Linköping University | Department of Computer Science
Master thesis, 30 ECTS | Datateknik
2017 | LIU-IDA/LITH-EX-A--17/022--SE

Tuning of machine learning algorithms for automatic bug assignment

Daniel Artchounin

Supervisor : Cyrille Berger
Examiner : Ola Leifler

LINKÖPING UNIVERSITY

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Abstract

In software development projects, bug triage consists mainly of assigning bug reports to software developers or teams (depending on the project). The partial or total automation of this task would have a positive economic impact on many software projects. This thesis introduces a systematic four-step method to find some of the best configurations of several machine learning algorithms intending to solve the automatic bug assignment problem. These four steps are respectively used to select a combination of pre-processing techniques, a bug report representation, a potential feature selection technique and to tune several classifiers. The aforementioned method has been applied on three software projects: 66 066 bug reports of a proprietary project, 24 450 bug reports of Eclipse JDT and 30 358 bug reports of Mozilla Firefox. 619 configurations have been applied and compared on each of these three projects. In production, using the approach introduced in this work on the bug reports of the proprietary project would have increased the accuracy by up to 16.64 percentage points.

Acknowledgments

I would like to thank my supervisor, **Daniel Nilsson**, and, my line manager, **Elisabeth Sjöstrand**, in the telecommunications company I have conducted my thesis work at, for having given me the opportunity to work on this fabulous project, and, for their answers to my numerous questions.

I would like to express my gratitude to my supervisor, **Associate Professor Cyrille Berger**, and, my examiner, **Associate Professor Ola Leifler**, from Linköping University, for their support, feedback and patience.

I would like to acknowledge all the employees in the telecommunications company who have helped me in the context of this project, in particular, **Jonas Andersson**, **Hanna Mårtensson**, **Sixten Johansson** and **Leif Jonsson**.

I am also very grateful to my opponent, **Tova Linder**, for having reviewed my thesis several times, and, having provided me with valuable and constructive remarks.

Contents

List of Figures

List of Tables

List of Abbreviations

AFL	Automatic fault localization
ANOVA	Analysis of variance
ICF	Iterative Case Filter
IDE	Integrated development environment
IR	Information retrieval
ITS	Issue tracking system
JDT	Java development tools
KL	Kullback-Leibler
LDA	Latent Dirichlet allocation
LSA	Latent semantic analysis
LSI	Latent semantic indexing
ML	Machine learning
MRR	Mean reciprocal rank
NLP	Natural language processing
NMF	Non-negative matrix factorization
OSS	Open source software
POS	Part of speech
QA	Quality assurance
RFE	Recursive feature elimination
SVD	Singular value decomposition
SVM	Support vector machines
WBFS	Weighted breadth first search

1 Introduction

Machine learning algorithms are becoming more widely used in the software engineering industry. In some tasks such as sentiment classification, these algorithms surpass human performance [26]. Sentiment classification consists of assigning a label to a textual document based on the opinion expressed inside of it. For instance, as in the paper of Pang et al. [26], predicting whether a movie review is positive or negative is a sentiment classification problem. This thesis will investigate the tuning of machine learning algorithms in the context of automatic bug assignment.

1.1 Motivation

When the size of a software development project increases, more bugs are found. Development teams generally use bug repositories to manage this growing number of discovered bugs. These repositories are also called issue tracking systems (ITS).

When a bug is found in a piece of software, a bug report is written. This artifact mainly describes the fault and the way to reproduce it. Generally, the person who writes a bug report is a user, a developer or a tester, and, he or she is called a reporter.

According to Anvik et al. [3], since additional bugs are found and fixed through the use of ITS, these systems might have a positive impact on the overall quality of software products.

Due to the ease of reporting bugs, a significant amount of bug reports is daily submitted and more resources need to be allocated to process them. During four consecutive months, around 29 bug reports have been daily submitted to the ITS of the Eclipse software development project [4]. If 5 minutes have been used for each bug, more than 2 working hours per day have been spent on handling these issues (the time needed to fix each bug is not included).

Handling bug reports is called bug triage [3]. This task is frequently done by a specific person called a bug triager. Bug triage is a combination of two subtasks. In the context of the first subtask, the triager has to decide whether or not to consider the content of the bug report based on its relevance. For instance, he or she has to identify the duplicate bug reports: if a fault in a bug report has already been reported or has already been fixed, the triager should not handle it. If the report is taken into account, then, the triager has to assign it to a person or a team. This person or this team will have the responsibility to fix the issue.

The aforementioned second subtask is arduous, time-consuming and prone to errors [3]. This subtask is often done manually by analyzing various artifacts such as previously fixed bug reports, source code and documents recording the skills of each developer.

The bug assignment task has a significant impact on the cost of the maintenance of a software product. If the assignee is not able to fix the bug described in a bug report, it will be reassigned. This phenomenon is called bug tossing [19]. As each time a bug is reassigned, more working hours are spent to fix it, bug assignment has a major role in maintenance cost.

Due to its economic issues, automating bug assignment would be beneficial for many software projects.

1.2 Aim

Many researchers have introduced various methods to solve the automatic bug assignment problem. In almost all of them, the introduced technique uses at least an instance of a specific type of machine learning algorithms called classifiers.

The topic of this thesis is mostly based on one of the main findings of Thomas et al. [34]: the configuration of a classifier has an impact on the results obtained in the context of automatic bug localization. I believe that this result is also valid in the context of automatic bug assignment. Within this framework, this thesis will therefore focus on introducing some systematic approaches to potentially find some of the best existing configurations in terms of two metrics, the accuracy and the mean reciprocal rank (MRR).

1.3 Research questions

The thesis will intend to answer the following research questions:

1. How can we select which pre-processing technique(s) to apply on a set of bug reports?

   In text classification, many pre-processing techniques could be applied on a set of documents to get better results (stop words removal, stemming, lemmatization, etc.). As the automatic bug triage problem could be considered as a text classification problem, the aforementioned techniques may also be used to achieve better results. According to Čubranić et al. [14], stemming has not a significant impact on the accuracy of a classifier in the context of automatic bug assignment. Nevertheless, in their works, some other researchers have used this pre-processing technique to probably improve their results [37, 9, 39]. I believe that the selection of the pre-processing techniques to use should be based on the bug reports of the data set. The answer to this research question will introduce a method to make a systematic optimal choice among several options.

2. How can we choose which model to use to represent a bug report?

   As any other text classification problem, many models can be used to extract features from the textual content of a bug report (by counting the occurrences of each word in each bug report, by using binary numbers to indicate the presence/absence of each word in each bug report, by using tf-idf weights, etc.). I also believe that the choice among the possible models should be based on the specificities of the bug reports in the data set. The answer to this research question will also introduce a method to make a systematic optimal choice.

3. How can we select which feature selection technique to apply on a given representation of a bug report?

   Due to the substantial number of distinct words that could be used in a set of bug reports, the dimension of the feature vectors representing them can be significant. Using all the features of these vectors might introduce some noise and have a negative impact

on the predictions of an automatic bug triage system. According to Xuan et al. [38], the use of feature selection can have a positive impact on the predictions of an automatic bug triage system, while reducing the size of the data set. Given a set of bug reports, some feature selection techniques may lead to better results than others. Selecting the size of the subset of the remaining features also influences the results. To answer this research question, a method will be proposed to select a feature selection technique among others and the size of the subset of the remaining features to get good results.

4. How can we tune an individual classifier on a set of bug reports?

   In the paper of Jonsson et al. [20], the selection of the classifiers on which further studies were made was based on their accuracies without having tuned them previously (using the default configurations of the library implementing them). I believe that the results may not have been the same if each classifier was tuned before the selection. The answer to this research question will introduce a method to efficiently tune any individual classifier on a set of bug reports.

1.4 Delimitations

In the framework of a software development project, several artifacts such as software architecture documents or project plans are produced. Some relevant data can be extracted from these documents and be used to train some algorithms which could solve the bug assignment problem. Nevertheless, in the framework of this thesis, only the titles and the descriptions of the previously fixed bug reports in the ITS will be considered to train the individual classifiers.

As mentioned above, several approaches have been introduced in order to solve the automatic bug triage problem. In this thesis, only the approach using machine learning classifiers will be studied.

Tuning and evaluating each algorithm used in this thesis will be done with 66 066 bug reports of a telecommunications company. To achieve replicability, and, as most of the prior works on automatic bug assignment have used them, the same analysis will be conducted on 24 450 bug reports of Eclipse JDT[1] and 30 358 bug reports of Mozilla Firefox[2].

The bug reports of the proprietary software development project will be used to solve the automatic bug assignment to teams problem whereas the bug reports of the two open source projects will be used to solve the automatic bug assignment to developers problem. Both problems are very similar: they could be both considered as text classification problems. The only major difference is the number of classes (lower in the context of automatic bug assignment to teams). As the focus of the thesis is on the benefits that the application of the introduced method could bring on the accuracies and the MRR values of the classifiers solving the automatic bug assignment problem, I believe that the use of the bug reports of these three projects in slightly different contexts will not cause any confusion.

[1] http://www.eclipse.org/jdt
[2] http://www.mozilla.org

2 Theory

In this chapter, the tools used to handle bug reports in software projects are first described. Some techniques and models used in a specific field of computer science called information retrieval are then presented. Next, some techniques generally used to solve text classification problems are described. Finally, some scientific publications related to the automatic bug assignment problem are presented.

2.1 Bug reporting and development tools

In this section, the process, and, the tools used to report and handle bugs in software development projects are described.

2.1.1 Bug report

When a bug is found in a software project, a bug report is written. This report mainly describes the problem and how to reproduce it. A bug report is usually written by a user, a developer or a tester. The author of a bug report is called the reporter.

Each bug report has some "pre-defined fields" [3], and, some other values such as a title or a description which should be filled in by the reporter. It could also contain some other relevant elements such as screenshots.

Two screenshots of the bug report 75119[1] of the Mozilla Firefox project could be consulted in the Figures 2.1 and 2.2. As can be seen, the reporter field, the reported date and the identifier of the bug report are some examples of "pre-defined fields" [3]. The values of these fields were automatically set when the bug was reported. The values of some other fields such as the product, the component or the importance of the bug report were manually set by the reporter. The other members of the project might have updated them later. The values of some other fields such as the status or the assignee of the bug report change frequently until the bug is fixed. The cc field generally contains the e-mail addresses of the members of the project who are interested in the bug report. As mentioned above, each bug report has mainly two textual fields: a title (Figure 2.1) and a description (Figure 2.2). Finally, the comments of the other members of the project related to the bug also appear at the bottom of the bug report (Figure 2.2).

[1] https://bugzilla.mozilla.org/show_bug.cgi?id=75119

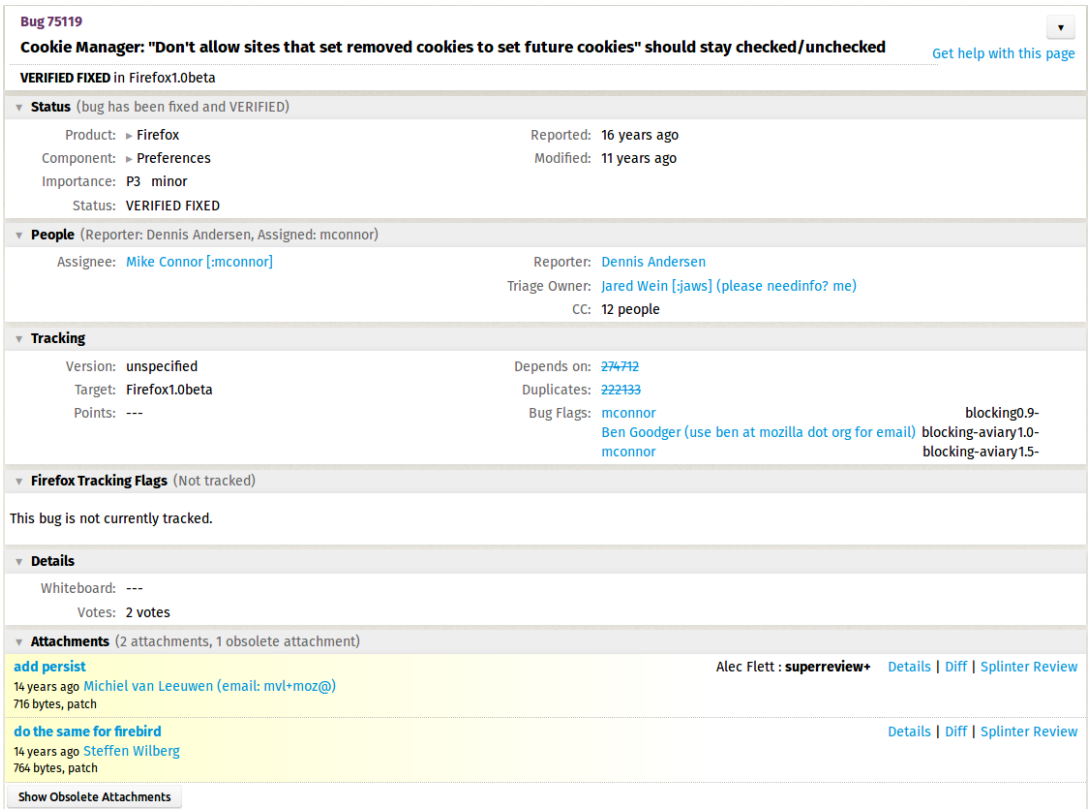


Figure 2.1: The fields (except the description and the comments) of the bug report 75119 of Mozilla Firefox

2.1.2 Issue tracking system

An issue tracking system (ITS) is mainly a system used to manage the bug reports and the names of the developers who have fixed them. This type of system is often used as a mean of communication among the developers as well as between the users and the developers [3]. According to Anvik et al. [3], as some additional bugs are found and fixed thanks to the ITS, these types of repositories might have a positive impact on the quality of software products.

Bugzilla[2] is an open source, cross platform and web-based ITS. It is one of the products of the software community Mozilla. This ITS is used to manage the bug reports of all the products of the Eclipse project[3] (including Eclipse JDT) and all the products of the Mozilla community[4] (including Firefox). Eclipse Java development tools (JDT) is a product of the Eclipse project which is an integrated development environment (IDE) mostly written in Java. The Eclipse project is open source and cross platform. Eclipse JDT contains several plug-ins that could be used for Java development. Firefox is another product of the software community Mozilla. Firefox is an open source web browser mostly written in C++.

In the context of this Master's thesis, the bug reports in the ITS of a telecommunications company and the bug reports in the ITS of both aforementioned products (Eclipse JDT and Mozilla Firefox) will be studied.

In any ITS, until it is eventually closed, each bug report goes through several states[5]. The ordered sequence of all the states of a bug report is called its life cycle.

[2] https://www.bugzilla.org/
[3] https://bugs.eclipse.org/bugs/
[4] https://bugzilla.mozilla.org/
[5] https://www.bugzilla.org/docs/3.6/en/html/lifecycle.html

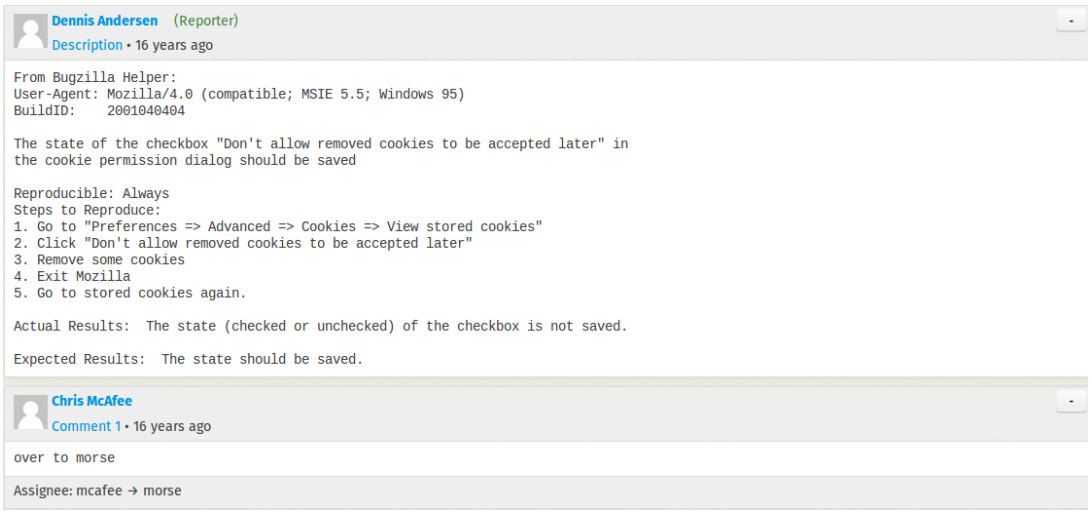


Figure 2.2: Description and comments of the bug report 75119 of Mozilla Firefox

More specifically, in Bugzilla, the state of a bug report is a combination of two pieces of information: the value of its status field and the value of its resolution field (used to know how a bug report has been resolved). The Figure 2.3 is a simplified graphical representation of the life cycle of a bug in Bugzilla[6]. When a bug is found, a report is filled in and its status is generally set to NEW. Next, the bug report is assigned to a developer and its state is modified to ASSIGNED. The bug is then generally either fixed (its status is modified to RESOLVED) or reassigned (its status is updated to NEW). If it has been resolved, the bug is then verified by the quality assurance (QA) team and its status is set to VERIFIED. Finally, the status of the bug is modified to CLOSED. If the status of the bug report was set to RESOLVED and the bug is still not fixed, the bug report is reopened (its status is updated to REOPENED). When a bug is resolved, its resolution is modified. If a modification has been made in the code base, the resolution field is set to FIXED. If the bug report is a duplicate, its resolution is updated to DUPLICATE. The resolution is set to WORKSFORME if the assignee was not able to reproduce the bug. If the bug report should not be taken into account, its resolution is set to INVALID. Finally, if the issue in the bug report will not be solved, the resolution field is set to WONTFIX.

The telecommunications company has its own ITS. The possible states of the life cycle of a bug report in this ITS are slightly different from Bugzilla. The Figure 2.4 is a simplified graphical representation of the life cycle of a bug in this ITS. When a bug is found, a report is submitted and its state is set to PRIVATE. If the bug report should be taken into account, its state is then set to REGISTERED. Otherwise, its state is updated to CANCELLED. If the bug report should be considered, it is then assigned to a team (the status is modified to ASSIGNED). Next, normally, a fix is proposed and the state is updated to PROPOSED. Generally, the fix is approved (the state is modified to PROPOSAL APPROVED). The fix is then verified (the state is set to CORRECTION VERIFIED). Next, the bug report is answered and the state is changed to ANSWERED. Finally, the bug report is closed (its status is set to FINISHED).

In any ITS, it is normally possible to search for a specific bug report thanks to its identifier or some key words inside of it. It is also generally possible to look for all the bug reports having some specific states. For instance, in the ITS of the Mozilla project, one can search for all the bug reports belonging to the Firefox product with a RESOLVED, VERIFIED or CLOSED status, and, a FIXED resolution.

[6] https://www.bugzilla.org/docs/3.6/en/html/lifecycle.html
[7] https://www.bugzilla.org/docs/3.6/en/html/lifecycle.html

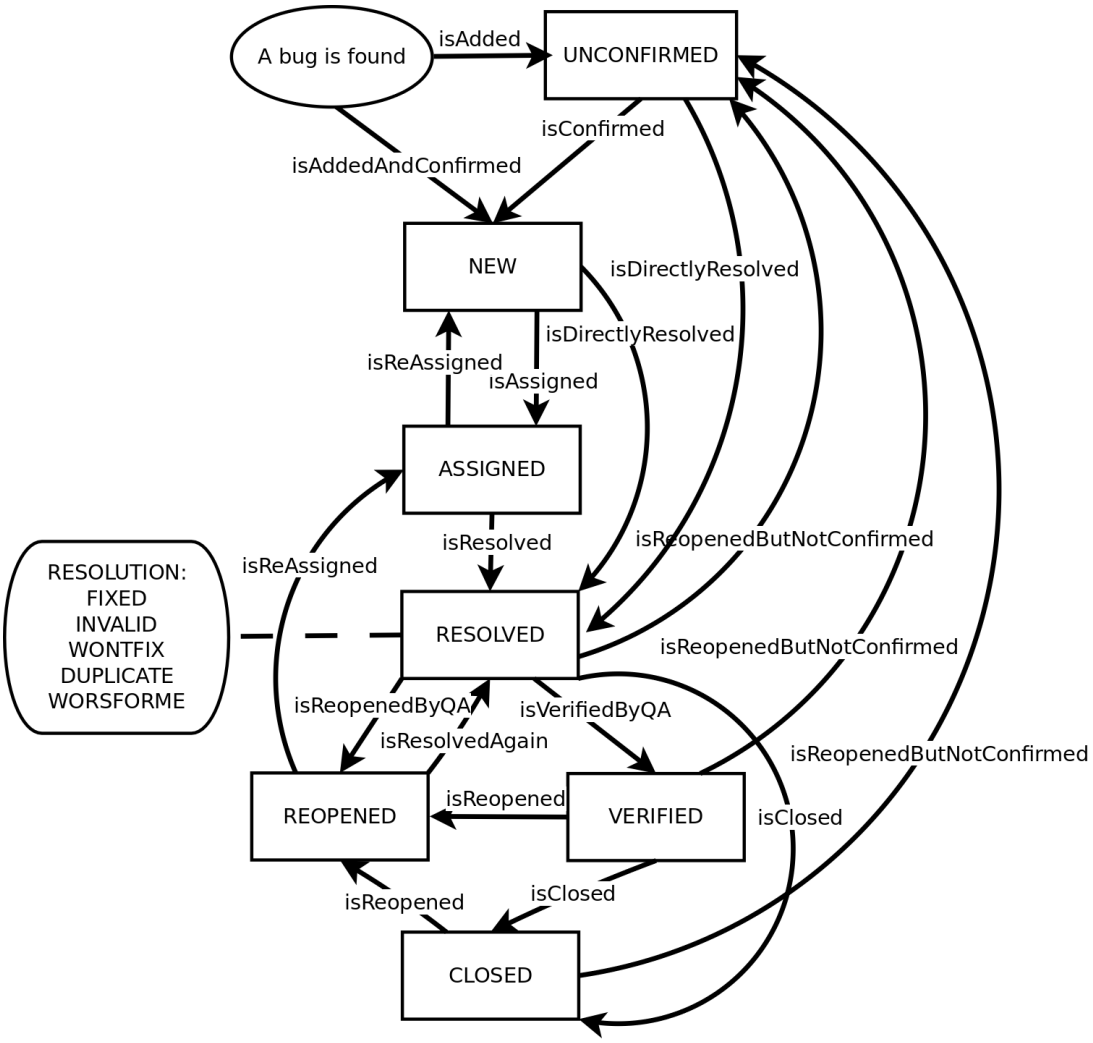


Figure 2.3: The simplified life cycle of a bug report in Bugzilla[7]

The text in the bug reports is a potentially valuable source of information that might be used to route them. It is however critical to choose an appropriate representation of the textual content for effective use by classification algorithms. Determining appropriate representations is conducted using techniques that are usually described as information retrieval.

2.2 Information retrieval

Information retrieval (IR) is an area of computer science. In this field, based on specific needs of a customer, the goal is to retrieve some relevant documents from a generally large set of documents.

In this section, first, a formal representation of an IR model will be presented. Some commonly used pre-processing techniques in this field will then be introduced. Finally, two classic IR models will be described.

2.2.1 Formal representation of any IR model

As stated by Baeza-Yates et al. [6], any IR model could be formally described as a quadruple $(D, Q, F, R(\vec{d}, \vec{q}))$, where D if a set of representations for the documents in the corpus; Q is a

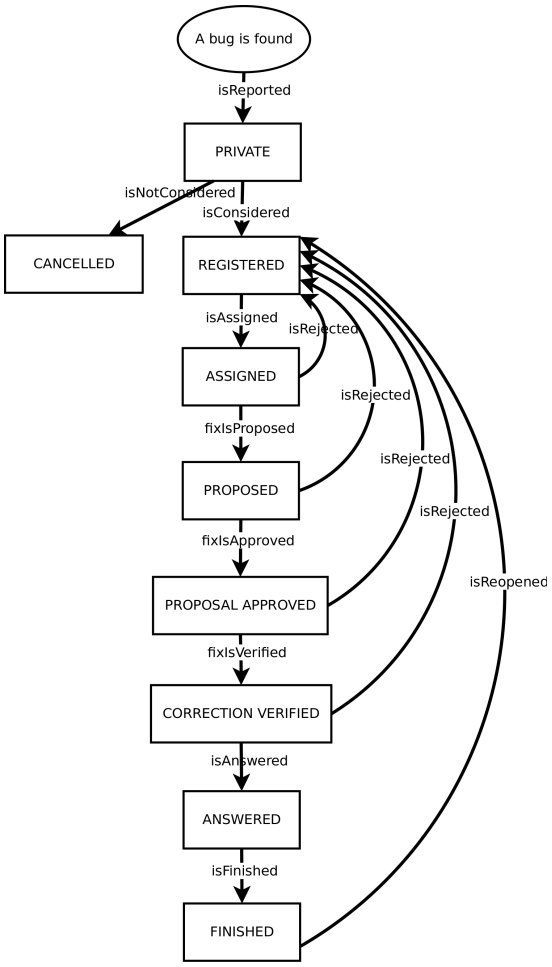


Figure 2.4: The simplified life cycle of a bug report in the ITS of the telecommunications company

set of queries (representations of the needs of the user); F is a framework used to model the documents in the corpus, the needs of the user and their relationships; $R(\vec{d}, \vec{q})$ is a ranking function which maps a real number to a document representation $\vec{d} \in D$ and a query $\vec{q} \in Q$. The goal of the function R is to order the elements in D with respect to a query $\vec{q} \in Q$.

2.2.2 Commonly used pre-processing techniques

Many IR models are based on the use of index terms [6]. These terms are generally keywords which represent sets of words in the corpus or words which appear at least in one document of the corpus. The goal of these terms is to simplify the IR problem by representing the documents in the corpus and the needs of the customer using some keywords or words of this set of documents. In this section, some commonly used techniques to build a set of index terms are presented.

Tokenization

Tokenization is the process of splitting an input string (it could be a document) into smaller meaningful strings which are called tokens [10]. The splitting task is generally based on some specific delimiters such as punctuation characters.

This task is not trivial. The management of some specific punctuation characters such as hyphens and apostrophes is complex. Splitting an input string based on its white spaces is not always an ideal solution. For instance, splitting the input string `Las Vegas` into two tokens, `Las` and `Vegas`, is probably not desirable.

Stemming

A word stem is a specific part of a term that does not change when the word is modified. This modification could be made to use a different grammatical category (add a suffix to an adjective to form a noun for example) or could be based on the grammatical category of the term (conjugate a verb for instance).

Stemming is the process of transforming a word into its stem [10].

For example, the stem of the word `helping` might be `help`.

Lemmatization

As stemming, the goal of lemmatization is to cluster related words together. Contrary to stemming, lemmatization will reduce each word to its lemma (its dictionary form) [10].

Unlike stemming, this reduction is based on the analysis of the context of the occurrence of each word. For instance, it could be based on the part of speech (POS) of each word. The POS is the grammatical category of a word. Adjectives or nouns are parts of speech.

For instance, the lemma of the word `tried` might be `try` whereas the stem of the same word might be `tri`.

Compared to stemming, one of the drawbacks of lemmatization is its increased computational cost due to the analysis needed to find the lemma of a word.

Stop words removal

A stop word is a word which will not be used by the selected model. Stop words are generally common in the language used in the documents of the corpus [10]. The gain made by a model analyzing these words is usually negligible.

For instance, the determiner `the` or the preposition `on` could be stop words.

Other techniques

The following techniques are also used to build a set of index terms:

- punctuation removal: all the tokens containing only punctuation characters such as dots or commas are removed;
- numbers removal: all the tokens only made up of numbers are removed;
- conversion to lower case: each character of each token is converted to lower case [10]. The goal of this pre-processing step is that the following steps (classification for instance) are not case-sensitive;
- in their paper, Naguib et al. [25] have introduced a new approach to solve the automatic bug assignment problem. Before applying their method, they have used some regular expressions to remove some non-discriminative tokens (HTML tags, hexadecimal numbers, etc.) from their bug reports.

2.2.3 Three classic IR models

Three classic IR models use index terms to describe the documents in the corpus: the Boolean model, the vector model and the probabilistic model. These index terms are generally retrieved using at least one of the pre-processing techniques introduced in the previous section. In this thesis, only the Boolean model and the vector model will be used. Only these two classic IR models will therefore be described.

Boolean model

As written in the book of Baeza-Yates et al. [6], the framework F of the Boolean model could be described as follows. Each document $\vec{d_i} \in D$ is a vector which elements represent the index terms of the corpus and which are binary. If the index term k_j is in the document $\vec{d_i}$, then, $d_{ij} = 1$. Otherwise, $d_{ij} = 0$. A query written by a user is a Boolean expression using the index terms of the corpus and the three following Boolean operators `and`, `or` and `not`. This query is converted to the disjunctive normal form (a disjunction of conjunctive vectors with all the index terms of the corpus). If one of the conjunctive vectors of the converted query is the same as a document $\vec{d_i}$, then, this document will be predicted as being relevant. Otherwise, it will be predicted as being irrelevant.

According to Baeza-Yates et al. [6], the main drawback is that the model is based on the occurrence of a perfect matching between a conjunctive vector of a query and a document $\vec{d_i}$. Too many documents or too few documents are therefore often retrieved. The documents in the corpus are also not ranked with respect to a query (they are either relevant or not relevant). However, the main advantage of the model is its intuitiveness.

Vector model

In the framework F of the vector model, the similarity sim between a document $\vec{d_i} \in D$ and a query $\vec{q} \in Q$ is computed using the Equation 2.1 [6].

$$sim(\vec{d_i}, \vec{q}) = \frac{\vec{d_i}.\vec{q}}{|\vec{d_i}| * |\vec{q}|} = \frac{\sum_{k=1}^{t} d_{ik} * q_k}{\sqrt{\sum_{k=1}^{t} {d_{ik}}^2} * \sqrt{\sum_{k=1}^{t} {q_k}^2}} \tag{2.1}$$

In the above equation, t is the number of index terms in the corpus. As can be seen in the Equation 2.1, the similarity sim is the cosine of the angle between a representation $\vec{d_i}$ of a document i of the corpus and a query $\vec{q}$.

Let N be the number of documents in the corpus, n_j the number of documents in the corpus in which the index term k_j appears and $freq_{ij}$ the frequency of k_j in $\vec{d_i}$. The weight d_{ij} of each word k_j in each document i is computed using the Equation 2.2:

$$d_{ij} = f_{ij} * idf_j, \tag{2.2}$$

where $f_{ij} = \frac{freq_{ij}}{\max_j(freq_{ij})}$ and $idf_j = \log\left(\frac{N}{n_j}\right)$.

Let $freq_{qj}$ be the frequency of k_j in the query q. The weight q_j of each word k_j in the query q is computed using the Equation 2.3:

$$q_j = f_{qj} * idf_j, \tag{2.3}$$

where $f_{qj} = 0.5 + \frac{0.5 * freq_{qj}}{\max_j(freq_{qj})}$ and $idf_j = \log\left(\frac{N}{n_j}\right)$.

The main disadvantage of this model is that the terms in each document and each query are assumed to be independent [6]. In practise, this assumption is not always true. For instance, the word `science` is likely to appear in a document containing the word `computer`.

The main advantages of the model are its simplicity, its performance due to the weights computed for each index term in each document, the facts that partial matching is taken into account in the similarity index and that documents are ranked.

After having removed the noise from the textual content of the bug reports and selected an appropriate model to represent them, the issues are routed using some algorithms of a specific field of computer science called machine learning.

2.3 Text classification

As automatic bug assignment could be considered as a text classification problem, this section will deal with some techniques generally used to solve this specific type of problems.

First, the machine learning field will be introduced and some algorithms generally used in text classification will be described. Second, some feature extraction techniques will be presented. Some feature selection techniques will then be described. The fourth section will deal with some commonly used tuning techniques. Finally, three metrics used to evaluate the performance of some text classification algorithms will be presented.

2.3.1 Machine learning

In machine learning (ML), the focus is on building computer systems which should perform a task and improve themselves in this task by learning from data.

The learning process can be achieved mainly via three different ways: unsupervised learning, supervised learning and reinforcement learning.

The focus of this section will be on supervised learning as this thesis mainly deals with this learning process.

In supervised learning, the computer systems learn from labeled data [28]. Each element of this set (the labeled data used to train the system) is a pair containing an input (called a feature vector) and an output (called a label). During the learning process called the training phase, each computer system builds a function which associates each input to its related output.

Supervised learning can mainly solve two types of problems: the classification problems and the regression problems.

As this thesis only deals with classification, some algorithms solving only this category of problems will be presented.

In classification, given the elements of the data set (the inputs and the outputs), the computer system called the classifier should predict the output of an unseen input [11]. The performance of the classifier is generally assessed using a subset of the data set not used to train the model: this subset is called the test set. Various classification algorithms exist. As only six of them will be used in the context of this thesis, only these algorithms will be introduced in the following sections.

Nearest centroid classifier

With the nearest centroid classifier, which is also called the Rocchio classifier [24], first, the centroid (mean) of each class is computed using the feature vectors of its observations. Given a new input, the prediction of this classifier is the class whose centroid is closest to its feature vector.

Naive Bayes classifier

The prediction of this model is the class which maximizes the probability $\mathbb{P}(Y = c_k|X_1 = x_1, \cdots, X_n = x_n)$ that the class of a feature vector $(x_1, \cdots, x_n)$ is c_k [11]. This model is based on the Bayes' theorem (Equation 2.4):

$$\mathbb{P}(Y = c_k|X_1 = x_1, \cdots, X_n = x_n) = \frac{\mathbb{P}(X_1 = x_1, \cdots, X_n = x_n|Y = c_k)\mathbb{P}(Y = c_k)}{\mathbb{P}(X_1 = x_1, \cdots, X_n = x_n)}, \tag{2.4}$$

where $\mathbb{P}(X_1 = x_1, \cdots, X_n = x_n)$ is the probability that the values of an unknown feature vector are $(x_1, \cdots, x_n)$; $\mathbb{P}(Y = c_k)$ is the probability that the class of any feature vector is c_k and $\mathbb{P}(X_1 = x_1, \cdots, X_n = x_n|Y = c_k)$ is the probability that, knowing that the class of an unknown feature vector is c_k, the values of its features are $(x_1, \cdots, x_n)$. This model also assumes that the set of features is pairwise independent. Based on this assumption, the Equation 2.4 could be simplified as follows:

$$\mathbb{P}(Y = c_k|X_1 = x_1, \cdots, X_n = x_n) = \frac{\mathbb{P}(Y = c_k)\prod_{i=1}^{n}\mathbb{P}(X_i = x_i|Y = c_k)}{\mathbb{P}(X_1 = x_1, \cdots, X_n = x_n)}. \tag{2.5}$$

For a given feature vector $(x_1, \cdots, x_n)$, the prediction of this algorithm is the class c_k which maximizes the numerator of the right-hand side of the Equation 2.5.

Support vector machines

In the support vector machines (SVM) algorithm, each feature vector of the data set is considered as a point in a n-dimensional space, where n is the number of features in each vector [11]. This algorithm learns a linear decision boundary which is the hyperplane that maximizes its distance to the nearest point from any class. This algorithm can also be extended in order to be able to build a non-linear decision boundary. It has been demonstrated that the linear decision boundary can be represented as a linear combination of inner products. By replacing each inner product by a kernel function applied on the two vectors normally involved in the initial inner product, a non-linear decision boundary can be learned.

Logistic regression

In this algorithm, the prediction is the class c_k which maximizes its posterior probability $\mathbb{P}(Y = c_k|\phi(X_1 = x_1, \cdots, X_n = x_n))$, where $x = (x_1, \cdots, x_n)$ is a feature vector and $\phi(x)$ is a fixed nonlinear transformation of each feature vector to a space where the classes are linearly separable [11].

In binary classification, the posterior probability of the class c_1 is given below:

$$\mathbb{P}(Y = c_1|\phi(X_1 = x_1, \cdots, X_n = x_n)) = \sigma\left(w^T\phi(X_1 = x_1, \cdots, X_n = x_n)\right), \tag{2.6}$$

where $\sigma(x) = \frac{1}{1+e^{-x}}$ is the sigmoid function and $w^T = (w_1, \cdots, w_n)$ is a vector of parameters which are estimated given a data set. The posterior probability of the class c_2 is $\mathbb{P}(Y = c_2|\phi(X_1 = x_1, \cdots, X_n = x_n)) = 1 - \mathbb{P}(Y = c_1|\phi(X_1 = x_1, \cdots, X_n = x_n))$.

In multiclass classification, the posterior probability of the class c_k is given below:

$$\mathbb{P}(Y = c_k|\phi(X_1 = x_1, \cdots, X_n = x_n) = \frac{\exp\left(w_k^T\phi(X_1 = x_1, \cdots, X_n = x_n)\right)}{\Sigma_{i=1}^{K}\exp\left(w_i^T\phi(X_1 = x_1, \cdots, X_n = x_n)\right)}, \tag{2.7}$$

where $w_i^T = (w_{i1}, \cdots, w_{in})$ are vectors of parameters which are estimated given a data set and K is the number of classes.

For both classification problems, in order to estimate the parameters of the model, first, the cross-entropy error function (the negative log-likelihood) is written. The Newton-Raphson algorithm is then used to find the parameters which minimize the aforementioned error function.

Perceptron

This machine learning algorithm solves a binary classification problem [11]. It could also be extended to solve a multiclass classification problem using the one-versus-the-rest technique or the one-versus-one technique. The prediction is made using the following equation:

$$t = f(w^T \phi\left(x_1, \cdots, x_n\right)), \tag{2.8}$$

where $x = (x_1, \cdots, x_n)$ is a feature vector, $\phi(x)$ is a fixed nonlinear transformation of each feature vector to a space where the classes are linearly separable and $w^T = (w_1, \cdots, w_n)$ is a vector of parameters which values are computed based on a given data set. f is a non linear activation function defined below:

$$f(a) = \begin{cases} +1, & a \geqslant 0 \\ -1, & a < 0 \end{cases} . \tag{2.9}$$

The first component $\phi_0(x)$ of $\phi(x)$ is generally a bias component ($\phi_0(x) = 1$). For convenience, the prediction $+1$ is related to the class c_1 whereas the prediction -1 is related to the class c_2. The values of the elements of w are selected so that they minimize the so-called perceptron criterion:

$$E(w) = -\sum_{i \in M} w^T \phi_i t_i, \tag{2.10}$$

where $\phi_i = \phi\left(x_{i1}, \cdots, x_{in}\right)$, $t_i = f(w^T \phi_i)$ and M is the set of the indexes of the misclassified elements of the training set.

By using the stochastic gradient descent algorithm, the value of w is iteratively computed:

$$w^{(\tau+1)} = w^{(\tau)} - \eta \nabla E(w) = w^{(\tau)} + \eta \phi_i t_i, \tag{2.11}$$

where η is the learning rate parameter and τ is used to represent the τ-th step of the algorithm.

Stochastic gradient descent

This algorithm is a technique used to find the parameter which minimizes a function. It is usually used for large-scale machine learning problems [40]. In this algorithm, the goal is to find the parameter w which minimizes the following equation:

$$\mathbb{E}_{X,Y} l(p(X), Y), \tag{2.12}$$

where X is the parent random variable of the feature vectors, Y is the parent random variable of the corresponding labels, $\mathbb{E}$ is the expectation, $p(X)$ is the prediction of a classifier such as SVM and l is a loss function measuring the quality of any prediction. For convenience, the stochastic gradient descent algorithm will only be described for linear classifiers (when $p(x) = w^T x$, where $w^T = (w_1, \cdots, w_n)$ is a vector of parameters which should be estimated given a data set).

As the Equation 2.12 may admit several solutions or no solution, some regularization parameters are generally added:

$$\mathbb{E}_{X,Y} l(w^T X, Y) + \frac{\lambda}{2} \|w\|_2^2, \tag{2.13}$$

where λ is a regularization parameter.

The stochastic gradient descent algorithm will iteratively solve the Equation 2.13 as follows:

$$w^{(\tau+1)} = w^{(\tau)} - \eta^{(\tau+1)}(S^{(\tau+1)})^{-1}\frac{\partial L_l}{\partial w}\left(w^{(\tau)}, X^{(\tau+1)}, Y^{(\tau+1)}\right), \tag{2.14}$$

where $\eta^{(\tau)} > 0$ is a learning rate parameter, $S^{(\tau)}$ is a symmetric positive definite matrix which could have an impact on the convergence rate and $L_l(w, x, y) = l(w^T x, y) + \frac{\lambda}{2}\|w\|_2^2$.

The Equation 2.14 could be written as follows:

$$w^{(\tau+1)} = w^{(\tau)} - \eta^{(\tau+1)}(S^{(\tau+1)})^{-1}\left(\lambda w^{(\tau)} + l_1'\left((w^{(\tau)})^T X^{(\tau+1)}, Y^{(\tau+1)}\right)X^{(\tau+1)}\right), \tag{2.15}$$

where $l_1'(p, y) = \frac{\partial l(p,y)}{\partial p}$.

2.3.2 Feature extraction

In this section, some commonly used techniques to extract features from data are described. The focus will mainly be on text classification feature extraction techniques.

Boolean representation

This technique is closely related to the Boolean model of information retrieval (Section 2.2) [29]. Each j-th column of a term-document matrix TD is a Boolean vector which elements represent the occurrences of the different terms of the whole corpus in the j-th document. If the i-th term occurs at least one time in the j-th document, $TD_{ij} = 1$, where TD_j is the vector modeling the j-th document of the corpus (the j-th column of the term-document matrix). Otherwise, $TD_{ij} = 0$.

Use of the tf weights

This technique is based on the vector model of information retrieval (Section 2.2). In this representation, each j-th column of a term-document matrix TD models the content of the j-th document in the corpus. The elements of this vector represent the different terms in the whole corpus. The values of the vector modeling the j-th document of the corpus (the j-th column of the term-document matrix TD) are computed using the first term on the right-hand-side of the Equation 2.2. Only the term frequencies (tf) of each document are taken into account to fill-in the matrix TD.

Use of the tf-idf weights

This technique is also based on the vector model of information retrieval (Section 2.2) [29]. The term-document matrix TD is filled-in using directly the Equation 2.2. The values of the vector TD_j modeling the j-th document of the corpus (the j-th column of the term-document matrix TD) are computed using the frequency of each term of the corpus inside the j-th document (tf) and the inverse document frequency of each term in the corpus (idf).

Latent semantic indexing

Also called latent semantic analysis (LSA), latent semantic indexing (LSI) is a technique used to reduce the dimension of a term-document matrix TD [15].

This technique is based on a mathematical theorem called singular value decomposition (SVD). This theorem states that, for all $m * n$ matrix X, there exists a decomposition:

$$X = U\Sigma V^T, \tag{2.16}$$

where U is a $m \times m$ orthogonal matrix, Σ is a rectangular $m \times n$ matrix and V is a $n \times n$ orthogonal matrix.

The diagonal elements of Σ are the singular values of X (the square roots of the eigenvalues of $X^T X$).

Due to this theorem, any $m * n$ matrix X admits also the following decomposition:

$$X = U_r \Sigma_r V_r^T, \tag{2.17}$$

where r is the rank of X, U_r is a $m \times r$ matrix which columns could be considered as an orthonormal set, Σ_r is a squared $r \times r$ matrix and V_r is a $n \times r$ matrix which columns could be considered as an orthonormal set.

Based on the Equation 2.17, a term-document matrix TD could also be decomposed. Given a number of features k desired by the user, the smallest submatrix in which the k highest diagonal terms of Σ_r appear will be extracted: the rest of Σ_r is discarded. The columns of U_r and the columns of V_r containing the terms which were multiplied with the terms of the discarded rows and columns of Σ_r are also discarded. Three new matrix are obtained: U_k (a $m \times k$ matrix) Σ_k (a $k \times k$ matrix) and V_k^T (a $k \times n$).

As can be seen in the Equation 2.18, the product of the three aforementioned matrices is an approximation of the initial term-document matrix.

$$TD = U_r \Sigma_r V_r^T \approx U_k \Sigma_k V_k^T \tag{2.18}$$

Each row i of U_k is the representation of a term i (the i-th row of TD) in a k-dimensional space. Each column j of V_k^T is a representation of a document j (the j-th column of TD) in the same k-dimensional space. The data in TD are projected in this k-dimensional space. With the Equation 2.18, an approximation of TD is computed using these projected data.

Using the Equation 2.19, the projection d_k of a new document d in the k-dimensional space could be computed. Thanks to this formula, in this new space, a classifier could be trained and could make some predictions.

$$d_k = \Sigma_k^{-1} U_k^T d \tag{2.19}$$

NMF

In text classification, non-negative matrix factorization (NMF) is generally used to reduce the dimension of a document-term matrix (DT) [21]. In this matrix, each row models a document of the corpus whereas each column represents a distinct term of the corpus.

This technique tries to find two non-negative matrices W and H such as:

$$DT \approx WH, \tag{2.20}$$

where DT is a $m * n$ matrix, W is a $m * k$ matrix, H is a $k * n$ matrix and $k \leqslant n$. The integer k is selected by the user. The matrix W obtained thanks to this factorization could be interpreted as a matrix which rows represent the documents of the corpus and which columns represent the k topics of the corpus. H is a matrix which rows represent the topics of the corpus and which columns represent the terms of the corpus. H might be used to determine the relative importance of each term of the corpus in each topic.

2.3.3 Feature selection

Some features of the feature vectors may be noisy and could have a negative impact on the classifiers which will use them. Sometimes, it could be relevant to train the classifiers on a subset of the feature vectors. The process of selecting a subset of features among all the available features in the feature vectors is called feature selection.

According to Xuan et al. [38], this process could increase the performance of a classifier. As the size of the feature vectors is reduced, it also has a positive impact on the computational cost related to the training phase of the classifiers.

In the following sections, some commonly used feature selection techniques are described.

Chi-squared

A chi-squared (χ^2) test is run between each feature and the vector of labels [29]. The χ^2 test is normally used to test the hypothesis of independence of two random variables based on their samples. Running this test between each feature and the vector of labels could be used to know which features have the most important impact on the vector of labels (their χ^2 scores are the highest). Based on this information, the aforementioned features could be selected.

ANOVA

As written in the paper of Surendiran et al. [33], the analysis of variance (ANOVA) is a statistical test which aims to know whether or not the expectations of a set of random variables are significantly different, based on their samples. This statistical test relies on the ratio between the variance related to the mean of each random variable's sample (called the between group sum of squares) and the variance in each random variable's sample (called the within group sum of squares).

ANOVA could be used to determine the impact of each feature on the total sum of squares (the sum of the within group sum of squares and the between group sum of squares) [33]. The ratio between the within group sum of squares of each feature and the total sum of squares, called the Wilks's lambda, is used to filter the features. The features with the highest ratios are generally selected.

Mutual information

Mutual information is a a measure used in probability theory to assess the dependency between two random variables [29]. The mutual information $I(X, Y)$ of two random variables X and Y is computed using the Equation 2.21:

$$I(X,Y) = \sum_{x \in X} \sum_{y \in Y} \mathbb{P}(X = x, Y = y) \log \left(\frac{\mathbb{P}(X = x, Y = y)}{\mathbb{P}(X = x)\mathbb{P}(Y = y)} \right). \tag{2.21}$$

As can be seen in the Equation 2.21, if X and Y are independent, $I(X, Y) = 0$. Moreover, the higher the value of $I(X, Y)$ is, the higher the dependency between X and Y is.

For each feature, the value of this measure is computed to assess its dependency with the labels.

The features with the highest mutual information values are selected.

Recursive feature elimination

Recursive feature elimination (RFE) is a recursive technique relying on a machine learning algorithm which should assign a weight to each feature during its training phase [17].

The aforementioned feature selection method uses these weights to recursively filter the features [17]. First, the ML algorithm is trained using all the features. The feature(s) with the lowest weight(s) is/are then removed. Recursively, the ML algorithm will be trained using the remaining features and its output will be used to remove some additional feature(s). This recursive process is repeated until the number of features wanted by the user is reached.

2.3.4 Tuning techniques

As seen in the Section 2.3.1, there exists many machine learning algorithms. Each of them has several parameters. In order to obtain some good predictions from at least one of the aforementioned models, optimal values for parameters need to be found. In this section, some commonly used techniques to achieve this goal are introduced.

Training set and test set

As written in the book of Bishop [11], evaluating the performance of a machine learning algorithm on the data set it has been trained on is not a good practice. Its performance will be overestimated compared to its real one on some unseen data. This phenomenon is called over-fitting. The data set is generally split into two subsets called the training set and the test set to avoid this problem. The elements of the training set are used to train each model whereas the elements of the test set are used to evaluate their performance.

Training set, validation set and test set

As the goal is to find the most suited model and the optimal values of its parameters for a given data set, several models with several configurations will be trained on the training set and evaluated on the test set [11]. As this procedure is repeated several times to find the best model and its optimal parameters, it is likely that the performance of the aforementioned model is overestimated on the test set. As over-fitting may have occurred on the test set, generally, when a model should be selected among several, the data set is split into three subsets: a training set, a validation set and a test set. As in the previous paragraph, the first set is used to train each model. The second subset is used to select the most suited model (based on its performance on this set). The third subset it used to evaluate the performance of the selected model (on unseen data).

Cross-validation

According to Bishop et al. [11], splitting the data set into three subsets has also some drawbacks. Bishop et al. claimed that, as the data set is a finite set in many machine learning problems, one wants to increase the size of the training set in order to be able to train the model on a representative set. One also wants to increase the size of the validation set to obtain a relevant evaluation of the performance of each model and be able to select the best one. In order to solve the above mentioned problem, a technique called cross-validation is generally used. In this technique, the test set is not affected. The former training set and the former validation set are merged. The resultant set is split into K subsets of equal size, where K is an integer selected by the user. Each model is then evaluated on each of the K subsets (for each evaluation, the model has previously been trained with the elements of the $K-1$ remaining subsets). Finally, for each model, its performance on all the K subsets is averaged. When the integer K is selected, this technique is called K-fold cross-validation. When the data set is small, sometimes, $K = L$, where L is the number of elements of the data set not in the test set. This particular instance of K-fold cross-validation is called the leave-one-out technique. The main disadvantage of K-fold cross-validation is its induced computational cost: the cost related to the model selection is multiplied by a coefficient proportional to K.

Consideration of the order of the elements in the data set

In 2008, Bettenburg et al. [8] published a paper on the impact of duplicate bug reports in ITS. In their paper, they showed that, by adding more information related to a bug, duplicate bug reports could be used to fix a bug more efficiently. They also showed that, by merging the data inside the different duplicate bug reports, the performance of a classifier intending to solve the automatic bug assignment problem might be improved. They used a new procedure

to evaluate the performance of their classifiers. First, they sorted the bug reports by their reporting dates (in the chronological order). Next, they split their data set into $K+1$ subsets of equal size. They then used K iterations to evaluate the performance of each classifier. During the iteration $i, i \in \{1; \cdots; K-1\}$, the ordered elements of the first i subset(s) is/are used as a training set whereas the $i+1$ subset is used as a test set. During the iteration $i+1, i \in \{1; \cdots; K-1\}$, the $i+1$ subset is added to the training set of the iteration i and the test set is now replaced by a new $i+2$ subset of the data set. Finally, as in cross validation, the performance of each model on all the K test sets is averaged.

Based on cross validation, the procedure of Bettenburg et al. [8] could be easily extended for model selection. First, the bug reports could be sorted by their reporting dates (in the chronological order). Next, a test set (the last bug reports of the data set) could be extracted. As in cross-validation, the procedure of Bettenburg et al. could then be applied to select the best performing model. Finally, the performance of the best model might be evaluated on the test set.

Grid search

Each machine learning algorithm has generally a set of parameters Θ which values are found during the training phase by solving an optimization problem. Each machine learning algorithm has also a set of hyper parameters λ. Finding good values for the aforementioned parameters is called "hyper-parameter optimization" [7]. The problem consists of finding the values which minimize the expectation of an error criterion for the parent distribution of the data set. As the parent distribution of the data set is unknown, cross validation is generally applied instead. One has to select a subset of Λ (the set of all the possible values of λ) and find which element of the subset achieves the best average performance on the validation sets of cross validation. Combining grid search and manual search is the most used technique to solve this problem. Let t be the number of hyper parameters to tune. Before using grid search, a set of values L_i has to be selected for each hyper parameter $\lambda_i, i \in \{1; \cdots; t\}$. When applying grid search, the model will be trained and evaluated with all the elements of the following t-ary Cartesian product: $L_1 \times \cdots \times L_t$. The number of models with different configurations to train and evaluate is therefore $|L_1 \times \cdots \times L_t| = \prod_{i \in \{1; \cdots; t\}} |L_i|$. As can be seen, the number of trials increases exponentially with respect to the number of hyper parameters t. Manual search is generally used to define the different sets $L_i, i \in \{1; \cdots; t\}$. For each selected machine learning algorithm, this technique should be applied to find its best configuration.

Random search

Random search assumes that all the trials are independent and identically distributed [7]. In this technique, based on the multivariate uniform distribution, a chosen number of elements of Λ are randomly selected. The model to tune is trained and evaluated with each of these possible configurations. According to Bergstra et al. [7], random search has the same advantages as grid search. Nevertheless, when the number of hyper parameters to tune is high, it is more efficient than grid search because several parameters have a minor impact on the performance of the model and grid search wastes a fraction of its trials on these parameters. However, Bergstra et al. also claimed that random search is slightly less efficient than the combination of manual search and grid search.

2.3.5 Learning curves

Learning curves are usually used to show the impact of a learning effort on the performance of a system [27]. In machine learning, it generally consists of plotting the accuracy of an algorithm on a test set over the size of the training set. When using neural networks, sometimes, one might plot the error of the algorithm on the test set over the number of iterations of the algorithm.

Plotting on the same chart the accuracy or the error on the training set could be used to know if increasing the size of the training set could have a major positive impact on the performance of the algorithm. If the difference between the performance of the model on the training set and the performance of the model on the test set is decreasing as the size of the training set is increasing, and, both curves are relatively close, adding more data to the training set might be unnecessary. However, if the performance of the model on the test set is much more lower than the performance of the model on the training set, and, the difference between both of them decreases when the size of the training set is increasing, adding more data to the training set could be useful.

2.3.6 Evaluation

In this section, some metrics generally used to evaluate some text classification models will be presented.

Accuracy

With the accuracy metric, which is also called "[t]op N rank" [41], if the developer who really fixes the bug is in the N developers recommended by the algorithm, the prediction of the algorithm is considered as a success. Otherwise, it is considered as a failure. The accuracy is the number of correct predictions (in our case, the predictions where the developer who has eventually fixed the bug is in the N developers recommended by the algorithm) divided by the total number of predictions. An algorithm trying to carry out the automatic bug assignment task has to try to maximize the value of the accuracy metric. This metric is calculated using the Equation 2.22:

$$accuracy = \frac{\sum_{i=1}^{m} \mathbb{1}_{CP_i}(y_i)}{m}, \tag{2.22}$$

where $accuracy$ is the value of the accuracy metric; m is the total number of predictions; y_i is the developer who has eventually fixed the i-th bug; CP_i is the set of the N developers recommended by the algorithm for the resolution of the i-th bug and $\mathbb{1}_{CP}(y)$ is an indicator function which is defined as follows:

$$\mathbb{1}_{CP}(y) = \begin{cases} 1 & , \text{ if } y \in CP \\ 0 & , \text{ if } y \notin CP \end{cases}. \tag{2.23}$$

Rank

The rank of the developer, who has eventually fixed the bug, in the predictions of the algorithm solving the automatic bug assignment problem, could be used as a metric. Using the rank is consistent as it takes into account the fact that, if a developer who should fix the bug has a good rank in the predictions of the algorithm, he or she is more likely to be selected by the bug triager. A model intending to solve the bug assignment problem has to try to minimize the value of this metric.

Mean reciprocal rank

The mean reciprocal rank (MRR) metric is the average of the inverse of the rank of the developer who has eventually fixed the bug in the predictions of the model [13]. As with the rank metric, the MRR metric is relevant because it considers that a developer who has the skills to fix a bug has to have a good rank to be assigned this task by the triager. The algorithm

which goal is to solve the automatic bug assignment problem has to maximize the value of this metric. The MRR metric could be calculated using the formula in the Equation 2.24:

$$MRR = \frac{1}{m} \sum_{i=1}^{m} \frac{1}{rank_i}, \tag{2.24}$$

where MRR is the value of the MRR metric, m is the total number of predictions and $rank_i$ is the rank of the developer who has eventually fixed the i-th bug.

Many researchers have intended to solve the automatic bug assignment problem. In almost all their publications, a new model has been introduced. Nevertheless, most of these models were based on a text classification approach.

