2017

Using Natural Language Processing and Machine Learning Techniques to Characterize Configuration Bug Reports: A Study

Wei Wen
University of Kentucky, wwen0@uky.edu
Digital Object Identifier: https://doi.org/10.13023/ETD.2017.047

STUDENT AGREEMENT:

I represent that my thesis or dissertation and abstract are my original work. Proper attribution has been given to all outside sources. I understand that I am solely responsible for obtaining any needed copyright permissions. I have obtained needed written permission statement(s) from the owner(s) of each third-party copyrighted matter to be included in my work, allowing electronic distribution (if such use is not permitted by the fair use doctrine) which will be submitted to UKnowledge as Additional File.

I hereby grant to The University of Kentucky and its agents the irrevocable, non-exclusive, and royalty-free license to archive and make accessible my work in whole or in part in all forms of media, now or hereafter known. I agree that the document mentioned above may be made available immediately for worldwide access unless an embargo applies.

I retain all other ownership rights to the copyright of my work. I also retain the right to use in future works (such as articles or books) all or part of my work. I understand that I am free to register the copyright to my work.

REVIEW, APPROVAL AND ACCEPTANCE

The document mentioned above has been reviewed and accepted by the student's advisor, on behalf of the advisory committee, and by the Director of Graduate Studies (DGS), on behalf of the program; we verify that this is the final, approved version of the student's thesis including all changes required by the advisory committee. The undersigned agree to abide by the statements above.

Wei Wen, Student

Dr. Jane Hayes, Major Professor

Dr. Miroslaw Truszczynski, Director of Graduate Studies

USING NATURAL LANGUAGE PROCESSING AND MACHINE LEARNING TECHNIQUES TO CHARACTERIZE CONFIGURATION BUG REPORTS:

A STUDY

THESIS

A thesis submitted in partial fulfillment of the requirements

for the degree of Master of Science in the

College of Engineering

at the University of Kentucky

By

Wei Wen

Lexington, Kentucky

Director: Dr. Jane Hayes, Professor of Computer Science

Co-Director: Dr. Tingting Yu, Assistant Professor of Computer Science

Lexington, Kentucky

2017

ABSTRACT OF THESIS

USING NATURAL LANGUAGE PROCESSING AND MACHINE LEARNING TECHNIQUES TO CHARACTERIZE CONFIGURATION BUG REPORTS: A STUDY

In this study, a tool is developed that achieves two purposes: (1) given bug reports, it identifies configuration bug reports from non-configuration bug reports; (2) once a bug report is identified to be a configuration bug report, the tool finds out what specific configuration option the bug report is associated.

This study starts with a review of related works that used machine learning tools to solve software bug and bug report related issues. It then discusses the natural language processing and machine learning techniques. Afterwards, the development process of the proposed tool is described in detail, including the motivation, the experiment design and setup, and results analysis. In order to evaluate the effectiveness of the tool, both cross-validation and a similar validation technique are performed. Results show that the tool is effective at both identifying configuration bug reports and the associated configuration options for the identified bug reports.

This study proves the usefulness of machine learning techniques in solving bug report related issues. It also shows that configuration and non-configuration bug reports have different characteristics that can be learned by machine learning tools. The developed tool can be improved in a number of areas to make it more effective.

KEYWORDS: Configuration Bug Reports, Natural Language Processing, Machine Learning, Weka, NLTK, Scikit-Learn

Wen Wei
Student's Signature

01 / 30 / 2017
Date

USING NATURAL LANGUAGE PROCESSING AND MACHINE LEARNING TECHNIQUES TO CHARACTERIZE CONFIGURATION BUG REPORTS: A STUDY

By

Wei Wen

Jane Hayes
Director of Thesis

Tingting Yu
Co-Director of Thesis

Miroslaw Truszczynski
Director of Graduate Studies

01 / 30 / 2017
Date

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank my advisors Dr. Jane Hayes and Dr. Tingting Yu for their guidance, insightful discussions, encouragement and support during my work on the Thesis. Their knowledge and experience in my field of study were very invaluable to me.

I would also like to thank Dr. Fuhua Cheng for taking time from his busy schedule to review the Thesis and serve as my committee member.

Lastly but not least, I would like to express my deep gratitude for my husband, Jinsong Chen, for his love, encouragement and eternal support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

As software technology advances, software products become more and more complex, and correspondingly maintenance is becoming more expensive and challenging. Maintenance costs account for more than two thirds of the life cycle costs of software products [1]. Essential maintenance activities include bug reporting and bug fixing. In order to facilitate bug maintenance, organizations use bug-tracking systems for users to submit bug reports, and for developers to collect bug information in order to fix bugs. Bug fixing involves both analyzing bug reports and modifying code to fix the bugs. Analyzing bug reports can be tedious and time-consuming, since bug reports can be lengthy and the description can be hard to understand. However, analysis is a very crucial step for developers to move closer to bug fixing. Thus, making this step efficient and effective can greatly reduce the maintenance cost.

Meanwhile, software engineering has advanced so much that nowadays medium to large software systems generally have many configuration options for users to customize in order to meet their needs. For example, users can augment their Mozilla browsers with sophisticated add-ins, change their Eclipse build settings (i.e., configuration options) to use different versions of the JDK or specified libraries depending on the project, build a specific Linux kernel configuration, etc. While such customizability provides benefit to users, the complexity of the configuration space and the sophisticated constraints among configuration settings complicates the process of testing and debugging. Thus, it is not surprising that many configuration bugs remain

undetected and later surface in the field. A study by Yin et al. [2] shows that up to 31% of bugs are related to misconfiguations in several open source and commercial software systems [2], where a majority of misconfigurations (up to 85.5%) are due to mistakes in setting configuration options.

A developer who is assigned to a given bug report first needs to determine the type of the bug, i.e., whether this bug is configuration-related or not. The next step is to use the anomalous configuration options to reproduce the bug. However, developers with insufficient domain knowledge may incorrectly label a bug report or spend time determining the bug type (time that could have been well spent elsewhere). In addition, to understand the bug, developers often need to look through the bug descriptions, which can be lengthy, verbose, and involve multiple developers and users. In fact, Rastkar et al. found that almost one third of the bug reports in the open source projects Firefox, Thunderbird, and Eclipse Platform in the 2011-2012 period were 300 words or longer (deemed lengthy) [3].

Furthermore, it is often non-trivial to determine which configuration options are relevant in order to reproduce a bug. For example, if a developer knows that a bug report describes a configuration bug related to javascript in a browser application, he/she may not be able to quickly determine what the real name of the configuration option is in the configuration database (e.g., Browser.urlbar.filter.javascript). If the developer wants to fix this bug, he/she may spend an exorbitant amount of time searching through the configuration database to find which option is relevant.

Therefore, there is a need for an effective technique to reduce the manual effort required to label configuration bug reports and to identify the root cause configuration options. This need inspired many researchers to study bug reports to find out useful bug information in order to localize and fix the bugs. With the increasing popularity of machine learning and its successful use in many applications, studying bug reports using machine learning techniques has also gained popularity and proven effective.

In this study, a framework is developed that aims to improve configuration-aware techniques and help ease developers' process of debugging and reproducing bugs that need specific configurations for exposition. It focuses on configuration bugs due to incorrect settings of configuration options. Given a bug report, it determines whether it is a configuration bug, and if it is, the approach suggests configuration options to help developers reproduce the bug. It provides at least two benefits. First, developers can label configuration bug reports in an automated and timely manner. Second, with the configuration query component, it allows developers to approximate configuration options that are relevant to the bugs. This can improve the configuration debugging and diagnosis process.

CHAPTER 2

Background and Related Work

This chapter presents the background knowledge in terms of software bugs, bug reports, natural language processing and machine learning, and discusses related work that used either or both natural language processing and machine learning to study software bugs and/or bug reports.

2.1 Software Bugs, Bug Reports and Related Research

This section gives brief definitions of software bugs and bug reports, and stresses the importance of studying them in order to put the ever-increasing software bugs under control. It also discusses related research activities at both the software bugs level and the bug reports level, and compares them with this research work.

2.1.1 Software Bugs and Bug Reports

A software bug is an error, flaw, failure, or fault in a computer program that causes the program to produce incorrect results, or to behave in an unintended way, even crash. As discussed in Chapter 1, software bugs are prevalent, and with the increasing number and complexity of software systems, both the amount and the types of bugs are growing too. Tremendous effort has been put into classifying bug reports, identifying and fixing bugs, as can be seen by the large volume of papers devoted to software bug/bug report research [4-15].

Software bug reports are plain text that can contain the error log, the steps to reproduce the bug, and the product, version, platform, and operating system information. To help track the progress of a bug report, it may be labeled as new, confirmed,

duplicated, fixed, closed, etc. A good bug report should be specific about the problem, and provide information that is as comprehensive as possible. However, there is no effective way to prevent bug reporters from writing free flowing, long bug reports with much extraneous information. Thus, having a tool that can extract useful information from these kinds of bug reports will be very useful.

2.1.2 Related Work

With the prevalence of software bugs, it is not surprising that there is much related research work. Researchers have been approaching the problems both from the code level and from the bug report level. On both levels, using machine learning techniques has become a common practice.

On the bug report level, researchers have been performing a lot of analyses on bug reports for all kinds of purposes. Dommati et al. [4] used machine learning tools to help identify duplicate bug reports so that less time is needed to classify the bug reports. Wang et al. [5] also studied duplicate bug reports. They used natural language and execution information to help detect duplicate reports. When a new bug report arrives, its natural language information is compared with existing ones, and the most similar existing bug reports are presented to the person in charge of marking a bug report as duplicate. Padberg and Pfaffe [6] studied the classification of concurrency bug reports on MySQL and Apache by applying keyword search and machine learning. For classification, they used a linear classifier and a neural network classifier and obtained encouraging results. Kim and Kim [7] proposed a two-phase prediction model that used information from bug reports to suggest the locations of the software that are likely to need fixing. They used the bag of words approach in NLP to extract word tokens from

bug reports to identify features, and then used the features in machine learning as input to train classifiers or to predict which location of the software needs to be fixed. Gegick et al. [8] applied text mining on bug reports to classify security bug reports. They trained the machine learning model on already manually and correctly labeled bug reports, and then used the trained model to identify security bug reports that were manually mislabeled as security bug reports. Evaluation of their models on a large Cisco software system showed moderately to high successful classification rates. Rastkar and Murphy [9] compared a few text mining classifiers called Email Classifier, Email and Meeting Classier, and their own Bug Report Corpus classifier based on which generates the most accurate bug report summaries. This is very helpful to developers since bug reports can be lengthy and loose; using classifiers that can generate concise and accurate summaries greatly alleviate developers' responsibilities. Sureka [10] studied bug reports by splitting them into components such as product name, version number, etc. and used machine learning tools to categorize bug reports into predefined lists of components and predict whether a given bug report is likely to be reassigned. His result shows the presence of correlation between terms in bug reports and components which can be exploited for the task of predicting the correct component of a bug report. Matter et al. [11] proposed an approach to automatically match and assign bug reports to the developers who have the expertise to work on the bugs. They compared the vocabulary used by a developer in his/her code with the vocabulary used in bug reports using a machine learning approach, and depending on how similar the vocabularies are, they recommend to assign or not assign a bug report to a developer.

On the code level, there are also many research activities. Briand et al. [12] applied machine learning techniques in identifying bugs in the code. They used C4.5 decision trees to identify various failure conditions based on information regarding the test cases' inputs and outputs. Their results showed improvement over their original technique. Zimmermann et al. [13] used Eclipse for their study, mapped the defects in its bug database to its source code locations, and built the bug data set and a description of its contents. They used logistic regression to train and classify the likelihood of a file/package as defect-prone. Lamkanf et al. [14] proposed a text mining approach to predict the severity of a bug report. Evaluation of their approach on three open source software products shows that using text mining for the prediction can achieve moderate to high accuracy. Turhan et al. [15] also used a machine learning approach to mine source code for locating and predicting software bugs. Compared to a non-machine learning, rule-based model which requires inspection of 45% of the source code, their machine learning-based model suggested that 70% of the defects could be detected by inspecting only 3% of the code. This suggests that a machine learning approach is a more practical and efficient way to identify bugs.

2.1.3 Contributions of This Research

Compared to the research activities discussed in 2.1.2, the current research is a study of using machine learning techniques to mine useful information from bug reports. Thus, it belongs to the first type of research activities. However, while the activities discussed in 2.1.2 encompass many areas, such as the duplicate bug reports, the security related bug reports, the concurrency bug reports, etc., this research is different in that it studies configuration bug reports, identifies them from non-configuration bug reports,

and finds the configuration options with which the bug reports are associated. It compares how the different software packages, machine learning classifiers, as well as the feature extraction techniques affect the learning outcomes. It takes into account the unique characteristics of bug reports, that is, different bug reports can contain very different words, and validates the classifiers' performance using a technique similar to cross validation. It does this by using only the training bug reports' features for training, since in reality if a trained classifier is used for testing, it will not be able to know the testing bug reports beforehand. It compares the classifiers' performances with using cross validation and with using this technique, and finds that there is generally a small decrease in performance using only the training bug reports for training. However, this is a more realistic use of a classifier in classifying bug reports.

This research also utilizes the NLP techniques to extract configuration options that are likely associated with a configuration bug report. It builds a corpus of configurations. Each configuration is considered a document in the corpus. It processes a configuration into a list of words by splitting the configuration according to the token used to connect the words together. By using a configuration as a document rather than a bug report, it greatly reduces the size of the corpus, and makes the configuration identification process faster.

The two-step processes developed in this study, i.e., configuration bug report identification and configuration option identification is very useful for a developer who is assigned to work on the bug. Being able to find out that the bug described in the bug report is configuration-related, and to further identify the associated configuration(s) will greatly shorten the developer's bug fixing time.

2.2 Natural Language Processing

Natural Language Processing (NLP) [16] is a computer science field of study that evolved from artificial intelligence and computational linguistics, among others. It involves the understanding of human languages and thus enabling computers to derive meaning from human or natural language input as well as to generate natural language and to translate between different languages. In this study, NLP is used to process bug reports, including the extraction of words, identification and removal of common and unimportant words (stop words), word tokenization, and lemmatization/stemmerization. Only after the proper NLP processing can we perform machine learning operations to learn and predict the types of bugs described in bug reports. NLP is also used in this study to identify the specific configuration(s) (the name of a configuration) associated with a bug report.

2.2.1. Tokenization, Lemmatization and Stopwords

The basic steps involved in NLP are word/sentence tokenization [17], stopword removal [18], stemming [19], part-of-speech tagging [20], lemmatization [21], and chunking and chinking [22]. In this work, some of these steps are employed to convert bug reports into a “bag of words” in preparation for machine learning.

Tokenization is the process of splitting paragraphs into sentences or splitting sentences into words. Words are frequently used as features in machine learning, specifically text mining. Thus, word tokenization is often necessary. After tokenization, there are still many common words which do not convey much meaning and should be removed. These are called stopwords, and they include words such as “the”, “this”, “a”, “is”. Removing stopwords reduces the dimensionality of the features (words) in machine

learning. Dimensionality of the features in this study is the number of words (features) used in machine learning. Dimensionality reduction [23] reduces the time and storage needed. It also improves the performance of the classifiers by removing multi-collinearity. To further reduce the dimensionality, stemming or lemmatization is performed. Stemming is the process of removing the ends of a derived word to hopefully get its root form. However, because of its simplicity, the transformed token may not be a linguistically correct word. Lemmatization, on the other hand, always returns the true root form of a word. In order for lemmatization to work correctly, the original word has to be tagged, which means that the word needs a tag to identify it as a noun, verb, or other part of speech so that lemmatization can restore the word to its correct root form. Since we use NLTK for this work, the stopwords are from NLTK's "corpus" package, and the lemmatization is from NLTK's "stem.wordnet" module.

2.2.2. Feature Extraction

In text mining, feature extraction [24] generally means extracting words as features from text. The steps discussed in section 2.2.1 are some of the essential ones to extract features for machine learning. In this section, we only concentrate on two additional techniques that will eliminate noisy features and reduce feature dimensionality, i.e., information gain with chi-sq and bigram.

Information gain [25] is a measure of how common a word is in a particular class (label) compared to how common it is in other classes (labels). In order to extract high information features, information gain needs to be calculated for each word. Chi-sq [26] can be used to score the commonness of a word in a class. The higher the score, the more likely the word is to be associated with the class. For simplicity, assume that there are

bug reports of two classes t1 and t2 (e.g., configuration vs. non-configuration in this study). Let n_{ii} be the counts that the word (say ***w***) in consideration occurs in bug reports of class t1, n_{io} be the counts that ***w*** occurs in reports of t2, n_{oi} be the counts of all words except ***w*** that occur in reports of class t1, n_{oo} be the counts of all words except ***w*** that occur in class t2, and n_{xx} be the counts of all words that occur in reports of both types. The Chi-sq score that shows how likely this word is associated with type t1 is calculated as:

$$Chi\_sq\_score = \frac{n_{xx} \times (n_{ii} \times n_{oo} - n_{io} \times n_{oi})^2}{(n_{ii} + n_{io}) \times (n_{ii} + n_{io}) \times (n_{io} + n_{oo}) \times (n_{io} + n_{oo})}$$

The score that shows how likely ***w*** is associated with class t2 can be calculated similarly.

Bigram [27] identifies two words that are likely to co-occur. For example, if a bug report contains "not configuration," it indicates that this report is not related to configuration bugs. If individual words are used as the only features, this report may be incorrectly classified as configuration-related. Thus, including bigrams increases the chance of correctly classifying bug reports. The likelihood of two words occurring together is calculated using chi-sq. The only change is that now instead of the association between a word and a label, the association is between two words.

2.3 Machine Learning and Its Tools

Machine learning [28] grew out of artificial intelligence, and is an interdiscipline of computer science and statistics. It started from people's quest to build a system that can learn and improve from experience, and thus be used for all kinds of

tasks. It has been used successfully in many fields, such as speech recognition, computer vision, e-commerce, etc. Other fields, from biology to control theory, have also shown increasing interest in how their systems can automatically adapt or optimize to their environment. With the exponential increase in online data, machine learning is becoming very popular to detect hidden patterns to support business success and to make users' online experience easier and more enjoyable.

With the successful application of machine learning in many fields and the pressing needs in software engineering to tackle the growing number of software bugs, recently machine learning has been gaining immense popularity in bug prediction and bug report classifications. This research takes advantage of the general usefulness of machine learning in problem solving and the prevalence of configurations in software as well as the need to put the ever increasing number of bugs under control. It utilizes machine learning tools to identify configuration bug reports based on known or labeled configuration/non-configuration bug reports. Although machine learning itself may be utilized for different purposes, the next step after processing text data with NLP is often to run machine learning tools on the processed data so that useful results can be extracted from the data. In our study, we use machine learning to build classifiers from labeled bug reports and use them to help identify a new bug report as configuration-related or non-configuration-related.

2.3.1. Classifiers in Machine Learning

There are three types of machine learning: supervised machine learning [29], unsupervised machine learning [30] and reinforcement learning [31]. In supervised

learning, there are labeled examples for the system to learn. A learned classifier is then constructed from the labeled examples, which are then used to label new inputs. In unsupervised learning, there are no labels in the given input data. The systems are supposed to detect patterns from the input data. Because of this, there is no obvious error metric to use to evaluate the classifiers. Reinforcement learning is useful in learning how to react given occasional reward or punishment signals. It finds uses in other disciplines such as game theory and genetic algorithms. In this work, we focus on supervised machine learning only.

There are many supervised machine learning algorithms (or classifiers) in use today. Popular ones include Naïve Bayes [32], Decision Trees [33], and Logistic Regression [34]. All these classifiers are generally used in natural language processing and text mining. The Maximum Entropy Classifier [35] is also commonly used and is provided in NLP. In addition to these classifiers, in order to establish a baseline classifier with which to compare, the simplest classifier, i.e., ZeroR [36], is also in use.

2.3.1.1 ZeroR

During training, ZeroR ignores the features and relies only on the labels for predicting. Although it does not have much predicting capability, it establishes the lowest possible predictability that a classifier can have. It works by constructing a frequency table for the labels in the training data and selects the most frequent values of the testing data in predicting.

2.3.1.2 Naïve Bayes

Naïve Bayes classifier uses Bayes algorithm and is statistic-based. In order to find the label (in our case, the configuration or non-configuration type of a bug report), it uses

the Bayes rule to represent P(label|features) in terms of P(label) and P(features|label), as shown below:

$$P(label|features) = \frac{P(features|label) \times P(label)}{P(features)}$$

Features are the input to the machine learning classifier. For text mining, they can be a bag of words, bigrams, or even trigrams. To simplify the classification work, the classifier also makes a "naïve" assumption, i.e., all the features are independent of each other. Thus, the above equation can be rewritten as:

$$P(label|features) = \frac{P(label) \times P(f1|label) \times \ldots \times P(fn|label)}{P(features)}$$

Where f1, f2, … fn are each individual features.

Although the naïve assumption is never really true, the classification results of Naïve Bayes can be quite good, as can be seen by the results shown in Tables 1, 3, 4 and 5 in 3.3.1).

2.3.1.3. Decision Tree

Decision tree is based on a tree structure, where the inner nodes represent the decision nodes and the leaf nodes represent the labels assigned. The decision nodes decide which branch to take based on the values of the features. Building up a decision tree starts with selecting the right features for the decision nodes, and there are a few choices in making this decision. The simplest method to pick a decision node is to consider all the available features and see which one is most accurate in predicting the training data's label and then use that feature. This is not effective though. A better

choice, and also the generally used one, is to measure how much more organized the input becomes after being divided using a given feature. One method to achieve this is to use entropy, which is the sum of the probability of each label times the log probability of the same label. The feature that achieves the maximum entropy [37] is selected as a decision node. Maximum entropy indicates the highest level that can be achieved from the initial unorganized input to the most organized input.

