Chapter 1

Introduction

"I discovered that a fresh start is a process. A fresh start is a journey - a journey that requires a plan." - Vivian Jokotade

This chapter presents the key concept used in the thesis and glossary for the entire document. Firstly, we introduce the research background in Section 1.1. Some general concepts and definitions related to software bug prediction (SBP) are presented in Section 1.2. The brief introduction of four studied SBP scenarios is described in Section 1.3, and the major challenges of these methods are in Section 1.4. Further, we express the motivation for this work in Section 1.5, and the goals/objectives of this thesis are given in Section 1.6. Finally, we elaborate on contributions made in this thesis in Section 1.7 and thesis organization in Section 1.8.

1.1 Research Background

1.1.1 The Existence and Harm of Software Bugs

In recent years, with the rapid advancement of information technology, the accumulation of software products has surged. Software has progressively become an

integral part of daily life and is widely utilized across various sectors, including mobile gaming, e-commerce applications, and air traffic control systems. We live in an era where software profoundly influences every aspect of our lives.

In a growing enterprise, the size of a software system keeps increasing to meet the user's requirements. As the size of software increases, its complexity also increases in the same proportion [1]. This increase in complexity causes more bugs to creep into the software system [2]. Such types of bugs may be the reason for the failure of the software or reduced software reliability [3]. The software quality assurance team allocates resources like a software tester to perform code review, software testing, and bug identification to improve the software reliability. Such type of activities cost approximately 80% of the total cost of the software system [4] and consume a lot of time.

Furthermore, software systems inevitably exhibit bugs due to various uncontrollable factors, such as vague requirements in the initial analysis phase, incomplete designs in the system planning phase, non-standard coding practices, and erroneous assessments during the testing phase [5]. In general, software engineers indicated that the bugs introduced during the software analysis and design stages constitute 64% of the total bugs, while the remaining 36% stem from other phases of development.

These bugs can adversely impact user experiences and lead to financial losses. For instance, bugs in mobile games can result in abrupt game interruptions, and bugs in e-commerce applications can potentially deceive consumers. Moreover, these bugs can have severe consequences, as evidenced by the Therac 25 radiotherapy device, where bugs in the input data sequence caused over ten fatalities between 1985 and 1987.

1.1.2 Software Bugs

A software fault is like a physical anomaly in the software that can make it fail to perform the required function. According to the ISO/IEC/IEEE 24765:2010 [6] definition, a software fault is "*a manifestation of an error in software*". Fention and Neill [7] defined software bugs as those artifacts that are an obstacle to fulfill the requirements and specifications of the software. In this thesis, we will be using the terms fault, bug, and defect interchangeably throughout the thesis.

1.1.3 Software Testing

Software testing is a time-consuming and costly task. It requires testing all the software modules to identify which module is non-buggy (clean) and which one is buggy that needs more effort. As users' requirements increase, the software systems' size and complexity increase. Therefore, manual detection of the location of a bug becomes time-consuming in case of a software system failure. The practical software testing approach states that 30 to 50% of the software development cost is spent on software testing [8]. The National Institute of Standards and Technology (NIST) estimates that software bugs cost the US economy $59.5 billion per year [9].

It is crucial to detect bugs early in the software development process to provide a cost-effective and high-quality software product [10]. Identifying bugs at an early stage of the software development life cycle (SDLC) can prevent the spreading of defects from one stage to later stages. In later stages, they may become more complex and expensive to fix [8]. E.g., the cost of a Java-based software project [11] having 100 function points and 10 KLOC are as follows: total effort of 20 staff per month across the five phases is distributed as 2 (10%) for requirement engineering, 4 (20%) for analysis and design, 6 (30%) for coding, 7 (35%) for testing, and finally, 1 (5%) for installation and training. From here, we can see the effort increases as the phases increase.

As the SDLC progresses, finding and fixing hidden bugs in the software becomes more costly. That's why software developers and testers use various testing techniques of software quality assurance (SQA) to analyze the software and detect as many bugs as possible before releasing it. These bugs can be at different granularity, like files, classes, or functions. However, checking all of them takes a lot of time and effort, especially when the software delivery date approaches. It's not practical to inspect every single software module with limited and precious testing resources. In such cases, developers and testers need to figure out which modules are most likely to have bugs and focus their testing efforts on those. This approach helps make testing more efficient. To accomplish this task, researchers have developed **S**oftware **B**ug **P**rediction (**SBP**) models. The SBP model is a classification model that predicts buggy or non-buggy modules of software projects, while the Software bug count prediction (SBCV) model is a regression model that predicts the precise number of bugs in each module.

1.2 Software Bug Prediction

SBP models are widely used to identify bug-prone modules in software systems. SBP model can help to reduce the testing cost, resource allocation, and improve the quality of software [10]. SBP/SBCV models help to predict bug/ bug count in each software system module before starting the software testing phase [12, 13].

SBP system is an attractive area of research in software engineering [14–18]. It enables focused testing of only those artifacts that are error-prone and saves a lot of time and resources. It also aids the software quality assurance (SQA) team who can focus and apply stringent quality measures on those artifacts that are error-prone. The software quality team also analyzes the software quality before the software's final release [19]. SBP models are actively adopted in the software industry to obtain software reliability prediction [20–22].

An SBP model developed using labeled datasets is known as a supervised SBP model. In supervised SBP, the software projects are collected from the software archives. These archives help to collect development artifacts and historical information of software projects. Each project is associated with a version control system and an issue-tracking system. A version control system (VCS) is also known as a source code management tool that helps software teams to manage changes in source code over time. VCS keeps track of every modification to the code in a special database. E.g., a popular VCS tool in use today is called Git[1] (Distributed VCS) and Subversion. Based on the collected data, the complexity of the dataset can be measured. The complexity measurement can be performed at different granularity levels like class, file, function(method), or package(subsystem). Issue or bug tracking systems like Jira and Bugzilla and committed messages in the version control system are used to collect bug information about the software projects. Then, finally, the dataset with the number of instances/modules, software metrics, and labels is generated. Each instance behaves like a class, file, function, or subsystem based on granularity [23].

After generating the labeled datasets, pre-processing techniques (like normalization, scaling, metric selection, and sampling) may be applied to the datasets as per the requirement [24–27]. This labeled dataset is separated into two parts: training datasets (historical datasets) and testing datasets. The training dataset is used to train the SBP model, and the testing dataset is used to test the performance of the model. Various SBP models can be built using Naïve Bayes, decision trees, logistic regression, random forest, etc. [19, 28–31].

