TU WIEN Informatics

Machine Learning for Interactive Performance Prediction

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Data Science

eingereicht von

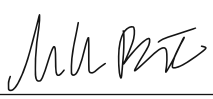
Markus Böck, BSc
Matrikelnummer 01634838

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Assistant Prof. Dipl.-Ing. Dr.sc. Jürgen Cito, BSc

Wien, 21. Juni 2022



Markus Böck

Jürgen Cito

TU WIEN Informatics

Machine Learning for Interactive Performance Prediction

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Data Science

by

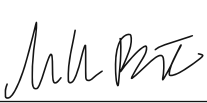
Markus Böck, BSc
Registration Number 01634838

to the Faculty of Informatics

at the TU Wien

Advisor: Assistant Prof. Dipl.-Ing. Dr.sc. Jürgen Cito, BSc

Vienna, 21st June, 2022



Markus Böck

Jürgen Cito

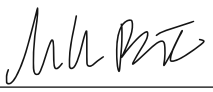
Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Markus Böck, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 21. Juni 2022



Markus Böck

Acknowledgements

First, I would like to express my gratitude to my supervisor, Jürgen Cito, who guided me throughout this project. I have benefited greatly from your expertise and insightful feedback. Thank you for taking me on as a student and for the opportunities created beyond this thesis.

I wish to extend my special thanks to Marco Castelluccio and Dave Hunt from Mozilla for providing me invaluable input about the Firefox project, without which the thesis would not have been possible in this form. I also want to thank Moritz Beller for helping with the bag-of-words approach.

I am grateful for my parents whose constant love and support keep me motivated. You encouraged my interest in science and technology since primary school. I could not have completed this dissertation without the support of my family!

Last, but not least, I would like to thank Caroline and Julian for always listening to my rants as well as the fun distractions from work.

Kurzfassung

Software Performance ist eine wichtige nicht-funktionelle Projektanforderung. Produkte wie Webbrowser oder Videospiele müssen ihre Ladezeit niedrig und User-Interaktion flüssig halten um am Markt kompetitiv zu bleiben. Weil vollständiges Testen und Benchmarking für große Projekte nicht möglich ist, müssen andere Tools in den Softwareentwicklungsprozess integriert werden um ein performantes System zu garantieren.

In dieser Arbeit betrachten wir den Open-Source-Webbrowser Mozilla Firefox und konstruieren ein Machine Learning Modell, das Programmcode identifizieren soll, der womöglich Performance-Probleme verursacht. So ein Modell kann verwendet werden um Softwareentwicklern durch frühe Hinweise auf verdächtigen Programmcode interaktives Feedback zu geben, oder etwa Code-Reviewern zu helfen ihre Aufmerksamkeit auf Programmcode zu lenken, der wahrscheinlich ein festgestelltes Peformance-Problem ausgelöst hat.

Die entscheidende Herausforderung beim Erkennen von Performance-Problemen ist das Data-Labeling - also zu bestimmen welcher Programmcode Probleme in der Vergangenheit verursacht hat. Da der SZZ Algorithmus, der häufig für diese Aufgabe bei der Software-Defekt-Vorhersage verwendet wird, nur unzureichend präzise war, stellen wir eine Labeling-Methode vor, die direkt auf Assoziationen von Bug-auslösenden und Bugbehebenden Issues im Bug-Tracking-System basiert. Obwohl viel Arbeit in traditionelles Feature-Engineering, wie das Berechnen von Programmkomplexitätsmetriken, gesteckt wurde, funktioniert ein Bag-of-words-Modell am besten und erzielt eine 5.7 mal höhere Genauigkeit als zufälliges Raten. Das finale Modell übertrifft das beste auf dem SZZ Algorithmus basierende Modell dreimal mit einem F1-Score von 0.1745, Genauigkeit von 0.2022 und Sensitivität von 0.1535.

Abstract

Software performance is an important non-functional project requirement. Products like web browsers or video games have to keep their loading times low and user interaction smooth to stay competitive in the market. Since exhaustive testing and benchmarking is infeasible for large-scale projects, other tools have to be integrated in the software development process to ensure a performant system.

In this thesis, we study the open source web browser Mozilla Firefox and build a machine learning model to predict which source code changes are prone to cause performance regressions. Such a model could be employed to give interactive feedback to developers by raising early warnings for suspicious code, or could be used to help code reviewers to focus their attention on code changes which are likely to have caused a detected performance problem.

The key challenge of predicting performance regressions is the difficulty of data labeling, i.e. determining which code change caused a regression in the past. After evaluating the SZZ algorithm, commonly used in software defect prediction for this task, to be insufficiently accurate, we present a labeling approach based directly on associations of bug-introducing and bug-fixing issues in the bug-tracking-system. Even though a lot of effort is put into traditional feature engineering, like computing source code complexity metrics, a bag-of-words model performs best and scores a 5.7 times higher precision than random guessing. The final model outperforms the best model based on the SZZ algorithm three times with a F1-score of 0.1745, precision of 0.2022 and recall of 0.1535.

Contents

CHAPTER 1

Introduction

A large-scale software product undergoes many changes during its development and lifetime. Faulty code changes may lead to a degradation in performance, e.g., longer loading times for websites or a drop in frame rate in video games. Once a product is released, it is critical to detect such defects as early as possible to ensure a satisfactory user experience. Due to frequent changes, exhaustive benchmarking is often too expensive and not a viable option.

A more feasible approach is to only selectively benchmark the product throughout its versions and learn from the collected data which kind of code changes are prone to cause performance regressions. For example, a code change consisting of hundreds of lines spread among multiple files is more probable to be faulty than one that makes only small changes in a single file.

The overall aim of this work is to create a machine learning model capable of detecting code changes that may introduce performance regressions. This model could be employed to give interactive feedback to developers by raising early warnings for suspicious code changes and prevent performance regression in the production environment. Once a performance regression has been detected, the model could also be used to help code reviewers to focus their attention on code changes which may have caused the problem. Therefore, a good model may help reduce software development effort and cost.

The central research question of this thesis is:

> *To what extent are machine learning models capable of detecting performance regression inducing code changes from source code features in a just in time manner, specifically at code edit or commit time?*

To answer the question, we consider the open-source web browser project Firefox by Mozilla. Web browsers are performance critical software products for various reasons.

For example, to provide a good user experience, it is important how fast a website loads, how smooth a video plays back, or how responsive the navigation of a website feels. The open access of the Firefox project and the performance aspects of a web browser make Firefox a perfect candidate for our research purposes.

In this work, we leverage the fact that large-scale software projects like Firefox are organised in form of repositories in *version-control-systems* (VCS). Developers make their edits in small chunks and *commit* their file and code changes to the repository with a commit message. This allows us to look at a software project throughout its incremental versions and extract the precise changes - the *diffs* - from one version to the next [Ott09].

In addition to version-control-systems, *bug-tracking-systems* (BTS) are used to organise and track tasks that have to be completed in the project. The tasks include the implementation of features, fix of software bugs, or other program enhancements [JPZ08]. Bug-tracking-systems are our primary source for collecting information about bugs of a software project, in particular performance regressions.

However, a big challenge for defect prediction is the fact that benchmark and bug data are separate from the source code changes. This means that to be able to fit a machine learning model, first, performance regressions detected in the past have to be carefully matched to the code changes which have introduced them. In the software defect prediction research field, the SZZ algorithm [SZZ05] is an automated way to bridge this gap. This leads to the question:

> *Is the SZZ algorithm suitable for labeling performance regression inducing code changes correctly?*

Another approach to the labeling problem is to make use of the fact, that since mid 2019, once they have fixed a performance problem, Firefox developers link the *bug number* of the issue in the bug-tracking-system which caused the problem. This number can be directly mapped to the commits which worked on the issue. However, since there are in general multiple commits per issue, we investigate:

> *Is a labeling based on bug numbers suitable for detecting performance regression inducing code changes?*

The labeling is not the only challenge for predicting performance regressions. Performance regression inducing code changes occur only rarely in comparison to sane commits. This leads to an extremely unbalanced data sets with only about 1% positive labels. We employ common sampling techniques described in Section 4.3 to deal with this problem and ask ourselves:

> *Which sampling technique is best for dealing with the imbalanced nature of performance regression data sets?*

Furthermore, we test a multitude of machine learning models and explore a large space of models with automated machine learning methods to answer:

> *Which machine learning model is best at generalising to unseen data for the task of performance regression prediction?*

In view of advances in natural language processing techniques for source code [FGT+20, ACRC21], one may ask whether it is better to manually engineer features that capture source code metrics relevant to the task, or if it is better to simply take the raw source code as input, as would be enough for human software engineers. Training a sophisticated natural language processing model is beyond the scope of this work, but to get an indicator to the full answer, we test a simple bag-of-words model, which was successfully used for performance regressions prediction in research [SWAK12, BLN+22] and investigate:

> *How does a bag-of-words model compare to a model trained on hand-crafted features?*

Performance bug prediction can be considered as a specialised case of general software defect prediction. Naturally, it is interesting whether the models trained on the performance regression subset are fundamentally different or roughly the same as general defect prediction models. By looking at the feature importance of the best models, we try to answer:

> *How does a performance regression prediction model differ from a general defect prediction model?*

The thesis is structured as follows: Chapter 2 gives an overview of the machine learning background for this thesis as well as an introduction to the SZZ algorithm. We list related work in Chapter 3. After presenting our approach to predicting performance regressions in Chapter 4, we describe our data labeling process in Chapter 5. Lastly, we report our experiment results in Chapter 6 and answer the research questions posed above in Chapter 7.

CHAPTER 2

Background

2.1 Machine Learning for Defect Prediction

Formally, in a supervised machine learning setting, we have observations in form of a feature matrix $\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_n]^T \in \mathbb{R}^{n \times d}$ and a target vector $\mathbf{y} \in \{0, 1\}^n$ [Bis06]. In our particular case, $\mathbf{x}_i \in \mathbb{R}^d$ represent code changes and $y_i = 1$ if the corresponding code changes are considered to have introduced a performance regression, otherwise $y_i = 0$. Methods for representing code changes with vectors $\in \mathbb{R}^d$ are discussed in the next section and our approach is described in Section 4.2.

The goal of machine learning is to find a function $\hat{\mathbf{y}}(\mathbf{x}) \in \{0, 1\}$, which predicts whether a new code change $\mathbf{x}$ is likely to introduce a performance regression. This prediction function is constructed by considering different models, see Section 2.3, and optimising their parameters - *training them* - on our observed data such that we minimise misclassification on past data, $\hat{\mathbf{y}}(\mathbf{x}_i) \neq y_i$. The models also have so called hyper-parameters - parameters which are fixed in the learning process, but also influence the performance of the model. We will perform a hyper-parameter search to find the best values for these parameters to obtain the best model for each model class.

2.2 Feature Engineering for Software Projects

2.2.1 Traditional Features

There are many different metrics used in the field of software fault prediction. They can be broadly split up into following categories [RHTŽ13]:

- Traditional metrics: size (e.g. number of lines) and complexity metrics.
- Object-oriented metrics: coupling, cohesion and inheritance metrics at a class-level.

- Process metrics: metrics extracted from the combination of source code and repository information (e.g. developer experience).

In this work, we want to make predictions at commit or code edit time. This requires the metrics to be fast to compute. For example, compiling the software project to extract metrics is not feasible. Furthermore, the metrics should be applicable to many programming languages.

2.2.2 Bag-Of-Words

As opposed to handcrafting features which capture source code and development characteristics, in theory it should be enough to just look at the source code to figure out which commits are susceptible to cause problems, just like a human software developer would do. For this reason, natural language processing methods (NLP), taking raw source code as input, have gained attention in recent software engineering research, for instance, large transformer-based models like CodeBERT [FGT+20] or PLBART [ACRC21].

Such large models are beyond the scope of this work, but we test the applicability of NLP methods with a simple bag-of-words model [MRS10], an approach which has been successfully implemented in existing literature [SWAK12, BLN+22]. In a bag-of-words model, text is represented as a list of words, so called *tokens*, ignoring the order or grammar of the words. The tokens belong to a vocabulary of fixed size N and the feature vector $\in \mathbb{R}^N$ is computed by simply counting the occurrences of each token in the text. The difficult part of this approach is to find an appropriate tokenization process for source code, see Section 4.2.2.

2.3 Machine Learning Models

In the field of software defect prediction many machine learning techniques have been applied [PMT21]. To be able to answer the central research question of this thesis, we made a selection of machine learning models with different properties, which will be briefly described in the following.

2.3.1 Logistic Regression

Recall that $\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_n]^T \in \mathbb{R}^{n \times d}$ is the feature matrix, where n is the number of samples and d is the number of features. The vector $\mathbf{y} \in \{0, 1\}^n$ is the target representing the true class labels. The logistic model is given by

$$p_i := P(y_i = 1 | \mathbf{w}, \mathbf{x}_i) = \sigma(\mathbf{w}^T \mathbf{x}_i), \quad \text{where} \quad \sigma(a) = \frac{1}{1 + \exp(-a)}.$$

The model coefficients $\mathbf{w} \in \mathbb{R}^d$ are found by minimising the regularised cross entropy loss,

$$-\ln P(\mathbf{y} | \mathbf{w}, \mathbf{X}) = \sum_{i=1}^{n} y_i \ln(p_i) + (1 - y_i) \ln(1 - p_i) + C \left\| \mathbf{w} \right\|_p^p, \tag{2.1}$$

where C and p are regularisation hyper-parameters. For more information see Bishop Chapter 4 [Bis06].

2.3.2 Support Vector Machine

Like the logistic regression model, the support vector machine (SVM) classifier is also linear in its parameters. However, the SVM maximises the margin of its decision boundary - the smallest distance between the boundary and any data point - to improve the separation of the classes. Data points lying on the margin are called support vectors.

Alongside the regularisation parameter C, the SVM allows us to specify a kernel to be used to measure the similarity of data points. This is equivalent to transforming to a (often higher dimensional) feature space and using the dot product $k(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)$. We will consider a radial basis function and a polynomial kernel as well as different values for C in the parameter tuning. For more details see Bishop Chapter 7 [Bis06]. To make the SVM work with our high dimensional data, we use the Nystroem approximation technique [WS00].

2.3.3 Multi-Layer Perceptron

A multi-layer perceptron (MLP) can be seen as a generalisation of the logistic model. Instead of only one coefficient vector, we now have multiple layers of weight matrices $\mathbf{W}^{(l)} \in \mathbb{R}^{n_l \times n_{l-1}}$, bias vectors $\mathbf{b}^{(l)} \in \mathbb{R}^{n_l}$, and a non-linear (element-wise) activation function $\sigma \colon \mathbb{R} \to \mathbb{R}$, such that the input $\mathbf{x}_i$ is passed through multiple linear and non-linear transformations,

$$\mathbf{x}_i^{(l)} = \sigma(\mathbf{W}^{(l)} \mathbf{x}_i^{(l-1)} + \mathbf{b}^{(l)}), \quad \mathbf{x}_i^{(0)} := \mathbf{x}_i.$$

Again, the cross entropy loss (2.1) is minimised where the output of the last layer is used for p_i.

For the handcrafted input features, the MLP does not have to extract features through learning and thus, only one hidden layer is sufficient for the classification task. Due to the high dimensionality the MLP is not feasible for the bag-of-words feature vectors. In parameter tuning, we change the number of nodes in the hidden layer, consider different activation functions (relu, logistic, tanh), as well as different regularisation parameters as before. For a detailed overview of MLPs see Bishop Chapter 5 [Bis06].

2.3.4 Random Forest

The random forest classifier [Bre01] is an ensemble of decision trees, a machine learning model where the feature space is divided by sequential decisions ordered in a tree-like fashion. This means that the final prediction is based on the aggregation of predictions from multiple decision trees trained on random subsets of features and bootstrap samples of the data. This process called *bagging* mitigates the problem of high variance common for decision trees.

There are several hyper-parameters to control the generation of the decision trees like the number of trees, maximum depth, or the minimum number of samples needed to further split the feature space, all of which will be explored in the search.

2.3.5 XGBoost

Like random forests, the XGBoost model [CG16] is also based on decision trees. However, it is a so called *gradient tree boosting* model. This means that in contrast to random forests, the decision trees are not generated independently, but models are sequentially added to correct the errors made by previous models via gradient descent methods.

Like before, there are also hyper-parameters for controlling the generation of decision trees. Furthermore, there are regularisation parameters and parameters that help to deal with imbalanced data.

2.4 Interpretability

Having a model that is capable of accurately predicting performance regressions is good, but it is also important to understand why the model thinks a particular code change is prone to cause problems. Telling a developer that their edit is likely to cause a regression by itself is not helpful. It would be more productive to be able to to give the reason for the decision, for example, that the code complexity increases significantly with their edit.

The SHAP (shapley additive explanations) method [LL17] takes a game theoretic approach to explain the output of any machine learning model. For each data point x we can assign a shapley value $\phi_j(x)$ to feature j, which can be interpreted as follows: The value of the j-th feature contributed $\phi_j(x)$ to the prediction of this particular instance x compared to the average prediction for the data set [Mol22].

Shapley values have the additive property that if we sum over all $\phi_j(x)$, we get the predicted value for the data point x minus the average predicted value,

$$\sum_{j=1}^{\#\text{features}} \phi_j(x) = \hat{f}(x) - \mathbb{E}\hat{f}(X).$$

This means that the prediction at a data point is distributed additively among all features. To interpret and explain a model as a whole, we can plot the shapley values alongside the feature values for all data points, which we will use extensively in Section 6.5.

2.5 SZZ Algorithm

In 2005, Sliwerski, Zimmermann, and Zeller (“SZZ”) proposed an algorithm for the automatic identification of bug-introducing code changes [SZZ05]. Since its introduction, the algorithm gained great popularity in the field of empirical software engineering and plays a foundational role in many software defect prediction methods [RRGB18].