The most noticeable advantage of decision trees is that they are easy to interpret. The main disadvantages are that: (1) they imply ordering of the features as decision nodes in the tree structure, even though there may not be any ordering in the features of the data; and (2) as the tree descends towards the leaves, there are fewer and fewer data available for the training, which easily leads to overfitting. Naïve Bayes does not have any of these issues, and may explain the better performance of Naïve Bayes in our results as compared to decision trees.

2.3.1.4 Logistic Regression

Logistic regression calculates the probability of a label likely to be assigned to a bug report when given an input feature set. It uses the Bernoulli distribution function for the probability and a sigmoid function for transforming the input:

$$p(y|x,w) = Ber(y|sigm(w^T x))$$

$$sigm(\alpha) \triangleq \frac{1}{1+\exp(-\alpha)}$$

where *Ber* represents the Bernoulli function, *sigm* represents the sigmoid function, *y* is the label, *x* represents the input features, and *w* is the model's weight vector. Model

parameter w is calculated in the training step. In the prediction step, the label with the highest probability is assigned to the bug report.

2.3.1.5. Maxent

Maxent is also called Maximum Entropy Classifier. It is similar to Naïve Bayes except that it uses search techniques to find the features that will maximize the performance of classification rather than using probabilities. To search for features, Maxent chooses those that contain the fewest unwarranted assumptions, which means the maximum entropy in the input. Maxent is equivalent to Logistic Regression [38].

2.3.2. Machine Learning Tools

There are many good open source machine learning tools available. Popular ones include Weka [36], NLTK [39], and Sklearn [40]. They are all used in this work because each one of them has its own strength. Weka has rich GUI functionalities and abundant tools to perform both machine learning and natural language processing. NLTK and Sklearn are for use with Python. Because Python is an easy to use language, this makes coding using NLTK and Sklearn easy as well. In addition, NLTK has a rich set of natural language processing modules, and Sklearn has many classification modules for machine learning. We also selected these tools for comparison purposes, to identify the best tool for this work.

2.3.2.1. Weka

Weka is a well-known suite of machine learning tools developed at the University of Waikato, New Zealand [41]. It is written in Java, and contains both command line and GUI operations. The GUI interface is easier and more convenient to use, and has three applications to suit different needs: Explorer [42], Experimenter [43], and

KnowledgeFlow [44]. Explorer is the easiest to get started with; however, it lacks some of the capabilities of Experimenter and KnowledgeFlow. For example, in Explorer, one can only perform one cross-validation, while in Experimenter there are more choices, such as 10 times cross validation. In addition, there is no easy way to save the classification results in Explorer.

Experimenter can be used for batches of experiments making it easy to compare the performance of different classifiers, and results can easily be saved as csv or arff files. Arff is short for Attribute-Relation File Format, and is the only file format that Weka recognizes. Although Weka can accept csv files as input, internally the csv files are still converted to arff files.

KnowledgeFlow supports the flow of information from one component to the next. The user chooses the components from a large selection. Components are functional blocks that perform certain tasks, such as DataSources, Filters, and Classifiers. In KnowledgeFlow, the components form the palette from which the user can select. Components can then be put on the canvas (editing screen) and connected together to perform processing to meet the specific needs of the user.

2.3.2.2. Sklearn

Sklearn, also called scikit-learn, started as a Google Summer of Code project and is a library of machine learning algorithms for Python programming [45]. It contains various classification, regression, and clustering algorithms, as well as text preprocessing facilities, such as TfidfVectorizer which we use to calculate the tfidf scores of words that occur in documents.

TFIDF [46] is short for Term Frequency – Inverse Document Frequency. Term Frequency ($tf_{t,d}$) is defined as the number of occurrences of a term, *t*, in the document *d*. If *t* is not in *d*, the value of $tf_{t,d}$ is zero. Document Frequency (df_t) is defined as the number of documents in the corpus that contains the term *t*. If *t* does not exist in any documents in the corpus, df_t is equal to zero. The Inverse Document Frequency (idf_t) is used to reduce the effect of terms that appear in many documents; it is defined as:

$$idf_t = log \frac{N}{df_t}$$

where *N* is the total number of documents in the corpus.

Thus, from the equation, a large value of df_t makes idf_t small.

TFIDF is used to measure the importance of a term in a document. If the term appears many times in only a few documents, but rarely in other documents, then it will have a high TFIDF score. This means that the term is very informative in conveying the topic(s) of the few documents in which the term appears. The definition of TFIDF is:

$$tf - idf_{t,d} = tf_{t,d} \times idf_t$$

2.3.2.3. NLTK

NLTK stands for Natural Language processing Tool Kit. It was originally developed at the University of Pennsylvania, and has since been contributed to by dozens of volunteer developers [47]. NLTK is also written in Python and is intended for NLP processing using Python.

Although NLTK is more often used for NLP, it also contains a few popular classifiers for machine learning purposes (for use after the natural language text is processed). These include NaiveBayes, DecisionTree, MaxEnt, and others. Since these classifiers are native to NLTK, they are easier to use after using NLTK's natural language processing algorithms; the classifiers in Sklearn and Weka have to be called from NLTK using NLTK's Sklearn and Weka wrapper methods.

2.3.3. Performance Evaluation Metrics for Classification

In order to pick the best classifier for predicting bug reports not yet labeled, it is necessary to have a set of evaluation metrics that can fully examine the performance of a classifier. The following metrics are commonly used in machine learning.

2.3.3.1. Accuracy

Accuracy [48] is the simplest metric used to evaluate a classifier. It measures the percentage of correctly predicted test data over all test data. Accuracy is not a good metric: when the two labeled data sets are hugely unbalanced, a bad classifier can blindly label every input to be in the majority class and can still achieve very high accuracy. For this reason, precision and recall or F-measure are preferred to accuracy.

2.3.3.2. Precision and Recall

For binary data (those that have only two labels), one can assume that one label is positive and that the other is negative. Even if the data is not binary, we can let the label in consideration be the positive label, and all other labels be negative. Precision [48] is then called the positive predictive value. It is the percentage of correctly predicted positive data (TP) over all predicted positive data. Recall [49], also known as sensitivity,

is the percentage of correctly predicted over all positive. Thus, they can be written in the following mathematic forms:

$$Precision = \frac{TP}{TP + FP} \times 100\%$$

$$Recall = \frac{TP}{TP + FN} \times 100\%$$

where *TP* stands for True Positive, *FP* stands for False Positive, *FN* stands for False Negative, *TP+FP* is all data that are predicted to be positive, and *TP+FN* is all positive data.

2.3.3.3. F-measure

F-measure [50] is a more comprehensive measure of performance as it takes into account the effect of both precision and recall and is the harmonic mean of both:

$$F\_measure = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

2.3.4. Common Practices in machine learning

Since machine learning is closely related to statistics, learning results need to be statistically evaluated. Common practices include cross-validation to validate the generalization of a classifier's result, T-test to check if two sets of result data are statistically significant, as well as other tests.

2.3.4.1. Cross-Validation

Cross-validation [51] in machine learning is a validation technique to assess the classifiers' performances. It evaluates how the results of the classifiers will generalized to

an independent data set. In other words, it overcomes the problem of overfitting and makes the predictions more general.

In machine learning, 10-fold cross-validation is commonly used. In 10-fold cross-validation, all training data is divided into 10 equal parts. Each time, 9 parts are used for training, with 1 part used for testing. This will repeat for 10 times. The average of the performance metrics (accuracy, precision, recall, and F-measure) as discussed in 2.3.3 are calculated to determine how good the classifiers are. **Error! Reference source not found.** shows this process. The advantage of cross-validation is that all data in the dataset are used for both training and testing, which reduces overfitting.

Figure 1. A sketch of 10-fold cross-validation.

Since machine learning is rooted in statistics, machine learning results naturally need to be tested statistically for its significance. Because of this, researchers more commonly perform 10 times 10-fold cross-validation. The process is similar to the 1 time 10-fold cross-validation discussed above. However, before each 1 fold cross-validation, the data is fully randomized. This is repeated 10 times. This further reduces overfitting. With 10-time 10-fold cross-validation, 100 data points for each performance metric are generated. With this large number of data, it can be applied on hypothesis tests such as T-test with data being generally normal. The normality of the performance metrics data in

this study will be shown later in section 3.2.2.3. Due to large number of data, only some samples will be shown.

2.3.4.2. Statistic Tests

In general statistic tests including T-tests [52] have two hypotheses: the null hypothesis [53], which assumes that the two data sets are statistically equal; and the alternative hypothesis [54], which assumes that the two data sets are statistically different. P-value [55] calculated in statistic tests is the probability of finding out if one data set is significantly different from the other. A commonly used p-value is 0.05, which indicates a 95% confidence level. A confidence interval is an interval estimate combined with a probability statement. It is the percentage of all possible samples that can be expected to include the true population parameter. When using p-value of 0.05, if the T-test results give us a p-value of less than 0.05, it means that the two data sets are significantly different, or the alternative hypothesis is true; otherwise, we cannot reject the null hypothesis (the two data sets are statistically equal).

2.3.4.2.1. T-test

Many statistic tests can be used to evaluate experiment data. However, in this study, we used T-test, which is a hypothesis test, on the mean of the data. In our case, the data is performance metrics data from the 10-time 10-fold cross validation results and other results from a test similar to cross-validation.

There are two types of T-test, paired T-test and unpaired T-test. Paired T-test is generally used for studies where the two pairs of data are generated before and after some treatment.

Unpaired T-test, also called student's T-test, is applied to two independent data sets. In this test, the two data sets do not have to have the same data size. It assumes that the data is from a normal distribution and that the standard deviation is approximately the same in the two data sets. It calculated the mean difference and p-value.

2.3.4.2.2. F-test

F-test [55] is used to test if two population variances are equal. It does this by comparing the ratio of two variances. Variances are a measure of dispersion, or how far the data are scattered from the mean. If the variances of two populations are equal, the ratio is 1.

In this study, F-test is used to test if the variations in the performance metrics results between two classifications are equal. For example, different techniques were used to improve classifiers performance. The baseline technique is to use all words (AW) of all the documents for classification. The others are using high information words (HIW) and using high information words plus bigram (HWB). Using F-test, we can determine if the metrics data from the different techniques have equal variance or not. The purpose of this is for T-test. There are two types of T-tests in Microsoft Excel. One is for equal variance and the other is for unequal variance. F-test results will determine which T-test to use in order to find out the statistical significance of the metrics data between the baseline (AW) and HIW/HWB.

Chapter 3

Predicting Configuration Bug Reports and Extracting Configuration Options

This chapter discusses the current research work in detail. It uses a Mozilla configuration bug report as an example to promote the importance of this study. It then delves into the design and the result analysis of this research.

3.1 Motivation of the Study

A configurable system is a software system with a core set of functionality and a set of variable features which are defined by a set of configuration options [56]. A configuration option can be specified in a configuration file, source code, and/or in a user input option. A configuration database (also called a configuration model) consists of all the configuration options in an application. Constructing an effective configuration model has been well discussed in recent work [56]. In this study, it is assumed that the configuration model is known. Changes to the value of a configuration option may change the program's behavior in some way. If such changes cause the system to behave incorrectly, a configuration bug occurs. Firefox, a popular web browser and also a highly configurable system, is used to motivate the approach in this study.

In Firefox, when the configuration option O1 = Browser.urlbar.filter.javascript is set to false, it allows "javascript:" URLs to appear in the autocomplete dropdown of the location bar. This can cause potential security threats. Figure 2 shows a bug report associated with the configuration option O1. This bug involves 22 comments and took 21

months to fix. In fact, at the end of the 21 months, a single change to the value of this configuration option fixed the problem.

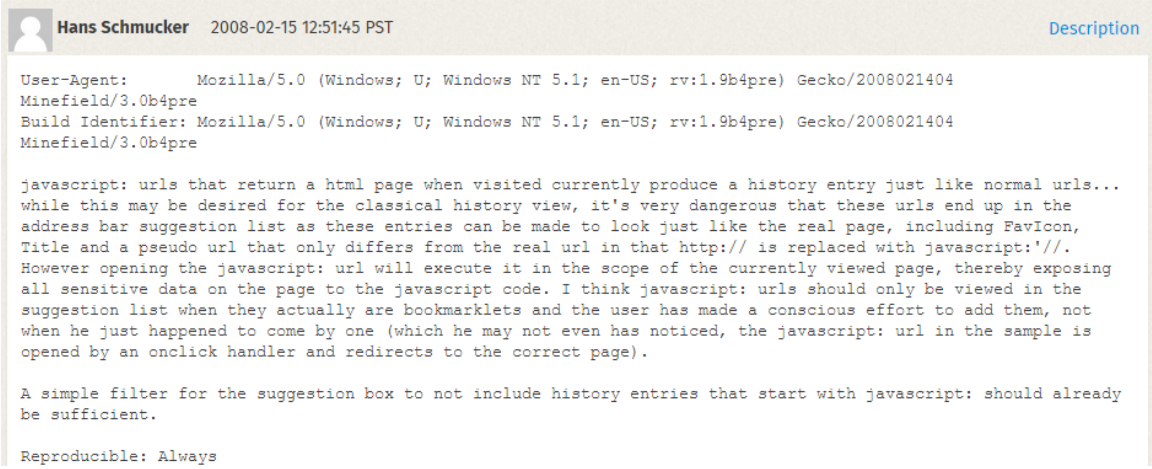


Figure 2. A configuration bug report.

Suppose that an inexperienced developer is assigned to work on this bug. He or she may spend much time figuring out that it is a configuration bug. In such a case, the developer has to inspect the source code and try various inputs and configurations to hopefully reproduce, locate, and fix the bug. Even if an experienced developer is assigned to work on this bug and notices that it is related to configurations based on the system's specific behavior (e.g., mouse scrolling events), she may not be able to quickly determine the real configuration option from the configuration database (i.e., O1) as there are approximately 1650 possible configuration options in the configuration model of Firefox. Therefore, to ease the process of configuration, debugging, and diagnosis, we need new techniques that can identify a configuration bug report and link the bug to specific configuration options.

In this study, it has been observed that natural language descriptions of a bug report provide information to indicate whether a bug is related to configuration options. In the running example, the word “bookmarklets” is likely to be an indicator of a configuration problem. The word “javascript” ties to the name of a configuration option O1 obtained from the configuration model. Based on these observations, it was determined that natural language processing (NLP) techniques can be used to process text reports and convert them into individual words to be used as features in machine learning. Developers can use the trained machine learning classifiers to label a bug report as either a configuration bug report or a non-configuration bug report. Also, with the help of NLP and information retrieval (IR), the classifier returns a list of ranked configuration options extracted from the configuration bug reports to the developers. In the above example, O1 is ranked at the top of the list.

3.2. Experiment Design and Setup

This section discusses the design and the setup of the empirical study. In the design, a top level diagram and two detailed diagrams are presented. The top level design describes the two main functional blocks of this study, i.e., the classification of the bug reports and the identification of the configuration options. The two detailed diagrams show how each of the functionalities are carried out.

In the setup, the data sources of the bug reports are discussed and the methods/tools used in the analyses of the classification results are presented.

3.2.1. Experiment Design

The design for this study includes two main steps, as shown in Figure 3. The first step is called **classification**, and it takes bug reports with known labels (configuration vs. non-configuration) as input, trains classifiers on these bug reports, then uses these classifiers to predict the un-labeled bug reports, i.e., identify them as either configuration or non-configuration bug reports. The second step is called **Configuration Identification**: it accepts the labeled configuration bug reports, uses NLP procedures to find the similarities between the bug reports and the configuration names, and outputs the associated configuration names ordered from more likely to less likely.

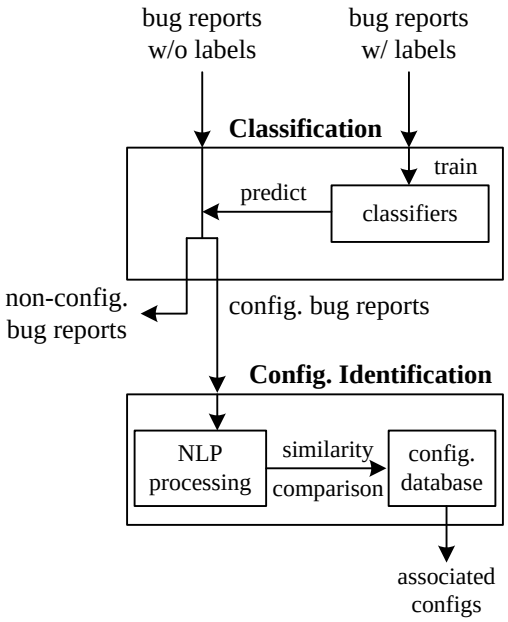


Figure 3. The Process flow of bug reports classification and configuration identification.

Figure 4 is a more detailed sketch of the classification step (step 1). A webpage that has bug report information also contains extraneous information that needs to be excluded. Thus, this step starts with extracting useful information from a webpage given

bug reports URLs. A python package called **Beautiful Soup** is used to accomplish this. A text file is thus created that contains only the relevant information from the webpage. Generally, only the title and the comment text from the webpage are included.

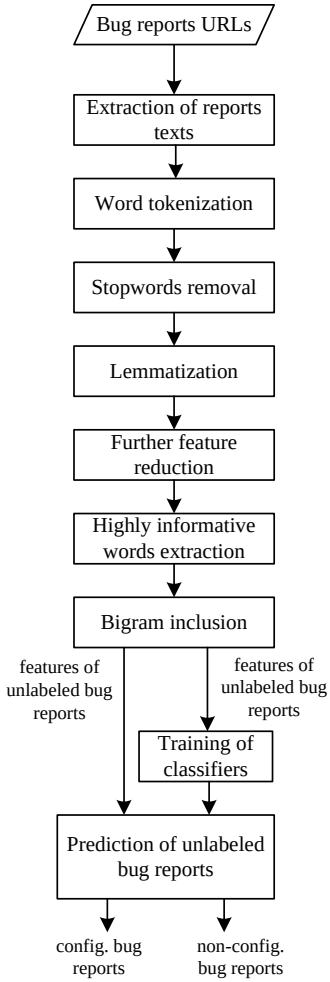


Figure 4. Bug reports classification process.

Features are then extracted from the cleaned bug reports (now the plain text files). The features in this study are words for machine learning. Feature extraction encompasses the next six small steps in Figure 4. These are NLP procedures, and Python's NLTK package is used to simplify the tasks. The bug report texts are first broken down into words. Then, stopwords are removed. It is found that just using the

default stopwords corpus in NLTK is not enough, so more stopwords are included that are specific in the bug reports, such as people's names. This step is followed by lemmatization and more feature reduction to further reduce the dimensionality of the features. Lemmatization restores a word to its dictionary form. Thus, for example, "configurations" will become "configuration." In addition, it is also found that people sometimes write "preference" as "pref," and "configuration" as "config." So words such as these are converted back to their original forms. Still there can be many words that are considered noisy data in machine learning and could not only reduce learning performance, but also misguide a classifier. As an example, Tables 1 and 2 in section 3.3.1 compare the classification results using all words as features and using only selected words as features. The result is much better in the case of the selected words (high information words and/or bigrams).

Thus, to improve the performance of the classifiers, a few more steps can be included, i.e., information gain using chi-sq and bigram. The fundamentals of chi-sq and bigram have been discussed in Chapter 2. In NLTK package, there are two modules called ***metrics*** and ***collocations.*** The metrics module has the ***BigramAssocMeasures*** class which contains an implementation of chi-sq. The ***collocations*** module has the ***BigramCollocationFinder*** class that can be used to find n-gram (n is a number, e.g. n=1 means unigram, n=2 means bigram). These classes are used to identify the highly informative words and commonly occurring bigrams. The selected features (words and bigrams) are arranged in the form of a dictionary with the words/bigrams as the key and the assigned values as the values of the keys. The format used is {word: True}, where word is the word selected, and the value is "True." As long as a word/bigram is selected

as a feature, "True" is assigned as the value for it to indicate that the word/feature appears in that report. Using any other values for the same key will be considered another feature, which is not correct. For example, {preference: True} and {preference: 1} are considered two features, even though the word feature is the same, namely "preference." Although only 100 high informative words are retained as features, the results can sometimes be much better than indiscriminately including all words. For this study, in order to evaluate the effectiveness of including only high information words and bigrams, three groups of word features are extracted for each bug report database, i.e., all words (AW), high information words (HIW), and high information words plus bigrams (HWB). AW is the control study, and the other two are compared to AW.

Text in bug reports can vary quite a lot, meaning that one bug report may contain words which are quite different from those in another report. Thus, the extracted features in one report may be quite different from the other reports. This creates a problem for some machine learning tools, e.g. Weka classifiers. In Weka, an ARFF file is provided as input to a classifier for learning, and the features (words in this study) in the ARFF file are fixed, although they can take on different but allowed values. Classifiers in NLTK are tailored to this special characteristic of text mining (varying words in different bug reports). Thus, using the above procedures to extract features does not work in Weka. Because of this, feature extraction for Weka classifiers is done inside Weka GUI. Although the steps are not exactly the same as Figure 4, the procedures are essentially the same.

In Weka, the first step is to convert the text files that are created in step 2 (extraction of reports texts) of Figure 4 into ARFF files. This is performed using the 4th

sub-application in Weka GUI, i.e., Simple CLI. The command used in CLI is: ***java.weka.core.converters.TextDirectoryLoader***. This is a fast operation and it creates the ARFF file for all the documents in a specified directory in less than a minute. In order to generate the correct ARFF file, inside the directory there should be two sub directories with the names ***config*** and ***nonconfig***. ***config*** subdirectory contains all the known configuration bug reports, while the ***nonconfig*** subdirectory contains all the known non-configuration bug reports. The subdirectory names are important since they are the clue for Weka to assign as values to the class. In this case, the class can take the value of either ***config*** or ***nonconfig***. For example, for all the Mozilla bug reports converted from URL links into 300 text files, the 150 configuration related text files are stored in the ***config*** directory, and the 150 non-configuration related are stored in the ***nonconfig*** directory. These two directories are subdirectories of "Mozilla" with prepending path.