If a module is predicted to have a high chance of bugs, it's recommended for closer examination by developers and testers. This way, SBP helps to optimize testing resource allocation and makes software testing more efficient. Additionally, from a software testing classification standpoint, SBP falls under the category of

[1] `https://git-scm.com/`

static testing, as it does not necessitate the execution of the software project's source code.

1.2.1 Types of SBP Based on Labeled Software Modules

SBP models can be grouped into three main types based on whether we have labeled software modules and how many of these labeled modules we use. These types are supervised SBP, semi-supervised SBP, and unsupervised SBP.

1. Supervised SBP is the most common study at present. It requires a sufficient number of labeled software modules to train a classification/regression-based SBP model. This model is then used to predict bugs in unlabeled or new software projects. We have proposed one supervised SBP in Chapter 3.

2. Semi-supervised SBP uses only a small set of labeled modules for training. However, this limited labeled data may not fully represent all the bug patterns, making it harder to build a good SBP model.

3. Unsupervised SBP doesn't use any labeled modules. Instead, it looks at unlabeled software modules and decides if they have bugs or not without any input from labeled modules. This is less common in research and practice.

In unsupervised SBP, we have two different approaches based on how we handle software modules. These are clustering-based and ranking-based unsupervised SBP.

1. Clustering-based unsupervised SBP uses a clustering algorithm to group unlabeled software modules into clusters. Typically, there are two clusters: one for potentially buggy modules and one for non-buggy modules. The bug label is assigned to each cluster based on certain rules, and all modules in a cluster get the same label.

2. Ranking-based unsupervised SBP arranges modules based on a specific feature value. Then, a threshold is set to determine which modules get the bug label. Modules above and below this threshold are given different labels. We have

used ranking-based unsupervised SBP approaches to propose unsupervised SBP models based on the metrics threshold.

Unsupervised bug prediction doesn't require labeled modules to work, but it often doesn't perform as well as supervised SBP. That's why there's less research and interest in this area. To overcome the limitation of lower performance of existing unsupervised SBP, we have proposed three unsupervised SBPs in Chapters 4, 5, 6. In this thesis, we focus on supervised and unsupervised SBP approaches.

1.2.2 Category of SBP Based on Bug Values

SBP can be divided into four main categories based on software module bug values.

1. Classification: This involves categorizing different software modules (like functions, files, and classes) into two groups: buggy or non-buggy.

2. Regression: In this category, the goal is to predict the number of bugs in the software system using various ML and deep learning methods.

3. Mining Association Rule: This category focuses on finding relationships between software metrics, software modules, and bugs using methods like relational association rules or algorithms such as CBA2 [32].

4. Ranking: Here, the aim is to rank different software modules based on the number of bugs they have. Modules with more bugs are prioritized for more testing efforts.

In this thesis, we are mainly working on classification-based SBP (Chapters 3, 4, 6.) and regression-based SBP (Chapter 5) approaches.

1.3 Brief Introduction of studied SBP Scenarios

In this thesis, we study ML-assisted SBP under four SBP scenarios. Specifically, we propose classification-based supervised SBP in Chapter 3 and brief introduction given in Section 1.3.1, classification-based unsupervised SBP in Chapter 4 and brief introduction given in Section 1.3.2, and regression-based unsupervised SBP in Chapter 5 and brief introduction given in Section 1.3.3 using publicly available object-oriented datasets. We have also proposed a classification-based unsupervised SBP on newly created Haskell (functional paradigm) datasets in Chapter 6 and a brief introduction given in Section 1.3.4.

1.3.1 Classification-Based Supervised SBP

SBP is an emerging area of research in software engineering [33, 34]. Many ML techniques, viz. Naive Bayes [35], Support vector machine [36], Random forest [37], Multilayer perceptron [38], Logistic regression [39], Decision tree [40], Evolutionary algorithm [41], Fuzzy inference based models [42], Association rule [43], Dictionary learning [44], and ensemble learning [45] are used to build SBP models. Semi-supervised models [46] and unsupervised models [47, 48] are used in SBP when limited or no labeled datasets (DSs) are available.

Software bug DSs suffer from class imbalance problem [45, 49]. Many researchers tried to mitigate this problem using sampling, ensemble, and cost-sensitive techniques [50–52]. Kamei et al. [53] implemented four sampling methods (ROS, SMOTE, RUS, and one-sided selection) with 4 ML models (Linear discriminant analysis (LDA), Logistic regression (LR), Neural network and Classification tree). The authors concluded that LDA and LR showed an improvement in f-measure by 0.078 to 0.224. Menardi et al. [54] stated that random oversampling example (ROSE) outperforms SMOTE in case of extreme levels of imbalance and small sample sizes.

Many research works are available in the literature that focus on the ensemble techniques used for SBP [45, 55]. Recently, Matloob et al. [56] presented a systematic literature review for SBP using ensemble techniques and concluded that in most cases, ensemble techniques outperformed base learners. They observed that bagging, boosting, and random forest are the most frequently employed ensemble techniques, and stacking, voting, and Extra trees are the less frequently employed ensemble techniques by the researchers. The most common performance parameters are accuracy, FM, precision, recall, MCC, and Area Under Curve (AUC). Most of the research works used the PROMISE and NASA MDP (Metric data program) repository DSs. Others also used the software bug DSs collected from SOFTLAB, AEEEM, ReLink, Apache, Eclipse, etc. [56]. The motivation for the classification-based ensemble model is given in Section 1.5.1.

1.3.2 Classification-Based Unsupervised SBP

The typical supervised methods have the limitation that they cannot be used for new software projects because they require labeled instances. Labeling of instances of new software projects is difficult due to the unavailability of sufficient historical information or bug information [25, 57]. When training and testing are performed on the same projects, it is known as within-project fault prediction (WPFP). When a model is trained on a source project and then is tested on any other target project, then it is called cross-project fault prediction (CPFP) [25, 27, 31, 58–61]. Zimmermann et al. [57] concluded that only 21 models proved to be useful out of 622 cross-project prediction models in an experimental study. However, Watanabe et al. [60] performed inter-language prediction experiments, which means training on project A and testing on other project B where projects A and B are written in different programming languages. They have proposed metrics compensation (To adjust the test data) formula. The compensated values become similar to the learning data, and hence, this approach has improved the result of the CPFP but could not perform better than WPFP. Turhan et al. [27] proposed the nearest neighbor

(NN) filter that selects the ten closest instances for each target instance to maintain the same distribution. However, its prediction performance is not better than that of WPFP.