The first part of the algorithm is concerned with finding bug-*fixing* code changes, which is done by syntactically analysing commit messages. For example, one may look for keywords like “bug”, “fix”, “problem”, etc. But ideally, each commit message references the bug number of the issue in the bug-tracking-system which was worked on in the code change in a predefined format, e.g., “Bug #123456: Further comments...”, like in the Firefox repository. This provides an exact mapping from bug-fix to the corresponding code change and even allows more filtering options by making use of the information in the bug-tracking-system. For example, this enables us to filter for performance regression fixes.

Having identified bug-fixing code changes, the algorithm employs the `diff` functionality of the version-control-system to find the source code lines that were modified in the fix and then applies the `annotate/blame` functions to determine the commits which last modified these particular lines. The commits detected this way are then filtered to match the bug report timeline (e.g. commits made after a bug was reported could not have caused the bug). The remaining commits are considered to be bug-introducing. The rationale behind this procedure is that a bug-fix has to correct specific lines that were previously added or modified and caused a problem. This is best explained with following example.

Bug 1717171 in Mozilla’s Firefox repository was a small fix. In Figure 2.1 on the left, you can see the code change of the fix, the *diff*, in the version-control format. By looking at the deleted (red) and added lines (green) we can see that this particular performance regression was solved by changing the way the variable `currentTime` is initialised. By tracing back which commit added line 1.12 to 1.14 we find the code change in Figure 2.1 on the right, which we accordingly label as bug-introducing.

```
1.1  --- a/dom/media/MediaDecoderStateMachine.cpp
1.2  +++ b/dom/media/MediaDecoderStateMachine.cpp
1.3  @@ -587,19 +587,17 @@ class MediaDecoderStateMachine::Decoding
1.4
1.5     void HandleAudioDecoded(AudioData* aAudio) override {
1.6       mMaster->PushAudio(aAudio);
1.7       DispatchDecodeTasksIfNeeded();
1.8       MaybeStopPrerolling();
1.9     }
1.10
1.11    void HandleVideoDecoded(VideoData* aVideo, TimeStamp aDecodeStart) override {
1.12 -    const auto currentTime = mMaster->mMediaSink->IsStarted()
1.13 -                               ? mMaster->GetClock()
1.14 -                               : mMaster->GetMediaTime();
1.15 +    const auto currentTime = mMaster->GetMediaTime();
1.16      if (aVideo->GetEndTime() < currentTime) {
1.17        SLOG("video %" PRId64 " is too late (current=%" PRId64 ")",
1.18             aVideo->GetEndTime().ToMicroseconds(), currentTime.ToMicroseconds());
1.19        mRequestNextVideoKeyFrame = true;
1.20      } else {
1.21        mRequestNextVideoKeyFrame = false;
1.22      }
1.23      mMaster->PushVideo(aVideo);
```

```
1.1  --- a/dom/media/MediaDecoderStateMachine.cpp
1.2  +++ b/dom/media/MediaDecoderStateMachine.cpp
1.3  @@ -587,25 +587,36 @@ class MediaDecoderStateMachine::Decoding
1.4
1.5     void HandleAudioDecoded(AudioData* aAudio) override {
1.6       mMaster->PushAudio(aAudio);
1.7       DispatchDecodeTasksIfNeeded();
1.8       MaybeStopPrerolling();
1.9     }
1.10
1.11    void HandleVideoDecoded(VideoData* aVideo, TimeStamp aDecodeStart) override {
1.12 +    const auto currentTime = mMaster->mMediaSink->IsStarted()
1.13 +                               ? mMaster->GetClock()
1.14 +                               : mMaster->GetMediaTime();
1.15 +    if (aVideo->GetEndTime() < currentTime) {
1.16 +      SLOG("video %" PRId64 " is too late (current=%" PRId64 ")",
1.17 +           aVideo->GetEndTime().ToMicroseconds(), currentTime.ToMicroseconds());
1.18 +      mRequestNextVideoKeyFrame = true;
1.19 +    } else {
1.20 +      mRequestNextVideoKeyFrame = false;
1.21 +    }
1.22      mMaster->PushVideo(aVideo);
1.23      DispatchDecodeTasksIfNeeded();
1.24      MaybeStopPrerolling();
1.25    }
```

Figure 2.1: Example application of the SZZ algorithm in Mozilla’s Firefox repository (Bug 1717171). Left: complete bug-fixing code change. Right: part of bug-introducing code change.

2.5.1 Limitations

Unfortunately, the core idea of the algorithm does not hold for all software defects and the procedure suffers from many limitations [CMS+16, RRGB18]. First, the mapping from bug reports to commits may be wrong. For instance, the reference to the bug report

may be missing in the commit message of the bug-fixing code change resulting in a false negative, or the bug report may not describe a real performance problem resulting in a false positive. These limitations can be largely avoided by carefully following the commit message and bug-tracking guidelines.

Secondly, tracing back the modified lines of a bug-fixing commit may not lead to only true bug-introducing commits. For example, modified lines in the fix could be just cosmetic changes, like variable renaming or indentation, or they could change the surrounding context of a regression and not the problematic lines themselves. Furthermore, the last commit to change a line may not actually be the one introducing the problem or there could be a change in a file which is not executed anymore but also traced back. All these examples mislead the SZZ algorithm and result in false positives and false negatives which are difficult to prevent. These limitations occur particularly often if the bug-fix modifies a lot of lines.

CHAPTER 3

Related Work

Performance is one of the most influential non-functional requirements of software products and research has addressed the importance of integrating quantitative validation in the software development process to meet performance requirements [BMI04]. Such requirements may be the throughput of databases [DCS17], system-level performance of embedded systems [CP03], or the economical use of resources in client-server architectures and computer networks [HHK02].

This work focuses on application performance, specifically the performance of the Firefox web browser. It is closely related to the scientific fields of model-based performance prediction and software defect prediction. In this chapter, we give a brief description of these fields and name relevant publications. Lastly, we list research work on software quality control based on the Firefox project.

3.1 Model-Based Performance Prediction

Software performance prediction can be defined as the the process of predicting and evaluating, based on performance models, whether the software system satisfies the user performance goals [BMI04]. In this work we do not directly predict the performance of a system, but try to indirectly predict whether a source code change degrades the performance. Nevertheless, there are a lot of parallels to the field of model-based performance prediction, which is why we give a brief review in the following.

Performance models can be divided into two categories: classical models and machine learning models. Classical models, also referred to as analytical models, are constructed by making mathematically formalised assumptions about the software system and crafting a model based on these assumptions. The most used performance models are queueing networks [DCS17], stochastic Petri nets [KP00], stochastic process algebra [HHK02], and

simulation models [AS00]. More approaches can be found in the survey of Balsamo et. al. [BMI04].

More recent approaches belong to the second category, where machine learning and other statistical modeling techniques are employed to predict system performance. These models do not require assumptions about the software system and therefore, no deep knowledge about the system is necessary. Examples of machine learning performance models can be as simple as linear regression models [SHN+15], but also sophisticated deep learning models [HZ19].

Note that there are also approaches which cannot be assigned to only one category as they combine both analytical models and machine learning methods [DQRT15].

3.2 Software Defect Prediction

Software defect prediction techniques aim to reduce software development cost by learning from the repository history and building models to predict whether or not changes to the project (commits, modules, files, methods, etc.) contain defects [TSR+20]. Since performance regressions are a a subset of general software defects, this work is best compared to methods in software defect prediction which leverage machine learning models to solve the prediction task within a project. It is worth mentioning that there are also approaches for cross-project defect prediction [KFM+16] as opposed to within project prediction.

Typically the predictions are based on software metrics [RHTŽ13]. There are source code complexity metrics, like number of lines or McCabe's cyclomatic complexity [McC76], object-oriented metrics like class coupling or the MOOD metrics set [HCN98], or process metrics like developer experience [KSA+12]. There is also progress on incorporating natural language processing methods in defect prediction models. Several approaches now use source code and commit message tokenization to compute bag-of-words features [SWAK12, BLN+22]. One publication even leverages deep belief networks to learn semantic source code features [WLNT18]. In the future, we might also see the power of large transformer-based models like CodeBERT [FGT+20] or PLBART [ACRC21] be successfully employed in defect prediction.

The range of machine learning methods in software defect prediction is also large. We have logistic regression [KSA+12, CZ11, ACDG07], decision trees [CZ11, ACDG07], random forests [NHL18, CZ11], naive Bayes [CZ11, SWAK12], support vector machines [SWAK12, ACDG07], or convolutional neural nets [LHZL17], to name a few.

Lastly, many approaches in the software defect prediction field rely on the automated labeling provided by the SZZ algorithm [SZZ05, KZP+06, SWAK12, KSA+12, TTDM15, WLNT18, ACDG07]. However, as discussed in Section 2.5.1, the SZZ algorithm has many limitations and in general there seems to be a lack of reproducibility for approaches using this labeling method [RRGB18]. This makes a direct comparison of methods and classification results difficult.

3.3 Software Engineering Research and Mozilla Firefox

The open availability of both the source code repository and bug-tracking-system makes the Firefox project attractive for research in software engineering. In this section we briefly describe a selection of publications working with data from Mozilla.

In 2011, Zaman et. al. [ZAH11] conducted a case study on Firefox to answer how different types of bugs differ from each other. By comparing security and performance bugs they found that performance bugs take more time to fix, are fixed by more experienced developers and require larger code changes. In 2012, they followed up with a qualitative study on performance bugs, where they concluded that developers face problems in reproducing performance bugs, have to spend more time discussing them, and that performance regressions are often tolerated as a trade-off to improve something else [ZAH12].

In 2015, An et. al [AKG18] built predictive models to help software developers detect crash-prone code. Their models achieve a remarkable precision of 61.4% and recall of 95.0%. Additionally, they found that developers with less experience are more likely to submit a crash-prone code change and that such changes are often larger in terms of added and deleted lines.

Similarly, Chowdhury et. al. [CZ11] trained machine learning models to detect vulnerability-prone files in 2010. Vulnerability in this context refers to weaknesses in the software that may be exploited to cause a security failure. Their models were able to correctly predict almost 75% of vulnerability-prone files with a false positive rate of below 30%.

There are also publications which do not focus exclusively on the Firefox project, but rather use it among other repositories to test their approach [KSA+12, SWAK12, KFM+16]. Again, a direct comparison to these results is not possible because the data mining process (extracting commits, labeling, computation of features) is not reproducible and because of different experiment setups.

Lastly, Mozilla developed a platform for machine learning projects on software engineering for Firefox [Moz22a]. For example, they built a model which helps triaging bugs by automatically assigning a product and component for each new bug [CL19]. Furthermore, they successfully use a model to figure out which of the around 85 000 unique test should be run for a single push. Reducing the number of executed test by 70% compared to previous heuristics makes continuous integration at the scale of Firefox possible [HC20].

CHAPTER 4

Methodology

4.1 Experiment Setup

4.1.1 Prediction Performance Measures

To evaluate and compare the various machine learning methods we use five common classification metrics: precision, recall, F1-score, average precision, and area-under-the-ROC-curve [HM15]. All of these metrics are derived from the following values:

- True Positive (TP): Performance regression correctly identified.
- False Positive (FP): Performance regression predicted when there is no regression.
- True Negative (TN): Non-regression correctly identified.
- False Negative (FN): Non-regression predicted when there is a performance regression.

Precision measures the efficiency of a prediction and is given by

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}.$$

A high precision means that the model produces few false positives in comparison to true positives. This is desirable because a lot of false positives would waste time of developers to check code changes which are not the cause of performance regressions. This is especially important if the model is employed as a background monitoring tool. When a lot of false alarms are given, then the predictions would not be trusted and most likely ignored.

Recall measures the effectiveness of a model and is given by

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} = \frac{\text{TP}}{\text{P}}.$$

A high recall indicates that the model is able to correctly identify most code changes which caused a performance regression if it is presented one. If a developer is tasked with finding the root cause of a detected performance regression and employs the model to filter for candidate code changes, it is more important to not miss any culprits than to get a few false positives. Thus, for this use case a model with high recall is desirable.

A good precision or recall alone are not meaningful as we can artificially get a perfect recall by exclusively predicting "positive" and a perfect precision by only predicting "positive" for a single data point where we are really sure about its label. Ideally, a model would maximise both metrics. We can assess the overall prediction capabilities with the F1-score given by

$$\text{F1} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}},$$

which is the harmonic mean of precision and recall.

Many machine learning models have the capability to predict at different confidence levels, also called thresholds. For example, if we predict a regression only when we are really certain about it, we have a higher precision, but we will miss many regressions and have lower recall. The reverse is true if we predict at a low confidence level. Again, a perfect recall at confidence level 0 is always possible. This allows a trade-off between precision and recall, and consequently leads to different F1-scores at different confidence thresholds. To assess the overall performance of models and to select the best ones, we compare them by their average precision (AP),

$$\text{average precision} = \sum_{t \in \text{thresholds}} (R_t - R_{t-1}) \cdot P_t,$$

an aggregate, where the precision at a threshold, P_t, is weighted by the gain in recall at the threshold, $R_t - R_{t-1}$.

In addition to the average precision, we report a second commonly used aggregate metric called area-under-the-ROC-curve (AUC). This curve is given by

$$\text{ROC-curve} = \{(\text{FPR}_t, \text{TPR}_t) : t \in \text{thresholds}\},$$

where $\text{FPR}_t = \text{FP}_t/\text{P}$ and $\text{TPR}_t = \text{TP}_t/\text{P}$ are the true positive and false positive rates at threshold t, respectively. As the name suggests the area under this curve is computed, where a value of 0.5 would be random guessing and a value of 1.0 would be a perfect prediction score at every threshold.

4.1.2 Time Sensitive Evaluation

In the design of the experiments, particular care has to be given to the correct estimation of the prediction capabilities of the models on unseen data. The standard cross-validation approach with shuffled data [BB12], commonly used in machine learning, is not suitable for software defect prediction, as argued by Lemaître et al. [TTDM15]. In spite of that, this approach is employed in the field [KZWG11, SWAK12] and shuffling of the data is even implicitly recommended in a popular survey of model validation techniques for defect prediction models [TMHM16].

This approach is wrong, because software repositories are evolving over time and randomly shuffling the data points, as it is standard in cross-validation, would induce a bias in our performance estimate by allowing the model to make predictions on past events with information from the future. For example, if we count the number of defects per file, the model could infer that a file is particularly susceptible for problems before the regressions even occurred in the project. The correct way to split the data for evaluation is by date [TTDM15].

Therefore, in this work we split the data by date and use 90% of the commits for training the models and 10% for testing them. Furthermore, the training set is further split multiple times into two sets - one used for fitting the model and one for testing the model on future commits. We refer to the second set as validation set and choose its size to match the test set. In total, we split the training data five times and the performance is averaged over the evaluation runs to determine the best hyper-parameters for each model. Lastly, the final comparison is done on the test set from which no information was used in the process of training and tuning the models. This experiment setup is similar to cross-validation but with the difference that the validation sets are subsets of each other and that predictions are only made into the future. This form of cross-validation is often used for the evaluation of time series predictors [BB12]. For an overview of the setup see Figure 4.1.

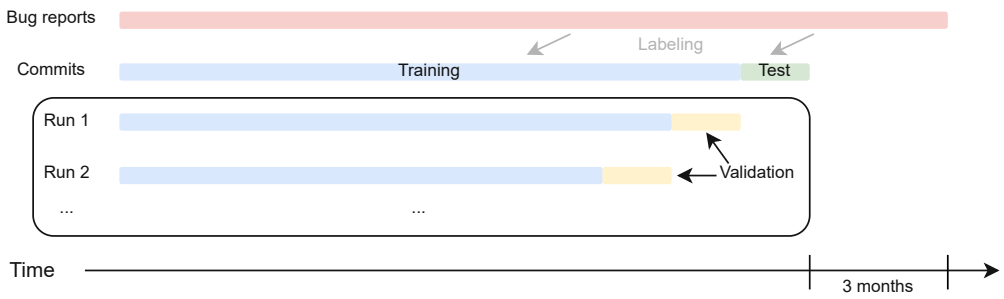


Figure 4.1: Overview of time-dependent data split in the experiments.

Furthermore, we have to keep in mind that it takes a bit of time to find and fix defects in a software project. This is important when we extract data from projects which are still under development. Because of this delay, there may be yet undiscovered bug-introducing

code changes in the most recent commits. As we precisely use the most recent commits to test the models, there would be a bias in the evaluation. To avoid this dilemma, commits younger than three months are completely discarded from training and testing. As performance regressions are critical bugs and usually fixed timely, three months leaves a large enough gap to assume that the vast majority of bugs are identified and fixed in this duration. Note that we do not discard any bug reports and use all of them to identify bug-introducing commits which are older than three months.

4.2 Feature Engineering for Software Projects

4.2.1 Traditional Feature Engineering

Recall that in this work, we want to make predictions at commit or code edit time to be able to give interactive feedback to developers. In this section we present our selection of metrics that are fast to compute and hopefully provide a good signal for performance regressions. These features are inspired by previous defect prediction models, e.g. Kamei et. al. [KSA+12], and a case study on performance bugs in the Firefox repository by Zaman et. al. [ZAH11], who found that performance bugs take more time to fix, are fixed by more experienced developers and require larger code changes.

In Table 4.1, you can see a description of all process metrics extracted from single commits. Since developers often modify multiple files in single commits, metrics that are calculated for each file are aggregated by calculating the minimum, maximum, average, and sum of the measures to get summary statistics. It is important to note that when calculating the features we only use information available at the time the code change was committed. For example, the number of unique changes to a file are only counted up to the commit for which we extract the features. This ensures that we do not introduce any bias by using information from the future we would not have at prediction time.

In addition to the process metrics, we also compute source code complexity metrics at the file level with the open-source tool rust-code analysis [ABC+20]. A commit with a higher associated complexity is more likely to cause regressions. An overview of the complexity metrics is given in Table 4.2. Again we aggregate the statistics over all files of a commit. Furthermore, we also compute the difference of the complexity metrics of a file before and after the code change to measure the complexity of a commit. We call these features "delta complexity metrics". The (file) complexity metrics capture the complexity of the files that are worked on, while the delta complexity metrics capture the added or removed complexity by the commit.

