Selection of Change-prone Software Components using the Expertise of Semantic Web and Intelligent Computing Methods

A

THESIS

Submitted in fulfilment of the

requirements for the award of the degree of

Doctor of Philosophy

Submitted By

Loveleen Kaur

(Registration No: 951403005)

Under the guidance of

Dr. Ashutosh Mishra

Assistant Professor

Computer Science and Engineering Department

Thapar Institute of Engineering and Technology, Patiala, India

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY

(Deemed to be University)

PATIALA – 147004 (INDIA)

January 2021

CERTIFICATE

I hereby certify that the work which is presented in this thesis entitled **Selection of Change-prone Software Components using the Expertise of Semantic Web and Intelligent Computing Methods**, in fulfilment of the requirements for the award of degree of **Doctor of Philosophy** submitted to the Computer Science and Engineering Department (CSED), Thapar Institute of Engineering and Technology, Patiala, Punjab, India is an authentic record of my own work carried out under the supervision of Dr. Ashutosh Mishra and refers to the work of other researchers, duly listed in the reference section.

The matter presented in this thesis has not been submitted for the award of any other degree of this or any other Institute.

(Loveleen Kaur)
Registration No. 951403005

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge and belief.

(Dr. Ashutosh Mishra)
Assistant Professor
Department of Computer Science and Engineering
Thapar Institute of Engineering and Technology
Patiala, India-147004
Supervisor

ACKNOWLEDGEMENT

I would like to express my deepest gratitude to Dr. Ashutosh Mishra, my supervisor, for his guidance, meticulous supervision, innovative feedback and priceless support throughout my PhD thesis. Being under his supervision has been a great privilege and blessing for me. I learnt a lot from him both about research and professionalism and his teachings will benefit me immensely in my future endeavours.

I would also like to acknowledge the Director and Dean (RSP), Thapar Institute of Engineering and Technology, Patiala for the indispensable academic and technical support. I express my gratitude to the doctoral committee comprising of Dr. Maninder Singh, Dr. Sreelekha Pandey, Dr. Parteek Bhatia, and Dr. M.D. Singh and PG coordinators during the course of my thesis: Dr. Rinkle Rani and Dr. Sushma Jain for monitoring the progress and providing valuable suggestions for improvement of my thesis work time to time. I am also grateful to all the Editors and anonymous reviewers of international research journals in the field of my work, who always patiently replied to my frequent enquiries on the "Manuscript Status". I would also like to thank my fellow research scholars, especially Dr. Sachendra Singh Chauhan and Mrs. Priya Arora for their constant inputs regarding my work, whenever needed.

Ultimately, I could not have achieved the major milestones of my life without the love and reassurance of the family I was born into. Deepest thanks to my parents who equipped me to reach where I am today. I would also like to express my profound appreciation to the family that I married, especially to Ramanjit, my husband, for his steady love and patience when it was most needed.

I am grateful to Waheguru, whose supreme holiness gave me the wisdom, resilience and determination to overcome all the roadblocks that have stood in my journey to this day.

ABSTRACT

Software component development, like generic software development, deals with "the construction of multi-version software" which will relentlessly be predisposed to changes either to add novel features or to reform source code for accommodating future changes or for removing existing defects as a part of corrective maintenance. Therefore, knowledge regarding the change-prone source code components of any software project or component is vital as they are prospective cradles of modifications and defects and could also depict possible design issues that need to be solved.

This thesis work proposes a novel change-prone software selection mechanism that employs Semantic web technology and Intelligent Computing Methods (ICM). The mechanism essentially selects those code components (Java files) of a software component that will be employed *with change* in the successive release of the component. Additionally, the work also identifies and establishes a cognitive aspect to the software change process.

We first begin by extensively surveying the existing research in order to understand the topic in concern from three primary standpoints:

(i) *Software change prediction* with an emphasis on the metrics used for software change prediction along with the prediction techniques employed, statistical tests used and validation strategies applied,

(ii) The *cognitive complexity metrics* introduced and validated in literature and their use to estimate any software development procedure, and

(iii) Application of *Semantic web technologies for software component-based tasks* with a key focus on the year wise trend of the research articles and possible justification of the usage of Semantic web technology and tools for a specific phase of component-based software development.

From the literature studied, we observed that it is challenging to modify code fragments from existing software that are difficult to comprehend. However, since systematic software maintenance includes an extensive human activity, cognitive complexity is one of the intrinsic factors that could potentially contribute to or impede an efficient software maintenance practice, the empirical validation of which remains vastly unaddressed. Thusly, we first conduct an experimental analysis in this thesis work in which the software developer's level of difficulty in comprehending the

software: the cognitive complexity, is theoretically computed and empirically evaluated for estimating its relevance to actual software change. For multiple successive releases of two software components (plugin projects written in Java), where the source code of a previous release has been substantively used in a novel release, we calculate the change results and the values of the cognitive complexity for each of the version's source code Java files. We construct eight datasets and build predictive models using statistical analysis and Machine Learning (ML) techniques. The pragmatic comparative examination of the estimated cognitive complexity against prevailing metrics of software change and software complexity clearly validates the cognitive complexity metric as a noteworthy measure of version to version source code change.

Secondly, several studies exist in literature that focus on finding the association between change-proneness and software metrics. Nevertheless, since the quest for the best classifier for change-prone source code elements is an ongoing process, we then present a pervasive framework for software change prediction by investigating various techniques from the categories of Intelligent Computing Methods (ICMs) and Statistical approaches for creating version to version change-prediction models with respect to Java files. The performance of the models is assessed during two validation scenarios: k-fold intra-release validation and inter-release validation, where the latter is useful for estimating the trend of change-proneness of files in the upcoming versions. Our experiments indicate that the prediction techniques perform differently under the selected validation settings, at the same time confirming the proficiency of the selected prediction techniques in lieu of developing change-proneness prediction models. Such models could aid the software engineers in the initial stages of software development for classifying change-prone Java files for the analyzed plugin project versions, in turn aiding in the trend estimation of change-proneness over future versions.

Lastly, we present the change-prone Java file selection mechanism developed using ICM and Semantic web technology. The framework proceeds with the construction of an ontology in Protégé with respect to the Java files in each of the selected software component versions using static source code metrics as attributes. It then employs Semantic Web Rule Language (SWRL) and Drools inference engine to formulate and induce potential rules acquired via the most appropriate ICM that classifies a Java file

as change-prone or stable. The relevant change-prone or stable files are then selected via the Semantic Query-Enhanced Web Rule Language (SQWRL) commonly employed for extracting information from ontologies. Additionally, we demonstrate how information (in the form of patterns or rules) which is constructed by a prediction technique to make relevant predictive decisions can be expressed in the ontology via valid SWRL rules. Since SWRL supports monotonic inference only, this can result in an incomplete inference should the rule contain more than one negation. We provide a methodology to express such rules in SWRL without any loss of relevant inference. The proposed selection mechanism is evaluated on various successive releases of the selected plugin projects and clearly establishes the applicability of Semantic web principles with respect to building a successful change-prone Java file selection mechanism.

More generally, the findings of this thesis identify if cognitive complexity plays a role in software change and if the existing prediction techniques for software change are cogent enough to predict the version to version change-proneness of Java files of software components. It also analyses if the two paradigms: Semantic web technology and an ICM could in fact be employed in unison to select change-prone Java files from a software component version.

TABLE OF CONTENTS

ABBREVIATIONS

Acc	Accuracy
ADB	ADaBoost
AM	Ant Miner
ANN	Artificial Neural Networks
AUC	Area Under the ROC Curve
BAG	Bagging
BCSs	Basic Control Structures
BN	Bayesian Network
CART	Classification & Regression Trees
CBFS	Correlation-Based Feature Selection
CBO	Coupling between Objects
CBSE	Component Based Software Engineering
CFS	Cognitive Functional Size
CHalsE	Cumulative Halstead's effort
CHIRP	Composite Hypercubes on Iterated Random Projections
CI	Continuous Integration
CK	Chidamber-Kemerer
CPJFS	Change-Prone Java file Selection
CWU	Cognitive Weight Units
CycloC	Cyclomatic Complexity
DAG	Dagging
DAML	DARPA Agent Markup Language
DIT	Depth of Inheritance Tree
DL	Description Logic
DTNB	Decision Table/Naive Bayes hybrid classifier
FACT	Fast Classification of Terminologies
FC	Filtered Classifier
FLDA	Fisher's Linear Discriminant Analysis
FLR	Fuzzy Lattice Reasoning
F-S	F-Measure or F-Score
FURIA	Fuzzy Unordered Rule Induction Algorithm

GANN	Genetic Algorithm with Neural Networks
G-m1	Geometric mean 1
G-m2	Geometric mean 2
HBT	Hybridized Techniques
HFT	Hoeffding Trees
ICMs	Intelligent Computing Methods
IQR	Interquartile Range
JFC	JFreeChart
LB	LogitBoost
LCOM	Lack of Cohesion of Methods
LMT	Logistic Model Trees
LR	Logistic Regression
MCCF	Matthews correlation coefficient
ML	Machine Learning
MLP	Multi-Layer Perceptron
MM	Majority margins matrix
MOD	Modlem
MOEFC	Multi Objective Evolutionary Fuzzy Classifier
NB	Naïve Bayes
NCBFS	No Correlation-Based Feature Selection
NCP	Non Change-Proneness
NNEP	Neural Net Evolutionary Programming
NNGE	Non-Nested Generalised Exemplars
NOC	Number of Children
OIL	Ontology Inference Layer
OKBC	Open Knowledge-Base Connectivity
OO	Object Oriented
OTS	Off-The-Shelf
OWL	Web Ontology Language
PART	PARTial decision lists
PCA	Principal Component Analysis
RBFN	Radial Basis Function Neural Network
RDF	Resource Description Framework

RDFS	Resource Description Framework Schema
RF	Random Forest
RFC	Response For Class
RQs	Research Questions
RSS	Random Sub Space
RV	Ranked Voting
SBT	Search-Based Techniques
SLOC	Source lines of code
SMO	Sequential Minimal Optimization
SMOTE	Synthetic Minority Oversampling Technique
SPEG	SPegasos
SQWRL	Semantic Query-enhanced Web Rule Language
SWRL	Semantic Web Rule Language
VFI	Voting Feature Intervals
WMC	Weighted Methods for Class
XML	Extensible Markup Language

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
INTRODUCTION AND STATEMENT OF THE PROBLEM

This thesis work proposes a novel change-prone software selection mechanism that employs Semantic web technology and Intelligent Computing Method. The mechanism essentially selects those Java files of a software component that will be employed with change in the successive release of the component. Additionally, the work also identifies and establishes a cognitive aspect to the software change process.

Along with providing an outline of the various chapters of the thesis, this chapter sets a brief context to the research conducted and provides a succinct overview to topic in concern.

Section 1.1 captures the background of the study and includes a brief report on the existing literature with respect to the topic. Section 1.2 explains the motivation for undertaking this study. Section 1.3 states the problem statement of the research along with the objectives to be covered. Section 1.4 elaborates the contributions of this work. Section 1.5 presents the organisation of the rest of the chapters in this thesis and Section 1.6 concludes the chapter with a summary.

1.1 Background

It has become ever more expensive to build and maintain modern software systems with the users of such systems becoming increasingly advanced with regard to the capability they expect [1, 2]. Developers need to use a large number of specifications, protocols, techniques, and tools to create these systems, each being complicated and challenging. Rather than developing it on their own, development organizations have tried to meet this challenge by incorporating off-the-shelf (OTS) software components developed outside of their organization that provide the necessary functional capability [1-3].

Software components [1-5] are capable of being:

- Independently Developed: Small, independent teams are capable of building small, independent software components. The time taken to understand a component is significantly reduced, and the development of new features becomes easier.
- Independently Deployed: Each single component can be implemented separately, thusly, allowing the release of new functionality over software components versions with higher speed and lower risk.
- Independently Scaled: Each software component can be scaled independently of other components in a system.
- Reused: Software components provide a limited, specific functionality. This signifies that they can be adapted more easily for use in other services, systems or products.

Software component development, like generic software development, comprises of a construction of multi-version software that is persistently subjected to changes, either to add novel features or to reform source code for accommodating future changes [5]. Additionally, removing existing defects as a part of corrective maintenance is another significant cause for change [6]. Therefore, knowledge regarding the change-prone source code elements of any software project or component is vital as they are prospective cradles of modifications and defects and could also depict possible design issues that need to be solved. Adaptation of such changes in any existing software version usually leads to a new version of the software which is why we see that there exist multiple versions of single software project or component these days [6].

It is vital to point out that although change-prone source code elements and bug-prone elements of a software component might consist of some relationship, they are conceptually different. Firstly, bug-proneness only specifies that there exists a possibility that a source code element would have bugs in future, hence, the mere point that a source code contains bugs in no way indicates that it gets modified frequently. Furthermore, bug-prone constituents could be subjected to changes in order to rectify faults, but then again a software change does not only happen because of corrections. A software change could also happen due to evolution of software. Therefore, bug- and change-proneness of source code could possess some connection, but they are not exactly equal [6, 7].

Prediction of software change models [6-14] signify a recognized approach to highlight change-prone source code elements of software and are one of the primary constituents in supporting software changes. An effective software change prediction mechanism predicts those code elements that are likely to be employed with some change from one version of software to the next. In an actual scenario, change prediction models are capable of being directly incorporated in software developers' analytics dashboards (e.g., BITERGIA[1]) via which these models could help in giving a continuous feedback about which files are more prone to undergo some future change [6, 7]. This feedback may be employed for executing preventive maintenance works prior to transitioning the code into production. For example, in case of a continuous integration (CI) milieu, it may be desired that the source code is refactored prior to start of CI pipeline in order to avoid warnings related to code quality by build failures or static analysis tools [15, 16]. Likewise, software change prediction models could also prove to be valuable for software project managers for the purpose of properly scheduling maintenance activities.

The research community, in the last decade, has often scrutinised the software change from two primary standpoints [8, 14]. On one hand, experimental analyses were performed in an effort to investigate the role of software metrics in predicting change-proneness, while other studies focussed on finding the most suitable technique to predict the change-prone source code elements.

1.1.1 Predictor variables for software change prediction

Numerous software metrics have been proposed to depict several features of the source code and design of software [10]. These metrics or a subset of these metrics, in conjunction with data related to change obtained from historical repositories has been employed in the development of the change-proneness prediction models [8, 14].

Software metrics can generally be segregated into two categories: Process and Product metrics [8, 14]. Product metrics have been normally observed to be computed at a Class-, Method- or Component-level. Class level metrics [8, 14] include metric suites such as Chidamber-Kemerer (CK) [17] metrics suite, QMOOD [18] and Lorenz and Kidd metric suite [19] among others. Method-level metrics consist of those proposed by Halstead [20] and McCabe [20, 21]. Component-level metrics measure

[1] https://bitergia.com

functional and non-functional features of a software component like its portability, reusability, usability, suitability and complexity in the context of a Component-Based Software System [8].

On the other hand, process metrics [8, 14] are related to the software process in which processes use products and produce new products used by some other process or activity. Some of the examples of process metrics can be the number of bugs discovered while reviewing the code, time taken to complete the process, etc.

Some of the software change prediction studies [22-26] conclude that the size of the source code elements has a significant impact on change-proneness. It has been found that large source code classes are more prone to changes, as compared to classes having comparatively smaller sizes (in terms of LOC, number of operations, etc.). Additionally, the past literature also establishes that there exists a significant relation between classes playing roles in design of the software and the classes that are likely to be changed [22, 27]. This is why most of the work conducted for predicting classes which are change-prone, discusses about using structural information obtained from source code. It has been also observed that product metrics are the most frequently used metrics for software change prediction wherein the metrics of the CK metric suite are the most common predictor variables [8, 14]. The basic conjecture regarding this is that classes that have poor quality of code are more likely to undergo some change in future. Additionally, some studies also opt for more than one type of metrics to predict software change [28-30].

Research efforts [6-7, 28-30] have also been made that offer supplementary knowledge regarding the factors that impact the change-proneness of source code classes. For example, authors in [28] investigate the role that process metrics play in regard to change prediction. In particular, they propose evolution metrics which characterize the historical background of a source code class from diverse perspectives such as the number of previous changes done to a class in a fixed time period.

Moreover, the past literature chiefly focuses on examining software change at finer-granularities of source code like at a class- and/or method-level or sometimes at a source code file level. Even though software components have been used as target projects for analysing software change [12, 27-29, 31-34], this change assessment has

not been performed in the context of a Component-based Software System. Hence, there is an absence of adequate studies examining change at a 'software component level' using component-level metrics.

However, authors argue that developer-related factors are equally relevant for change prediction. For instance, Catolino [7, p. 15] comments:

> *"Despite the effort devoted by the research community over the years, we believe that current approaches miss an important piece of information, i.e., they do not consider developer-related factors, which can provide information on how developers perform modifications and how complex this process is."*

There have been a few models [35, 36] proposed for bug prediction that rely on the usage of developer-related factors to predict buggy source code elements. However, these models only monitor and evaluate the metrics corresponding to the developer activity that has been performed on the source code, for instance, the entropy of modifications done by developers or the count of developers that changed a code element over time. These metrics might work for bug prediction but as far as change-prediction is concerned, project managers and developers call for the comprehension of the software project's structure via its source code, and the relationships that exist among its source code elements in order to make preventive, adaptive and corrective changes.

Thusly, the ability of the software developer to comprehend the given source code also forms an important quality aspect influencing software change as Borstler et al. [37, p. 231] states:

> *"About 50% of software maintenance costs are spent on code comprehension and more than 40% of the comprehension time is spent on plain code reading alone."*

Although conjectured to be vital with respect to software maintenance, this effort needed by the software developer to understand the source code before modifying it still remains unexplored especially in the software change prediction milieu.

1.1.2 Techniques for software change prediction

The name "Intelligent Computing" has been used for over 20 years, with different interpretations. However, Duch [38, p. 5] surveyed various journals and books related to Computational Intelligence (CI) and confirmed that:

> *"There are many books applying CI to diverse areas, such as Computational Intelligence in Design and Manufacturing [39, 40], Computational intelligence in*

Software Quality Assurance [41], Computational intelligence in Control Engineering [42], Computational Intelligence in Economics and Finance [43] and many others. They all tend to see computational intelligence as a consortium of data-driven methodologies which includes fuzzy logic, genetic algorithms, probabilistic belief networks and machine learning (ML)".

ML has been by far the most popular category of Computational Intelligence (CI) methodologies and when employed in industrial contexts, is known as predictive analytics or predictive modelling [44]. It concentrates on the development of computer programs that change when exposed to new data and can teach themselves to grow. ML programs perform pattern detection in data and program actions are adjusted accordingly.

As far as software change is concerned, the techniques employed for the prediction of change-prone software elements can be chiefly segregated into two categories: ML and statistical [8, 14]. Some of the studies [8] conducted in the nascent years employ neither a ML method, nor a statistical technique for software change prediction and instead rely on descriptive statistical analysis or traditional graphical methods such as boxplots and scattergrams.

However, the last one and a half decade has witnessed many researchers employing various methods for conducting an empirical study to investigate and confirm the capability of Object Oriented (OO) metrics in predicting change-prone source code elements via ML techniques like Bayesian classifiers [6, 12, 30, 32, 45-48], Decision Trees [6, 23, 33, 47- 49, 50-56], Artificial Neural Networks [6, 11, 12, 32, 33, 34, 47, 55, 57, 58], Support Vector Machines [10, 12, 56] and meta-classifiers [7, 30, 33, 47, 56, 57]. Statistical methodologies such as Logistic Regression [6, 11, 28, 33, 49, 51, 55, 57, 59-61] have also been analysed for the same. Authors have often incorporated classifiers from each category and have compared their performances with respect to predicting change-prone software classes or files of a certain software project using suitable performance measures. Additionally, other ICMs [9, 54, 61, 62] like Particle Swarm Optimization, Ant Colony Optimization and evolutionary methodologies like Hybridized Techniques (HBT), and Search-Based Techniques (SBT) [45] have also been evaluated and their efficacy has been compared with the previously examined ML techniques with respect to software change prediction.

However, as the quest for identifying the best classifier with respect to a certain prediction scenario is an on-going process [61], there exist many techniques in literature that have still not been scrutinized in regard to software change prediction. Moreover, the validation findings of different projects do not need to be similar and it is therefore imperative to assess the efficacy of the prediction techniques with respect to specific software projects [47].

Post evaluating the model performance using many measures, it is imperative to perform an analysis statistically to verify if the differences in the performance of the models are random or real. There has also been some application of statistical tests with authors in [31, 47, 63] employing the t-test and some of the other studies employing the Friedman's test along with the post hoc Wilcoxon Signed Rank test [9, 30, 48, 61, 62] for establishing the disparities between the performances of several prediction techniques.

Additionally, in an ideal scenario of model validation, empirical studies need to have an inter-release validation wherein historical data from the previous versions are utilized for training a model which in turn predicts change-prone source code elements of a new release. Yet, almost all the previous articles only employ a k-fold cross validation (intra-release) or an inter-project validation to validate the results of the prediction models.

1.1.3 Semantic web techologies with respect to software change

The development process of software at high speed with the help of conventional software engineering principles and practices was deemed to be insufficient and inefficient. This compelled researchers to develop the most important component technology, Component Based Software Engineering (CBSE) [1-3] which promotes software systems development by selecting or creating reliable, reusable and robust software components and integrating these components within appropriate software architectures.

Even though component development has vast benefits, it has been generally afflicted with different issues due to which it has not been able to achieve its full potential. This has primarily been due to the absence of clear protocols to specify a software component's description, configuration, integration, and modification, so that it can be selected and accommodated within other development environments [64].

Software components are housed in software reuse repositories on the Web and several works have been proposed in literature, such as those given in [65-70], that depict how a Semantic web technology, 'Ontology', can be utilized to model common qualitative features of these software components in order to aid in effective selection of the required components from such repositories. Ontologies have been by far the most commonly used Semantic web technology in CBSE. Ontology has shown to facilitate software component selection, storage as well as interoperability, thus catering to almost all the aspects of CBSE. With the help of additional practical features like reasoning, ontologies are able to validate the developed component-based software ontology and detect the possible vulnerabilities and threats [71].

The next section provides the motivations for this thesis work that drive the research reported.

1.2 Motivation

The following three factors motivate the research undertaken in this thesis:

Firstly, numerous research efforts have attempted to quantify the change-proneness of a code element for software change prediction. These usually involve static code metrics gathered automatically by various software tools [6-14, 22-24, 26, 28-30, 32-34, 45-63]. Although all these measures have been validated to substantially aid in executing a successful maintenance or development activity, the human cognition involved in changing an existing code, which may hamper development efforts due to limitations of cognitive resources, inadequate modelling abilities or ineffective reasoning, has not been empirically scrutinized. The interpretation of source code is one of the most cognitively arduous tasks that humans perform in the software change process since before software components or projects can be changed, its constituent source code has to be understood by the potential developer, thus inherently rendering the software change activity as a common cognitive task [72-74] .

Secondly, the last two decades have witnessed many researchers employing various prediction methods for conducting empirical studies to examine and confirm the capability of the software metrics in predicting change-prone source code elements. However, there exists a necessity to re-evaluate this topic for the following reasons:

- The authenticity of empirical analyses can be only ascertained via statistical testing. Previously employed tests for software change prediction have their own shortcomings which hinder these tests from being all-pervasive. For instance, like the t-test relies on many presumptions like the normal distribution of data and it is not advisable to use Wilcoxon's test sans Bonferroni correction, as family-wise error is not considered.
- Most of the former findings validate the generated predictive models on the same data that was used for their training and consequently do not assess the efficacy of these models to predict the change-proneness of the source code elements of software components across imminent versions.
- Estimation of the performance proficiency of various change-proneness prediction models is essential for assessing their pragmatic pertinence. There are other ICMs in literature that could produce better results which still remain unexplored as far as change-proneness prediction is involved.

Lastly, the past literature argues that Semantic web technologies can be utilized to context model basic information about the software components housed in reuse repositories. This context modelling is done in a manner that facilitates reasoning on this information for the purpose of adapting to the behavior of system. Semantic web employs Ontology as a knowledge representation scheme to store knowledge. Ontologies are among the most appropriate solutions for context modelling [65-70]. In addition, the ease with which ontologies create context-aware repositories is shown in many works [75-80]. However, the work performed in these articles [65-70, 75-80] is insufficient because:

- It focuses on employing ontology only for the purpose of describing the basic descriptive aspects of software components so as to make the selection and retrieval procedure uncomplicated.
- The ontological modelling of information within the software component(vis-à-vis the source code elements like classes or files) housed in these software repositories that can be utilized for performing software maintenance activities, for instance, change prediction, bug prediction, software testing etc. has not been performed in the existing literature.

Moreover, attributes or metrics that impact or affect a potential change of software have not been included in these ontology-based repositories, in spite of there being adequate research and literature for the same [6-14, 22-24, 26, 28-30, 32-34, 45-63].

Having identified the motivation for performing the research work conducted in this study, the next section introduces the problem statement formulated for this thesis along with the research objectives that are intended to be covered.

1.3 Problem Statement and Research Objectives

The proposed problem statement for this thesis reads as follows:

> *"To establish a cognitive aspect to the software change process and to propose a novel change-prone software selection mechanism that employs Semantic web technology and Intelligent Computing Method and selects those Java files of a software component that will be employed with change in the successive release of the component."*

In order to perform this task, following objectives are proposed to be carried out.

1. To study the various source code change-proneness determination techniques.
2. To find out the change-proneness in the source code components of various versions of a component-based software system using Cognitive parameter.
3. To predict trend of change-proneness in the versions to come by using the expertise of Intelligent Computing techniques.
4. To propose, implement and validate a software component selection mechanism for various large scale Component Based Software Systems in order to select the prospective change-prone code components from the individual versions with the help of Semantic web technologies.

Albeit the experiments in this research article have been conducted on Java files, the conclusions drawn are applicable to the OO standard in general. However, the reasons for selecting Java files as opposed to any other OO language for our analysis have been specified below:

- Generalizability: Every enterprise uses Java in one way or other. It is also cross-platform in nature that enables programmers to write one code and run it across desktop, mobile, and embedded systems. Moreover, the object-oriented nature of Java allows developers to create modular programs and write reusable codes. This saves lots of efforts and time, improving the productivity of the development process.

As per Oracle, more than 3 billion devices run Enterprise, Scientific, Web, Mobile applications including Android applications etc. designed on the development platform.

Conducting an empirical analysis on Java-based code would ensure that the findings from this research work could be applied to a larger population.

- Stability: Java is a mature language that has immensely evolved over the years. Hence, it's more stable and predictable. Additionally, it offers a relatively seamless forward compatibility from one version to the next, thus making it a popular choice for software development wherein there are lot of updates/changes across infrequent version releases.

The next section states the contributions of the research reported in this thesis work.

1.4 Contributions of the Thesis

In the proposed work, we show that the key to an effective selection of change-prone Java files of software component relies on a set of appropriate software metrics and rules which eloquently isolate the change-prone files from the not change-prone ones. Through this combination of software metrics and rules, we develop a Semantic web-based mechanism for change-prone Java file selection, wherein every such file belongs to a software component and its features are modelled in the form of an ontology.

Along with some of the previously introduced software metrics for software change prediction, the proposed Semantic web-based software selection mechanism evaluates and incorporates a novel metric capturing the cognitive complexity that exists with respect to every Java file of the software component.

The principal contributions of this thesis are:

- Highlighting that the measurement and evaluation of software complexity from a human-oriented perspective of software engineering has been missing in literature. This form of software complexity is often touted to be essential for controlling high degrees of software maintainability.
- Demonstrating in detail how cognitive complexity can be calculated and evaluated for a Java file of a software component utilizing the premise of the

cognitive functional size (CFS) proposed by Wang and Shao [81] and the cognitive complexity measure proposed by Crasso et al. [82].

- Conducting a comprehensive and comparative empirical evaluation of the calculated cognitive complexity metric with respect to software maintenance, specifically for estimating its prediction capability with respect to version to version change-proneness of Java files using various predictive models and statistical testing.
- Establishing the cognitive complexity metric's credibility as one of the important independent variables for evaluating the change-proneness of a Java file of a software component.
- Providing a comparative analysis of 31 predictive models in the context of version to version change-proneness prediction of Java files under the intra-release (VS1) and inter-release (VS2) validation scenarios. The prediction techniques have been carefully chosen in a way so that each category has a minimum of one technique being analysed. Additionally, techniques that have not been explored vis-à-vis change-proneness prediction in the existing literature have also been analysed for their competency.
- In particular, we incorporate an inter-release validation, therefore imitating with this validation a predicament that is normally encountered in an actual software maintenance milieu, wherein historical data from the previous versions are utilized for training a model which in turn predicts change-prone Java files of a new release. Such a validation scenario aids the software developers in examining the efficiency of the selected methodologies in predicting the trend of change-proneness of Java files of the upcoming component versions. Moreover, the testimony acquired from such exhaustive data-based comparative pragmatic analyses can assist the software researchers and practitioners to cultivate ample corpus of knowledge to accept/reject a given hypothesis.
- Based on the performance of the prediction techniques, we propose a selection mechanism that uses the Semantic web technologies to select the change-prone Java files of software components. This mechanism provides insights so as to how software developers and practitioners can model supplementary source code related information regarding a software at a finer granularity (file

level, in our case) in an ontology-based software repository. Such ontologies are capable of transforming existing content management systems to knowledge-bases wherein information regarding software component versions are not only stored but also processed for making relevant software maintenance decisions.

- Additionally, we demonstrate how information (in the form of patterns or rules) which is constructed by a prediction technique to make relevant predictive decisions, can be expressed in the ontology via valid SWRL (Semantic Web Rule Language) rules [83]. Since SWRL supports monotonic inference only, (i.e. SWRL rules cannot modify existing ontology information and SWRL executes one condition at a time serially) this can result in incomplete inference should the rule contain more than one negation. We provide a methodology to express such rules in SWRL without any loss of relevant inference.

1.5 Organisation of the Thesis

This thesis shows how a prediction technique can be utilized along with Semantic web technology in order to develop a version to version change-prone Java file selection mechanism. These Java files of various successive releases of software components are quantified via appropriate change-proneness prediction metrics, including a cognitive parameter, to facilitate the selection. The organization of the thesis is as follows:

Chapter 2 begins by explaining software components and their code constituents. It then defines software change and the possible why, when and where factors of software change. The chapter then splits into three broad literature reviews:

- First review consists of a study of the existing literature that focuses on determining software change, which includes a dissection on: (a) the software metrics reported and validated to be useful for change-proneness prediction, (b) prediction techniques employed, (c) validation strategies utilized and, (d) the statistical tests incorporated,
- Second literature review presents the research background that exhibits the status of cognitive complexity with respect to software development and software change, and

- Third literature review examines the usage history of Semantic web technologies to achieve various tasks of the CBSE process. It provides details regarding the purpose of employing Semantic web technology for CBSE, type of Semantic web technology employed and specific tools and methods incorporated.

The literature review is followed by a summary of major observations drawn from the review along with the shortcomings identified.

The work done in this chapter caters to the research objective 1 proposed in Section 1.3.

A significant portion from this chapter (Chapter 2) has been published as a systematic review article titled "Software component and the Semantic web: An in-depth content analysis and integration history" in Journal of Systems and Software, Elsevier.

Chapter 3 conducts an experimental analysis in which the software developer's level of difficulty in comprehending the software: the cognitive complexity, is theoretically computed and empirically evaluated for estimating its relevance to actual software change of source code Java files. For eight datasets constructed from successive releases of two software components, where the source code of a previous release has been substantively used in a novel release, we calculate the change results and the values of the cognitive complexity for each of the version's source code Java files. Analysis of the effect of the calculated cognitive complexity metric on this change statistic is performed via appropriate statistical and ML methodologies. Two previously introduced complexity measures and six change prediction measures are also evaluated on the same eight datasets to provide an exhaustive comparative analysis. Although the model results are reported using some commonly employed performance measures, AUC, with the aid of Friedman's statistical test, is primarily used to compare and rank the relative predictive capability of the each of the metrics. Through the consistency in the empirical results drawn in this analysis, it is established that the evaluated cognitive complexity metric is a distinguishable measure of version to version source code change of Java files.

The work done in this chapter caters to the research objective 2 proposed in Section 1.3.

This chapter (Chapter 3) in its entirety has been published as a research article titled "Cognitive complexity as a quantifier of version to version Java-based source code change: An empirical probe." in Information and Software Technology, Elsevier.

In *Chapter 4*, we extend our analysis pertaining to change-proneness prediction by providing a thoroughly pervasive framework which is capable of assisting the software developers in an expert selection of a prediction approach amid a massive choice of prevailing methods. We examine and compare thirty one techniques (which include statistical and ICMs) for creating version to version change-prediction models on Java files of software components. We also analyse the selected techniques using an inter-release validation for estimating the trend of prediction. Additionally, Kruskal-Wallis test and analyses like the Scott-Knott cluster analysis with Borda Counting statistically scrutinize the results obtained in this chapter with the purpose of identifying groups of techniques that are statistically dissimilar in their prediction performance and in ranking the selected techniques in accordance to their performance with respect to predicting software change- proneness of Java files.

The work done in this chapter caters to the research objective 3 proposed in Section 1.3.

This chapter (Chapter 4) has been published as a research article titled "A Pragmatic Framework for Predicting Change Prone Files Using Machine Learning Techniques with Java-based Software," in Asia Pacific Journal of Information Systems (SCOPUS Indexed).

Chapter 5 presents the change-prone Java file selection mechanism developed using Semantic web technology. The mechanism constructs an OWL-based ontology in Protégé with respect to the Java files in each of the selected software component versions using static source code metrics as attributes. It then employs Semantic Web Rule Language (SWRL) and Drools inference engine to formulate and induce potential rules acquired via the most appropriate ICM that classifies a Java file as change-prone or stable. The relevant change-prone or stable files are then selected via the Semantic Query-Enhanced Web Rule Language (SQWRL) commonly employed for extracting information from OWL ontologies. The proposed selection mechanism is evaluated on various successive releases of software components and clearly

establishes the applicability of Semantic web principles with respect to building a successful change-prone Java file selection mechanism.

The work done in this chapter caters to the research objective 4, the last objective of our thesis work proposed in Section 1.3.

This chapter (Chapter 5) in its entirety has been communicated as a research article titled "An ontology-based approach to change-prone software selection" in Applied Ontology, IOS Press.

Chapter 6 gives a summary of this thesis as well as discusses on directions for future work.

1.6 Summary

The work performed in this thesis chiefly aims to identify if cognitive complexity plays a role in software change and if the existing prediction techniques for software change are cogent enough to predict the version to version change-proneness of Java files of software components. It also analyses if the two paradigms: Semantic web technology and ICM such as a ML technique could in fact be employed in unison to select change-prone Java files from a software component version.

This chapter presents a background to the thesis work that follows. It provides a brief introduction to the topic in concern and concisely discusses the factors and techniques used to perform software change prediction. It also succinctly reports the means by which Semantic web technologies have been stated to be useful to CBSE. It presents the motivation for the work performed in this thesis on the basis of the literature studied and reports the problem formulated to be catered to in this thesis work. The research objectives covered as a part of solving the problem are stated along with the contributions that this thesis work makes to the software engineering research community though the inferences drawn. The chapter concludes with an overview of how the rest of this thesis is organised.

The next chapter details the literature studied apropos to the topics in concern.

CHAPTER 2
LITERATURE SURVEY

This chapter surveys the existing research in order to understand:

- Software change in the context of prediction techniques,
- The cognitive complexity metric and its role in software development, and
- Application of Semantic web technologies in the component-based development scenario.

This chapter begins by providing a brief background of software components in Section 2.1. The subsections explain the software components in CBSE in Section 2.1.1, which is followed by a brief breakdown on the code constituents of a software component in Section 2.1.2. Section 2.1.3 defines software change and the possible "why", "when" and "where" factors of software change.

The literature survey conducted in this chapter begins from Section 2.2 and, given the broad nature of the problem domain, is split into three comprehensive literature reviews as follows:

- Section 2.2 contains a study of the existing literature that focuses on determining software change, which includes a dissection on: (a) the software metrics reported and validated to be useful for change-proneness prediction, (b) prediction techniques employed, (c) validation strategies utilized and, (d) the statistical tests incorporated. For this literature review we extensively refer to the very recent in-depth literature review done by Malhotra and Khanna [14] and extend their analysis by including some of the articles overlooked by the authors.
- Section 2.3 depicts the research background that exhibits the status of cognitive complexity with respect to software development and software change in literature.
- Section 2.4 reviews and examines the usage history of Semantic web technologies to achieve various tasks of the CBSE process and Software development. It provides details regarding the purpose of employing Semantic

web technology for CBSE, type of Semantic web technology employed and specific tools and methods incorporated.

The literature review is followed by a summary of major observations drawn from the literature studied and the gaps identified in Section 2.5. Finally, Section 2.6 summarises the chapter.

2.1 Background

The following subsections provide an explanation of the software components in CBSE, which is followed by a brief breakdown on the code constituents of a software component and the possible "why", "when" and "where" factors of software change.

2.1.1 Software component in CBSE

A Component-based architecture focuses on the decomposition of the design of a software system into individual software components that represent well-defined communication interfaces containing methods, events, and properties. A software component [1, 4, 5, 84] can be:

- a *user-interface* software for processing different requests views and scenarios (for example, different components can be employed to display the same information in a web page and a mobile app), or
- a *software model* that handles requests for events including data processing and business rules (for example, a model handling a bill payment request for an internet banking website), or
- *software controller* which decides what other components to call for a particular request or event (for example, a software controller dynamically loading different views for a bill payment based on factors such as language or transaction status), or,
- a *plugin* which has been designed to extend the functionality of an application or system (for example, a plugin for media player to visualise music).

Figure 2.1 presents a straight forward case of various software components encapsulated within the purview of a proposed ATM component-based system characterized in UML 2.0. ATM is a machine at a bank branch or other location which enables customers to perform basic banking activities. All system tasks like the

processing of the ATM card details, linking the card to the appropriate bank and printing of the customer receipt post processing or transaction etc. are placed into separate components. Additionally, components use interfaces to connect with each other. In order to extend services to the other parts of the system, a component uses a *provided* interface [5, 84]. This interface clearly details the services extended and the manner in which the other components can avail them. This interface acts as a signature of the specified component and there is no requirement for the client to understand about the inner mechanism of the component (implementation) for using it. The UML illustrations in Figure 2.1 represent interfaces using a symbol resembling a lollipop tagged to the component on one of its outer edges.

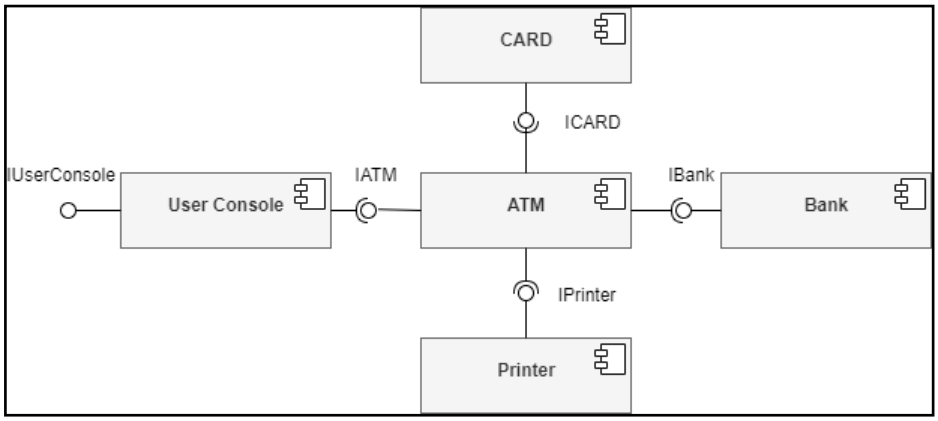

Figure 2.1 UML ATM Component-based system

2.1.2 Code components of a software component

Component-based software engineering (CBSE) uses almost similar kind of methods, tools, and principles as used in software engineering. However, there are certain differences. The prime difference is that the CBSE distinguishes the process of "component development" from that of "system development with components" [5, 84, 85]. Figure 2.2 presents an elaborative breakdown of the component development lifecycle given by Lau and Ukis [85]. It depicts that once the business and technical requirements related to the software component are established, the focus is then put on determining how the software component will be designed, what component model the software component will adhere to, how the component interfaces will be specified etc. Following this, the source code related to the software component is developed and the software component is implemented after which it is packaged and

made available to the users or released via various software repositories. Post release, the software component is configured and installed onto a target system and made ready for execution.

As observed, code components of a software component consist of the following [5]:

- *Source code*: The source code for a software component is the full set of machine-readable software files containing procedures, modules and classes.
- *Executable files*: These machine-executable files contain run-time libraries and precompiled object code.
- *Executable component model*: A component model defines specific interaction and composition standards. A component model's implementation is the dedicated set of executable software elements required to support the execution of components that conform to the model.

All the three above mentioned constituents are required to package the software component into a machine-readable software [5].

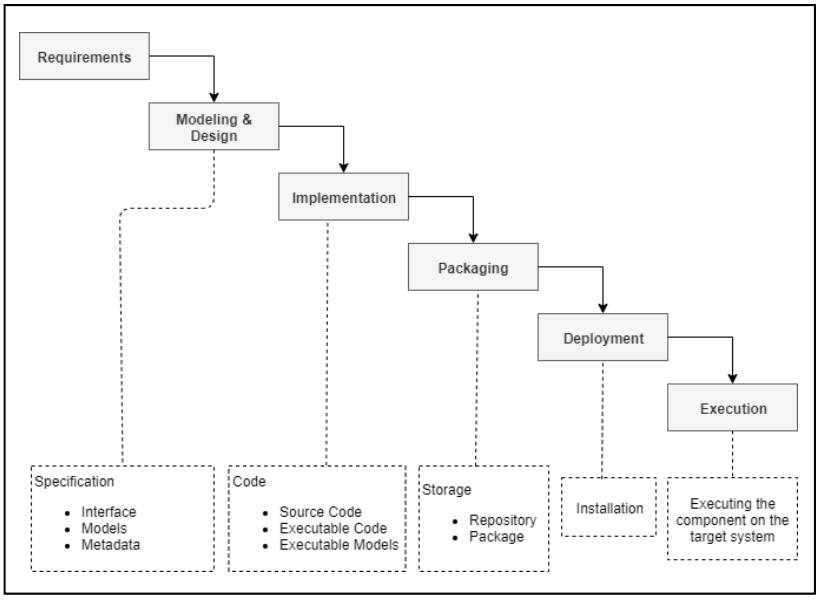

Figure 2.2 Software component lifecycle [85]

The focus of our analysis will be on the source code component of the software component, which does not include any executable code or models.

2.1.3 Software change

Software of every size requires changes and once a software component version is released, it is subjected to novel requirements and changes to the existing requirements [8, 14]. There are various reasons for the changes to occur which may include correction of errors which are discovered during operation, or enhancements to optimize the performance or some non-functional characteristics [8, 14, 86].

The following sub-sections focus on the why, when and where aspects of software changes, common to a software project or a software component.

2.1.3.1 Reason for a software change

There are a number of different reasons [8, 14, 86] why software undergoes a change:

- *Software maintenance*: This means that software is subjected to changes owing to varied requirements keeping the fundamental structure stable. This happens to be the most common use case for software change.
- *Architectural transformation*: This is a more radical approach to software change than maintenance as it involves making significant changes to the architecture of the software. Most commonly, software evolves from a centralised, data-centric architecture to client-server architecture.
- *Software re-engineering*: This is different from other strategies in that no new functionality is added to the system. Rather, the software is modified to make it easier to understand and change. Re-engineering may involve some structural modifications but does not usually involve major architectural change.

2.1.3.2 Temporal properties affecting software change

Temporal properties [86] concerned with software changes can be segregated into:

- *Time of Change*: Depending on when the software undergoes a change, it can be categorised as a Static, Load-time or a Dynamic change. Static change is the customary way to maintain software. Here the source code is edited or extended by the programmer and then the re-compiled into a novel executable version. Changes pertaining to Load-time occur when software components are put in an executable system. Generally, the evolution of load-time does not need source code access. It directly applies modifications to the binaries of the

system. A leading illustration for a mechanism based on load-time evolution was captured as Java's ClassLoader architecture [87] which allowed for class file changes at load-time. Dynamic change occurs during execution of the software. On the contrary, in case of dynamic evolution modifications are made at run-time.

- *Change History:* This term from the perspective of a software system indicates compilation of all modifications made to the software. In completely unversioned systems, changes are applied destructively so that new versions of a component overwrite old ones. In systems that support static versioning, new and old versions can physically coexist at compile-time but they are identified at load- and run-time. On the contrary, dynamically versioned systems [86, 88] permit two separate versions of single component being used simultaneously in parallel, under the same name space. Old or legacy components may be required to be used side by side the new ones during the transition period for business purposes. Safe updates meant for the legacy components are required to be accomplished in an environment where the new clients employ the newer version and only the existing clients make use of the old components.
- *Anticipation:* While making design decisions some software modifications which can be anticipated in the initial phase of the system development are in a position to be incorporated. On the other hand changes that could not be anticipated ensue mainly due to dynamic business contexts or varying technical requirements or are the result of end-users' experience with novel systems. On the whole, an anticipated change is much easier to incorporate as compared to an unanticipated one [89].

2.1.3.3 Object of software change

The objects [86] of software change are chiefly concerned with the location of the software changes and can be assessed from the viewpoints of:

- *Artifact*: Static evolution can lead to modification of various types of software artifacts. These changes can be due to new requirements arising with respect to architecture and design, source code, documentation or test suites.
- *Granularity*: Granularity refers to the scale of the artifacts to be changed and can range from very coarse, through medium, to a very fine degree of granularity. For example, coarse granularity might refer to changes at a

system, subsystem or package level, medium granularity might refer to changes at class or object level and fine granularity might refer to changes at variable, method, or statement level which is mostly smaller than a single file.

- *Change Type:* The features of the change or modification itself can impact the way that modification is effectuated [90]. Changes which modify the structure of the software are called structural changes and these can in turn also impact the software behavior. These changes can be categorized as addition (inculpating new elements), subtraction (taking away elements) and alteration (modifying a current element, like renaming). Next to structural changes, a distinction should be made between semantics-modifying and semantics-preserving changes.

2.2 Software change prediction

As discussed in Section 2.1.3, a software is relentlessly predisposed to changes which are obligatory to acclimate to a different environment, to add new features, refactor the source code, or to fix bugs [8, 14, 86]. An effective software change prediction [8, 14] mechanism predicts those source code elements that are likely to be employed with some change from one version of software to the next. Software change prediction models might prove to be useful for project managers for the purpose of properly scheduling maintenance activities [28, 29, 33]. Sufficient allocation of limited resources to change-prone source code elements ensures that they are carefully designed and rigorously verified. Such activities would result in a good quality, easily maintainable and cost-effective software product.

Various empirical studies in literature have successfully developed software quality models to predict change-prone code elements of a software. These studies have explored a variety of software metrics, numerous classification algorithms and extensive software datasets for empirical validation. We analysed 50 such research articles related to software change prediction, published from the year 2000 to 2019. Table A.1 detailed in Appendix A summarises the studied articles on the basis of the metrics used, prediction techniques employed, validation methodologies and statistical tests used and major observations drawn from the empirical analysis.

We assessed the 50 articles given in Table A.1 in terms of the sub-sections as follows:

2.2.1 Independent variables for software change prediction

Software metrics are related to measures or parameters of quantitative assessment used for measurement purposes to produce better product and improve its related process. The metrics used in the milieu of software change prediction can be broadly separated into the following five groups [14]: Structural metrics, Network metrics, Evolution-based metrics, Word vector metrics, and Developer related metrics. These belong to the category of either Product or Process metrics.

2.2.1.1 Product metrics

The metrics related to products are called product metrics. A product can be any deliverable produced during the software development.

- The literature on software change prediction demonstrates the heavy usage of *structural metrics* in the form of source code design metrics, which depict the structural attributes of a class such as its inheritance, cohesiveness, size, etc. Many such metric suites have been proposed in literature such as Chidamber and Kemerer (CK) metrics suite [17], Quality Models for Object Oriented Design metrics suite [18], Lorenz and Kidd metrics suite [19] and many others.
- The *network metrics* [32], on the other hand, are extracted from the dependency graph of the software and identifies files which are "more central" and are more likely to change, e.g. Degree centrality, Closeness centrality, Reachability, etc.
- Only one study [56] in the literature has been observed to work with *word vector metrics*. These metrics quantify the terms used in the source code and their names by using bag of words approach.

2.2.1.2 Process metrics

The metrics related to the process are known as process metrics. A process uses the products and produces new products used by some other process or activity. Following kinds of process metrics were found in the literature studied in Table A.1:

- *Evolution metrics* [6, 7, 28-30] characterizing evolution history of a class, i.e. release by release history of how a class has evolved in previous versions. e.g. Birth of a Class, Frequency of changes, Change density, etc. and

- *Developer metrics* [6, 7] quantifying various developer related factors such as entropy of changes introduced by a developer in a given time period, number of developers employed on a specific software segment in a specific time, structural and semantic scattering of developers in a specific time period, etc.

An analysis of primary studies conducted in Table A.1 depicted that product metrics especially structural metrics extracted from source code design have been widely used for software change prediction in literature. Also, it was observed that the CK metrics suite (consisting of Weighted Methods of a Class (WMC), Lack of Cohesion among Methods (LCOM), Coupling Between Objects (CBO), Response for a Class (RFC), Depth of Inheritance Tree (DIT) and Number of Children (NOC)) was the most commonly used structural metrics suite in primary studies. The CK metric suite in its entirety has been employed in twenty-six studies out of the fifty studies analysed. Even those articles [6,7, 28-30, 32, 56] that proposed process metrics like evolution-based, network, developer-related, etc. assessed and compared their proposed predictors with structural metrics as they are well established and successfully used by numerous studies. Apart from the CK metric suite, the SLOC metric (a measure of size) has also been used in fifteen studies included in Table A.1. If not SLOC, then other size measures like NOO (No. of Operations), NOA (No. of Attributes) etc. have been employed by authors to quantify size of the source code in order to predict software change.

Some studies do not use any of the standard metric suites. For, e.g., the paper by Askari and Holt [95] employed CVS logs of six open source software systems and introduced three probabilistic models to predict changes in the systems. Also, the empirical study conducted by Posnett et al. [25] used pattern roles as independent variable.

Only 14% of the articles surveyed in Table A.1 used process metrics. While some studies like those done by Elish et al. [28] and Malhotra and Khanna [30] used evolution-based metrics which characterize the evolution history of a class, Catolino and Ferrucci [6] and Catolino et al. [7] used metrics specifically depicting the development process complexity by quantifying developer related factors. It may be noted that authors in [28-30] advocated the combination of both process as well as product metrics for determining change-prone nature of a class. It may also be noted that certain studies [23, 26, 34, 46, 47, 52-54, 94] analyzed a large number of OO

metrics with respect to different dimensions (cohesion, coupling, size and inheritance) in order to obtain generalized results.

We also analyzed the granularity level over which these metrics were collected. The articles read in Table A.1 exhibit that the metrics have been usually calculated at a class-, method- or a file-level, and not at a project or a component-level even though software components have been employed as target projects for software change prediction. Five studies, those performed by Liu and Khoshgoftaar [49], Khoshgoftaar et al. [50], Giger et al. [32], Kaur and Mishra [61] and Zhu et al. [56] collected file-level metrics, one study by Romano and Pinzger [12] collected structural metrics at interface level and that done by Sharafat and Tahvildari [98] collected structural metrics at method level. However, all other studies analysed class-level metrics.

Other structural metrics such as probability of change based on inheritance, reference and dependency used by Tsantalis et al. [51] and instability and maintainability index used by Kaur and Mishra [61] have also been employed along with other structural metrics to predict software change.

2.2.2 Techniques for software change prediction

Prediction models require the aid of data analysis algorithms. Figure 2.3 depicts the percentage of primary studies using the various categories of algorithms. Ensemble algorithms were used by 32% of the studies, which were ensemble of several base learning algorithms. For instance, Elish et al. [33] used an ensemble of Multilayer Perceptron (MLP), Support Vector Machine (SVM), Genetic Programming (GP), Logistic Regression (LR) and k-means techniques which were aggregated using majority voting. According to Figure 2.3, ML algorithms are the most popular category, followed by the statistical algorithms. The disadvantage of using statistical algorithms over ML ones is that the models developed using statistical techniques are not easily interpretable [14]. Another disadvantage of statistical models is that they are highly reliant on data distribution and are based on assumptions which may not be fulfilled by the software product data whose change-proneness is to be predicted [14]. Out of the 50 studies, three studies did not use any specific algorithm but predicted classes using a certain set of equations, as done by Sharafat and Tahvildari in [98], by using a combined rank list as done by authors in [100] or by using random effect meta-analysis model [26]. Other ICM techniques like evolutionary classifiers and

PSO-based techniques were employed in 30% of the articles, albeit in conjunction with other ML classifiers.

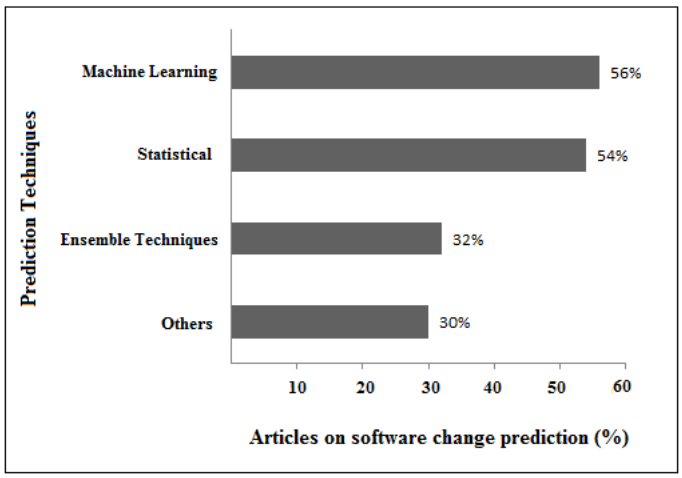

Figure 2.3 Percentage of primary studies using the various categories of algorithms

The ML algorithms can be further divided into several categories in accordance with Malhotra and Khanna [14]. Figure 2.4 states the that most popular categories of ML techniques that have been employed for software change prediction in the past literature are those summarized in column 3 of Table A.1.

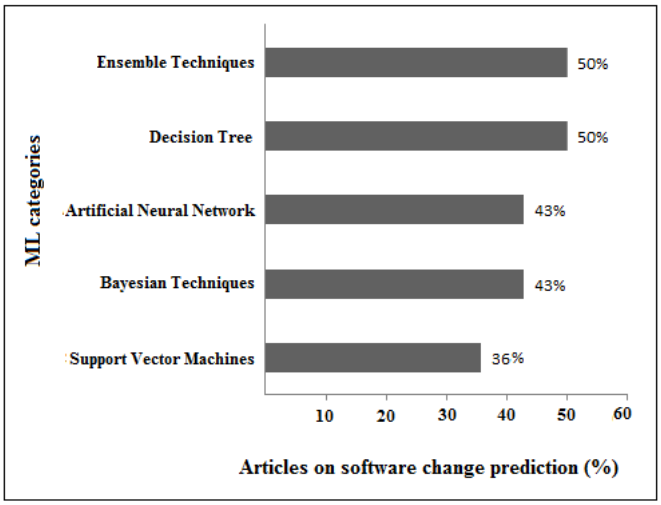

Figure 2.4 Percentage of primary studies using the various categories of ML techniques

These sub-categories are Bayesian algorithms, Decision Trees (DT), Support Vector Machines (SVM), Ensemble or meta-classification techniques, and Artificial Neural Networks (ANN). Other remaining algorithms such as the evolutionary techniques have been grouped into a miscellaneous category.

It was noted that Ensemble techniques and Decision Trees are the most popular categories of ML algorithms and are each employed by 50% of studies that have

employed ML techniques for software change prediction. The next popular category of techniques were ANNs and Bayesian algorithms each used by 43% of the studies. ANNs are capable of modeling complex non-linear relationships and are adaptive in nature making them suitable for change prediction tasks. A Bayesian classification algorithm like the Naïve Bayes classifier performs better compared to other models like logistic regression when assumption of independence holds.

Additionally, as previously elaborated in [14], ML techniques like Bagging (BAG), Random Forest (RF) and ADaBoost (ADB) have been observed to obtain median values of 0.76 and higher in the past literature wherever assessed for software change prediction. These median values are reported to be higher than other employed ML techniques for software change prediction in the past literature and could be due to the fact that the BAG and ADB techniques belong to the ensemble category of ML algorithms. Therefore, their effective predictive capability is a result of aggregation of results of several base models. Moreover, although being a DT classifier, RF draws random bootstrap samples from the training set in addition to drawing random subsets of features for training the individual trees; similar to what is observed in BAG. Due to the random feature selection, the trees are more independent of each other compared to a regular BAG technique, which often results in better predictive performance (due to better variance-bias trade-offs) [14, 47].

2.2.3 Validation methods

Literature studies have been using different techniques of validation to build software change prediction models that can be loosely classified into intra-project and inter-project validation methodologies.

Intra-project validation methodologies are of two kinds: Intra-release and Inter-release validation. Intra-release validation models use training and testing data of the same software project version to validate the model's performance. Inter-release validation models use training and testing data of the same software project. The training data used by the model is obtained from the previous versions of the same project and is validated on the later versions. On the contrary, in inter-project validation, the prediction model is trained using data from one project (say Project A) and is validated on another project (Project B). Inter-project validation is useful in case historical data of the same software project is not available.

An analysis of Table A.1 reveals that the majority of studies developed software change prediction models using either a hold-out validation, K-fold cross validation or Leave-one-out Cross Validation (LOOCV), all of which are intra-project validation approaches. A summary on the number of articles that have employed a validation strategy has been provided in Figure 2.5.

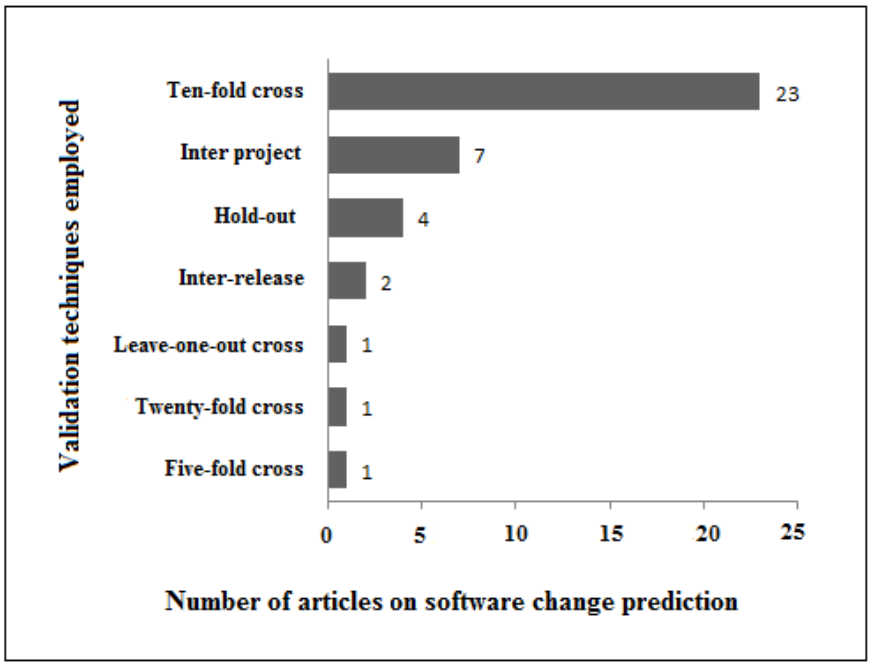

Figure 2.5 Number of primary studies using the various validation strategies

- Only one study conducted by Elish et al. [33] used LOOCV, this method requires N iterations should there be N datapoints in the dataset. In each iteration, all data points except one are used as training instances. The remaining data point is used for validation. It is ensured that all data points are used at least once for validating the developed model.
- On the other hand, K-fold cross validation is the most popularly used validation methodology and is employed in twenty-five studies of Table A.1. The most frequently used value for K is 10. One study [63] used K = 20 and another study [59] conducted the validation using K = 5. The whole dataset is randomly split into K parts, which are nearly equal in size. Thereafter, K iterations are performed. In each iteration, only one partition is excluded for validation, while all others are used for training the model. We found that k-fold cross validation is the most popular validation method as it provides the mean results obtained in various partitions, thereby reducing variability. As a

result, the data is insensitive to the created partitions as in the case of hold-out validation.

- Hold-out Validation has been employed by four studies [29, 33, 49, 50]. During a hold-out validation, data points are randomly split into testing and training sets using a specific ratio. One of the most common ratio used for partitioning is 75:25. In such a case, 75% of data points are used while training and the remaining 25% of data points are used while validation. However, the method has high variability due to random division of training and test sets. The points which make the training and test sets may affect the performance of the developed model.
- Also, inter-version/ release validation, where different releases of the same dataset are used for training and validation was used by two studies [13, 48].
- Apart from intra-project validation, inter-project validation was used by seven studies [13, 34, 47, 55, 58, 60, 103].
- Time dimension was also employed as a means to validate a developed change prediction model in only one study conducted by Catolino et al. [7]. The authors used a three month sliding window to train and test the software change prediction models as the developers metrics used by the study encapsulate developer dynamics in a given time period.

2.2.4 Statistical tests used for validating the results

In order to produce accurate conclusions, statistical confirmation of the results of a study is necessary.

In thirty out of fifty studies in Table A.1, authors employed parametric or non-parametric statistical tests to confirm their findings. In the thirty studies, seven types of non-parametric tests and six types of parametric tests have been applied. We can observe that most studies prefer a non-parametric test to a parametric test. This pattern has been noted as parametric tests involve strict suppositions to be accommodated before applying them such as distribution of the population to verify the assumptions of normal testing. Although these aspects make parametric tests effective versus non-parametric, tests they are more difficult to implement. It is easy to understand and use non-parametric steps. Therefore, the scientific community supports them.

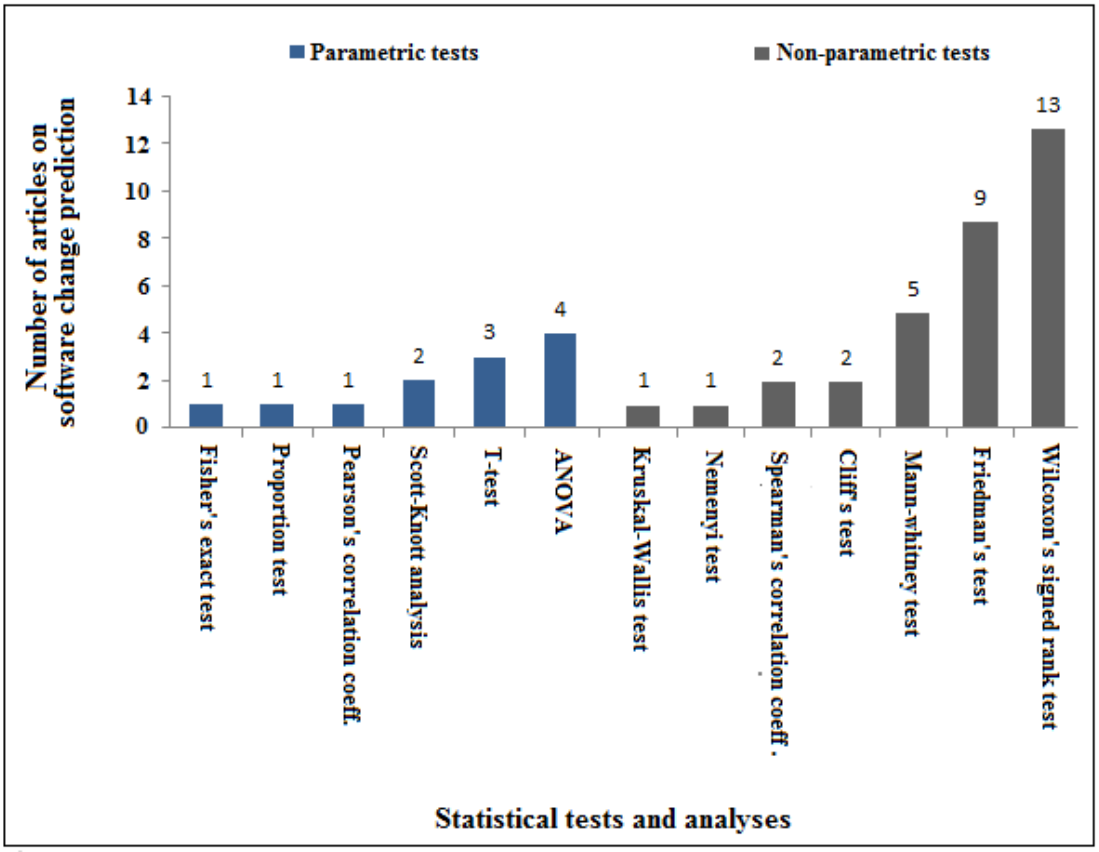

Figure 2.6 Number of primary studies using the various statistical tests and analyses

As seen in Figure 2.6, the most popular non-parametric tests employed by researchers for software change prediction are observed to be the Wilcoxon signed rank test, Friedman test, Mann-Whitney's test used by 13, 9, 5 studies respectively. Although the Wilcoxon signed rank test can be used individually for pairwise comparisons, it has also mainly been employed as a post-hoc test after the application of Friedman test for six out of thirteen studies in Table A.1. On the other hand from the parametric tests category, tests like ANOVA and the t-test have been used more frequently than the others, with ANOVA being used in four research articles and t-test being employed in three articles.

In summary, the extent of research carried out on software change prediction provides sufficient testament to its importance because organisations are now completely dependent on their software systems. Their systems are critical business assets and they must invest in system change to maintain the value of these assets. The existing research articles elaborated and discussed in this section and in Table A.1 on software change prediction claim to solve the key problem for organisations revolving around

predicting change to their existing systems in order to effectively support their business operations.

Having investigated the literature on software change and software change determination, the next section elucidates the cognitive complexity metric and the research background that exhibits the status of cognitive complexity with respect to software development in literature.

2.3 Role of cognitive complexity in software development

The metrics proposed vis-à-vis measurement of software complexity have been often viewed either from a machine-oriented or a human-oriented perspective of software engineering. According to Wang and Shao [81], *cognitive complexity* is a metric measuring the psychological and cognitive complexity of software in the form of a human intelligence artifact.

Complexity measurement has always been a focal point for the research practitioners ever since the commencement of software engineering. Beginning from the control flow-based and considerably harangued McCabe's Cyclomatic complexity [20, 21] to Halstead's science measures [20] which take the operators/operands into account to the structural Fan-in and Fan-out measures [104] which calculate the outbound and inbound dependency of a function, academics have mostly attempted to quantify complexity by means of design aspects. From the results of an online survey gathered from 100 respondents at two universities and seven companies, authors [105] established that software complexity is observed to strongly and negatively influence certain internal quality characteristics like readability, comprehension, and adaptability of code.

Some researchers have also attempted to investigate the cognitive aspect of complexity and the influence that it could have on different programmer tasks (e.g. understanding, maintaining, and testing the code). Cant et al. [106] proposed that for analyzing the procedures concerned with programmer tasks, one should follow the laws of cognitive sciences which map the mental tasks connected to programming into "chunking" and "tracing".

The use of Cognitive Informatics (CI) with respect to software development was first identified by Wang and Shao [81] who modelled one of the central software qualities,

complexity, via the examination of the cognitive weights of Basic Control Structures (BCSs) and formulated a novel theory of software Cognitive Functional Size (CFS) to estimate the cognitive complexity of software. Comparative case studies to test the robustness of CFS were performed using three programs based on the same algorithm developed using Pascal, Java, and C. Results indicated that the cognitive complexity measure (CFS) has a higher degree of robustness than the physical-size metric and is not dependent on language/implementation, since the program implementations conforming to the same algorithm specification had the same CFS, contrary to measures like LOC where the number of lines of code needed to develop a program might may vary vastly from programmer to programmer and from language to language. This cognitive complexity measure, CFS, calculated in terms of Cognitive Weight Units (CWU) was also observed to satisfy eight out of nine Weyuker's properties [107] , thus providing adequate theoretical validation for the same.

This metric essentially prompted the research in cognitive informatics-based software complexity measurement. Kushwaha and Misra [108] proposed the Cognitive Information Complexity Measure (CICM) as the product of CFS and the weighted information count which was measured in terms of the number of operators and identifiers in the software. Analysis of CICM was performed by applying it to 15 'C' programs. These 'C' programs were also given to a group of 25 students who were asked to analyze it and decipher the problem that the program dealt with. Time taken to comprehend the program was noted. Results indicated that there exists a linear relationship between the time expended in understanding a given piece of code and its cognitive complexity, thus modeling the comprehension strategy of the user. The CICM metric was also observed to fulfil all the nine Weyuker's properties [109]. Later Misra [110] modified the CFS measure into Modified Cognitive Complexity Measure (MCCM), which eased the complex weighted information count of every line in CICM by measuring it as the number of instances of operators and operands. MCCM was also validated [111] using the principles of measurement theory and its calculation was demonstrated using various programs written in 'C' language. Furthermore, an all-inclusive metric called as the Unified Complexity Measure (UCM) was proposed by Misra and Akman [112] which took into account all the internal attributes precisely influencing the complexity, for example: the number of control structures, number of lines, function calls, and total occurrence of operators

and operands. The authors employed eight different ‘C’ programs for the comparative scrutiny of the UCM methodology with other four different complexity measures like CFS, Halstead’s effort, Cyclomatic complexity and statement count.

In all the previously introduced complexity metrics, the factors or the BCSs were quantified for the measurement of complexity without considering the dependencies among them. Auprasert and Limpiyakorn [113] therefore proposed the Structured Cognitive Information Measure of software (SCIM) which attempts to classify the information contained in the source code in accordance with the dependencies transpiring within the BCSs. The proposed measure was theoretically validated using the Weyuker’s properties, in which all the nine properties were satisfied. Multiple Java programs were also selected to comparatively evaluate SCIM to existing metrics like LOC, Cyclomatic Complexity, Halstead’s Effort, CFS, and MCCM.

The most recent improvement to the CFS metric has been proposed by Crasso et al. [82] in which the authors include six novel BCSs, four of which examine the special operators of the Object-Oriented (OO) standards, whereas the remaining two take the Java language operators into account for handling exceptions. The authors also present a theoretical validation of this extended measure with respect to a framework specifically devised for validating the software complexity measures [114] and the appropriateness of the metric is demonstrated by exhibiting it in ten Java projects.

Overall, from the literature on the cognitive complexity metric, it is observed that most of the studies effectively propose and, at best, theoretically validate the various cognitive complexity measures. Furthermore, there exists a severe lack of related work empirically testifying the applicability of the cognitive complexity measure on any of the software development phases. However, it has been often argued that it is challenging to modify code fragments from existing software when it comprises files that are difficult to comprehend. Since systematic software maintenance, which often involves software change, includes an extensive involvement of human activity, therefore cognitive complexity is one of the intrinsic factors that could potentially contribute to or impede an efficient software maintenance practice, the empirical validation of which remains vastly unaddressed.

2.4 Application of Semantic web technology, Ontology, in the component-based development scenario

Even though software components have vast benefits, their use in practice has been relatively low for the following reasons :

• Firstly, there is no clear protocol to specify a OTS component's description, configuration, integration, and modification, so that it can be accommodated within third-party system requirements [64].

• Secondly, due to the absence of proper query methods and techniques, OTS component integration also involves the problem of locating and retrieving such components which from reuse repositories that fulfil the user's requirements [115].

A suitable solution to this lies in Semantic web technologies [116] which "help machines comprehend additional information located on the Web to make them support automation of tasks and richer data discovery, data navigation and integration". In other words, Semantic web contains the ability to methodically express the intended semantics and aids in automated reasoning, supporting, sharing, integration and management of information from heterogeneous sources [117]. These capabilities of Semantic web perfectly satisfy all those general requirements of exploring, retrieving and describing the software component relationships in reuse libraries.

We conducted a systematic literature review (SLR) to describe the latest state-of-the-art Semantic web technology, Ontology, employed to ease and improve the process of exploring, retrieving and describing software components. To identify the most important studies in accordance with relevance and quality, we have performed the search in four primary digital libraries. The SLR conducted in this research work is performed in accordance with the guidelines proposed by Kitchenham, 2004 [118].

2.4.1 Ontology in Semantic Web: a brief background check

The notion of Semantic Web was introduced by Tim Berners-Lee and was made popular with the help of the World Wide Web consortium (W3C). It was due to the Semantic Web that computers could deal with the information on WWW, understand and couple it, to help humans find required knowledge.

Often considered as the premise of the Semantic Web, an ontology [119] can be interpreted as a collection of logical axioms constructed to explain the intended meaning of a vocabulary, to express something significant within a particular area of interest and promote sharing of information [120]. XML [121] is the commonly used format for data and documents on the Web. Also proposed by W3C, is the Resource Description Framework (RDF) [122], a data model based on the concepts of graph which employs URIs and has several different syntax, including XML.

- Ontology makes use of *Ontological Languages* [123] like OIL (Ontology Inference Layer), DAML+OIL (DARPA Agent Markup Language + OIL), CycL, OWL (Web Ontology Language), etc. for representation of knowledge. Among all the ontological languages presented, OWL [124] has been the most popular one. OWL provides an additional vocabulary with the help of formal representation of semantic. This makes it support greater interpretability of the content on the Web by the machine than that provided by RDF, XML and RDF Schema (RDF-S). It inherits the advantages of DAML+OIL and provides the same level of reasoning and sharing capabilities. Contrary to other languages, OWL also provides built-in versioning functionalities. It supports a rich expressive power and achieves scalability due to a layered architecture.
- Ontology employs a variety of *Ontology editing tools* [125] for ontology editing and development. However, Protégé [126], an integrated and platform-independent system for maintenance and development of knowledge-based systems, has been used repeatedly for ontology editing. Protégé consists of a frame-based knowledge model, ensuring compatibility with OKBC (the Open Knowledge-Base Connectivity protocol) which enables interoperability with other knowledge-representation systems.
- *Reasoners* determine the quality and correctness of an ontology. A high quality and correct ontology is devoid of inconsistency and uncertainty. Out of all the ontology reasoners, RACER [127], the first OWL reasoner to be developed; Pellet [128], an OWL-DL reasoner based on Java, and FACT ++ [129] (an updated version of Fast Classification of Terminologies) which supports OWL and uses the principles of description logic have gained larger acceptance.

- On the basis of ontology languages and reasoners described above, many *query systems and languages* have already been developed. When ontologies are employed in collaboration with Semantic web rules, SQWRL [130] is employed as the query engine.

Apart from the Ontology, other two major constituents of Semantic web technologies are the Semantic Web Services and Linked Data. Semantic web services [131] are used for machine-to-machine communication through the World Wide Web, much like the usual Web services. Whereas, Linked Data [132] pertains to the technical interoperability of data, empowering data connection from a variety of sources and provides the best approaches for data publishing on the Web. Linked Data is one of the steps to achieve the objective of creating a Web of Data or in other words, the vision of the Semantic web.