2.4 Related work

In this section, some research related to the automatic bug assignment problem is presented. As stated by Shokripour et al. [31], most of it has only used one of the following approaches: the activity-based approach or the location-based approach.

First, several papers introducing some algorithms using the activity-based approach will be presented. Second, some models based on the location-based approach will be described. Third, a new algorithm introduced by Tian et al. [35] which combines the two aforementioned approaches will be presented. Next, some solutions relying on probabilistic graphs will be described. Finally, some work on automatic fault localization will be introduced.

2.4.1 Activity-based approach

In the activity-based approach, the algorithms make their predictions based on the expertise of the developers of the project.

Like my work, Čubranić et al. [14] treated the automatic bug assignment problem as a text classification problem in 2004. They trained a naive Bayes classifier on a training set made up of bug reports and the developers assigned to them. When evaluating the performance of the classifier, they reached 30 % accuracy on the test set. In their experiments, they measured the accuracy of their classifier with different training/test sets ratios. They obtained some relatively good results when splitting their data set into a training set made up of 90 % of their bug reports and a test set containing the remaining ones (10 % of their data set). Based on their result, in this thesis, the same proportions have been selected for the training set and the test set. Additionally, similar to one result of the aforementioned authors, the best configurations of this thesis, in terms of accuracy and MRR value, were not using stemming.

In 2006, Anvik et al. [4] used almost the same approach. Nevertheless, they used several classifiers: naive Bayes, SVM and C4.5. The latter classifier generates a decision tree which is a particular type of tree [11]. In decision trees, each interior node can be considered as a decision made on a specific feature. Each arc incident from an interior node corresponds to a set of possible values for a particular feature. Each branch of the decision tree is related to a class predicted based on some possible values of each feature. The decision tree is built based on the feature vectors of the training set. Each time a decision has to be made, based on the value of each feature of the given feature vector, a branch of the decision tree will be followed and the class related to the leaf of this branch will be the prediction of the algorithm. Anvik et al. [4] have observed that SVM has the best performance on the data set they had used. Instead of proposing only one developer, the classifier also proposed a short-list of developers and the triager had to pick one of them. The goal of the classifier of Anvik et al. was to help the triager in his or her task, not to replace him or her. In this thesis, I have similarly compared the performance of several classifiers. Like their work, the best performing classifier, in terms of accuracy, on the bug reports of the telecommunications company was SVM. In contrast to their work, in this study, the classifiers were tuned before being compared.

In their paper, Lin et al. [22] have made two experiments on a Chinese software project. First, they used SVM on the textual fields of the bug reports. They then used C4.5 on the non textual fields of the bug reports. The results of the second experiment were better than the first one because one of the non textual fields, the module id, was a good indication to know which developer had fixed a given bug. In this project, each developer had generally worked on only one module. According to the authors, the use of textual fields is therefore more relevant than the use of non textual fields. Based on their result and due to the time constraints related to this thesis work, I decided to use only the textual content of the bug reports to make my predictions.

Ahsan et al. [1] used different feature extraction techniques in 2009: they used tf-idf, and, they combined tf with LSI (with several configurations). They also used seven machine learning algorithms after having extracted the features. They obtained the best results when combining tf, LSI and SVM. Like their work, the results obtained with different feature extraction configurations and different classifiers have been compared in this thesis. On the bug reports of the telecommunications company, the best performing classifier, in terms of accuracy, was also SVM. For any of the three software projects studied in this thesis, the best feature extraction technique, in terms of accuracy or MRR value, was however not based on LSI.

In 2010, Helming et al. [18] worked on a new model which is only usable in a specific repository called UNICASE. In this repository, the issues, the bug reports, the tasks and the functional requirements are stored. The work items (the issues, the bug reports and the tasks) are linked to the functional requirements in UNICASE. The functional requirements stored in the repository are also hierarchically structured. Given a new work item, they retrieved from the repository all the work items related to the functional requirement, to the ancestors of the functional requirement and to the descendants of the functional requirement of this new work item. Their new model ranked all the developers based on the number of related work items each of them had fixed. These authors have compared the performance of their model with the predictions of several classifiers. Similar to my results on the bug reports of the telecommunications company, their best performing classifier was SVM.

In 2011, Anvik et al. [5] made a more thorough study than their previous one [4]. In their paper, they focused on the recommender systems that might help to process the bug reports. For instance, these types of recommender systems could recommend a developer to fix a bug, recommend the component in which the bug related to a bug report is located or recommend the developers who might be interested in the resolution of a bug report. According to them, before building such a system, six questions should be considered where one of them is related to what machine learning algorithm that should be used. Like my results on one of the three studied projects in this thesis, in their work, they decided to use SVM because they obtained the best results with this classifier after having compared the performance of five machine learning algorithms on two open source software (OSS) projects. After having answered the aforementioned six questions, they evaluated their recommender system on 5 OSS projects. In their study, they only considered the automatic bug assignment to developers problem. They also implemented a tool especially for the bug triagers. This tool was an interface between an ITS and the triagers which displayed the predictions of a recommender system. This interface was tested by four triagers during four months. They reported that the outcome was positive. Finally, they described two techniques to help the triagers to configure such a recommender system. The first technique consists of grouping the bug reports to allow the triagers to define some heuristics used by the recommender system to automatically find the labels in the bug reports (in the context of automatic bug assignment, the developers who eventually fixed each bug) before its training. The second technique was related to the selection of the bug reports the recommender system should be trained on.

Xie et al. [37] introduced Developer REcommendation based on TOpic Models (DRETOM) in 2012. First, they extracted the developers who had written some comments

in each previously fixed bug report. Their model then used topic modeling to find the main topic of each previously fixed bug report. Finally, using a probabilistic model, DRETOM ranked each developer based on its prior contributions to each topic (to the previously fixed bug reports of each topic) and the topic proportions of a new bug report. Based on their work, for the two OSS projects studied in this thesis, only the bug reports with a FIXED resolution, and, with a RESOLVED, VERIFIED or CLOSED status have been downloaded.

In 2013, Alenezi et al. [2] compared the use of five feature selection techniques before applying the naive Bayes machine learning algorithm. Like for two of the three projects studied in this thesis, they obtained their best results with the χ^2 feature selection technique.

In 2015, Shokripour et al. [30] extracted the identifiers and the nouns from the code base, and, associated them with their authors. The recommendations of their model were using the similarity between the nouns and the identifiers used in a new bug report, and, the same type of data used by each developer in the code base. In their similarity index, they took into account the time when each noun was used (based on the reporting date of the bug report in which it had been found). Their index was based on the formula used to compute the tf-idf weights (Equation 2.2), the best feature extraction technique, in terms of accuracy, on two of the three software projects studied in this thesis.

The same year, using two instance selection methods and two feature selection methods, Xuan et al. [38] tried to find the optimal data reduction rates for both types of techniques. An instance selection technique consists of filtering some elements of the data set which could mislead the prediction of a machine learning algorithm. According to the authors, keeping 30 % of the features and 50 % of the bug reports were good choices. With these rates, they then compared the results obtained with 4 feature selection methods and 4 instance selection methods. They achieved the best results with χ^2 as a feature selection method and Iterative Case Filter (ICF) as an instance selection method. They found that, in the context of automatic bug assignment, feature selection techniques might increase the accuracy of a classifier whereas instance selection techniques might decrease the accuracy of a classifier. They then compared the accuracies of three classifiers (SVM, kNN and naive Bayes) when combining ICF and χ^2. They obtained the best results with naive Bayes. They also noticed that combining both types of data reduction techniques might increase the accuracy. They reported that using both types of data reduction techniques in different orders had an impact on the performance of the classifiers. They therefore trained a classifier to predict which order should be used to get the best results (which type of data reduction technique should be used first). Like their work, the application of some feature selection techniques on the bug reports of an OSS project has increased the accuracy of a classifier used in this thesis.

In 2015, Jonsson et al. [20] tried to solve the automatic bug assignment to teams problem using around 50 000 bug reports of 2 companies. These bug reports belonged to 5 projects of these 2 companies. In their study, they conducted 5 experiments. The first one was to compare the performance of 28 classifiers on the 5 data sets. They then applied stacked generalization (an ensemble method algorithm) using 3 configurations: the first configuration used the 5 worst performing classifiers of the previous experiment, the second one used the 5 best ones and the last configuration was based on using 5 classifiers of different types (the best one of each group). In stacked generalization, a higher level machine learning algorithm is trained on the predictions of some lower level machine learning algorithms. In their study, they obtained the best results with the last and second last configurations. Using learning curves, they also showed that the number of bug reports used to train their model had a minor impact on its performance when they exceeded a certain threshold. Furthermore, they showed that the performance of their model tended to decrease when it was trained on older data. Finally, by training their model on more and more sorted data, they showed that there was a trade off between training a model with a bigger amount of data and older data. The positive impact of using a bigger amount of data was counterbalanced by the negative impact of using older data. The first sub experiment of the preliminary experiment of this thesis (Section 3.3.1) was based on the last experiment of the aforementioned authors. In terms of accuracy, their best

performing classifier was furthermore similar to the best performing one on the bug reports of the proprietary project studied in this thesis.

As stated by Tian et al. [35], some promising results have been obtained via the activity-based approach. Nevertheless, this method does not take into account the relationship between the bug reports and the modifications made in the code base. Using the aforementioned information would probably increase the performance of the activity-based algorithms.

2.4.2 Location-based approach

By guessing the locations of the bugs, the models presented in this section make their predictions.

Linares-Vásquez et al. [23] introduced a new method in 2012. First, they extracted the features from each source code file of a version of a software product using LSI. In their approach, each file was considered as a document. Given a new bug report or a new feature request, they extracted its features using LSI as well, and, they ranked all the source code files based on their similarities with the new change request using IR. They then selected the most similar source code files and extracted their authors from their headers. Based on these extracted data, the developers were ranked. In this thesis, LSI was one of the feature extraction techniques applied on the bug reports of the three studied software projects.

In 2014, Wang et al. [36] introduced FixerCache. According to the authors, many developers work only on few components of a software project. For each component, they therefore estimated the probability of each developer to fix a bug based on his or her activity(ies) related to this component in the ITS during the last days. Based on the value of the component field in a new bug report, they used the previously estimated probabilities of the given component to make their recommendations. The above mentioned authors compared the performance of their model with two of the six classifiers used in this thesis.

As written in the paper of Tian et al. [35], the location-based algorithms have achieved good results. Nevertheless, the quality of the recommendations of these algorithms is highly based on the guesses made on the location(s) of the bug. If the algorithm does not find the correct location(s), it is likely that the second step of the algorithm will fail.

2.4.3 Combined activity-based and location-based approaches

This section will present the algorithm introduced by Tian et al. [35] combining the activity-based and the location-based approaches.

The paper of Tian et al. [35] was based on the fact that both aforementioned approaches have advantages. They should therefore be both used to solve the automatic bug assignment problem. In their paper, their model was based on 16 features. These features learned the similarity between each bug report and each developer. The twelve first features were related to the activity-based approach whereas the four last features were related to the location-based approach. They have evaluated the performance of their model on some bug reports of three OSS projects: Eclipse JDT, Eclipse SWT and ArgoUML. They have compared the performance of their model with two instances of their model using only the subset of features related to one of the two approaches. They have proved that a model combining the two approaches achieves better performance than a model using only one of the two approaches. They have also compared their model with two state of the art models using only one of the two approaches. In almost all the cases, their model has achieved better performance than the two other algorithms. Finally, they have also identified the features among the 16 features which had played a major role in the predictions of their model. In this thesis, the method used by Tian et al. to build their data sets has also been applied: the bug reports have thus been sorted based on their reported dates.

2.4.4 Use of tossing graphs

In this section, some publications intending to solve the automatic bug assignment problem using some probabilistic graphs will be introduced.

In 2009, Jeong et al. [19] used probabilistic graphs to store the tossing events of bug reports. In these graphs called tossing graphs, the nodes represent the developers and the weights on the edges estimate the tossing probabilities between them. Their model was based on Markov chains. For visualization purpose and in order to improve the quality of their model, they removed the edges with low probabilities and the edges representing a small number of tossing events between some pairs of developers. According to the authors, the tossing graphs could be used as a management tool to visualize the main interactions between the developers. Thanks to these graphs, using the weighted breadth first search (WBFS) algorithm, they could also predict the developer to which an assigned bug report could be tossed. Using their model and WBFS, they have shown that it was possible to reduce the number of tossing events when processing a bug report. They have also tried to combine the predictions made by their tossing graph with the predictions made by two classifiers. One of these machine learning algorithms has been used in this thesis. Thanks to the use of their graph, the accuracy of the predictions of the two classifiers were improved.

In 2012, Bhattacharya et al. [9] tried to improve the model of Jeong et al. [19]. They used five machine learning algorithms: naive Bayes, Bayesian networks, C4.5 and two SVM classifiers (differing in their kernel functions) as well as a tossing graph to solve the automatic bug assignment problem. They have used some information retrieval techniques (stop words removal, non-alphabetic words removal, stemming and tf-idf) to extract and clean the data from the bug reports. The goal of their tossing graph was to store the fact that some bug reports were misassigned to some developers, the activity of each developer in the project (the degree to which they had contributed to the project) as well as the product and component of each bug report. They have also used an incremental learning approach to improve the accuracy of their model. They have concluded that the performance of each classifier is highly correlated with the data set used to train it. According to them, no particular machine learning algorithm can therefore always be recommended to solve the automatic bug assignment problem. Like their work, the naive Bayes and SVM classifiers have been used in this thesis.

2.4.5 Automatic fault localization

In this section, some publications aiming to find the location of a bug are presented. This problem is called automatic fault localization (AFL).

In 2012, Somasundaram et al. [32] focused on the component assignment problem. In this problem, the goal is to find the software component in which the bug related to a bug report should be fixed. In this context, they compared the results obtained using three different combinations: tf-idf and SVM, latent Dirichlet allocation (LDA) and SVM, and, LDA and Kullback-Leibler (KL) divergence. The LDA model is a specific topic modeling technique [12]. Topic models are based on the fact that documents related to certain topics are likely to contain some terms related to some specific contexts. In topic models, it is assumed that each document deals with several topics. Thanks to topic modeling, the topic proportions of an unseen document can be inferred. In LDA, it is assumed that each document is a combination of a subset of the topics extracted from the whole corpus. Each word of a document is a sample of a word-topic distribution. Based on this generative model, the word-topic distributions and the topic proportions of each document of a corpus are inferred using some other algorithms. In the study of Somasundaram et al. [32], the KL divergence, a metric generally used to compute the difference between two distributions, was applied to classify the bug reports. They measured the difference between the average topic proportions of the bug reports of each component and the topic proportions of a new bug report. This information was used to make the predictions. They found that they had less performance issues on the

less represented components with LDA and KL divergence than with the two other methods. As in two of the three projects studied in this thesis, they found that, in average, in terms of accuracy, their best results were nevertheless obtained when using the tf-idf weights.

In 2013, Thomas et al. [34] worked on the effect of a classifier configuration on its performance in the context of AFL. For each of the 3 OSS projects used in their work, they have compared the results obtained with 3172 configurations. In their different configurations, they have used different textual fields of the bug reports (the title, the description or both), different features related to the source code files, different pre-processing techniques (different combinations among splitting, stemming and stop words removal), different feature extraction techniques (tf-idf, LSI and LDA) and different values for the parameters of each of these techniques. They have also tried to find the source code file which may contain a bug using some software metrics. They found that the classifier configuration has an impact on the results they could reach with it. They have also tried to combine the predictions of different combinations of the aforementioned classifiers. They used two approaches. In the first one, they combined the predictions of the best individual classifiers trained on different types of data from the bug reports and the source code files. In the second one, they randomly selected the individual classifiers trained on different types of data from the bug reports and the source code files. They also used two techniques: the Borda Count technique (which is based on the ranks of the predictions of each individual classifier) and the score addition technique (which takes into account the scores of the predictions of each individual classifier). As stated in the Chapter 1, the topic of this thesis was based on the claim of the above mentioned authors on the impact of a classifier configuration on its performance. Based on this result, the goal of this thesis was nevertheless to introduce a systematic method to find some of these best configurations in the context of automatic bug assignment (not in the context of AFL).

Ye et al. [39] worked on a new model to solve the AFL problem in 2014. Their model gave a suspiciousness score to each file of the project using a linear combination of six features. The first feature of their model was based on the textual similarity between a new bug report and each source code file. The second one used the similarity between the concatenation of the documentation of each variable used in each method of each class and the documentation of the classes inside each source file, and, a new bug report. The third one was using the similarity between the set of bug reports which have been fixed via some modifications in each source file and a new bug report. The other features were respectively using the potential occurrence of a class name inside a new bug report, the date of the last fix made on each source file and the number of time(s) each source file had already been fixed. They evaluated their approach on 6 OSS projects. Like their work, in this thesis, for the Eclipse JDT and Mozilla Firefox projects, only the bug reports with a RESOLVED, VERIFIED or CLOSED status, and, with a FIXED resolution have been downloaded.

3 Method

In this chapter, the data sets used in this Master's thesis are first presented. The computer resources and the main libraries used to conduct the experiments of this study are then described. Next, the method used in the preliminary experiment and the main experiments of this thesis is introduced. Finally, the metrics used to evaluate my approach are presented.

3.1 Data sets

In this thesis, all the experiments have been conducted on some bug reports of the ITS of three software projects: a project of a telecommunications company, Eclipse JDT[1] and Mozilla Firefox[2]. These three data sets contain respectively 66 066, 24 450 and 30 358 bug reports.

The first project has mainly been selected because it is a proprietary project and few papers (compared to the OSS projects) have been written on trying to solve the automatic bug assignment problem for these types of projects [18, 20, 22, 34].

The other projects have been selected for various reasons. First, these projects (Eclipse and Mozilla) have been used in many prior works to train and evaluate the models introduced [1, 2, 4, 5, 8, 9, 14, 19, 30, 32, 34, 35, 36, 37, 38, 39]. Training and evaluating the models introduced in prior works would be simplified. Then, since these projects were likely to be representative instances of OSS projects, the findings made on these data sets should be generalized to any large-scale OSS development project. Finally, as stated in the Chapter 1, since the data sets are open source, they can be easily accessed by other researchers to reproduce the results of this study.

In the context of this thesis, only the textual content of the bug reports has been used: only the title and the description of each bug report have been considered (Section 2.1.1).

For the project of the telecommunications company, only the bug reports which were in a FINISHED state have been downloaded (Figure 2.4). This choice was based on a discussion with an expert triager of the company. According to him, all the fixed bug reports of the ITS should be in this state.

For the Eclipse JDT and Mozilla Firefox projects, only the bug reports with a FIXED resolution, and, with a RESOLVED, VERIFIED or CLOSED status have been downloaded (Figure

[1] http://www.eclipse.org/jdt
[2] http://www.mozilla.org

Project	Time range	# bug reports	Size of the vocabulary	# of classes
A project of a telecoms company	2013/01/01 - 2017/02/13	66066	1406720	18
Eclipse JDT	2001/10/10 - 2017/01/25	24450	148589	186
Mozilla Firefox	2001/04/07 - 2017/02/09	30358	83795	1483

Table 3.1: Data sets used

2.3). More precisely, after having formulated a query to retrieve the aforementioned bug reports, the relevant data have been extracted from the results via the use of web scraping. According to Bhattacharya et al. [9], not only considering FIXED bug reports adds noise to the data set. In their papers, Ye et al. [39] and Xie et al. [37] have made the same choices regarding their data sets: they have only downloaded the bug reports which had a RESOLVED, VERIFIED or CLOSED status, and, a FIXED resolution.

Solving the automatic bug assignment to teams problem has been investigated via the bug reports of the telecommunications company whereas the bug reports of the OSS projects have been used to solve the automatic bug assignment to developers problem. Both problems are text classification problems. The only major difference is thus the number of classes (lower in the context of automatic bug assignment to teams).

Concerning the telecommunications company, the bugs reported until 2017/02/13 have been extracted from its ITS. Regarding the Eclipse JDT and Mozilla Firefox projects, the same type of data has been respectively downloaded until 2017/01/25 and 2017/02/09. After having pre-processed them, the bug reports with an empty title field, an empty description field or an empty assignee field were removed from the data sets of Eclipse JDT and Mozilla Firefox. The details of the three data sets used in this thesis are described in the Table 3.1.

The same approach as the one of Tian et al. [35] has been used to build the data set: the bug reports have been sorted based on their reported dates.

3.2 Experimental setup

All the experiments of this thesis have been conducted on a virtual machine with the following specifications:

- x86_64 (architecture),
- 8 Intel® Xeon® Processor E5-2680 v3 @ 2.50 GHz (8 vCPUs),
- 31.3 GiB (RAM).

All the source code related to the experiments of this thesis has been written in Python 3.6.

The library NLTK[3] has been used to pre-process the bug reports in all the experiments of the thesis. NLTK is a well-documented natural language processing (NLP) Python library. NLTK offers the possibility to easily apply some pre-processing techniques such as tokenization, stemming, POS tagging or lemmatization on textual data. NLTK is a cross platform and open source library. The creators of the library have written a book [10] which aims to be an introduction to NLP and which is mostly based on Python and NLTK.

In all the experiments of the thesis, the classifiers implemented in scikit-learn[4] have been used. The implementations of the feature extraction and the feature selection techniques of this library have also been used. scikit-learn is a Python machine learning library. In this widely used open source library, many machine learning algorithms and techniques have been implemented. A large community is continuously improving scikit-learn and this library is easy-to-use.

[3] https://www.nltk.org/
[4] http://scikit-learn.org/stable/

3.3 Preliminary experiment

Before conducting the main experiments of this thesis, a preliminary experiment based on one of the findings of L. Jonsson et al. [20] had been carried out. In the context of automatic bug assignment to teams, using learning curves (Section 2.3.5), they showed that the performance of machine learning algorithms tend to decrease when they are trained using an amount of sorted data exceeding a certain threshold. If a machine learning algorithm is trained on too many bug reports, its performance could decrease. On the other hand, if there are not enough bug reports in its training set, its performance might be poor.

The goal of this preliminary experiment was to find the optimal number of bug reports that should be used to train a machine learning algorithm.

The experiment was made up of two sub experiments. In each of them, tokenization (based on the function `word_tokenize` of the module `nltk.tokenize` of NLTK) has been applied on the textual content (the title and the description fields) of each bug report (Section 2.2.2). Only tokenization has been used so that this type of technique would not have a major impact on the results. Next, the features of each bug report were extracted by using the tf weight of each token in each bug report (Section 2.3.2). TF weights have been used because it was the most intuitive way to extract features from some textual content and this step should not have played a major role in the results.

Several SVM classifiers with a linear kernel have been trained and evaluated. The decision related to the selected classifier was based on three facts. First, in several papers, the performance of many classifiers including SVM have been compared: in most of these publications, SVM has outperformed the other classifiers [4, 20, 1, 18]. Using directly SVM might therefore be representative of the best performance that could be reached with any other classifier. Second, in many scientific publications, the authors have compared the performance of their newly introduced models with the one reached with a SVM-based classifier in order to know if the predictions of their models were reasonable [36, 23, 18]. This means that they have considered the SVM-based classifiers as a baseline. Third, some authors have also decided to directly use SVM for the classification part of their model [22, 32, 39]. Their choice was based on the fact that, in many prior works, SVM had outperformed some other classifiers. Due to the aforementioned three reasons, I have decided to use several linear SVM classifiers in both sub experiments. More precisely, I have used the default configuration of the `LinearSVC` class of scikit-learn. This class implements a linear SVM based on the liblinear[5] open source library.

3.3.1 First sub experiment

The first sub experiment was based on the last experiment of L. Jonsson et al. [20]. Each set of bug reports was split into K subsets of equal size. In the remainder of this report, the term 'fold' will be used to designate any of the subsets defined when applying the cross-validation technique (Section 2.3.4). The performance of the machine learning algorithm was then evaluated several times on the bug reports of each $Fold_i, i \in \{2; \cdots; K\}$ (on the bug reports of each i-th fold). For each of the aforementioned i-th fold, the machine learning algorithm was trained and evaluated $i-1$ times using each element of the following set:

$$\{Folds_{j,i-1}, j \in \{1; \cdots; i-1\}\}, \tag{3.1}$$

where $Folds_{k,l}, k \leqslant l$ could be formally defined as follows $Folds_{k,l} = \bigcup Fold_{m,m\in\{k;\cdots;l\}}, k \leqslant l$.

The selected machine learning algorithm has been trained $\sum_{i=2}^{K} i-1 = \sum_{i=1}^{K-1} i = \frac{(K-1)K}{2}$ times. The details related to the different training sets and test sets used in the context of this sub experiment are in the Table 3.2.

[5] http://www.csie.ntu.edu.tw/~cjlin/liblinear/

Number of runs	Size of each training set	Training set	Test set
$K-1$	1 fold	$Folds_{1,1}$	$Fold_2$
		$Folds_{2,2}$	$Fold_3$
		$\vdots$	$\vdots$
		$Folds_{K-2,K-2}$	$Fold_{K-1}$
		$Folds_{K-1,K-1}$	$Fold_K$
$K-2$	2 folds	$Folds_{1,2}$	$Fold_3$
		$Folds_{2,3}$	$Fold_4$
		$\vdots$	$\vdots$
		$Folds_{K-3,K-2}$	$Fold_{K-1}$
		$Folds_{K-2,K-1}$	$Fold_K$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
2	$K-2$ folds	$Folds_{1,K-2}$	$Fold_{K-1}$
		$Folds_{2,K-1}$	$Fold_K$
1	$K-1$ folds	$Folds_{1,K-1}$	$Fold_K$

Table 3.2: The different training sets and test sets of the first sub experiment of the preliminary experiment

For each training set size, the average performance of the model was computed based on its performance on the different test sets when trained on a training set having that given size. Thanks to all the averages previously computed, a set of pairs $PERF\_TE\_SET_K = \{(tr\_s\_size_1, te\_s\_avg\_perf_1); (tr\_s\_size_2, te\_s\_avg\_perf_2); \cdots; (tr\_s\_size_{K-2}, te\_s\_avg\_perf_{K-2}); (tr\_s\_size_{K-1}, te\_s\_avg\_perf_{K-1})\}$, where $tr\_s\_size_i$ is the number of bug reports in i folds and $te\_s\_avg\_perf_i$ is the average performance of the model when trained on a training set made up of i folds, was built.

The average performance of the model on the different training sets of same size was also computed. Another set of pairs $PERF\_TR\_SET_K = \{(tr\_s\_size_1, tr\_s\_avg\_perf_1); (tr\_s\_size_2, tr\_s\_avg\_perf_2); \cdots; (tr\_s\_size_{K-2}, tr\_s\_avg\_perf_{K-2}); (tr\_s\_size_{K-1}, tr\_s\_avg\_perf_{K-1})\}$, where $tr\_s\_size_i$ is the number of bug reports in i folds and $tr\_s\_avg\_perf_i$ is the average performance of the model on a training set made up of i folds, was built.

Based on the results of L. Jonsson et al. [20], the bigger the test fold is, the more the performance of the model should decrease (on its last predictions) as the last predictions of the model are made using older data.

Due to this idea, I decided to plot several learning curves for different values of $K, K \in \{4; 6; 8; 10; 15; 25; 50\}$. For each $K \in \{4; 6; 8; 10; 15; 25; 50\}$, two learning curves were plotted in the same chart based on the elements of $PERF\_TR\_SET_K$ and $PERF\_TE\_SET_K$. The standard deviation related to each point plotted in each chart has also been represented. Based on the obtained results, if the time needed to train the machine learning algorithm was reasonable, the model could be periodically re-trained so that its performance would not continuously decrease.

The Figure 3.1 depicts the method used in the first sub experiment of the preliminary experiment of this thesis.

Figure 3.1: The method used in the first sub experiment of the preliminary experiment

3.3.2 Second sub experiment

In the second sub experiment, I assumed that the machine learning algorithm would be periodically re-trained and that making the choice related to the optimal size of the training set was more relevant when evaluating it on recent data. Only the last fold was therefore used to plot the learning curves. In this sub experiment, as shown in the Table 3.3, the model has been evaluated $K-1$ times on the $Fold_K$ (the K-th fold). Each evaluation was made using a model trained on each element of the following set:

$$\{Folds_{j,K-1}, j \in \{1; \cdots ; K-1\}\}. \tag{3.2}$$

Based on the idea mentioned in the previous section, several learning curves have been plotted for different values of $K, K \in \{4;6;8;10;15;25;50\}$. For each $K \in \{4;6;8;10;15;25;50\}$, two learning curves have been plotted in the same chart based on the elements of $PERF\_TR\_SET_K$ and $PERF\_TE\_SET_K$ (the two sets have been defined in the previous section).

The Figure 3.2 shows the method used in the second sub experiment of the preliminary experiment of this study.

Number of runs	Size of each training set	Training set	Test set
1	$K-1$ folds	$Folds_{1,K-1}$	$Fold_K$
1	$K-2$ folds	$Folds_{2,K-1}$	$Fold_K$
⋮	⋮	⋮	⋮
1	2 folds	$Folds_{K-2,K-1}$	$Fold_K$
1	1 fold	$Folds_{K-1,K-1}$	$Fold_K$

Table 3.3: The different training sets and test sets of the second sub experiment of the preliminary experiment

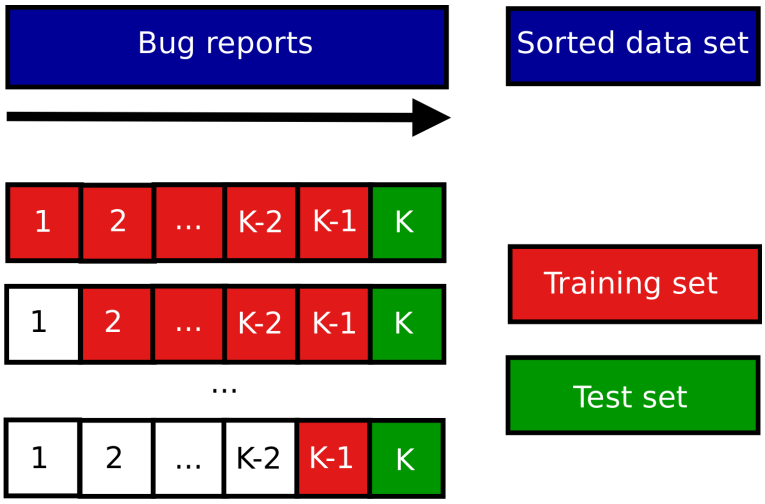


Figure 3.2: The method used in the second sub experiment of the preliminary experiment

3.4 Main experiments

In this section, the main experiments which have been conducted in the context of the thesis are described. The results related to each experiment is used to answer each research question.

In all the experiments described below, first, each data set was split into a training set (90 %) and a test set (10 %). This training set/test set ratio has been selected for mainly two reasons. First, in the data set of the telecommunications company, 10 % represent approximately 5 months of bug reports. I judged that evaluating the performance of a classifier on the issues reported during the last 5 months of the data set would probably be representative. Second, in their experiments, Čubranić et al. [14] measured the accuracy of their classifier with different training/test sets ratios. They obtained some relatively good results with this particular ratio. As a consequence, I decided to select these proportions for the training set and the test set.

As explained in the Section 2.3.4, the training set was then split into $K+1$ folds (based on the implementation of the class `TimeSeriesSplit` of scikit-learn). In each experiment, each configuration of a classifier was trained and evaluated K times: the performance of each machine learning algorithm was evaluated exactly one time on the bug reports of each $Fold_j, j \in \{2; \cdots; K+1\}$. Each model evaluated on the fold $Fold_j$ had been trained on $Folds_{1,j-1}$. For each model, the average of its performance related to each evaluation was computed. Finally, the model with the best performance was selected. In this thesis, the value of K was set to 10.

The Figure 3.3 illustrates the method used in each main experiment of this thesis.

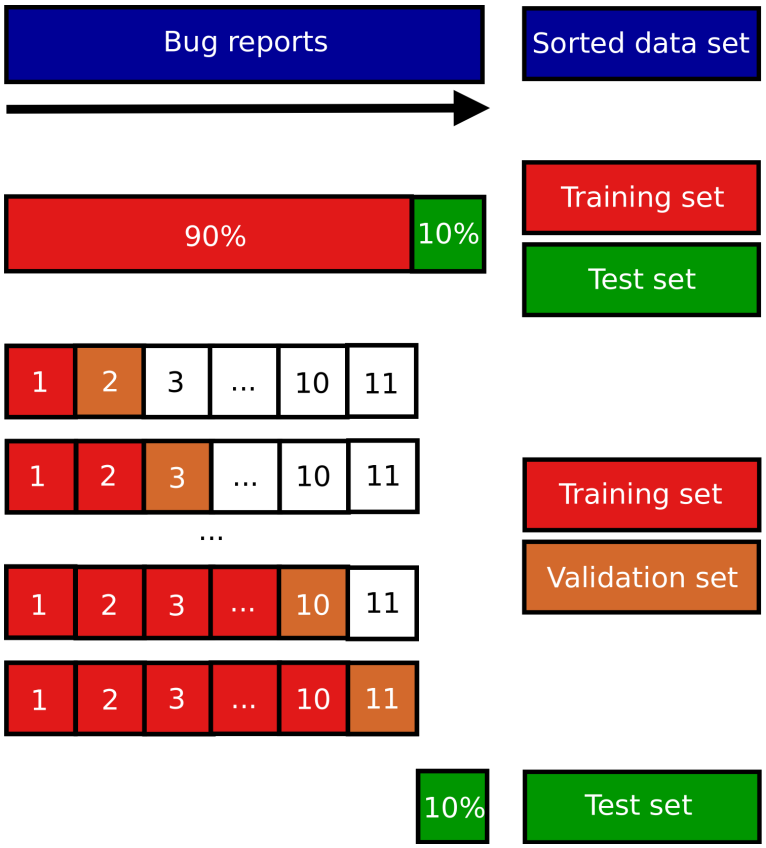


Figure 3.3: The method used in all the main experiments

In all the experiments, at least tokenization (based on the function `word_tokenize` of the module `nltk.tokenize` of NLTK) has been applied on the textual content of each bug report to later be able to extract the features from them (Section 2.2.2).

For the same reasons mentioned in the Section 3.3, in the first three experiments, the default configuration of the scikit-learn implementation of a linear SVM (the `LinearSVC` class) has been used for the classification.

3.4.1 Experiment 1

The goal of this experiment was to find which combination of pre-processing techniques should be used to obtain the best results (Section 2.2.2).

In the context of this experiment, 96 combinations have been tested. These combinations were based on six parameters: five of them had two possible values whereas one of them had three possible values.

Cleaning or not cleaning the bug reports was the first parameter. Cleaning the bug reports consists of removing the elements which appear in a considerable number of bug reports (the titles in the descriptions of the bug reports, the subtitles, etc.) using some regular expressions, converting the HTML character entity references to Unicode characters, solving the encoding problems, etc.

The second parameter was related to the use of a stemmer (the class `PorterStemmer` of the module `nltk.stem.porter` of NLTK), a lemmatizer (the class `WordNetLemmatizer` of the module `nltk.stem.wordnet` and the POS given by the function `pos_tag` of the module `nltk.tag` of NLTK) or nothing (three possible values).

Parameter	Possible value
Cleaning bug reports	Yes/No
Stemmer vs. Lemmatizer	Stemmer/Lemmatizer/Nothing
Stop words removal	Yes/No
Punctuation characters removal	Yes/No
Numbers removal	Yes/No
Conversion to lower case	Yes/No

Table 3.4: The possible values for the parameters of the experiment 1

Removing or not removing stop words (using the English stop words list of NLTK) was also considered as a parameter.

The fourth parameter was based on the choice of removing or not the punctuation characters.

Removing or not the numbers from the textual content of the bug reports was used as a fifth parameter.

The sixth parameter was based on the choice of converting or not each token to lower case.

Based on the possible choices related to the aforementioned six parameters, $96 = 2 * 3 * 2 * 2 * 2 * 2$ configurations of parameters have been obtained (Table 3.4).

In this experiment, the bug reports were pre-processed with all the different configurations. For the same reasons mentioned in the Section 3.3, the features of each bug report were then extracted by using the tf weight of each token in each bug report (Section 2.3.2). Next, 96 linear SVM classifiers (96 instances of the `LinearSVC` class) were trained and evaluated on the feature vectors extracted from the output of the distinct configurations.

3.4.2 Experiment 2

The aim of this experiment was to find the best way to extract the features from the bug reports of an ITS.

In order to do that, several configurations were tested and the one which gave the best performance was selected.

First of all, the best configuration of the experiment 1 (the best combination of pre-processing techniques) was applied on the bug reports of the ITS.

The experiment was mainly based on a comparison between the results obtained using three different representations: a Boolean representation (using the class `CountVectorizer` of scikit-learn), a representation based on the tf weights (the class `TfidfTransformer` of scikit-learn) and a representation based on the tf-idf weights (the class `TfidfTransformer` of scikit-learn) of the words in the textual content of the bug reports (Section 2.3.2).

Next, the LSI algorithm (the class `TruncatedSVD` of scikit-learn) was separately run on each of the three aforementioned representations. The NMF algorithm (the class `NMF` of scikit-learn) was also separately run on each of the three representations. Each time the LSI or the NMF algorithm was used, $f \in \{10; 30; 50; 70; 90\}$ features were selected. This choice was motivated by the time and the space required to use these algorithms on my virtual machine (Section 3.2).

33 linear SVM classifiers (33 instances of the `LinearSVC` class) were trained on the $33 = 3 + 6 * 5$ different representations of the bug reports. The details related to the aforementioned 33 configurations are provided in the Table 3.5.

Finally, all the possible pairs of extracted features were generated. For each experiment involving, either LSI or NMF, only the configuration with the optimal number of selected features was considered (the selected number of features which gave the best result). Only the performance related to the $\binom{9}{2} = 36$ combined configurations were compared. 36 linear SVM

Representation	Remaining features
Boolean	All
TF	All
TF-IDF	All
Boolean + LSI	{10; 30; 50; 70; 90}
TF + LSI	{10; 30; 50; 70; 90}
TF-IDF + LSI	{10; 30; 50; 70; 90}
Boolean + NMF	{10; 30; 50; 70; 90}
TF + NMF	{10; 30; 50; 70; 90}
TF-IDF + NMF	{10; 30; 50; 70; 90}

Table 3.5: The different configurations of the first part of the experiment 2

Representation
Boolean
TF
TF-IDF
Best Boolean + LSI
Best TF + LSI
Best TF-IDF + LSI
Best Boolean + NMF
Best TF + NMF
Best TF-IDF + NMF

Table 3.6: The different configurations of the second part of the experiment 2

classifiers (36 instances of the `LinearSVC` class) were trained on all the features extracted from these 36 combined configurations (Table 3.6).

In the context of the experiment 2, the results related to $69 = 33 + 36 = (3 + 6 * 5) + \binom{9}{2}$ configurations have been compared.

3.4.3 Experiment 3

The goal of this experiment was to know if using a feature selection technique would increase the performance of a classifier. If it was the case, the experiment would also allow us to find the size of the subset of the remaining features which should be selected to get the best results, in terms of accuracy and MRR value.

First, the best configuration of the experiment 1 (the best combination of pre-processing techniques) and the best configuration of the experiment 2 (the best feature extraction technique) were applied on the bug reports. The following feature selection techniques were then separately applied on the set of bug reports: the χ^2 test (the function `chi2` of the module `sklearn.feature_selection.univariate_selection` of scikit-learn), the ANOVA test (the function `f_classif` of the module `sklearn.feature_selection.univariate_selection` of scikit-learn), the mutual information (the function `mutual_info_classif` of the module `sklearn.feature_selection.mutual_info_` of scikit-learn) and the Recursive Feature Elimination (the class `RFECV` of scikit-learn) (Section 2.3.3). For each feature selection technique, $f \in \{0.1; 0.3; 0.5; 0.7; 0.9\}$ of the initial features were selected. In total, $20 = 4 * 5$ configurations have been compared (Table 3.7). 20 linear SVM classifiers (20 instances of the `LinearSVC` class) were trained on the remaining features of the different configurations. The performance of the best feature selection technique with the best percentage of remaining features was compared with the performance obtained without applying any feature selection technique (the performance of the best configuration of the experiment 2). If a particular

Feature selection technique	Remaining features
Chi-2	{0.1; 0.3; 0.5; 0.7; 0.9}
ANOVA	{0.1; 0.3; 0.5; 0.7; 0.9}
Mutual information	{0.1; 0.3; 0.5; 0.7; 0.9}
Recursive feature elimination	{0.1; 0.3; 0.5; 0.7; 0.9}

Table 3.7: The different configurations of the experiment 3

feature selection technique and a particular percentage of remaining features improved the performance of a linear SVM classifier, this configuration would be selected.

3.4.4 Experiment 4

The aim of this experiment was to find the best performing classifier and its best configuration on each data set. This goal was achieved by tuning a set of selected classifiers: a nearest centroid classifier, a naive Bayes classifier based on a multinomial distribution, a linear SVM classifier, a logistic regression classifier, a perceptron classifier and a classifier based on stochastic gradient descent. The aforementioned classifiers have been selected because they had been used in many prior works in the context of automatic bug assignment or they were implemented in scikit-learn.

First, the best configuration of the experiment 1 (the best combination of pre-processing techniques), the best configuration of the experiment 2 (the best feature extraction technique) and the best configuration of the experiment 3 (the potential use of a feature selection technique) were applied on the bug reports. Two strategies were then used to tune each classifier: a grid search and a random search (Section 2.3.4). For each classifier and each strategy, an upper bound of 150 configurations was selected. The choice regarding this upper bound was based on several parameters such as the specifications of my virtual machine (Section 3.2), the computational cost related to the training phase of each classifier and the time constraints related to the Master's thesis. Based on this upper bound and their relevance, for each hyper-parameter of each classifier, a set of possible values has been selected.

If it was complicated to predict the behaviour of a classifier when one of its hyper-parameter(s) was tuned, this hyper-parameter was considered as relevant in the context of this experiment. For example, if the number of iterations of the stochastic gradient descent algorithm is increased, it is well known that the performance of the classifier will increase and that the computational cost related to its training phase will also increase. In the context of this experiment, this hyper-parameter was considered as irrelevant as it should be tuned based on the needs of the customer. Tuning this hyper-parameter could be relatively easily achieved by using some learning curves (Section 2.3.4). As tuning the aforementioned hyper-parameter was highly related to the context of the problem (the needs of the customer), it was not considered.

If it was well-known that one of the possible values of a hyper-parameter would definitely increase the performance of its classifier, this value was selected. For instance, if its value is set to `"balanced"`, the parameter `class_weight` of the constructor of the class `LinearSVC` will allow the linear SVM classifier to take into account that some classes are more represented than others. As it would likely improve the performance of the linear SVM classifier, this parameter was always set to `"balanced"`.

Using the above mentioned approach, for each relevant hyper-parameter of each of the six classifiers, a set of values was built (Table 3.8). As they were considered as irrelevant in the framework of this experiment, the default values of the parameters not in the Table 3.8 were used. A grid search strategy (based on the `GridSearchCV` class of scikit-learn) was used to try to find the best configuration of each classifier. A random search strategy (based on the `RandomizedSearchCV` class of scikit-learn) requiring the same number of runs was

Algorithm	Implementation	Number of runs	Hyper-parameter
Nearest centroid classifier	`NearestCentroid`	2	`metric` ∈ {"manhattan";"euclidean"}
Naive Bayes classifier (multinomial distribution)	`MultinomialNB`	22	`alpha` ∈ `np.linspace(0,1,11)` `fit_prior` ∈ {True;False}

Linear SVM	`LinearSVC`	20	`C` ∈ `np.logspace(-4, 4, 10)`
			`loss` ∈ {"squared_hinge";"hinge"}
			`class_weight="balanced"`

Logistic regression	`LogisticRegression`	40	`dual=False`
			`C` ∈ `np.logspace(-4, 4, 10)`
			`class_weight="balanced"`
			`solver` ∈ {"newton-cg";"sag";"lbfgs"}
			`multi_class="multinomial"`
			--- 	
			`dual=True`
			`C` ∈ `np.logspace(-4, 4, 10)`
			`class_weight="balanced"`
			`solver="liblinear"`
			`multi_class="ovr"`

Perceptron	`Perceptron`	13	`penalty` ∈ {"l2";"elasticnet"}
			`alpha` ∈ `10.0**-np.arange(1,7)`
			`class_weight = "balanced"`
			--- 	
			`penalty=None`
			`class_weight="balanced"`

Stochastic gradient descent	`SGDClassifier`	120	`loss` ∈ {"hinge";"log";"modified_huber";
			"squared_hinge";"perceptron"}
			`penalty` ∈ {"l2";"elasticnet"}
			`alpha` ∈ `10.0**-np.arange(1,7)`
			`class_weight="balanced"`
			`average` ∈ {True;False}

Table 3.8: The different configurations of the experiment 4

also used to try to find the best configuration of each classifier. For each set of values, a random distribution was used to apply the random search. If a set contained some non-numeric values, a discrete uniform distribution was used. Otherwise, a continuous distribution was used to exploit the full potential of the random search strategy (Section 2.3.4). For each classifier, the best configuration among the configurations compared via the grid search strategy and the random search strategy was eventually selected.

In total, in this experiment, $434 = 217 * 2 = (2 + 22 + 20 + 40 + 13 + 120) * 2$ configurations have been compared.

3.5 Evaluation

In the preliminary experiment (Section 3.3), only the value of the top 1 accuracy metric (Section 2.3.6) has been computed.

Except for the nearest centroid classifier used in the forth experiment, in all the main experiments of this thesis (Section 3.4), the value of the top 1 accuracy metric and the value of the MRR metric (Section 2.3.6) have been computed in order to be able to make some decisions related to the selection of the different types of techniques. As the classes were not ranked during the validation and testing phases of the nearest centroid classifier implemented in scikit-learn (the class `NearestCentroid`) and adding this feature would have required significant development effort, the value of the MRR metric was not computed for this classifier.

4 Results

In this chapter, first, the results related to the preliminary experiment of this thesis (Section 3.3) are described. Second, the results related to the main experiments of this thesis (Section 3.4) are presented.

4.1 Preliminary experiment

The results related to the preliminary experiment described in the Section 3.3 are presented below.

4.1.1 First sub experiment

In this section, the results of the first sub experiment of the preliminary experiment of the thesis are presented.

The graphs in the Figures 4.1, 4.2 and 4.3 illustrate the learning curves obtained with different numbers of folds $K, K \in \{4; 8; 50\}$. For each $K \in \{4; 8; 50\}$, two learning curves are represented: the red one is related to the accuracy on the training set whereas the green one shows the accuracy on the test set. The standard deviation related to each point plotted in each chart is also represented. The graphs related to the configurations with 6, 10, 15 and 25 folds can be found in the Figures A.1, A.2, A.3, A.4, A.5 and A.6 of the Appendix A.1.

Telecommunications company

In all the subfigures of the Figure 4.1, the accuracy on the training set has raised with an increase in the size of the same set.

With less than 8 folds, the accuracy on the test set has decreased when using more bug reports. With at least 8 folds, the accuracy on the test set has fluctuated, but generally, it has increased, then, decreased. In the latter case, the accuracy has always reached its maximum when the number of bug reports in the training set was between 10 000 and 20 000.

Eclipse JDT

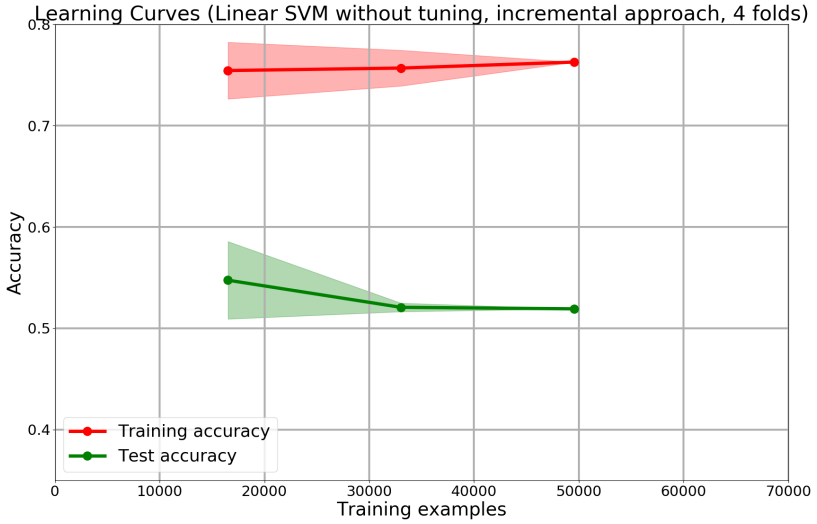
Contrary to the bug reports of the telecommunications company, in all the subfigures of the Figure 4.2, the accuracy on the training set has decreased with an increase in the size of the same set.

With no more than 8 folds, the accuracy on the test set has decreased when using more bug reports. With strictly more than 8 folds, the accuracy on the test set has fluctuated, but generally, it has increased, then, decreased: it has always reached its maximum when the number of bug reports in the training set was between 2 500 and 5 000.

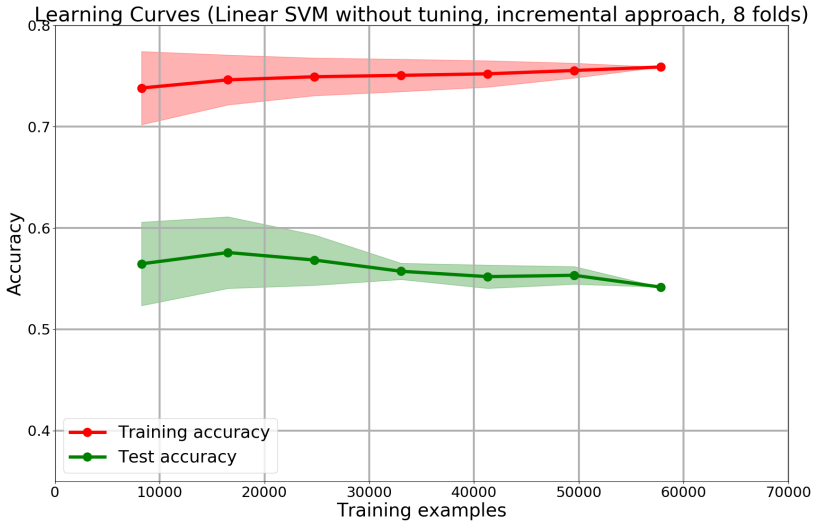
Mozilla Firefox

As with the bug reports of Eclipse JDT, in all the subfigures of the Figure 4.3, the accuracy on the training set has decreased with an increase in the size of the same set.

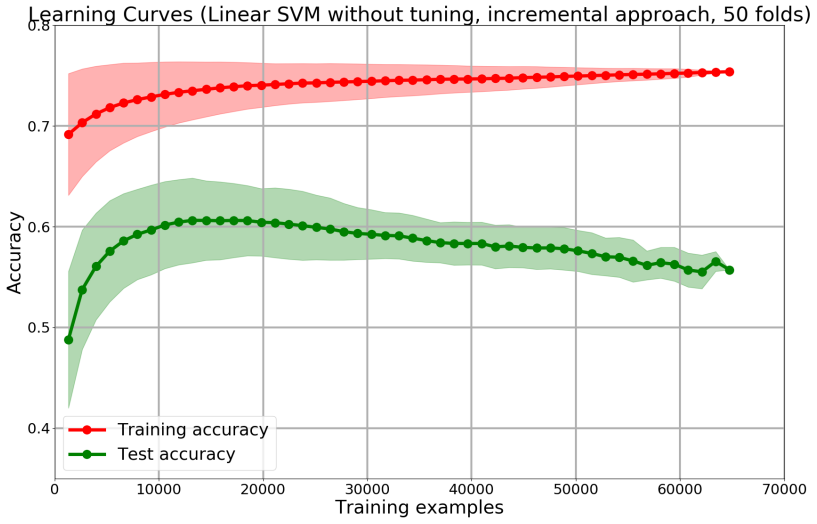
With less than 8 folds, the accuracy on the test set has fluctuated, but, been relatively constant. As with the data set of the telecommunications company, with at least 8 folds, the accuracy on the test set has fluctuated, but generally, it has increased, then, decreased: it has always reached its maximum when the number of bug reports in the training set was between 3 000 and 10 000. For convenience, in the remainder of this report, the expression 'telecommunications company' will be used to designate the data set of this company.