After computing the feature matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$, where n is the number of samples and d is the number of features, we apply a min-max scaling given by

$$\frac{\mathbf{X}_{ij} - \min_{i \in \text{train}} \mathbf{X}_{ij}}{\max_{i \in \text{train}} \mathbf{X}_{ij} - \min_{i \in \text{train}} \mathbf{X}_{ij}} \in [0, 1],$$

to map all features to the same value range.

Definition	Description and Justification
Number of modified files	Commits with many modified files are more likely to cause regressions.
Number of modified directories	Directories are defined as folders containing the files touched in the commit. Again, a high number indicates an error prone commit.
Number of modified subsystems	Subsystems are defined as root folders containing the files touched in the commit. Again, a high number indicates an error prone commit.
Entropy of all modified lines across files	Measures the distribution of the code changes across the modified files. High entropy indicates that the change is scattered across many files and thus more error prone.
Lines of code added	Total number of lines added in the commit diff. A higher number indicates a more complicated change.
Lines of code deleted	Total number of lines deleted in the commit diff. A higher number indicates a more complicated change.
Lines of code modified	Sum of lines of code added and deleted.
Number of developers per file	Files modified by many developers are more error prone.
Number of commits since file was last touched	If the last change of a file is a long time ago, a developer is more likely to forget all details and introduce an error.
Number of unique changes to file	If the number of changes is high, the developers have to keep track of many versions of the files and are more likely to introduce errors.
(Recent) developer experience (overall, directory, subsystem)	Less experienced developers are more likely to introduce an error. To calculate the recent experience the commits by the developer in the last three months are counted.
Developer seniority in repository	Number of commits since first commit of developer. A high number indicates an experienced developer.
(Recent) number of reverted changes (*backouts*) from developer	If a lot of code changes of a developer have to be undone, their code changes are more likely to be faulty.
(Recent) number of reverted changes (*backouts*) in subsystem/directory	Many code changes in a particular subsystem/directory that have to be undone indicate high code complexity.
Comment length	A high number of words in a commit message indicate a complex code change, which may be error prone.

Table 4.1: Summary of implemented process metrics, majority of which where adapted from Kamei et. al. [KSA+12].

Definition	Description
Cyclomatic complexity	McCabe's cyclomatic complexity measures the complexity as the number of linearly independent paths through a piece of code [McC76].
Number of lines of code	Counts the total number of lines in a file, its number of logical lines (statements), number of comment lines, and number of blank lines.
Halstead measures	A collection of metrics derived from the number of operators and operands in a piece of code. These measures provide estimates for the difficulty to understand, effort and time required to implement, and number of expected bugs in the code [Hal77]
Maintainability index	Measures the maintainability of a file as described by Welker [Wel01].
Number of methods	Number of methods in a file.
Number of arguments	Number of arguments of each method in a file.
Number of exits	Number of possible exit points of each method in a file.

Table 4.2: Summary of used source code complexity metrics obtained by the rust-code-analysis tool [ABC+20].

4.2.2 Bag-Of-Words

The tokenization of commit diffs and source code in general is not straightforward. We follow the approach of Beller et al. [BLN+22].

For each diff:

1. Disregard symbols not belonging to source code files.
2. Split the strings into words (consisting of letters, numbers, and underscores) and other symbols (e.g. + or &).
3. Disregard numeric tokens (e.g. `123`) or tokens consisting only of an operator symbol (parenthesis, boolean logic operators, equal sign, etc.).
4. Split names written in camel- or snake case (e.g., `camelCase` → `camel` and `case`; and `snake_case` → `snake` and `case`).
5. Only keep tokens longer than two characters.
6. Transform tokens to lower case characters.
7. Prefix tokens with `added_`, `deleted_`, and `context_` if the token belongs to an added, deleted, or other *context* line in the diff, respectively.

The tokenization process is best illustrated with an example. The diff given below

```
 bool MediaDecoderStateMachine::HaveEnoughDecodedVideo() {
   MOZ_ASSERT(OnTaskQueue());
-  return VideoQueue().GetSize() >= GetAmpleVideoFrames() * mPlaybackRate + 1;
+  bool isVideoEnoughComparedWithAudio = true;
+  if (HasAudio()) {
+    isVideoEnoughComparedWithAudio =
+        VideoQueue().Duration() >= AudioQueue().Duration();
+  }
+  return VideoQueue().GetSize() >= GetAmpleVideoFrames() * mPlaybackRate + 1 &&
+         isVideoEnoughComparedWithAudio;
 }
```

will be turned into following tokens:

```
context_   +    bool media decoder state machine have enough decoded video
context_   +    moz assert task queue
deleted_   +    return video queue get size get ample video frames playback rate
added_     +    bool video enough compared with audio true
added_     +    has audio
added_     +    video enough compared with audio
added_     +    video queue duration audio queue duration
added_     +    return video queue get size get ample video frames playback rate
added_     +    video enough compared with audio
```

After counting the tokens for all commits, the final vocabulary is restricted to the 50 000 most frequent tokens (about 5% of all unique tokens). Finally, the token counts are transformed to the term frequency–inverse document frequency (TF-IDF) [MRS10] statistic given by

$$\text{TF-IDF} = \text{term frequency} \cdot \log\left(\frac{N}{\text{document frequency}}\right),$$

where N is the number of diffs, term frequency is the relative frequency of the token within the diff, and the document frequency is the number of diffs containing the token. With this statistic, diffs can be identified if they use some tokens more frequently then other commits, or if they use tokens which are very rare in the entire code base. Lastly, the TF-IDF vectors are divided by their L2 norm.

4.3 Tackling Imbalanced Data

Software engineers only rarely introduce performance regressions and consequently, the vast majority of code changes are not bug-introducing. In fact, only approximately $1-2\%$ of all commits in the Firefox repository will be labeled as regression inducing. This means a model predicting that all commits are not faulty would score a high accuracy, $(\text{TP} + \text{TN})/(\text{P} + \text{N})$, but would be completely useless in practice. There are a lot of methods to overcome the challenge of imbalanced data in machine learning, which have also been applied in previous research in defect prediction [TTDM15]. In this work, the following methods are tested:

1. Original data: use the imbalanced data set as is.
2. Under-sampling: randomly sample a subset of the majority class to match the size of the minority class.
3. Over-sampling: randomly sample with replacement from the minority class to match the size of the majority class.
4. SMOTE: synthetic minority over-sampling technique based on k-nearest neighbours [CBHK02].

4.4 Parameter Tuning and Model Selection

As described in Sections 4.1.1 and 4.1.2, we want to find models that score the best average precision on future data. To achieve this we employ two approaches which are described in this section.

4.4.1 Bayesian Optimisation

First, we tune the hyper-parameters of our selected models, see Section 2.3, and test the different sampling techniques presented in the previous section. For the logistic regression, support vector machine, random forest, and multi-layer perceptron, we use the implementations of the open source library scikit-learn [PVG+11]. In addition, we use the openly available XGBoost python package, which is compatible with the scikit-learn interface. Furthermore, to integrate the sampling methods in our machine learning pipeline, the open source scikit-learn extension imbalanced-learn [LNA17] is used. The compatibility with the scikit-learn interfaces makes it particularly easy to optimise the hyper-parameters and select the best sampling method with an unified approach.

We leverage a sequential model-based Bayesian optimisation algorithm from the scikit-learn extension scikit-optimize [HML+18]. The advantage of this algorithm is the efficient sampling of the parameter search space guided by a Bayesian model. Thus, we avoid an exhaustive grid search or inefficient randomised search. The complete parameter search space configuration can be seen in Table 4.3. For all models, we sample the search space 100 times.

4.4.2 Automated Machine Learning

In addition to the hyper-parameter optimisation of the selected models, we use the open source tool TPOT [OM16], a tree-based pipeline optimisation tool for automated machine learning. At its core, TPOT uses a genetic programming algorithm to find the best machine learning pipeline of scikit-learn operators. The operators in consideration are:

- Classifiers: Prediction models like logistic regression, XGBoost, etc.
- Feature preprocessors: scaling, principal-component analysis, etc.
- Feature selectors: based on variance, recursive elimination, etc.

We let the algorithm run for 100 generations with a population size of 100.

Note that due to the high dimensionality of the bag-of-words feature vector, the TPOT search as well as the multi-layer perceptron hyper-parameter search were not feasible.

Parameter	Search Space
Sampling	No sampling, random under-sampling, random over-sampling, SMOTE
Logistic Regression	
Regularisation parameter C	LogUniform($[10^{-4}, 10^{3}]$)
Penalty	L1, L2
Support Vector Machine	
Regularisation parameter C	LogUniform($[10^{-4}, 10^{3}]$)
Kernel	Linear, Radial Basis Functions, Polynomial
Polynomial Kernel Degree	3, 5
Multi-Layer Perceptron	
Regularisation parameter α	LogUniform($[10^{-4}, 10^{3}]$)
Initial learning rate	LogUniform($[10^{-4}, 10^{-1}]$)
Hidden layer size	10, 20, 50, 100, 200
Activation function	ReLU, Sigmoid, tanh
Random Forest	
Maximum depth	No bound, 3, 5, 10, 15, 20
Minimum samples to split leaf	2, 5
Number of estimators	UniformInteger($[5, 150]$)
XGBoost	
Maximum depth	UniformInteger($[3, 10]$)
Minimum child weight	1, 2, 5
Maximum delta step	0, 1
Number of estimators	UniformInteger($[5, 150]$)
Regularisation parameter γ	Uniform($[0, 1]$)

Table 4.3: Search spaces for hyper-parameter optimisation.

CHAPTER 5

Data and Labeling

5.1 Mozilla Firefox

The Mozilla Project was founded in 1998 and the first major version of the browser was released in 2002 with the goal of providing the best possible browsing experience to the widest possible set of people. In 2004, Firefox 1.0 reached over 100 million downloads in less than a year. In 2013, celebrating the 15th anniversary of the project, Firefox was brought to smartphones [Moz22b]. The browser reached its peak market share in November 2009 with almost 32%. Nowadays the desktop share hovers at 8% while the share across all platforms is about 4% [Sta22].

Mozilla Firefox is free and open-source. The mercurial[1] repository can be found at https://hg.mozilla.org/mozilla-central/. As of March 2022, the code base ranks up over 786 000 commits made by over 8 500 contributors. This amounts to almost 25 million lines of code which took an estimated 8 000 years of joint effort [Ope22].

A web browser is a performance critical software project. The developers of Firefox aim to consistently improve the performance in the following points (among others) [Smy20]:

- Startup time: how quickly the browser launches.
- Page load time: how quickly websites load.
- Responsiveness: how smooth the interaction with a website feels.
- Resource usage: how much power the browser uses.

The open availability of source code and bug reports, as well as the performance aspects of a web browser, make Firefox a perfect candidate for our research purposes.

[1] https://www.mercurial-scm.org

5.2 Mozilla's Bugzilla

In Mozilla's bug-tracking-system Bugzilla[2] all software development tasks, like the implementation of a new feature, refactoring of source code, or the fix of a defect, are tracked with their unique bug number. Almost every code change in the Firefox repository comes with a commit message that references the bug number that was worked on. This makes it particularly easy to map from bug issue to commits and vice-versa. We can query bug reports for their type (enhancement, defect, task), status (new, assigned, resolved, etc) and resolution (fixed, won't fix, invalid, etc). Additionally, the reports are tagged with keywords to be able to categorise them. Examples for keywords are "memory-leak", "crash", or "sec-critical" for a critical security bug. The keywords relevant to us are "perf", "topperf", and "perf-alert", which are all used for performance regressions.

Furthermore, in early 2019, Mozilla introduced the guideline for developers that whenever they fix a problem they should link the bug number of the code change which caused the defect in the `regressed-by` field of the issue-tracker. This provides important information about performance regression inducing code changes which we would like to predict. The fact that we have associations from bug-fixing to bug-introducing code changes labeled by software engineers, who are experts on the code base, makes this information especially valuable. But unfortunately, this information is not perfect. In general, bug numbers map to multiple commits and not all of them may have necessarily something to do with the problem. Nevertheless, in Section 5.6 we investigate if we can leverage the bug number links of the `regressed-by` field in a meaningful way.

5.3 Mozilla's Perfherder

In an effort to find and report performance regressions, Mozilla developed the interactive dashboard Perfherder[3] which allows monitoring and analysis of automated Firefox performance benchmark tests [Moz19]. Whenever Perfherder detects changes in a performance test series, it automatically sends alerts which are then further processed by the responsible software engineers. In each alert two revision hashes (commit identifiers) are given. The first one corresponds to the commit, for which the benchmark still looked good, while the second one corresponds to the commit, for which the benchmark already shows a worse score. All commits in between are candidates for the bug-introducing commit. Furthermore, if the alert is valid, a bug report is created and is given the "perf-alert" keyword. Therefore, Perfherder provides yet another source of information which we can use in the labeling process.

Since not all performance regressions are detected with Perfherder alerts and because the revision ranges of alerts are only estimated in an automated way with insufficient accuracy, we cannot use these commit intervals to directly label the code changes. However, we

[2] https://bugzilla.mozilla.org/
[3] https://treeherder.mozilla.org/perf.html

will use the information to evaluate our labeling approach based on the SZZ algorithm, see Section 5.5.

For an overview of the bug report and alert system as described see Figure 5.1.

Figure 5.1: Overview of information used in the labeling process categorised by source: mercurial repository, Bugzilla, and Perfherder. Bug numbers are referenced in the commit messages. Both the `regressed-by` field and performance alerts can be mapped to commits and can be used to evaluate the SZZ algorithm.

5.4 Data Selection and Labeling

To be able to predict which code changes are likely to induce performance regressions in the future, we first have to know which code changes have caused these kind of problems in the past. In an ideal world, developers would document in the bug-tracking-system which source code lines were the root cause of the problem once they have fixed it. However, this is not the case and we have to carefully label the code changes. The performance of a prediction model is highly dependent on the correctness of the labeling. A model may do well with respect to statistical metrics, but if the labeling is wrong its predictions may still be useless.

We present two approaches to the labeling problem. First, we examine the applicability of the SZZ algorithm - the automated labeling tool commonly used in defect prediction, see Section 2.5. Secondly, we try to leverage the `regressed_by` field of the bug-tracking-system, where developers keep track of causes of bugs, on the *bug-level*, see Section 5.2.

As we want to be able to compare these two approaches and because the `regressed_by` field was only introduced in early 2019, we only consider commits from 2019/07/01 to 2021/11/01. We choose the start date to make sure that developers had enough time to adapt the guideline and use the new bug report field. Again, we fetched bug reports up to 2022/02/01 as discussed in Section 4.1.2. After excluding merge commits, commits from side branches, and commits which could not be matched to a bug number, we are left with 99 694 commits to be labeled.

5.5 SZZ Labeling

There have been various improvements to the SZZ algorithm to mitigate the limitations discussed in Section 2.5.1. In this thesis, we use the open source implementation called SZZ-Unleashed [BSBH19] for the automated labeling process. This version of the algorithm is based on the enhancements proposed by Williams and Spacco [WS08], where line number maps are built that track unique source lines as they change over the lifetime of the software project. We modified the algorithm to only consider source code files, the most prominent file types being `.cpp`, `.js`, `.html`, `.c`, `.rs`, and `.py`. We converted the mercurial repository to a git[4] repository with fast-export [Dre21] to be able to apply the SZZ-Unleashed tool.

We select all bug reports, which are valid, have type “defect”, have resolution “fixed”, and contain at least one of the the keywords “perf”, “perf-alert”, or “topperf”. The resulting 961 performance bug reports are then mapped to the corresponding bug-fixing commits by selecting all code changes which reference one of the 961 bug numbers in the commit message. SZZ Unleashed finds 1544 bug-introducing commits from which only 589 (38%) belong to our selected commits. The vast majority of the other suspected

[4] https://git-scm.com/

commits were made before 2019/07/01. This fact already sounds like bad news and thus, a more rigorous evaluation of the results is called for.

5.5.1 Evaluation

From the `regressed_by` field and from Perfherder alerts we can reduce the candidate bug-introducing commits for certain bugs. This allows us to evaluate the SZZUnleashed results by checking whether the found commits are indeed among the candidate commits. If a bug report has a bug number in the `regressed_by` field, we define the candidate commits to be the ones which reference this number in the message. If there exists a Perfherder alert for a bug, we define the candidate commits to be the ones given by the range in which the benchmark score went bad (see Section 5.3). This evaluation approach is illustrated in Figure 5.1.

Let n be the number of *bug reports* that have either a bug number in the `regressed_by` field or a Perfherder alert. We call a bug-introducing *commit* identified by SZZUnleashed true positive (TP) if it is among the corresponding candidate commits, false positive (FP) otherwise. For this evaluation we consider all commits, not only those selected in Section 5.4. Lastly, we count the number of *bug reports* for which SZZUnleashed did not identify any bug-introducing commits. This number is a lower bound on the number of false negative (FN) *commits*, if we assume that there exists at least one bug-introducing commit for every bug report.

The results can be seen in Table 5.1. If the commits identified by SZZUnleashed are accurate, we would see at least noticeable agreement with the candidates derived by the bug report associations from developers and benchmark based alerts. Unfortunately, this is not the case: true positives are greatly outnumbered by false positives and the number of false negatives is also high. Nevertheless, in Section 6.1 we will investigate how our machine learning models perform with this labeling approach anyways.

candidate source	**n**	**TP**	**FP**	**FN**
`regressed_by` field	371	134	576	≥ 207
Perfherder alerts	192	38	691	≥ 105

Table 5.1: SZZUnleashed evaluation results.

5.6 BugBug Labeling

Disappointed by the evaluation of the SZZUnleashed results, one may ask if we can leverage the information from the `regressed_by` field or Perfherder field directly and get better results. As already mentioned in Section 5.3, the Perfherder alerts cannot be used to label the commits directly, because the given revision ranges of alerts are only estimated in an automated way with insufficient accuracy.

In contrast, the `regressed_by` field can be used directly, because the information comes from software engineers, who are experts on the code base and can be trusted to establish correct associations between bug-introducing and bug-fixing code, and because the `regressed_by` field is used consistently since 2019.