Ma et al. [31], and Nam et al. [25] have proposed transfer learning methods to overcome CPFP. Ma et al. [31] gave more weights to the source instances similar to the target instances when building the Naïve Bayes model. It performs better than the NN filter. Nam et al. [25] proposed state-of-the-art transfer component analysis (TCA+) techniques to build SBP model for CPFP. They concluded that the performance of TCA+ is comparable to the WPFP.

Zhang et al. [61] built a Universal Fault Prediction (UFP) model. In UFP, the context-aware rank transformation method was implemented to scale the metric values ranging from 1 to 10 across all projects. In this way, UFP was applied to 1398 software projects collected from SourceForge and Google Code, and results were comparable with that of WPFP. So, CPFP suffers from the same issue, that construction of the CPFP model requires sufficient software projects with historical information and similar distribution of source and target dataset. However, this problem has been dealt with by trying to make the distribution of the source and target dataset similar by transferring metric values [60, 62], selecting similar instances [63] and using transfer learning [25, 31]. But, still, these approaches are not practical and suffer from dataset shift problems [64]. Nam et al. [62] proposed CPFP using a heterogeneous metric known as Heterogeneous fault prediction (HFP), which could overcome the results of WPFP, CPFP, and UFP. But, all the aforementioned approaches, viz. WPFP, CPFP, and UFP need labeled datasets, so to overcome this problem, various research studies have been conducted to build SBP models on the unlabelled datasets [65–69]. The motivation for the classification-based unsupervised SBP model is given in Section 1.5.2.

1.3.3 Regression-Based Unsupervised SBP

Researchers have extensively studied classification-based SBP models over the past three decades [8, 56, 70]. However, in recent years, there has been growing interest among researchers to develop regression-based Software Bug Count Vector (SBCV) prediction models [12, 71–73]. SBCV prediction models offer superior advantages in terms of optimizing limited testing resource allocation, reliability evaluation, and maintenance effort estimation compared to SBP models [1, 74]. SBCV prediction models have been found to be more effective than traditional SBP models [71]. However, predicting the precise number of bugs in each software module is a challenging task. All the existing SBCV prediction models are developed using labeled datasets. It was observed that the majority of authors [1, 71, 75–78] utilized standard ML algorithms such as LM, SVR, DTR, KNN, NBR, GA, BRR, and MLP to compare the performance of their proposed methods. As far as our knowledge extends, no previous research has introduced an unsupervised approach for predicting the number of bugs in each module of a software system. The motivation for the regression-based unsupervised SBCV prediction model is given in Section 1.5.3.

1.3.4 Classification-Based Unsupervised SBP in Functional Paradigm (FP)

Nowadays, a wide range of modern programming languages like Python or Java support functional operations [79]. Haskell is a general-purpose, strongly typed, and purely functional language [80]. Functional programming (FP) performs mathematical operations without changing state and mutating data. Khanfor and Yang [81] have advocated that programs written in FP are more modular and reusable than those written in OO programming. FP also supports abstraction, encapsulation, inheritance, and polymorphism. The Haskell code is easy to understand for the developer and the user due to the written code being shorter and more independent [81].

FP is now adopted as a new generation of programming technologies. It is also gaining popularity and influence [82]. In the real world, FP languages are also progressively adopted by industry[2], e.g., Facebook, WhatsApp, Twitter, LinkedIn, Google, etc. The progressively increasing popularity of the FP has sparked new discourse between FP and OO programming communities [83]. Based on STREW-J (Software Testing and Reliability Early Warning for JAVA) metric suit, Sherriff et al. [84] proposed the STREW-H (Software Testing and Reliability Early Warning for Haskell) SM. They present a feasibility study to predict defect density using in-process metrics by analyzing the source code on seven versions of the Glasgow Haskell Compiler[3] [84]. Later, they designed the defect density prediction model and showed that STREW-H metrics were capable of predicting the defect density [85].

To design the SDP system for an unlabeled dataset, many researchers have used clustering approaches to cluster the modules [86]. These clusters were then labeled by experienced software engineering experts [87] having experience of 15 years or more. But, firstly, it is difficult to find software engineers with that kind of experience, and secondly, manual labeling is costly and cumbersome. Zhong et al. [87] provide an approach based on manual labeling, which is both expensive and subjective. Therefore, our proposed work focuses on providing a fully automated SDP system for a functional paradigm without the need for software experts. The motivation for the classification-based unsupervised SBP model for Haskell is given in Section 1.5.4

1.4 Major Challenges Faced in SBP Scenario

Every SBP domain has many challenges. These are common across SBP/SBCV domains, such as obstacles of high dimensionality, overfitted results, and class imbalance problems. The main difficulties to be solved under different SBP scenarios are described as follows:

[2]https://wiki.haskell.org/Haskell_in_industry
[3]http://www.haskell.org/ghc/

The software metrics of the bug dataset typically represent the basic characteristics of the source code, such as complexity, coupling, cohesion, and changing characteristics [88]. These software metrics usually cannot demonstrate the intrinsic structure details concealed behind the datasets; that is, they may make it complex to distinguish the modules of different labels linearly. This will remarkably reduce the results of the SBP model. In addition, the class imbalance problem in a dataset refers to the condition where instances of the minority class (buggy modules) are much less than those of the majority class (non-buggy modules). In that case, the SBP may not perform well and can lead to biased results towards the majority class [89]. The variation in the distribution of data points leads to the poor performance of some ML techniques, mainly for the minority class. Thus, how to learn suitable metric representation to characterize the bug data and reduce the negative impacts of class imbalance for performance improvement are the main challenges in SBP scenarios.

Further, supervised SBP models have the limitation that they require labeled bug datasets. Labeling of datasets is a time-consuming process. So, collecting the bug datasets with the utmost care and consistency (100% accuracy) is also challenging in the SBP scenario [47]. Further, tuning the hyperparameters of the ML models for better performance is a challenging task [90]. The performance of individual ML models has reached its saturation point, and it has become difficult to improve the performance of SBP further using the existing techniques [70]. Additionally, selecting the effective software metrics (SMs) for the SBP model is also a challenging task [91, 92].