(a) 4 folds

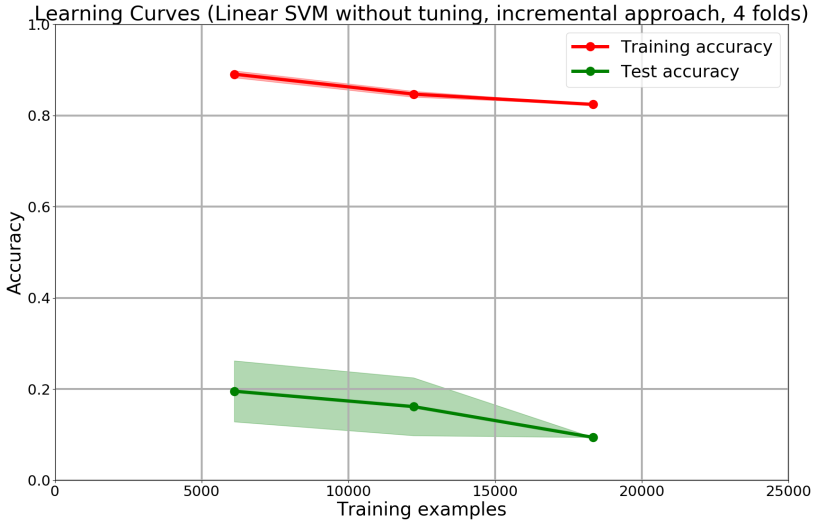


(b) 8 folds

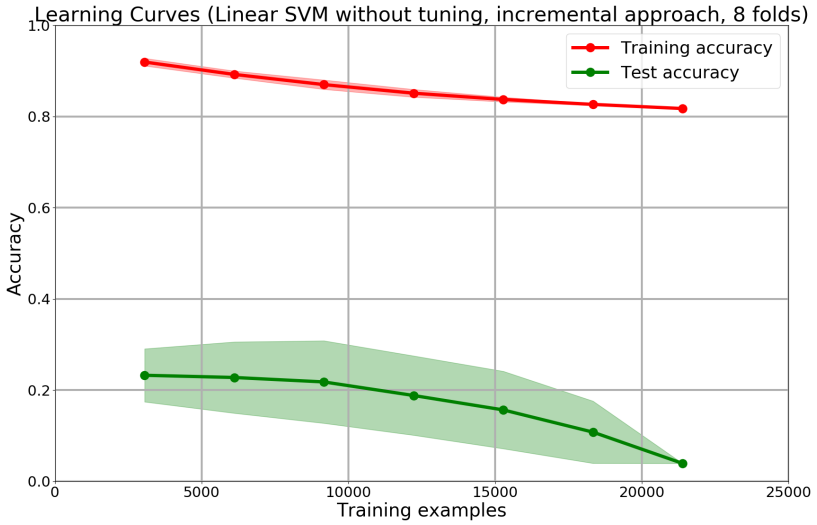


(c) 50 folds

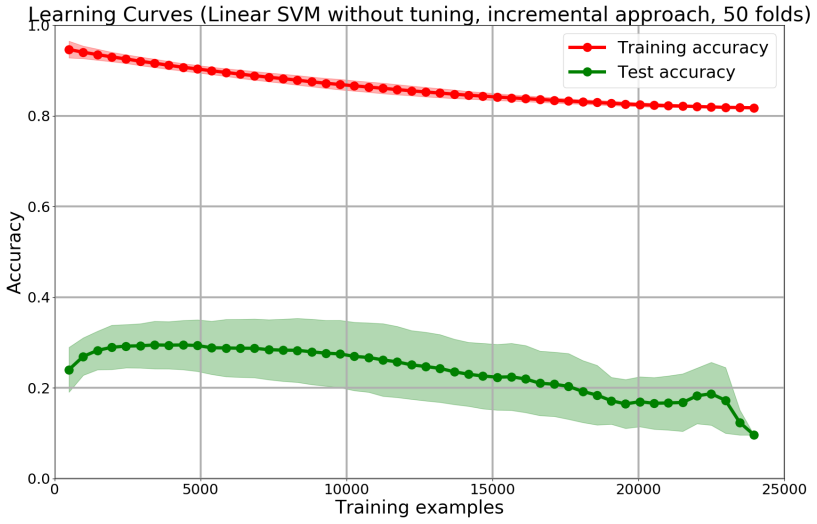
Figure 4.1: Learning curves of the first sub experiment of the preliminary experiment conducted on the bug reports of the telecommunications company (obtained with 4, 8 and 50 folds)



(a) 4 folds

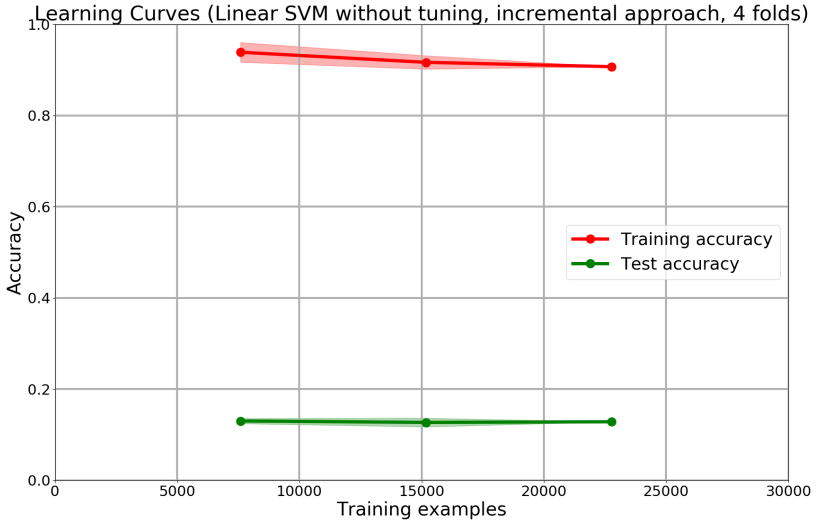


(b) 8 folds

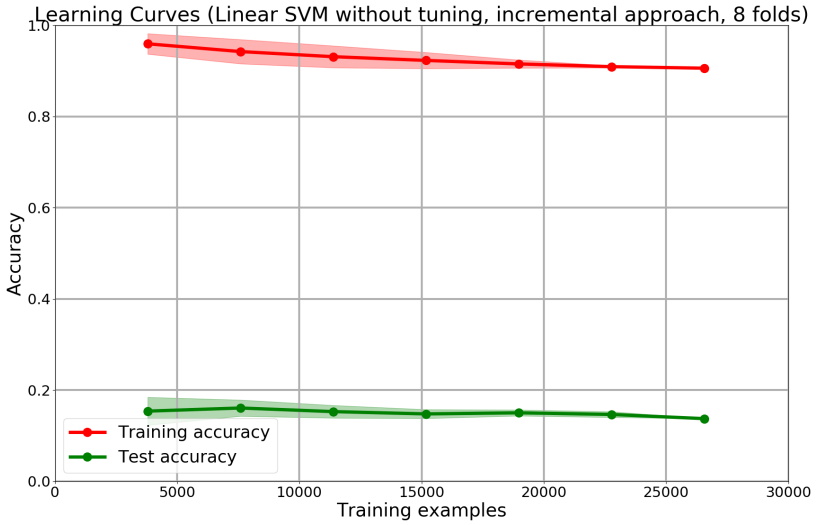


(c) 50 folds

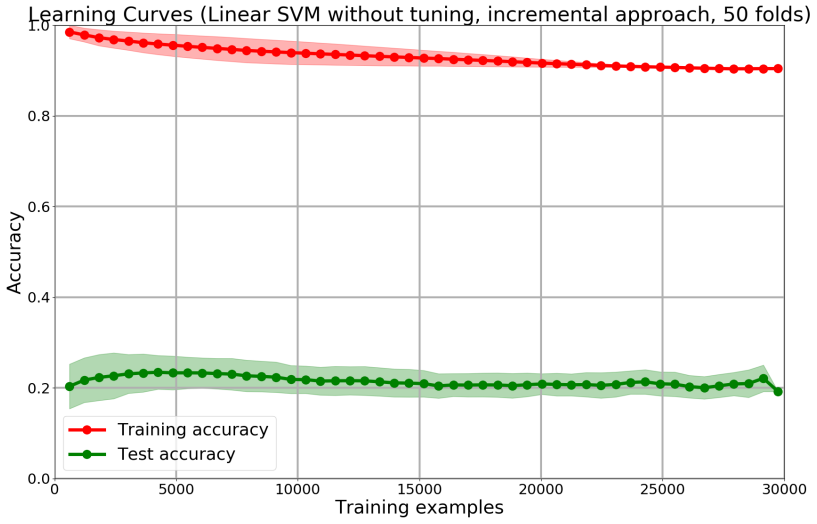
Figure 4.2: Learning curves of the first sub experiment of the preliminary experiment conducted on the bug reports of Eclipse JDT (obtained with 4, 8 and 50 folds)



(a) 4 folds



(b) 8 folds



(c) 50 folds

Figure 4.3: Learning curves of the first sub experiment of the preliminary experiment conducted on the bug reports of Mozilla Firefox (obtained with 4, 8 and 50 folds)

4.1.2 Second sub experiment

In this section, the results of the second sub experiment of the preliminary experiment of the thesis are described.

The graphs in the Figures 4.4, 4.5 and 4.6 depict the learning curves obtained with different numbers of folds $K, K \in \{4; 8; 50\}$. For each $K \in \{4; 8; 50\}$, two learning curves are plotted: the red one is related to the accuracy on the training set whereas the green one shows the accuracy on the test set. The standard deviation related to each point plotted in each chart is also represented. The graphs related to the configurations with 6, 10, 15 and 25 folds can be found in the Figures A.7, A.8, A.9, A.10, A.11 and A.12 of the Appendix A.2. As explained in the Section 3.3.2, contrary to the first sub experiment, only the last folds are used to plot the green learning curves.

Telecommunications company

In all the subfigures of the Figure 4.4, with an increase in the size of the training set, the accuracy on the same set has raised.

With 4 folds, the accuracy on the test set has steadily decreased with an increase in the number of bug reports in the training set. With no more than 8 folds (and more than 4 folds), the accuracy on the test set has increased, then, decreased. Its maximum was reached when the number of bug reports in the training set was between 30 000 and 40 000. With strictly more than 8 folds, the accuracy on the test set has fluctuated, but generally, it has increased with an increase in the size of the training set.

Eclipse JDT

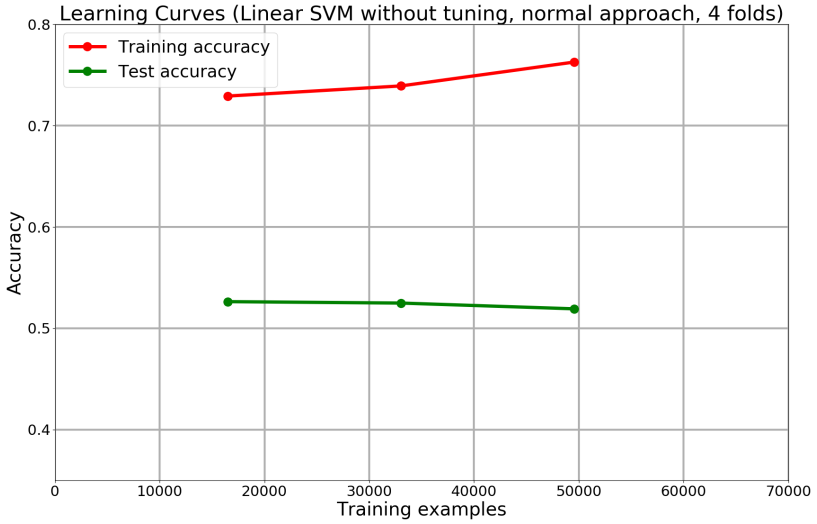
With an increase in the size of the training set, the accuracy on the same set has sometimes fluctuated, but generally, it has decreased in all the subfigures of the Figure 4.5

With no more than 8 folds, the accuracy on the test set has steadily decreased with an increase in the number of bug reports in the training set. With strictly more than 8 folds, the accuracy on the test set has fluctuated, but generally, it has decreased with an increase in the size of the training set.

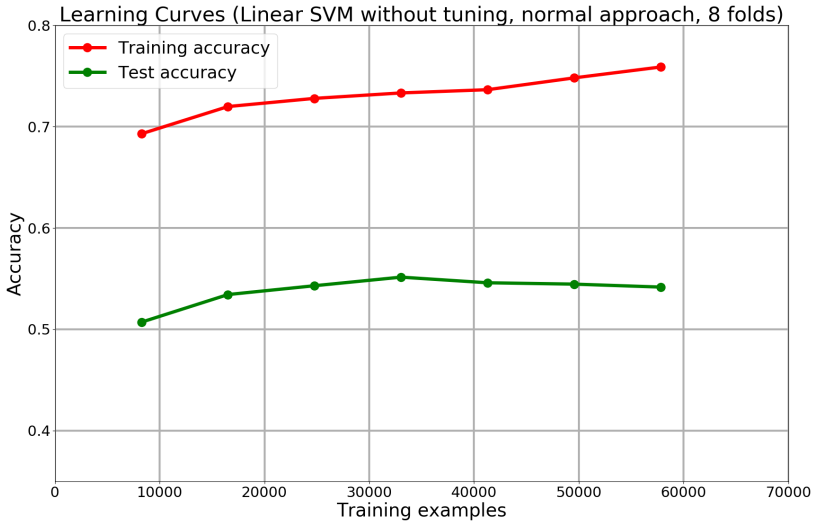
Mozilla Firefox

In all the subfigures of the Figure 4.6, with an increase in the size of the training set, the accuracy on the same set has sometimes fluctuated, but generally, decreased.

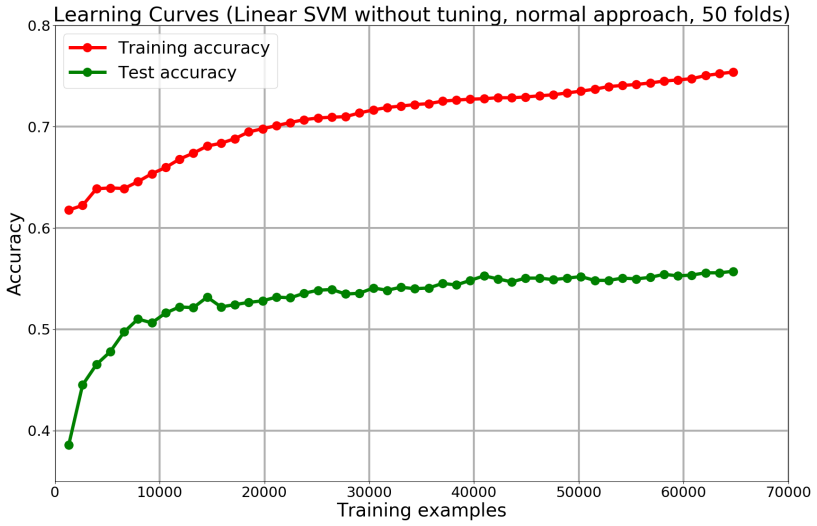
As with the first sub experiment conducted on the same data set, with less than 8 folds, the accuracy on the test set has fluctuated, but, been relatively constant. With at least 8 folds, the accuracy on the test set has fluctuated, but generally, it has increased, then, decreased: it has reached its maximum when the number of bug reports in the training set was between 3 000 and 10 000.



(a) 4 folds

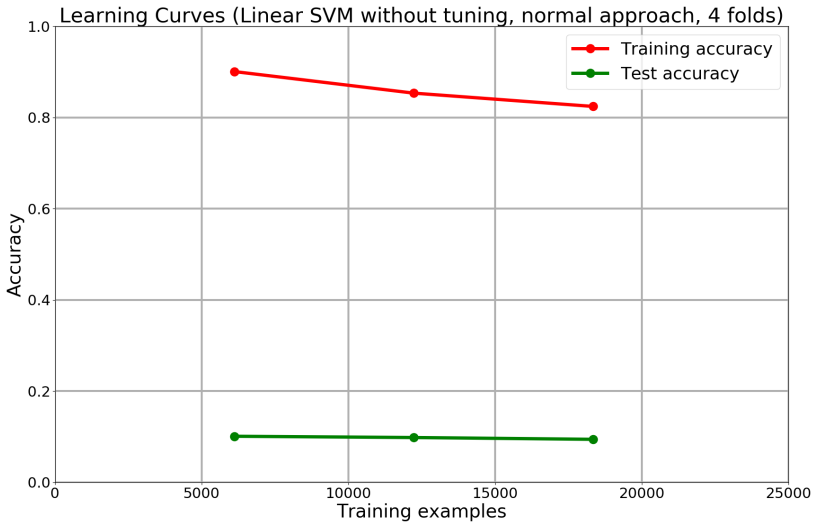


(b) 8 folds

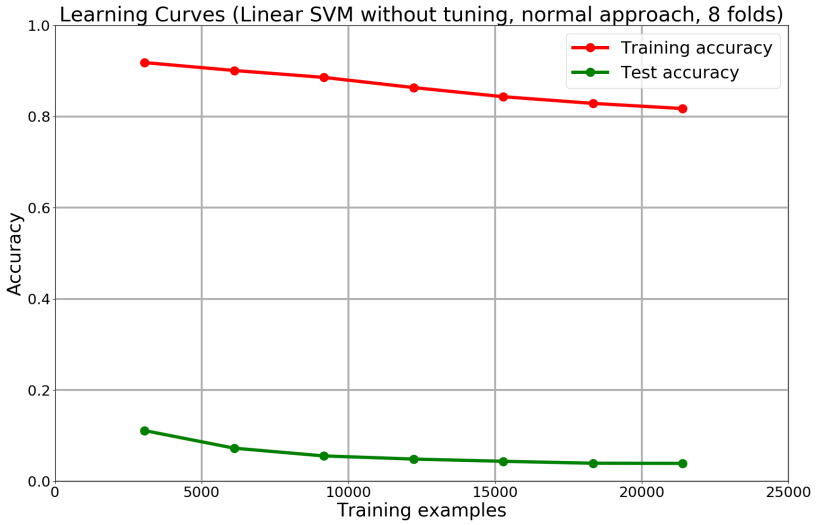


(c) 50 folds

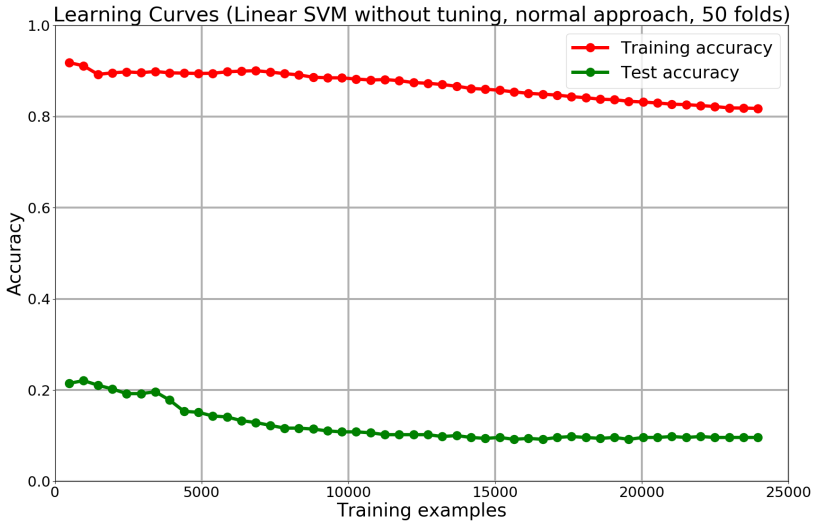
Figure 4.4: Learning curves of the second sub experiment of the preliminary experiment conducted on the bug reports of the telecommunications company (obtained with 4, 8 and 50 folds)



(a) 4 folds

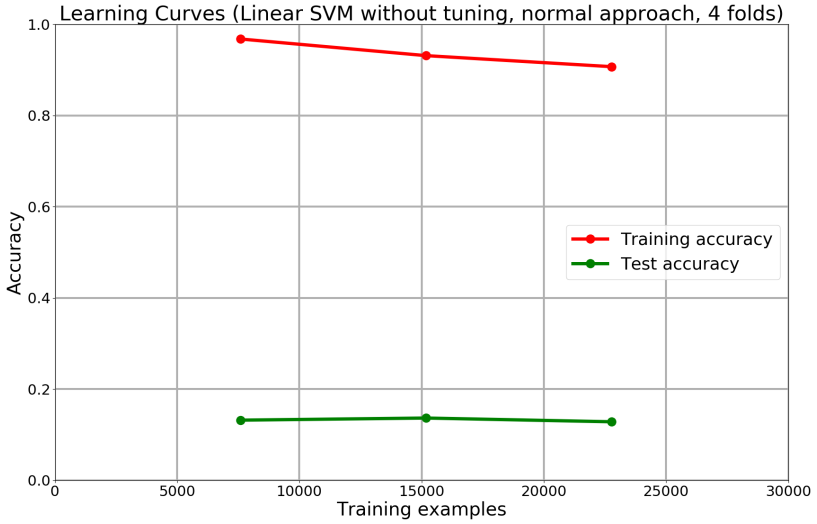


(b) 8 folds

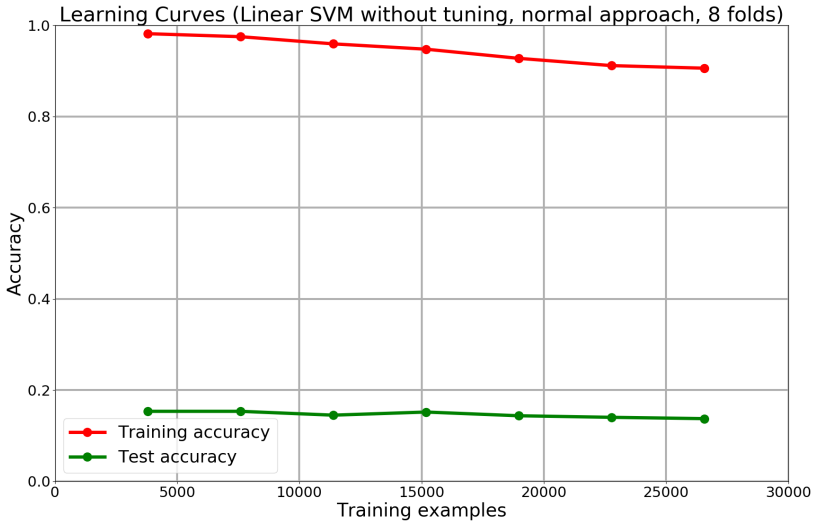


(c) 50 folds

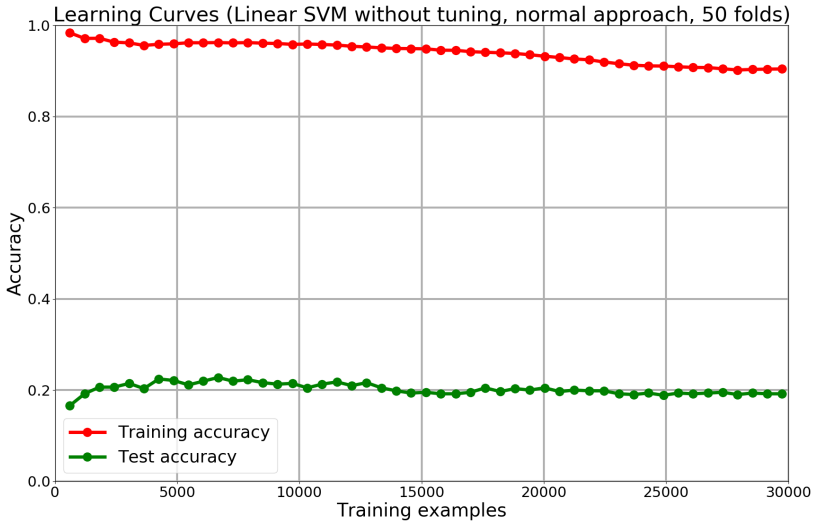
Figure 4.5: Learning curves of the second sub experiment of the preliminary experiment conducted on the bug reports of Eclipse JDT (obtained with 4, 8 and 50 folds)



(a) 4 folds



(b) 8 folds



(c) 50 folds

Figure 4.6: Learning curves of the second sub experiment of the preliminary experiment conducted on the bug reports of Mozilla Firefox (obtained with 4, 8 and 50 folds)

4.2 Main experiments

The summarized results related to the main experiments of the thesis (Section 3.4) are presented below.

The detailed results related to the same experiments can be found in the Appendix B.

4.2.1 Experiment 1

In this section, the summarized results related to the first main experiment of the thesis (Section 3.4.1) are described.

The detailed results related to the same main experiment can be found in the Appendix B.1.

For readability purposes, the results will be described using a mapping of acronyms to pre-processing techniques (Table 4.1). When a pre-processing technique is used, its acronym is directly written. If not, it is preceded by the string `NOT` and enclosed within parenthesis. All obtained strings are then concatenated and the character `|` is used as a delimiter between them.

If the string `C|NOT(S)|L|SW|P|N|NOT(LC)` is used for instance, it means that the bug reports have first been cleaned. The tokens have then been lemmatized. The stop words, the tokens containing only punctuation characters and the tokens containing only numbers have eventually been removed.

Acronym	Pre-processing technique
C	Cleaning bug reports
S	Stemmer
L	Lemmatizer
SW	Stop words removal
P	Punctuation characters removal
N	Numbers removal
LC	Conversion to lower case

Table 4.1: The mapping of acronyms to pre-processing techniques

Telecommunications company

The horizontal bar chart in the Figure 4.7 depicts the accuracy of the worst and best pre-processing configurations on the project of the telecommunications company.

The horizontal bar chart in the Figure 4.8 shows the MRR value of the worst and best pre-processing configurations on the bug reports of the telecommunications company.

As the best accuracy, 74.74 % (Figure 4.7), and, the best MRR value, 83.43 % (Figure 4.8), were reached when cleaning the bug reports, removing the stop words, removing the tokens containing only punctuation characters, removing the tokens containing only numbers and performing a conversion to lower case, only the aforementioned pre-processing techniques have been used in the following experiments conducted on the bug reports of the telecommunications company.

Eclipse JDT

The horizontal bar chart in the Figure 4.9 shows the accuracy of the worst and best pre-processing configurations on the bug reports of the Eclipse JDT project.

The horizontal bar chart in the Figure 4.10 depicts the MRR value of the worst and best pre-processing configurations on Eclipse JDT.

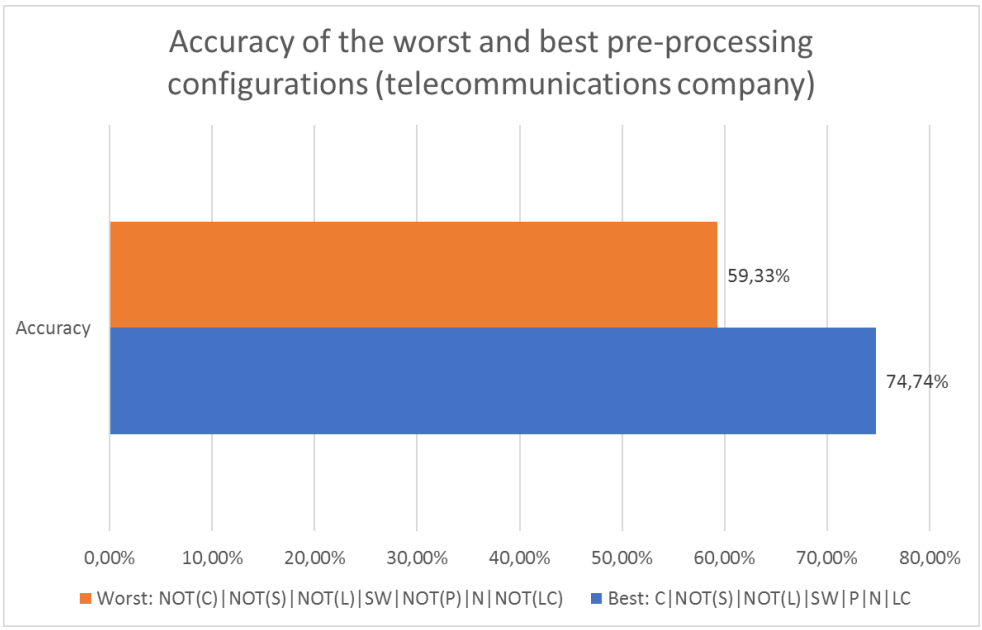


Figure 4.7: Accuracy of the worst and best pre-processing configurations on the bug reports of the telecommunications company (the mapping of acronyms to pre-processing techniques defined in the Table 4.1 has been used)

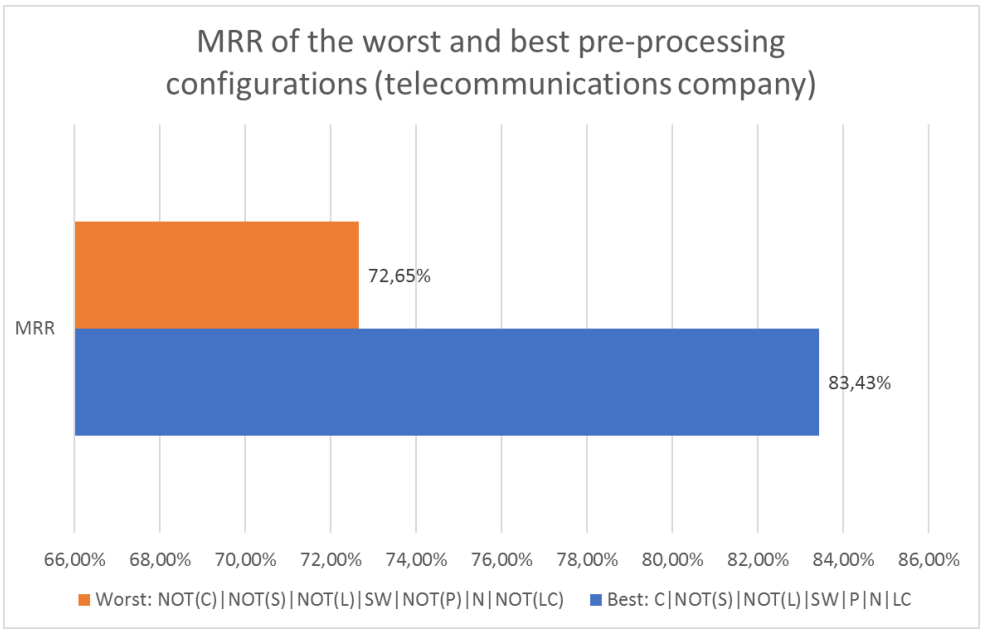


Figure 4.8: MRR of the worst and best pre-processing configurations on the bug reports of the telecommunications company (the mapping of acronyms to pre-processing techniques defined in the Table 4.1 has been used)

Contrary to the bug reports of the telecommunications company, the best configuration, in terms of accuracy, was obtained, only when cleaning the bug reports, removing the stop words and using a conversion to lower case (Figure 4.9). This configuration has therefore been used when conducting the other experiments on the Eclipse JDT data set.

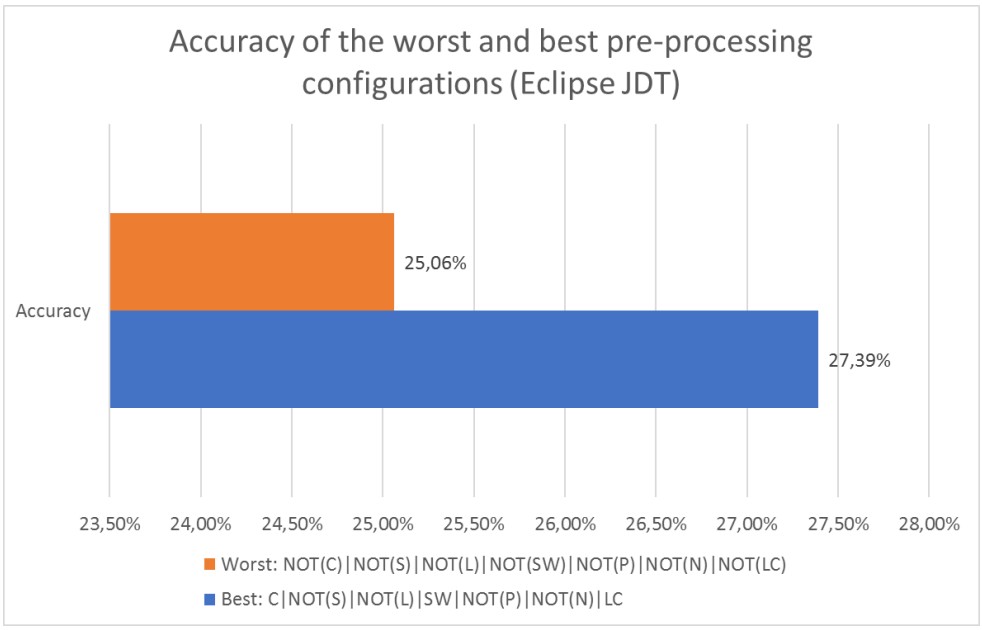


Figure 4.9: Accuracy of the worst and best pre-processing configurations on the bug reports of Eclipse JDT (the mapping of acronyms to pre-processing techniques defined in the Table 4.1 has been used)

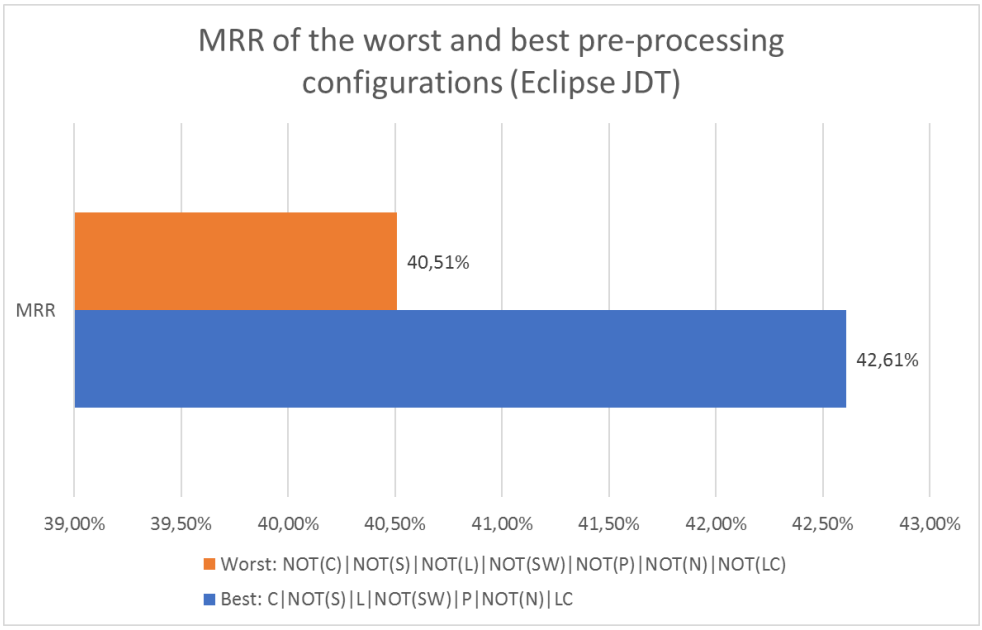


Figure 4.10: MRR of the worst and best pre-processing configurations on the bug reports of Eclipse JDT (the mapping of acronyms to pre-processing techniques defined in the Table 4.1 has been used)

Mozilla Firefox

The horizontal bar chart in the Figure 4.11 illustrates the accuracy of the worst and best pre-processing configurations on the bug reports of the Mozilla Firefox project.

The horizontal bar chart in the Figure 4.12 depicts the MRR value of the worst and best pre-processing configurations on Mozilla Firefox.

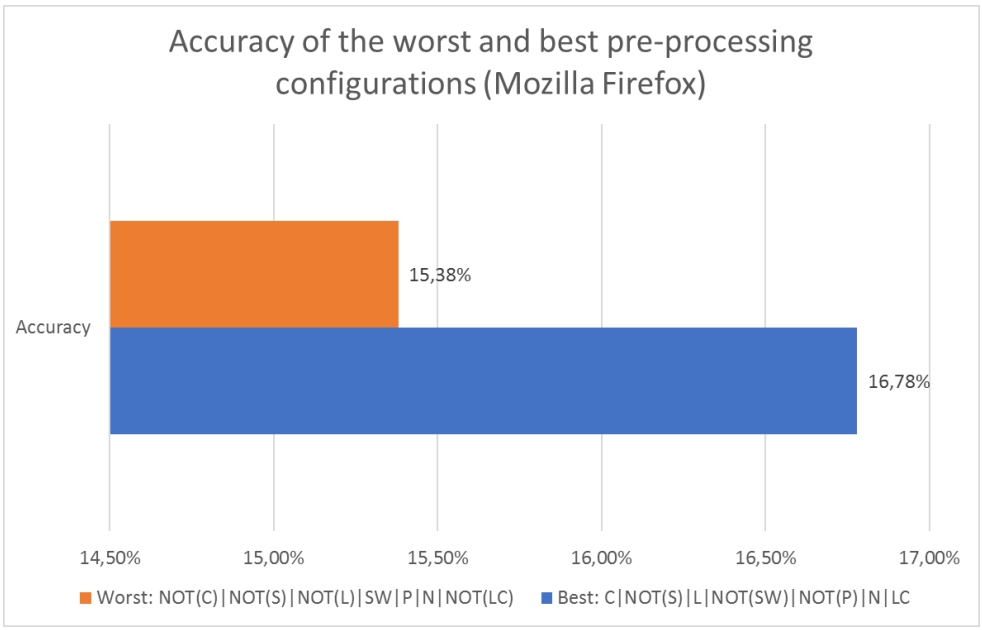


Figure 4.11: Accuracy of the worst and best pre-processing configurations on the bug reports of Mozilla Firefox (the mapping of acronyms to pre-processing techniques defined in the Table 4.1 has been used)

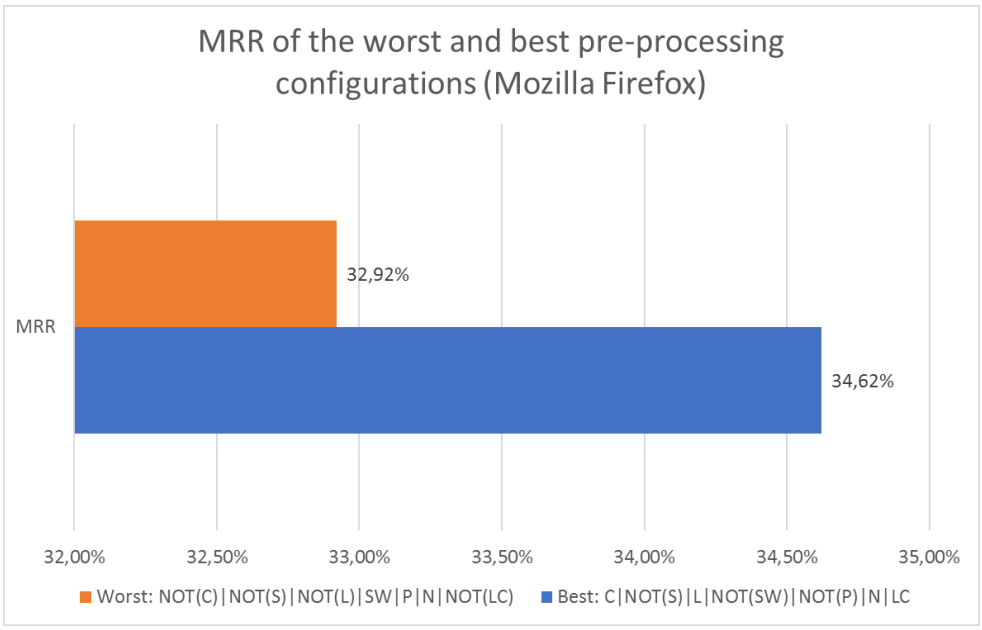


Figure 4.12: MRR of the worst and best pre-processing configurations on the bug reports of Mozilla Firefox (the mapping of acronyms to pre-processing techniques defined in the Table 4.1 has been used)

Contrary to the bug reports of the telecommunications company and the bug reports of Eclipse JDT, the best configuration, in terms of accuracy and MRR value, has been obtained, only when cleaning the bug reports, using a lemmatizer, removing the tokens containing only numbers and using a conversion to lower case (Figures 4.11 and 4.12). This configuration has been used when conducting the following experiments on the Mozilla Firefox project.

4.2.2 Experiment 2

The summarized results related to the second main experiment of the thesis (Section 3.4.2) are presented in this section.

The detailed results related to the same main experiment can be found in the Appendix B.2.

For readability purposes, the results will be described using a mapping of acronyms to feature extraction techniques (Table 4.2). If a feature extraction technique is used, its acronym is directly written. Each time the LSI or the NMF algorithm is used, the acronym is followed by an hyphen - and the new number of features.

If the string `TF+LSI-50` is used for instance, it means that the tf weights have first been used. The LSI algorithm has then been run to extract 50 new features.

Acronym	Feature extraction technique
BOOL	Boolean
TF	TF
TF-IDF	TF-IDF
BOOL+LSI	Boolean + LSI
TF+LSI	TF + LSI
TF-IDF+LSI	TF-IDF + LSI
BOOL+NMF	Boolean + NMF
TF+NMF	TF + NMF
TF-IDF+NMF	TF-IDF + NMF

Table 4.2: The mapping of acronyms to feature extraction techniques

When all the possible pairs of extracted features have been generated, the acronyms in the Table 4.3 are concatenated and the character `&` is used as a delimiter between them.

If the string `TF-IDF+LSI&BOOL+NMF` is used, it means that a combination of two feature extraction techniques has been used. First, the LSI algorithm has been applied on the tf-idf weights of the bug reports. Second, a Boolean representation of the bug reports followed by the application of the NMF algorithm has been used. As explained in the Section 3.4.2, when LSI or NMF is used in a combination of two feature extraction techniques, only the number of features which has given the best accuracy is considered.

Telecommunications company

The horizontal bar chart in the Figure 4.13 shows the accuracy of the worst (without combination), best (without combination), worst (with combination) and best (with combination) feature extraction techniques on the project of the telecommunications company.

As can be seen in the Figure 4.13, the best configuration (without combination), in terms of accuracy, is using tf-idf weights.

The horizontal bar chart in the Figure 4.14 indicates the MRR value of the worst (without combination), best (without combination), worst (with combination) and best (with combination) feature extraction techniques on the bug reports of the telecommunications company.

The best configuration (without combination), in terms of MRR, is only based on tf-idf weights (Figure 4.14).

The accuracy of the best configuration, when combining feature extraction techniques, is 75.90 % whereas the best one, when not combining these techniques, is 75.64 % (Figure 4.13). When combining feature extraction techniques, the MRR value of the best configuration is 84.25 % whereas the best one, when not combining these techniques, is 84.06 % (Figure 4.14). As the differences between both accuracies and both MRR values are relatively low, in the following experiments conducted on the project of the telecommunications company, the features have been extracted via only the use of tf-idf weights.

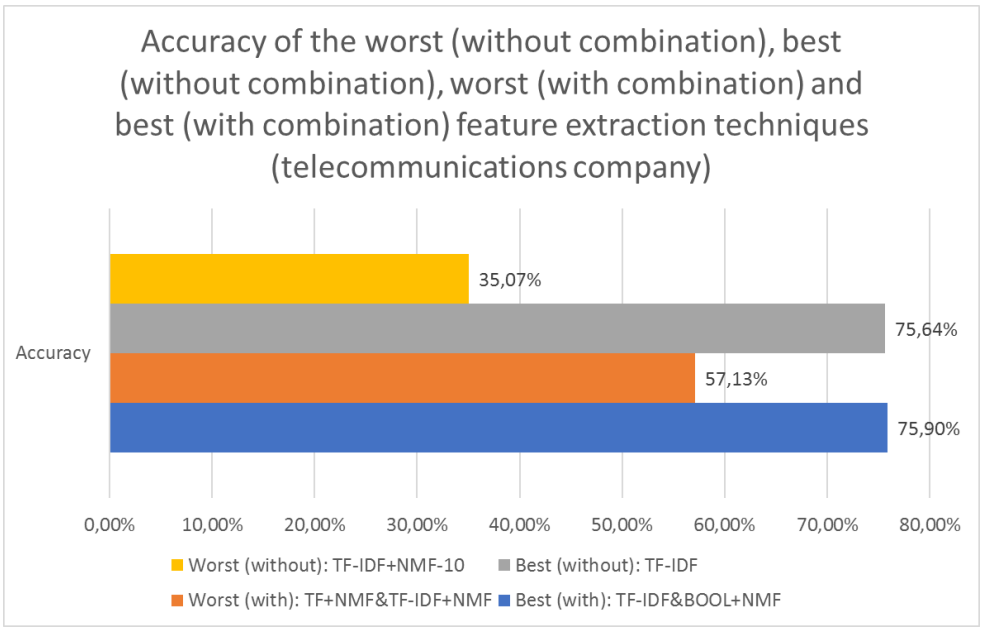


Figure 4.13: Accuracy of the worst (without combination), best (without combination), worst (with combination) and best (with combination) feature extraction techniques on the bug reports of the telecommunications company (the mapping of acronyms to feature extraction techniques defined in the Table 4.2 has been used)

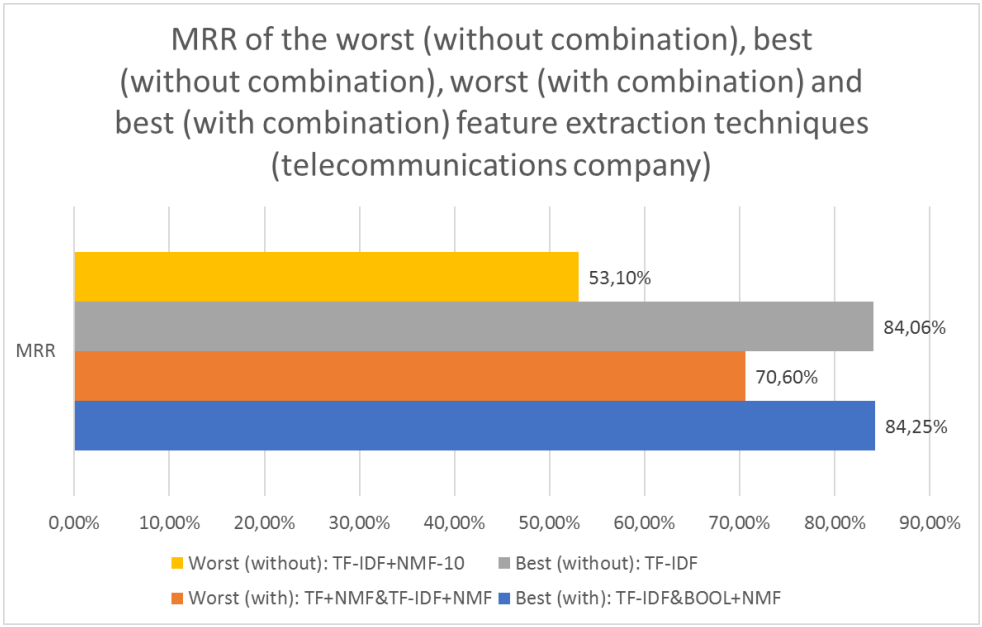


Figure 4.14: MRR of the worst (without combination), best (without combination), worst (with combination) and best (with combination) feature extraction techniques on the bug reports of the telecommunications company (the mapping of acronyms to feature extraction techniques defined in the Table 4.2 has been used)

Eclipse JDT

The horizontal bar chart in the Figure 4.15 illustrates the accuracy of the worst (without combination), best (without combination), worst (with combination) and best (with combination) feature extraction techniques on the bug reports of Eclipse JDT.

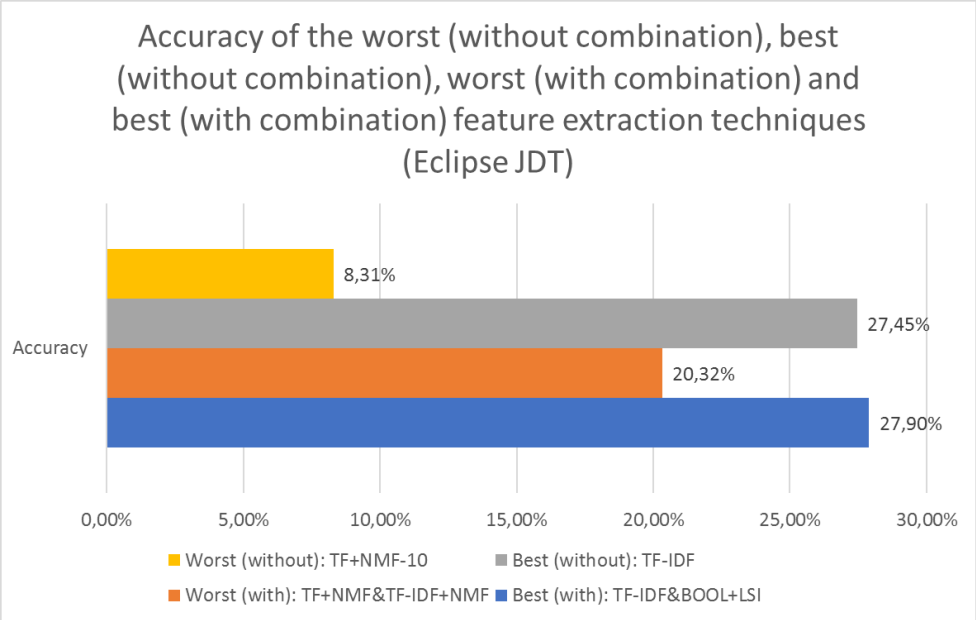


Figure 4.15: Accuracy of the worst (without combination), best (without combination), worst (with combination) and best (with combination) feature extraction techniques on the bug reports of Eclipse JDT (the mapping of acronyms to feature extraction techniques defined in the Table 4.2 has been used)

The best configuration (without combination), in terms of accuracy, is using tf-idf weights (Figure 4.15).

The horizontal bar chart in the Figure 4.16 indicates the MRR value of the worst (without combination), best (without combination), worst (with combination) and best (with combination) feature extraction techniques on Eclipse JDT.

When combining feature extraction techniques, the accuracy of the best configuration is 27.90 % whereas the best one, when not combining these techniques, is 27.45 % (Figure 4.15). The MRR value of the best configuration is 43.02 % when combining feature extraction techniques, whereas the best one, when not combining these techniques, is 42.56 % (Figure 4.16). Moreover, the MRR value of the best configuration (in terms of accuracy) is 42.39 % when not combining feature extraction techniques (Figure B.12), whereas the best one, when combining these techniques, is 42.93 % (Figure B.14). As the differences between both accuracies and the different MRR values are relatively low, in the following experiments conducted on the bug reports of Eclipse JDT, the features have been extracted via only the use of tf-idf weights.

Mozilla Firefox

The horizontal bar chart in the Figure 4.17 shows the accuracy of the worst (without combination), best (without combination), worst (with combination) and best (with combination) feature extraction techniques on Mozilla Firefox.

The horizontal bar chart in the Figure 4.18 depicts the MRR value of the worst (without combination), best (without combination), worst (with combination) and best (with combination) feature extraction techniques on the bug reports of Mozilla Firefox.

Contrary to the bug reports of the telecommunications company and Eclipse JDT, the best configuration (without combination), in terms of accuracy and MRR, is only based on the tf weights (Figures 4.17 and 4.18).

The accuracy of the best configuration when combining feature extraction techniques is 16.79 %, whereas the best one, when not combining these techniques, is 16.78 % (Figure 4.17).

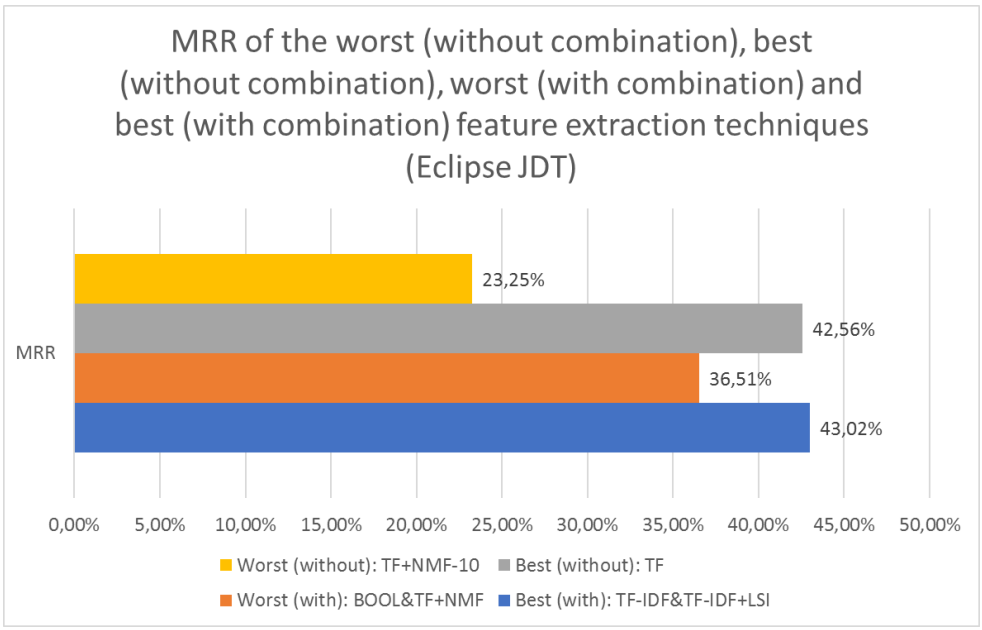


Figure 4.16: MRR of the worst (without combination), best (without combination), worst (with combination) and best (with combination) feature extraction techniques on the bug reports of Eclipse JDT (the mapping of acronyms to feature extraction techniques defined in the Table 4.2 has been used)

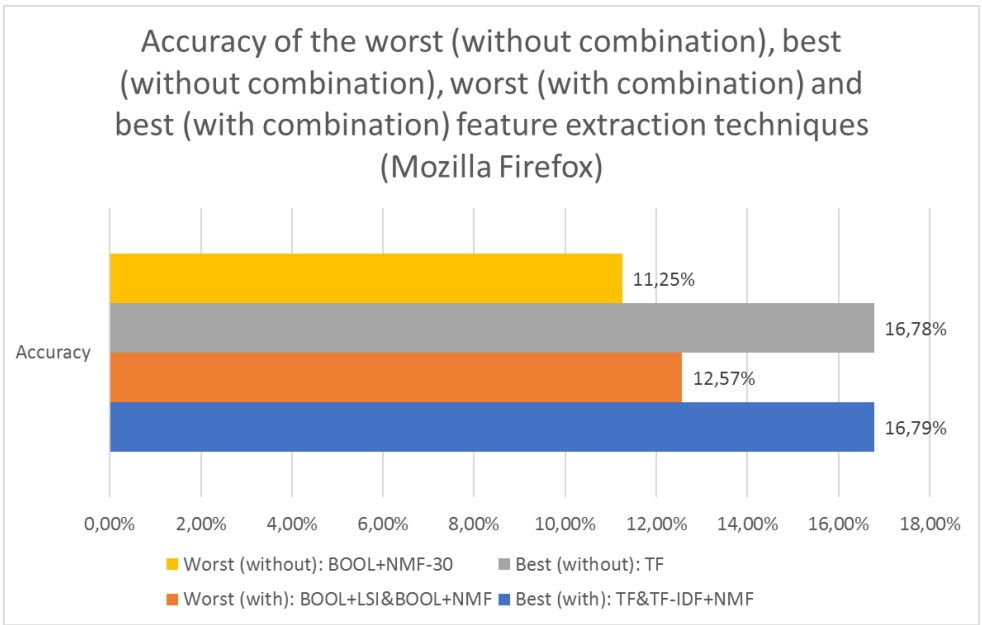


Figure 4.17: Accuracy of the worst (without combination), best (without combination), worst (with combination) and best (with combination) feature extraction techniques on the bug reports on the bug reports of Mozilla Firefox (the mapping of acronyms to feature extraction techniques defined in the Table 4.2 has been used)

When not combining feature extraction techniques, the MRR value of the best representation is 34.62 %, whereas the best one, when combining them, is 34.63 % (Figure 4.18). As the best accuracy and the best MRR value reached when combining feature extraction techniques are almost the same as the ones reached when not combining these techniques, the features have

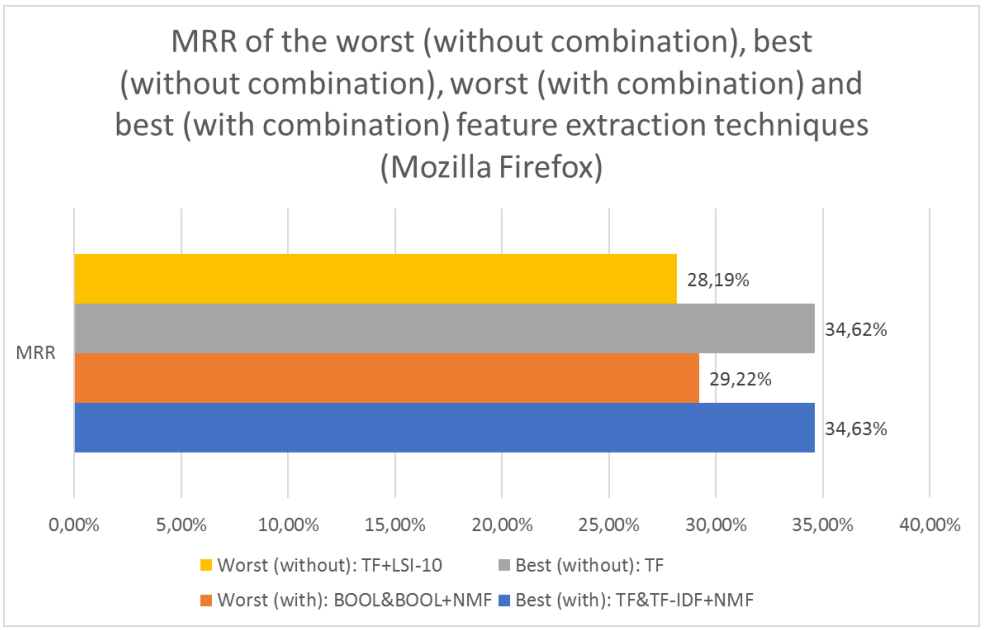


Figure 4.18: MRR of the worst (without combination), best (without combination), worst (with combination) and best (with combination) feature extraction techniques on the bug reports on the bug reports of Mozilla Firefox (the mapping of acronyms to feature extraction techniques defined in the Table 4.2 has been used)

been extracted via only the use of the tf weights in the following experiments conducted on the bug reports of Mozilla Firefox.

4.2.3 Experiment 3

In this section, the summarized results of the third main experiment of the thesis (Section 3.4.3) are presented.

The detailed results related to the same main experiment can be found in the Appendix B.3.

For readability purposes, the results are presented with a mapping of acronyms to feature selection techniques (Table 4.3). If a feature selection technique is applied, its acronym is used. This string is then concatenated with an hyphen - and the percentage of remaining features.

If the string `CHI-2-30` is used for instance, it means that the χ^2 test has been applied. Based on the results of this test, 30 % of the features have then been selected.

Acronym	Feature selection technique
CHI-2	Chi-2
ANOVA	ANOVA
MI	Mutual information
RFE	Recursive Feature Elimination

Table 4.3: The mapping of acronyms to feature selection techniques

Telecommunications company

The horizontal bar chart in the Figure 4.19 illustrates the accuracy of the worst three and best three feature selection techniques on the project of the telecommunications company.

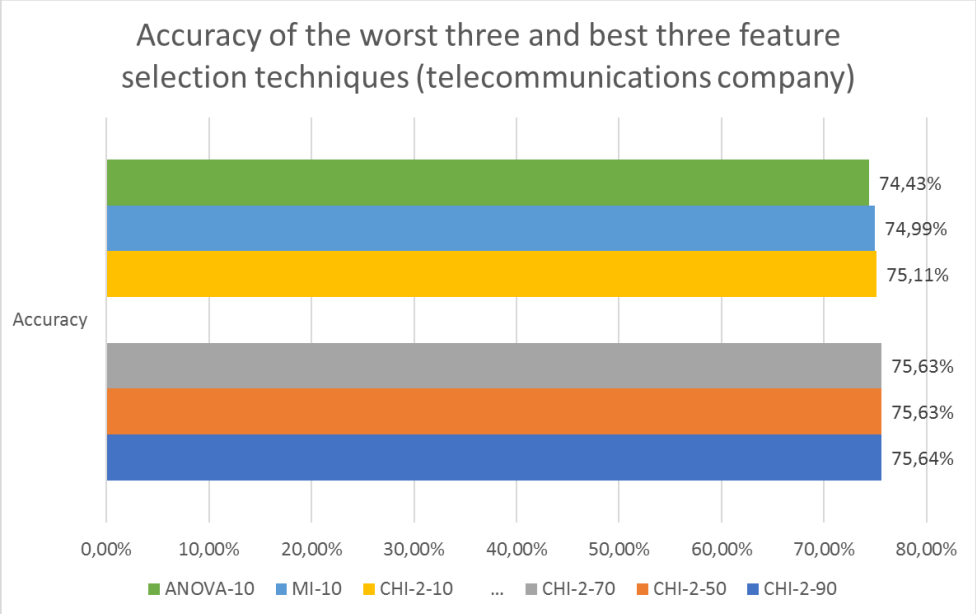


Figure 4.19: Accuracy of the worst three and best three feature selection techniques on the bug reports of the telecommunications company (the mapping of acronyms to feature selection techniques defined in the Table 4.3 has been used)

The horizontal bar chart in the Figure 4.20 depicts the MRR value of the worst three and best three feature selection techniques on the bug reports of the telecommunications company.