The above-discussed Semantic Web technologies are the most popular ones and Section 2.4.3 depicts how they have been widely incorporated with respect to software components. A brief discussion on the research methodology followed to review the application of Semantic Web in CBSE is presented in the next section.

2.4.2 Research methodology to perform the review process

The following steps were undertaken to conduct the review process:

2.4.2.1 Developing the review protocol

The review was performed by identifying the appropriate research articles (pertaining to the domain of interest with the help of a keyword-based search) from the various electronic databases. The inclusion and exclusion criteria were applied post this step to narrow down the identified articles after which analysis of the articles, selected on the basis of the research questions formulated, was done. Finally, the results drawn from this analysis were compiled.

2.4.2.2 Research questions

This systematic review aims to determine and analyse the existing literature concentrating on the application areas of Semantic web tools and techniques when applied to the field of CBSE. In order to execute the review in an effective way, a set of research questions were formulated as stated below:

- What has been the year wise status of publications since the inception of the amalgamation of Semantic web and CBSE?
- Which aspect of CBSE has been most catered to by Semantic web?
- What Semantic web technique has been used the most?
- What are the frequently used tools and languages?

2.4.2.3 E-databases as sources of information

Before the commencement of the search process, a suitable set of electronic databases were selected so as to augment the chances of finding proper articles related to the domain of search. The following databases were selected for finding the research articles: ScienceDirect (www.sciencedirect.com), ACMDigital Library (www.acm.org/dl), IEEE eXplore (ieeexplore.ieee.org) and Google Scholar (scholar.google.co.in/).

2.4.2.4 Search, inclusion and exclusion criteria

A comprehensive and methodical keyword-based search strategy was undertaken to gather relevant data from various e-databases as shown in Table 2.1. The keyword 'component' led to a high number of irrelevant research articles. Also, the term 'semantic' retrieved research articles on linguistic semantics as well. Based on titles and abstracts, inapposite papers like these were excluded with the help of a manual elimination as shown in Figure 2.7. Qualitative and quantitative research articles written in English and published including and up to 2018 beginning from the commencement year of Semantic web i.e. 2001 were included to ensure that the review was performed in its entirety.

The total search results were brought down to 119 research articles on the basis of their titles, and further reduced to 76 research articles on the basis of their abstracts. Following this step, these 76 research articles were analysed in their entirety to settle on a final list of 54 research articles on the basis of the inclusion and exclusion criteria.

2.4.2.5 Threats to validity

The most crucial threat to validity is concerned with the fact that the papers reporting on empirical work related to the topic of our concern might be published in other venues as well. We did not search for papers on an issue-by-issue basis and through a

manual reading of titles of all journal-published research articles. This means that we probably have excluded some papers relevant to the topic of concern which are

Table 2.1 Keyword-based search strategy

Information Source	Keywords	Time-Frame	Publication type	No. of results
www.sciencedirect.Com	*Abstract+title+keyword*: semantic web used in software component	2001-2018	Journal publication	23
	Abstract+title+keyword: ontology used in software component	2001-2018	Journal publication	58
	Abstract+title+keyword: semantic web service used in software	2001-2018	Journal publication	19
	Abstract+title+keyword: linked data used in software component	2001-2018	Journal publication	53
ieeexplore.ieee.org	*Abstract:* semantic web used in software component	2001-2018	Journals and magazines	13
		2001-2018	Conference publications	77
	Abstract: ontology used in software component	2001-2018	Journals and magazines	9
		2001-2018	Conference publications	148
	Abstract: semantic web services used in software component	2001-2018	Journals and magazines	8
		2001-2018	Conference publications	43
	Abstract: linked data used in software component	2001-2018	Journals and magazines	17
		2001-2018	Conference publications	165
www.acm.org/dl	*Abstract:* semantic web used in software component	2001-2018	Journals	-
		2001-2018	Conference proceedings	17
	Abstract: ontology used in software component	2001-2018	Journals	2
		2001-2018	Conference proceedings	23
	Abstract: semantic web services used in software component	2001-2018	Journals	-
		2001-2018	Conference proceedings	10
	Abstract: linked data used in software component	2001-2018	Journals	-
		2001-2018	Conference proceedings	5
scholar.google.co.in/	Semantic web used in component based software engineering	2001-2018	All sources (Journals, conference proceedings, patents, book chapters etc.)	Over 88,000

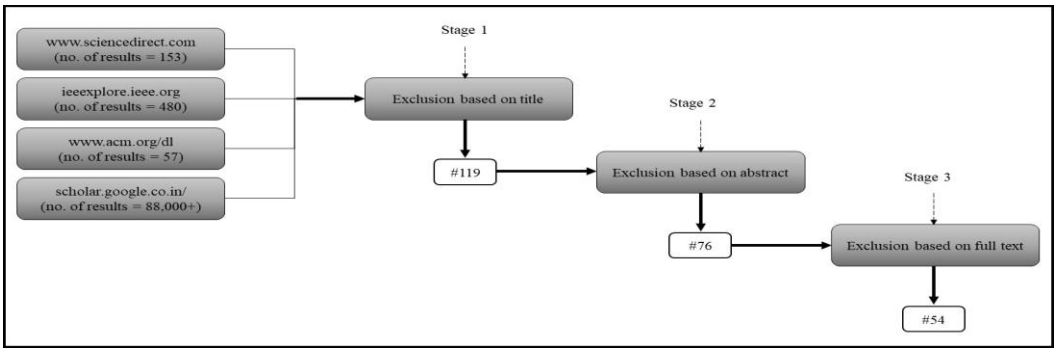

Figure 2.7 The exclusion process to select the papers for review

located in some conference proceedings or journals. Conference papers were not excluded because a majority of the work is mostly published in conference proceedings. However, research articles which did not associate CBSE and Semantic web in a significant manner or reprised the work already done with slight modifications were not included.

2.4.3 Literature survey: Semantic web technology for component-based software engineering

To enhance understanding, the literature review of the research articles is presented in a tabular format (Table A.2 in Appendix A) along the categories of: *Source*-the author and the year of publishing, *Technology incorporated*- Semantic web Technology used, *Tools used*- in relation to the Semantic web technology, *Purpose*- what aspect of CBSE does the Semantic web technology and its tools cater to, and *Outcome/Conclusion*.

2.4.4 Discussions of Research Questions

This section constitutes of a discussion and an assessment of what has been presented in Table A.2. This assessment will be performed from the component based software engineering's perspectives and attempts to answer the research questions posed earlier in Section 2.4.2.2.

2.4.4.1 What has been the year wise status of publications since the inception of the amalgamation of Semantic web and CBSE?

Twenty one journal papers and thirty-three conference proceedings were systematically analyzed in Section 2.4.3 from 2001 to 2018. Figure 2.8 is a line-graph plotting the year of publication on the x-axis and the number of papers published in that year on the y-axis for papers in review(mentioned in the table), thus representing the year-wise status of publications since the inception of the amalgamation of Semantic web and CBSE. As seen in Figure 2.8, the number of papers involving the application of semantic web in CBSE have varied significantly from year to year thereby resulting in the formation of drastic peaks and dips in the line graph. Considering the year 2009 to be the midpoint, 50% of the research articles have been published in and before the year 2009 where as 50% of the research articles have been published after the year 2009. It is evident that the popularity of Semantic Web

technology for CBSE was highest in the year 2006, and has been on a consistent irregularity over the years. Thus one can recommend that in order to analyze and access the most relevant papers, papers published after year 2003 should be examined by researchers to reach the most germane information.

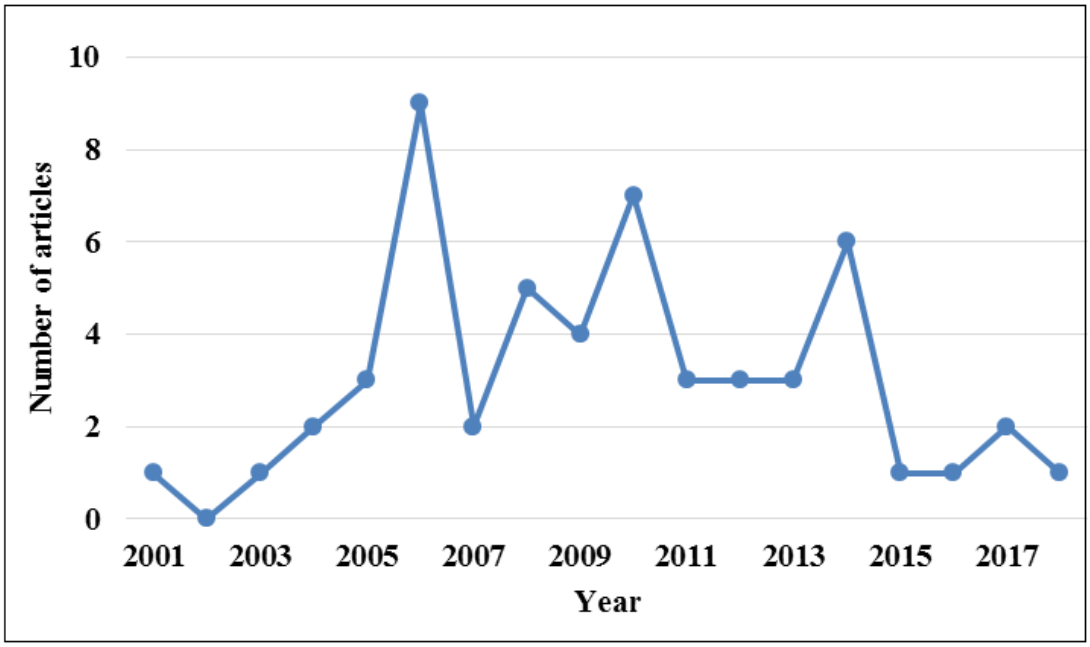

Figure 2.8 Distribution of papers according to year

2.4.4.2 Which aspect of CBSE has been most catered to by Semantic web?

Out of the 54 research articles studied in Table A.2, 47 of them were directly useful in catering to at least one of the processes of CBSE. As shown in Figure 2.9, the CBSE process constitutes of six phases:- the Requirement analysis, System and software design, Implementation and unit testing, System integration, System verification and validation, and finally the Operation and maintenance phase.

i. Requirement analysis

4% of the total articles studied in Table A.2 find their relevance in the process of requirement analysis in the form of reuse of the component requirements. Apart from workflows, another important by-product of software development process is the requirements. Karatas et al. [166] proposed a method by which systematic reuse of software requirements can be made using an automated ontology based graphical user interface tool. This tool disables the user from configuring a product violating domain constraints or domain scope. Rastgoo et al. [167] presented another approach to

promote the automatic generation of software component requirements in the form of an ontology using the UML diagrams. Abbasipour et al. [67] presented a model based technique in which the requirements stated by the user are decomposed into configuration (lower-level) requirements using an ontology comprising of the traceability links between the desired system and the user requirements. An ontology is used to specify the particulars about the decomposition of different functionalities and a separate model is used to define the functionalities provided by the available component types.

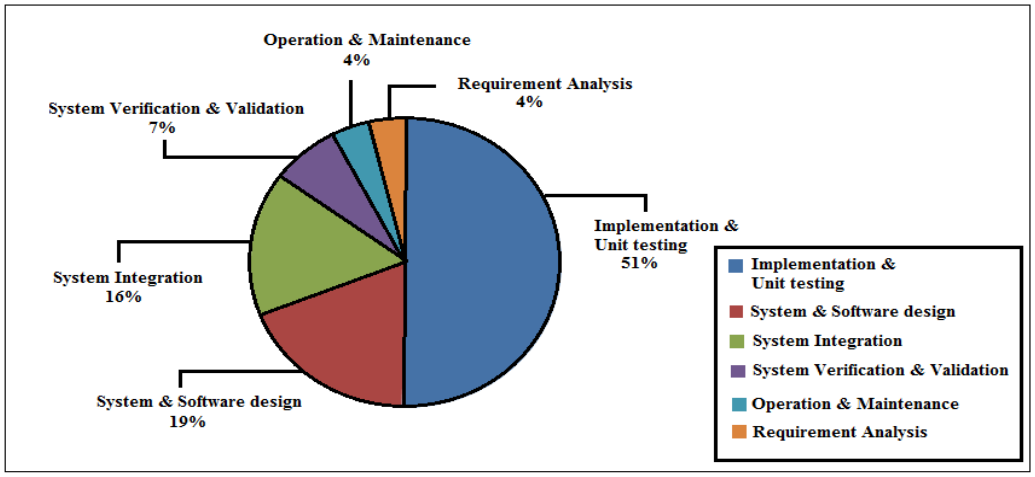

Figure 2.9 Distribution of the literature survey according to the phases of CBSE

ii. System and software design

19 % of the research articles contribute to the System and software design phase in which the architectural/model compliance of the desired component with the target system is sought to avoid the interoperability related issues. It must be noted that, though less in number, exceptional work has been carried out relevant to this phase, with authors exploiting the descriptive benefits of semantic web to ease the working of software component registries like ebXML and repositories, and validating the software component models and architectural styles. Efforts have made to simplify and better the management and architecture of the software components using Semantic web approaches. Software component registries are one of the means to manage software components effectively since they provide mechanisms for every software component to be assigned a unique numeric address and developer given public names. These mechanisms help in distinguishing it from all other components participating in a system. Ontologies have been employed by authors in order to supplement the working of software component registry. Song et al. [138] presented

an approach to enhance the semantics of the ebXML software component registry by exploiting the benefits of an ontology-based representation. On the other hand, since the current registration standards are syntactic in nature and are weak in terms of describing the logical relationship between the various heterogeneous information resources, Liu et al. [76] proposed a mechanism by virtue of which ontological languages and reasoners could be used to develop complex information registry knowledge base which would help in realizing the complex information resources interoperability at a semantic level. Talevski et al. [150] proposed a methodology which could be used to express the communication architecture of a component-based software in which the component specifications, compositions, relationship constraints between components, interfaces and their interconnections (connectors and adaptors) are expressed using an ontology. This explicit ontology based model provides the features of navigation, querying, and manipulation of the services provided by the software components. Zhang et al. [152] proposed an ontology-based knowledge base for dynamically validating software architectural styles and component configurations, focusing on modelling the software architectural constraints. In order to verify the architectural design of a newly proposed component model, Wang and Sun [83] used ontology and Semantic Web Rule Language (SWRL) to describe the relationships, dependencies and constraints between its entities. The authors validated the proposed approach by modelling the computation units, methods, connectors and the components as OWL classes to achieve their desired task. Dai et al. [164] proposed a multi-layered ontology model to validate software component architecture function blocks. Jayasudha et al. [168] highlighted that software companies produce lots of reusable components or artefacts which should be stored in a structured way. The authors made use of two separate ontologies: domain ontology- to understand the domain of the project and, the folder ontology- to create the index of the actual storage of data which helps in easy retrieval of the corresponding component.

iii. Implementation and unit testing

Out of the 45 papers in consideration, 51 % of the papers contribute to the implementation and unit testing phase of CBSE. This sector is primarily composed of the research articles that involve the use of Semantic web as far as specification,

description, selection and retrieval of the desired software components is concerned. For instance, authors like Sugumaran and Storey [75] presented an approach which exploits the domain knowledge in domain models and ontologies in order to select a component from a reuse repository. Li et al. [78] identified that for an effective component selection process, an evolvable ontology-based component repository (in-house or open commercial) is to be developed. On the other hand, based on the argument that if we add machine-processable semantic intelligence to the components and publish this information systematically using standard techniques, classifying, searching, and selection of the software components from the repository will significantly improve, Zygkostiotis et al. [79], presented a Semantic web-based approach for the purpose of annotation, publishing and locating of the reusable Java software components. Java source code domain ontologies in OWL-DL are annotated and in order to publish and locate these semantic descriptions, the authors use a Description Logic (DL) based semantic registry to perform matchmaking between the component advertisement and request. Unit testing of the selected components is vital because the component testing performed in isolation is not adequate and although the components themselves are correct, often the assembly of correct components may be incorrect. The unit testing of a software component, however, has not been analyzed much with the help of Semantic web technologies.

iv. System integration

The system integration phase of CBSE involves component integration and system testing and might lead to minor customization of the component so as to be accommodated into the whole system, resulting in the final version of the desired software system. 16% of the research articles in consideration contribute to the smooth composition of components. Paulheim and Erdogan [154] presented a prototype framework to facilitate the smooth composition of heterogeneous User Interface (UI) components using RDF and ontology. Ontologies are primarily used to provide a universal representation of both the UI components and the component based system along with the information which they process, thereby minimizing dependencies between integrated components and facilitating data exchange between them. Attempts were also made by authors to remove the conflicts which arise during the process of component integration. Kzaz et al. [158], presented an approach for business component integration using the concept of ontology alignment. Ontology

was incorporated to detect the naming conflicts which arise during the integration process of conceptual business components, which are candidates for reuse in an information system project. Verification methods for component-based systems and component matching are difficult to add to the standard software development process because these can require specialized expertise, could be error-prone, and time-consuming. As a solution to this, Castillo-Barrera et al. [84] proposed a Semantic web-based verification process for syntactic and behavioural interface contract conformance of a component with respect to the target system. The invariants, pre- and post-conditions of the interface contracts are expressed as classes, properties, and relations in an ontology which is checked for consistency using a reasoner. Paquette and Masmoudi [153], presented an ontology-based framework for the purpose of software component aggregation to assist the programmers during the process of software development. This involves the metadata extraction from the software component packages that the developers want to aggregate and uses an ontology to categorize the kind of aggregation (coordination, collection or fusion) to be done on the prospective components. Khemakhem et al. [80] proposed a mechanism called SEC++ employing ontologies at the phase of component discovery and integration. The software components are enriched with semantics and a shared ontology is created by mapping components and other applicable concepts. This shared ontology acts as a knowledge repository for software components and provides ontological heuristics so as to facilitate the dynamic and automated component integration. Castillo-Barrera et al. [165] present an ontology-based software domain metric to measure the amount of applicability that a software component possesses with respect to a specific domain, based on the vocabulary used in that domain.

v. System verification and validation

However none of the articles, reviewed in the table, have examined the success of the integration techniques with an overall system test. Standard testing and verification methodologies are employed in the system verification and validation phase. 7% of the research articles investigate the privacy concerns and the chances of vulnerability that might arise after the component has been incorporated into the system. Wu et al. [71] presented an approach for analysis of the vulnerabilities in software code component repositories using ontologies and used the concept of Common Weaknesses Enumeration (CWE) and Common Vulnerability Enumeration (CVE) in

order to furnish a measurable, unified and interdependent set of software weaknesses. Semantic web services have been specifically used to aid in the testing of the software components in this regard. Jiang et al. [155] proposed a hierarchical software component interface model that supports component testing and reuse by extending the existing semantic specification of the testing details in the component interface using WSDL. In order to cater to the privacy concerns which might arise when components exchange data, Kost and Freytag [160] introduced an ontology- based system model in which, for the evaluation of privacy aspects, annotation of the system model is done. This annotation is performed using data classifications which map system data types to corresponding domain specific ontological concepts, which are further converted to instances using the rule-based transformations. These instances form the instances of the privacy extended ontologies and the domain- specific privacy ontologies, which are used to analyse the system's privacy specific characteristics. Li and Zhang [161] proposed a reusable test case knowledge management model to aid in reusing knowledge which enables the test engineers in retrieving and reusing test cases with flexibility. The reusable test cases are described using an ontology for representing the concepts set and its inner relationships and for the formal representation, axioms are used. As an aid to facilitate effective component-based software development, Francisco et al. [163] proposed the automatic generation of QuickCheck test models from its web services WSDL specification supplemented with OCL semantic constraints, which results in generation and execution of great amounts of automatically generated test cases useful for testing web services. These web services, in turn, are used to integrate software components and make the communication between them possible.

vi. Operation and maintenance phase

The Operations and maintenance phase is constituted of some steps that are similar to the integration process. An existing component is altered or a new version of the same component is added into the system in most of the cases. About 4% of the articles contribute to the Operations and maintenance phase to monitor the changes made to the software components and to establish valid and useful links between the various software component artifacts that help in maintaining the complete software system. Hyland-wood et al. [143], explored the use of ontology to monitor the information changes made to the software component and its metadata through a date-

time OWL object property lastModifiedAt which can be accessed through appropriate SPARQL queries. This would indicate when the component and its specifications were last modified or validated. Witte et al. [148], identified that semantic connection among software component artifacts like source code and documents could prove to be useful for maintenance. With the cross linking of the ontological representation of source code and documentation, traceability links could be established between the code and documentation, thus helping in the maintenance process and for the detection of inconsistencies between information present in the actual code vs natural language texts.

2.4.4.3 What Semantic web technique has been used the most?

Distribution of the Semantic web technologies which have been studied in Table A.2 is shown in Figure 2.10. Ontology has been, without a doubt, the most used Semantic web technology employed in CBSE albeit in association with other technologies. 81% of papers used Ontology. 13% of papers applied Semantic web services. Only 6% of papers have employed Linked Data.

i. Application of ontology in CBSE

Ontology has been proved to be useful in all the phases of CBSE since they inherit from various associated and precursory technologies, e.g. semantic nets, theorem provers, logics, deductive databases, conceptual modelling languages etc. and it is this combination of several technological features which render ontologies distinct in contrast to related technologies and explains its wide usage in CBSE as seen in Table A.2.

- Firstly, ontologies form a unique category of conceptual models which relies on easily understandable modelling constructs, thus acting as an aid for the specification and description of software components. Formalized conceptual modelling also facilitates the information exchange between implemented computational systems, fostering integration along with interoperability of software components.

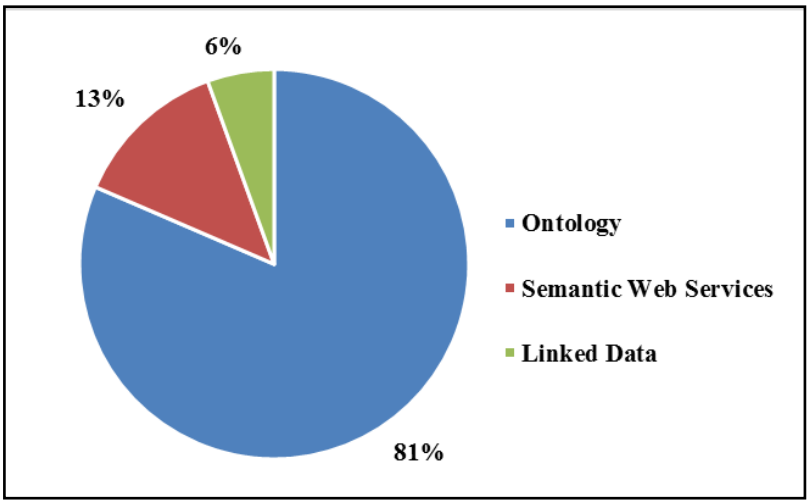

Figure 2.10 Distribution of the use of Semantic web technologies in software component-based tasks

- Secondly, ontology with its instances can be generated, altered, and populated by generic tools with editing possibilities. This provides for an unambiguous meaning of modelling constructs helpful to verify the software models and architecture. Also, for having a proof theory, a model theory stands as a prerequisite, and, thus, reasoning becomes an additional practical feature, facilitating the validation of the developed component-based software and to find the possible vulnerabilities and threats posed.
- Thirdly, since ontologies are declarative in nature, this renders them specifically acceptable for reuse. This implies that the effort of conceptually modelling a domain is undertaken once and the resulting ontology is eligible for reuse in several applications.
- Fourthly, with the recommendation of W3C, Semantic Web Activity provides ontological Web Compliance and allows for publication (making the components available online), querying (discovery of appropriate components) and annotation.

ii. Application of Semantic Web services in CBSE

Semantic Web services (SWS) have also helped in achieving certain tasks of CBSE. Although originally, SWS seek to achieve the automation of building the Web Service based applications using the Semantic web technology, web services are a logical evolution of software components and middleware and, Web services, therefore, are middleware and components in one [131]. In the research articles, the objectives of Semantic Web services, i.e. locating, mediation and web services integration, have

shown to be useful to accomplish the tasks of CBSE. Here again, ontologies are an inherent part of SWS and is a fundamental component in providing a solution to the semantic heterogeneity issue, thereby facilitating the semantic interoperability between disparate web services and applications. Ontology provides an extensive description of these terms and services and the associations between them in the application domain, progressing towards the acquirement of the knowledge domain in an explicitly representative manner.

iii. Application of Linked Data in CBSE

Linked Data, though not much, has been employed true to its purpose. Linked Data uses the Hypertext Transfer Protocol (HTTP) and the Resource Description Framework (RDF) for publishing structured data on the Web and to couple data present in various data sources, enabling data in two different data sources to be effectively linked to each other. Based on this, Linked Data aids in making the components specification (in OWL/RDF) available online in terms of linked datasets like DBpedia. Moreover, tools like DBpedia spotlight assist in an annotation of this specification, thus catering to the effective location and search of components on the web.

2.4.4.4 Frequently used tools and languages for the Semantic web techniques employed in CBSE

As far as the frequently used tools and languages are concerned, they have been employed according to the evolution of Semantic web technologies. This implies that there has been no particular preference of Semantic web tools and languages as far as their application to achieve the various tasks of CBSE is concerned. Although introduced in the year 2004, OWL has been the most vastly used ontology language till now for ontology development. OWL has three increasingly-expressive sublanguages [124] which have been designed to be used for specific purposes: OWL-DL, OWLFull and OWL Lite. Protégé continues to be the popular choice among the ontology editors. However, from Table A.2, it has been observed that Protégé has been preferred over other equally adept ontology editors such as the TopBraid Composer. A likely reason for this could be that Protégé provides flexible modelling components like meta-classes and is open source as opposed to TopBraid Composer. As seen in Table A.2, authors have not mentioned the choice of reasoner employed

for checking the consistency of the ontologies developed. SPARQL has been the preferred for querying the ontology. Also, since Semantic web technology is a relatively new area with respect to CBSE, there has not been much experimentation concerning the employment of the tools of Semantic web services and Linked Data in CBSE.

2.5 Observations drawn and gaps identified from the literature studied

Having reviewed the literature with respect to: software components, their code constituents and software change in Sections 2.1, software change prediction in Section 2.2, the role of cognitive complexity in software development in Section 2.3, and finally, application of semantic web technologies to software components in Section 2.4, we now summarize the complete literature studied topic-wise and highlight the gaps identified.

2.5.1 Software components and its code constituents

Component-based architecture focuses on the decomposition of the design of a software system into individual software components that communicate with each other via interfaces. These off-the-shelf (OTS) software components are developed outside their organization and provide the functionality and capability required. These can be in the form of a user-interface, a software model, software controller, a plugin etc. and can be independently developed, independently deployed, independently scaled and reused for use in other systems, services or products.

A software component consists of components like the source code files, executable files and an executable component model. All the three constituents are required to package the software component into a machine-readable software so that it becomes available for installation and execution in the target system.

2.5.2 Software change and software change prediction

Once a software component version or a software project is released, new requirements emerge and existing requirements change due to which the software undergoes changes. These changes might occur owing to software maintenance or software re-engineering or an architectural transformation.

Many kinds of software elements of a software project can be subject to changes which can range from requirements pertaining to architecture and design, to source code, documentation and test suites. Source code changes happening at a granularity less than a file are considered fine-grained changes. Such changes can be structural that alter the structure of the software and might alter the software behaviour as well, in addition to semantics-modifying and semantics-preserving changes.

A software change can be static, load-time or dynamic depending on when it manifests and even might result in a new software component that completely overwrites its old versions. However, dynamically versioned systems also exist that allow two different versions of one component being deployed simultaneously side by side, within the same name space. Additionally changes that are anticipated or predicted typically involve much less effort to implement than the unanticipated changes.

We performed an extensive review in Section 2.2 to analyze the current state of existing literature in the domain of software change prediction and to further identify research gaps in this domain. 50 primary studies were chosen and analysed along the lines of the software metrics reported and validated to be useful for change-proneness prediction, prediction techniques employed, validation strategies utilized and, the statistical tests incorporated.

The following observations were drawn from the literature survey from Table A.1:

- Product metrics especially the CK metrics suite have been the most extensively used in primary studies for developing software change prediction models. Nevertheless, the validation of process metrics that take the developer related factors into account and their combination with product metrics as a metric set for the prediction of change-proneness prediction is insufficient.
- Most of the datasets used by the primary studies have been open–source in nature and have been subjected to various balancing techniques before determination of the change-prone elements.
- It was observed that a majority of studies used ML algorithms and these algorithms are effective in the domain of software change prediction. However, as the quest for identifying the best classifier with respect to a certain prediction scenario is an on-going process [61], there are many

techniques in literature that have still not been scrutinized in regard to software change prediction. Additionally, as highlighted by Malhotra and Khanna [14], there exists limited data in literature which compares the performance of different algorithms for developing effective software change prediction models Moreover, the validation findings of different projects do not need to be similar and it is therefore imperative to assess the efficacy of the prediction techniques with respect to specific software projects [47].

- Furthermore, it was found that evolutionary techniques exhibited effective accuracy results. The authors in [14] encourage additional investigation of the effectiveness of such ICMs in the domain of software change prediction in order to yield conclusive results about their capability by comparing their performance with other established ML and statistical techniques.
- Intra-release validation has become a common standard while validating software change prediction models. Inter-project validation has also been investigated, however, studies have not extensively explored inter-release validation of the change prediction models. In an ideal scenario of model validation, empirical studies need to have an inter-release validation wherein historical data from the previous versions are utilized for training a model which in turn predicts change-prone source code elements of a new release. Yet, almost all the previous articles only employ a k-fold cross validation (intra-release) or an inter-project validation to validate the results of the prediction models.
- The results indicate that a majority (62%) of the articles on software change prediction employ statistical tests and analysis to obtain additional pragmatic validation with respect to the obtained prediction model results. This is advisable and should be pursued in future studies as well. However, most of the studies opt for non-parametric tests over parametric tests. This could be due to the fact that parametric tests assume underlying statistical distributions in the data. Therefore, several conditions of validity must be met so that the result of a parametric test is reliable. For example, the t-test for two independent samples is reliable only if each sample follows a normal distribution and if sample variances are homogeneous. On the other hand, non-

parametric tests do not rely on any distribution and therefore can be applied even if parametric conditions of validity are not met [175].

- However, there was no assessment of the normality of the data in most of the studied software change prediction articles before selecting and applying the appropriate statistical tests. Although non-parametric tests are valid in a broader range of situations (fewer conditions of validity), parametric equivalent of a non-parametric test has more statistical power than the latter [175]. In other words, a parametric test is more able to lead to a rejection of a null hypothesis since, most of the times; the p-value associated to a parametric test will be lower than the p-value associated to a nonparametric equivalent one that is run on the same data.

2.5.3 The role of cognitive complexity in software development

As observed in the literature read, most of the studies effectively propose and, at best, theoretically validate the various cognitive complexity measures and therefore there exists a severe lack of work that pragmatically establishes the applicability of the cognitive complexity measure on any of the software development phases.

Although the past literature includes various studies such as those conducted in [61, 98, 102] that establish associations between complexity metrics and change-proneness, a comprehensive and pragmatic validation based on real world software engineering projects has not been conducted on the cognitive complexity metric and predictive models have not been generated particularly for studying the relevance of the cognitive complexity as a predictor of source code change. Also, the literature that provides an appraisal of the cognitive complexity metric with respect to other variables belonging to the software change prediction milieu is extremely scarce. We feel that it is only because the cognitive aspect involved in software development has been largely ignored and not that the cognitive complexity is weak as a measure.

2.5.4 Application of Semantic web technologies to CBSE

We reviewed and examined the usage history of Semantic web technologies to achieve various tasks of the CBSE process in Section 2.4. 54 articles directly pertaining to the topic of research, published in conference proceedings and journals, were analysed to assess the advancement and direct future research on the employment of ontology, semantic web services and linked data technology in CBSE.

The research articles were evaluated with a specific emphasis on types of Semantic web technology, specific tool, methods, and purpose in detail.

Based on Table A.2, it could be construed that:

- Over time, more authors realized the potential benefits that Semantic web technologies held to ease the process of CBSE (which explains an increase in the number of research articles) therefore, more Semantic web-based frameworks for CBSE were proposed. Also, while in the early years, most of these frameworks catered for software component discovery related issues, a shift towards software component integration, software component reuse, and component testing could be noted.
- Ontologies have been by far the most commonly used Semantic web technology in CBSE. Ontology has shown to facilitate software component selection, storage as well as interoperability, thus catering to almost all the aspects of CBSE. With the help of additional practical features like reasoning, ontologies are able to validate the developed component-based software and detect the possible vulnerabilities and threats. Semantic Web Activity provides ontological Web Compliance, thus catering for the effective location and selection of components on the Web which further propels the reuse of software components. Also, Semantic web services aid in solving the semantic heterogeneity problem, thereby facilitating the semantic integration between disparate software components.

However, perhaps, since Semantic web is still an emerging area, straightforward and freely accessible CBSE applications that employ Semantic web are limited. Although many researchers have highlighted in the literature studied in Table A.2 how the beneficial features of ontology can be leveraged to specify and describe software components in a reuse repository, the source code within the software component has been overlooked by the Semantic web community.

We believe that ontologies are capable of transforming existing content management systems to knowledge-bases wherein information regarding the source code of software are not only stored but also processed for making relevant software development decisions. Moreover, attributes or metrics that impact or affect a potential change of software have not been included in these repositories, in spite of

there being adequate research and literature for the same [10-25]. None of the Semantic web technologies have been explored with respect to modelling supplementary source code related information regarding a software project or component at a finer granularity in an ontology-based software repository.

2.6 Summary

The purpose of this chapter was to review the research literature in a bid to identify research gaps that built the research objectives of this thesis stated in Chapter 1 Section 1.3. This chapter commenced with a basic explanation of software components, its examples and its code components. It then provided an introduction to software change, possible causes for a software change, when does it occur and what software component elements undergo a change.

We then conducted the actual in-depth literature review by performing three separate literature studies:

- First, a rigorous study of the existing literature that focuses on determining and predicting software change was performed which specifically discussed: the software metrics reported and validated to be useful for change-proneness prediction, the prediction techniques employed, validation strategies and the statistical tests incorporated.
- Second, a review of the literature related to the history of cognitive complexity in software development was presented.
- Third, the usage history of Semantic web technologies to achieve various tasks of the CBSE process was examined using a systematic literature review, providing details regarding the purpose of employing Semantic web technology for CBSE, type of Semantic web technology employed and specific tools and methods incorporated.

The major observations drawn and the potential gaps identified from the three extensive literature reviews in this chapter provided a background for the development of the research objectives of this thesis work given in Section 1.3 of Chapter 1.

The next chapters cater to providing empirical analyses in order to achieve research objectives[2] stated (second research objective onwards).

[2] The following chapters (Chapter 3 onwards) begin by catering to the second research objective since the first objective pertaining to examining the software change prediction studies has already been covered in Section 2.2 of this chapter.

CHAPTER 3
A COGNITIVE ASPECT OF SOFTWARE CHANGE

This chapter consists of an experimental analysis in which the software developer's level of difficulty in comprehending the software: the cognitive complexity, is theoretically computed and empirically evaluated for estimating its relevance to actual software change. This chapter essentially caters for the research objective 2 of our thesis.

This chapter is structured as follows: Section 3.1 provides a summary of the research background that exhibits the status of cognitive complexity with respect to software development in literature (a detailed literature survey regarding the same has been provided in Chapter 2 Section 2.3) and also presents the research questions to be answered in this chapter. Section 3.2 describes the cognitive complexity metric considered and evaluated in the study. Section 3.3 describes the empirical design employed in this analysis comprising of the independent and dependent variables, the target projects undertaken for evaluation and empirical data collection. Section 3.4 discusses the research methodology incorporated. Section 3.5 consists of the results of the analysis conducted with the help of statistical and ML techniques to estimate the performance of the cognitive complexity as a quantifier of version to version source code change-proneness of Java files. Section 3.6 discusses the research questions stated in Section 3.1. Section 3.7 summarises this chapter.

3.1 Background and Research Questions

Chapter 2 describes the numerous research efforts that have been attempted to quantify the change-proneness of a code element for software change prediction. These usually involve static code metrics gathered automatically by various software tools, provided in Section 2.2.1 of Chapter 2. Although all these measures have been validated to substantially aid in executing a successful maintenance or development activity, the human cognition involved in changing an existing code, which may hamper development efforts due to limitations of cognitive resources, inadequate

modelling abilities or ineffective reasoning, has not been empirically scrutinized. The interpretation of source code is one of the most cognitively arduous tasks that humans perform in the software change process since before the software components or projects can be changed, its constituent source code has to be understood by the potential developer, thus inherently rendering the software change activity as a common cognitive task [72-74] .

Although the past literature includes various studies contained within Chapter 2 Section 2.2, conducted for establishing associations between complexity metrics and change-proneness, a comprehensive and pragmatic validation based on real world software engineering projects conducted on the cognitive complexity metric has not been found and statistical and ML techniques have not been employed particularly for studying the relevance of the cognitive complexity as a predictor of source code change. Also, the literature that provides an appraisal of the cognitive complexity metric with respect to other variables belonging to the Object Oriented (OO) paradigm is extremely scarce.

Additionally as observed in the literature discussed in Section 2.3 of Chapter 2, most of the studies effectively propose and, at best, theoretically validate the various cognitive complexity measures. Furthermore, researchers [173] note that there exists a severe lack of related work empirically testifying the applicability of a cognitive complexity measure in any of the software development phases.

Apropos to this, the study conducted in this chapter focuses on examining if the cognitive complexity involved in analyzing a source code Java file belonging to a particular version of a software component, can predict its change-proneness for the next release. This cognitive complexity is theoretically calculated in terms of the cognitive functional size (CFS) of the code which finds its origins in cognitive informatics. It is explained as the intellectual onus on the user dealing with the code, such as the developer, tester, and maintenance staff [81, 174]. This cognitive complexity is estimated via the cognitive weights which take the internal structural flow of source code into account to measure the extent of effort required and difficulty in understanding given software. Multiple successively released versions of two plugin projects have been selected in this chapter and the cognitive complexity corresponding to each of the version's Java files has been calculated *ad-modum* the cognitive weights of its Basic Control Structures (BCSs). Also, a binary variable

indicating the change statistic corresponding to each of the Java files is estimated to depict if a particular file belonging to a particular version of the project is used with or without change or modification in the subsequent release of the software. Analysis of the effect of the calculated cognitive complexity metric on this change statistic is performed via appropriate statistical and Machine Learning (ML) methodologies.

Predictive statistical and ML models [44] aid in providing the degree of relevance of an input variable (in our case, the cognitive complexity metric) thus exhibiting if it is cogent enough to have a high impact on the target variable (in our case, the binary variable indicating the change statistic of a Java file). Also, a comparative study of the prediction results of the cognitive complexity measure with some of the other previously introduced complexity measures and existing software change prediction metrics has been performed and their relative performances have been ranked using Friedman statistical test [175] .

The work conducted in this chapter would aid in providing factual answers to the following research questions (RQs):

RQ 1 : Can cognitive complexity as a metric predict software change and is cognitive complexity, as a quantifier of version to version change-proneness of Java files, analogous or superior to some of the other similar previously introduced complexity and change-proneness prediction measures?, and;

RQ 2: Which of the techniques attests to be the best for change-proneness prediction when the selected complexity and change prediction metrics are used as independent variables?

Having formulated the research questions to be answered in this chapter, the next section explains the cognitive metric computed.

3.2 Cognitive complexity metric computation

The cognitive complexity metric evaluated in this empirical study utilizes the premise of the cognitive functional size (CFS) proposed by Wang and Shao [81] and the cognitive complexity measure proposed by Crasso et al. [82]. In both the referenced articles, cognitive weights are assigned to the Basic Control Structures (BCSs) of the source code with the motive to determine the cognitive complexity involved in analysing the logic of the software. The authors in [81] defined BCSs as a collection

of flow control techniques which are utilized for developing the functional structure of a software and classified it along the categories of *sequential, branch, iteration, embedded component,* and *concurrency* structures.

Since in a generic source code there exists a possibility that either all the BCSs are in a linear arrangement or certain BCSs are set in other BCSs, then, as given in [81] the total cognitive weight of a source code element in terms of cognitive weights of the BCSs, W_{SC}, can be stated as follows:

$$W_{SC} = \sum_{k=1}^{x}[\prod_{j=1}^{m}\sum_{i=1}^{n} W_c(k,j,i)] \quad \ldots\ldots\ldots\ldots\ldots\ldots \text{Eqn(i)}$$

where W_{SC} is the sum of cognitive weights of x linear blocks of the code embedded in specific BCSs wherein every block might be composed of m layers of nesting BCS's, with every layer having n linear BCSs.

Additionally, the authors [81] defined the cognitive functional size (CFS) to measure the cognitive complexity of a basic source code, consisting of only one method, as a product of the sum of its inputs and outputs and its total cognitive weight as shown in Eq. (ii).

$$CFS_{SC} = N_{i/o} * W_{SC} \quad \ldots\ldots\ldots\ldots\ldots\ldots \text{Eqn(ii)}$$

The BCSs introduced in [81] did not take the influence of exception occurrences and exception-handling into account while evaluating the overall cognitive complexity. Crasso et al. [82] proposed to extend the Sequence and Branch categories of the BCSs earlier considered to include these concepts for the calculation of cognitive complexity. This was primarily done in order to be pertinent to the OO mechanism, particularly to Java programs in which the exception handling mechanism is built upon three keywords: try, catch, and finally. The justification behind this addition is that an exception throw tends to disrupt the natural flow of control of a program, thereby increasing the complexity of the code.

Therefore, the BCSs employed in this work to evaluate the cognitive complexity of a Java source code file in terms of CFS are a combination of all of those introduced by Wang and Shao [81] and some from Crasso et al. [82] and have been summarized in Table 3.1. The authors established the weights for each BCS in such a manner that greater is the assigned weight of the structure, the higher is its complexity.

Table 3.1 Basic control structures (BCS) employed in our analysis for the measurement of cognitive complexity

Category	Basic control structure(s)	Weight
Sequence	Sequence *[SEQ]*	1
	Try-catch (no finally block)	2
	Try-catch-finally	2
Branch	If-then-[else] *[ITE]*	2
	Case	3
	catch *[catch]	3
	Throws*[TH]*	2
Iteration	For-do, Repeat-until, While-do *[ITR]*	3
Embedded component	Method call (local, non-local)*[MC]*	2
	Recursion	2
Concurrency	Parallel, Interrupt	4

Also, since the research work conducted in this chapter focuses on computing the cognitive complexity involved in analyzing a Java language program code file (which could consist of more than one method) of a particular software version and whether it can predict the file's use with or without change in the next release, hence apropos to our context of application, we do not take into account the number of inputs and outputs (as considered in Eq. (ii)) and evaluate the cognitive complexity *CogC* of a Java file only in terms of its total cognitive weight in cognitive weight units (CWU).

$$CogC_{SC} = W_{SC} \quad \ldots\ldots\ldots\ldots\ldots\ldots \text{Eqn(iii)}$$

It is noteworthy that while calculating *CogC*, only one sequential structure *[SEQ]* is considered for a given code file, which can be either a class or an interface. The methods in a Java file may be local, non-local or recursive method calls. Every recursive method call is deemed to be a distinct call. Even if the method (that is recursively called) is within the same file of the method or in another file that triggers the first call, simply the complexity surfaced due to method calls is added, not the complexity of the called method. We also introduce the BCS 'throws' as we perform our evaluation on a Java file where the keyword 'throws' might as well be used for throwing an exception from within a method. Additionally, for every method declared *[MD]* in the file we introduce a cognitive weight of '1' for simplicity in the calculation. The step-by-step demonstration of how the metric is calculated on actual Java files of the selected plugin projects has been explained in later sections.

The next section elaborates on the empirical design of the study performed in this chapter.

3.3 Empirical Design

Harrison and Counsell [176] proposed that a metric should be both theoretically validated and empirically evaluated before it can be employed reliably. We found extensive evidence for the theoretical validation of the CogC metric employed in this chapter to appraise the cognitive complexity of the Java files. This measure is essentially based on the cognitive complexity metric proposed by Crasso [82], which is solely an enhancement to the CFS metric proposed by Wang and Shao [81] and only constitutes of supplementary BCSs to those previously introduced. Section 2.3 of Chapter 2 already details how the cognitive complexity metrics discussed in both the research articles [81, 82] have been theoretically validated in [107, 114] . Therefore, the theoretical validation of the CogC metric has not been included in this study due to the existence of published works providing adequate testimony for the same. Instead, the research work carried out in this chapter aims at empirical validation because to the best of our knowledge there are still no published works giving an empirical comparative validation to the CogC metric against existing software metrics, specifically for version to version source code change.

Thusly, this section details the empirical design employed in this analysis which comprises of the independent and dependent variables used in this study (Section 3.3.1) along with the target projects undertaken for evaluation (Section 3.3.2) and the empirical data collection (Sections 3.3.3 and 3.3.4).

3.3.1 Variables in the study

Independent variables are employed for the prediction of the dependent variable. In our study, the independent variable chiefly corresponds to the cognitive complexity metric, CogC, discussed in Section 3.2.

Since we wanted to establish this metric as a significant and distinguishable measure of version to version Java file change-proneness, therefore we decided to compare the prediction results obtained by the selected CogC metric to those obtained by existing measures of software complexity and software change. We particularly chose two commonly employed software complexity metrics: McCabe's Cyclomatic Complexity *(CycloC)* [20, 21] and Cumulative Halstead's effort (*CHalsE)* [20], and the Chidamber and Kemerer (C&K) Metrics [17] for the performance comparison. The

C&K metric suite comprises of six software metrics: Coupling between Objects *(CBO),* Depth of Inheritance Tree *(DIT),* Lack of Cohesion of Methods *(LCOM),* Number of Children *(NOC),* Response For Class *(RFC),* and Weighted Methods for Class *(WMC)* which have been often employed and validated as some of the significant predictors of software change, as observed in Chapter 2. Therefore, these eight software metrics (two complexity and six C&K metrics) are also selected as the independent variables along with the CogC metric.

Change-proneness [6-9, 11-14, 22-32, 45-63] is the binary dependent variable in this research work and pertains to the likelihood of manifestation of a change in a Java file belonging to a software after the software has been released. The term 'change' in the applied context only checks if there has been any addition, deletion or modification of a source line of code in each Java file from one version to the next. Any change in the space characters, comment lines and string constants present in the source code files are not considered. The change-proneness of a file is coded as either "Yes" or "No", with "Yes" indicating that the file of an existing version has been used with a change in its successive release and "No" demonstrating that the file has been used in the successive version without any modification or change. Files not found to be used with or without a change in the subsequent version are omitted from the analysis.

3.3.2 Description of the target projects

The of use of code without modification (over versions and in the same product family) was found in the JFreeChart and the Heritrix plugin projects, as many of their Java files have been consistently and widely used in their future versions with/without modification or change. As read in Section 2.1 of Chapter 2, a plugin is a ready to use software component that can be added to existing software to provide supplementary features [1, 4, 5, 84]. JFreeChart [177] is a free Java chart library and makes it easy for developers to display professional quality charts in their applications. On the other hand, Heritrix [178] is the Internet Archive's open-source, extensible, scalable, archival-quality Web crawler. Both the selected projects are available as Eclipse plugins under a free software license.

It was found that the Java files were changed from version to the next in the above selected plugin projects due to one or more of the following reasons [177, 178]:

- To accommodate new features/requirements:

Tooltips, crosshairs and zooming functions were added to the JfreeChart version 0.7.0 to create the upgraded version of 0.7.1. On the other hand, Command-line options for setting web user interface username/password were added to Heritrix 0.4.0 to create the upgraded version of 0.4.0

- To fix bugs:

The code in JfreeChart version 0.7.1 had a bug with the legend that resulted in a loop at small chart sizes. This was rectified to create the upgraded version JfreeChart 0.7.2. Similary, the "NoClassDefFoundError" the was encountered when starting a job in Heritrix 0.6.0 was fixed to create Heritrix 0.8.0.

- To improve coding standards:

The responsibility for category distribution was moved to the CategoryAxis class in JfreeChart 0.7.4, which tidied up the code in the CategoryPlot classes in JfreeChart 0.7.3.

Five consecutively released versions of JFreeChart and three consecutive versions of the Heritrix software were investigated, the specifics of which are presented in Table 3.2.

Table 3.2 Target projects

Software projects	Total size in LOC	Total number of Java files
JFreeChart 0.6.0	5,700	86
JFreeChart 0.7.0	7,870	105
JFreeChart 0.7.1	8,894	128
JFreeChart 0.7.2	9,222	130
JFreeChart 0.7.3	9,318	131
Heritrix 0.2.0	8,331	125
Heritrix 0.4.0	11,688	168
Heritrix 0.6.0	12,751	200

3.3.3 Calculation of the Independent variables

This sub-section demonstrates how the numerical values of each of the nine independent variables were gathered to complete the datasets and to check their applicability against version to version change-proneness of Java files.

3.3.3.1 Calculation of the CogC metric:

The CogC metric was calculated manually according to the specifications stated in Section 3.2 for each of the Java files present in the selected JFreeChart and Heritrix versions. One Java file from the JFreeChart version 0.6.0 (See Appendix B) has been used as an example to illustrate how CogC has been estimated in cognitive weight units (CWU). The calculation is explained as follows:

`Class AbstractTitle` (given in Appendix B) has a total of 18 linear structures as per the specification provided in Section 3.2. These linear structures are basically the seventeen method declarations (MD_1-MD_{17}), each having a cognitive weight of 1 and a sequence (SEQ) again holding a cognitive weight of 1. Therefore, according to Eqn(iii) of Section 3.2,

$CogC_{AbstractTitle}$ = SEQ

+ MD_1*(((ITE$_{1,1}$+MC$_{1,1}$)*TH$_{1,1,1}$)+ (ITE$_{1,2}$*TH$_{1,2,1}$)+ (ITE$_{1,3}$*TH$_{1,3,1}$)+ MC$_{1,1}$)

+ MD_{2*}(MC$_{2,1}$ + MC$_{2,1}$)

+ MD_{3*}MC$_{3,1}$

+ MD_4* (TC$_{4,1}$* ((MC$_{4,1}$) +(TH$_{4,1}$))+MC$_{4,1}$+MC$_{4,2}$+MC$_{4,3}$+MC$_{4,4}$+MC$_{4,5}$)

+ MD_5+ MD_6+ MD_7

+ MD_8* (ITE$_{8,1}$*(MC$_{8,1,1}$ + MC$_{8,1,2}$))

+ MD_9

+ MD_{10} * (ITE$_{10,1}$*(MC$_{10,1,1}$ + MC$_{10,1,2}$))

+ MD_{11}

+ MD_{12} * (ITE$_{12,1}$*(MC$_{12,1,1}$ + MC$_{12,1,2}$))

+ MD_{13}

+ MD_{14}*((ITE$_{14,1}$ +MC$_{14,1}$)*(MC$_{14,1,1}$+MC$_{14,1,2}$))

+ MD_{15}* (MC$_{15,1}$)

+ MD_{16}* (MC$_{16,1}$)

+ MD_{17} *(ITE$_{17,1}$ *(MC$_{17,1,1}$ + (ITR$_{17,1,1}$+MC$_{17,1,2}$)*(MC$_{17,1,1,1}$ + MC$_{17,1,1,2}$)))

=1+18+4+2+18+3+8+1+8+1+8+1+16+2+2+44= 137 CWU

Although the CFS [81] and the Cognitive Complexity metric [82] was invented to be a language-independent measure, it cannot be overlooked that diverse languages offer a diverse set of features. For example, until recently C++ lacked a try-finally structure because it earlier relied on concepts like RAII "Resource Acquisition Is Initialization" [179] for memory de-allocation.

However, such drawbacks seldom thwart software developers from requiring those structures or from attempting to build something comparable with the software tools available at that point. In situations like these, an exact exercise of the Cognitive Complexity's rules would give rise to unreasonably high scores. For instance, the class `AbstractTitle` explained before is essentially the subclass of the class `Object` and uses the interface `Cloneable`. However, since we tested the change-proneness of each of the files (interfaces and classes) contained in the software versions independently using the clone detection tool, we have considered all the files (superclasses, subclasses and interfaces) contained in the JFreeChart and Heritrix projects as individual entities for change-proneness analysis, and did not take into account the additional complexity caused due to inheritance or implementation of an interface while calculating the CogC of a file.

3.3.3.2 Calculation of the remaining eight independent variables

In order to gather the numerical values of the other eight independent variables (two complexity metrics and six C&K metrics), with respect to each of the Java files present in all the selected JFreeChart and Heritrix versions, we employed two open source static code analysis tools: JHawk 6.1.3 [180] and Stan4J [181]. The tools calculated the metrics[3] according to their standard definitions given in [17, 20, 21].

3.3.4 Calculation of the dependent variable

The dependent variable in this analysis denotes if a Java file of a software version has been used with or without any change or modification in the successively released version of the software or not. Since without modification applies to exact copies of source code, therefore, in our study, the "stable" or "unchanged" Java files have been identified using the AntiCutandPaste software which is fundamentally a plagiarism

[3] We conduct our analysis at a file-level, although most of the metrics selected as a part of our research are class-level metrics. In situations where a file is composed of more than one class, the sum of the numerical values of the metrics obtained by each of its constituent class is considered.

detection tool. This tool is also categorized under code clone detection tools [182] and accurately returns the files having exactly similar content when two source code packages are run through it, i.e. the code fragments (Java files, in our case) that have been used without change and sans any modification from version $_i$ to version $_{i+1}$.

Post the application of the AntiCut&Paste tool on two consecutive software releases, a dichotomous value of Yes/No is computed as the change statistic and entered corresponding to each Java file of the selected JFreeChart and Heritrix versions, with "Yes" implying that the Java file of an existing version has been changed in its successive release and "No" signifying that the file has been used in the successive version without any modification or change. The results from the AntiCut&Paste tool were further verified manually by comparing the full file names (i.e. package name + class name) and their numeric metric values obtained from the tools JHawk and Stan4j of all the Java files of the previous release with the Java files of the later release (exactly similar files between two versions have the same numerical values of metrics like number of calls, cyclomatic complexity etc.). This is primarily done to ensure that the software does not bias the credibility of our results in any way.

The numerical values of each of the nine metrics (CogC + two complexity metrics+ six C&K metrics) and the binary value of the change statistic are then combined to generate data points corresponding to each Java file in each of the eight versions. A summary on change-proneness of the Java files contained in the eight versions employed in our study is given in Table 3.3.

Table 3.3 Change statistics of the Java files

Versions	Total number of Java files	Number of Java files used in the next version	Number of Java files used without change in the next version	Number of Java files used with change in the next release
JFreeChart 0.6.0	86	86	67	19
JFreeChart 0.7.0	105	102	42	60
JFreeChart 0.7.1	128	122	94	28
JFreeChart 0.7.2	130	130	112	18
JFreeChart 0.7.3*	131	130	92	38
Heritrix 0.2.0	125	113	48	65
Heritrix 0.4.0	168	155	70	85
Heritrix 0.6.0*	200	195	128	67

* *The successive version analysed for change statistics of JFreeChart 0.7.3 is JFreeChart 0.7.4 and for the change statistics of Heritrix 0.6.0, we analysed Heritrix 0.8.0.*

We gathered the numerical values of each of the nine independent variables and change statistic corresponding to only those Java files that were employed in the next

release (see column 3 of Table 3.3) for each of the eight individual versions, generating a total of 1,033 data points corresponding to the eight datasets.

3.4 Research Methodology

In this section, we explain the Statistical and ML techniques employed to generate predictive models and the parameters used for the performance evaluation of the models. The results of correlation analysis conducted between the independent variables are also reported in this section.

3.4.1 Prediction techniques employed

Table 3.4 gives a short description of the prediction algorithms incorporated for the purpose of identifying the association between the change statistic of the Java files and each of the independent variables gathered over the eight datasets (as shown in Table 3.3).

It is imperative to note that the prediction techniques were selected in a manner such that minimum one technique gets selected from each category. Moreover, the past literature stated in Table A.1 (provided in Appendix A) of Section 2.2 of Chapter 2 is indicative that the techniques selected have been examined for change-proneness prediction in situations where the dependent binary variable of change was learned and an appropriate change prediction metric set comprising of multiple independent variables was employed. However, none of these techniques have ever been inspected for those cases where the

Table 3.4 Description of the prediction techniques employed in this study

Category	Prediction Technique	Description
Bayesian classifier	NaiveBayes	The *NaiveBayes* [12, 45-48] classifier is concerned with the extent to which the probability of a hypothesis being accurate is dependent on previous unfamiliar information.
Functions	MultiLayer Perceptron (MLP)	The *MLP* [11, 32-34, 55] artificial neural network self-learns via identifying a set of connection weights which minimize the sum of squared errors on the training sample, therefore discovering various input characteristics that play the most important role in the target function's learning process. We employ the standard error back-propagation algorithm to train the network with the default learning rate value of 0.3 and a momentum value of 0.2, the minimum square error being the training stopping criterion.
Meta-classifiers	Bagging AdaBoost	The *Bagging* [11, 30, 47, 48, 55] technique uses a plurality voting scheme to conjoin numerous results of a learning algorithm for the purpose of obtaining a combined single prediction. We employed 10 iterations, a bag size percentage of 100, and REPTree as the classifier in our study.

		Analogous to the bagging algorithm, the boosting methodology also syndicates the varied results of a learning technique in order to obtain an aggregated prediction. However, the weights of training cases are altered in every iteration to compel the learning algorithms to give lesser importance to the instances that were estimated correctly previously and more emphasis on instances that were estimated incorrectly previously. This study includes the adaptive boosting *AdaBoost* technique for experimental analysis.
Decision Trees	Random Forest	The *Random Forest* [11, 34, 47, 48, 55, 57] classifier is a collection of un-pruned regression or classification trees constructed via a random selection of samples from the training data. Training dataset samples are selected with replacement. However, the trees are developed in a manner to limit the correlation between each of the classifiers.
Statistical classifier	Logistic Regression (LR)	*LR* [6, 11, 28, 33] is a conventional modeling approach that aids in describing the association of a number of Xs (independent variables) with a binary categorical dependent variable (Y), exhibiting the empirical occurrence or non-occurrence of an event.

independent variables have individually been used to predict the target variable of change-proneness.

Hence, these models were selected as candidates for applying a univariate analysis to examine if the evaluated CogC metric individually performs better than or similar to the existing complexity and software change metrics when predicting the version to version change-proneness of a Java file. Moreover, results drawn from the analysis would aid the software developers in knowing which algorithm to employ when assessing the prediction capability of a single independent variable against a binary change statistic of a Java file.

3.4.2 Performance evaluation measures

The ensuing subsections state the performance measures used for the valuation of the quality of the predicted models generated in this chapter.

3.4.2.1 Accuracy

The accuracy of the model is given as the percentage of the number of Java files that are accurately predicted to the total number of Java files in the dataset.

3.4.2.2 Area Under the ROC Curve (AUC)

The ROC curve [47] is an efficient technique for the evaluation of the performance or quality of the predicted models and is obtained by plotting sensitivity versus 1-specificity. The Area Under the ROC Curve (AUC) is a composite estimate of the specificity and sensitivity, depicting the point where both sensitivity and specificity

are maximized. To establish the final observations related to the performance of the models, we followed the conventional interpretations of the AUC values which state that: AUC values <0.6 exhibit unacceptable classification; AUC values>=0.6 and <0.7 signify poor classification; AUC values>=0.7 and <0.8 signify acceptable classification; AUC values >=0.8 and <0.9 signify excellent classification; and AUC >= 0.9, signifies outstanding classification between the changed and stable files by the prediction algorithm.

Additionally, for the purpose of attaining a supplementary pragmatic estimation, a ten-fold cross-validation [12] of all the models generated in this chapter is performed.

3.4.3 Correlation Analysis of the independent variables

A correlation analysis has been performed in this sub-section to investigate the extent of correlation between the independent/predictor variables since this correlation often disrupts the assumption of independence among the predictor variables and leads to misleading results. In particular, we wanted to verify the absence of correlation between the estimated CogC metric with the other eight selected independent variables so that it can be efficiently incorporated in the version to version change prediction metric set. The correlation is calculated with the help of Spearman's correlation, a non-parametric correlation test.

Tables 3.5 (a-e) and 3.6 (a-c**)** depict the Spearman's Rho coefficient of correlation between each of the nine metrics for every selected version of JFreeChart and Heritrix projects wherein, a correlation coefficient value higher than 0.8 depicts a high correlation, a correlation coefficient value between 0.5 and 0.8 depicts a moderate correlation, and a correlation coefficient value less than 0.5 depicts a low correlation [183]. The diagonal elements in the tables hold a correlation coefficient value of 1, following the definition of correlation. For the JFreeChart releases (Tables 3.5(a-e)) and the selected Heritrix releases (Tables 3.6(a-c)) there are certain metrics that display high correlations with each other. CogC metric, in particular, is not observed to be highly correlated with any of the other eight metrics for all the selected version datasets except for JFreeChart 0.7.0. However, the metrics RFC and WMC are observed to be highly correlated to one another for all the eight datasets. Also, the CogC measure holds a negative correlation only with the metric NOC and is observed

to have a positive correlation with the remaining seven independent variables for all the selected version datasets for JFreeChart and Heritrix.

As mentioned earlier, a high degree of correlation between metrics represents redundant information and may cause multicollinearity. Therefore, we further calculate the condition number for the metrics using the Principal Component Analysis (PCA). PCA technique generates the minimum and the maximum eigenvalue (e_min and e_max) when employed on the predictor or independent variables. The conditional number (λ) is then calculated as √(e_max/ e_min). A λ value lesser than 30 indicates the absence of multicollinearity between the independent variables [183, 184].

Table 3.7 shows the conditional number (λ) estimated for all the metrics for each of the selected releases wherein it can be observed that for every selected software version, λ is well below 30. This indicates that the multicollinearity between the independent variables for all the selected releases of JFreeChart and Heritrix is tolerable and therefore all the nine selected independent variables (including the CogC metric) can be employed altogether for predicting the version to version change-proneness of Java files.

Table 3.5(a-e) Correlation analysis for the independent variables corresponding to the five selected versions of JFreeChart (The correlation coefficient values higher than 0.8 are indicated in bold)

Table 3.5(a) Correlation analysis of JFreeChart 0.6.0

	CBO	CHalsE	CogC	CycloC	DIT	LCOM	NOC	RFC	WMC
CBO	1.000								
ChalsE	-.078	1.000							
CogC	.180	.707	1.000						
CycloC	.395	.544	.773	1.000					
DIT	.198	.032	.307	.319	1.000				
LCOM	.341	.755	.766	**.846**	.073	1.000			
NOC	.634	-.053	-.071	.273	.041	.250	1.000		
RFC	.576	.412	.650	**.893**	.444	.721	.404	1.000	
WMC	.614	.366	.598	**.879**	.416	.706	.444	**.982**	1.000

Table 3.5(b) Correlation analysis of JFreeChart 0.7.0

	CBO	CHalsE	CogC	CycloC	DIT	LCOM	NOC	RFC	WMC
CBO	1.000								
CHalsE	-.109	1.000							
CogC	.033	**.802**	1.000						
CycloC	.368	.146	.603	1.000					
DIT	.170	-.043	.047	.169	1.000				
LCOM	.337	.119	.584	**.902**	-.007	1.000			
NOC	.689	-.053	-.009	.317	.069	.271	1.000		
RFC	.553	.063	.362	.756	.493	.623	.488	1.000	
WMC	.622	.081	.384	**.823**	.349	.702	.550	**.900**	1.000

Table 3.5(c) Correlation analysis of JFreeChart 0.7.1

	CBO	CHalsE	CogC	CycloC	DIT	LCOM	NOC	RFC	WMC
CBO	1.000								
CHalsE	-.079	1.000							
CogC	.163	.645	1.000						
CycloC	.437	.171	.735	1.000					
DIT	.188	-.015	.236	.221	1.000				
LCOM	.418	.149	.670	**.912**	.049	1.000			
NOC	.616	-.027	-.102	.308	.071	.276	1.000		
RFC	.598	.094	.545	.772	.520	.658	.459	1.000	
WMC	.667	.107	.584	.834	.374	.729	.521	**.901**	1.000

Table 3.5(d) Correlation analysis of JFreeChart 0.7.2

	CBO	CHalsE	CogC	CycloC	DIT	LCOM	NOC	RFC	WMC
CBO	1.000								
CHalsE	-.084	1.000							
CogC	.123	.712	1.000						
CycloC	.458	.179	.609	1.000					
DIT	.197	-.013	.196	.211	1.000				
LCOM	.452	.144	.572	**.904**	.049	1.000			
NOC	.624	-.031	-.035	.341	.074	.314	1.000		
RFC	.609	.110	.458	.773	.511	.674	.477	1.000	
WMC	.682	.113	.469	**.828**	.346	.742	.554	**.896**	1.000

Table 3.5(e) Correlation analysis of JFreeChart 0.7.3

	CBO	CHalsE	CogC	CycloC	DIT	LCOM	NOC	RFC	WMC
CBO	1.000								
CHalsE	-.076	1.000							
CogC	.183	.711	1.000						
CycloC	.458	.187	.569	1.000					
DIT	.193	-.013	.191	.212	1.000				
LCOM	.461	.147	.516	**.906**	.050	1.000			
NOC	.616	-.023	-.038	.333	.063	.314	1.000		
RFC	.605	.119	.450	.762	.518	.665	.467	1.000	
WMC	.681	.127	.465	**.826**	.347	.744	.546	**.886**	1.000

Table 3.6(a-c) Correlation analysis for the independent variables corresponding to the three selected versions of Heritrix (The correlation coefficient values higher than 0.8 are indicated in bold)

Table 3.6(a) Correlation analysis of Heritrix 0.2.0

	CBO	CHalsE	CogC	CycloC	DIT	LCOM	NOC	RFC	WMC
CBO	1.000								
CHalsE	.125	1.000							
CogC	.086	.265	1.000						
CycloC	.259	.727	.269	1.000					
DIT	.094	-.027	.176	.028	1.000				
LCOM	.352	.393	.164	.694	-.028	1.000			
NOC	.274	-.072	-.044	-.027	.059	-.023	1.000		

RFC	.420	.601	.140	**.868**	.057	.593	.042	1.000	
WMC	.374	.440	.175	**.843**	.120	.581	.040	**.863**	1.000

Table 3.6(b) Correlation analysis of Heritrix 0.4.0

	CBO	CHalsE	CogC	CycloC	DIT	LCOM	NOC	RFC	WMC
CBO	1.000								
CHalsE	.246	1.000							
CogC	.377	.771	1.000						
CycloC	.462	**.846**	.621	1.000					
DIT	.067	.145	.134	.063	1.000				
LCOM	.511	.640	.555	.779	.042	1.000			
NOC	.295	-.060	-.030	.011	.098	-.017	1.000		
RFC	.681	.525	.470	**.813**	.037	.676	.082	1.000	
WMC	.595	.537	.420	.797	-.022	.610	.051	**.914**	1.000

Table 3.6(c) Correlation analysis of Heritrix 0.6.0

	CBO	CHalsE	CogC	CycloC	DIT	LCOM	NOC	RFC	WMC
CBO	1.000								
CHalsE	.198	1.000							
CogC	.038	.418	1.000						
CycloC	.445	.766	.380	1.000					
DIT	.112	.071	.009	.040	1.000				
LCOM	.489	.472	.191	.785	.024	1.000			
NOC	.335	-.015	-.026	.034	.086	.010	1.000		
RFC	.716	.462	.103	.776	.086	.657	.125	1.000	
WMC	.633	.473	.091	.763	.020	.580	.097	**.904**	1.000

Table 3.7 Multi-collinearity analyses

Release	Condition number(λ)
JFreeChart 0.6.0	√(5.016/0.014) = 18.93
JFreeChart 0.7.0	√(4.373/0.035) =11.18
JFreeChart 0.7.1	√(4.685/0.053) = 9.42
JFreeChart 0.7.2	√(4.602/0.064) = 8.48
JFreeChart 0.7.3	√(4.683/0.065) = 8.49
Heritrix 0.2.0	√(3.939/0.058) = 8.24
Heritrix 0.4.0	√(4.732/0.041) = 10.74
Heritrix 0.6.0	√(4.196/0.062) = 8.23

3.5 Analysis of results

This section is comprised of the results of a detailed comparative empirical study performed in order to gauge the capability of the CogC metric as a quantifier of version to version change-proneness of Java files. This section also compares the capability of the selected predictive techniques employed.

3.5.1 Individual metric evaluation

We employ one statistical technique and five ML algorithms (described in Section 3.4.1) to assess the individual performance of CogC measure with respect to version to version change-proneness of a Java file. Since we needed to conduct a comparative analysis, two other complexity measures and six commonly employed software change prediction measures are also assessed individually for their performance.

The results obtained from the prediction models constructed via the six techniques on the five JFreeChart and three Heritrix datasets have been presented in Tables 3.8 and 3.9. All the model results have been generated using the default settings of Weka 3.8.0 [185] and have been assessed via the ten-fold cross-validation technique. The columns in Tables 3.8 and 3.9 indicate the version-wise Accuracy (*Acc.*) and Area under Curve (*AUC*) attained by each of the nine independent variables for each of the prediction techniques for a specific version. (Version-wise highest accuracy and AUC values for each of the models have been indicated in bold.)