5.6.1 Labeling on the Commit-Level

The straightforward labeling process based on the `regressed_by` field consists of following steps:

1. Collect bugs from the bug-tracking-system that have a non-empty `regressed_by` field.
2. Get performance bugs by filtering for keywords “perf”, “perf-alert”, and “topperf”.
3. Collect all bug numbers from the `regressed-by` field of the selected bugs.
4. Label all commits which worked on a bug number from step 3 as performance regression inducing.
5. Label all other commits as non-regression.

Again, this labeling approach should have less noise than the one produced with the SZZ algorithm, because the bug-introducing and bug-fixing code links come directly from developers. Note that Mozilla also uses a similar approach in their own machine learning framework bugbug [Moz22a] and therefore, we call this labeling approach “bugbug”. However, the approach has one major drawback: bug numbers can be mapped to multiple (consecutive) commits. An example of this pattern in the Firefox repository can be seen in Figure 5.2. And in fact, a quick analysis shows that about 60% of all performance problems are split into more than one commit. This may not sound like a huge problem, but it makes the labeling unusable in this form as we shall see.

Figure 5.2: Example of bug being split up into multiple commits.

These bug-introducing labeled groups of commits introduce a *positional bias* which may be exploited by machine learning models. To investigate to what extent this effect takes place, we train a XGBoost model on both a randomly shuffled data split (case 1) and a time-based data split (case 2). If the model can pick up on the position of these groups, then the model will be able to predict well in case 1, because it can match randomly left out commits back to their group. On the other hand, the model will perform badly in

case 2, because it can not extrapolate from the groups to future commit groups. This line of reasoning is illustrated in Figure 5.3.

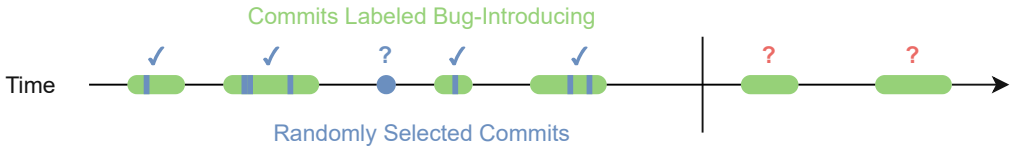


Figure 5.3: Illustration of positional bias from bug-number-based labeling.

And indeed, the model achieves a precision 0.94 and recall 0.38 in case 1 which vastly outperforms the precision 0.07 and recall 0.01 in case 2. Furthermore, the most important features are "developer seniority", "developer experience", and "number of overall backouts of developer", see Table 4.1. Looking at these features for the commit group of Figure 5.2 immediately shows how the model can exploit the positional bias, see Figure 5.4. As the features either increment by one or stay the same for all commits in the group, the model can use them as a proxy for the position in the version history.

revision hash	Commit Id	Developer Seniority	Recent Developer Experience	Recent Backouts Developer
3af530ddacc9ea83551722a8d9cb454473c0804b	574357	490884	230	27
d5eb51b405c4bdfe5d6cec7d0e7fe287929d6c32	574358	490885	231	27
56f4a105ca5299049d31091659dc861dc450027c	574359	490886	232	27
a4d18ab82feff8faea191ab5eaa82b7655ecb2d3	574360	490887	233	27
b3219fb7aa9b3c48b03ab4db71f6ce739dee8686	574361	490888	234	27

Figure 5.4: Positional bias implicitly encoded through small set of features.

Pushing this experiment to absurdity, we train the model just on the commit id, an integer enumerating all code changes. The fact that even in this ridiculous case the model is able to achieve precision 0.96 and recall 0.12 confirms that this labeling approach is not suitable for predicting performance defects on the commit level, because the model just exploits the position of the commits. It also highlights the importance of the correct evaluation setup. If we would not have used the correct time-based test split as discussed in Section 4.1.2, we might have been happy with the good classification results and called it a day.

5.6.2 Labeling on the Bug-Level

There is a way to work around the positional bias discovered in the previous section. By changing the prediction level from single code changes to commit groups as entities, we are able to still take advantage of the data manually labeled by the software engineers. In other words, we aggregate the commit groups to single data points, which completely eliminates the positional information exploited by the models.

The commits are grouped by the following rules:

- The commits have to appear consecutively in the repository.
- The commits have to reference the same bug number.
- The commits have to be created by the same author.

In essence, one commit group represents the work of a developer on a bug in one session (consecutive commits). The grouping not only eliminates the positional bias, but it also makes sense from a software engineering perspective. Bugs may be introduced not only by single commits, but also by the interaction of multiple code changes.

Finally, we adapt our labeling process:

1. Collect bugs from the bug-tracking-system that have a non-empty `regressed_by` field.
2. Get performance bugs by filtering for keywords “perf”, “perf-alert”, and “topperf”.
3. Collect all bug numbers from the `regressed-by` field of the selected bugs.
4. *Group the commits according to the rules.*
5. Label all *commit groups* which worked on a bug number from step 3 as performance regression inducing.
6. Label all other *commit groups* as non-regression.

This reduces the 99 694 commits to 67 921 groups, where 1880 (2.77%) groups are labeled as performance regression inducing. We adjust the computation of the traditional features by aggregating (mean, max, min, sum) over all commits of the group. Additionally, we introduce a feature counting the number of commits of the groups. For the bag-of-words feature vectors, we count the occurrences of tokens in the diffs of all commits of the group.

Note that the classification performance is still way better on a random data split with this labeling approach. However, the reason for this cannot be the positional bias of consecutive commits as previously. Instead, we think that as the features evolve over time (e.g. developer seniority will go up for all software engineers over time), the model may pick up on certain time frames (by remembering certain values of these evolving features) where there was an unusual high amount of performance regressions. We emphasise that this phenomenon cannot be reflected in our reported classification results, as we evaluate on a *time-dependent* data split, see Section 4.1.2.

CHAPTER 6

Results

6.1 SZZ Labeling

In Table 6.1, you can see the results of the hyper-parameter tuning for the SZZ labeling (Section 5.5) with traditional features (Section 4.2.1) in form of average precision (AP) and area-under-the-ROC curve (AUC). We report these metrics on the training set and test set, where the model is trained on the full training set. Additionally, we give the average of the metrics on the five validation splits, Section 4.1.2, which were used to select the best hyper-parameters for each model. To have a reference, we include a *dummy classifier*, which predicts that all commits are performance regressions. By definition this classifier achieves a perfect 1.0 recall, $\frac{P}{P+N}$ (average) precision, and 0.5 AUC.

	training		validation		test	
	AP	AUC	AP	AUC	AP	AUC
Dummy Classifier	0.0063	0.5000	0.0060	0.5000	0.0022	0.5000
Logistic Regression	0.0208	0.7136	0.0304	0.7033	0.0049	0.5944
Support Vector Machine	0.0525	0.8128	0.0365	0.7482	0.0052	0.6750
Random Forest	**0.9952**	**1.0000**	0.0528	0.7621	0.0054	0.6335
XGBoost	0.3865	0.9817	0.0590	0.7453	0.0069	**0.6767**
Multi-Layer Perceptron	0.0402	0.7844	0.0376	0.7502	0.0071	0.6566
TPOT	0.0916	0.8420	**0.0674**	**0.7969**	**0.0089**	0.6714

Table 6.1: Results of hyper-parameter tuning for SZZ labeling with traditional features.

The first thing to notice is that all models perform significantly better on the training set than on the test set. The overfitting is most pronounced with the tree-based models, random forest and XGBoost. Despite this, the best model found by TPOT, and best performing model on the validation splits, is a XGBoost model trained on the stacked output of a naive Bayes [Bis06, Chapter 8] and decision tree classifier after removing features of low variance.

The aggregate metrics AP and AUC alone are not that meaningful for the final classification performance of the models. Therefore, we show the ROC-curve and the precision-recall curve in Figure 6.1 to investigate the classification at different confidence thresholds. A good ROC-curve is bent to the top-left corner of the plot, while a precision-recall curve is good if it is far to the top-right of the origin (better F1 score).

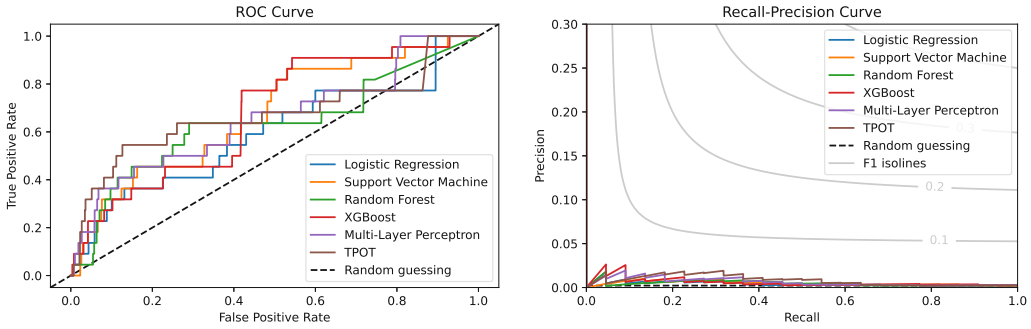


Figure 6.1: ROC and precision-recall curves for the SZZ labeling with traditional features on the test set. The y-axis for the precision-recall plot is scaled for better separation.

Even though the ROC-curves are clearly separated from the random guessing reference, the precision at all confidence thresholds is very low. Note that we cannot simple read off the best precision-recall trade-off in terms of F1-score from the plot, because this would introduce a bias by selecting the best confidence threshold based on data in the test set. Instead, we determine the best threshold on the training set and read the metrics for this threshold on the test set. In fact, for all models, except the multi-layer perceptron, perform very poorly with this threshold, see Appendix Table 8.1. However, the scores with 0.0909 recall, 0.0177 precision, and 0.0292 F1 are still very bad, even though they greatly outperform the dummy classifier with 1.0 recall, precision 0.0022, and 0.0044 F1 in terms of F1 score.

Interestingly, the simple bag-of-words approach worked better for the SZZ labeling approach. Not only are almost all hyper-parameter test scores better than with traditional features, see Table 6.2, but the ROC and precision-recall curves look better too, see Figure 6.2. Remember that due to the high dimensionality of the bag-of-words feature vector, parameter search for the multi-layer perceptron and TPOT model search were not feasible. There is also an interesting spike to 0.25 precision for the logistic regression model, which we cannot explain. But note that this precision is not achieved with the confidence threshold determined on the training set, which demonstrates the aforementioned bias when reading directly from the precision-recall curve. Furthermore, the models also all have a greater F1 score on the test set compared to before. The best model is now the support vector machine with 0.1818 recall, 0.0288 precision and 0.0497 F1, see Appendix Table 8.3. This is an almost 2× improvement over the traditional features. To investigate why the bag-of-words approach works better, we take a look at the most important tokens in Section 6.5.

	training		validation		test	
	AP	AUC	AP	AUC	AP	AUC
Dummy Classifier	0.0063	0.5000	0.0060	0.5000	0.0022	0.5000
Logistic Regression	0.3403	**0.9535**	0.0670	**0.8227**	**0.0246**	0.7134
Support Vector Machine	0.1086	0.8957	0.0563	0.8187	0.0172	**0.7154**
Random Forest	**0.7216**	0.9504	0.0627	0.7937	0.0058	0.6689
XGBoost	0.3491	0.8802	**0.0765**	0.8013	0.0077	0.6947

Table 6.2: Results of hyper-parameter tuning for SZZ labeling with bag-of-words features.

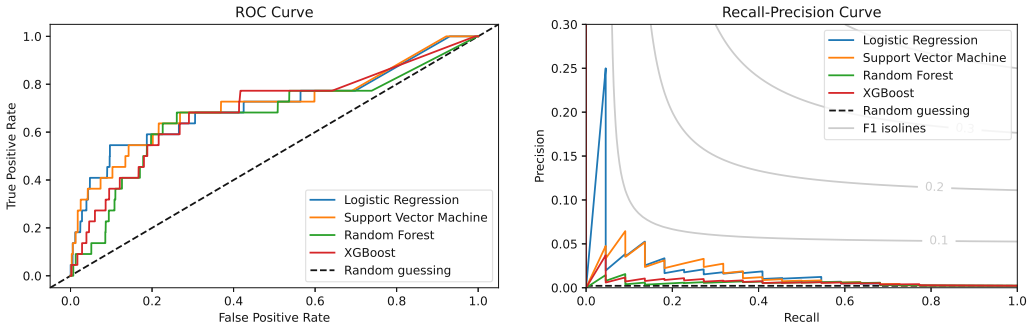


Figure 6.2: ROC and precision-recall curves for the SZZ labeling with bag-of-words features on the test set. The y-axis for the precision-recall plot is scaled for better separation.

In summary, even though the best model outperforms the dummy classifier and random guessing by more than 10× in terms of precision and F1, it is safe to say that this model is far from useful in application. Interestingly, the relative frequency of performance regressions shrinks from 0.6% in training and validations sets to 0.2% in the test set. This fact, together with the bad prediction scores and our estimates of high false positive and false negative rates in the evaluation of the SZZ labeling, see Section 5.5.1, brings us to the conclusion that this approach fails in predicting performance regressions.

6.2 BugBug Labeling

In this section we report the results for the bugbug labeling approach, see Section 5.6. Remember that this labeling is now based on groups of commits rather than single commits like the SZZ labeling. Again, we start by looking at the average precision (AP) area-under-the-ROC-curve (AUC), see Table 6.3. While the logistic regression and support vector machine models perform fairly equally on training, validation, and test sets, the tree-based models still severely overfit on the training data. The best model found with the automated machine learning tool TPOT is an extra trees classifier, a model similar to random forests [GEW06], again with feature selection based on variance. It achieves a near perfect score on the training set and the best average precision on the test set, albeit far from a good score.

	training		validation		test	
	AP	AUC	AP	AUC	AP	AUC
Dummy Classifier	0.0268	0.5000	0.0297	0.5000	0.0355	0.5000
Logistic Regression	0.0751	0.6966	0.0767	0.6375	0.0929	**0.7015**
Support Vector Machine	0.0853	0.7116	0.0804	0.6419	0.0929	0.6992
Random Forest	0.1844	0.7981	0.1055	0.6804	0.0972	0.6814
XGBoost	0.6059	0.9208	0.1088	0.6842	0.0860	0.6481
Multi-Layer Perceptron	0.1224	0.7568	0.0870	0.6404	0.0830	0.6934
TPOT	**0.9640**	**0.9985**	**0.1163**	**0.6881**	**0.1131**	0.6875

Table 6.3: Results of hyper-parameter tuning for bugbug labeling with traditional features.

Surprisingly, the good average precision and precision-recall curve, see Figure 6.3, of the extra trees model do not carry over to a good F1 score based on the confidence threshold determined on the training set. In fact, the logistic regression model performs best in this regard with 0.2946 recall, 0.1084 precision, and 0.1585 F1. However, due to the increased relative frequency of performance regression inducing commit groups of 2.77%, this is now only about two times better than the dummy classifier in terms of F1, but more than five times better in terms of precision. For the classification results of all models see Appendix Table 8.4.

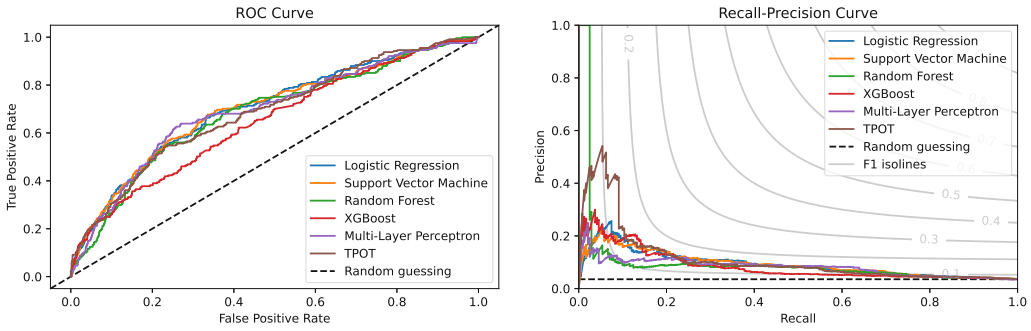


Figure 6.3: ROC and precision-recall curves for the bugbug labeling with traditional features on the test set.

The results for the bag-of-words models look really similar to the models based on traditional features but are slightly better, see Table 6.4 and Figure 6.4. While the random forest model dominates in terms of the aggregate metrics AP and AUC, this performance vanishes when evaluating at the best confidence threshold. In this case, it was again the logistic regression model which performed best with 0.1535 recall, 0.2022 precision, 0.1745 F1, beating the logistic regression model based on the traditional features by a small margin. Again, we investigate how the simple bag-of-words features can seemingly encode the same information as carefully handcrafted source code metrics in Section 6.5.

	training		validation		test	
	AP	AUC	AP	AUC	AP	AUC
Dummy Classifier	0.0268	0.5000	0.0297	0.5000	0.0355	0.5000
Logistic Regression	0.4549	0.8713	0.1249	0.7210	0.0958	0.6420
Support Vector Machine	0.2246	0.7901	0.1181	0.7099	0.0919	0.6351
Random Forest	**0.7485**	**0.9492**	0.1343	**0.7233**	**0.1072**	**0.6839**
XGBoost	0.5933	0.8530	**0.1358**	0.6795	0.0865	0.6441

Table 6.4: Results of hyper-parameter tuning for bugbug labeling with bag-of-words features.

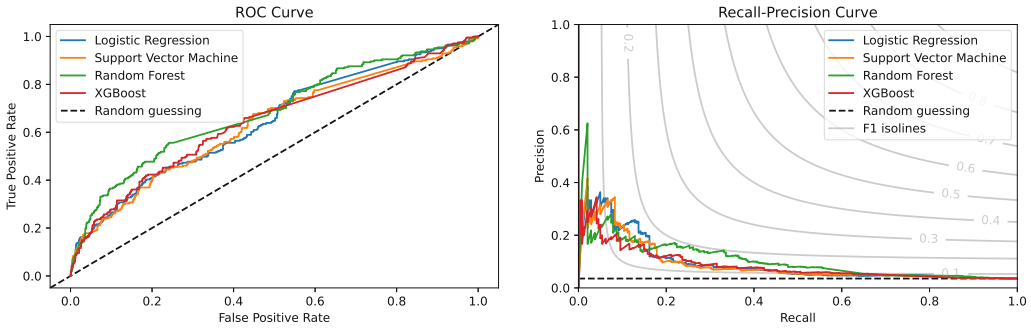


Figure 6.4: ROC and precision-recall curves for the bugbug labeling with bag-of-words features on the test set.

