Chapter 1

Feature Generation and Engineering for Software Analytics

Xin Xia

Faculty of Information Technology, Monash University, Australia

David Lo

School of Information Systems, Singapore Management University, Singapore

Abstract

This chapter provides an introduction on feature generation and engineering for software analytics. Specifically, we show how domain-specific features can be designed and used to automate three software engineering tasks: (1) detecting defective software modules (defect prediction), (2) identifying crashing mobile app release (crash release prediction), and (3) predicting who will leave a software team (developer turnover prediction). For each of the three tasks, different sets of features are extracted from a diverse set of software artifacts, and used to build predictive models.

1.1 Introduction

As developers work on a project, they leave behind many digital artifacts. These digital trails can provide insights into how software is developed and provide a rich source of information to help improve development practices. For instance, GitHub hosts more than 57M repositories, and is currently used by more than 20M developers [1]. As another example, Stack Overflow has more than 3.9M registered users, 8.8M questions, and 41M comments [58]. The productivity of software developers and testers can be improved using information from these repositories.

There have been a number of studies in software engineering which focus on building predictive models by mining a wide variety of software data collected from systems, their repositories and relevant online resources [3, 6, 35, 59, 60]. For example, in defect prediction [35,60], developers aim to predict whether a module/class/method/change contains bugs, and they build a predictive model by extracting features from historical modules/classes/methods/changes with known labels (i.e., buggy or clean). In bug priority prediction [53], developers aim to predict the priority level of a bug when it is submitted, and they build a predictive model by leveraging features from historical bug reports with known priority levels. In practice, the performance of a predictive model will be largely affected by the features used to build the model. For example, Rahman and Devanbu investigated different types of features on the performance of defect prediction, and they found that process features performed better than the code features in defect prediction [45]. However, feature identification and generation from software artifacts and repositories is challenging since (1) software engineering data are complex, and (2) it requires domain knowledge to identify effective features.

Features can be extracted from various types of software artifacts, e.g., source code, bug reports, code reviews, commit logs, and email lists. Even in

the same software artifacts, there are various ways to extract features. For example, to extract features from source code, trace features (e.g., statement coverage) can be extracted by running the source code and analyzing its execution trace, code features (e.g., code complexity) by leveraging static analysis tools (e.g., SciTool[1]), textual features (e.g., readibility and term frequency) by using text mining techniques, and process features (e.g., number of developers who changed the code) by mining the change history of the code.

In this chapter, we aim to provide an introduction on feature generation and engineering for software analytics, and show how domain-specific features are extracted and used for three software engineering use cases, i.e., defect prediction, crash release prediction, and developer turnover prediction. These three case studies extract different kinds of features from software artifacts, and build predictive models based on these features. Some features used in these three case studies are related, while others are problem-specific.

The remainder of the chapter is structured as follows. Section 1.2 describes features used in defect prediction. Section 1.3 presents features used in crash release prediction for apps. Section 1.4 elaborates features generated from monthly report for developer turnover prediction. Section 1.5 concludes the chapter and discusses future directions.

1.2 Features for Defect Prediction

Defect prediction techniques are proposed to help prioritize software testing and debugging; they can recommend likely defective code to developers. Most defect prediction studies proposed prediction models built on various types of features (e.g., process or code features), and predicted defects at coarse granularity level (e.g., file), which refered to as file-level defect prediction [17, 19, 28, 39, 45, 54, 60]. Besides file-level defect prediction, Mockus and Weiss proposed a prediction model which focuses on identifying defect-prone software changes instead of files or packages [37], which was also referred as just-in-time (JIT) defect prediction by Kamei et al. [24].

The difference between file-level and just-in-time defect prediction lies on the development phase when they are employed. File-level defect prediction is usually conducted before a product release. It aims to be a quality control step before a release [63]. Just-in-time defect prediction is conducted when each change is submitted. It aims to be a continuous activity of quality control, which leads to smaller amount of code to be reviewed, and developers can review and test these risky changes while they are still fresh in their minds (i.e., at commit time) [21]. They can complement each other to improve the quality of the upcoming release. Considering the difference in the usage scenarios of

[1] https://scitools.com/

these two types of defect prediction techniques, their corresponding features are different. In the remaining sections, the details of features for these two types of defect prediction techniques are introduced.

1.2.1 File-level Defect Prediction

In general, there are two types of features for file-level defect prediction: code features, which measure properties of the code (e.g., code complexity, and lines of code), and process features (e.g., developer experience, and number of changes), which are extracted from the software development process. A number of papers in the software engineering literature have investigated the effectiveness of each feature type. Menzies et al. concluded that code metrics are efffective for defect prediction [36]. Moser et al. compared the performance of code and process features on Eclipse JDT project, and they found that process features outperform code features [38]. Arisholm et al. performed an empirical study on various types of features and techniques on several releases of a Java middleware system named COS, and they found that process and code features perform similarly in terms of AUC, but process features are cost-effective [4]. Finally, Rahman and Devanbu performed a large-scale empirical study to investigate why and how process features performed better than code features [45].

1.2.1.1 Code Features

Jureczko and Madeyski proposed 20 code features to predict defective files [23]. These features have been empirically demonstrated to be effective in defect prediction [8, 23, 41]. They can be categorized according to the researchers who first proposed as follows:

1. Features proposed by Chidamber and Kemerer [9]:

1. **Weighted methods per class (WMC)**: the number of methods used in a given class.
2. **Depth of Inheritance Tree (DIT)**: the maximum distance from a given class to the root of an inheritance tree.
3. **Number of Children (NOC)**: the number of children of a given class in an inheritance tree.
4. **Coupling between object classes (CBO)**: the number of classes that are coupled to a given class.
5. **Response for a Class (RFC)**: the number of distinct methods invoked by code in a given class.
6. **Lack of cohesion in methods (LCOM)**: the number of method pairs in a class that do not share access to any class attributes.