Table 3.8 Results of the predictive models generated corresponding to the nine metrics over the five versions of JFreeChart

	JFreeChart 0.6.0		JFreeChart 0.7.0		JFreeChart 0.7.1		JFreeChart 0.7.2		JFreeChart 0.7.3	
	Naïve Bayes		*Naïve Bayes*		*Naïve Bayes*		*Naïve Bayes*		*Naïve Bayes*	
Metric	Acc.	AUC	Acc.	AUC	Acc.	AUC	Acc.	AUC	Acc.	AUC
CogC	77.91	0.730	50.98	0.616	74.59	0.694	**86.92**	0.687	79.23	0.764
CycloC	81.39	0.775	60.78	0.669	77.05	0.753	83.85	0.500	76.92	0.741
CHalsE	82.56	0.653	52.94	0.532	78.68	0.660	85.38	0.577	69.23	0.599
LCOM	80.23	0.680	60.78	0.641	**79.51**	0.724	83.85	0.583	78.46	0.687
CBO	77.91	0.663	61.74	0.575	78.69	0.533	81.61	0.607	79.23	0.788
RFC	83.72	0.785	70.59	0.661	71.31	**0.761**	82.31	0.664	80.77	0.830
NOC	79.07	0.570	55.88	0.498	74.59	0.414	86.15	0.444	70.77	0.628
WMC	**84.88**	**0.797**	69.6	0.674	76.23	0.741	85.38	0.674	79.23	**0.817**
DIT	76.74	0.581	**71.57**	**0.704**	76.23	0.517	84.62	**0.726**	**82.31**	0.717
	MLP		*MLP*		*MLP*		*MLP*		*MLP*	
	Acc.	AUC	Acc.	AUC	Acc.	AUC	Acc.	AUC	Acc.	AUC
CogC	77.91	0.742	**72.54**	**0.756**	76.23	0.757	**86.15**	0.699	79.23	0.781
CycloC	**82.55**	0.742	62.75	0.708	76.23	0.742	**86.15**	**0.709**	76.15	0.767
CHalsE	81.39	0.686	50.00	0.493	**79.51**	0.702	84.62	**0.709**	70.77	0.629
LCOM	80.23	0.646	57.84	0.640	**79.51**	**0.762**	84.62	0.553	76.92	0.688
CBO	80.23	0.672	63.73	0.675	77.87	0.561	86.15	0.637	80.77	0.771
RFC	81.39	**0.774**	70.89	0.717	74.59	0.740	85.38	0.619	**83.08**	**0.829**
NOC	81.39	0.574	53.92	0.570	74.59	0.470	**86.15**	0.475	74.62	0.602
WMC	80.23	0.762	70.59	0.725	77.05	0.743	**86.15**	0.681	80.00	0.805
DIT	77.91	0.547	71.57	0.744	76.23	0.498	**86.15**	0.707	82.31	0.732
	AdaBoost		*AdaBoost*		*AdaBoost*		*AdaBoost*		*AdaBoost*	
	Acc.	AUC	Acc.	AUC	Acc.	AUC	Acc.	AUC	Acc.	AUC
CogC	79.07	0.752	71.56	0.700	72.95	0.686	**87.69**	0.752	73.07	0.728
CycloC	79.07	0.730	64.71	0.651	77.05	**0.725**	86.15	0.596	76.15	0.729

Table 3.8 Results of the predictive models generated corresponding to the nine metrics over the five versions of JFreeChart

CHalsE	75.58	0.731	**75.49**	**0.711**	77.86	0.685	85.38	**0.794**	67.69	0.670
LCOM	80.23	0.625	53.92	0.540	**78.69**	0.714	83.08	0.719	73.85	0.651
CBO	**81.39**	0.618	60.78	0.672	77.05	0.545	85.38	0.609	83.08	0.730
RFC	80.23	**0.779**	68.63	0.709	76.23	0.718	86.92	**0.794**	**83.85**	**0.820**
NOC	**81.39**	0.592	52.94	0.519	74.59	0.391	86.15	0.473	73.85	0.590
WMC	79.07	0.775	66.67	0.693	74.59	0.713	84.62	0.788	80.00	0.762
DIT	77.91	0.552	71.57	0.686	76.23	0.491	86.15	0.661	82.31	0.638
	Bagging		*Bagging*		*Bagging*		*Bagging*		*Bagging*	
	Acc.	AUC	Acc.	AUC	Acc.	AUC	Acc.	AUC	Acc.	AUC
CogC	76.74	0.703	68.62	0.711	77.05	0.68	**86.15**	**0.842**	79.23	0.762
CycloC	80.23	0.745	70.59	0.713	77.87	0.745	**86.15**	0.678	74.62	0.732
CHalsE	**82.56**	0.679	**77.45**	**0.766**	75.4	0.738	84.62	0.785	70.00	0.679
LCOM	80.23	0.687	58.82	0.627	**79.51**	0.739	**86.15**	0.720	75.38	0.678
CBO	81.39	0.618	61.76	0.720	76.23	0.675	**86.15**	0.646	**82.31**	0.783
RFC	80.23	0.757	62.75	0.736	77.87	**0.752**	84.62	0.812	78.46	**0.806**
NOC	77.91	0.596	53.92	0.572	74.59	0.516	**86.15**	0.436	72.31	0.609
WMC	81.39	**0.767**	65.69	0.704	75.41	0.735	83.85	0.789	75.38	0.782
DIT	77.91	0.517	71.57	0.746	76.23	0.493	**86.15**	0.687	**82.31**	0.649
	Random Forest		*Random Forest*		*Random Forest*		*Random Forest*		*Random Forest*	
	Acc.	AUC	Acc.	AUC	Acc.	AUC	Acc.	AUC	Acc.	AUC
CogC	70.93	0.651	72.58	0.717	**81.15**	0.671	83.07	**0.819**	77.69	0.764
CycloC	72.39	0.595	66.67	0.697	**81.15**	0.699	80.77	0.588	73.08	0.666
CHalsE	70.93	0.717	**78.43**	**0.813**	70.49	0.679	78.46	0.798	66.15	0.687
LCOM	79.07	0.698	58.82	0.558	77.05	0.681	**88.46**	0.68	69.23	0.651
CBO	**83.72**	0.572	61.76	0.672	72.13	0.569	87.69	0.509	81.54	0.761
RFC	73.26	0.753	65.69	0.725	76.23	0.647	83.85	0.813	78.46	**0.785**
NOC	80.23	0.608	57.84	0.521	74.59	0.452	86.15	0.496	73.08	0.551
WMC	79.07	**0.785**	62.75	0.666	77.87	**0.717**	81.54	0.751	70.77	0.714
DIT	77.91	0.456	71.57	0.722	76.23	0.46	87.69	0.735	**82.31**	0.702
	LR		*LR*		*LR*		*LR*		*LR*	
	Acc.	AUC	Acc.	AUC	Acc.	AUC	Acc.	AUC	Acc.	AUC
CogC	79.07	0.744	50.98	0.686	75.41	0.779	**87.69**	**0.824**	78.41	0.778
CycloC	82.56	0.764	67.65	0.734	78.69	0.782	86.15	0.667	76.92	0.793
CHalsE	82.55	0.720	53.92	0.357	79.50	0.755	86.15	0.794	70.00	0.663
LCOM	80.23	0.683	60.78	0.635	**79.51**	**0.790**	85.38	0.577	76.92	0.687
CBO	82.32	0.704	57.84	0.705	76.23	0.530	85.38	0.656	80.77	0.780
RFC	**83.72**	**0.795**	68.63	**0.753**	72.13	0.768	88.46	0.704	80.77	**0.845**
NOC	79.07	0.574	51.96	0.551	74.59	0.449	86.15	0.444	70.00	0.602
WMC	82.56	**0.795**	68.63	0.743	74.59	0.775	85.38	0.690	80.00	0.830
DIT	79.07	0.592	**71.57**	0.723	76.23	0.444	84.62	0.724	**82.31**	0.731

Table 3.9 Results of the predictive models generated corresponding to the nine metrics over the three versions of Heritrix

	Heritrix 0.2.0		Heritrix 0.4.0		Heritrix 0.6.0	
	Naïve Bayes		*Naïve Bayes*		*Naïve Bayes*	
Metrics	Acc.	AUC	Acc.	AUC	Acc.	AUC
CogC	61.00	**0.732**	62.32	0.638	68.75	0.659
CycloC	54.87	0.635	58.06	0.632	69.00	0.600

Table 3.9 Results of the predictive models generated corresponding to the nine metrics over the three versions of Heritrix

CHalsE	54.87	0.671	52.26	0.534	67.50	0.588
LCOM	46.12	0.508	48.39	0.508	66.50	0.554
CBO	**61.06**	0.640	**63.87**	**0.734**	**72.00**	**0.683**
RFC	60.17	0.657	56.77	0.576	68.00	0.650
NOC	45.13	0.477	48.39	0.495	66.50	0.664
WMC	60.18	0.648	50.97	0.524	67.00	0.607
DIT	**61.06**	0.588	57.42	0.565	70.50	0.617

	MLP		*MLP*		*MLP*	
	Acc.	AUC	Acc.	AUC	Acc.	AUC
CogC	**73.00**	**0.765**	63.76	0.632	68.18	0.675
CycloC	67.26	0.730	63.28	0.625	67.00	0.645
CHalsE	71.68	0.760	55.48	0.611	68.50	0.628
LCOM	55.75	0.516	54.84	0.524	64.50	0.593
CBO	65.49	0.737	**69.68**	**0.751**	**74.00**	**0.703**
RFC	64.62	0.665	58.06	0.624	66.00	0.681
NOC	55.75	0.473	54.19	0.524	67.00	0.512
WMC	62.83	0.638	57.42	0.554	63.00	0.634
DIT	61.95	0.628	56.13	0.614	66.00	0.627

	AdaBoost		*AdaBoost*		*AdaBoost*	
	Acc.	AUC	Acc.	AUC	Acc.	AUC
CogC	67.00	0.681	61.59	0.619	64.77	0.619
CycloC	**73.45**	**0.753**	63.23	0.614	66.50	0.613
CHalsE	72.57	0.735	65.16	0.615	70.00	0.610
LCOM	57.52	0.460	50.97	0.519	65.00	0.579
CBO	62.83	0.688	**72.26**	**0.731**	**75.00**	**0.674**
RFC	70.79	0.693	58.06	0.598	67.00	0.662
NOC	57.52	0.464	54.84	0.508	67.00	0.520
WMC	67.26	0.616	56.13	0.531	64.00	0.642
DIT	59.29	0.585	60.00	0.680	70.50	0.578

	Bagging		*Bagging*		*Bagging*	
	Acc.	AUC	Acc.	AUC	Acc.	AUC
CogC	71.00	0.679	62.32	0.630	62.50	0.617
CycloC	69.91	0.709	65.16	0.631	69.50	0.613
CHalsE	**71.68**	**0.755**	58.71	0.613	67.50	0.633
LCOM	55.75	0.526	57.42	0.624	66.50	0.638
CBO	65.49	0.722	**70.32**	**0.757**	**75.50**	**0.691**
RFC	70.79	0.688	64.52	0.709	65.00	0.671
NOC	57.52	0.519	52.90	0.471	67.00	0.508
WMC	66.37	0.643	58.71	0.606	65.00	0.642
DIT	57.52	0.612	60.65	0.675	69.00	0.590

	Random Forest		*Random Forest*		*Random Forest*	
	Acc.	AUC	Acc.	AUC	Acc.	AUC
CogC	51.00	0.580	54.35	0.624	60.79	0.600
CycloC	66.37	0.694	63.23	0.612	68.00	0.525
CHalsE	**67.26**	**0.692**	56.13	0.535	56.50	0.606
LCOM	59.29	0.615	58.71	0.589	66.50	0.565
CBO	66.37	0.707	**72.26**	**0.731**	**73.00**	**0.669**
RFC	67.26	0.626	65.16	0.715	61.50	0.575
NOC	56.64	0.462	52.90	0.439	66.50	0.488
WMC	62.83	0.646	56.13	0.565	64.00	0.589
DIT	56.64	0.565	60.00	0.644	69.00	0.588

	LR		*LR*		*LR*	
	Acc.	AUC	Acc.	AUC	Acc.	AUC

Table 3.9 Results of the predictive models generated corresponding to the nine metrics over the three versions of Heritrix

CogC	73.00	0.778	63.04	0.640	66.48	0.689
CycloC	70.79	0.758	61.94	0.654	69.50	0.684
CHalsE	**76.11**	**0.785**	64.52	0.657	67.50	0.659
LCOM	57.52	0.519	53.84	0.538	67.00	0.608
CBO	69.03	0.753	**70.97**	**0.773**	**74.00**	**0.723**
RFC	70.79	0.683	60.65	0.644	67.50	0.714
NOC	57.52	0.431	54.84	0.497	67.00	0.517
WMC	64.61	0.664	58.06	0.596	67.00	0.670
DIT	54.87	0.611	54.19	0.597	70.50	0.626

To summarise,

Out of a total of 270 models generated in Table 3.8 (30 models generated for each of the nine metrics) over the five JFreeChart versions, as far as the AUC values are concerned,

- The highest numbers of excellent divisions (i.e. AUC>= 0.8 and <0.9) are shown by the RFC metric for 7 models. This is followed by the CogC and WMC metrics which show excellent divisions for 3 models.
- The WMC metric exhibits 21 acceptable divisions (i.e. AUC>= 0.7 and <0.8), the highest among the nine selected metrics, between the changed and the stable files. This is followed by the CycloC and the RFC metrics with 19 acceptable divisions. The CogC metric comes next with 18 acceptable divisions.
- The NOC metric exhibits highest number of unacceptable divisions (i.e. AUC<0.6) for 25 out of 30 models and is thus adjudged as the worst performing metric among the nine independent variables to predict change-prone files over the five selected versions of JFreeChart software.

Out of a total of 162 models generated in Table 3.9 (18 models generated for each of the nine metrics) over the three Heritrix versions, as far as the AUC values are concerned,

- None of the metrics are observed to exhibit excellent divisions over any of the three version datasets for any of the models.
- The CBO metric exhibits 12 acceptable divisions (i.e. AUC>= 0.7 and <0.8), the highest among the nine selected metrics, between the changed and the stable files. This is followed by the CycloC and the CHalsE metrics with 4

acceptable divisions. The CogC and RFC metrics come next with 3 acceptable divisions.

- Likewise, as observed for the selected JFreeChart versions, the worst performing metric to predict change-prone files over the three selected versions of Heritrix software is the NOC metric exhibiting unacceptable divisions (i.e. AUC<0.6) for 17 out of 18 models, the highest among the nine independent variables.

However, none of the metrics exhibit outstanding divisions between the changed and stable files for both the JFreeChart and Heritrix datasets. Moreover, contrary to the results observed in Table 3.8 where there is no consistency among any of the metrics in terms of predictive performance over the five JFreeChart versions, the results in Table 3.9 depict the CBO metric displaying consistently highest results for both the performance measures over most of the predictive models especially for Heritrix 0.4.0 and Heritrix 0.6.0.

3.5.2 Metric evaluation using statistical testing

As observed in Tables 3.8 and 3.9, multiple predictive models are generated corresponding to each of the nine metrics. Therefore, it is difficult to clearly ascertain the performance of the CogC metric against other eight independent variables. In order to resolve this, we employ the Friedman statistical test [175] on the AUC values obtained by the metrics via the six prediction models over selected JFreeChart and Heritrix plugin project versions. The Friedman statistical test allocates ranks to the metrics for the purpose of identifying if, as an individual predictor, CogC is statistically similar or superior to the other eight selected complexity and change prediction metrics.

Table 3.10 Friedman's statistical test values for AUC corresponding to the nine metrics

Independent Variables	Mean Ranks for JFreeChart versions	Mean Ranks for Heritrix versions
CBO	4.13	**7.89**
CHalsE	4.90	5.33
CogC	*6.28*	*5.94*
CycloC	5.60	5.94
DIT	3.93	3.44
LCOM	4.23	5.33
NOC	1.36	1.44
RFC	**7.60**	5.94
WMC	6.95	3.72

The highest ranks are indicated in bold and the ranks obtained by CogC are represented in italics.

Table 3.10 reports the mean ranks scored by each of the nine metrics after applying the Friedman test. Considering that the test is conducted appertaining to the AUC values obtained by each of the models with respect to each of the nine metrics on the eight datasets, therefore the metric obtaining the highest mean rank is deemed to be the best in terms of performance.

The Friedman's statistical test results indicate that the metric with the best performance with respect to change-proneness in terms of AUC on all the five JFreeChart datasets is RFC with a mean rank of 7.60. The WMC metric follows next with a mean rank of 6.95, which is then followed by our estimated CogC metric with a mean rank of 6.28. Both the selected complexity metrics CycloC and CHalsE perform poorly in comparison to the CogC metric, obtaining mean ranks of 5.60 and 4.90. On the other hand, the CBO metric, with a mean rank of 7.89, is observed to have the best performance with respect to change-proneness on the three Heritrix data sets. The CogC metric comes second along with the CycloC metric and RFC metric, each obtaining a mean rank of 5.94. This is followed by the CHalsE and LCOM metrics, each obtaining mean ranks of 5.33.

Metrics DIT and NOC (with mean ranks of 3.93 and 1.36 for JFreeChart and 3.44 and 1.44 for Heritrix) are adjudged as the worst two performers among the nine selected independent variables for predicting version to version change-proneness of Java files for both JFreeChart and Heritrix datasets.

The Friedman statistical value for AUC with a degree of freedom eight was evaluated to be 113.16 (for JFreeChart) and 68.19 (for Heritrix), true for $\alpha = 0.05$. Additionally, p-values of 0.000 for both JFreeChart and Heritrix projects exhibit that the results obtained are true at a 95% confidence interval. This indicates that the performance of all the nine independent variables in terms of AUC is statistically significantly different; therefore, the null hypothesis of Friedman test stating that all parameters perform the same is rejected.

3.5.3 Evaluation using Feature Selection technique

The results from Table 3.10 empirically validate the CogC metric to be a better measure than most of the other selected independent variables for estimating version to version change-proneness of Java files belonging to JFreeChart and Heritrix versions. However, in a real-world setting there does not always exist a single

independent variable but a set of efficient independent variables that are employed in combination to predict a target variable. Therefore, with the motive to provide additional empirical substantiation regarding the efficacy of the CogC metric, we also consider performing Correlation-Based Feature Selection (CBFS) [186] on all the eight version datasets by using all the nine independent variables altogether. CBFS aids in a quick identification and screening of extraneous, unnecessary, and noisy features, and selects the best subset of features from the original feature domain that have a correlation with the dependent variable but do not have any relation with each other.

The metrics corresponding to each version obtained after the application of CBFS with the best first search technique have been stated in Table 3.11. It can be observed that the estimated CogC metric, along with the CBO metric, are the only measures out of the nine independent variables to be selected in six out of eight version datasets of JFreeChart and Heritrix as two of the relevant features that contribute to the accuracy of the model in predicting version to version change-proneness of Java files. Such consistent results across most of the selected versions give us an extremely compelling reason to employ the CogC metric as one of the independent variables for predicting the change-prone Java files of any future release of software projects like JFreeChart and Heritrix.

Additionally, the accuracy and AUC values obtained by the six prediction models on the five JFreeChart and three Hertrix datasets when no Correlation-Based Feature Selection (NCBFS) is applied versus after CBFS is applied are stated in Tables 3.12 and 3.13. Again, all the model results have been generated using Weka 3.8.0 and estimated using the ten-fold cross-validation technique. Highest accuracy and AUC values per column are indicated in bold. As observed from the Tables 3.12 and 3.13, for some cases like MLP and RF results on JFreeChart 0.6.0, CBFS degrades ML performance in terms of accuracy, albeit slightly. This, as explained by [187] is common with feature selection techniques and happens in those cases where some features that are predictive of very small areas of a specific instance space get eliminated. However, for most cases, the classification accuracy or AUC obtained by means of the reduced feature set (CBFS) is better than or equal to the accuracy or AUC obtained with the help of the complete feature set of nine metrics, thus

Table 3.11 CBFS results

Version Datasets	Metrics selected after CBFS
JFreeChart 0.6.0	CBO, CHalsE, *CogC*, CycloC, LCOM, NOC, RFC, WMC
JFreeChart 0.7.0	CBO, CHalsE, *CogC*, DIT, RFC
JFreeChart 0.7.1	CHalsE, LCOM, RFC
JFreeChart 0.7.2	CHalsE, *CogC*, CycloC, DIT, RFC
JFreeChart 0.7.3	CBO, *CogC*, DIT, RFC,
Heritrix 0.2.0	CBO, CHalsE, *CogC*, CycloC
Heritrix 0.4.0	CBO, CHalsE, DIT
Heritrix 0.6.0	CBO, CHalsE, *CogC*, CycloC, DIT, RFC

Table 3.12 Results of the predictive models generated over the five versions of JFreeChart

	JFreeChart 0.6.0				**JFreeChart 0.7.0**				**JFreeChart 0.7.1**				**JFreeChart 0.7.2**				**JFreeChart 0.7.3**			
	NCBFS		*CBFS*		*NCBFS*		*CBFS*		*NCBFS*		*CBFS*		*NCBFS*		*CBFS*		*NCBFS*		*CBFS*	
	Acc.	AUC	Acc.	AUC	Acc.	AUC	Acc.	AUC	Acc.	AUC	Acc.	AUC	Acc.	AUC	Acc.	AUC	Acc.	AUC	Acc.	AUC
NaïveBayes	81.39	**0.864**	**81.39**	**0.862**	**74.51**	0.731	66.67	0.753	**80.32**	0.758	81.15	0.756	84.62	0.756	84.62	0.756	**87.69**	0.860	**87.69**	**0.907**
MLP	**81.53**	0.768	**81.39**	0.786	69.61	0.757	**74.51**	0.769	78.69	0.731	**82.79**	0.721	86.92	0.715	**89.23**	0.750	83.84	0.826	86.92	0.827
AdaBoost	81.39	0.767	**81.39**	0.767	70.59	0.770	71.57	0.777	74.59	0.717	77.87	0.725	87.69	0.904	88.46	0.875	**87.69**	**0.898**	87.69	0.904
Bagging	76.74	0.805	76.74	0.822	72.55	0.808	**74.51**	**0.818**	76.23	0.765	77.87	0.780	88.46	0.870	84.62	0.845	84.62	0.843	85.38	0.864
Random Forest	79.07	0.825	77.91	0.829	71.57	**0.819**	73.52	0.816	78.69	**0.794**	77.87	0.775	**90.79**	**0.921**	87.69	**0.916**	84.62	0.865	83.84	0.862
LR	80.23	0.688	79.07	0.694	66.67	0.745	69.61	0.779	79.50	0.744	81.15	**0.786**	88.46	0.815	86.15	0.790	86.15	0.872	85.38	0.886

Table 3.13 Results of the predictive models generated over the five versions of Heritrix

	Heritrix 0.2.0				**Heritrix 0.4.0**				**Heritrix 0.6.0**			
	NCBFS		*CBFS*		*NCBFS*		*CBFS*		*NCBFS*		*CBFS*	
	Acc.	AUC	Acc.	AUC	Acc.	AUC	Acc.	AUC	Acc.	AUC	Acc.	AUC
Naïve Bayes	66.99	0.778	66.99	0.784	64.03	0.723	**73.38**	0.726	74.87	0.735	71.79	0.752
MLP	71.84	0.787	**79.61**	0.801	68.35	0.730	69.06	**0.786**	78.46	0.763	**78.91**	0.786
AdaBoost	71.84	0.797	72.82	**0.809**	**71.22**	0.750	68.35	0.768	76.92	0.745	77.95	0.774
Bagging	**76.69**	**0.816**	71.84	0.797	65.47	0.742	69.78	0.750	76.41	0.728	76.92	0.721
Random Forest	70.88	0.777	72.82	0.782	67.63	**0.771**	68.35	0.783	76.92	0.778	77.44	0.799
LR	72.82	0.770	71.84	0.763	65.67	0.756	65.47	0.731	**79.49**	**0.804**	77.44	**0.822**

indicating the efficacy of the reduced feature set (which includes CogC for most of the versions) for the prediction of the version to version change-proneness of Java files.

3.5.4 Performance evaluation of the prediction techniques

In order to evaluate the performance of the six employed prediction techniques, we conducted boxplot analyses with respect to the AUC values obtained by the techniques over the various JFreeChart and Heritrix plugin project versions. The small circle in the boxplot indicates an outlier and the extreme values are marked with a star. The *X*-axis indicates the prediction techniques under consideration, and the *Y*-axis indicates the range of the AUC values produced by the employed prediction techniques for each of the nine metrics corresponding to the selected versions of the software projects.

3.5.4.1 Performance evaluation when a single independent variable is employed

We select the AUC values gathered in Tables 3.8 and 3.9 by the selected prediction algorithms and visualize their results using boxplots (generated in Figures 3.1(a) and (b)) for ascertaining the best predictor out of the six prediction techniques when a single independent variable is employed for change-proneness prediction of Java files. Overall, with respect to the AUC measure; the statistical technique of Logistic Regression (LR) performs the best out of the six selected techniques for predicting version to version change-proneness of Java files for the JFreeChart and Heritrix versions when a single independent variable is employed. This is because it obtains the highest median values (indicated by the dotted reference line) for AUC, equal to 0.726 for JFreeChart and 0.658 for Heritrix. The Bagging and MLP techniques come next with median values slightly lesser than LR as seen in Figures 3.1(a) and (b). The Naïve Bayes technique with a median value of approximately 0.669 is adjudged as the worst predictor for version to version change-proneness of Java files for the JFreeChart versions. Whereas, the Random Forest technique, with the least median value of 0.600 performs the worst among the six selected techniques for predicting version to version change-proneness of Java files for the Heritrix versions.

3.5.4.2 Performance evaluation when the independent variables are employed in combination

The AUC values obtained corresponding to columns NCBFS for the JFreeChart and Heritrix versions in Tables 3.12 and 3.13 are utilized to build boxplots illustrated in Figures 3.2(a) and (b). As observed from the Figure 3.2, the Random Forest technique performs the best out of the six prediction techniques for predicting version to version change-proneness of Java files for both the JFreeChart and Heritrix versions when the nine metrics are employed in combination. This is indicated by the dotted reference line signifying the highest median values for AUC, equal to 0.825 for JFreeChart and 0.777 for Heritrix.

Additionally, from Tables 3.12 and 3.13, we also conducted boxplot analyses for the AUC values obtained when only the metrics selected post the application of CBFS were employed as independent variables, indicated in Figures 3.3(a) and (b). The Random Forest technique yet again performs the best out of the six prediction techniques for predicting version to version change-proneness of Java files, but only for the JFreeChart project. On the other hand, the MLP marginally outperforms the Random Forest technique and is adjudged as the best prediction technique for predicting version to version change-proneness of Java files over the selected versions of Heritrix. Thusly, it can be said that the results are not alike for different datasets. Yet, if datasets have many similar characteristics, then similar results might be obtained (i.e. we may get the same ML methodology as being the best one). Nevertheless, the highest median values for AUC for both the Random Forest technique (0.831 for JFreeChart) and the MLP technique (0.786 for Heritrix) are higher than those obtained when NCBFS is applied.

Note: We prefer AUC to Accuracy for conducting the performance comparison of the metrics and the prediction techniques. This is because accuracy evaluates, for a given threshold, the percentage of points correctly classified, regardless of which class they belong to. However, in situations where there exists a large imbalance between the classes, the phenomenon of 'Accuracy Paradox' is commonly encountered in which the model predicts the value of the majority class for all predictions, therefore attaining a high classification accuracy. In contrast, AUC does not bias on the size of test or evaluation data. Since as observed in Table 3.3, there does exist an imbalance between the changed and the unchanged files for most of the JFreeChart and Heritrix

versions selected, we select AUC as the comparison measure, which has been statistically proven to be more discriminant than accuracy, especially for imbalanced datasets [188].

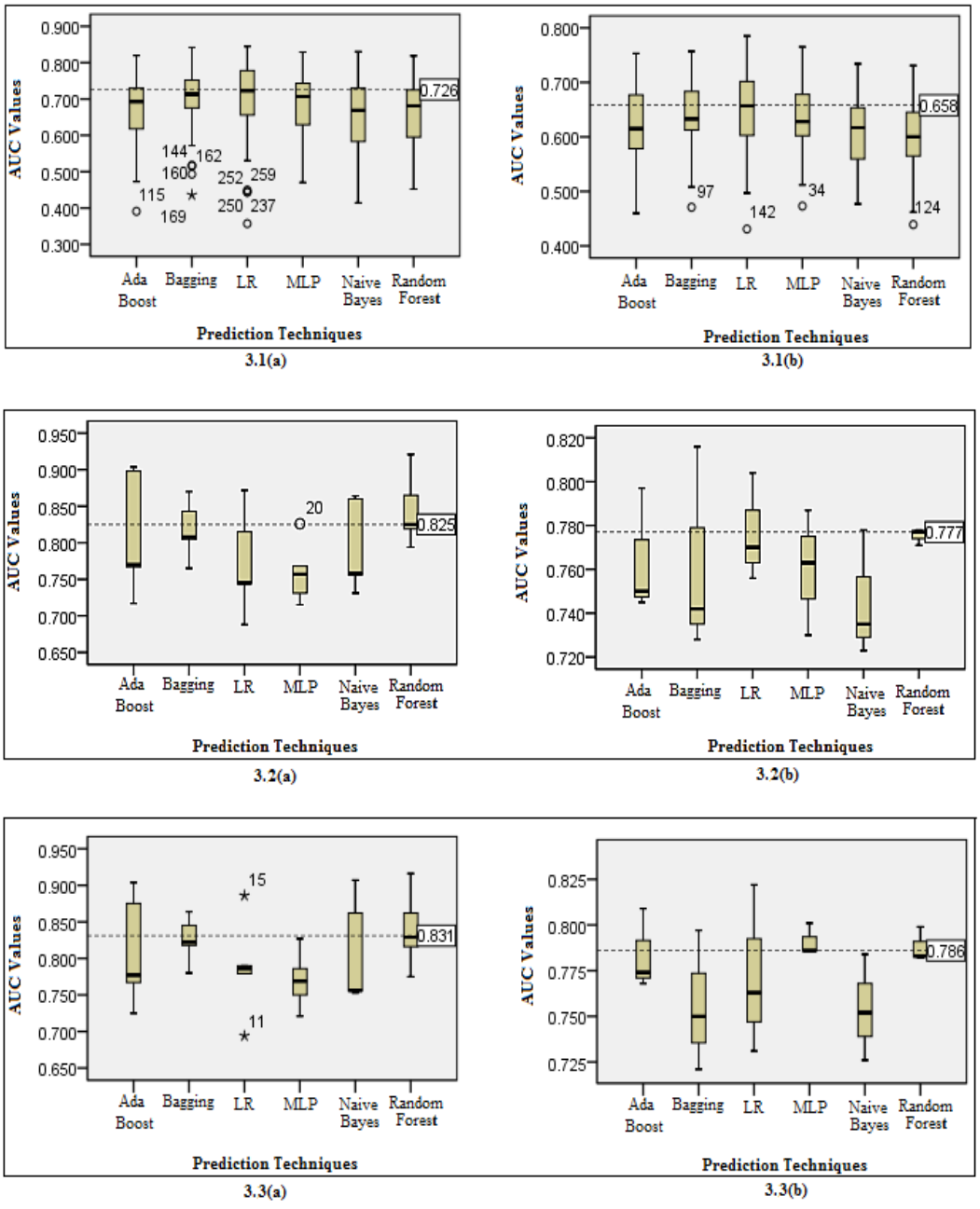

Figure 3.1 Boxplot analysis of the prediction techniques when the metrics are employed individually; **Figure 3.2** Boxplot analysis of the prediction techniques when the metrics are employed altogether and without any Feature Selection (NCBFS); **Figure 3.3** Boxplot analysis of the prediction techniques when the metrics are employed altogether after applying Feature Selection (CBFS); for: (a) Five JFreeChart versions, and (b) Three Heritrix versions.

3.6 Discussion of Research Questions

This section comprises of a discussion of the research questions (RQs) specified in Section 3.1. Although answers have been provided to both of these RQs in various sub-sections of Section 3.5 of this chapter, we try to synopsize them below.

RQ 1: *Can cognitive complexity as a metric predict software change and is cognitive complexity, as a quantifier of version to version change-proneness of Java files, analogous or superior to some of the other similar previously introduced complexity and change-proneness prediction measures?*

We initially attempted to examine the efficacy of the Cognitive Complexity (CogC) and each of the other eight independent variables selected with respect to version to version change-proneness of Java files using six prediction techniques. As observed from the model results stated in Tables 3.8 and 3.9 for the JFreeChart and Heritrix plugin projects, the CogC metric exhibits an above par predictive performance with respect to certain metrics like DIT, LCOM, and NOC for both the software projects. Yet it is difficult to evidently establish if it, in fact, demonstrates one of the best performances as an individual predictor for version to version change-proneness of Java files.

Therefore, we chose an appropriate statistical test to lucidly determine the relative performances of the nine selected metrics. The results from the Friedman statistical test provided in Table 3.10 depict that the CogC metric exhibits third-best performance for the selected JFreeChart versions and second-best performance (in conjunction with the CycloC and RFC metrics) for Heritrix versions. Also, the results obtained are true at a 95% confidence interval. This confirms that the performance difference between the nine selected independent variables is not random and the evaluated CogC metric does prove to be superior to most of the other selected metrics for version to version change-proneness prediction of Java files over the selected JFreeChart and Heritrix versions.

Furthermore, the results stated in Table 3.11 also demonstrate the CogC metric to be a good indicator of change-proneness since it is one of the metrics to be selected in the reduced feature sets for six out of eight dataset versions of the JFreeChart and Heritrix software after the CBFS technique is applied. Additionally, it can be observed from

Tables 3.12 and 3.13 that the performance of a majority of the models, in terms of the selected performance measures, improved after the metrics obtained from CBFS method were used as independent variables versus when they were employed altogether without feature selection (NCBFS). This validates that the CogC metric does contribute to the accuracy of the models and should indeed be employed for predicting version to version change-prone Java files for future releases of JFreeChart and Heritrix software.

RQ 2: Which of the techniques attests to be the best for change-proneness prediction of Java files when the selected complexity and change prediction metrics are used as independent variables?

Among the six selected prediction techniques, the boxplot results from Section 3.5.4 indicate that the statistical technique of LR performs the best in terms of AUC for both the JFreeChart and Heritrix datasets when the selected complexity and change prediction metrics are used individually as single independent variables, thus outperforming all the ML techniques employed. From the remaining techniques, Bagging follows LR by exhibiting the second-best performance, again consistent for both the JFreeChart and Heritrix versions employed. This concludes that LR and Bagging can be employed to predict change-prone Java files of any future release of JFreeChart and Heritrix or of any similar dataset, especially in those cases where one wants to test the prediction efficacy of a single independent variable against version to version change of a Java file.

On the other hand, when all the nine independent variables are used altogether and without any feature selection, the Random Forest technique performs the best in terms of AUC for both the JFreeChart and Heritrix datasets for predicting the change-prone files. This result, in particular, is accordant with the results observed in [11], where the Random Forest technique was adjudged as the best method for three Java-based software datasets for change-prone class prediction. However, the boxplot results in Figure 3.3 of Section 3.5.4 are not consistent for the JFreeChart and Heritrix versions when the metrics obtained via CBFS are applied as independent variables, with the Random Forest technique performing the best for the JFreeChart software versions and MLP technique performing the best for the Heritrix software versions. This indicates that the prediction results of the techniques might vary from software to

software even though these techniques are being analyzed on source code developed using the same language and are being assessed using the same set of independent and dependent variables.

The next section summarises this chapter.

3.7 Chapter summary

The purpose of this chapter was to report the results of a preliminary experiment probing the importance of Cognitive Complexity (CogC) metric as a quantifier for estimating version to version change-proneness of of Java files.

We employed five successive versions of JFreeChart and three successive versions of Heritrix plugin projects as target projects, and estimated the *change* results and the values of CogC for each of the version's source code Java files to construct eight datasets. One statistical analysis and five ML techniques were used to build models with the motive to draw inferences regarding the importance of the CogC metric with respect to the version to version source code change-proneness. Two previously introduced complexity measures and six change prediction measures were also evaluated on the same eight datasets to provide an exhaustive comparative analysis. Although the model results were reported using some commonly employed performance measures, AUC, with the aid of Friedman's statistical test, were primarily used to compare and rank the relative predictive capability of the each of the measures.

Observing the results drawn from the comparative analysis conducted in Section 3.5, it was found that:

- CogC, as an individual quantifier of version to version change-proneness of Java files, clearly outperforms certain commonly employed metrics for change prediction like Cumulative Halstead Effort (CHalsE), Depth of Inheritance Tree (DIT), Lack of Cohesion in Methods (LCOM) and Number of Children (NOC) for both the target projects analyzed.
- The Friedman test results indicate that the CogC metric exhibits comparable yet competitive predictive ability against widely used change prediction metrics like Cyclomatic Complexity (CycloC) and Response For Class (RFC), again for both the target projects. Moreover, the test results are significant which confirms that the performance difference between the selected

independent variables is not random and the evaluated CogC metric does prove to be superior to most of the over the other selected metrics for version to version change-proneness prediction of Java files.

- The feature selection results validate the contribution of the CogC metric to the accuracy of the change-proneness prediction models generated. Hence, albeit the CogC metric individually cannot be adjudged as the best quantifier of version to version change-proneness of source code files across the selected versions of the two projects, the consistency in the empirical results drawn in this analysis establish its contribution as a distinguishable measure of version to version source code change. Therefore, cognitive complexity should be incorporated as one of the independent variables among other independent variables when evaluating source code with respect to change-proneness.

CHAPTER 4
PREDICTING SOFTWARE CHANGE USING INTELLIGENT COMPUTING METHODS

This chapter presents a pervasive framework for software change prediction by investigating 31 techniques from the categories of Intelligent Computing Methods (ICMs) and Statistical approaches for creating version to version change-prediction models with respect to Java files. The performance of the models is assessed during two validation scenarios: k-fold intra-release validation and inter-release validation, where the latter is useful for estimating the trend of change-proneness of files in the upcoming versions.

This chapter is structured as follows: Section 4.1 provides a brief revisit to the research background studied in Section 2.2 of Chapter 2 and presents the research questions (RQs) formulated for this chapter. Section 4.2 reports the pragmatic framework designed for change-proneness prediction of Java files and also discusses the dependent and independent variables, software projects employed in this chapter, and empirical data collection. Section 4.3 elaborates the experimental setup of the study conducted in this chapter; while Section 4.4 states and discusses the results obtained in accordance with each of the given RQs. Section 4.6 re-counts the conclusions of the analysis in a summary and therefore concludes the chapter.

4.1 Background and Research Questions

It can be seen from the literature presented in Section 2.2 of Chapter 2 that most of the articles only consist of a ten-fold validation of the selected techniques for the prediction of software change. Seven studies [13, 34, 47, 55, 58, 60, 103] analyse the effectiveness of the techniques using inter-project evaluation and only two studies [13, 48] employ an inter-release validation. Additionally, there have been certain articles [9, 45, 62] that state that the ML techniques obtain comparable performances and even sometimes underperform in comparison to ICMs like evolutionary and search-based techniques. However, these articles have not included the Random Forest (RF) technique in their comparative analysis, which, has often been touted as the most efficient classifier to predict change-proneness of software among most of

the available techniques. There has also been some application of statistical tests with authors in [31, 47, 63] employing the t-test and most of the other studies [9, 30, 32, 48, 55, 61, 62] employing the Friedman test along with the port-hoc Wilcoxon test for establishing the disparities between the performances of several techniques. However, the t-test relies on many presumptions like the normal distribution of data and it is not advisable to use Wilcoxon's test sans Bonferroni correction, as family-wise error is not considered.

The pragmatic assessment conducted in this chapter aids in supplying valid answers to the research questions (RQs) stated as follows:

- RQ1: What is the predictive capability of the various selected prediction techniques, by and large, with respect to predicting version to version change-proneness of Java files when 'k'-fold cross-validation with feature selection is employed?
- RQ2: What is the performance of the models with respect to predicting the trend of version to version change-proneness of Java files?
- RQ3: Is the predictive performance of the change-proneness prediction models developed via k-fold cross-validation statistically similar to or different from the performance of the models constructed by means of an inter-release validation?
- RQ4: Which are the best and the worst techniques for change-proneness prediction of Java files of the selected target projects?

We believe that no former research has been performed which specifically: (1) Conducts a broad comparative analysis of statistical approaches and ICMs in the context of version to version change-proneness prediction, (2) Employs data gathered from multiple releases of two plugin projects using 17 OO metrics to obtain generalized and unbiased results, (3) Statistically analyses the attained results using multiple statistical tests for the performance comparison of the selected prediction techniques, and (4) Performs an inter-release validation of the models with the purpose of examining the efficiency of the selected techniques in predicting the trend of change-proneness of Java files in the upcoming versions. Moreover, the testimony acquired from such exhaustive data-based comparative pragmatic analyses can assist the software researchers and practitioners to cultivate ample corpus of knowledge to accept/reject a given hypothesis [57].

The next section elaborates the software change prediction framework proposed in this chapter.

4.2 Pragmatic framework for software change prediction

A pragmatic framework for change-proneness prediction is presented in this section for the purpose of performing an extensive assessment and comparison of the selected prediction techniques. Figure 4.1 illustrates the framework proposed in the study. As seen in Figure 4.1, we employ multiple releases of the same plugin projects employed in Chapter 3: JFreeChart and Heritrix for analysis in this chapter as well. Real world datasets commonly show the particularity to have anomalies along with a number of samples of a given class under-represented compared to other classes [189]. Therefore, datasets generated from the target projects are pre-processed to solve the problem of class-imbalance, after which the outliers of the datasets are identified and removed. The datasets are also individually scrutinized for finding the best subset of variables for prediction via an appropriate feature selection technique. After this, all the selected prediction techniques are exclusively applied on the each of the selected version datasets and the models generated are subjected to two validation settings (VS1 and VS2).

In VS1, every prediction model generated apropos to version to version change-proneness is trained and tested using the same version dataset. In VS2, the model is trained using version $_{i}$ and validated using the successively released version $_{i+1}$.

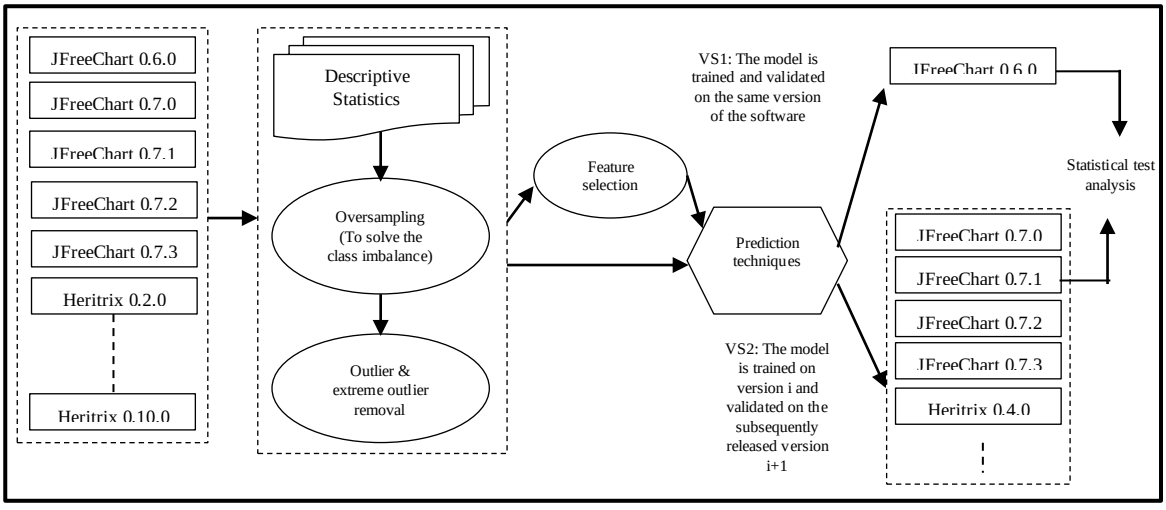

Figure 4.1 Framework for predicting the change-proneness of Java files using two validation settings

For instance, model is trained using JFreeChart version 0.6.0 and validated on JFreeChart 0.7.0. Similarly, model is trained using JFreeChart 0.7.0 and validated on JFreeChart 0.7.1. The procedure is repeated over all the chosen releases of the two target projects. The results of change prediction models generated are assessed via various performance metrics. We further employ Kruskal-Wallis and Scott-Knott cluster analysis with Borda ranking for statistically scrutinizing the results of the study.

It is imperative to note here that we do not directly proceed with an inter-release validation (VS2) before analysing the selected prediction techniques via a 'k-fold' validation (VS1). This is due to the fact that a prediction model that exhibits a very poor performance during VS1 i.e. when it has been trained and tested using the same version dataset, will not exhibit high performance when validated via VS2 i.e. trained using version $_{i}$ and validated using the successively released version $_{i+1}$. Such models are therefore identified via VS1 and not included for analysis with VS2.

The following sub-sections consist of the description of the variables employed, as well as the target projects selected for analysis. The procedure employed to gather the data corresponding to the variables employed is also provided.

4.2.1 Variables in the study

Figure 4.1 makes it clear that the primary objective of our prediction analysis in this chapter is to test the competency of the various selected prediction techniques in the classification of the change-prone Java files for a specific release during two different validation settings: VS1 and VS2. Therefore, we utilize the same binary dependent variable of change-proneness that has also been employed in the Chapter 3.

As far as the independent variables are concerned, this chapter employs the eight metrics analysed along with the CogC metric in Chapter 3, in addition to some of the Martin's metrics [190], size metrics like SLOC along with some of the rarely employed metrics like Maintainability Index [191] and Instability [190]. Each metric is employed in such a way that it exploits a different source of code information and represents some fundamental software traits such as such as size, cohesion, coupling, inheritance, and complexity and are summarized in Table 4.1.

Table 4.1 Software metrics[4] selected

Source code analysis tool	Software Metric & Description
JHawk 6.1.3 [180]	*Source lines of code(SLOC)*: Sum of lines in the code file that have source code. However, a line can have source code as well as a comment and therefore amounts to several metrics. *Total Cyclomatic Complexity(CycloC)* [20, 21]: CycloC computes the number of linearly independent paths within a particular piece of code. The tool calculates CycloC by adding 1 to the total number of keywords for decision points. *Cumulative Halstead Length(CHL)* [20] : The total of number of operators(OP) and operands(OD) in a code is called as the Cumulative Halstead Length. *Cumulative Halstead Volume(CHV)*[20]: CHV indicates the complete information that a person reading a source code needs to comprehend so as to understand its meaning. CHV is calculated as = CHL * $\log_2$(V); where V is Halstead Vocabulary and dignifies the total of the number of distinct operatands(UOD) and operators(UOP). *Cumulative Halstead Effort(CHE)* [20]: CHE of a source code denotes the total intellectual effort required for its refabricating and is given as = CHV* DIF; where DIF is Halstead Difficulty and is calculated as (UOP/2) * (OD/UOD). *Cumulative Halstead Bugs(CHB)*[20]: CHB calculates the number of bugs that could be present in a given source code and is calculated as CHV/3000. *Maintainability Index (MI)* [191]: MI indicates the ease of maintaining a particular source code file and is given as – MI = 171 - 3.42ln(aE) - 0.23aV(g') - 16.2ln(aLOC); Where aE indicates the average Halstead Effort of the code file, aV(g') indicates the average extended cyclomatic complexity of the code file and aLOC is the average numbers of lines of code per module in that file.
Stan4J [181]	*Afferent Coupling(AC)* [190]: The AC metric identifies the number of interfaces and classes from other files that depend on the classes in a given file. *Efferent Coupling(EC)* [190] : EC is determined as the number of types inside a file's class which depend on other classes' types.

[4] Again, granting most of the metrics included in our analysis are class-level measures, our study is performed at a file-level. When files are found to be constituted of more than one class, the sum of numeric values of the metrics achieved by its every constituent class is taken into consideration.

Table 4.1 Software metrics[4] selected

Source code analysis tool	Software Metric & Description
	Instability [190]: Instability is estimated by calculating the effort needed to alter a file sans the alteration of other files in the software. It is computed as: Instability = EC / (AC + EC). *Weighted Methods per Class(WMC)* [17]: WMC measures the sum of the complexities of all class methods. It gauges the amount of effort needed for the development and maintenance of a specific class. *Depth of inheritance (DIT)* [17]: The depth of a class in the inheritance hierarchy is evaluated as the highest count of nodes originating from the class to the root node. *Number of children (NOC)* [17]: Number of Children (NOC) is the total count of direct subclasses of a given class. *Coupling between objects (CBO)* [17]: CBO relates to the notion that if the declared methods of one class use instance variables or methods defined by the other class, then the two classes are termed to be coupled to each other. It is evaluated according to the formula: CBO= AC+EC *Response for a class (RFC)* [17]: This metric is the sum of all the methods present in a class and all the other methods which get called by methods of this class. Since this is a set, every called method is evaluated just once irrespective of the number of times it is called. *Lack of cohesion (LCOM)* [17]: This metric calculates what percentage of methods contained in a class use a given instance variable belonging to the class. A low percentage indicates a high cohesion between class methods and data. *Cognitive Complexity (CogC)*: CogC evaluates the software developer's amount of complication in understanding a source code component. It is appraised via calculating the cognitive weights of the Basic Control Structures(BCS) contained in the source code of the software and is given as: $$CogC = \sum_{k=1}^{x} [\prod_{j=1}^{m} \sum_{i=1}^{n} W_c(k,j,i)]$$ where CogC is the total of cognitive weights of x linear chunks of the software lines present in specific BCSs where each chunk may well be comprising of m levels of nesting BCS's, wherein every level includes n linear BCSs.

4.2.2 Target projects

The plugin software, JFreeChart and Heritrix, previously analysed in Chapter 3 are again selected for the evaluation of the various prediction techniques under the two validation settings. Eight releases (five JFreeChart and three Heritrix) were analysed to establish the efficacy of the CogC metric with respect to version to version change-proneness prediction of Java files in Chapter 3. Two additional successional releases of the Heritrix software: Heritrix 0.8.0 and Heritrix 0.10.0 are included in the analysis conducted in this chapter, making the total number of software versions analysed equal to ten, details of which have been given in Table 4.2.

Table 4.2 Version specifics of the selected software projects

Software projects	Total size in LOC	Total number of Java files
JFreeChart 0.6.0	5,700	86
JFreeChart 0.7.0	7,870	105
JFreeChart 0.7.1	8,894	128
JFreeChart 0.7.2	9,222	130
JFreeChart 0.7.3	9,318	131
Heritrix 0.2.0	8,331	125
Heritrix 0.4.0	11,688	168
Heritrix 0.6.0	12,751	200
Heritrix 0.8.0	42,351	223
Heritrix 0.10.0	49,441	249

4.2.3 Empirical data collection

The Cognitive complexity (CogC) metric has been calculated manually in terms of Cognitive Weight Units for all the Java files of the selected JFreeChart and Heritrix datasets according to the lucid guidelines given in our previous chapter, Chapter 3. The work done in the previous chapter essentially consisted of a preliminary analysis identifying the significance of the CogC metric in regard to change-proneness wherein the empirical results were drawn using fewer releases of the Heritrix software and using six prediction algorithms. The work performed in this chapter acts a major extension to it by assessing the prediction techniques with respect to two extra versions of the Heritrix software. Moreover, sixteen metrics proven to be useful for change-proneness prediction are employed along with the CogC metric and the results are analysed using the 31 prediction techniques which not only include statistical and ML approaches but also include other ICMs such as Fuzzy techniques and

Evolutionary methods. Also, we carry out statistical tests for proving the effectuality of the results obtained. Furthermore, determining the statistical significance and performance comparison of the prediction techniques during inter-release validation (VS2) for change-proneness trend estimation is also conducted.

The same static code analysis tools employed in Chapter 3 have been employed in this Chapter to gather the numeric values corresponding to the remaining sixteen independent variables, in regard to every Java file that exists in all the selected JFreeChart and Heritrix software versions. These tools: JHawk 6.1.3 [180] and Stan4J [181] compute the metrics according to their standard definitions.

As mentioned in Section 4.2.1, we utilize the binary dependent variable of change-proneness that has also been employed in the Chapter 3. Since the analysis conducted in Chapter 3 consisted of the same eight versions of JFreeChart and Heritrix that have been employed for analysis in this chapter, we utilize the same change-statistics of versions JFreeChart 0.6.0 to JFreeChart 0.7.3 and Heritrix 0.2.0 to Heritrix 0.6.0 that were gathered using the AntiCutandPaste tool [182]. Two additional versions of Heritrix have been added in this chapter for analysis, the change-statistics of which have also been evaluated with the same tool.

An outline on the change-proneness of Java files present in the ten releases considered in our analysis is provided in Table 4.3. We grouped the binary values of the change statistic along with the numeric values of the seventeen independent variables corresponding to vis-à-vis just those Java files that have been included in the

Table 4.3 Change statistics of the Java files in the selected software project releases

Versions	Total number of Java files	Number of Java files used in the next version	Number of Java files used without change in the next version	Number of Java files used with change in the next release
JFreeChart 0.6.0	86	86	67	19
JFreeChart 0.7.0	105	102	42	60
JFreeChart 0.7.1	128	122	94	28
JFreeChart 0.7.2	130	130	112	18
JFreeChart 0.7.3*	131	130	92	38
Heritrix 0.2.0	125	113	48	65
Heritrix 0.4.0	168	155	70	85
Heritrix 0.6.0	200	195	128	67
Heritrix 0.8.0	223	213	89	124
Heritrix 0.10.0*	249	231	171	60

** We examined the successional version JFreeChart 0.7.4 for the change statistics of JFreeChart 0.7.3 and the successional version Heritrix 1.0.0 for the change information of Heritrix 0.10.0.*

successional version (refer to column 3 of Table 4.3) for each of the ten specific releases, making the total number of data points equal to 1,477.

4.3 Experimental Setup

The following steps are undertaken for the purpose of empirically answering the RQs stipulated in Section 4.1 of this chapter:

1. Use of an efficient resampling method to rectify the imbalance in the selected version datasets.
2. Elimination of outliers in the resampled datasets.
3. Selection of an appropriate feature section technique to identify the most relevant features to the dependent variable.
4. Selection of apposite techniques (comprising of ICMs and statistical approaches) for predicting the dependent variable.
5. Selection of performance measures for gauging the predictive competence of the techniques chosen at step 4.
6. Usage of competent validation methodologies for determining the factual appropriateness of the predicted models.
7. Choosing fitting statistical tests for deciding the performance excellence of one prediction technique over the others.

The subsequent sub-sections consist of a description of resampling technique, the outlier detection and removal methodology, the feature selection technique, prediction techniques selected, performance evaluation metrics, validation approaches and statistical tests employed for an additional experimental substantiation.

4.3.1 Resampling of unbalanced datasets

Datasets collected corresponding to the various software development tasks normally experience the class imbalance [48, 189] problem, such as the NASA MDP datasets [192] which contain very few defective examples in comparison to the non-defective examples. Such imbalance creates difficulties for the prediction techniques as there is an under-portrayal of one class and an over-depiction of the other. Resampling techniques can be employed for the purpose of catering to the problem of class imbalance and include the alteration of samples in a dataset via either an under-sampling of the majority class or an oversampling of the minority class.

In this chapter, we utilize SMOTE (synthetic minority oversampling technique) [193] for achieving an identical class distribution in each of the ten datasets. SMOTE operates by creating new "synthetic" examples for the under-represented class instead of duplicating prevailing ones. For the creation of a synthetic example, SMOTE looks for the nearest neighbours of a minority-class example that also have the same class category. Post this step, a novel synthetic sample is fabricated at a random point on the space that connects the two authentic samples and this novel class is labelled as the minority class. This is done assuming that every dimension of feature space that similar samples belong to develops a ratio scale. Different synthetic samples are then generated using different pairs of neighbours.

Since SMOTE employs a user-defined factor that determines the number of new records to create i.e. 100% more examples will be created with a value of 100, therefore a different parameter value has been employed for each of the ten datasets as stated in column two of Table 4.4. The parameter values have been decided in such a way that SMOTE creates sufficient synthetic instances that render the total number of minority class occurrences equal to the number to occurrences in the majority class.

4.3.2 Outlier detection and removal

Another indispensable step in data pre-processing consists of an outlier examination. Outliers are defined as prevalence estimates that lie considerably beyond the acceptable range. With the objective of obtaining equitable results, we efficiently detect and eliminate all the outliers and extreme outliers from each of the ten datasets by means of the Inter Quartile Range filter. Interquartile Range (IQR) [194] is a statistical technique representing the distance in the middle of the 25th and the 75th percentile. This distance comprising of 50% of the data is called as the IQR which remains immune to the effect of outliers.

IQR are also compared to the length of the box in a boxplot wherein data can be easily grouped into quartiles. WEKA 3.8.2 [185] specifies an attribute value filter for the IQR which sorts the data by the mean of the sample's distance from all data points with respect to an attribute. Data points with distance more than 1.5 times the IQR lie outside the boxplot and are called as outliers, and those more than 3 times the IQR are called as extreme outliers.

Table 4.4 Change statistics after resampling and outlier and extreme outlier detection and removal

Versions	SMOTE	Inter Quartile Range filter		Change statistics after data pre-processing	
	% of minority class balancing	*Number of outliers detected*	*Number of extreme outliers detected*	*Number of Java files used without change in the next release*	*Number of Java files used with change in the next release*
JFreeChart 0.6.0	253	17	5	50	66
JFreeChart 0.7.0	17	19	9	39	51
JFreeChart 0.7.1	221	24	12	70	88
JFreeChart 0.7.2	522	22	17	92	103
JFreeChart 0.7.3	142	27	11	67	89
Heritrix 0.2.0	34	22	32	34	39
Heritrix 0.4.0	20	23	41	58	61
Heritrix 0.6.0	91	39	69	68	103
Heritrix 0.8.0	39	47	61	76	90
Heritrix 0.10.0	185	63	93	90	141

Columns three and four in Table 4.4 show the number of outliers and extreme outliers identified and removed in each data set after the resampling technique of SMOTE is applied. Table 4.4 also indicates the final values of the change statistic obtained post these two steps of data pre-processing.

4.3.3 Feature selection method

Research articles in effect indicate that extraneous attributes, in conjunction with redundant attributes, are capable of adversely affecting the accuracy of the predictors [6, 28, 55, 59]. Hence, the application of a pertinent feature selection technique to remove all unnecessary, redundant and noisy features is vital to render the classification of new instances more accurate. This chapter employs one of the most commonly used techniques in software change prediction, Correlation based Feature Selection (CBFS) [11, 12, 26, 32, 45, 100], to effectually shortlist the best independent variables for change-proneness prediction of files out of the seventeen selected independent variables. As mentioned earlier in Section 3.5.3 of Chapter 3, CBFS relies on the hypothesis that an efficient attribute subset holds attributes/features highly correlated with the outcome, but uncorrelated with one another, and chooses an efficient subset of attributes by scrutinizing their individual predictive capability and their redundancy among one another.

4.3.4 Prediction techniques incorporated

Table 4.5 provides a brief report of the 31 prediction techniques employed in this chapter comprising of statistical techniques such as LR and FLDA and ICMs such as Machine Learning techniques, evolutionary techniques and fuzzy-based methodologies. We also analyse AntMiner, an ant colony based optimization technique. As detailed in Section 2.2.2 of Chapter 2, ICMs are "a consortium of data-driven methodologies which include Machine Learning (ML), fuzzy logic, and evolutionary computing among others". It is crucial to state here that the prediction techniques employed in this analysis have been carefully chosen in a way so that each classification category has a minimum of one technique being analysed. Additionally, ICMs that have not been explored vis-à-vis change-proneness prediction in the existing literature have also been analysed for their competency.

Table 4.5 Prediction techniques incorporated in this analysis

Prediction Technique		Description
Bayesian Classification		
BN	Bayesian Network [32, 45, 47, 57, 103]	The Bayesian classifiers are focused on determining the degree to which the probability that a hypothesis is correct depends on former unaware information. The *BN* allows the user to specify which attributes are conditionally independent. *NB,* on the other hand assumes conditional independence among the attributes.
NB	Naïve Bayes [12, 45-48]	
Functions		
FLDA	Fisher's Linear Discriminant Analysis [9, 61]	*FLDA* performs linear classification by projecting the data to a lower dimension in a manner that the projected means of categories are distant keeping the range of the projected data minor. *LR* is a conservative modelling methodology that assists in the description of the relationship between certain Xs (independent variables) and a twofold categorical dependent variable (Y), demonstrating an event's pragmatic occurrence or non-occurrence. *MLP* uses back-propagation algorithms to train the connection weights eventuslly taking longer time to train, because the back-propagation algorithms rely on greedy search algorithms like gradient decent. *RBFN* differs from MLP, because in RBF networks the hidden layer performs some computation using a Gaussian Radial Basis Function (RBF) which has linear parameters. The *SMO* technique picks the size of the working set to be two and employs a simple method for solving the reduced minor quadratic programming issues which occur while training the Support Vector Machines (SVMs). The *SPegasos* algorithm, on the other hand, is a SVM model taking advantage of the Stochastic Gradient Descent. The
LR	Logistic Regression [11, 33, 34, 46, 47, 49, 55]	
MLP	Multi-Layer Perceptron[11, 32-34, 47, 55]	
RBFN	Radial Basis Function Neural Network [195]	
SMO	Sequential Minimal Optimization [195]	
SPEG	SPegasos [196]	

Table 4.5 Prediction techniques incorporated in this analysis

Prediction Technique		Description
		SPegasos technique has the straightforwardness and rapidity that online learning techniques possess but is sure to approach to a realistic dichotomous SVM result.
Meta-classification		
ADB	ADaBoost [6,30, 47, 48, 55, 61, 62]	The *ADB* methodology groups diverse results from learning techniques in an effort to obtain a combined prediction wherein the training case weights are changed in each cycle. This is done to coerce the learning algorithms in giving extra emphasis on instances that were assessed erroneously before and reduced importance to those instances that were predicted correctly. A plurality voting scheme is employed by the *Bagging* technique to group various results from a learning technique with the aim to find a joint single prediction. *Dagging* generates various disjoint, stratified folds from the dataset and every fold is submitted to a copy of a given base classification technique. A majority voting scheme is employed to make the final predictions. *FC* conducts classification on dataset that has been processed via an arbitrary filter that employs some mathematical evaluation. *LB* classifies via a regression technique as the base learner that is principally enhanced to cater to noisy data and handles multi-class problems. The *RSS* classifier resembles the bagging technique but in RSS, random subsets from the given dataset are selected to be random subsets of the attributes as opposed to in bagging wherein the samples are selected with replacement.
BAG	Bagging [6, 30, 48, 56, 57, 62]	
DAG	Dagging [197]	
FC	Filtered Classifier [198]	
LB	LogitBoost [30, 48, 58, 62]	
RSS	Random Sub Space [198]	
Miscellaneous		
CHIRP	Composite Hypercubes on Iterated Random Projections [199]	*CHIRP* is a non-parametric classifier that deals with nonlinear separability, computational complexity, and the curse of dimensionality. For categorized inputs in a high-dimensional setting, CHIRP uses computationally-effective approaches for generating 2D projections and sets of rectangular regions on projections comprising of data from a particular class category. CHIRP systematizes these region and projection sets into a list of decisions for allocating scores to new data points. The *FLR* classifier induces expressive, decision-making rules in a mathematical lattice data domain including space R^N. Learning is performed quickly and incrementally via the computation of disjunctions of join-lattice interval conjunctions (a hyperbox in R^N). It has been claimed to be a better classifier in comparison to C4.5 decision trees well as back-propagation neural networks. *VFI*, on the other hand enables each feature to participate in the classification. Every feature submits a vote for one of the classes out of the 'n' available classes and the class with the highest votes is declared to be the predicted class. *Ant-Miner* [14] is a data mining algorithm based on Ant
FLR	Fuzzy Lattice Reasoning [200]	
VFI	Voting Feature Intervals [185]	
AM	Ant Miner [201]	

Table 4.5 Prediction techniques incorporated in this analysis

Prediction Technique		Description
		Colony Optimization (ACO) paradigm and works on the principle of an ant colony's foraging patterns. It begins with an empty rule set and creates one rule at a time by adding terms to the partial rule in a probabilistic manner. The rules are updated and pheromone is added iteratively based on the quality of the rule. Finally, the rules above a certain pheromone threshold are selected as the optimal rules. Standard parameter values provided by KEEL were used to execute the Ant Miner algorithm.
Decision Rules		
DTNB	Decision Table/Naive Bayes hybrid classifier [202]	The *DTNB* hybrid classifier appraises the importance of segregating the features into two disjoint groups: one for NB and the other one for the decision table. All features are modeled by the decision table to begin with. At every step of a forward selection search, certain features are using the NB and the remaining use the decision table. *FURIA* extends the well-known RIPPER algorithm, while conserving its benefits. In addition, instead of conventional rules and rule lists, FURIA learns via fuzzy rules and consists of unordered rule sets. Furthermore, it utilizes an effective rule stretching approach for dealing with uncovered examples. The *Modlem* algorithm performs induction of decision rules within Cluster Splitting based Resource Allocation and directly handles numerical attributes during rule induction. *NNGE* algorithm uses non-nested generalized exemplars (that can exhibit just one example from the training database or hyperrectangles signifying two or more examples of a particular class from the training database that can be viewed as if-then rules). This new example is then categorized as a class member of the closest exemplar using Euclidean distance. *PART* algorithm uses a divide-and-rule method, constructs a partial C4.5 decision tree during every run and labels the most appropriate leaf node as a rule.
FURIA	Fuzzy Unordered Rule Induction Algorithm [203]	
MOD	Modlem [204]	
NNGE	Non-Nested Generalised Exemplars [47]	
PART	PARTial decision lists [57]	
Decision Trees		
HFT	Hoeffding Trees [197]	The *HFT* makes use of a pre-pruning policy based on the Hoeffding bound to incrementally grow a decision tree. The *J48* technique creates a decision tree by iterative data splitting and Depth-first strategy is employed for decision growth. *RF* is an assemblage of unpruned classification or regression trees, produced from bootstrap training dataset samples, by means of random feature selection during the tree generation procedure. The predictions of the ensemble are aggregated via a voting scheme to deliver a conclusive prediction. The *CART* employs Gini Index to form subsections of the dataset via all independent variables and creates two child nodes repetitively. The ultimate objective is to construct dataset subsets that are of maximum
J48	J48 [9, 53, 54, 56]	
RF	Random Forest [6, 30, 57, 62]	
CART	Classification & Regression Trees [197]	
SYSFOR	SysFor [205]	

Table 4.5 Prediction techniques incorporated in this analysis

Prediction Technique		Description
		homogeneity with the predictor variable. *SYSFOR* is a gain ratio-based multiple tree building algorithm. Multiple trees are built with the purpose of gaining enhanced knowledge via the extraction of multiple patterns which continues up until the user stated numbers of trees are constructed.
Evolutionary approach		
GANN	Genetic Algorithm with Neural Network [61]	*GANN* is the modified hybridized genetic algorithm (GA) and artificial neural network (ANN) method which work in unison to discover the most optimal feature set that is capable of efficiently classifying the data. The GA locates the ANN with the most optimum feature set that will accurately segregate the classes, whereas the fitness values of GA are optimized by ANN. *NNEP* is a unique variant of feed-forward neural network, known as product-unit neural networks in which the model insinuated changes the network‟s weights and the structure with the aid of an evolutionary algorithm. This change in the network structure partly eases the difficulty involved in not knowing the best network structure for a given problem in advance. NNEP and GANN were run using KEEL and for both of them the maximum number of generations was brought down to 10. 100 backpropagation cycles were employed to generate the models corresponding to GANN. Both GANN and NNEP were run for 30 iterations to produce the results for the purpose of obtaining rigorous statistical evidence with respect to their performance. *MOEFC* constructs a fuzzy rule based classifier by using the ENORA or NSGA-II Multi-objective Evolutionary Algorithm. We employed ENORA in our work as it is configured to maximize accuracy, to maximize area under ROC curve, and to minimize root mean squared error.
NNEP	Neural Network Evolutionary Programming [61]	
MOEFC	Multi Objective Evolutionary Fuzzy Classifier [206]	

4.3.5 Performance evaluation metrics

A binary classification technique is capable of making two possible errors: false negatives (FN) and false positives (FP). True negative (TN) stands for a correctly classified stable file and a true positive (TP) stands for a correctly classified change-prone file. Sensitivity or Recall indicates the proportion of “changed Java files” of a version that are classified correctly as “change-prone” and as is calculated as (TP/(TP+FN)). Specificity, on the other hand, is the proportion of “unchanged Java files” of a version that are classified correctly as “stable” or “not change-prone” and is estimated as (TN/(TN+FP)). Precision presents the positive predictive value or the fraction of the classified change-prone files that are change-prone and is calculated as

(TP/(TP+FP)). Apart from including Accuracy and AUC (Area Under the ROC) as performance measures (as employed[5] in Chapter 3), we evaluated binary classification results from the standpoint of, F-Measure or F-Score (*F-S*), G-mean1 (*G-m1*), G-mean2 (*G-m2*), and Matthews correlation coefficient (*MCCF*), which are described as follows:

4.3.5.1 F-measure

The F-measure [207], or F-score, is the harmonic mean, or the weighted average, of precision and recall, where the score value between 0 and 100 indicate worst and best match performances, respectively. Put another way, the F- score conveys the balance between the precision and the recall.

$$\text{Fmeasure} = 2 * \frac{(precision * recall)}{(precision + recall)}$$

4.3.5.2 Geometric mean 1 (g-mean1) and Geometric mean 2 (g-mean2)

G-mean1 [207] is a single-measure statistic where the resulting value between 0 and 100 indicates poor to perfect match performance of the prediction model. It is calculated as:

$$\text{G} - \text{mean1} = \sqrt{Sensitivity * Precision}$$

G-mean2 on the other hand is calculated as:

$$\text{G} - \text{mean2} = \sqrt{Sensitivity * Specificity}$$

In software change prediction, it is important to identify more change-prone modules which mean a high probability of detection is needed. So if two classifiers predict with a same or similar sensitivity, the one with higher specificity would be preferred

4.3.5.3 Matthews correlation coefficient

The Matthews correlation coefficient (MCCF) [208], also referred to as the phi coefficient (φ) or mean square contingency coefficient, takes into account any classification size differences and is particularly useful when the two classes are of very different sizes. The calculated coefficient value is between -1 and 1, with 0 as a random match performance, 1 as a perfect match performance, and -1 as a perfect inverse match performance. It is calculated as:

[5] The description of Accuracy and AUC measures has been provided in Chapter 3, Section 3.4.2.

$$\text{MCC} = \frac{TP * TN - FP * FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

Let's consider a case where there exists an extreme imbalance in the dataset wherein the number of cases for either the positive or negative class are too low. In such a situation the classifier might evaluate the value of TP or TN to be 0. Upon averaging the True Positive Rate and True Negative Rate, a score without any direction will be returned. On the contrary, since MCCF involves values of all the four quadrants of a confusion matrix, it is therefore a balanced measure and returns a value with a direction (+ve and -ve).

The performance measures MCCF and AUC are robust to data imbalance.

4.3.6 Validation Methodologies

It is imperative to verify the predictive models generated on data sets dissimilar from the datasets that were used for their training to obtain generalizable and unbiased results.

Two validation approaches: k-fold cross-validation and inter-release validation are employed in our study in order to acquire an additional pragmatic and conclusive estimation regarding the efficacy of the selected techniques with respect to version to version change-proneness prediction of Java files. As mentioned in the literature studied [12, 23, 52, 54] in Section 2.2.3 of Chapter 2, in a k-fold cross validation (VS1 in Figure 4.1), the dataset is arbitrarily segregated into approximately ‘k’ equal subsets. For every evaluation, the test set is created from either of the k subsets and the training set is created using the rest of the ‘k-1’ subsets. This procedure is performed for all the ‘k’ subsets. In other words, the predictive model is created and assessed ‘k’ times in which every time a dissimilar subsample is employed for the assessment of the performance measures of the model and the rest of the subsamples are employed to create the training data to develop the prediction model. In our work, the model generated results are validated with the value of ‘k’ equal to 10.

On the contrary, in an inter-release validation [13, 48] (VS2 in Figure 4.1), the model training is performed via a certain release and this trained model is validated on its subsequently released version. Therefore as per our context of application, we employ every selected version of the two target projects, construct models with respect to

each of the selected 31 techniques and validate these models on their subsequent releases. E.g., JFreeChart version 0.6.0 is employed to train a model by means of a prediction technique and then the developed model is further validated on JFreeChart version 0.7.0. This validation methodology is primarily useful to predict the change-prone files of a specific release of a software whose successive version is yet to be released. A predictive analysis like so would aid in providing insights to the software developers apropos to the trend of change-proneness of files over upcoming releases of a software project.

4.3.7 Statistical evaluations

We employ various statistical tests and analyses to evaluate the findings of the results obtained in this chapter and to provide statistical substantiation to the answers corresponding to the RQs stated in Section 4.1 of this chapter.

4.3.7.1 Kruskal-Wallis test

The *Kruskal-Wallis test* [27] is used to answer RQ3 to test if the selected prediction techniques perform with a significant difference when they are subjected to different validation settings over the various JFreeChart and Heritrix releases using the AUC and MCCF performance measure values. The Kruskal-Wallis test relies on the rank-ordering of data and allows you to test whether the mean ranks are the same in all the groups of three or more independent samples. If each group consists of 4 or more observations, the Kruskal-Wallis test approaches a chi-square distribution. The null hypothes is indicating that at least one of the samples belong to a dissimilar population is accepted if the estimated value of the Kruskal-Wallis test is less in comparison to the chi-square's critical value. If the estimated value of Kruskal-Wallis test is higher in comparison to the critical chi-square value, then the alternative hypothesis is agreed upon and the null hypothesis is overruled.

4.3.7.2 Scott-Knott cluster analysis

We employ the *Scott-Knott cluster analysis* [6, 56, 209] in RQ4 to statistically compare all the 31 prediction techniques over a specific project according to their AUC values and then cluster them into homogenous subgroups, where each subgroup contains the techniques that are significantly indifferent. The Scott-Knott is a multiple comparison statistical procedure based on the notion of clustering where the criterion

of clustering is the significance test between the prediction techniques' AUC value. We apply the Scott-Knott analysis using the 95% confidence level (i.e., $\alpha = 0.05$). This analysis can overcome the issue of overlapping multiple comparisons that are obtained from other tests, such as the Mann-Whitney U test [31, 52, 93]. It recursively ranks the evaluated classifiers through hierarchical clustering analysis. In each iteration, the Scott-Knott test separates the evaluated classifiers into two groups based on the performance measure (i.e., the AUC value). If the two groups have statistically significant difference in the AUC value, the Scott-Knott test executes again within each group. If no statistically distinct groups can be created, the Scott-Knott test terminates [6, 56, 209].

Since the Scott-Knott presumes provisionally that the performance measure values should be normally distributed, we made sure that the AUC values obtained with respect to each prediction technique over each of the software projects follow normal distribution. Tests like Kolmogorov-Smirnov test and Shapiro-Wilk are designed to test normality by comparing your data to a normal distribution with the same mean and standard deviation of your sample. If the test is not significant, then the data are normal, so any value above .05 indicates normality. However, the power of both the tests is still low for small sample size.

Since the sample size of our data to perform a Scott-Knott cluster analysis is low, we therefore opt for another rapid way to check if a distribution is normal. We compare the mean and median of AUC values obtained per prediction technique per project. In a normal distribution mean is equal to median and thusly the ratio mean/median should be 1. We took a confidence interval of 95% and therefore ascertained a normal distribution of data if the mean to median ratio for every such selected prediction technique lied between 0.95-1.05.

4.3.7.3 Ranked Voting (RV) using Borda counting

As explained in previous section, Scott-Knott allows us to identify the prediction techniques that have highest AUC for each of the validation scenarios for both the plugin project versions. However, when it comes to identifying the best techniques with respect to software change-proneness prediction, we should assess the models using not only AUC, but taking other performance measures in consideration.

Therefore, *Ranked Voting (RV)* [210, 211] method is employed in this chapter post the application of the Scott Knott's test in RQ4 to aggregate ranks across multiple performance measures. Methods like win-tie-loss cannot work well because their ranking mechanism depends on the significance test between different methods. The RV method has never been used before to rank software change prediction models, which is considered part of novelty in this research. RV is a measure of individual interests and preferences as an aggregate towards collective decision.

For example, suppose we have 5 prediction techniques (P1, P2, P3, P4, P5) and 4 performance measure (pm1, pm2, pm3, pm4). Every experimental condition ranks candidates in a definitive order as follows:

pm1: P2≻P1≻P4≻P3≻P5;

pm2:P1≻P4≻P2≻P3≻P5;

pm3: P2≻P4≻P1≻P5≻P3; and

pm4: P1≻P4≻P5≻P2≻P3.

Among many RV methods, *Borda counting* [212] has been employed in this chapter. In order to calculate the collective decision using Borda, we first construct a majority margins matrix (MM) as shown in the illustrative example in Table 4.6. Each entry in MM represents how many times a prediction technique 'x' preceeds another prediction technique 'y' across all performance measures. This can be accomplished by subtracting the times that x beats y (|x≻y|) from the times that y beats x (|y≻x|). For example, the first row and third column tell us that $MM_{P1,P3}$ = |P1≻P3| -|P3≻P1| = 4-0=4, which indicates that the prediction technique P1 beats the technique P3 by a margin of four.

Post the calculation of the summation of votes for every such technique over each performance, the techniques are then ranked in a descending order based on the final summation where the prediction technique with the largest score is the best performing technique.

Table 4.6 Majority margin matrix where rows and columns headers represent the prediction techniques

	P1	P2	P3	P4	P5	Score
P1		0	4	2	4	**10**
P2	0		4	0	2	**6**
P3	-4	-4		-4	0	**-12**
P4	-2	0	4		4	**6**
P5	-4	-2	0	-4		**-10**

Each entry represents precedence degree between two techniques across all performance measures. The last column represents the overall score that techniques are sorted upon. The resulted aggregated scores for every technique are represented in bold in the last column of Table 4.6. Therefore, for the above profile we get the following ranking: P1≻(P2~P4)≻P5≻P3 where techniques on the left hand side are ranked higher and ~ symbol means indifference between two techniques (i.e. they have same rank). The final ranking suggests that out of the five prediction techniques, P1 the best performing technique according to the selected performance measures.

4.4 Empirical results and analysis

The following sub-sections exhibit the results of the selected prediction techniques for version to version change-proneness prediction of Java files using the seventeen OO metrics. The empirical results gathered with respect to the prediction techniques are provided as responses to the various RQs specified in Section 4.2. The default settings of Weka 3.8.2 [185] and KEEL [213] have been utilized to generate all the results, unless specified otherwise in the description given in Table 4.5 of Section 4.3.4. Post the rectification of the class imbalance problem in our JFreeChart and Heritrix datasets using SMOTE, we conducted another vital data pre-processing activity of feature selection to reduce the number of OO metrics using the CBFS technique (as mentioned in Section 4.3.3). Table 4.7 indicates the relevant OO metrics that were selected after applying CBFS with the best first search methodology over all the selected JFreeChart and Heritrix versions.

Table 4.7 CBFS results

Datasets	Metrics selected after CBFS
JFreeChart 0.6.0	SLOC, CHV, AC, DIT,NOC,CBO,RFC
JFreeChart 0.70	EC, Instability, CBO, CogC
JFreeChart 0.7.1	CHL,CHV, MI, AC, Instability, CBO,RFC,LCOM
JFreeChart 0.7.2	CHL,CHV, CHB, MI, EC,Instability, DIT,NOC,CBO, LCOM, CogC
JFreeChart 0.7.3	EC, DIT, ,CBO,RFC, CogC
Heritrix 0.2.0	SLOC, CHL, CHB, EC, WMC, CBO, CogC
Heritrix 0.4.0	CHV, EC, DIT, CBO,RFC
Heritrix 0.6.0	CHE, EC, WMC,DIT,CBO, CogC
Heritrix 0.8.0	SLOC, CHE, EC, CBO,RFC, CogC
Heritrix 0.10.0	SLOC, CHE, EC, WMC,CBO

As observed from Table 4.7, the CBO metric is found to be selected in all the ten version datasets of JFreeChart and Heritrix software, signifying it to be the most important feature among the seventeen selected features with respect to change-proneness of Java files. Apart from the CBO metric, the EC metric is also observed to be selected in all the five releases of the Heritrix software. Other than the CBO and EC metric, overall, it can be seen that CogC, RFC, DIT, CHV and SLOC metrics are commonly found to be relevant to change-proneness of files in most of the selected releases of both the selected target projects. These results are consistent with the results of feature selection performed in some of the previous analyses [11, 24, 25, 47, 55, 100, 101] with respect to change-proneness prediction.