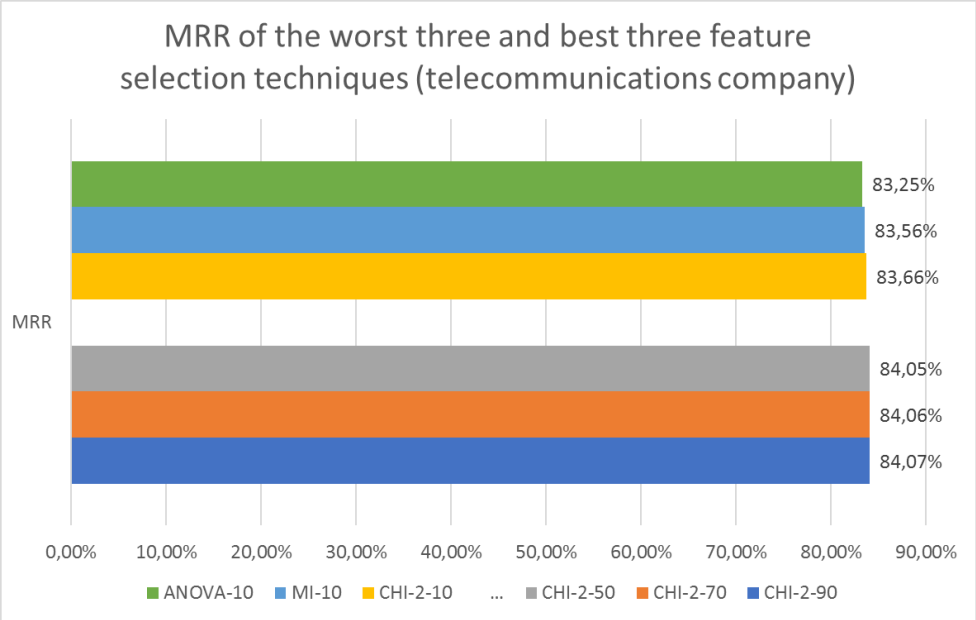


Figure 4.20: MRR of the worst three and best three feature selection techniques on the bug reports of the telecommunications company (the mapping of acronyms to feature selection techniques defined in the Table 4.3 has been used)

As can be seen in the Figures 4.19 and 4.20, the best three configurations are the ones using the χ^2 test as a feature selection technique. The worst three configurations are obtained when selecting 10 % of the initial features.

The accuracy of the best configuration is 75.64 % (Figure 4.19). Its MRR value is 84.07 % (Figure 4.20). As this accuracy and this MRR value have almost been reached in the experiment 2, no feature selection technique has been used in the last experiment conducted on the bug reports of the project of the telecommunications company.

Eclipse JDT

The horizontal bar chart in the Figure 4.21 shows the accuracy of the worst three and best three feature selection techniques on Eclipse JDT.

The horizontal bar chart in the Figure 4.22 indicates the MRR value of the worst three and best three feature selection techniques on the bug reports of Eclipse JDT.

In the Figures 4.21 and 4.22, the best configuration is using the χ^2 test to select 90 % of the features. The worst configuration is obtained when selecting 10 % of the initial features via the use of the mutual information technique.

The accuracy of the best configuration is 27.43 % (Figure 4.21). Its MRR value is 42.38 % (Figure 4.22). As this accuracy and this MRR value have already been reached in the experiment 2, no feature selection technique has been used in the last experiment conducted on the bug reports of Eclipse JDT.

Mozilla Firefox

The horizontal bar chart in the Figure 4.23 depicts the accuracy of the worst three and best three feature selection techniques on the bug reports of Mozilla Firefox.

The horizontal bar chart in the Figure 4.24 indicates the MRR value of the worst three and best three feature selection techniques on Mozilla Firefox.

Contrary to the bug reports of the telecommunications company and the bug reports of Eclipse JDT, in the figures 4.23 and 4.24, the best three configurations are using recursive

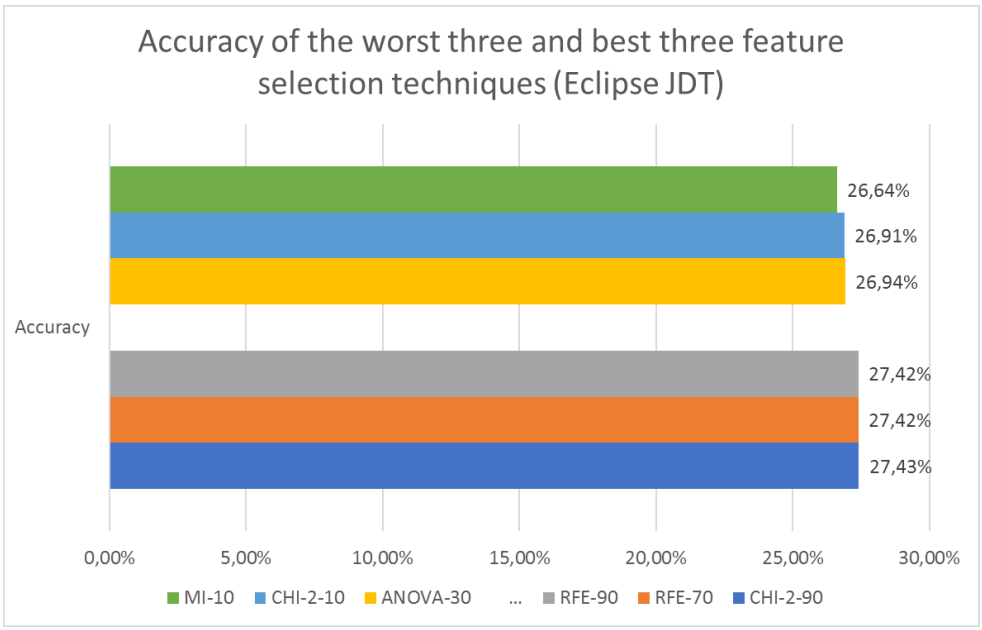


Figure 4.21: Accuracy of the worst three and best three feature selection techniques on the bug reports of Eclipse JDT (the mapping of acronyms to feature selection techniques defined in the Table 4.3 has been used)

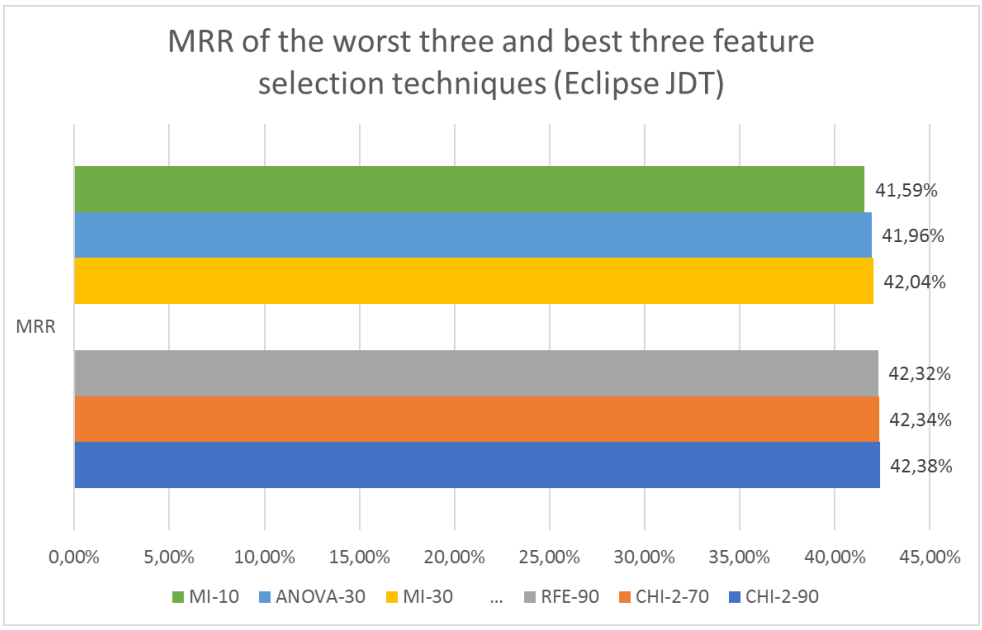


Figure 4.22: MRR of the worst three and best three feature selection techniques on the bug reports of the bug reports of Eclipse JDT (the mapping of acronyms to feature selection techniques defined in the Table 4.3 has been used)

feature elimination as a feature selection technique. The worst two configurations, in terms of accuracy and MRR value, are selecting 10 % of the initial features.

As can be seen in the Figures 4.23 and 4.24, the accuracy of the best configurations is 16.84 % and their MRR value is 34.65 %. As this accuracy and this MRR value have almost been reached in the experiment 2, no feature selection technique has been used in the last experiment conducted on the bug reports of Mozilla Firefox.

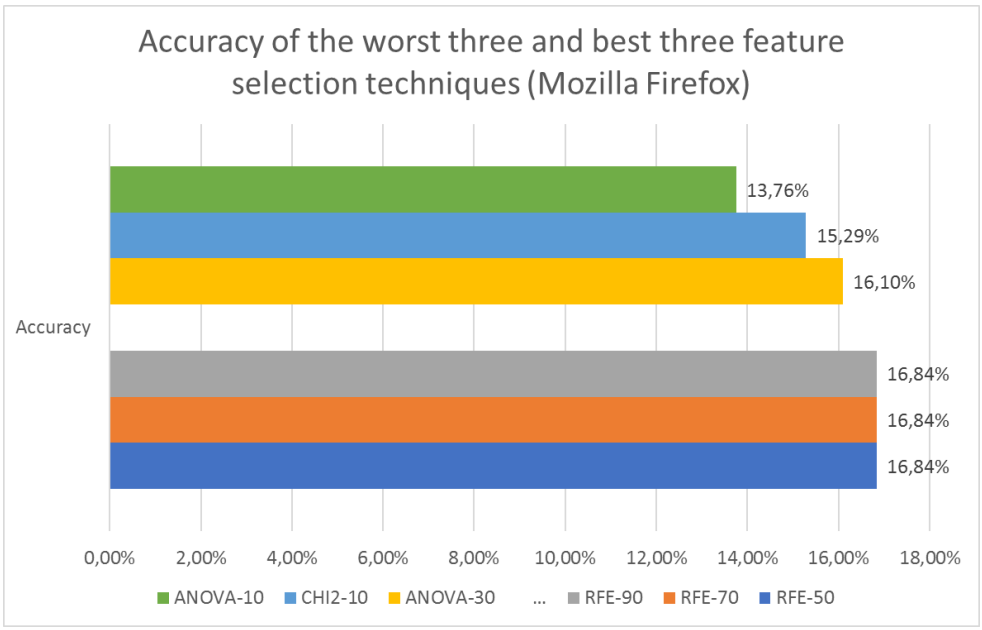


Figure 4.23: Accuracy of the worst three and best three feature selection techniques on the bug reports of Mozilla Firefox (the mapping of acronyms to feature selection techniques defined in the Table 4.3 has been used)

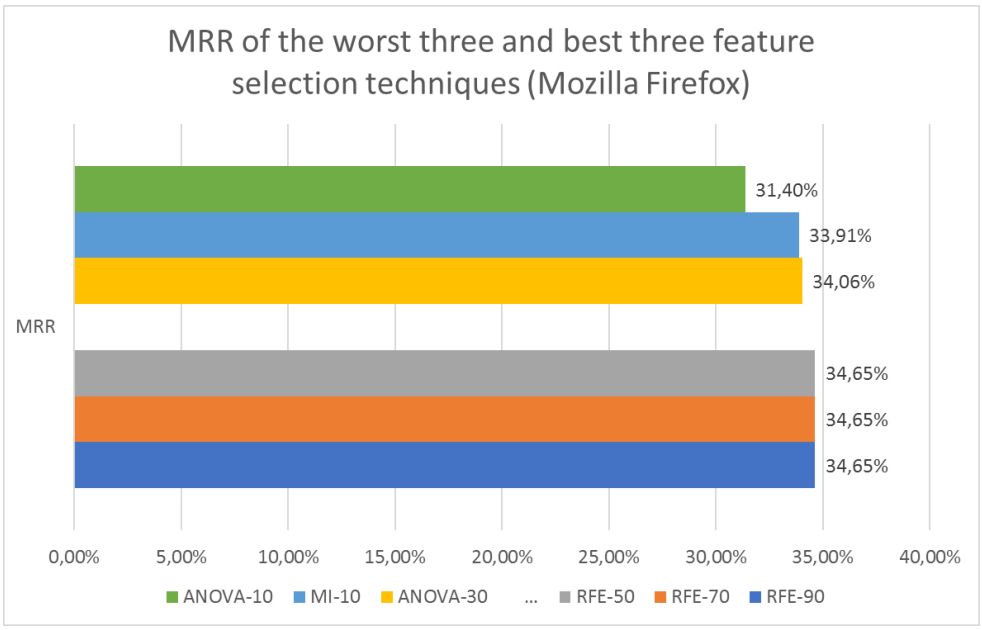


Figure 4.24: MRR of the worst three and best three feature selection techniques on the bug reports of Mozilla Firefox (the mapping of acronyms to feature selection techniques defined in the Table 4.3 has been used)

4.2.4 Experiment 4

The summarized results of the forth main experiment of the thesis (Section 3.4.4) are described in this section.

The detailed results related to the same main experiment can be found in the Appendix B.4.

As mentioned in the Section 3.5, the MRR values of the nearest centroid classifier have not been computed due to practical reasons.

Telecommunications company

The horizontal bar chart in the Figure 4.25 illustrates the best accuracy of the different classifiers (grid search and random search strategies) on the project of the telecommunications company.

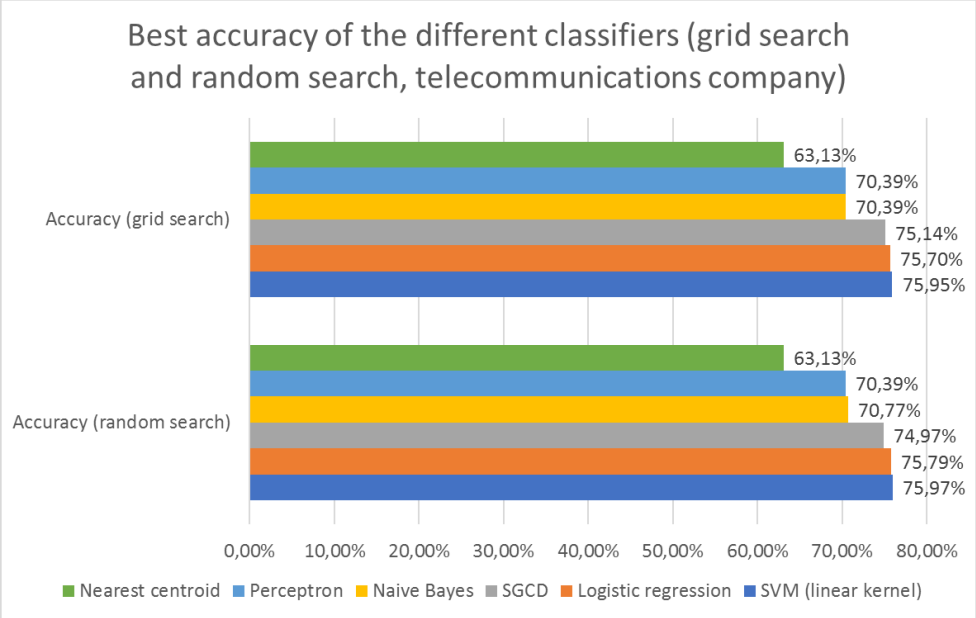


Figure 4.25: Best accuracy of the different classifiers (grid search and random search) on the bug reports of the telecommunications company

Except for the stochastic gradient descent algorithm, the accuracies reached with the best configuration of each classifier, using a random search strategy, are greater than or equal to the ones reached when using a grid search strategy (Figure 4.25). In terms of accuracy, when using a random search strategy, the classifiers are ordered the same way as the one obtained when using a grid search strategy.

The horizontal bar chart in the Figure 4.26 depicts the best MRR value of the different classifiers (grid search and random search strategies) on the project of the telecommunications company.

In terms of accuracy and MRR value, when using a grid search strategy, the best three classifiers are the linear SVM, the logistic regression and the stochastic gradient descent algorithm (Figures 4.25 and 4.26). In terms of accuracy, the worst configuration is obtained when using the nearest centroid classifier (Figure 4.25).

Eclipse JDT

The horizontal bar chart in the Figure 4.27 depicts the best accuracy of the different classifiers (grid search and random search strategies) on the bug reports of Eclipse JDT.

Except for the perceptron, the stochastic gradient descent algorithm, the linear SVM classifier and the logistic regression, the accuracies reached with the best configuration of each classifier, using a random search strategy, are greater than or equal to the ones reached when using a grid search strategy (Figure 4.27).

The horizontal bar chart in the Figure 4.28 indicates the best MRR value of the different classifiers (grid search and random search strategies) on Eclipse JDT.

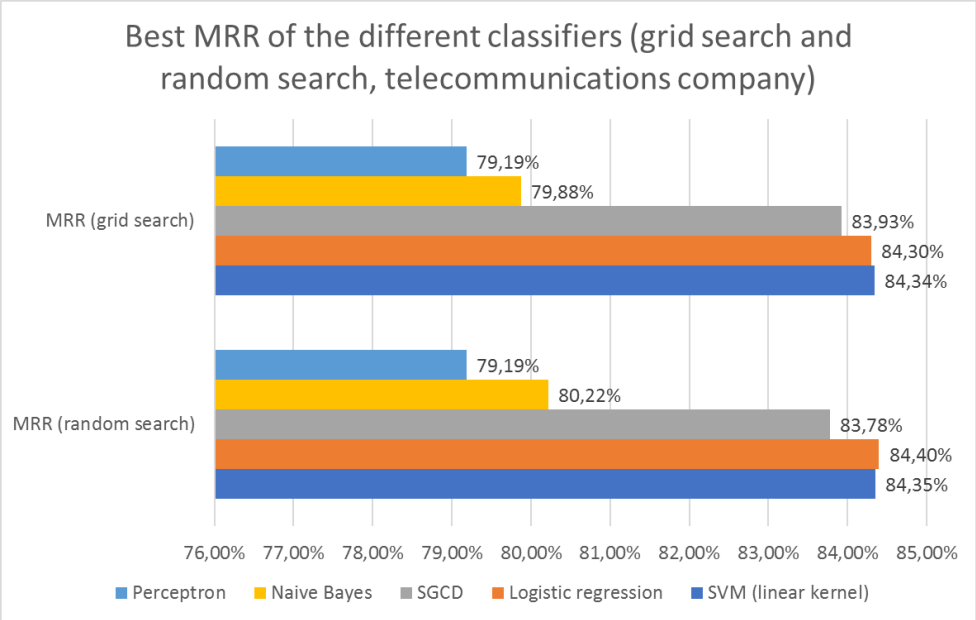


Figure 4.26: Best MRR of the different classifiers (grid search and random search) on the bug reports of the telecommunications company

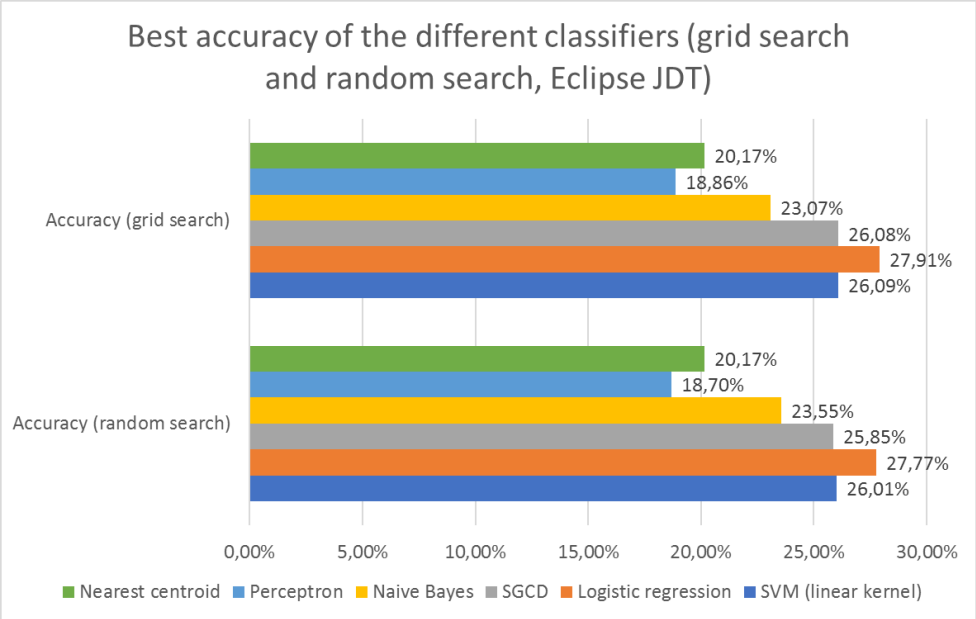


Figure 4.27: Best accuracy of the different classifiers (grid search and random search) on the bug reports of Eclipse JDT

When using a grid search strategy or a random search strategy, the best classifier, in terms of accuracy and MRR value, is the logistic regression (Figures 4.27 and 4.28). The worst classifier, in terms of accuracy, is the perceptron (Figure 4.27).

Mozilla Firefox

The horizontal bar chart in the Figure 4.29 shows the best accuracy of the different classifiers (grid search and random search strategies) on Mozilla Firefox.

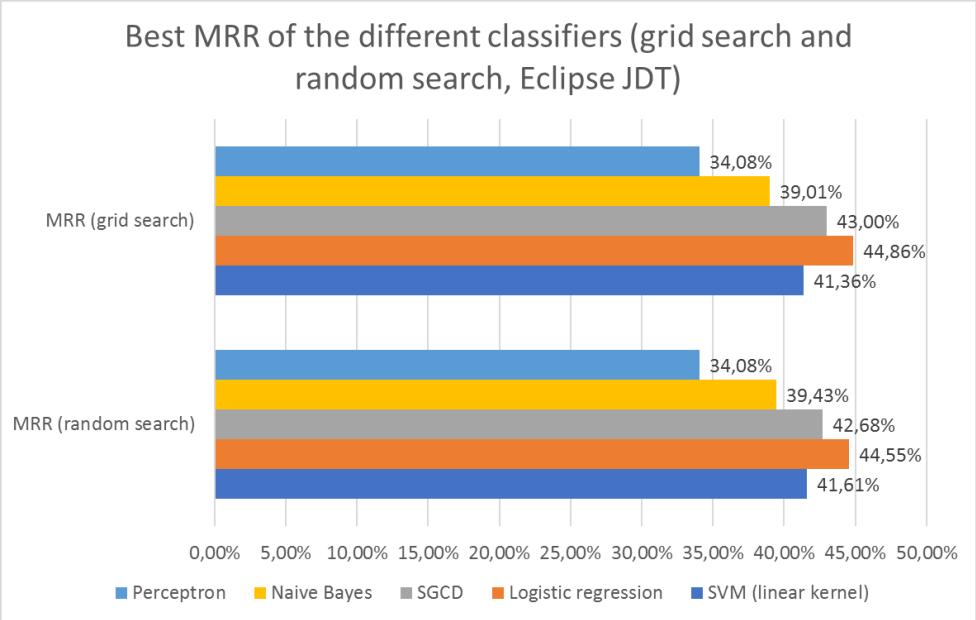


Figure 4.28: Best MRR of the different classifiers (grid search and random search) on the bug reports of Eclipse JDT

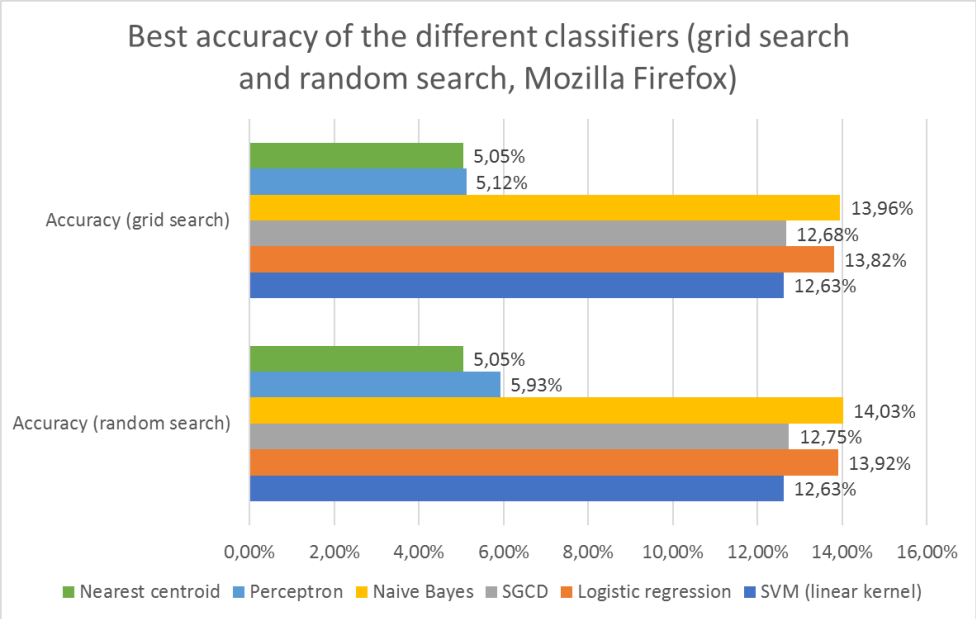


Figure 4.29: Best accuracy of the different classifiers (grid search and random search) on the bug reports of Mozilla Firefox

The accuracies reached with the best configuration of each classifier, using a random search strategy, are greater than or equal to the ones reached when using a grid search strategy (Figure 4.29). In terms of accuracy, when using a random search strategy, the classifiers are ordered the same way as the one obtained when using a grid search strategy.

The horizontal bar chart in the Figure 4.30 shows the best MRR value of the different classifiers (grid search and random search strategies) on Mozilla Firefox.

When using a grid search strategy, the best four classifiers, in terms of accuracy and MRR value, are the naive Bayes classifier (multinomial distribution), the logistic regression, the

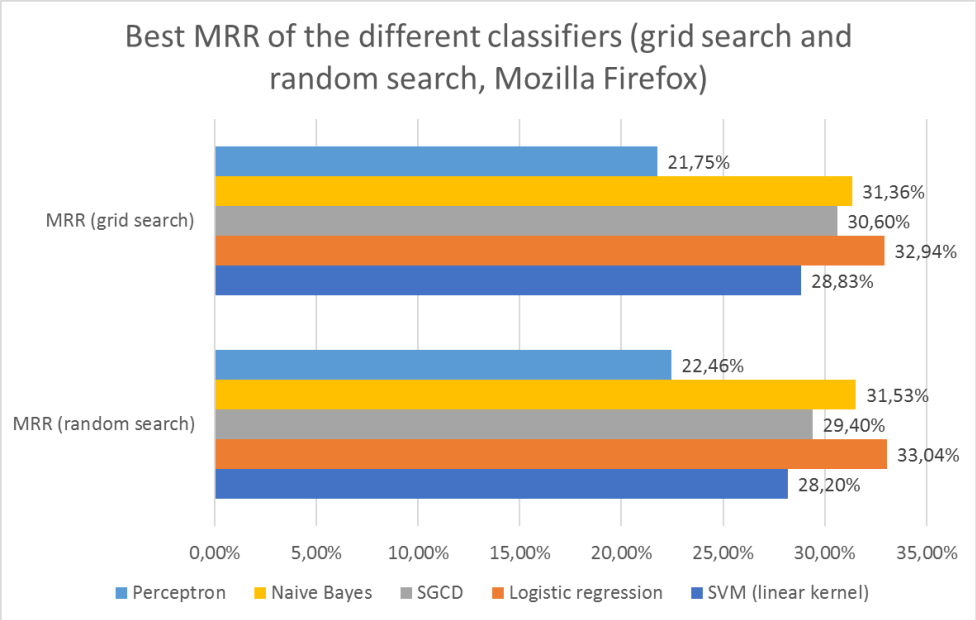


Figure 4.30: Best MRR of the different classifiers (grid search and random search) on the bug reports of Mozilla Firefox

stochastic gradient descent algorithm and the linear SVM classifier (Figures 4.29 and 4.30). The worst configuration, in terms of accuracy, is obtained when using the nearest centroid classifier (Figure 4.29).

5 Discussion

In this chapter, the results obtained in this work are first analyzed. The applied method is then discussed. Finally, the ethical and societal aspects related to this thesis are discussed.

5.1 Results

The results obtained in this study are discussed in this section.

5.1.1 Preliminary experiment

The results related to the preliminary experiment of this thesis (Section 4.1 and Appendix A) are analyzed below.

First sub experiment

In this section, the results of the first sub experiment of the preliminary experiment of the thesis (Section 4.1.1 and Appendix A.1) are discussed.

In all the configurations of the telecommunications company, the accuracy on the test set was above 40 % (Figures 4.1, A.1 and A.2). The accuracies on the test sets of all the configurations of Eclipse JDT and Mozilla Firefox was however below 30 % (Figures 4.2, A.3, A.4, 4.3, A.5 and A.6). The observed difference between the performance of a classifier on the bug reports of the telecommunications company and the other data sets is probably due to the fact that the number of classes is significantly lower in the first data set than in the other ones. Increasing the number of classes has thus a negative impact on the performance of a classifier. This observation is also valid for the second sub experiment of the preliminary experiment and the other main experiments of the thesis.

Regarding the configurations based on less than 8 folds, and, applied on the bug reports of the telecommunications company, as the accuracy on the test set decreases with an increase in the number of bug reports in the training set, a machine learning algorithm should make its predictions on a number of bug reports not greater than one eighth of the data set before being re-trained (Figures 4.1 and A.1). One eighth of the data set represents approximately the number of bug reports submitted during 6 months and a quarter. In their work, L. Jonsson et al. [20] have shown that the accuracy of a machine learning algorithm decreases when

it is trained on older data. When making its last predictions on the bug reports of more than one eighth of the data set, the decisions of a machine learning algorithm are based on knowledge learned from significantly old bug reports. This phenomenon is probably the cause of the drop in its accuracy. When the configuration is based on no more than 8 folds, the same observation could be made for the bug reports of Eclipse JDT (Figures 4.2 and A.3). On this data set, a classifier should make its predictions on a number of bug reports lower than one eighth of the data set before being re-trained. This observation is also valid for the configurations using less than 8 folds, and, applied on the bug reports of Mozilla Firefox (Figures 4.3 and A.5). On this data set, a classifier should not make its predictions on more than one eighth of the bug reports before being re-trained.

Regarding the configurations based on at least 8 folds of the telecommunications company (Figures 4.1, A.1 and A.2), the results are consistent with the findings made by L. Jonsson et al. [20]. The accuracy on the test set has generally increased, then, decreased. The maximum is reached when the number of bug reports in the training set is between 10 000 and 20 000. One fiftieth of the data set represents approximately the number of bug reports submitted during 1 month. Based on the results of this experiment, if the model is re-trained every month in production, it should be trained on 15 000 bug reports so that it could reach its optimum accuracy. Concerning the configurations based on more than 8 folds, the same observation could be made on the bug reports of Eclipse JDT (Figures 4.2, A.3 and A.4). The maximum accuracy is always reached when the number of bug reports in the training set is between 2 500 and 5 000. If a classifier is re-trained every $3\frac{3}{4}$ months (around one fiftieth of the data set is submitted during this period), each time, it should be trained on around 4 000 bug reports to reach its maximum accuracy. Concerning the configurations using at least 8 folds, and, applied on the bug reports of Mozilla Firefox, the maximum accuracy is reached when the size of the training set is between 3 000 and 10 000 (Figures 4.3, A.5 and A.6). If a classifier is re-trained every 3.8 months (during this period, around one fiftieth of the data set is generated), it should be trained on around 4 500 bug reports.

For four of the five data sets used by L. Jonsson et al. [20], the maximum accuracy was reached when the number of bug reports in the training set was between 1 000 and 2 000. For the remaining data set, the accuracy on the test set has steadily increased with an increase in the size of the training set. The authors have considered that the aforementioned behaviour was a special case. The differences between the results obtained on four of the five data sets of the above mentioned authors and the results obtained in this work are probably due to the following reason: the data sets used in the study of these authors are not the same as the ones used in this thesis.

Second sub experiment

In this section, the results of the second sub experiment of the preliminary experiment of the thesis (Section 4.1.2 and Appendix A.2) are discussed.

Concerning the bug reports of the telecommunications company and Eclipse JDT (Figures 4.4, A.7, A.8, 4.5, A.9 and A.10), as the results are not exactly similar to the ones obtained in the first preliminary experiment (Figures 4.1, A.1, A.2, 4.2, A.3 and A.4), the bug reports of the last folds are not representative of the whole data sets. Regarding the predictions made on the last quarters of the aforementioned data sets (Figures 4.4 and 4.5), the same observation as the one made regarding the first sub experiment of the preliminary experiment could be made. The drop in the accuracy of the classifier is probably related to the significant size of the test set.

Regarding the other configurations of the telecommunications company (Figures 4.4, A.7 and A.8), it seems that the general behaviour of the classifier is similar to the one observed in the first preliminary experiment (Figures 4.1, A.1 and A.2). The accuracy on the test set seems to increase, then, decrease. More specifically, when making its predictions on the bug reports of no more than the last eighth of the data set, the maximum accuracy of the machine

learning algorithm is reached if it has been trained on an amount of between 30 000 and 40 000 bug reports. Regarding the remaining configurations, the maximum is not reached with the bug reports in the data set. It is probably related to the fact that a significant proportion of the bug reports in the test set are not similar to the bug reports in the training set. Based on a discussion with an expert bug triager of the telecommunications company, when the experiment was conducted, the firm was developing a new software product. A growing proportion of bug reports related to this new product were consequently being submitted.

Concerning the other configurations of Eclipse JDT, although the accuracy on the test set has fluctuated on the last one, it has generally decreased with an increase in the size of the training set until stabilizing (Figures 4.5, A.9 and A.10). It seems that the increase in the size of the training set has added some noise. As with the bug reports of the telecommunications company, it is probably related to the fact that a significant proportion of the bug reports in the test set is different from the bug reports added to the training set.

Regarding the bug reports of Mozilla Firefox (Figures 4.6, A.11 and A.12), the behaviour of the classifier on the last fold is close to the behaviour observed in the first sub experiment conducted on the same data set (Figures 4.3, A.5 and A.6). It means that the last fold is relatively representative of the whole data set. When at least 8 folds were used, as in the results of the first sub experiment, the maximum accuracy was thus reached when the size of the training set was between 3 000 and 10 000.

5.1.2 Main experiments

The results related to the main experiments of this thesis (Section 4.2 and Appendix B) are analyzed below.

Experiment 1

In this section, the results related to the first main experiment of the thesis (Section 4.2.1 and Appendix B.1) are discussed.

The method used in this experiment is the answer to the first research question (Section 1.3). In the context of this experiment, several combinations of pre-processing techniques were applied on the bug reports. Several instances of the same classifier were trained on the bug reports differently pre-processed. The combination of pre-processing techniques which gave the best results was then selected.

When the conversion to lower case was used, the results were always better for all the software development projects. This observation is consistent with the choices made in some related scientific publications. In their papers, Helming et al. [18], Jonsson et al. [20] and Čubranić et al. [14] have applied the aforementioned pre-processing technique on their bug reports before training and testing some classifiers. The reasons behind the choices of the authors of these publications concerning this pre-processing technique were not explicitly mentioned. One can guess that, after having made some minor experiments, they noticed that they obtained better results when converting each token to lower case.

For the telecommunications company (Figures 4.7, B.1, 4.8 and B.2) and Mozilla Firefox (Figures 4.11, B.5, 4.12 and B.6), the best results, in terms of accuracy and MRR, have been reached when the tokens containing only numbers were removed. In terms of accuracy, the best results were reached when removing the stop words from the bug reports of the telecommunications company (Figures 4.7 and B.1) and Eclipse JDT (Figures 4.9 and B.3).

As can be seen, the best combinations of pre-processing techniques are not similar for the three projects studied in this thesis.

For the bug reports of any project, the best configuration of this experiment was not based on the use of a stemmer. This result is coherent with one of the findings of Čubranić et al. [14]. According to them, using a stemmer does not significantly affect the accuracy of a classifier used to solve the automatic bug assignment problem. In some other prior works, the

researchers have nevertheless used this pre-processing technique to probably improve their results [37, 9, 39]. In the aforementioned papers, the reasons related to the choices of their authors regarding this pre-processing technique were not precisely explained. If their choices were based on a comparison of the performance of their model when using and not using this technique, the difference in their results could be explained by the following reasons:

- the aforementioned authors might have used a stemmer different from the one used in this thesis;
- all the aforementioned researchers have introduced a new model not solely based on a classifier;
- the focus of Ye et al. [39] was on automatic fault localization and not on automatic bug assignment;
- their data sets were not exactly the same as the ones used in this thesis.

Using the method introduced in this thesis would have allowed the above mentioned authors to justify their choices regarding the use of a stemmer and would have allowed them to make the right choice based on their data sets.

Based on the results of the first experiment, the tokens which are not discriminative should be removed and this might be done via the use of regular expressions. As the use of the conversion to lower case had a positive impact on the performance of the linear SVM classifier, this pre-processing technique should probably also be used. For all the projects studied in the thesis, the best configuration of the first experiment was not based on the use of a stemmer. Nevertheless, as there exist various stemmers, and, only one instance of this pre-processing technique has been used in this thesis, no conclusion can be drawn from this experiment regarding them.

Experiment 2

In this section, the results related to the second main experiment of the thesis (Section 4.2.2 and Appendix B.2) are discussed .

The method used in the experiment 2 is the answer to the second research question (Section 1.3). Several feature extraction techniques have been used on the bug reports of three software development projects. A classifier was then trained on the output of each feature extraction technique. The feature extraction technique which output allowed us to obtain the best results, in terms of accuracy and MRR, was eventually selected.

In the first part of the experiment 2, except for the Mozilla Firefox project, the worst five configurations were extracting ten features using either NMF or LSI (Figures B.7, B.8, B.11 and B.12). For all the projects, the best two feature extraction configurations were not based on NMF or LSI, and, not relying on a Boolean representation of the bug reports (Figures B.7, B.8, B.11, B.12, B.15 and B.16). Except for the Mozilla Firefox project, the best configuration, in terms of accuracy, was only relying on the tf-idf weights (Figures 4.13, B.7, 4.15 and B.11). Regarding the bug reports of the Firefox project (Figure B.15), the second best configuration, in terms of accuracy, is also based on tf-idf weights (this configuration is less than 0.5 percentage point behind the first configuration which is only using tf weights). The LSI algorithm seems to be more suited for a representation only based on the tf-idf weights. Except for the Firefox project, the NMF algorithm gave its best results when running on a Boolean representation of the bug reports (Figures B.7, B.8, B.11 and B.12). Except with a Boolean representation of the bug reports of Mozilla Firefox, when extracting more features via NMF or LSI, the accuracy and the MRR value have increased (Figures B.7, B.8, B.11, B.12, B.15 and B.16).

In the second part of the experiment 2, except for Mozilla Firefox, the best configurations were the ones combining the tf-idf weights of the bug reports with another feature extraction

technique (Figures B.9, B.10, B.13 and B.14). In all the software development projects, the worst configurations were using NMF (Figures B.9, B.10, B.13, B.14, B.17 and B.18).

At least for the bug reports of the telecommunications company and Eclipse JDT, the results obtained in the second experiment are consistent with the choices made in some prior works regarding the feature extraction technique to use (Figures 4.13, B.7, 4.14, B.8, 4.15, B.11, 4.16 and B.12). The similarity index used by Shokripour et al. [30] was indeed based on the formula used to compute the tf-idf weights (Equation 2.2). Their choice was probably based on a comparison with the results obtained using some other similarity indices based on some other feature extraction techniques. In their approaches used to solve the automatic bug assignment problem, Bhattacharya et al. [9] and Jonsson et al. [20] have used the tf-idf weights to extract the features from the bug reports. In the aforementioned papers, the reasons related to the choices of their authors regarding this feature extraction technique were not precisely explained. One can guess that it was probably based on the results related to some minor experiments made using some other feature extraction techniques. In their paper, Thomas et al. [34] compared the results obtained using different feature extraction techniques. For all the projects studied in their work, even if they have not used any classifiers, the best results of their first experiment was obtained when using the tf-idf weights in the vector space model. Their results are similar to the results obtained in this thesis even if the data sets are not similar. In the context of the component assignment problem, Somasundaram et al. [32] compared the accuracies obtained using three different configurations: tf-idf and SVM, LDA and SVM, and, LDA and KL divergence. As in this thesis, the best results, in average, in terms of accuracy, were obtained when using a SVM classifier trained on tf-idf weights. In their work, Ahsan et al. [1] have compared the results obtained using different feature extraction techniques (tf-idf weights or tf weights combined with different configurations of LSI) and different machine learning algorithms. In their paper, they obtained their best results, in terms of accuracy, precision and recall, when using a SVM algorithm trained on 100 features extracted via the LSI algorithm run on the tf weights of the tokens of the bug reports. The difference in their results could be explained by the following reasons:

- they have not used a linear SVM classifier as in this thesis, but, a normal SVM classifier;
- they have used some feature selection techniques between the computation of the tf weights and the use of the LSI algorithm;
- they have filtered out more than 60 % of the bug reports they had downloaded so that their text classification problem contains only 18 classes (instead of 186);
- they have not only used the title and the description fields of the bug reports (they have also used the product, the component and the operating system fields of the bug reports);
- even if they have also used some bug reports of the Mozilla project, their data set was not the same as the ones used in this thesis.

Regarding the bug reports of Mozilla Firefox, I believe that it was a special case (Figures 4.17, B.15, 4.18 and B.16). The results are nevertheless still consistent with the above mentioned prior works. The accuracy and the MRR value of the configuration based on the tf-idf weights are thus just behind the same metrics values of the configuration using only tf weights. The tf weights are, furthermore, calculated with the first term of the equation used to compute the tf-idf weights (Equation 2.2). The obtained results could probably be justified by the fact that some tokens occur in a significant proportion of the bug reports of Mozilla Firefox, but, are still discriminative. Using the inverse document frequencies would have indeed a negative impact on the predictions of the classifier.

Experiment 3

In this section, the results related to the third main experiment of the thesis (Section 4.2.3 and Appendix B.3) are discussed.

The method used to conduct the experiment 3 is the answer to the third research question (Section 1.3). Several feature selection techniques with various proportions of remaining features have been used. The configuration on which a classifier gave the best results was eventually kept.

For all the software development projects studied in this thesis, the worst configuration was obtained when selecting 10 % of the initial features (Figures 4.19, B.19, 4.20, B.20, 4.21, B.21, 4.22, B.22, 4.23, B.23, 4.24 and B.24).

Except for Mozilla Firefox, for all the software development projects, the best configurations were the ones relying on the χ^2 feature selection technique (Figures 4.19, B.19, 4.20, B.20, 4.21, B.21, 4.22, and B.22). Before conducting further studies on the χ^2 technique, Xuan et al. [38] had compared the accuracy they could reach using four different feature selection techniques. As with the bug reports of the telecommunications company and Eclipse JDT, their best results were reached when using the χ^2 feature selection technique. In their publication, Alenezi et al. [2] had compared the use of five feature selection techniques before applying the naive Bayes classifier. They have also obtained their best results when using the χ^2 feature selection technique.

In their paper, Xuan et al. [38] have found that using a feature selection technique can increase the accuracy of a classifier. As mentioned in the section describing the results of the experiment 3 (Section 4.2.3), except with the bug reports of Mozilla Firefox, none of the feature selection techniques used in this thesis has increased the accuracy of the predictions of the classifier. The difference observed between both results could be explained by the following reasons:

- the finding made by the above mentioned authors was mainly based on the use of the naive Bayes machine learning algorithm (not on the linear SVM classifier);
- they have filtered out the bug reports fixed by the developers who had fixed less than 10 bug reports in their data sets;
- even if they obtained their best results when selecting 30 % or 50 % of the features, other percentages than the ones used in this thesis might have eventually improved the accuracy of the classifier;
- their data sets were not exactly the same as the ones used in this thesis.

For the bug reports of the telecommunications company, selecting 90 % of the features with the χ^2 test has, nevertheless, increased the value of the MRR metric (Figures 4.20 and B.20). Selecting 50 % of the features via the recursive feature elimination technique has, furthermore, increased the values of both the accuracy and the MRR metrics on the bug reports of Mozilla Firefox (Figures 4.23, B.23, 4.24 and B.24).

Even if the results obtained on the bug reports of Mozilla Firefox are consistent with the aforementioned prior works, I believe that it is a special case (Figures 4.23, B.23, 4.24 and B.24). The results are thus drastically different from the results obtained on the two other projects studied in this thesis. Using recursive feature elimination has, indeed, increased the accuracy of the linear SVM classifier only on this project. Contrary to the other projects, the best configuration was based on recursive feature elimination and not on χ^2.

Contrary to the studies of Xuan et al. [38], in this thesis, the best results were reached when selecting a number of features close to the original number of features. The difference observed in my results might also be due to the fact that I have used the linear SVM classifier. When Xuan et al. [38] used the SVM classifier in their study, they thus obtained some results similar to the results obtained in this thesis. Among three algorithms, when they applied both

some instance selection and feature selection techniques, the only machine learning algorithm for which the accuracy decreased, was SVM.

For all the projects studied in this thesis, there was at least one feature selection configuration which had divided the size of the feature vectors by 10, and, had decreased the values of the accuracy and the MRR metrics by less than 0.5 percentage point (Figures B.19, B.20, B.21, B.22, B.23 and B.24). This result is interesting because it shows that one can drastically reduce the computation cost related to the training phase of a classifier without having a significant negative impact on its performance.

Experiment 4

In this section, the results related to the forth main experiment of the thesis (Section 4.2.4 and Appendix B.4) are discussed.

The method used in the context of the experiment 4 is the answer to the forth research question (Section 1.3). Several classifiers have been tuned on the bug reports of the three projects, using a grid search strategy and a random search strategy. The best performing classifier, in terms of accuracy and MRR, has eventually been selected.

Except for Mozilla Firefox, for all the projects studied in this thesis, the best configuration, in terms of accuracy, was obtained when tuning either the linear SVM classifier or the logistic regression classifier (Figures 4.25, B.25, B.27, 4.27, B.29 and B.31). This result is consistent with the findings made in several prior works [4, 20, 1, 18]. In the context of automatic bug assignment, the authors of the aforementioned papers have compared the performance of several classifiers including SVM: in these papers, the SVM classifier has outperformed the other ones. Even if it seems that the above mentioned authors had not tuned the different classifiers before comparing them (except in the paper of Jonsson et al. [20], the authors did not explicitly mention that they had not used any tuning), the above mentioned results confirm the reliability of this thesis. Among the aforementioned authors, only Jonsson et al. [20] have tried to use a logistic regression classifier in their comparison. They have nevertheless not obtained all the results related to this classifier due to some out of memory issues, and, the time needed to train this classifier.

Regarding the bug reports of Mozilla Firefox, as with the previous experiments, I believe that it was a special case (Figures 4.29, B.33, B.35, 4.30, B.34 and B.36). The results are nevertheless still consistent with the aforementioned prior works. In terms of accuracy and MRR value, the linear SVM classifier is thus always one of the best five classifiers used in this study. More specifically, in terms of accuracy, the best configurations of the logistic regression (primal problem) are furthermore just behind the best configurations of the naive Bayes classifier based on a multinomial distribution (Figures B.33 and B.35). For both the grid search and random search strategies, the highest MRR values were reached when tuning the logistic regression solving a primal problem (Figures B.34 and B.36). In the papers of Anvik et al. [4], Jonsson et al. [20] and Ahsan et al. [1], the accuracy reached with a naive Bayes classifier was significantly lower than the accuracy reached when using a SVM classifier. Nevertheless, in the study of Helming et al. [18], the performance of a naive Bayes classifier was almost similar to the performance of a SVM classifier. For two of the three projects studied in the aforementioned paper, the naive Bayes classifier had thus better accuracy than the linear SVM classifier. The results of Helming et al. [18] therefore confirm the reliability of this thesis.

One of the findings of Bergstra et al. [7] was that the use of the random search strategy is more efficient than the use of the grid search strategy when tackling the hyper-parameter optimization problem. The results related to the experiment 4 of the thesis are consistent with the finding of the above mentioned authors. With the same computational budget, in terms of accuracy and MRR value, the results obtained with the random search strategy are almost similar to the ones obtained with the grid search strategy. For the bug reports of the telecommunications company and Mozilla Firefox, the configurations which reached the highest accuracy were furthermore found using a random search strategy.

5.2 Method

In all the experiments of the study, the approach of Tian et al. [35] has been used to build the data set. The bug reports have thus been sorted based on their reported dates. Training a machine learning algorithm on a shuffled data set would have had a positive impact on its accuracy. I nevertheless believe that the calculated accuracy on the validation set or the test set would not have been representative of its true performance. This algorithm would have made its predictions based on knowledge partly learned from bug reports historically submitted after the ones being analyzed. As this situation would not occur in production, I decided to sort the bug reports using their reported dates.

In the experiments 1, 2 and 3 of the thesis, only the linear SVM classifier, respectively, has been used to select a combination of pre-processing techniques, a feature extraction technique and a potential feature selection technique. This choice might have had an impact on the results obtained in this thesis. In their study, Xuan et al. [38] have compared the accuracies of three classifiers (SVM, kNN and naive Bayes) when combining the ICF instance selection technique with the χ^2 feature selection technique. They obtained the best results when using the naive Bayes classifier. The only classifier for which the accuracy decreased, was, however, SVM. This finding shows that the usefulness of a feature selection technique might be related to the classifier which is used. There might also be some correlations between the selected classifier and the optimal combination of pre-processing techniques, and, the selected classifier and the optimal feature extraction technique. The results of the first three experiments may also have impacted the following experiments. Conducting a full factorial experiment on all the factors of all the experiments of the thesis would be the ideal method. This approach is nevertheless practically not feasible due to combinatorial explosion and some continuous factors. The method introduced in this thesis is an heuristic mainly used to avoid the combinatorial explosion problem.

Although the results related to the first experiment of the thesis are consistent with the findings of Čubranić et al. [14], some other researchers have used stemming in their works [37, 9, 39]. This choice could be based on the fact that a specific stemmer had a positive impact on the predictions of their models. As the aforementioned authors might have used other stemmers than the one used in this thesis (the Porter stemming algorithm), these decisions are not considered as threats for the reliability of this Master's thesis. The focus of my work was on introducing a method to improve the accuracies of several classifiers solving the automatic bug assignment problem and not on the results themselves. The approach introduced in this work could be extended by comparing the impacts of several types of stemmers and lemmatizers in the framework of the first experiment.

In the context of the first experiment, the regular expressions used to remove less discriminative tokens might have a negative impact on the replicability of this study. Writing these regular expressions was based on my common sense and some discussions with some bug triagers of the telecommunications company. For each of these regular expressions, its impact on the accuracy of the linear SVM classifier was assessed. If the impact was positive, the regular expression was retained. Otherwise, it was deleted. This parameter of the first experiment should nevertheless not be considered as a threat to the reliability of the study. Using exactly the same regular expressions as the ones written in the context of this work, on the same data sets, should lead to the same results.

In their work, Anvik et al. [4] have used some heuristics[1] to label the bug reports of Mozilla Firefox and Eclipse. Due to the time constraints related to this thesis work, these heuristics have not been applied. In their study, Anvik et al. [4] have downloaded every bug report of Eclipse and Mozilla Firefox, resolved or assigned within a specific period of time. Contrary to my work, they have not only downloaded the bug reports with a FIXED resolution, and, with a RESOLVED, VERIFIED or CLOSED status. As most of their heuristics

[1] http://www.cs.ubc.ca/labs/spl/projects/bugTriage/assignment/heuristics.html

were used to label the bug reports not with a FIXED resolution, this decision should not have a major impact on my results. Even if the results of the thesis were impacted by this decision, as the focus of this study is mainly on the introduced approach and not on the results themselves, this choice should not be considered as a treat to the validity of this work. As in the work of Anvik et al. [4], if one wants to significantly improve the accuracy of a classifier, one could remove from the data set the bug reports solved by the developers who have not fixed at least a certain amount of bug reports recently. As in the paper of Shokripour et al. [30], one could also use a larger N ($N > 1$) when evaluating the performance of his or her algorithm with the top N accuracy metric. In this thesis, $N = 1$ has been selected because the emphasis was not on the accuracy itself, but on the benefits that the application of the introduced method could bring on the accuracies and the MRR values of the classifiers solving the automatic bug assignment problem. In production, using the approach introduced in this work on the bug reports of the telecommunications company would have increased the accuracy by up to 16.64 percentage points (the difference between the accuracy of the best configuration of the last experiment and the accuracy of the worst configuration of the first experiment). Applying the approach on the bug reports of Eclipse JDT would have increased the accuracy by up to 2.85 percentage points (the difference between the accuracy of the best configuration of the last experiment and the accuracy of the worst configuration of the first experiment). However, using this method on the bug reports of Mozilla Firefox would have decreased the accuracy by up to 2.75 percentage points (the difference between the accuracy of the best configuration of the last experiment and the accuracy of the best configuration of the first experiment). After having conducted some minor experiments, I have noticed that this decrease was due to the fact that the parameter `class_weight` of the constructors of several classifiers used in the forth experiment had been set to `"balanced"` (Section 3.4.4). As the data set of Mozilla Firefox was imbalanced, the accuracy has decreased when conducting the forth experiment. This problem could have been solved by adding some additional configurations to tune the hyper parameter `class_weight` (in the experiment 4). One could also ignore the last experiment of the approach introduced in this thesis if it has a negative impact on the accuracy. When ignoring the forth experiment, applying the approach on the bug reports of Mozilla Firefox would have thus increased the accuracy by up to 1.46 percentage point (the difference between the accuracy of the best configuration of the third experiment and the accuracy of the worst configuration of the first experiment).

5.3 The work in a wider context

According to Frey et al. [16], due to the abundance of data related to certain tasks and recent advances in artificial intelligence, not only the tasks that are explicitly definable with rules could be automated. Only the tasks, in which an agent needs to interact with an unstructured environment, use its creativity or interact with other people, are less likely to be automated. In their work, these authors have intended to estimate the probabilities of 702 jobs to be automated via the use of a Gaussian process classifier. They have found that the high-income jobs and the jobs that are usually done by more highly educated people are respectively less likely to be automated than the low-income jobs and the jobs that are done by less educated people. To the authors' mind, this result is a radical change with the present trend. During the nineteenth century, due to advances in production technologies, a significant proportion of tasks requiring more skills have been replaced by tasks requiring less skills. During the twentieth century, there has been a decrease in the demand of middle-income jobs due to automation. Based on the results of the aforementioned authors, the low-income jobs will mainly be impacted by automation during the twenty-first century.

Bug triage is usually not a full-time activity. It is thus a task done by one or several person(s) called bug triager(s). These people have generally a broad knowledge of the software development project and are sometimes managers. Although this task is generally not done

by a low-paid worker or less educated person of the software development project, this task is susceptible to being automated as the bug triager does not need to significantly interact with his or her environment, use his or her creativity, or, his or her social skills. The triager mainly needs to use his or her knowledge of the project to assign a bug report to a developer, or, a team having the required skills to solve the problem. As shown in this study, using the prior decisions taken by the human bug triagers, a significant proportion of this knowledge can be learned by some machine learning algorithms.

Due to the stressful, arduous and time consuming aspects of bug triage, carrying out this task was not appreciated by the bug triagers I have worked with in the telecommunications company. Automating, partially or totally, this task would allow them to save some time to do some other tasks that are less likely to be automated and less arduous. Automating this task would also have an obvious positive economic impact on all the software development projects relying on manual bug assignment.

6 Conclusion

Due to its significant cost in the maintenance of a product, the partial or total automation of bug triage via the use of machine learning algorithms would be advantageous for many software development projects.

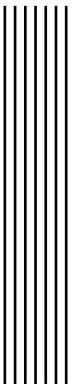
According to Thomas et al. [34], the configuration of a classifier impacts the quality of its predictions when solving the automatic bug localization problem. Based on the result of the aforementioned authors, the purpose of this thesis was to introduce a systematic approach to find some of the best existing configurations, in terms of accuracy and MRR, of several classifiers intending to solve the automatic bug assignment problem.

