Feature Defect Prediction

Stefan Strüder
University of Koblenz-Landau, Germany

Daniel Strüber
Radboud University Nijmegen, Netherlands

Thorsten Berger
Chalmers | University of Gothenburg, Sweden

Mukelabai Mukelabai
Chalmers | University of Gothenburg, Sweden

ABSTRACT

Software errors are a major nuisance in software development and can lead not only to damage to reputation but also to considerable financial losses for companies. For this reason, numerous techniques for detecting and predicting defects have been developed over the past decade, which are largely based on machine learning methods. These techniques are usually aimed at predicting defects at the file level. However, in recent years the popularity of feature-based software development has been increasing: a paradigm that relies on functional increments of a software system (features) and thus ensures a wide variability of the software product. A common implementation technique for features is based on annotations with preprocessor instructions, such as `#IFDEF` and `#IFNDEF`, whose code is spread over several files of the software source code ("code scattering"). A bug in such feature code can have far-reaching consequences for the functionality of the entire software. If a part of the feature code contains defects, the entire function of the feature becomes faulty and may lead to the failure of the entire functionality of the software (features are "cross-cutting"). This problem is the subject of this thesis. A prediction technique for defective features is developed, which is based on machine learning methods. The evaluation of eight classifiers, each based on an individual classification algorithm, shows that the feature-based dataset created for this thesis allows an accuracy of up to 84% for the prediction of defective or defect-free features. It is also shown how the feature oriented aspect was integrated into the creation of the dataset and what results were achieved compared to the traditional file-based methodology. This comparison showed that the additional inclusion of the feature aspect in the file-based defect prediction does not have a significant impact on the prediction results.

ACM Reference Format:
Stefan Strüder, Daniel Strüber, Thorsten Berger, and Mukelabai Mukelabai. 2020. Feature Defect Prediction. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 10 pages.

1 INTRODUCTION

Software errors are a significant trigger for financial damage and damage to the reputation of companies. Such errors range from minor bugs to serious security vulnerabilities. For this reason, there is a great deal of interest in warning a developer when they release updated software code that may contain one or more bugs. To this end, researchers and software developers have developed various techniques for error detection and error prediction over the past decade, which are largely based on methods and techniques of Machine Learning [3]. These techniques usually use historical data of defective and defect-free changes to software systems in combination with a carefully compiled set of attributes (usually called features[1]) to train a given classifier [1, 6]. This can then be used to make an accurate prediction of whether a new change to a piece of software is defective or clean. The choice of algorithms for classification is large. Studies show that, out of the pool of available algorithms, both tree-based (e.g. J48, CART or Random Forest) and Bayesian algorithms (e.g. Naïve Bayes (NB), Bernoulli-NB or multinomial NB) are the most widely used [16]. Alternative algorithms include regression, k-nearest-neighbors or artificial neural networks [3]. It should be noted, however, that there is no consensus on the best available algorithm, since each algorithm has different strengths and weaknesses for specific use cases. The goal of this work is to develop such a prediction technique for software errors based on software features. These features describe increments of the functionality of a software system. The software systems developed in this way are called software product lines and consist of a set of similar software products. They are characterized by having a common set of features and a common code base [18]. By having different features along the software products, a wide variability within a product line can be achieved. In the development of the prediction technique, the implementation of features is performed by means of preprocessor instructions, such as `#IFDEF` and `#IFNDEF` (also called preprocessor directives). This approach [11], which has so far only been considered once in a case study, is promising for several reasons:

(1) Since a given feature might be historically more or less error-prone, a change that updates the feature may be more or less eror-prone as well.
(2) Features more or less likely to be error-prone might have certain characteristics that can be harnessed for defect predictoin.
(3) Code that contains a lot of feature-specific code (specifically, the so-called feature interactions) might be more error-prone than others.

The above-mentioned goal of the work consists of several subgoals. Among them is the creation of a dataset including the feature aspect. This dataset in turn serves to train a representative selection of classifiers with a subsequent comparative evaluation of these. In addition, the feature-based data set is compared with a classical file-based data set, the development of which was taken from a scientific paper [9].

2 BACKGROUND / RELATED WORK

Lorem ipsum.

[1]To avoid ambiguous and misleading use of the feature term, the term "attribute" will be used throughout this paper to describe the characteristics of data.

Conference'17, July 2017, Washington, DC, USA
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00

3 METHODOLOGY

3.1 Creation of Dataset

The data set forms the basis for the training of the Machine Learning Classifiers and is created specifically for this work using commit data from 13 feature-based software projects. The software projects are selected based on previous use in scientific literature [7, 8, 11, 12]. However, it was also important that the variability in the source code of the software projects was implemented by means of preprocessor directives and that their git repositories had a clear "revision history" regarding release numbers. Another sufficient criterion was the use of a common language. All software projects originate from the English-speaking world. The software projects used for this work are listed in Table 1 together with their purpose and data sources.