Even though the prediction capabilities of the models are still far from usable, we are happy to see that our approach to label based on coherent commit groups rather than single commits is better suited than the labeling of the commonly used SZZ algorithm and leads to a 3.5× improvement in terms of F1 score. We think there are several reasons for this success: First, the main signal for linking bug-introducing code to bug-fixing code comes directly from the Firefox software engineers. Second, the key assumptions of the SZZ algorithm that bugs are fixed at the same code lines they were introduced at seems to be wrong for performance regressions. Third, performance bugs may be introduced by the interplay of code changes of several commits, which may be captured by the representation of code changes in terms of our grouping method.

6.3 Improving the Models with Feature Selection

Previous research in software defect prediction has shown that removing unimportant features can significantly improve the performance of the models [SWAK12]. We find the most important features by looking at summary plots of shapley values. A shapley value $\phi_j(x)$ is the value of the j-th feature contribution to the prediction of the instance x compared to the average prediction for the data set, see Section 2.4. Unfortunately, due to the high dimensionalty of our feature matrices, it was not feasible to compute the shapley values for every model type. Therefore, we use the best logistic regression

models for all plots. In Figure 6.5 you can see the summary plots for the traditional features for both the SZZ and bugbug labeling. For each sample in the test set x, a point is drawn for each feature with its value color coded (high value: red, low value: blue) and with a horizontal offset according to the corresponding shapley value. Points far to the right or left mean that there is sample for which the feature has a particularly strong importance towards predicting regression or non-regression, respectively.

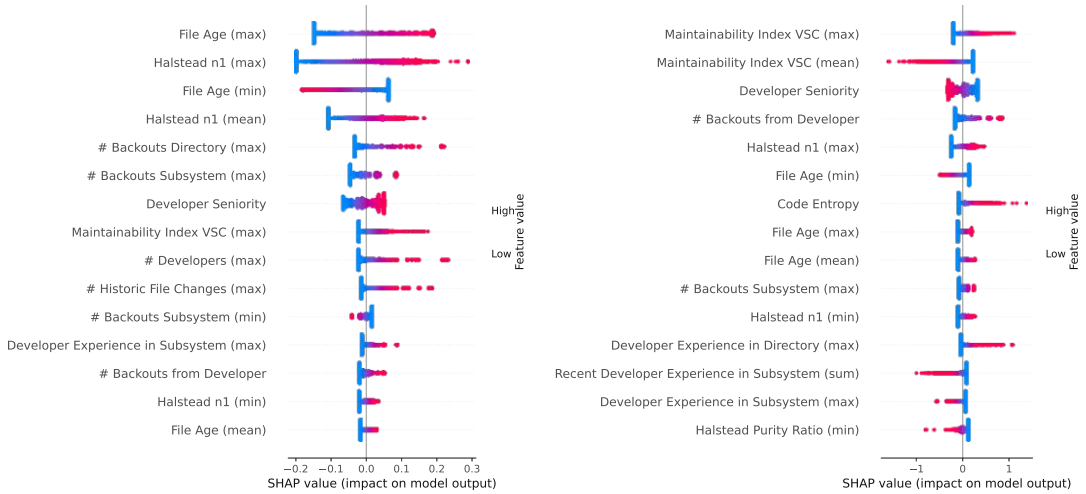


Figure 6.5: SHAP summary plots for traditional features. Left: SZZ labeling; right: bugbug labeling.

We immediately see that there are contradictions in the feature importance. Recall that we compute several metrics for each file in a commit (group) and then calculate minimum, maximum, mean, and sum, to get summary statistics, see Section 4.2.1. For example, this is done for the feature "File Age" and "Maintainability Index". We see that once each of the features has a positive impact on the prediction and once a negative impact depending on the aggregation method. The reason may be that these features correlate strongly which the model exploits. To avoid this, we select the mean as only aggregation method. There are also no "delta complexity metrics" among the most important features. This tells us that the change in complexity introduced by a commit is not as important as the overall complexity of the files that were worked on. Thus, we also remove the delta complexity metrics from the feature matrix. For the updated shapley value summary plots refer to Section 6.5.

The hyper-parameter tuning results with the reduced features can be seen in Tables 6.5 and 6.6. Compared to Tables 6.1 and 6.3, we can see similar validation scores. The best average precision for the SZZ labeling increased two-fold on the test set and decreased slightly for the bugbug labeling. The best TPOT models are now a combination of multi-layer perceptron and XGBoost for the SZZ labeling and a combination of naive Bayes and extra trees classifier for the bugbug labeling. Interestingly, TPOT now omits a feature selection step with the reduced features unlike before.

	training		validation		test	
	AP	AUC	AP	AUC	AP	AUC
Dummy Classifier	0.0063	0.5000	0.0060	0.5000	0.0022	0.5000
Logistic Regression	0.0196	0.7377	0.0278	0.7365	0.0066	0.6233
Support Vector Machine	0.0318	0.8001	0.0362	0.7753	0.0047	0.6731
Random Forest	**0.9918**	**1.0000**	0.0482	0.7652	0.0059	0.6715
XGBoost	0.1584	0.9513	0.0546	0.7772	0.0039	0.6624
Multi-Layer Perceptron	0.0469	0.8287	0.0368	0.7710	0.0035	0.6614
TPOT	0.3281	0.9408	**0.0687**	**0.7922**	**0.0172**	**0.7076**

Table 6.5: Results of hyper-parameter tuning for SZZ labeling with traditional features after feature reduction.

	training		validation		test	
	AP	AUC	AP	AUC	AP	AUC
Dummy Classifier	0.0268	0.5000	0.0297	0.5000	0.0355	0.5000
Logistic Regression	0.0707	0.6940	0.0747	0.6403	0.0921	**0.7027**
Support Vector Machine	0.1233	0.7655	0.0880	0.6602	0.0873	0.6974
Random Forest	0.1798	0.7856	0.1044	0.6754	0.0934	0.6717
XGBoost	0.5541	0.9136	0.1037	0.6654	0.0814	0.6590
Multi-Layer Perceptron	0.2250	0.8584	0.1040	0.6560	0.0848	0.6693
TPOT	**0.9877**	**0.9997**	**0.1228**	**0.6913**	**0.0942**	0.6756

Table 6.6: Results of hyper-parameter tuning for bugbug labeling with traditional features after feature reduction.

The biggest improvement of the models can be seen in terms of F1-score. In Table 6.7, you can see the best models for each labeling and feature type configuration. With the feature reduction the performance gap to the bag-of-words models has been closed. Still, the models based on the bugbug labeling outperform the models based on the SZZ labeling 3× in terms of F1 score. We do not use further feature selection for the bag-of-words model, because we already select only 5% of all unique tokens, see Section 4.2.2.

Labeling	**Feature Type**	**Model**	**Sampling**	**Recall**	**Precision**	**F1**
SZZ	traditional	Multi-Layer Perceptron	No sampling	0.0909	0.0177	0.0296
	traditional + reduction	TPOT (MLP + XGBoost)	No sampling	0.1364	0.0326	0.0526
	bag-of-words	Support Vector Machine	Over-Sampling	0.1818	0.0288	0.0497
bugbug	traditional	Logistic Regression	SMOTE	**0.2946**	0.1084	0.1585
	traditional + reduction	Logistic Regression	SMOTE	0.2324	0.1383	0.1734
	bag-of-words	Logistic Regression	No sampling	0.1535	**0.2022**	**0.1745**

Table 6.7: Comparison of best models for each labeling and feature type configuration.

Furthermore, from Table 6.7 and from the hyper-parameter search results in Tables 6.1-6.6, we can see the general trend that the linear models logistic regression and support

vector machine, as well as the multi-layer perceptron are the best at generalising to future data in the sense that their classification scores for the validation and test set are often close to the scores on the training data set. Even though the scores are still higher on the training set for these models, the tree-based models random forest, XGBoost, and extra trees, show much more overfitting with sometimes near perfect scores on the training set. For the classification results in more detail see Appendix Sections 8.1 and 8.2. The best hyper-parameters for all models can be found in Appendix Section 8.4.

6.4 Best Sampling Method

Until now we did not investigate the sampling techniques used in the machine learning pipelines for the best models. In Table 6.7, you can see that for the best models all sampling techniques except random under-sampling are present. However, this could just be a coincidence and we will take a more quantitative look into finding the best sampling method.

For each labeling, feature type (only reduced traditional features), and model, we select the best validation score for each of the four sampling methods (Section 4.3) out of the 100 investigated points of the hyper-parameter search. We then rank the sampling techniques from 1 to 4 based on their best validation score and average the ranks over all labeling, feature type, and model configurations (=18 combinations).

Sampling Method	Average Rank
Random Over-Sampling	1.9444
No sampling	2.2778
Random Under-Sampling	2.7222
SMOTE	3.0556

Table 6.8: Average ranks of sampling methods taken over all labeling, feature type, and model configurations.

From Table 6.8, it seems that the best way to tackle the problem of highly imbalanced data in performance regression prediction is to either use over-sampling or to not tackle it at all. It makes sense that these two methods are better than under-sampling, because they do not throw away any data. We believe that no sampling is also at the top, because it allows the models to learn the real class distribution, which may have an effect when selecting the best confidence threshold for prediction. The synthetic data generation of SMOTE seems to fail in general, but is surprisingly effective for the bugbug labeling with traditional features.

For the best sampling technique of all labeling, feature type, and model configurations see Appendix Table 8.14. Note that our result that sometimes no sampling is a viable option and sometimes a sampling technique is better for defect prediction is in agreement with the results of Tan et. al. [TTDM15].

6.5 Interpreting the Models

As we have discussed in Section 2.4, having a good performance regression prediction model is meaningless if we cannot communicate to the software engineer why the model thinks a code change is suspicious. Disappointingly, our model results presented in Section 6.1and 6.2 are not nearly as good as would be needed for a deployment in the software development process. Nevertheless, it is still interesting to investigate what the models learned to make their predictions, most importantly to answer the question why the simple bag-of-words approach works so well.

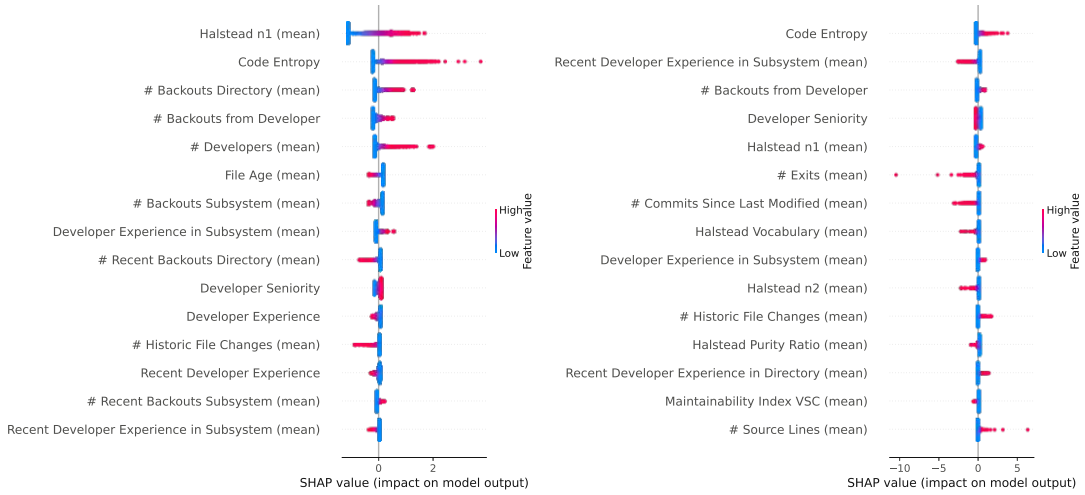


Figure 6.6: SHAP summary plots for reduced traditional features. Left: SZZ labeling; right: bugbug labeling.

First, in Figure 6.6 you can see the SHAP summary plots for the traditional features for both the SZZ and bugbug labeling. Recall that points far to the right or left mean that there is sample for which the feature has a particularly strong importance towards predicting regression or non-regression, respectively. For example, we can see that if the code entropy (distribution of source code among files) is high that the models will predict on average towards regression, while a low number of operators in the source code (Halstead n1) will bias the prediction towards non-regression.

While some interpretations make sense to us, like a high number of developers working on the same file is bad (# Developers for SZZ labeling), others are beyond explanation, like a high developer experience in the subsystem is bad while a high experience in the directory is good (bugbug labeling). We have fixed the contradictions concerning the different interpretation of metrics based on the aggregation method with feature selection, see Section 6.3, but there are still contradictions regarding the number of backouts and experience features as described before. Furthermore, there are disagreements between the labelings: for the SZZ labeling high developer seniority is good while for the bugbug labeling high seniority is bad.

In summary, the models generally seem to use the features as they were intended, see Section 4.2, however, there are many contradictions and counter-intuitive features, such that we cannot conclude that the model learned a meaningful classification.

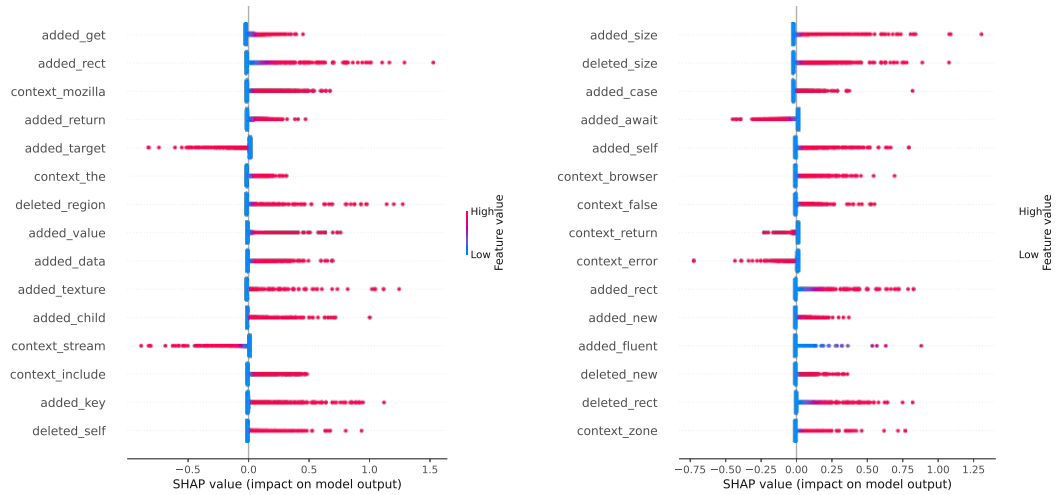


Figure 6.7: SHAP summary plots for bag-of-words features. Left: SZZ labeling; right: bugbug labeling.

Now we turn our attention to the most important bag-of-words tokens for prediction, see Figure 6.7. In this case a red point means that token is present in the commit code, and a blue point means that the token is missing. It is immediately clear that it is more important which tokens are present rather than missing for prediction. Also there are more tokens which bias the prediction towards regression than towards non-regression. There are mostly tokens from added and context lines, rather than from deleted lines. However, the tokens themselves are rather generic. It would be ridiculous to tell a developer that he should not use the `return` keyword, because it causes performance regressions. Furthermore, it is rather weird that the `await` keyword in particular indicates non-regression, because we associate this keyword with asynchronous programs, which we would think are more susceptible to performance problems.

Just from the summary plots we cannot explain why the bag-of-words model can match the performance of the handcrafted features. For this reason, we will now take a look at a specific example and investigate the prediction of the models in more detail.

We consider bug 1718755, which fixed a problem regarding the customisation of the graphical user interface, but caused a 20% degradation on a benchmark concerning load times on the Wikipedia web page. It was fixed in a single commit with bug number 1722487 and revision hash 5235f051d952. For the sake of brevity we now will refer to the models as traditional/bag-of-words SZZ/bugbug model, depending on the feature type and labeling they are based on.

This bug is interesting for many reasons:

- SZZUnleashed was able to correctly find the bug-introducing commit (revision hash 9ac290ec5884).
- The commit belongs to the test set (was not involved in the training of the models).
- The traditional SZZ model was not able to correctly predict regression, while the bag-of-words model was able to correctly predict regression.
- There are four commits working on bug 1718755 (9ac290ec5884 among them), which are grouped together in our bugbug labeling. Both the traditional and bag-of-words bugbug models correctly predict regression.

Therefore, we hope that this data point gives insights into why the bag-of-words approach works better, especially in the SZZ labeling case, and why the bugbug models outperform the SZZ models.

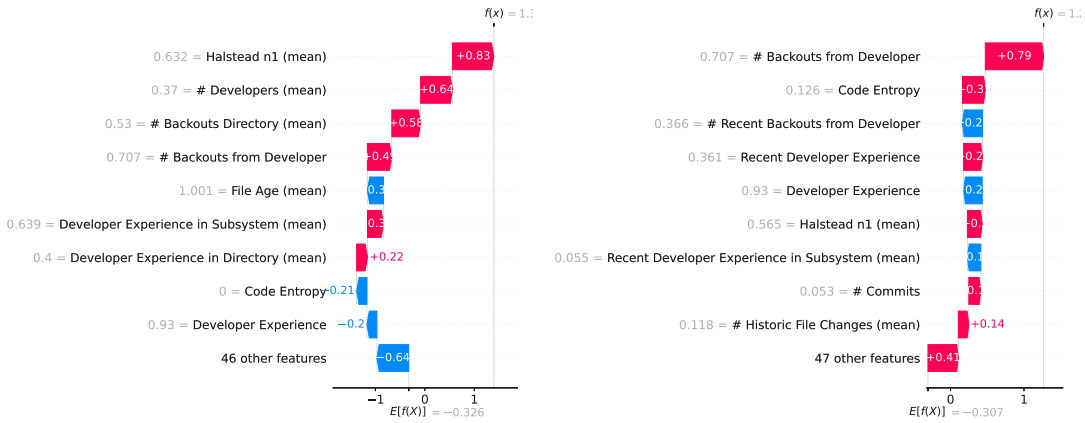


Figure 6.8: SHAP waterfall plots for traditional features. Left: commit for revision 9ac290ec5884; right: commit group for bug 1718755.