The method introduced in this thesis is an heuristic made up of four steps. In each of them, several configurations are compared on the first 90 % of the bug reports in the data set. The best configuration of each of the first three experiments is used in the following one(s). The accuracy and the MRR value of a linear SVM classifier are used to select the best configuration in each of the first three steps. In the first step, several combinations of pre-processing techniques are compared, and, the one which leads to the best accuracy is selected. The second step consists of comparing different feature extraction techniques and selecting the technique which gives the best results in terms of accuracy. In the context of the third step, different feature selection configurations are applied on the bug reports and the configuration which leads to the best accuracy is selected. In the last step, via the use of the grid search strategy and the random search strategy, several classifiers are tuned and the best performing one is eventually selected. Its performance is finally evaluated on the remaining 10 % of the bug reports.

The above mentioned method has been applied on three software development projects: 66 066 bug reports of a proprietary project of a telecommunications company, 24 450 bug reports of Eclipse JDT and 30 358 bug reports of Mozilla Firefox. 619 configurations have been applied and compared on each of these three projects. In production, when solving the automatic bug assignment problem, the application of the method introduced in this thesis on the bug reports of the telecommunications company has the potential to increase the accuracy by up to 16.64 percentage points.

Future work could focus on extending the method introduced in this thesis by adding more factors in each of its four steps. This could be achieved by using other pre-processing techniques, other bug reports representations, other feature selection techniques or other classifiers. Another possibility could be the use of the other fields of the bug reports. The use of

several classifiers in the first three steps of the method instead of the only use of the linear SVM classifier could be another alternative. Practitioners can also apply the method introduced in this work on other proprietary, or, open source projects, and, then, assess the impact of the use of the approach.



Bibliography

[1] Syed Nadeem Ahsan, Javed Ferzund, and Franz Wotawa. "Automatic software bug triage system (bts) based on latent semantic indexing and support vector machine". In: *Software Engineering Advances, 2009. ICSEA'09. Fourth International Conference on*. IEEE. 2009, pp. 216–221.

[2] Mamdouh Alenezi, Kenneth Magel, and Shadi Banitaan. "Efficient Bug Triaging Using Text Mining." In: *JSW* 8.9 (2013), pp. 2185–2190.

[3] John Anvik, Lyndon Hiew, and Gail C Murphy. "Coping with an open bug repository". In: *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*. ACM. 2005, pp. 35–39.

[4] John Anvik, Lyndon Hiew, and Gail C Murphy. "Who should fix this bug?" In: *Proceedings of the 28th international conference on Software engineering*. ACM. 2006, pp. 361–370.

[5] John Anvik and Gail C Murphy. "Reducing the effort of bug report triage: Recommenders for development-oriented decisions". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 20.3 (2011), p. 10.

[6] Ricardo Baeza-Yates, Berthier Ribeiro-Neto, et al. *Modern information retrieval*. Vol. 463. ACM press New York, 1999, pp. 19–34.

[7] James Bergstra and Yoshua Bengio. "Random search for hyper-parameter optimization". In: *Journal of Machine Learning Research* 13.Feb (2012), pp. 281–305.

[8] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. "Duplicate bug reports considered harmful... really?" In: *Software maintenance, 2008. ICSM 2008. IEEE international conference on*. IEEE. 2008, pp. 337–345.

[9] Pamela Bhattacharya, Iulian Neamtiu, and Christian R Shelton. "Automated, highly-accurate, bug assignment using machine learning and tossing graphs". In: *Journal of Systems and Software* 85.10 (2012), pp. 2275–2292.

[10] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.", 2009, pp. 60–62, 107–112.

[11] Christopher M.. Bishop. *Pattern recognition and machine learning*. Springer, 2006, pp. 1–4, 32–38, 179–196, 203–210, 325–345, 378–383, 663–666.

[12] David M Blei, Andrew Y Ng, and Michael I Jordan. "Latent dirichlet allocation". In: *Journal of machine Learning research* 3.Jan (2003), pp. 993–1022.

[13] Nick Craswell. "Mean reciprocal rank". In: *Encyclopedia of Database Systems*. Springer, 2009, pp. 1703–1703.

[14] Davor Čubranić. "Automatic bug triage using text categorization". In: *In SEKE 2004: Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*. Citeseer. 2004.

[15] Scott Deerwester, Susan T Dumais, George W Furnas, Thomas K Landauer, and Richard Harshman. "Indexing by latent semantic analysis". In: *Journal of the American society for information science* 41.6 (1990), p. 391.

[16] Carl Benedikt Frey and Michael A Osborne. "The future of employment: how susceptible are jobs to computerisation?" In: *Technological Forecasting and Social Change* 114 (2017), pp. 254–280.

[17] Isabelle Guyon, Jason Weston, Stephen Barnhill, and Vladimir Vapnik. "Gene selection for cancer classification using support vector machines". In: *Machine learning* 46.1 (2002), pp. 389–422.

[18] Jonas Helming, Holger Arndt, Zardosht Hodaie, Maximilian Koegel, and Nitesh Narayan. "Semi-automatic Assignment of Work Items." In: *ENASE* 10 (2010), pp. 149–158.

[19] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. "Improving bug triage with bug tossing graphs". In: *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM. 2009, pp. 111–120.

[20] Leif Jonsson, Markus Borg, David Broman, Kristian Sandahl, Sigrid Eldh, and Per Runeson. "Automated bug assignment: Ensemble-based machine learning in large scale industrial contexts". In: *Empirical Software Engineering* (2015), pp. 1–46.

[21] Daniel D Lee and H Sebastian Seung. "Learning the parts of objects by non-negative matrix factorization". In: *Nature* 401.6755 (1999), pp. 788–791.

[22] Zhongpeng Lin, Fengdi Shu, Ye Yang, Chenyong Hu, and Qing Wang. "An empirical study on bug assignment automation using Chinese bug data". In: *Empirical software engineering and measurement, 2009. ESEM 2009. 3rd international symposium on*. IEEE. 2009, pp. 451–455.

[23] Mario Linares-Vásquez, Kamal Hossen, Hoang Dang, Huzefa Kagdi, Malcom Gethers, and Denys Poshyvanyk. "Triaging incoming change requests: Bug or commit history, or code authorship?" In: *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE. 2012, pp. 451–460.

[24] Christopher D Manning, Prabhakar Raghavan, Hinrich Schütze, et al. *Introduction to information retrieval*. Vol. 1. 1. Cambridge university press Cambridge, 2008, pp. 269–273.

[25] Hoda Naguib, Nitesh Narayan, Bernd Brügge, and Dina Helal. "Bug report assignee recommendation using activity profiles". In: *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*. IEEE. 2013, pp. 22–30.

[26] Bo Pang, Lillian Lee, and Shivakumar Vaithyanathan. "Thumbs up?: sentiment classification using machine learning techniques". In: *Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10*. Association for Computational Linguistics. 2002, pp. 79–86.

[27] Claudia Perlich. "Learning curves in machine learning". In: *Encyclopedia of Machine Learning*. Springer, 2011, pp. 577–580.

[28] Stuart Jonathan Russell, Peter Norvig, John F Canny, Jitendra M Malik, and Douglas D Edwards. *Artificial intelligence: a modern approach*. 3rd ed. Vol. 2. Prentice hall Upper Saddle River, 2003, pp. 693–697.

[29] Fabrizio Sebastiani. "Machine learning in automated text categorization". In: *ACM computing surveys (CSUR)* 34.1 (2002), pp. 1–47.

[30] Ramin Shokripour, John Anvik, Zarinah M Kasirun, and Sima Zamani. "A time-based approach to automatic bug report assignment". In: *Journal of Systems and Software* 102 (2015), pp. 109–122.

[31] Ramin Shokripour, John Anvik, Zarinah M Kasirun, and Sima Zamani. "Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation". In: *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press. 2013, pp. 2–11.

[32] Kalyanasundaram Somasundaram and Gail C Murphy. "Automatic categorization of bug reports using latent dirichlet allocation". In: *Proceedings of the 5th India software engineering conference*. ACM. 2012, pp. 125–130.

[33] B Surendiran and A Vadivel. "Feature selection using stepwise ANOVA discriminant analysis for mammogram mass classification". In: *International J. of Recent Trends in Engineering and Technology* 3.2 (2010), pp. 55–57.

[34] Stephen W Thomas, Meiyappan Nagappan, Dorothea Blostein, and Ahmed E Hassan. "The impact of classifier configuration and classifier combination on bug localization". In: *IEEE Transactions on Software Engineering* 39.10 (2013), pp. 1427–1443.

[35] Yuan Tian, Dinusha Wijedasa, David Lo, and Claire Le Gouesy. "Learning to rank for bug report assignee recommendation". In: *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*. IEEE. 2016, pp. 1–10.

[36] Song Wang, Wen Zhang, and Qing Wang. "FixerCache: Unsupervised caching active developers for diverse bug triage". In: *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM. 2014, p. 25.

[37] Xihao Xie, Wen Zhang, Ye Yang, and Qing Wang. "Dretom: Developer recommendation based on topic models for bug resolution". In: *Proceedings of the 8th international conference on predictive models in software engineering*. ACM. 2012, pp. 19–28.

[38] Jifeng Xuan, He Jiang, Yan Hu, Zhilei Ren, Weiqin Zou, Zhongxuan Luo, and Xindong Wu. "Towards effective bug triage with software data reduction techniques". In: *IEEE transactions on knowledge and data engineering* 27.1 (2015), pp. 264–280.

[39] Xin Ye, Razvan Bunescu, and Chang Liu. "Learning to rank relevant files for bug reports using domain knowledge". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2014, pp. 689–699.

[40] Tong Zhang. "Solving large scale linear prediction problems using stochastic gradient descent algorithms". In: *Proceedings of the twenty-first international conference on Machine learning*. ACM. 2004, p. 116.

[41] Jian Zhou, Hongyu Zhang, and David Lo. "Where should the bugs be fixed?-more accurate information retrieval-based bug localization based on bug reports". In: *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press. 2012, pp. 14–24.

A Preliminary experiment

This appendix presents some of the results related to the preliminary experiment of the thesis.

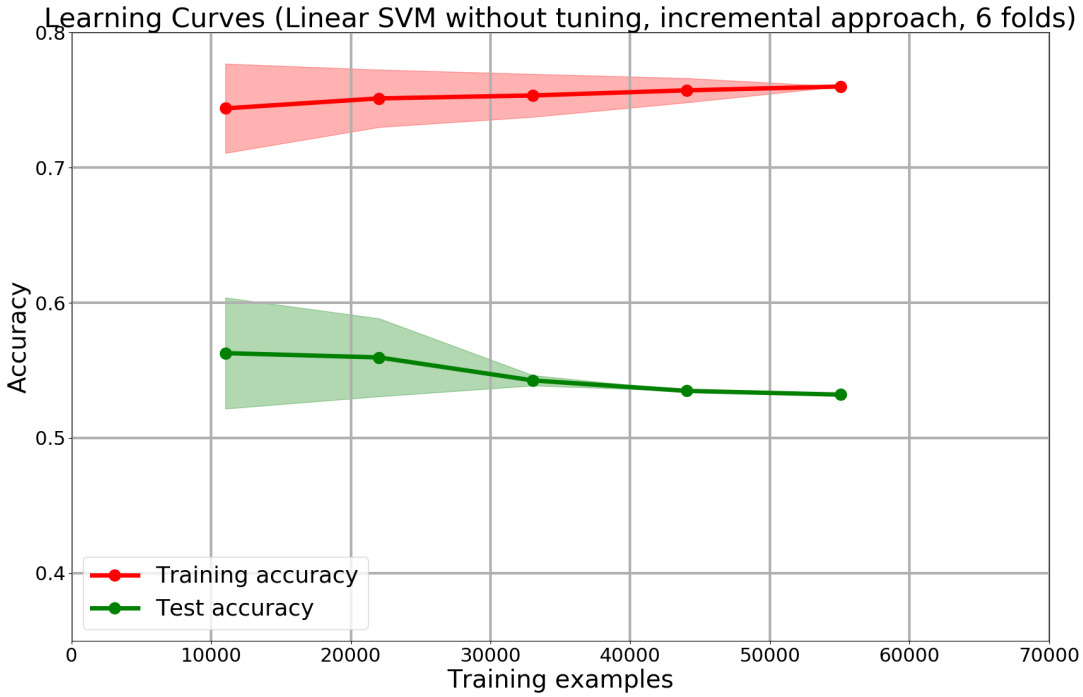
A.1 First sub experiment

In this section, some of the results of the first sub experiment of the preliminary experiment of the thesis are presented.

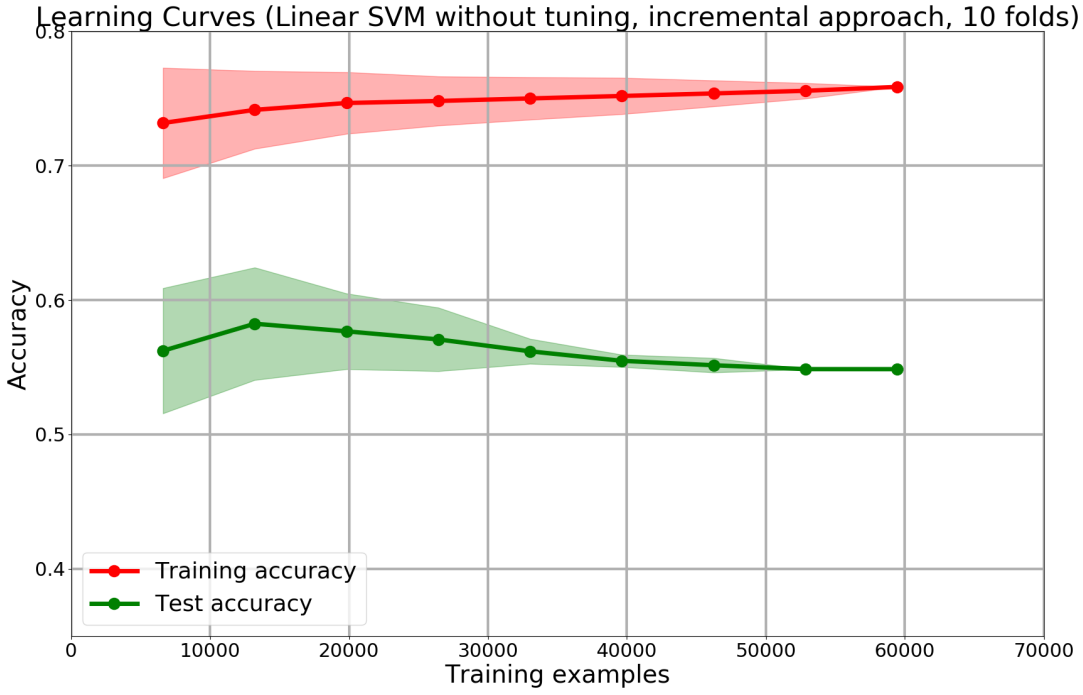
The graphs in the Figures A.1, A.2, A.3, A.4, A.5 and A.6 illustrate the learning curves obtained with different numbers of folds $K, K \in \{6; 10; 15; 25\}$. For each $K \in \{6; 10; 15; 25\}$, two learning curves are represented: the red one is related to the accuracy on the training set whereas the green one shows the accuracy on the test set. The standard deviation related to each point plotted in each chart is also represented.

A.1.1 Telecommunications company

In the Figures A.1 and A.2, the graphs show the learning curves obtained with 6, 10, 15 and 25 folds on the bug reports of the telecommunications company.

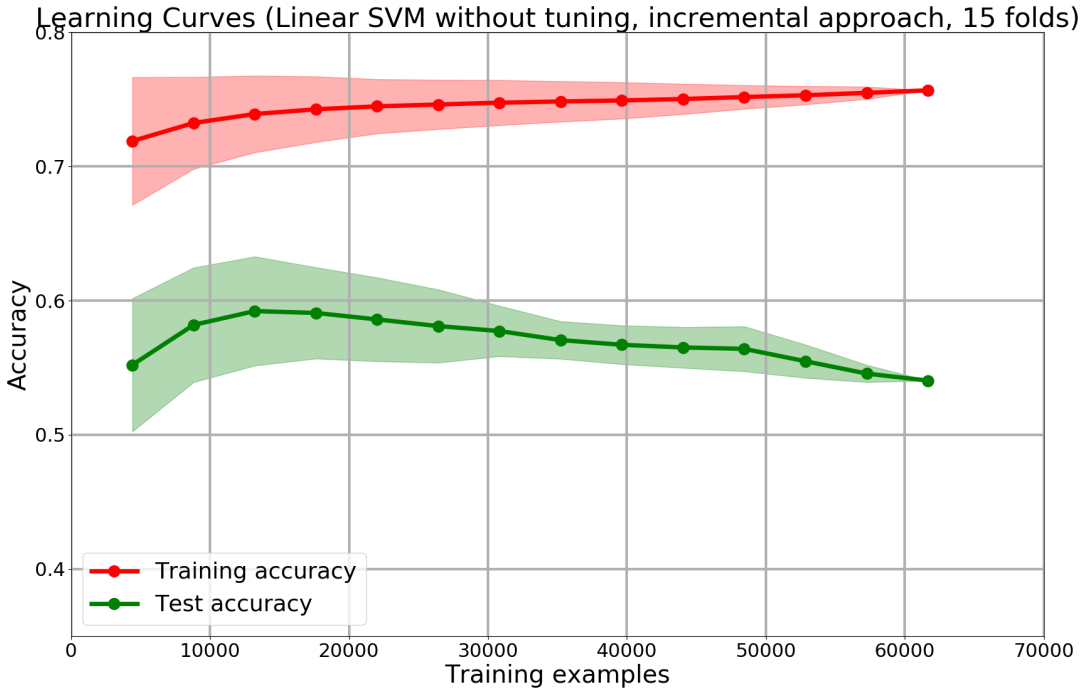


(a) 6 folds

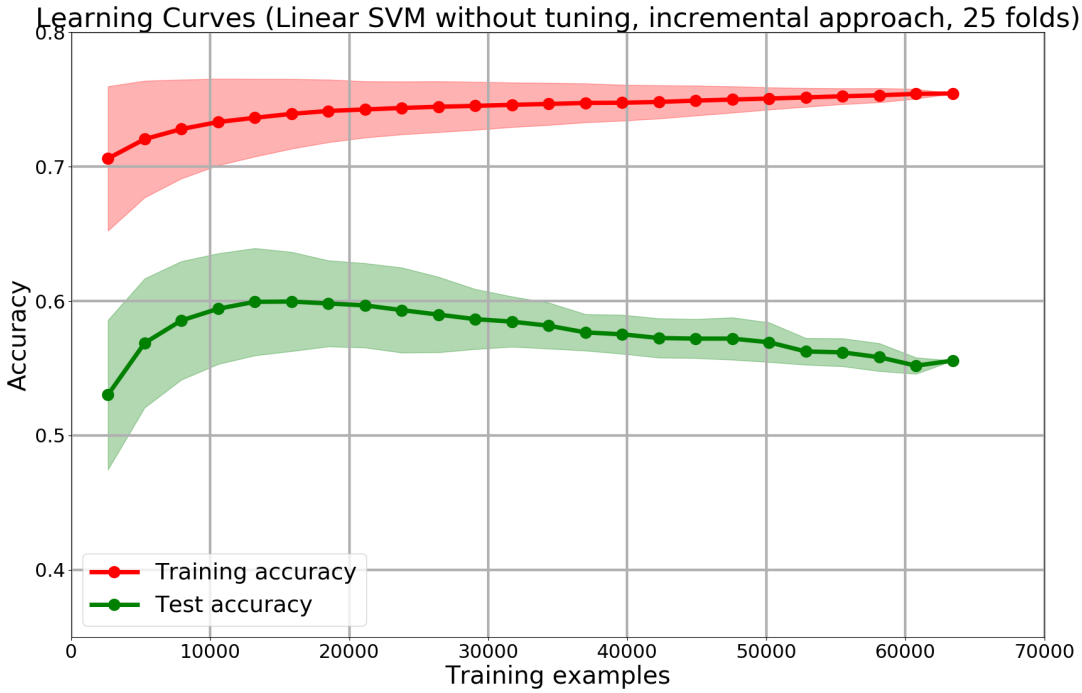


(b) 10 folds

Figure A.1: Learning curves of the first sub experiment of the preliminary experiment conducted on the bug reports of the telecommunications company (obtained with 6 and 10 folds)



(a) 15 folds

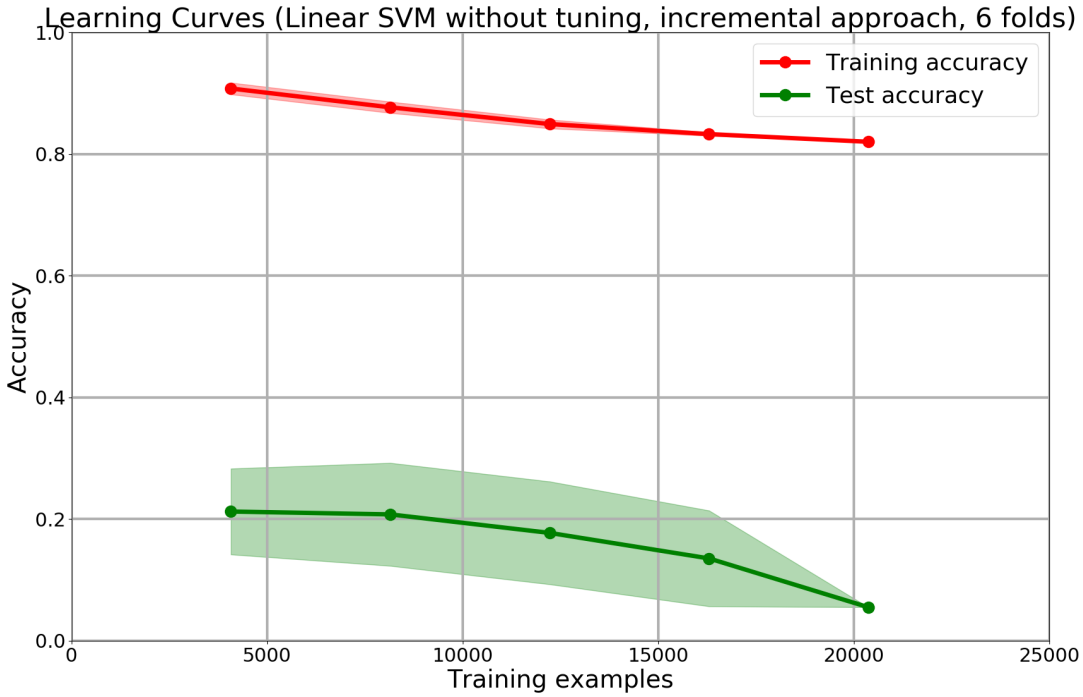


(b) 25 folds

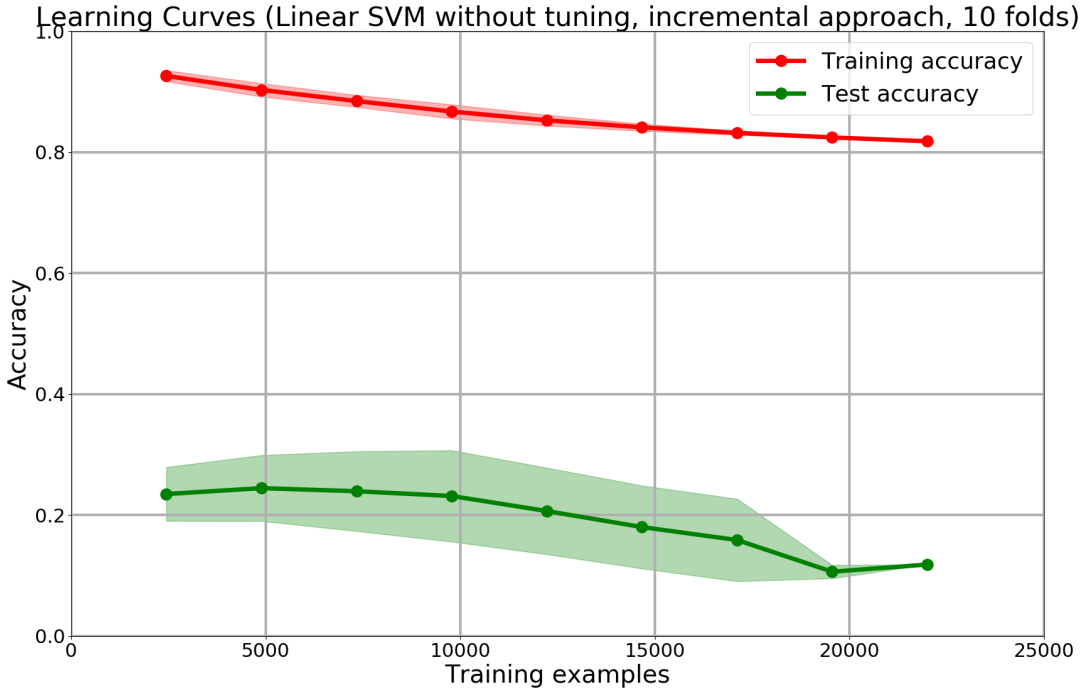
Figure A.2: Learning curves of the first sub experiment of the preliminary experiment conducted on the bug reports of the telecommunications company (obtained with 15 and 25 folds)

A.1.2 Eclipse JDT

The graphs in the Figures A.3 and A.4 depict the learning curves obtained with 6, 10, 15 and 25 folds on the bug reports of Eclipse JDT.

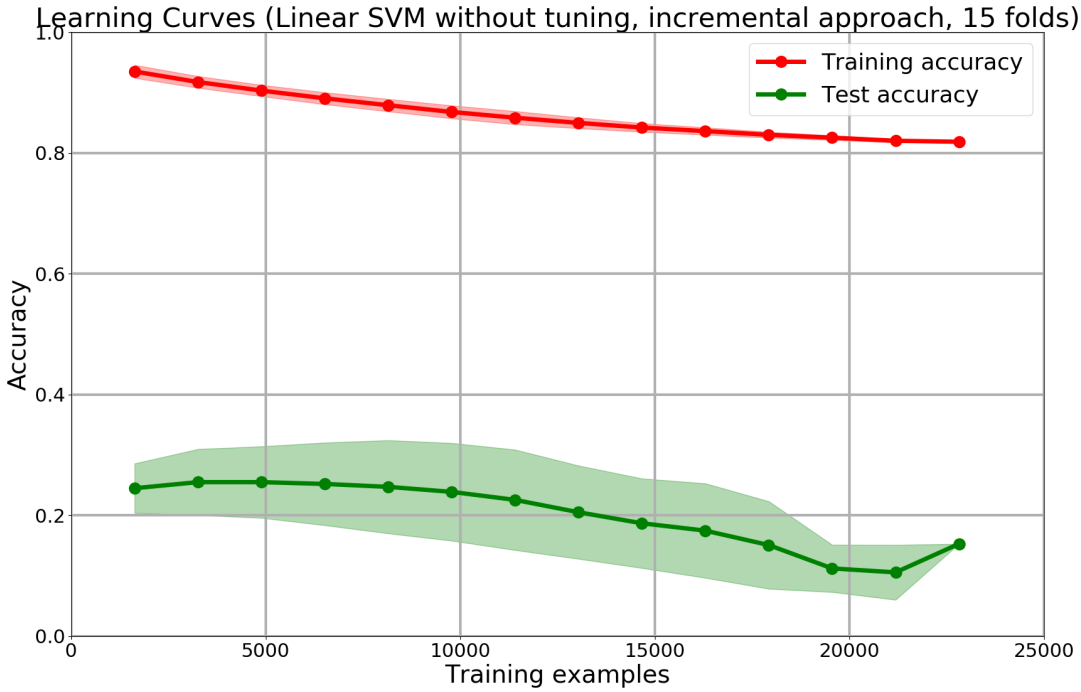


(a) 6 folds

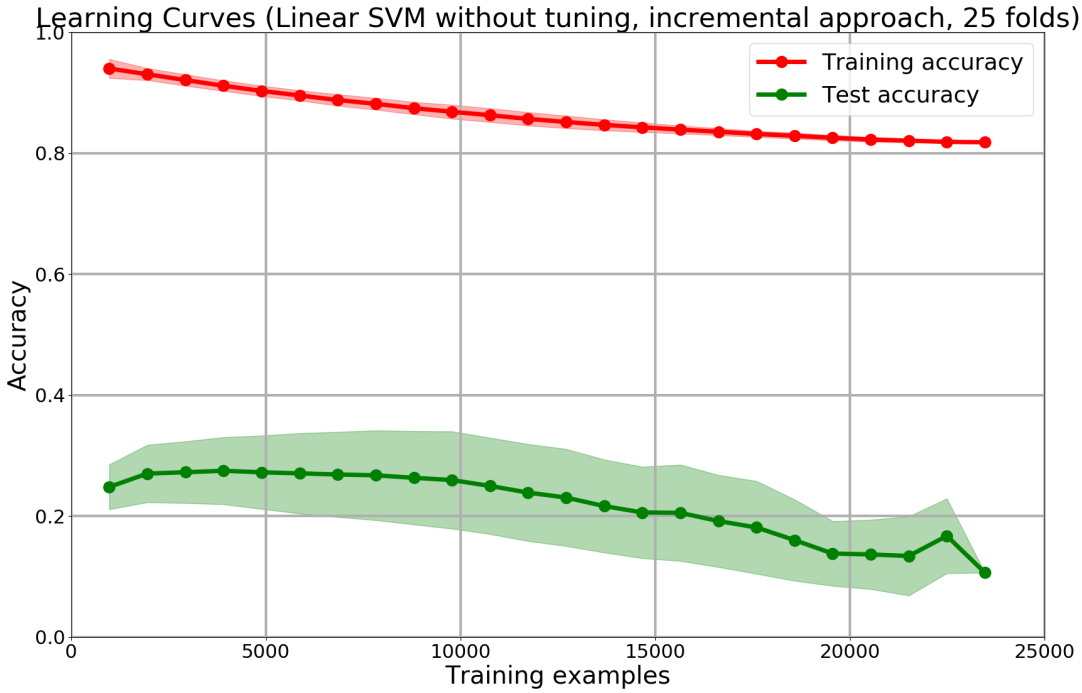


(b) 10 folds

Figure A.3: Learning curves of the first sub experiment of the preliminary experiment conducted on the bug reports of Eclipse JDT (obtained with 6 and 10 folds)



(a) 15 folds

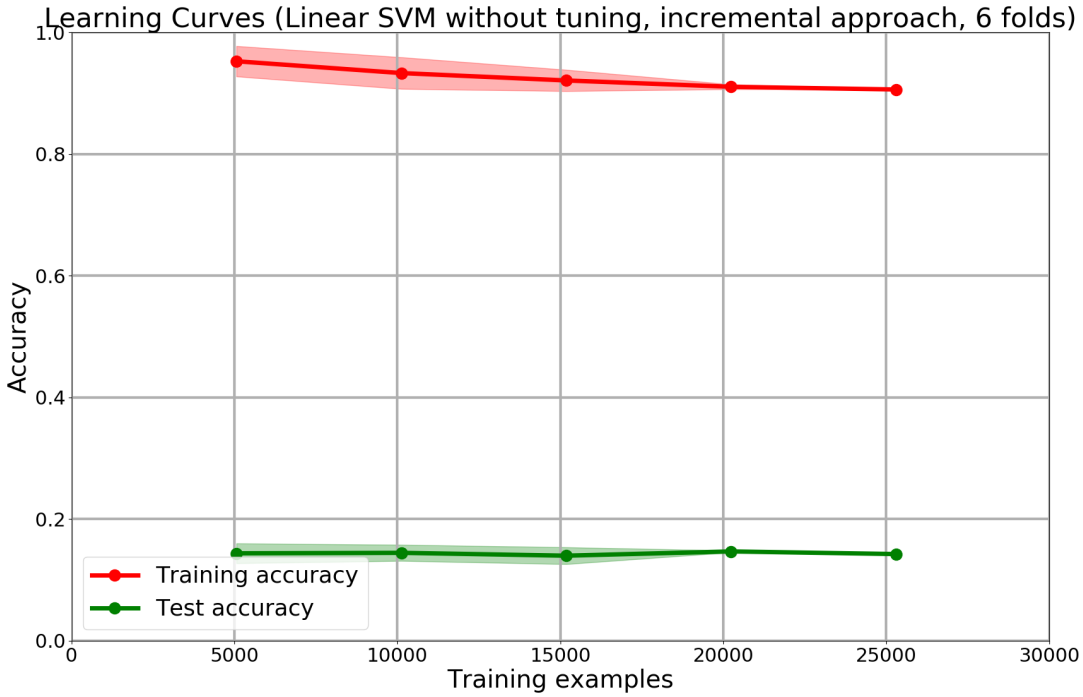


(b) 25 folds

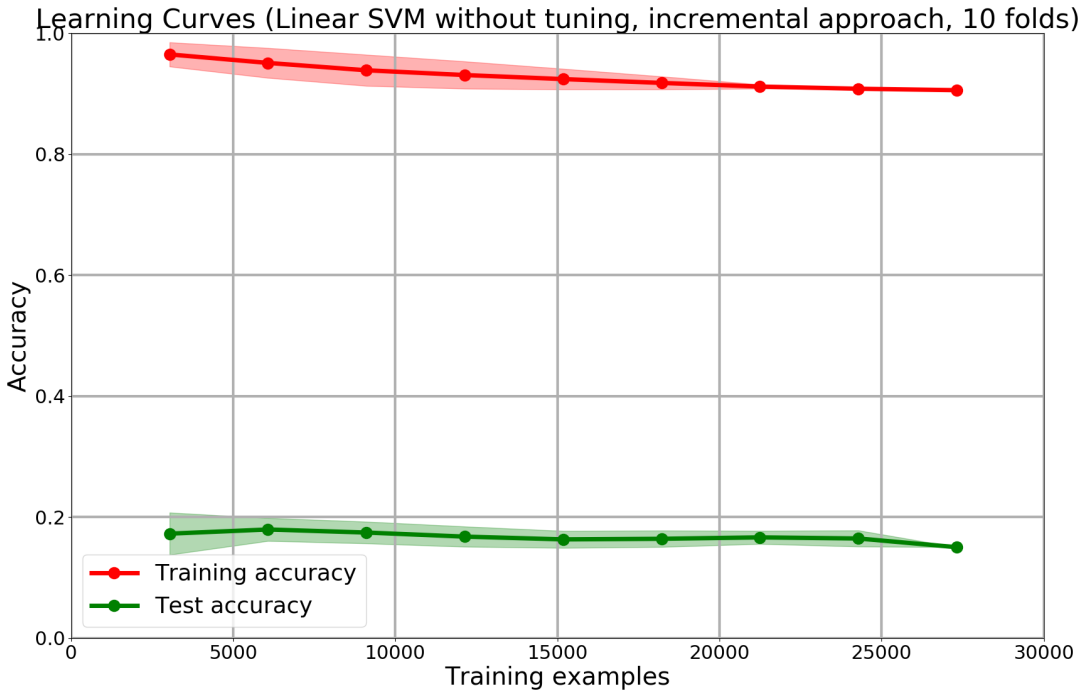
Figure A.4: Learning curves of the first sub experiment of the preliminary experiment conducted on the bug reports of Eclipse JDT (obtained with 15 and 25 folds)

A.1.3 Mozilla Firefox

In the Figures A.5 and A.6, the graphs show the learning curves obtained with 6, 10, 15 and 25 folds on the bug reports of Mozilla Firefox.

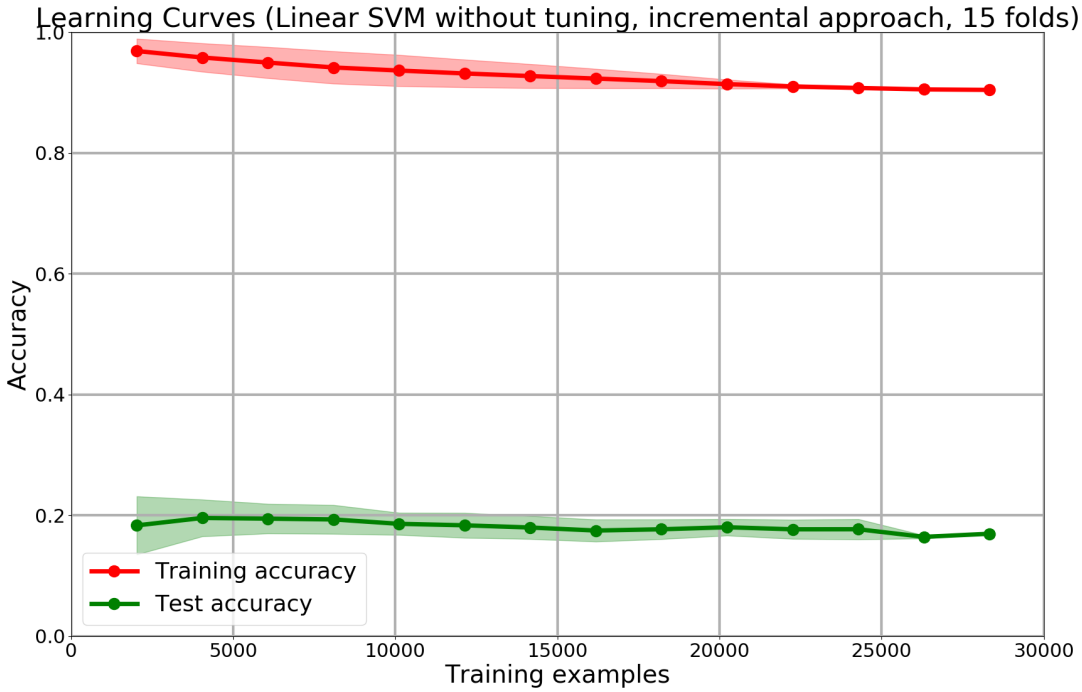


(a) 6 folds

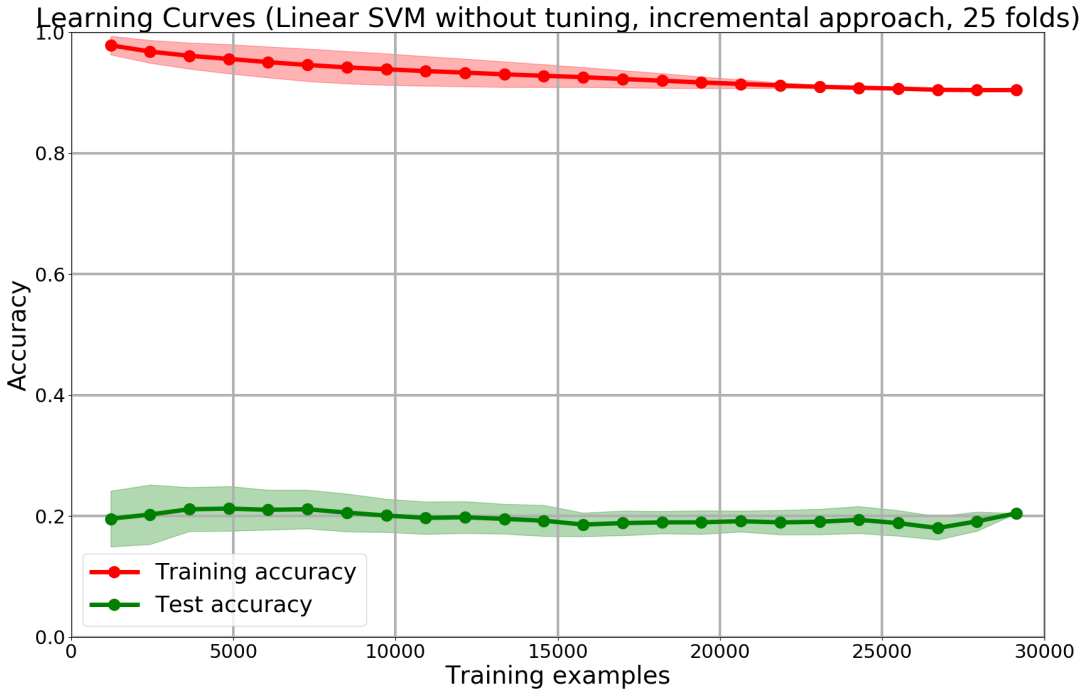


(b) 10 folds

Figure A.5: Learning curves of the first sub experiment of the preliminary experiment conducted on the bug reports of Mozilla Firefox (obtained with 6 and 10 folds)



(a) 15 folds



(b) 25 folds

Figure A.6: Learning curves of the first sub experiment of the preliminary experiment conducted on the bug reports of Mozilla Firefox (obtained with 15 and 25 folds)

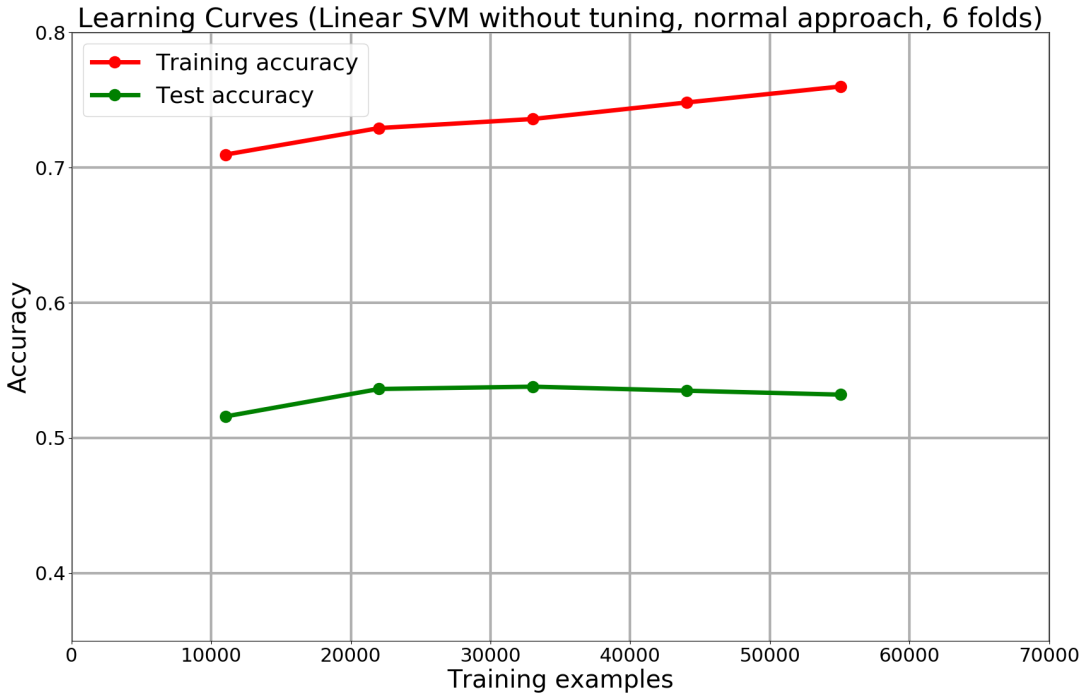
A.2 Second sub experiment

In this section, some of the results of the second sub experiment of the preliminary experiment of the thesis are presented.

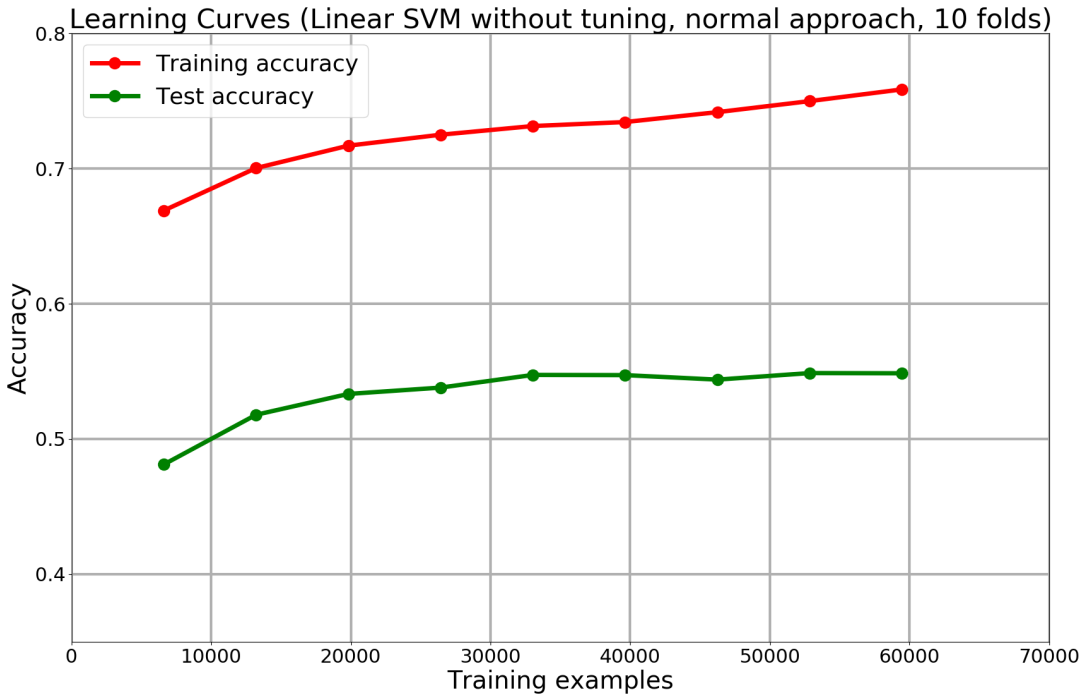
The graphs in the Figures A.7, A.8, A.9, A.10, A.11 and A.12 depict the learning curves obtained with different numbers of folds K, $K \in \{6; 10; 15; 25\}$. For each $K \in \{6; 10; 15; 25\}$, two learning curves are plotted: the red one is related to the accuracy on the training set whereas the green one shows the accuracy on the test set. The standard deviation related to each point plotted in each chart is also represented. As explained in the Section 3.3.2, contrary to the first sub experiment, only the last folds are used to plot the green learning curves.

A.2.1 Telecommunications company

The graphs in the Figures A.7 and A.8 depict the learning curves obtained with 6, 10, 15 and 25 folds on the bug reports of the telecommunications company.

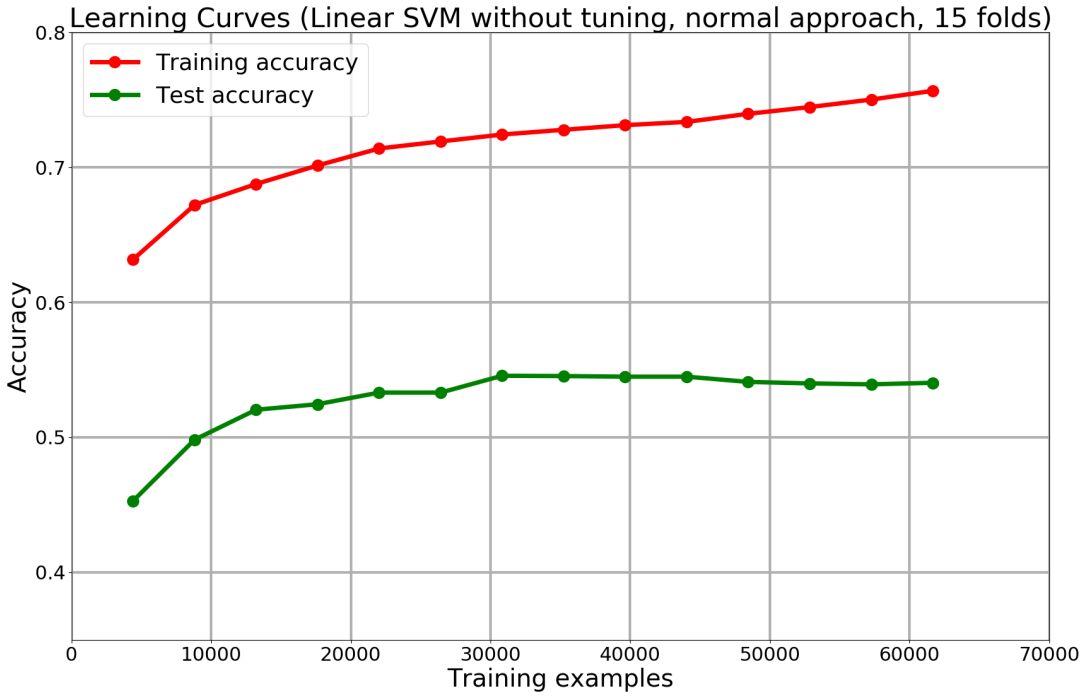


(a) 6 folds

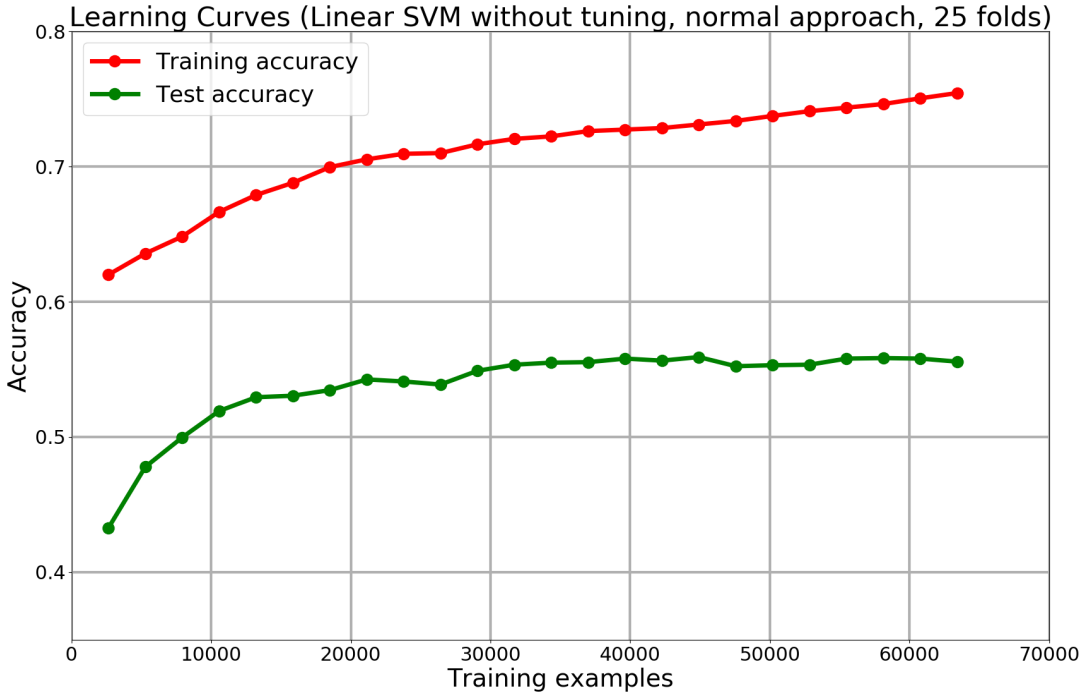


(b) 10 folds

Figure A.7: Learning curves of the second sub experiment of the preliminary experiment conducted on the bug reports of the telecommunications company (obtained with 6 and 10 folds)



(a) 15 folds

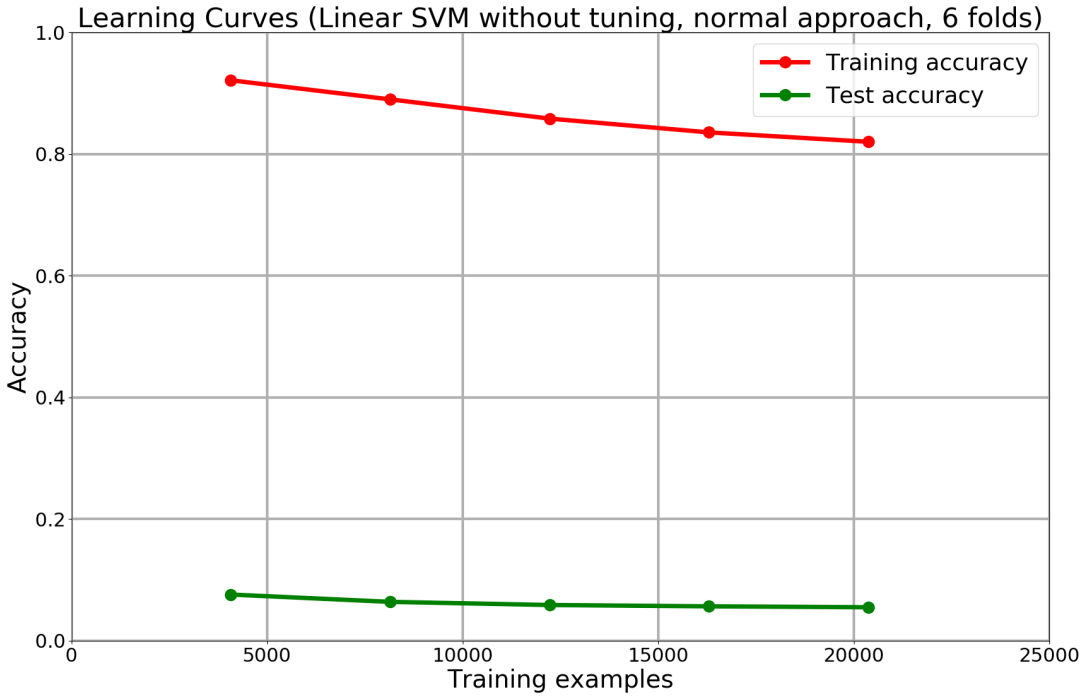


(b) 25 folds

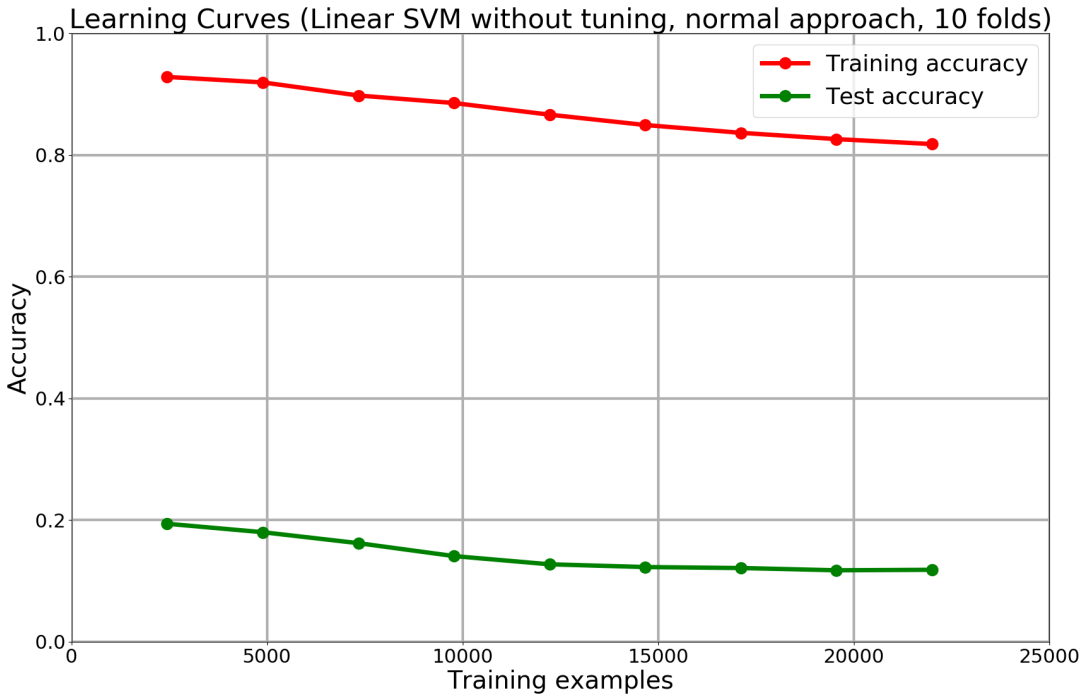
Figure A.8: Learning curves of the second sub experiment of the preliminary experiment conducted on the bug reports of the telecommunications company (obtained with 15 and 25 folds)

A.2.2 Eclipse JDT

In the Figures A.9 and A.10, the graphs show the learning curves obtained with 6, 10, 15 and 25 folds on the bug reports of Eclipse JDT.