Table 1: Used software projects

Project	Purpose	Data source
Blender	3D modelling tool	GitHub mirror
Busybox	UNIX toolkit	Git repository
Emacs	text editor	GitHub mirror
GIMP	graphics editor	GitLab repository
Gnumeric	spreadsheet	GitLab repository
gnuplot	plotting tool	GitHub mirror
Irssi	IRC client	GitHub repository
libxml2	XML parser	GitLab repository
lighttpd	web server	Git repository
MPSolve	polynom solver	GitHub repository
Parrot	virtual machine	GitHub repository
Vim	text editor	GitHub repository

To retrieve the commit data of the software projects the Python library PyDriller[2] was used [17]. This allows easy data extraction from Git repositories to obtain commits, commit messages, developers, diffs, and more (called "metadata" in the following). The URLs to the Git repositories of the software projects were used as input for the specially created Python scripts for receiving the commit metadata. Furthermore, the metadata was divided into commits per release. This was made possible by specifying release tags in the PyDriller code, based on the tag structure of Git repositories. For each modified file within a commit and a release, the following metadata was retrieved using PyDriller:

- commit hash (unique identifier of a commit)
- commit author
- commit message
- filename
- lines of code
- cyclomatic complexity
- number of added lines
- number of removed lines
- diff (changeset)

[2]https://github.com/ishepard/pydriller

The data obtained in this way was stored in a MySQL database after retrieval. For each software project, a separate table was created in which, in addition to the metadata above, the name of the software project and the release numbers associated with the commits were stored. Each modified file of a commit receives one row of the database tables. The further construction of the data set is divided into several phases of data processing and optimization.

The first phase consists of extracting the features involved in a modified file. This was done by using a Python script to identify the preprocessor statements `#IFDEF` and `#IFNDEF` in the diffs of the modified files, and then saving the string following the directives as a feature until the end of the line of code. The identification was done using regular expressions. The features identified per file are stored in an additional column in the respective MySQL tables. In case a feature is identified after the `#IFNDEF` directive, the feature is stored with a preceding "not". It will be saved as a separate feature, along with its non-negated form. Combinations of features are stored in their identified form. If no feature could be identified, `"none"` is saved accordingly.

This way of identification has some obstacles. In some C programming paradigms, it is common to include header files in the source code using preprocessor directives, so that they appear to be features. However, these "header features", as they will be referred to later, should be ignored as they do not create variability throughout the source code. In general, these header features are identifiable by their naming in the form of an appended `_h_` to the feature name, such as `featurename_h_`. This appended part allows the header features to be identified and filtered out using regular expressions. It is also possible that "wrong" features can be identified. Examples of this can come from `#IFDEFs` used in comments. Such false features were removed in a manual review of the identified features and replaced with `"none"`.

The next phase of processing consists of identifying corrective commits. A common method used for this, and one that was used in this paper, is to analyze commit messages for the presence of certain keywords [20]. The keywords used are "bug", "bugs", " bugfix", "error", "fail", "fix", "fixed" and "fixes". The analysis was performed using a Python script that checks whether any of the keywords are in the commit message alone or within a combination of words. The identification was limited to the first line of each commit message, since these contain the relevant information of the message. The results were stored in another column ("corrective") of the MySQL tables (true = corrective, false = uncorrective).

The search for corrective commits is followed by an analysis for commits that introduced bugs. A PyDriller implementation of the SZZ algorithm according to Sliwerski, Zimmermann and Zeller was used [15, 17]. This algorithm allows to find commits that introduce bugs in locally stored software repositories [2]. It requires that the corrective commits have already been identified, as they serve as the algorithm's input [2].

An overview of the number of corrective and bug-introducing commits and the number of features identified per software project is given in Table 2.

A real example from the data of the Vim software project, showing the diffs of a corrective (A) and a bug-introducing (B) commit to a feature `FEAT_TEXT_PROP`, is shown in Figure 1. The section of the diff shows that the method call `vim_memset` has been replaced

Table 2: Number of corrective and bug-introducing commits and number of identified features

Project	#corrective	#bug-introducing	#features
Blender	7.760	3.776	1.400
Busybox	1.236	802	628
Emacs	4.269	2.532	718
GIMP	1.380	854	204
Gnumeric	1.498	1.191	637
gnuplot	854	1.215	558
Irssi	52	22	9
libxml2	324	88	200
lighttpd	1.078	929	230
MPSolve	151	211	54
Parrot	3.109	3.072	397
Vim	371	696	1.158

with different arguments. According to the associated commit message, the original method call caused a "memory access error". This commit was identified as corrective because the commit message contains the keyword "error". The corresponding entry in Vim's main table thus gets the value `true` in the "corrective" column. Using the SZZ algorithm, specifying the commit hash of the corrective commit, the error-initiating commit (B) of the file concerned. In its portion of the diff, you can see that this commit has put the feature `FEAT_TEXT_PROP` in the file with the incorrect method call. As a result, it is assigned the value `true` in the "bug_introducing" column in the main table.

3.2 Selection of Metrics

As already mentioned, attributes are used to train the machine learning classifiers. In this scenario, so-called metrics are used as attributes. Metrics are numerical values that quantify properties of a software project. In this case, the metrics are divided into the usual categories code metrics and process metrics and each is calculated using the existing raw data of the main tables [13]. Code metrics are used to measure properties of source code, such as "size" or complexity [13]. Process metrics are used to measure properties that can be discussed using metadata from software repositories [13]. Examples include the number of changes made to a particular file or the number of active developers on a project. For this work, eleven feature-based metrics were calculated, divided into seven process and four code metrics. Five of the process metrics were taken from scientific papers [11, 13]. The other six metrics were calculated based on the commit metadata obtained from PyDriller. A list of the eleven metrics and their descriptions can be found in Table 4. The individual metrics were calculated either directly using SQL queries or combined using SQL queries and calculations by a Python script.

In view of the evaluation of the feature-based dataset that follows in the next section, two additional datasets were created to make the impact of the feature-based metrics on the classifiers' predictions comparable. Both further data sets follow a classical file-based approach as it is common in machine learning based error detection and are based on the methodology developed by Moser et al. [9]. The 17 process metrics of this scientific work were also adopted and are listed in Table 5. A visualization to distinguish the three data sets is shown in Figure 2.