The ARFF files created contains only two features, one is the whole text, and the other one is the class (config vs nonconfig). To convert the text into words, the ***StringToWordVector*** filter is used in the Weka Explorer sub-application to further divide the text feature into word features. There are a number of options to select to get high information words and high information words plus bigrams. The steps are similar to steps 2-8 in Figure 4 with some differences. For example, Weka does not contain lemmatization, but stemmer. And thus, in HIW and HWB the LovinsStemmer is used. LovinsStemmer is the oldest in use and is also the fastest [57]. For AW, no stemming is used except that the extracted features are all words; numbers and other symbols are discarded. High information words are extracted using Tfidf.

With the feature sets prepared, they are then passed to the classifiers for training. The classifiers included in this study are those discussed in Chapter 2. NLTK classifiers used are Naïve Bayes, Decision Tree, and MaxEnt; Sklearn classifiers are Naïve Bayes, Decision Tree, Logistic Regression and SVC (Support Vector Machine for Classification); Weka classifiers are ZeroR, NaiveBayes, Decision Tree (J48), Logistic Regression and SVM.

In Weka as in NLTK and Sklearn, classification is performed programmatically rather than using Weka's Explorer or Experimenter. In Explorer, only one round of 10-fold cross validation can be performed, which generates too little data for statistical significance analysis. Experimenter can run 10 times 10-fold cross validation; however, it contains too much irrelevant information and can be hard to locate the performance metrics data in the exported csv files. Since Weka is written in Java, in this study Java programs are written to run Weka classifiers.

Figure 5 shows the flow of the second step in Figure 3. There are a configuration database and the configuration bug reports. The configuration database is a list of configuration names and, depending on the software, the configuration names can use camel case or have dots or underlines separating the words. So NLP processing of the configurations is different from that shown in Figure 3. In processing configurations, the words are split by the camel case, the underlines, and the dots. Sometimes when two words are combined without any of the above, regular expressions are used to split them. The words are restored to their root forms with lemmatization. Then the words are combined by spaces to become a string of words. The end result of this is one document is one string of words. For example, the Mozilla configuration

Browser.urlbar.filter.javascript discussed in section 3.1 is split into the words: ***browser***, ***url***, ***bar***, ***filter***, ***javascript***. These words are combined to become ***browser url bar filter javascript***. This is one document in the configuration database. After all the configurations are converted, the configuration corpus is built up and ready for the next step, i.e., Tfidf fitting and transformation.

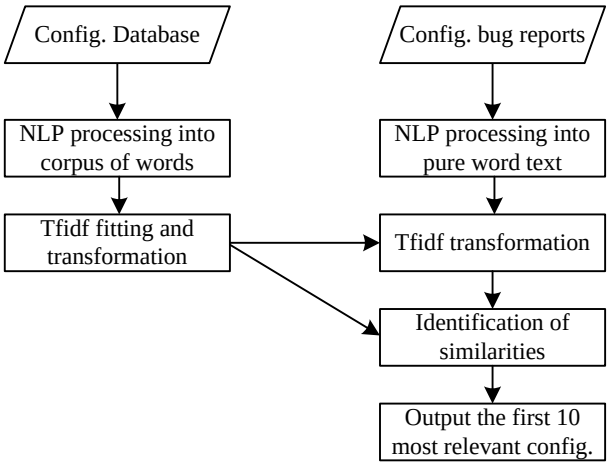


Figure 5. Configuration identification of a configuration bug report.

For configuration bug reports, in addition to similar processing described above for the configuration database, they are first processed as in Figure 6 (steps 2-8). The configuration corpus is then processed with **TfidfVectorizer** in Sklearn to convert the collection of documents (each document is one configuration transformed into a string of words) to a matrix of TF-IDF features. Figure 6 lists part of the configurations in the three open source projects. Before discussing the procedures, it is necessary to provide definitions of the terms to be used.

In this step, the **TfidfVectorizer** learns the vocabulary from the configuration corpus, counts the frequency of the words and finds the inverse term frequency of the

words in each document of the corpus. In **TfIdfVectorizer**, both unigrams and bigrams for the **ngram_range** parameter are included to help increase the chance of finding the right similarities between bug reports and configurations. Including bigrams will significantly increase the size of the matrix; however, since a configuration's size is very small (only a few words) compared to a bug report's size, this is not much of a penalty. The trained **TfidfVectorizer** is then used to transform (construct matrix form) the bug reports. Each bug report is processed one at a time in the whole flow. After the bug report transformation, **TfidfVectorizer** presents the Tfidf score of each word that occurs in both a configuration and a bug report. A naive approach is adopted to calculate the similarity score. This is done by adding the scores of all words that occur in each configuration and the bug report and output the configurations with the first 10 highest scores.

Apache	Mozilla	MySQL
AuthFormLogoutLocation	Accessibility.accesskeycausesactivation	Audit_log_events_filtered
AuthFormMethod	Accessibility.blockautorefresh	Audit_log_events_lost
AuthFormMimetype	Accessibility.browsewithcaret	Audit_log_events_written
AuthFormPassword	Accessibility.disablecache	audit_log_exclude_accounts
AuthFormProvider	Accessibility.disableenumvariant	audit_log_file
AuthFormSitePassphrase	Accessibility.tabfocus	audit_log_flush
AuthFormSize	Accessibility.tabfocus applies to xul	audit_log_format
AuthFormUsername	Accessibility.typeaheadfind	audit_log_include_accounts
AuthGroupFile	Accessibility.typeaheadfind.autostart	audit_log_policy
AuthLDAPAuthorizePrefix	Accessibility.typeaheadfind.casesensitive	audit_log_rotate_on_size
AuthLDAPBindAuthoritative	Accessibility.typeaheadfind.enablesound	audit_log_statement_policy
AuthLDAPBindDN	Accessibility.typeaheadfind.enabletimeout	audit_log_strategy
AuthLDAPBindPassword	Accessibility.typeaheadfind.flashBar	Audit_log_total_size
AuthLDAPCharsetConfig	Accessibility.typeaheadfind.linksonly	Audit_log_write_waits
AuthLDAPCompareAsUser	Accessibility.typeaheadfind.prefillwithselection	auto_generate_certs
AuthLDAPCompareDNOnServer	Accessibility.typeaheadfind.soundURL	auto_incrment
AuthLDAPDereferenceAliases	Accessibility.typeaheadfind.startlinksonly	auto_increment_increment
AuthLDAPGroupAttribute	Accessibility.typeaheadfind.timeout	auto_increment_offset
AuthLDAPGroupAttributeIsDN	Accessibility.usebrailledisplay	autocommit
AuthLDAPInitialBindAsUser	Accessibility.usetexttospeech	automatic_sp_privileges
AuthLDAPInitialBindPattern	Accessibility.warn on browsewithcaret	avoid_temporal_upgrade
AuthLDAPMaxSubGroupDepth	Advanced.mailftp	back_log
AuthLDAPRemoteUserAttribute	Advanced.system.supportDDEExec	basedir
AuthLDAPRemoteUserIsDN	Alerts.slideIncrement	big-tables
AuthLDAPSearchAsUser	Alerts.slideIncrementTime	bind-address

Figure 6. Some configurations in the three open source projects.

3.2.2. Experiment Setup

3.2.2.1 Data for Classification

The efficacy of classification and configuration identification of bug reports in this design is evaluated on bug reports from three open source software projects, i.e., Mozilla, MySQL, and Apache. For diversity of data, we are not restricted to a few specific components of the software. The reason to choose bug reports from these open source projects is that they are popular software, have bug reports generally available, and some Mozilla bug reports are already labeled as associated with some configurations in Mozilla's website. The last characteristic is especially helpful since it can be used as the ground truth to evaluate our design. For those configuration bug reports collected that are not identified as associated with configurations, they are identified manually. In addition, labeling a bug report as configuration or non-configuration is also done manually. This involves both using key word search in bug report database and reading through the reports.

For each software project, 300 bug reports are collected, with equal number of configuration and non-configuration bug reports. Thus, the total number of bug reports collected for all three software projects is 900. Collecting an equal number of configuration and non-configuration bug reports is to ensure that the classifiers that are trained on these bug reports are not biased. For machine learning, it is always desirable to have more bug reports for training However, collecting bug reports and correctly labeling them is very time consuming; besides, this number has been shown to be enough for machine learning [58].

3.2.2.2 Cross Validation

As discussed in Chapter 2, cross validation reduces overfitting and makes the results of trained classifiers generalized to independent bug report prediction. Thus, in the first part of the study, 10-fold cross validation is performed for all the classifiers on all the data sets. In addition,

in order to get enough data to analyze the statistical significance of the results, the 10-fold cross validation is run 10 times in each case to get 100 data points for each performance metric (accuracy, precision, recall and F-measure). Before each run, the bug reports, or the word features of them, are completely randomized so that the result is not a duplicate of the previous run.

Although cross-validation is generally considered a good measure of a classifier's performance, the features are also known for both training data and testing data. In classifying bug reports, if a classifier is used to predict a truly unknown bug report, it would not be possible for the classifier to know the features of the unknown bug report. This is because one bug report can use quite different words than another, and words are the features for classifiers to train. It is therefore predicted that 10 times 10-fold cross validation results will be better than if a classifier is trained only on the features of the trained reports but used to test unknown bug reports. Due to this concern, another type of validation is performed that is similar to the 10 times 10-fold cross validation but takes this into consideration. In addition, in order to get more bug reports for testing, unlike 10-fold cross validation, 5-fold is used. Thus, of each bug report type (configuration or non-configuration), 30 bug reports are used for testing, while 120 are used for training. To get 100 data points for each performance metric, the 5-fold validation has to run 20 times.

Here is how the training and testing process is conducted: In this validation, the bug reports will be shuffled 20 times. After each shuffle, the bug reports are divided into five parts for both configuration and non-configuration. Four parts are copied into a directory that is used for training, and one part is copied into another directory that is used for testing. This is rotated five times similar to Figure 1. Feature extraction for training is done only on the reports in the training directory, while features for testing are extracted from reports in the testing directory. This makes sure that the classifiers do not know the testing bug reports, which is more representative of reality. After one training and testing, one data point is obtained for all the metrics. Then the

training and testing directory's contents are cleared and filled with the next rotated data. After the five rotations are done, it is similar to the completion of one 10-fold cross validation. Then the bug reports are shuffled, and this process repeats for 20 times. To differentiate this from the 10 times 10-fold cross validation, it will be called 20x5 training and testing.

The 20x5 training and testing runs much slower than 10 times 10-fold CV. The main step that makes it slow is in feature extraction. In 10 times10-fold CV, feature extraction is performed only one time, while in the 20x5 training and testing, feature extraction is performed 100 times.

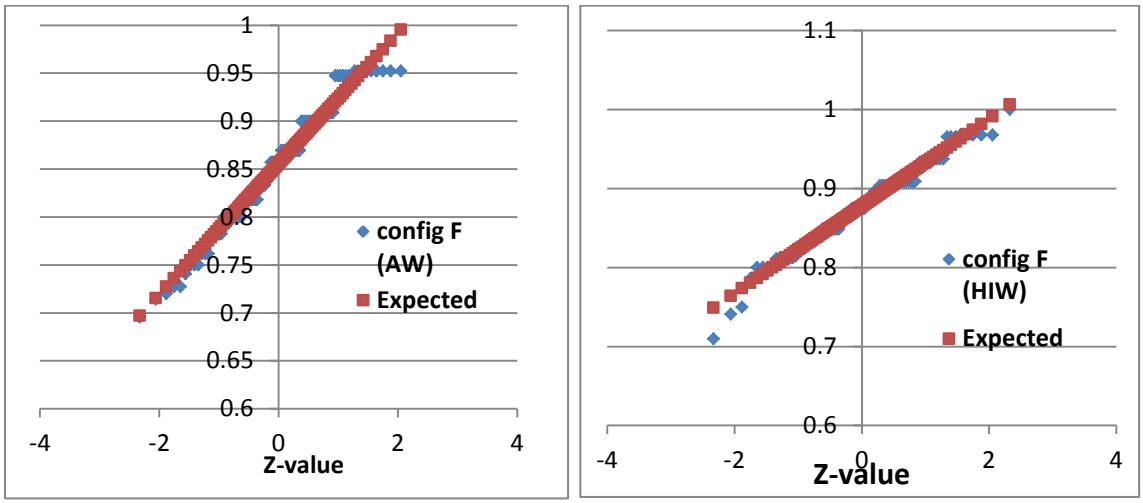
3.2.2.3. Statistical Significance Test

As discussed before, unpaired T-test is performed to evaluate the statistical significance of the performance metrics. In this study, the AW method is considered the baseline or control group, while HIW and HWB are the treatment groups. This is because one of the objectives of this study is to evaluate the effectiveness of using high information words and bigrams as features in classifying bug reports compared to using all words as features.

Microsoft Excel Data Analysis package contains a lot of statistical analysis tools, and it includes both T-test and F-test. F-test is used to compare the variance of the control data vs. the treatment data. When the control and the treatment data have unequal variances, T-test with unequal variance in Excel is used to test for significance; otherwise, T-test with equal variance is used.

F-test assumes that the data is normally distributed and the data points are independent of each other. The second assumption is automatically satisfied since each classification run is independent of others. Thus, in order to use F-test, data normality has to be verified. In this study, Q-Q plot [59] is used to examine data normality. This is done in Excel as well. The raw performance metrics data to be plotted is sorted first. Then Cumulative Distribution Function (CDF) is calculated which is used for calculating expected value and Z-value. Expected value is

calculated using the NORM.INV function in Excel with parameters of CDF and the mean and standard deviation of the raw data. Z-value is calculated using the function NORM.S.INV with parameter CDF. Finally, the data is plotted with Z-value on the x-axis, while the raw data and the expected values on the y-axis. The plot for the configuration F-measure data for Mozilla classified by NLTK Naïve Bayes using HIW is shown in Figure 7 as an example. Expected values in the plot represent data in normal distribution. Raw data and the expected values do not generally deviate much, which is an indication that the data is normally distributed. Thus, using F-test for variance test is valid.



(a) (b)

Figure 7. Q-Q plots of the Mozilla configuration F-measure classified using NLTK Naïve Bayes using HIW.

Figure 8 shows the statistical significance test on configuration F-measure of Apache result. The top table shows the F-test result on the variance of configuration F-measure. Since p-value is great than 0.05, the null hypothesis is assumed true, which means that there is no statistical significance between the two variances. Thus, T-test can be performed, which is shown

as the second table in Figure 8. In this result, p-value is significantly smaller than 0.05. Thus, the null hypothesis is rejected, which means the two means are significant different.

Apache 10 times 10-fold CV, config F-measure

F-Test Two-Sample for Variances

	HIW	*AW*
Mean	0.877576027	0.780063
Variance	0.003054887	0.002735
Observations	100	100
df	99	99
F	1.116899007	
P(F<=f) one-tail	0.291680056	
F Critical one-tail	1.394061257	

P(F<=f) one-tail > 0.05, equal variance

t-Test: Two-Sample Assuming Equal Variances

	AW	*HIW*
Mean	0.780063492	0.877576
Variance	0.00273515	0.003055
Observations	100	100
Pooled Variance	0.002895019	
Hypothesized Me	0	
df	198	
t Stat	-12.8150341	
P(T<=t) one-tail	4.48949E-28	
t Critical one-tail	1.652585784	
P(T<=t) two-tail	8.97898E-28	
t Critical two-tail	1.972017478	

P(T<=t) two-tail << 0.05, reject null hypothesis

Figure 8. F-test on the variance and T-test on the mean of configuration F-measure of Apache 10 times 10-fold CV using NLTK Naïve Bayes.

3.3. Results and Analyses

This section presents results and analyses of classification and configuration identification. Through the results and analyses we can see that using the approaches in

the study to classify configuration bug reports and to identify configuration options is effective.

3.3.1 Classification of Bug Reports into Configuration and Non-configuration Related

The mean values of the performance metrics are shown in Tables 1 to 15. One table contains the results for one open source project (Apache, Mozilla, or MySQL) using one classification software (NLTK, Sklearn, or Weka) with either 10 times 10-fold CV or 50x2 training and testing. HIW and HWB data are compared with AW to see if using high information words and bigrams can improve classifiers' performance. Adopting the format in Weka Experimenter, the HIW and HWB data in the tables are postfixed with either "*" or "v". The symbol "*" indicates that the HIW/HWB data are statistically worse than the AW data, while the symbol "v" indicates that they are statistically better than the AW data. If there is no statistical difference between HIW/HWB and AW, then there is no symbol appended to the data.

Tables 1-5 show the Apache results. Tables 1 and 2 are both from NLTK classifications. In both cases, we can see that using HIW or HWB improves the prediction performance. Since F-measure is the more comprehensive representation of the overall performance of a classifier, F-measure results are used in analysis from now on. The greatest increase in performance is Maxent with 20x5 training and testing. From AW to HIW, the non-configuration F-measure increases by 50%. It is also noted that by not including testing data in feature selection as done in 50x2 training and testing, all three classification schemes (AW, HIW and HWB) show some decrease in the results, as can be seen by comparing Table 1 to Table 2, and Table 3 to Table 4. Although 10 times

10-fold CV (10x10) and 50x2 training and testing use different documents for training and testing, the more number of testing bug reports used in 50x2 should stabilize any variation in results due to too few testing bug reports, and the 120 bug reports of each type in 50x2 should be sufficient for training. Thus, it is believed that the different number of training and testing bug reports in 50x2 as compared to the 10x10 is not the factor contributing to the decrease in performance; it is the fact that not using testing bug reports' features for training that leads to the decrease. This is understandable, since if a classifier uses all features to make a decision, it is like it has seen the bug reports that are used for testing. The result certainly should be better. However, the decrease is minor, which makes 10x10 still a good validation metric. Besides, by using the two different approaches to evaluate the classifiers' performances, it also proves the validity of either one, since the results show very little difference.

Table 1. Apache 10 times 10-fold CV using NLTK classifiers

Eval. Metrics		Maxent			NaiveBayes			Decision Tree		
		AW	HIW	HWB	AW	HIW	HWB	AW	HIW	HWB
Accuracy		0.725	0.872v	0.88v	0.726	0.87v	0.88v	0.812	0.845v	0.834
Precision	**Conf.**	0.7	0.858v	0.867v	0.661	0.84v	0.854v	0.856	0.87	0.853
	Non.	0.806	0.897v	0.906v	0.932	0.921	0.919	0.787	0.833v	0.83v
Recall	**Conf.**	0.862	0.9v	0.906v	0.959	0.925*	0.921*	0.755	0.818v	0.815v
	Non.	0.588	0.845v	0.854v	0.493	0.815v	0.838v	0.868	0.872	0.853
F-meas.	**Conf.**	0.755	0.875v	0.883v	0.78	0.878v	0.884v	0.798	0.839v	0.829v
	Non.	0.643	0.867v	0.876v	0.63	0.86v	0.874v	0.822	0.849v	0.837v

Table 2. Apache 20 times 5-fold training and testing using NLTK classifiers

Eval. Metrics		Maxent			NaiveBayes			Decision Tree		
		AW	HIW	HWB	AW	HIW	HWB	AW	HIW	HWB
Accuracy		0.679	0.857v	0.845v	0.725	0.837v	0.833v	0.808	0.848v	0.838v
Precision	**Conf.**	0.701	0.838v	0.806v	0.657	0.799v	0.783v	0.847	0.861v	0.848
	Non.	0.749	0.885v	0.902v	0.93	0.898*	0.916	0.783	0.842v	0.845v
Recall	**Conf.**	0.81	0.889v	0.912v	0.963	0.91*	0.93*	0.757	0.835v	0.828v
	Non.	0.548	0.825v	0.778v	0.488	0.763v	0.736v	0.859	0.861	0.848
F-meas.	**Conf.**	0.701	0.861v	0.854v	0.779	0.849v	0.849v	0.797	0.846v	0.835v
	Non.	0.567	0.852v	0.833v	0.632	0.821v	0.813v	0.817	0.85v	0.839v

In Tables 3 and 4, we can see that except for Naïve Bayes and Logistic Regression, using either HIW or HWB, the classifiers perform better than using AW. However, compared to NLTK, Sklearn classifiers generally perform better. Even when using AW, the F-measure is still mostly greater than 0.8. The largest increase in performance is SVM from using AW to HIW, which is 22.4%.