Software metrics (SMs) are skewed towards the positive [47]. Hence, SMs are not always characterized by descriptive statistics. So, the implementation and usage of the SBP model based on SMs are also affected by the distribution of SMs. Overfitting is also a prominent challenge in such predictive models. SBP model predicts only whether the specific module is buggy or non-buggy but does not predict the number

of bugs within a particular module. So, developing an SBCV prediction model that predicts bug count in each module is a potential challenge.

Most of the existing SBP models used JAVA, C, and C++ software projects. The most publicly available datasets are in OOP. No bug datasets are available for the functional paradigm. Therefore, a major challenge is creating novel functional parading datasets and proposing a novel SBP model for FP.

1.5 Motivation for the Thesis

In the motivation section, we reported the reasons for proposing the four SBP models into different categories. The primary goal of these SBP models is to predict modules with bugs or bug count, making testing easier and helping the SQA/development team leader to allocate testing resources effectively. These are a few common motivations that apply to all SBP domains. Motivation for classification-based supervised SBP is given in Section 1.5.1, classification-based unsupervised SBP is given in Section 1.5.2, and regression-based unsupervised SBP is given in Section 1.5.3. The motivation for classification-based unsupervised SBP for newly created Haskell (functional paradigm) datasets is given in Section 1.5.4.

1.5.1 Motivation for Classification-Based Supervised SBP

In recent years, many SBP models have been developed using various ML techniques [10,93]. A few of them use Naive Bayes, Support vector machines, K-nearest neighbors, Decision trees, Neural networks, etc. The average performance accuracy of these algorithms is almost 80% to 85% [10,94]. However, these algorithms' performance is varying across different datasets (DSs) [2,95]. Earlier researches [56] have revealed that standalone ML techniques for SBP have reached the peak of their performance. So, there emerged a need to combine the results of multiple techniques in model building process to improve the performance [96,97]. To address this emerging need, Menzies et al. [98] enriched the training dataset (DS) with weights assigned

by human experts with business knowledge to improve the quality of the DS. They further customized the SBP model as per the specific business goals [96]. The above performance-enhancing techniques helped to cross the performance bottleneck significantly.

Also, individual ML models are limited in identifying a certain bug under specific circumstances [99]. For this reason, hybrid learning techniques [56] are used to combine the strength of multiple classifiers that provide higher bug identification in the DS. In the last decades, many researchers have given empirical evidence to show that hybrid/ensemble techniques provide higher performance than individual classifiers [55, 100–102].

Broadly, ensemble techniques are divided into homogeneous (Bagging, Boosting) and heterogeneous (Voting, Averaging, Stacking) techniques [77]. Few studies analyzed the stacking, voting, bagging and boosting techniques to develop the SBP models [103–105]. However, these models are validated on limited datasets (DSs) and still have the scope to enhance the performance.

Recently, Kumar et al. [55], and Rathore et al. [106], presented an empirical study about the improved ensemble techniques. Matloob et al. [56] presented a systematic literature review on SBP using ensemble learning. From this literature review, we found that there is a scope for improvement in majority voting ensemble technique. We analyzed and found that there is a need for a weight evaluation scheme to decide the weight of base classifiers (BCs) in the Simple Majority Voting technique.

Further, the existing SBP techniques are suffering from two significant obstacles. First is class imbalance problem [49] and second is the poor performance of ML models [10,94]. No WMV SBP models have evaluated the individual weights of BCs in ensemble techniques. The contribution related to WMV is given in Section 1.7.1.

1.5.2 Motivation for Classification-Based Unsupervised SBP

Many researchers have already proposed several software metrics and SBP models to predict software bug [19, 28, 107–111]. Software metrics are typically divided into two parts: code metrics and process metrics [112]. Code metrics give the internal complexity of the source code, and process metrics define the complexity of developing the software [24].

Various SBP models are proposed based on supervised learning [24, 57, 113]. In the past few decades, many SBP models have been proposed to enhance the reliability of software quality. Most of the ML algorithms are applied to the labeled datasets. But, practically collecting faulty data and labeling the software dataset is a tedious task. Such SBP models are challenging to apply to new software projects or projects with limited historical software archives. So, supervised SBP models have a limitation that they cannot be applied to a new dataset (dataset without labels).

To overcome the aforementioned limitations, existing researchers have focussed on providing various approaches to enable SBP on new datasets or datasets with limited historical information. Many researchers have proposed cross-project fault prediction (CPFP) models. In CPFP, software project history or fault information was used to train the model and test on new datasets [25, 27, 31, 58, 60, 113, 114]. Unfortunately, CPFP approaches have a limitation that CPFP is only feasible for software projects that have exactly the same metric set. Finding other software projects with exactly the same metric set can be challenging. Publicly available fault datasets that are widely used in SBP literature usually have heterogeneous metric sets.

To overcome the above limitation, researchers have proposed heterogeneous defect prediction (HDP) models in which datasets having heterogenous metric sets (in practice, different metrics have different distributions and physical meanings) are

converted to achieve similar distribution [61, 62, 64]. Still, these approaches of making the dataset distribution similar may not be practical every time for SBP [25, 27]. An expert-based SBP model was proposed by Zhang et al. [87], in which a software expert labeled the dataset after clustering. A threshold-based SBP model was proposed by Catal et al. [115], in which the datasets were labeled based on the specific threshold of the software metrics. However, the approaches proposed by Zhang et al. [87] and Catal et al. [115] were not affected by different distributions of the dataset, but they do require human effort and intervention to decide the threshold.

To mitigate this requirement of a human intervention, Nam et al. [116] proposed the CLA/CLAMI methods to design an automated SBP model on unlabelled datasets. Another automatic change-prone class SBP model on unlabelled datasets was proposed by Yan et al. [117]. The goal of the proposed work is to design an automated SBP model using unsupervised learning methods. This method does not require any human effort. Based on such automated SBP models, we have proposed an unsupervised SBP model. The contribution related to this classification-based unsupervised SBP is given in Section 1.7.2.

1.5.3 Motivation for Regression-Based Unsupervised SBP Prediction

In the modern software world, programs are getting bigger, more intricate, and more sophisticated. To minimize testing effort, it's not enough to just predict whether a module has bugs or not. Predicting the number of bugs in each module gives us more useful information and helps reduce the amount of testing needed. If a team leader knows the total bugs in the entire project, then this information can efficiently help to allocate testing resources. Efficient use of testing resources helps to reduce the overall cost and time of developing software.