The figure shows the ways of creating the feature-based dataset (path A - G) and the file-based data sets according to [9].These are divided into the "simple" file-based dataset (paths A, B, H, I) and the extended file-based dataset (combination of both paths). The creation of the feature-based dataset has already been explained in detail up to this section. More information about finalizing the datasets will follow as this section progresses.

The creation of the file-based datasets is also based on the raw data of the software project commits obtained with PyDriller and the modified files (A + B) extracted from them. Thus no further processing of the raw data is necessary. They then serve as a basis for the calculation of the 17 metrics (H) according to [9]. To calculate the time-based metrics `AAGE` and `WAGE`, the raw data or the commits listed in the raw data had to be supplemented with their publication date. This was also done with PyDriller. The date of the first commit of a release was chosen as the starting point for calculating the past weeks.

The feature-based and the "simple" file-based dataset (G + I) consist of the calculated feature- (F) and file-related (H) metrics and the labels of the target class. The values of the metrics are calculated for the data of each software project. The resulting tables contain as columns the values of the eleven or 17 metrics and the label (target class) "defective" or "clean" and as rows the features or files aggregated by release. This means that if a feature or file has been edited multiple times in a release (i.e. it is edited in multiple commits), the average value of the respective metrics within the release is calculated and stored. The procedure for determining the label is performed for each change to a feature or file using the following pattern:

bug-introducing	+	corrective	=	defective
bug-introducing	+	not corrective	=	defective
not bug-introducing	+	corrective	=	clean
not bug-introducing	+	not corrective	=	clean

The information on the status of a feature is based on the files on which it is based. If a feature or a file has been edited several times within a release, the label is determined according to the following rules:

- in the case of features, it is checked whether the feature was marked as "defective" at least once within the release. If this is the case, it is assumed that the feature is broken in the release in question. If this is not the case, the feature is assumed to be clean.
- in the case of files, the last commit of the files in that release is checked. If the file is marked "clean" there, it is assumed to be error-free. If it is marked as "defective", it is assumed that it is defective in this release.

The individual tables created in this way are then concatenated into a common table, resulting in a comprehensive listing of metrics including the associated labels. This list specifies the characteristics a feature or file must have in order to be classified as "defective" or "clean" and serves as a training basis for classifiers for future predictions.

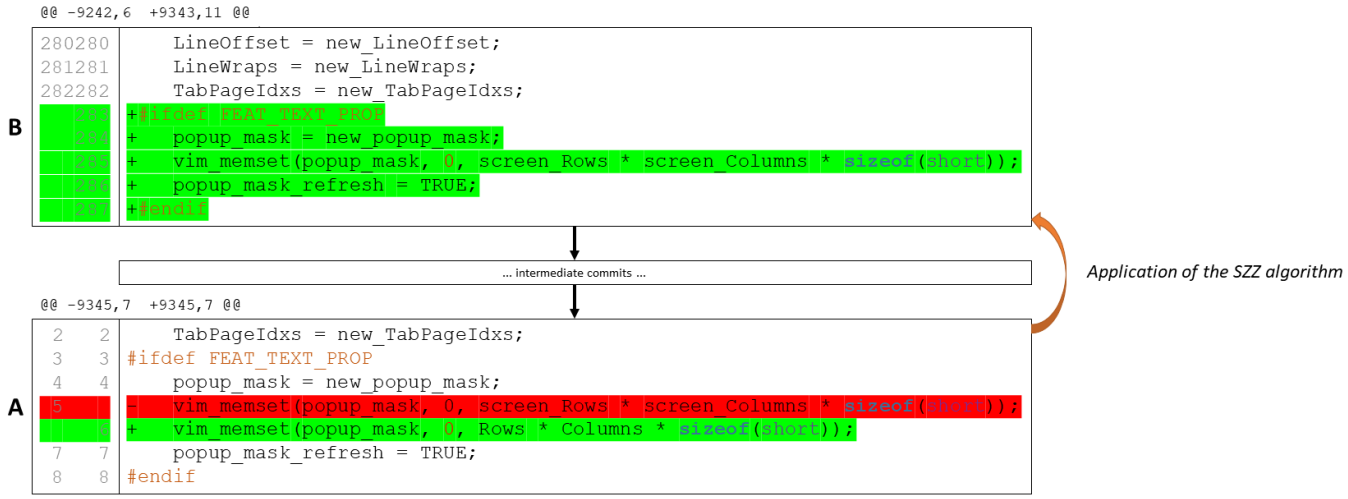


Figure 1: Real example of a defect with corrective (A) and bug-introducing (B) commit

Table 3: Overview of used feature-based metrics

	Metric	Description	Source
Process metrics	Number of commits	number of commits associated with the feature in a release.	[11, 13]
	Number of active developers	number of developers who have edited (changed, deleted or added) the feature within a release.	[11, 13]
	Number of distinct developers	cumulative number of developers who have edited (changed, deleted or added) the feature within a release.	[11, 13]
	Experience of all develepoers	geometric mean of the experience of all developers who have edited (changed, deleted or added) the feature within a release. Experience is defined as the sum of the changed, deleted or added lines in the commits associated with the feature.	[11, 13]
	Experience of the most involved developers	experience of the developer who has edited (changed, deleted or added) the feature most often within a release. Experience is defined as the sum of changed, deleted, or added lines in the commits associated with the feature.	[11, 13]
	Degree of modifications	number of edits (change, removal, extension) of the feature within a release.	*
	Scope of modifications	number of edited features within a release (feature overlapping value). Idea: The more features have been edited in a release, the more error-prone they seem to be.	*
Code metrics	Lines of code	average number of lines of code of the files associated with the feature within a release.	*
	Cyclomatic Complexity	average cyclomatic complexity of the files associated with the feature within a release.	*
	Number of added lines	average number of lines of code added to the files associated with the feature within a release.	*
	Number of removed lines	average number of lines of code deleted from the files associated with the feature within a release.	*
** These values were calculated based on the metadata obtained with PyDriller. Feature-level metrics were calculated based on the metadata of the underlying files.*			