Table 3. Apache 10 times 10-fold CV using Sklearn classifiers

Eval. Metrics		Logistic			NaiveBayes		
		AW	HIW	HWB	AW	HIB	HWB
Accuracy		0.869	0.836*	0.913v	0.892	0.877*	0.896
Precision	Conf.	0.899	0.845*	0.946v	0.872	0.869	0.88
	Non.	0.856	0.84*	0.892v	0.925	0.895*	0.924
Recall	*Conf.*	0.838	0.834	0.88v	0.925	0.894*	0.923
	Non.	0.9	0.839*	0.947v	0.858	0.859	0.868
F-meas.	Conf.	0.862	0.835*	0.909v	0.895	0.879*	0.898
	Non.	0.873	0.836*	0.917v	0.887	0.874	0.892
Eval. Metrics		Decision Tree			SVM		
		AW	HIW	HWB	AW	HIW	HWB
Accuracy		0.801	0.853v	0.848v	0.621	0.871v	0.812v
Precision	Conf.	0.814	0.87v	0.867v	0.895	0.96v	0.974v
	Non.	0.804	0.833v	0.842v	0.575	0.817v	0.739v
Recall	Conf.	0.791	0.818v	0.831v	0.268	0.776v	0.642v
	Non.	0.811	0.872v	0.865v	0.974	0.967	0.981
F-meas.	Conf.	0.796	0.839v	0.844v	0.396	0.855v	0.766v
	Non.	0.803	0.849v	0.85v	0.722	0.884v	0.841v

Table 4. Apache 20 times 5-fold training and testing using Sklearn classifiers

Eval. Metrics		Logistic			NaiveBayes		
		AW	HIW	HWB	AW	HIB	HWB
Accuracy		0.871	0.835*	0.9v	0.882	0.867*	0.84*
Precision	Conf.	0.902	0.836*	0.94v	0.853	0.863	0.798*
	Non.	0.85	0.841	0.871v	0.922	0.879*	0.905*
Recall	Conf.	0.835	0.837	0.857v	0.927	0.877*	0.917
	Non.	0.907	0.833*	0.943v	0.836	0.857v	0.763*
F-meas.	Conf.	0.865	0.834*	0.895v	0.887	0.868*	0.852*
	Non.	0.875	0.834*	0.905v	0.875	0.866	0.826*
Eval. Metrics		Decision Tree			SVM		
		AW	HIW	HWB	AW	HIW	HWB
Accuracy		0.801	0.836v	0.844v	0.615	0.854v	0.791v
Precision	Conf.	0.827	0.861v	0.866v	0.918	0.956v	0.972v
	Non.	0.786	0.82v	0.833v	0.569	0.792v	0.714v
Recall	Conf.	0.768	0.807v	0.821v	0.254	0.742v	0.601v
	Non.	0.833	0.865v	0.868v	0.977	0.965*	0.981
F-meas.	Conf.	0.793	0.83v	0.84v	0.387	0.833v	0.737v
	Non.	0.806	0.84v	0.848v	0.718	0.869v	0.826v

Table 5 is the Weka classification results. When using Weka, feature extraction is done in Weka Explorer. Thus, it is not possible to perform 50x2 training and testing programmatically where the feature selections are carried out 100 times. Because of this, only 10 times 10-fold CV is done using Weka.

In general, Weka ZeroR classifier has the worst performance of all classifiers (Weka, NLTK and Sklearn). It does not do any real prediction, but simply labels all bug reports as configuration bug reports, as shown in the 1.0 configuration recall and 0.0 non-configuration recall. It does so regardless of using AW, HIW or HWB. Using HIW or HWB does not provide much performance improvement as in NLTK and Sklearn. We can see that two classifiers (Decision Tree and Logistic Regression) perform better without using high information words and bigram. Weka classifiers may be optimized to work with all words. However, even though Weka's Logistic Regression has much better performance using AW than the HIW and HWB, it sacrifices time for the number. In general, Logistic Regression takes hours (three or more hours) to complete using AW, while it takes only about 10 minutes using HIW or HWB. When time is of a concern, sacrificing a little performance degradation for timely results is a great trade-off. In the worst case, the decrease in performance from using AW to HWB is 15.8% in non-configuration F-measure with Logistic Regression.

Table 5. Apache 10 times 10-fold CV using Weka classifiers

Eval. Metrics		ZeroR			NaiveBayes			Decision Tree		
		AW	HIW	HWB	AW	HIB	HWB	AW	HIW	HWB
Accuracy		0.5	0.5	0.5	0.814	0.854v	0.863v	0.844	0.82*	0.822*
Precision	Conf.	0.5	0.5	0.5	0.844	0.896v	0.865v	0.856	0.855	0.849
	Non.	0.0	0.0	0.0	0.801	0.832v	0.873v	0.846	0.801*	0.811*
Recall	Conf.	1.0	1.0	1.0	0.783	0.809v	0.869v	0.838	0.779*	0.795*
	Non.	0.0	0.0	0.0	0.845	0.899v	0.857v	0.851	0.861	0.849
F-meas.	Conf.	0.667	0.667	0.667	0.807	0.845v	0.863v	0.843	0.811*	0.816*
	Non.	0.0	0.0	0.0	0.818	0.86v	0.861v	0.844	0.827*	0.826*
Eval. Metrics		Logistic Regression			Support Vector Machine					
		AW	HIW	HWB	AW	HIW	HWB			
Accuracy		0.887	0.755*	0.756*	0.886	0.898	0.891			
Precision	Conf.	0.927	0.765*	0.756*	0.908	0.945*	0.933*			
	Non.	0.863	0.759*	0.771*	0.877	0.868	0.865			
Recall	Conf.	0.845	0.749*	0.77*	0.865	0.849*	0.847*			
	Non.	0.929	0.762*	0.743*	0.908	0.947v	0.935v			
F-meas.	Conf.	0.88	0.752*	0.758*	0.882	0.891	0.885			
	Non.	0.892	0.755*	0.751*	0.889	0.903	0.896			

Tables 6-10 show the Mozilla classification results. The benefit of using high information words and bigram is not as significant as in Apache. In some cases, there is somewhat of a decrease in performance. However, as in Logistic Regression in Weka, using all words as features increases the classification time. This is especially so when the number of bug reports to predict is increasing.

Tables 6-7 are the NLTK classifiers results. As in Apache, Maxent has the worst performance when all words are used as features. Not only does it predict very poorly, but it also takes a very long time to complete (sometimes a few days). In the worst case, it has zero non-configuration F-measure. But when using HIW, F-measure increases significantly to 0.875.

Table 6. Mozilla 10 times 10-fold CV using NLTK classifiers

Eval. Metrics		Maxent			NaiveBayes			Decision Tree		
		AW	HIW	HWB	AW	HIW	HWB	AW	HIW	HWB
Accuracy		0.5	0.888v	0.878v	0.5	0.862v	0.862v	0.877	0.891v	0.88
Precision	Conf.	0.5	0.843v	0.841v	0.5	0.8v	0.82v	0.877	0.91v	0.881
	Non.	0.0	0.96v	0.935v	0.0	0.975v	0.935v	0.889	0.882	0.889
Recall	Conf.	1.0	0.965*	0.94*	1.0	0.98*	0.942*	0.885	0.872*	0.885
	Non.	0.0	0.811v	0.815v	0.0	0.745v	0.782v	0.87	0.909v	0.875
F-meas.	Conf.	0.667	0.898v	0.886v	0.667	0.879v	0.874v	0.878	0.887	0.88
	Non.	0.0	0.875v	0.868v	0.0	0.839v	0.847v	0.876	0.893v	0.879

Table 7. Mozilla 20 times 5-fold training and testing using NLTK classifiers

Eval. Metrics		Maxent			NaiveBayes			Decision Tree		
		AW	HIW	HWB	AW	HIW	HWB	AW	HIW	HWB
Accuracy		0.5	0.875v	0.895v	0.5	0.832v	0.873v	0.879	0.894v	0.881
Precision	**Conf.**	0.5	0.822v	0.852v	0.5	0.768v	0.835v	0.889	0.904v	0.884
	Non.	0.0	0.957v	0.955v	0.0	0.959v	0.929v	0.876	0.889v	0.881
Recall	**Conf.**	1.0	0.962*	0.96*	1.0	0.966*	0.935*	0.871	0.884v	0.878
	Non.	0.0	0.788v	0.829v	0.0	0.698v	0.811v	0.888	0.903v	0.883
F-meas.	**Conf.**	0.667	0.886v	0.902v	0.667	0.853v	0.881v	0.878	0.892v	0.88
	Non.	0.0	0.862v	0.886v	0.0	0.802v	0.863v	0.881	0.895v	0.881

When using Sklearn classifiers, as shown in Tables 8 and 9, using high information words and bigrams does not provide much of a benefit. Except for SVM, Logistic Regression classifier actually performs better with all words as features; Naïve Bayes and Decision Tree do not show any difference using AW and HIW/HWB. Using HIW or HWB, we generally see a time saving benefit, especially when the number of bug reports to be classified is large.

Table 8. Mozilla 10 times 10-fold CV using Sklearn classifiers

Eval. Metrics		Logistic			NaiveBayes		
		AW	HIW	HWB	AW	HIB	HWB
Accuracy		0.94	0.822*	0.926*	0.887	0.887	0.87
Precision	**Conf.**	0.96	0.816*	0.927*	0.85	0.86	0.824*
	Non.	0.928	0.84*	0.932	0.943	0.93*	0.942
Recall	**Conf.**	0.921	0.843*	0.928	0.948	0.933*	0.95
	Non.	0.959	0.802*	0.924*	0.826	0.841v	0.791*
F-meas.	**Conf.**	0.938	0.826*	0.926*	0.894	0.893	0.881
	Non.	0.942	0.817*	0.926*	0.877	0.88	0.857*
		Decision Tree			**SVM**		
Accuracy		0.872	0.875	0.877	0.732	0.926v	0.869v
Precision	**Conf.**	0.886	0.869*	0.876*	0.931	0.925	0.861*
	Non.	0.866	0.892v	0.889v	0.666	0.937v	0.89v
Recall	**Conf.**	0.859	0.887v	0.885v	0.505	0.934v	0.888v
	Non.	0.884	0.862*	0.868*	0.96	0.918*	0.85*
F-meas.	**Conf.**	0.87	0.875	0.877	0.644	0.927v	0.871v
	Non.	0.872	0.83*	0.875	0.784	0.924v	0.865v

Table 9. Mozilla 20 times 5-fold training and testing using Sklearn classifiers

Eval. Metrics		Logistic			NaiveBayes		
		AW	HIW	HWB	AW	HIB	HWB
Accuracy		0.941	0.838*	0.926*	0.885	0.887	0.88
Precision	Conf.	0.959	0.834*	0.927*	0.843	0.85	0.84
	Non.	0.927	0.851*	0.932	0.944	0.94	0.936
Recall	Conf.	0.922	0.852*	0.928	0.95	0.945	0.943
	Non.	0.959	0.827*	0.924*	0.819	0.828	0.815
F-meas.	Conf.	0.939	0.841*	0.926*	0.892	0.894	0.887
	Non.	0.942	0.837*	0.926*	0.875	0.878	0.87
		Decision Tree			SVM		
Accuracy		0.874	0.864	0.877	0.728	0.927v	0.869v
Precision	Conf.	0.883	0.855*	0.872	0.927	0.924	0.861*
	Non.	0.871	0.879	0.889v	0.658	0.933v	0.89v
Recall	Conf.	0.865	0.879v	0.885v	0.496	0.932v	0.888v
	Non.	0.883	0.848*	0.868*	0.96	0.921*	0.85*
F-meas.	Conf.	0.872	0.865	0.877	0.642	0.927v	0.871v
	Non.	0.875	0.861*	0.875	0.78	0.926v	0.865v

Using Weka classifiers, there is generally not much difference in performance when using AW compared with HIW/HWB, except for Logistic Regression which always performs better but at the cost of taking a much longer time. Classification results of Mozilla bug reports using Weka classifiers are shown in Table 10.

Table 10. Mozilla 10 times 10-fold CV using Weka classifiers

Eval. Metrics		ZeroR			NaiveBayes			Decision Tree		
		AW	HIW	HWB	AW	HIB	HWB	AW	HIW	HWB
Accuracy		0.5	0.5	0.5	0.809	0.832v	0.836v	0.89	0.873*	0.882
Precision	Conf.	0.5	0.5	0.5	0.876	0.89	0.885	0.906	0.874*	0.878*
	Non.	0.0	0.0	0.0	0.772	0.799v	0.807v	0.886	0.886	0.898v
Recall	Conf.	1.0	1.0	1.0	0.727	0.764v	0.778v	0.875	0.883	0.893v
	Non.	0.0	0.0	0.0	0.891	0.899	0.894	0.905	0.863*	0.87*
F-meas.	Conf.	0.667	0.667	0.667	0.789	0.817v	0.824v	0.887	0.875	0.882
	Non.	0.0	0.0	0.0	0.824	0.843v	0.845v	0.892	0.87*	0.88
Eval. Metrics		Logistic Regression			Support Vector Machine					
		AW	HIW	HWB	AW	HIW	HWB			
Accuracy		0.86	0.791*	0.822*	0.94	0.931	0.933			
Precision	Conf.	0.866	0.803*	0.842*	0.951	0.96	0.96			
	Non.	0.869	0.795*	0.817*	0.935	0.912*	0.915*			
Recall	Conf.	0.864	0.787*	0.803*	0.93	0.903*	0.907*			
	Non.	0.855	0.796*	0.841*	0.949	0.959	0.959			
F-meas.	Conf.	0.861	0.789*	0.818*	0.939	0.928	0.931			
	Non.	0.857	0.79*	0.825*	0.94	0.933	0.935			

Tables 11 and 12 show the classification results of MySQL bug reports using NLTK classifiers. As in the classification of Apache and MySQL bug reports, Maxent is sensitive to using AW or HIW/HWB as features. It always performs much worse when

using AW. Decision Tree performs better using HIW/HWB in the 20x5 training and testing but not statistically different in the case of 10x10 CV.

Table 11. MySQL10 times 10-fold CV using NLTK classifiers

Eval. Metrics		Maxent			NaiveBayes			Decision Tree		
		AW	HIW	HWB	AW	HIW	HWB	AW	HIW	HWB
Accuracy		0.5	0.838v	0.811v	0.867	0.818*	0.799*	0.764	0.773	0.752
Precision	**Conf.**	0.02	0.795v	0.771v	0.873	0.762*	0.756*	0.782	0.786	0.773
	Non.	0.5	0.917v	0.88v	0.877	0.926v	0.878	0.763	0.774	0.748*
Recall	**Conf.**	0.001	0.926v	0.895v	0.87	0.941v	0.9	0.743	0.759v	0.728*
	Non.	1.0	0.751v	0.727v	0.865	0.695*	0.702*	0.785	0.787	0.777
F-meas.	**Conf.**	0.003	0.852v	0.826v	0.867	0.84*	0.818*	0.756	0.767	0.744
	Non.	0.667	0.819v	0.791v	0.866	0.788*	0.774*	0.769	0.776	0.757

Table 12. MySQL 20 times 5-fold training and testing using NLTK classifiers

Eval. Metrics		Maxent			NaiveBayes			Decision Tree		
		AW	HIW	HWB	AW	HIW	HWB	AW	HIW	HWB
Accuracy		0.504	0.817v	0.804v	0.855	0.799*	0.798*	0.735	0.766v	0.757v
Precision	**Conf.**	0.068	0.771v	0.761v	0.856	0.742*	0.746*	0.753	0.784v	0.776v
	Non.	0.505	0.892v	0.872v	0.867	0.909v	0.889v	0.726	0.758v	0.745v
Recall	**Conf.**	0.046	0.91v	0.894v	0.863	0.93v	0.915v	0.708	0.745v	0.725v
	Non.	0.962	0.723*	0.715*	0.846	0.668*	0.681*	0.761	0.787v	0.789v
F-meas.	**Conf.**	0.036	0.833v	0.821v	0.855	0.824*	0.82*	0.726	0.76v	0.747v
	Non.	0.647	0.795v	0.783v	0.852	0.764*	0.768*	0.74	0.769v	0.764v

In the case of Sklearn classifiers, again SVM is more sensitive than the others when using AW as compared to using HIW/HWB. It always performs worse with AW. Again, Logistic Regression works better with AW at the cost of time. Decision Tree does not show much difference using either AW or HIW/HWB, while there is slightly worse performance in Naïve Bayes with HIW/HWB.

Table 13. MySQL 10 times 10-fold CV using Sklearn classifiers

Eval. Metrics		Logistic			NaiveBayes		
		AW	HIW	HWB	AW	HIB	HWB
Accuracy		0.882	0.785*	0.86*	0.864	0.853	0.816*
Precision	**Conf.**	0.911	0.808*	0.876*	0.827	0.819*	0.778*
	Non.	0.865	0.778*	0.856	0.925	0.908*	0.885*
Recall	**Conf.**	0.851	0.759*	0.847	0.931	0.915*	0.898*
	Non.	0.913	0.811*	0.873*	0.797	0.791	0.734*
F-meas.	**Conf.**	0.877	0.777*	0.858*	0.874	0.862	0.83*
	Non.	0.886	0.79*	0.861*	0.852	0.841	0.797*
Eval. Metrics		**Decision Tree**			**SVM**		
		AW	HIW	HWB	AW	HIW	HWB
Accuracy		0.736	0.737	0.752	0.819	0.861v	0.823
Precision	**Conf.**	0.763	0.773	0.771	0.872	0.901v	0.908v
	Non.	0.723	0.721	0.747v	0.808	0.836v	0.778*
Recall	**Conf.**	0.697	0.683	0.728	0.771	0.817v	0.724*
	Non.	0.774	0.791v	0.776	0.867	0.905v	0.921v
F-meas.	**Conf.**	0.723	0.719	0.743	0.805	0.854v	0.798
	Non.	0.743	0.75	0.757	0.825	0.867v	0.84v

Table 14. MySQL 20 times 5-fold training and testing using Sklearn classifiers

Eval. Metrics		Logistic			NaiveBayes		
		AW	HIW	HWB	AW	HIB	HWB
Accuracy		0.88	0.769*	0.849*	0.864	0.835*	0.814*
Precision	Conf.	0.906	0.783*	0.855*	0.826	0.816	0.768*
	Non.	0.862	0.765*	0.849*	0.919	0.866*	0.892*
Recall	Conf.	0.851	0.753*	0.845	0.928	0.873*	0.909*
	Non.	0.909	0.785*	0.852*	0.799	0.798	0.719*
F-meas.	Conf.	0.876	0.764*	0.848*	0.873	0.841*	0.831*
	Non.	0.883	0.772*	0.849*	0.853	0.828*	0.792*
Eval. Metrics		Decision Tree			SVM		
		AW	HIW	HWB	AW	HIW	HWB
Accuracy		0.735	0.74	0.735	0.811	0.848v	0.792*
Precision	Conf.	0.757	0.77v	0.756	0.864	0.888v	0.897v
	Non.	0.725	0.722	0.723	0.802	0.816	0.739*
Recall	Conf.	0.7	0.691	0.701	0.763	0.798v	0.667*
	Non.	0.77	0.789	0.769	0.859	0.897v	0.918v
F-meas.	Conf.	0.724	0.725	0.725	0.795	0.839v	0.759*
	Non.	0.744	0.752	0.743	0.818	0.855v	0.816

In the case of Weka classifiers, again SVM performs better using HIW/HWB while Logistic Regression and Naïve Bayes are better with AW, as shown in Table 15.

Table 15. MySQL 10 times 10-fold CV using Weka classifiers

Eval. Metrics		ZeroR			NaiveBayes			Decision Tree		
		AW	HIW	HWB	AW	HIB	HWB	AW	HIW	HWB
Accuracy		0.5	0.5	0.5	0.876	0.822*	0.721*	0.754	0.752	0.743
Precision	Conf.	0.5	0.5	0.5	0.887	0.806	0.67*	0.755	0.77v	0.771v
	Non.	0.0	0.0	0.0	0.878	0.855v	0.842v	0.768	0.748*	0.734*
Recall	Conf.	1.0	1.0	1.0	0.872	0.859	0.893v	0.767	0.731*	0.704*
	Non.	0.0	0.0	0.0	0.881	0.785*	0.549*	0.741	0.772v	0.782v
F-meas.	Conf.	0.667	0.667	0.667	0.876	0.827*	0.763*	0.756	0.745	0.729*
	Non.	0.0	0.0	0.0	0.876	0.813*	0.656*	0.749	0.755	0.752
Eval. Metrics		Logistic Regression			Support Vector Machine					
		AW	HIW	HWB	AW	HIW	HWB			
Accuracy		0.89	0.735*	0.73*	0.867	0.851	0.871			
Precision	Conf.	0.882	0.74*	0.734*	0.903	0.852*	0.896			
	Non.	0.908	0.745*	0.74*	0.844	0.864v	0.86v			
Recall	Conf.	0.907	0.742*	0.737*	0.827	0.86v	0.845v			
	Non.	0.872	0.728*	0.722*	0.907	0.843*	0.897			
F-meas.	Conf.	0.892	0.736*	0.729*	0.861	0.852	0.866			
	Non.	0.886	0.73*	0.725*	0.872	0.849*	0.875			

Overall, the NLTK classifiers are more sensitive to using AW or HIW/HWB than the NLTK and the Weka classifiers. This is especially true with the NLTK classifier Maxent. NLTK classifiers perform better when using high information words and bigrams. In addition, bug reports from different projects seem to respond differently to

AW or HIW/HWB. Apache bug reports are generally classified better when using HIW/HWB rather than using AW for all three classifiers (NLTK, Sklearn and Weka). In Weka classifiers, Logistic Regression generally performs better with AW, while in Sklearn SVM performs better with HIW/HWB. Even when a classifier does not perform well with HIW/HWB, the performance difference is not large compared to AW. However, when a classifier does not perform well with AW, the performance can be very bad, as can be seen in all three types of bug reports when classified with Maxent using AW.

One of the reasons that lead to the not so good performance in some cases when using HIW/HWB may be due to the fact that in some bug reports (e.g. MySQL) there is too much extraneous information such as the execution error results. These execution error results are certainly helpful for developers to investigate, but they have many repetitive words that are not helpful in identifying the type of bug report; unfortunately, the high frequency of these words makes them highly informative words to the classifier.

The other likely reason is that the highly informative words do not occur predominantly in one class (configuration) versus the other class (non-configuration). So for example, even though words such as "set" or "value" do appear in the first 100 high informative words produced from chi-sq in MySQL, they are not used, or are on the lower level of the list by the classifiers as key features for identifying configuration bug reports. The reason they make it into the 100 high informative words is likely due to the reason they appear more frequently in some reports, which makes their count high. In NLTK, once a word is selected as a feature, it is treated equally as other features, regardless of how often it appears in some reports. This can be understood by the explanation of how to represent features in NLTK in section 3.2.1, "experiment design",

where all features selected have the value of "TRUE". With these selected features, it is up to the classifier to decide which features are the important ones for determining a report as configuration type or not. In the collected bug reports, it is likely that in the training bug reports, the words "set" and "value" do not appear in enough bug reports to merit being treated as important features, so they do not appear at all in Table 16. On the contrary, even though the word "preference" is not high on the list of chi-sq scores in Mozilla (as shown in Table 16), it becomes high on the list of features selected by classifiers as shown in Figure 9. This is likely because "preference" appears in the majority of configuration bug reports. Thus, even though there are not many words that we consider very informative in Table 16 for Mozilla, the result in Mozilla is good. Note that Table 16 lists the features ordered in their chi-sq scores, not the order from the classifiers' informative feature list. Figure 9 shows the first few most informative features identified by NLTK NaiveBays for Mozilla bug reports. The ratios on the right indicate how likely a feature is to appear in configuration bug reports vs. in non-configuration bug reports.