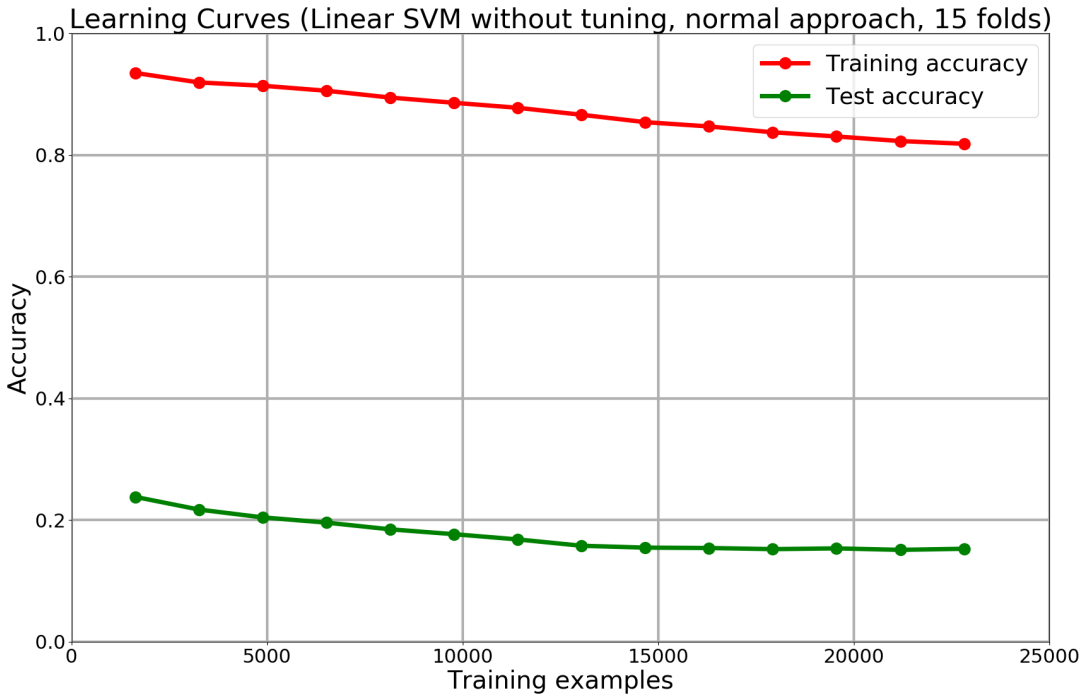


(a) 6 folds

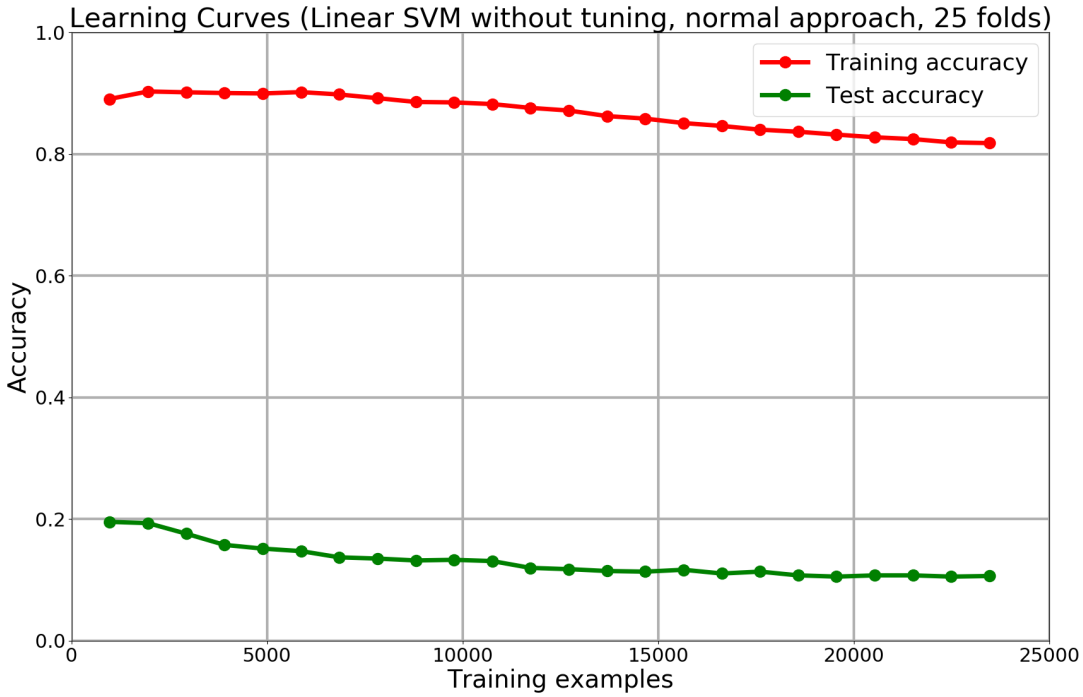


(b) 10 folds

Figure A.9: Learning curves of the second sub experiment of the preliminary experiment conducted on the bug reports of Eclipse JDT (obtained with 6 and 10 folds)



(a) 15 folds

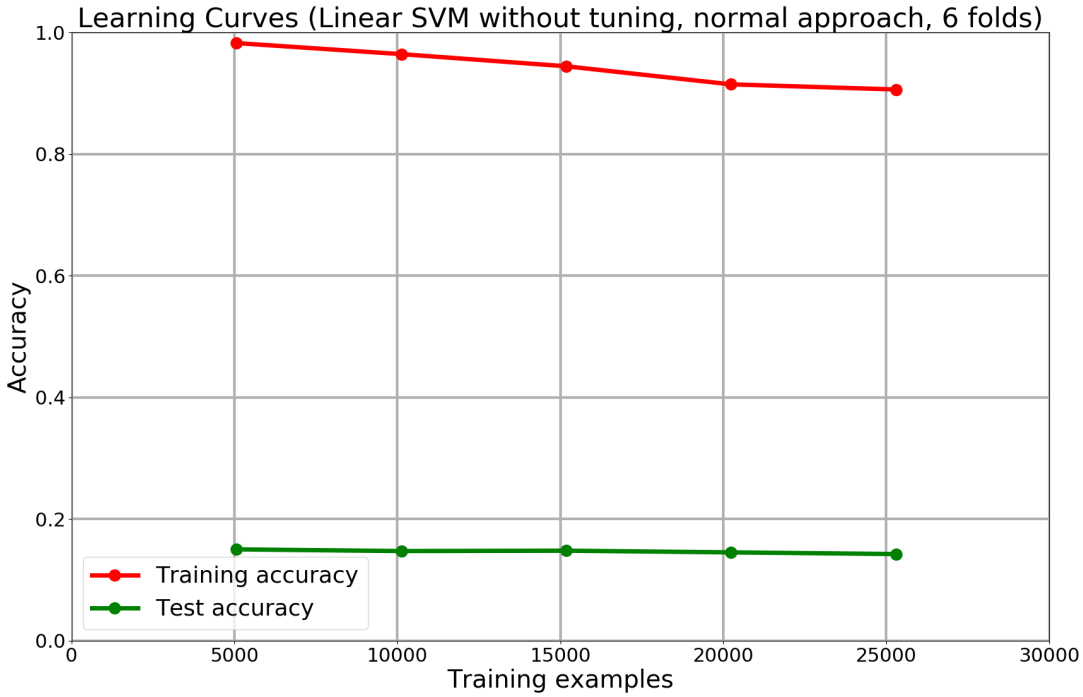


(b) 25 folds

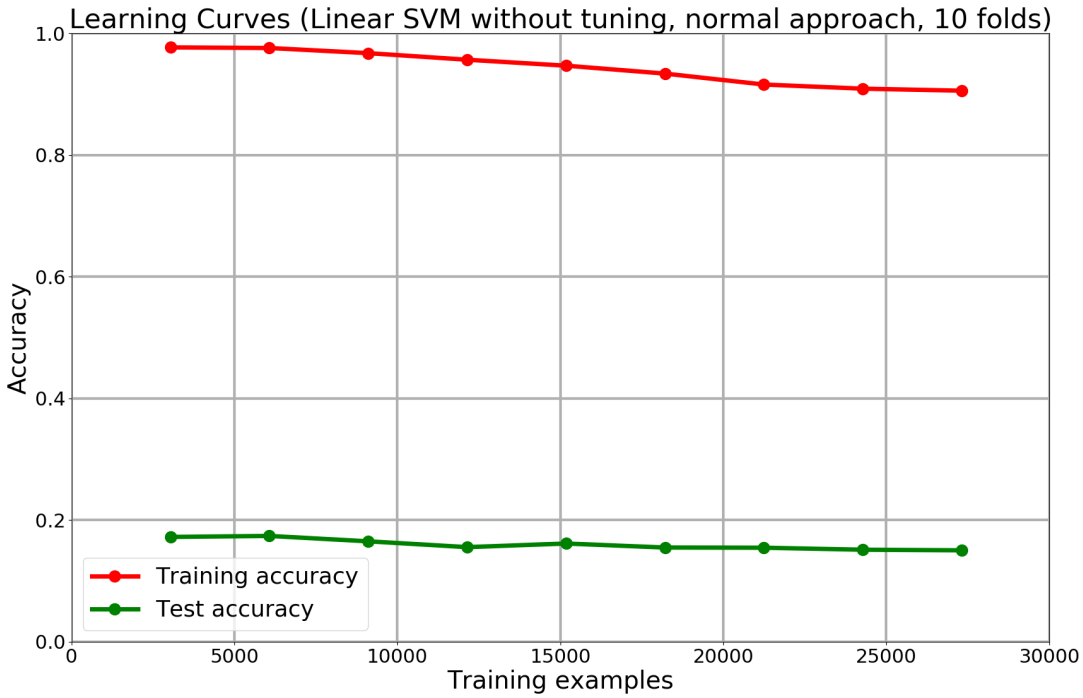
Figure A.10: Learning curves of the second sub experiment of the preliminary experiment conducted on the bug reports of Eclipse JDT (obtained with 15 and 25 folds)

A.2.3 Mozilla Firefox

The graphs in the Figures A.11 and A.12 depict the learning curves obtained with 6, 10, 15 and 25 folds on the bug reports of Mozilla Firefox.

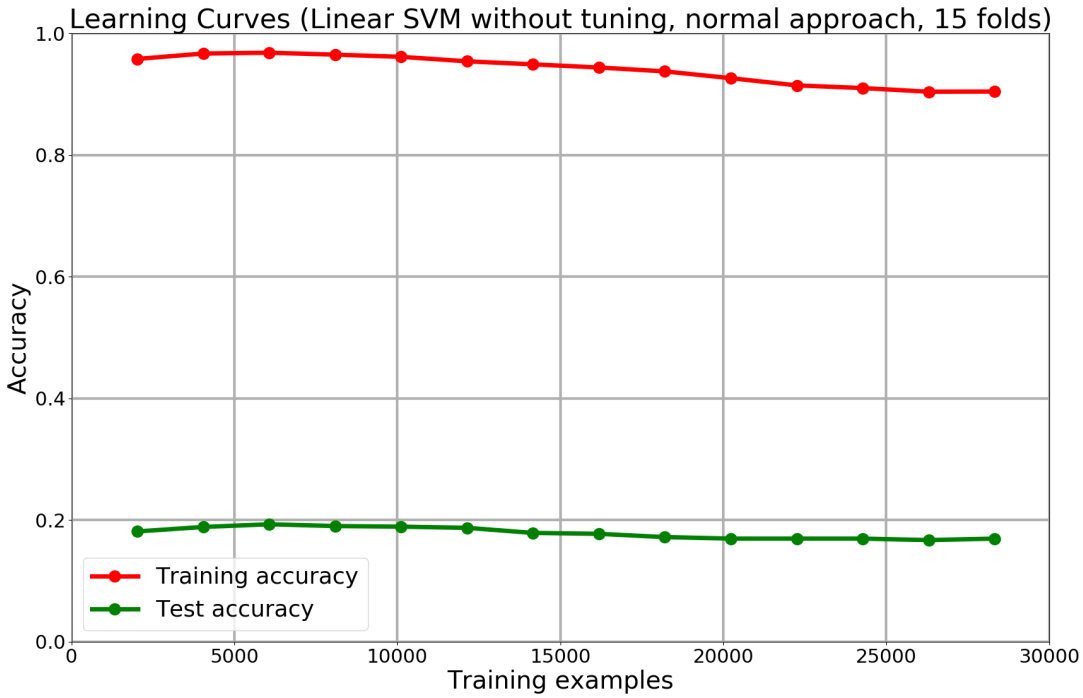


(a) 6 folds

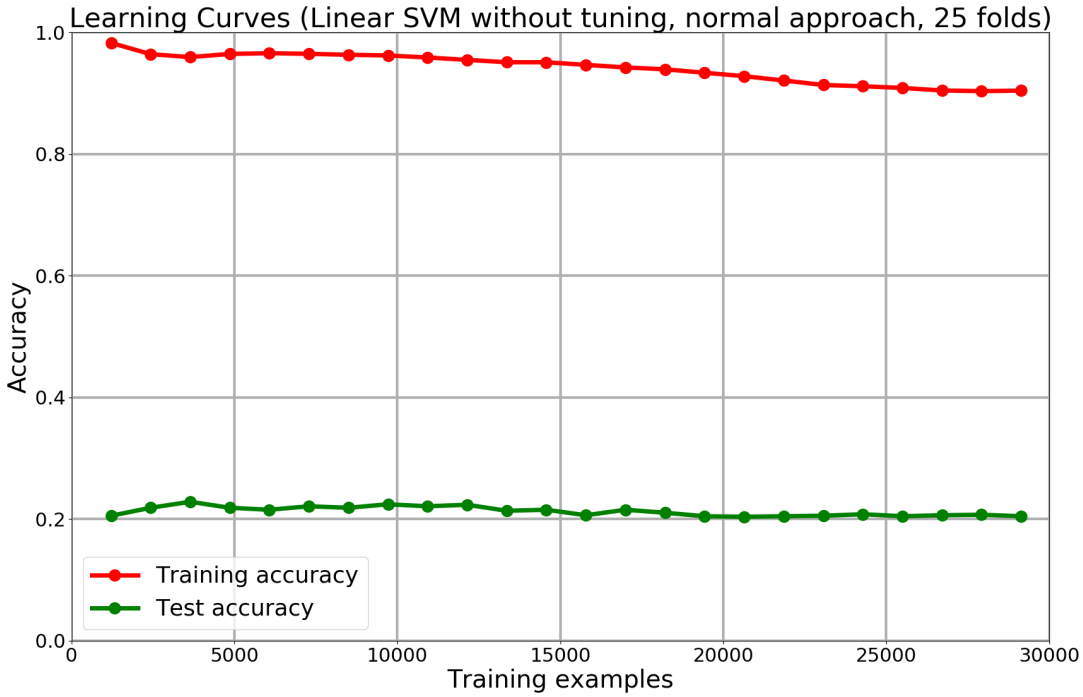


(b) 10 folds

Figure A.11: Learning curves of the second sub experiment of the preliminary experiment conducted on the bug reports of Mozilla Firefox (obtained with 6 and 10 folds)



(a) 15 folds



(b) 25 folds

Figure A.12: Learning curves of the second sub experiment of the preliminary experiment conducted on the bug reports of Mozilla Firefox (obtained with 15 and 25 folds)

B Main experiments

This appendix presents the detailed results related to the main experiments of the thesis.

B.1 Experiment 1

In this section, the detailed results related to the first main experiment of the thesis (Section 3.4.1) are described.

For readability purposes, the results are presented using the mapping of acronyms to pre-processing techniques defined in the Table 4.1. When a pre-processing technique is used, its acronym is directly written. If not, it is preceded by the string `NOT` and enclosed within parenthesis. All obtained strings are then concatenated and the character `|` is used as a delimiter between them. Next, the string related to each configuration is followed by a hash character `#` and a unique identifier.

B.1.1 Telecommunications company

The horizontal bar chart in the Figure B.1 depicts the accuracy of the different pre-processing configurations on the project of the telecommunications company.

The horizontal bar chart in the Figure B.2 shows the MRR value of the different pre-processing configurations on the bug reports of the telecommunications company.

In the best ten configurations of both charts (Figures B.1 and B.2), it can be seen that the bug reports have been cleaned. In the best five configurations, the stop words, the tokens containing only punctuation characters and the tokens containing only numbers have been removed. A conversion to lower case has been performed in the best three configurations. The best one was not based on stemming or lemmatization. Stemming has not been used in the best two configurations. In the worst twenty-four configurations, neither the bug reports have been cleaned nor the tokens containing only punctuation characters have been removed. The stop words have been removed in the worst ten configurations. The worst five configurations were not based on a conversion to lower case.

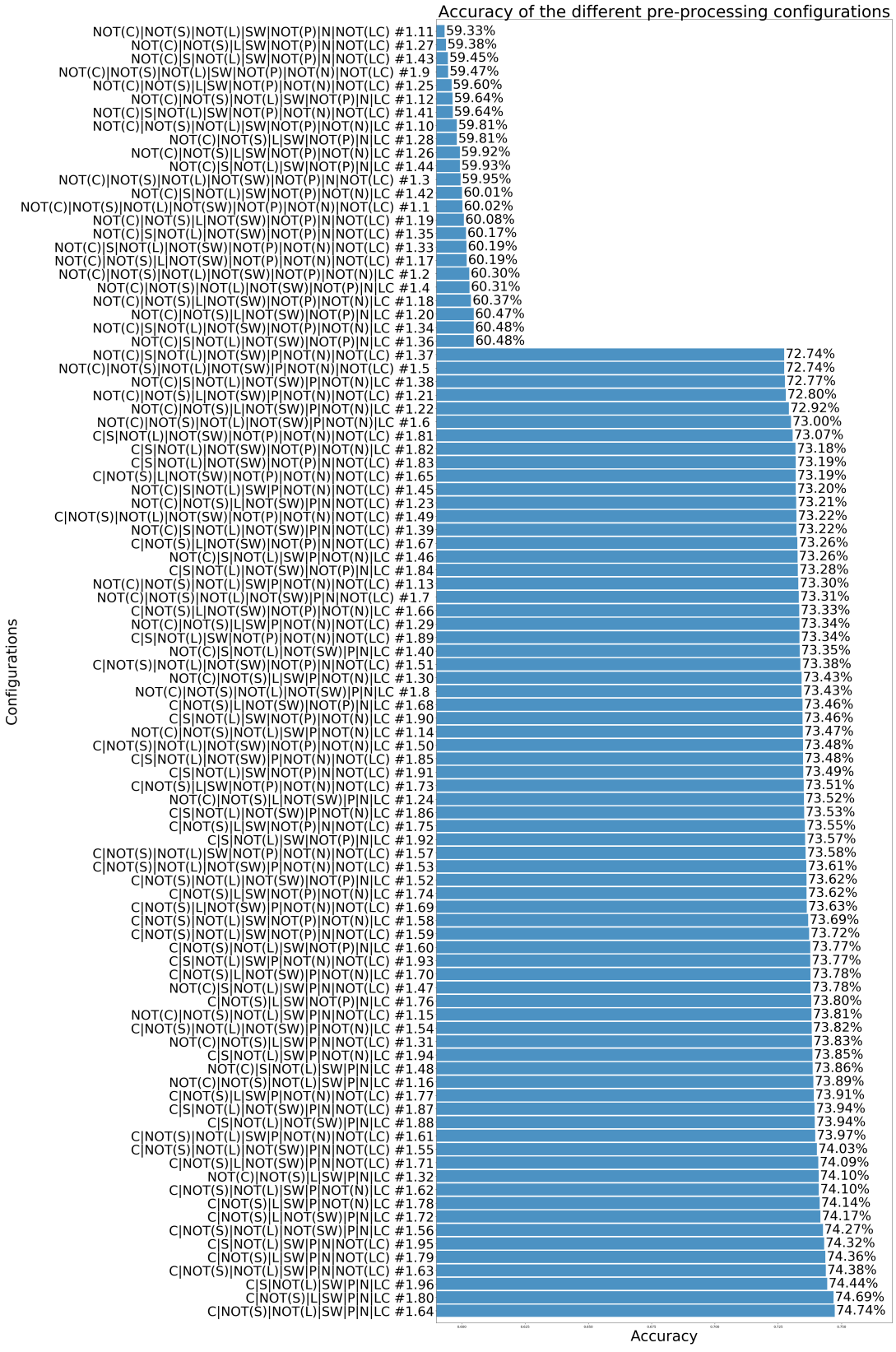


Figure B.1: Accuracy of the different pre-processing configurations on the bug reports of the telecommunications company (the mapping of acronyms to pre-processing techniques defined in the Table 4.1 has been used)

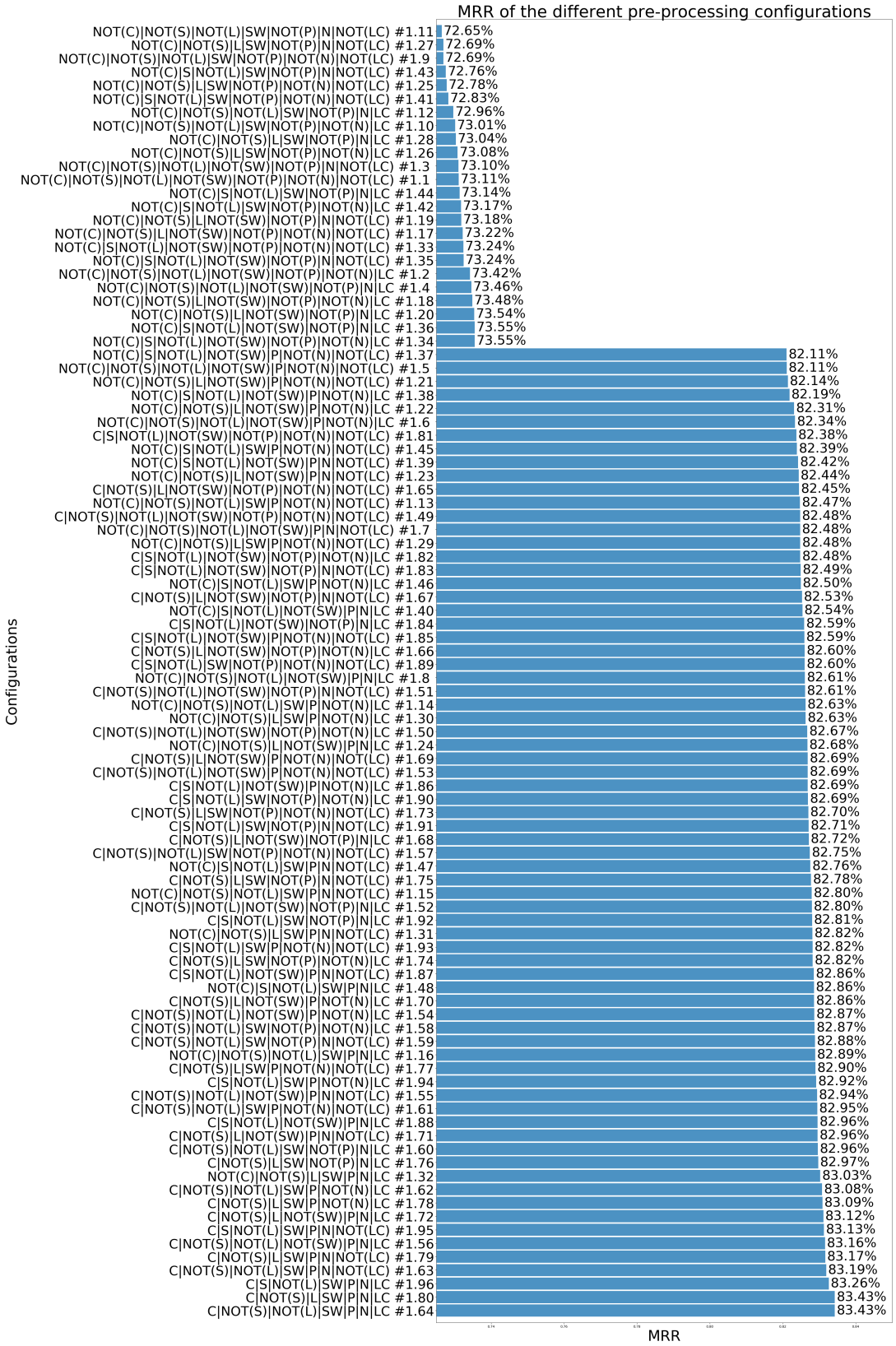


Figure B.2: MRR of the different pre-processing configurations on the bug reports of the telecommunications company (the mapping of acronyms to pre-processing techniques defined in the Table 4.1 has been used)

B.1.2 Eclipse JDT

The horizontal bar chart in the Figure B.3 shows the accuracy of the different pre-processing configurations on the bug reports of the Eclipse JDT project.

The horizontal bar chart in the Figure B.4 depicts the MRR value of the different pre-processing configurations on Eclipse JDT.

In the best configurations of both charts (Figures B.3 and B.4), the bug reports have been cleaned. In the best twenty configurations, a conversion to lower case has been performed. Stemming has not been used in the best ten configurations. In the worst fourteen configurations, no conversion to lower case has been used. The configurations are not ordered exactly the same way in both charts.

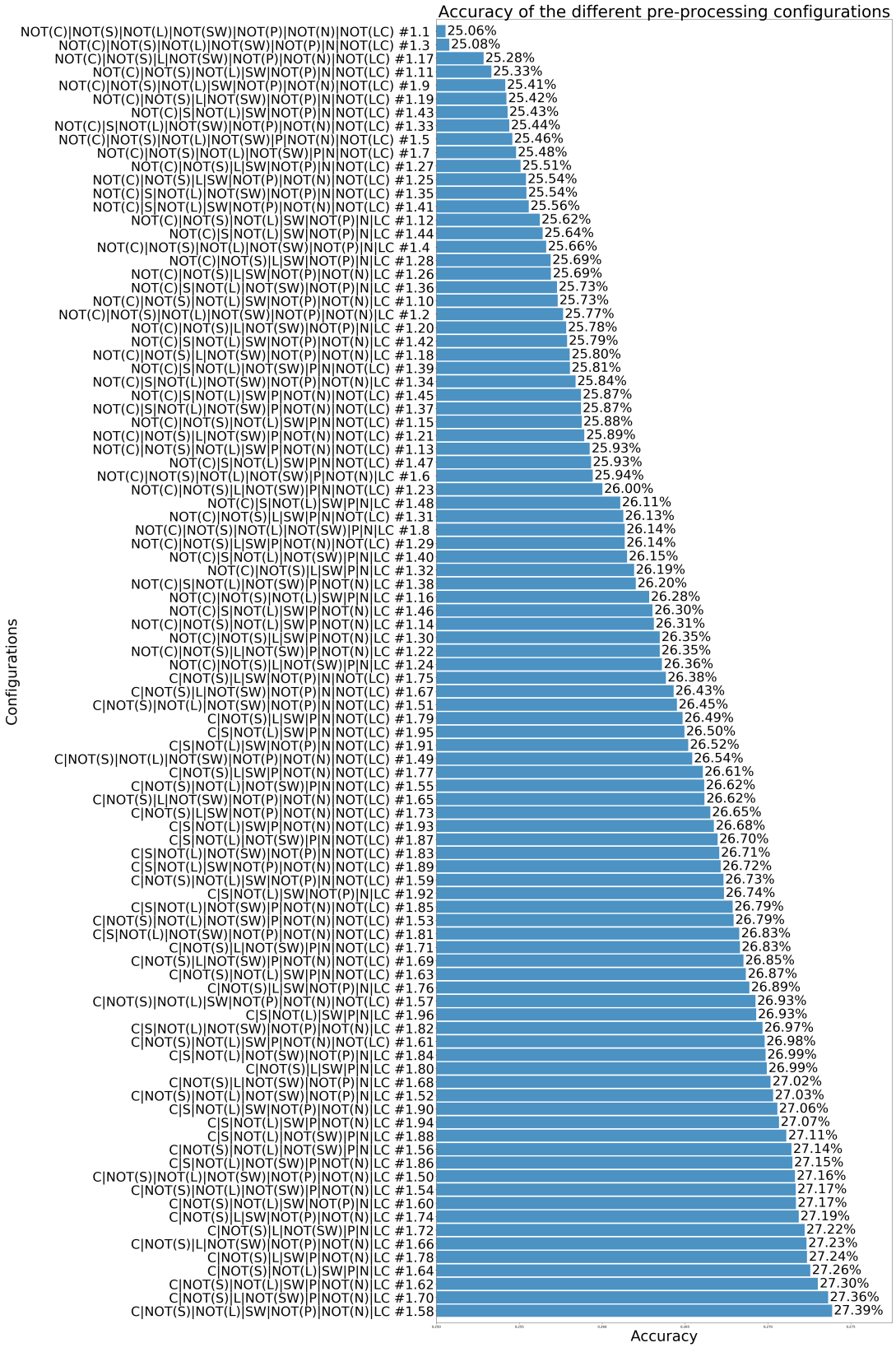


Figure B.3: Accuracy of the different pre-processing configurations on the bug reports of Eclipse JDT (the mapping of acronyms to pre-processing techniques defined in the Table 4.1 has been used)

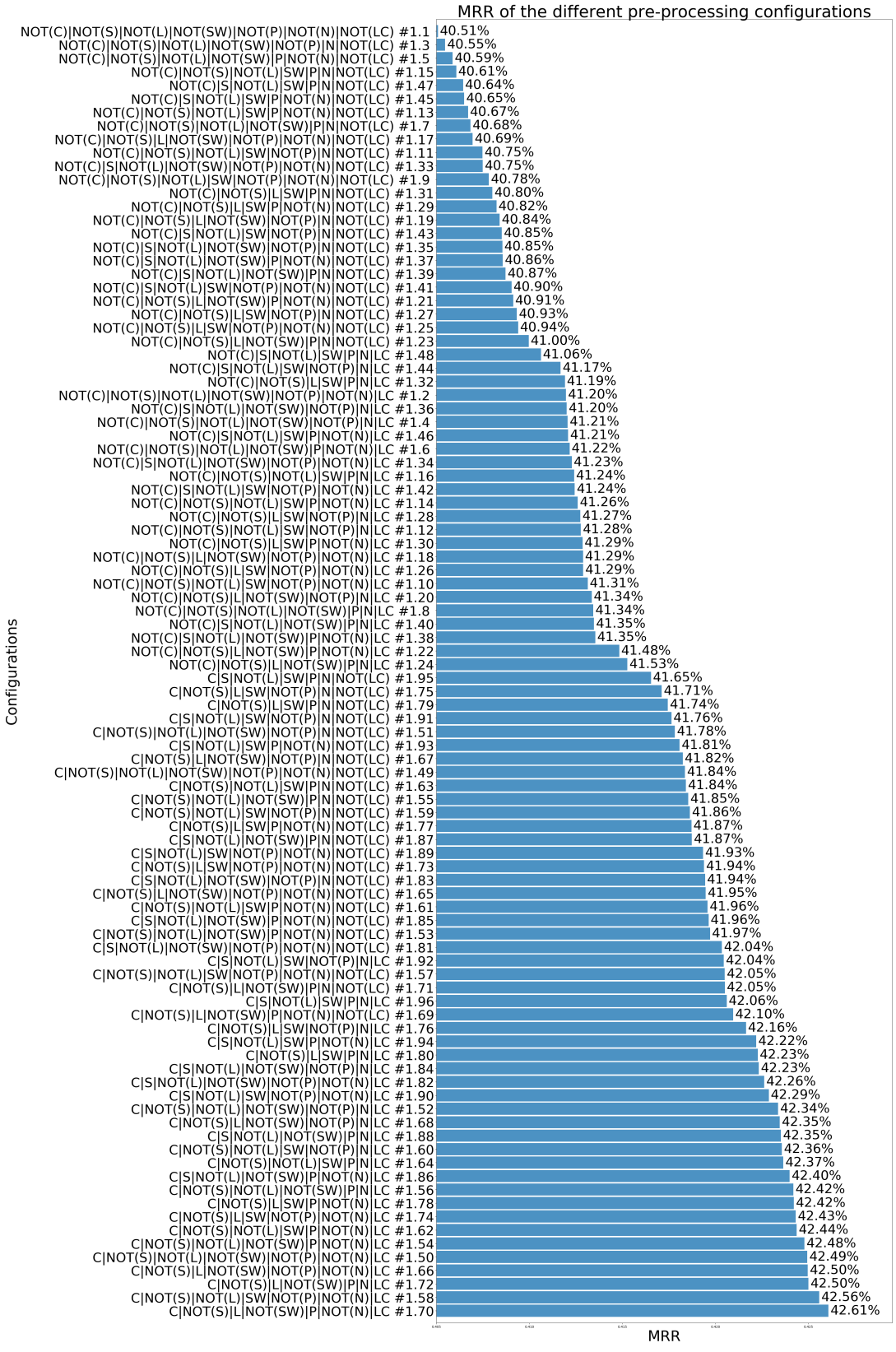


Figure B.4: MRR of the different pre-processing configurations on the bug reports of Eclipse JDT (the mapping of acronyms to pre-processing techniques defined in the Table 4.1 has been used)

B.1.3 Mozilla Firefox

The horizontal bar chart in the Figure B.5 illustrates the accuracy of the different pre-processing configurations on the bug reports of the Mozilla Firefox project.

The horizontal bar chart in the Figure B.6 depicts the MRR value of the different pre-processing configurations on Mozilla Firefox.

The configurations are not ordered exactly the same way in both charts (Figures B.5 and B.6). In the best three configurations of both charts, the bug reports have nevertheless been cleaned. In the best sixteen configurations, a conversion to lower case has been performed. In the worst six configurations, the bug reports have not been cleaned and no conversion to lower case has been used. Stemming has not been used in the best three configurations.

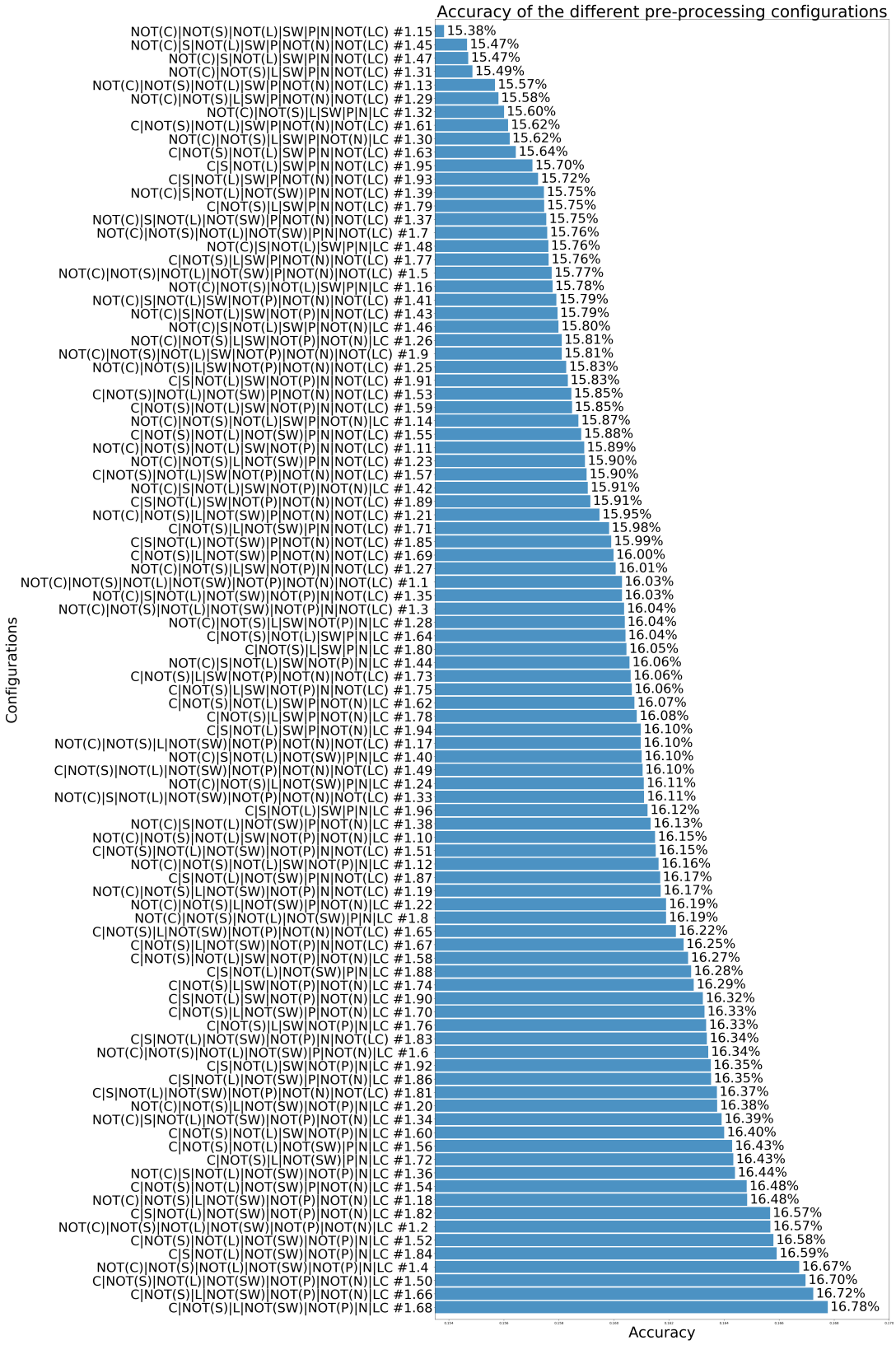


Figure B.5: Accuracy of the different pre-processing configurations on the bug reports of Mozilla Firefox (the mapping of acronyms to pre-processing techniques defined in the Table 4.1 has been used)

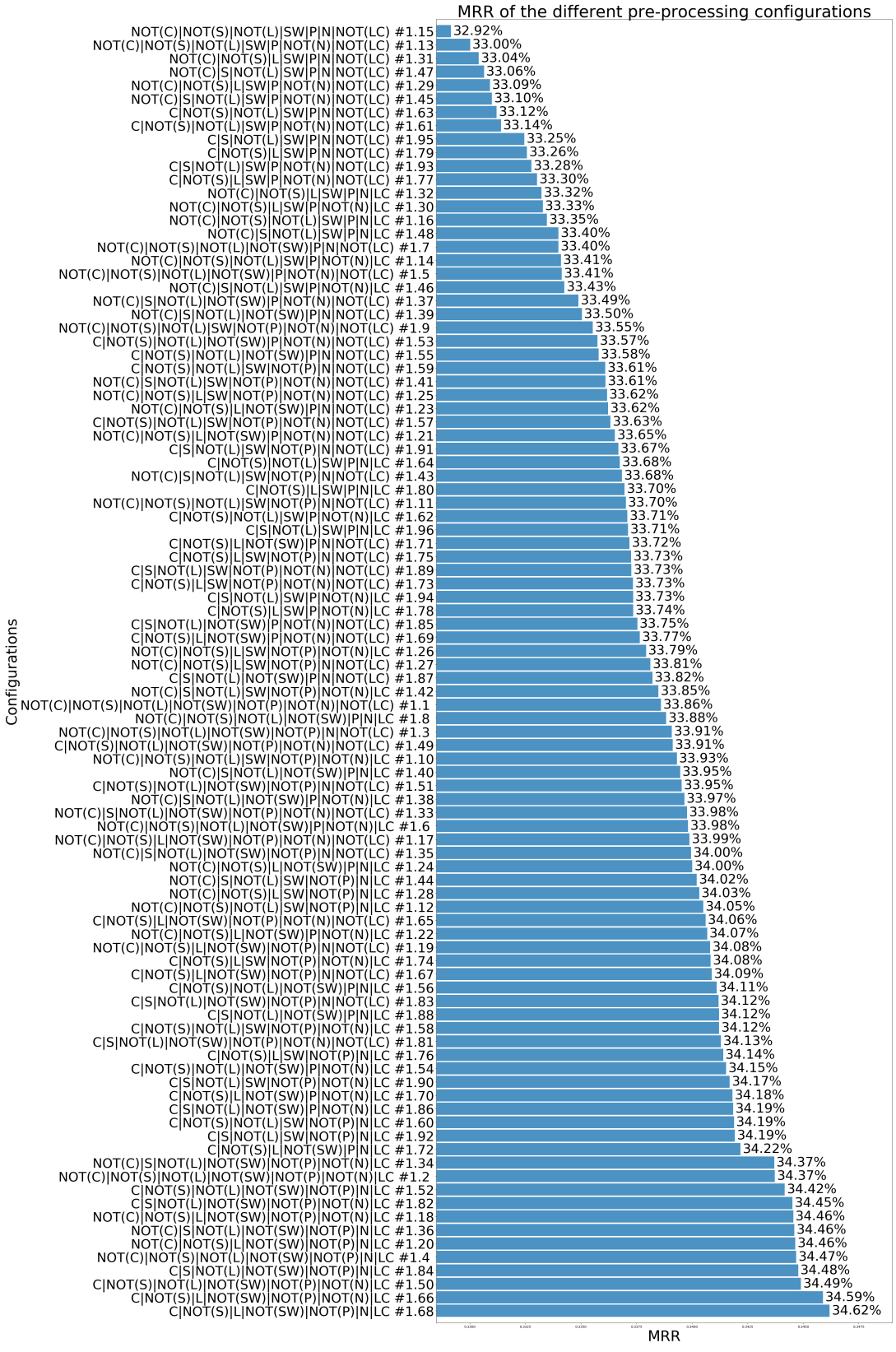


Figure B.6: MRR of the different pre-processing configurations on the bug reports of Mozilla Firefox (the mapping of acronyms to pre-processing techniques defined in the Table 4.1 has been used)

B.2 Experiment 2

The detailed results related to the second main experiment of the thesis (Section 3.4.2) are presented in this section.

For readability purposes, the results are presented using the mapping of acronyms to feature extraction techniques defined in the Table 4.2. If a feature extraction technique is used, its acronym is directly written. Each time the LSI or the NMF algorithm is used, the acronym is concatenated with an hyphen - and the new number of features. The string related to each configuration is then followed by a hash character # and a unique identifier.

B.2.1 Telecommunications company

The horizontal bar chart in the Figure B.7 shows the accuracy of the different feature extraction configurations (without combination of features) on the project of the telecommunications company.

As can be seen in the Figure B.7, the worst five configurations in terms of accuracy are using NMF or LSI to extract 10 new features. The best three configurations are not based on NMF or LSI. When using NMF, the best configurations are obtained with a boolean representation of the bug reports. When using LSI, the best configurations are obtained with tf-idf weights. When NMF or LSI is used, and, more features are extracted, the accuracy increases.

The horizontal bar chart in the Figure B.8 shows the MRR value of the different feature extraction configurations (without combination of features) on the project of the telecommunications company.

As with the accuracy, the best three configurations, in terms of MRR, are not based on NMF or LSI (Figure B.8). The worst five configurations are using NMF or LSI to extract 10 features. When using LSI, the best configurations are obtained with tf-idf weights. When using NMF, the best configurations are obtained with a boolean representation of the bug reports. When NMF or LSI is used, and, more features are extracted, the MRR value increases.

The horizontal bar chart in the Figure B.9 indicates the accuracy of the different feature extraction configurations (with combination of features) on the project of the telecommunications company.

The horizontal bar chart in the Figure B.10 shows the MRR value of the different feature extraction configurations (with combination of features) on the bug reports of the telecommunications company.

In both charts (Figures B.9 and B.10), it can be seen that the best two representations are combining tf-idf weights with a boolean representation of the bug reports followed by the use of the NMF or the LSI algorithm. In the best six configurations, the tf-idf weights are used and combined with another feature extraction technique. The worst three representations are combining feature extraction techniques relying on NMF.

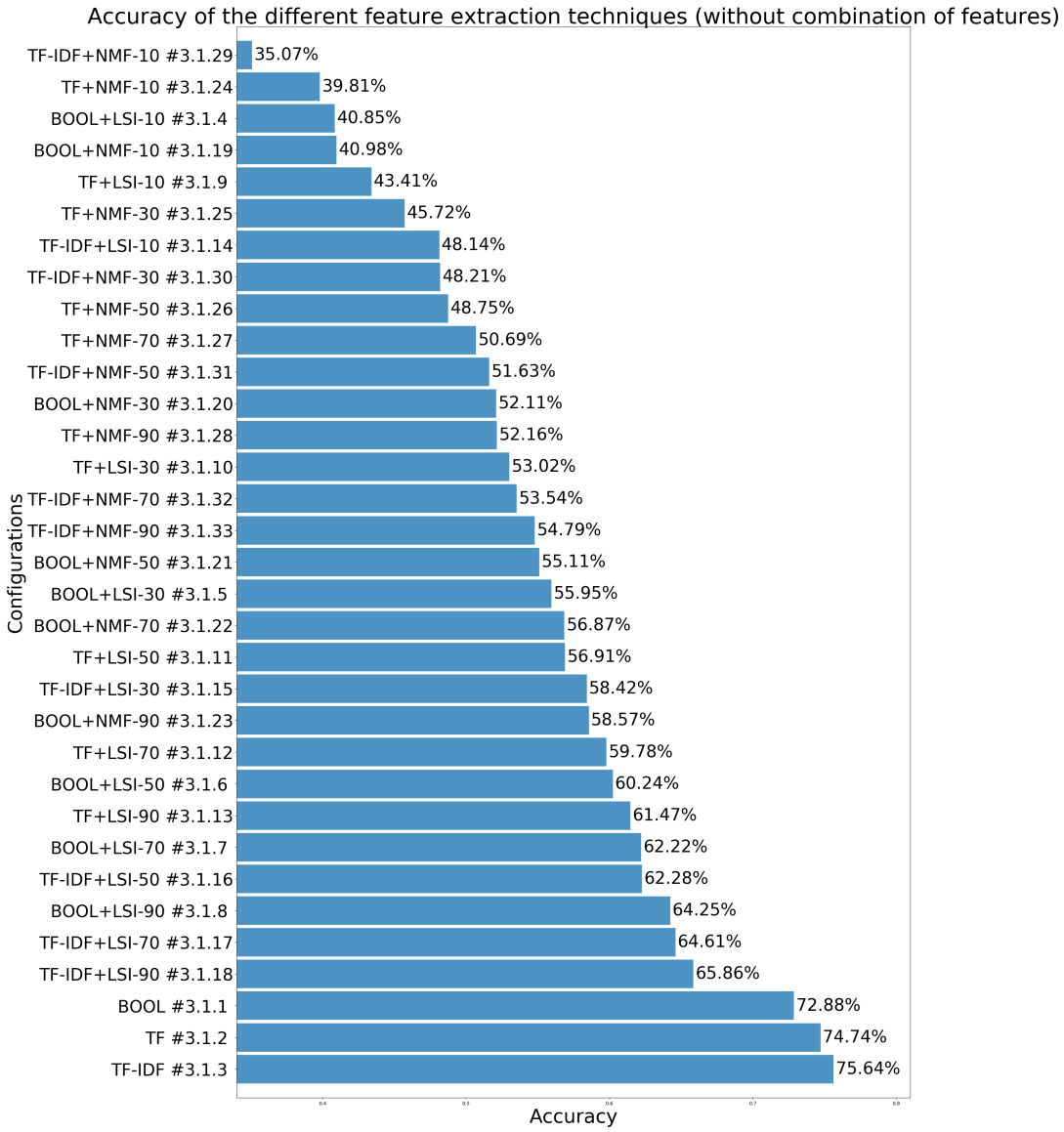


Figure B.7: Accuracy of the different feature extraction techniques (without combination of features) on the bug reports of the telecommunications company (the mapping of acronyms to feature extraction techniques defined in the Table 4.2 has been used)

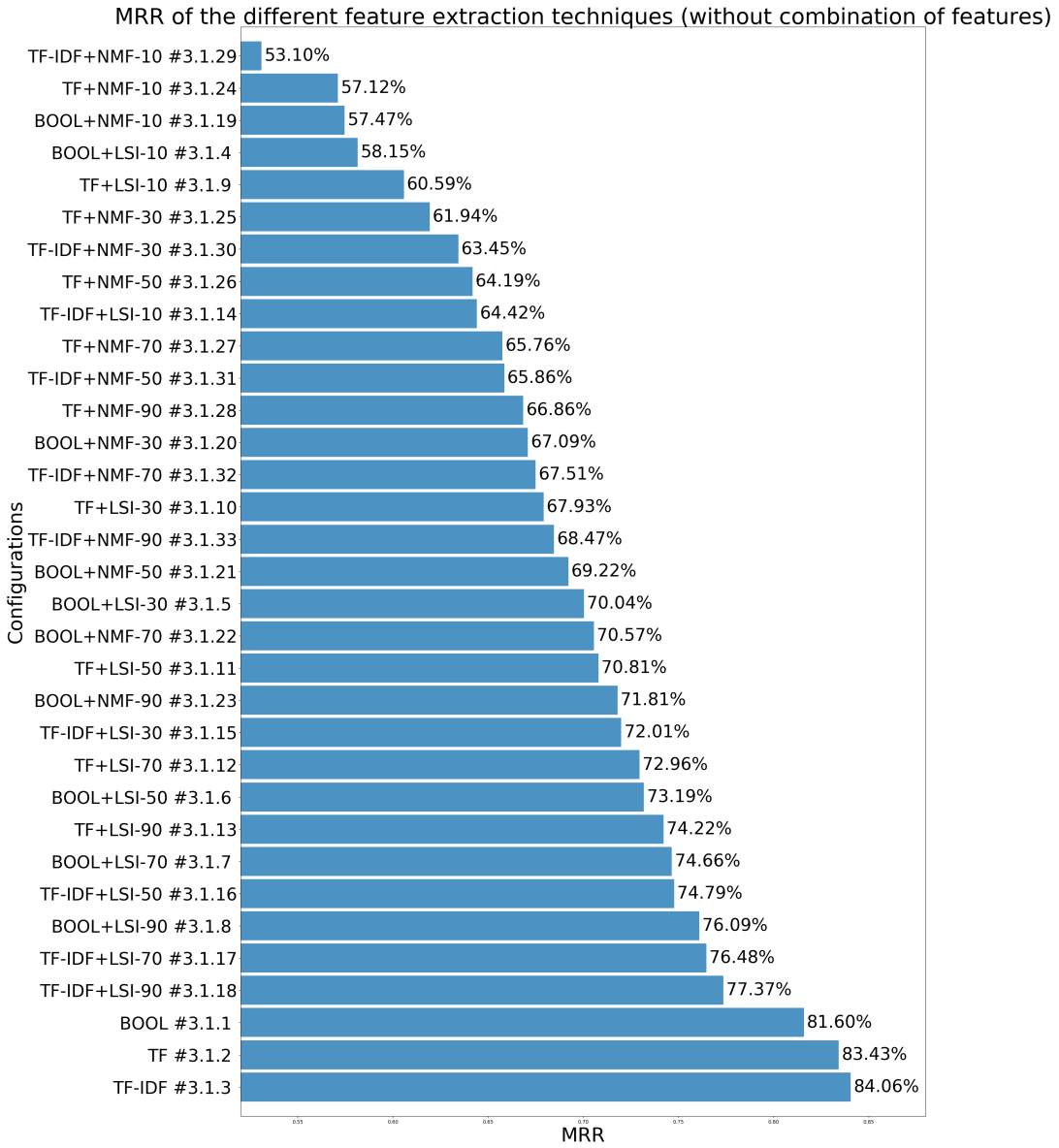


Figure B.8: MRR of the different feature extraction techniques (without combination of features) on the bug reports of the telecommunications company (the mapping of acronyms to feature extraction techniques defined in the Table 4.2 has been used)

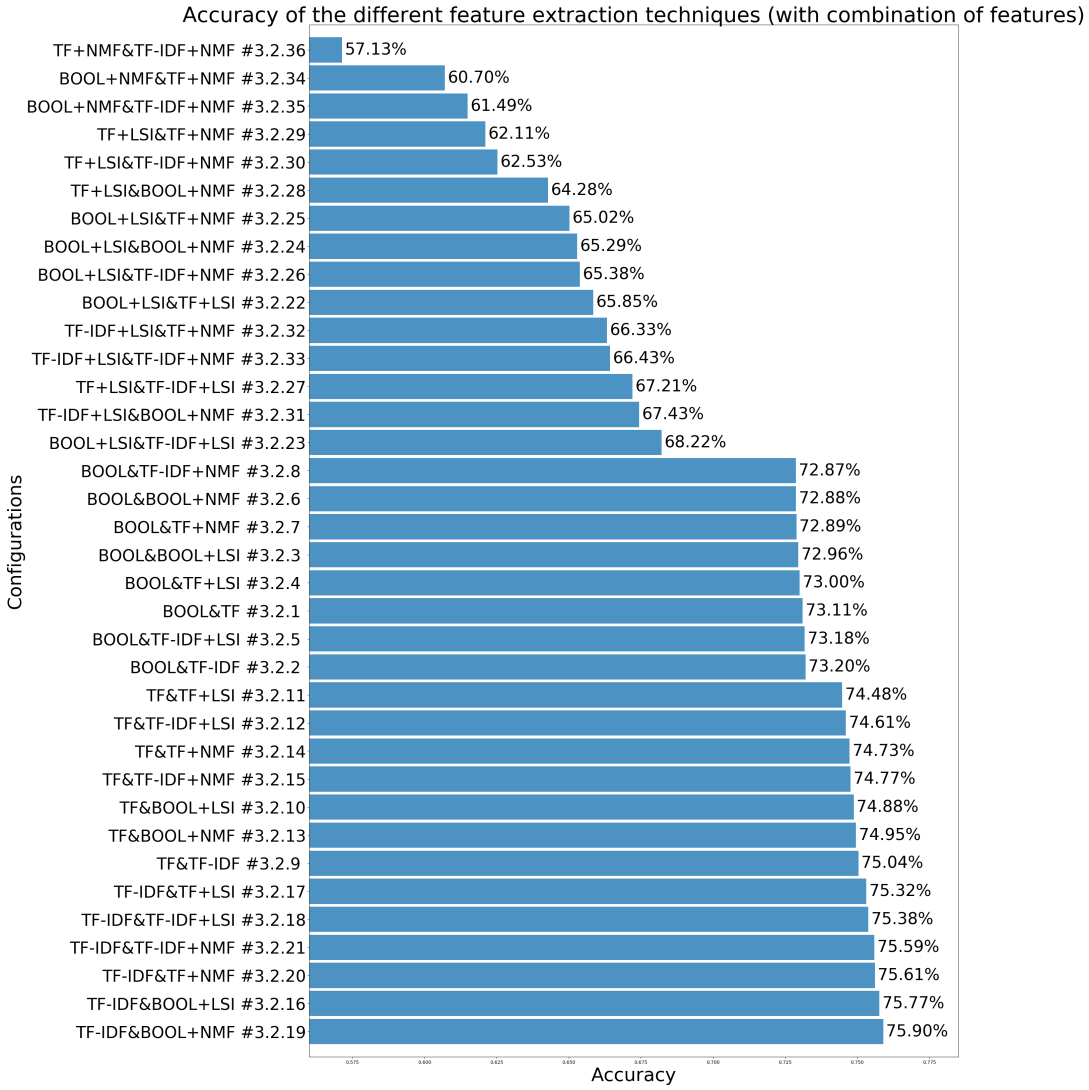


Figure B.9: Accuracy of the different feature extraction techniques (with combination of features) on the bug reports of the telecommunications company (the mapping of acronyms to feature extraction techniques defined in the Table 4.2 has been used)

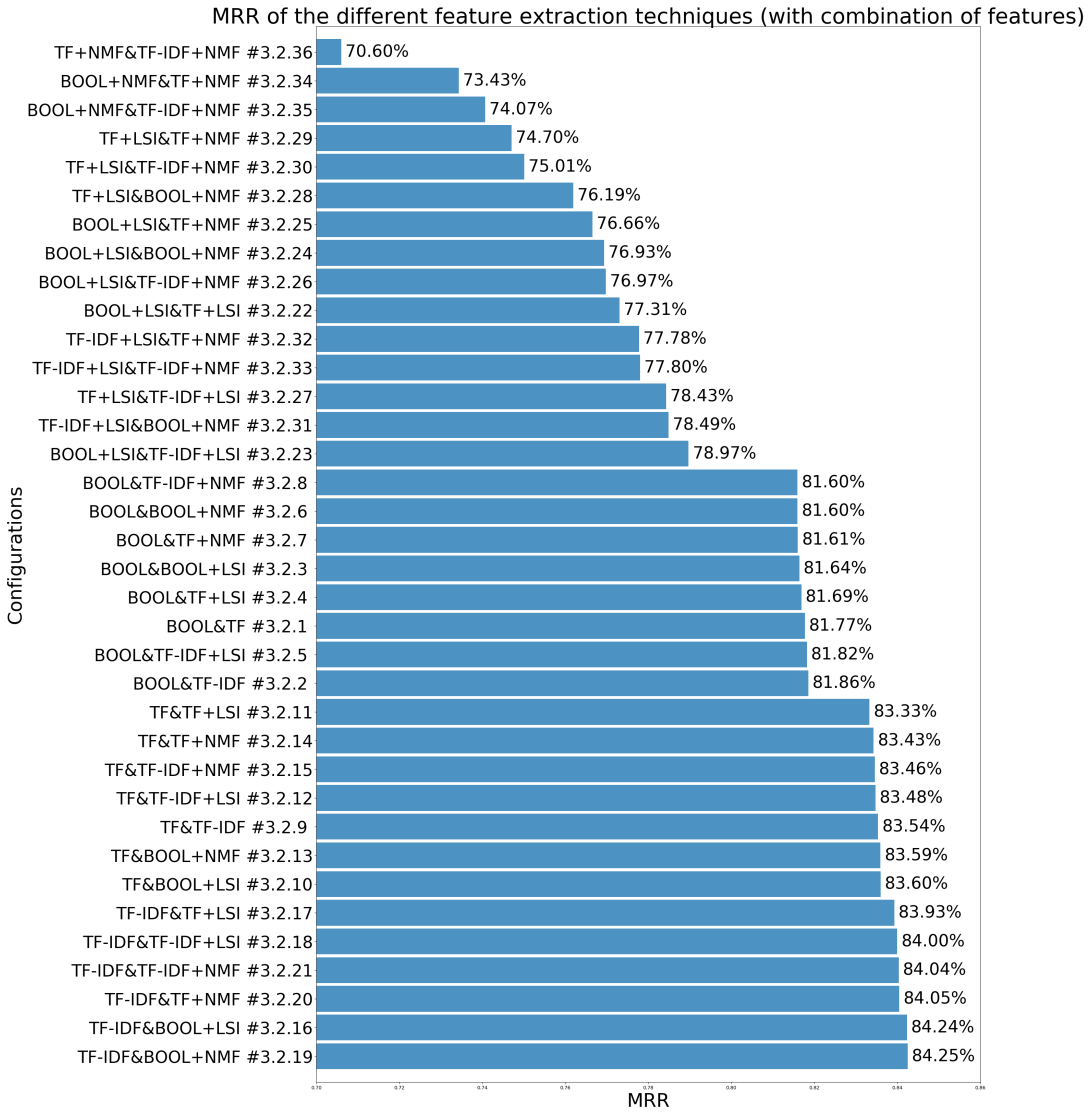


Figure B.10: MRR of the different feature extraction techniques (with combination of features) on the bug reports of the telecommunications company (the mapping of acronyms to feature extraction techniques defined in the Table 4.2 has been used)

B.2.2 Eclipse JDT

The horizontal bar chart in the Figure B.11 illustrates the accuracy of the different feature extraction configurations (without combination of features) on the bug reports of Eclipse JDT.