A special feature is the second file-based data set created for the comparison within the evaluation (J). It is based on the "simple" file-based dataset (I) with the metrics from [9], but additionally includes the eleven metrics of the feature-based dataset (G) from Table 4. For this purpose, the values of the feature-based metrics were mapped or transferred at file level. This means that for each file of a release referenced in the file-based dataset, it was analyzed which features were mentioned in the respective file. From these features, the corresponding metrics of the feature-based dataset were determined and the average value was calculated and entered into the entered dataset. If no features were mentioned in a file, the value 0 is stored for the feature-based metrics. As mentioned at the beginning of this section, this way the feature-based dataset can be compared with a classical file-based dataset, because the conditions

Table 4: Feature metrics. To apply the metrics to a file, we aggregate the metric values of all features associated to the given file, using the listed aggregation operator.

	Metric	Description	Aggregation operator	Source
Process metrics	Number of commits (FCOMM)	number of commits associated with the feature in a release.	Mean	[11, 13]
	Number of active developers (FADEV)	number of developers who have edited (changed, deleted or added) the feature within a release.	Mean	[11, 13]
	Number of distinct developers (FDDEV)	cumultative number of developers who have edited (changed, deleted or added) the feature within a release.	Mean	[11, 13]
	Experience of all develepoers (FEXP)	geometric mean of the experience of all developers who have edited (changed, deleted or added) the feature within a release. Experience is defined as the sum of the changed, deleted or added lines in the commits associated with the feature.	Mean	[11, 13]
	Experience of the most involved developers (FOEXP)	experience of the developer who has edited (changed, deleted or added) the feature most often within a release. Experience is defined as the sum of changed, deleted, or added lines in the commits associated with the feature.	Mean	[11, 13]
	Degree of modifications (FMODD)	number of edits (change, removal, extension) of the feature within a release.	Mean	*
	Scope of modifications (FMODS)	number of edited features within a release (feature overlapping value). Idea: The more features have been edited in a release, the more error-prone they seem to be.	Mean	*
Code metrics	Lines of code (FNLOC)	average number of lines of code of the files associated with the feature within a release.	Mean	*
	Cyclomatic Complexity (FCyCO)	average cyclomatic complexity of the files associated with the feature within a release.	Mean	*
	Number of added lines (FADDL)	average number of lines of code added to the files associated with the feature within a release.	Mean	*
	Number of removed lines (FREML)	average number of lines of code deleted from the files associated with the feature within a release.	Mean	*
** These values were calculated based on the metadata obtained with PyDriller. Feature-level metrics were calculated based on the metadata of the underlying files.*				

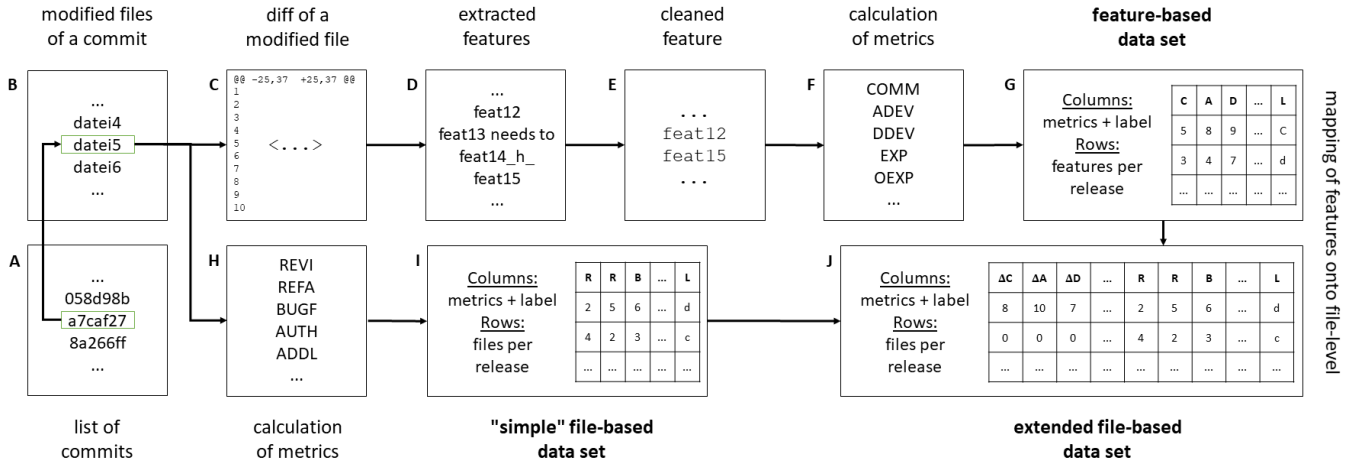


Figure 2: Visualization to distinguish the three data sets

for the comparison have been created. A direct comparison between the different datasets is not practical for the reasons mentioned above.

Some supplementary key figures for the datasets are listed in Table 6. The row "unique" indicates how many unique rows are contained in the datasets. The datasets created in the manner described above can now be used to train the classifiers. This process will be explained in the next section.