We take a look at so called SHAP waterfall plots for both the traditional and bag-of-words features, see Figure 6.8 and 6.9, respectively. These plots show which features pushed the predictions towards regression (to the right and red) or non-regression (to the left and blue) *for a particular data point*. The colors no longer encode the feature values, however, their numeric value is given. From the plots we can see that for the SZZ model the prediction was pushed towards regression but not far enough. On the other hand, for the bugbug model the most important feature is the number of backouts from the developer, which caused the model to predict regression in the end. Many developer experience features move the prediction in different directions, which is hard to understand. It makes no sense to us why the experience should have a positive *and* negative influence on the prediction.

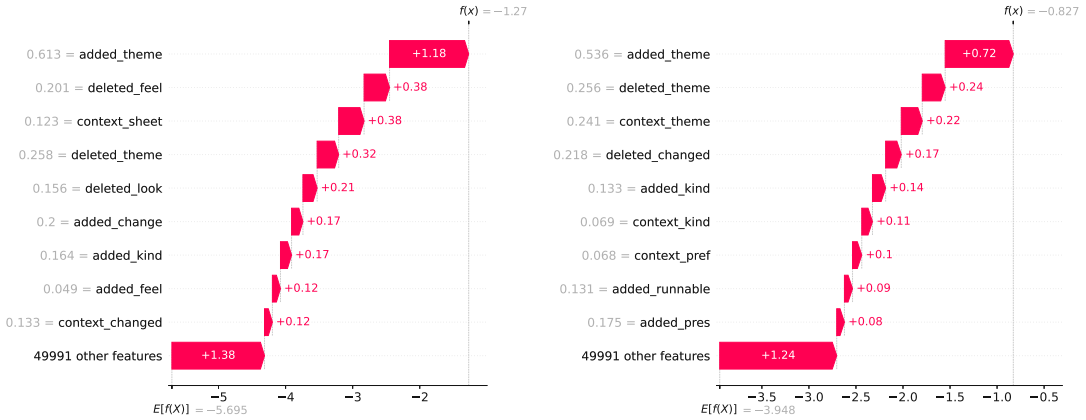


Figure 6.9: SHAP waterfall plots for bag-of-words features. Left: commit for revision 9ac290ec5884; right: commit group for bug 1718755.

By taking a look at the bag-of-words feature importance in Figure 6.9, we confirm that it is the presence of certain tokens that stir the prediction towards regression rather than the absence. It seems that the tokens “theme”, “kind” and “changed”, whether they appear in added, deleted, or context lines, are most important to the models for our data point. These words make sense, as the commit was concerned with the customisation of the user interface in form of browser themes.

In summary, while the handcrafted features are generally used in the intended way, we have found many contradictions and counter-intuitive features, which makes it hard to believe that the models learned a meaningful classification. They are certainly not good enough to give pointers to developers on how to improve their code. By investigating a particular prediction we could see that the most important tokens are fitting to the corresponding software development task, but it is not clear why the bag-of-words model works equally well.

6.6 Performance Regressions Versus General Bugs

In the last section we have seen that the interpretation of the models in terms of feature importance is difficult to make sense of. Next we will compare the performance regression prediction models to models trained for general bug prediction to investigate whether there is a difference between the models in terms of prediction capability and feature importance, or if the models look the same and there is no significant difference in the learned classification.

As the SZZ labeling did not yield good results, see Section 5.5.1 and 6.1, we make the comparison only on the bugbug labeling, see Section 5.6, which is readily adjusted for general regressions:

1. Collect bugs from the bug-tracking-system that have a non-empty `regressed_by` field.
2. ~~Get performance bugs by filtering for keywords "perf", "perf-alert", and "topperf".~~
3. Collect all bug numbers from the `regressed-by` field of the selected bugs.
4. Group the commits according to the rules.
5. Label all commit groups which worked on a bug number from step 3 as ~~performance~~ regression inducing.
6. Label all other commit groups as non-regression.

This leaves us with 14287 out of 67921 (21.03%) positively labeled commit groups. By definition this also contains the 1880 performance regression inducing commit groups.

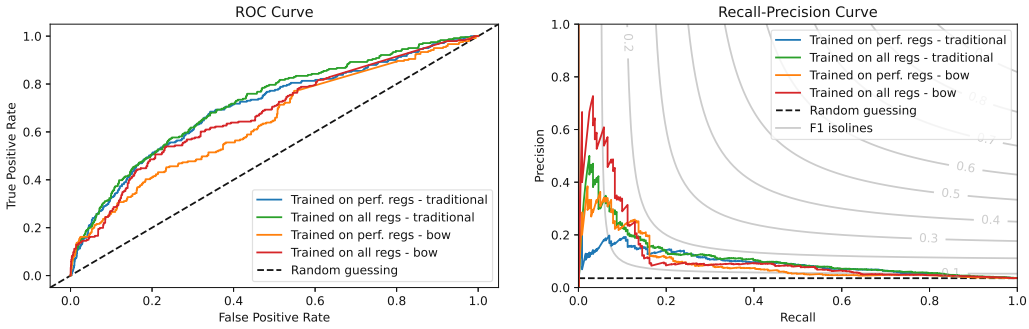


Figure 6.10: Models trained on all regressions compared to models trained only on performance regressions on the performance regression test set.

After a full hyper-parameter search (for results see Appendix Table 8.7 and 8.8), we compare the best model trained on all regressions, in this case a random forest model for the reduced traditional features and a XGBoost model for the bag-of-words features, on the performance regression test set. It seems like there is a good transfer of prediction capabilities, and for some recall ranges even better precision, see Figure 6.10. However, if we compare the scores at the best confidence level determined on the training set, we see that the models trained specifically on performance regressions have a slight edge over the models trained on all regressions, see Table 6.9. For the traditional features the F1 score decreased from 0.1734 only to 0.1396 and for the bag-of-words features from 0.1745 to 0.1417. It is interesting that in both cases the recall remains higher at the cost of precision.

Now one may wonder what happens if we reverse the roles and evaluate the performance regression model on all regressions. In Figure 6.11 we can see that in this case the prediction capabilities are significantly worse. For the traditional features the F1 score decreased from 0.4307 only to 0.2030 and for the bag-of-words features from 0.4114 to

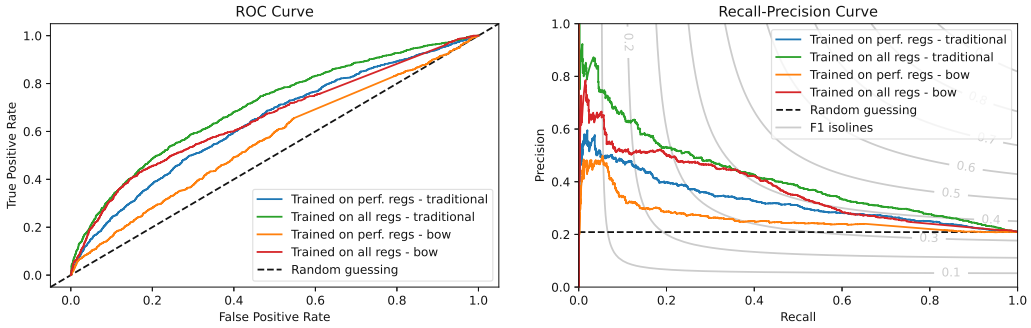


Figure 6.11: Models trained on all regressions compared to models trained only on performance regressions on the all regression test set.

0.1062. Interestingly, this is only due to a drop in recall, whereas the precision even increases a bit, see Table 6.9. However, at least for the traditional features the F1 score remains relatively high. This means that a model trained on performance regressions is capable to predict at least some general regressions. The same conclusion is not true for the bag-of-words feature type.

Feature Type	Trained on	Evaluated on	Recall	Precision	F1
traditional	perf. regressions	perf. regressions	0.2324	0.1383	0.1734
	all regressions	perf. regressions	0.5643	0.0797	0.1396
bag-of-words	perf. regressions	perf. regressions	0.1535	0.2022	0.1745
	all regressions	perf. regressions	0.4772	0.0832	0.1417
traditional	perf. regressions	all regressions	0.1305	0.4568	0.2030
	all regressions	all regressions	0.4746	0.3943	0.4307
bag-of-words	perf. regressions	all regressions	0.0599	0.4645	0.1062
	all regressions	all regressions	0.4062	0.4168	0.4114

Table 6.9: Comparison of best models trained and evaluated on performance regressions and general regressions.

Lastly, we take a look at the most important features to see if there is a significant difference between a model trained on performance regressions and a model trained on all regressions. Like before, we show the the SHAP summary plot and the waterfall plot for the performance bug 1718755 described in Section 6.5, which is correctly predicted by both models trained on all regressions.

In Figure 6.12, you can see the SHAP plots for the best logistic regression model trained on all regressions with traditional features. While the features code entropy, number of backouts, and developer experience, also appear among the most important features, it seems that the complexity metrics like the Halstead measures and maintainability index are now missing, compare Figure 6.6. Once again we can see contradictions in the interpretation of the features: one time high developer experience pushes the prediction towards regression, one time towards non-regression. By comparing the waterfall plot

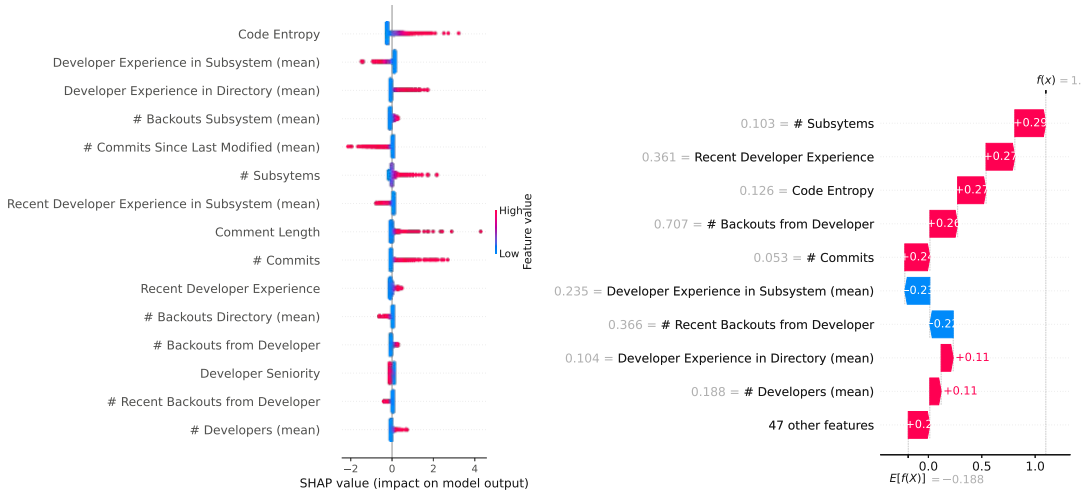


Figure 6.12: SHAP summary plot and waterfall plot of bug 1718755 for model trained on all regressions with traditional features.

of bug 1718755 to the one of the performance regression model, Figure 6.8, we see that while previously the number of backouts of the developer was the biggest push towards regression, now the number of subsystems touched, code entropy and developer experience are important.

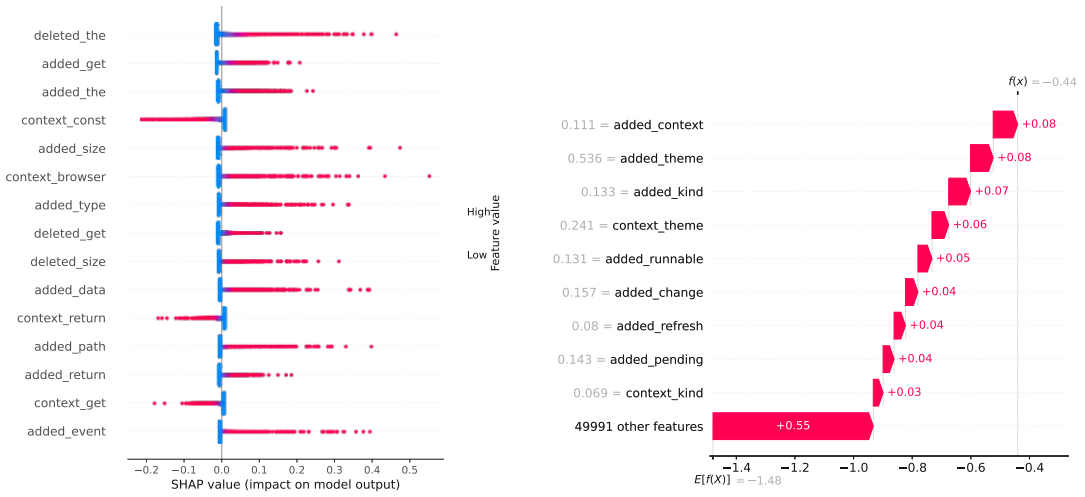


Figure 6.13: SHAP summary plot and waterfall plot of bug 1718755 for model trained on all regressions with bag-of-words features.

In Figure 6.13, you can see that the most important tokens for the bag-of-words model are once again very generic words. In particular, it is odd that the token “the” is on top of the list as we would assume that it should not hold any significant information for

predicting regressions. We could filter out words like these, but a good model should be able to tell that these words are only noise and focus on more important tokens.

Comparing the waterfall plot to Figure 6.9, we are happy to see that there is at least some consistency in the sense that the tokens “theme”, “kind”, and “change” are once again present. However, there are also new tokens like “context” or “refresh” which we did not see before. Furthermore, the overall behavior of bag-of-words model to mainly evaluate the presence rather than absence of tokens as an indicator for a regression still holds for the general defect prediction models.

In summary, a model trained on general regressions can transfer its prediction capabilities to the subset of performance regressions with classification scores just a bit shy of the scores of a model trained specifically on this subset (approximately -20%). The reverse is not true: a performance regression model cannot successfully predict general regressions. Although there is some transfer with traditional features, there is almost none for bag-of-words features. We conclude that there is a significant difference between the models for performance regressions and general defects. The performance regression model seems to value complexity metrics higher than the general regression model. The most important bag-of-words tokens stay similar to the performance regression model.

CHAPTER 7

Conclusion

7.1 Summary

We summarise the key findings of this thesis by revisiting the research questions posed in the introduction, see Chapter 1.

> *Is the SZZ algorithm suitable for labeling performance regression inducing code changes correctly?*

Although the SZZ algorithm is commonly used in the field of software defect prediction [RRGB18], we could not successfully leverage the improved version SZZUnleashed [BSBH19] to label performance regressions correctly. In Section 5.5.1, we used information from the benchmark monitoring tool Perfherder and manually reported links from bug-fixing to bug-introducing code changes to estimate the number of true positives and false positives of the labeling obtained by SZZUnleashed to be very high. In addition, the highest achieved F1-score of the machine learning models is 0.0526, see Table 6.7. This makes us conclude that the SZZ algorithm is not suitable for predicting performance regressions in the Firefox repository.

> *Is a labeling based on bug numbers suitable for detecting performance regression inducing code changes?*

Since 2019, Mozilla software engineers provide associations of bug-introducing and bug-fixing source code by linking the bug number of a commit which caused a regression to their bug-fix issue in the bug-tracking-system. As a bug number maps to multiple consecutive commits in general, we showed that labeling singular commits based on bug

numbers is not appropriate, because this introduces a positional bias in the data, see Section 5.6.

However, we presented an approach to collect commits into groups, which essentially represent the work of a developer on a particular issue in one session. In addition to removing the positional bias, the grouping also makes sense from a software engineering perspective: bugs may be introduced not only by single commits, but also by the interaction of multiple code changes. The best model trained on this labeling achieved a 0.1745 F1-score, a 3× improvement over the SZZ algorithm, see Table 6.7.

We gave three reasons which may explain why this approach works better than the SZZ algorithm: First, the main signal for linking bug-introducing code to bug-fixing code comes directly from the Firefox software engineers. Second, the key assumptions of the SZZ algorithm that bugs are fixed at the same code lines they were introduced at seems to be wrong for performance regressions. Third, performance bugs may be introduced by the interplay of code changes of several commits, which may be captured by the representation of code changes in terms of our grouping method.

Even though the classification scores are far from perfect, we conclude that our labeling approach, which we call *bugbug* in homage to the Mozilla machine learning framework [Moz22a], is suitable for performance regression prediction. It outperforms the automatic labeling of the SZZ algorithm, makes sense from a software engineering perspective, and is based directly on expert knowledge, namely the bug number links made by Mozilla developers.

> *Which sampling technique is best for dealing with the imbalanced nature of performance regression data sets?*

In Section 6.4, we took a quantitative approach to finding the best sampling technique by ranking the best models for each of following methods: no sampling, over-sampling, under-sampling, and SMOTE. After averaging the ranks over multiple labeling and feature type configurations, we found that no sampling and over-sampling perform the best. These results seem reasonable as these methods work with all the original data. They do not throw away any data or synthetically generate new data, like under-sampling or SMOTE, respectively. However, note that the best model with hand-crafted features based on the bugbug labeling uses SMOTE, see Table 6.7.

> *Which machine learning model is best at generalising to unseen data for the task of performance regression prediction?*

By looking at the best models after hyper-parameter tuning for both labeling approaches (SZZ and bugbug) and feature types (traditional and bag-of-words), we observed the general trend that the classification scores of the linear models logistic regression and support vector machine are the most similar on training, validation, and test data splits,

indicating the best generalisation capability, see Section 6.1, 6.2, and 6.3. On the other hand, tree-based models like random forest, XGBoost, or the extra trees classifier overfit severely on the training data. Nonetheless, they sometimes still outperform the linear models on the test set. The multi-layer perceptron lies somewhere between both extremes in terms of overfitting.