The predictive effectiveness of prediction techniques is analyzed using: (a) CBFS+10-fold validation (VS1), and (b) inter-release validation (VS2) for the evaluation of the models ability in the prediction of the change-proneness of Java files. The fidelity of the prediction results is assessed via statistical tests as elucidated in Section 4.3.7. AUC is preferred to other measures for summarizing the performance of the techniques in RQ1 and RQ2, since AUC is not biased towards the size of evaluation or test data. As noted in Table 4.4, an imbalance between the unchanged and the changed files for majority of the JFreeChart and Heritrix releases still exists because we eliminate the outliers after the application of SMOTE to avoid errors in classification. AUC has been statistically confirmed to be more discriminatory in contrast to other performance measures, specifically for imbalanced datasets [188] and is therefore selected as the deciding metric for the performance comparison of the prediction techniques. Although MCCF as a performance measure is also robust to the imbalance in dataset, we employ it along with other performance measures to answer RQ3 where we evaluate statistically significant difference between the validation scenarios and in RQ4 wherein the best technique with respect to the two validation scenarios is evaluated.

4.4.1 RQ1

What is the predictive capability of the various selected techniques, by and large, with respect to predicting version to version change-proneness of Java files when 'k'-fold cross-validation with feature selection is employed?

After the application of the CBFS method, the results acquired from the models developed by means of the 31 prediction techniques on the datasets gathered from five JFreeChart and five Heritrix plugin project versions have been stated in Tables 4.8 and 4.9. A 10-fold cross-validation technique has been utilized to assess the results, after the application of CBFS (VS1 as explained in Figure 4.1). The columns in Tables 4.8 and 4.9 specify the Accuracy (*Acc.*), Area under Curve (*AUC*), F-Measure or F-Score (*F-S*), G-mean1 (*G-m1*), G-mean2 (*G-m2*), and Matthews correlation coefficient (*MCCF*) attained by every selected prediction technique on each of the selected version datasets. (Highest performance measure value for every prediction technique apropos to each version has been showed in bold)

As studied from Tables 4.8 and 4.9, the RF technique shows highest values with respect to all the six performance measures out of the 31 techniques, consistent for two out of five JFreeChart versions and three out of five Heritrix versions. Apart from this, even though the evolutionary technique of MOEFC obtains acceptable values of AUC for six out of ten (five JFreeChart and one Heritrix) version datasets, very high Root Mean Squared Error values (>0.7) (though not included in the table) are observed for the same for all the ten datasets.

To abridge,

From total of 310 models developed in Table 4.8 and 4.9 (31 models constructed for each of the five JFreeChart versions and five Heritrix versions), insofar as the AUC values are involved,

- The prediction techniques exhibit 31 outstanding divisions (i.e. AUC >= 0.9) in totality between the change-prone and unchanged files for the five JFreeChart datasets, highest being 21 outstanding divisions for the JFreeChart version 0.7.2. On the contrary, not one of the prediction techniques display outstanding classifications apropos to the five Heritrix version datasets.
- The prediction techniques, in general, perform better for the JFreeChart datasets than the Heritrix datasets, with the minimum number of excellent

divisions (i.e. AUC >= 0.8 and <0.9) being 10 for the JFreeChart version 0.7.2 (the highest number of excellent divisions being 24 for JFreeChart version 0.7.3). On the other hand the highest number of excellent divisions, with respect to the Heritrix datasets, is observed for 8 techniques. Also, none of the selected techniques show excellent divisions for Heritrix 0.8.0.

- Overall, a total of 43 unacceptable divisions (i.e. AUC < 0.7) are observed out of the 155 models generated by the 31 prediction techniques for the five Heritrix version datasets. Whereas, only 4 unacceptable divisions are observed for the five JFreeChart datasets.

Overall, taking an average of the average AUC values obtained across the versions of the two target projects, it can be observed that:

- Among the 31 selected prediction techniques, the results of CBFS + 10-fold validation indicate that there is no one technique that consistently proves to be the best or worst for change-proneness prediction of Java files for all the versions of JFreeChart and Heritrix projects with respect to any of the performance measures used.

- BN, ADB, BAG, LB, RSS, GANN, RF, and SYSFOR techniques exhibit the highest prediction performances in terms of AUC, exhibiting an excellent division (i.e. AUC >= 0.8 and <0.9) between the change-prone and stable Java files for the ten datasets.

- Although techniques like SMO, SPEG, FLR, VFI, MOD and MOEFC exhibit acceptable AUC values over the JFreeChart versions, they underperform for most of the Heritrix releases.

Table 4.8 Prediction results of the 31 prediction techniques over the five versions of JFreeChart using CBFS + 10 fold validation.

	JFreeChart 0.6.0						JFreeChart 0.7.0						JFreeChart 0.7.1					
Tech.	Acc.	AUC	F-S	G-m1	G-m2	MCCF	Acc.	AUC	F-S	G-m1	G-m2	MCCF	Acc.	AUC	F-S	G-m1	G-m2	MCCF
BN	81.9	0.915	78.8	78.8	81.4	0.630	78.9	0.833	70.4	70.8	74.4	0.522	75.9	0.806	73.2	73.3	75.8	0.514
NB	84.4	0.912	82.0	82.0	84.2	0.684	76.7	0.813	71.4	71.9	75.2	0.546	73.4	0.808	65.6	66.2	70.3	0.445
FLDA	82.8	0.901	80.0	80.0	82.4	0.649	72.2	0.825	67.5	67.5	71.4	0.433	70.3	0.771	66.7	66.7	70.5	0.420
LR	85.3	0.891	83.2	83.2	85.2	0.702	73.3	0.803	68.5	68.6	72.7	0.475	72.8	0.800	66.7	68.3	72.4	0.444
MLP	82.8	0.900	80.0	80.0	82.4	0.649	82.2	0.768	75.0	76.8	77.6	**0.659**	75.9	0.817	71.8	71.9	75.1	0.523
RBFN	81.9	0.884	78.8	78.8	81.4	0.630	78.9	0.806	69.7	70.9	73.8	0.553	74.7	0.767	69.2	69.4	73.1	0.484
SMO	82.8	0.819	79.2	79.2	81.7	0.647	75.6	0.745	68.4	68.5	72.3	0.454	67.1	0.645	50.0	52.4	58.1	0.303
SPEG	81.9	0.814	78.8	78.8	81.4	0.630	74.4	0.729	69.3	69.4	73.2	0.476	73.4	0.719	66.1	67.3	70.7	0.491
ADB	83.6	0.884	81.2	81.2	83.4	0.667	76.7	0.814	71.2	71.4	75.0	0.521	76.6	0.776	71.9	71.9	74.8	0.499
BAG	86.2	0.887	84.9	85.0	86.6	0.727	81.1	0.832	78.9	79.3	81.4	0.662	74.7	0.799	73.0	73.0	75.9	0.524
DAG	81.9	0.898	81.2	81.2	83.4	0.667	75.6	0.798	74.6	75.8	77.7	0.623	75.3	0.791	71.4	71.4	74.3	0.487
FC	82.8	0.859	82.1	82.6	**88.4**	0.673	74.4	0.743	68.6	69.0	72.9	0.499	72.3	0.756	69.0	69.0	71.9	0.438
LB	86.2	**0.924**	84.6	84.7	86.4	0.723	77.8	0.865	74.7	74.7	77.8	0.568	77.2	0.824	73.0	73.0	75.9	0.524
AM	85.3	0.891	83.2	83.2	85.2	0.702	73.3	0.803	68.5	68.6	72.7	0.475	72.8	0.800	66.7	66.9	70.9	0.444
RSS	83.6	0.888	81.9	82.0	83.9	0.673	76.7	0.813	76.7	76.9	79.6	0.614	77.2	0.840	75.8	75.9	78.5	0.588
CHIRP	80.2	0.797	76.8	76.8	79.6	0.595	73.3	0.719	64.8	65.1	69.7	0.428	72.8	0.724	71.5	71.5	74.6	0.498
FLR	66.4	0.685	68.3	69.5	66.7	0.380	62.2	0.621	59.3	59.3	63.1	0.261	62.7	0.618	60.0	56.0	60.9	0.266
VFI	80.2	0.879	78.1	78.2	80.4	0.603	73.3	0.754	67.6	68.4	72.1	0.501	60.1	0.773	67.4	70.1	56.3	0.322
DTNB	77.6	0.872	70.5	70.5	74.9	0.516	**84.4**	0.805	**78.5**	78.5	80.9	0.617	77.8	0.798	75.9	75.9	77.9	0.556
FURIA	82.8	0.863	81.1	81.3	83.1	0.657	80.0	0.814	75.0	75.2	78.1	0.591	72.2	0.767	67.2	67.2	70.7	0.421
MOD	**86.2**	0.857	83.7	83.3	85.6	0.718	77.8	0.771	78.9	**79.0**	**81.5**	0.363	82.4	0.818	**78.5**	**78.6**	**80.9**	**0.627**
MOEFC	81.9	0.863	78.4	78.4	81.0	0.629	80.0	0.711	65.7	66.6	70.5	0.578	71.5	0.776	72.3	72.5	75.6	0.455
NNGE	83.6	0.837	81.6	81.6	83.7	0.669	76.7	0.758	76.3	76.4	79.2	0.591	77.2	0.771	72.9	72.9	75.6	0.513
PART	82.8	0.857	82.1	82.6	83.5	0.694	75.6	0.769	69.4	69.7	73.5	0.498	77.9	0.740	69.6	69.7	73.0	0.471
NNEP	81.9	0.889	79.6	79.6	81.9	0.634	82.2	0.843	75.3	75.3	78.3	0.569	74.7	0.789	69.2	69.4	73.1	0.484
HFT	84.5	0.910	82.0	82.0	84.2	0.684	77.8	0.814	72.5	73.1	76.0	0.571	73.4	0.808	65.6	66.1	70.2	0.445
J48	82.7	0.826	82.1	82.6	83.5	0.673	78.9	0.753	72.5	73.1	76.0	0.571	72.2	0.733	72.1	72.2	74.2	0.481
GANN	85.3	0.889	82.5	82.5	84.6	0.700	78.9	0.773	72.5	73.1	76.0	0.571	75.3	0.816	75.2	75.2	77.6	0.552
RF	**86.2**	0.911	84.6	84.7	86.4	0.723	78.9	**0.872**	75.3	75.3	78.3	0.569	**84.2**	**0.873**	**78.5**	**78.6**	**80.9**	**0.627**
CART	83.6	0.835	82.6	82.9	84.2	0.681	78.9	0.795	72.0	72.0	75.5	0.522	77.2	0.744	71.5	71.5	74.6	0.498
SYSFOR	**86.2**	0.895	**85.7**	**86.2**	87.0	**0.743**	75.6	0.840	70.3	70.4	74.1	0.498	71.5	0.826	71.1	71.3	73.1	0.459

Table 4.8 (contd.) Prediction results of the 31 prediction techniques over the five versions of JFreeChart using CBFS + 10 fold validation.

Tech.	JFreeChart 0.7.2						JFreeChart 0.7.3						Average values over the JFreeChart versions					
	Acc.	AUC	F-S	G-m1	G-m2	MCCF	Acc.	AUC	F-S	G-m1	G-m2	MCCF	Acc.	AUC	F-S	G-m1	G-m2	MCCF
BN	89.7	0.959	90.0	90.3	89.8	0.807	84.6	0.881	80.0	80.5	82.2	0.690	82.2	0.879	78.5	78.7	80.7	0.633
NB	78.5	0.863	76.1	76.2	78.0	0.568	81.4	0.884	76.0	76.5	79.1	0.621	78.9	0.856	74.2	74.6	77.4	0.573
FLDA	83.6	0.903	83.2	83.2	83.7	0.673	82.8	0.877	76.5	77.6	79.2	0.656	78.3	0.855	74.8	75.0	77.4	0.566
LR	82.6	0.893	82.5	82.6	82.7	0.656	82.1	0.872	76.7	77.2	79.5	0.635	79.2	0.852	75.5	76.0	78.5	0.582
MLP	90.8	0.948	90.5	90.6	90.9	0.817	80.8	0.844	75.4	75.8	78.6	0.607	82.5	0.855	78.5	79.0	80.9	0.651
RBFN	82.1	0.881	81.7	81.7	82.2	0.634	80.8	0.846	75.0	75.5	78.2	0.608	79.7	0.837	74.9	75.3	77.7	0.582
SMO	81.5	0.817	81.1	81.1	81.6	0.632	82.7	0.804	76.1	77.5	78.8	0.660	77.9	0.766	71.0	71.7	74.5	0.539
SPEG	83.1	0.836	83.7	84.1	83.1	0.677	81.4	0.795	75.2	76.0	78.3	0.625	78.8	0.779	74.6	75.1	77.3	0.580
ADB	92.1	0.983	92.6	92.6	92.9	0.857	81.4	0.888	77.5	77.6	80.3	0.619	82.1	0.869	78.9	78.9	81.3	0.633
BAG	91.3	0.974	90.4	90.7	91.5	0.822	81.4	0.912	76.8	77.0	79.7	0.619	82.9	0.881	80.8	81.0	83.0	0.671
DAG	86.2	0.923	87.0	87.5	86.0	0.765	83.3	0.891	79.0	79.3	81.6	0.659	80.5	0.860	78.6	79.0	80.6	0.640
FC	90.3	0.939	90.4	90.5	90.4	0.813	80.1	0.792	76.3	76.3	79.3	0.593	80.0	0.818	77.3	77.5	80.6	0.603
LB	**95.4**	**0.992**	**95.2**	**95.2**	**95.4**	**0.908**	81.4	0.910	77.5	77.6	80.3	0.619	83.6	0.903	81.0	81.0	83.2	0.668
AM	82.6	0.893	82.5	82.6	82.7	0.656	82.1	0.872	76.7	77.2	79.5	0.635	79.2	0.852	75.5	75.7	78.2	0.582
RSS	92.3	0.982	92.2	92.3	92.4	0.850	83.3	0.898	79.0	79.3	81.6	0.659	82.6	0.884	81.1	81.3	83.2	0.677
CHIRP	89.7	0.901	89.9	90.1	89.9	0.804	83.3	0.821	79.0	79.3	81.6	0.659	79.9	0.792	76.4	76.6	79.1	0.597
FLR	83.1	0.825	80.2	80.6	81.9	0.668	71.2	0.720	69.8	70.1	71.7	0.435	69.1	0.694	67.5	67.1	68.9	0.402
VFI	88.2	0.953	88.4	88.7	88.4	0.775	80.8	0.837	73.2	74.7	76.4	0.620	76.5	0.839	74.9	76.0	74.7	0.564
DTNB	91.9	0.956	91.3	91.3	91.7	0.835	80.8	0.859	74.6	75.3	77.8	0.610	82.5	0.858	78.2	78.3	80.6	0.627
FURIA	91.8	0.927	91.6	91.6	91.9	0.838	80.8	0.800	75.8	76.0	78.9	0.606	81.5	0.834	78.1	78.3	80.5	0.623
MOD	92.8	0.929	92.5	92.5	92.8	0.856	83.9	0.828	80.0	80.2	82.4	0.672	84.6	0.841	**82.7**	**82.7**	**84.6**	0.647
MOEFC	84.6	0.899	83.7	83.7	84.5	0.691	82.1	0.861	77.1	77.4	79.9	0.633	80.0	0.822	75.4	75.7	78.3	0.597
NNGE	84.6	0.847	84.0	84.1	84.7	0.693	80.1	0.806	78.3	78.5	80.5	0.605	80.4	0.804	78.6	78.7	80.7	0.614
PART	93.3	0.951	93.0	93.0	93.3	0.866	80.8	0.853	78.9	79.0	81.1	0.617	82.1	0.834	78.6	78.8	80.9	0.629
NNEP	92.8	0.982	92.6	92.6	92.9	0.857	80.8	0.886	77.6	77.6	80.3	0.608	82.5	0.878	78.9	78.9	81.3	0.630
HFT	78.9	0.863	76.8	76.9	78.6	0.578	81.4	0.883	76.0	76.5	79.1	0.603	79.2	0.856	74.6	74.9	77.6	0.576
J48	91.8	0.930	91.5	91.5	91.9	0.836	79.5	0.784	75.8	75.8	78.7	0.580	81.0	0.805	78.8	79.0	80.9	0.628
GANN	90.8	0.963	90.4	90.4	90.8	0.816	79.5	0.870	75.0	75.1	78.2	0.579	82.0	0.862	79.1	79.3	81.4	0.644
RF	92.3	0.990	92.1	92.2	92.4	0.848	**85.3**	**0.926**	**82.4**	**82.5**	**84.6**	**0.698**	**85.4**	**0.914**	82.6	**82.7**	84.5	**0.693**
CART	91.3	0.927	91.2	91.3	91.5	0.830	80.8	0.833	76.6	76.6	79.5	0.605	82.4	0.827	78.8	78.9	81.1	0.627
SYSFOR	91.3	0.984	91.1	91.2	91.4	0.828	83.3	0.89	79.4	79.5	81.9	0.659	81.6	0.887	79.5	79.7	81.5	0.637

Table 4.9 Prediction results of the 31 prediction techniques over the five versions of Heritrix using CBFS + 10 fold validation.

	Heritrix 0.2.0						Heritrix 0.4.0						Heritrix 0.6.0					
Tech.	Acc.	AUC	F-S	G-m1	G-m2	MCCF	Acc.	AUC	F-S	G-m1	G-m2	MCCF	Acc.	AUC	F-S	G-m1	G-m2	MCCF
BN	73.9	0.799	72.5	72.4	73.9	0.478	64.7	0.731	63.8	63.8	64.7	0.294	70.7	0.766	63.2	63.2	69.2	0.390
NB	72.6	0.805	68.8	68.9	71.7	0.448	74.8	0.796	71.7	72.0	74.0	0.501	70.9	0.701	58.6	59.5	65.7	0.397
FLDA	72.6	0.764	68.8	68.9	71.7	0.448	64.7	0.763	61.1	61.3	64.1	0.294	69.6	0.75	59.4	59.5	66.3	0.354
LR	68.5	0.795	65.7	65.7	68.2	0.366	68.1	0.764	66.1	66.1	67.8	0.361	70.8	0.742	57.6	58.3	65.0	0.371
MLP	68.5	0.727	62.3	62.7	66.7	0.365	74.8	0.790	72.2	72.4	74.2	0.498	66.7	0.715	55.1	55.3	62.8	0.290
RBFN	72.6	0.737	70.6	70.6	72.5	0.449	77.3	0.856	75.7	75.7	77.1	0.547	70.2	0.691	59.2	59.4	66.2	0.363
SMO	73.9	0.736	70.8	70.8	73.3	0.476	71.4	0.711	66.7	67.3	70.0	0.437	71.4	0.677	58.1	58.9	65.3	0.384
SPEG	71.2	0.710	68.7	68.6	70.9	0.421	62.2	0.621	60.2	60.2	62.0	0.243	70.8	0.680	59.7	60.0	66.6	0.375
ADB	75.3	0.816	73.5	73.5	75.2	0.505	68.1	0.752	66.7	66.7	68.0	0.361	74.9	0.758	64.5	65.0	70.4	0.463
BAG	72.6	0.794	70.6	70.6	72.5	0.449	73.1	0.800	70.9	71.0	72.7	0.463	77.2	0.797	67.2	67.9	72.5	0.515
DAG	**80.8**	0.812	**79.4**	**79.4**	**80.7**	**0.615**	71.4	0.746	67.3	67.7	70.3	0.434	72.5	0.743	61.2	61.6	67.8	0.411
FC	76.7	0.721	75.4	75.4	76.7	0.533	61.3	0.645	58.2	58.3	60.9	0.226	72.5	0.733	60.5	61.1	67.2	0.411
LB	69.9	0.804	66.7	66.7	69.4	0.393	65.6	0.738	63.7	63.8	65.4	0.310	77.8	0.786	70.3	70.5	75.2	0.529
AM	68.5	0.795	65.7	65.7	68.2	0.366	68.2	0.764	66.1	66.1	67.8	0.361	70.8	0.742	57.6	58.3	65.0	0.371
RSS	72.6	0.759	71.4	71.4	72.6	0.452	68.9	0.772	66.1	66.2	68.4	0.379	78.4	0.826	67.8	69.0	72.7	0.544
CHIRP	75.3	0.748	70.8	70.8	73.3	0.476	71.4	0.712	67.3	67.7	70.3	0.434	69.6	0.653	53.6	54.8	61.7	0.342
FLR	65.8	0.655	62.7	62.7	65.4	0.311	66.4	0.662	63.6	63.7	65.9	0.327	63.2	0.614	53.3	53.3	60.8	0.229
VFI	69.9	0.679	59.3	61.4	65.0	0.412	70.6	0.732	59.8	63.4	65.3	0.465	53.2	0.696	61.9	66.2	49.1	0.272
DTNB	76.7	0.763	73.8	73.9	76.1	0.531	69.8	0.717	65.4	65.8	68.6	0.400	73.1	0.741	58.9	60.3	65.8	0.424
FURIA	75.3	0.778	73.5	73.5	75.2	0.505	70.6	0.716	66.7	67.0	69.6	0.416	74.3	0.744	63.3	63.9	69.5	0.450
MOD	68.5	0.684	66.7	66.6	68.4	0.368	70.6	0.705	69.0	69.0	70.4	0.411	79.5	0.768	71.1	71.6	75.5	0.566
MOEFC	72.6	0.782	67.7	68.1	71.2	0.449	66.4	0.660	63.0	63.1	65.8	0.328	71.4	0.671	58.1	58.9	65.3	0.384
NNGE	79.5	0.794	78.3	78.2	79.4	0.588	72.3	0.723	71.8	71.8	72.2	0.445	71.4	0.692	62.0	62.1	68.4	0.393
PART	71.2	0.693	66.7	66.9	70.1	0.421	74.8	0.807	73.7	73.7	74.7	0.495	73.1	0.721	56.6	59.0	63.8	0.428
NNEP	76.7	0.808	73.8	73.9	76.1	0.531	71.4	0.802	70.2	70.2	71.4	0.428	77.8	0.787	67.2	68.3	72.4	0.530
HFT	73.9	0.804	69.8	70.1	72.9	0.477	73.9	0.795	71.0	71.3	73.3	0.482	72.5	0.700	59.1	60.1	66.1	0.410
J48	69.9	0.642	64.5	64.8	68.4	0.393	73.9	0.819	73.5	73.5	73.9	0.479	73.1	0.726	59.6	60.8	66.4	0.423
GANN	72.6	0.829	69.7	69.7	72.1	0.448	73.1	0.809	69.8	70.1	72.3	0.466	71.4	0.748	58.1	58.9	65.3	0.384
RF	75.3	0.791	72.7	72.8	74.9	0.503	**78.9**	**0.831**	**77.9**	**77.9**	**78.9**	**0.580**	**81.9**	**0.856**	**74.8**	**75.2**	**78.6**	**0.617**
CART	72.6	0.722	71.4	71.4	72.6	0.452	68.1	0.710	66.1	66.1	67.8	0.361	70.2	0.712	57.9	58.3	65.2	0.360
SYSFOR	75.3	**0.850**	73.5	73.5	75.2	0.505	69.8	0.796	67.3	67.4	69.4	0.395	75.4	0.803	61.1	63.3	67.2	0.482

Table 4.9 (contd.) Prediction results of the 31 prediction techniques over the five versions of Heritrix using CBFS + 10 fold validation.

	Heritrix 0.8.0						Heritrix 0.10.0						Average values over the Heritrix versions					
Tech.	Acc.	AUC	F-S	G-m1	G-m2	MCCF	Acc.	AUC	F-S	G-m1	G-m2	MCCF	Acc.	AUC	F-S	G-m1	G-m2	MCCF
BN	69.3	0.731	67.5	67.5	69.3	0.385	70.1	0.774	65.2	65.2	71.2	0.428	69.7	0.760	66.4	66.4	69.7	0.395
NB	67.5	0.709	57.1	58.6	63.4	0.345	64.9	0.684	45.9	47.1	56.0	0.233	70.1	0.739	60.4	61.2	66.2	0.385
FLDA	60.8	0.674	54.5	54.6	59.5	0.205	69.3	0.730	62.6	62.6	69.0	0.384	67.4	0.736	61.3	61.4	66.1	0.337
LR	63.9	0.676	55.2	55.7	61.1	0.265	68.4	0.727	52.3	53.5	61.0	0.326	67.9	0.741	59.4	59.9	64.6	0.338
MLP	60.8	0.635	51.1	51.6	57.7	0.201	70.9	0.722	58.2	58.8	65.7	0.380	68.3	0.718	59.8	60.2	65.4	0.347
RBFN	68.7	0.714	62.3	62.7	66.8	0.365	65.8	0.675	46.8	47.5	56.6	0.219	70.9	0.735	62.9	63.2	67.8	0.389
SMO	65.7	0.634	49.6	52.8	57.5	0.321	65.4	0.617	44.8	47.7	54.8	0.291	69.6	0.675	58.0	59.5	64.2	0.382
SPEG	62.0	0.602	54.1	54.7	60.3	0.252	68.4	0.652	54.2	54.9	62.5	0.329	66.9	0.653	59.4	59.7	64.5	0.324
ADB	71.1	0.748	68.4	68.4	70.8	0.418	74.5	0.738	60.9	61.3	67.8	0.411	72.8	0.762	66.8	67.0	70.4	0.432
BAG	68.7	0.746	64.4	64.4	67.8	0.366	76.6	0.805	66.3	66.5	72.1	0.480	73.6	0.788	67.9	68.1	71.5	0.455
DAG	67.5	0.737	62.0	62.1	66.2	0.341	69.7	0.716	51.6	52.3	60.4	0.290	72.4	0.751	64.3	64.6	69.1	0.418
FC	62.1	0.628	54.0	54.4	59.7	0.228	71.9	0.755	64.7	64.8	70.9	0.445	68.9	0.706	62.6	62.8	67.1	0.369
LB	**72.9**	**0.765**	**70.2**	**70.2**	**72.6**	**0.453**	74.9	0.780	62.2	62.5	68.9	0.422	72.2	0.775	66.6	66.7	70.3	0.421
AM	63.9	0.676	55.2	55.7	61.1	0.265	68.4	0.727	52.3	53.5	61.0	0.326	68.0	0.741	59.4	59.9	64.6	0.338
RSS	68.7	0.759	62.9	63.1	67.1	0.365	74.1	0.802	65.0	65.4	71.1	0.469	72.5	0.784	66.6	67.0	70.4	0.442
CHIRP	68.7	0.678	62.9	63.1	67.1	0.365	71.9	0.678	58.2	58.8	65.7	0.380	71.4	0.694	62.6	63.0	67.6	0.399
FLR	54.2	0.520	34.5	36.3	45.2	0.048	58.1	0.554	45.6	45.6	54.5	0.108	61.5	0.601	51.9	52.3	58.4	0.205
VFI	53.1	0.656	63.2	65.9	45.3	0.149	49.4	0.653	59.5	64.3	43.6	0.218	59.2	0.683	60.7	64.2	53.7	0.303
DTNB	68.1	0.758	63.9	64.0	67.3	0.354	74.9	0.795	69.0	69.1	74.4	0.511	72.5	0.755	66.2	66.6	70.4	0.444
FURIA	66.7	0.673	65.4	65.5	66.4	0.330	74.1	0.722	62.1	62.6	68.8	0.430	72.2	0.727	66.2	66.5	69.9	0.426
MOD	69.3	0.687	64.8	64.9	68.4	0.378	77.5	0.732	65.4	66.4	71.1	0.507	73.1	0.715	67.4	67.7	70.8	0.446
MOEFC	57.8	0.623	44.4	45.4	52.7	0.135	71.9	0.434	57.9	58.4	65.4	0.371	68.0	0.634	58.2	58.8	64.1	0.333
NNGE	64.5	0.642	60.9	60.9	64.0	0.283	77.1	0.726	65.9	66.0	71.9	0.463	73.0	0.715	67.8	67.8	71.2	0.434
PART	67.5	0.690	66.3	66.3	67.6	0.352	67.9	0.754	61.5	61.6	67.5	0.344	70.9	0.733	65.0	65.5	68.7	0.408
NNEP	67.5	0.701	62.5	62.6	66.4	0.341	76.2	0.791	65.1	65.2	71.2	0.446	73.9	0.778	67.8	68.0	71.5	0.455
HFT	68.7	0.711	58.7	60.0	64.6	0.372	64.9	0.672	47.0	48.0	56.8	0.245	70.8	0.736	61.1	61.9	66.7	0.397
J48	66.9	0.683	65.8	65.9	67.0	0.341	74.5	0.745	65.9	66.1	71.8	0.450	71.7	0.723	65.9	66.2	69.5	0.417
GANN	62.7	0.660	55.7	55.9	60.9	0.241	72.7	0.751	60.7	60.9	67.7	0.387	70.5	0.759	62.8	63.1	67.7	0.385
RF	68.1	0.745	63.4	63.5	67.1	0.354	**80.5**	**0.862**	**69.8**	**69.9**	**75.0**	**0.528**	**76.9**	**0.817**	**71.7**	**71.9**	**74.9**	**0.516**
CART	68.1	0.740	64.4	64.5	67.6	0.355	73.2	0.717	60.9	61.0	67.8	0.374	70.4	0.720	64.1	64.3	68.2	0.380
SYSFOR	69.9	0.758	64.3	64.5	68.3	0.390	73.2	0.791	62.8	62.9	69.3	0.409	72.7	0.800	65.8	66.3	69.9	0.436

4.4.2 RQ2

What is the performance of the models with respect to predicting the trend of version to version change-proneness of files?

The efficacy of selected prediction techniques is also assessed via an inter-release validation (refer to VS2 in Figure 4.1 and Section 4.3.6) for predicting the trend of version to version change-proneness of Java files over the JFreeChart and Heritrix versions, wherein, a model generated via a particular version is validated on its successional release. This procedure is implemented on all the selected versions data sets. Table 4.10 and 4.11 display the inter-release validation results in terms of Accuracy (*Acc.*), Area under Curve (*AUC*), F-Measure or F-Score (*F-S*), G-mean1 (*G-m1*), G-mean2 (*G-m2*), and Matthews correlation coefficient (*MCCF*) for the two software projects. Having evaluated the performance of the 31 prediction techniques using CBFS +10-fold validation in RQ1 Section 4.4.1, it is vital to note here that we eliminated those techniques which were deemed unsuitable for change-proneness prediction of files over the selected releases of the two projects and therefore we were left with twenty five techniques. These techniques[6] (SMO, SPEG, FLR, CHIRP, VFI, MOEFC) have been excluded from further analysis owing to their poor predictive ability in terms of AUC as per the results in Tables 4.8 and 4.9. Also, since the features in the training dataset should match to the features of the testing dataset, therefore only 10-fold validation (sans the CBFS) is employed to train the models and no feature is eliminated from the training or the testing datasets. The results corresponding to the performance of the techniques with 10-fold validation have been provided in Appendices C and D.

[6] Even though acceptable prediction (in terms of cumulative average AUC) is exhibited by these techniques for the five releases of the JFreeChart datasets, this is not true for the Heritrix software project with these techniques underperforming for majority of the Heritrix releases. Since we were looking for apposite prediction techniques for the prediction of version to version change-proneness of Java files in general, therefore, only those techniques that majorly exhibited acceptable performances for both the selected software projects are examined further for additional empirical analysis in this research work.

Table 4.10 Prediction results of the 25 shortlisted prediction techniques over the five releases of JFreeChart using inter-release validation (VS2)

Tech.	JFreeChart 0.7.0 using JFreeChart 0.6.0						JFreeChart 0.7.1 using JFreeChart 0.7.0					
	Acc.	AUC	F-S	G-m1	G-m2	MCCF	Acc.	AUC	F-S	G-m1	G-m2	MCCF
BN	71.1	0.717	53.6	58.3	60.8	**0.437**	70.3	0.706	67.6	67.6	70.2	0.402
NB	61.1	0.667	25.5	34.0	38.5	0.200	70.3	0.706	67.6	67.6	70.2	0.402
FLDA	70.3	0.706	23.6	29.6	31.2	0.202	70.3	0.706	67.6	67.6	70.2	0.402
LR	55.6	0.580	23.1	26.7	36.5	0.023	70.3	0.703	64.9	65.1	70.2	0.334
MLP	67.8	**0.791**	**61.3**	**61.4**	**66.3**	0.339	70.3	0.706	67.6	67.6	70.2	0.402
RBFN	64.4	0.776	33.3	42.7	44.8	0.306	58.9	0.601	55.8	55.8	58.8	0.176
ADB	60.0	0.604	47.1	47.6	55.3	0.165	70.9	0.717	67.6	67.6	70.6	0.412
BAG	51.1	0.522	26.7	27.9	39.1	-0.058	70.3	0.719	68.0	68.1	70.3	0.405
DAG	60.0	0.613	21.7	30.2	35.1	0.165	65.8	0.648	65.4	65.7	66.2	0.330
FC	68.9	0.738	50.0	54.4	58.1	0.380	67.7	0.689	67.5	67.9	68.2	0.370
LB	52.2	0.524	29.5	30.7	41.5	-0.028	66.5	0.730	67.1	67.3	68.1	0.374
AM	55.6	0.580	23.1	26.7	36.5	0.023	60.1	0.576	54.7	54.7	59.3	0.191
RSS	61.1	0.649	31.4	37.0	43.5	0.185	72.2	0.720	69.4	69.5	72.0	0.440
DTNB	55.6	0.500	23.1	26.7	36.5	0.023	72.8	0.708	70.3	70.4	72.8	0.453
FURIA	58.9	0.552	27.5	32.3	40.2	0.119	60.8	0.629	58.7	58.8	61.0	0.218
MOD	67.3	0.696	70.2	71.3	67.6	0.405	67.3	0.696	70.2	71.3	67.6	0.405
NNGE	62.2	0.582	39.3	42.7	49.9	0.208	67.1	0.677	66.2	66.5	67.5	0.352
PART	60.0	0.522	18.2	28.7	31.8	0.179	63.9	0.596	62.7	63.0	64.3	0.286
NNEP	62.2	0.553	26.1	36.3	38.8	0.248	63.9	0.669	57.8	57.8	62.7	0.264
HFT	66.7	0.772	44.4	49.6	53.8	0.331	**74.1**	**0.762**	71.3	71.4	**73.9**	**0.477**
J48	61.1	0.509	25.5	34.0	38.5	0.200	63.9	0.622	62.3	62.4	64.2	0.283
GANN	64.4	0.688	44.8	47.7	54.2	0.262	68.4	0.706	67.6	67.6	67.3	0.402
RF	65.6	0.595	39.2	46.2	49.6	0.317	72.8	0.733	**71.9**	**72.2**	73.2	0.465
CART	**69.1**	0.647	48.1	53.7	47.9	0.391	70.9	0.706	67.6	67.6	70.6	0.412
SYSFOR	65.6	0.766	43.6	48.1	53.3	0.297	70.3	0.706	67.6	67.6	70.2	0.402

Table 4.10 (contd.) Prediction results of the 25 shortlisted prediction techniques over the five releases of JFreeChart using inter-release validation (VS2)

Tech.	JFreeChart 0.7.2 using JFreeChart 0.7.1						JFreeChart 0.7.3 using JFreeChart 0.7.2					
	Acc.	AUC	F-S	G-m1	G-m2	MCCF	Acc.	AUC	F-S	G-m1	G-m2	MCCF
BN	**77.5**	0.831	**77.3**	**77.4**	**77.6**	**0.553**	75.6	0.826	69.4	69.6	73.6	0.498
NB	64.6	0.764	56.1	56.9	47.8	0.291	73.1	0.747	58.8	62.0	65.0	0.465
FLDA	65.4	0.646	66.7	66.7	64.5	0.289	65.4	0.646	66.7	66.7	64.5	0.289
LR	63.6	0.684	52.3	53.9	59.1	0.273	64.7	0.742	47.6	49.5	56.4	0.262
MLP	65.1	0.597	45.2	51.6	54.1	0.358	55.8	0.635	57.1	58.0	56.3	0.149
RBFN	65.6	0.794	58.9	59.4	63.7	0.310	66.7	0.774	58.1	58.3	64.1	0.310
ADB	72.3	0.832	70.7	70.7	71.9	0.444	70.5	0.786	58.2	59.6	64.7	0.392
BAG	65.6	0.797	57.9	58.6	63.1	0.311	69.2	0.761	54.7	56.8	62.0	0.366
DAG	68.7	0.764	59.6	61.1	65.0	0.384	65.4	0.799	42.6	47.1	52.5	0.288
FC	73.3	0.771	66.2	67.5	70.3	0.480	69.2	0.637	46.7	53.5	55.3	0.406
LB	65.6	0.731	56.2	57.4	62.1	0.315	66.7	0.798	45.8	49.9	55.0	0.318
AM	63.6	0.684	52.3	53.9	59.1	0.273	64.7	0.742	47.6	49.5	56.4	0.262
RSS	67.2	0.749	59.5	60.3	64.6	0.344	66.7	0.745	48.0	51.0	56.7	0.312
DTNB	70.8	0.800	67.1	67.2	70.0	0.412	75.6	0.819	62.7	66.1	68.0	0.527

Table 4.10 (contd.) Prediction results of the 25 shortlisted prediction techniques over the five releases of JFreeChart using inter-release validation (VS2)

Tech.	JFreeChart 0.7.2 using JFreeChart 0.7.1						JFreeChart 0.7.3 using JFreeChart 0.7.2					
	Acc.	AUC	F-S	G-m1	G-m2	MCCF	Acc.	AUC	F-S	G-m1	G-m2	MCCF
FURIA	66.7	0.671	59.1	59.9	64.2	0.333	69.9	0.676	48.4	54.8	56.6	0.420
MOD	75.6	0.752	71.2	71.7	74.2	0.523	64.7	0.604	42.1	46.2	52.2	0.269
NNGE	69.7	0.687	61.4	62.7	66.4	0.404	70.5	0.659	48.9	56.0	57.0	0.443
PART	66.7	0.777	51.9	55.6	59.2	0.365	66.7	0.707	43.5	48.9	53.1	0.327
NNEP	69.2	0.814	58.3	60.8	64.2	0.406	66.7	0.804	49.0	51.6	57.6	0.309
HFT	65.1	0.781	55.3	56.6	61.4	0.305	**82.7**	**0.849**	**77.3**	**78.0**	**80.1**	**0.650**
J48	76.9	0.724	75.7	75.7	76.9	0.537	64.7	0.642	42.1	46.2	52.2	0.269
GANN	65.6	0.606	55.6	57.0	61.8	0.317	67.3	0.770	47.4	51.3	56.2	0.332
RF	75.9	**0.862**	75.1	75.2	76.0	0.518	70.5	0.806	64.1	64.1	68.9	0.393
CART	59.5	0.530	47.0	48.3	54.7	0.183	69.2	0.627	52.0	55.3	59.8	0.375
SYSFOR	65.1	0.782	55.3	56.6	61.4	0.305	**82.7**	**0.849**	**77.3**	**78.0**	**80.1**	**0.650**

Table 4.11 Prediction results of the 25 shortlisted prediction techniques over the five releases of Heritrix using inter-release validation (VS2)

Tech.	Heritrix 0.4.0 using Heritrix 0.2.0						Heritrix 0.6.0 using Heritrix 0.4.0					
	Acc.	AUC	F-S	G-m1	G-m2	MCCF	Acc.	AUC	F-S	G-m1	G-m2	MCCF
BN	67.7	0.687	74.0	74.5	63.7	0.346	60.2	0.724	60.5	61.8	61.5	0.261
NB	69.0	0.743	75.8	76.5	63.5	0.380	70.8	0.705	63.8	63.8	69.6	0.393
FLDA	65.4	0.646	66.7	66.7	64.5	0.289	64.9	0.667	55.9	55.9	63.0	0.268
LR	72.9	0.736	77.4	77.7	70.5	0.452	66.1	0.654	59.2	59.2	65.3	0.303
MLP	68.4	0.687	72.6	72.7	67.0	0.357	69.0	0.721	62.9	63.0	68.5	0.365
RBFN	69.0	0.685	72.4	72.4	68.3	0.372	68.4	0.715	60.3	60.3	66.7	0.341
ADB	69.0	0.751	76.5	77.5	61.6	0.391	68.1	0.728	61.3	61.6	66.3	0.320
BAG	69.7	0.731	75.9	76.6	65.1	0.391	69.0	0.715	63.9	64.1	69.0	0.374
DAG	71.0	0.749	76.7	77.2	66.9	0.417	66.1	0.727	61.8	62.2	66.6	0.325
FC	66.5	0.640	74.5	75.6	58.7	0.330	69.6	0.696	60.0	60.1	66.8	0.357
LB	72.3	**0.783**	77.5	78.0	68.8	0.443	64.3	**0.746**	60.1	60.5	64.8	0.292
AM	72.9	0.736	77.4	77.7	70.5	0.452	66.1	0.654	59.2	59.2	65.3	0.303
RSS	67.1	0.682	74.1	74.8	61.7	0.335	69.6	0.728	62.9	62.9	68.6	0.372
DTNB	67.1	0.754	74.1	74.8	61.7	0.335	69.6	0.723	65.3	65.7	70.0	0.392
FURIA	65.8	0.682	70.4	70.5	64.2	0.304	68.4	0.687	63.0	63.2	68.2	0.359
MOD	71.0	0.689	77.4	78.2	65.3	0.426	67.8	0.686	64.1	64.4	68.5	0.363
NNGE	71.6	0.693	78.2	79.2	65.2	0.447	70.2	0.710	**66.7**	**67.1**	**70.9**	**0.411**
PART	67.7	0.644	74.0	74.5	63.7	0.346	**70.8**	0.688	60.9	61.1	67.6	0.379
NNEP	**73.5**	0.770	**78.8**	**79.3**	**69.7**	**0.473**	63.2	0.701	60.4	61.0	64.2	0.283
HFT	68.4	0.702	75.1	75.9	63.1	0.364	64.9	0.712	57.7	57.8	64.0	0.279
J48	68.4	0.639	73.8	74.1	65.5	0.357	64.3	0.678	60.1	60.5	64.8	0.292
GANN	70.3	0.756	77.7	78.9	62.4	0.427	65.5	0.711	58.2	58.2	64.5	0.289
RF	66.5	0.752	73.2	73.7	61.8	0.319	64.3	0.690	61.1	61.7	65.2	0.302
CART	69.0	0.645	74.2	74.5	66.4	0.370	68.4	0.665	59.1	59.1	65.9	0.335
SYSFOR	68.4	0.694	75.1	75.9	63.1	0.364	65.5	0.712	58.2	58.2	64.5	0.289

Table 4.11 (contd.) Prediction results of the 25 shortlisted prediction techniques over the five releases of Heritrix using inter-release validation (VS2)

Tech.	Heritrix 0.8.0 using Heritrix 0.6.0						Heritrix 0.10.0 using Heritrix 0.8.0					
	Acc.	AUC	F-S	G-m1	G-m2	MCCF	Acc.	AUC	F-S	G-m1	G-m2	MCCF
BN	66.9	**0.733**	**58.6**	**59.2**	**64.1**	0.329	58.4	0.691	60.0	62.0	59.8	0.252
NB	56.6	0.651	33.3	36.5	44.7	0.103	64.9	0.667	59.3	59.6	65.0	0.294
FLDA	59.6	0.680	42.7	44.8	52.0	0.175	62.8	0.633	56.6	56.8	62.6	0.248
LR	59.6	0.670	39.6	42.6	49.7	0.177	62.3	0.618	54.9	55.1	61.6	0.230
MLP	59.6	0.569	41.7	44.1	51.3	0.175	61.9	0.622	56.0	56.3	61.9	0.234
RBFN	65.7	0.679	54.4	55.7	61.0	0.307	54.1	0.639	48.5	49.0	54.4	0.085
ADB	**70.5**	0.705	56.6	60.3	63.0	0.438	62.8	0.637	57.0	57.3	62.8	0.251
BAG	60.2	0.693	36.5	41.2	47.4	0.200	63.2	0.662	57.7	58.0	63.4	0.262
DAG	61.4	0.684	41.8	45.3	51.6	0.223	62.3	0.622	58.4	59.0	63.2	0.260
FC	62.1	0.664	30.8	41.4	42.7	0.301	65.4	0.641	51.2	51.5	60.1	0.251
LB	66.3	0.676	48.1	52.7	56.5	0.348	62.3	0.661	59.5	60.3	64.0	0.273
AM	59.6	0.670	39.6	42.6	49.7	0.177	62.3	0.618	54.9	55.1	61.6	0.230
RSS	58.4	0.663	28.9	35.0	41.2	0.160	**67.5**	0.669	61.9	62.2	67.6	0.344
DTNB	62.7	0.648	47.5	49.5	55.7	0.244	61.9	0.677	58.5	59.2	62.9	0.257
FURIA	63.9	0.647	42.3	47.7	51.9	0.296	55.4	0.612	59.3	61.9	56.0	0.224
MOD	59.0	0.561	32.0	37.5	43.8	0.172	63.6	0.644	59.2	59.7	64.3	0.281
NNGE	64.5	0.624	49.6	52.0	57.5	0.287	58.9	0.611	57.4	58.5	60.3	0.219
PART	62.7	0.635	38.0	44.5	48.6	0.275	62.3	0.565	56.7	57.0	62.5	0.244
NNEP	60.1	0.710	38.8	44.1	49.2	0.250	58.4	0.657	52.9	53.3	58.7	0.170
HFT	63.3	0.714	39.6	45.9	49.8	0.289	64.1	0.674	57.4	57.6	63.7	0.269
J48	62.7	0.640	39.2	45.0	49.5	0.269	64.1	0.658	59.5	59.9	64.7	0.288
GANN	61.5	0.696	38.5	43.3	48.9	0.232	58.9	0.612	50.3	50.3	57.7	0.155
RF	66.9	0.719	45.5	52.8	54.4	**0.391**	67.4	**0.712**	**64.8**	**65.6**	**68.8**	**0.375**
CART	68.1	0.656	56.9	58.6	63.2	0.362	58.9	0.627	48.1	48.1	56.5	0.141
SYSFOR	62.1	0.711	36.4	43.1	47.3	0.261	64.1	0.675	57.4	57.6	63.7	0.269

Tables 4.10 and 4.11 again show that none of the twenty five shortlisted prediction techniques exhibit a consistently highest predictive performance over the selected version datasets so as to be ascertained as the best performing technique for predicting the trend of change-proneness of Java files. AUC values range from 0.5 to 0.791 for JFreeChart 0.7.0, 0.622 to 0.796 for JFreeChart 0.7.1, 0.579 to 0.862 for JFreeChart 0.7.2, and 0.604 to 0.849 for JFreeChart 0.7.3. However, the results of the predictive models generated corresponding to JFreeChart 0.7.2 (when JFreeChart 0.7.1 was employed as a training set) in Table 4.10 display the best AUC values, with only five out of twenty five techniques exhibiting AUC below the acceptable range.

On the other hand, as observed from Table 4.11, AUC values vary from 0.639 to 0.783 for Heritrix 0.4.0, 0.654 to 0.746 for Heritrix 0.6.0, 0.561 to 0.733 for Heritrix 0.8.0, and 0.565 to 0.712 for Heritrix 0.10.0. However, 24 out of the 25 shortlisted

techniques exhibit their worst performance on Heritrix 0.10.0, with only RF exhibiting acceptable AUC values. This could be due to the fact that the prediction techniques, in general, underperform for Heritrix 0.8.0 even during the CBFS + 10 fold validation with 17 out of 31 techniques exhibiting unacceptable AUC values as seen in Table 9. Since in Table 4.11, the prediction techniques are trained on Heritrix 0.8.0 using 10 fold-validation and tested on Heritrix 0.10.0 during inter-release validation, therefore, poor results in terms of predictive performance are observed. This is because majority of the prediction techniques perform below the acceptable range of AUC for Heritrix 0.8.0 even during ten-fold validation indicating poor efficiency on the part of the techniques for predicting change-proneness of files for that version. Such poorly trained models when tested on a fresh set of data such as Heritrix 0.10.0 therefore severely underperform.

Overall, taking an average of the average AUC values obtained across the versions of the two target projects, it can be found that:

- As expected the results obtained during inter-release validation for trend estimation (VS2) are worse than those achieved by the intra-release analysis (VS1), with the shortlisted prediction techniques, in general, performing better for the JFreeChart datasets in comparison to the Heritrix datasets.
- Although techniques like NB, BAG, LR, NNEP etc. obtain acceptable values for AUC for estimating the trend of change-proneness of Java files over the versions of the JFreeChart software, they underperform when applied for the same on the selected Heritrix version datasets.
- However, the models developed using BN, ADB, HFT, RF and SYSFOR techniques show an average AUC of 0.7 or higher over the versions of the JFreeChart and Heritrix projects. This strengthens the potency of the employed prediction techniques for the development of models for estimating the trend of change-proneness of Java files over the versions of JFreeChart and Heritrix projects.

4.4.3 RQ3

Is the predictive performance of the change-proneness prediction models developed via k-fold cross-validation statistically similar to or different from the performance of the models constructed by means of an inter-release validation?

We validate if there is any statistical difference amongst the results obtained using 10-fold, CBFS + 10-fold validation (VS1) and inter-release validation (VS2) on all the data sets using Kruskal-Wallis test [27], the results of which are presented in Tables 4.12(a) and (b). We used PASW Statistics 18 (SPSS, Chicago, IL, USA) [214] to perform the Kruskal-Wallis test. The AUC and MCCF values obtained during the three validation procedures by the prediction techniques over the various versions of each of the two software projects have been analysed in the test.

It can be seen from Tables 4.12(a) and (b) that there exists a vast difference among the mean ranks obtained by the prediction techniques during the three validation scenarios. The techniques exhibit the lowest performance with respect to the AUC

Table 4.12(a) Kruskal–Wallis test results for comparison between the validation techniques over the JFreeChart and Heritrix versions using AUC as a performance measure

Kruskal–Wallis test results			**Mean Ranks**		
Test Statistics	JFreeChart	Heritrix	Validation tech.	JFreeChart	Heritrix
Chi-square value	130.91	68.35	10-fold validation	188.75	163.98
DF	2	2	CBFS+10-fold validation (VS1)	193.25	193.11
p-value	0.000	0.000	Inter-release validation (VS2)	69.51	94.42

H_0: The samples do not belong to the same population;
H_a: The samples belong to the same population;
Since the evaluated *p*-value is less than the threshold of significance α, therefore, the null hypothesis H_0 is accepted

Table 4.12(b) Kruskal–Wallis test results for comparison between the validation techniques over the JFreeChart and Heritrix versions using MCCF as a performance measure

Kruskal–Wallis test results			**Mean Ranks**		
Test Statistics	JFreeChart	Heritrix	Validation tech.	JFreeChart	Heritrix
Chi-square value	163.16	51.19	10-fold validation	192.18	160.68
DF	2	2	CBFS+10-fold validation (VS1)	199.21	188.40
p-value	0.000	0.000	Inter-release validation (VS2)	60.12	102.43

H_0: The samples do not belong to the same population;
H_a: The samples belong to the same population;
Since the evaluated *p*-value is less than the threshold of significance α, therefore, the null hypothesis H_0 is accepted

and MCCF values during the Inter-release validation (VS2) even though the models are trained using the 10-fold validation wherein the models are seen to perform well. This is a common phenomenon in predictive analytics and is called as the "Model performance mismatch" problem wherein model skills on the training dataset do not match the skills of the model on the test dataset. This occurs because some small overfitting of the training dataset is inevitable given hyper-parameter tuning thus rendering the training scores optimistic. Therefore even though the models trained using 10-fold validation indicate higher MCCF values and acceptable to outstanding AUC values, there could exist a major disparity in performance when they are tested on a new sample during the inter-release validation (VS2).

Additionally as observed in Tables 4.12(a) and (b), the test does yield significant results for both the performance measures, due to which we accept the null hypothesis that states that that the performance of the developed models using the three validation techniques is not comparable to each other.

4.4.4 RQ4

Which are the best and the worst techniques for change-proneness prediction of Java files of the selected target projects?

As observed in Tables 4.8-4.11, each of the selected prediction techniques is evaluated on multiple versions of the two plugin projects. As a result, it is hard to evidently establish the predictive pre-eminence of one technique over the other. Therefore, we apply the Scott-Knott cluster analysis [209] (as detailed in Section 4.3.7.2) to statistically compare the performance of the prediction techniques over each of the specific validation scenarios (VS1 and VS2) according to their AUC values and then cluster them into subdivisions, where each subdivision or subgroup consists of those techniques that are significantly similar.

All the shortlisted prediction techniques analysed in RQ2 are further analysed using Scott-Knott cluster analysis for the purpose of identifying similar groups of methods and to select the group with highest AUC (i.e., the best group). We used R (version 3.6) and RStudio (version 1.1.456) [215] to perform the Scott-Knott analysis. One could therefore obtain a group of techniques which have a statistically similar highest performance in terms of AUC values via Scott-Knott analysis, as opposed to non-

parametric tests like the Friedman's test which allocates mean ranks to individual techniques on the basis of their values with respect to a performance measure.

As the Scott-Knott algorithm assumes that the distribution of AUC values is approximately normal, we tested the distribution of AUCs from each of the twenty five prediction techniques using the Shapiro-Wilk test and found that they were normally distributed. We also compared the mean and median of AUC values obtained per prediction technique per project and observed that mean to median ratio for every technique lies between 0.96-1.05. This indicates that the AUC values are normally distributed and could be utilized as input to the Scott-Knott test.

Figure 4.2 shows graphical plot of Scott-Knott cluster analysis based on ANOVA significance test and using AUC values obtained by the JFreeChart and Heritrix datasets via the twenty five shortlisted prediction techniques. The x-axis indicates the prediction technique organized as per their ranks where better performing techniques start from left hand side. The AUC values are indicated on the y-axis and mean of the AUC values are represented by the small circles on each vertical line. The gray box on the left in each figure indicates the best subgroup of prediction techniques that exhibits significantly highest AUC. The Scott-Knott analysis resulted in eight homogeneous clusters wherein techniques in each of the clusters show statistically similar performance in both the Figures 4.2(a) and 4.2(b).

Figure 4.2(a) presents the Scott Knott analysis test results for the prediction techniques during the CBFS+10 fold validation for both the software projects. As observed from Figure 4.2(a) the RF technique obtains the highest AUC values for software change-prediction over the various JFreeChart and Heritrix versions and this performance is statistically significantly higher from the remaining techniques. SYSFOR, LB, BAG and RSS techniques follow the RF technique and are observed to exhibit statistically similar performance to each other in terms of AUC values.

Figure 4.2(b) presents the Scott Knott analysis test results for the prediction techniques during the inter-release validation for both the target projects. As observed from Figure 4.2(b) the HFT, BN, SYSFOR and RF techniques perform statistically significantly higher for estimating the trend of software change-prediction over the various JFreeChart and Heritrix versions. The ADB technique follows next.

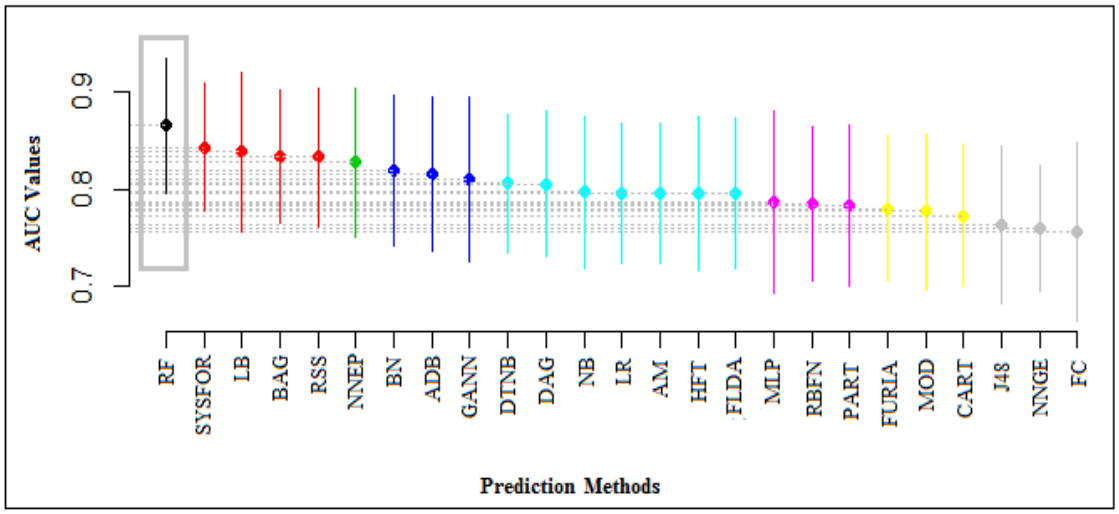

Figure 4.2(a) Scott Knott analysis for prediction techniques during the CBFS+10 fold validation (VS1)

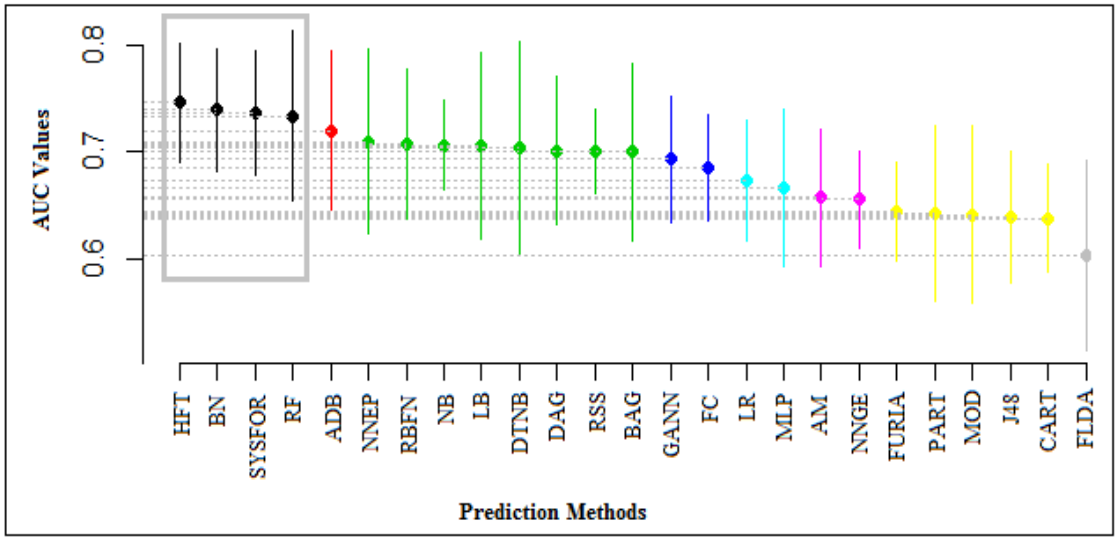

Figure 4.2(b) Scott Knott analysis for prediction techniques during the inter-release validation (VS2)

The Scott-Knott cluster analysis concludes the RF technique to be the best performing technique for software change prediction over the selected JFreeChart and Heritrix datasets during CBFS + 10 fold validation (VS1), with no other selected prediction technique exhibiting a statistically similar high predictive performance. However, for the inter-release validation (VS2), four techniques are observed to occupy the best performing cluster. Since each homogeneous cluster consist of prediction techniques that have a statistical similarity in performance, thus one cannot just rely only on the

ranking obtained by Scott-Knott as shown in Figure 4.2(b) to ascertain the best performing prediction technique. Therefore, we consult Borda count method [210, 212] to rank all the techniques in the best performing cluster across all the remaining performance measures which are Accuracy (*Acc.*), F-Measure or F-Score (*F-S*), G-mean1 (*G-m1*), G-mean2 (*G-m2*), and Matthews correlation coefficient (*MCCF*).

Table 4.13 shows the prediction techniques sorted by the calculated Borda score seen in all performance measures over each validation scenario. The Borda score for each technique is calculated in the same way as explained in the example given in 4.7.4. The technique with the largest score is ranked first. We also analysed the techniques in the second best performing cluster using the Borda score, simply to assess their relative performance with respect to each other vis-à-vis the remaining performance measures.

Table 4.13 Ranks allotted to techniques after Borda ranking

CBFS+10 fold validation (VS1)		**Inter release validation (VS2)**	
Rank	prediction technique	Rank	prediction technique
1	RF	1	BN
2	BAG	2	RF
3	RSS	3	ADB
4	SYSFOR	4	HFT
5	LB	5	SYSFOR

As observed from Table 13, the RF technique performs the best for VS1 even when the remaining performance measures are taken into consideration and is concluded to be the best prediction technique among the selected techniques for version to version change-proneness prediction of Java files over the selected JFreeChart and Heritrix versions. This is followed by the techniques from the second cluster wherein the ensemble technique of BAG (Bagging) performs the best followed by another technique, RSS (Random Sub Space).

Unlike the Scott-Knott results, it is clear from the final scores form the Ranked Voting using Borda counting scheme that as far as the Inter-release validation(VS2) is concerned, the BN technique outperforms all other techniques in regard to the selected performance measures for version to version change prediction of Java files.

This is followed by the RF technique which is also ascertained to perform the best for the VS1. This is followed by the ensemble technique of ADB (ADaBoost) which was observed to occupy the second best performing cluster in the Scott-Knott analysis as seen in Figure 4.2(b). This indicates that one should analyse the performance of the prediction techniques taking maximum number of performance measures into consideration as a technique performing well as per one measure might not perform well with respect to other measures.

Summarizing the results drawn from the Scott-Knott cluster analysis:

The RF technique outperforms all other ICMs and statistical approaches including those selected under the Decision Tree category with respect to change-proneness prediction of Java files during VS1 and obtains the second highest performance (the highest among the decision tree classifiers) during VS2 over the selected releases of both the target projects. Even though the BAG algorithm comes close in terms of prediction performance during VS1, it is outperformed by the RF technique. This is because RF technique is an improvement over BAG. The chief drawback with decision trees, such as CART, which are observed to underperform in both the validation scenarios, is that they employ the greedy algorithm to choose the splitting feature that minimizes error. Even in the case of the BAG technique, the decision nodes can comprise of many structural semblances and as a result have strong correlations in their estimates. Ensembles merge predictions from several models which is effective only if the predictions of the sub-trees are weakly correlated or uncorrelated. RF alters this behaviour in a way that the sub-models are learned and therefore there is a lesser correlation among the subsequent predictions from all of the sub-trees. In CART the learning algorithm scrutinizes all the features and their values for the purpose of selecting the optimum split-point. The RF technique alters this process as well so as to limit the search of the learning algorithm to a random sample of attributes.

Another previously unexamined technique, SYSFOR, also exhibits high performance in predicting version to version change-proneness of Java files and is observed to be one of the top five techniques for both the validation scenarios. SYSFOR, also a decision tree classifier, outperforms other selected techniques because contrasting to the prevailing methodologies it does not need to construct a tree that employs a feature or an attribute that has a poor gain ratio. This results in the generation of

superior logic rules and prediction accuracy of the trees that is higher than those generated by trees that are constructed by means of poor attributes. Also, disparate from other techniques, SYSFOR permits the selection of a numeric feature multiple times provided the gain ratios are prominently high and the split points are well segregated.

The evolutionary technique of NNEP was observed to follow the top five techniques in terms of AUC values for both the validation scenarios during the Scott-Knott cluster analysis. This is because the NN is inherently a deterministic technique and one of the advantages of such techniques is that they are at a lower risk of over-fitting than non-deterministic algorithms [44]. Thusly, even though AUC values obtained by the NNEP technique during the inter-release validation scenario are lower than those obtained during the CBFS+10 fold validation, the NNEP technique (relative to the other techniques) performs with the same prediction prowess.

On the other hand, the other evolutionary techniques of GANN and AM underperformed in comparison to NNEP and other ML techniques in terms of AUC values, especially for the inter-release validation scenario. GANN and AM being non-deterministic techniques are more prone to overfitting and have more flexibility when learning a target function [30, 61]. Overfitting happens when the models learn the detail and noise in the training data to the extent that it negatively impacts the performance of the model on new data. This means that the noise or random fluctuations in the training data is picked up and learned as concepts by the model. The problem is that these concepts do not apply to new data and negatively impact the models ability to generalize. This is why GANN and AM perform better for VS1 when they are evaluated on the same version’s dataset versus during VS2 when they are subjected to an inter-release validation.

Classifier BN is observed to perform the best for both the target projects during VS2 taking all the performance measures into consideration. This is because the Bayesian classifiers, as opposed to other prediction techniques, do not rely on asymptotic approximation and provide accurate conclusions that are restricted to the data being analyzed. Bayesian classifiers also evaluate the parameter functions directly, sans the employment of the "plug-in" method (a method for the estimation of the functionals by plugging the projected parameters in the functionals).

This efficiency of the Bayesian classifiers in both the validation settings is utilized for improving the predictive accuracy of the HFT technique by virtue of which it performs the best overall for VS2 during the Scott-Knott cluster analysis with AUC as the deciding performance measure. This is done in accordance with the empirical inference drawn by Gama and Medas [216] which states that the BN technique when employed at the leaves of an HFT instead of the majority class classifier leads to a significant improvement in its classification accuracy.

Ensemble learners like RSS, ADB and LB methodologies show exceptional results for a majority of the selected versions for predicting the trend of version to version change-proneness of Java files, obtaining high mean ranks during the Scott-Knott cluster analysis. These techniques have also shown to exhibit acceptable and sometimes even excellent AUC values during the CBFS+ 10 fold validation over the JFreeChart and Heritrix versions. Ensemble learners generate successful prediction models since they employ various prediction techniques in unison to find improved prediction ability than the individual prediction techniques. Therefore, ensemble learners could similarly be employed on other data sets for generating version to version change-proneness prediction models with respect to Java files.

The next section summarises this chapter.

4.5 Chapter Summary

This chapter aimed at extending our analysis pertaining to software change prediction by providing a framework which is capable of assisting the software developers in an expert selection of a prediction approach amid a massive choice of prevailing methods.

31 prediction techniques belonging to the category of ICMs and statistical classifiers were assessed on datasets gathered from multiple successional releases of two plugin projects: JFreeChart and Heritrix for evaluating their efficacy in predicting version to version change-proneness of Java files. Apposite pre-processing techniques like SMOTE, Inter-Quartile Range filter and Correlation-Based Feature Selection were applied on the datasets for the purpose of class-balancing, removal of outliers and extreme values, and attribute reduction. The generated models were empirically validated using intra-release and inter-release scenarios on the various selected releases of the two target projects in order to acquire all-pervading results. Measures

like Accuracy, Area under Curve, F-Score, G-mean1, G-mean2, and Matthews correlation coefficient were employed as metrics to depict the predictive performance of the prediction techniques. Additionally, Kruskal-Wallis test and Scott-Knott cluster analysis with Borda counting were employed for assessing the statistical significance of the acquired results and to gather valid and generalized inferences.

The primary conclusions of the analysis, most of which were gathered while answering the RQs in Section 4.4, are stated as follows:

- Among the 17 selected OO metrics, CBO, EC, CogC, RFC, DIT, CHV and SLOC were found to be significant predictors of change-proneness of Java files over the JFreeChart and Heritrix plugin projects using the CBFS method.
- The work authenticates the overall predictive potency of the selected prediction techniques with respect to change-proneness prediction of Java files over JFreeChart and Heritrix releases under the intra-release (VS1) and inter-release (VS2) validation scenarios. The Kruskal-Wallis test results however indicate that the employment of SMOTE considerably augments the predictive ability of the models against those situations where no feature selection is applied. Moreover, granting that the prediction techniques demonstrate their capability for estimating the version to version trend of Java files for most of the selected versions during inter-release validation (VS2), their results are inferior in comparison to those obtained during CBFS+ 10-fold validation (VS1).
- Furthermore, certain previously unexamined techniques like the techniques based on the concept of support vector machines like SMO and SPEG, evolutionary technique of MOEFC, the fuzzy classifier FLR, techniques under the miscellaneous category like CHIRP and VFI, show extremely poor performances and are therefore deemed to be highly unsuitable for prediction of version to version change-proneness of Java files.
- The Scott-Knott cluster analysis results also conclude the RF technique to be the best performing technique for software change prediction over the selected JFreeChart and Heritrix datasets during CBFS + 10 fold validation (VS1), with no other selected prediction technique exhibiting a statistically similar high predictive performance. However, for the inter-release validation (VS2), four techniques: HFT, BN, SYSFOR and RF are observed to statistically similarly highest performance with respect to the AUC values via the Scott-Knott analysis.

- The RF technique performs the best for VS1 even when the remaining performance measures are taken into consideration with the Borda count method and is concluded to be the best prediction technique among selected for version to version change-proneness prediction of Java files over the selected JFreeChart and Heritrix versions. This is followed by the techniques from the second cluster wherein the ensemble technique of BAG (Bagging) performs the best followed by another prediction technique RSS (Random Sub Space). Unlike the Scott-Knott results, it is clear from the final scores form the Borda ranking that as far as the Inter-release validation(VS2) is concerned, the BN technique outperforms all other techniques in regard to the selected performance measures for predicting the trend of version to version software change prediction of Java files. This is followed by the RF technique which is also ascertained to perform the best for the VS1.

Therefore, this pragmatic analysis provides a confirmation with respect to the ability of the techniques in predicting version to version change-proneness of Java files. In particular, we incorporate an inter-release evaluation, therefore imitating with the second validation a predicament that is normally encountered in an actual software maintenance milieu, wherein historical data from the previous versions are utilized for training a model which in turn predicts change-prone components of a new release. Moreover this study also establishes the predictive ability of a previously unexamined technique: SYSFOR, which is seen to be one of the top five techniques for both the validation scenarios.

CHAPTER 5
CHANGE-PRONE CODE SELECTION AND EVALUATION: A SEMANTIC WEB PERSPECTIVE

In this chapter we present a selection mechanism, based on a niche domain ontology (one of the chief constituents of Semantic web technology) that constructs a knowledge base from multiple successive releases of software component, to select change-prone code components from individual software component releases. These code components are source code Java files that have been substantively used with/without change from a previous release in a novel release.

The chapter proceeds as follows: Section 5.1 provides some background. Section 5.2 presents the proposed change-prone Java file selection mechanism. Section 5.3 describes the data collected, pre-processing performed and the construction of a Semantic web-based knowledge-base. Section 5.4 details the rules formulated for context reasoning in order to infer knowledge. Section 5.5 explains the empirical analysis conducted to validate the proposed framework. This is followed by Section 5.6 that concludes the chapter.

5.1 Adding a Semantic web perspective to software change

As read in Section 2.4 of Chapter 2, the foundations of Semantic web technologies have, in the recent past, thrived as artifacts to depict domain knowledge and have emerged as a vital decision-making element in the area of Component-Based Software Engineering (CBSE). Ontologies, in particular, have been extensively employed to classify, select, match and integrate software components and have been considered to be effective in acquiring and utilizing the software component knowledge on the retrieval system and matching procedure, afore and thereafter.

However, application and assessment of Ontology with respect to the source code components present within the software component remains vastly unexamined. This is due to the fact that researchers have focussed on building the software component

ontological knowledge-base using generic qualitative component specifications rather than specific quantitative features suitable or relevant to an exclusive software development setting. Moreover existing reuse repositories housing software components have been reduced to mere content management systems from which knowledge related to source code of software projects and components has to be mined to allow managers or maintainers to better aid the software evolution process and improve the quality of the software in the long run [145, 147].

Moreover, attributes or metrics that impact or affect a potential change of the source code within a software component have not been included in these repositories, in spite of there being adequate research and literature for the same as reviewed in Section 2.2.1 of Chapter 2.

This leads us to the following research question:

"Having already demonstrated their efficacy in specification, selection and retreival of software components albeit based on genric specifications, can Semantic web technologies in any way be extended so as to be useful to select code components within these software components vis-à-vis a specific criteria (software change proneness, as per our context of application)?"

A suitable solution to this lies in one of the primary constituents to Semantic web: Ontologies. Ontology [119], as described in Section 2.4.1 of Chapter 2, are among the most appropriate solutions for context modelling information with respect to software components in reuse repositories. This is because of the high and formal expressiveness that they offer along with the benefits that come with the application of ontology reasoning techniques. In addition, the ease with which ontologies create context-aware repositories is shown in many works [75, 122, 137, 145, 147, 159].

Building on this perspective and to answer the research question stated above, we propose a mechanism for Change-Prone Java file Selection (CPJFS) in this chapter which employs Semantic web technology. This mechanism begins by constructing an ontology in Protégé corresponding to the Java files contained in five JFreeChart and five Heritrix plugin projects. OWL [123] is selected as the description language for modelling the relevant change-proneness related contextual information with respect to the Java files in an efficient way. SWRL [83] and Drools Rule Engine is employed to analyse the contextual information that would subsequently infer these Java files as

change-prone or stable and populate the ontology temporarily. Finally, SWQRL [130] is employed to query the now populated ontology and select the relevant change-prone Java files belonging to the software components (selected versions of JFreeChart and Heritrix plugin projects). The chapter concludes with a comparative study of the proposed change-prone Java file selection (CPJFS) mechanism results against the actual change statistics of the software project versions.

The findings of Chapter 4 are extensively utilized to construct the CPJFS mechanism in this chapter. The contextual information modelled in OWL corresponds to the values of the source code metrics that have been validated in the Chapter 4 to be the most effective for the change-proneness prediction of Java files vis-à-vis the two selected plugin projects: JFreeChart and Heritrix. The SWRL rules employed to classify a Java file as change-prone or stable are constructed from the technique validated to be the most effective for predicting software change, also in Chapter 4. However, as oppposed to Chapter 4, the SWRL rules utilized in Chapter 5 are constructed by executing the prediction technique post the normalization and unsupervised discretization of the datasets of the plugin projects.

The proposed mechanism additionally:

- Provides insights so as to how software developers and practitioners can model supplementary source code related information regarding a software component at a finer granularity (file level, in our case) in an ontology-based software repository.
- Demonstrates how information (in the form of patterns or rules) which is constructed by an Intelligent Computing Methodology (ICM) to make relevant predictive decisions can be expressed in the form of valid SWRL rules in an ontology.

Such ontologies are capable of transforming existing reuse repositories from being content management systems to knowledge-bases wherein information regarding the source code components, like Java files, of software components are not only stored but also processed for making relevant software maintenance decisions.

5.2 Proposed Mechanism

The Semantic web-based methodology for CPJFS not only provides a systematic platform for describing quantitative features related to the source code Java files of a software component but is also enriched with reasoning ability and retrieval functions. Factors specifically responsible for the change-proneness classification of Java files have been employed to methodically construct the ontology. The CPJFS procedure has been shown in Figure 5.1 and can be described as:

Step 1. *Knowledge acquisition and pre-processing:*

The change-proneness related empirical data corresponding to the individual Java files of the software components is acquired to build the knowledge base. This acquisition is done via statically analysing each version of the software component over various source-code analysis tools and then gathering the numerical values generated by it. The acquired information is then pre-processed in a manner suitable for it to be included in constructing a normative framework for ontology.

Step 2. *Knowledge Representation:*

Ontology is employed for representing the Java files and their corresponding features. The main constituents of ontology are classes, attributes, and relationships. Classes represent the concept of domain knowledge. Attributes describe the features of classes. Relationships describe the relationship between class and class. Each of the software component, its version, the Java files in that version and the Java file's constituent features are context modelled to form an ontology.

Step 3. *Knowledge Reasoning and Inference:*

Ontologies are extended with rules written in SWRL, an ontology-based rule language. SWRL rules assert domain restrictions on an OWL ontology that merely represents facts. Modelling and implementation of the proposed ontology of the Java files and SWRL rules is carried using the Protégé-OWL editor and SWRLTab extension to Protégé. The rules indicate the following: Which of the Java files in the ontology belonging to a software component version are change-prone according to the values of their attributes?