3.3 Selection of Classifiers

Although programming with the Python programming language was often used to create the data sets, the choice of a machine learning tool was not the library scikit-learn [10], but an independent solution. The WEKA-Workbench[3] is used as a machine learning tool. This tool proved to be suitable for the underlying task by numerous citations in scientific papers (among others in [6, 11, 14]). The WEKA-Workbench (WEKA as acronym for **W**aikato **E**nvironment for **K**nowledge **A**nalysis) was developed at the University of Waikato in New Zealand and offers a large collection

[3] https://www.cs.waikato.ac.nz/ml/weka/

Table 5: Process metrics according to [9]

Metric	Description
REVISIONS	Number of revisions of a file.
REFACTORINGS	Number of times a file has been refactored.
BUGFIXES	Number of times a file was involved in bug-fixing.
AUTHORS	Number of distinct authors that checked a file into the repository.
LOC_ADDED	Sum over all revisions of the lines of code added to a file.
MAX_LOC_ADDED	Maximum number of lines of code added for all revisions.
AVE_LOC_ADDED	Average lines of code added per revision.
LOC_DELETED	Sum over all revisions of the lines of code deleted from a file.
MAX_LOC_DELETED	Maximum number of lines of code deleted for all revisions.
AVE_LOC_DELETED	Average lines of code deleted per revision.
CODECHURN	Sum of (added lines of code – deleted lines of code) over all revisions.
MAX_CODECHURN	Maximum CODECHURN for all revisions.
AVE_CODECHURN	Average CODECHURN per revision.
MAX_CHANGESET	Maximum number of files committed together to the repository.
AVE_CHANGESET	Average number of files committed together to the repository.
AGE	Age of a file in weeks (counting backwards from a specific release).
WEIGHTED_AGE	$WeightedAge = \frac{\sum_{i=1}^{N} Age(i) * LOC\_ADDED(i)}{\sum_{i=1}^{N} LOC\_ADDED(i)}$

Table 6: Key figures of the data sets

	feature-based data set	"simple" file-based data set	"extended" file-based data set
#attributes	11 + label	17 + label	28 + label
#data records	14.059	76.986	76.986
thereof defective	2.735	1.899	1.899
thereof clean	11.324	75.087	75.087
thereof unique	8.447	52.564	52.783

of machine learning algorithms and preprocessing tools for use within a graphical user interface [5]. There are also interfaces for the Java programming language [5]. An overview of the selected classification algorithms can be found in Table 7. This also includes the abbreviations of the classifiers that will be used in the following.

All classification algorithms presented above are already integrated in the WEKA tool. It receives as input the final data sets in a proprietary file format. The 13 calculated metrics form the attributes, whereas the target class is represented by the labels "defective" and "clean".

The training process of the classifiers took place within the graphical user interface of WEKA. Before the training, the split ratios for the division of the data sets into training data and test data were determined. It was determined individually for each of the software projects used and is based on the number of releases available in each case. However, care was always taken to approximate the commonly used split ratios of 80:20 and 75:25 as well as 70:30. The training data ranges from 67% to 80% of the data records of the data sets. The earlier releases were assigned to the training data. The training data contains the data records of the later releases. An overview of the division into training and test data per software project is shown in Figure 3. The resulting split ratios for the entire data sets are: 69:31 (feature-based data set) and 71:29 ("simple" and extended file-based data set).

The training of the classifiers in WEKA was carried out for each classification algorithm with the respective standard settings. Only for the algorithms NN and RF further configurations were made. For the RF-algorithm a number of instances of 200 was defined. This means that 200 decision trees perform parallel processing at the same time. There are no clear recommendations on how many instances should be selected. The selected value of 200 was therefore determined independently, taking into account the scope of the data sets and the high number of attributes. For the NN algorithm an independently determined hidden layer structure of (13,13,13 was chosen. This means that the artificial neural network has three hidden layer layers of 13 hidden layer neurons each. This allows them to process the large number of attributes more efficiently. Furthermore, no validation data was generated, since it is not intended to perform attribute selection or to adjust further classifier settings.

An analysis of the training process also showed that the file-based data sets are highly unbalanced with regard to the target class. With a value of about 97%, there are far more entries assigned to the label "clean". Balancedness, i.e. a balanced ratio (50:50 is not mandatory in the binary case) within the target classes, is however a prerequisite for the correct training of most classifiers. Ignoring this problem can lead to misleading accuracy, since most records are correctly assigned to the over-represented class and is a fundamental problem of accuracy metrics. To solve this problem, the SMOTE algorithm was applied to the file-based datasets [4]. The algorithm, whose acronym stands for **S**ynthetic **M**inority **O**ver-sampling **Te**chnique, performs an oversampling of the underrepresented class [4]. Using next-neighbor calculations based on the Euclidean distance between the attribute values of each dataset's datasets, new synthetic datasets are added (oversampling) so that the number of datasets of the relevant class increases [4]. In this case, the percentage of synthetic records generated is 2000, so for each existing record of the underrepresented class, 20 additional synthetic records are generated. Thus, the percentage of records with the label "defective" was increased to about 41%. At the same time the number of records grew by about 60%. The algorithm was only applied to the training data [4] . It is not intended to be applied to the test data so that the "ground truth" is not distorted or falsified. As a result, the metrics of the file-based datasets also changed. These are shown in Table 8.

The results obtained using the test data, which reflect the performance of the individual classifiers, are presented in the following chapter as part of the evaluation.

4 EVALUATION

4.1 Evaluation metrics

Lorem ipsum.