So, to minimize this cost, time and enhance software reliability, SBP plays a vital role in predicting buggy software modules [56, 118]. Precise predictions aid

software testers in optimizing testing resources by targeting predicted buggy modules [1, 119]. However, relying solely on a classification-based SBP model for predicting bug-proneness falls short in real-world scenarios, as illustrated in Fig. 1.1. Fig. 1.1 shows a software project that has 500 classes (modules). It is required to be tested by a software tester. Software testers aim not only to identify which software classes should be inspected first but also to assess the reliability and maintenance effort of each class. To achieve this, they can utilize historical data to build either a classification-based SBP model or a regression-based SBP model (SBCV). Subsequently, the two trained models can be employed to predict either the bug-proneness or the bug counts in the modules or classes [71].

Let the SBP model predict that 35% of classes are buggy (e.g., C3, C498, C500, etc.), while testers can only examine a limited percentage (e.g., 25%) of classes due to resource constraints. The challenge lies in choosing which classes among the predicted buggy ones to test based on the prediction result of the SBP model. However, top-indexed 25% classes can be tested to identify more bugs using the prediction results of the SBCV prediction model, which indexes the 500 classes based on the predicted bug counts. Further, software unreliability and maintenance effort are directly proportional to the bug counts [74,120]. According to SBP model, the software unreliability and maintenance effort for C3, C498, and C500 are deemed equal, as these are predicted to be buggy. Conversely, the SBCV prediction model predicts that C3, C498, and C500 contain 5, 3, and 4 bugs, respectively, indicating that 498 is more reliable and requires less maintenance effort than C3 and C500 (Fig. 1.1). In essence, the SBP model fails to differentiate and rank the modules on the basis of number of bugs in each module.

In summary, SBCV prediction models offer superior advantages in terms of optimizing limited testing resource allocation, reliability evaluation, and maintenance effort estimation compared to SBP models [1, 74].

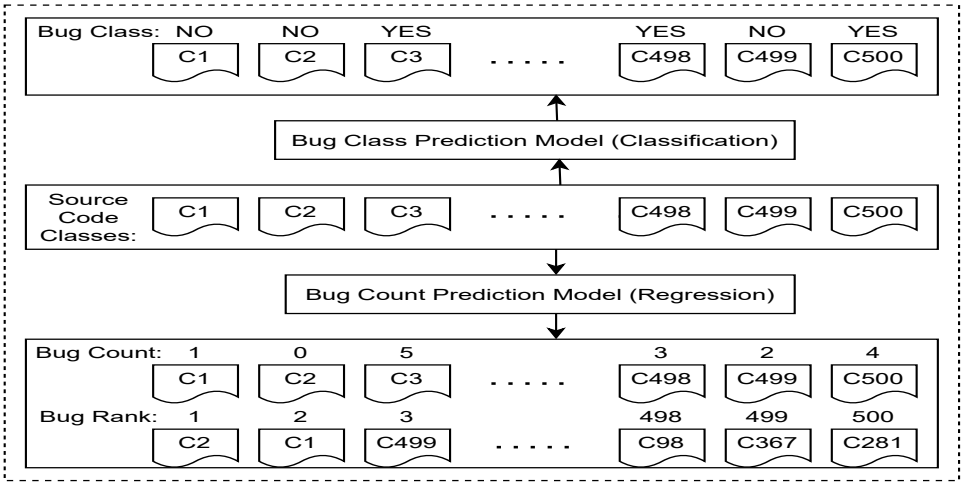

FIGURE 1.1: Bug class (SBP) VS. bug count vector prediction (SBCV)

Precisely predicting the bug count in a new software release can facilitate software managers in intelligent decision-making. This includes determining the appropriate timing for the software release based on the number of potential bugs. Apriori knowledge of the number of possible bugs enables the users to take timely actions to address any losses or issues that might arise due to potential software bugs [74,121]. As a result, many SBP models have been proposed [56, 122]. According to Wahono [123], 77.46% of the models investigated are related to SBP, while only 14.08% pertain to SBCV prediction models.

Various SBCV prediction models based on labeled datasets have been proposed [1, 71, 77, 118]. Various ML algorithms like linear regression [78], multi-layer perception [77], decision tree regression [75], k-nearest neighbor [75], support vector regression [74] deep neural network [1], ensemble regression [77], etc. have been used to build the SBCV prediction models. These SBCV prediction models built using labeled datasets are not much effective for new software projects due to different distributions of training and testing datasets. Aforesaid algorithms need labeled datasets to train the ML models. However, practically collecting the software dataset and labeling them with bug count information is difficult and requires a lot of effort and time. SBCV prediction on a new software project (unlabeled) is an interesting problem in software engineering [34, 124]. As far as our knowledge extends, no

previous research has introduced an unsupervised approach for predicting the number of bugs in each module of a software system. Therefore, we have proposed a regression-based unsupervised SBCV prediction model based on unlabeled datasets. The contribution related to regression-based unsupervised SBP is given in Section 1.7.3.

1.5.4 Motivation for Classification-Based Unsupervised SBP in FP

Much research has been done in the field of SBP in the OOP paradigm. However, few empirical studies have been conducted to analyze the impact of functional programming languages on software quality (in terms of bugs) [125].

Any research on debugging and tracing techniques is costly and time-consuming because of the requirement of runtime monitoring of the program. It also does not guarantee the execution of each section of code, so it is useful to focus on debugging and testing the most crucial area of the program only. Unfortunately, it is difficult to identify such critical regions of large software. Research conducted in the OOP paradigm has proved that SMs provide much information about these crucial areas. Thus, SMs are an essential factor in auditing the internal quality of software projects [126]. SMs are used as a vital tool to measure the defect proneness of components, their reusability, and their maintenance.

Detecting and fixing the defects in software after delivering it to the customer is a complicated and expensive task. Instead, early detection of defect proneness can reduce the efforts involved in testing, diagnostics, and final resolution of the defects [127, 128]. The tools that enable early detection of defects can reduce the cost incurred during software development and maintenance stages [90,129]. Manjula et al. [130] state that most rework can be avoided by early detection of defects, an inspection of code, and peer review. Peters and Pedrycz [131] advocate that 30 to 50% of the software development cost comes from testing.

To the best of our knowledge, we could not find any research where SMs have been used to develop an SDP system in the FP. In this research, we have focused on detecting the defect proneness in Haskell packages. Since no research dataset was available in Haskell, so we developed datasets of our own from Haskell packages. These Haskell datasets were prepared by extracting the six software metric values from the Haskell packages. The functions that contain high metrics values require much refactoring, reviewing, and testing [132]. Programmers can use SM evaluation to get an idea of those functions that require improvement to maintain the quality of the code. Metrics can also be considered analogous to compiler warnings; for example, metrics can provide a hint that a specific code may require an inspection.