The rules have been retrieved from analysing the data gathered and pre-processed Step 1 by means of a prediction technique, the details of which have been provided in future subsections.

Direct inference cannot be done using SWRL rules format, so format transformation of SWRL rules to the reasoning engine acceptable format is required for which Drools Rule Engine is employed. Drools is a Production Rule System that uses the rule-based approach to implement an Expert System. Expert Systems are knowledge-based systems that use knowledge representation to process acquired knowledge into a knowledge-base that can be used for reasoning.

Through the inference engine (Pellet), the knowledge class and concept explanation in proposed ontology are translated into an appropriate format so that the ontology instances may go into operation. The reason is that the instance in ontology cannot be used when making real inferences about rules. In addition, the inference engine recognizes the conflicting and contradictory knowledge in ontology.

Step 4. *Knowledge Querying:*

The novel knowledge corresponding to the SWRL rules generated during the process of reasoning is stored in the working memory area temporarily. The now populated ontology is then queried with the SQWRL to identify which Java files belonging to a particular version of software component meet the query results. The change-prone Java files selected according to the SQWRL are displayed as results.

The empirical evaluation of the CPJFS framework has been discussed in later sections.

5.3 Semantic web preliminaries with respect to the proposed CPJFS mechanism

5.3.1 Construction of a Semantic web-based knowledge-base

The following sub-sections detail the *Steps 1* and *2* of the proposed CPJFS mechanism, mentioned in Section 5.2, that have been undertaken to construct the Semantic web-based knowledge-base in Protégé.

5.3.1.1 Empirical data acquisition & Pre-processing

In order to construct a Semantic web-based mechanism that selects change-prone Java files, we need a set of metrics or features that are most relevant or exclusive to the qualification of a Java file as change-prone. Therefore, elaborating on the premise established in Chapter 4, we employ the seven software metrics evaluated to be most efficient for change-proneness prediction of Java files contained in the selected versions of JFreeChart and Heritrix plugin projects in the proposed mechanism. As read in Section 2.1 of Chapter 2, a plugin is a ready to use software component that can be added to an existing software to add features [5]. The descriptive statistics of the seven metrics with respect to the selected versions of JFreeChart and Heritrix have been provided in the Tables 5.1.

As observed from the Table 5.1, the data gathered corresponding to the independent variables is numeric and therefore indicates quantity. However, we decided to transform the metrics to quality indicating categorical variables. This is because we needed to simplify the data in order to restrict the number of rules that are generated when a prediction technique is applied on this data [127]. Additionally, as observed from the data profiling carried out in Table 5.1, most of our independent variables lie in different numeric ranges. Therefore, in order to segregate them into similar categories, we need to normalize the variables within same data boundaries. Normalization changes the values of dataset attributes having numeric values to a common scale, without any altering the variances in the ranges of values.

The descriptive statistics of the data obtained after data normalization is given below in Tables 5.2 (a) and (b). Post normalization, we perform an unsupervised discretization of the datasets into five categories: Very Low (VL), Low (L), Medium (M), High (H), Very High (VH). An equal-frequency binning methodology is employed and the number of bins is fixed to five for each of the independent variables in each of the datasets using the Discretize unsupervised attribute filter in Weka 3.8.2. This normalized and discretized data has been employed to generate rules for classifying a Java file as change-prone (explained in later sections) and has been used to construct the proposed ontology. It is vital to note here that the bin edges vary for each of the attributes over each of the plugin project version's independent variables. For example, the bin edges for the normalized CE variable for Heritrix 0.2.0 created after the Discretize filter are as follows:

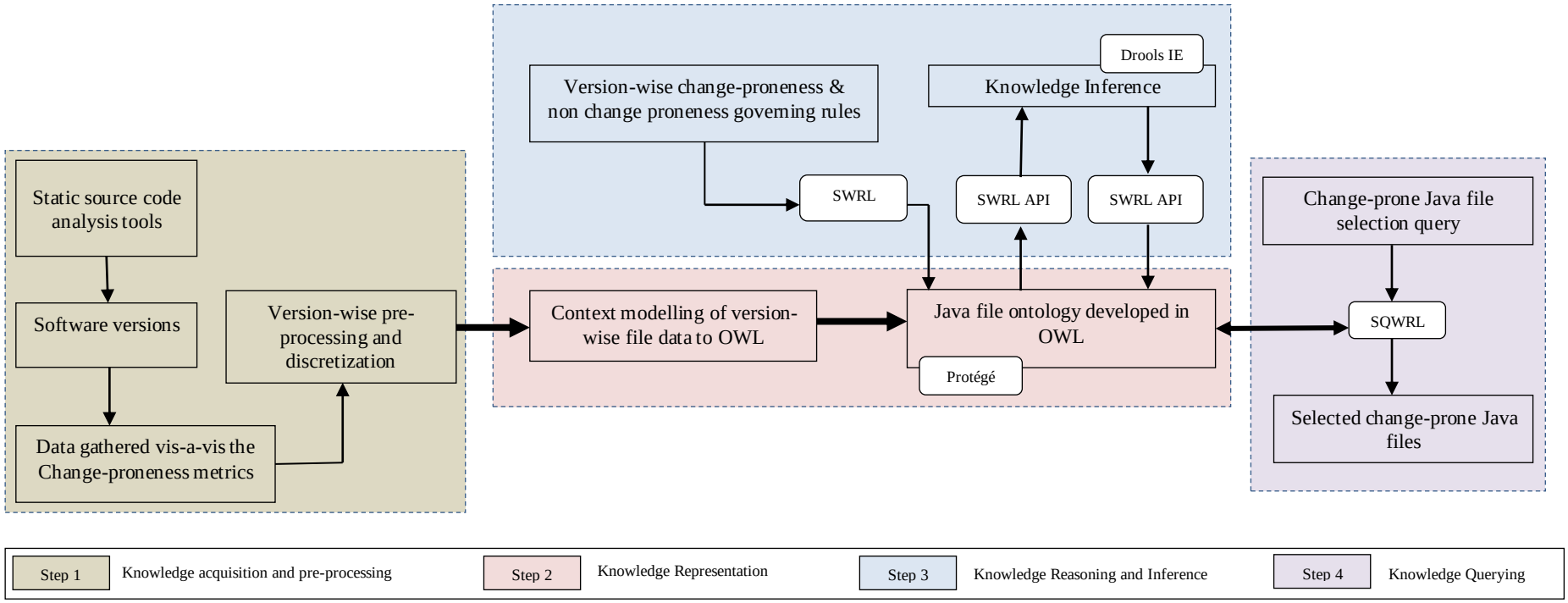

Figure 5.1 The flow of the Semantic web–based CPJFS procedure

Table 5.1 Descriptive statistics of the numerical values gathered corresponding to change-proneness software metrics for the selected versions of: (a) JFreeChart, and (b) Heritrix plugin projects

Table 5.1 (a) Metric statistics corresponding to JFreeChart versions

	JFreeChart 0.6.0				JFreeChart 0.7.0				JFreeChart 0.7.1				JFreeChart 0.7.2				JFreeChart 0.7.3			
	Min	Max	Avg.	Std.	Min	Max	Avg.	Std.	Min	Max	Avg.	Std.	Min	Max	Avg.	Std.	Min	Max	Avg.	Std.
SLOC	4	1281	100.8	155.4	4	1156	105.2	156.7	3	1296	100.7	160.7	3	1341	100.8	163.5	3	1331	101.6	163.7
CE	0	12	2.7	2.9	0	12	2.9	3.1	0	14	2.7	3.1	0	14	2.7	3.1	0	14	2.8	3.1
CHV	35.2	7334.2	3394.4	8211.9	35.2	99434.6	4019.8	11028.5	2	99434.6	3699.9	10411.9	2	102567.5	3659.6	10430.8	2	103482.5	3658.1	10467.9
CBO	0	31	5.4	5.4	0	40	5.9	6.4	0	41	5.6	6.5	0	39	5.5	6.2	0	39	5.9	6.2
DIT	0	5	1.5	1.1	0	6	1.7	1.4	0	6	1.5	1.3	0	6	1.5	1.6	0	6	1.8	1.3
RFC	1	47	10.6	10.9	1	47	11.4	11.8	0	51	10.6	12.9	0	56	10.6	12.4	0	56	10.7	12.5
COGC	1	3226	305.9	542.3	1	8274	431.2	1118.1	0	3024	264.7	509.9	0	4775	295.8	670.3	0	4025	282.2	593.5

Table 5.1 (b) Metric statistics corresponding to Heritrix versions

	Heritrix 0.2.0				Heritrix 0.4.0				Heritrix 0.6.0				Heritrix 0.8.0				Heritrix 0.10.0			
	Min	Max	Avg.	Std.	Min	Max	Avg.	Std.	Min	Max	Avg.	Std.	Min	Max	Avg.	Std.	Min	Max	Avg.	Std.
SLOC	3	950	79.9	124.2	3	778	96.9	107.7	3	949	97.1	119.3	3	926	103.1	126.6	3	1001	103.9	125.4
CE	0	14	2.4	2.9	0	28	2.68	3.5	0	31	2.76	3.7	0	29	2.9	3.6	0	19	2.6	3.2
CHV	2	65057.2	2879.9	7185.2	2	22514.3	2200.9	2783.9	2	25096.5	2214.6	3113.5	2	23780.1	2472.9	3466.5	2	87448.4	3051.9	6967.1
CBO	0	41	4.8	5.9	0	50	5.5	7.2	0	49	5.5	7.3	0	51	5.7	7.6	0	58	5.1	6.9
DIT	0	4	1.4	0.9	0	6	1.9	1.6	0	6	1.9	1.7	0	6	2.1	1.6	0	6	1.9	1.6
RFC	0	95	11.9	13.2	0	92	12.6	13.4	0	107	12.2	13.1	0	104	12.6	14.1	0	118	12.5	13.9
COGC	0	23555	609.2	2408.1	0	66792	1462.9	7442.5	0	77432	951.1	5758.5	0	8648	614.9	1451.3	0	10523	461.3	1090.6

Table 5.2 Descriptive statistics of the change-proneness software metrics after normalization for the selected versions of: (a) JFreeChart, and (b) Heritrix plugin projects

Table 5.2 (a) Normalized metric statistics corresponding to JFreeChart versions

	JFreeChart 0.6.0				JFreeChart 0.7.0				JFreeChart 0.7.1				JFreeChart 0.7.2				JFreeChart 0.7.3			
	Min	*Max*	*Avg.*	*Std.*	*Min*	*Max*	*Avg.*	*Std.*	*Min*	*Max*	*Avg.*	*Std.*	*Min*	*Max*	*Avg.*	*Std.*	*Min*	*Max*	*Avg.*	*Std.*
SLOC	0	1	0.08	0.12	0	1	0.09	0.14	0	1	0.08	0.13	0	1	0.07	0.12	0	1	0.07	0.12
CE	0	1	0.22	0.25	0	1	0.24	0.26	0	1	0.19	0.26	0	1	0.19	0.22	0	1	0.19	0.22
CHV	0	1	0.05	0.11	0	1	0.04	0.11	0	1	0.04	0.15	0	1	0.05	0.12	0	1	0.04	0.12
CBO	0	1	0.17	0.17	0	1	0.15	0.16	0	1	0.14	0.16	0	1	0.14	0.16	0	1	0.14	0.16
DIT	0	1	0.29	0.22	0	1	0.28	0.23	0	1	0.26	0.22	0	1	0.25	0.21	0	1	0.25	0.21
RFC	0	1	0.21	0.24	0	1	0.23	0.26	0	1	0.21	0.24	0	1	0.19	0.22	0	1	0.19	0.22
COGC	0	1	0.10	0.17	0	1	0.05	0.14	0	1	0.09	0.17	0	1	0.06	0.14	0	1	0.07	0.15

Table 5.2 (b) Normalized metric statistics corresponding to Heritrix versions

	Heritrix 0.2.0				Heritrix 0.4.0				Heritrix 0.6.0				Heritrix 0.8.0				Heritrix 0.10.0			
	Min	*Max*	*Avg.*	*Std.*	*Min*	*Max*	*Avg.*	*Std.*	*Min*	*Max*	*Avg.*	*Std.*	*Min*	*Max*	*Avg.*	*Std.*	*Min*	*Max*	*Avg.*	*Std.*
SLOC	0	1	0.08	0.13	0	1	0.12	0.14	0	1	0.09	0.13	0	1	0.10	0.14	0	1	0.10	0.13
CE	0	1	0.17	0.19	0	1	0.09	0.12	0	1	0.08	0.12	0	1	0.09	0.12	0	1	0.13	0.17
CHV	0	1	0.04	0.11	0	1	0.09	0.12	0	1	0.08	0.12	0	1	0.10	0.15	0	1	0.03	0.08
CBO	0	1	0.18	0.14	0	1	0.11	0.14	0	1	0.11	0.15	0	1	0.11	0.15	0	1	0.09	0.12
DIT	0	1	0.36	0.24	0	1	0.33	0.26	0	1	0.32	0.27	0	1	0.34	0.27	0	1	0.32	0.26
RFC	0	1	0.12	0.12	0	1	0.14	0.15	0	1	0.11	0.12	0	1	0.12	0.14	0	1	0.11	0.12
COGC	0	1	0.03	0.11	0	1	0.03	0.11	0	1	0.01	0.07	0	1	0.07	0.17	0	1	0.04	0.10

- Very Low (VL) if CE > = 0 and <0.035
- Low (L) if CE >= 0.035 and <0.1
- Medium (M) if CE >= 0.1 and <0.2
- High (H) if CE >= 0.2 and <0.3
- Very High (VH) if CE >=0.3

Whereas, for the normalized CE variable for Heritrix 0.4.0 the bin edges are created in the manner like so:

- Very Low (VL) if CE > = 0 and <0.02
- Low (L) if CE >= 0.035 and <0.05
- Medium (M) if CE >= 0.05 and <0.1
- High (H) if CE >= 0.1 and <0.2
- Very High (VH) if CE >=0.2

This is primarily because the descriptive statistics of the independent variables in the version datasets differ in terms of mean and standard deviation over versions and across variables. Therefore we do not employ a standardized method for discretizing the normalized independent variables that is common for all the variables over all the selected version datasets and instead opt for bin edges specific to the independent variable for its discretization into five categories.

5.3.1.2 Context Modelling in terms of Ontology (Knowledge Representation)

The seven static source code metrics gathered corresponding to every Java file form a basis of the ontology to support change-prone Java file selection. The ontologies have been built using the Protégé application in this empirical work. Developed at Stanford University's Information Center, Protégé [126] is used to connect, create and maintain ontology. RDF (Resource Description Framework) follows Protégé. RDF explains the relationship between web pages and other tools and recognizes the key ontological components: classes and attributes [218]. Class stands for the concept of knowledge. Attribute describes the features and association of a class and is linked to a basic data type. The relationship can link another Instance or Class.

Data has been described using classes, domain, attributes, and scopes. Moreover, since Protégé is written in Java programming language, the expansibility of the ontology developed is unlimited.

(a) Ontology Classes and their instances

In ontology, classes are a concrete representation of concepts [118, 219]. The ontology constructed in this chapter is composed of three classes depicting concepts of suitable contextual information that are essential for selecting a change-prone Java file. These classes are; "Java Files" class, "Change Prone Files" class and "Not Change Prone Files" class. The “Change Prone Files” class and the "Not Change Prone Files" class have been included explicitly as classes in the ontology primarily to aid in the classification of a file as change-prone or not change-prone. The classes have been explained below:

- The *Java Files class* represents all the Java files that are present in the individual software component versions and has its subclass as the software component name whose versions are being analysed. Each of the Java file within a software component is represented as an instance via a unique File Code that further describes features like file name, file type, software version that it belongs to, structural attributes like CBO, CE, CogC, etc.
- The *Change Prone Files class* represents those Java files that have been selected as change-prone according to the individual SWRL rules. These rules exhibit the conditions defined by the selected prediction technique. Each of the “change-proneness” governing rule obtained from the prediction technique is named as an instance exhibiting the software component, nature of the rule (change-prone or not change-prone), and rule number. Each of the Change Prone Files class consists of sub-classes indicating the version of the software component that the rules belong to.

  For example, the instance of the Change Prone Files class expressing Rule 1 for change-proneness of a JFreeChart(JFC) plugin project version 0.6.0 would be labelled as : JFC_CP_Rule1.
- The *Not Change Prone Files class,* as the name suggests, represents those Java files that have been selected as stable or not change-prone according to the individual SWRL rules. The structure of the Not Change Prone Files class is similar to that of the Change Prone Files class except the rules pertain to classifying a Java file as not change-prone. Again, each of the Not Change Prone Files class consists of sub-classes indicating the software component version of that the rules belong to.

Taking the example given above, for a version of the JFreeChart (JFC) plugin project, the instance of the Not Change Prone Files class would be labelled as JFC_NCP_Rule1.

After the rules are fired and the reasoner is synchronized, the instances of *Change Prone Files class* and the *Not Change Prone Files class* will consist of all those Java files that have satisfied the change-proneness and not change-proneness governing rules of the selected software version respectively.

(b) Ontology Relationships

Relationships depict the communication amongst the concepts in the ontology [118, 219]. They are defined by the attributes and properties that a class has. "Object properties" are the relationships existing between classes wherein each object property comprises of a source and target. Table 5.3 states the object properties along with their source and target classes used in the proposed ontology. Some key points to be noted here is that none of these properties have sub-properties and both the mentioned object properties are inferred using SWRL rules and the Drools inference engine. However, any sub-classes of a Domain class for an Object Property, also become a Domain for that Object Property [219].

Attributes characterizing the classes or class instances are known as "data type properties" [219]. They define the association between class instances (individuals) and data values. Table 5.4 states the twelve data type properties included in our analysis, the class they belong to and the data type they have. Data type properties, like object properties, can too have sub-properties. However, none of the data type properties employed in our analysis have a sub-property. Ten object properties belong to class Java Files and primarily aid in describing features related to a Java file such as its name, the version it belongs to, the file type, and the values of the seven software metrics. The remaining two properties chiefly aid in the version-wise classification of the rule instances. More is explained related to this in Section 5.4 of this chapter wherein the experimentation is performed on actual software datasets.

Table 5.3 Object Properties included for the selection of change-prone Java files

Object Property	Domain	Range
hasCPFilesAs	Change Prone Files	Java Files
hasNCPFilesAs	Not Change Prone Files	Java Files

Table 5.4 Data Type Properties included for the selection of change-prone Java files

Class	Data Type Property	Data Type
Java Files	BelongsToVersion	String
	HasFileName	
	HasFileType	
	HasSLOC	
	HasCE	
	HasCHV	
	HasCBO	
	HasDIT	
	HasRFC	
	HasCOGC	
Change Prone Files	hasCPRuleName_Number	
Not Change Prone Files	hasNCPRuleName_Number	

(c) Ontology Language

There are many languages [123] available in the literature to represent or express ontologies, including OWL (Web Ontology Language), DAML+OIL, Resource Description Framework Schema (RDFS) etc. As stated in the Section 2.4 of Chapter 2, OWL is one of the primary technologies of Semantic web and was proposed by the Web Ontology Working Group of W3C [123]. OWL is an ontology language that includes all the required constructors to systematically define almost all of the concepts of information management such as classes and property, (with hierarchies), scope and domain restrictions; and therefore in this work OWL language has been incorporated for describing the ontology vis-à-vis the Java files.

5.3.2 Context knowledge reasoning for novel knowledge inference and querying for the selection of change-prone Java files

The main objective of constructing the ontology corresponding to the Java files is not only to represent useful contextual information with respect to version to version change-proneness of Java files of software components, but also to enable reasoning on this contextual information via inferential rules in the form of "if...then...", as mentioned in *Step 3* and *4* of Section 5.2.

5.3.2.1 Novel knowledge inference using SWRL rules

Inference rules are a set of rules that define a general mechanism, based on existing ones, for an automatic discovery and generation of novel associations between concepts [220]. As per our context of application, the rules employed have been segregated into two categories: Change Proneness rules and Not Change Proneness rules.

- *Change Proneness rules* represent those rules that aid in classifying a particular Java source code files as change-prone in the forthcoming version, on the basis of its properties specified in the ontology.

  Example 1: *IF* (JavaFile hasCBO equal to "H") *THEN* (JavaFile is Change Prone).

  Example 2: *IF* (JavaFile hasCBO equal to "VH") AND *IF* (JavaFile hasRFC equal to "VH") *THEN* (JavaFile is Change Prone).
- *Not Change Proneness rules* represent those rules that aid in classifying a particular Java source code files as not change-prone in the forthcoming version, on the basis of its properties specified in the ontology.

  Example 3: *IF* (JavaFile hasCBO equal to "L") *THEN* (JavaFile is Not Change Prone).

  Example 4: *IF* (JavaFile hasCBO equal to "VL" AND *IF (*JavaFile hasRFC equal to "L") *THEN* (JavaFile is Not Change Prone).

As mentioned earlier, we incorporate a prediction technique to define the rules pertaining to the change-proneness and stability of Java files belonging to a software component release with respect to the forthcoming version. We need to express these rules in a semantic language in order to apply them to our ontology. The Semantic Web Rule Language (SWRL) [83] is a language proposed for the Semantic web that can be used to express rules and logic, combining OWL with a Rule Markup Language sub-set. SWRL [83, 221] expands the OWL axioms set by including conditional rules (Horn clauses), in the form of "if...then...". A rule axiom comprises of an antecedent and a consequent. A rule expressed using SWRL's "human readable" syntax has the form: antecedent ⇒ consequent and can be interpreted as: if the antecedent is true, then the consequent must also be true.

SWRL has been employed in this work to capture the full semantic richness of OWL and construct appropriate rules to aid in the selection of change-prone Java files. As mentioned in the previous sub-sections, Protégé has been employed to construct the proposed ontology for Java files. The development environment 'Protégé-OWL' facilitates the construction of SWRL rules via the SWRLTab [222].

SWRL rules can only be used to add new information to an ontology and not modify existing information in an ontology. The SWRLTab [222] comprises of the following (1) an *editor* supporting an interactive creation of SWRL rules, (2) a *rule engine* bridge providing the set-up required to include third-party rule engines in the SWRLTab for the execution SWRL rules, and (3) a set of *built-in libraries* comprising of temporal, string, and mathematical operators.

Suppose the rule stated in Example 1 is the first change-proneness governing rule generated from JFreeChart 0.6.0 and we want to extract the Java files belonging to JFreeChart version 0.7.0 for change-proneness with respect to it. The SWRL rule when constructed in the SWRLTab would be represented in the following manner:

```
JavaFile(?f)^ belongsToVersion(?f,"JFreeChart0.7.0")^
ChangeProneFiles(?cp)^ hasCPRuleName_Number(?cp,"JFC0.6.0_CP_Rule1")
^ hasCBO(?f,"H")-> hasCPFilesAs(?cp,?f)
```

Once the rule has been evaluated with the reasoner and the inference engine, the instance of the ChangeProneFiles that has the rule name and number "JFC0.6.0_CP_Rule1" will consist of all the Java File instances that belong to version JFreeChart0.7.0 and have CBO value equal to "H".

5.3.2.2 Selection of change-prone software components using SQWRL

In order to select the change-prone Java files from a software version, querying of the new knowledge added to the ontology by the SWRL rules is done, as mentioned *in Step 4* of Section 5.2. The SWRL can be extended to **S**emantic **Q**uery-enhanced **W**eb **R**ule **L**anguage (SQWRL) [129] that contains a set of built-in libraries allowing it to be used as a query language. SQWRL enables the SWRL rules to be employed as a means to query OWL based ontologies. It comprises of SQL-like built-ins that can be added to a rule to create selection and retrieval actions. As an example, the query stated below selects and displays all Java Files and their version in an ontology whose CBO value is equal to "H:

```
JavaFile(?f)^ belongsToVersion(?f,?v)^ hasCBO(?f,"H")->
sqwrl:select(?f,?v)
```

Thusly, as observed, the ontology proposed in this chapter is not complex to implement and can include all the contextual information that is required. Additionally, the ontology proposed and rules-model is scalable since there can be an easy addition of novel contextual information and reasoning on this newly added information. For instance, in a case where new software project version's Java files is added to the reuse repository, we can incorporate this information by just altering the ontology structure and forming new rules corresponding to this information.

5.4 Experiment and Discussion

The experimental data employed in our analysis has been collected from successively released versions of five JFreeChart and five Heritrix plugin projects. Albeit identical in structure, separate ontologies have been developed for the Java files of the two projects for simplification in evaluation. Each of the developed ontologies has three classes, two object properties and twelve variables from the twelve data properties. However, the ontology constructed with respect to the Java files of the JFreeChart versions has a total of 98 rules and 570 Java files. Whereas the ontology constructed with respect to the Java files of the Heritrix versions has a total of 92 rules and 907 Java files.

Although all the JFreeChart and Heritrix versions selected have been evaluated to validate the proposed Semantic web-based CPJFS mechanism, the Protégé result screenshots have been included in Appendices E and F for two versions each from the Heritrix software project so as to avoid repetition of content. However, only Protégé result screenshots with respect to JFreeChart version 0.7.0 have been employed in Sections 5.4.1 through 5.4.3 and with respect to JFreeChart version 0.7.1 in Section 5.4.4 to detail the ontological modelling of the empirical data, explain the construction of relevant SWRL rules in the SWRLTab, and to build SQWRL queries for change-prone Java file selection.

Since there are five releases for each of the JFreeChart and Heritrix plugin projects, four versions from each software projects are used to train and validate the proposed mechanism. This is because there is no version to train the first version of each project and no version to validate the proposed mechanism using the last release.

5.4.1 Ontological modelling of the empirical data

Figures 5.2 and 5.3 show the Protégé OWL ontology editor for the JFreeChart versions, the "class hierarchy" presents the three classes mentioned in Section 5.3.1.2: JavaFiles, ChangeProneFiles, and NotChangeProneFiles class. The root node of the ontology is ''owl: Thing'' The child nodes of root node are JavaFiles, ChangeProneFiles, and NotChangeProneFiles class which build the individual categories. The sub-categories of these classes include sub-classes and instances/ individuals.

- The *JavaFiles* class consists of the JFreeChart subclass which further is composed of individuals that represent each of the Java files of the selected JFreeChart version. Each of the individual is named in the following manner: SoftwareNameAndVersion_FileNumber. For example, as seen in Figure 5.2, individual representing the 75th Java file analysed in the JFreeChart plugin project version JFreeChart0.7.3 is named as JFreeChart0.7.3_75.

  Also, as seen in Figure 5.2, the individual file data discretized with respect to the seven independent variables is added to each of these individual in the form of the 12 object properties stated in Table 5.4 of Section 5.3.1.2. These object properties, as stated before, describe the features of the Java file with respect to change-proneness.
- The *ChangeProneFiles* class has subclasses and individuals that get populated every time a relevant rule is fired. These subclasses and individuals are named in a manner that indicates the target projects' version, the nature of rule and the rule number. Each of these subclasses further has a data property called as the *hasCPRuleName_Number* that indicates the version and number of the rule.
- The *NotChangeProneFiles* class, similar to the ChangeProneFiles class, has subclasses that get populated every time a relevant rule is fired. These subclasses and individuals are named in a manner that indicates the target projects' version, the nature of rule and the rule number. Each of these subclasses further has a data property called as the *hasNCPRuleName_Number* that indicates the version and number of the rule.

For example, as observed in Figure 5.3, the subclass of the NotChangeProneFiles class consisting of the Java files that are updated according to the rules of JFreeChart 0.6.0 is named as NotChangeProneFilesJFC060. This subclass further consists of individual that has a value to data property *hasNCPRuleName_Number* as "JFreeChart0.6.0_NCP_Rule6", which gets updated with the appropriate Java files every time the sixth Not Change-Proneness (NCP) governing rule of JFreeChart0.6.0 is fired.

The properties *hasCPRuleName_Number and hasNCPRuleName_Number* have been included only for aiding in the change-prone classification procedure.

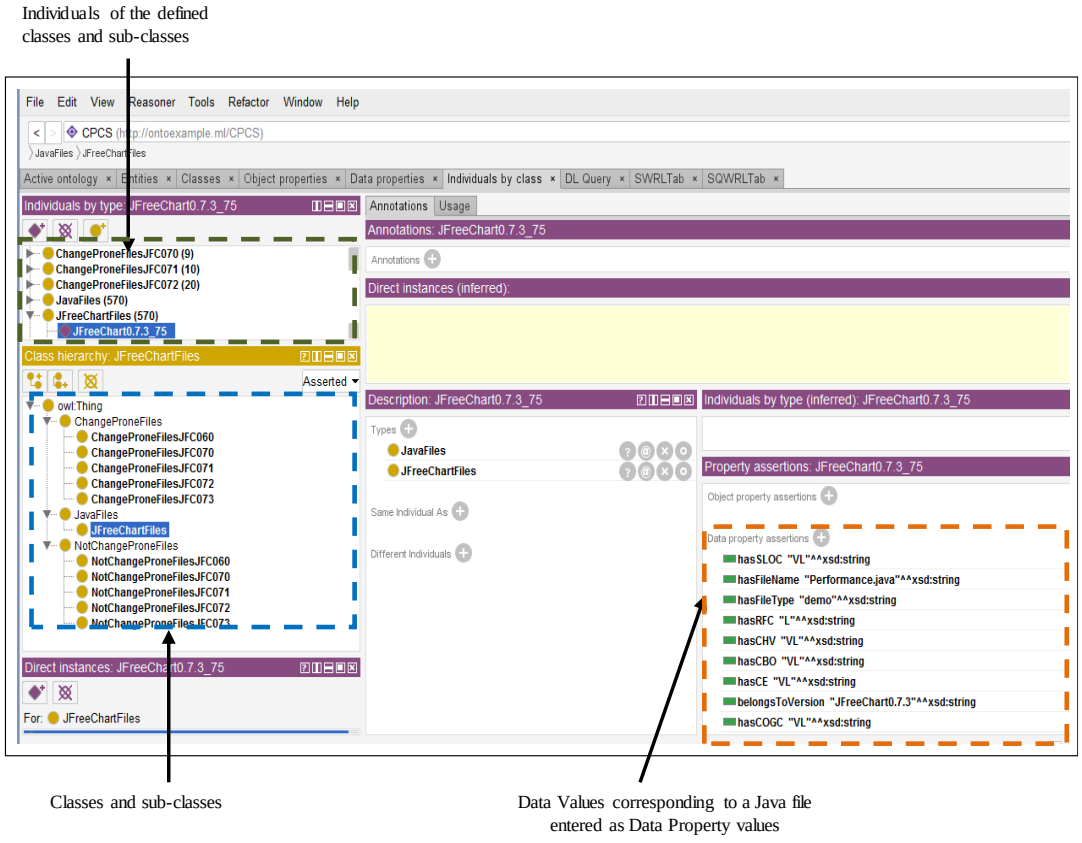

Figure 5.2 The developed OWL ontology for JFreeChart Java files in Protégé editor

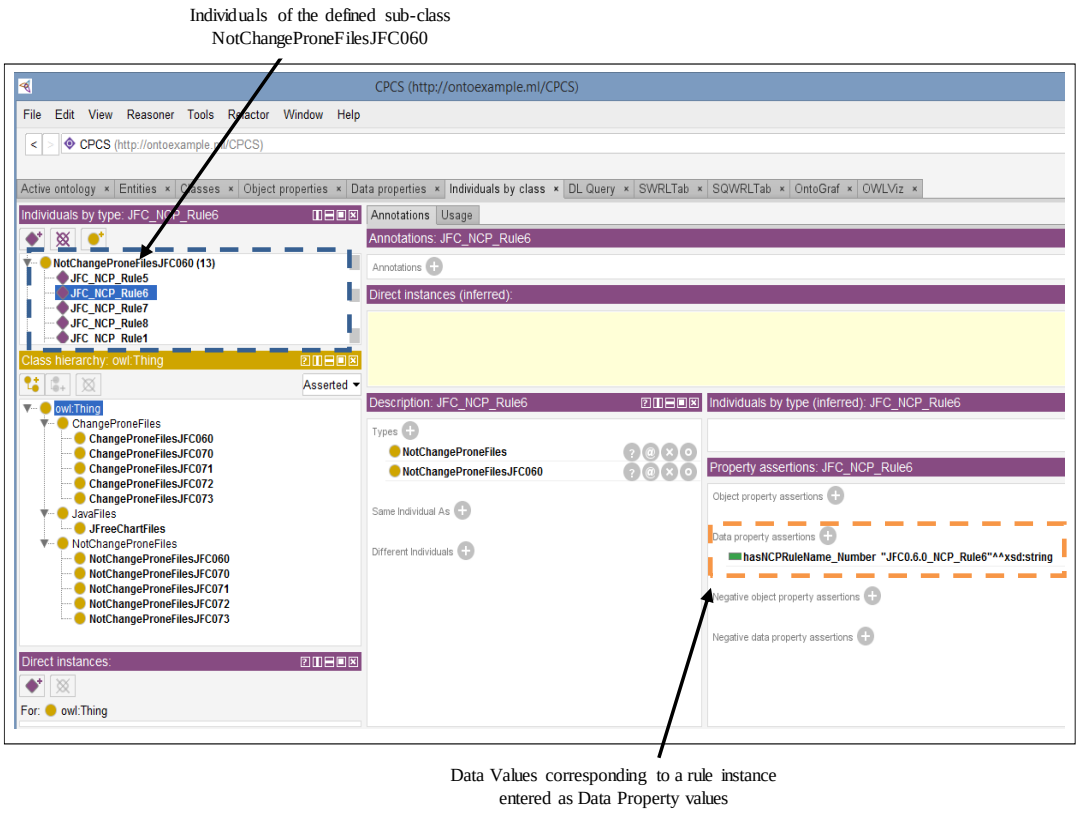

Figure 5.3 The developed OWL ontology indicating data property assertion for rules

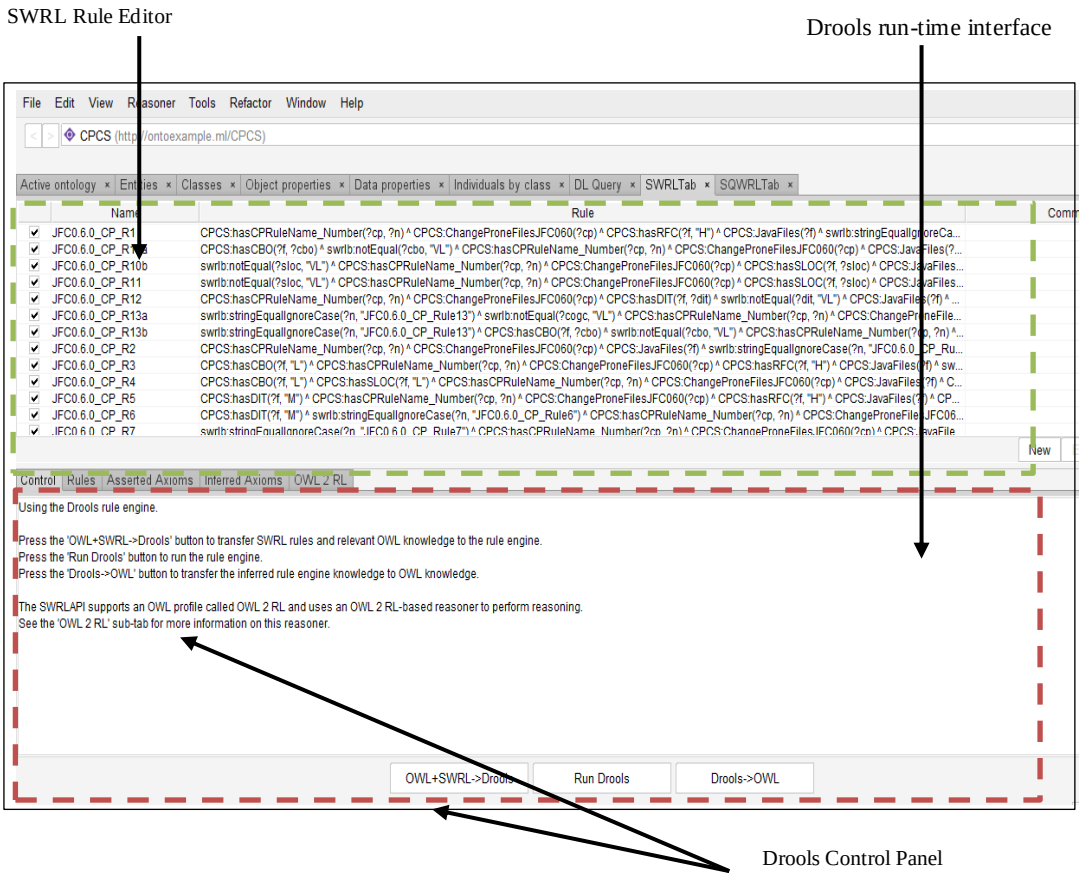

Figure 5.4 The SWRLTab containing rules and the Drools connections in Protégé

The proposed CPJFS mechanism returns names for those Java files of a project that would be used with change in the subsequent release of the project. As stated earlier, the ‘‘SWRLTab’’ [222] plug-in in Protégé enables the creation of the SWRL rules on top of OWL ontology. Along with SWRL, our study incorporated the Drools Rule Engine, also offered with Protégé, to execute the rule inference. Figure 5.4 shows the SWRL operation interface, in which the SWRL rule editor and the Drools run-time interface have been expressed via the two dashed rectangles. The rule editor allows users to add rules derived from the prediction technique in the form of SWRL, as shown in the editor. These rules are triggered or fired every time the Drools rule engine is invoked.

5.4.2 Classifying the components using the rule-base

Semantic web is one of the widely adopted standards of knowledge representation. Nevertheless, knowledge engineers are always looking for sophisticated methods to discover and represent knowledge in a Semantic Web form.

Recent studies [223-225] show that prediction techniques such as Association rules, Decision Trees have a vital role in gathering, depicting and augmenting existing knowledge in Semantic web technologies. These techniques are available on various publicly accessible data mining applications like Rapid Miner, WEKA, R, etc. in a humanly readable format for the purpose of data mining and information exploration.

Tools like Weka even employ Predictive Model Markup language (PMML) to represent knowledge discovery. PMML represents a formalized syntax of knowledge mining. This language, however, does not address the depiction of the semantics of the knowledge discovered [226]. Consequently, there is a shortage of resources to tackle the translation of inferences in suitable Semantic web formats such as OWL. Although one might attempt to break down steps of converting XML to OWL, there is a dearth of information that makes the process all the more complicated [227].

A prediction technique learns from past knowledge, reuses this knowledge to solve new problems and also builds model for the same for future use. As per the analysis conducted in the previous chapter, the RF technique is found to be the most effective ICM for the prediction of version to version change proneness of Java files of the various JFreeChart and Heritrix plugin projects among the various analysed prediction techniques, both during k-fold and an inter-release validation.

The OWL ontology CPJFS developed till now (as per Section 5.4.1) merely represents facts in regard to the various Java files of the selected JFreeChart and Heritrix versions. Therefore, in order to select the relevant change-prone Java files, the rules generated by the RF technique with respect to each version analysed for change-proneness prediction are employed as SWRL rules in the ontology. These rules when applied on the Java files instances of a software version essentially infer the relevant instances according to the given rules, thereby classifying the instances as change-prone or not change-prone according to the individual rules. The ontology is then queried for the selection of change-prone Java files of the given version.

Note: The SWRL rules derived from the RF algorithm are employed in the ontology in an incremental manner for selection of change-prone Java files. That is, the SWRL rules with respect to change-proneness prediction generated till Version $_{i-1}$ are employed, if one needs to select the Java files of a Version $_{i}$ that will be used with change in Version $_{i+1}$. This will ensure that the rule-base in the ontology always remains updated with the most recent rules with respect to version to version change-proneness prediction of Java files.

5.4.2.1 Extraction of most accurate change-proneness prediction rules from the ML technique generated rules

We constructed change-proneness prediction models using the Random Forest (RF) technique on each of the ten plugin versions, the results of which in terms of Sensitivity, Specificity and Accuracy are provided in Tables 5.5 and 5.6. These results were generated after applying the RF technique on the ten datasets obatined after the normalization and discretization of the seven independent variables as explained in Section 5.3.1.1. The prediction models are generated using Weka 3.8.2 and although the rest of the model settings are kept as default, the numbers of trees vary for each of the software project versions. The optimal number of trees that the RF model should build corresponding to each version which yields maximum accuracy possible were found out by hit and trial and varied for each version.

Table 5.5 RF results for the JFreeChart versions

	JFreeChart 0.6.0	**JFreeChart 0.7.0**	**JFreeChart 0.7.1**	**JFreeChart 0.7.2**	**JFreeChart 0.7.3**
Sensitivity	88.0	74.4	75.7	95.7	80.6
Specificity	84.5	82.4	86.4	89.3	88.8
Accuracy	86.2	78.9	84.2	92.3	85.3

Table 5.6 RF results for the Heritrix versions

	Heritrix 0.2.0	Heritrix 0.4.0	Heritrix 0.6.0	Heritrix 0.8.0	Heritrix 0.10.0
Sensitivity	70.6	75.9	67.6	60.5	65.5
Specificity	79.5	82.0	91.3	74.4	85.8
Accuracy	75.3	78.9	81.9	68.1	80.5

Post generating prediction models with respect to each plugin project version, we formulated rules out of the decision trees that were constructed by the RF technique for each of the version. We utilized the methodology proposed by Mitchell [228, p.72] that states:

> *"To generate the rules, trace each path in the decision tree from root node to leaf node and record the test outcomes as antecedents and the leaf-node classification as the consequent."*

Additionally, Mitchell [228] pointed out that conversion of a decision tree to rules before pruning has three main advantages:

- Transforming to rules makes it possible to differentiate between the various contexts where a decision node will be used.
- Dissimilar to trees, there is no distinction between the attribute tests occuring near the tree's root with that of happening near the tree leaves.
- Reading and understanding rules is easier.

Columns 2, 3 and 4 of Table 5.7 indicate the total number of rules, number of change-prone rules and not-change-prone rules that are generated for each of the five JFreeChart and Heritrix plugin project versions. The individual 10 datasets were balanced, normalized and discretized as explained earlier in Section 5.3.1.1, before they were analysed for change-proneness prediction via the RF technique.

As observed, a large number of rules corresponding to every version are generated. However, instead of including all the rules in the proposed ontology for Java files, rules derived from RF are further screened according to their individual strength of accuracy. The following steps have been incorporated to extract the most accurate rules:

- Execute every rule individually on the dataset. If the rule satisfies for a particular Java file then an appropriate change statistic of "yes" or "no" is entered.

- Match the outcome of the execution of the rule with the target variable of each Java file. If the target variable corresponding to a file is equal to the change statistic evaluated by the individual rule, then it is considered to be an accurate prediction by the rule for that specific file.
- Evaluate the overall accuracy of individual rule using the formula:

$$\frac{number\ of\ files\ accurately\ predicted\ by\ the\ rule}{total\ number\ of\ Java\ files\ in\ the\ dataset}$$

Only those rules that have rule strength higher than 60% have been included in the proposed ontology.

Table 5.7 RF rule statistics

Version	Statistics of the rules generated from RF			Statistics of the rules shortlisted from RF		
	No. of rules	No. of Change-Proneness governing rules	No. of not Change-Proneness governing rules	Number of rules shortlisted	No. of Change-Proneness governing rules	No. of not Change-Proneness governing rules
JFreeChart0.6.0	57	31	26	13	7	6
JFreeChart0.7.0	32	22	10	9	4	5
JFreeChart0.7.1	47	25	22	10	5	5
JFreeChart0.7.2	59	23	36	17	10	7
JFreeChart0.7.3	40	20	20	10	5	5
Heritrix0.2.0	113	70	43	16	5	11
Heritrix0.4.0	127	61	66	10	5	5
Heritrix0.6.0	160	95	65	10	4	6
Heritrix0.8.0	143	83	60	10	9	1
Heritrix0.10.0	162	97	65	8	0	8

5.4.2.2 Modelling of the extracted change-proneness prediction rules to SWRL rules

As earlier mentioned, the most accurate rules extracted from the RF technique are written in SWRL using the SWRLTab extension to Protégé. The rules essentially indicate the following: Which of the Java files of a software component version are change-prone according to the values of their attributes? As observed from column 5 in Table 5.7, 13 rules are extracted with respect to JFreeChart 0.6.0. However, it is also observed that six of these rules also are actually those pertaining to non-change-

proneness of a Java file. Since for an effective change-proneness selection mechanism it is required that the prospect of a file being not change prone is also evaluated, therefore for every such rule extracted, its inverse rule is also included and entered as a SWRL rule for that version.

Therefore, the 13 rules with respect to JFreeChart0.6.0 now are a total of 26 rules that form an equal number of change-prone as well as not change-prone rules, essentially signifying that initially there are equal prospects of a Java file to be used in the successive version with or without change. The final 26 SWRL rules corresponding to the JFreeChart 0.6.0 are given in Tables 5.8 and 5.9 below.

Table 5.8 Rules entered corresponding to JFreeChart 0.6.0 in order to identify change prone files of JFreeChart 0.7.0

Rule Name	JFreeChart 0.6.0 CP governing rules
JFC0.6.0_CP_R1	CPCS:hasCPRuleName_Number(?cp, ?n) ^ CPCS:ChangeProneFilesJFC060(?cp) ^ CPCS:hasRFC(?f, "H") ^ CPCS:JavaFiles(?f) ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.6.0_CP_Rule1") ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.0") -> CPCS:hasCPFilesAs(?cp, ?f)
JFC0.6.0_CP_R2	CPCS:hasCPRuleName_Number(?cp,?n) ^ CPCS:ChangeProneFilesJFC060(?cp) ^ CPCS:JavaFiles(?f) ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.6.0_CP_Rule2") ^ CPCS:hasRFC(?f,"VH") ^ CPCS:belongsToVersion(?f,"JFreeChart0.7.0") -> CPCS:hasCPFilesAs(?cp, ?f)
JFC0.6.0_CP_R3	CPCS:hasCBO(?f, "L")^ CPCS:hasCPRuleName_Number(?cp, ?n) ^ CPCS:ChangeProneFilesJFC060(?cp) ^ CPCS:hasRFC(?f,"H") ^ CPCS:JavaFiles(?f) ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.6.0_CP_Rule3") ^ CPCS:belongsToVersion(?f,"JFreeChart0.7.0") -> CPCS:hasCPFilesAs(?cp, ?f)
JFC0.6.0_CP_R4	CPCS:hasCBO(?f, "L") ^ CPCS:hasSLOC(?f, "L") ^ CPCS:hasCPRuleName_Number(?cp, ?n) ^ CPCS:ChangeProneFilesJFC060(?cp) ^ CPCS:JavaFiles(?f) ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.0") ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.6.0_CP_Rule4") -> CPCS:hasCPFilesAs(?cp, ?f)
JFC0.6.0_CP_R5	CPCS:hasDIT(?f, "M") ^ CPCS:hasCPRuleName_Number(?cp, ?n) ^ CPCS:ChangeProneFilesJFC060(?cp) ^ CPCS:hasRFC(?f, "H") ^ CPCS:JavaFiles(?f) ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.0") ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.6.0_CP_Rule5") -> CPCS:hasCPFilesAs(?cp, ?f)
JFC0.6.0_CP_R6	CPCS:hasDIT(?f, "M") ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.6.0_CP_Rule6") ^ CPCS:hasCPRuleName_Number(?cp, ?n) ^ CPCS:ChangeProneFilesJFC060(?cp) ^ CPCS:JavaFiles(?f) ^ CPCS:hasRFC(?f, "VH") ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.0") -> CPCS:hasCPFilesAs(?cp, ?f)
JFC0.6.0_CP_R7	swrlb:stringEqualIgnoreCase(?n, "JFC0.6.0_CP_Rule7") ^ CPCS:hasCPRuleName_Number(?cp, ?n) ^ CPCS:ChangeProneFilesJFC060(?cp) ^ CPCS:JavaFiles(?f) ^ CPCS:hasCOGC(?f, "L") ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.0") -> CPCS:hasCPFilesAs(?cp, ?f)
	(a)

JFC0.6.0_CP_R8*	swrlb:notEqual(?rfc, "VL") ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.6.0_CP_Rule8") ^ CPCS:hasCPRuleName_Number(?cp, ?n) ^ CPCS:ChangeProneFilesJFC060(?cp) ^ CPCS:hasRFC(?f, ?rfc) ^ CPCS:JavaFiles(?f) ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.0") -> CPCS:hasCPFilesAs(?cp, ?f)
	(b) swrlb:stringEqualIgnoreCase(?n, "JFC0.6.0_CP_Rule8") ^ CPCS:hasCPRuleName_Number(?cp, ?n) ^ CPCS:ChangeProneFilesJFC060(?cp) ^ CPCS:hasDIT(?f, ?dit) ^ swrlb:notEqual(?dit, "VL") ^ CPCS:JavaFiles(?f) ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.0") -> CPCS:hasCPFilesAs(?cp, ?f)
JFC0.6.0_CP_R9*	swrlb:stringEqualIgnoreCase(?n, "JFC0.6.0_CP_Rule9") ^ swrlb:notEqual(?rfc, "L") ^ CPCS:hasCPRuleName_Number(?cp, ?n) ^ CPCS:ChangeProneFilesJFC060(?cp) ^ CPCS:hasRFC(?f, ?rfc) ^ CPCS:JavaFiles(?f) ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.0") -> CPCS:hasCPFilesAs(?cp, ?f)
JFC0.6.0_CP_R10*	(a) CPCS:hasCBO(?f, ?cbo) ^ swrlb:notEqual(?cbo, "VL") ^ CPCS:hasCPRuleName_Number(?cp, ?n) ^ CPCS:ChangeProneFilesJFC060(?cp) ^ CPCS:JavaFiles(?f) ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.6.0_CP_Rule10") ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.0") -> CPCS:hasCPFilesAs(?cp, ?f)
	(b) swrlb:notEqual(?sloc, "VL") ^ CPCS:hasCPRuleName_Number(?cp, ?n) ^ CPCS:ChangeProneFilesJFC060(?cp) ^ CPCS:hasSLOC(?f, ?sloc) ^ CPCS:JavaFiles(?f) ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.6.0_CP_Rule10") ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.0") -> CPCS:hasCPFilesAs(?cp, ?f)
JFC0.6.0_CP_R11*	swrlb:notEqual(?sloc, "VL") ^ CPCS:hasCPRuleName_Number(?cp, ?n) ^ CPCS:ChangeProneFilesJFC060(?cp) ^ CPCS:hasSLOC(?f, ?sloc) ^ CPCS:JavaFiles(?f) ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.0") ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.6.0_CP_Rule11") -> CPCS:hasCPFilesAs(?cp, ?f)
JFC0.6.0_CP_R12*	CPCS:hasCPRuleName_Number(?cp, ?n) ^ CPCS:ChangeProneFilesJFC060(?cp) ^ CPCS:hasDIT(?f, ?dit) ^ swrlb:notEqual(?dit, "VL") ^ CPCS:JavaFiles(?f) ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.0") ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.6.0_CP_Rule12") -> CPCS:hasCPFilesAs(?cp, ?f)
JFC0.6.0_CP_R13*	(a) swrlb:stringEqualIgnoreCase(?n, "JFC0.6.0_CP_Rule13") ^ swrlb:notEqual(?cogc, "VL") ^ CPCS:hasCPRuleName_Number(?cp, ?n) ^ CPCS:ChangeProneFilesJFC060(?cp) ^ CPCS:JavaFiles(?f) ^ CPCS:hasCOGC(?f, ?cogc) ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.0") -> CPCS:hasCPFilesAs(?cp, ?f)
	(b) swrlb:stringEqualIgnoreCase(?n, "JFC0.6.0_CP_Rule13") ^ CPCS:hasCBO(?f, ?cbo) ^ swrlb:notEqual(?cbo, "VL") ^ CPCS:hasCPRuleName_Number(?cp, ?n) ^ CPCS:ChangeProneFilesJFC060(?cp) ^ CPCS:JavaFiles(?f) ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.0") -> CPCS:hasCPFilesAs(?cp, ?f)

Table 5.9 Rules entered corresponding to JFreeChart 0.6.0 in order to identify non-change prone files of JFreeChart 0.7.0

Rule Name	**JFreeChart 0.6.0 NCP governing rules**
JFC0.6.0_NCP_R1*	swrlb:notEqual(?rfc, "H") ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.6.0_NCP_Rule1") ^ CPCS:hasNCPRuleName_Number(?ncp, ?n) ^ CPCS:hasRFC(?f, ?rfc) ^ CPCS:JavaFiles(?f) ^ CPCS:NotChangeProneFilesJFC060(?ncp) ^

	CPCS:belongsToVersion(?f, "JFreeChart0.7.0") -> CPCS:hasNCPFilesAs(?ncp, ?f)
JFC0.6.0_NCP_R2*	swrlb:stringEqualIgnoreCase(?n, "JFC0.6.0_NCP_Rule2") ^ CPCS:hasNCPRuleName_Number(?ncp, ?n) ^ CPCS:hasRFC(?f, ?rfc) ^ CPCS:JavaFiles(?f) ^ CPCS:NotChangeProneFilesJFC060(?ncp) ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.0") ^ swrlb:notEqual(?rfc, "VH") -> CPCS:hasNCPFilesAs(?ncp, ?f)
JFC0.6.0_NCP_R3*	(a) CPCS:hasCBO(?f, ?cbo) ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.6.0_NCP_Rule3") ^ CPCS:hasNCPRuleName_Number(?ncp, ?n) ^ CPCS:JavaFiles(?f) ^ CPCS:NotChangeProneFilesJFC060(?ncp) ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.0") ^ swrlb:notEqual(?cbo, "L") -> CPCS:hasNCPFilesAs(?ncp, ?f)
	(b) swrlb:notEqual(?rfc, "H") ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.6.0_NCP_Rule3") ^ CPCS:hasNCPRuleName_Number(?ncp, ?n) ^ CPCS:hasRFC(?f, ?rfc) ^ CPCS:JavaFiles(?f) ^ CPCS:NotChangeProneFilesJFC060(?ncp) ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.0") -> CPCS:hasNCPFilesAs(?ncp, ?f)
JFC0.6.0_NCP_R4*	(a) swrlb:stringEqualIgnoreCase(?n, "JFC0.6.0_NCP_Rule4") ^ CPCS:hasNCPRuleName_Number(?ncp, ?n) ^ CPCS:hasSLOC(?f, ?sloc) ^ CPCS:JavaFiles(?f) ^ CPCS:NotChangeProneFilesJFC060(?ncp) ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.0") ^ swrlb:notEqual(?sloc, "L") -> CPCS:hasNCPFilesAs(?ncp, ?f)
	(b) CPCS:hasCBO(?f, ?cbo) ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.6.0_NCP_Rule4") ^ CPCS:hasNCPRuleName_Number(?ncp, ?n) ^ CPCS:JavaFiles(?f) ^ CPCS:NotChangeProneFilesJFC060(?ncp) ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.0") ^ swrlb:notEqual(?cbo, "L") -> CPCS:hasNCPFilesAs(?ncp, ?f)
JFC0.6.0_NCP_R5*	(a) swrlb:stringEqualIgnoreCase(?n, "JFC0.6.0_NCP_Rule5") ^ CPCS:hasNCPRuleName_Number(?ncp, ?n) ^ CPCS:hasDIT(?f, ?dit) ^ swrlb:notEqual(?dit, "M") ^ CPCS:JavaFiles(?f) ^ CPCS:NotChangeProneFilesJFC060(?ncp) ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.0") -> CPCS:hasNCPFilesAs(?ncp, ?f)
	(b) swrlb:notEqual(?dit, "H") ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.6.0_NCP_Rule5") ^ CPCS:hasNCPRuleName_Number(?ncp, ?n) ^ CPCS:hasRFC(?f, ?dit) ^ CPCS:JavaFiles(?f) ^ CPCS:NotChangeProneFilesJFC060(?ncp) ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.0") -> CPCS:hasNCPFilesAs(?ncp, ?f)
JFC0.6.0_NCP_R6*	(a) CPCS:hasNCPRuleName_Number(?ncp, ?n) ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.6.0_NCP_Rule6") ^ CPCS:hasDIT(?f, ?dit) ^ swrlb:notEqual(?dit, "M") ^ CPCS:JavaFiles(?f) ^ CPCS:NotChangeProneFilesJFC060(?ncp) ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.0") -> CPCS:hasNCPFilesAs(?ncp, ?f)
	(b) swrlb:notEqual(?dit, "VH") ^ CPCS:hasNCPRuleName_Number(?ncp, ?n) ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.6.0_NCP_Rule6") ^ CPCS:hasRFC(?f, ?dit) ^ CPCS:JavaFiles(?f) ^ CPCS:NotChangeProneFilesJFC060(?ncp) ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.0") -> CPCS:hasNCPFilesAs(?ncp, ?f)
JFC0.6.0_NCP_R7*	CPCS:hasNCPRuleName_Number(?ncp, ?n) ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.6.0_NCP_Rule7") ^ CPCS:JavaFiles(?f) ^ swrlb:notEqual(?cogc, "L") ^ CPCS:hasCOGC(?f, ?cogc) ^ CPCS:NotChangeProneFilesJFC060(?ncp) ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.0") -> CPCS:hasNCPFilesAs(?ncp, ?f)

JFC0.6.0_NCP_R8	CPCS:hasNCPRuleName_Number(?ncp, ?n) ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.6.0_NCP_Rule8") ^ CPCS:hasRFC(?f, "VL") ^ CPCS:hasDIT(?f, "VL") ^ CPCS:JavaFiles(?f) ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.0") ^ CPCS:NotChangeProneFilesJFC060(?ncp) -> CPCS:hasNCPFilesAs(?ncp, ?f)
JFC0.6.0_NCP_R9	CPCS:hasRFC(?f, "L") ^ CPCS:hasNCPRuleName_Number(?ncp, ?n) ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.6.0_NCP_Rule9") ^ CPCS:JavaFiles(?f) ^ CPCS:NotChangeProneFilesJFC060(?ncp) ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.0") -> CPCS:hasNCPFilesAs(?ncp, ?f)
JFC0.6.0_NCP_R10	CPCS:hasNCPRuleName_Number(?ncp, ?n) ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.6.0_NCP_Rule10") ^ CPCS:JavaFiles(?f) ^ CPCS:hasCBO(?f, "VL") ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.0") ^ CPCS:NotChangeProneFilesJFC060(?ncp) ^ CPCS:hasSLOC(?f, "VL") -> CPCS:hasNCPFilesAs(?ncp, ?f)
JFC0.6.0_NCP_R11	CPCS:hasNCPRuleName_Number(?ncp, ?n) ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.6.0_NCP_Rule11") ^ CPCS:JavaFiles(?f) ^ CPCS:NotChangeProneFilesJFC060(?ncp) ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.0") ^ CPCS:hasSLOC(?f, "VL") -> CPCS:hasNCPFilesAs(?ncp, ?f)
JFC0.6.0_NCP_R12	CPCS:hasNCPRuleName_Number(?ncp, ?n) ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.6.0_NCP_Rule12") ^ CPCS:hasDIT(?f, "VL") ^ CPCS:JavaFiles(?f) ^ CPCS:NotChangeProneFilesJFC060(?ncp) ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.0") -> CPCS:hasNCPFilesAs(?ncp, ?f)
JFC0.6.0_NCP_R13	CPCS:hasNCPRuleName_Number(?ncp, ?n) ^ CPCS:hasCOGC(?f, "VL") ^ CPCS:JavaFiles(?f) ^ CPCS:hasCBO(?f, "VL") ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.6.0_NCP_Rule13") ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.0") ^ CPCS:NotChangeProneFilesJFC060(?ncp) -> CPCS:hasNCPFilesAs(?ncp, ?f)

Rule Names without an *(asterisk) indicate the extracted rule, while those with an * are the inverse rules that have been additionally included. For example, JFC0.6.0_CP_R1 is the original change-proneness rule that is extracted, whereas JFC0.6.0_NCP_R1* is its inverse rule that is additionally included for non-change-proneness. Similarly, JFC0.6.0_CP_R8* is the inverse rule for change-proneness included corresponding to the actual extracted rule JFC0.6.0_NCP_R8 which pertains to non-change-proneness.

5.4.2.3 Addressing the monotonicity issue in SWRL

It can be observed that there exist two rules corresponding to a rule name for some cases in Tables 5.8 and 5.9. For example: JFC0.6.0_CP_R8*. This is because, like OWL, SWRL supports monotonic inference only [229], implying that it executes one condition at a time serially. Rule name JFC0.6.0_CP_R8* is an inverse of the original extracted rule JFC0.6.0_NCP_R8. JFC0.6.0_NCP_R8 states that for every Java File that belongs to version "`JFreeChart0.7.0`" and that has datatype property values of both `hasRFC` and `hasDIT` equal to `'VL'` is a not change

prone file corresponding to the instance of NCPFilesJFC060 that has NCPRuleName_Number of JFC0.6.0_NCP_Rule8. JFC0.6.0_NCP_R8 is written in SWRL in the following manner:

```
CPCS:hasNCPRuleName_Number(?ncp,                ?n)                ^
swrlb:stringEqualIgnoreCase(?n,        "JFC0.6.0_NCP_Rule8")       ^
CPCS:hasRFC(?f, "VL") ^ CPCS:hasDIT(?f, "VL") ^ CPCS:JavaFiles(?f) ^
CPCS:belongsToVersion(?f,             "JFreeChart0.7.0")           ^
CPCS:NotChangeProneFilesJFC060(?ncp) -> CPCS:hasNCPFilesAs(?ncp, ?f)
```

Since the rule JFC0.6.0_NCP_R8 has two non-negation conditions: hasRFC and hasDIT equal to 'VL', that need to hold true simultaneously, therefore the SWRL rule works fine and renders correct inferences. However, its inverse would imply the negation of both the conditions to hold true at the same time ie. hasRFC and hasDIT not equal to 'VL'. This, unfortunately, does not work for SWRL as it, instead of computing both the conditions as a unit, computes the conditions sequentially and by doing so fails to include several correct inferences.

In order to obviate this issue, each of the rules that consist of more than one negative condition is split into multiple rules according to the number of negations that it needs to satisfy. Therefore JFC0.6.0_CP_R8* is split into two rules addressing the negations separately but updating the same instance of CPFilesJFC060. This ensures that the final inference consists of the union of the inferred Java file instances from the two rules JFC0.6.0_CP_R8*(a) and JFC0.6.0_CP_R8*(b).

5.4.2.4 Execution of SWRL rules and viewing the inferences

Every rule mentioned in Tables 5.8 and 5.9 is entered as part of the developed ontology using the SWRLTab of Protégé. As mentioned earlier, SWRL [222] does not permit a direct inference, to overcome which Drools Rule Engine is utilized. Therefore, post the inclusion of relevant rules in SWRL, Drools Rule Engine is run as shown in Figures 5.5, 5.6 and 5.7.

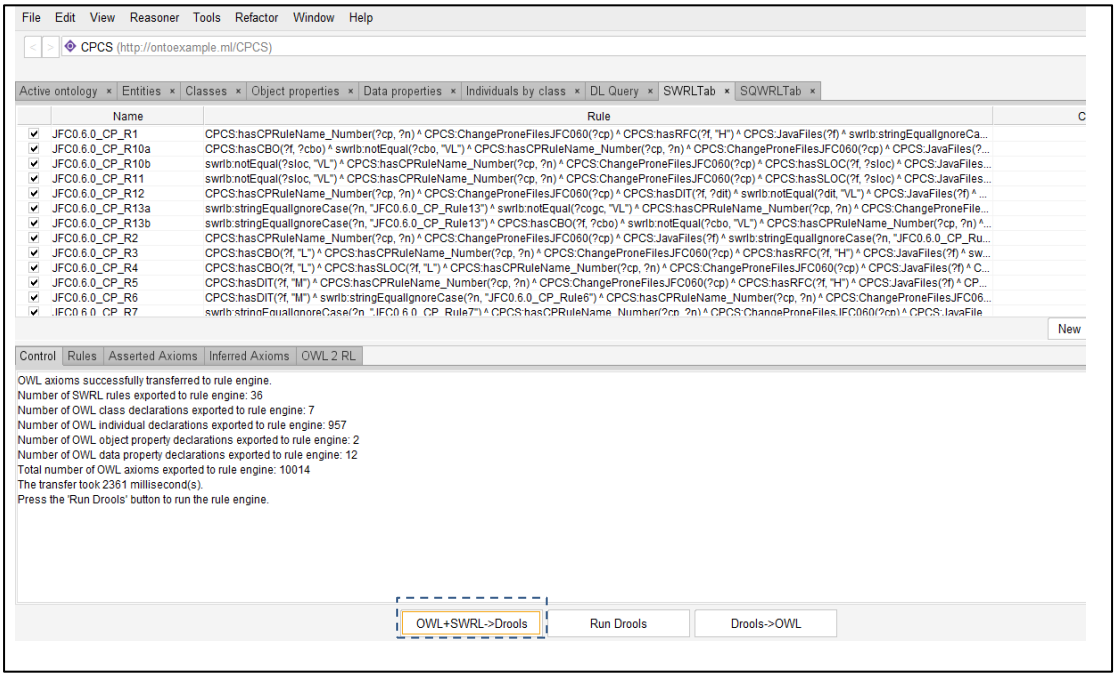

Figure 5.5 Screenshot indicating the export of the OWL ontology and the SWRL rules from SWRLTab to the Drools rule engine.

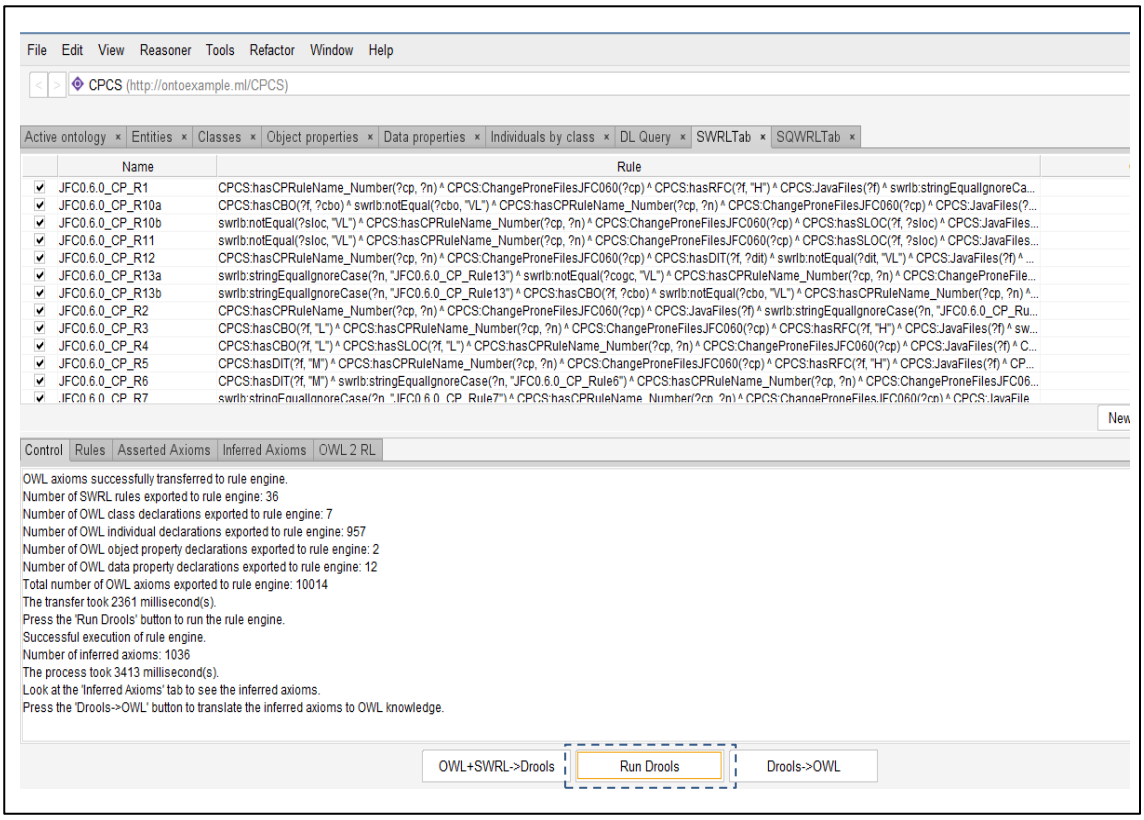

Figure 5.6 The screenshot indicating the execution of the Drools rule engine

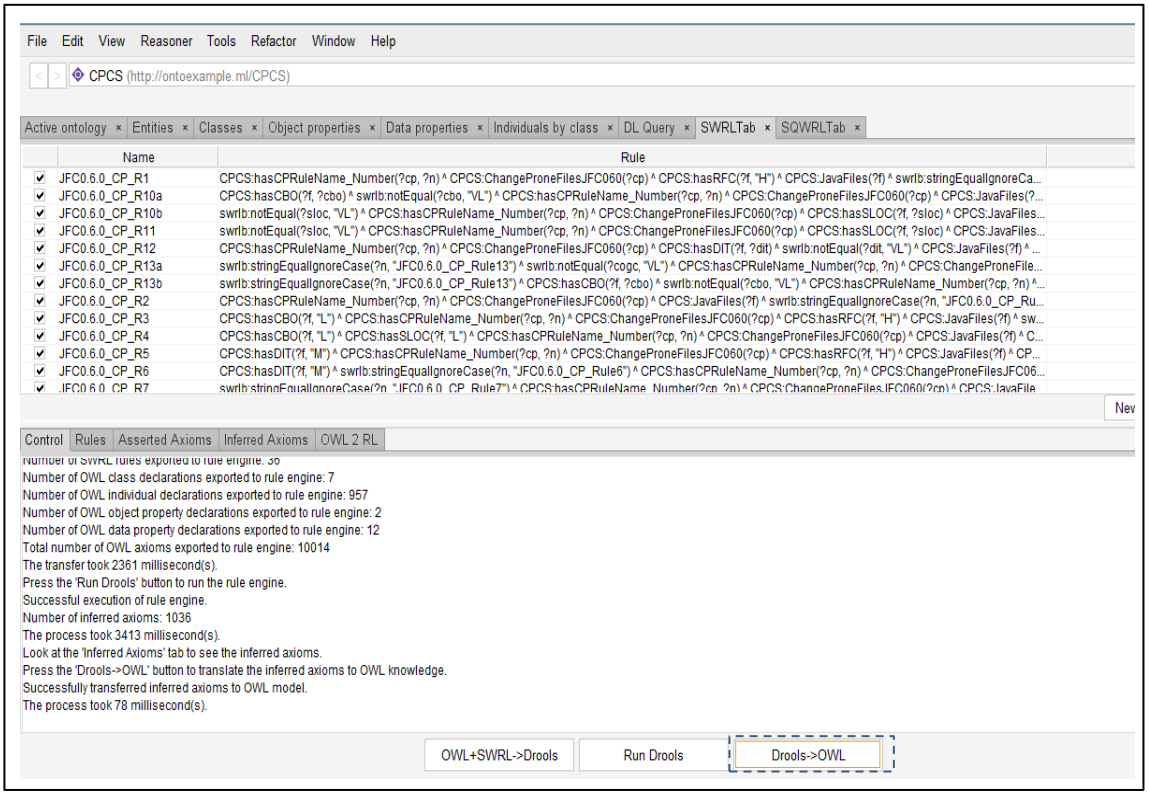

Figure 5.7 The screenshot indicating the successful transfer of the new information drawn from the rules via the Drools Engine back to the OWL ontology

After the rules are run and the reasoner (Pellet) is synchronized, the knowledge class and concept explanation in the proposed Java file ontology are translated into an appropriate format so that the ontology instances may go into operation. Additionally, the reasoner indentifies the inconsistent and conflicting knowledge in ontology [128]. The now populated ontology comprises of the object properties inferred from the SWRL rules.

For example, as mentioned in Table 5.8, the first rule derived from JFreeChart0.6.0 that governs the change-proneness of Java Files is stated in SWRLTab as:

```
CPCS:hasCPRuleName_Number(?cp, ?n) ^ CPCS:ChangeProneFilesJFC060(?cp)
^ CPCS:hasRFC(?f, "H") ^ CPCS:JavaFiles(?f) ^
swrlb:stringEqualIgnoreCase(?n, "JFC0.6.0_CP_Rule1") ^
CPCS:belongsToVersion(?f, "JFreeChart0.7.0") ->
CPCS:hasCPFilesAs(?cp, ?f)
```

Therefore, as seen in Figure 5.8 after the rules are fired using the Drools inference engine and the reasoner is synchronized, the instance JFC_CP_Rule1 of the sub-class ChangeProneFilesJFC060 is now updated with those Java files from JFreeChart version 0.7.0 that satisfy the datatype property *hasRFC* equal to *"H"*. This new

inference is represented via the rectangle drawn using dashed lines in the Figure 5.8. All the instances with respect to ChangeProneFilesJFC060 and NotChangeProneFilesJFC060 are updated in a similar manner comprising of the Java Files from JFreeChart 0.7.0 that satisfy the SWRL Rules mentioned in Tables 5.8 and 5.9. The populated ontology with new inferences is then queried to select the relevant Java files from JFreeChart 0.7.0 that are going to be used with change in its successive release.

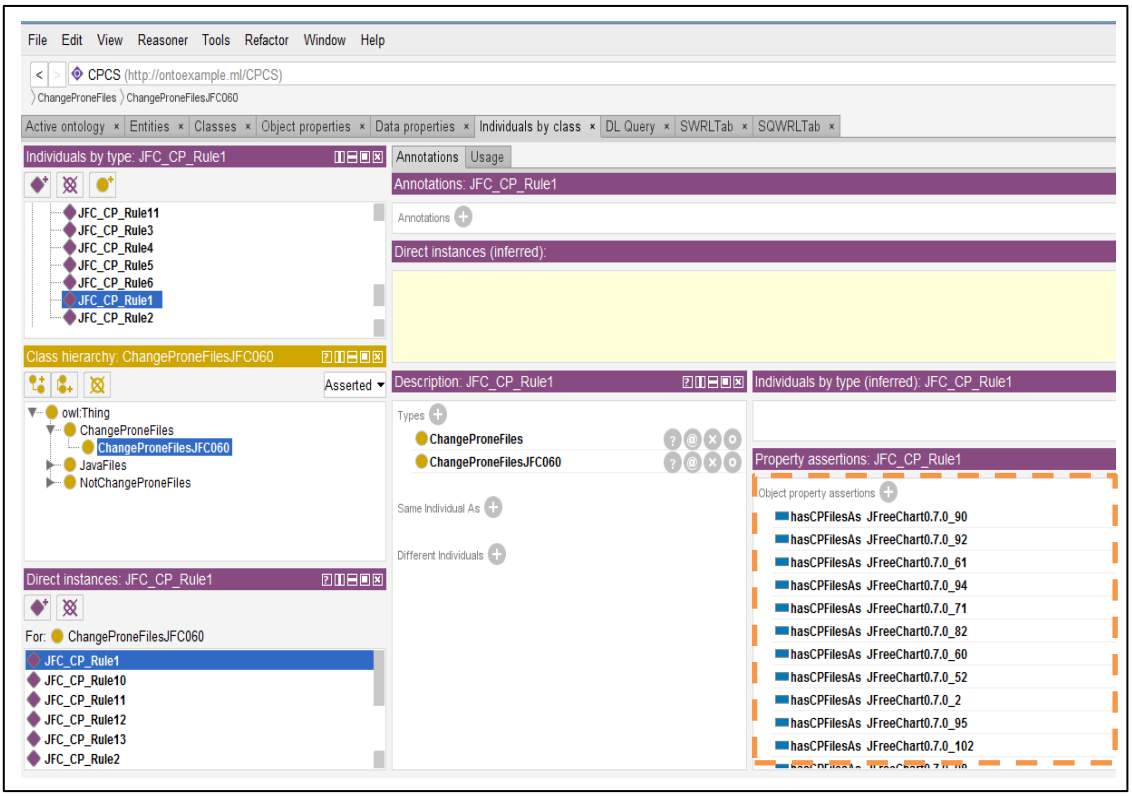

Figure 5.8 Screenshot exhibiting the Java Files from JFreeChart 0.7.0 that satisfy the change-proneness governing Rule1 of JFreeChart 0.6.0

5.4.3 Querying the populated knowledge-base for change-prone component selection

Post the derivation of the inferred relationships which state so as to which Java files from JFreeChart 0.7.0 are change-prone and not-change prone as per the shortlisted rules from JFreeChart 0.6.0, the populated ontology is now queried to find the possibility of change-proneness and non-change-proneness of each of the JFreeChart 0.7.0 Java files. This possibility is computed using the SQWRL and basically consists of calculating the number of times a JFreeChart 0.7.0 file is estimated to be change-prone and number of times it has been estimated to be non-change-prone. Finally, if

the computed change-proneness count of a file is found to be higher than its non-change-proneness count, it is selected as change-prone.