4.2 Results

Lorem ipsum.

Feature-based data set. Lorem ipsum.

File-based data sets. Lorem ipsum.

5 DISCUSSION

5.1 Challenges and limitations

Lorem ipsum.

Identification of features. The basic question that arose in the process of identifying the features was: "What is counted as a feature? As mentioned above, the identification of features presented some

Table 7: Selection of classification algorithms

Algorithm	Abbreviation
J48 Decision Trees	J48
k-Nearest-Neighbors	KNN
Logistic Regression	LR
Naïve Bayes	NB
Artificial Neural Networks	NN
Random Forest	RF
Stochastic Gradient Descent	SGD
Support Vector Machines	SVM

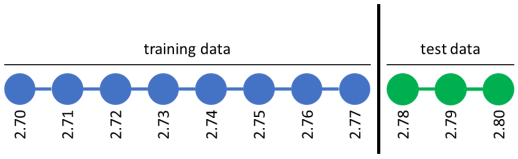
Table 8: Key figures of the data sets

	"simple" data set		extended data set	
	before	after	before	after
#attributes	17 + label	17 + label	28 + label	28 + label
#data records	76.986	111.706	76.986	112.706
thereof defective	1.899	37.619	1.899	37.619
thereof clean	75.087	75.087	75.087	75.087
thereof unique	52.564	86.155	52.783	86.381
overall split ratio	71:29	81:19	71:29	81:19

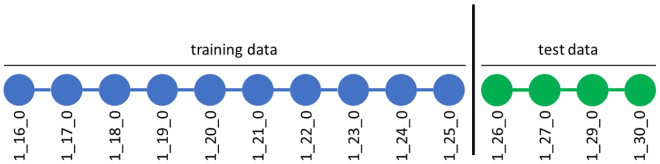
Table 9: Let R be a release consisting of q commits: $R = \{c_1, c_2, ...c_q\}$, F be the set of all p files changed by commits in R: $F = \{f_1, f_2, ...f_p\}$. For each file $f \in F$, let T be the set of all n features (in f) affected by changes in R: $T = \{feat_1, feat_2, ...feat_n\}$, and each feature $feat \in T$ has a set A of m files which implement it: $A \subseteq F$: $A = \{featfile_1, featfile_2, ...featfile_m\}$ We define our feature based metric as follows for each file f:

metric	description	equation	supplementary eq.
$FCOMM$[1]	Average number of commits associated to the changed features of a file within a release.	$FCOMM_R(f) = \frac{1}{n}\sum_{i=1}^{n} comm(feat_i, A_i)$	$comm(feat_i, A_i) = \sum_{j=1}^{m} comm(featfile_j)$
$FADEV$[2]	Average number of developers who changed the features of a files within a release.	$FADEV_R(f) = \frac{1}{n}\sum_{i=1}^{n} adev(feat_i, A_i)$	$adev(feat_i, A_i) = \sum_{j=1}^{m} adev(featfile_j)$
$FDDEV$[3]	Average cumulated number of developers who changed the features of a files within a release.	$FDDEV_R(f) = \frac{1}{n}\sum_{i=1}^{n} ddev(feat_i, A_i)$	$ddev(feat_i, A_i) = \sum_{j=1}^{m} ddev(featfile_j)$
$FEXP$[4]	Average experience[5] of all developers who changed the features of a file within a release.	$FEXP_R(f) = \frac{1}{n}\sum_{i=1}^{n} exp(feat_i, A_i)$	$exp(feat_i, A_i) = \sum_{j=1}^{m} exp(featfile_j)$
$FOEXP$[6]	Average experience[5] of the developer who changed the features of a file most often within a release.	$FOEXP_R(f) = \frac{1}{n}\sum_{i=1}^{n} oexp(feat_i, A_i)$	$oexp(feat_i, A_i) = \sum_{j=1}^{m} oexp(featfile_j)$
$FMODD$[7]	Average number of changes to the features of a file within a release.	$FMODD_R(f) = \frac{1}{n}\sum_{i=1}^{n} modd(feat_i, A_i)$	$modd(feat_i, A_i) = \sum_{j=1}^{m} modd(featfile_j)$
$FMODS$[8]	Average number of changed features of a file within a release.	$FMODS_R(f) = \frac{1}{n}\sum_{i=1}^{n} mods(feat_i, A_i)$	$mods(feat_i, A_i) = \sum_{j=1}^{m} mods(featfile_j)$
$FNLOC$[9]	Average number of lines of code of the underlying files of the changed features of a file within a release.	$FNLOC_R(f) = \frac{1}{n}\sum_{i=1}^{n} nloc(feat_i, A_i)$	$nloc(feat_i, A_i) = \sum_{j=1}^{m} nloc(featfile_j)$
$FCYCO$[10]	Average cyclomatic complexity of the underlying files of the changed features of a file within a release.	$FCYCO_R(f) = \frac{1}{n}\sum_{i=1}^{n} cyco(feat_i, A_i)$	$cyco(feat_i, A_i) = \sum_{j=1}^{m} nloc(featfile_j)$
$FADDL$[11]	Average number of lines of code added to the underlying files of the changed features of a file within a release.	$FADDL_R(f) = \frac{1}{n}\sum_{i=1}^{n} addl(feat_i, A_i)$	$addl(feat_i, A_i) = \sum_{j=1}^{m} addl(featfile_j)$
$FREML$[12]	Average number of lines of code deleted from the underlying files of the changed features of a file within a release.	$FREML_R(f) = \frac{1}{n}\sum_{i=1}^{n} reml(feat_i, A_i)$	$reml(feat_i, A_i) = \sum_{j=1}^{m} reml(featfile_j)$