Table 16. Some Most Informative Words In Mozilla, Apache And MySQL Bug Reports That Are Used By Classifiers

Mozilla		Apache		MySQL	
crash	8375.5	configuration	542.4	option	253.4
build	1675.2	module	200.3	global	212.7
talkback	736.3	conf	196.4	configuration	208.1
reproducible	676.0	directive	175.8	cnf	171.2
identifier	553.7	enable	170.2	usr	136.2
option	442.9	src	112.3	ref	93.4
agent	376.5	xindice	99.0	connector	81.9
preference	357.3	ssl	95.0	socket	80.0
code	272.8	configure	45.8	sock	68.2

```
Most Informative Features:
                talkback = True              noncon : config =     13.0 : 1.0
              preference = None              noncon : config =     10.6 : 1.0
              preference = True              config : noncon =      8.8 : 1.0
                    text = True              config : noncon =      5.9 : 1.0
                  change = None              noncon : config =      5.8 : 1.0
                   crash = True              noncon : config =      5.2 : 1.0
                     pub = True              noncon : config =      4.3 : 1.0
                  debian = True              noncon : config =      4.3 : 1.0
                inbound = True               noncon : config =      4.3 : 1.0
             identifier = True               noncon : config =      4.3 : 1.0
           reproducible = True               noncon : config =      4.3 : 1.0
             unexpected = True               config : noncon =      4.2 : 1.0
                  think = None               noncon : config =      3.9 : 1.0
               password = True               config : noncon =      3.8 : 1.0
                 player = True               noncon : config =      3.7 : 1.0
                   tree = True               config : noncon =      3.7 : 1.0
                  build = None               config : noncon =      3.1 : 1.0
                  agent = True               noncon : config =      3.1 : 1.0
                   code = True               config : noncon =      3.1 : 1.0
                 option = True               config : noncon =      3.1 : 1.0
```

Figure 9. The first few most informative features identified by NLTK NaiveBayes in Mozilla bug reports.

Table 17 shows the mean of configuration and non-configuration F-measures using the six classifiers. These numbers are the average of the F-measures in Tables 1-15. As we can see, performance does not change in ZeroR regardless of which method is used, and it is also the worst classifier since its non-configuration F-measures are all 0s. For the other five classifiers, with the exception of Logistic Regression, all other classifiers perform better using HIW/HWB compared to using AW. Maxent makes the greatest improvement when the feature extraction method is changed from AW to HIW, with non-configuration F-measure value doubled. The classifier that benefits the second most from using HIW/HWB is SVM, with an increase of 24% in configuration F-measure. Logistic Regression actually does not perform as well when using HIW/HWB. However, it takes much longer time to finish when using AW rather than HIW/HWB. Overall, using HIW/HWB improves the performance of a classifier.

Table 17 also reveals that most classifiers perform much better than ZeroR which is the baseline classifier used for comparison. The F-measure values of these classifiers,

with the exception of Maxent with AW, are mostly more than 0.8. This implies that the classification of bug reports as configuration or non-configuration is effective.

Table 17. Average configuration and non-configuration F-measures of the five classifiers

	Config. F-measure			Nonconfig. F-measure		
	AW	HIW	HWB	AW	HIW	HWB
ZeroR	0.667	0.667	0.667	0	0	0
Maxent	0.472	0.868	0.862	0.42	0.845	0.84
NaiveBayes	0.827	0.857	0.85	0.714	0.837	0.823
DecisionTree	0.806	0.818	0.817	0.816	0.824	0.824
Logistic	0.888	0.795	0.852	0.893	0.796	0.854
SVM	0.706	0.878	0.832	0.816	0.89	0.862

When comparing the three software packages, Weka is the most consistent regardless of using AW or HIW/HWB, while NLTK is the most sensitive to these methods (AW, HIW, HWB). However, NLTK classifiers can perform very well when the right method (HIW/HWB) is used.

Of all the classifiers (not considering ZeroR), Logistic Regression and Maxent appear to be the slowest classifiers, especially when using AW. Naïve Bayes is usually very fast, and its performance sometimes is better than Decision Tree. However, in some cases, it is moderately sensitive to the method used. SVM is also sensitive to the method used.

The Figures A1-A15 in Appendix compare F-measures among the different classifiers. We can see how the F-measure values of the 100 classifications vary. In Tables 1-15, differences less than 0.1 (most of the time around 0.02 and 0.03) are considered statistically different. In the figures, the differences such as these are not so easily discernable. The most obvious differences between AW and HIW/HWB are NLTK classification results on Apache (Figures A1-A4, NLTK classifiers and some Sklearn

classifiers on Apache). When a classifier performs poorly, most of the time we can see that the data points vary widely.

3.3.2 Identification of a configuration associated with a configuration bug report

In section 3.3.1, in order to evaluate the performance of classification and how it can be generalized, 10 times 10-fold CV and a similar validation procedure are carried out. In this section, the steps in Figure 3 are followed. First, for each bug report corpus 200 bug reports (100 configuration and 100 non-configuration) are used for training and the rest are used as unlabeled bug reports for prediction (testing). After a bug report is predicted to be configuration-related, it is used to identify its associated configurations.

For configuration option identification, all the configuration bug reports that are used in testing are considered. There are 50 configuration bug reports used in testing in each project. In Table 18, the accuracy ratio is calculated by the number of correct identifications to the total number of configuration bug reports, which is 50. For Mozilla, those configuration bug reports are already identified on Mozilla configuration website (http://kb.mozillazine.org/Category:Preferences), so that information is used as ground truth to test the accuracy of the configuration identification. For bug reports from the other two projects, the association of a bug report with a configuration or a few configurations is manually identified and is verified to be the correct identification.

Since a bug report could be associated with more than one configuration, the configuration identification tool will present the first 10 most likely configurations. This helps in the case that when it is not possible to identify the exact configuration, it is still possible to narrow down the number of configurations. This will help a developer who

works on the bug since he/she still gets relevant bug reports to consider rather than none at all. If the already known to be related configuration occurs in the first 10 listed configurations, it is considered a correct identification in this study. Although this type of identification is loose, it is found that the tool developed in this study can identify the correct configurations in the top five most of the time. In fact, it is not uncommon to see that it finds the correct configuration as the first one on the list.

As discussed in section 2.3.2.2, Term Frequency-Inverse Document Frequency (TFIDF) is used to measure the importance of a term in a document. This tool makes use of TFIDF to calculate the similarity of a bug report and the configurations. The configurations are ordered according to similarity score from highest to lowest. The similarity is defined as:

$$similarity(b, d) = \sum_{t \in b} tf - idf_{t,d}$$

where *b* is the bug report and *d* is the configuration.

Table 18 is an example of the tf-idf calculations the tool makes in identifying the configuration associated with the bug report in Figure 2. In this example, the configuration is browser.urlbar.filter.javascript. Its similarity to the example bug report breaks down to the individual tf-idf scores of the five terms. Adding them together, we get 0.16+0.18+0.15+0.14=0.64 which turns out to be the highest score of all other configurations. Thus, the tool correctly identifies this configuration as the number one on the list of all possible configurations associated with the example bug report.

Table 18. Ranking terms in the example configuration option

term	tf	df	idf	tf-idf
browser	1	28	0.16	0.16
url	1	31	0.18	0.18
bar	0	-	-	-
filter	1	18	0.15	0.15
javascript	1	22	0.14	0.14
…	…	…	…	…

Results in Table 19 show that overall, this configuration identification tool is effective in finding out which configuration is associated with the configuration bug report under investigation. This is particularly true in Mozilla, with a high value of 0.92. The tool performs worst in MySQL. This is mainly due to the fact that MySQl configurations have irregular number of words in the configurations. For example, it has configurations: "ssl", "ssl_accepts", "ssl_accept_renegotiates" in server configurations. These configurations vary in size significantly. In these cases, our tool tends to choose the longer configuration if the words "accept" and "renegotiate" also appear in the bug report. In the collected bug reports, there are a few cases that the bug reports are associated with "ssl", and our tool identified the longer ones. Since an identification is considered to be correct only if the correct configuration is in the first 10 of the identified ones, those are all considered incorrect identifications. Correct identification of such configurations is a direction for future improvement of the tool, where it should not only consider a bag of words but may also consider semantic of the text.

Table 19. Accuracy of relating a configuration bug report with a configuration

	Mozilla	Apache	MySQL
Accuracy	0.92	0.88	0.74

Chapter 4

Conclusions and Future Work

This chapter draws conclusions based on the results and analyses discussed in Chapter 3 and outlines future work to improve classification and configuration identification performances.

4.1 Conclusions

In this study, a tool is developed that can classify configuration bug reports and extract configuration options. It involves two steps. The first step trains classification models on the labeled bug reports to predict a given unlabeled bug report as being either a configuration or non-configuration bug report. The second step employs natural language processing and information retrieval to extract configuration options from the identified configuration bug reports. A total of 900 bug reports from three open source projects are used for the study. The results show that the approach adopted in the study discriminates configuration bug reports from non-configuration bug reports with high accuracy, and that it is effective at extracting configuration options.

The study also compares three machine learning software packages in classifying bug reports. NLTK's classifiers are more sensitive to the different methods used to extract features; using high information words and bigrams as features works better in most NLTK classifiers used in this study. Sklearn classifiers are moderately affected by the methods used; however, its SVM classifier performs much better with HIW/HWB than AW. Weka classifiers do not show much difference using AW or HIW/HWB; in fact,

its Logistic Regression classifier works much better with AW but takes much longer time to complete.

The classifiers are also compared. ZeroR has no prediction power and is used as baseline classifier. Maxent is very sensitive to the method used; when using HIW/HWB it performs its best and the time spent in classification is much shorter than using AW. SVM is also sensitive to the methods used, especially the one implemented in Sklearn. Naïve Bayes is somewhat affected by the methods used, in particular the NLTK Naïve Bayes; but generally its performance is quite good, sometimes better than Decision Tree and the time spent is generally shorter than Decision Tree. Logistic Regression generally performs best in terms of the performance metric numbers, but it is rather slow. In general, all the classifiers perform much better than the baseline ZeroR. Exceptions are in Maxent and NaiveBayes when they are used to predict Mozilla bug reports using AW. In that case, their prediction ability seems to deteriorate to that of ZeroR, blindly labeling all bug reports to be one type.

4.2 Future Work

The current research shows promising results in using NLP and machine learning techniques to characterize configuration bug reports. However, there are improvements to make to the current characterization framework, and this will be the future work.

We saw that on Apache bug reports, using HIW/HWB improved the performances of the classifiers significantly, especially the NLTK classifiers, while there was not much difference using AW or HIW/HWB on MySQL bug reports. Study of the bug reports revealed that MySQL bug reports contained much information that was

irrelevant to the type of the bug reports; however, the irrelevant words occurred many times, which made them appear to be important and thus they entered into the high information words list. This irrelevant information is considered one of the possible reasons that lead to the little to no improvement in classification using HIW/HWB. Thus, one part of the future work will be to clean the bug reports first before performing any of the operations in the current research.

In this research, for each open source project, only 135 bug reports of each type are used for training and 15 bug reports of each type are used for testing in the 10-fold cross validation, and in the 20x5 training and testing validation, 120 bug reports of each type are used for training and 30 bug reports of each type are used for testing. Although these numbers are considered enough for machine learning [8], generally the more data there are the better. In addition, considering the potential great differences in words and phrases used in one bug report as compared to those in another, using more bug reports for training and testing will capture more information, improve the classifiers' performance and make the testing results more convincing. This will be another part of the future research work. To make this part of the work more conclusive, it may need to study how the classifiers' performance metrics vary as the numbers of the bug reports for training and testing are increased. It is possible there is a maximum in the number of bug reports beyond which further increase in bug reports does not improve classifiers' performance much. If that number can be found, then there is no need to look for more bug reports to improve classification performance, and we are confident that the results we have reflect the true performance of the classifiers.

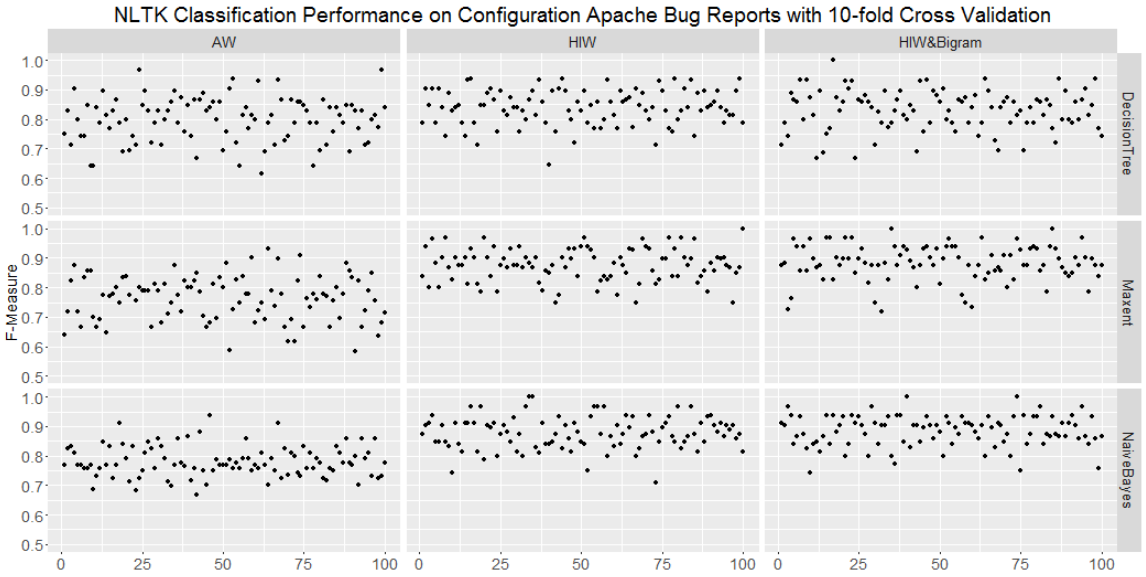
In addition to varying the number of the bug reports, it is also necessary to include the bug reports from more software projects, both open software and proprietary, to make the results generalizable.

The configuration identification part of the current research has its limitations. As discussed in 3.3.2, for configurations with varying words, the configuration identification tool tends to choose the configuration with more words, even though the shorter one is the correct configuration. Improvement on this may involve considering the sematic of the bug reports, synonyms and using n-gram. The n in n-gram is greater than two since unigram and bigram have already been used.

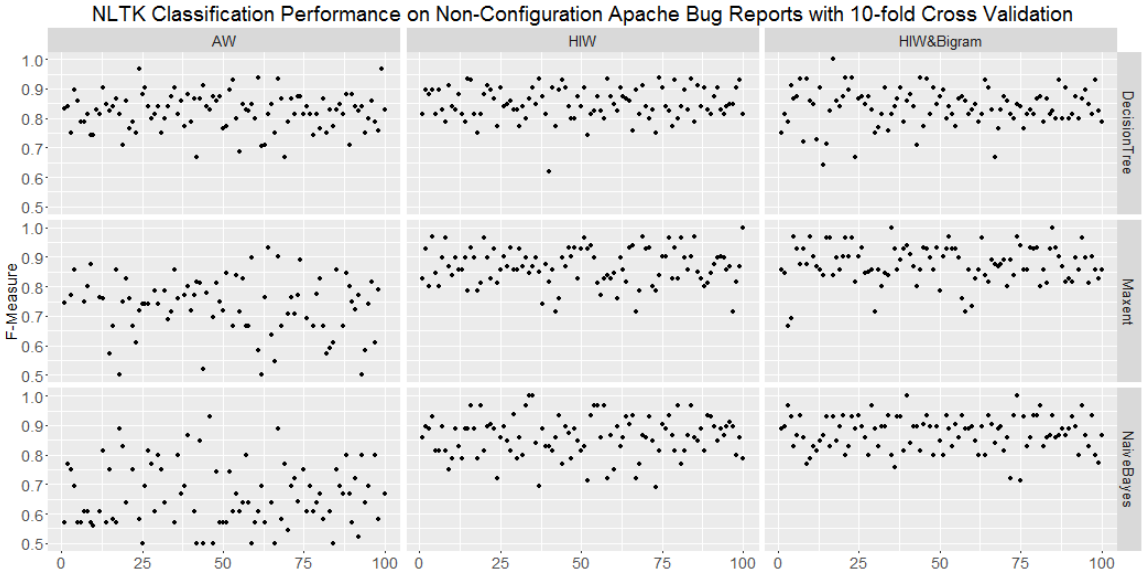
The last part of the future work is to combine the currently discrete parts of the code into a fully integrated software piece. These parts include the code to generate pure text from the bug reports URLs, the code to perform classification, and the code to calculate Tfidf values of the configuration bug reports and output the most closely associated configurations, etc. It would be desirable to include Weka classifiers in the python code rather than in a separate Java code. This involves fully understanding the current difficulties in calling Weka classifiers from the python code and finding an alternative solution. To make the integrated tool more user friendly, it can provide the user with the option to choose which classifier(s) he/she wants to use for classification.

Appendix

Figures that compare the configuration and non-configuration F-measure between the classifiers

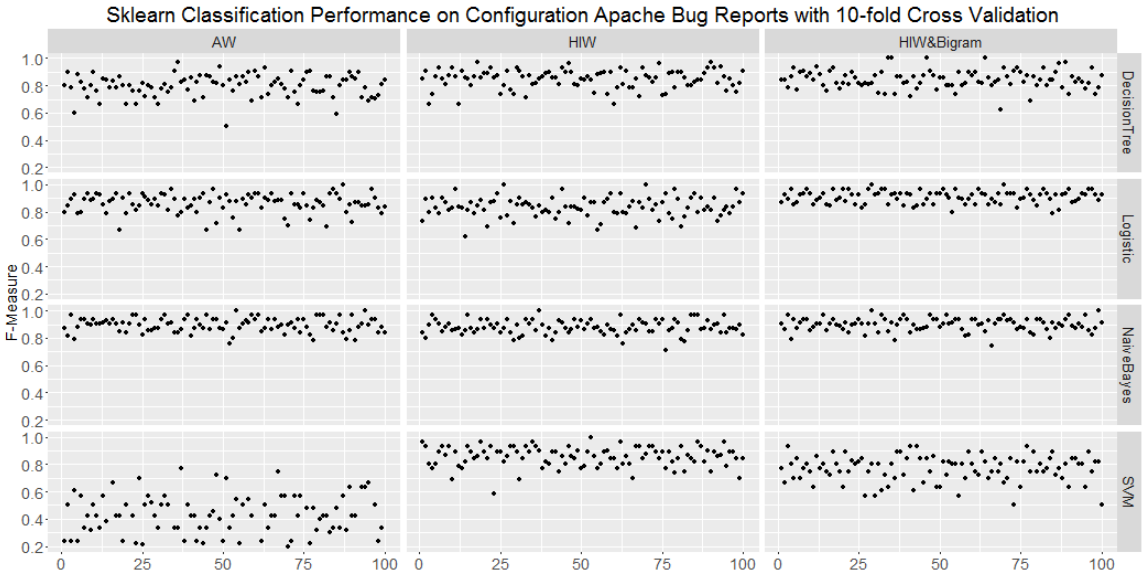


(a) Configuration F-measure

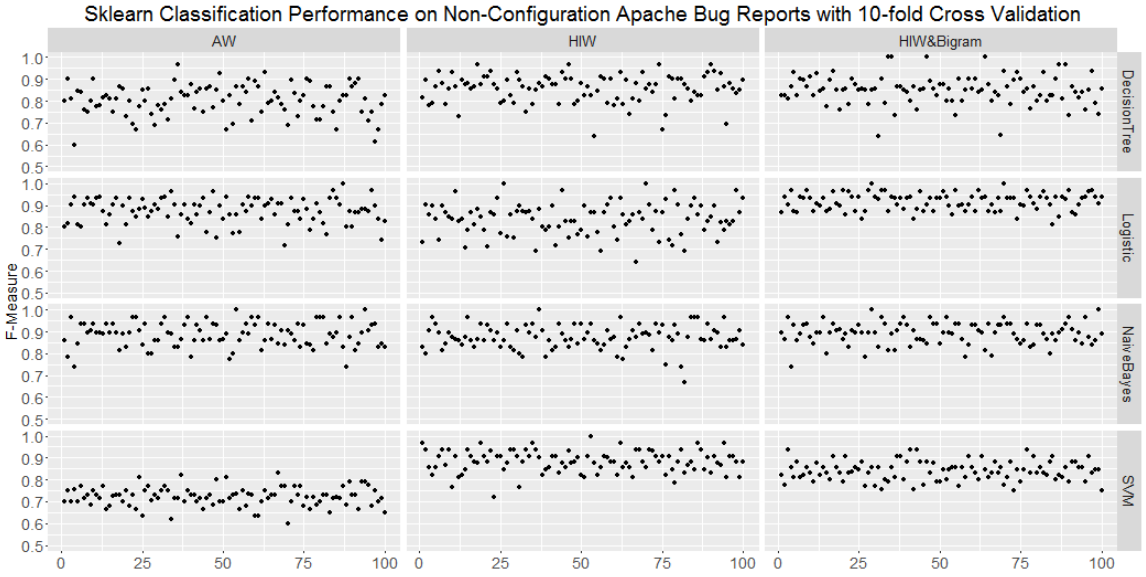


(b) Non-configuration F-measure

Figure A1. Apache F-measure comparison with 10x10 CV using NLTK.