As detailed earlier, SQWRL is a library extension to SWRL and is established on the fact that a rule antecedent can be viewed as a pattern-matching mechanism, i.e., a query. It allows queries directed at OWL classes, individuals and properties. SQWRL queries provide additional self-explanatory structures such as orderBy, selectDistinct, and countDistinct (i.e., similar to SWRL). However, it should be noted that SQWRL is designed to be a query language only, whereas SWRL can be used to create new knowledge. It should be remembered that SQWRL is meant to be a language for querying only, as opposed to SWRL which can be employed to create new knowledge. Figure 5.9 exhibits how a typical SQWRLTab in Protégé looks like wherein the dashed rectangle represents the area where the results from the SQWRL queries are displayed.

Note that a query can be transformed to a SWRL rule and the outcomes put into the relevant class provided there is no SQWRL construct employed in the antecedent of the query statement.

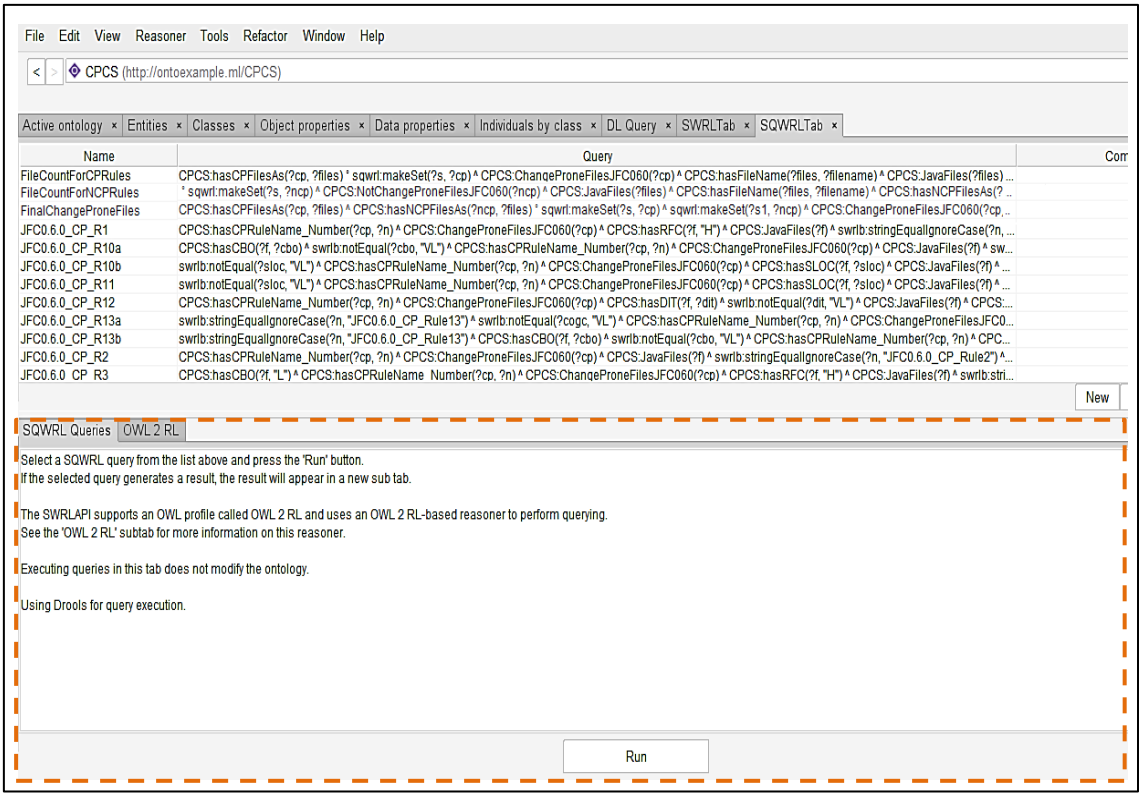

Figure 5.9 SQWRLTab in Protégé

The three SQWRL queries named as: *FileCountforCPRules*, *FileCountforNCPRules* and *FinalChangeProneFiles*; are built for each version analyzed and can be viewed along with the built SWRL rules in the SQWRLTab. This is because SQWRL honours the SWRL rules and their inferences during query execution. The following sub-sections address these queries in detail with screenshots of actual execution.

- FileCountforCPRules

Query *FileCountforCPRules* displays the file instances that exist corresponding to JFreeChart 0.7.0. It also captures their file name and their total count with respect to each of the change-prone rules derived from its previously released version JFreeChart 0.6.0.

```
CPCS:hasCPFilesAs(?cp, ?files) ^ CPCS:ChangeProneFilesJFC060(?cp)
^ CPCS:hasFileName(?files, ?filename) ^ CPCS:JavaFiles(?files) ^
CPCS:belongsToVersion(?files, "JFreeChart0.7.0")  .
sqwrl:makeSet(?s, ?cp) ^ sqwrl:groupBy(?s, ?files)  .
sqwrl:size(?CountAsPerCPRules, ?s) -> sqwrl:select(?files,
?filename, ?CountAsPerCPRules)
```

It can be read as:

For:

- the asserted property 'hasCPFilesAs' that links a change prone file 'cp' to a Java file 'files',
- where 'cp' are Java files inferred to be change-prone according to the change proneness governing rules of JFreeChart0.6.0,
- where each of the 'files' are of version 'JFreeChart0.7.0' via an asserted datatype property of 'belongsToVersion' and 'files' have a 'filename' via an asserted datatype property of 'hasFileName',
- where 's' is a set constructed of all the change prone files 'cp' using the makeSet built-in construct of SQWRL, which is further grouped according to 'files' and where 'CountAsPerCPRules' is the count/size of each of the 'files' in the grouped set 's',

Output all conforming 'files' and their 'filename' along with their count as 'CountAsPerCPRules'.

Figures 5.10 and 5.11 are the results of the *FileCountforCPRules* query. Figure 5.10 indicates that the query returns 102 rows. This statistic is expected since as seen in column 3 of Table 4.3 of Chapter 4, the JFreeChart 0.7.0 has a total of 102 Java files. Figure 5.11 indicates the output from the query in which the file instances and their filenames along with their number of times each of the file satisfied a change-proneness governing rule of JFreeChart 0.6.0 given in Table 5.8.

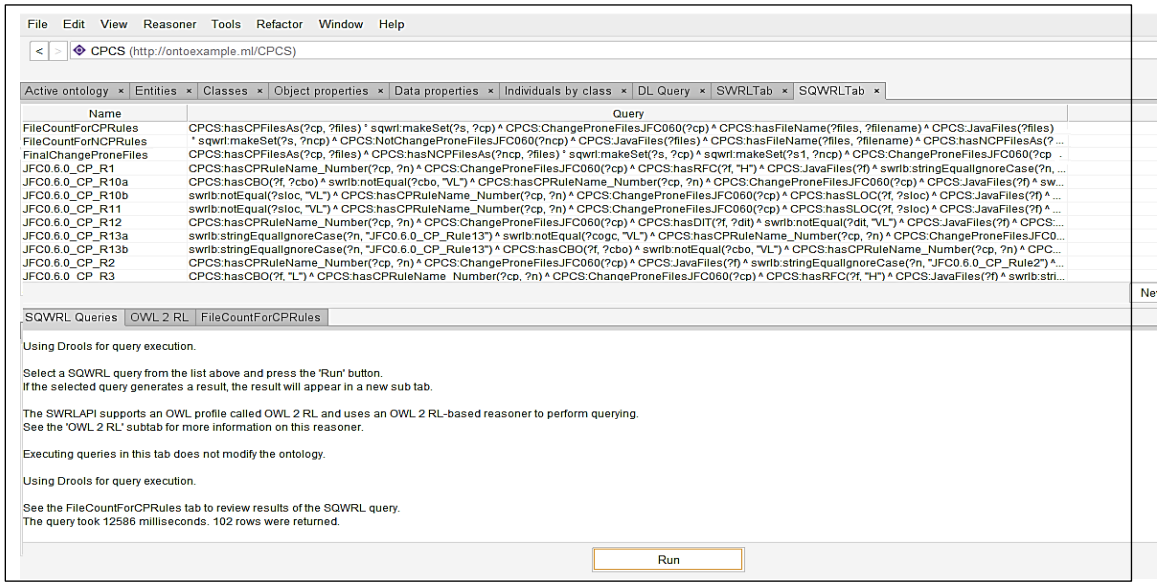

Figure 5.10 Screenshot exhibiting the succesful execution of the FileCountforCPRules query

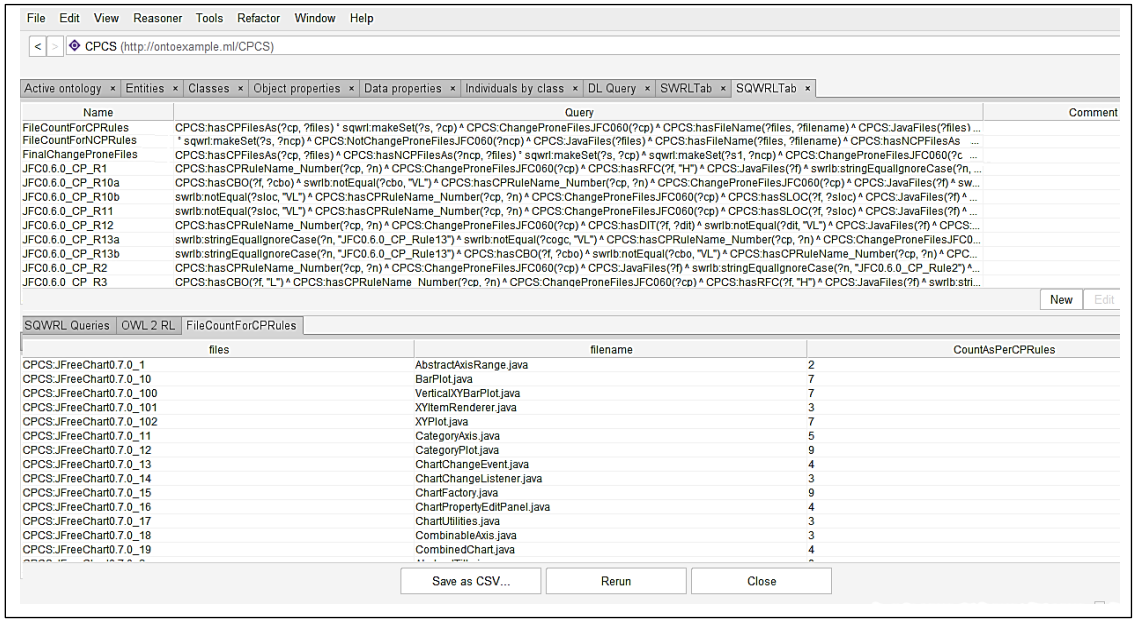

Figure 5.11 Screenshot exhibiting the results of the FileCountforCPRules query

- FileCountforNCPRules

Query FileCountforNCPRules displays the file instances that exist corresponding to JFreeChart 0.7.0, their file name and their total count with respect to each of the not change-prone rules derived from JFreeChart 0.6.0.

```
  .  sqwrl:makeSet(?s, ?ncp) ^
CPCS:NotChangeProneFilesJFC060(?ncp) ^ CPCS:JavaFiles(?files) ^
CPCS:hasFileName(?files, ?filename) ^ CPCS:hasNCPFilesAs(?ncp,
?files) ^ CPCS:belongsToVersion(?files, "JFreeChart0.7.0") ^
sqwrl:groupBy(?s, ?files)  .  sqwrl:size(?CountAsPerNCPRules, ?s)
-> sqwrl:select(?files, ?filename, ?CountAsPerNCPRules)
```

It can be read as:

For:

- the asserted property 'hasNCPFilesAs' that links a change prone file 'ncp' to a Java file 'files',
- where 'ncp' are Java files inferred to be not change-prone according to the not change-proneness governing rules of JFreeChart0.6.0,
- where each of the 'files' are of version 'JFreeChart0.7.0' via an asserted datatype property of 'belongsToVersion' and 'files' have a 'filename' via an asserted datatype property of 'hasFileName',
- where 's' is a set constructed of all the non-change-prone files 'ncp' using the makeSet built-in construct of SQWRL, which is further grouped according to 'files' and where 'CountAsPerNCPRules' is the count/size of each of the 'files' in the grouped set 's',

Output all conforming 'files' and their 'filename' along with their count as 'CountAsPerNCPRules'.

Figures 5.12 and 5.13 are the results of the *FileCountforNCPRules* query. Figure 5.12 indicates that the query returns 102 rows. Again, this statistic is expected. Figure 5.13 indicates the output from the query in which the file instances and their filenames along with their number of times each of the file satisfied a non-change-proneness governing rule of JFreeChart 0.6.0 given in Table 5.9.

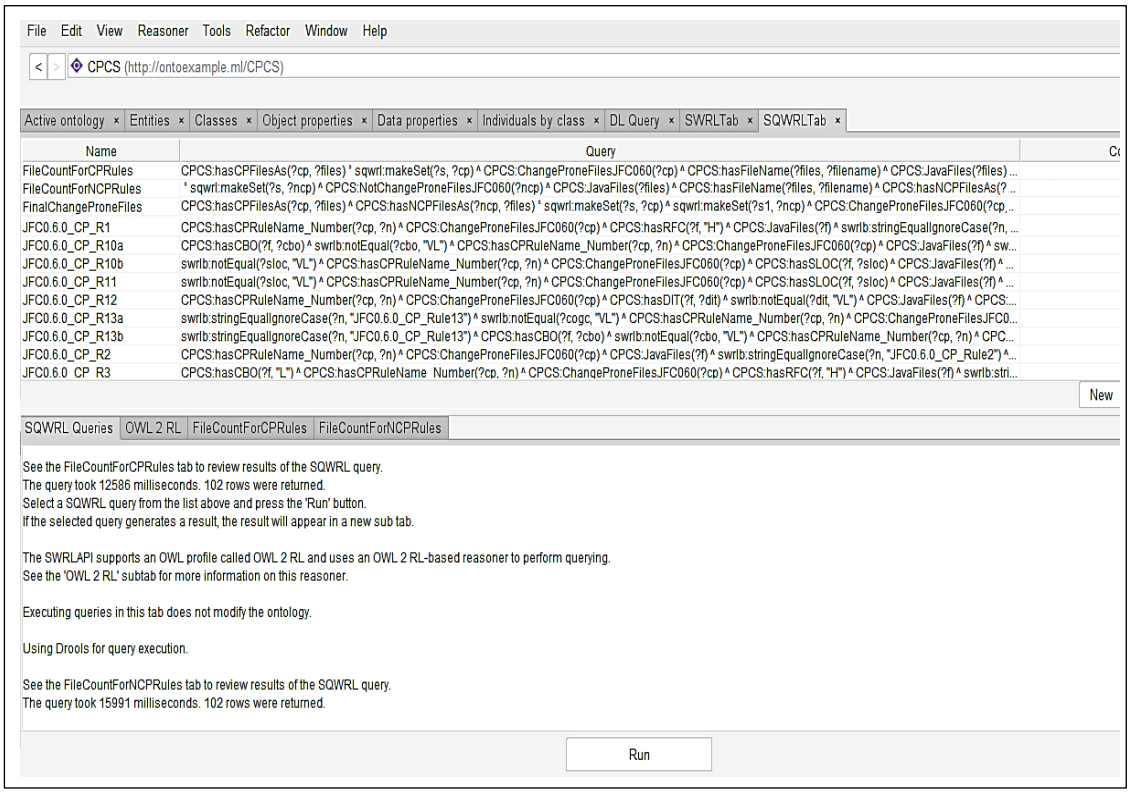

Figure 5.12 Screenshot exhibiting the succesful execution of the FileCountforNCPRules query

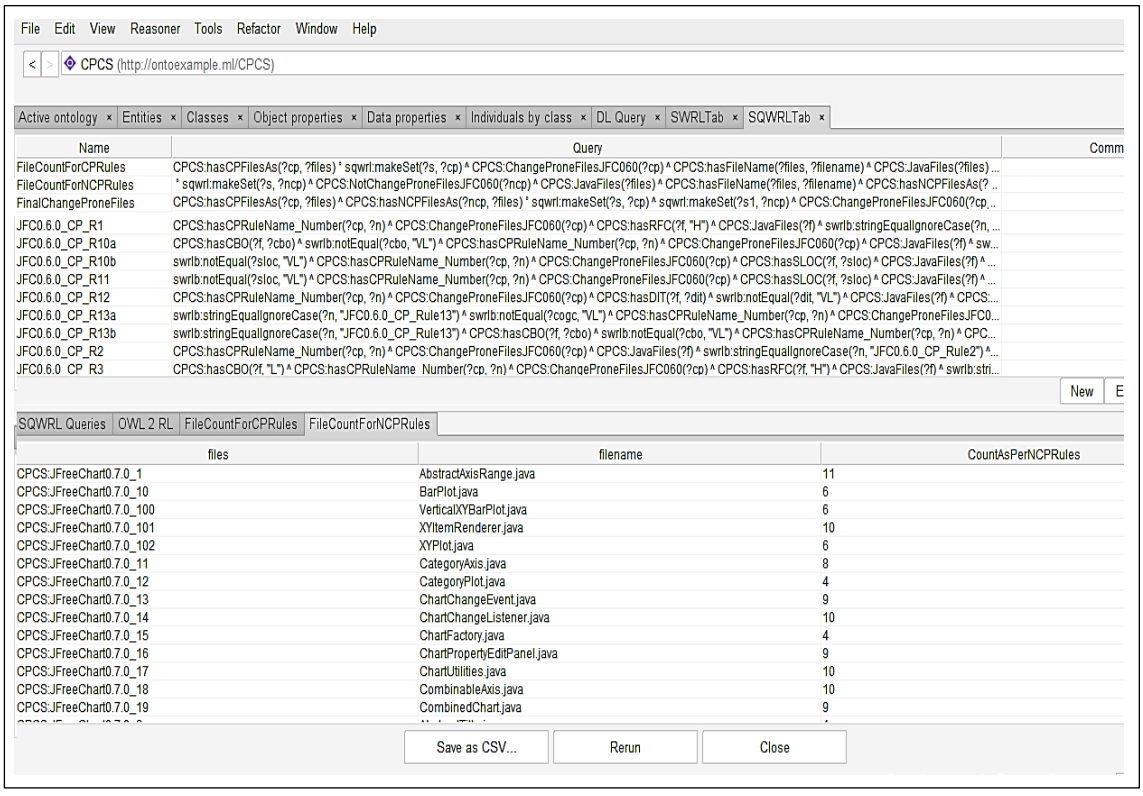

Figure 5.13 Screenshot exhibiting the results of the FileCountforNCPRules query

- FinalChangeProneFiles

Query FinalChangeProneFiles essentially employs the computed change-prone and not change-prone counts of each of the files, calculated using the queries FileCountforCPRules and FileCountforNCPRules, and compares these counts with respect to each file. The Java file having the CountAsPerCPRules higher than or equal to its CountAsPerNCPRules is selected as a Java file which has the possibility to be used with change in its upcoming version. As per our current context of application, the query displays the Java file instances corresponding to JFreeChart 0.7.0 that are selected to be change-prone in their successive version along with their file name. The query for the same is formulated as follows:

```
CPCS:hasCPFilesAs(?cp, ?files) ^ CPCS:hasNCPFilesAs(?ncp, ?files)
.  sqwrl:makeSet(?s, ?cp) ^ sqwrl:makeSet(?s1, ?ncp) ^
CPCS:ChangeProneFilesJFC060(?cp) ^
CPCS:NotChangeProneFilesJFC060(?ncp) ^ CPCS:hasFileName(?files,
?filename) ^ CPCS:JavaFiles(?files) ^ CPCS:belongsToVersion(?files,
"JFreeChart0.7.0") ^ sqwrl:groupBy(?s, ?files) ^ sqwrl:groupBy(?s1,
?files)  .  sqwrl:size(?CountAsPerCPRules, ?s) ^
sqwrl:size(?CountAsPerNCPRules, ?s1)^
swrlb: greaterThanOrEqual(?CountAsPerCPRules, ?CountAsPerNCPRules)
-> sqwrl:select(?files, ?filename)
```

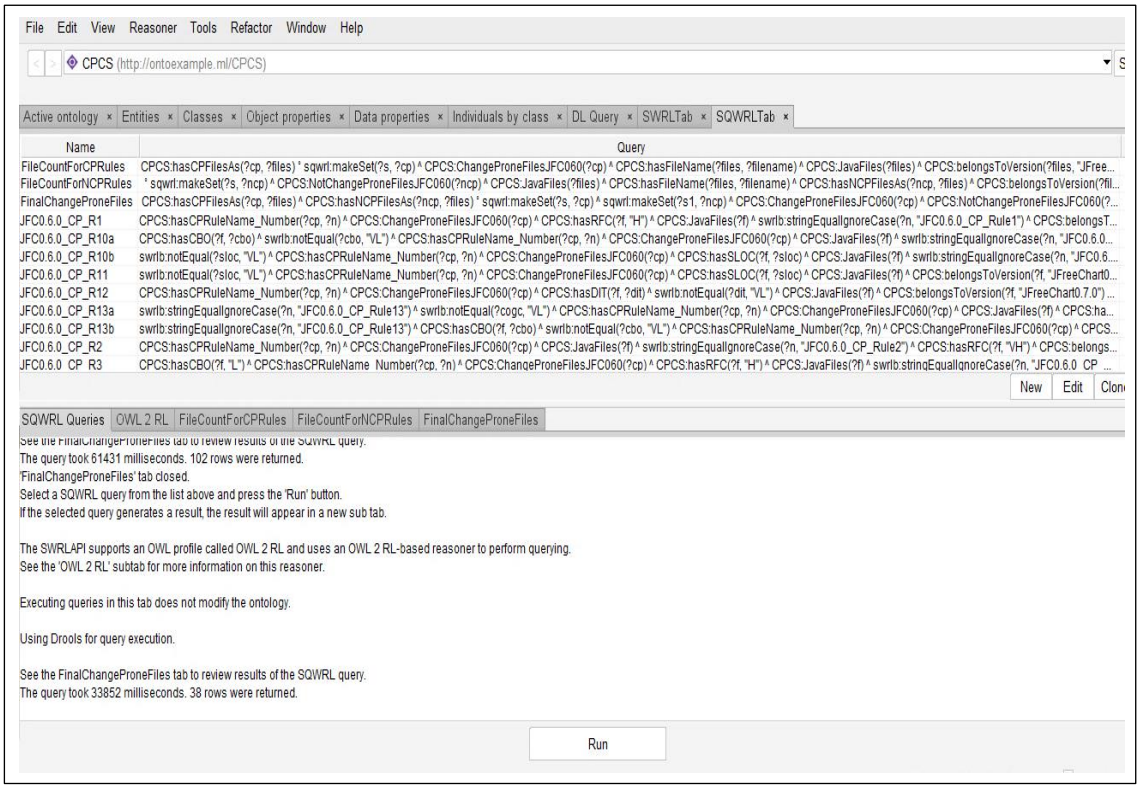

Figure 5.14 Screenshot exhibiting the successful execution of the FinalChangeProneFiles query

Figures 5.14 and 5.15 are the results of the FinalChangeProneFiles query. Figure 5.14 indicates that the query returns 38 rows, thus exhibiting that 38 Java files are selected to be change-prone from the 102 Java files in JFreeChart 0.7.0, according to the rules derived from its previous version JFreeChart 0.6.0. Figure 5.15 displays the output from the query which consists of the 38 selected change-prone file instances and their filenames.

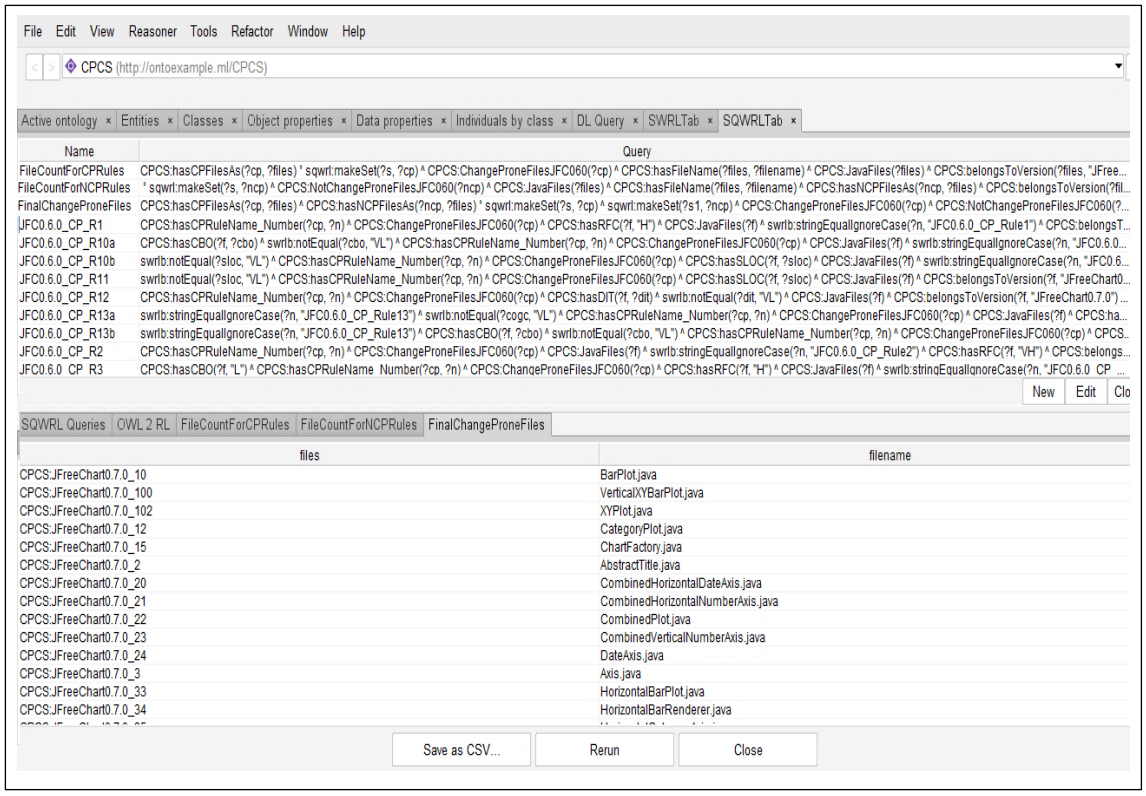

Figure 5.15 Screenshot exhibiting the final 38 change–prone files of JFreeChart 0.7.0 dervied via the FinalChangeProneFiles query

5.4.4 Execution of SWRL rules and viewing the inferences for Version $_i$, where i >= 3

As mentioned in Section 5.4.2, the SWRL rules derived from the RF algorithm are employed in the ontology in an incremental manner for selection of change-prone Java files. That is, the SWRL rules with respect to change-proneness prediction generated till Version $_{i-1}$ are employed, if one needs to select the Java files of a Version $_i$ that will be used with change in Version $_{i+1}$. Therefore, for the selection of Java files from JFreeChart 0.7.1 that could be used with change in its successive release, the extracted rules from the previously released versions i.e. JFreeChart 0.6.0

and JFreeChart 0.7.0 are utlized. These rules when applied on the Java files instances of JFreeChart 0.7.1 essentially classify the instances as change-prone or not change prone according to the individual rules. The ontology is then queried for the selection of change-prone Java files of JFreeChart 0.7.1.

5.4.4.1 Modifying the rules from JFreeChart 0.6.0 to be used for JFreeChart 0.7.1

The 26 rules in Tables 5.8 and 5.9 that are extracted corresponding to JFreeChart 0.6.0 for classifying the Java files of JFreeChart 0.7.0 are now utilized for classification of Java files from JFreeChart 0.7.1. This is simply done by modifying just the `belongsToVersion` property for each of the 26 rules like exhibited in the query given below: For instance, the `belongsToVersion` property in rule JFC0.6.0_CP_Rule1 stated in Table 5.8 is modified in the manner below to assert that every Java file (?f) now belongs to version JFreeChart0.7.1. This is done for each of the 26 rules.

```
CPCS:hasCPRuleName_Number(?cp, ?n) ^ CPCS:ChangeProneFilesJFC060(?cp)
^ CPCS:hasRFC(?f, "H") ^ CPCS:JavaFiles(?f) ^
swrlb:stringEqualIgnoreCase(?n, "JFC0.6.0_CP_Rule1") ^
CPCS:belongsToVersion(?f, "JFreeChart0.7.1") ->
CPCS:hasCPFilesAs(?cp, ?f)
```

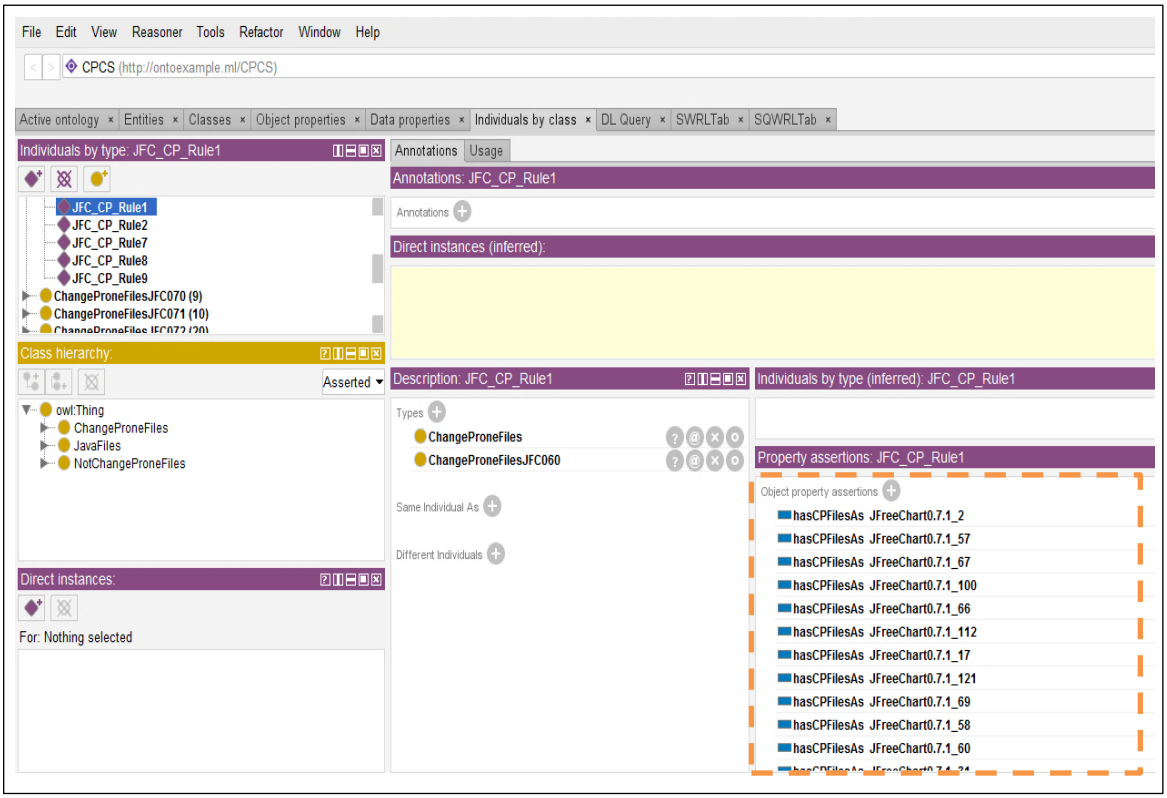

Figure 5.16 Screenshot exhibiting the Java Files from JFreeChart 0.7.1 that satisfy the change-proneness governing Rule1 of JFreeChart 0.6.0

As seen in Figure 5.16 after the rules are fired using the Drools inference engine and the reasoner is synchronized, the instance JFC_CP_Rule1 of the sub-class ChangeProneFilesJFC060 is now updated with those Java files from JFreeChart version 0.7.1 that satisfy the datatype property hasRFC equal to “H”. This new inference is represented via the rectangle drawn using dashed lines in the Figure 5.16. All the instances with respect to ChangeProneFilesJFC060 and NotChangeProneFilesJFC060 are updated in a similar manner comprising of the Java Files from JFreeChart 0.7.1 that satisfy the 26 SWRL Rules.

5.4.4.2 Adding the rules from JFreeChart 0.7.0 to be used for JFreeChart 0.7.1

Additionally, nine change-proneness governing rules that are shortlisted with respect to JFreeChart 0.7.0 are also utilized to classify the Java files from JFreeChart 0.7.1. As mentioned earlier, some of the extracted rules also are those pertaining to non-change-proneness of a Java file. Therefore, for every rule extracted, its inverse rule is also included and entered as a SWRL rule for that version, making the nine rules a total of 18 rules that form an equal number of change-prone as well as not change-prone rules. The final 18 SWRL rules corresponding to the JFreeChart 0.7.0 that are used to find are given in Tables 5.10 and 5.11.

Additionally, as seen in Figure 5.17 after the rules are fired using the Drools inference engine and the reasoner is synchronized, the instance JFC_CP_Rule20 (which denotes the 7th Change Prone rule of JFreeChart 0.7.0) of the sub-class ChangeProneFilesJFC070 is now updated with those Java files from JFreeChart version 0.7.1 that satisfy the datatype property hasDIT not equal to “L”. This new inference is represented via the rectangle drawn using dashed lines in the Figure 5.17. All the instances with respect to ChangeProneFilesJFC070 and NotChangeProneFilesJFC070 are updated in a similar manner comprising of the Java Files from JFreeChart 0.7.1 that satisfy the 18 SWRL rules.

Table 5.10 Rules entered corresponding to JFreeChart 0.7.0 in order to identify change prone files of JFreeChart 0.7.1

Rule Name	JFreeChart 0.7.0 CP governing rules
JFC0.7.0_CP_R1	CPCS:hasCPRuleName_Number(?cp, ?n) ^ CPCS:ChangeProneFilesJFC070(?cp) ^ CPCS:hasCE(?f, "VH") ^ CPCS:JavaFiles(?f) ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.7.0_CP_Rule1") ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.1") -> CPCS:hasCPFilesAs(?cp, ?f)
JFC0.7.0_CP_R2	CPCS:hasCPRuleName_Number(?cp, ?n) ^ CPCS:ChangeProneFilesJFC070(?cp) ^ CPCS:hasRFC(?f, "H") ^ CPCS:JavaFiles(?f) ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.7.0_CP_Rule2") ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.1") -> CPCS:hasCPFilesAs(?cp, ?f)
JFC0.7.0_CP_R3	CPCS:hasCPRuleName_Number(?cp, ?n) ^ CPCS:ChangeProneFilesJFC070(?cp) ^ CPCS:hasRFC(?f, "VH") ^ CPCS:JavaFiles(?f) ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.7.0_CP_Rule3") ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.1") -> CPCS:hasCPFilesAs(?cp, ?f)
JFC0.7.0_CP_R4	CPCS:hasCPRuleName_Number(?cp, ?n) ^ CPCS:ChangeProneFilesJFC070(?cp) ^ CPCS:hasDIT(?f, "VH") ^ CPCS:JavaFiles(?f) ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.7.0_CP_Rule4") ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.1") -> CPCS:hasCPFilesAs(?cp, ?f)
JFC0.7.0_CP_R5*	CPCS:hasCPRuleName_Number(?cp, ?n) ^ CPCS:ChangeProneFilesJFC070(?cp) ^ CPCS:hasCE(?f, ?ce) ^ swrlb:notEqual(?ce, "VL") ^ CPCS:JavaFiles(?f) ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.7.0_CP_Rule5") ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.1") -> CPCS:hasCPFilesAs(?cp, ?f)
JFC0.7.0_CP_R6*	CPCS:hasCPRuleName_Number(?cp, ?n) ^ CPCS:ChangeProneFilesJFC070(?cp) ^ CPCS:hasRFC(?f, ?rfc) ^ swrlb:notEqual(?rfc, "VL") ^ CPCS:JavaFiles(?f) ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.7.0_CP_Rule6") ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.1") -> CPCS:hasCPFilesAs(?cp, ?f)
JFC0.7.0_CP_R7*	CPCS:hasCPRuleName_Number(?cp, ?n) ^ CPCS:ChangeProneFilesJFC070(?cp) ^ CPCS:hasDIT(?f, ?dit) ^ swrlb:notEqual(?dit, "L") ^ CPCS:JavaFiles(?f) ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.7.0_CP_Rule7") ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.1") -> CPCS:hasCPFilesAs(?cp, ?f)
JFC0.7.0_CP_R8*	CPCS:hasCPRuleName_Number(?cp, ?n) ^ CPCS:ChangeProneFilesJFC070(?cp) ^ CPCS:hasDIT(?f, ?dit) ^ swrlb:notEqual(?dit, "VL") ^ CPCS:JavaFiles(?f) ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.7.0_CP_Rule8") ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.1") -> CPCS:hasCPFilesAs(?cp, ?f)
JFC0.7.0_CP_R9*	JFC0.7.0_CP_R9a CPCS:hasCPRuleName_Number(?cp, ?n) ^ CPCS:ChangeProneFilesJFC070(?cp) ^ CPCS:hasSLOC(?f, ?sloc) ^ swrlb:notEqual(?sloc, "VL") ^ CPCS:JavaFiles(?f) ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.7.0_CP_Rule9") ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.1") -> CPCS:hasCPFilesAs(?cp, ?f)
	JFC0.7.0_CP_R9b CPCS:hasCPRuleName_Number(?cp, ?n) ^ CPCS:ChangeProneFilesJFC070(?cp) ^ CPCS:hasCE(?f, ?ce) ^ swrlb:notEqual(?ce, "VL") ^ CPCS:JavaFiles(?f) ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.7.0_CP_Rule9") ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.1") -> CPCS:hasCPFilesAs(?cp, ?f)

Table 5.11 Rules entered corresponding to JFreeChart 0.7.0 in order to identify non-change prone files of JFreeChart 0.7.1

Rule Name	JFreeChart 0.7.0 NCP governing rules
JFC0.7.0_NCP_R1*	CPCS:hasNCPRuleName_Number(?ncp, ?n) ^ CPCS:NotChangeProneFilesJFC070(?ncp) ^ CPCS:hasCE(?f, ?ce) ^ swrlb:notEqual(?ce, "VH") ^ CPCS:JavaFiles(?f) ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.7.0_NCP_Rule1") ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.1") -> CPCS:hasNCPFilesAs(?ncp, ?f)

JFC0.7.0_NCP_R2*	CPCS:hasNCPRuleName_Number(?ncp, ?n) ^ CPCS:NotChangeProneFilesJFC070(?ncp) ^ CPCS:hasRFC(?f, ?rfc) ^ swrlb:notEqual(?rfc, "H") ^ CPCS:JavaFiles(?f) ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.7.0_NCP_Rule2") ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.1") -> CPCS:hasNCPFilesAs(?ncp, ?f)
JFC0.7.0_NCP_R3*	CPCS:hasNCPRuleName_Number(?ncp, ?n) ^ CPCS:NotChangeProneFilesJFC070(?ncp) ^ CPCS:hasRFC(?f, ?rfc) ^ swrlb:notEqual(?rfc, "VH") ^ CPCS:JavaFiles(?f) ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.7.0_NCP_Rule3") ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.1") -> CPCS:hasNCPFilesAs(?ncp, ?f)
JFC0.7.0_NCP_R4*	CPCS:hasNCPRuleName_Number(?ncp, ?n) ^ CPCS:NotChangeProneFilesJFC070(?ncp) ^ CPCS:hasDIT(?f, ?dit) ^ swrlb:notEqual(?dit, "VH") ^ CPCS:JavaFiles(?f) ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.7.0_NCP_Rule4") ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.1") -> CPCS:hasNCPFilesAs(?ncp, ?f)
JFC0.7.0_NCP_R5	CPCS:hasNCPRuleName_Number(?ncp, ?n) ^ CPCS:NotChangeProneFilesJFC070(?ncp) ^ CPCS:hasCE(?f, "VL") ^ CPCS:JavaFiles(?f) ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.7.0_NCP_Rule5") ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.1") -> CPCS:hasNCPFilesAs(?ncp, ?f)
JFC0.7.0_NCP_R6	CPCS:hasNCPRuleName_Number(?ncp, ?n) ^ CPCS:NotChangeProneFilesJFC070(?ncp) ^ CPCS:hasRFC(?f, "VL") ^ CPCS:JavaFiles(?f) ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.7.0_NCP_Rule6") ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.1") -> CPCS:hasNCPFilesAs(?ncp, ?f)
JFC0.7.0_NCP_R7	CPCS:hasNCPRuleName_Number(?ncp, ?n) ^ CPCS:NotChangeProneFilesJFC070(?ncp) ^ CPCS:hasDIT(?f, "L") ^ CPCS:JavaFiles(?f) ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.7.0_NCP_Rule7") ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.1") -> CPCS:hasNCPFilesAs(?ncp, ?f)
JFC0.7.0_NCP_R8	CPCS:hasNCPRuleName_Number(?ncp, ?n) ^ CPCS:NotChangeProneFilesJFC070(?ncp) ^ CPCS:hasDIT(?f, "VL") ^ CPCS:JavaFiles(?f) ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.7.0_NCP_Rule8") ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.1") -> CPCS:hasNCPFilesAs(?ncp, ?f)
JFC0.7.0_NCP_R9	CPCS:hasCE(?f, "VL") ^ CPCS:hasNCPRuleName_Number(?ncp, ?n) ^ CPCS:NotChangeProneFilesJFC070(?ncp) ^ CPCS:JavaFiles(?f) ^ swrlb:stringEqualIgnoreCase(?n, "JFC0.7.0_NCP_Rule9") ^ CPCS:belongsToVersion(?f, "JFreeChart0.7.1") ^ CPCS:hasSLOC(?f, "VL") -> CPCS:hasNCPFilesAs(?ncp, ?f)

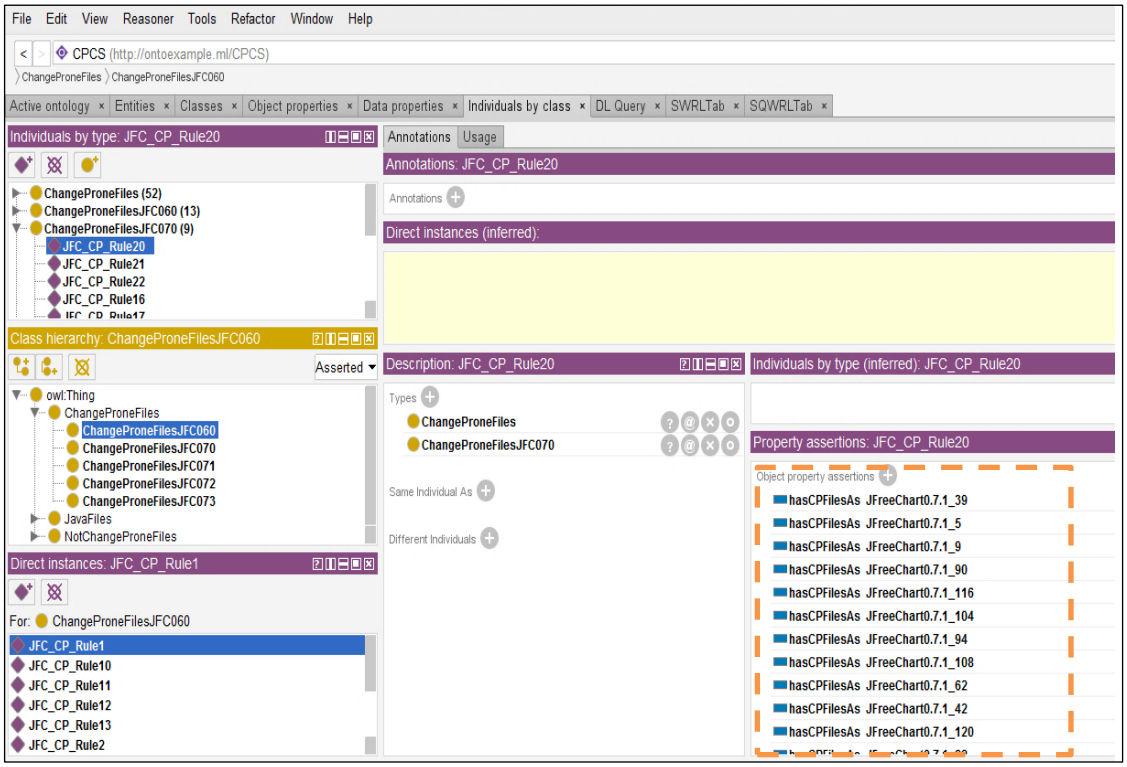

Figure 5.17 Screenshot exhibiting the Java Files from JFreeChart 0.7.1 that satisfy the change-proneness governing rule of JFreeChart 0.7.0

5.4.4.3 Querying the populated knowledge-base for selection of change-prone components from JFreeChart 0.7.1

The three SQWRL queries: FileCountforCPRules, FileCountforNCPRules and FinalChangeProneFiles; are again executed for the selection of the change-prone Java files of JFreeChart 0.7.1. However, each of these queries is slightly modified to suit the context.

- FileCountforCPRules

As read previously, query FileCountforCPRules displays the existing file instances corresponding to a version. It also captures their file name and their total count with respect to each of the change-prone rules derived from its previously released versions. Therefore, for finding out the change-proneness count of every Java file in JFreeChart 0.7.1, the existing query is modified to consider Java files that only belong to version JFreeChart 0.7.1. Also, as seen in the ontology, the classes ChangeProneFilesJFC060 and ChangeProneFilesJFC070 are essentially the sub-classes of the ChangeProneFiles class. Therefore, in order to obtain the cumulative count of the individual JFreeChart0.7.1 Java files that are inferred to be change prone as per the change-proneness governing rules of JFreeChart0.6.0 and JFreeChart0.7.0, we include the superclass ChangeProneFiles to link the relevant ?cp files to the Java Files(?files). This is done keeping in mind that any sub-classes of a Domain class for an Object Property, also become a Domain for that Object Property. The SQWRL query for the same is as follows:

```
CPCS:hasCPFilesAs(?cp, ?files) ^ CPCS:ChangeProneFiles (?cp) ^
CPCS:hasFileName(?files, ?filename) ^ CPCS:JavaFiles(?files) ^
CPCS:belongsToVersion(?files, "JFreeChart0.7.1")  .
sqwrl:makeSet(?s, ?cp) ^ sqwrl:groupBy(?s, ?files)  .
sqwrl:size(?CountAsPerCPRules, ?s) -> sqwrl:select(?files,
?filename, ?CountAsPerCPRules)
```

- FileCountforNCPRules

As read previously, query FileCountforNCPRules displays the file instances that exist corresponding to a version. It also captures their file name and their total count with respect to each of the non-change-prone rules derived from its previously released versions. Thusly, for finding out the not change-proneness count of every Java file in

JFreeChart0.7.1, the existing query is modified to consider Java files that only belong to version JFreeChart0.7.1. Also, as seen in the ontology, the classes NotChangeProneFilesJFC060 and NotChangeProneFilesJFC070 are essentially the sub-classes of the NotChangeProneFiles class. Therefore, in order to obtain the cumulative count of the individual JFreeChart0.7.1 Java files that are inferred to be "not change prone" as per the non-change-proneness governing rules of JFreeChart0.6.0 and JFreeChart0.7.0, we include the superclass NotChangeProneFiles to link the relevant ?ncp files to the Java Files(?files). The SQWRL query for the same is as follows:

```
CPCS:hasNCPFilesAs(?ncp, ?files) ^ CPCS:ChangeProneFiles (?ncp) ^
CPCS:hasFileName(?files, ?filename) ^ CPCS:JavaFiles(?files) ^
CPCS:belongsToVersion(?files, "JFreeChart0.7.1")  .
sqwrl:makeSet(?s, ?ncp) ^ sqwrl:groupBy(?s, ?files)  .
sqwrl:size(?CountAsPerNCPRules, ?s) -> sqwrl:select(?files,
?filename, ?CountAsPerNCPRules)
```

- FinalChangeProneFiles

Query FinalChangeProneFiles essentially employs the computed change-prone and not change-prone counts of each of the files, calculated using the queries FileCountforCPRules and FileCountforNCPRules, and compares these counts with respect to each file. The Java file having the CountAsPerCPRules higher than or equal to its CountAsPerNCPRules is selected as a Java file which has the possibility to be used with change in its upcoming version. As per our current context of application, the query displays the Java file instances corresponding to JFreeChart 0.7.1 that are selected to be change-prone in their successive version along with their file name.

```
CPCS:hasCPFilesAs(?cp, ?files) ^ CPCS:hasNCPFilesAs(?ncp, ?files)
.  sqwrl:makeSet(?s, ?cp) ^ sqwrl:makeSet(?s1, ?ncp) ^
CPCS:ChangeProneFiles(?cp) ^ CPCS:NotChangeProneFiles(?ncp) ^
CPCS:hasFileName(?files, ?filename) ^ CPCS:JavaFiles(?files) ^
CPCS:belongsToVersion(?files, "JFreeChart0.7.1") ^
sqwrl:groupBy(?s, ?files) ^ sqwrl:groupBy(?s1, ?files)  .
sqwrl:size(?CountAsPerCPRules, ?s) ^
sqwrl:size(?CountAsPerNCPRules, ?s1)^
swrlb: greaterThanOrEqual(?CountAsPerCPRules, ?CountAsPerNCPRules)
-> sqwrl:select(?files, ?filename)
```

Figures 5.18 and 5.19 show the results of the FinalChangeProneFiles query. Figure 5.18 indicates that the query returns 40 rows, thus exhibiting that 40 Java files are selected to be change-prone from the 122 Java files in JFreeChart 0.7.1, according to

the rules derived from its previous versions JFreeChart 0.6.0 and JFreeChart 0.7.0. Figure 5.19 displays the output from the query which consists of the 40 selected change-prone file instances and their filenames.

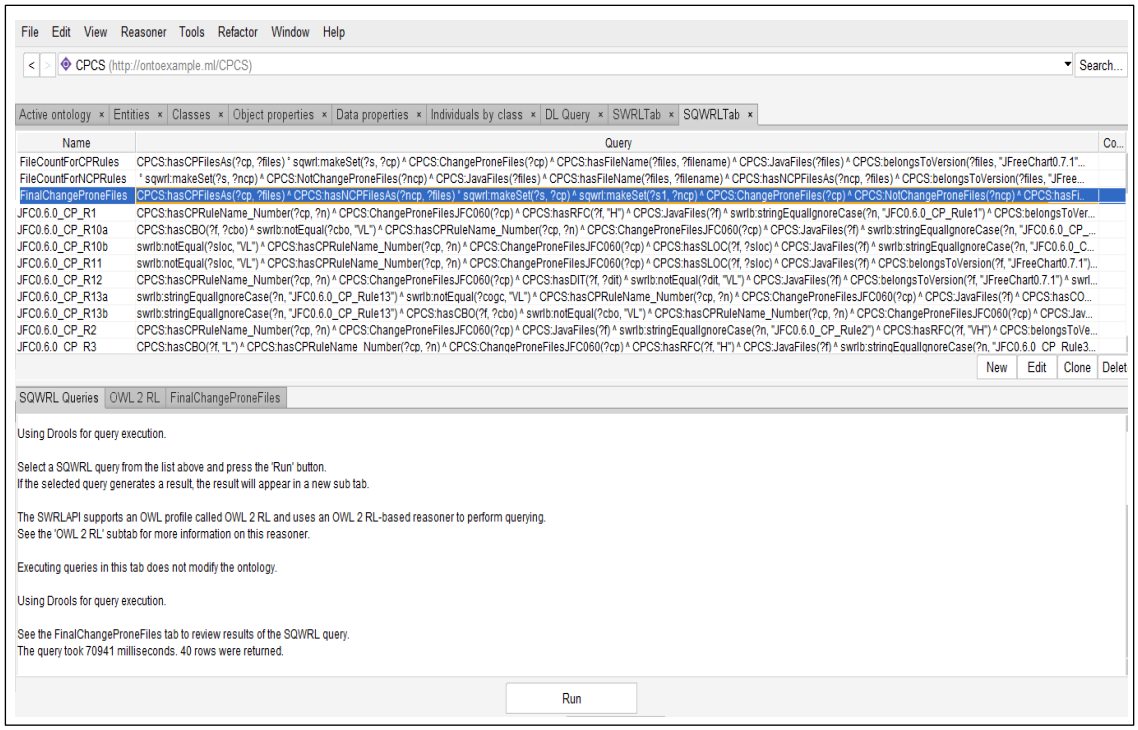

Figure 5.18 Screenshot of successful execution of the FinalChangeProneFiles query

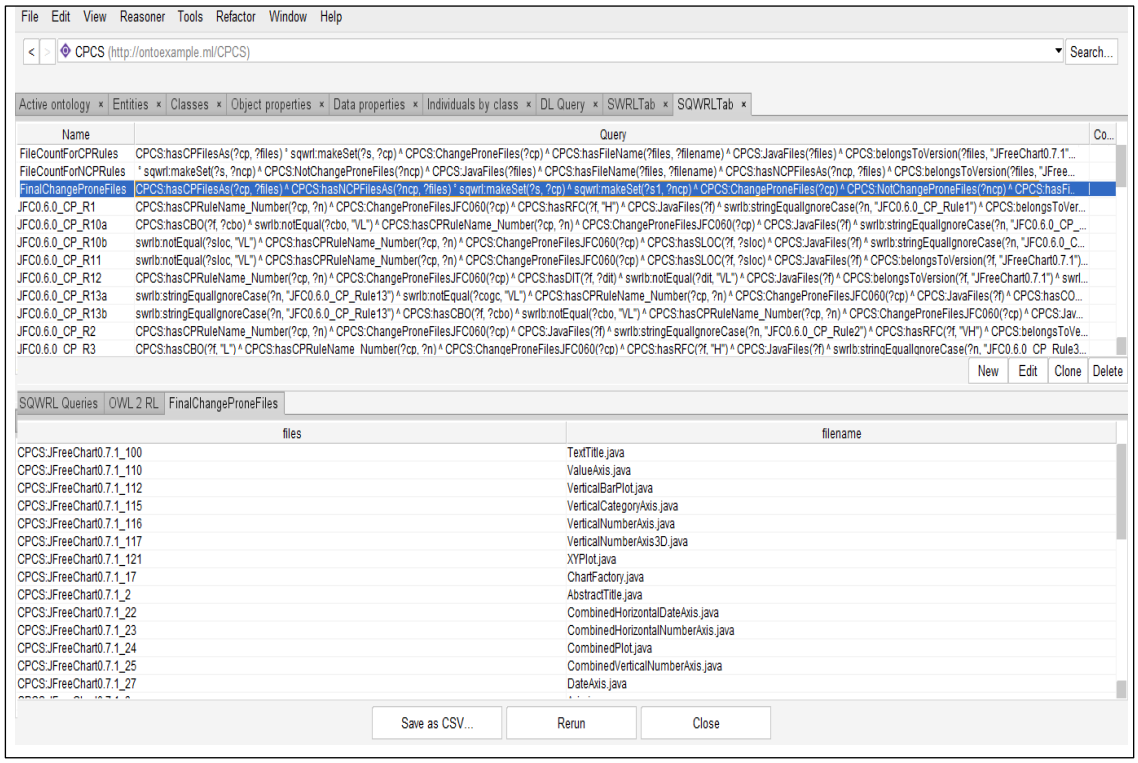

Figure 5.19 Screenshot of the final 40 change–prone files of JFreeChart 0.7.1 dervied via the FinalChangeProneFiles query

This process is repeated for all the remaining JFreeChart versions. The Heritrix ontology is also built in similar fashion and the screenshots from the results for Heritrix 0.4.0 and Heritrix 0.6.0 have been added in Appendices E and F.

5.5 Evaluation of the proposed CPJFS mechanism

Accuracy is a very important performance measure for a mechanism that claims to select change-prone Java files. The efficacy of the proposed Semantic web-based CPJFS mechanism has been evaluated using the results that were drawn via the eight version datasets of JFreeChart and Heritrix software projects. The mechanism attempted to select change-prone Java files of JFreeChart version 0.7.0 to 0.7.3 and Heritrix versions 0.4.0 to 0.10.0. As mentioned earlier, the SWRL rules derived from the RF algorithm are employed in the ontology in an incremental manner for selection of change-prone Java files. That is, the SWRL rules with respect to change-proneness prediction generated till Version $_{i-1}$ are employed, if one needs to select the Java files of a Version $_{i}$ that will be used with change in Version $_{i+1}$.

The results drawn from the proposed CPJFS mechanism have been given in Table 5.12, along with the actual change statistics and those obtained after the application of the ML[7] technique. Recall represents the change-prone files predicted/ selected which actually were change-prone and accuracy indicates the percentage of Java files that are correctly predicted/ selected to the total number of Java files that exist in the version.

A high recall by a selection or prediction process indicates that most of the relevant results are selected by the mechanism, whereas a high precision suggests that the mechanism generated significantly more relevant results than irrelevant ones [230-232].

Evaluation of a selection or prediction mechanism via accuracy as a performance measure is suitable only when both false positive and false negative costs are similar (which is unlikely to be the case) [230].

[7] These results have been obtained after performing the inter release validation of the selected plugin versions after the datasets were balanced, normalized and discretized. i.e. JFreeChart version 0.6.0 is employed to train the RF model and then the developed model is further validated on JFreeChart version 0.7.0. Subsequently, JFreeChart version 0.7.0 is employed to train the RF model and then the developed model is further validated on JFreeChart version 0.7.1 and so on.

Table 5.12 Performance statistics of the proposed Semantic web-based CPJFS mechanism for eight plugin versions.

Plugin version	Actual Change-Prone statistics	ML result statistics			CPJFS result statistics		
		Number of Change-Prone files selected	*Recall*	*Acc.*	*Number of Change-Prone files selected*	*Recall*	*Acc.*
JFreeChart0.7.0	60	26	40.0	63.7	38	66.1	81.4
JFreeChart0.7.1	28	20	31.0	72.8	40	75.9	77.9
JFreeChart0.7.2	18	87	38.9	63.8	35	61.1	75.6
JFreeChart0.7.3	38	103	55.3	80.7	14	31.6	78.5
Heritrix0.4.0	85	60	79.4	61.5	116	92.9	74.8
Heritrix0.6.0	67	72	75.4	69.1	89	77.6	73.3
Heritrix0.8.0	124	66	45.2	63.2	125	80.6	76.7
Heritrix0.10.0	60	148	66.7	63.7	132	92.6	69.8

A common aim of every such mechanism, be it prediction or selection, would be to ensure high values of both recall and precision. But prediction methods can be adjusted give importance to one metric over the others in a way that either the prediction methodlogy can be precise or sensitive. However, this decision primarily depends on the business goal. Foe example, in those cases where the mechanism needs to detect a fraud, recall is a more suitable or preferred metric since it is definitely essential to identify every conceivable fraud even if that entails the possibility of going through some false positives [233]. Conversely, if the goal we are dealing with is sentiment analysis wherein we need a high-level idea of emotions depicted in tweets then precision is a more suitable metric [232].

As mentioned in the earlier chapters, change-proneness [8, 14, 86] is a quality indicating feature of software that depicts its likelihood of changing in the future because of: (a) bug fixing, (b) evolving requirements, or (c) ripple effects. Change proneness has been often linked to several negative consequences along software evolution [6]. For instance, change-prone softwarwe artifacts are likely to also have

bugs, and therefore hoard high technical liability [6, 7]. Therefore, it becomes of paramount importanve to identify and monitor those parts of the system that are change-prone. This is the reason why we evaluate the efficacy of the proposed mechanism using recall as a performance measure.

From the results of Table 5.12, it can be observed that for most of the version datasets, the proposed CPJFS mechanism selects more number of files to be change-prone than the actual change prone statistic. This in turn increases the value of recall as the chances of an actual change-prone file to get selected as change-prone by the proposed mechanism increase.

That being said, our purpose is not to improve the accuracy of selection with respect to that obtained by RF, since the SWRL rules being employed are those generated from RF. There is no algorithm or rote translation you can make from ML to Semantic web to increase the accuracy but there are clearly advantages that can be gained when you integrate the two paradigms [227].

However, the accuracy values are also observed to be higher than the values obtained by the ML technique of Random Forest (RF). This could be due to the fact that we employ rules in an incremental manner for the proposed CPJFS mechanism wherein the SWRL rules (obtained from RF) with respect to change-proneness prediction generated till Version $_{i-1}$ are employed, if one needs to select the Java files of a Version $_{i}$ that will be used with change in Version $_{i+1}$.

Having evaluated the proposed CPJFS mechanism, the next section summarises the chapter.

5.6 Summary

The purpose of this chapter was to propose a mechanism for Change-Prone Java file Selection (CPJFS) of software components. Ontology, the chief constituents of Semantic web, is employed to develop this selection mechanism. The ontology was constructed in Protégé corresponding to the Java files contained in the five versions of JFreeChart and five versions of Heritrix plugin projects (software components) using OWL as the description language. The developed ontology consisted of three classes, two object properties and twelve data properties The seven metrics validated to be the most useful for the JFreeChart and Heritrix plugin projects for the prediction of

version to version change-proneness of Java files were selected and added to the data properties of each of the Java file in the ontology after normalization and discretization.

In order to classify the relevant change-prone Java files present in the ontology, the rules generated by the RF technique with respect to each version analysed for change-proneness prediction were employed as SWRL rules in the ontology in an incremental manner for selection of change-prone Java files. In order to generate the rules from the results of the RF algorithm, each path in the decision tree was traced from root node to leaf node and the test outcomes were recorded as antecedents and the leaf-node classification as the consequent. Additionally, instead of including all the rules in the proposed ontology for Java files, rules derived from RF were further screened according to their individual strength of accuracy.

Post the derivation of the inferred relationships which stated so as to which Java files from Version $_{i}$ are change-prone and not-change prone as per the shortlisted rules from Version $_{i-1}$, the populated ontology was then queried to find the possibility of change-proneness and non-change-proneness of each of the Java files in Version $_{i}$ using SQWRL. Finally, if the computed change-proneness count of a file was found to be higher than its non-change-proneness count, it was selected as change-prone.

The chapter concluded with a comparative study of the proposed change-prone Java file selection (CPJFS) mechanism results of the selected versions against the actual change statistics of the versions. Results demonstrated that the proposed mechanism selected the change-prone Java files of the selected JFreeChart and Heritrix versions with high recall, higher than that delivered by the RF algorithm on the versions. With that being said, our primary motive was to propose a mechanism for the selection of change-prone Java files present in various versions of software components. There was adequate literature demonstrating how software components could leverage the semantic capabilities of Semantic web technologies such as ontology to develop reuse repositories, thus acting as an aid for the specification and description of software components [65-70]. The sole purpose of this chapter was to act as an extension to this research by describing how the source code related information within the software components could also be added to the ontologies and how reasoning (using rules from an ICM) and querying could be done to select the change-prone source code components. The evaluation of the proposed mechanism clearly shows us that

even though the performance results could not be improved for all the versions, but a competent change-prone selection mechanism can be developed by integrating the two very diverse paradigms of Semantic web technologies and Intelligent Computing Methods like the ML technique of Random Forest.

Having satisfied all the research objectives, the next chapter concludes this thesis and presents future directions.

CHAPTER 6
CONCLUSION AND FUTURE WORK

This chapter begins by stating the threats to the validity of the research work presented in this thesis. This followed by a summary and discussion on the future directions in the domains of software change prediction in general and Semantic web-based software change estimation in particular. In the summary section, we give an overview of the work conducted, the effect of our empirical study, and the discourse on how our research satisfied the proposed objectives. The future work section emphasises on the possible outlets to further polish and improve the conclusions drawn in the empirical examinations of this research.

6.1 Threats to validity

The most important threats to the work done have been categorized as construct, internal, and external, and are customary in approximately all of the experiments in the literature that aim at carrying out an empirical analysis.

Threats to Construct validity: The chief threat to the construct validity is related to the authenticity of the variables utilized to develop the ontology to select the change-prone Java files. The independent variables that have been employed in this chapter, or rather in this research work, have been previously introduced in the literature. Former studies [11, 20, 82, 107] contain a validation for these metrics, that is if they are a true measure of the theories they imply. Moreover, the consistency in the empirical results drawn validated the contribution of the proposed CogC metric as a distinguishable measure of version to version source code change. Nonetheless, the employment of the seven best adjudged metrics as datatype properties in this chapter, open for comparison, is viable as previous findings [9, 24, 25, 48, 51, 61, 91, 98] have employed a majority of these metrics for the estimation of change-prone source code elements. Thusly, there is no threat to the construct validity of our work.

Threats to Internal validity: The internal validity denotes the magnitude to which precise implications can be drawn apropos to the contributory consequence of the independent variables on the change statistic. The ontology in this chapter has been

constructed using the data gathered from the source code files of consecutive versions of JFreeChart and Heritrix plugin projects, which are two of the many open source projects hosted on SourceForge.net. The aim of our research work is to not claim causation of the independent variables employed as datatype properties vis-à-vis software change, but to:

(i) depict relations between the selected independent variables (chiefly the CogC metric) and the change statistic of a Java file for the purpose of building efficient software change prediction models in Chapter 3,

(ii) to assess the performance of various prediction techniques during the two validation settings using the selected independent variables and the change statistic of a Java file with the aim of constructing prediction models for software change in Chapter 4, and

(iii) to build a change-prone Java file selection mechanism using independent variables validated to be effective for change-proneness prediction of Java files as datatype properties in this chapter.

Hence, the threat to internal validity does not exist in this research work.

Threats to External validity: Threats to external validity concern with the generality of the results acquired, more so, in terms of the mechanism performing with a comparable accuracy in selecting change-prone files of other projects. The threat to external validity could chiefly pertain to the plugin projects and versions selected for the purpose of validating the proposed change-prone Java file selection mechanism. The successional versions are selected from two software plugin projects: JfreeChart and Heritrix containing Java files, and there are many other projects available online from which data sets can be constructed in order to build ontology of Java files of software components.

However, most of the conclusions drawn are applicable to the OO standard in general:

- For instance, in Chapter 4, the BCSs involved in the calculation of the cognitive complexity metric represent some fundamental concepts common to all OO languages like C++, Python, Smalltalk etc. However, the keywords employed to depict these concepts might differ from language to language. For instance, Java handles exception-handling using the *try-catch-finally* block, whereas Python utilizes the *try-except-finally* block to perform the exception-

handling task, thus implying that exceptions that may be caught, go into an *except* block in Python much like the Java *catch* equivalent.

Having said that, there are certain limitations to this applicability:

- Although Ruby, Python and Java etc. are all OO languages, the some of the pragmatic inferences made via an OO language like Java could not entirely be applicable to others. This is because of the inherent syntax difference between the OO languages. For example, SLOC, which is one of the seven features selected to be highly correlated to software change in the research work conducted might not work for predicting change proneness in Python or Ruby code. This is because Python has rich built-in high-level data types and even supports dynamic typing, like Ruby. This is the most significant difference and affects how you design, write, and troubleshoot programs in a fundamental way, albeit making it one of the new preferred choices of programmers as they have to write less code. But same is not the case with Java, as developers are required to define the type of each variable before using it.

Hence, the results drawn in this research work could be limited to software components with Java files. In addition to this, the choice of software metrics utilized for the construction of datatype properties could also pose as another threat to external validity. However, these metrics have been included in the ontology only after conducting the CBFS in Chapter 4 where they are found to be most appropriate for predicting the software change of Java files for the two given plugin projects. Additionally, the SWRL rules constructed in the ontology to classify the change-prone and not change-prone files have been derived from the RF technique also adjudged to be the best for predicting the change-prone Java files for the two selected plugin projects. Hence, there exists a partial threat to the external validity of this work owing to the bespoke nature of the ontology proposed specific to the selected plugin projects.

6.2 Research Summary

The primary aim of this thesis work was to offer a mechanism for selection of change-prone Java files from software component versions using Semantic web technologies and Intelligent Computing Method. Additionally, we also wanted to verify if there

existed a cognitive aspect to the software change process. The identified research problem was divided into various sub-problems stated as objectives of this work.

The *first sub-problem* was resolved by conducting an extensive literature review in Chapter 2. The literature was primarily conducted with respect to the main topics covered in our research work and was segregated into three primary sections. The first Section presented the software change prediction in general and reported an outline of previous research published pertaining to the software metrics used and prediction techniques employed for software change. Having read about this, the second section discussed the various cognitive complexity metrics reported in literature, highlighting the role that cognitive complexity metric plays in software change and software development in general. The third section of this chapter presented an exhaustive and systematic literature survey researching the applicability of Semantic web technologies in performing the various software component-based tasks.

The probe into the *second sub-problem* resulted in a study elaborated in Chapter 3, focussing on examining if the cognitive complexity involved in analyzing a source code component (Java files in our context), belonging to a particular version of a software, could predict its change-proneness for the next release. This cognitive complexity metric was estimated via the cognitive weights which take the internal structural flow of the software into account to measure the extent of effort required and difficulty in understanding a given software. Multiple successively released versions of two plugin projects (JFreeChart and Heritrix) were selected and the cognitive complexity corresponding to each of the version's source code files was calculated. Statistical and ML models were built with the motive to draw inferences regarding the importance of the CogC metric with respect to the version to version source code change-proneness. Eight commonly used change prediction measures were also evaluated on the same eight datasets to provide an exhaustive comparative analysis and AUC, with the aid of Friedman's statistical test, were primarily used to compare and rank the relative predictive capability of the each measure.

The derived empirical results exhibited some interesting observations regarding the calculated cognitive complexity (CogC) metric, with it outperforming certain commonly employed metrics for change prediction like Cumulative Halstead Effort (CHalsE), Depth of Inheritance Tree (DIT), Lack of Cohesion in Methods (LCOM) and Number Of Children (NOC) for both the target projects analysed. Additionally,

significant Friedman test results indicated that the CogC metric exhibits comparable yet competitive predictive ability against metrics like Cyclomatic Complexity (CycloC) and Response For Class (RFC), again for both the software projects. The feature selection results also validated the contribution of the CogC metric to the accuracy of the change-proneness prediction models generated. All the results obtained in Chapter 3 provided enough testaments to the fact that although the CogC metric individually cannot be adjudged as the best quantifier of version to version change-proneness of source code files, its contribution as a distinguishable measure of version to version source code change cannot be denied.

Towards the *third sub-problem*, we extended our analysis pertaining to software change prediction by proposing, implementing and validating a thoroughly pervasive framework in Chapter 4. The framework was implemented using 31 prediction techniques (statistical and ICM) on datasets gathered from multiple successional releases of the same target projects employed in Chapter 3. Apposite pre-processing techniques were applied on the datasets and six performance measures were employed to depict the predictive performance of the prediction techniques. The performance of the developed prediction models was validated using two scenarios: k-fold validation (VS1) and inter-release validation (VS2), where the latter is useful for estimating the trend of change-proneness of files in the upcoming versions. Additionally, appropriate tests were employed for assessing the statistical significance of the acquired results and to gather valid and generalized inferences.

In addition to empirically establishing CBO, EC, CogC, CHV, RFC, DIT and SLOC as significant predictors of software change, the work authenticated the overall predictive potency of the selected prediction techniques with respect to change-proneness prediction of Java files over JFreeChart and Heritrix releases under the intra-release (VS1) and inter-release (VS2) validation scenarios. The Kruskal-Wallis test results however indicated that the employment of SMOTE considerably augmented the predictive ability of the models against those situations where no feature selection was applied. Also, although the prediction techniques demonstrated their capability for estimating the version to version trend proneness of Java files for most of the selected versions during inter-release validation (VS2), their results were inferior in comparison to those obtained during CBFS+ 10-fold validation (VS1). Furthermore, certain previously unexamined techniques like the techniques like SMO

and SPEG, ICM of MOEFC, the fuzzy classifier FLR, techniques under the miscellaneous category like CHIRP and VFI were found to be highly unsuitable for prediction of version to version change-proneness of Java files. Overall, the RF technique performed the best for VS1 when all the performance measures were taken into consideration with the Borda count method post the Scott-Knott analysis and was concluded to be the best prediction technique among the selected techniques for version to version change-proneness prediction of Java files over the selected JFreeChart and Heritrix versions. Moreover, the BN technique outperformed all other techniques in regard to all the selected performance measures for predicting the trend of version to version software change prediction of Java files during VS2, which was followed by the RF technique.

The *fourth and final sub-problem* was undertaken via proposing, implementing and evaluating a novel mechanism in Chapter 5 for a scalable Change-Prone Java File Selection (CPJFS) which employs Semantic web technology. Authors in the past literature elaborate how a Semantic web technology such as ontology can be employed to represent qualitative features of software housed in Web-based software repositories for an efficient and accurate matching and selection of required software. The work performed in Chapter 5 only attempted to extend the past research with respect to modelling software features using ontologies by providing insights on how software developers and practitioners could model supplementary information related to source code components of a software component at a finer granularity (file level, in our case) in an ontology-based software repository. Also, the mechanism demonstrated how information (in the form of patterns or rules) which is constructed by an ICM such as an ML technique to make relevant predictive decisions could be expressed in the form of valid SWRL rules in ontology.

Software component ontologies in Protégé were constructed corresponding to the Java files contained in the JFreeChart and Heritrix software project versions (analysed in Chapter 4). OWL was selected as the description language for modelling the relevant change-proneness related contextual information. SWRL and Drools Rule Engine were employed to analyse the contextual information that would subsequently infer these Java files as change-prone or stable. Finally, SWQRL was employed to query the now populated ontology and select the relevant change prone Java files. The chapter concluded with a comparative study of the proposed mechanism results

against the actual change statistic of the software project versions. The results obtained from the proposed Semantic web-based CPJFS mechanism indicated that even though the performance results could not be improved for all the versions, but a comptent change-prone source code component selection mechanism could be developed by integrating the two very diverse paradigms of Semantic web technologies and Intelligent Computing Methods like the ML technique of Random Forest.