[1] $comm(featfile_j)$ counts commits in which the file $featfile_j$ was changed. [2] $adev(featfile_j)$ counts the developers who changed the file $featfile_j$. [3] $ddev(featfile_j)$ cumulates the counts of developers who changed the file over commits $featfile_j$. [4] $exp(featfile_j)$ returns the geometric mean of the experience[5] of all developers who changed the file $featfile_j$. [5] Experience is defined as the sum of the changed, deleted or added lines in the commits associated with the files. [6] $oexp(featfile_j)$ returns the experience[5] of the developer who changed the file $featfile_j$ most often within a release. [7] $modd(featfile_j)$ returns the number of changes to the file $featfile_j$. [8] $mods(featfile_j)$ returns the number of changed files in the release of the file $featfile_j$. [9] $nloc(featfile_j)$ returns the number of lines of code of the file $featfile_j$. [10] $cyco(featfile_j)$ returns the cyclomatic complexity of the file $featfile_j$. [11] $addl(featfile_j)$ returns the number of added lines of code to the file $featfile_j$. [12] $reml(featfile_j)$ returns the number of removed lines of code from the file $featfile_j$.

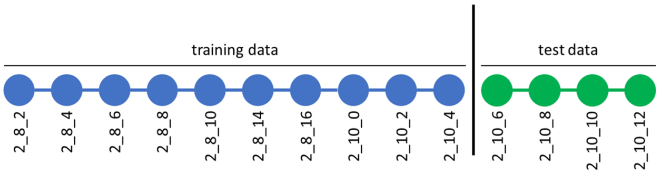


(a) Blender
split ratio: 73:27

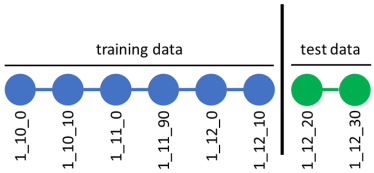


(b) Busybox
split ratio: 71:29

(c) Emacs
split ratio: 71:29



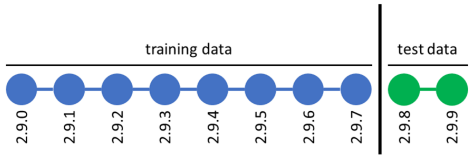
(d) GIMP
split ratio: 71:29



(e) Gnumeric
split ratio: 75:25

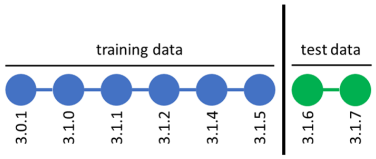
(f) gnuplot
split ratio: 80:20

(g) Irssi
split ratio: 71:29

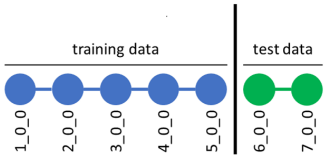


(h) libxml2
split ratio: 80:20

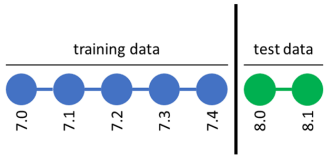
(i) lighttpd
split ratio: 67:33



(j) MPSolve
split ratio: 75:25



(k) Parrot
split ratio: 71:29



(l) Vim
split ratio: 71:29

Figure 3: Overview of the division into training and test data

challenges. The first challenge was the filtering out of "header features", which are used in some programming paradigms to include header files in the source code. However, these header features do not create variability in the code, so they are undesirable. Most of these header features were identifiable by their assigned names, which had a _h_. This way they could be quickly identified and filtered out using regular expressions. However, it is possible that in some software projects the header features are not explicitly identified by their names. This makes it difficult to identify them, for example by manually viewing the contexts of the features in the source code. In this case, however, this would be very time-consuming due to the large number of features and was therefore not performed. The removal of the recognizable header features showed that a considerable part of the previously identified features was undesirable. This method therefore proved to be effective. A solution that could contribute to the above problem would be a tool for parsing source code to correctly identify features by means of automated analysis of the context of the feature. This would also detect the "wrong" features mentioned above. Such a tool does not exist at the moment. Available feature identification tools use a similar approach as used in this thesis (regular expressions).

Integration of the reference to features. As mentioned above, the reference to the features is based on the underlying files. Therefore the diffs of the modified files were analyzed. Thus, a feature is considered relevant if it is mentioned in a diff. However, it may also be the case that the feature code it contains was not involved in the change described in the diff, but was only mentioned in the part of the diff referred to as "hunk". This denotes an overhang of the actual context of the described changes in the form of additional lines of code that follow this change. Thus, there is no "in-depth" analysis of the source code. However, this approach was also chosen by the scientific work underlying this paper [11]. Also, the metrics of the features are calculated based on the metadata of the underlying files. Thus an over-approximation takes place.

Heuristic for identifying corrective Commits. The heuristic for detecting corrective commits was changed during the course of the work. First, the entire commit message was analyzed for the presence of the keywords. However, this resulted in some commits being incorrectly identified as corrective. The reason for this was that the commit messages in some of the software projects used were very large in word count. These messages mentioned all the changes made by the commits, without exception. However, most of these changes were irrelevant for this purpose. It was found that the main statement or reason for the commit was in the first line of the commit message. The heuristics were then adjusted accordingly. A sample of corrective commits before and after the heuristic adjustment showed that the change caused irrelevant corrective commits to be removed.