This proposed research can help to predict defects in Haskell packages and can be utilized by the software industry to automate the SDP system. The contribution related to classification-based unsupervised SBP in FP is given in Section 1.7.4.

1.6 Thesis Objective

The main goal of this thesis is to propose novel supervised and unsupervised SBP/SBCV prediction models by using different machine-learning techniques. This thesis mainly focuses on classification and regression analysis-based SBP approaches using supervised and unsupervised models. However, as already discussed in Section 1.4, there are several concerns with the existing techniques. Hence, we attempted to find answers to the following common research questions (RQs) as part of our research:

1. RQ-1: Is the prediction performance of the proposed supervised approach comparable to that of standard supervised ML models?

2. RQ-2: Is the performance of unsupervised approaches comparable to the performance obtained using the standard supervised / unsupervised learning approaches?

3. RQ-3: Are the results obtained by proposed techniques comparable to those obtained by best-performing SOTA advanced techniques?

4. RQ-4: Is the performance of proposed approaches without sampling techniques comparable to that of proposed approaches with sampling techniques?

5. RQ-5: Is the prediction performance of the proposed regression-based SBP comparable to that of standard supervised regression ML models?

6. RQ-6: Are the extracted SMs capable of predicting defect proneness in the functions of Haskell?

We have also added similar/common RQs in each chapter to answer/justify such RQs in each chapter of our thesis. To address these RQs, we have provided detailed explanations/justifications for them in each chapter. Answers to these common research questions are given in Chapter 7.

1.7 Contributions to the Thesis

This thesis is committed to developing effective SBP and SBCV prediction models using supervised and unsupervised approaches. We're accomplishing this to answer the RQs mentioned in Section 1.6. To tackle the problem of predicting software bugs, we're developing SBP models one after another by overcoming some limitations of all the previous SBP models.

First, we're exploring the most common approach, classification-based supervised SBP, using the WMV ensemble technique in Chapter 3 and detail contribution given in Section 1.7.1. Here, we examine that creating and collecting the label bug datasets is a tedious and costly process. So, to overcome the problem of labeled datasets, we're proposing a classification-based unsupervised SBP model using software metrics threshold derivation in Chapter 4 and detail contribution given in Section 1.7.2. Now, we found that only predicting buggy and non-buggy modules is not sufficient to rank the software module or not provide more information, like

bug count in each module. Hence, we are proposing a regression-based unsupervised approach for predicting bug count vectors using metric selection and metric threshold calculation in Chapter 5 and detail contribution given in Section 1.7.3. At last, we investigate that the above three mentioned SBP models are developed using the public datasets only that are related to OOP; therefore, we explore new paradigm FP and create novel FP datasets using Haskell packages and proposing a classification-based unsupervised SBP using UMV technique in Chapter 6, and detail contribution given in Section 1.7.4.

The main focus of this thesis involves developing, testing, and evaluating SBP approaches. We're using new and innovative approaches to address existing issues. We've developed new methods for classification-based SBP and regression-based SBP in different software projects collected from different dataset groups. We've also created four novel datasets from Haskell packages and developed an ensemble approach UMV to predict software bugs in the Haskell projects. We're also addressing challenges like class imbalance, overfitting, and model stability. Additionally, we're considering factors like model complexity, prediction performance, statistical tests, and more.

1.7.1 WMV: Classification-Based Supervised SBP

The main goal of this contribution is to address the dataset imbalance problem and develop an effective SBP model using a supervised approach. An SBP system built using ML on the labeled dataset is known as supervised learning. The motivation behind this chapter is discussed in Section 1.5.1.

An accurate prediction of bugs in software projects can help to improve the software project's quality. A simple majority voting ensemble (SMV) is an effective technique for bug prediction. SMV combines the results of base classifiers (BCs) based on the majority voting of class. All the standalone BCs do not perform equally well, yet all BCs in SMV are given equal weights. Therefore, in order to improve

the performance of SMV, BCs should be assigned different weights. So, here, we propose a novel reward-based weighted majority voting (WMV) ensemble technique to build a bug prediction model. In WMV, each BC performance in the ensemble is evaluated, and then a reward-based mechanism is used to calculate the weights of each classifier. When a base classifier (BC) predicts the correct class of an instance, then a reward is provided, but no punishment is given for the wrong prediction. A BC will get a higher weight in an ensemble that predicts more instances correctly.

To show the effectiveness of the proposed WMV technique over individual classifiers, SMV technique, and other state-of-the-art (SOTA) techniques. Five individual algorithms viz. Naive Bayes (NB), Support vector machine (SVM), K-nearest neighbor (KNN), Random forest (RF), and C5.0, are chosen as the BCs in WMV. The experiments are performed on 28 SBP DSs collected from five groups. Accuracy, F-measure, and MCC are considered for evaluating the performance of the proposed WMV technique. The statistical comparison of the models is done using the Wilcoxon signed-rank test and the Nemenyi test. The results of the proposed WMV technique are also compared with the recently published works. Our experimental study can help software practitioners in developing effective SBP models. The main contributions of this work are listed as follows:

1. We have proposed a novel WMV ensemble technique for SBP. The proposed approach has not been explored before to the best of our knowledge.

2. We have implemented our technique on 28 software projects collected from five repositories to establish the feasibility and usefulness of WMV. We have compared the results based on statistical evaluation.

3. To show the effectiveness of the proposed WMV technique, the performance of WMV is compared with the other recent papers published in reputed journals in the last five years (2017-2022).

Our DSs are imbalanced; therefore, we have applied three sampling techniques: ROSE (Random over-sampling examples) [54, 133], SMOTE (Synthetic minority over-sampling technique) [134], and RUS (Random under-sampling) [133] to balance the DSs. We, then, applied WMV on the datasets balanced by these three techniques independently and compared the results to find which sampling technique was giving the best results. The detailed description is given in Chapter 3.

1.7.2 TCL/TCLP: Classification-Based Unsupervised SBP

The basic inside of this contribution is to design an automated SBP using unsupervised learning methods. This method does not require any human effort. This chapter also addresses the dataset imbalance problem. The motivation behind this chapter is discussed in Section 1.5.2.