Researchers and practitioners from the software industry can effectually utilize the results generated and observations drawn from this work. This is because, the pragmatic analyses performed in this thesis, in the bounds of its validity, provide:

- A critical empirical evaluation of the role human cognition plays in change-proneness of a particular software, thinking from the perspective of the developers working to maintain and upgrade a plethora of software daily.
- A repeatable empirical framework and confirmation with respect to the ability of various ICMs and statistical methods in predicting version to version change-proneness of Java files of software components.
- A scalable mechanism based on ontologies to select the change-prone Java files from software components.

Experts from software industry and academic researchers can extensively make use of the findings from the prediction models conceptualized in this study to efficiently allocate resources by concentrating on the change-prone files identified. These source code elements happen to be plausible sources of current defects and future modifications. This makes a business case for such files to be provided higher resources and attention in the early phase of the development life cycle to ensure a quality product. In particular, the inter-release validation conducted imitates the situation that is normally encountered in an actual software maintenance milieu, wherein historical data from the previous versions is utilized for training a model which in turn predicts change-prone components of a new release. Moreover this study also establishes the predictive ability of a previously unexamined ML technique: SYSFOR, which is seen to be one of the top five techniques for both the validation scenarios. Additionally, the proposed Semantic web-based mechanism essentially presents a platform for future researchers and practitioners to utilize ontologies as a means of transforming existing web-based software repositories

housing software versions from content management systems to knowledge-bases wherein information regarding software version histories is not only stored but also processed for making relevant software maintenance decisions. This approach would, therefore, make it handy for the software industry to secure the right and adequate resources for maintenance activities driving cost efficiency and quality simultaneously.

6.3 Future Work

As a part of the future work, one could expand the research conducted in this thesis in the directions stated below:

- The change-proneness statistic employed in this work was a binary variable simply indicating if a source code component (Java file in our case) of one version was employed in the second release with a change or not. Future work related to software change could focus on predicting software change using a categorical change statistic (depicting the type or degree of change) as the dependent variable. Since the number of modifications in a code component is proportional to the component's expected number of faults, an evaluation of the software metrics against a dependent variable signifying the degree of change ("no change envisioned", "subtle changes envisioned", "moderate changes envisioned", "heavy changes envisioned") would aid in further suggesting if the metric can be used to envision the number of bugs in the future releases.
- Additionally, an accurate prediction of types of changes provide certain insights on the overall software design and aids in observing the evolution of stability across consecutive versions. For example, a high number of perfective changes predicted corresponding to the source code components of a software could be indicative of a high coupling between the components that may need to be reduced. Also, common preventive changes include optimizing, restructuring the code and revising documentation. Predicting preventive changes condenses the total unanticipated effects a software can have in the long-term and aids it in becoming, stable, scalable, maintainable and understandable.

- Different metrics have individual features which could be exploited for software change prediction. As a part of the future work, other developer related metrics can be employed versus and along with the Cognitive Complexity metric as independent variables for the prediction of version to version change-proneness of source code files. Such metrics have been recently validated to be useful for change-proneness prediction of classes [6] and measure specific developer related factors like:
  - The entropy of changes applied by developers in a fixed time window. A complicated development process is expected to be present in case the changes' entropy is high. In such scenarios the developers may make modifications in a manner which is not disciplined and can result in the source code becoming hard to maintain and even more prone to changes in the subsequent versions.
  - The scattering of developers, both from the structural as well as the semantic standpoint working on an element of the code in a stipulated period of time, taken as indicators, and
  - The count of unique developers working on a specific part of the source code in a stipulated period of time.
- Multiple versions of two Java-based software projects having reasonable number of records were considered in this study. Software projects effectuated in a variety of other OO languages can be used to substantiate the derived results.
- Perhaps, since Semantic web is an emerging area, straightforward and freely accessible software engineering applications that employ Semantic web technologies for any of their software engineering tasks are limited. The KOntoR approach [77], one of the fewest Semantic web-based approaches which makes use of a knowledge base for storing semantic descriptions of software projects and software artefacts and runs SPARQL-based semantic queries on it to retrieve particular software, is an ideal example which ought to be followed. More of such applications are needed which hold the capability to make a change in the manner in which software and software repository information is managed. Finally, additional case studies could be conducted to determine the effectiveness of the proposed concept of amalgamating

Intelligent Computing Methods and Semantic web technologies in different types of applications and domains.

- Additionally, the ontology and rules-model proposed in Chapter 5 to select the change-prone Java files from software components is scalable since there can be an easy addition of novel contextual information and reasoning on the already added information. It could be particularly extended as a part of future work to support enterprise-level data analytics research related to a software change management system that captures multiple aspects of the data processing pipeline's expert knowledge, similar to the recently proposed Research Variable Ontology [234]. This knowledge repository could be designed to store metrics that impact software change, link domain knowledge related to data, facts and findings related to ML models that work best on certain data sourced for software change and information about available resources (i.e., data sources, ML models and transformations) together. The data analysts can interrogate the knowledge repository to learn and get recommendations from accumulated knowledge and use this knowledge for their analysis."

REFERENCES

1. A. K. Tripathi, R. Gupta, and M. Gupta, “Some observations on software processes for CBSE,” *Software Process: Improvement and Practice*, vol. 13, no. 5, pp. 411–419, 2008.
2. M.L. Griss, "CBSE Success Factors: Integrating Architecture, Process, and Organization," in *Component-Based Software Engineering: Putting the Pieces Together.* Addison Wesley, 2001, pp. 143-160.
3. A. K. Tripathi, “Some Pertinent Issues and Considerations on CBSE,” *International Journal of Social, Behavioral, Educational, Economic, Business and Industrial Engineering*, vol. 9, no. 2, pp. 693-700, 2015.
4. I. Crnkovic, J. Stafford, and C. Szyperski, “Software Components beyond Programming: From Routines to Services,” *IEEE Software*, vol. 28, no. 3, pp. 22–26, 2011.
5. T. Ravichandran and M. A. Rothenberger, “Software reuse strategies and component markets,” *Communications of the ACM*, vol. 46, no. 8, pp. 109–114, 2003.
6. G. Catolino and F. Ferrucci, “Ensemble techniques for software change prediction: A preliminary investigation,” in *Proceedings of the 2018 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*. IEEE, 2018, pp. 25-30.
7. G. Catolino, F. Palomba, A. D. Lucia, F. Ferrucci, and A. Zaidman, “Enhancing change prediction models using developer-related factors,” *Journal of Systems and Software*, vol. 143, pp. 14–28, 2018.
8. R. Malhotra and A. J. Bansal, “Software change prediction: a literature review,” *International Journal of Computer Applications in Technology*, vol. 54, no. 4, p. 240, 2016.
9. R. Malhotra and M. Khanna, “An exploratory study for software change prediction in object-oriented systems using hybridized techniques,” *Automated Software Engineering*, vol. 24, no. 3, pp. 673–717, 2017.
10. T. Honglei, S. Wei and Z. Yanan, “The research on software metrics and software complexity metrics,” in *Proceedings of the International Forum on Computer Science-Technology and Applications*. IEEE, 2009, pp. 131–136.
11. R. Malhotra and M. Khanna, “Investigation of relationship between object-oriented metrics and change proneness,” *International Journal of Machine Learning and Cybernetics*, vol. 4, no. 4, pp. 273–286, 2013.
12. D. Romano and M. Pinzger, “Using source code metrics to predict change-prone Java interfaces,” in *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2011 pp. 303-312.

13. R. Malhotra and A. Bansal, “Prediction of Change-Prone Classes Using Machine Learning and Statistical Techniques,” in *Proceedings of the Conference on Advanced Research and Trends in New Technologies, Software, Human-Computer Interaction, and Communicability*. IGI Global, 2014, pp. 193-202.

14. R. Malhotra and M. Khanna, “Software Change Prediction: A Systematic Review and Future Guidelines,” *e-Informatica Software Engineering Journal*, vol. 13, no. 1, pp. 227-259, 2019.

15. C. Vassallo, S. Panichella, F. Palomba, S. Proksch, A. Zaidman, and H. C. Gall, “Context is king: The developer perspective on the usage of static analysis tools,” in *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018 pp. 38-49.

16. M. Beller, G. Gousios and A. Zaidman, “Travistorrent: synthesizing Travis CI and GitHub for full-stack research on continuous integration,” in *Proceedings of the International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 447–450.

17. S.R. Chidamber and C.F. Kemerer, “A metrics suite for object oriented design,” *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

18. J. Bansiya and C.G. Davis, “A hierarchical model for object-oriented design quality assessment,” *IEEE Transactions on Software Engineering*, vol. 28, no. 1, pp. 4–17, 2002.

19. M. Lorenz and J. Kidd, *Object-oriented software metrics: A practical guide*. Prentice-Hall, Inc., 1994.

20. B. Curtis, S.B. Sheppard, P. Milliman, M.A. Borst and T. Love, “Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics,” *IEEE Transactions on Software Engineering*, vol. 2, pp. 96–104, 1979.

21. T.J. McCabe, “A complexity measure,” *IEEE Transactions on Software Engineering*, vol. 4, pp. 308–320, 1976.

22. J. Bieman, G. Straw, H. Wang, P. Munger, and R. Alexander, “Design patterns and change proneness: an examination of five evolving systems,” in *Proceedings of the 5th International Workshop on Enterprise Networking and Computing in Healthcare Industry*. IEEE, 2003, pp. 40-49.

23. A.G. Koru and H. Liu, “Identifying and characterizing change-prone classes in two large-scale open-source products,” *Journal of Systems and Software*, vol. 80, no. 1, pp. 63–73, 2007.

24. Y. Zhou, H. Leung, and B. Xu, “Examining the potentially confounding effect of class size on the associations between object-oriented metrics and change-proneness,” *IEEE Transactions on Software Engineering*, vol. 35, no. 5, pp. 607–623, 2009.

25. D. Posnett, C. Bird, and P. Dévanbu, “An empirical study on the influence of pattern roles on change-proneness,” *Empirical Software Engineering*, vol. 16, no. 3, pp. 396–423, 2011.

26. H. Lu, Y. Zhou, B. Xu, H. Leung and L. Chen, “The ability of object-oriented metrics to predict change-proneness: a meta-analysis,” *Empirical Software Engineering*, vol. 17, no. 3, pp. 200–242, 2011.

27. M.D. Penta, L. Cerulo, Y.G. Gueheneuc, and G. Antonio, “An empirical study of the relationships between design pattern roles and class change proneness,” in *Proceedings of the IEEE International Conference on Software Maintenance*. IEEE, 2008, pp. 217–226.

28. M.O. Elish and M. Al-Rahman Al-Khiaty, “A suite of metrics for quantifying historical changes to predict future change-prone classes in object-oriented software,” *Journal of Software: Evolution and Process*, vol. 25, no. 5, pp. 407–437, 2013.

29. M. Al-Khiaty, R. Abdel-Aal, and M.O. Elish, “Abductive network ensembles for improved prediction of future change-prone classes in object-oriented software,” *International Arab Journal of Information Technology*, vol. 14, no. 6, pp. 803–811, 2017.

30. R. Malhotra and M. Khanna, “Prediction of change prone classes using evolution-based and object-oriented metrics,” *Journal of Intelligent and Robotic Systems Fuzzy Systems*, vol. 34, no. 3, pp. 1755–1766, 2018.

31. F. Khomh, M. D. Penta, and Y.-G. Gueheneuc, “An Exploratory Study of the Impact of Code Smells on Software Change-proneness,” in *Proceedings of the 16th Working Conference on Reverse Engineering*. IEEE, 2009 pp. 75-84.

32. E. Giger, M. Pinzger, and H.C. Gall, “Can we predict types of code changes? An empirical analysis,” in *Proceedings of the 9th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2012, pp. 217–226.

33. M.O. Elish, H. Aljamaan, and I. Ahmad, “Three empirical studies on predicting software maintainability using ensemble methods,” *Soft Computing*, vol. 19, no. 9, pp. 2511–2524, 2015.

34. L. Kumar, R.K. Behera, S. Rath, and A. Sureka, “Transfer learning for cross-project change-proneness prediction in object-oriented software systems: A feasibility analysis,” *ACM SIGSOFT Software Engineering Notes*, vol. 42, no. 3, pp. 1–11, 2017.

35. R. M. Bell, T. J. Ostrand and E. J. Weyuker, “The limited impact of individual developer data on software defect prediction,” *Empirical Software Engineering*, vol. 18, no. 3, pp. 478–505, 2013.

36. D. Di Nucci, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto and A. De Lucia, “A developer centered bug prediction model,” *IEEE Transactions on Software Engineering*, vol. 44, no.1, pp. 5-24, 2017.

37. J. Börstler, M. E. Caspersen and M. Nordström, “Beauty and the Beast: on the readability of object-oriented example programs,” *Software Quality Journal*, vol. 24, no. 2, pp. 231-246, 2016.

38. W. Duch, “What is computational intelligence and where is it going?,” in *Challenges for Computational Intelligence (Studies in Computational Intelligence)*. Springer, pp. 1-13, 2007

39. A. Konar, *Computational intelligence principles, techniques, and applications*. New York: Springer, 2005.

40. A. Kusiak, *Computational intelligence in design and manufacturing*. New York: Wiley, 2000.

41. S. Dick and A. Kandel, *Computational intelligence in software quality assurance*. Series in Machine Perception and Artificial Intelligence, vol. 63, World Scientific 2005.

42. R.E. King, *Computational intelligence in control engineering*, Marcel Dekker, Inc., New York, 1999.

43. S.H. Chen, P. Wang, and P.P. Wang, *Computational Intelligence in Economics and Finance*. Advanced Information Processing Series, Springer 2006

44. R. Mohanta, R. Vadlamani and M. R. Patra, “The application of intelligent and soft-computing techniques to software engineering problems: a review,” *International Journal of Information and Decision Sciences*, vol. 2, no. 3, pp. 233-272, 2010.

45. A. Bansal, “Empirical analysis of search based algorithms to identify change prone classes of open source software,” *Computer Languages, Systems and Structures*, vol. 47, pp. 211–231, 2017.

46. L. Kumar, S.K. Rath, and A. Sureka, “Empirical analysis on effectiveness of source code metrics for predicting change-proneness,” in *Proceedings of the 10th Innovations in Software Engineering Conference*. ACM, 2017, pp. 4–14.

47. R. Malhotra and R. Jangra, “Prediction and assessment of change prone classes using statistical and machine learning techniques,” *Journal of Information Processing Systems*, vol. 13, no. 4, pp. 778–804, 2017.

48. R. Malhotra and M. Khanna, “An empirical study for software change prediction using imbalanced data,” *Empirical Software Engineering*, vol. 22, no. 6, pp. 2806–2851, 2017.

49. Y. Liu and T.M. Khoshgoftaar, “Genetic programming model for software quality classification,” in *Proceedings of the International Symposium on High Assurance Systems Engineering. Special Topic: Impact of Networking*. IEEE, 2001, pp. 127–136.

50. T.M. Khoshgoftaar, N. Seliya, and Y. Liu, “Genetic programming-based decision trees for software quality classification,” in *Proceedings of the 15th*

International Conference on Tools with Artificial Intelligence. IEEE, 2003, pp. 374–383.

51. N. Tsantalis, A. Chatzigeorgiou, and G. Stephanides, "Predicting the probability of change in object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 31, no. 7, pp. 601–614, 2005.
52. A.G. Koru and J. Tian, "Comparing high-change modules and modules with the highest measurement values in two large-scale open-source products," *IEEE Transactions on Software Engineering*, vol. 31, no. 8, pp. 625–642, 2015.
53. D. Azar, "A genetic algorithm for improving accuracy of software quality predictive models: a search-based software engineering approach," *International Journal of Computational Intelligence and Applications*, vol. 9, no. 2, pp. 125–136, 2010.
54. D. Azar and J. Vybihal, "An ant colony optimization algorithm to improve software quality prediction models: Case of class stability," *Information and Software Technology*, vol. 53, no. 4, pp. 388–393, 2011.
55. R. Malhotra and M. Khanna, "Mining the impact of object oriented metrics for change prediction using machine learning and search-based techniques," in *Proceedings of the International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. IEEE, 2015, pp. 228–234.
56. X. Zhu, Q. Song, and Z. Sun, "Automated identification of change-prone classes in open source software projects," *Journal of Software*, vol. 8, no. 2, pp. 361–366, 2013.
57. A. Agrawal and R.K. Singh, "Empirical validation of OO metrics and machine learning algorithms for software change proneness prediction," in *Proceedings of the Towards Extensible and Adaptable Methods in Computing conference*. Springer, 2018, pp. 69–84.
58. M. Yan, X. Zhang, C. Liu, L. Xu, M. Yang, and D. Yang, "Automated change-prone class prediction on unlabeled dataset using unsupervised method," *Information and Software Technology*, vol. 92, pp. 1–16, 2017.
59. L. Kumar, S. Lal, A. Goyal, and N. Murthy, "Change-proneness of object-oriented software using combination of feature selection techniques and ensemble learning techniques," in *Proceedings of the Innovations on Software Engineering Conference*. ACM, 2019, pp. 8-19.
60. Y. Ge, M. Chen, C. Liu, F. Chen, S. Huang, and H. Wang, "Deep metric learning for software change-proneness prediction," in *Proceedings of the International Conference on Intelligent Science and Big Data Engineering*. Springer, 2018, pp. 287–300.
61. L. Kaur, A. Mishra, "A comparative analysis of evolutionary algorithms for the prediction of software change," in *Proceedings of the 2018 International*

Conference on Innovations in Information Technology (IIT). IEEE, 2018, pp. 187-192.

62. R. Malhotra and M. Khanna, "Particle Swarm Optimization-Based Ensemble Learning for Software Change Prediction," *Information and Software Technology,* vol. 102, pp. 65-84, 2018.

63. L. Kumar, S.K. Rath, and A. Sureka, "Using source code metrics to predict change-prone web services: A case-study on ebay services," in *Proceedings of the Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*. IEEE, 2017, pp. 1–7.

64. I. Crnkovic, "Component-based software engineering challenges in software development," *Software Focus,* vol. 2, no. 4, pp. 127–133, 2001.

65. A. Alnusair and T. Zhao, "Retrieving reusable software components using enhanced representation of domain knowledge," in *Proceedings of the Conference on Recent Trends in Information Reuse and Integration*. Springer, 2012, pp. 363–379.

66. F. Siddiqui and M.A Alam, "Ontology based feature driven development life cycle," *International Journal of Computer Science Issues,* vol. 9, pp. 207–212, 2013.

67. M. Abbasipour, M. Sackmann, F. Khendek and M. Toeroe, "Ontology-based user requirements decomposition for component selection for highly available systems", in *Proceedings of the 15th International Conference on Information Reuse and Integration (IRI).* IEEE, 2014, pp. 44–51.

68. M. Dostal, M. Nykl and K. Jezek, "Semantic analysis of software specifications with linked data," *Journal of Theoretical and Applied Information Technology*, vol. 67, no. 2, pp. 368–376, 2014.

69. H. Samimi, C. Deaton, Y. Ohshima, A. Warth and T. Millstein, "Call by meaning," in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. ACM, 2014, pp. 11–28.

70. A. Konys, "A framework to cots component selection and evaluation processes," *Przegl Elektrotechniczny*, vol. 91, no. 2, pp. 85–88, 2015.

71. Y. Wu, R. A. Gandhi, and H. Siy, "Using semantic templates to study vulnerabilities recorded in large software repositories," in *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems - SESS 10*. ACM, 2010, pp. 22-28.

72. M.A. Eierman and M.T. Dishaw, "The process of software maintenance: a comparison of object-oriented and third-generation development languages," *Journal of Software: Evolution and.process,* vol. 19, no. 1, 2007, pp. 33–47.

73. A.J. Ko, H.H. Aung and B.A. Myers, “Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks,” in *Proceedings of the 27th International Conference on Software Engineering*. IEEE, 2005, pp. 126–135.

74. Z. Soh, F. Khomh, Y.G. Guéhéneuc and G. Antoniol, “Towards understanding how developers spend their effort during maintenance activities,” in *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. IEEE, 2013, pp. 152–161.

75. V. Sugumaran and V.C. Storey, “A semantic-based approach to component retrieval,” *ACM SIGMIS Database,* vol. 34, no. 3, pp. 8–24, 2003

76. W. Liu, K. He and W. Liu, “Design and realization of ebxml registry classification model based on ontology,” in *Proceedings of the International Conference on Information Technology: Coding and Computing*. IEEE, 2006, pp. 809–814.

77. H.J. Happel, A. Korthaus, S. Seedorf and P. Tomczyk, "KOntoR: An Ontology-Enabled Approach to Software Reuse,” in *Proceedings of the* 18TH *International Conference on software engineering and knowledge engineering*, pp. 349-354, 2006.

78. C. Li, X. Liu and J. Kennedy, “Semantics-based component repository: current state of arts and a calculation rating factor based framework,” in *Proceedings of the 32nd Annual IEEE International conference on Computer Software and Applications*. IEEE, 2008, pp. 751–756.

79. Z. Zygkostiotis, D. Dranidis and D. Kourtesis, “Semantic annotation, publication, and discovery of java software components: an integrated approach,” in *Proceedings of the AIAI-2009 Workshops*, 2009, pp. 168–178.

80. S. Khemakhem, K. Drira, and M. Jmaiel, “Semantic matching to achieve software component discovery and composition,” *Laboratory for Analysis and Architecture of Systems*, Technical. Report, 2013.

81. Y. Wang, J. Shao, “Measurement of the cognitive functional complexity of software,” in *Proceedings of the Second International Conference on Cognitive Informatics.* IEEE, 2003, pp. 67–74.

82. M. Crasso, C. Mateos, A. Zunino, S. Misra and P. Polvorín, “Assessing cognitive complexity in java-based object-oriented systems: metrics and tool support,” *Computing and Informatics*, vol. 35, no. 3, pp. 497–527, 2016.

83. H.H. Wang and J. Sun, “A semantic web environment for components,” *The Knowledge Engineering Review,* vol. 24, no. 1, pp. 59-75, 2009.

84. F. E. Castillo-Barrera, R. C. M. Ramırez, and H. A. Duran-Limon, “Knowledge capitalization in a component-based software factory: a semantic viewpoint,” in *Proceedings of the LA-NMR*, 2011, pp. 105–114.

85. K. K. Lau and V. Ukis, "Defining and checking deployment contracts for software components," *Component-based software engineering*, vol. 16, 2006.

86. J. Buckley, T. Mens, M. Zenger, A. Rashid, and G Kniesel, "Towards a taxonomy of software change," *Journal on Software Maintenance and Evolution*, vol. 17, no. 5, pp. 309-332, 2005.

87. T. Lindholm and F. Yellin, *The Java Virtual Machine*. MA: Addison-Wesley, 1999.

88. J. Kramer and J. Magee, "The evolving philosophers problem: Dynamic change management," *IEEE Transactions on Software Engineering*, vol. 16, no. 11, pp. 1293–1306, 1990.

89. G. Kniesel, J. Noppen, T. Mens and J. Buckley, "Unanticipated Software Evolution," in *Proceedings of the European Conference for Object Oriented Programming*. Springer, 2002, pp. 92-106.

90. N. Chapin, J. Hale, K. Khan, J. Ramil, and W.G. Than, "Types of software evolution and software maintenance," *Journal of software maintenance and evolution*, vol. 13, pp. 3–30, 2001.

91. M.A. Chaumun, H. Kabaili, R.K. Keller and F. Lustman, "A change impact model for changeability assessment in object-oriented software systems," *Science of Computer Programming*, vol. 45, no. 2, pp. 155–174, 2002.

92. J. M. Bieman, A. A. Andrews, and H. J. Yang, "Understanding change-proneness in OO software through visualization," in *Proceedings of the 11th IEEE International Workshop on Program Comprehension*. IEEE, 2003, pp. 44–53.

93. J. M. Bieman, G. Straw, H. Wang, P. W. Munger, and R. T. Alexander, "Design patterns and change proneness: An examination of five evolving systems," in *Proceedings of the 9th International Software Metrics Symposium (METRICS'03)*. IEEE, 2003, pp. 40–49.

94. E. Arisholm, L.C. Briand, A. Foyen, "Dynamic coupling measurement for object-oriented software," *IEEE Transactions on Software Engineering*, vol. 30, no. 8, pp. 491–506, 2004.

95. M. Askari and R. Holt, "Information theoretic evaluation of change prediction models for large-scale software," in *Proceedings of the International workshop on Mining software repositories*. ACM, 2006, pp. 126-132.

96. A.R. Sharafat, and L. Tahvildari, "A probabilistic approach to predict changes in object-oriented software system," in *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*. CMSR, 2007, pp. 27-38.

97. A.R. Han, S.U. Jeon, D.H. Bae, and J.E. Hong, "Behavioral dependency measurement for change-proneness prediction in UML 2.0 design models," in

Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference. IEEE, 2008, pp.76-83.

98. A.R. Sharafat and L. Tahvildari, "Change prediction in object-oriented software systems: a probabilistic approach," *Journal of Software*, vol. 3, no. 5, pp.26–39, 2008.

99. A.R. Han, S.U. Jeon, D.H. Bae, and J.E. Hong, "Measuring behavioral dependency for improving change-proneness prediction in UML-based design models," *The Journal of Systems and Software*, vol. 83, no. 2, pp.222–234, 2010.

100. S. Eski and F. Buzluca, "An empirical study on object-oriented metrics and software evolution in order to reduce testing costs by predicting change-prone classes," in *Proceedings of the Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2011, pp. 566–571.

101. R. Malhotra and M. Khanna, "A new metric for predicting software change using gene expression programming," in *Proceedings of the 5th International Workshop on Emerging Trends in Software Metrics*. ACM, 2014, pp. 8–14.

102. C. Marinescu, "How good is genetic programming at predicting changes and defects?" in *Proceedings of the 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE, 2014, pp. 544–548.

103. C. Liu, Y. Dan, X. Xin, Y. Meng, and Z. Xiaohong, "Cross-project change-proneness prediction," in *Proceedings of the 42nd Annual Computer Software and Applications Conference (COMPSAC)*. IEEE, 2018, pp. 64–73.

104. V. Antinyan, M. Staron, J. Derehag, M. Runsten, E. Wikström, W. Meding, A. Henriksson and J. Hansson, "Identifying complex functions: By investigating various aspects of code complexity," in *Proceedings of the Science and Information Conference (SAI)*. IEEE, 2015, pp. 879–888.

105. V. Antinyan, M. Staron and A. Sandberg, "Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3057–3087, 2017.

106. S.N. Cant, D.R. Jeffery and B. Henderson-Sellers, "A conceptual model of cognitive complexity of elements of the programming process," *Information and software technology,* vol. 37, no. 7, pp. 351-362, 1995.

107. S. Misra and A.K. Misra, "Evaluating cognitive complexity measure with Weyuker properties," in *Proceedings of the Third International Conference on Cognitive Informatics*. IEEE, 2004, pp. 103–108.

108. D.S. Kushwaha and A.K. Misra, "Improved cognitive information complexity measure: a metric that establishes program comprehension effort," *ACM SIGSOFT Software Engineering Notes*, vol. 31, no.5, pp. 1–7, 2006.

109. D.S. Kushwaha and A.K. Misra, “Robustness analysis of cognitive information complexity measure using Weyuker properties,” *ACM SIGSOFT Software Engineering Notes,* vol. 31, no. 1, pp. 1-6, 2006.

110. S. Misra, “Modified cognitive complexity measure,” in *Proceedings of the International Symposium on Computer and Information Sciences*. Springer, 2006, pp. 1050-1059.

111. S. Misra, “Validating modified cognitive complexity measure,” *ACM SIGSOFT Software Engineering Notes,* vol. 32, no. 3, pp. 1–5, 2007.

112. S. Misra and I. Akman, “A model for measuring cognitive complexity of software,” in *Proceedings of the International Conference on Knowledge-Based, Intelligent Information and Engineering Systems*. Springer, 2008, pp. 879–886.

113. B. Auprasert and Y. Limpiyakorn, “Structuring cognitive information for software complexity measurement,” in *Proceedings of the WRI World Congress on Computer Science and Information Engineering*. IEEE, 2009, pp. 830–834.

114. S. Misra, I. Aman and R. Palacios, “Framework for evaluation and validation of software complexity measures,” *IET Software*, vol. 6, no. 4, pp. 323–334, 2012.

115. P. Vitharana, “Risks and challenges of component-based software development,” *Communications of the ACM*, vol. 46, no. 8, pp. 67–72, 2003.

116. J. Cardoso, *Semantic Web Services: Theory, Tools and Applications: Theory, Tools and Applications*. IGI Global, 2007.

117. J. Cardoso, and A. Sheth, “The semantic web and its applications,” in *Semantic Web Services, Processes and Applications*. Springer, pp. 3–33, 2006.

118. B. A. Kitchenham, “Procedures for performing systematic reviews,” in *Technical Report TR/SE-0401, Keele University, and Technical Report 0400011T.1, National ICT Australia,* 2004.

119. N. Guarino, “Formal ontology and information systems,” in *Proceedings of the 1st International conference on Formal Ontology in Information Systems.* IOS Press, 1998, pp. 81–97.

120. T.R. Gruber, “The role of common ontology in achieving sharable, reusable knowledge bases,” in *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning.* Cambridge, 1991, pp. 601–602.

121. X. Wang, R. Gorlitsky and J.S. Almeida, “From xml to rdf: how semantic web technologies will change the design of ‘omic’ standards,” *Nature Biotechnology*, vol. 23, no. 9, pp. 1099–1103, 2005.

122. L. Quan, J. Xinjuan and L. Yihong, “Research on ontology based representation and retrieval of components,” in *Proceedings of the 8th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*. IEEE, 2007, pp. 494–499.

123. A.N. Sireteanu, “A survey of web ontology languages and semantic web services,” *Annals Alexandru Ioan Cuza University-Econ*, vol. 60, no. 1, pp. 42–53, 2013.

124. D.L. McGuinness and F. Van Harmelen, “Owl web ontology language overview,” *W3C Recommendation*, vol. 10, no. 10, 2004.

125. R.B. Mishra and S. Kumar, “Semantic web reasoners and languages,” *Artificial Intelligence Review*, vol. 35, no. 4, pp. 339-368, 2011.

126. N.F. Noy, R.W. Fergerson and M.A. Musen, “The knowledge model of protege-2000: combining interoperability and flexibility,” in *Proceedings of the International Conference on Knowledge Engineering and Knowledge Management*. Springer, 2000, pp. 17–32.

127. V. Haarslev and R. Moeller, “Description of the Racer System and Its Applications,” in *Proceedings of the International Workshop in Description Logics*, 2001, pp. 132–142.

128. B. Parsia, and E. Sirin, “Pellet: an owl dl reasoner,” in *Proceedings of the 3rd International Semantic Web Conference*, Citeseer, 2004.

129. D. Tsarkov, and I. Horrocks, “Fact++ description logic reasoner: system description,” *Automated Reasoning*, pp. 292–297, 2006.

130. M. J. O’Connor and A.K. Das, “Sqwrl: a query language for owl,” *OWLED*, vol. 529, pp. 1-8, 2009.

131. S. Kumar, and R.B. Mishra, “Semantic web service composition,” *IETE Technical Review*, vol. 25, no. 3, pp. 105-121, 2008.

132. C. Bizer, T. Heath, K. Idehen and T. Berners-Lee, “Linked data on the web,” in *Proceedings of the 17th International Conference on World Wide Web*. ACM, 2008, pp. 1265–1266.

133. R.M. Braga, M. Mattoso and C.M. Werner, “The use of mediation and ontology technologies for software component information retrieval,” in *ACM SIGSOFT Software Engineering Notes*. ACM, 2001, pp. 19–28.

134. L. Aroyo and D. Dicheva, “The new challenges for e-learning: the educational semantic web,” *Educational Technology and Scociety*, vol. 7, no. 4, pp. 59–69, 2004.

135. H. Yao and L. Etzkorn, “Towards a semantic-based approach for software reusable component classification and retrieval,” in *Proceedings of the 42nd Annual Southeast Regional Conference*. ACM, 2004, pp. 110–115.

136. J. Lu, Y. Yu and J. Mylopoulos, “A lightweight approach to semantic web service synthesis,” in *Proceedings of the International Workshop on Challenges in Web Information Retrieval and Integration*. IEEE, 2005, pp. 240–247.

137. C. Zhangjian and Q. Nen, “Research of a semantic-based approach to improve software component reuse,” in *Proceedings of the 1st International Conference on Semantics, Knowledge and Grid*. IEEE, 2005, pp. 93-100.

138. D. Song, W. Liu, Y. He and K. He, “Ontology application in software component registry to achieve semantic interoperability,” in *Proceedings of the Conference on Information Technology: Coding and Computing*. IEEE, 2005, pp. 181–186.

139. A. Korthaus, M. Schwind and S. Seedorf, “Leveraging semantic web technologies for business component specification,” *Web Semantics Science Services and Agents on the World Wide Web*, vol. 5, no. 2, pp. 130–141, 2007 .

140. J. Sun, H. Miao and X. Cao, “A domain formal ontology and the application in service component retrieval,” in *Proceedings of the International Conference on Software Engineering Advances*. IEEE, 2006, pp. 33-39.

141. Y. Ha and R. Lee, “ Integration of semantic web service and component-based development for e-business environment,” in *Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*. IEEE, 2006, pp. 315–323.

142. Z. Laliwala, R. Khosla, P. Majumdar and S. Chaudhary, “Semantic and rules based event-driven dynamic web services composition for automation of business processes,” in *Proceedings of the Services Computing Workshops*. IEEE, 2006, pp. 175–182.

143. D. Hyland-Wood, D. Carrington and S. Kaplan, “ A semantic web approach to software maintenance,” in *Proceedings of the 15th International Conference on Knowledge Engineering and Knowledge Management*, 2006, pp. 11–12.

144. M. Sjachyn and L. Beus-Dukic, “Semantic component selection-semacs,” in *Proceedings of the 5th International Conference on Commercial-off-the-Shelf (COTS)-BasedSoftware Systems*. IEEE, 2006, pp. 7–13.

145. S. Khemakhem, K. Drira and M. Jmaiel, “Sec: a search engine for component-based software development,” in *Proceedings of the Symposium on Applied computing*. ACM, 2006, pp. 1745–1750.

146. P. Graubmann and M. Roshchin, “Semantic annotation of software components,” in *Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE, 2006, pp. 46–53.

147. C. Pahl, “An ontology for software component matching,” *International journal on software tools for Technology Transfer*, vol. 9, no. 2, pp. 169–178, 2007.

148. R. Witte, Y. Zhang and J. Rilling, “Empowering software maintainers with semantic web technologies,” in *Proceedings of the European Semantic Web Conference*. Springer, 2007, pp. 37-52.

149. D. Retkowitz and M. Pienkos, “Ontology-based configuration of adaptive smart homes,” in *Proceedings of the 7th Workshop on Reflective and Adaptive Middleware*. ACM, 2008, pp. 11–16.

150. A. Talevski, P. Wongthongtham and S. Komchaliaw, “Towards a software component ontology,” in *Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services*. ACM, 2008, pp. 503–507.

151. S. Yang and J. Yan, “Ontology component development,” in *Proceedings of the International Conference on Computational Intelligence and Natural Computing*. IEEE, 2009, pp. 92–95.

152. W. Zhang, K.M. Hansen and J. Fernandes, “Towards open world software architectures with semantic architectural styles, components and connectors,” in *Proceedings of the 14th IEEE International Conference on Engineering of Complex Computer Systems*. IEEE, 2009, pp. 40–49.

153. G. Paquette and A. Masmoudi, “Ontology-based software component aggregation,” *Computer Engineering: Concepts, Methodologies, Tools and Applications*, pp. 223–238, 2011.

154. H. Paulheim and A. Erdogan, “Seamless integration of heterogeneous ui components,” in *Proceedings of the 2nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. ACM, 2010, pp. 303–308.

155. Y. Jiang, Y.K. Lu, H.R. Wang, Y.N. Li and X.D. Fu, “Extended software component model for testing and reuse,” in *Proceedings of the 2nd IEEE International Conference on Information Management and Engineering (ICIME)*. IEEE, 2010, pp. 310–313.

156. W. Long, “Research on development method of mes based on component and driven by ontology,” *Journal of Software*, vol. 5, no. 11, pp. 1228–1235, 2010.

157. C. Li, R. Pooley and X. Liu, “Ontology-based quality attributes prediction in component-based development,” *International Journal of Computing Science and Information Technology*, vol. 2, no. 5, pp. 12–29, 2010.

158. L. Kzaz, H. Elasri and A. Sekkaki, “A model for semantic integration of business components,” *International Journal of Computing Science and Information Technology*, vol. 2, no. 1, pp. 1–12, 2010.

159. D. Garijo and Y. Gil, “A new approach for publishing workflows: abstractions, standards, and linked data,” in *Proceedings of the 6th workshop on Workflows in Support of Large-Scale Science*. ACM, 2011, pp. 47–56.

160. M. Kost and J.C. Freytag, “Privacy analysis using ontologies,” in *Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy*. ACM, 2012, pp. 205–216.

161. X. Li and W. Zhang, “Ontology-based testing platform for reusing,” in *Proceedings of the Sixth International Conference on Internet Computing for Science and Engineering*. IEEE, 2012, pp. 86–89.

162. F.E. Castillo-Barrera, C. Medina-Ramírez, H.A. Durán-Limón, J.E.L Gayo and S.M. Sadjadi, “Verifying the behavioural contracts among components by means of semantic web techniques,” in *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*, 2012, pp. 1–7.

163. M.A. Francisco, M. Lopez, H. Ferreiro and L.M. Castro, “ Turning web services descriptions into quick-check models for automatic testing,” in *Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang*. ACM, 2013, pp. 79–86.

164. W. Dai, V.N. Dubinin and V. Vyatkin, “Automatically generated layered ontological models for semantic analysis of component-based control systems,” *IEEE Transactions on Industrial Informatics***,** vol. 9, no. 4, pp. 2124–2136, 2013.

165. F.E. Castillo-Barrera, H.G. Pérez-González and S.M. Sadjadi, “Towards a software domain metric based on semantic web techniques,” in *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*, 2013, pp. 1–6.

166. E.K. Karatas, B. Iyidir and A. Birturk, “Ontology-based software requirements reuse: case study in fire control software product line domain,” in *Proceedings of the IEEE International Conference on Data Mining Workshop (ICDMW)*. IEEE, 2014, pp. 832–839.

167. V. Rastgoo, M.S. Hosseini and E. Kheirkhah, “Semantic web-based software engineering by automated requirements ontology generation in soa,” *International Journal of Web & Semantic Technology***,** vol. 5, no. 2, pp. 1–11, 2014.

168. R. Jayasudha, S. Subramanian and L. Sivakumar, “A novel software reuse methodan ontological approach,” *International journal of scientific and engineering research*, vol. 5, no. 5, pp. 138–142, 2014.

169. Y. Lv, Y. Ni, H. Zhou and L. Chen, “Multi-level ontology integration model for business collaboration,” *International Journal of Advanced Manufacturing Technology*, vol. 84, pp. 1–7, 2016.

170. M. Bhat, K. Shumaiev, A. Biesdorf, U. Hohenstein, M. Hassel and F. Matthes, "An ontology-based approach for software architecture recommendations,” in *Proceedings of the 23rd conference on information systems*, 2017.

171. N. Yanes, S. Ben Sassi, and H. Hajjami Ben Ghezala, “Ontology-based recommender system for COTS components,” *Journal of Systems and Software*, vol. 132, pp. 283–297, 2017.

172. D. Du, X. Ren, Y. Wu, J. Chen, W. Ye, J. Sun, X. Xi, Q. Gao and S. Zhang, “Refining traceability links between vulnerability and software component in a

vulnerability knowledge graph," in *Web engineering*. Springer, 2018, pp 33–49.

173. V. Gruhn and R. Laue, "Adopting the cognitive complexity measure for business process models," in *Proceedings of the 5th International Conference on Cognitive Informatics*, IEEE, 2006, pp. 236–241.

174. Y. Wang and S. Patel, "Exploring the cognitive foundations of software engineering," *International Journal of Software Science and Computational Intelligence*, vol. 1, no. 2, pp. 1–19, 2009.

175. E. McCrum-Gardner, "Which is the correct statistical test to use?," *British Journal of oral and maxillofacial surgery*, vol. 46, no. 1, 2008, pp. 38–41.

176. R. Harrison and S. Counsell, "Theoretical validation and empirical evaluation of object-oriented design metrics," *IEEE Transactions on Software Engineering*, vol. 18, no. 5, pp. 410–422, 1992.

177. *JFreeChart Java chart library*. Accessed: January 2018. Available: http://www.jfree.org/jfreechart

178. *Heritrix archival crawler project*. Accessed: January 2018. Available: http://crawler.archive.org/articles/releasenotes/

179. A. Koenig and B. Stroustrup, "Exception handling for C++," in *Proceedings of Usenix C++ Conference*, 1990, pp. 149–176.

180. *JHawk- the Java metrics tool*. Accessed: January 2018. Available: http://www.virtualmachinery.com/jhawkprod.htm

181. *STAN4J- Structure analysis for Java*. Accessed January 2018. Available: http://stan4j.com/

182. J.Y. Kuo, F.C Huang, C. Hung and L.H.Z. Yang, "The study of plagiarism detection for object-oriented programming," in *Proceedings of the Sixth International Conference on Genetic and Evolutionary Computing (ICGEC)*. IEEE, 2012, pp. 188–191.

183. R. Shatnawi and W. Li, "The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process," *Journal of Systems and Software*, vol. 81, no. 11, pp. 1868–1882, 2008.

184. D.A. Belsley, E. Kuh and R.E Welsch, *Regression Diagnostics: Identifying Influential Data and Sources of Collinearity*. John Wiley & Sons, 2005.

185. E. Frank, M.A. Hall and I.H. Witten, *Data Mining: Practical Machine Learning Tools and Techniques.* Morgan Kaufmann, 2016.

186. K. Michalak and H. Kwasnicka, "Correlation-based feature selection strategy in neural classification," in *Proceedings of the Sixth International Conference on Intelligent Systems Design and Applications*. IEEE, 2006, pp. 741–746.

187. M.A. Hall, “Correlation-based feature selection for discrete and numeric class machine learning,” in *Proceedings of the Seventeenth International Conference on Machine Learning*, 2000, pp. 359–366.

188. J. Huang and C.X. Ling, “Using AUC and accuracy in evaluating learning algorithms,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 3, pp. 299–310, 2005.

189. T.M. Khoshgoftaar, K.Gao and N. Seliya, “Attribute selection and imbalanced data: Problems in software defect prediction,” in *Proceedings of the 22nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, 2010, pp. 137-144.

190. R.C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003

191. P. Oman and J. Hagemeister, “Metrics for assessing a software system’s maintainability,” in *Proceedings of the Conference on Software Maintenance*. IEEE, 1992, pp. 337–344.

192. D. Gray, D. Bowes, N. Davey, Y. Sun and B. Christianson, “Reflections on the NASA MDP data sets,” *IET software*, vol. 6, no. 6, pp. 549-558, 2012.

193. N. Chawla, K.W. Bowyer, L.O. Hall and W.P. Kegelmeyer, “SMOTE: synthetic minority over-sampling technique,” *Journal of artificial intelligence research*, vol. 16, pp. 321-357, 2012.

194. A.J. Tallón-Ballesteros and J.C. Riquelme, “Deleting or keeping outliers for classifier training?,” in *Proceedings of the 2014 Sixth World Congress on Nature and Biologically Inspired Computing (NaBIC)*. IEEE, 2014, pp. 281-286.

195. Y. Peng, G. Kou, G. Wang, W. Wu, and Y. Shi, “Ensemble of software defect predictors: an AHP-based evaluation method,” *International Journal of Information Technology & Decision Making*, vol. 10, no. 1, pp. 187-206, 2011.

196. S. Shalev-Shwartz, Y. Singer and N. Srebro, “Pegasos: Primal Estimated sub-GrAdient SOlver for SVM,” *Mathematical programming*, vol. 127, no. 1, pp. 3-30, 2011.

197. I. Myrtveit, E. Stensrud and M. Shepperd, “Reliability and validity in comparative studies of software prediction models,” *IEEE Transactions on Software Engineering*, vol. 31, pp. 380–391, 2005.

198. L. Kaur and A. Mishra, “An Empirical Analysis for Predicting Source Code File Reusability Using Meta-Classification Algorithms,” in *Proceedings of the International Conference on Advanced Computational and Communication Paradigms*. Springer, 2018, pp. 493-504.

199. L. Wilkinson, A. Anand, and D.N. Tuan, “CHIRP: a new classifier based on composite hypercubes on iterated random projections,” in *Proceedings of the*

17th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2011, pp. 6-14.

200. V.G. Kaburlasos, I.N. Athanasiadis and P.A. Mitkas, "Fuzzy lattice reasoning (FLR) classifier and its application for ambient ozone estimation," *International journal of approximate reasoning*, vol. 45, no. 1, pp. 152-188, 2007.

201. R.S. Parpinelli, H.S. Lopes and A.A. Freitas, "Data Mining With an Ant Colony Optimization Algorithm," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 4, pp. 321-332, 2002.

202. S. Sheng and C. X. Ling, "Hybrid Cost-sensitive Decision Tree, Knowledge Discovery in Databases," in *Proceedings of the 9th European Conference on Principles and Practice of Knowledge Discovery in Databases*. Springer, 2005.

203. J. Hühn and E. Hüllermeier, "FURIA: an algorithm for unordered fuzzy rule induction," *Data Mining and Knowledge Discovery*, vol. 19, no. 3, pp. 293-319, 2009.

204. J. Stefanowski, " The rough set based rule induction technique for classification problems," in *Proceedings of the 6th European Congress on Intelligent Techniques and Soft Computing*, 1998, pp. 109-113.

205. Z. Islam and H. Giggins, "Knowledge discovery through SysFor: a systematically developed forest of multiple decision trees," in *Proceedings of the Ninth Australasian Data Mining Conference*. ACM, 2011, pp. 195-204.

206. F. Jiménez, C. Martínez, E. Marzano, J. Palma, G. Sánchez and G, Sciavicco, "Multiobjective Evolutionary Feature Selection for Fuzzy Classification," *IEEE Transactions on Fuzzy Systems*, vol. 27, no. 5, pp. 1085-1099, 2019.

207. R.P. Espíndola and N.F.F. Ebecken, "On extending f-measure and g-mean metrics to multi-class problems," *WIT Transactions on Information and Communication Technologies*, vol. 35, 2005.

208. X. Xuan, D. Lo, X. Xia and Y. Tian, "Evaluating defect prediction approaches using a massive set of metrics: An empirical study," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. ACM, 2015, pp. 1644-1647.

209. E. Jelihovschi, J. C. Faria and I. B. Allaman, "Scottknott: A package for performing the scott-knott clustering algorithm in r," *Trends in Applied and Computational Mathematics*, vol. 15, no. 1, pp. 3–17, 2014.

210. B. Ghotra, S. McIntosh and A.E. Hassan, "Revisiting the impact of classification techniques on the performance of defect prediction models," in *Proceedings of the 37th International Conference on Software Engineering*. IEEE, 2015, pp. 789-800.

211. E. Bauer and R. Kohavi, "An empirical comparison of voting classification algorithms: Bagging, boosting, and variants," *Journal of Machine Learning*, vol. 36, no. 2, pp. 105–139, 1999.

212. C. Klamler, “On the closeness aspect of three voting rules: Borda, Copeland, Maximin,” *Journal of Group Decision and Negotiation*, vol. 14, no. 3, pp. 233–240, 2005.

213. J. Alcalá-Fdez, L. Sánchez, S. García, M.J. del Jesus, S. Ventura, J.M. Garrell, J. Otero, C. Romero, J. Bacardit, V.M. Rivas, J.C. Fernández and F. Herrera. “KEEL: A Software Tool to Assess Evolutionary Algorithms to Data Mining Problems,” *Soft Computing*, vol. 13, no. 3, pp. 307-318, 2009.

214. *PASW Statistics for Windows* 18.0. (2009). SPSS Inc, IBM.

215. *RStudio: Integrated Development for R.* (2015). RStudio, Inc.

216. J. Gama and P. Medas, “Learning decision trees from dynamic data streams,” *Journal of Universal Computer Science*, vol. 11, no. 8, pp. 1353–1366, 2005.

217. P. Singh and S. Verma, “An investigation of the effect of discretization on defect prediction using static measures,” in *Proceedings of International Conference on Advances in Computing, Control, and Telecommunication Technologies*. IEEE, 2009, pp. 837-839.

218. W. Guo, “Ontology design for supporting matchmaking in E-commerce,” in *International Symposium on Information Science and Engineering.* IEEE, 2008, pp. 410-413.

219. J Evermann and Y. Wand, “Ontology based object-oriented domain modelling: fundamental concepts,” *Requirements engineering*, vol. 10, no. 2, pp. 146-160, 2005.

220. M. O’Connor, H. Knublauch, S. Tu, B. Grosof, M. Dean, W. Grosso, and M. Musen, “Supporting rule system interoperability on the semantic web with SWRL,” in *Proceedings of International semantic web conference*. Springer, 2005, pp. 974-986.

221. A. Guerrero, V.A. Villagrá, J.E.L. De Vergara and J. Berrocal, “Ontology-based integration of management behaviour and information definitions using SWRL and OWL,” in *Proceedings of International Workshop on Distributed Systems: Operations and Management*. Springer, 2005, pp. 12-23.

222. M.J. O'Connor and A. Das, “The SWRLTab: An Extensible Environment for working with SWRL Rules in Protégé-OWL,” in *Proceedings of 2nd International Conference on Rules and Rule Markup Languages for the Semantic Web*, 2006, pp. 1-2.

223. X.B. Tang, G.C. Liu, J. Yang and W. Wei, “Knowledge-based financial statement fraud detection system: based on an ontology and a decision tree,” *Knowledge Organization*, vol. 45, no. 3, pp. 205-219, 2018.

224. B. Yang, “Construction of logistics financial security risk ontology model based on risk association and machine learning,” *Safety Science*, to be published. https://doi.org/10.1016/j.ssci.2019.08.005

225. B. Liu, L. Yao and D. Han, “Harnessing ontology and machine learning for RSO classification,” *SpringerPlus,* vol. 5, no. 1, pp. 1-22, 2016.

226. M. Bandara and F.A. Rabhi, “Semantic modeling for engineering data analytics solutions,” *Semantic Web,* (Preprint), pp. 1-23, 2018.

227. E. Aman, “Bridging data mining and semantic web,” Master’s Thesis, Department of Mathematical Information Technology, University of Jyväskylä, Finland, 2016. Accessed on: June 16, 2018. [Online]. Available: http:// https://jyx.jyu.fi/bitstream/handle/123456789/52274/URN:NBN:fi:jyu-201612125050.pdf?sequence=1

228. T. Mitchell, “Decision tree learning,” *Machine learning*, vol. 414, pp. 52-78, 1997.

229. E. Jajaga, L. Ahmedi and L. Abazi-Bexheti, “Semantic Web trends on reasoning over sensor data,” in *Proceedings of the 8th South East European Doctoral Student Conference,* 2013, pp. 284-293.

230. M. Buckland and F. Gey, “The relationship between recall and precision,” *Journal of the American society for information science*, vol. 45, no. 1, pp. 12-19, 1994.

231. F. Salfner, M. Lenk and M. Malek, “A survey of online failure prediction methods,” *ACM Computing Surveys (CSUR),* vol. 42, no. 3, pp. 1-42, 2010.

232. J. Goldstein, M. Kantrowitz, V. Mittal, and J. Carbonell, “Summarizing text documents: sentence selection and evaluation metrics,” in *Proceedings of the 22nd Annual international SIGIR Conference on Research and Development in information Retrieval*. ACM, 1999, pp. 121-128.

233. Monika and A. Kaur, “Fraud Prediction for credit card using classification method,” *International Journal of Engineering and Technology*, vol. 7, no. 3, pp. 1083-1086, 2018.

234. M. Bandara, A. Behnaz and F.A. Rabhi, “RVO - The Research Variable Ontology”, in *Proceedings of the European Semantic Web Conference*. Springer, 2019, pp. 412-426.

LIST OF PUBLICATIONS

Articles published:

SCI/ SCIE Indexed Journals

L. Kaur and A. Mishra, "Software component and the Semantic web: An in-depth content analysis and integration history," Journal of Systems and Software, vol. 125 pp. 152-169, 2017. (Impact Factor: 2.559)

L. Kaur and A. Mishra, "Cognitive complexity as a quantifier of version to version Java-based source code change: An empirical probe," Information and Software Technology, vol. 106, pp. 31-48, 2019. (Impact Factor: 2.921)

Conferences

L. Kaur and A. Mishra, "A comparative analysis of evolutionary algorithms for the prediction of software change," in *Proceedings of the 2018 International Conference on Innovations in Information Technology (IIT)*. IEEE, 2018, pp. 187-192.

SCOPUS

L. Kaur and A. Mishra, " A Pragmatic Framework for Predicting Change Prone Files Using Machine Learning Techniques with Java-based Software," Asia Pacific Journal of Information Systems, vol. 30, pp. 457-496, 2020.

Articles under review:

L. Kaur and A. Mishra, "An ontology-based approach to change-prone software selection" in *Applied Ontology*, IOS Press. (SCIE Indexed)

APPENDIX

Appendix A:

Table A.1 Software change prediction studies

Source	Metrics	Technique for prediction	Datasets	Validation; statistical test	Outcome
Liu and Khoshgoftar (2001) [49]	NOCI, 4 measures pertaining to lines of code	GP, LR	Windows based software application (VLWA dataset)	Hold-out; -	This paper introduced the prior probability and costs of misclassification into the fitness function of the GP technique for software change prediction. As the cost ratio increased, type I errors increased and type II errors decreased. Additionally, GP was observed to perform more accurately than LR.
Chaumun et al. (2002) [91]	CK metrics (WMC, DIT, NOC, CBO, RFC, LCOM)	Stratified sampling approach	Telecommu nication–based system, built in C++; unknown	-; Correlation coefficient, ANOVA	The impact of change in one class due to change in another class of the target project was calculated by the authors for which links between classes of the system were considered. Results indicated a relation between WMC, a design metric, and the mean change impact of a class i.e. higher the WMC value, higher the mean change impact.
Bieman et al. (2003a) [92]	Size measures: NOO, NOA; Relation metrics: NFM, NOMO, DIT, NOC, NOD	Design patterns Box plots, outlier analysis	Two versions of a C++ based system	-; -	The authors showed how to quantify the degree to which classes are change-prone both locally and in their interactions with others by visualising local versus cluster change-proneness through the change architecture diagram.
Khoshgoftar et al. (2003)	NOCI, 4 measures pertaining to lines of code	GP, Decision tree based GP	Windows based Software Application (VLWA Dataset)	Hold-out; -	The authors established that the GP-based decision tree modeling approach achieved better results than the models calibrated using standard GP. Moreover, the GP-based decision tree model achieved better optimization with respect to the tree size and the model selection strategy of balancing the Type I and Tyoe II errors.
Bieman et al. (2003b) [93]	Size measures: NOO, NOA. Design patterns	Scattergram	Two versions each from C++ and Java systems, Netbeans and JRefactory versions; 3 commercial and 2 open source	-; Spearman rank correlation, Mann-Whitney test	Results from datasets generated from the C++ system and the Netbeans system indicated that classes with greater size have the most change-prone source code. Hoever, this inference was not applicable to the other analysed systems by the author. Additionally, classes with more children or more descendents also were also observed to undergo more change.
Arisholm et al. (2004) [94]	Various dynamic coupling measures (11), static coupling measures	Principal component analysis (PCA)	4 subreleases of Velocity (Apache Jakarta	-; -	Some dynamic coupling measures were observed to be significant predictors of change-proneness. The authors argued that dynamic coupling measures combined with the existing static coupling measures would enhance software

Table A.1 Software change prediction studies

Source	Metrics	Technique for prediction	Datasets	Validation; statistical test	Outcome
	and size measures (27).		Project)		change-proneness estimation.
Tsantalis et al. (2005) [51]	NOO, CK metrics: WMC, CBO, DIT, RFC, NOC, LCOM	Boxplot, LR	13 versions of JFlex and 9 versions of JMol (both written in Java)	-; -	Statistical analysis showed a correlation between the extracted probabilities and the actual changes in a system. Aditionally, the proposed approach improved the prediction accuracy over a simple model that relied only on past data.
Koru and Tian (2005) [52]	37 class level static metrics for OpenOffice ; 67 class level static metrics for Mozilla	Tree-based model	Two projects, Mozilla and OpenOffice (both in C++)	Ten-fold; Mann-Whitney U test	The chief outcome from the analysis was that, contrary to the popular belief, the source code modules which underwent the maximum number of changes were not those with the highest values of the metrics taken as independent variables.
Askari and Holt (2006) [95]	CVS logs	Maximum likelihood estimation, Reflexive exponential decay (RED), RED-co-change (REDCC)	Versions of OpenBSD, FreeBSD, KDE, Koffice: NetBSD, Postgres	-; -	Out of the three probabilistic prediction models, the REDCC model was observed to predict the distribution closest to the actual distribution for all the studied systems.
Koru and Liu (2007) [23]	37 class level etrics for Open Office; 67 class level static metrics for Mozilla	Tree-based model	Mozilla and OpenOffice (both in C++)	Ten-fold; -	The results strongly supported Pareto's law that states that large majority (around 80%) of problems are rooted in a small proportion (around 20%) of the modules. The authors used tree-based models for identification of change-prone classes and discussed a prioritisation strategy which can help practitioners to use resources in an efficient manner. The model results exhibited that size metrics and the static metrics related to size were more associated with change.
Sharafat and Tahvildari (2007) [96]	AID, ALD, LOC, MNOB, MPC, NIC, NOLV, NOP	Own proposed probabily-based approach	14 versions of JFlex	-; -	The authors proposed a novel probabilistic approach to predict changes by calculating the total probability of change for each class using internal change probability and propagation probability. The authors also employed change log of the software system to get another measure of the likelihood of change of classes. Finally, the total probability of change for each class and history-based time normalised probability were combined to predict whether or not each class will change in the future generation. The proposed probabilistic approach was concluded to be simple and accurate in the comparison with existing methods in the literature.
Han et al. (2008) [97]	BDM, CK metrics	Stepwise multiple regression	JFreechart versions	-; ANOVA	The authors established BDM to be an useful indicator of change-proneness prediction and concluded that inclusion of BDM in the metrics set along with CK metrics improved predictive model 's accuracy.
Sharafat and Tahvildari	AID, ALD, CycloC, SLOC,	Own proposed probabily-	JFlex versions	-; -	The authors did another study based on the same concept [97] wherein the probabilistic approach to determine whether classes will change in future

Table A.1 Software change prediction studies

Source	Metrics	Technique for prediction	Datasets	Validation; statistical test	Outcome
(2008) [98]	MNOB, MPC, NIC, NOLV, NOP	based approach			release was proposed. However, in this research, authors also presented additional techniques to deal with cyclic dependencies in classes, which could be linked to software change.
Penta et al. (2008) [27]	12 design patterns	Design motif identificat-ion multi-layered approach (DeMIMA)	3 object-oriented systems: JHotDraw, Eclipse-JDT plugins, and Xerces	-; Kruskal-Wallis test, MannWhitney test	The authors presented an empirical study based on the premise that source code classes likely to change more must be designed carefully as their change- proneness can make other parts of the system less robust to changes. They concluded that the classes playing a particular role in a given design motif were more change-prone than the classes playing some other roles. For, e.g., in the adaptor motif, classes playing the role of adaptor were more change-prone. Similar results were observed for the abstract factory and command motif. Therefore, design patterns play a crucial role in the application.
Khomh et al. (2009) [31]	29 different kinds of code smells	LR	2 open source system: 9 releases of Azureus and 13 releases Eclipse (Both written in Java)	-; Fisher's exact test, Mann-Whitney test, t-test	The authors concluded that source code classes with code smells are observed to be more change-prone than other classes. Additionally, the authors highlighted that some specific code smells should be scrutinized by the developers and maintainers during software evolution as they could likely indicate the change-prone classes.
Zhou et al. (2009) [24]	3 size metrics: SLOC, NOM, NOP; 18 cohesion metrics; 20 coupling metrics; 17 inheritance metrics	-	Two releases of Eclipse Written in Java	-, spearman correlation	The authors established (through empirical analysis) the confounding effect of class size on the associations between object-oriented metrics and change-proneness of classes.
Azar (2010) [53]	22 OO metrics; 4 cohesion metrics; 4 coupling metrics; 7 inheritance metrics; 7 size metrics	C4.5, GA	Bean Browser, Ejbvoyager, Free, Javamapper, Jchempaint, Jedit, Jetty, Jigsaw, Jlex, Lmjs, Voji, 4 versions of JDK	Ten-fold; -	The authors proposed a rule-based classifier which has a lower complexity than those created by C4.5. Empirical results exhibited that the proposed approach outperformed algorithms like C4.5, random guess and the majority classifier but more significantly on the imbalanced dataset.
Han et al. (2010) [99]	CK metrics: WMC, DIT, RFC, NOC, LCOM; Lorenz and Kidd metrics; MOOD metrics (20	PCA, stepwise multiple regression	9 releases of JFlex	–; ANOVA	The authors established the BDM metric to be a useful indicator change-prone class prediction. Additionally, BDM was observed to improve accuracy of change-prone class prediction over that of metrics (CK, Lorenz and Kidd, and MOOD) when the target system contained high degree of inheritance relationships and polymorphism. • However, when inheritance relationships and polymorphism were less in the system, BDM made no difference in the

Table A.1 Software change prediction studies

Source	Metrics	Technique for prediction	Datasets	Validation; statistical test	Outcome
	metrics); BDM				prediction of change-prone classes.
Azar and Vybihal (2011) [54]	22 OO metrics: 4 cohesion metrics, 4 coupling metrics, 7 inheritance metrics, 7 size metrics	C4.5, ACO	Bean Browser, Ejbvoyage, Free, Javamapper Jchempaint, Jedit, Jetty, Jigsaw, Jlex, Voji	Ten-fold; Wilcoxon Signed Rank	The proposed ACO technique was observed to out-perform both C4.5 and random guessing. Moreover, the proposed approach also preserved the white-box nature of the obtained models, hence providing the classification label as well as its source.
Posnett et al. (2011) [25]	Pattern roles, project release, SLOC	Multiple regression model	3 Object-oriented open source projects: JHotDraw, Xerces, and Eclipse JDT	-; ANOVA	The empirical results indicated size in terms of SLOC to be a strong determinant of change-proneness, better than design pattern or meta-pattern roles.
Romano and Pinzger (2011) [12]	CK metrics, Set of metrics to measure the complexity and the usage of interfaces, 2 metrics to measure external cohesion: interface usage cohesion, clustering metric	SVM, NB, ANN	8 plugins from Eclipse and hibernate 2 and hibernate 3 systems	Ten-Fold; Correlation analysis Wilcoxon Signed rank	The results indicated that the IUC metric possessed the strongest correlation with the number of source code changes and therefore had a major role in improving the performance of prediction models to classify Java interfaces into change-prone and not change-prone.
Lu et al. (2011) [26]	62 object-oriented metrics: 7 size, 18 cohesion, 20 coupling, and 17 inheritance	Random effect model, boxplots	102 Java software systems and for each software, two versions were taken; Open source	-, -	The results from the empirical analysis indicated that the predictive capability of size metrics is moderate, coupling/cohesion metrics is low and inheritance metrics is poor for the selected software projects.
Eski and Buzluca 2011 [100]	CK, QMOOD	Combined Rank List Mechanism	Yari (3 versions), UCdetector (4 versions), JFreeChart (4 versions)	-, -	The results indicated that a combination of metrics gives the most optimal result for the change estimation of studied projects and this combination differs from project to project as each project has its own nature and the project history. Also, the combination of WMC, LOC and RFC works for the selected project in prediction of their change-prone classes.
Giger et al. 2012 [32]	CK, 7 Network centrality	MLP, BN	19 Eclipse plug-in projects,	Ten-Fold; Friedman and	The results indicated that Centrality measures from Social Network Analysis (SNA) and OO metrics positively correlate with number of source

Table A.1 Software change prediction studies

Source	Metrics	Technique for prediction	Datasets	Validation; statistical test	Outcome
	measures from social network analysis		Azureus	Wilcoxon Signed rank	code changes. Moreover, Degree Centrality measures and complexity (WMC) were observed to be strongly correlated with each other. Also, models based on MLP using SNA and OOM as independent variables can predict categories of code change types, better than those developed using BN models.
Elish and Al-Khiaty 2013 [28]	CK, 16 Evolution-based metrics	LR	PeerSim (9 versions), VSSPlugin (13 versions)	Ten-Fold; Wilcoxon Signed Rank	Since the evolution-based metrics measure different dimensions than the C&K metrics, thusly they do not measure redundant data, and hence complement the C&K metrics. Additionally, several evolution-based metrics were observed to be significantly correlated with class change-proneness and empirical results indicated a more accurate prediction when the evolution-based metrics were combined with the C&K metrics.
Malhotra and Khanna (2013) [11]	CK, 16 other class-level metrics	LR, RF, MLP, BAG	Two versions each of Frinika, FreeMind, OrDrumBox	Ten-Fold; -	RFC was observed to be a significant indicator of change-proneness for all the three data sets, post the application of the feature subset selection method. Also, RF and Bagging methods outperformed the LR model, in terms of AUC.
Malhotra and Bansal (2014) [13]	CK, SLOC	LB	Four versions each of Apache Abdera, Apache POI, and Apache Rave	Ten-Fold and Inter-version -	The authors concluded that inter-version predictions are more accurate than the inter-project predictions, with the success rate of inter-release prediction being 67% and the success rate of inter-project predictions being 30%. Also, the authors established that the distributional characteristics of the datasets should match for a high prediction success rate.
Malhotra and Khanna (2014) [101]	CK, SLOC	GEP	Simutrans, Glest	Ten-Fold; -	The results indicated that change in a class of software was dependent on the values of SLOC, LCOM, WMC and RFC. Also, the GEP algorithm was observed to be highly efficient for developing models which predict software change and its results were better than univariate results of individual metrics.
Marinescu (2014) [102]	NOM, DIT, RFC, NOC, CBO, CycloC, SLOC	GP-Symbolic Regression	ArgoUmL, Findbugs, FOP, FreeCol	-; Proportion test	Symbolic regression is one of the earliest applications of Genetic Programming. The results show that Symbolic Regression is able to predict change-prone classes with a precision that tends to exceed 0.7506 and a recall that tends to exceed 0.99, and defect-prone classes with a precision that tends to exceed 0.0859 and a recall that tends to be 1.
Elish et al. (2015) [33]	CK	LR, MLP, RBF, SVM, DT, GEP, k-means, BAG, Boosting, Majority Voting, Non-linear Decision tree Forest)	PeerSim, VSSPlugin	Hold-out and leave-one out; -	The empirical results indicated that the performance of the individual prediction models might vary from dataset to dataset wherein the ensemble methods achieved comparable or higher prediction accuracy compared to individual models. Moreover, the results indicated that MLP, when employed as base classifier, improved classification accuracy of bagging and boosting ensemble methods, whereas SVM decreased the accuracy as a base classifier.
Malhotra	CK metrics,	LR, RF,	Simutrans,	Ten-fold	The results indicated the CBO and SLOC metrics

Table A.1 Software change prediction studies

Source	Metrics	Technique for prediction	Datasets	Validation; statistical test	Outcome
and Khanna (2015) [55]	SLOC, NOM, NIV, NPM, NIM, NOA	BAG, MLP, ADB, CPSO, HIDER, MPLCS, SUCS, GFS-SP, NNEP	Glest, Celestia	and Inter-project; Friedman's test	to be highly correlated with change in a class. The model predicted using the BAG technique outperformed statistical techniques of ML and SBT. The two SBTs, the NNEP technique and the MPLCS technique also yield good results, better than the statistical LR technique.
Bansal (2017) [45]	CK metrics, SLOC	NB, BN, LB, ADB, GFS-ADB, GFS-LB, GFS-MLB, HIDER, NNEP, PSO-LDA, GFS-GP, GFS-SP, SLAVE	Apache Rave, Apache Math	Ten-fold; Wilcoxon Signed Rank	The authors concluded that although the SBTs outperformed the ML techniques in the empirical analysis, Wilcoxon Signed Rank statistical test indicated that that there does not exist any statistically significant dissimilarity in the results of the best performing SBT and the selected ML models.
Elish et al. (2017) [29]	CK, 16 Evolution-based metrics	GMDH	13 versions of VSSPlugin	Hold-out; -	The combination of the two set of metrics (evolutionary set, and C&K) as independent variables to a GMDH-based abductive classifier improves classification accuracy compared to that obtained with a single set of predictors. Additionally, the best performance was obtained when fusing the outputs of the three single models using majority voting to form an ensemble GMDH abductive classifier.
Kumar et al. (2017) [46]	62 OO metrics: 19 cohesion metrics, 19 coupling metrics, 17 inheritance metrics, 7 size metrics	LR, NB, Extreme Machine Learning (Linear, Polynomial and RBF kernels), SVM (Linear, Polynomial and Sigmoid kernels), Ensembles of Techniques (Best in Training, Majority Voting)	Eclipse	Ten-fold; -	The authors establish coupling metrics to have higher predictive ability than to size metrics, cohesion metrics and inheritance metrics. From experiments, it was observed that the performance of the feature selection techniques varied with the different classification methods used.
Kumar et al. (2017) [63]	21 OO metrics including CK metrics	Least Square SVM (Linear, Polynomial and RBF kernels)	5 versions of Ebay Services	Twenty-fold; t-test	The authors concluded that the model developed using all metrics obtains slightly better results compared to models on selected set of metrics. Also, the predictive model developed using LS-SVM linear kernel yielded better results compared to other kernels. It was also observed that the performance of the selected set of source code metrics varied with the different classification methods (such as linear, polynomial and RBF kernel).
Kumar et al.	61 OO	LR, MLP, RBF, DT,	10 Eclipse	Ten-fold and Inter-	The empirical results validate the GA to be useful in providing better predictive performance

Table A.1 Software change prediction studies

Source	Metrics	Technique for prediction	Datasets	Validation; statistical test	Outcome
(2017) [34]	metrics	RF, Ensembles of Techniques (Best in Training, Majority Voting, Non-Linear Decision Tree Forest)	plug-ins	project; Wilcoxon Signed Rank	especially during across project validation. The study also reveals the performance of the selected classifiers to be different across the same project validation setting.
Malhotra and Jangra (2017) [47]	13 OO metrics including CK metrics	LR, RF, ADB, BAG, MLP, NB, BN, J48, NNGE	AOI, SweetHome 3D	Ten-fold and Inter-project; t-test	NPRM, SLOC and WMC metrics were established to be significant indicators of change-proneness in both the data sets using feature selection and CBO and NPRM metrics were found significant using univariate LR analysis. Also, the models developed using the Bagging, RF, Bayes Net, LogitBoost and MLP methods outperformed the model developed using the LR method. Moreover, the authors concluded that it is not necessary that the results of validation for different projects will be similar with Naïve Bayes performing best for 'AIO' project and Bagging performing best for 'Sweet-Home-3D'.
Malhotra and Khanna (2017) [9]	18 OO metrics including CK metrics, SLOC, QMOOD suite, AC, EC, AMC, IC, CBM	PSO-LDA, NNEP, GFS-LB, CART, SUCS, CPSO, C4.5, GA-ADI, HIDER, MLP-CG, MPLCS, LDA, DT-GA, XCS, SVM	Six Android application packages	Ten-fold; Friedman and Wilcoxon Signed Rank test	The comparative assessment of search-based techniques and hybridized techniques along with the statistical validation of the test established that the hybridized techniques are better at predicting change-prone classes than search-based techniques.
Malhotra and Khanna (2017) [48]	18 OO metrics including CK metrics, SLOC, QMOOD suite, AC, EC, AMC, IC, CBM	MLP, RF, NB, ADB, LB, BAG	Three Android application packages, Net, IO, Log4j	Ten-fold and Inter-version; Friedman and Wilcoxon Signed Rank test	The authors confirmed that the performance of different ML techniques improved considerably after utilizing varied sampling methods, with the resampling method along with replacement giving the best results when compared on the basis of AUC, G-mean, and balance performance metrics. However, the two other methods of sampling which were SMOTE and spread subsample also exhibited satisfactory results. The authors also established that the use of MetaCost learners could also be used to handle imbalanced learning problem as they work to sensitize the ML method by furnishing varied costs for diverse errors of misclassification in order to ensure that the overall cost of misclassification is reduced thereby improving the performance of the model developed. This results also indicated that in comparison of the cost-sensitizing of learners, provision of effective training data utilizing resampling is the better approach.
Yan et al. (2017) [58]	10 OO metrics including CK suite,	LB, MLP, RBF, SVM, k-means, CLAMI,	Ant, Antlr, Argouml, Azureus, Freecol,	Intra-version and Inter-project;	The authors applied a cutting edge unsupervised method (CLAMI: Clustering, Labeling, Metric selection and Instance selection) to the change-prone class prediction thereby proving that as

Table A.1 Software change prediction studies

Source	Metrics	Technique for prediction	Datasets	Validation; statistical test	Outcome
	Li and Henry [65] metrics	CLAMI+	Freemind, Hibernate, Jgraph, Jmeter, Jstock, Jung, Junit, Lucene, Weka	Friedman and Nemenyi test	compared to the baselines like LB, MLP etc., the unsupervised methods provide either better or comparable results for both the validation scenarios, intra-project and inter-project.
Agrawal and Singh (2018) [57]	15 OO metrics including CK suite	MLP, LR, RF, KStar, PART, BAG, BN	GATE, Tuxguitar, FreeCol, KolMafia, Legatus	Ten-fold; Friedman's test	The Friedman's test results suggested Bagging to be the best algorithm when applied to the five datasets. RF, BN, and KStar obtained higher accuracies than LR. However, MLP was the observed to perform the worst. Also, the results obtained by authors contradicted few studies previous studies in literature [22] where ML techniques outperformed LR in change-proneness prediction.
Catolino et al. (2018) [7]	Entropy of changes, number of developers, structural and semantic scattering of developers, evolution-based metrics, OO metrics	LR	Ant, Cassandra, Lucene, POI, Synapse, Velocity, Xalan, Xerces, ArgoUML, aTunes, FreeMind, JEdit, JFreeChart, JHotDraw, JVLT, pBeans, pdfTranslator, Redaktor, Serapion, Zuzel	3 month sliding window to train and test models, Mann–Whitney test and Cliff's test	The authors concluded that developer-based models to predict change exhibited the best performance. This was especially relevant for the model proposed by Di Nucci et al. (2017). These models were sometimes even better than a model based on code metrics.
Ge et al. (2018) [60]	CK, SLOC	LR, NB, DT, SVM, Decision Table, Deep Metric Learning	4 versions each of ArgoUML, FreeCol, JMeter, Jung, and Weka	Inter-project; -	To better the Inter-project prediction, the authors proposed a Deep Metric Learning (DML) model to lessen feature distinction between projects before classification of files into change-prone or not. The authors also leverage an over-sampling approach to work with the highly imbalanced dataset for training the model. Experimental results suggest that DML is much better than baselines.
Liu et al. (2018) [103]	CK	BN, CLAMI+	Ant, Antlr, ArgoUML, Azureus, FreeCol, FreeMind, Hibernate, JGraph, JMeter, JStock, Jung, JUnit, Lucene, Weka	Inter-project; Wilcoxon Signed Rank test	The authors proposed a selective Inter-project (SCP) model for change-proneness prediction that focussed on selecting projects with similar data distributions as the source and target. The authors compared it with related change-proneness models like CLAMI+. Experiment showed both AUC and cost-effectiveness got improved due to SCP for CLAMI+.
Kaur and Mishra	CycloC, CHL, CHV, CHE, CHB,	LR, LDA, GFS-GP, GFS-LB,	4 versions of	Ten-fold, Friedman and	The results indicated that the HBTs obtain similar performance to classifiers like LDA and LR and in some cases, even outperform them. The

Table A.1 Software change prediction studies

Source	Metrics	Technique for prediction	Datasets	Validation; statistical test	Outcome
(2018) [61]	MI, AC, EC, Instability	GFS-SP, GFS-ADB, NNEP, GANN	JFreeChart	Wilcoxon Signed Rank test	Friedman test show that evolutionary technique of GFS-LB performed better than the rest on change-proneness of files on all (4) data sets.
Malhotra and Khanna (2018) [30]	CK, 16 Evolution-based metrics	LR, MLP, NB, RF, ADB, BAG, LB	Five versions of Android Contacts and four versions of Android Gallery2	Ten-fold; Friedman and Wilcoxon Signed Rank test	TACH (Total Amount of Changes) and CSBS (Changes since Birth normalized by Class size) both evolution-based metrics are concluded to be good indicators of change-prone class. Also, the performance of the models using both OO metrics and evolution-based metrics is better for developing change prediction models as it incorporates two separate dimensions (structural properties as well as evolution history).
Malhotra and Khanna (2018) [62]	CK, SLOC	RF, BAG, ADB, LB, 4 CPSO voting based fitness ensembles	Six Android application packages, IO, Net, Math, Log4j	Ten-fold; Friedman, Wilcoxon Signed Rank	Authors evaluated and compared the performance of the four proposed Particle Swarm Optimization ensemble classifiers with ML classifiers (Random Forests (RF), AdaBoost (ADB), Bagging (BAG) and LogitBoost (LB)) for developing prediction models which determine the change-prone classes in a software. The individual CPSO classifiers were based on seven different fitness functions (performance metrics) and empirical results exhibited them to be accurate and diverse. Moreover, the performance of the proposed classifiers was found comparable to four ML ensemble classifiers.
Zhu et al. (2018) [56]	Complexity metrics, Word metrics, Network Metrics	C4.5, NB, SVM, BAG	Java projects: Ant, Eclipse, JEdit, Itextpdf, Liferay, Lucene, Struts, Tomcat	Ten-fold; Scott–Knott	The authors employed C4.5, NB, and SVM as base classifiers of the Bagging algorithm. The experimental result showed that Bagging with NB as the base classifier could identify the most number of change-prone files, while the one with J48 as the base classifier could identify the least.
Catolino and Ferrucci (2019) [6]	OO metrics, Process metrics, Developer related factors	LR, Simple Logistic, NB, MLP, ADB, BAG, RF, Voting	Ant, Log4j, Lucene, Pbeans, POI, Synapse, Velocity, Xalan, Xerces, JEdit	Ten-fold; Scott–Knott, Clif	The authors performed an extensive comparison between the performances of the prediction techniques on 33 releases of 10 open-source systems. The results indicated the superiority of ensemble methods and in particular RF to predict software change in terms of F-measure.