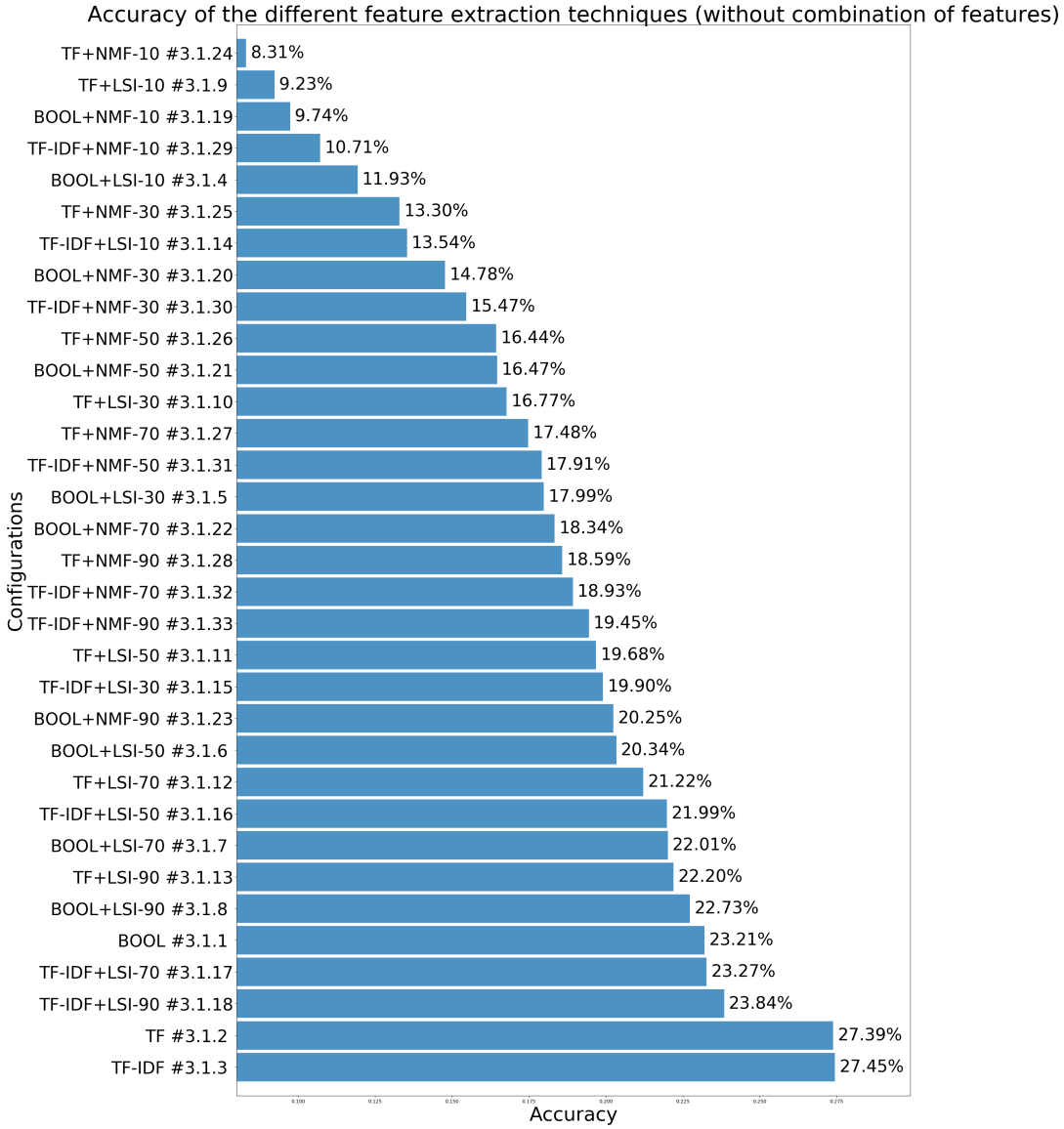


Figure B.11: Accuracy of the different feature extraction techniques (without combination of features) on the bug reports of Eclipse JDT (the mapping of acronyms to feature extraction techniques defined in the Table 4.2 has been used)

The horizontal bar chart in the Figure B.12 depicts the MRR value of the different feature extraction techniques (without combination of features) on Eclipse JDT.

In the Figures B.11 and B.12, it can be seen that the worst five configurations are using NMF or LSI to extract 10 new features. The best two configurations are not based on NMF or LSI, and, are not using a boolean representation of the bug reports. As with the data set of the telecommunications company, when using LSI, the best configurations are obtained with tf-idf weights. When using NMF, the best configurations are obtained with a boolean representation of the bug reports. When NMF or LSI is used, and, more features are extracted, the accuracy and the MRR value increase.

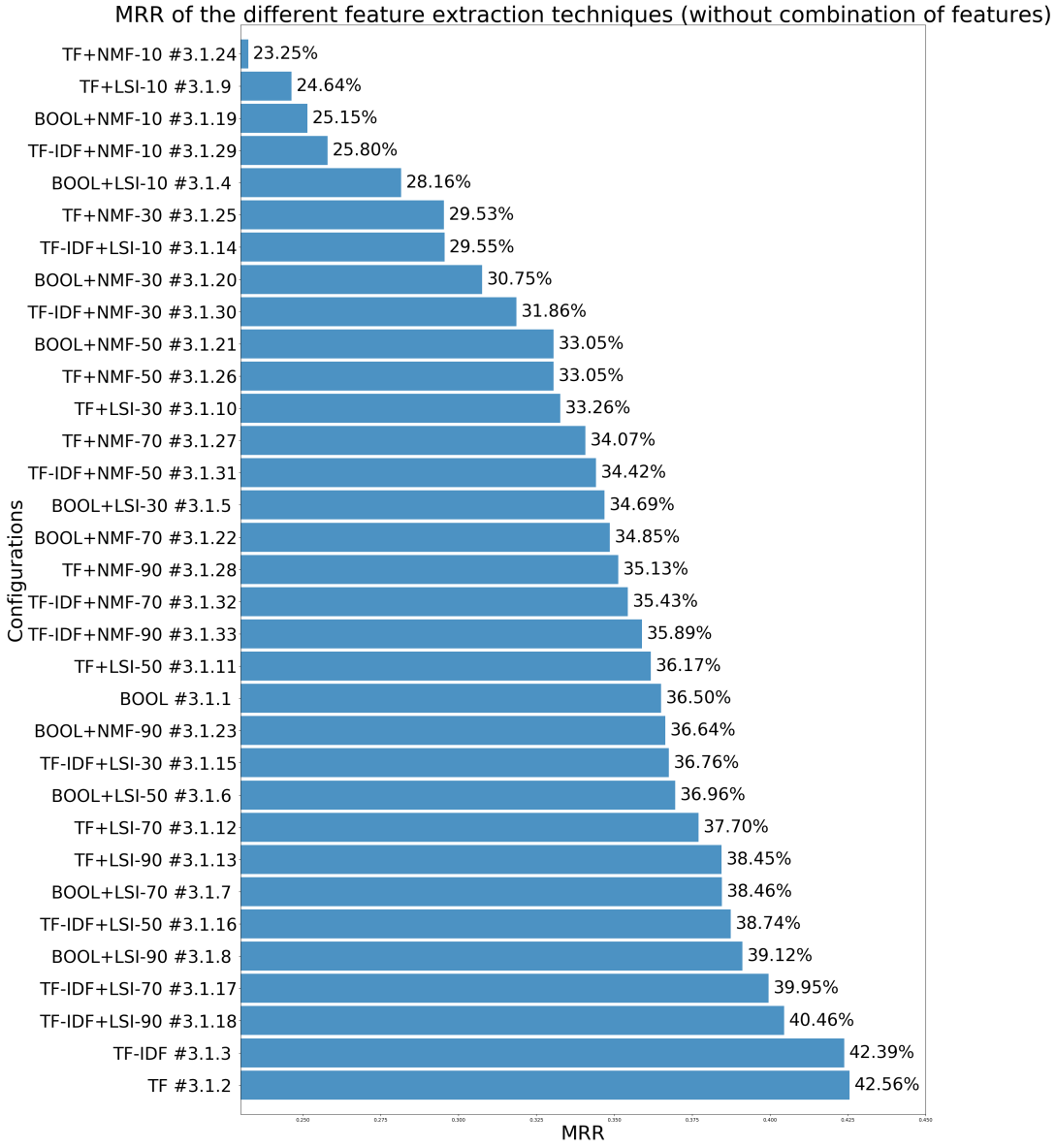


Figure B.12: MRR of the different feature extraction techniques (without combination of features) on the bug reports of Eclipse JDT (the mapping of acronyms to feature extraction techniques defined in the Table 4.2 has been used)

The horizontal bar chart in the Figure B.13 shows the accuracy of the different feature extraction configurations (with combination of features) on the bug reports of Eclipse JDT.

The horizontal bar chart in the Figure B.14 indicates the MRR value of the different feature extraction configurations (with combination of features) on Eclipse JDT.

In the four best configurations of the Figure B.13, the tf-idf weights are used and combined with another feature extraction technique. The tf-idf weights are used and combined with another feature extraction technique in two of the three best configurations of the Figure B.14. In both figures, at least the three worst representations are using NMF.

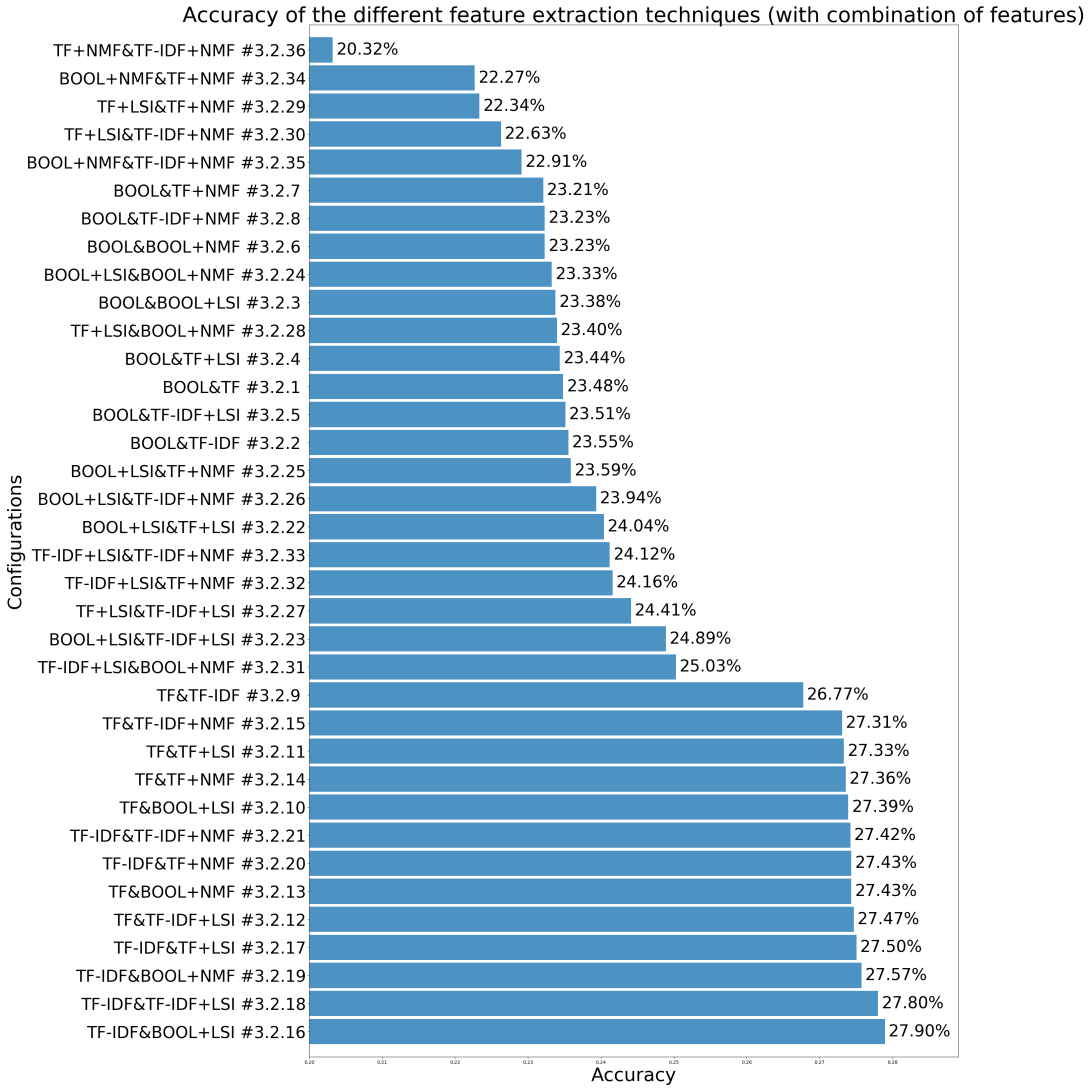


Figure B.13: Accuracy of the different feature extraction techniques (with combination of features) on the bug reports of Eclipse JDT (the mapping of acronyms to feature extraction techniques defined in the Table 4.2 has been used)

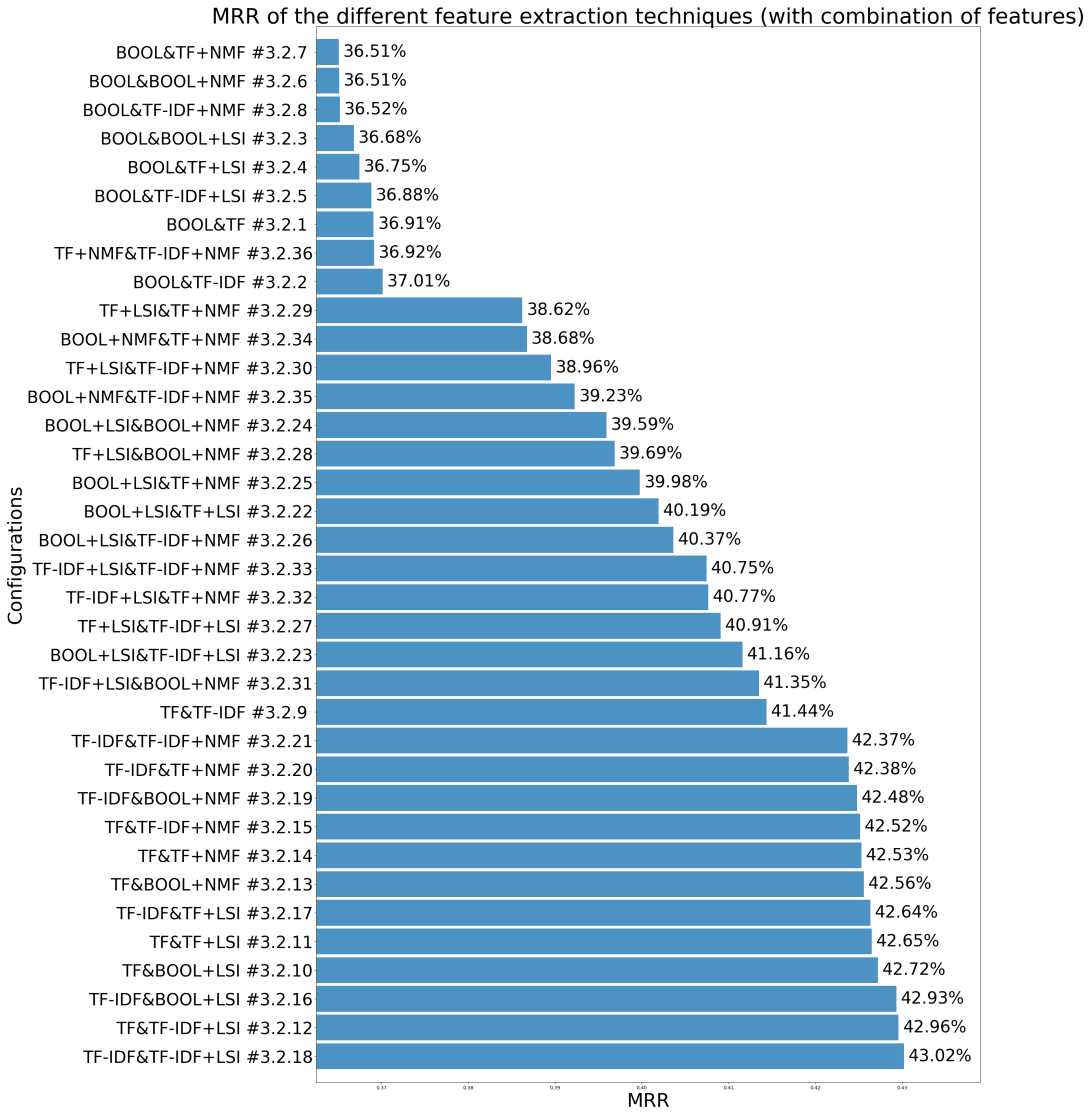


Figure B.14: MRR of the different feature extraction techniques (with combination of features) on the bug reports of Eclipse JDT (the mapping of acronyms to feature extraction techniques defined in the Table 4.2 has been used)

B.2.3 Mozilla Firefox

The horizontal bar chart in the Figure B.15 shows the accuracy of the different feature extraction techniques (without combination of features) on Mozilla Firefox.

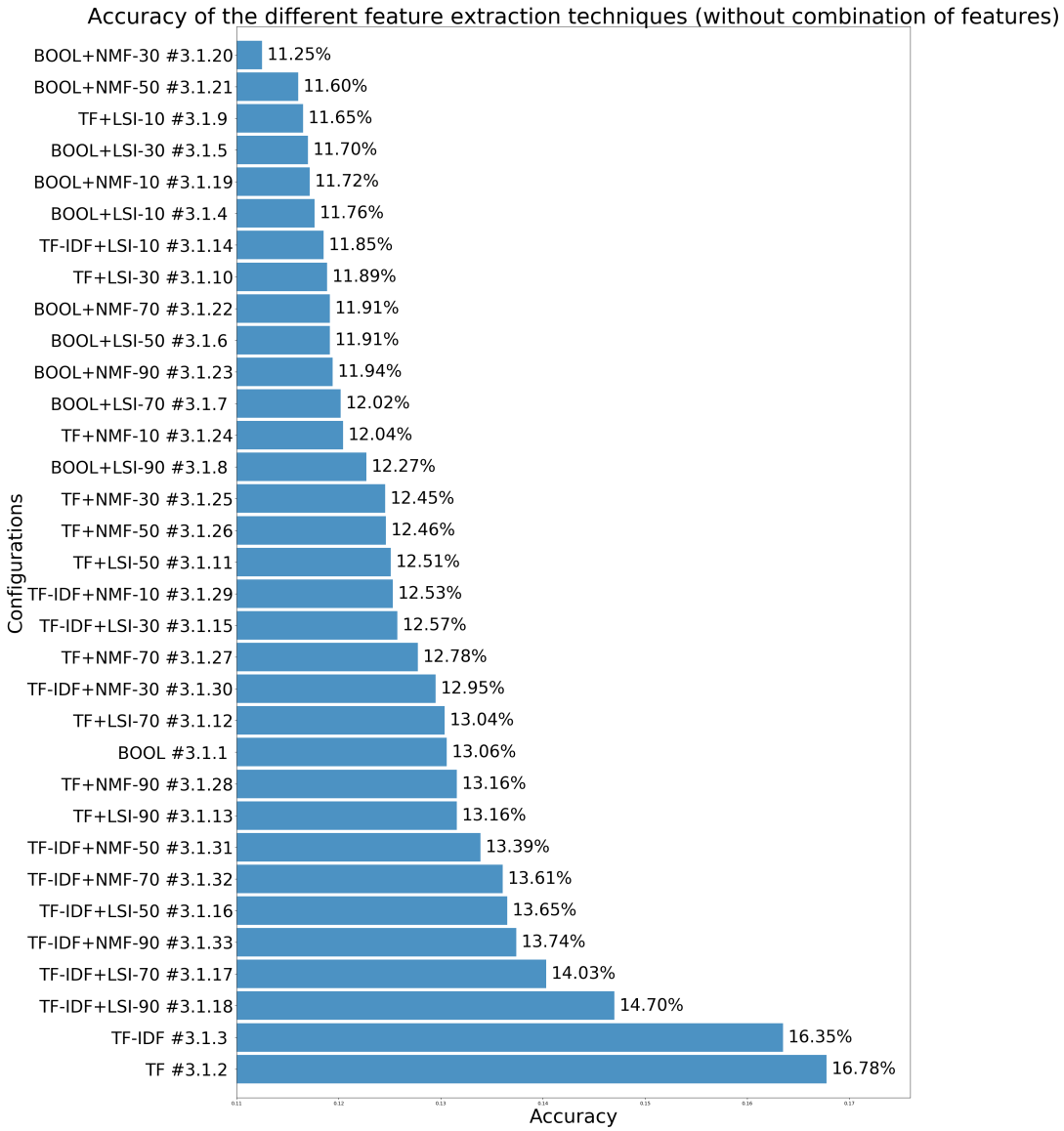


Figure B.15: Accuracy of the different feature extraction techniques (without combination of features) on the bug reports of Mozilla Firefox (the mapping of acronyms to feature extraction techniques defined in the Table 4.2 has been used)

The horizontal bar chart in the Figure B.16 illustrates the MRR value of the different feature extraction configurations (without combination of features) on the bug reports of Mozilla Firefox.

In the Figures B.15 and B.16, as with the bug reports of Eclipse JDT, the best two configurations are not relying on NMF or LSI, and, are not using a boolean representation of the bug reports. When using LSI, the best configurations are based on the tf-idf weights. Contrary to the telecommunications company and Eclipse JDT, the best configurations, when using NMF, are also based on the tf-idf weights. Except with a boolean representation of the bug reports, when NMF or LSI is used, and, more features are extracted, the accuracy and the MRR value

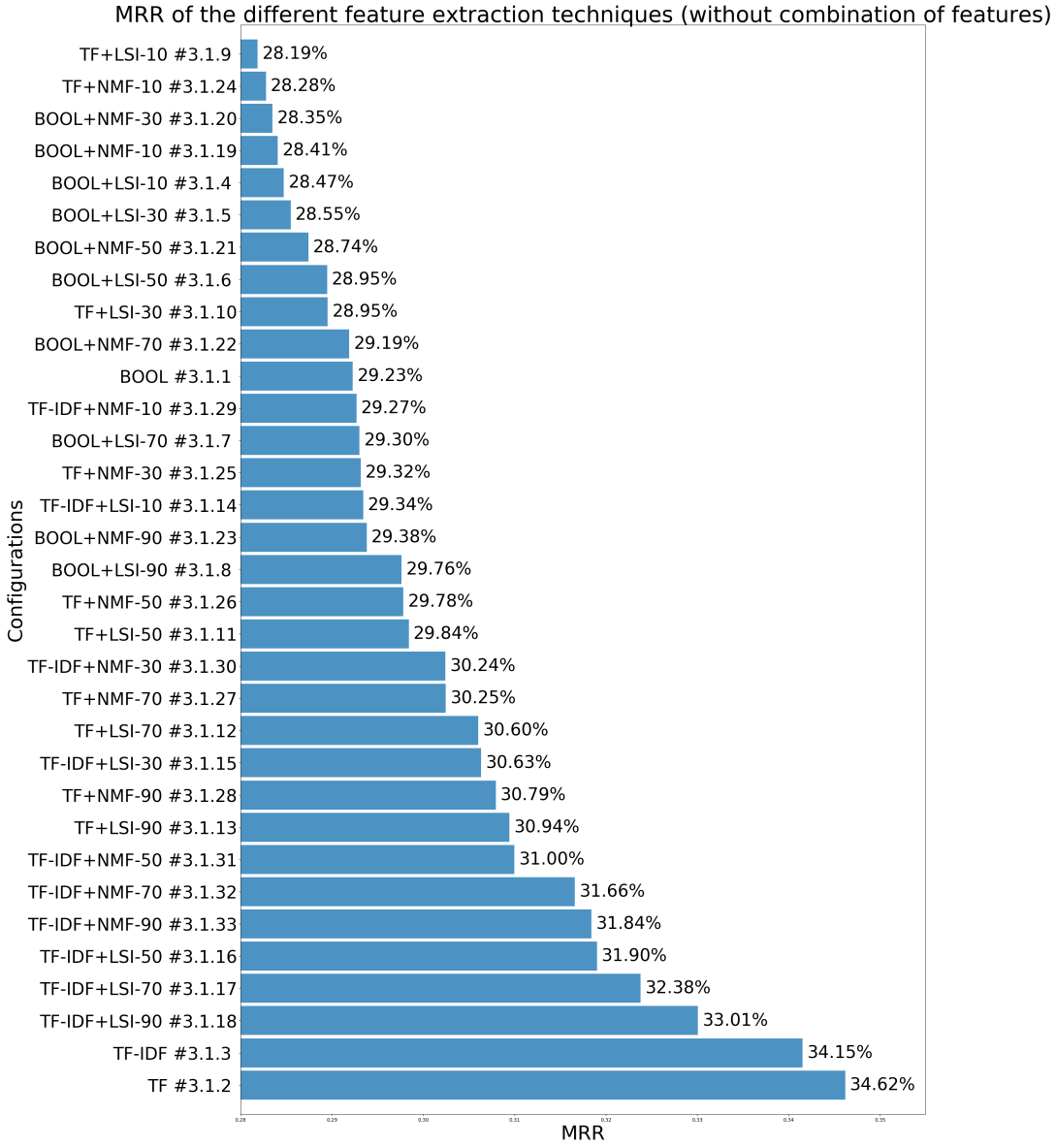


Figure B.16: MRR of the different feature extraction techniques (without combination of features) on the bug reports of Mozilla Firefox (the mapping of acronyms to feature extraction techniques defined in the Table 4.2 has been used)

increase. Contrary to the two other software development projects, the worst five configurations, in terms of accuracy and MRR value, are not only using NMF or LSI to extract 10 new features.

The horizontal bar chart in the Figure B.17 indicates the accuracy of the different feature extraction configurations (with combination of features) on Mozilla Firefox.

The horizontal bar chart in the Figure B.18 depicts the MRR value of the different feature extraction techniques (with combination of features) on the bug reports of Mozilla Firefox.

In both charts (Figures B.17 and B.18), in five of the six best configurations, the tf weights are used and combined with another feature extraction technique. At least three of the four worst configurations are, furthermore, using NMF in both figures.

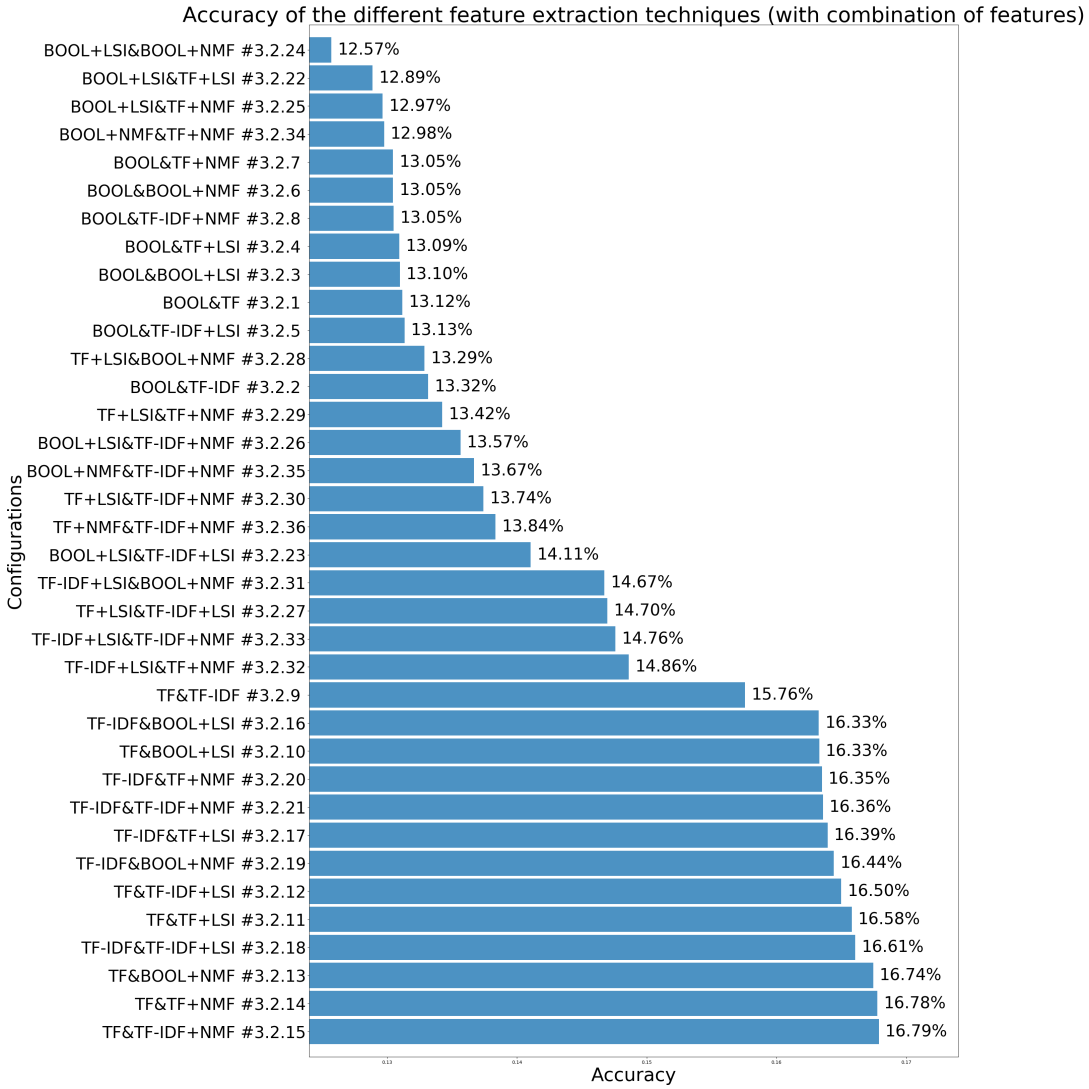


Figure B.17: Accuracy of the different feature extraction techniques (with combination of features) on the bug reports of Mozilla Firefox (the mapping of acronyms to feature extraction techniques defined in the Table 4.2 has been used)

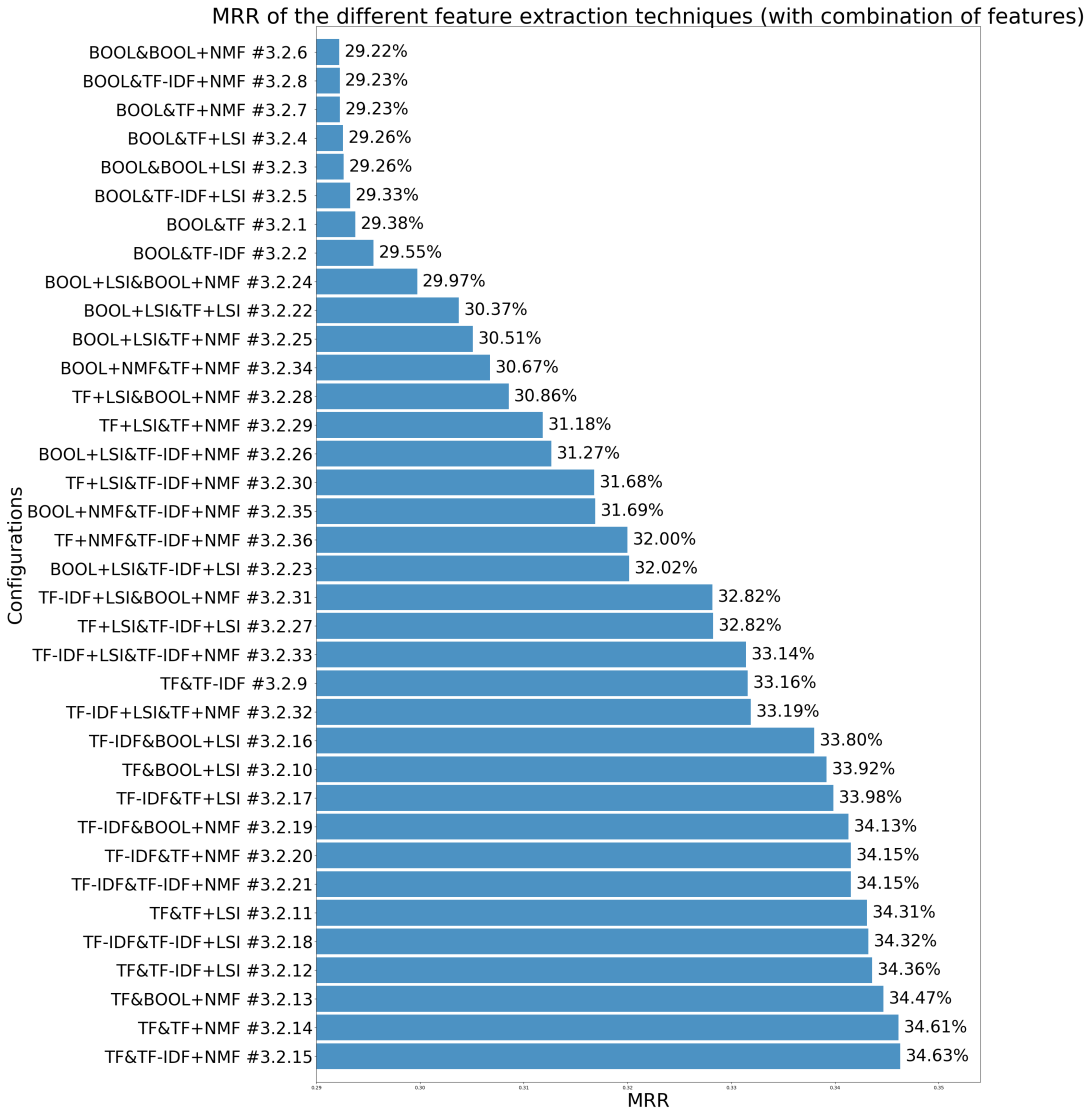


Figure B.18: MRR of the different feature extraction techniques (with combination of features) on the bug reports of Mozilla Firefox (the mapping of acronyms to feature extraction techniques defined in the Table 4.2 has been used)

B.3 Experiment 3

The detailed results related to the third main experiment of the thesis (Section 3.4.3) are presented in this section.

For readability purposes, the results are presented using the mapping of acronyms to feature selection techniques defined in the Table 4.3. If a feature selection technique is applied, its acronym is used. This string is then concatenated with an hyphen - and the percentage of remaining features. Next, the string related to each configuration is followed by a hash character # and a unique identifier.

B.3.1 Telecommunications company

The horizontal bar chart in the Figure B.19 illustrates the accuracy of the different feature selection configurations on the project of the telecommunications company.

The horizontal bar chart in the Figure B.20 depicts the MRR value of the different feature selection configurations on the bug reports of the telecommunications company.

Except for recursive feature elimination, for each feature selection technique, when more features are selected, the accuracy and the MRR value increase (Figures B.19 and B.20).

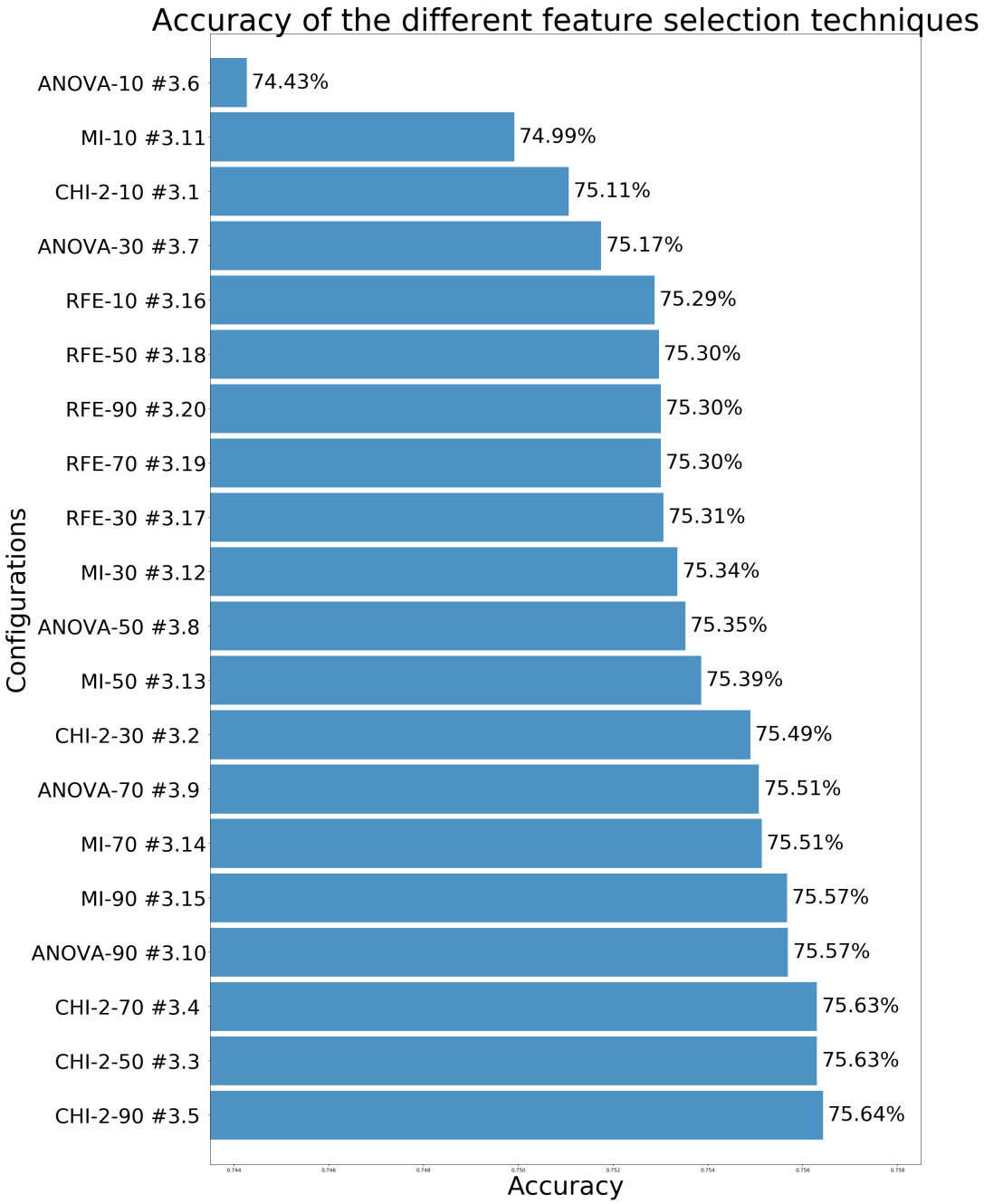


Figure B.19: Accuracy of the different feature selection techniques on the bug reports of the telecommunications company (the mapping of acronyms to feature selection techniques defined in the Table 4.3 has been used)

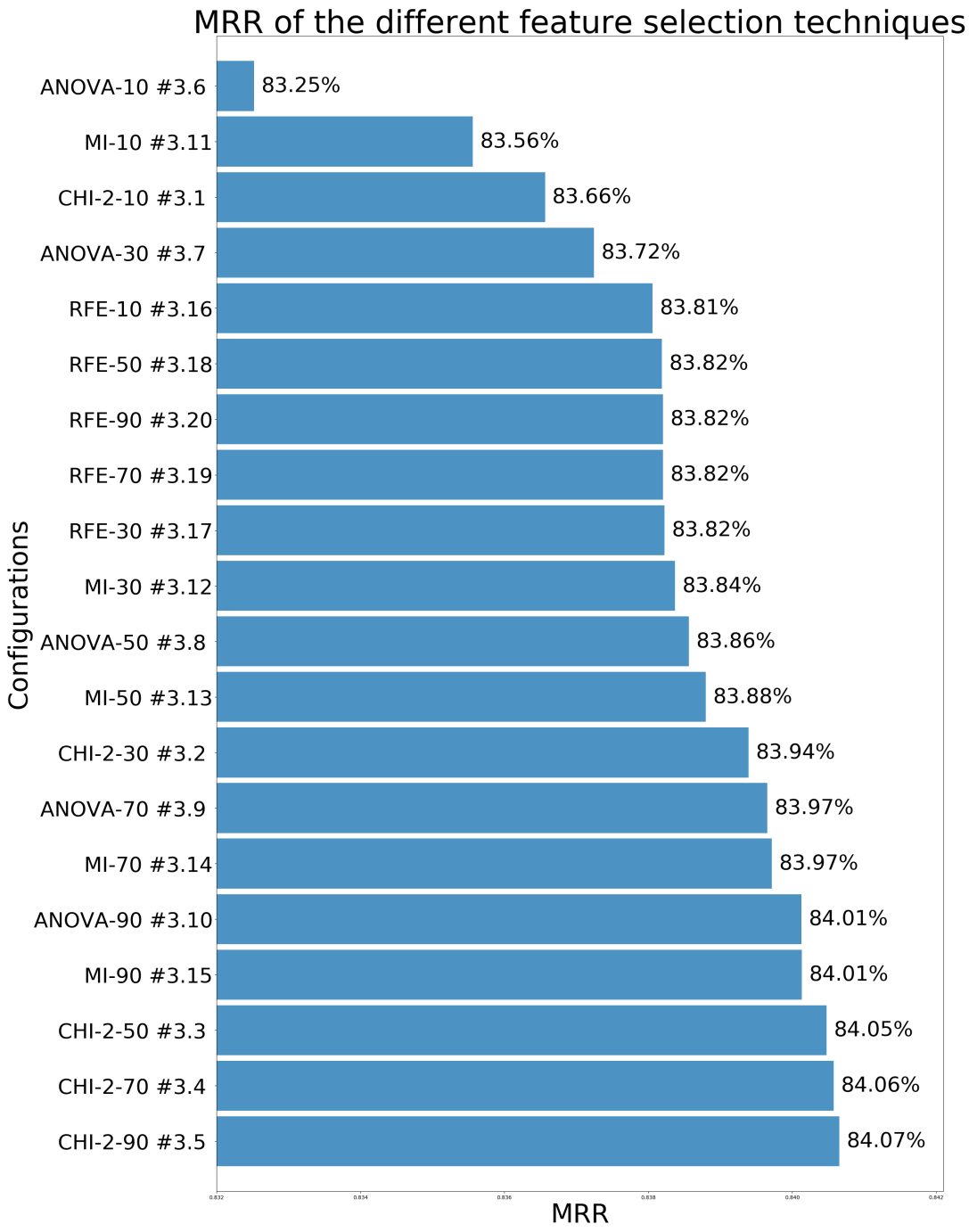


Figure B.20: MRR of the different feature selection techniques on the bug reports of the telecommunications company (the mapping of acronyms to feature selection techniques defined in the Table 4.3 has been used)

B.3.2 Eclipse JDT

The horizontal bar chart in the Figure B.21 shows the accuracy of the different feature selection configurations on Eclipse JDT.

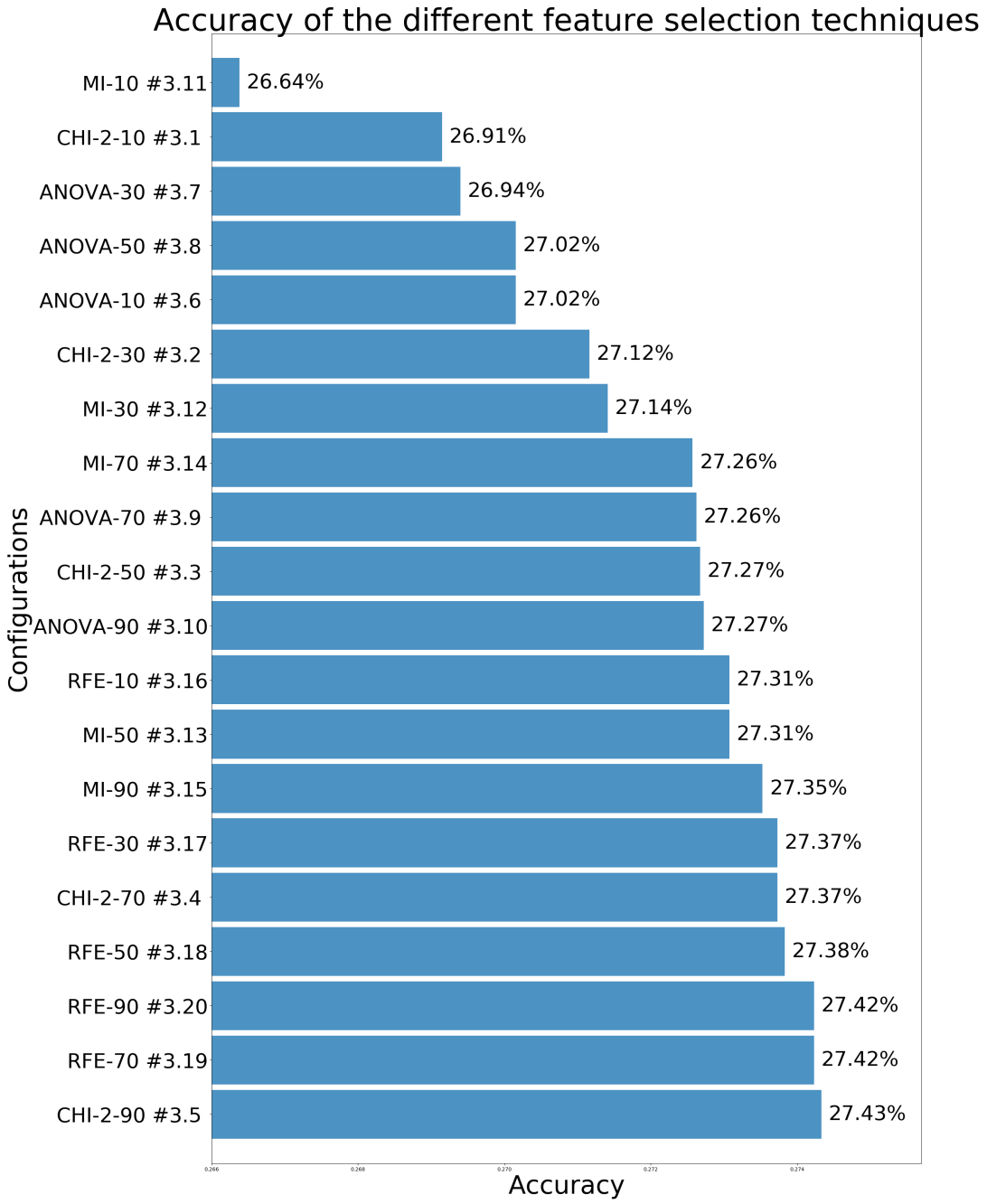


Figure B.21: Accuracy of the different feature selection techniques on the bug reports of Eclipse JDT (the mapping of acronyms to feature selection techniques defined in the Table 4.3 has been used)

The horizontal bar chart in the Figure B.22 indicates the MRR value of the different feature selection techniques on the bug reports of Eclipse JDT.

In the Figures B.21 and B.22, the orders of the different configurations are not similar.

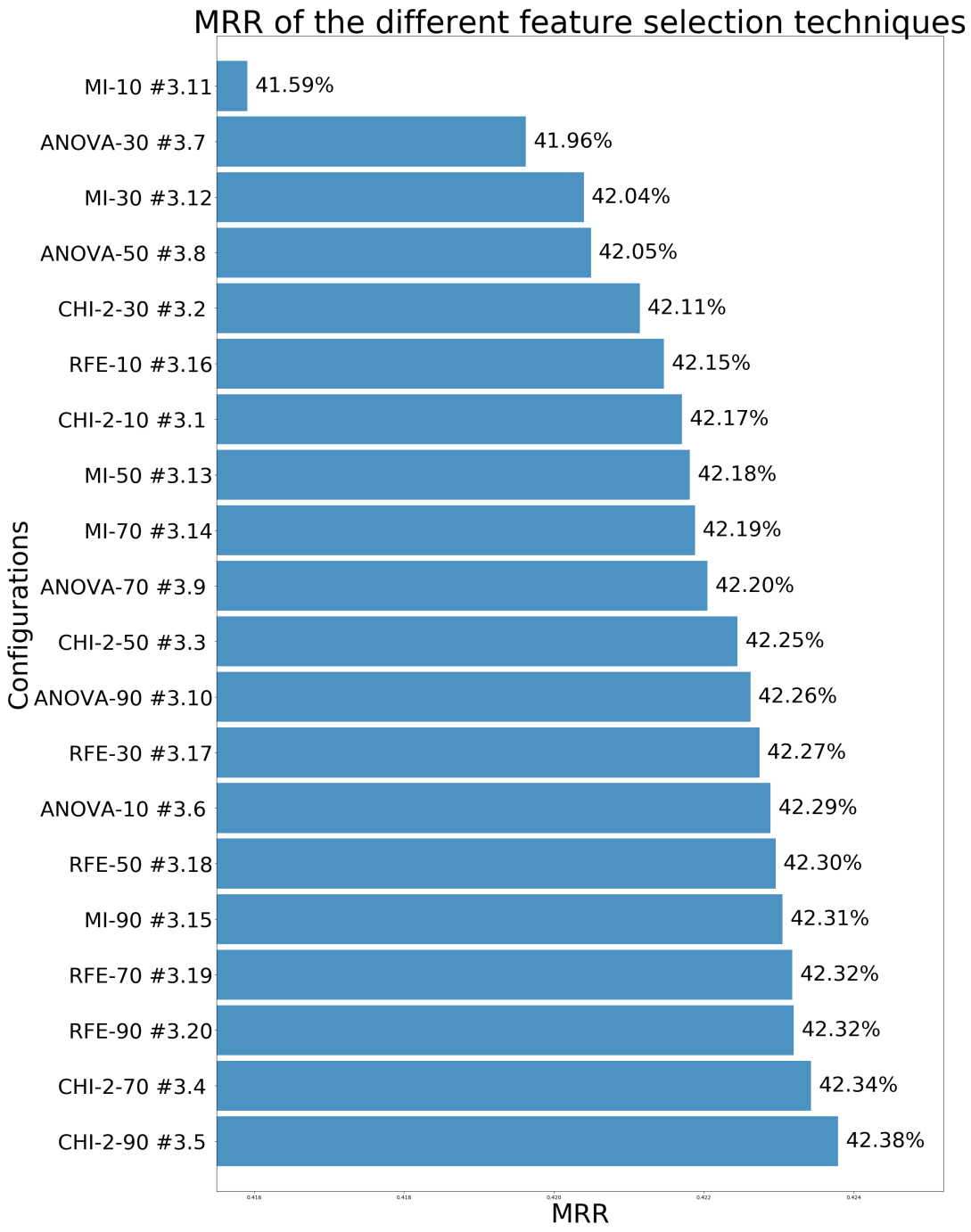


Figure B.22: MRR of the different feature selection techniques on the bug reports of Eclipse JDT (the mapping of acronyms to feature selection techniques defined in the Table 4.3 has been used)

B.3.3 Mozilla Firefox

The horizontal bar chart in the Figure B.23 depicts the accuracy of the different feature selection configurations on the bug reports of Mozilla Firefox.

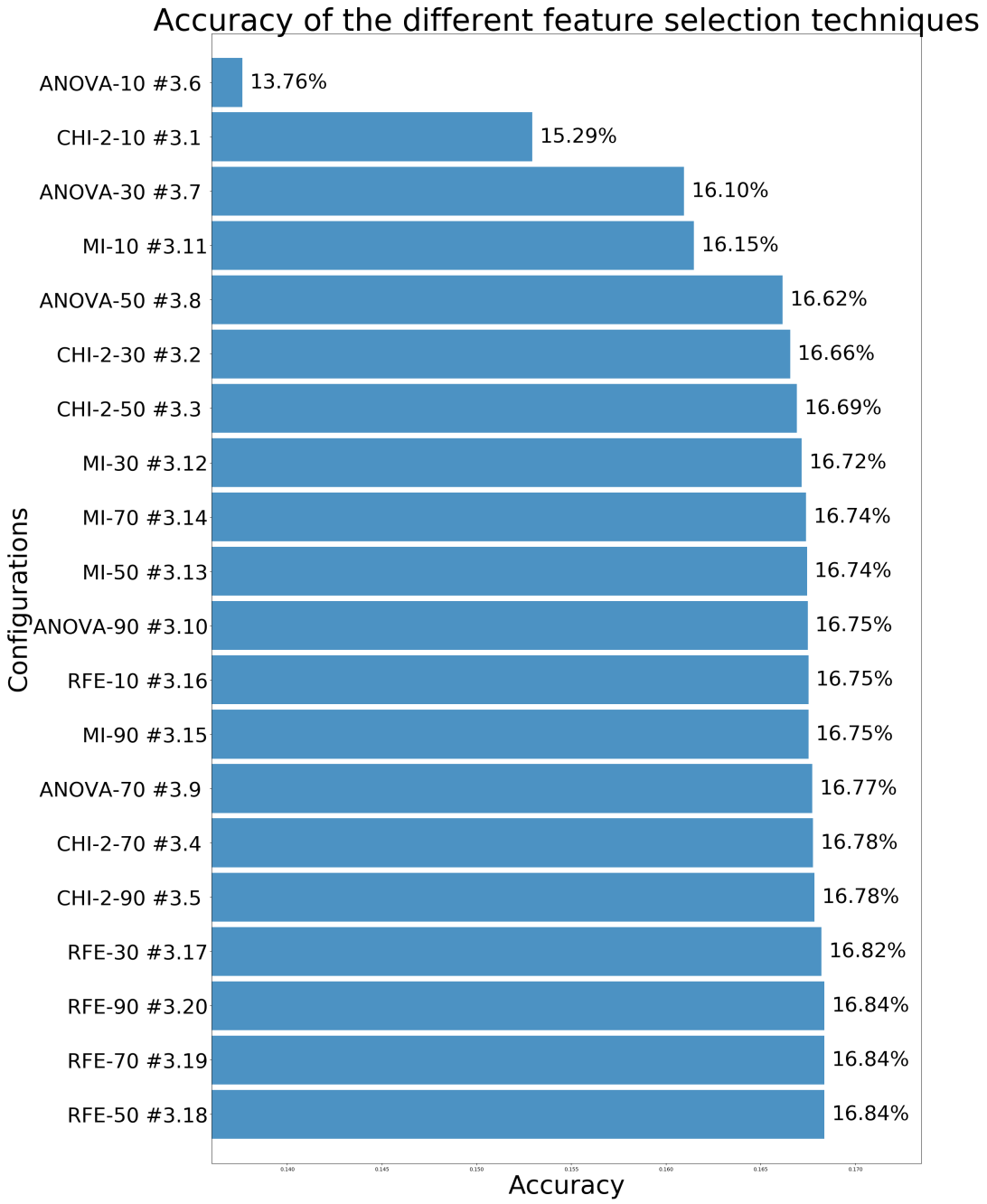


Figure B.23: Accuracy of the different feature selection techniques on the bug reports of Mozilla Firefox (the mapping of acronyms to feature selection techniques defined in the Table 4.3 has been used)

The horizontal bar chart in the Figure B.24 indicates the MRR value of the different feature selection configurations on Mozilla Firefox.