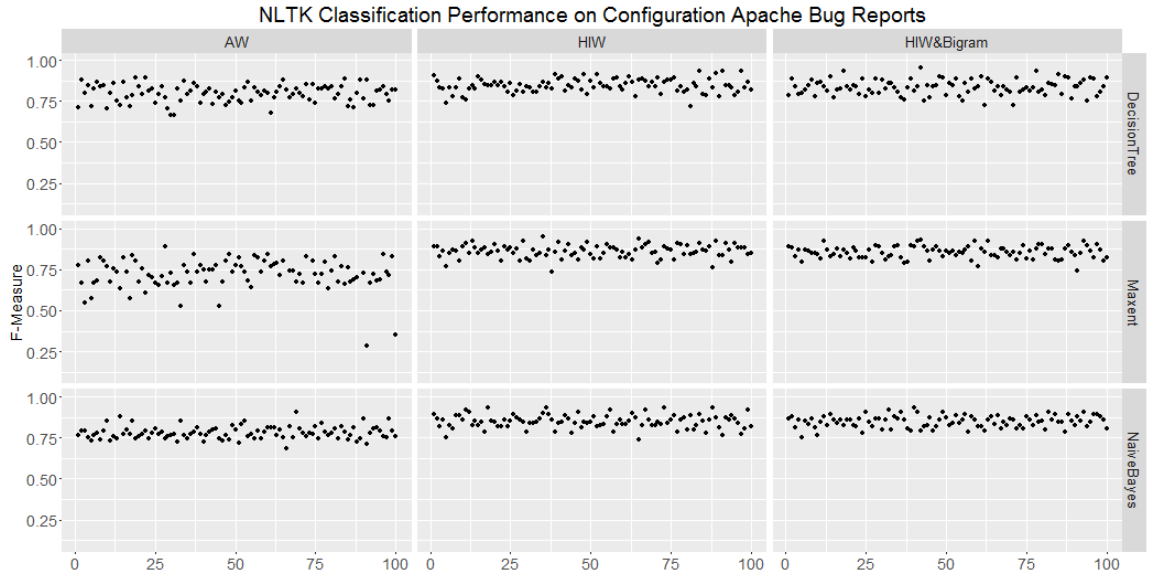


(a) Configuration F-measure

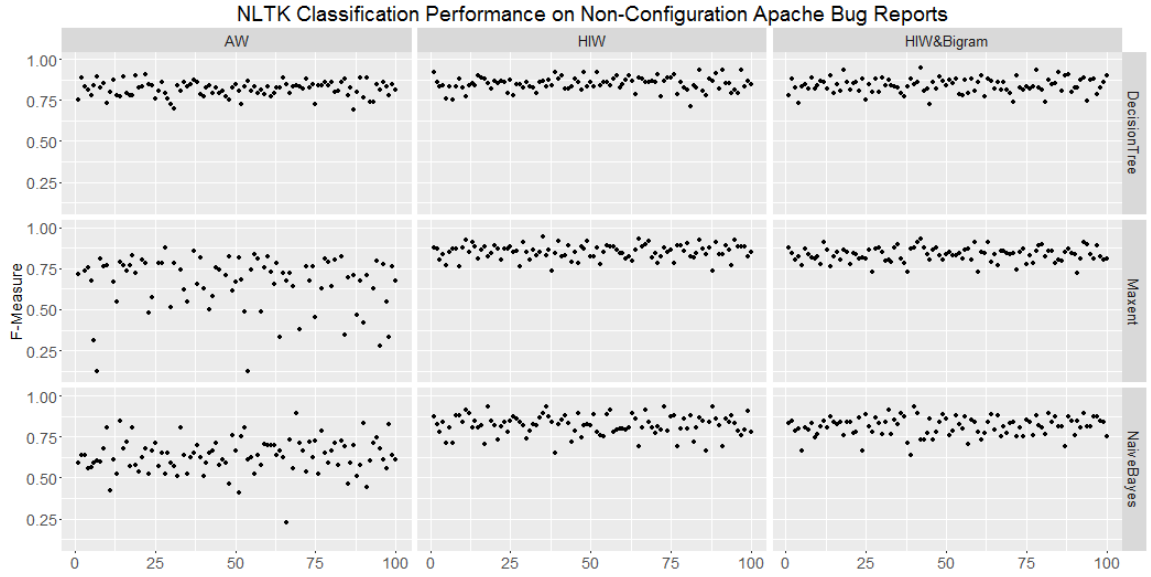


(b) Non-configuration F-measure

Figure A2. Apache F-measure comparison with 10x10 using Sklearn

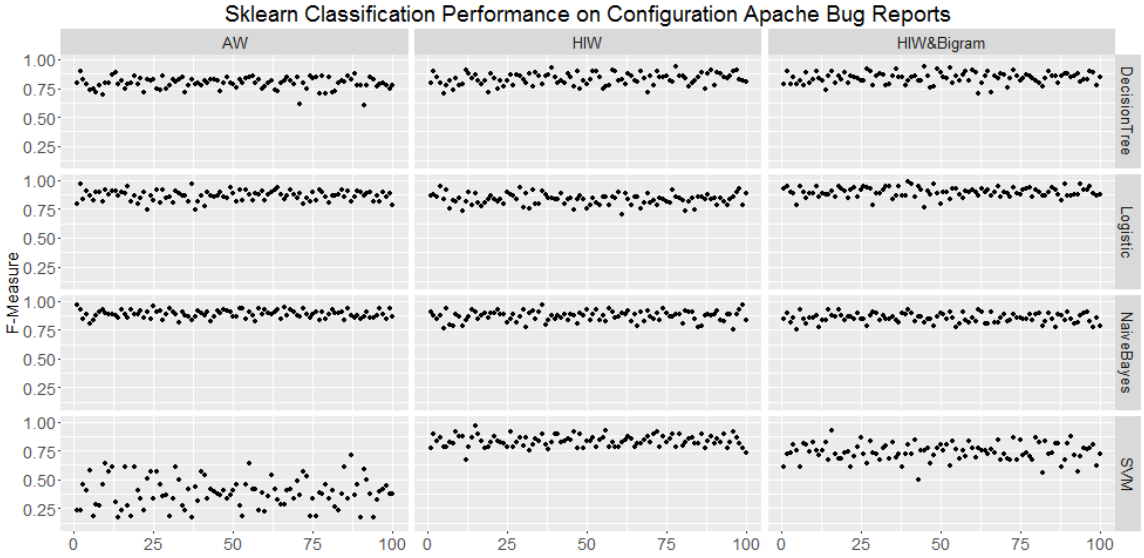


(a) Configuration F-measure

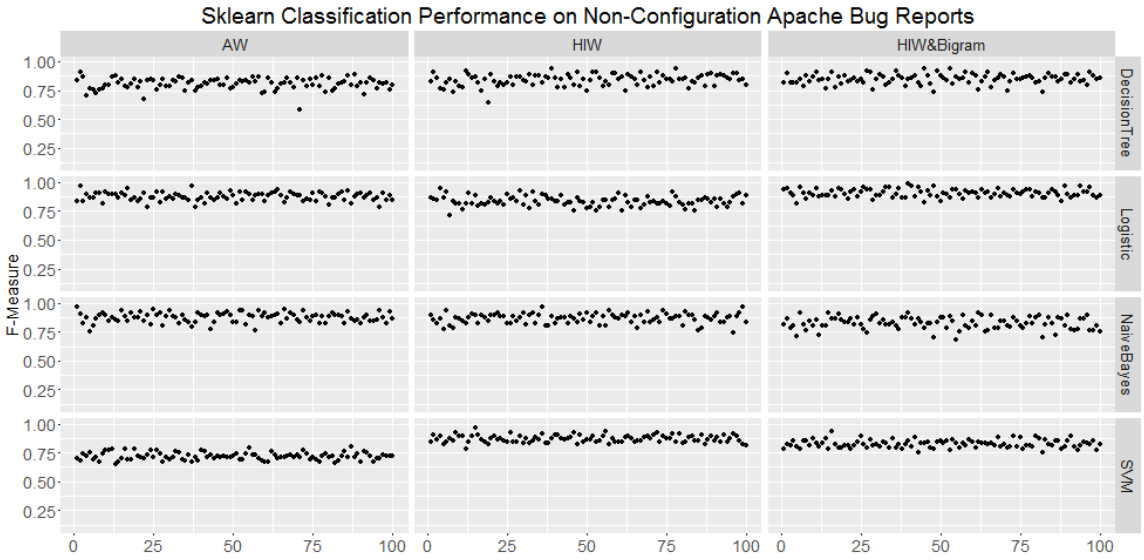


(b) Non-configuration F-measure

Figure A3. Apache F-measurement comparison with 20x5 training-and-testing using NLTK

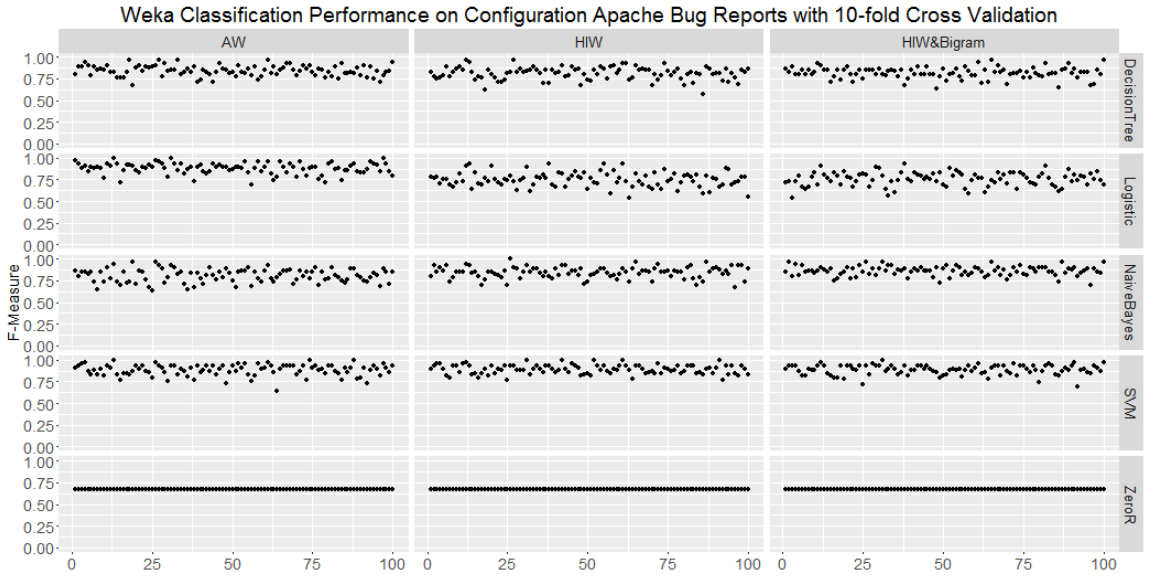


(a) Configuration F-measure

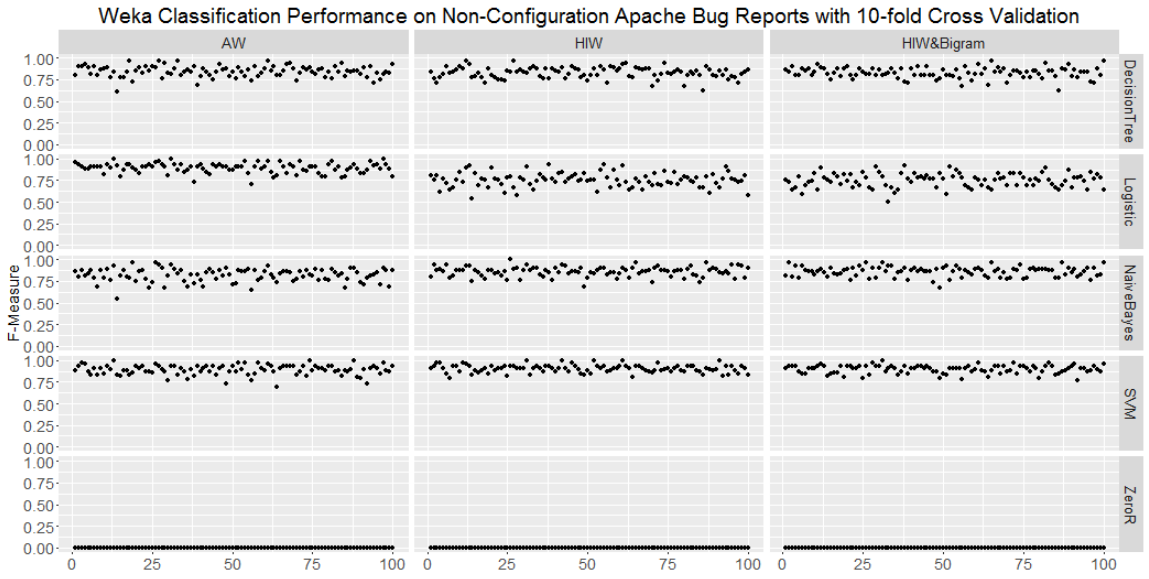


(b) Non-configuration F-measure

Figure A4. Apache F-measurement comparison with 20x5 training-and-testing using Sklearn.

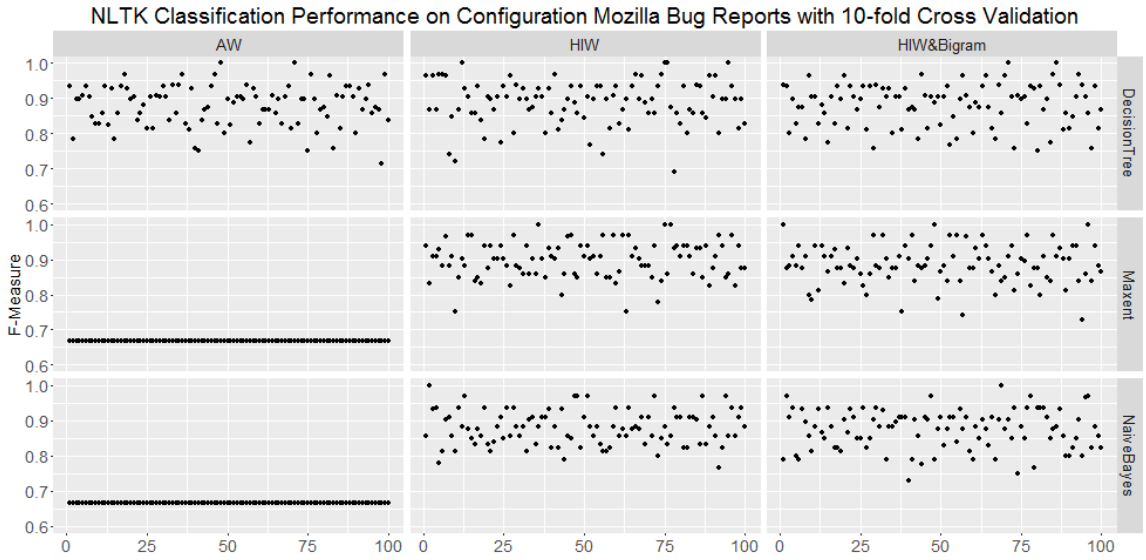


(a) Configuration F-measure

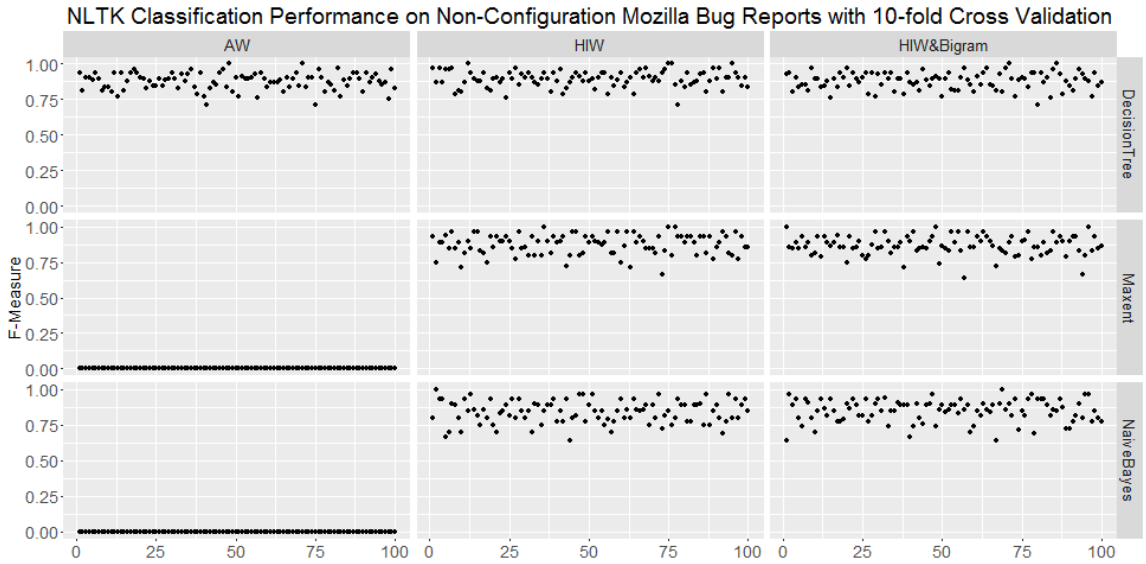


(b) Non-configuration F-measure

Figure A5. Apache F-measure comparison with 10x10 CV using Weka.