SBP models help the software quality assurance team (SQA) to manage the resources optimally during software maintenance. Most SBP approaches, recently proposed, are helpful only on labeled datasets. Recently, several threshold-based SBP approaches have been proposed. However, these approaches do not incorporate the distribution of software metrics for metric threshold derivation and hence demonstrate poor performance. To fill this gap, we have developed an automated SBP approach, namely TCL/TCLP (Threshold Clustering Labelling/Threshold Clustering Labelling Plus), that doesn't need a labeled dataset. It can identify the buggy and non-buggy artifacts on unlabelled datasets by self-learning. Our proposed approach is an extension of the state of art technique known as CLAMI [116]. Unlike CLAMI, we have derived the metrics threshold using logarithmic transformation. Thereafter, we labeled the instances into binary classes (buggy/non-buggy) using the metrics threshold values. TCLP extends this approach one step further by performing SBP using a random forest algorithm. The empirical evaluation of the proposed approach on 28 datasets (with different numbers of metrics and granularity) collected from five software groups shows that the proposed unsupervised method obtains significantly better results than the state-of-the-art methods. The proposed approach

impressively enhances the performance of CLAMI in terms of accuracy, f-measure, and MCC. The contributions of this research work are as follows;

1. We have proposed a novel approach, TCL and TCLP, to design an automated SBP system on unlabelled datasets.

2. We have presented an empirical study to demonstrate the significance of the proposed model and have compared our results with five supervised learning algorithms, four unsupervised learning algorithms, and three threshold-based methods. The presented results have been highlighted in terms of performance measures (accuracy, f-measure, MCC) and statistical tests (Wilcoxon signed rank test and Nemenyi test).

Our datasets are imbalanced, so we have applied the SMOTE (Synthetic Minority Oversampling Technique) [134] to balance the datasets and have compared the results of our proposed methods on balanced and imbalanced datasets. SMOTE is an effective and influential oversampling technique to increase the instances of minority classes in order to balance the datasets. The datasets are balanced by applying an interpolation among the neighboring minority class instances. The detailed description is presented in Chapter 4.

1.7.3 MTB/MTBP: Regression-Based Unsupervised SBP

The main goal of this contribution is to design an automated Software Bug Count Vector (SBCV) prediction technique using unsupervised learning. This method also utilizes the software metric selection technique to make the proposed approach effective. The motivation behind this chapter is discussed in Section 1.5.3.

Software Bug Count Vector (SBCV) prediction technique is a regression model that aims to predict the precise number of bugs in each module of a software system. In contrast, an SBP model focuses on predicting whether or not a module is buggy. Predicting the exact number of bugs in each module brings efficiency in software

test resource allocation, maintenance, and release time. Many researchers have conducted empirical studies to predict SBCV using regression algorithms on labeled datasets. However, accurately collecting and labeling buggy data poses multiple challenges. To address the limitation of labeled data and the absence of an unsupervised regression model for SBCV prediction, we propose a novel unsupervised regression model. The key idea behind our approach is to predict SBCV on unlabeled datasets by deriving independent thresholds for each software metric. The average performance of our proposed technique over 22 datasets surpasses the majority of state-of-the-art regression techniques (8 standard supervised algorithms) in terms of mean absolute error (0.45), mean relative error (0.20), and Pred(l)_Error (0.29). The results of statistical tests, including the Wilcoxon signed-rank test, CohenD test, and Nemenyi test, demonstrate the significance of our proposed technique. The key contributions are listed as follows:

1. We have proposed an MTB/MTBP model that stands for **M**etric selection, **T**hreshold derivation, **B**ug count vector generation, and **P**lus. MTB is used to provide bug count value to each module based on software metric threshold. Then, we have utilized the MTB model to propose the MTBP model. Fig. 5.1 presents the step-by-step process to implement the MTB/MTBP model. This is the first unsupervised SBCV prediction model to the best of our knowledge.

2. We have implemented the proposed approach MTB/MTBP on 22 datasets collected from two repositories (MORPH and AEEEM). We have compared the results with the 8 supervised ML algorithms and applied three statistical tests to evaluate the significant performance of the models.

The detailed description of this contribution is presented in Chapter 5.

1.7.4 UMV: Classification-Based Unsupervised SBP in FP

The main focus of this contribution is to design an automated software metric threshold-based ensemble approach to predict defect proneness in the FP for those

cases where no labeled dataset is available. Four novel Haskell datasets were created from Haskell packages using different software metric extraction tools. The motivation behind this chapter is discussed in Section 1.5.4.

Numerous SDP systems have been explored to predict software defects within the OOP. In contrast, there are no experimental/empirical investigations to predict defects in Haskell, which is a functional programming (FP) language. Our first objective is to create a novel software bug dataset using Haskell packages. We demonstrate how source code metrics values are extracted/derived solely from a given function in Haskell code snippets. Secondly, we aim to build a novel SDP system that can predict defects in each function of Haskell using software metrics (SMs) threshold calculation. We analyzed the SMs statistically and found that these SMs are right-skewed. So, we reduced the skewness using three transformation techniques and then calculated the corresponding SM threshold. Then, we proposed a novel unsupervised majority voting (UMV) ensemble method for SDP in Haskell and performed a comparative analysis. Our experimental results demonstrate that the proposed SDP method (UMV) performs well on Haskell. The experimental results of UMV on four Haskell package datasets are validated using accuracy, f-measure (FM), Mathew's correlation coefficient (MCC), average silhouette width coefficient, and defect-wise boxplot of SM values. The average performance of UMV is superior to all the unsupervised algorithms in terms of accuracy (93.43%), FM (0.965), and MCC (0.733). Various factors can influence the performance of an SDP model, such as statistical characteristics of dataset. However, in case of unlabelled datasets, our SMs-based SDP model UMV can be used in industry to predict software reliability. The main contributions are listed as follows:

1. We have created four novel Haskell datasets with six software metrics from Haskell packages.

2. We have proposed a novel SDP model based on the unsupervised majority voting (UMV) ensemble technique.

3. Presented empirical evidence to support the utility of the proposed approach on Haskell. The outcomes of our proposed UMV technique are rigorously compared with five established supervised ML models, four unsupervised ML models, and six threshold-based unsupervised SDP models.

This proposed research can help predict defects in Haskell packages and can be utilized by the software industry to automate the SDP system. The obtained results enable us to conclude that SMs in the FP are pretty valuable for predicting defects in functions using SM thresholds and ensemble clustering approaches. The detailed description of this contribution is presented in Chapter 6.