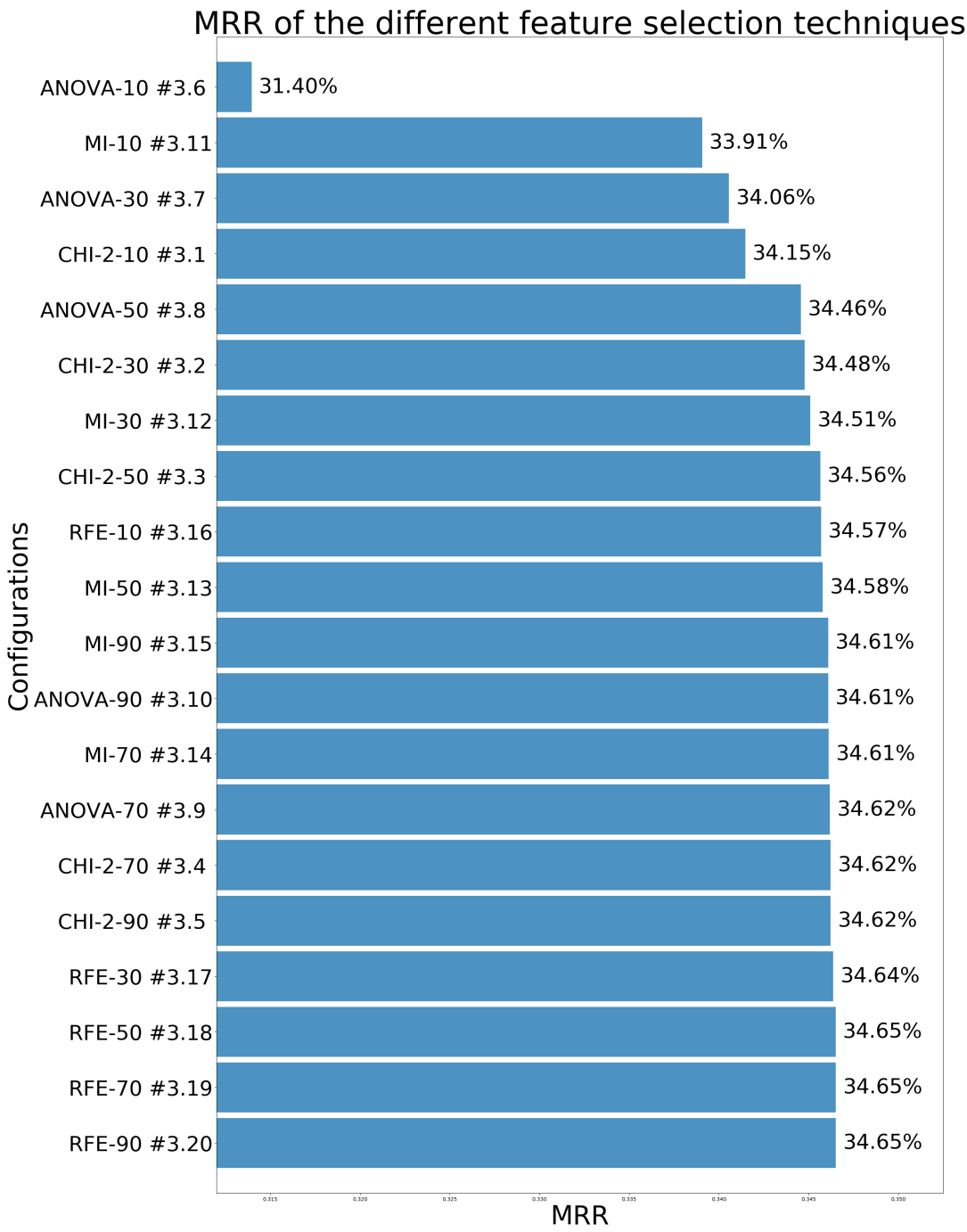


Figure B.24: MRR of the different feature selection techniques on the bug reports of Mozilla Firefox (the mapping of acronyms to feature selection techniques defined in the Table 4.3 has been used)

The orders of the different feature selection configurations are not similar in the Figures B.23 and B.24. Only for χ^2, when more features are selected, the accuracy and the MRR value increase.

B.4 Experiment 4

In this section, the detailed results related to the forth main experiment of the thesis (Section 3.4.4) are presented.

For readability purposes, the results will be described using a mapping of acronyms to classifiers (Table B.1). If a classifier has been trained and has made some predictions, its acronym is used. This string is then concatenated with a pipe `|` and another string containing the value(s) of its hyper parameter(s). The second sub-string has the following form: `hyper_param_1=val_1|...|hyper_param_n=val_n`, where `hyper_param_i` is the name of the *i*-th hyper parameter and `val_i` is its value.

If the string `NC|alpha=0|fit_prior=True` is written, for instance, it means that a naive Bayes classifier with a multinomial distribution has been used with the following values for its hyper parameters: `0` for `alpha` and `True` for `fit_prior`.

Acronym	Classifier
NC	Nearest centroid classifier
MNB	Naive Bayes classifier (multinomial distribution)
LSVC	Linear SVM
PLR	Logistic regression (primal problem)
DLR	Logistic regression (dual problem)
PWP	Perceptron (with penalty)
P	Perceptron (without penalty)
SGCD	Stochastic gradient descent

Table B.1: The mapping of acronyms to classifiers

B.4.1 Telecommunications company

The horizontal bar chart in the Figure B.25 illustrates the accuracy of the best configuration (in terms of accuracy) of each classifier on the project of the telecommunications company.

The horizontal bar chart in the Figure B.26 shows the MRR value of the best configuration (in terms of MRR value) of each classifier on the bug reports of the telecommunications company.

The aforementioned best configurations have been obtained via the use of a grid search strategy. In both charts, the classifiers are ordered in the same way.

Except for the stochastic gradient descent algorithm, for both the accuracy and the MRR value, the best configurations of each classifier are similar (Figures B.25 and B.26). In terms of accuracy and MRR value, the four best classifiers are, in descending order, the linear SVM, the logistic regression (dual problem), the logistic regression (primal problem) and the stochastic gradient descent algorithm. In terms of accuracy, the worst configuration is obtained when using the nearest centroid classifier.

The horizontal bar chart in the Figure B.27 depicts the accuracy of the best configuration (in terms of accuracy) of each classifier on the project of the telecommunications company.

The horizontal bar chart in the Figure B.28 illustrates the MRR value of the best configuration (in terms of MRR value) of each classifier on the bug reports of the telecommunications company.

The above mentioned best configurations have been obtained via the use of a random search strategy.

Except for the perceptron with penalty, the stochastic gradient descent algorithm and the logistic regression (primal problem), the accuracies reached with the best configuration of each classifier, using a random search strategy, are greater than or equal to the ones reached when using a grid search strategy. In terms of accuracy, when using a random search strategy, the classifiers are ordered the same way as the one obtained when using a grid search strategy.

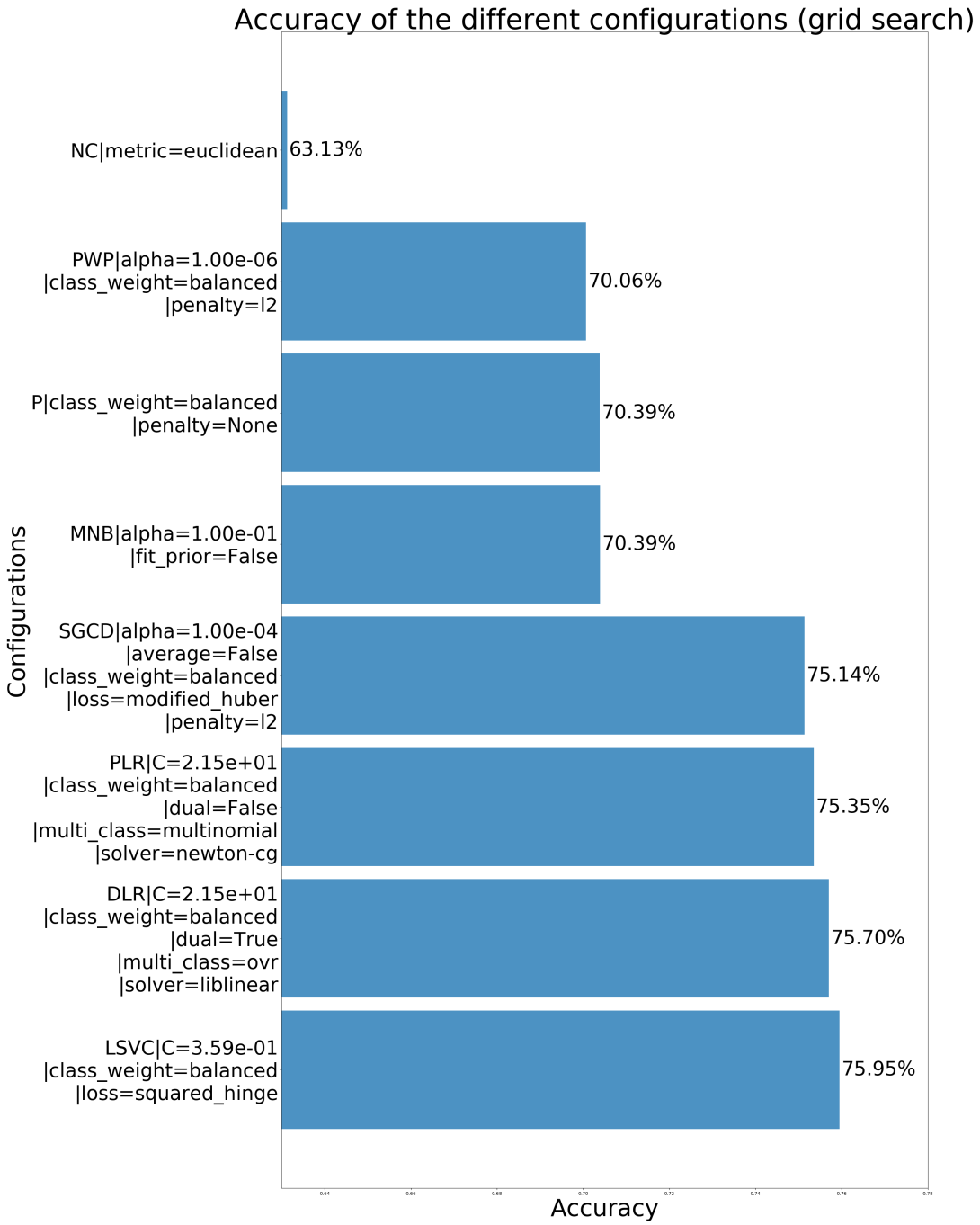


Figure B.25: Accuracy of the best grid search configurations (in terms of accuracy) on the bug reports of the telecommunications company (the mapping of acronyms to classifiers defined in the Table B.1 has been used)

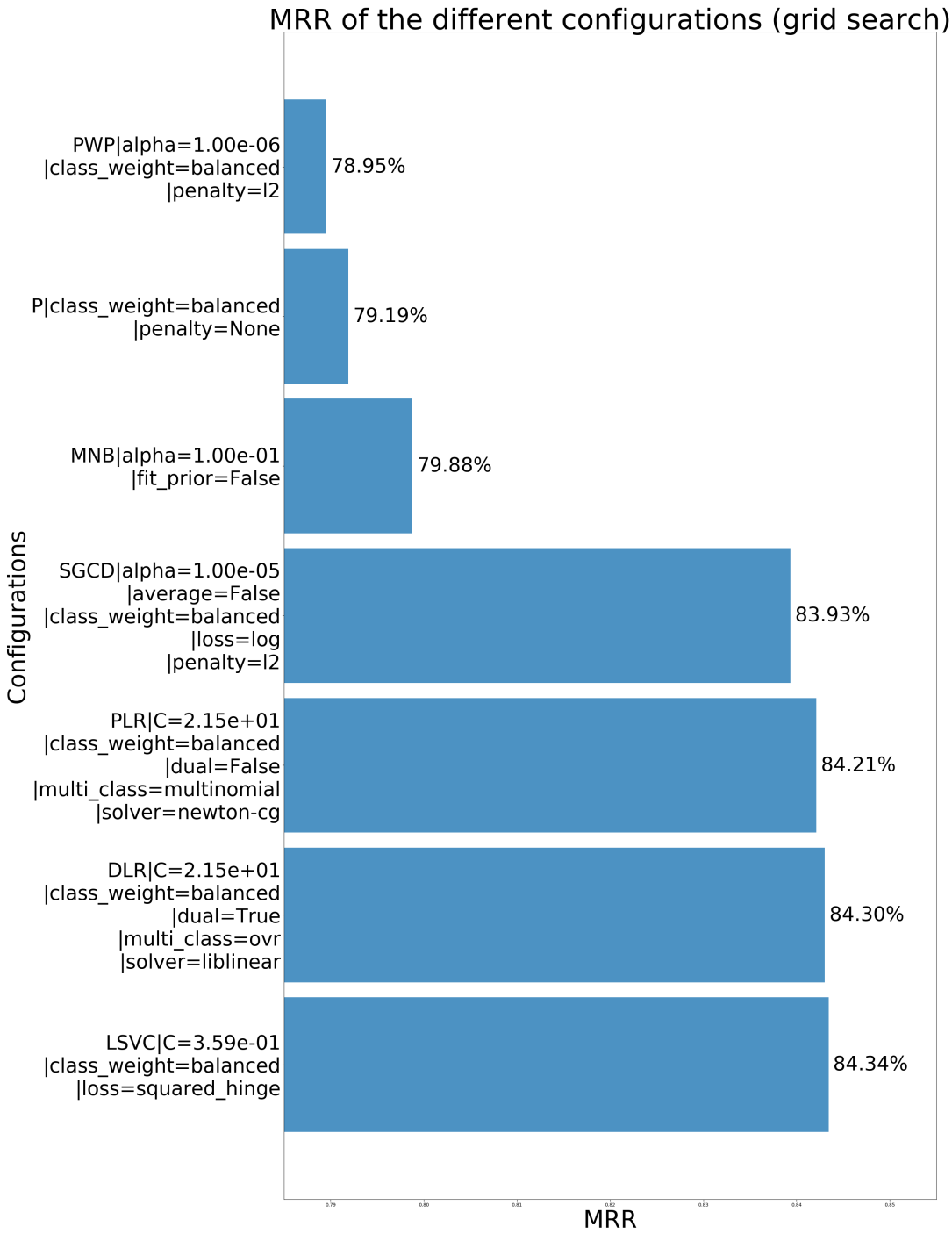


Figure B.26: MRR of the best grid search configurations (in terms of MRR value) on the bug reports of the telecommunications company (the mapping of acronyms to classifiers defined in the Table B.1 has been used)

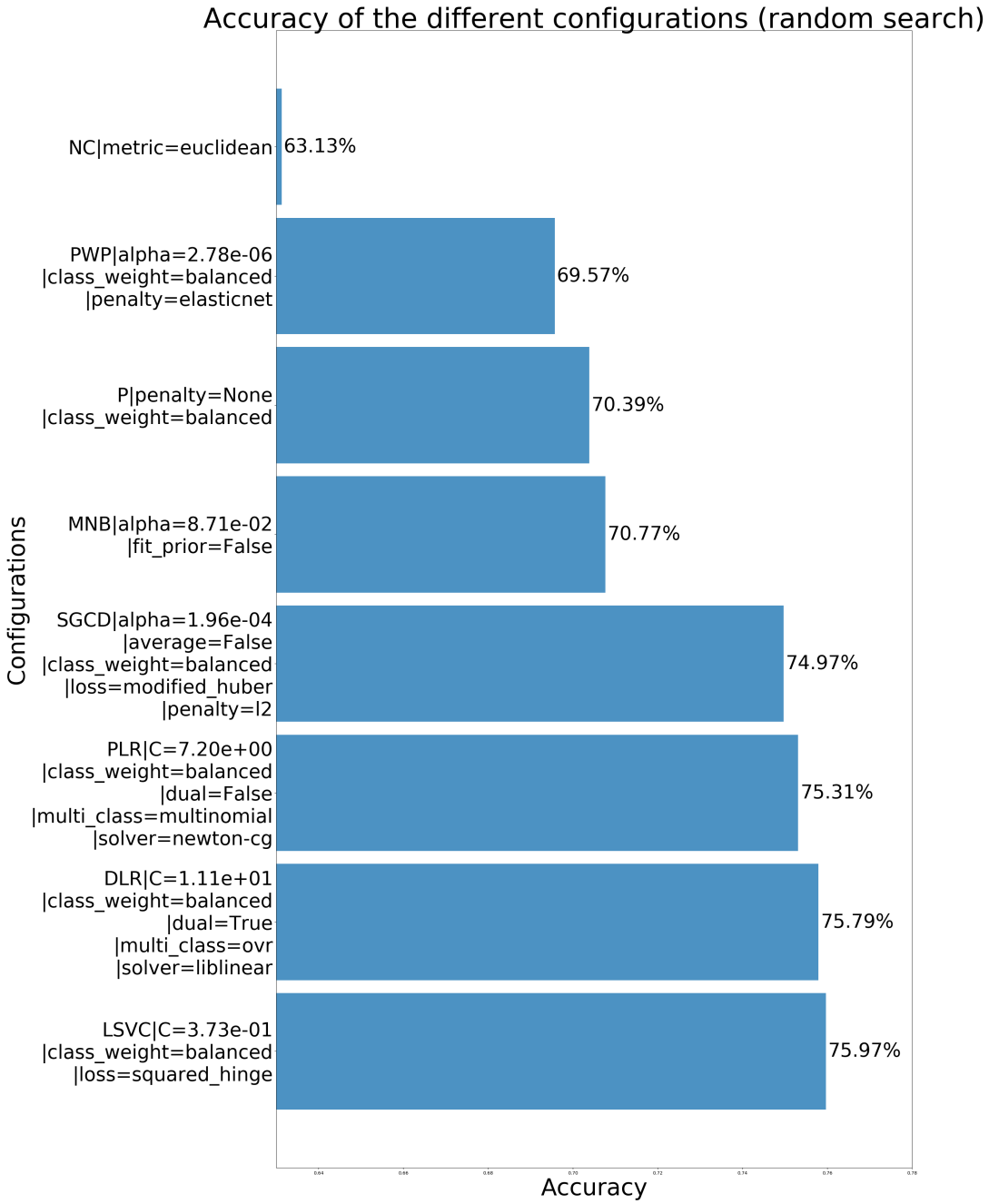


Figure B.27: Accuracy of the best random search configurations (in terms of accuracy) on the bug reports of the telecommunications company (the mapping of acronyms to classifiers defined in the Table B.1 has been used)

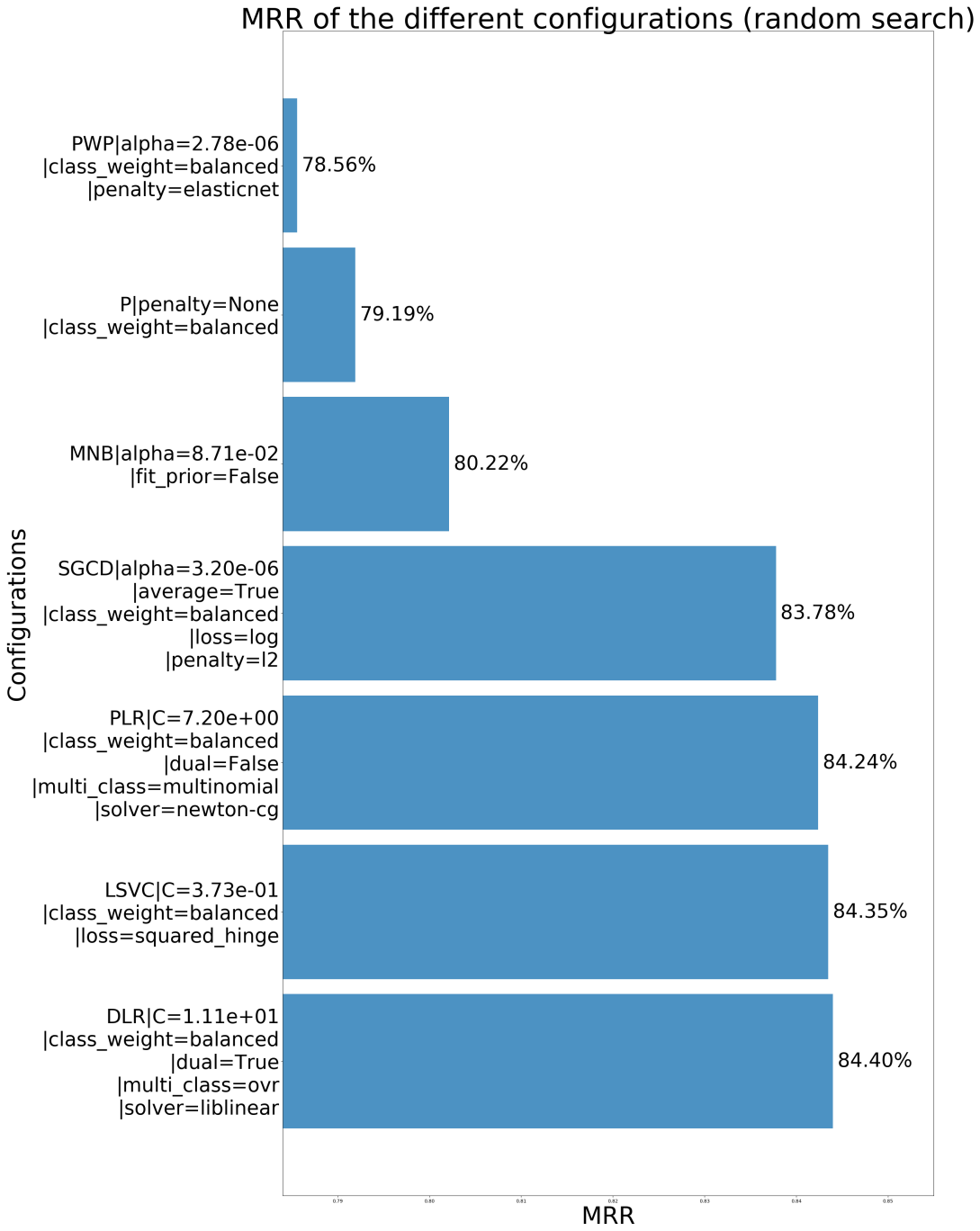


Figure B.28: MRR of the best random search configurations (in terms of MRR value) on the bug reports of the telecommunications company (the mapping of acronyms to classifiers defined in the Table B.1 has been used)

B.4.2 Eclipse JDT

The horizontal bar chart in the Figure B.29 depicts the accuracy of the best grid search configuration (in terms of accuracy) of each classifier on the bug reports of Eclipse JDT.

The horizontal bar chart in the Figure B.30 indicates the MRR value of the best grid search configuration (in terms of MRR value) of each classifier on Eclipse JDT.

Except for the linear SVM classifier, the best configurations of each classifier are the same for both the accuracy and the MRR value (Figures B.29 and B.30). The two best classifiers, in terms of accuracy and MRR value, are, in descending order, the logistic regression (primal problem) and the logistic regression (dual problem). The worst classifier, in terms of accuracy, is the perceptron without penalty.

The horizontal bar chart in the Figure B.31 illustrates the accuracy of the best random search configuration (in terms of accuracy) of each classifier on Eclipse JDT.

The horizontal bar chart in the Figure B.32 shows the MRR value of the best random search configuration (in terms of MRR value) of each classifier on the bug reports of Eclipse JDT.

Except for the linear SVM classifier, the stochastic gradient descent algorithm and the logistic regression (dual problem), the best configurations of each classifier are the same for both the accuracy and MRR metrics (Figures B.31 and B.32). Except for the perceptron with penalty, the stochastic gradient descent algorithm, the linear SVM classifier and the logistic regression (primal problem), the accuracies reached with the best configuration of each classifier, using a random search strategy, are greater than or equal to the ones reached when using a grid search strategy. In terms of accuracy, when using a random search strategy, the classifiers are not ordered exactly the same way as the one obtained when using a grid search strategy (at least two transpositions should be used).

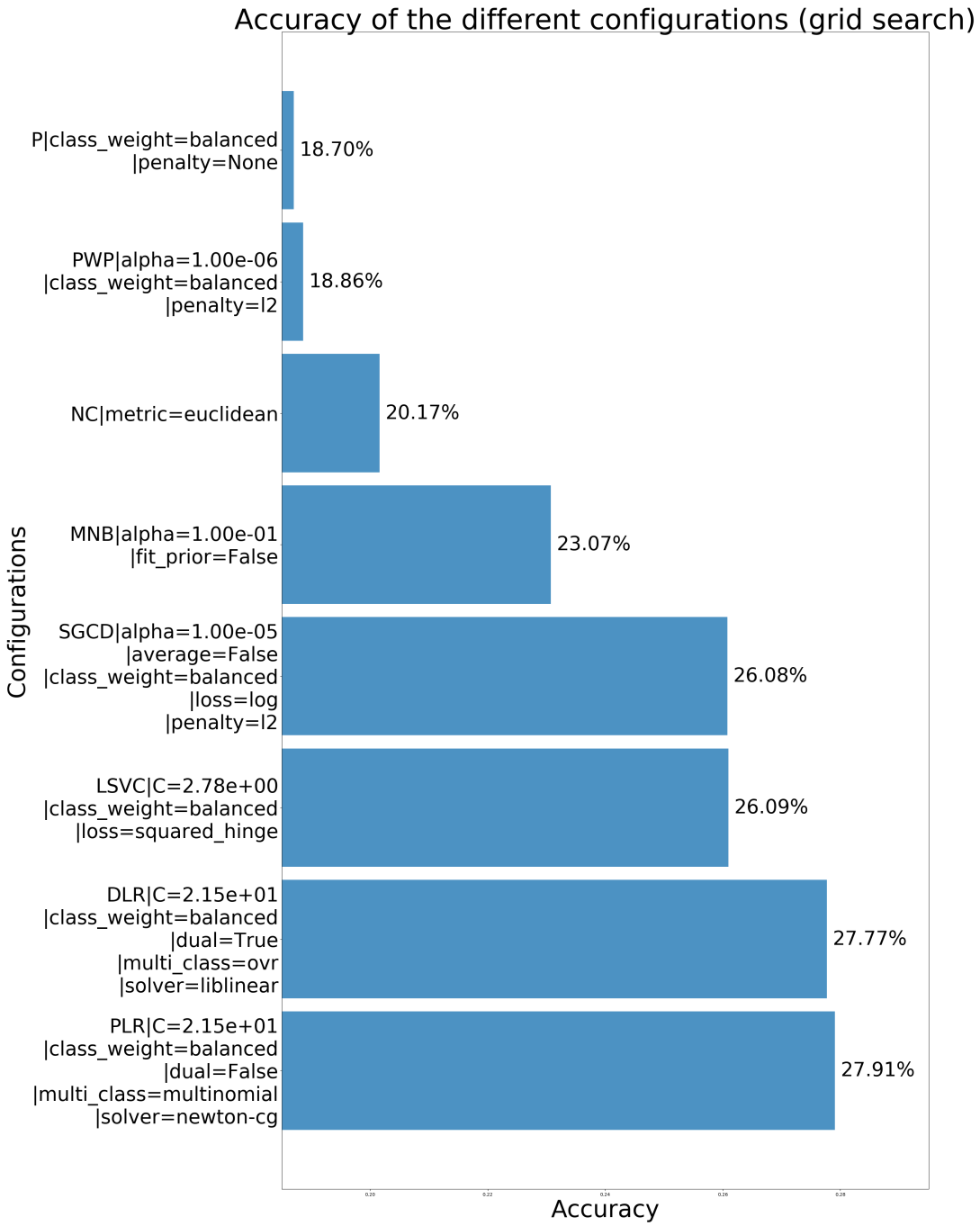


Figure B.29: Accuracy of the best grid search configurations (in terms of accuracy) on the bug reports of Eclipse JDT (the mapping of acronyms to classifiers defined in the Table B.1 has been used)

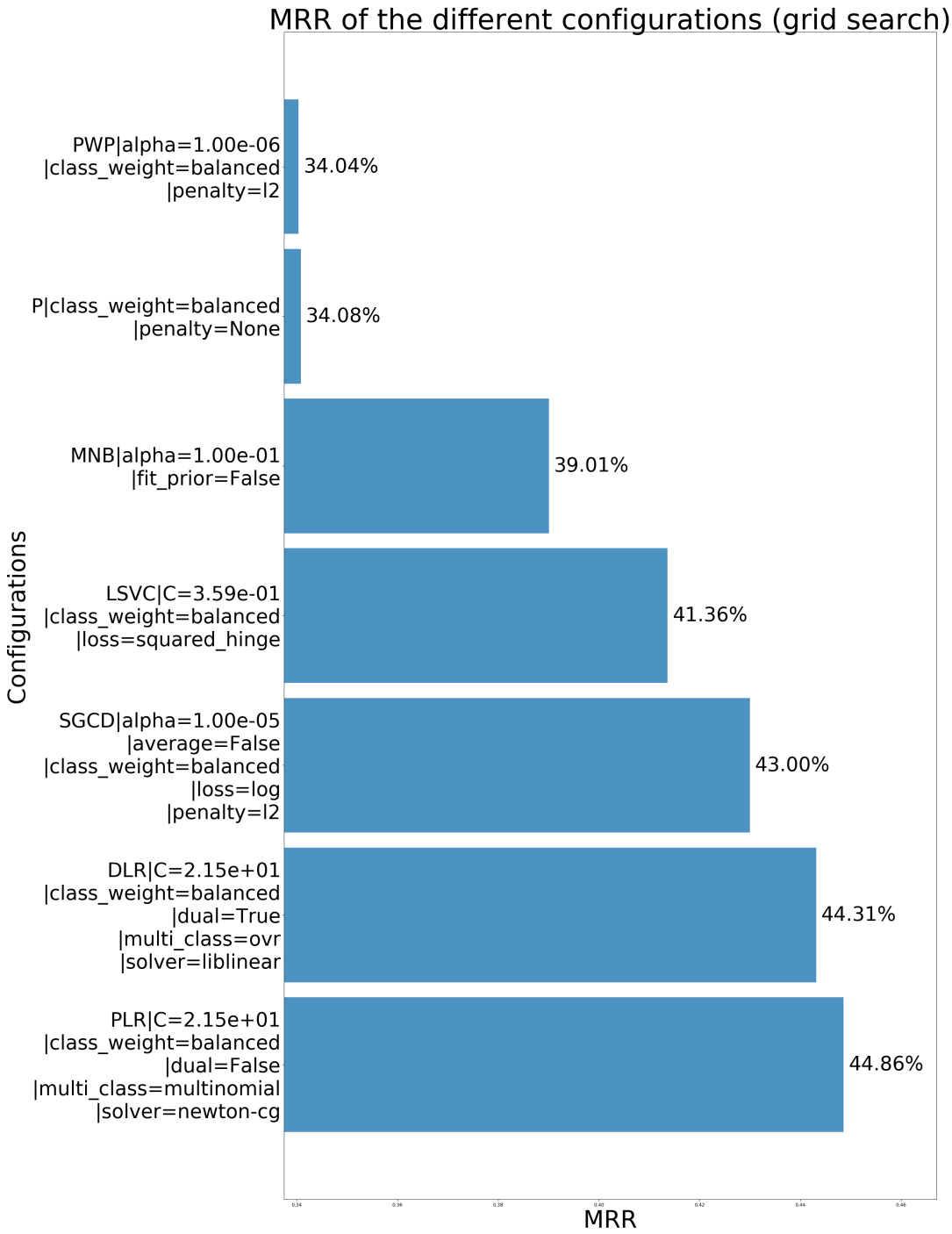


Figure B.30: MRR of the best grid search configurations (in terms of MRR value) on the bug reports of Eclipse JDT (the mapping of acronyms to classifiers defined in the Table B.1 has been used)

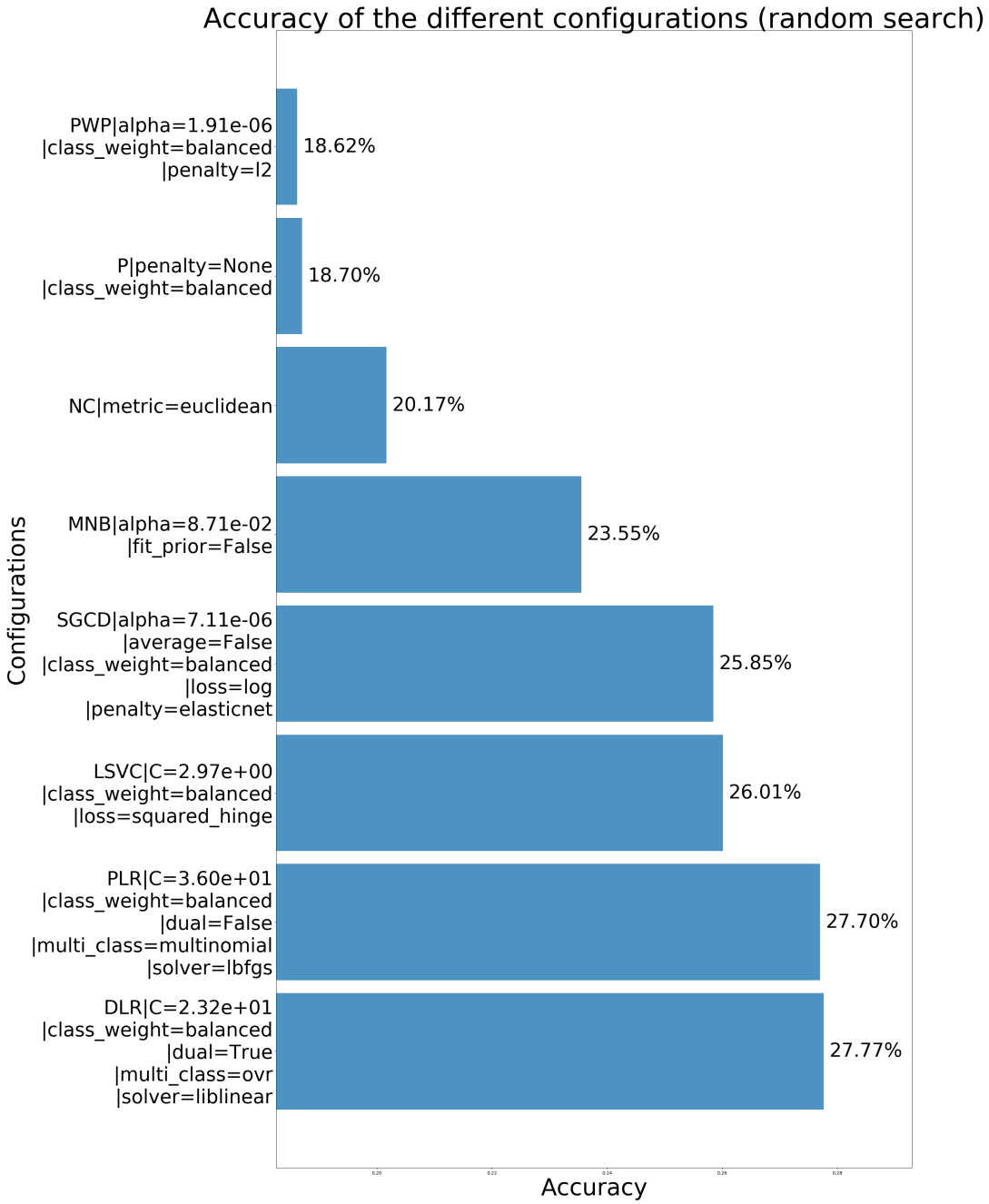


Figure B.31: Accuracy of the best random search configurations (in terms of accuracy) on the bug reports of Eclipse JDT (the mapping of acronyms to classifiers defined in the Table B.1 has been used)

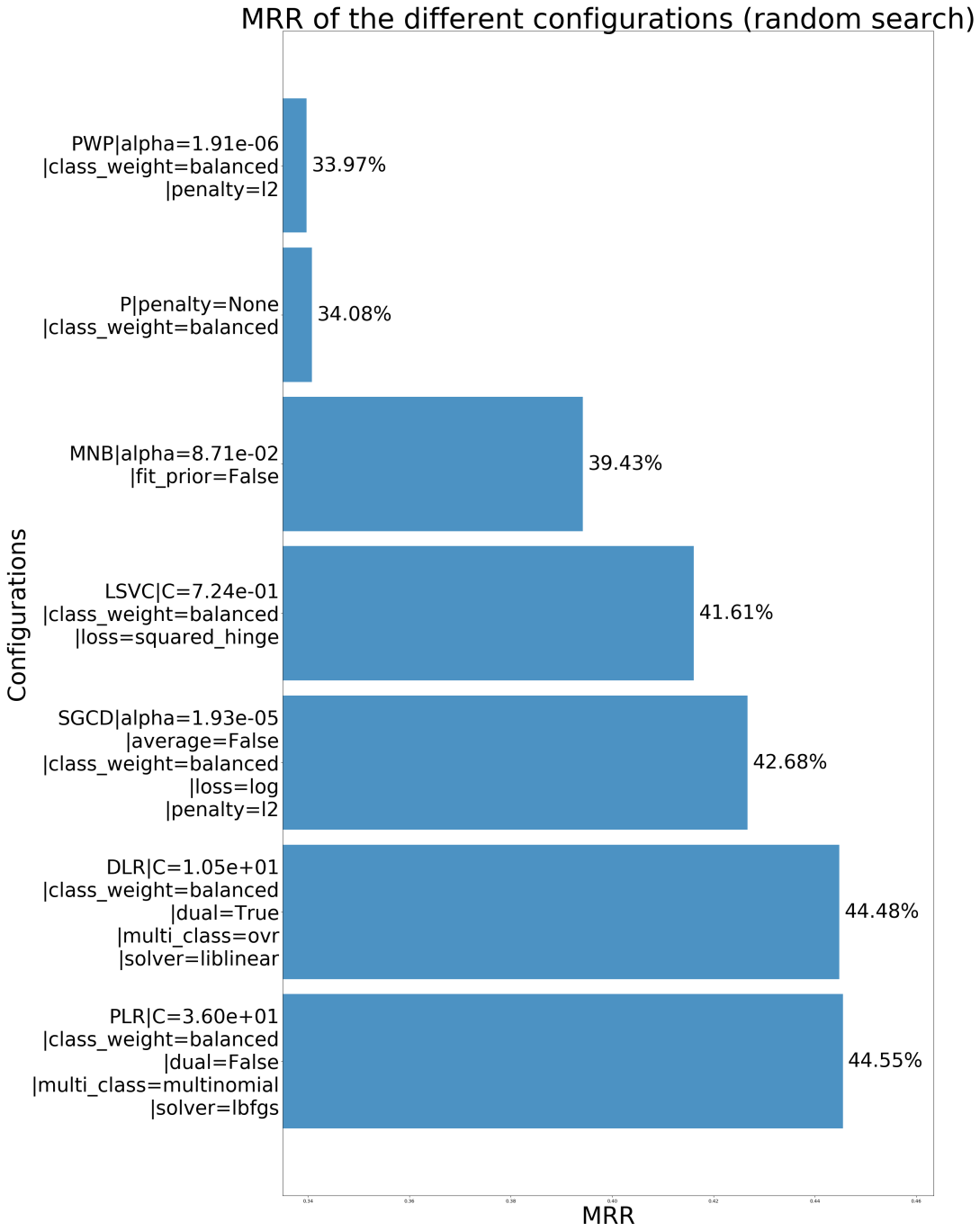


Figure B.32: MRR of the best random search configurations (in terms of MRR value) on the bug reports of Eclipse JDT (the mapping of acronyms to classifiers defined in the Table B.1 has been used)

B.4.3 Mozilla Firefox

The horizontal bar chart in the Figure B.33 shows the accuracy of the best grid search configuration (in terms of accuracy) of each classifier on Mozilla Firefox.

The horizontal bar chart in the Figure B.34 depicts the MRR value of the best grid search configuration (in terms of MRR value) of each classifier on the bug reports of Mozilla Firefox.

Except for the stochastic gradient descent algorithm, the logistic regression (primal problem), the logistic regression (dual problem) and the linear SVM classifier, the best configurations of each classifier are similar for both the accuracy and the MRR value (Figures B.33 and B.34). The best five classifiers, in terms of accuracy and MRR value, are the naive Bayes classifier (multinomial distribution), the logistic regression (primal problem), the logistic regression (dual problem), the stochastic gradient descent algorithm and the linear SVM classifier. The worst configuration, in terms of accuracy, is obtained when using the perceptron without penalty.

The horizontal bar chart in the Figure B.35 shows the accuracy of the best random search configuration (in terms of accuracy) of each classifier on Mozilla Firefox.

The horizontal bar chart in the Figure B.36 illustrates the MRR value of the best random search configuration (in terms of MRR value) of each classifier on the bug reports of Mozilla Firefox.

Except for the stochastic gradient descent algorithm, when using the random search strategy, the best configurations of each classifier are the same for both the accuracy and MRR metrics (Figures B.35 and B.36). Except for the logistic regression (dual problem), the accuracies reached with the best configuration of each classifier, using a random search strategy, are greater than or equal to the ones reached when using a grid search strategy. In terms of accuracy, when using a random search strategy, the classifiers are ordered the same way as the one obtained when using a grid search strategy.

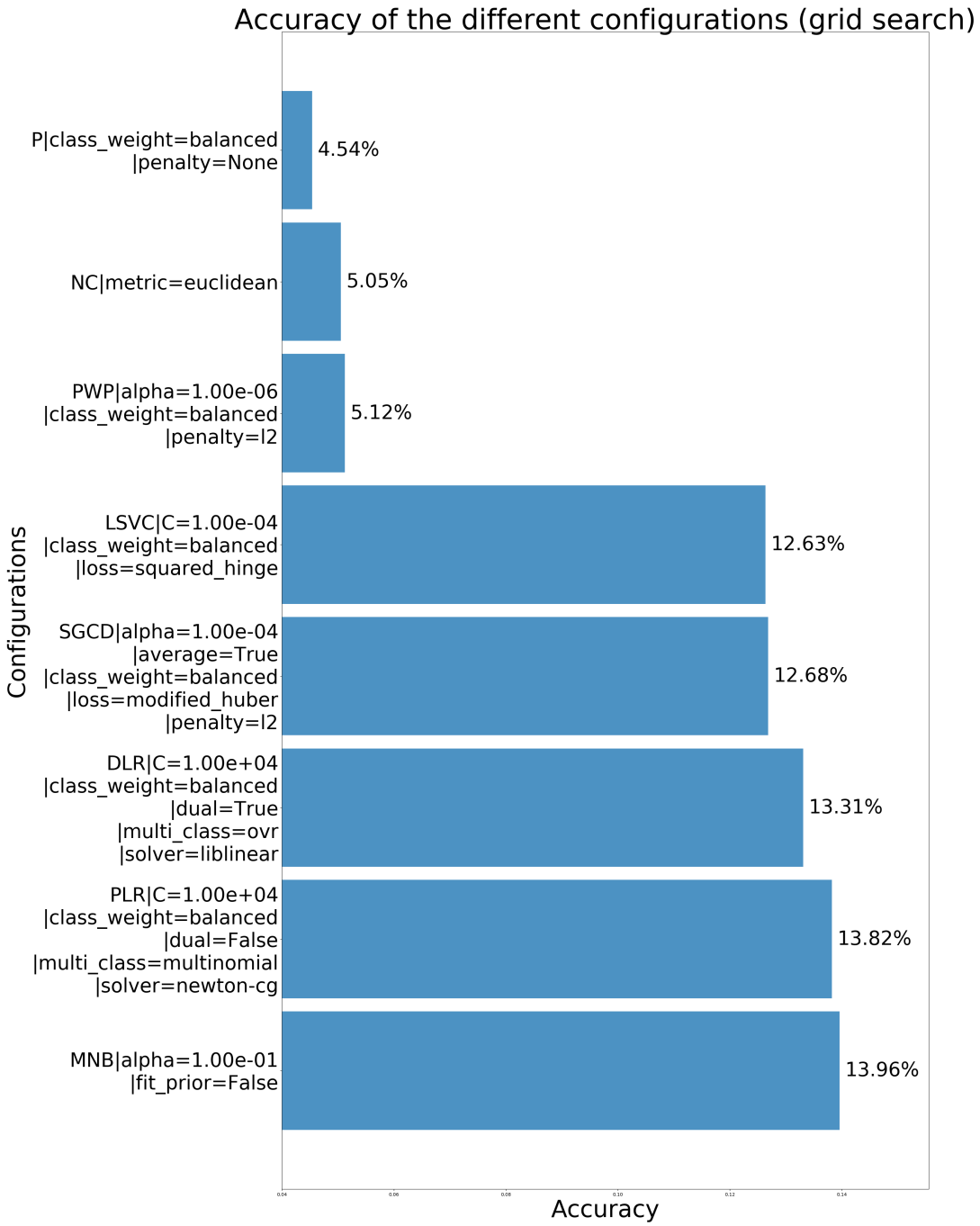


Figure B.33: Accuracy of the best grid search configurations (in terms of accuracy) on the bug reports of Mozilla Firefox (the mapping of acronyms to classifiers defined in the Table B.1 has been used)

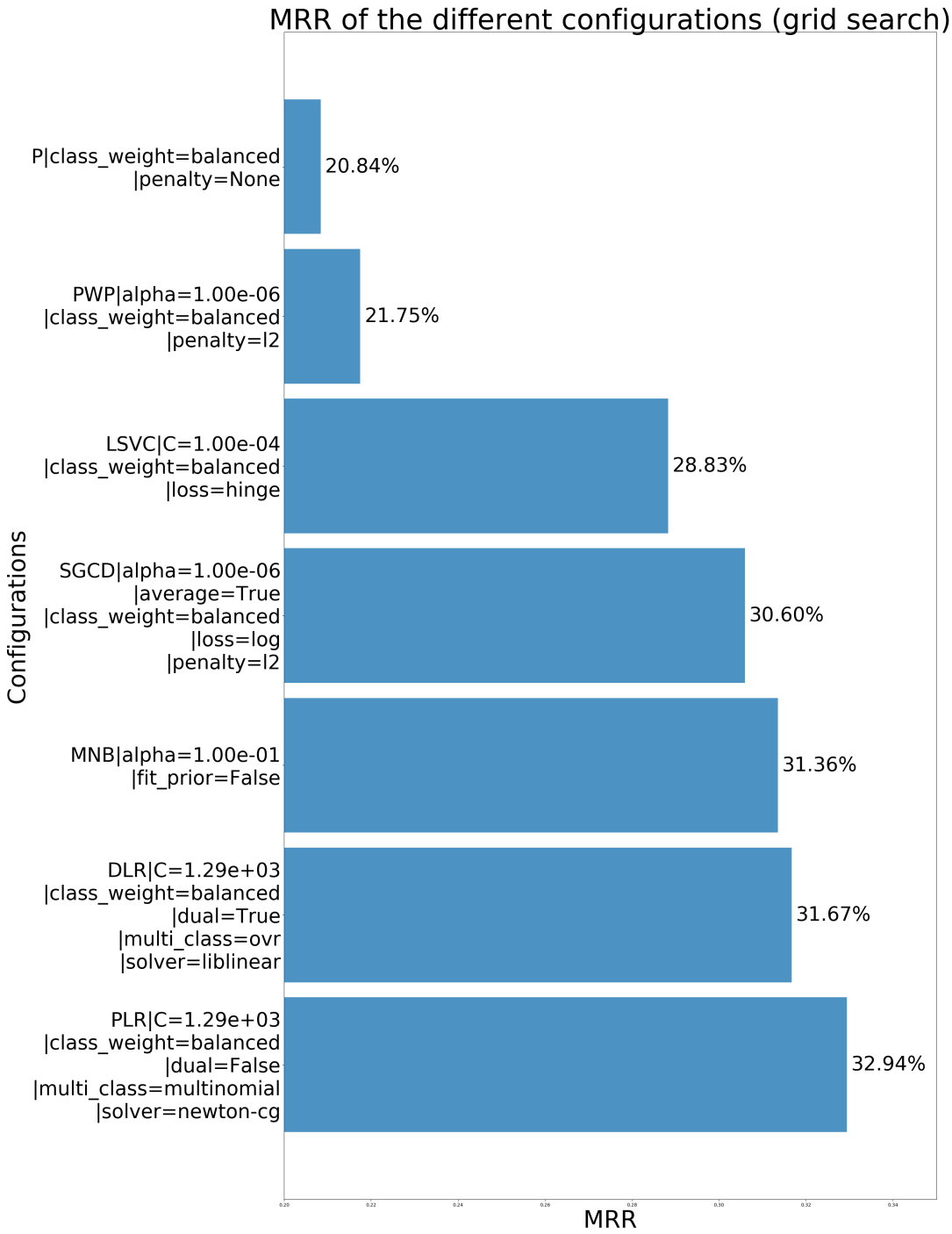


Figure B.34: MRR of the best grid search configurations (in terms of MRR value) on the bug reports of Mozilla Firefox (the mapping of acronyms to classifiers defined in the Table B.1 has been used)

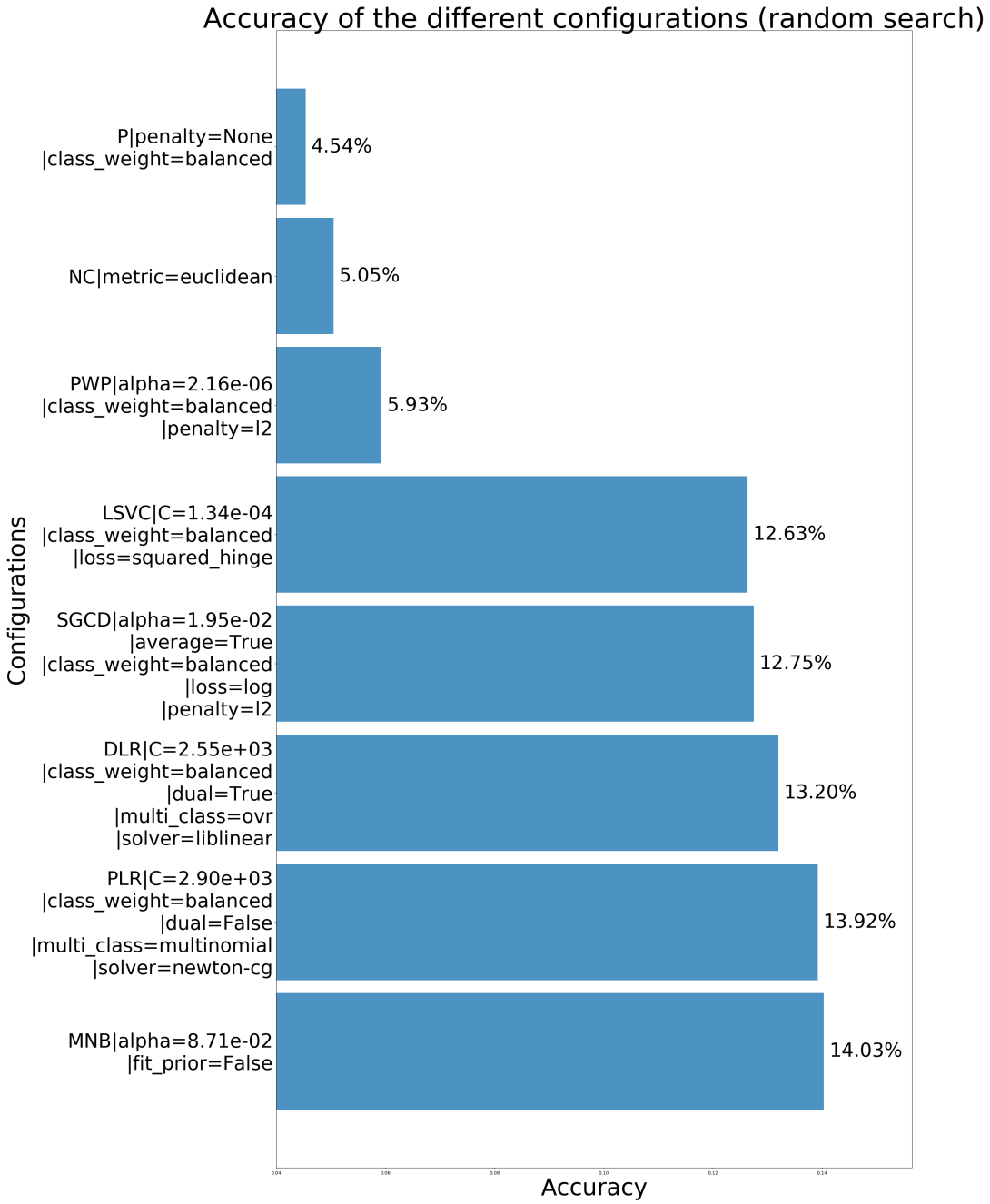


Figure B.35: Accuracy of the best random search configurations (in terms of accuracy) on the bug reports of Mozilla Firefox (the mapping of acronyms to classifiers defined in the Table B.1 has been used)

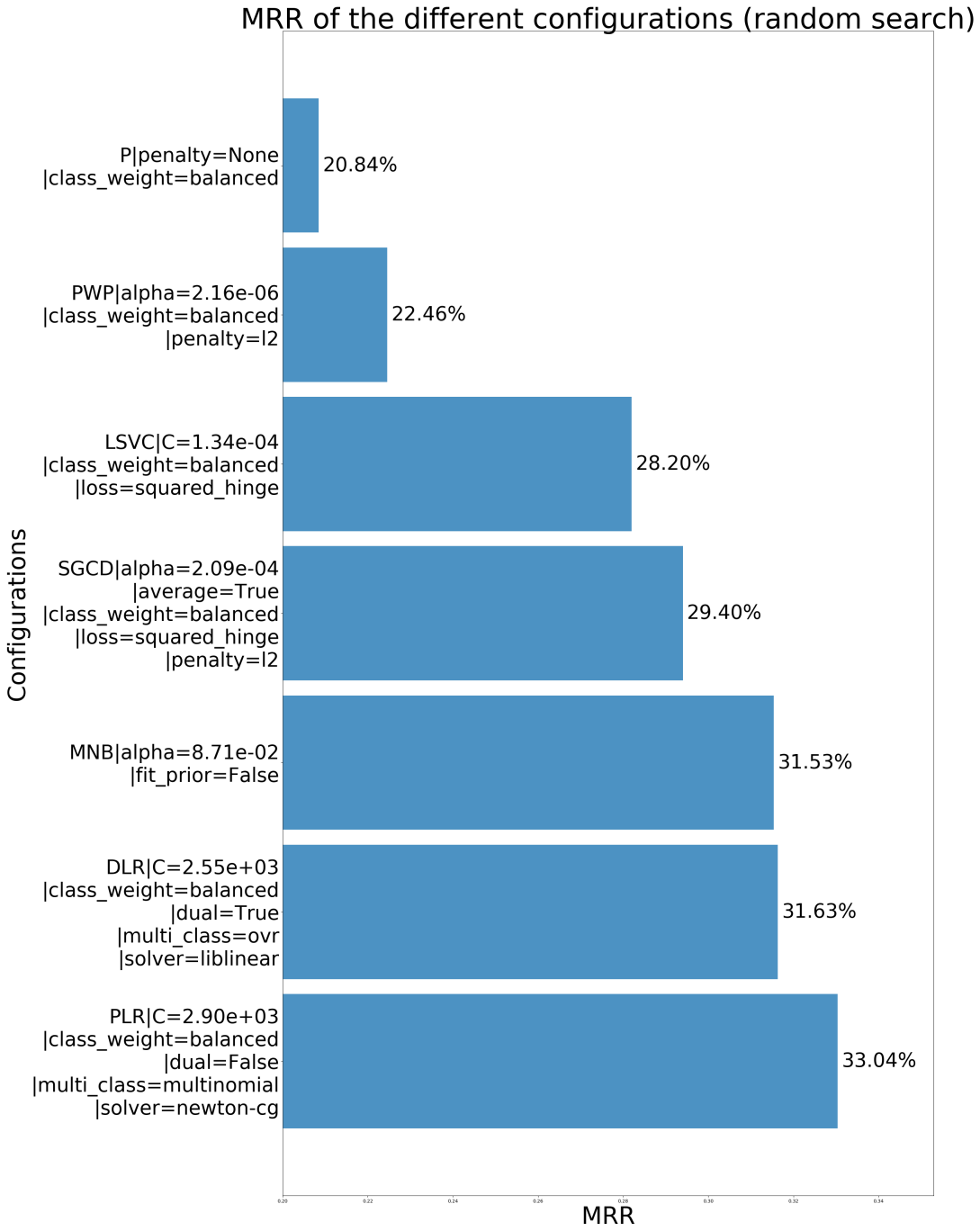


Figure B.36: MRR of the best random search configurations (in terms of MRR value) on the bug reports of Mozilla Firefox (the mapping of acronyms to classifiers defined in the Table B.1 has been used)