> *How does a bag-of-words model compare to a model trained on hand-crafted features?*

Surprisingly, the simple bag-of-words approach described in Section 4.2.2 produced models on par to the models trained on hand-crafted features for both the SZZ and bugbug labeling. In fact, it was a bag-of-words logistic regression model that achieved the best F1-score of 0.1745 closely followed by a logistic regression model based on traditional features with a F1-score of 0.1734. In our efforts to understand how the bag-of-words models work so well, we found that the overall most important tokens are just generic programming words like "return" or "get". However, by looking at a specific data point, we saw that the tokens, which contributed the most to the prediction, were fitting to the corresponding software engineering task. A final explanation for the performance of bag-of-words models in the context of performance regression prediction remains for future research.

> *How does a performance regression prediction model differ from a general defect prediction model?*

In Section 6.6, we concluded that models trained for general regression prediction can transfer their capabilities to the subset of performance regressions with classification scores just 20% shy of the scores of a model trained specifically for performance regressions. This is true for both traditional and bag-of-words features. The transfer of prediction capability in the reverse direction fails. There are more complexity metrics, like the Halstead metrics, among the most important features for the performance regression model than for the general regression model, where more general features like developer experience or number of reverted changes dominate. We could not find significant differences between the most important tokens of the bag-of-words models.

> *To what extent are machine learning models capable of detecting performance regression inducing code changes from source code features in a just in time manner, specifically at code edit or commit time?*

Building a machine learning pipeline for performance regression prediction is particularly difficult, because there is uncertainty in the correctness of the data labeling and the suitability of the selected modeling methods. In Section 5.6, we have seen that if we are

not careful when labeling the data and with choosing the correct evaluation setup, it is easy to produce misleading good classification scores. Furthermore, there is constantly the question if the constructed features provide a good signal for performance regressions or if there can even exist such a signal. In summary, there is a permanent concern that either the labeling is bad or the model is bad, see Figure 7.1.

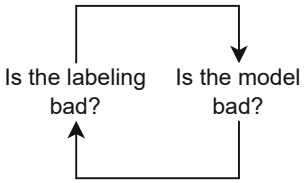


Figure 7.1: The dilemma of performance regression prediction.

Nevertheless, we were able to build a model that can predict performance regression 5.7× more precisely than random guessing. With our approach we were able to improve the labeling which led to an increase of precision to 0.2022 and recall to 0.1535, which makes a 0.1745 F1-score for our best model. Still, there is room for improvement and we list ideas worth pursuing in future work in Section 7.3, for example, an in-depth investigation of natural language processing methods for predicting performance regressions.

7.2 Contributions

The main contributions of this thesis to the research field of software defect prediction are as follows:

- An open source implementation of a full machine learning pipeline for performance regression prediction from data labeling, to feature extraction and model training. All experiments are reproducible at `https://github.com/ipa-lab/firefox-performance-regressions`.
- An evaluation of the applicability of the SZZ algorithm for performance regressions in the Firefox repository.
- An approach to labeling based on grouping of coherent commits, which makes sense from a software engineering perspective, is based directly on expert knowledge, and improves the performance of models significantly.
- A discussion on the correct evaluation setup for defect prediction and an example where the standard cross-validation approach produced highly misleading results.
- An in-depth investigation of feature importance of performance regression prediction models in comparison to general defect prediction model both for hand-crafted features and bag-of-words tokens.

7.3 Limitations and Future Work

A shortcoming of this work is that we only considered performance regression prediction for Mozilla Firefox. However, this allowed us to put a lot of effort in evaluating and adjusting the labeling process specifically for Firefox, testing several machine learning methods, as well as interpreting the final models in detail. Still, extending our approach to other performance critical software projects will be necessary for future work. In particular, it would be interesting to test whether the bugbug labeling can be translated to other repositories.

Furthermore, the surprising predictive capabilities of the bag-of-words models make natural language processing methods look very promising for the task of software defect prediction. It remains for future work to find the reasons why this approach works so well, for example, with counterfactual explanations [CDMC21]. Also, more sophisticated natural language processing methods, like large transformer-based models or methods that take the semantics of source code into account, should be considered for performance regression prediction.

Lastly, it could be fruitful to consult software engineers to find new insights into how performance bugs happen and how they are fixed to build a model following their expertise. From our perspective it would also be interesting to conduct an experiment where the root cause of performance problems is recorded accurately either on the commit or even on the source code line level. This would remove the uncertainty in the labeling which would allow us to focus on the machine learning part of the prediction task.

7.4 Threats to Validity

External Validity is the extent to which the findings of this work can be generalised. In this thesis we only considered one performance critical software project, namely the Mozilla Firefox web browser. Therefore, it is possible that our results and conclusions may not transfer to other repositories. However, Firefox is a very large project, comprised of many modules covering several aspects of software engineering (graphical interfaces, networking, etc.) which are written in multiple programming languages. This makes us confident that our conclusions are likely to hold for other software projects.

Internal Validity is the extend to which the findings of this work are indeed explained by the presented data and not by other factors. The biggest uncertainty in our results comes from the labeling process. In Section 2.5.1, we described the limitations of the SZZ algorithm and in Section 5.5.1, we demonstrated a high false positive and false negative rates of the labeling produced by the SZZ algorithm. Even though our bugbug labeling directly leverages the knowledge of Mozilla developers, see Section 5.6, there is still the possibility of inaccurate or insufficient bug reporting leading to an incorrect data labeling, which directly influences the experiment results. Furthermore, while particular care was taken to not include any future information about the source code in the prediction, this was not possible for the bug reports. As we do not have the exact state of the

bug-tracking-system at each time step, we extracted all information from the system, but only consider commits older than three months, see Section 4.1.2. This way we ensure consistent and (almost) complete bug reporting for all considered commits, however, the most recent state of the repository is slightly different due to undetected bugs. Lastly, we decided to use the SHAP framework for model interpretation, however, there are also other methods that could lead to slightly different conclusions [Mol22].

CHAPTER 8

Appendix

In Section 8.1, 8.2, and 8.3, you can find the classification scores for the best machine learning models for the SZZ labeling, bugbug labeling, and bugbug labeling for general regressions, respectively. The best models are chosen based on their average precision on the validation splits in the hyper-parameter search. We report recall, precision, and F1 score on training and test set with the confidence threshold determined on the training data. Additionally, we report these three metrics on the test set with the confidence threshold determined on the test data, which can be found in the "test pareto" column. By comparing these scores to the unbiased test scores, we can estimate the extend of overfitting present in the models.

In Section 8.4, we report the best hyper-parameters for all combinations of labeling, model, and feature type. The traditional features refer to the reduced features described in Section 6.3.

In Section 8.5, we report the best sampling method for all combinations of labeling, model, and feature type with the corresponding average precision on the validation splits. Again, the traditional features refer to the reduced features described in Section 6.3.

8.1 SZZ Labeling

		training			test			test pareto		
		recall	precision	F1	recall	precision	F1	recall	precision	F1
Dummy Classifier	regression	1.0000	0.0063	0.0126	1.0000	0.0022	0.0044	1.0000	0.0022	0.0044
	non-regression	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Logistic Regression	regression	0.0723	0.0401	0.0516	0.0455	0.0066	0.0116	0.0455	0.0179	0.0256
	non-regression	0.9890	0.9941	0.9915	0.9849	0.9979	0.9913	0.9945	0.9979	0.9962
Support Vector Machine	regression	0.1411	0.0963	0.1144	0.0000	0.0000	0.0000	0.1364	0.0102	0.0189
	non-regression	0.9916	0.9945	0.9930	0.9903	0.9978	0.9940	0.9706	0.9980	0.9842
Random Forest	regression	0.9965	0.9843	**0.9904**	0.0000	0.0000	0.0000	0.0455	0.0182	0.0260
	non-regression	0.9999	1.0000	0.9999	1.0000	0.9978	0.9989	0.9946	0.9979	0.9962
XGBoost	regression	0.5009	0.3994	0.4444	0.0000	0.0000	0.0000	0.0909	0.0256	**0.0400**
	non-regression	0.9952	0.9968	0.9960	0.9974	0.9978	0.9976	0.9924	0.9980	0.9952
Multi-Layer Perceptron	regression	0.1217	0.0752	0.0930	0.0909	0.0177	**0.0296**	0.0909	0.0194	0.0320
	non-regression	0.9905	0.9944	0.9924	0.9888	0.9980	0.9934	0.9898	0.9980	0.9939
TPOT	regression	0.1958	0.1512	0.1706	0.0455	0.0116	0.0185	0.3182	0.0191	0.0361
	non-regression	0.9930	0.9949	0.9939	0.9915	0.9979	0.9947	0.9639	0.9984	0.9809

Table 8.1: Classification scores of all models for SZZ labeling with traditional features.

		training			test			test pareto		
		recall	precision	F1	recall	precision	F1	recall	precision	F1
Dummy Classifier	regression	1.0000	0.0063	0.0126	1.0000	0.0022	0.0044	1.0000	0.0022	0.0044
	non-regression	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Logistic Regression	regression	0.0476	0.0374	0.0419	0.0455	0.0078	0.0133	0.1818	0.0221	0.0394
	non-regression	0.9922	0.9939	0.9931	0.9872	0.9979	0.9925	0.9822	0.9982	0.9901
Support Vector Machine	regression	0.1429	0.0502	0.0743	0.0455	0.0068	0.0118	0.0455	0.0137	0.0211
	non-regression	0.9828	0.9945	0.9886	0.9853	0.9979	0.9916	0.9928	0.9979	0.9953
Random Forest	regression	0.9877	0.9672	**0.9773**	0.0000	0.0000	0.0000	0.0909	0.0215	0.0348
	non-regression	0.9998	0.9999	0.9999	1.0000	0.9978	0.9989	0.9909	0.9980	0.9944
XGBoost	regression	0.2769	0.2240	0.2476	0.0000	0.0000	0.0000	0.3182	0.0054	0.0107
	non-regression	0.9939	0.9954	0.9946	0.9959	0.9978	0.9968	0.8709	0.9983	0.9303
Multi-Layer Perceptron	regression	0.1499	0.0928	0.1146	0.0000	0.0000	0.0000	0.4545	0.0046	0.0091
	non-regression	0.9907	0.9946	0.9926	0.9910	0.9978	0.9944	0.7817	0.9985	0.8769
TPOT	regression	0.3686	0.3821	0.3752	0.1364	0.0326	**0.0526**	0.0909	0.0714	**0.0800**
	non-regression	0.9962	0.9960	0.9961	0.9911	0.9981	0.9946	0.9974	0.9980	0.9977

Table 8.2: Classification scores of all models for SZZ labeling with traditional features after feature reduction.

		training			test			test pareto		
		recall	precision	F1	recall	precision	F1	recall	precision	F1
Dummy Classifier	regression	1.0000	0.0063	0.0126	1.0000	0.0022	0.0044	1.0000	0.0022	0.0044
	non-regression	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Logistic Regression	regression	0.4339	0.3739	0.4016	0.0455	0.0204	0.0282	0.0455	0.2500	**0.0769**
	non-regression	0.9954	0.9964	0.9959	0.9952	0.9979	0.9965	0.9997	0.9979	0.9988
Support Vector Machine	regression	0.3563	0.1563	0.2173	0.1818	0.0288	**0.0497**	0.0909	0.0645	0.0755
	non-regression	0.9878	0.9959	0.9918	0.9864	0.9982	0.9923	0.9971	0.9980	0.9975
Random Forest	regression	0.7637	0.6541	**0.7046**	0.0455	0.0095	0.0157	0.0909	0.0156	0.0267
	non-regression	0.9974	0.9985	0.9980	0.9895	0.9979	0.9937	0.9873	0.9980	0.9926
XGBoost	regression	0.3122	0.5822	0.4064	0.0455	0.0312	0.0370	0.0455	0.0370	0.0408
	non-regression	0.9986	0.9956	0.9971	0.9969	0.9979	0.9974	0.9974	0.9979	0.9976

Table 8.3: Classification scores of all models for SZZ labeling with bag-of-words features.

8.2 BugBug Labeling

		training			test			test pareto		
		recall	precision	F1	recall	precision	F1	recall	precision	F1
Dummy Classifier	regression	1.0000	0.0268	0.0522	1.0000	0.0355	0.0685	1.0000	0.0355	0.0685
	non-regression	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Logistic Regression	regression	0.2465	0.0916	0.1336	0.2946	0.1084	**0.1585**	0.3402	0.1107	0.1670
	non-regression	0.9326	0.9782	0.9549	0.9109	0.9723	0.9406	0.8994	0.9737	0.9351
Support Vector Machine	regression	0.1916	0.1188	0.1467	0.2199	0.1183	0.1538	0.2075	0.1401	0.1672
	non-regression	0.9608	0.9773	0.9690	0.9397	0.9704	0.9548	0.9531	0.9703	0.9617
Random Forest	regression	0.1891	0.2967	0.2310	0.0539	0.1354	0.0772	0.4274	0.0891	0.1475
	non-regression	0.9876	0.9779	0.9827	0.9873	0.9660	0.9765	0.8393	0.9755	0.9023
XGBoost	regression	0.4802	0.7560	0.5873	0.0788	0.1959	0.1124	0.2158	0.1313	0.1633
	non-regression	0.9957	0.9858	0.9908	0.9881	0.9668	0.9774	0.9475	0.9705	0.9588
Multi-Layer Perceptron	regression	0.2245	0.1518	0.1811	0.1618	0.1071	0.1289	0.2448	0.1180	0.1592
	non-regression	0.9654	0.9783	0.9719	0.9504	0.9686	0.9594	0.9327	0.9711	0.9515
TPOT	regression	0.8566	0.9784	**0.9135**	0.0581	0.5000	0.1041	0.1950	0.1483	**0.1685**
	non-regression	0.9995	0.9961	0.9978	0.9979	0.9664	0.9819	0.9588	0.9700	0.9644

Table 8.4: Classification scores of all models for bugbug labeling with traditional features.

		training			test			test pareto		
		recall	precision	F1	recall	precision	F1	recall	precision	F1
Dummy Classifier	regression	1.0000	0.0268	0.0522	1.0000	0.0355	0.0685	1.0000	0.0355	0.0685
	non-regression	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Logistic Regression	regression	0.1696	0.1100	0.1334	0.2324	0.1383	**0.1734**	0.2407	0.1404	**0.1774**
	non-regression	0.9622	0.9768	0.9694	0.9467	0.9710	0.9587	0.9458	0.9713	0.9584
Support Vector Machine	regression	0.2965	0.1454	0.1951	0.2822	0.1151	0.1635	0.3029	0.1199	0.1718
	non-regression	0.9520	0.9800	0.9658	0.9202	0.9721	0.9454	0.9182	0.9728	0.9447
Random Forest	regression	0.3087	0.1719	0.2208	0.1037	0.0806	0.0907	0.3942	0.0946	0.1526
	non-regression	0.9590	0.9805	0.9697	0.9565	0.9667	0.9616	0.8613	0.9748	0.9145
XGBoost	regression	0.4790	0.6250	0.5423	0.0830	0.1550	0.1081	0.1826	0.1155	0.1415
	non-regression	0.9921	0.9857	0.9889	0.9834	0.9668	0.9750	0.9486	0.9693	0.9588
Multi-Layer Perceptron	regression	0.3008	0.2490	0.2725	0.1494	0.1053	0.1235	0.3237	0.1022	0.1554
	non-regression	0.9750	0.9806	0.9778	0.9533	0.9682	0.9607	0.8955	0.9730	0.9326
TPOT	regression	0.9811	0.9110	**0.9448**	0.0373	0.2000	0.0629	0.1909	0.1407	0.1620
	non-regression	0.9974	0.9995	0.9984	0.9945	0.9656	0.9798	0.9571	0.9698	0.9634

Table 8.5: Classification scores of all models for bugbug labeling with traditional features after feature reduction.

		training			test			test pareto		
		recall	precision	F1	recall	precision	F1	recall	precision	F1
Dummy Classifier	regression	1.0000	0.0268	0.0522	1.0000	0.0355	0.0685	1.0000	0.0355	0.0685
	non-regression	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Logistic Regression	regression	0.4875	0.4719	0.4796	0.1535	0.2022	**0.1745**	0.1618	0.1980	0.1781
	non-regression	0.9850	0.9859	0.9854	0.9777	0.9691	0.9734	0.9759	0.9694	0.9726
Support Vector Machine	regression	0.2849	0.3029	0.2936	0.1203	0.1768	0.1432	0.1743	0.1687	0.1714
	non-regression	0.9819	0.9803	0.9811	0.9794	0.9680	0.9737	0.9684	0.9696	0.9690
Random Forest	regression	0.6919	0.7354	**0.7130**	0.0830	0.2174	0.1201	0.3320	0.1423	**0.1993**
	non-regression	0.9931	0.9915	0.9923	0.9890	0.9670	0.9779	0.9264	0.9742	0.9497
XGBoost	regression	0.5308	0.7311	0.6151	0.0871	0.1795	0.1173	0.2324	0.1244	0.1621
	non-regression	0.9946	0.9872	0.9909	0.9853	0.9670	0.9761	0.9399	0.9708	0.9551

Table 8.6: Classification scores of all models for bugbug labeling with bag-of-words features.