1.8 Thesis Organization

The overall thesis is organized into eight chapters. Fig. 1.2 illustrates the overview and interconnection between each chapter. Fig. 1.3 shows a concept map of the thesis to present the cohesive nature of the thesis.

Chapter 1 demonstrates a brief introduction to SBP and its different categories. This chapter also presents limitations, motivation, thesis objective, and list of contributions. In Chapter 2, we present a literature review and preliminary information related to the thesis. Preliminary includes software metrics brief description, various bug datasets descriptions, employed performance measures, and baseline methods. In Chapter 3, we demonstrate our first proposed ensemble-based classification model for labeled datasets. Further, in Chapter 4, we overcome the limitation of Chapter 3 and proposed an unsupervised classification SBP approach for unlabeled datasets. After that, in Chapter 5, we overcome the limitation of Chapter 3 and Chapter 4 and proposed an unsupervised SBCV prediction model (regression-based) for unlabeled datasets. Later on, we analyze that the previous chapters are dealing with only object-oriented datasets publicly available for researchers, so to explore other paradigms, in Chapter 6, we have created four novel datasets from Haskell packages and proposed an unsupervised SBP model for predicting bug in Haskell (functional

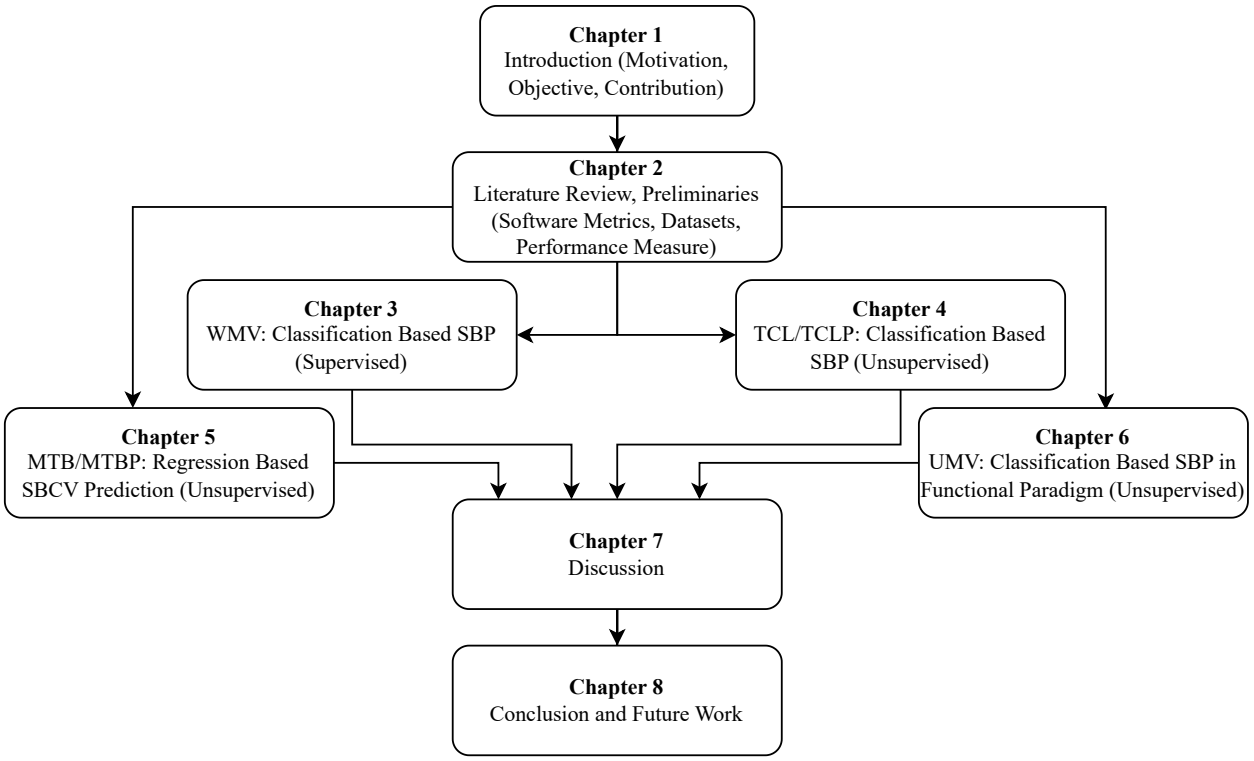

FIGURE 1.2: Thesis structure

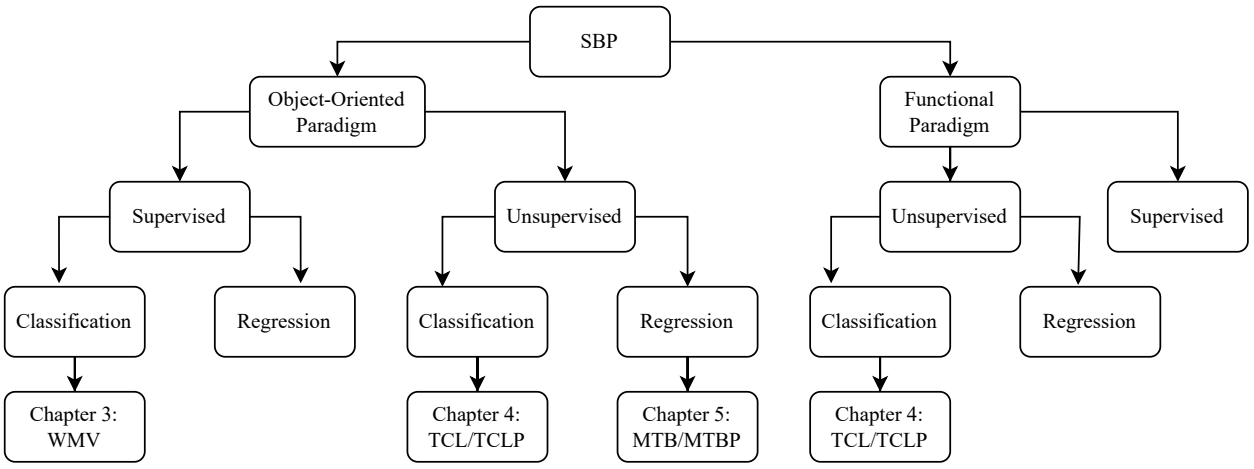

FIGURE 1.3: Concept map of thesis

paradigm datasets). At last, in Chapter 7 and Chapter 8, we present the discussion and conclude the thesis with future research work, respectively.