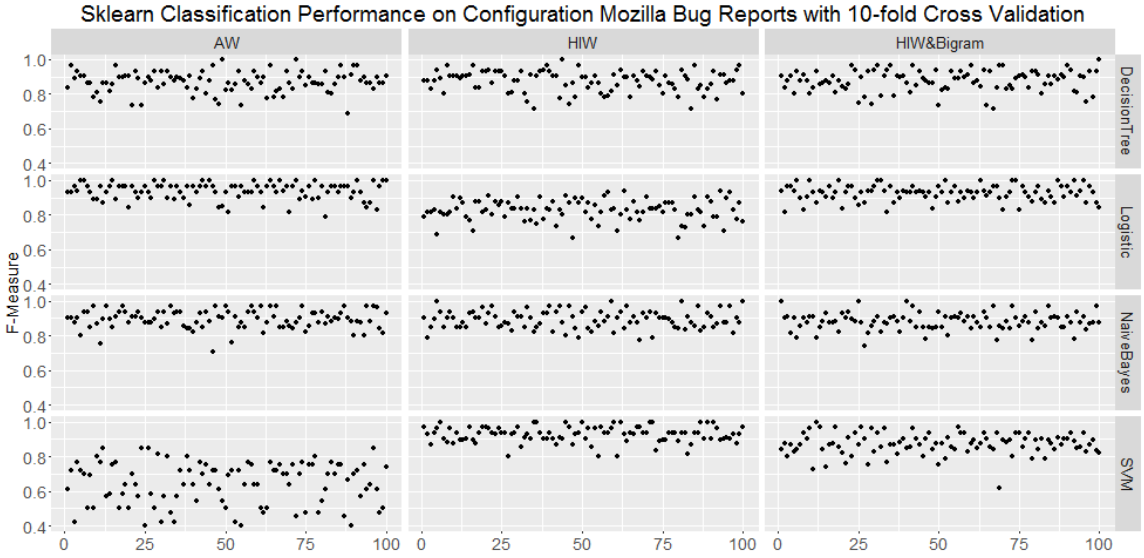


(a) Configuration F-measure

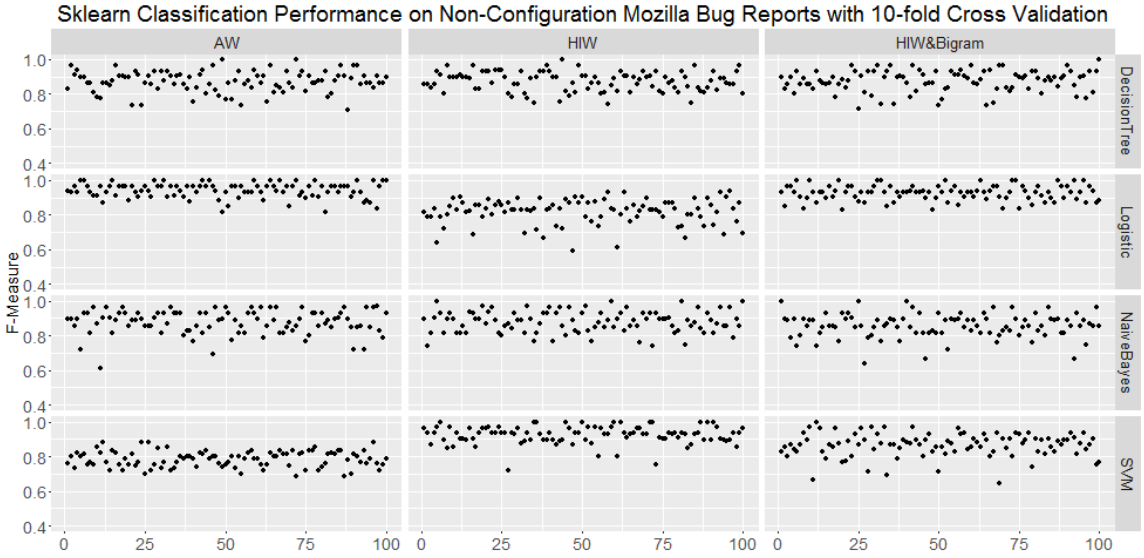


Non-configuration F-measure

Figure A6. Mozilla F-measure comparison with 10x10 CV using NLTK

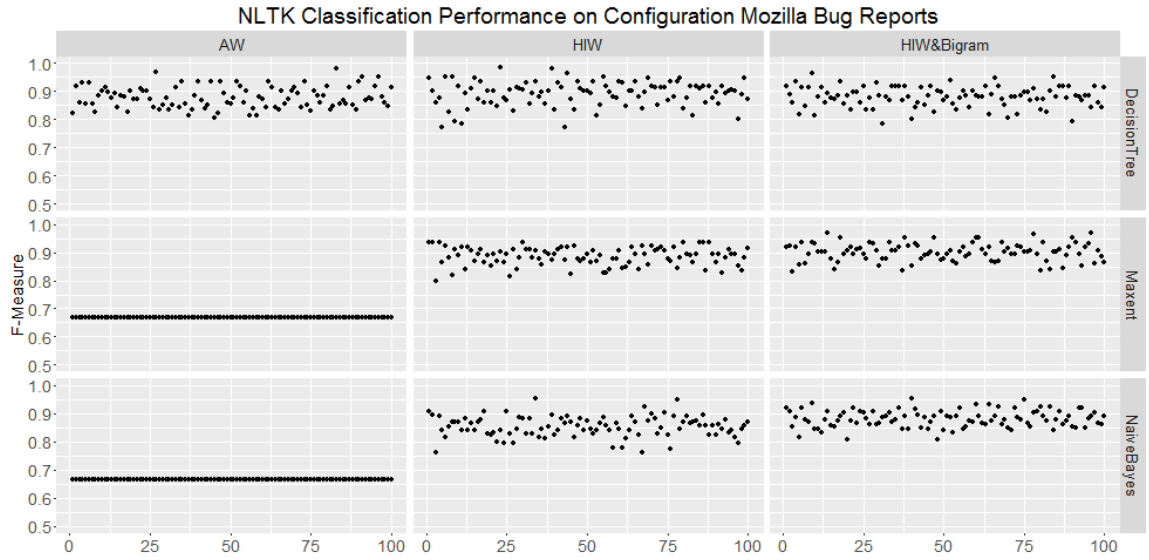


(a) Configuration F-measure

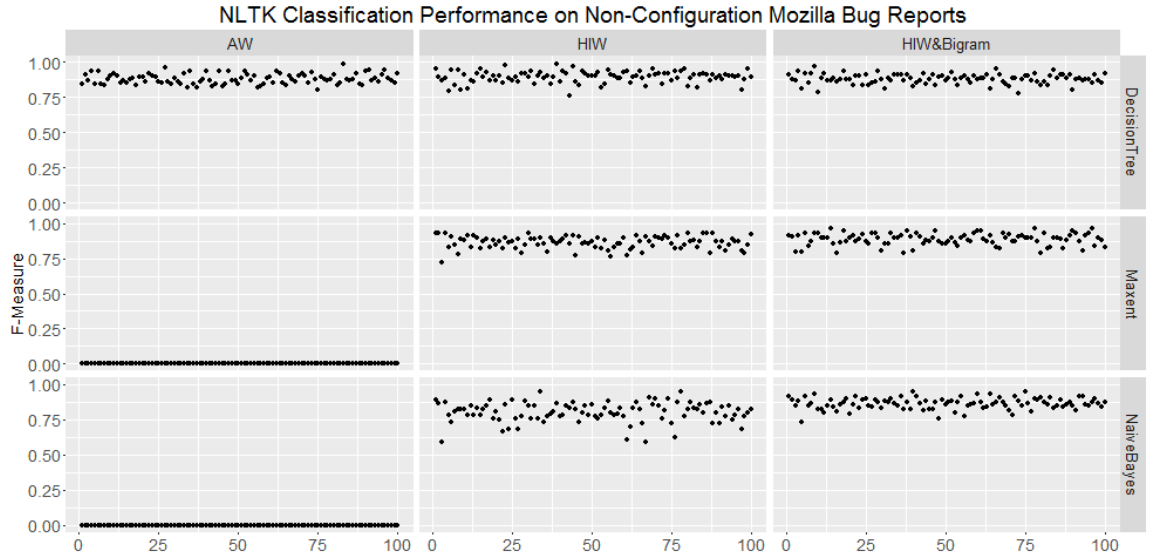


(b) Non-configuration F-measure

Figure A7. Mozilla non-configuration F-measure comparison with 10x10 CV using Sklearn.

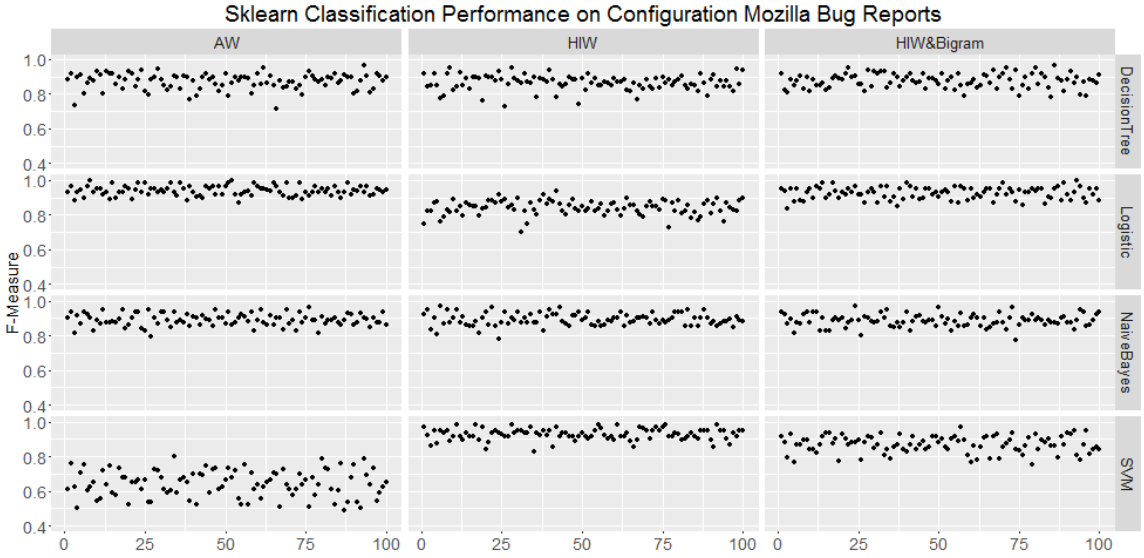


(a) Configuration F-measure

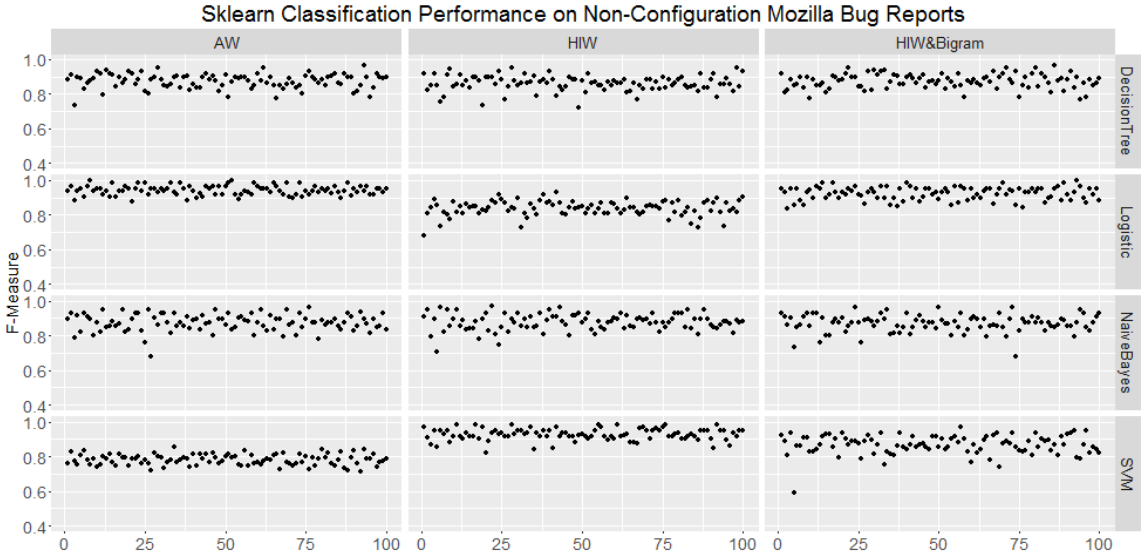


(b) Non-configuration F-measure

Figure A8. Mozilla F-measure comparison with 20x5 training-and-testing using NLTK.

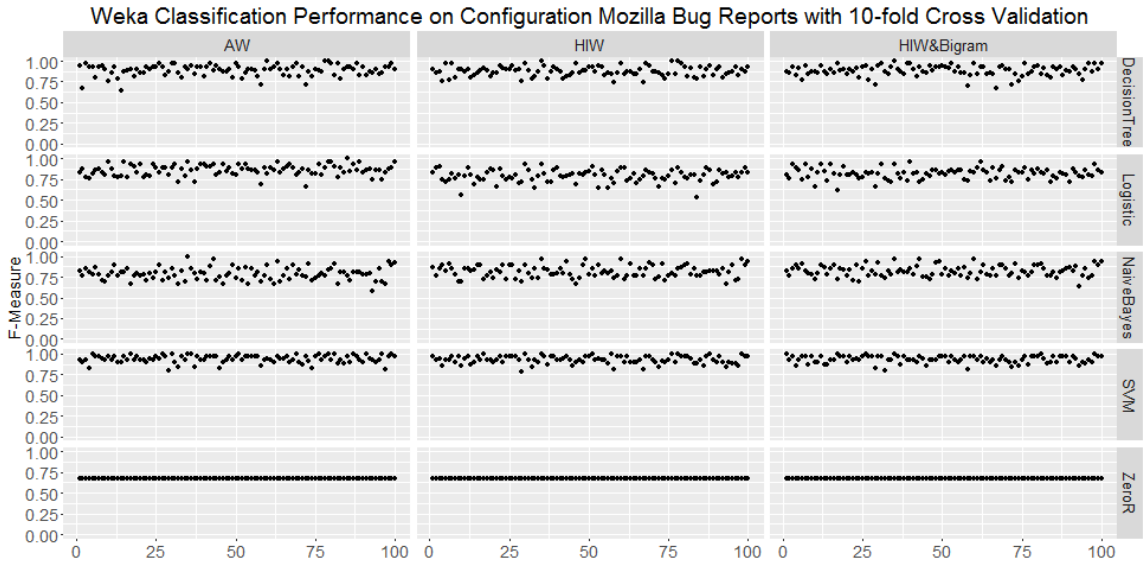


(a) Configuration F-measure

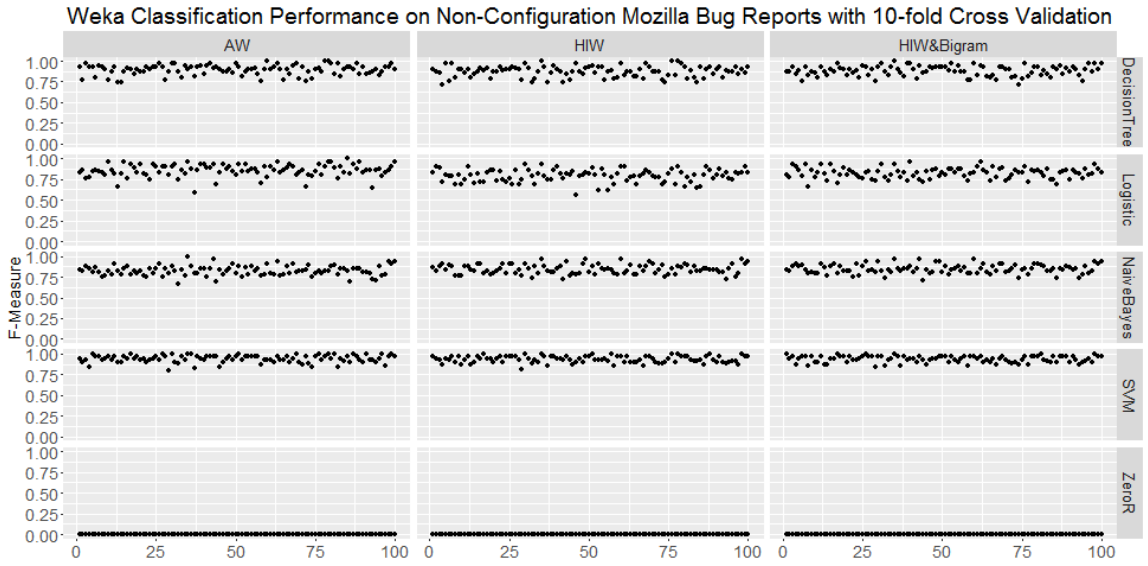


(b) Non-configuration F-measure

Figure A9. Mozilla F-measure comparison with 20x5 training-and-testing using Sklearn.

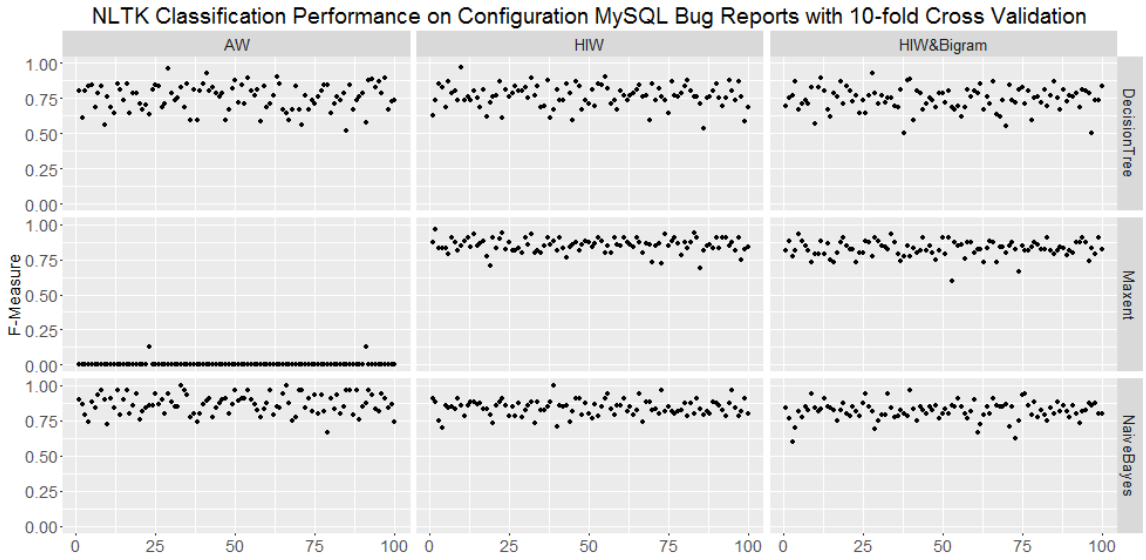


(a) Configuration F-measure

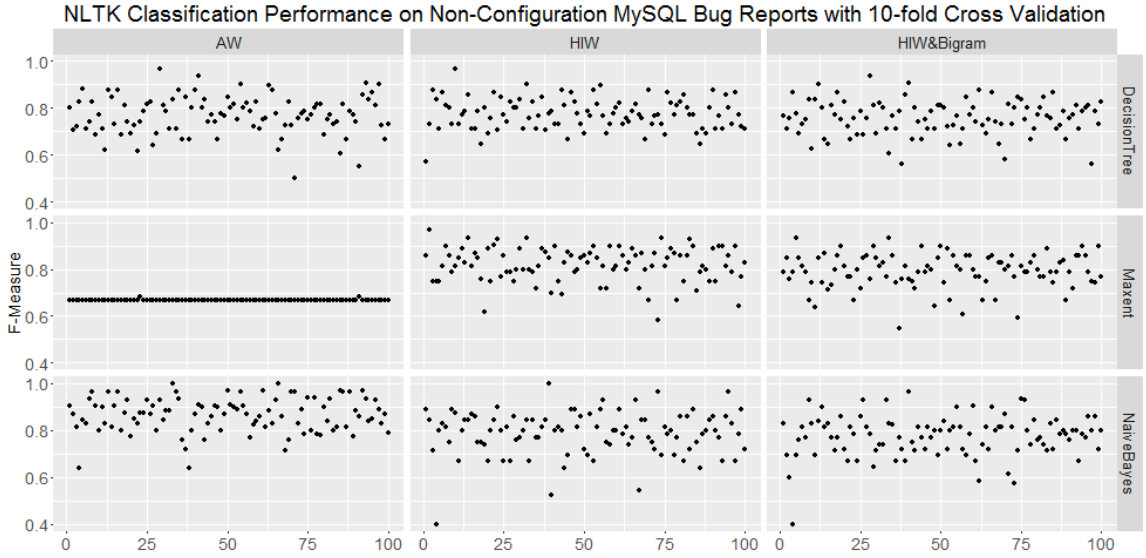


(b) Non-configuration F-measure

Figure A10. Mozilla F-measure comparison with 10x10 CV using Weka.

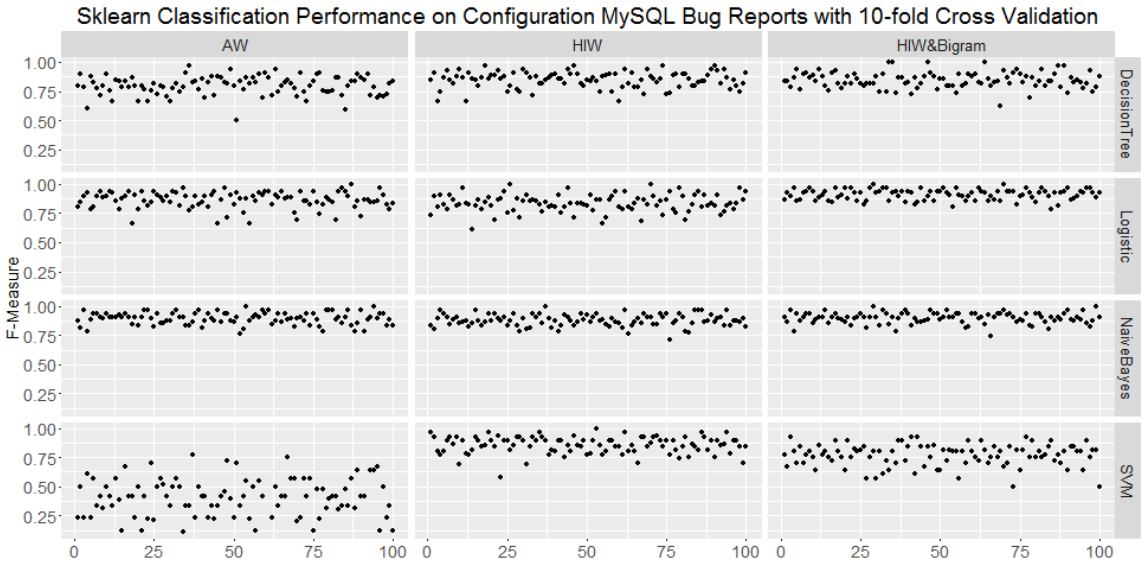


(a) Configuration F-measure

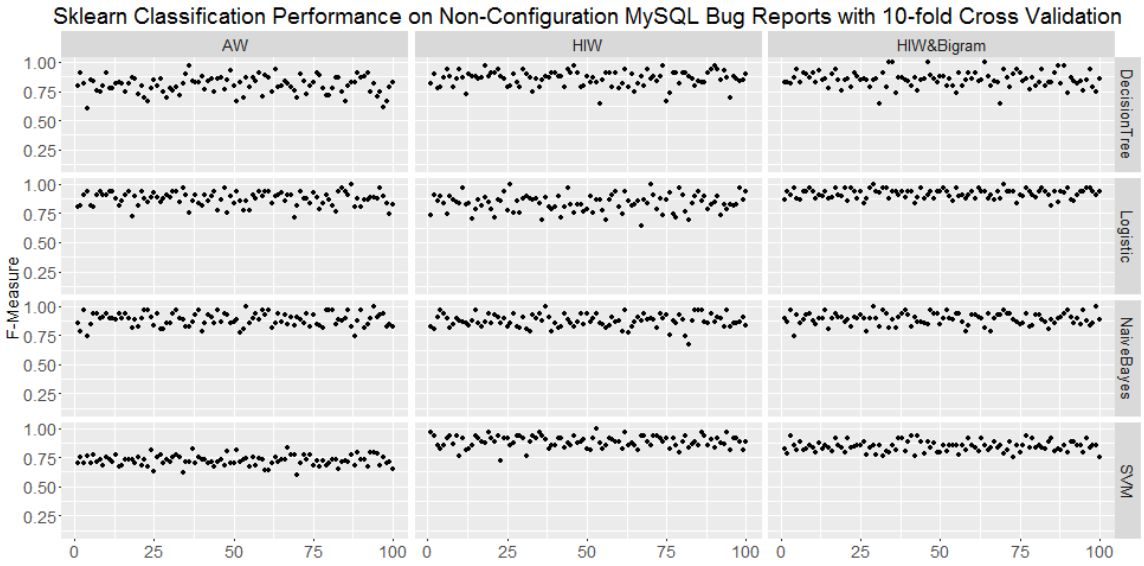


(b) Non-configuration F-measure

Figure A11. MySQL F-measure comparison with 10x10 CV using NLTK.