8.3 BugBug Labeling - General Regressions

		training			test			test pareto		
		recall	precision	F1	recall	precision	F1	recall	precision	F1
Dummy Classifier	regression	1.0000	0.2105	0.3478	1.0000	0.2087	0.3454	1.0000	0.2087	0.3454
	non-regression	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Logistic Regression	regression	0.5852	0.3277	0.4201	0.5515	0.3001	0.3887	0.4492	0.3517	0.3945
	non-regression	0.6798	0.8601	0.7594	0.6607	0.8481	0.7427	0.7816	0.8432	0.8112
Support Vector Machine	regression	0.5620	0.3636	0.4415	0.5071	0.3243	0.3956	0.6551	0.2916	0.4036
	non-regression	0.7377	0.8633	0.7956	0.7213	0.8472	0.7792	0.5801	0.8644	0.6943
Random Forest	regression	0.7458	0.8729	**0.8044**	0.4746	0.3943	**0.4307**	0.5402	0.3683	**0.4380**
	non-regression	0.9710	0.9348	0.9525	0.8076	0.8535	0.8299	0.7555	0.8617	0.8051
XGBoost	regression	0.5659	0.4592	0.5070	0.4986	0.3652	0.4216	0.6333	0.3234	0.4281
	non-regression	0.8223	0.8766	0.8486	0.7713	0.8536	0.8104	0.6504	0.8705	0.7445
Multi-Layer Perceptron	regression	0.5785	0.3594	0.4434	0.5635	0.3339	0.4193	0.5395	0.3476	0.4228
	non-regression	0.7250	0.8658	0.7892	0.7034	0.8593	0.7736	0.7328	0.8578	0.7904
TPOT	regression	0.6020	0.4745	0.5307	0.5099	0.3611	0.4228	0.5550	0.3493	0.4288
	non-regression	0.8222	0.8857	0.8527	0.7620	0.8549	0.8058	0.7273	0.8610	0.7885

Table 8.7: Classification scores of all models for bugbug labeling for all regressions with traditional features after feature reduction.

		training			test			test pareto		
		recall	precision	F1	recall	precision	F1	recall	precision	F1
Dummy Classifier	regression	1.0000	0.2105	0.3478	1.0000	0.2087	0.3454	1.0000	0.2087	0.3454
	non-regression	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Logistic Regression	regression	0.5814	0.4424	0.5024	0.4140	0.3164	0.3587	0.6206	0.2807	0.3866
	non-regression	0.8046	0.8782	0.8397	0.7641	0.8317	0.7965	0.5805	0.8529	0.6908
Support Vector Machine	regression	0.5606	0.3484	0.4298	0.4908	0.3053	0.3764	0.5409	0.2989	0.3850
	non-regression	0.7204	0.8601	0.7841	0.7053	0.8400	0.7668	0.6653	0.8460	0.7448
Random Forest	regression	0.8535	0.9987	**0.9204**	0.1368	0.5791	0.2213	0.5402	0.3477	**0.4231**
	non-regression	0.9997	0.9624	0.9807	0.9738	0.8105	0.8846	0.7327	0.8580	0.7904
XGBoost	regression	0.4752	0.5032	0.4888	0.4062	0.4168	**0.4114**	0.4386	0.3952	0.4158
	non-regression	0.8749	0.8621	0.8685	0.8500	0.8444	0.8472	0.8229	0.8475	0.8350

Table 8.8: Classification scores of all models for bugbug labeling for all regressions with bag-of-words features.

8.4 Best Hyper-Parameters

Data	Target	Feature Type	C	Penalty	Sampling
SZZ	perf. regressions	traditional	0.0167	l2	Random Over-Sampling
		bag-of-words	2.3731	l2	No sampling
bugbug	perf. regressions	traditional	0.6689	l2	SMOTE
		bag-of-words	0.7420	l2	No sampling
	all regressions	traditional	0.2075	l1	SMOTE
		bag-of-words	0.1658	l2	No sampling

Table 8.9: Best hyper-parameters for the logistic regression model.

Data	Target	Feature Type	Degree	Kernel	C	Sampling
SZZ	perf. regressions	traditional	3	rbf	6.2824	SMOTE
		bag-of-words	3	poly	593.4792	Random Over-Sampling
bugbug	perf. regressions	traditional	3	rbf	98.3742	Random Over-Sampling
		bag-of-words	5	linear	0.1330	Random Under-Sampling
	all regressions	traditional	3	rbf	95.2045	SMOTE
		bag-of-words	5	poly	97.9066	Random Over-Sampling

Table 8.10: Best hyper-parameters for the support vector machine model.

Data	Target	Feature Type	Max_depth	Min_samples_split	N_estimators	Sampling
SZZ	perf. regressions	traditional	20.0000	5	37	Random Over-Sampling
		bag-of-words	15.0000	2	150	No sampling
bugbug	perf. regressions	traditional	5.0000	5	121	Random Over-Sampling
		bag-of-words	None	5	150	Random Under-Sampling
	all regressions	traditional	15.0000	5	150	No sampling
		bag-of-words	None	5	150	No sampling

Table 8.11: Best hyper-parameters for the random forest model.

Data	Target	Feature Type	Gamma	Maximum delta step	Max_depth	Minimum child weight	N_estimators	Sampling
SZZ	perf. regressions	traditional	0.0052	1	3	1	78	Over-Sampling
		bag-of-words	0.5270	1	3	1	40	No sampling
bugbug	perf. regressions	traditional	0.2433	1	5	1	42	No sampling
		bag-of-words	1.0000	0	3	1	56	No sampling
	all regressions	traditional	0.0000	0	4	5	44	No sampling
		bag-of-words	1.0000	1	3	5	53	No sampling

Table 8.12: Best hyper-parameters for the XGBoost model.

Data	Target	Feature Type	Activation	Alpha	Hidden_layer_sizes	Learning_rate_init	Sampling
SZZ	perf. regressions	traditional	tanh	0.0003	50	0.0001	SMOTE
bugbug	perf. regressions	traditional	relu	0.0001	200	0.0001	Random Over-Sampling
	all regressions	traditional	relu	0.0001	10	0.0015	No sampling

Table 8.13: Best hyper-parameters for the multi-layer perceptron model.

8.5 Best Sampling Methods

Data	Target	Feature Type	Model	Sampling	AP (val.)
SZZ	perf. regressions	traditional	Logistic Regression	Over-Sampling	0.0278
		traditional	Support Vector Machine	SMOTE	0.0362
		traditional	Random Forest	Over-Sampling	0.0482
		traditional	XGBoost	Over-Sampling	0.0546
		traditional	Multi-Layer Perceptron	SMOTE	0.0368
		bag-of-words	Logistic Regression	No sampling	0.0670
		bag-of-words	Support Vector Machine	Over-Sampling	0.0563
		bag-of-words	Random Forest	No sampling	0.0627
		bag-of-words	XGBoost	No sampling	0.0765
bugbug	perf. regressions	traditional	Logistic Regression	SMOTE	0.0747
		traditional	Support Vector Machine	Over-Sampling	0.0880
		traditional	Random Forest	Over-Sampling	0.1044
		traditional	XGBoost	No sampling	0.1037
		traditional	Multi-Layer Perceptron	Over-Sampling	0.1040
		bag-of-words	Logistic Regression	No sampling	0.1249
		bag-of-words	Support Vector Machine	Under-Sampling	0.1181
		bag-of-words	Random Forest	Under-Sampling	0.1343
		bag-of-words	XGBoost	No sampling	0.1358
	all regressions	traditional	Logistic Regression	SMOTE	0.3551
		traditional	Support Vector Machine	SMOTE	0.3820
		traditional	Random Forest	No sampling	0.4187
		traditional	XGBoost	No sampling	0.4144
		traditional	Multi-Layer Perceptron	No sampling	0.3935
		bag-of-words	Logistic Regression	No sampling	0.3753
		bag-of-words	Support Vector Machine	Over-Sampling	0.3666
		bag-of-words	Random Forest	No sampling	0.4155
		bag-of-words	XGBoost	No sampling	0.4027

Table 8.14: Best sampling method for every data and model configuration.

List of Figures

List of Tables

Index

Bibliography

[ABC+20] Luca Ardito, Luca Barbato, Marco Castelluccio, Riccardo Coppola, Calixte Denizet, Sylvestre Ledru, and Michele Valsesia. rust-code-analysis: A rust library to analyze and extract maintainability information from source codes. *SoftwareX*, 12:100635, 2020.

[ACDG07] Lerina Aversano, Luigi Cerulo, and Concettina Del Grosso. Learning from bug-introducing changes to prevent fault prone code. In *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, pages 19–26, 2007.

[ACRC21] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*, 2021.

[AKG18] Le An, Foutse Khomh, and Yann-Gaël Guéhéneuc. An empirical study of crash-inducing commits in mozilla firefox. *Software Quality Journal*, 26(2):553–584, 2018.

[AS00] L. Arief and Neil Speirs. A uml tool for an automatic generation of simulation programs. pages 71–76, 01 2000.

[BB12] Christoph Bergmeir and José M Benítez. On the use of cross-validation for time series predictor evaluation. *Information Sciences*, 191:192–213, 2012.

[Bis06] Christopher M Bishop. Pattern recognition. *Machine learning*, 128(9), 2006.

[BLN+22] Moritz Beller, Hongyu Li, Vivek Nair, Vijayaraghavan Murali, Imad Ahmad, Jürgen Cito, Drew Carlson, Ari Aye, and Wes Dyer. Learning to learn to predict performance regressions in production at meta (experience paper). 2022.

[BMI04] Simonetta Balsamo, Antinisca Marco, and Paola Inverardi. Model-based performance prediction in software development: A survey. *Software Engineering, IEEE Transactions on*, 30:295 – 310, 06 2004.

[Bre01] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[BSBH19] Markus Borg, Oscar Svensson, Kristian Berg, and Daniel Hansson. Szz unleashed: An open implementation of the szz algorithm-featuring example usage in a study of just-in-time bug prediction for the jenkins project. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation*, pages 7–12, 2019.

[CBHK02] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[CDMC21] Jürgen Cito, Isil Dillig, Vijayaraghavan Murali, and Satish Chandra. Counterfactual explanations for models of code. *arXiv preprint arXiv:2111.05711*, 2021.

[CG16] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

[CL19] Marco Castelluccio and Sylvestre Ledru. Teaching machines to triage firefox bugs. https://hacks.mozilla.org/2019/04/teaching-machines-to-triage-firefox-bugs/, 2019. Accessed: 2022-03-32.

[CMS+16] Daniel Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E. Hassan. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering*, PP:1–1, 10 2016.

[CP03] Joseph E Coffland and Andy D Pimentel. A software framework for efficient system-level performance evaluation of embedded systems. In *Proceedings of the 2003 ACM symposium on Applied computing*, pages 666–671, 2003.

[CZ11] Istehad Chowdhury and Mohammad Zulkernine. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, 57(3):294–313, 2011.

[DCS17] Salvatore Dipietro, Giuliano Casale, and Giuseppe Serazzi. A queueing network model for performance prediction of apache cassandra. 01 2017.

[DQRT15] Diego Didona, Francesco Quaglia, Paolo Romano, and Ennio Torre. Enhancing performance prediction robustness by combining analytical modeling and machine learning. 01 2015.

[Dre21] Frej Drejhammar. fast-export. https://github.com/frej/fast-export, 2021.

[FGT+20] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[GEW06] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine learning*, 63(1):3–42, 2006.

[Hal77] Maurice H Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

[HC20] Andrew Halberstadt and Marco Castelluccio. Testing firefox more efficiently with machine learning. https://hacks.mozilla.org/2020/07/testing-firefox-more-efficiently-with-machine-learning/, 2020. Accessed: 2022-03-32.

[HCN98] Rachel Harrison, Steve J Counsell, and Reuben V Nithi. An evaluation of the mood set of object-oriented software metrics. *IEEE Transactions on Software Engineering*, 24(6):491–496, 1998.

[HHK02] Holger Hermanns, Ulrich Herzog, and Joost-Pieter Katoen. Process algebra for performance evaluation. *Theor. Comput. Sci.*, 274:43–87, 2002.

[HM15] Mohammad Hossin and Sulaiman M.N. A review on evaluation metrics for data classification evaluations. *International Journal of Data Mining & Knowledge Management Process*, 5:01–11, 03 2015.

[HML+18] Tim Head, MechCoder, Gilles Louppe, Iaroslav Shcherbatyi, fcharras, Zé Vinícius, cmmalone, Christopher Schröder, nel215, Nuno Campos, Todd Young, Stefano Cereda, Thomas Fan, rene rex, Kejia (KJ) Shi, Justus Schwabedal, carlosdanielcsantos, Hvass-Labs, Mikhail Pak, SoManyUsernamesTaken, Fred Callaway, Loïc Estève, Lilian Besson, Mehdi Cherti, Karlson Pfannschmidt, Fabian Linzberger, Christophe Cauet, Anna Gut, Andreas Mueller, and Alexander Fabisch. scikit-optimize/scikit-optimize: v0.5.2, March 2018.

[HZ19] Huong Ha and Hongyu Zhang. Deepperf: Performance prediction for configurable software with deep sparse neural network. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1095–1106, 2019.

[JPZ08] Sascha Just, Rahul Premraj, and Thomas Zimmermann. Towards the next generation of bug tracking systems. In *2008 IEEE symposium on visual languages and human-centric computing*, pages 82–85. IEEE, 2008.

[KFM+16] Yasutaka Kamei, Takafumi Fukushima, Shane McIntosh, Kazuhiro Yamashita, Naoyasu Ubayashi, and Ahmed E Hassan. Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering*, 21(5):2072–2106, 2016.

[KP00] Peter King and Rob Pooley. Derivation of petri net performance models from uml specifications of communications software. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 262–276. Springer, 2000.

[KSA+12] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2012.

[KZP+06] Sunghun Kim, Thomas Zimmermann, Kai Pan, E James Jr, et al. Automatic identification of bug-introducing changes. In *21st IEEE/ACM international conference on automated software engineering (ASE'06)*, pages 81–90. IEEE, 2006.

[KZWG11] Sunghun Kim, Hongyu Zhang, Rongxin Wu, and Liang Gong. Dealing with noise in defect prediction. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 481–490. IEEE, 2011.

[LHZL17] Jian Li, Pinjia He, Jieming Zhu, and Michael R Lyu. Software defect prediction via convolutional neural network. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 318–328. IEEE, 2017.

[LL17] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 4765–4774. Curran Associates, Inc., 2017.

[LNA17] Guillaume Lemaître, Fernando Nogueira, and Christos K. Aridas. Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. *Journal of Machine Learning Research*, 18(17):1–5, 2017.

[McC76] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[Mol22] Christoph Molnar. *Interpretable Machine Learning*. 2 edition, 2022.

[Moz19] MozillaWiki. Engineeringproductivity/projects/perfherder. https://wiki.mozilla.org/EngineeringProductivity/Projects/Perfherder, 2019. Accessed: 2022-03-03.

[Moz22a] Mozilla. bugbug. https://github.com/mozilla/bugbug, 2022.

[Moz22b] Mozilla. History of the mozilla project. https://www.mozilla.org/en-US/about/history/, 2022. Accessed: 2022-03-02.

[MRS10] Christopher Manning, Prabhakar Raghavan, and Hinrich Schütze. Introduction to information retrieval. *Natural Language Engineering*, 16(1):100–103, 2010.

[NHL18] Mathieu Nayrolles and Abdelwahab Hamou-Lhadj. Clever: Combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 153–164, 2018.

[OM16] Randal S Olson and Jason H Moore. Tpot: A tree-based pipeline optimization tool for automating machine learning. In *Workshop on automatic machine learning*, pages 66–74. PMLR, 2016.

[Ope22] OpenHub. Mozilla Firefox: Project summary. https://www.openhub.net/p/firefox, 2022. Accessed: 2022-03-02.

[Ott09] Stefan Otte. Version control systems. *Computer Systems and Telematics*, pages 11–13, 2009.

[PMT21] Sushant Kumar Pandey, Ravi Bhushan Mishra, and Anil Kumar Tripathi. Machine learning based methods for software fault prediction: A survey. *Expert Systems with Applications*, 172:114595, 2021.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[RHTŽ13] Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič. Software fault prediction metrics: A systematic literature review. *Information and software technology*, 55(8):1397–1418, 2013.

[RRGB18] Gema Rodriguez, Gregorio Robles, and Jesus Gonzalez-Barahona. Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm. *Information and Software Technology*, 99, 03 2018.

[SHN+15] Weiyi Shang, Ahmed E. Hassan, Mohamed Nasser, Parminder Flora, Blackberry Waterloo, and Canada Ontario. Automated detection of performance regressions using regression models on clustered performance counters. 01 2015.

[Smy20] Eric Smyth. Our year in review: How we've made firefox faster in 2020. https://blog.mozilla.org/performance/2020/12/15/2020-year-in-review/, 2020. Accessed: 2022-03-02.

[Sta22] StatCounter Global Stats. Browser market share worldwide. https://gs.statcounter.com/browser-market-share#monthly-200901-202111, 2022. Accessed: 2022-03-02.

[SWAK12] Shivkumar Shivaji, E James Whitehead, Ram Akella, and Sunghun Kim. Reducing features to improve code change-based bug prediction. *IEEE Transactions on Software Engineering*, 39(4):552–569, 2012.

[SZZ05] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? volume 30, 07 2005.

[TMHM16] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering*, 43(1):1–18, 2016.

[TSR+20] Mahesh Kumar Thota, Francis H Shajin, P Rajesh, et al. Survey on software defect prediction techniques. *International Journal of Applied Science and Engineering*, 17(4):331–344, 2020.

[TTDM15] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. Online defect prediction for imbalanced data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 99–108, 2015.

[Wel01] Kurt D Welker. The software maintainability index revisited. *CrossTalk*, 14:18–21, 2001.

[WLNT18] Song Wang, Taiyue Liu, Jaechang Nam, and Lin Tan. Deep semantic feature learning for software defect prediction. *IEEE Transactions on Software Engineering*, 46(12):1267–1293, 2018.

[WS00] Christopher Williams and Matthias Seeger. Using the nyström method to speed up kernel machines. *Advances in neural information processing systems*, 13, 2000.

[WS08] Chadd Williams and Jaime Spacco. Szz revisited: Verifying when changes induce fixes. In *Proceedings of the 2008 workshop on Defects in large software systems*, pages 32–36, 2008.

[ZAH11] Shahed Zaman, Bram Adams, and Ahmed E Hassan. Security versus performance bugs: a case study on firefox. In *Proceedings of the 8th working conference on mining software repositories*, pages 93–102, 2011.

[ZAH12] Shahed Zaman, Bram Adams, and Ahmed E Hassan. A qualitative study on performance bugs. In *2012 9th IEEE working conference on mining software repositories (MSR)*, pages 199–208. IEEE, 2012.