Table A.1 Software change prediction studies

Source	Metrics	Technique for prediction	Datasets	Validation; statistical test	Outcome
Kumar et al. (2019) [59]	20 OO metrics including CK metrics suite	LR, Polynomial Regression, DT, Linear, Polynomial & RBF variants of SVM, Extreme ML and Least-Square SVM, Simple Logistic, ANN with 5 training algorithms, Ensembles (Best in Training, Majority Voting, Non-Linear Decision Tree Forest)	compare, webdav, debug, update, core, swt, team, pde, ui, jdt	Five-fold; Wilcoxon Signed Rank	The authors conducted an extensive assessment of 11 feature selection techniques to identify the suitable set of source code metrics, out of 21 metrics, for change-proneness prediction. It was established that the performance of different feature selection techniques varied with use of different classifiers. Additionally, the Least-Square SVM-RBF based model was observed to give best results as compared to model developed using other techniques, followed by the Non-Linear Decision Tree Forest method.

AC: Afferent Coupling; AID: Access of Imported Data; ALD: Access of Local Data; AMC: Average Method Complexity ;BDM: Behavioural Dependency Measure; CBM: Coupling Between Methods of a Class; CBO: Coupling Between Objects; CHB: Cumulative Halstead Bugs; CHE: Cumulative Halstead Effort; CHL: Cumulative Halstead Length; CHV: Cumulative Halstead Volume; CK: Chidamber and Kemerer metrics; CLAMI: Clustering Lableing Metric selection and Instance selection; CycloC: Cyclomatic Complexity; DIT: Depth of Inheritance Tree; EC: Efferent Coupling; IC: Inheritance Coupling; LCOM: Lack of Cohesion in Methods; MI: Maintainability Index; MNOB: Maximum Number Of Branches; MOOD: Metrics for Object-Oriented Design; MPC: Message Passing Coupling; NFM: Number of Friend Methods; NIC: Number of Imported Classes; NIM: Number of Instance Methods; NIV: Number of Instance Variables; NOA: Number of Attributes; NOC: Number of Children; NOCI: Number of Times Source File was Inspected; NOD: Number of Descendants; NOLV: Number of Local Variables; NOM: Number of Methods; NOM: Number of Methods per Class; NOMO: Number of Methods Over-ridden; NOO: Number of Operations; NOP: Number of Parameters; NPM: Number of Public Methods; QMOOD: Quality Model for Object-Oriented Design; RBF: Radial Basis Function; RFC: Response For a Class; SLOC: Lines of Code; WMC :Weighted Method per Class; OO: Object-Oriented metrics;

ACO: Ant Colony Optimization; ADB: ADaBoost; ANOVA: Analysis Of Variance; ANN: Artificial Neural Network; BAG: Bagging; BN: Bayesian Network; CART: Classification And Regression Tree; CPSO : Constricted for Particle Swarm Optimization; DT: Decision Tree; DT-GA: Decision Tree- GA; GA: Genetic Algorithm; GMDH: Group Method of Data Handling; GP: Genetic Programming; GA-ADI: Genetic Algorithm with Adaptive Discretization Intervals; GANN: Genetic Algorithm with Neural Networks; GFS-ADB: Fuzzy Learning based on ADaBoost ; GFS-GP: Fuzzy Learning based on Genetic Programming; GFS-LB: Genetic Fuzzy System- LogitBoost; GFS-MLB: Genetic Fuzzy System MaxLogitBoost ; GFS-SP: Fuzzy Learning based on Genetic Programming Grammar Operators and Simulated Annealing; HIDER: Hierarchical Decision Rules; LB: Logit Boost; LDA: Linear Discriminant Analysis; LR: Logistic Regression; MLP: Multi Layer Perceptron; MLP-CG: MLP with Conjugate Learning; MPLCS: Memetic Pittsburgh Learning Classifier System; NB: Naïve Bayes; NNGE: Non-Nested Generalized Exemplars; NNEP: Neural Net Evolutionary Programming; PART: PARTial decision lists; PCA: Principal Component Analysis ; PSO-LDA: Particle Swarm Optimization- Linear Discriminant Analysis; RF: Random Forest ; SLAVE: Structural Learning Algorithm in a Vague Environment with Feature Selection; SUCS: Supervised Classifier System; SVM: Support Vector Machine; XCS: X Classifier System

Table A.2 Semantic web in Component Based Software Engineering

Source	Technology	Tools	Purpose	Outcome
R.M.M Braga et al. (2001) [133]	Ontology	Ontology Language-XML	Software component information integration from heterogeneous sources using ontology	The Odyssey Mediation Layer(OML) is proposed which uses ontology to access software component information from various heterogeneous information sources. This mediation layer facilitates the integration of domain information and provides support for component request translation across ontologies.
Sugumaran and Storey (2003) [75]	Ontology	Ontology editor-Jess	Software component retrieval	The domain knowledge present in domain models and ontology is exploited in order to retrieve a component from a reuse repository. Ontology provides an explanation of the use of the application domain terms (for both the domain model and reusable repository).
Aroyo and Dicheva (2004) [134]	Ontology	Ontology language:-XML with RDF	Incorporating ontology in the knowledge components of an e-learning based environment	The content is modeled in the form of software components thus authoring tools that are ontology driven and mirror the educational system's modularization consider the entire authoring process and provide semi-wide automation of the difficult authoring tasks.
Yao et al. (2004) [135]	SWS+ Ontology	Ontology language-DAML+OIL SW Language-WSDL & DAML-S	Software Component Classification and Retrieval	Semantic representation mechanisms are employed for the purpose of annotation of a software component with a semantic description of the operations and services that it provides. This approach helps to automate reusable software searches on the World Wide Web.
Lu et al. (2005) [136]	SWS	Language: WSDL Query specification: Datalog	Composing and brokering software components on the web.	A web service specification is proposed to control a larger number of existing web services and to define a web service synthesizer for the purpose of dynamic generation of the implementation of a service specification. Here the web service is basically a software component described using XML.
Zhangjian et al. (2005) [137]	SWS+ Ontology	Ontology language-DAML+OIL SW Language-WSDL & DAML-S	Software component reuse and retrieval	Ontology & SWS help to provide same semantic description on the user's side as well as the component repository's end which makes the retrieval process much easier.
Song et al. (2005) [138]	Ontology	-	Enhance the semantics of software component registry using ontology	The ebXML(Electronic Based XML) information model is extended to accommodate a ontology attribute of the software component thereby increasing the semantics without compromising on the interoperability ability of the registry.
Korthaus et al. (2006) [139]	Ontology	Ontology language-OWL	Software component specification	Ontology helps to yield machine readable & machine-processible component artifacts to stakeholders for effective software development
Sun et al. (2006) [140]	Ontology	RDF, Ontology editor-Protégé 2.1	Software component retrieval	The initial request query generated by the user is extended with the help of query reasoning into appropriate RDF formats and retrieval of the software component is based on the degree of similarity and relatedness using fuzzy logic.
Ha et al. (2006) [141]	SWS+ Ontology	Ontology Language:- OWL UML is used to map OWL to WSDL	Web service and CBSE integration for E-business environment.	This approach is useful for dynamic service generation from components.
Laliwala et al. (2006) [142]	SWS+ Ontology	Ontology language-OWL, OWL-S, SWS Standards-WSRF, WSN	Publish and access software components as services on the	The concept of Semantic Web Services and Service-oriented architecture was used to publish and access services as components on the web. Event condition rules were developed using the SWRL.

Table A.2 Semantic web in Component Based Software Engineering

Source	Technology	Tools	Purpose	Outcome
		Other-BPEL & SWRL	web	ForwardChaining and BackwardChaining algorithms are employed for dynamic composition and for the dynamic generation of BPEL schema at run time.
Hyland-Wood et al. (2006) [143]	Ontology	Ontology Language-OWL-DL Query language-SPARQL	Software component maintenance	Ontology helps to segregate the software component from the system metadata and enables a language independent relational navigation through the CBSS for high understanding & thus, maintenance.
Liu et al. (2006) [76]	Ontology	Ontology Language- OWL	Software component registration with the help of ontology	Ontological languages and reasoners could be used to develop complex information registry knowledge base which would help in realizing the complex information resources interoperability at a semantic level.
Sjachyn and Beus-Dukic (2006) [144]	Ontology	-	Software component identification & classification	Ontology is used to represent the hierarchy of domain knowledge in the form of a generic taxonomy of classes and sub-classes and describes general specifications of components. Taxonomies are continually edited and expanded due to the dynamic nature of the methodology proposed by the authors, SemaCS.
Khemakhem et al. (2006) [145]	Ontology	Language-RDF/XML Editor-Protégé	Automatic software component discovery & retrieval	The process of automatic generation of query from developer specification and searching through a repository of software components is modeled as ontology.
Graubmann et al. (2006) [146]	Ontology	Language: OWL+SWRL	Software component annotation	Use of logic-on-demand and triple semantic model is done to build dynamic & flexible software annotation process for component composition.
Pahl (2007) [147]	Ontology	Language-DAML+OIL	Description of software component	Specific properties about the safety and business of software components are encoded into description logic and ontology framework in order to match the required and the provided components from the repository.
Quan et al. (2007) [122]	Ontology	Language-OWL-DL Editor-Protégé -2	Software component retrieval	Translation of users query into the same ontology as the repository's, efficiently augments the retrieval process.
Li et al. (2008) [78]	Ontology	OWL-DL	Software Component retrieval	Mismatch issue is tackled in software component retrieval process
Witte et al. (2008) [148]	Ontology	Language-OWL-DL Reasoner-Racer Editor-Protégé -2 Query language-nRQL	Software Component maintenance	Ontological representation of source code and documentation to eases the process of software maintenance.
Retkowitz et al. (2008) [149]	Ontology	Language-OWL	Interoperability of software components which deliver services in an Adaptive Smart Home environment.	Ontology are used to represent services based on OSGi in an Adaptive Smart Home to resolve the service heterogeneity issues due to service interface incompatibility.
Talevski et al. (2008) [150]	Ontology	Language –OWL Framework- Jena	Software component management	Software Component Ontology proposed defines common sharable software component knowledge like component concepts & relationships for better component based software management.
Yang et al.	Ontology	8-tuple ontology	Discovery, match	Addition of a semantic ability function helps to

Table A.2 Semantic web in Component Based Software Engineering

Source	Technology	Tools	Purpose	Outcome
(2009) [151]		instead of a 3-tuple ontology like RDF or OWL	& selection of software components.	analyze how well a component can cater to the user's request.
Zhang et al. (2009) [152]	Ontology	Ontology language-OWL+SWRL Editor-Protege-OWL/SWRL APIs	Improvement of the classical CBSE architectural styles with dynamic runtime semantic validation	The authors introduce extensible knowledge base called SACoCo by applying semantic web technology. Validation of the proposed methodology is done with SWRL rules.
Wang et al. (2009) [83]	Ontology	Ontology language-OWL and SWRL Editor-Prote´ge´ SWRL tab Reasoner-Jess Query-Prote´ge´ SWRL query tab	Modeling and verification of software component models	Reasoning engine is shown to detect inconsistencies and queries could be used to understand the desired properties of the model. However, current evaluation of the proposed work is primarily based on manually generated or some relatively small component models.
Zygkostiotis et al. (2009) [79]	Ontology	Ontology language: OWL-DL Software for tool development: Eclipse	Annotation, publication ad discovery of software components.	Authors have incorporated an existing Semantic Web Service publication and discovery solution into Semantic web component annotation and discovery tool in which functional and non-functional profiles of Java components are developed using an OWL.
Paquette and Masmoudi et al. (2010) [153]	Ontology	Ontology language-OWL-DL Editor- Protégé	Software component aggregation for the development of software packages	This aggregation framework for ontology is developed using TFLOS scenario editor. The use of ontology facilitates software component classification and inference when it is coupled with the DL inference engine and query language.
Wu et al. (2010) [71]	Ontology	Ontology language-OWL Editor- Protégé	Detecting vulnerable software components in large software repositories	Semantic templates record the faults in software elements & vulnerabilities which caused them & classify the vulnerability types like denial of service and injection.
Paulheim et al. (2010) [154]	Ontology	Ontology language-RDF/XML	Integration heterogeneous UI software components	The system, its UI components and the data exchanged by them is modeled in an ontological format for the integration of components. The proposed technique has been applied to integrate Java and Flex based components without loss of modularity and code-tangling.
Jiang et al. (2010) [155]	SWS	WSDL-S	Software component testing	The authors developed a prototype using .NET for black-box testing of semantic web services which could be also used for testing a software component.
Long et al. (2010) [156]	Ontology	Ontology language-OWL Editor- Protégé	Components & Ontology are used together for the purpose of building MES (Manufacturing Execution System)	Difficulties in traditional software development methods are removed with the help of MES. Pharmaceutics MES is currently being used by the pharmaceutical industry.
Li, et al. (2010) [157]	Ontology	Ontology language-OWL Editor- Protégé	Precision analysis of a software component retrieval	Ontology based retrieval eliminates the component mismatch problem, also helps to predict quality attributes for CBD by analyzing the search precision using the QAOCS model.
Kzaz et al. (2010) [158]	Ontology	Ontology language-OWL; XSLT	Integration of business software components	Semantic integration of business components using ontology ensures that meaning is assigned to the data and services of each component and smooth

Table A.2 Semantic web in Component Based Software Engineering

Source	Technology	Tools	Purpose	Outcome
		transformation is used for automatic generation of OWL from UML description.		exchange of data and services can take place between these heterogeneous components without the occurrence of any naming conflicts.
Castillo-Berrera et al. (2011) [84]	Ontology	Language-OWL Reasoner-Pellet Editor-Protégé Query language-SPARQL	Classification, matching & Search of software components	Ontologies make for a useful approach to capture and use the knowledge of components in a component based software factory.
Alnusair et al. (2011) [65]	Ontology	Language-OWL & RDF Framework-Jena Query language-SPARQL	Identification and retrieval of software components from software reuse libraries	Authors propose COMPRE(based on OWL), a component representation ontology which defines its own relations and class hierarchy for semantic component description in order to promote an easy and accurate retrieval of software components.
Garijo et al. (2011) [159]	LD + Ontology	Triple Store /Data Source- Allegro20 Query language-SPARQL LD browser-Pubby	Promote reusability of software component workflows.	LD makes possible to publish abstract and executable workflows in the form of open accessible workflow repositories, making the workflows reusable for different environment.
Kost et al. (2012) [160]	Ontology	Language-OWL Reasoner-Pellet Editor-Protégé	Privacy analysis of a CBS	Ontologies helps to semantically analyze a component based system's privacy leakages and privacy indicators, thus preventing the system from potential misuse.
Siddiqui et al. (2012) [66]	Ontology	Language-OWL, SWRL Query engine-SQWRL Editor- Protégé	Component description and architecture design.	Smooth conversion the domain model elements(i.e. features) to the architecture model elements (i.e. components) for CBS design can be performed using ontology.
Li and Zhang (2012) [161]	Ontology	-	Reusability of test cases for component based development	Generation of test cases with ontology and knowledge management model enables test engineers to retrieve and reuse test case flexibly and also to improve the design efficiency of test case.
Castillo-Barrera et al. (2012) [162]	Ontology	Language-CORBA-IDL (later converted to OWL-DL), Reasoner-Pellet, Query language-SPARQL	Interface contract matching among components for the purpose of a component integration	Ontology is employed to represent the interface's invariants, pre- and post-conditions as classes, properties, and relations which eases the process of verifying interface conformance of a software component with respect to the target system.
Francisco et al. (2013) [163]	SWS	WSDL & WSDL-S	Automatic testing of Web services, used to develop CBS.	Efficient test cases development of for the Web Services has been proposed to ensure the quality & behavior for efficient software component integration.
Khemakhem et al. (2013) [80]	Ontology	Ontology language-RDF +OWL Editor-Protégé	Software component integration	SEC++, a component retrieval engine, developed by authors covers the complete set of ontologies in order to discover and integrate atomic components.
Dai et al. (2013) [164]	Ontology	Ontology language-SWRL Query Engine-SQWRL	Semantic analysis of component based software(IEC 61499 standard)	The authors have proposed an multiple-layered ontological approach of automatic syntactic & semantic checking of component based control systems to ensure quality and functionality.
Castillo-Barrera et al. (2013) [165]	Ontology	Ontology LanguageCORBA-IDL++ (later converted	Software component integration	Ontology is used to calculate a software metric which evaluates the degree to which a software component is suitable to be integrated into a target system.

Table A.2 Semantic web in Component Based Software Engineering

Source	Technology	Tools	Purpose	Outcome
		to RDF), Reasoner-Pellet, Query language-SPARQL		
Kamer Karatas et al. (2014) [166]	Ontology	Frame ontology paradigm Editor-Protégé 3.4.7	Requirement reuse in component based software product lines	Proposed mechanism is applied at the requirement elicitation phase of software development & enables requirements reuse & management in a product line to eliminate the disadvantages of not reusing software requirements.
Rastgoo et al. (2014) [167]	Ontology	Languages – RDF/OWL Editor-Protégé	Reusing of software component requirements	Automated ontology generation using a standard design format like UML diagrams can increase speed and exactness of requirements reusing dramatically. Also, this ontology can measure completeness, preciseness and unambiguous of a set of requirements.
Abbasipour et al. (2014) [67]	Ontology	-	User requirement decomposition for software component selection.	Apart from automating the decomposition, ontology also aids in considering alternative decompositions and takes optional COTS components into consideration thus making it practical to develop many sets of COTS components which could be composed within a system in order to cater to user requirements
Dostal et al. (2014) [68]	LD + Ontology	Ontology language-XML Data analysis-DBPedia Spotlight	Software component specification analysis	This approach is useful for semantic enrichment of text for the purpose of software component detection, selection and disambiguation.
Samimi et al. (2014) [69]	Ontology	Ontology language-CycL	Software component discovery based on functionality.	The authors introduced semantic reasoning at the programming languages level thus facilitating semantic & dynamic communication among components.
Jayasudha et al. (2014) [168]	Ontology	-	Software component storage	The authors make use of two separate ontologies for the purpose of storage and retrieval of software components i.e. the Domain and Folder Ontology helps in the finding the needed and available relevant reuse artifacts for the project.
Konys (2015) [70]	Ontology	Languages –OWL Editor-Protégé	COTS component selection and evaluation	The chief benefit of the proposed COTS ontologies is to give repeatable and systematic knowledge with respect to available COTS ERP components and methodologies supporting COTS component selection process.
Lv et al., 2016 [169]	Ontology	Ontology language: OWL, Rule language: SWRL	Software component integration for enterprise business collaboration.	Two ontologies (Software component ontology and Manufacturing software component (MSC) core ontology) are constructed to map the relevant components that are suitable for integration. Post construction of ontologies, a novel semantic similarity algorithm for mapping the two ontologies is developed.
Bhat et al., 2017 [170]	Ontology + Linked Data	Data analysis-DBPedia Ontology Language-OWL-DL Query language-SPARQL	Automatic annotation of architectural elements of software components	The authors describe the generation recommendations pertaining to alternate architectural solutions for software components on the basis of factors like architectural style, pattern, and software technology using look-up queries formulated using the SPARQL query language.
Yanes et al., 2017 [171]	Ontology	Ontology Language-OWL-DL Query language-SPARQL Editor-Protégé	Recommender system for COTS component selection	The recommender system uses an ontology of COTS components, named ONTOCOTS, that describes COTS components and unifies their heterogeneous descriptions available on the Web along with a user model that represents user preferences and interest domains. Results list is ranked according to the satisfaction degree of user requirements that the

Table A.2 Semantic web in Component Based Software Engineering

Source	Technology	Tools	Purpose	Outcome
				component satisifies and preferences.
Du et al., 2018 [172]	Ontology	NeoJ (graph-based data management system) Query language-Cypher	Identification of vulnerable software components	A software vulnerability knowledge graph model is proposed that integrates CVE (Common Vulnerabilities and Exposures) information, Java Component metadata in Maven repository and project collaboration data on Github. Two ontology matching approaches are constructed to develop the knowledge graph. Experimental results show that matching between CVE project version and Maven project version are highly promising with high accuracies.

Appendix B:

Class used to demonstrate the calculation of the cognitive complexity metric

Class AbstractTitle

```
public abstract class AbstractTitle extends Object implements Cloneable {

public static final int TOP = 0;                                          //SEQ=1
public static final int BOTTOM = 1;
public static final int RIGHT = 2;
public static final int LEFT = 3;
public static final int NORTH = 0;
public static final int SOUTH = 1;
public static final int EAST = 2;
public static final int WEST = 3;
public static final int CENTER = 4;
public static final int MIDDLE = 4;
protected boolean notify;
protected int position;
protected int horizontalAlignment;
protected int verticalAlignment;
protected Insets insets;
protected List listeners;

   protected AbstractTitle(int position, int horizontalAlignment,          //MD1=1
   int verticalAlignment, Insets insets) {
      if (!this.isValidPosition(position)) {                    //ITE1,1+MC1,1= 2+2
      throw new IllegalArgumentException("AbstractTitle: Invalid        TH1,1,1=2
       position.");
        }

     if ((horizontalAlignment!=LEFT) &&                                   //ITE1,2= 2
         (horizontalAlignment!=CENTER) &&
         (horizontalAlignment!=RIGHT)) {
      throw new IllegalArgumentException("                              //TH1,2,1=2
      AbstractTitle: Invalid horizontal alignment.");
        }

    if ((verticalAlignment!=TOP) &&                                       //ITE1,3= 2
        (verticalAlignment!=BOTTOM) &&
        (verticalAlignment!=MIDDLE)) {
     throw new IllegalArgumentException("                               //TH1,3,1= 2
     AbstractTitle: Invalid vertical alignment.");
        }

       this.position = position;
       this.horizontalAlignment = horizontalAlignment;
       this.verticalAlignment = verticalAlignment;
       this.insets = insets;
       this.listeners = new java.util.ArrayList();                        //MC11=2
       this.notify = true;
    }

   protected AbstractTitle(int position,                                  //MD2=1
   int horizontalAlignment, int verticalAlignment) {
      this(position, horizontalAlignment,                    //MC2,1 + MC2,2 =2+2
      verticalAlignment, new Insets(2, 2, 2, 2));
    }

   protected AbstractTitle() {                                            //MD3=1
      this(TOP, CENTER, MIDDLE);                                          //MC3,1=2
    }
```

```
public Object clone() {                                                    //MD4=1
   AbstractTitle duplicate = null;
   try {                                                                   //TC4,1=2
     duplicate = (AbstractTitle)(super.clone());                           //MC4,1,1=2
     }
   catch (CloneNotSupportedException e) {
     throw new RuntimeException("AbstractTitle.clone()");                  //TH4,1=2
     }
   duplicate.setNotify(false);                                             //MC4,1=2
   duplicate.setInsets((Insets)this.getInsets().clone());//MC4,2+ MC4,3+MC4,4=2+2+2
   duplicate.setNotify(true);                                              //MC4,5=2
   return duplicate;

 }

public boolean getNotify() {                                               //MD5=1
   return this.notify;
 }

public void setNotify(boolean flag) {                                      //MD6=1
   this.notify = flag;
 }

public int getPosition() {                                                 //MD7=1
   return this.position;
 }

public void setPosition(int position) {                                    //MD8=1
   if (this.position!=position) {                                          /ITE8,1=2
     this.position = position;
     notifyListeners(new TitleChangeEvent(this));                          //MC8,1,1 + MC8,1,2 =2+2
   }
 }

public int getHorizontalAlignment() {                                      //MD9=1
   return this.horizontalAlignment;
 }

public void setHorizontalAlignment(int alignment) {                        //MD10=1
   if (this.horizontalAlignment!=alignment) {                              //ITE10,1=2
     this.horizontalAlignment = alignment;
     notifyListeners(new TitleChangeEvent(this));                          //MC10,1,1 + MC10,1,2 =2+2
   }
 }

public int getVerticalAlignment() {                                        //MD11=1
   return this.verticalAlignment;
 }

public void setVerticalAlignment(int alignment) {                          //MD12=1
     if (this.verticalAlignment!=alignment)                                //ITE12,1=2
     this.verticalAlignment = alignment;
     notifyListeners(new TitleChangeEvent(this));                          //MC12,1,1 + MC12,1,2 =2+2
   }
 }

public Insets getInsets()                                                  //MD13=1
   return this.insets;
 }

public void setInsets(Insets insets)                                       //MD14=1
   if (!this.insets.equals(insets)) {                                      //ITE14,1 +MC14,1=2+2
     this.insets = insets;
     notifyListeners(new TitleChangeEvent(this));                          //MC14,1,1+MC14,1,2=2+2
   }
 }
```

```
   public abstract boolean isValidPosition(int position);
   public abstract double getPreferredWidth(Graphics2D g2);
   public abstract double getPreferredHeight(Graphics2D g2);
   public abstract void draw(Graphics2D g2, Rectangle2D titleArea);

   public void addChangeListener(TitleChangeListener listener) {          //MD15=1
      listeners.add(listener);                                            //MC15,1=2
    }
   public void removeChangeListener(TitleChangeListener listener) {       //MD16=1
      listeners.remove(listener);                                         //MC16,1=2
    }

   protected void notifyListeners(TitleChangeEvent event) {               //MD17=1
      if (this.notify) {                                                  //ITE17,1=2
       java.util.Iterator iterator = listeners.iterator();               //MC17,1,1=2
       while (iterator.hasNext()) {                                      ITR17,1,1+MC17,1,2=3+2
           TitleChangeListener listener = (TitleChangeListener)
           iterator.next();                                              //MC17,1,1,1=2
           listener.titleChanged(event);                                 //MC17,1,1,2=2
     }
    }
  }
}
```

Appendix C:

Table C.1 Prediction results of the shortlisted techniques over four releases of JFreeChart using 10-fold validation

Tech.	JFreeChart 0.7.0						JFreeChart 0.7.1					
	Acc.	AUC	F-S	G-m1	G-m2	MCC	Acc.	AUC	F-S	G-m1	G-m2	MCC
BN	72.2	0.770	67.5	67.5	71.4	0.433	76.6	0.800	74.1	74.1	76.5	0.528
NB	75.6	0.760	66.7	67.8	71.3	0.504	73.4	0.801	65.6	66.3	70.2	0.460
FLDA	70.0	0.727	64.9	64.9	69.1	0.387	77.9	0.831	74.8	74.8	77.4	0.551
LR	67.8	0.741	61.3	61.4	66.3	0.339	79.1	0.836	75.9	75.9	78.5	0.575
MLP	76.7	0.761	72.7	72.7	76.0	0.524	82.9	0.882	81.1	81.2	82.9	0.656
RBFN	74.4	0.740	69.3	69.4	73.2	0.476	76.0	0.768	71.2	71.3	74.6	0.510
ADB	76.7	0.813	70.4	69.2	72.8	0.522	72.8	0.763	69.1	69.1	72.3	0.448
BAG	78.9	0.822	72.5	73.1	76.0	0.571	77.9	0.848	73.7	73.8	76.7	0.549
DAG	68.9	0.767	61.1	61.3	66.5	0.358	73.4	0.771	70.4	70.4	73.2	0.463
FC	72.2	0.733	65.8	65.9	70.3	0.429	72.2	0.739	69.0	69.0	71.9	0.438
LB	73.3	0.803	66.7	66.9	71.2	0.451	77.2	0.826	74.6	74.6	77.0	0.540
AM	67.8	0.741	61.3	61.4	66.3	0.339	79.1	0.836	75.9	75.9	78.5	0.575
RSS	76.7	0.796	71.2	71.4	75.0	0.521	75.3	0.834	70.7	70.8	74.1	0.497
DTNB	77.8	0.848	73.7	73.7	76.9	0.545	74.7	0.794	71.4	71.4	74.3	0.487
FURIA	76.7	0.756	69.6	70.1	73.6	0.523	76.0	0.775	72.5	72.4	75.3	0.511
MOD	78.9	0.781	74.7	74.7	77.8	0.568	82.9	0.820	79.4	79.6	81.7	0.654
NNGE	77.8	0.777	75.0	75.0	77.6	0.551	77.9	0.778	75.5	75.5	77.7	0.554
PART	75.6	0.717	70.3	70.4	74.1	0.498	81.0	0.810	78.9	78.9	80.9	0.617
GANN	76.7	0.785	72.0	72.0	75.5	0.522	79.8	0.829	76.5	76.5	79.0	0.588
HFT	75.6	0.762	67.6	68.4	72.1	0.501	72.8	0.803	64.5	65.3	69.4	0.447
J48	68.9	0.661	62.2	62.3	67.2	0.360	78.5	0.783	75.4	75.4	78.0	0.563
NNEP	78.9	0.739	71.6	72.6	75.3	0.575	79.8	0.843	77.1	77.1	79.4	0.590
RF	76.7	0.849	71.2	71.4	75.0	0.521	84.8	0.898	82.1	82.2	84.0	0.692
CART	76.7	0.787	70.4	70.8	74.4	0.522	76.6	0.778	73.4	73.4	76.1	0.525
SYSFOR	73.3	0.829	66.7	66.9	71.2	0.451	76.0	0.839	72.5	72.4	75.3	0.511

Table C.1 Prediction results of the shortlisted techniques over four releases of JFreeChart using 10-fold validation (contd.)

Tech.	JFreeChart 0.7.2						JFreeChart 0.7.3					
	Acc.	AUC	F-S	G-m1	G-m2	MCC	Acc.	AUC	F-S	G-m1	G-m2	MCC
BN	89.23	0.958	89.7	90.1	89.3	0.801	80.1	0.839	75.6	75.7	78.7	0.592
NB	75.89	0.856	73.1	73.3	75.4	0.516	83.3	0.843	78.7	79.1	81.2	0.661
FLDA	86.15	0.927	86.3	86.5	86.3	0.731	79.5	0.825	71.9	73.1	75.6	0.587
LR	85.64	0.924	84.9	84.9	85.6	0.712	76.3	0.850	71.3	71.4	75.1	0.513
MLP	90.26	0.943	89.8	89.8	90.3	0.805	82.7	0.928	80.6	80.6	82.8	0.651
RBFN	81.03	0.886	80.4	80.5	81.1	0.621	78.9	0.801	71.8	72.6	75.5	0.570
ADB	91.28	0.986	91.1	91.2	91.4	0.828	76.9	0.877	72.3	72.3	75.8	0.526
BAG	92.31	0.984	92.2	92.3	92.4	0.850	81.4	0.913	77.2	77.3	80.0	0.618
DAG	86.15	0.915	86.8	87.3	86.1	0.743	84.0	0.876	80.0	80.2	82.4	0.672
FC	90.77	0.943	90.9	91.1	90.9	0.825	78.2	0.810	73.8	73.9	77.1	0.553
LB	95.38	0.987	95.3	95.3	95.5	0.910	80.8	0.873	76.6	76.6	79.5	0.605
AM	85.64	0.924	84.9	84.9	85.6	0.712	76.3	0.850	71.3	71.4	75.1	0.513
RSS	91.28	0.972	91.4	91.6	91.4	0.834	80.8	0.919	75.8	76.0	78.9	0.606
DTNB	93.85	0.961	93.8	93.9	94.0	0.882	80.1	0.870	75.2	75.4	78.4	0.592
FURIA	91.28	0.930	91.0	91.1	91.4	0.827	80.1	0.816	75.2	75.4	78.4	0.592
MOD	93.33	0.933	93.0	93.0	93.3	0.866	84.0	0.830	80.3	80.4	82.7	0.672
NNGE	89.74	0.899	89.5	89.5	89.9	0.796	82.7	0.821	79.4	79.4	81.9	0.645
PART	93.85	0.936	93.7	93.7	94.0	0.879	82.1	0.860	78.8	78.8	81.4	0.633
GANN	90.77	0.985	90.5	90.6	90.9	0.817	84.0	0.897	80.6	80.7	83.0	0.671
HFT	75.89	0.856	73.1	73.3	75.4	0.516	83.3	0.844	78.7	82.3	84.5	0.661
J48	92.30	0.925	92.1	92.1	92.4	0.847	80.8	0.833	77.6	74.5	77.1	0.608
NNEP	91.79	0.961	91.7	91.8	91.9	0.839	81.4	0.853	75.2	82.6	85.1	0.625
RF	93.33	0.992	93.1	93.2	93.5	0.868	84.0	0.932	80.3	74.7	76.9	0.672
CART	91.28	0.932	91.2	91.3	91.5	0.830	78.5	0.823	75.0	77.4	80.6	0.579
SYSFOR	92.31	0.990	92.1	92.2	92.4	0.848	80.1	0.900	76.3	76.3	79.3	0.593

Appendix D:

Table D.1 Prediction results of the shortlisted techniques over four releases of Heritrix using 10-fold validation (contd.)

Tech.	Heritrix 0.4.0						Heritrix 0.6.0					
	Acc.	AUC	F-S	G-m1	G-m2	MCC	Acc.	AUC	F-S	G-m1	G-m2	MCC
BN	63.0	0.7190	65.1	65.3	62.8	0.267	69.0	0.747	62.4	62.5	68.2	0.362
NB	70.6	0.7780	67.3	67.6	69.9	0.414	69.6	0.699	55.2	56.0	63.1	0.343
FLDA	65.6	0.7410	63.1	63.1	65.2	0.310	69.6	0.717	61.8	61.8	68.0	0.365
LR	68.1	0.7540	66.1	66.1	67.8	0.361	71.9	0.721	60.7	61.0	67.4	0.399
MLP	69.7	0.7770	69.0	69.0	69.7	0.395	69.0	0.675	59.5	59.6	66.4	0.345
RBFN	68.1	0.7400	66.1	66.1	67.8	0.361	66.0	0.616	55.4	55.4	62.9	0.282
ADB	68.9	0.7370	65.4	65.6	68.1	0.380	75.4	0.758	65.6	66.0	71.3	0.476
BAG	72.3	0.8170	70.3	70.3	71.9	0.445	71.9	0.738	60.7	61.0	67.4	0.399
DAG	69.7	0.7510	68.4	68.4	69.6	0.394	71.4	0.725	59.5	59.9	66.5	0.386
FC	61.3	0.6230	55.8	56.1	60.0	0.227	71.9	0.734	61.3	61.6	67.9	0.401
LB	65.5	0.7470	65.5	65.5	65.5	0.312	77.2	0.819	69.8	69.9	74.8	0.517
AM	68.1	0.7540	66.1	66.1	67.8	0.361	71.9	0.721	60.7	61.0	67.4	0.399
RSS	72.3	0.7450	69.2	69.4	71.6	0.448	76.6	0.792	66.1	66.9	71.6	0.502
DTNB	68.1	0.7210	65.5	65.6	67.7	0.361	70.2	0.725	59.8	60.0	66.7	0.365
FURIA	68.9	0.6890	65.4	65.6	68.1	0.380	73.1	0.749	58.9	60.3	65.8	0.424
MOD	72.3	0.7210	70.3	70.3	71.9	0.445	72.2	0.687	59.1	60.1	66.1	0.410
NNGE	69.7	0.6990	70.5	70.6	69.7	0.398	70.2	0.682	61.1	61.1	67.6	0.370
PART	71.4	0.7460	70.7	70.7	71.4	0.428	70.2	0.735	54.9	56.1	62.8	0.356
GANN	64.7	0.7140	61.1	61.3	64.1	0.294	76.0	0.735	66.1	66.6	71.7	0.489
HFT	71.4	0.7790	68.5	68.7	70.9	0.430	70.2	0.700	55.7	56.6	63.4	0.356
J48	69.7	0.7410	68.4	68.4	69.6	0.394	69.0	0.698	56.9	57.2	64.4	0.336
NNEP	65.5	0.7540	63.7	63.8	65.4	0.310	68.4	0.716	55.7	56.1	63.5	0.322
RF	69.8	0.7870	67.3	67.4	69.4	0.395	79.5	0.829	72.9	72.9	77.3	0.567
CART	66.4	0.6700	64.3	64.4	66.2	0.327	69.0	0.664	56.2	56.7	63.9	0.334
SYSFOR	68.9	0.8030	67.3	67.3	68.7	0.377	74.9	0.782	63.9	64.5	69.9	0.463

Table D.1 Prediction results of the shortlisted techniques over four releases of Heritrix using 10-fold validation

Tech.	Heritrix 0.8.0						Heritrix 0.10.0					
	Acc.	AUC	F-S	G-m1	G-m2	MCC	Acc.	AUC	F-S	G-m1	G-m2	MCC
BN	66.3	0.713	65.9	73.3	66.5	0.338	68.8	0.739	61.7	61.7	67.9	0.356
NB	64.6	0.706	70.4	55.9	61.6	0.278	66.7	0.673	52.8	53.1	61.3	0.278
FLDA	62.7	0.679	66.7	59.8	61.7	0.244	66.7	0.701	55.5	55.5	63.3	0.290
LR	63.9	0.684	68.4	59.2	62.4	0.266	67.1	0.705	47.9	49.3	57.5	0.273
MLP	59.6	0.629	64.2	56.4	58.5	0.182	73.2	0.789	61.7	62.1	68.5	0.421
RBFN	65.7	0.670	68.5	64.9	65.3	0.308	68.8	0.681	49.3	51.2	58.5	0.313
ADB	71.1	0.758	74.7	66.2	69.8	0.415	70.6	0.721	60.0	60.1	67.1	0.370
BAG	68.1	0.722	71.7	64.7	67.1	0.354	73.2	0.773	62.2	62.5	68.9	0.422
DAG	66.3	0.731	69.2	65.1	65.8	0.319	68.4	0.727	56.3	56.4	64.1	0.320
FC	61.5	0.636	67.3	54.3	58.9	0.215	73.2	0.709	61.2	61.7	68.0	0.420
LB	69.3	0.745	72.4	66.8	68.6	0.379	71.4	0.735	61.2	61.3	68.0	0.389
AM	63.9	0.684	68.4	59.2	62.4	0.266	67.1	0.705	47.9	49.3	57.5	0.273
RSS	69.9	0.743	72.8	67.8	69.3	0.391	73.6	0.800	61.1	61.8	67.9	0.428
DTNB	61.5	0.696	64.0	61.9	61.2	0.225	77.1	0.753	68.6	68.8	74.0	0.509
FURIA	68.7	0.690	71.1	68.4	68.4	0.369	75.8	0.770	64.1	64.9	70.2	0.477
MOD	67.5	0.671	70.3	66.3	67.0	0.343	75.8	0.713	62.2	63.7	68.4	0.479
NNGE	57.2	0.565	62.4	53.0	55.6	0.131	68.0	0.669	60.2	60.2	66.7	0.335
PART	62.1	0.594	64.8	62.1	61.7	0.236	69.3	0.712	62.0	62.1	68.2	0.363
GANN	66.3	0.701	68.9	66.0	66.0	0.320	76.2	0.788	68.6	68.6	74.1	0.495
HFT	64.5	0.708	70.4	55.9	61.6	0.278	62.3	0.622	49.1	49.2	58.1	0.194
J48	63.3	0.568	67.0	60.8	62.4	0.256	74.5	0.699	67.0	67.0	72.8	0.462
NNEP	64.5	0.633	68.1	62.0	63.7	0.281	73.2	0.765	65.2	65.1	71.2	0.433
RF	67.5	0.750	71.3	63.6	66.4	0.341	80.1	0.861	72.3	72.5	76.9	0.574
CART	68.7	0.724	71.7	66.6	68.1	0.367	74.0	0.751	66.3	66.3	72.2	0.452
SYSFOR	69.3	0.767	73.8	62.3	67.3	0.378	76.2	0.820	68.9	68.9	74.4	0.497

Appendix E:

The screenshots for the selection of change-prone Java files of Heritrix 0.4.0 have been included in this appendix. Same methodology detailed in the sub-sections of Section 5.4.2 and 5.4.3 of Chapter 5 is employed to select the change-prone Java files of the Heritrix versions.

As mentioned in Section 5.4.2 of Chapter 5, in order to select the relevant change-prone Java files, the rules generated by the RF technique with respect to each version analysed for change-proneness prediction are employed as SWRL rules in the ontology. These rules when applied on the Java files instances of a software version essentially infer the relevant instances according to the given rules, thereby classifying the instances as change-prone or not change-prone according to the individual rules. The ontology is then queried for the selection of change-prone Java files of the given version.

Post generating prediction models with respect to each plugin project version, the methodology proposed by Mitchell [228, p.72] is utilized to formulate rules out of the decision trees that were constructed by the RF technique for each of the Heritrix version.

However, instead of including all the rules in the proposed ontology for Java files, rules derived from RF are further screened according to their individual strength of accuracy and as earlier stated in Table 5.7, a total of 16 rules are shortlisted from Heritrix 0.2.0 and are employed as SWRL rules to classify the Java files of Heritrix 0.4.0 as change-prone and not change-prone.

Post the execution of SWRL rules, the same three SQWRL queries named as: *FileCountforCPRules*, *FileCountforNCPRules* and *FinalChangeProneFiles*; built for each version of JFreeChart in Section 5.4.3 of Chapter 5 have been employed to select the change-prone Java files for Heritrix versions. The following sub-sections address these queries in detail with screenshots of actual execution for Heritrix 0.4.0.

- FileCountforCPRules and FileCountforNCPRules

Query *FileCountforCPRules* displays the file instances that exist corresponding to Heritrix 0.4.0. It also captures their file name and their total count with respect to each of the change-prone rules derived from its previously released version Heritrix 0.2.0.

Similarly Query *FileCountforNCPRules* displays the file instances that exist corresponding to Heritrix 0.4.0 and its results show the file name and their total count with respect to each of the not change-prone rules derived from its previously released version Heritrix 0.2.0.

Figures E.1 are the results of the *FileCountforCPRules* and *FileCountforNCPRules* query. Figure E.1 indicates that 155 rows each are returned for both the queries. This statistic is expected since as seen in column 3 of Table 4.3 of Section 4.2.3 of Chapter 4, the Heritrix 0.4.0 has a total of 155 Java files that have been employed in the subsequently released version Heritrix 0.6.0.

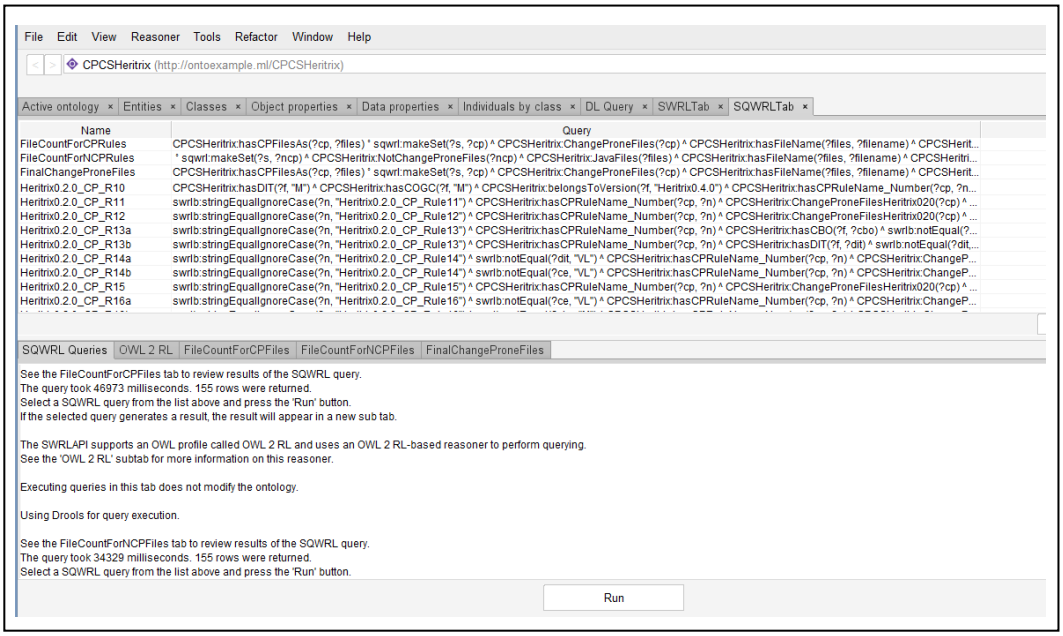

Figure E.1 Screenshot of the succesful execution of the FileCountforCPRules and FileCountforNCPRules query for Heritrix 0.4.0

Figure E.2 indicates the output from the *FileCountforCPRules* query in which the file instances and their filenames along with their number of times each of the file satisfied a change-proneness governing rule of Heritrix 0.2.0.

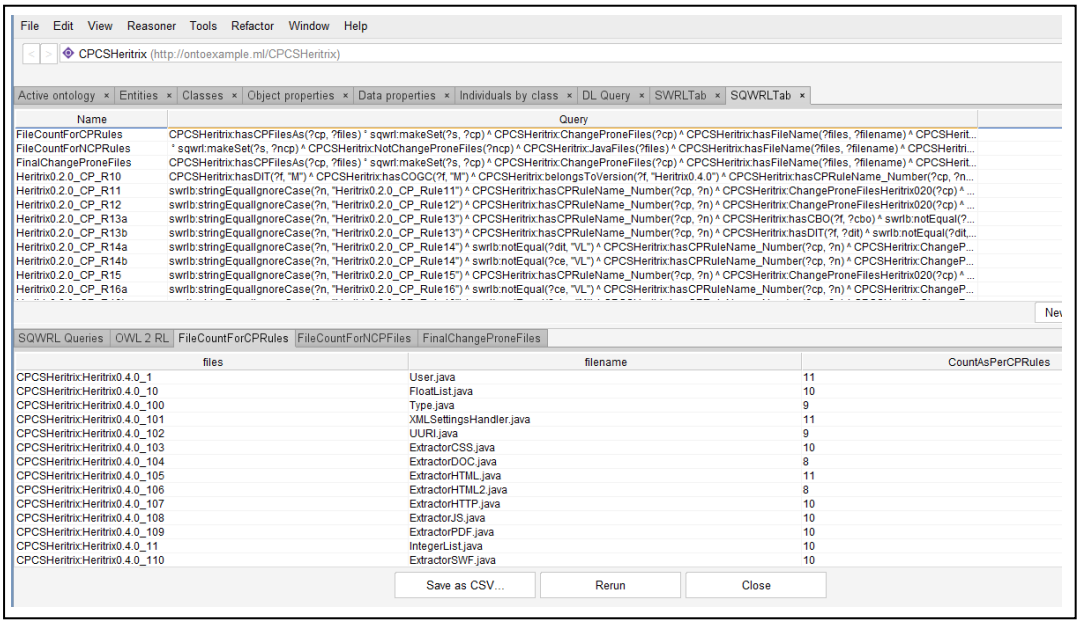

Figure E.2 Screenshot of the results of the FileCountforCPRules query for Heritrix 0.4.0

Figure E.3 indicates the output from the query in which the file instances and their filenames along with their number of times each of the file of Heritrix 0.4.0 satisfied a non-change-proneness governing rule of Heritrix 0.2.0.

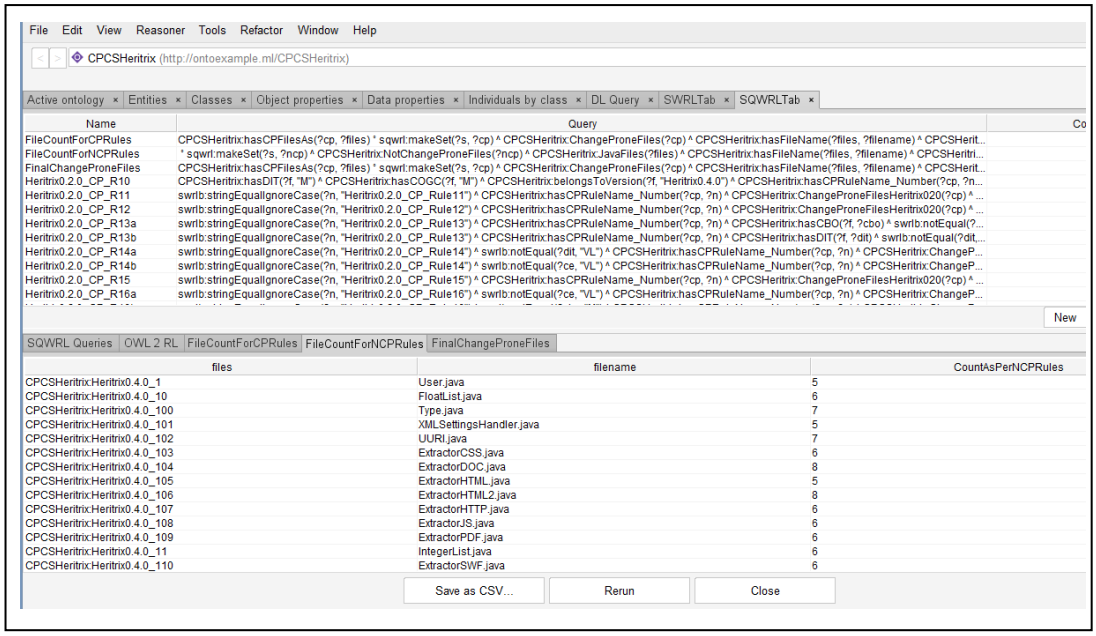

Figure E.3 Screenshot of the results of the FileCountforNCPRules query for Heritrix 0.4.0

- FinalChangeProneFiles

As elucidated for JFreeChart version 0.7.0 in Section 5.4.3 of Chapter 5, query *FinalChangeProneFiles* essentially employs the computed change-prone and not change-prone counts of each of the files, calculated using the queries FileCountforCPRules and FileCountforNCPRules, and compares these counts with

respect to each file. The Java file having the `CountAsPerCPRules` higher than or equal to its `CountAsPerNCPRules` is selected as a Java file which has the possibility to be used with change in its upcoming version. As per our current context of application, the query displays the Java file instances corresponding to Heritrix 0.4.0 that are selected to be change-prone in their successive version along with their file name.

Figures E.4 and E.5 are the results of the *FinalChangeProneFiles* query. Figure E.4 indicates that the query returns 116 rows, thus exhibiting that 116 Java files are selected to be change-prone from the 155 Java files in Heritrix 0.4.0, according to the rules derived from its previous version Heritrix 0.2.0. Figure E.5 displays the output from the query which consists of the 116 selected change-prone file instances and their filenames.

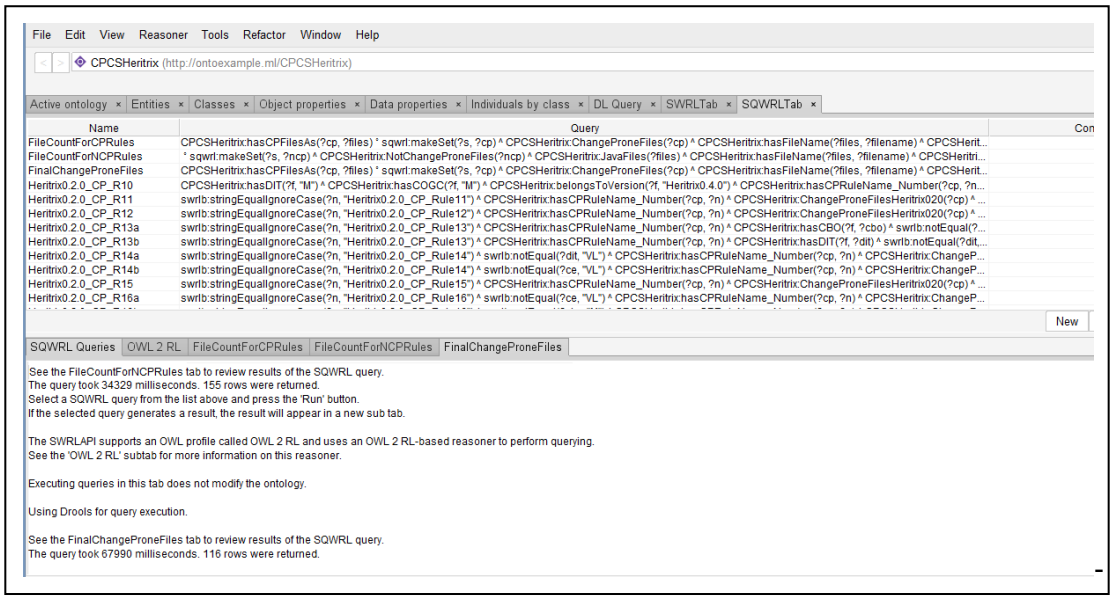

Figure E.4 Screenshot exhibiting the successful execution of the FinalChangeProneFiles query for Heritrix 0.4.0

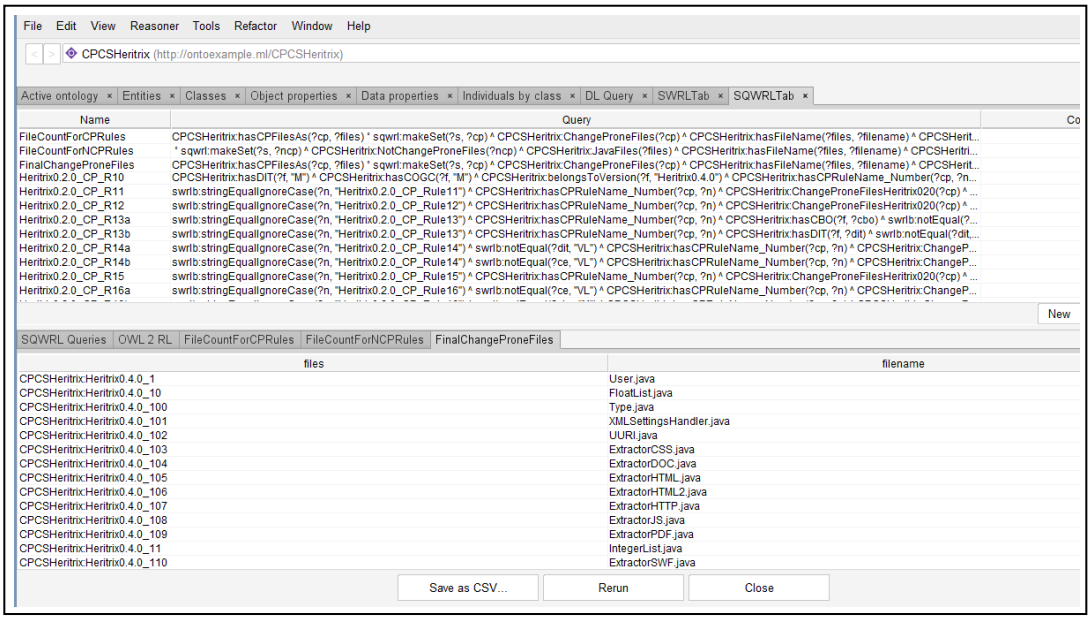

Figure E.5 Screenshot exhibiting the final 116 change-prone files dervied via the FinalChangeProneFiles query for Heritrix 0.4.0

Appendix F:

The screenshots for the selection of change-prone Java files of Heritrix 0.6.0 have been included in this appendix. As mentioned in Section 5.4.2, the SWRL rules derived from the RF algorithm are employed in the ontology in an incremental manner for selection of change-prone Java files. Therefore, for the selection Java files from Heritrix 0.6.0 that could be used with change in its successive release, the extracted rules from the previously released versions i.e. Heritrix 0.2.0 and Heritrix 0.4.0 are utlized. The ten rules shortlisted from Heritrix 0.4.0 (mentioned in Table 5.7) are utlized along with the rules from Heritrix 0.2.0 to classify Java files of Heritrix 0.6.0. These rules when applied on the Java files instances of Heritrix 0.6.0 essentially classify the instances as change-prone or not change-prone according to the individual rules. The ontology is then queried for the selection of change-prone Java files of Heritrix 0.6.0.

The three SQWRL queries: FileCountforCPRules, FileCountforNCPRules and FinalChangeProneFiles; are again executed for the selection of the change-prone Java files of Heritrix 0.6.0. However, each of these queries is slightly modified to suit the context as already detailed in Section 5.4.4 wherein change-prone Java files from JFreeChart 0.7.1 were selected using SWRL rules from JFreeChart 0.7.0 and JFreeChart 0.6.0.

- FileCountforCPRules

Query *FileCountforCPRules* displays the file instances that exist corresponding to Heritrix 0.6.0. It also captures their file name and their total count with respect to each of the change-prone rules derived from its previously released version Heritrix 0.2.0 and Heritrix 0.4.0.

Figures F.1 and F.2 the results of the *FileCountforCPRules* query. Figure F.1 indicates that 195 rows each are returned for the query. This statistic is expected since as seen in column 3 of Table 4.3 of Section 4.2.3 of Chapter 4, the Heritrix 0.6.0 has a total of 155 Java files that have been employed in the subsequently released version Heritrix 0.8.0.

Figure F.2 indicates the output from the *FileCountforCPRules* query in which the file instances and their filenames along with their number of times each of the file satisfied a change-proneness governing rule of Heritrix 0.4.0 and Heritrix 0.2.0.

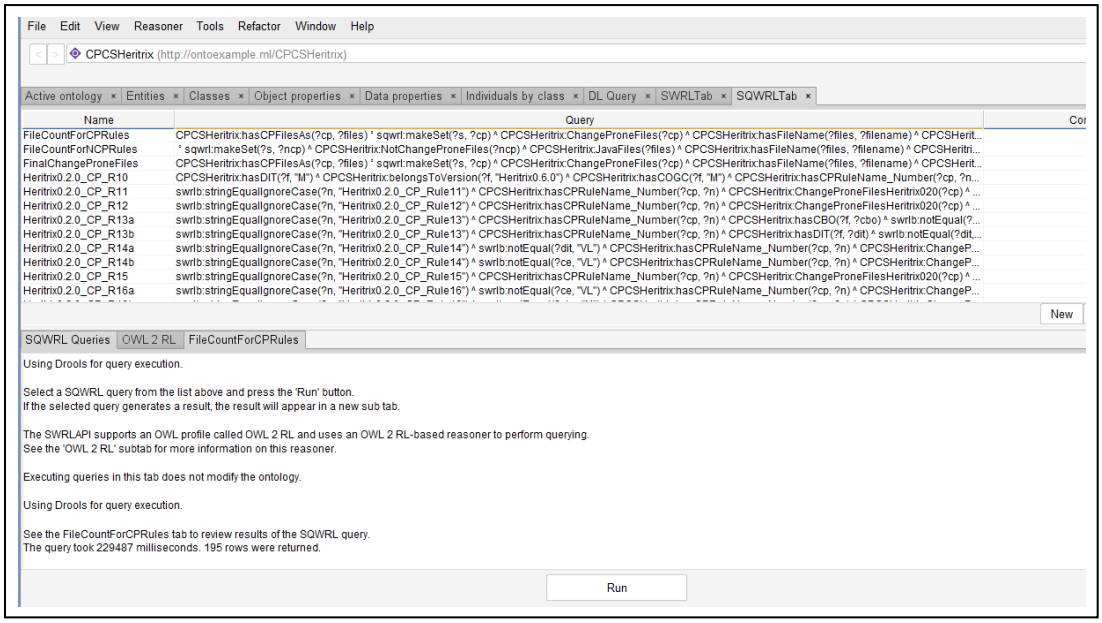

Figure F.1 Screenshot exhibiting the succesful execution of the FileCountforCPRules query for Heritrix 0.6.0

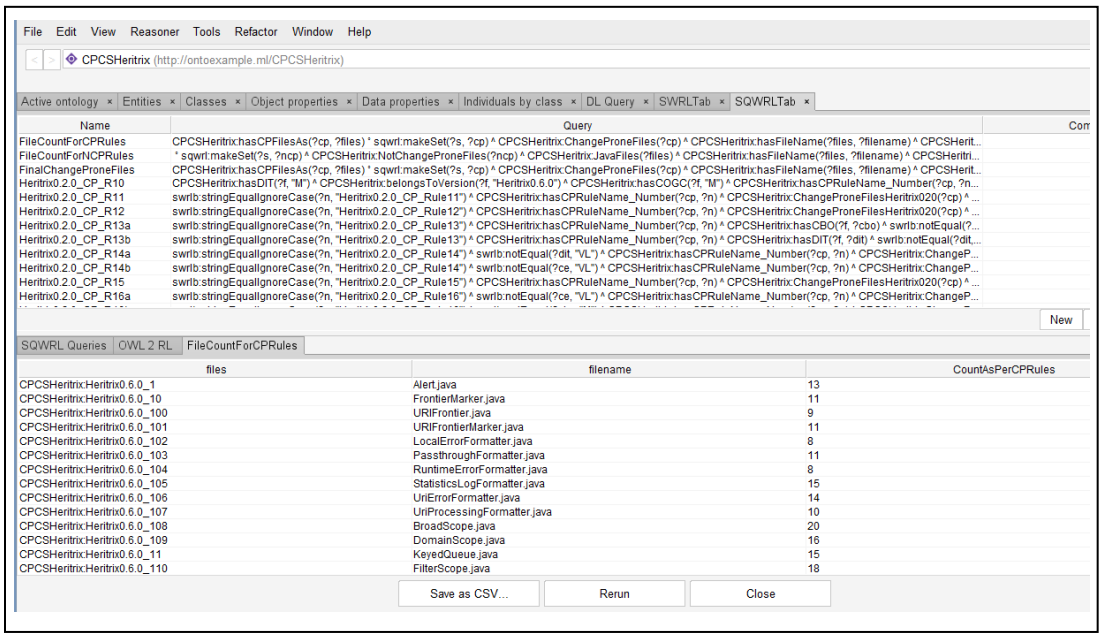

Figure F.2 Screenshot exhibiting the results of the FileCountforCPRules query for Heritrix 0.4.0

- FileCountforNCPRules

As read previously, query *FileCountforNCPRules* displays the file instances that exist corresponding to a version. It also captures their file name and their total count with respect to each of the non-change-prone rules derived from its previously released versions. Figures F.3 are the results of the *FileCountforNCPRules* query. Figure F.3 indicates that 195 rows each are returned for the query as expected.

Figure F.4 indicates the output from the *FileCountforNCPRules* query in which the file instances and their filenames along with their number of times each of the file satisfied a not change-proneness governing rule of Heritrix 0.4.0 and Heritrix 0.2.0.

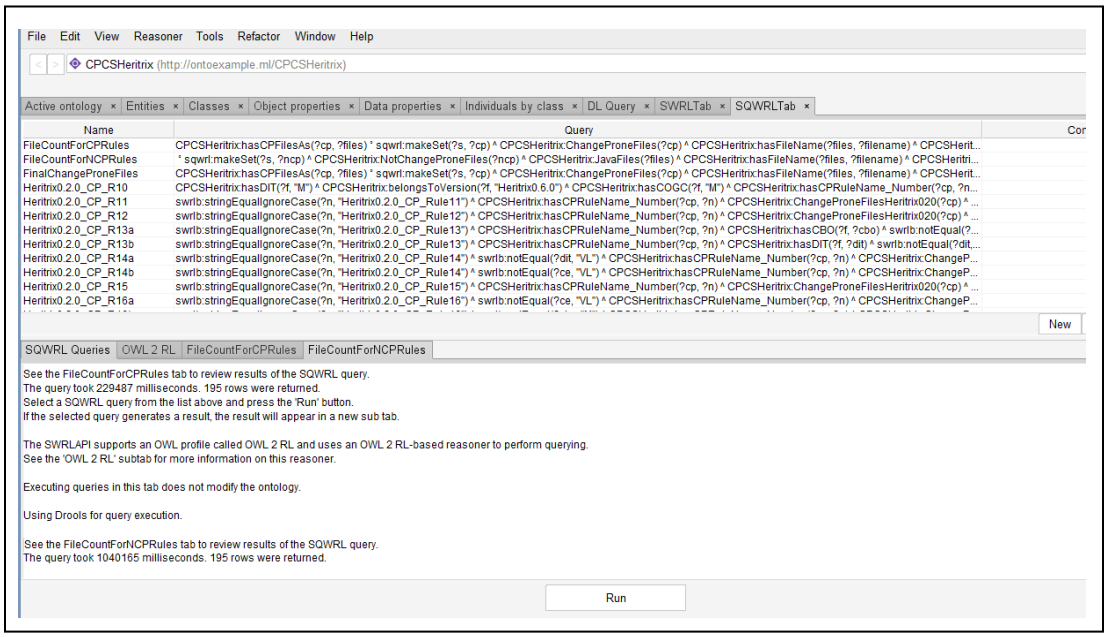

Figure F.3 Screenshot exhibiting the succesful execution of the FileCountforNCPRules query for Heritrix 0.6.0

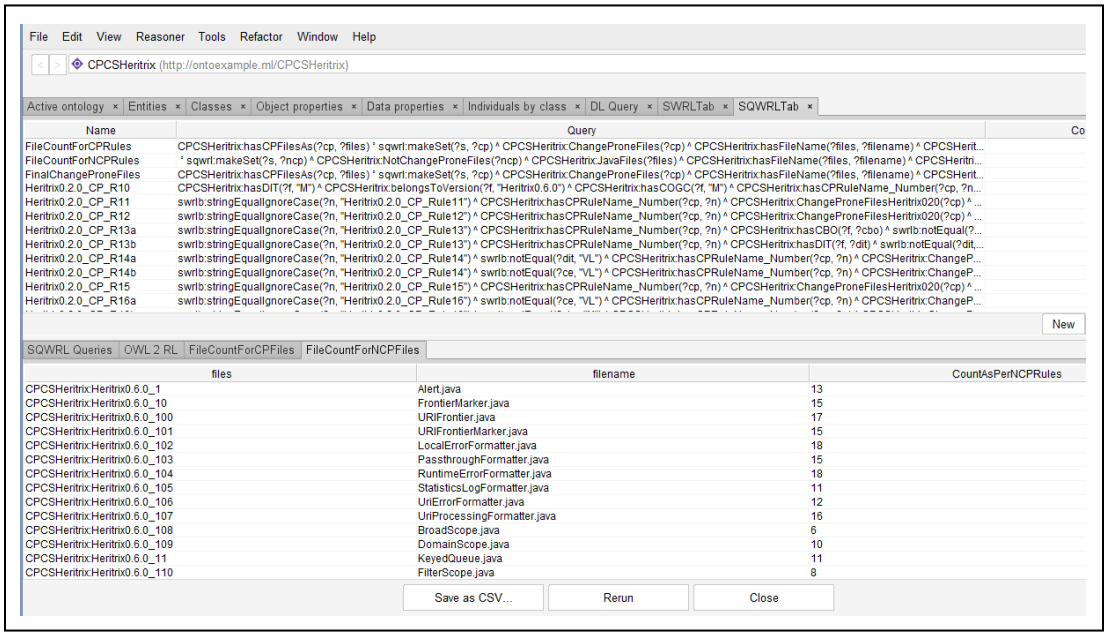

Figure F.4 Screenshot exhibiting the results of the FileCountforNCPRules query for Heritrix 0.6.0

- FinalChangeProneFiles

Figures F.5 and F.6 are the results of the *FinalChangeProneFiles* query. Figure F.5 indicates that the query returns 89 rows, thus exhibiting that 89 Java files are selected to be change-prone from the 195 Java files in Heritrix 0.6.0, according to the rules derived from its previous versions Heritrix 0.4.0 and Heritrix 0.2.0. Figure F.6 displays the output from the query which consists of the 89 selected change-prone file instances and their filenames.

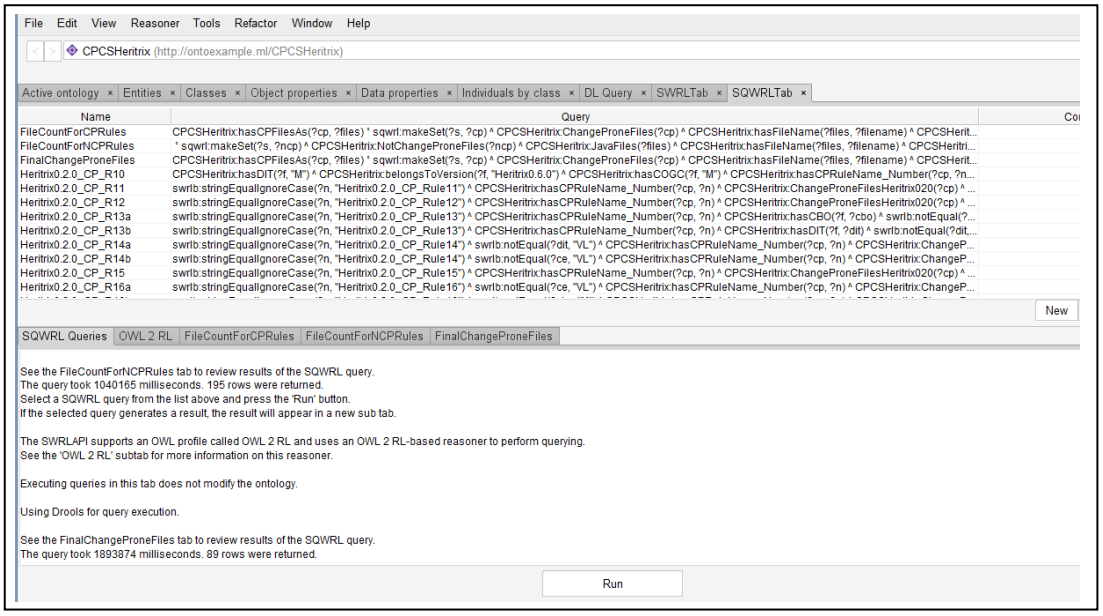

Figure F.5 Screenshot exhibiting the successful execution of the FinalChangeProneFiles query for Heritrix 0.6.0

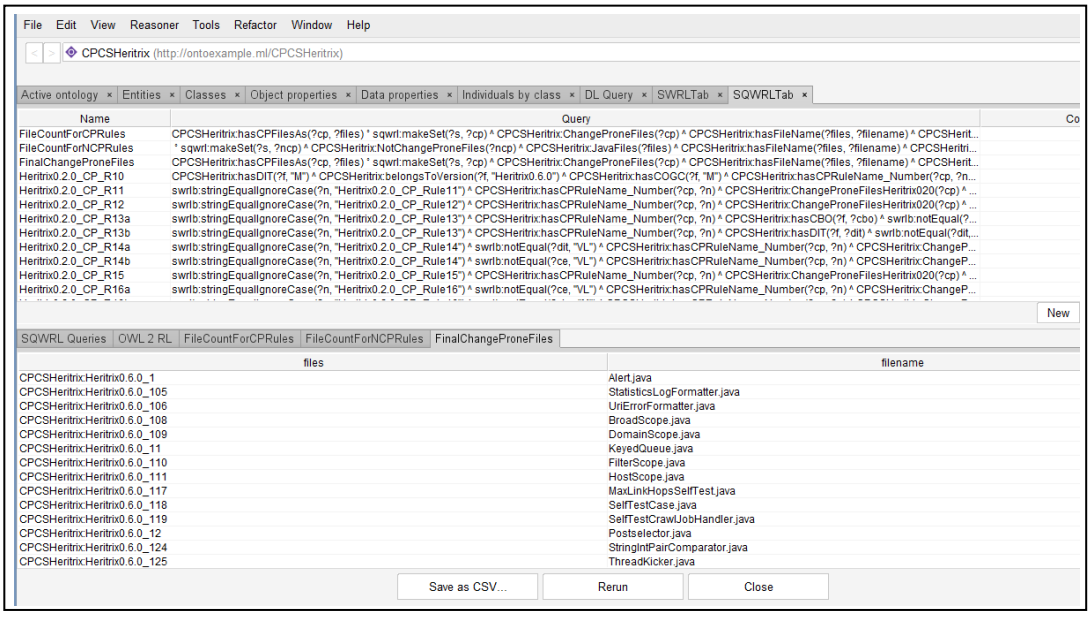

Figure F.6 Screenshot exhibiting the final 89 change-prone files dervied via the FinalChangeProneFiles query for Heritrix 0.6.0