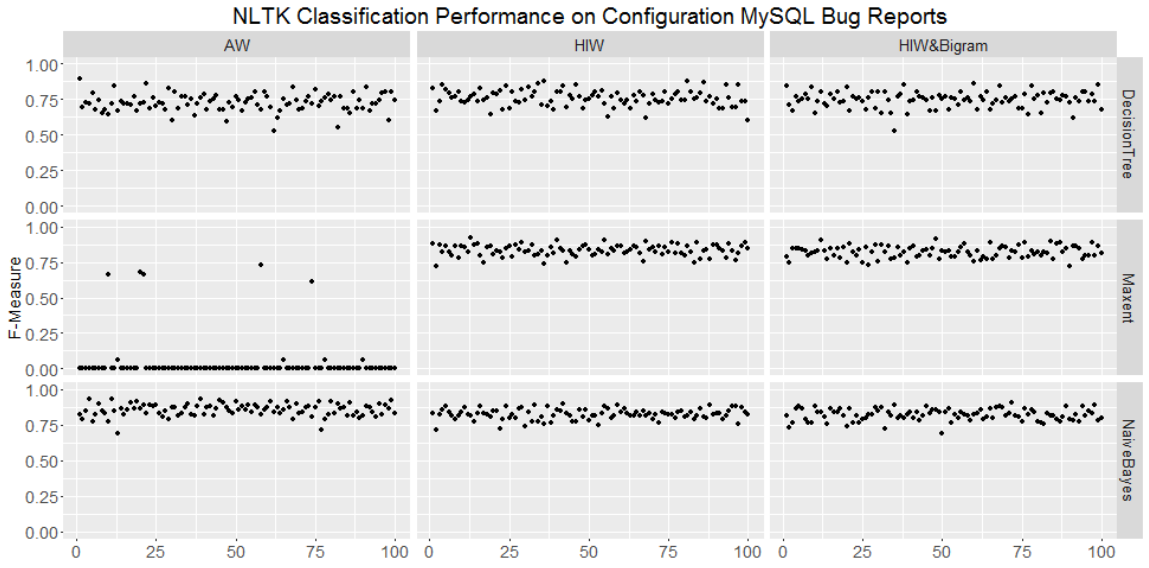


(a) Configuration F-measure

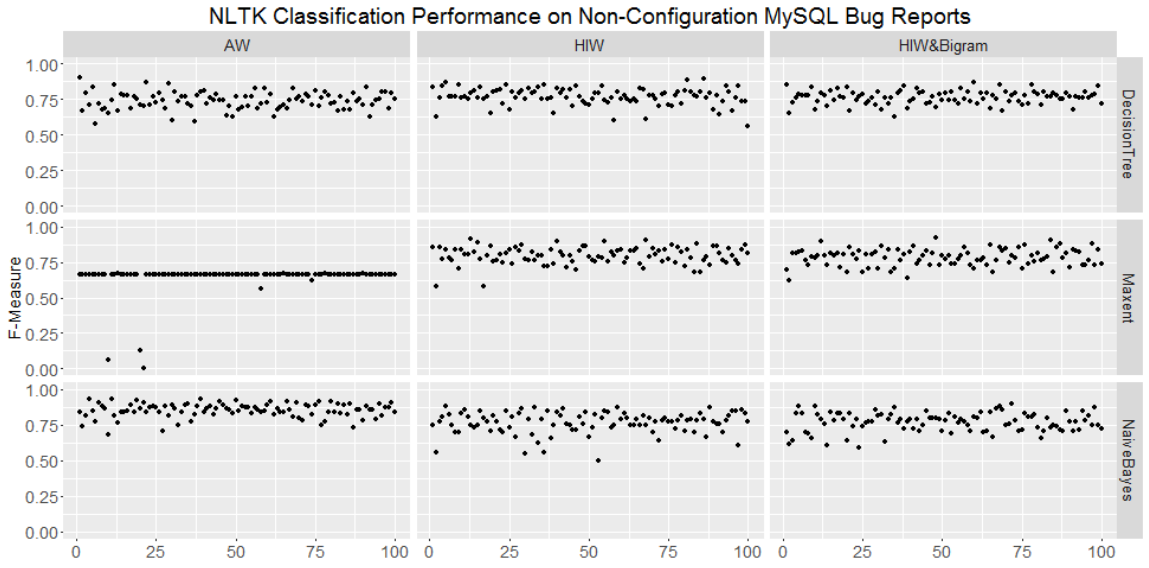


(b) Non-configuration F-measure

Figure A12. MySQL F-measure comparison with 10x10 CV using Sklearn.

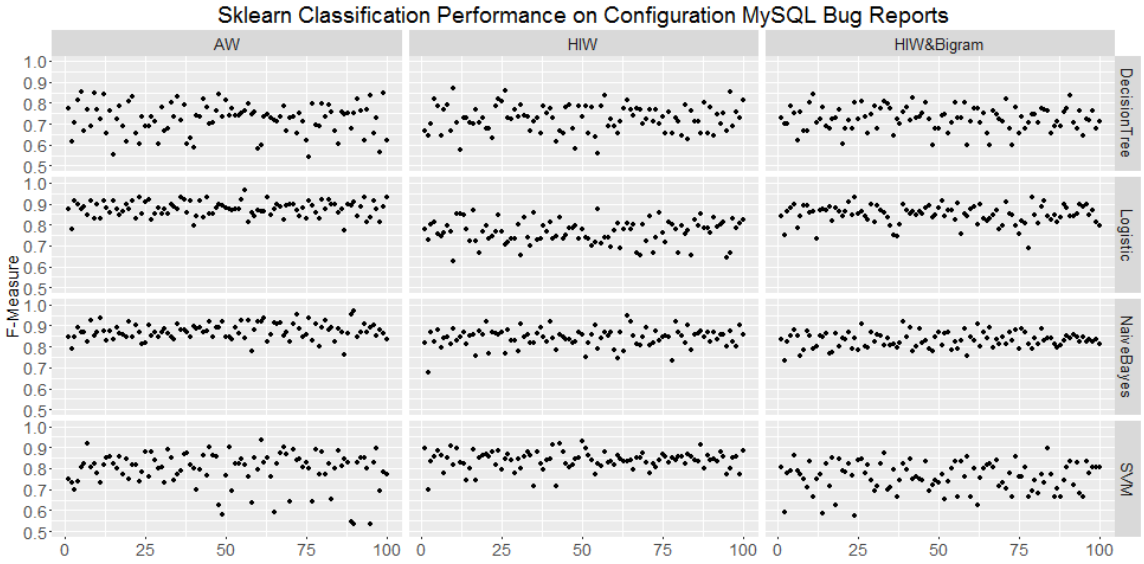


(a) Configuration F-measure

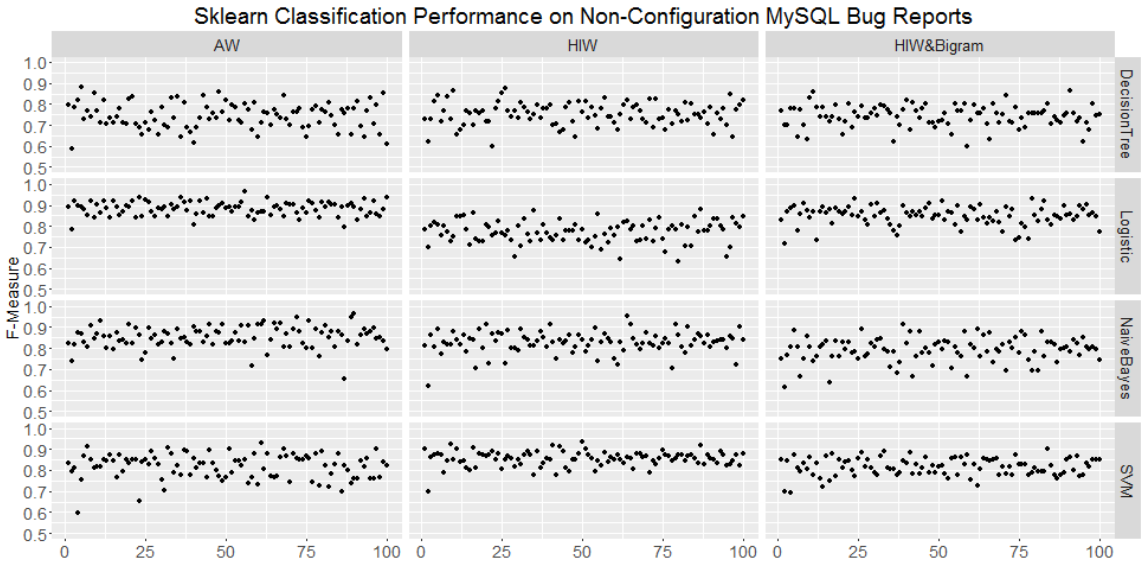


(b) Non-configuration F-measure

Figure A13. MySQL F-measurement comparison with 20x5 training-and-testing using NLTK

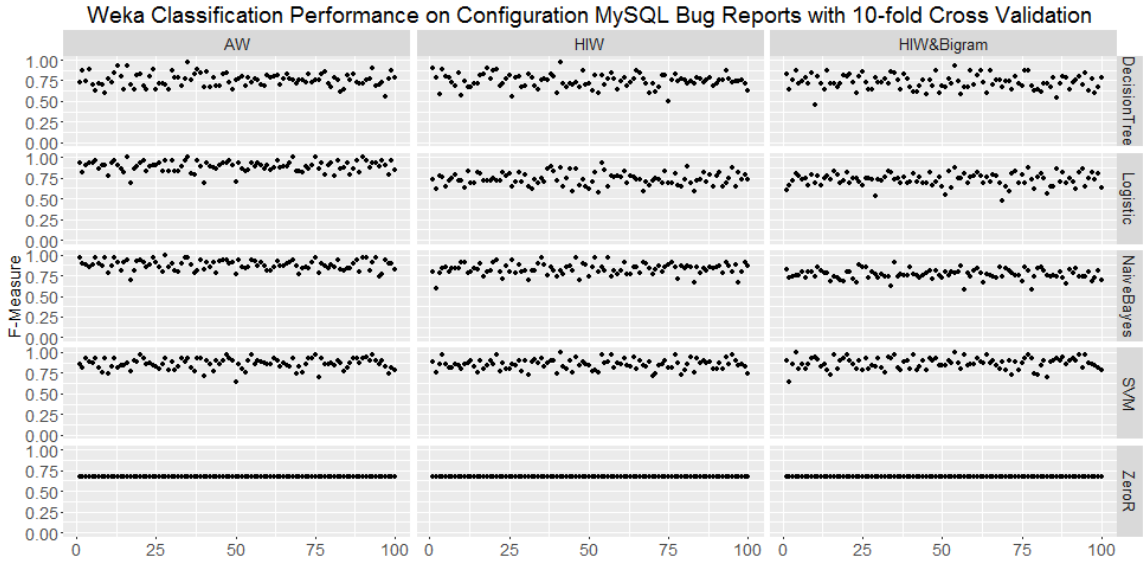


(a) Configuration F-measure

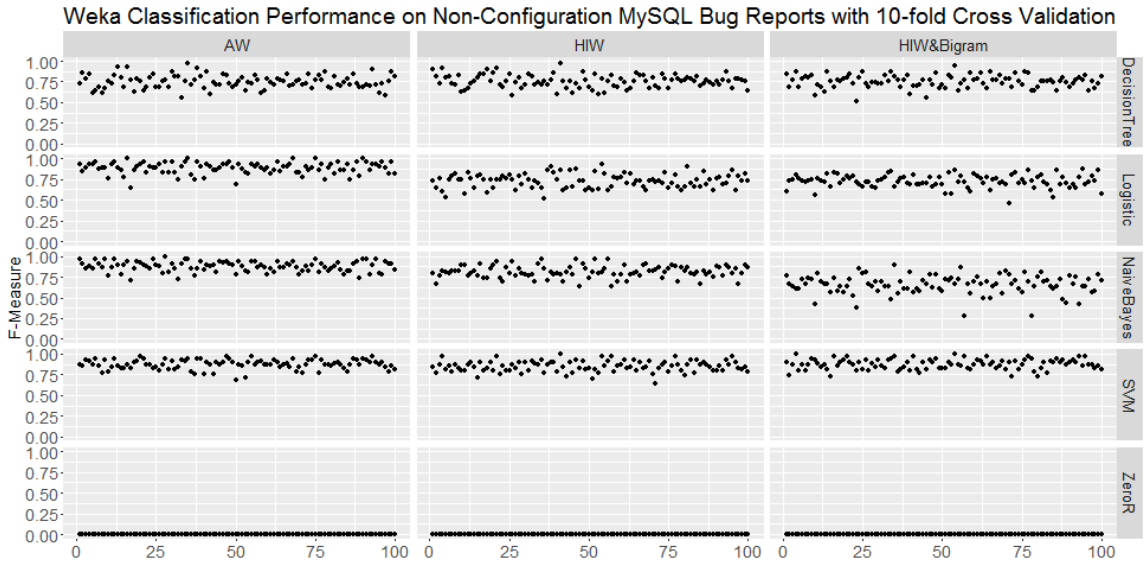


(b) Non-configuration F-measure

Figure A14. MySQL F-measurement comparison with 20x5 training-and-testing using Sklearn.



(a) Configuration F-measure



(b) Non-configuration F-measure

Figure A15. MySQL F-measure comparison with 10x10 CV using Weka.

References

1. Smith, B. "An approach to graphs of linear forms." *Referencia de un trabajo no publicado), sin publicar* (1982).
2. Yin, Zuoning, et al. "An empirical study on configuration errors in commercial and open source systems." *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011.
3. Rastkar, Sarah, Gail C. Murphy, and Gabriel Murray. "Automatic summarization of bug reports." *IEEE Transactions on Software Engineering*40.4 (2014): 366-380.
4. Dommati, Sunil Joy, Ruchi Agrawal, and S. Sowmya Kamath. "Bug Classification: Feature Extraction and Comparison of Event Model using Naive Bayes Approach." *arXiv preprint arXiv:1304.1677* (2013).
5. Wang, Xiaoyin, et al. "An approach to detecting duplicate bug reports using natural language and execution information." *Proceedings of the 30th international conference on Software engineering*. ACM, 2008.
6. Padberg, Frank, Philip Pfaffe, and Martin Blersch. "On Mining Concurrency Defect-Related Reports from Bug Repositories."
7. Kim, Dongsun, et al. "Where should we fix this bug? a two-phase recommendation model." *IEEE transactions on software Engineering* 39.11 (2013): 1597-1610.
8. Gegick, Michael, Pete Rotella, and Tao Xie. "Identifying security bug reports via text mining: An industrial case study." *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, 2010.

9. Rastkar, Sarah, Gail C. Murphy, and Gabriel Murray. "Automatic summarization of bug reports." *IEEE Transactions on Software Engineering*40.4 (2014): 366-380
10. Sureka, Ashish. "Learning to classify bug reports into components." *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Springer Berlin Heidelberg, 2012.
11. Matter, Dominique, Adrian Kuhn, and Oscar Nierstrasz. "Assigning bug reports using a vocabulary-based expertise model of developers." *2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE, 2009.
12. Briand, Lionel C., Yvan Labiche, and Xuetao Liu. "Using machine learning to support debugging with tarantula." *The 18th IEEE International Symposium on Software Reliability (ISSRE'07)*. IEEE, 2007.
13. Zimmermann, Thomas, Rahul Premraj, and Andreas Zeller. "Predicting defects for eclipse." *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on*. IEEE, 2007.
14. Lamkanfi, Ahmed, et al. "Predicting the severity of a reported bug." *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, 2010.
15. Turhan, Burak, Gozde Kocak, and Ayse Bener. "Data mining source code for locating software bugs: A case study in telecommunication industry." *Expert Systems with Applications* 36.6 (2009): 9986-9990.
16. Chowdhury, Gobinda G. "Natural language processing." *Annual review of information science and technology* 37.1 (2003): 51-89.
17. Grefenstette, Gregory, and Pasi Tapanainen. "What is a word, what is a sentence?: problems of Tokenisation." (1994): 79.

18. Wilbur, W. John, and Karl Sirotkin. "The automatic identification of stop words." *Journal of information science* 18.1 (1992): 45-55.
19. Ahmed, Shabbir, and Farzana Mithun. "Word Stemming to Enhance Spam Filtering." *CEAS*. 2004.
20. Brill, Eric. "Part-of-speech tagging." *Handbook of natural language processing* (2000): 403-414.
21. Plisson, Joël, Nada Lavrac, and Dunja Mladenic. "A rule based approach to word lemmatization." *Proceedings C of the 7th International Multi-Conference Information Society IS 2004*. Vol. 1. No. 1. 2004.
22. Arellano, Andres, Edward Carney, and Mark A. Austin. "Natural Language Processing of Textual Requirements." *The Tenth International Conference on Systems (ICONS 2015), Barcelona, Spain*. 2015.
23. Fukumizu, Kenji, Francis R. Bach, and Michael I. Jordan. "Dimensionality reduction for supervised learning with reproducing kernel Hilbert spaces." Journal of Machine Learning Research 5.Jan (2004): 73-99.
24. Gawade, Trunal. "Feature Extraction using Text mining." *International Journal Of Emerging Technology and Computer Science* 1.2 (2016).
25. Lee, Changki, and Gary Geunbae Lee. "Information gain and divergence-based feature selection for machine learning-based text categorization." *Information processing & management* 42.1 (2006): 155-165.
26. Liu, Huan, and Rudy Setiono. "Chi2: Feature selection and discretization of numeric attributes." *ICTAI*. 1995.

27. Tan, Chade-Meng, Yuan-Fang Wang, and Chan-Do Lee. "The use of bigrams to enhance text categorization." *Information processing & management* 38.4 (2002): 529-546.
28. Sebastiani, Fabrizio. "Machine learning in automated text categorization." *ACM computing surveys (CSUR)* 34.1 (2002): 1-47.
29. Kotsiantis, Sotiris B., I. Zaharakis, and P. Pintelas. "Supervised machine learning: A review of classification techniques." (2007): 3-24.
30. Gentleman, R., and V. J. Carey. "Unsupervised machine learning." *Bioconductor Case Studies*. Springer New York, 2008. 137-157.
31. Sutton, Richard S., and Andrew G. Barto. *Reinforcement learning: An introduction*. Vol. 1. No. 1. Cambridge: MIT press, 1998.
32. Murphy, Kevin P. "Naive bayes classifiers." *University of British Columbia*(2006).
33. Safavian, S. Rasoul, and David Landgrebe. "A survey of decision tree classifier methodology." (1990).
34. Hosmer, David W., and Stanley Lemeshow. "Introduction to the logistic regression model." *Applied Logistic Regression, Second Edition* (2000): 1-30.
35. Liu, Ting, et al. "Semantic role lableing system using maximum entropy classifier." *Proceedings of the Ninth Conference on Computational Natural Language Learning*. Association for Computational Linguistics, 2005.
36. Witten, Ian H., et al. "Weka: Practical machine learning tools and techniques with Java implementations." (1999).
37. Wang, Fu, Jiazheng Xu, and Zhide Liang. "Maximum Entropy Method." *Textures and Microstructures* 19 (1992): 55-58.

38. Mount, John. "The equivalence of logistic regression and maximum entropy models." *URL: http://www. win-vector. com/dfiles/LogisticRegressionMaxEnt. pdf* (2011).
39. Bird, Steven. "NLTK: the natural language toolkit." *Proceedings of the COLING/ACL on Interactive presentation sessions*. Association for Computational Linguistics, 2006.
40. Pedregosa, Fabian, et al. "Scikit-learn: Machine learning in Python." *Journal of Machine Learning Research* 12.Oct (2011): 2825-2830.
41. Hall, Mark, et al. "The WEKA data mining software: an update." *ACM SIGKDD explorations newsletter* 11.1 (2009): 10-18.
42. Kirkby, Richard, Eibe Frank, and Peter Reutemann. "WEKA Explorer User Guide for Version 3-5-6." (2007).
43. Scuse, David, and Peter Reutemann. "Weka experimenter tutorial for version 3-5-5." *University of Waikato* (2007).
44. Bouckaert, Remco R., et al. "WEKA Manual for Version 3-7-8." *Hamilton, New Zealand* (2013).
45. https://en.wikipedia.org/wiki/Scikit-learn.
46. Blei, David M., Andrew Y. Ng, and Michael I. Jordan. "Latent dirichlet allocation." *Journal of machine Learning research* 3.Jan (2003): 993-1022.
47. http://www.nltk.org/book/
48. Van Halteren, Hans, Jakub Zavrel, and Walter Daelemans. "Improving accuracy in word class tagging through the combination of machine learning systems." *Computational linguistics* 27.2 (2001): 199-229.

49. Davis, Jesse, and Mark Goadrich. "The relationship between Precision-Recall and ROC curves." *Proceedings of the 23rd international conference on Machine learning*. ACM, 2006.

50. Powers, David Martin. "Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation." (2011).

51. Krogh, Anders, and Jesper Vedelsby. "Neural network ensembles, cross validation, and active learning." *Advances in neural information processing systems* 7 (1995): 231-238.

52. Glantz, Stanton A. "Primer of biostatistics." (2002): 246.

53. Anderson, David R., Kenneth P. Burnham, and William L. Thompson. "Null hypothesis testing: problems, prevalence, and an alternative." *The journal of wildlife management* (2000): 912-923.

54. Box, George EP, William Gordon Hunter, and J. Stuart Hunter. "Statistics for experimenters." (1978).

55. Moore, David S. *The basic practice of statistics*. Vol. 2. New York: WH Freeman, 2007.

56. Jin, Dongpu, et al. "Configurations everywhere: Implications for testing and debugging in practice." *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014.

57. http://snowball.tartarus.org/algorithms/lovins/stemmer.html

58. Michael Gegick, Pete Rotella, Tao Xie. Identifying Security Bug Reports via Text mining: An Industry Case Study. *InMining software repositories (MSR), 2010 7th IEEE working conference on 2010 May 2 (pp. 11-20). IEEE.*

59. Kratz, Marie, and Sidney I. Resnick. "The QQ-estimator and heavy tails." *Stochastic Models* 12.4 (1996): 699-724.

VITA

Wei Wen was born in Chengdu, SiChuan, China.

Education:

M.S. in Electrical Engineering, University of Kentucky Jul. 2010

Ph.D. in Materials Science and Engineering, University of Kentucky Dec. 2004

Conference Publications:

1. Wei Wen, Tingting Yu, Jane Hayes: "CoLUA: Automatically Predicting Configuration Bug Reports and Extracting Configuration Options", In International Symposium on Software Reliability Engineering (ISSRE), 2016

2. Tingting Yu, Wei Wen, Xue Han, Jane Hayes, "Predicting Testability of Concurrent Programs", In Proceedings of the 9th International Conference on Software Testing, Verification and Validation (ICST), Pages 168-179, 2016.