Imprecision of the SZZ-Algorithm. A limitation, which was established in the course of the creation of the data set through literature research, refers to the SZZ algorithm. This was used to identify commits that introduced bugs based on the commit hashes of the corrective commits. Analysis of the algorithm revealed that currently available implementations, and thus those of the Python tool PyDriller, can identify only about 69% of the actual number of bug-introducing commits that exist [19]. In addition, about 64% of the identified commits were found to be incorrectly identified [19]. The algorithm is therefore considered imprecise [19]. The reasoning behind this is as follows:

> The reason is that the implicit assumptions of the SZZ algorithm are violated by the insufficient file coverage and statement direct coverage between bug-inducing and bug-fixing commits. - [19]

Furthermore, the authors of the study found in tests conducted by themselves that the results of eight out of ten earlier studies were significantly influenced by the imprecise algorithm [19]. This may therefore also apply to this work. However, there is currently no alternative method for identifying bug-introducing commits. Should a new method or an improved version of the SZZ algorithm be published, it would be a good idea to repeat the main steps of this work, taking the new method into account, to compare it with the results presented here in order to highlight the influences of the SZZ algorithm.

6 CONCLUSION

Lorem ipsum.

REFERENCES

[1] Abdullah Alsaeedi and Mohammad Zubair Khan. 2019. Software Defect Prediction Using Supervised Machine Learning and Ensemble Techniques: A Comparative Study. *Journal of Software Engineering and Applications* 12, 05 (2019), 85–100. https://doi.org/10.4236/jsea.2019.125007

[2] Markus Borg, Oscar Svensson, Kristian Berg, and Daniel Hansson. 2019. 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 - MaLTeSQuE 2019*. ACM Press. https://doi.org/10.1145/3340482.3342742

[3] Venkata Udaya B. Challagulla, Farokh B. Bastani, I. Ling Yen, and Raymond A. Paul. 2008. Empirical assessment of machine learning based software defect prediction techniques. *International Journal on Artificial Intelligence Tools* 17, 2 (2008), 389–400. https://doi.org/10.1142/S0218213008003947

[4] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. 2002. SMOTE: Synthetic Minority Over-sampling Technique. *Journal of Artificial Intelligence Research* 16 (jun 2002), 321–357. https://doi.org/10.1613/jair.953

[5] Eibe Frank, Mark A. Hall, and Ian H. Witten. 2016. *The WEKA Workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques"* (fourth edition ed.). Morgan Kaufmann.

[6] Awni Hammouri, Mustafa Hammad, Mohammad Alnabhan, and Fatima Alsarayrah. 2018. Software Bug Prediction using Machine Learning Approach. *International Journal of Advanced Computer Science and Applications* 9, 2 (2018). https://doi.org/10.14569/ijacsa.2018.090212

[7] Claus Hunsen, Bo Zhang, Janet Siegmund, Christian Kästner, Olaf Leßenich, Martin Becker, and Sven Apel. 2015. Preprocessor-based variability in open-source and industrial software systems: An empirical study. *Empirical Software Engineering* 21, 2 (apr 2015), 449–482. https://doi.org/10.1007/s10664-015-9360-1

[8] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*. ACM Press. https://doi.org/10.1145/1806799.1806819

[9] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. 2008. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 13th international conference on Software engineering - ICSE '08*. ACM Press. https://doi.org/10.1145/1368088.1368114

[10] 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. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[11] Rodrigo Queiroz, Thorsten Berger, and Krzysztof Czarnecki. 2016. Towards predicting feature defects in software product lines. In *Proceedings of the 7th*

International Workshop on Feature-Oriented Software Development - FOSD 2016. ACM Press. https://doi.org/10.1145/3001867.3001874

[12] Rodrigo Queiroz, Leonardo Passos, Marco Tulio Valente, Claus Hunsen, Sven Apel, and Krzysztof Czarnecki. 2015. The shape of feature code: an analysis of twenty C-preprocessor-based systems. *Software & Systems Modeling* 16, 1 (jul 2015), 77–96. https://doi.org/10.1007/s10270-015-0483-z

[13] Foyzur Rahman and Premkumar Devanbu. 2013. How, and why, process metrics are better. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. https://doi.org/10.1109/icse.2013.6606589

[14] Jacek Ratzinger, Thomas Sigmund, and Harald C. Gall. 2008. On the relation of refactorings and software defect prediction. In *Proceedings of the 2008 international workshop on Mining software repositories - MSR '08*. ACM Press. https://doi.org/10.1145/1370750.1370759

[15] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes? *ACM SIGSOFT Software Engineering Notes* 30, 4 (jul 2005), 1. https://doi.org/10.1145/1082983.1083147

[16] Le Son, Nakul Pritam, Manju Khari, Raghvendra Kumar, Pham Phuong, and Pham Thong. 2019. Empirical Study of Software Defect Prediction: A Systematic Mapping. *Symmetry* 11, 2 (feb 2019), 212. https://doi.org/10.3390/sym11020212

[17] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2018. PyDriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*. ACM Press. https://doi.org/10.1145/3236024.3264598

[18] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *Comput. Surveys* 47, 1 (jul 2014), 1–45. https://doi.org/10.1145/2580950

[19] Ming Wen, Rongxin Wu, Yepang Liu, Yongqiang Tian, Xuan Xie, Shing-Chi Cheung, and Zhendong Su. 2019. Exploring and exploiting the correlations between bug-inducing and bug-fixing commits. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2019*. ACM Press. https://doi.org/10.1145/3338906.3338962

[20] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. 2007. Predicting Defects for Eclipse. In *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*. IEEE. https://doi.org/10.1109/promise.2007.10