2. A Feature proposed by Henderson-Sellers [20]:

1. **Lack of cohesion in methods (LCOM3)**: another type of lcom metric proposed by Henderson-Sellers [20], i.e.,

$$LOCM3 = \frac{\frac{1}{a}\sum_{j=1}^{a} m(A_j) - m}{1 - m} \tag{1.1}$$

   In the above equation, m is the number of methods in a class, a is the number of attributes in a class, and $m(A)$ is the number of methods that access the attribute A.

3. Features proposed by Bansiy and Davis [5]:

1. **Number of Public Methods (NPM)**: the number of public methods in a given class.
2. **Data Access Metric (DAM)**: the ratio of the number of private/protected attributes to the total number of attributes in a given class.
3. **Measure of Aggregation (MOA)**: the number of attributes in a given class which are of user-defined types.
4. **Measure of Functional Abstraction (MFA)**: the number of methods inherited by a given class divided by the total number of methods that can be accessed by the member methods of the given class.
5. **Cohesion Among Methods of Class (CAM)**: the ratio of the sum of the number of different parameter types of every method in a given class to the product of the number of methods in the given class and the number of different method parameter types in the whole class.

4. Features proposed by Tang et al. [52]:

1. **Inheritance Coupling (IC):** the number of parent classes that a given class is coupled to.
2. **Coupling Between Methods (CBM):** the total number of new or overwritten methods that all inherited methods in a given class are coupled to.
3. **Average Method Complexity (AMC):** the average size of methods in a given class.

5. Features proposed by Martin [29]:

1. **Afferent couplings (Ca)**: number of classes that depends upon a given class.

2. **Efferent couplings (Ce)**: number of classes that a given class depends upon.

6. Features proposed by McCabe [32]:

1. **McCabe's cyclomatic complexity (CC)**: CC is equal to the number of different paths in a method (function) plus one. The cyclomatic complexity of a method is defined as: $CC_M = E \times \times N + P$; where E is the number of edges of the graph, N is the number of nodes of the graph, and P is the number of connected components. Based on CC_M, two variants of CC can be computed for a class as follows:

   (a) **Maximum McCabe's cyclomatic complexity (MAX_CC)**: maximum McCabe's cyclomatic complexity (CC) score of methods in a given class.

   (b) **Average McCabe's cyclomatic complexity (AVG_CC)**: arithmetic mean of the McCabe's cyclomatic complexity (CC) scores of methods in a given class.

7. Others:

1. **Lines of Code (LOC)**: a popular feature in defect prediction, which calculates the number of lines of code of a class under investigation.

These 20 code features can be also categorized on other categories. Table 1.1 categorizes these features based on how they are derived, including complexity, coupling, cohesion, abstraction, and encapsulation.

1.2.1.2 Process Features

Various process features can be extracted for a source code file, and generally they can be grouped into 3 categories: developer's behavior, change entropy, and commit history [45].

1. Developer's Behavior: Since source code files are created/revised by different developers, features extracted from developers' commit behavior can potentially be used to predict likely buggy files. Many developer behavior features have been proposed in prior studies [4, 7, 37, 43, 45]; they include the following:

1. Number of commits made to a file (COM)

2. Number of developers who changed a file (NumDev)

3. Number of distinct developers who contributed to a file (DisDev)

4. Number of lines of code added or deleted or modified in a file in the previous release (AddLoc, DelLoc, and ModiLoc)

TABLE 1.1: List of of Code Features

Category	Code Features
Complexity	Lines of Code (LOC) Weighted Methods per Class (WMC) Number of Public Methods (NPM) Average Method Complexity (AMC) Max McCabe's Cyclomatic Complexity (Max_cc) Avg McCabe's Cyclomatic Complexity (Avg_cc) Measure of Aggregation (MOA)
Coupling	Coupling between object classes (CBO) Response of a Class (RFC) Afferent Couplings (CA) Efferent Couplings (CE) Inheritance Coupling (IC) Coupling Between Methods (CBM)
Cohesion	Lack of cohesion in methods (LCOM) Lack of cohesion in methods (LCOM3) Cohesion Among Methods of Class (CAM)
Abstraction	Depth of Inheritance Tree (DIT) Number Of Children (NOC) Measure of Functional Abstraction (MFA)
Encapsulation	Data Access Metric (DAM)

5. Geometric mean of experiences of all developers working on a file (Exp)

2. Change Entropy: Scattered changes could be more complex to manage, and prior study showed that scattered changes are good indicators of defects [19]. Rahman and Devenbu proposed a simple line based change entropy feature named SCTR, which measures the scattering of changes to a file [45]. SCTR is the standard position deviation of changes from the geographical centre theme.

3. Commit History: Features extracted from the commit history of a source code file can also potentially help to predict defective files [25]. Features in this category include:

1. Number of defects in previous version (NDPV), which measures the number of defects reported in a given file in the previous release of a project.
2. Number of commits which modified a file in the previous release (NCOM)
3. Number of commits which aimed to fix bugs in a file in the previous release (NCOMBUG)

1.2.2 Just-in-time Defect Prediction

In general, the features used by file-level defect prediction can also be adapted for just-in-time defect prediction. Also, there are some specific features for just-in-time defect prediction. Kamei et al. proposed 14 features for just-in-time defect prediction, which are divided into five dimensions: diffusion, size, purpose, history, and experience [24]. Table 1.2 presents the details of these 14 features. Features in the diffusion, size, history, and experience dimensions are similar to those originally defined for file-level defect prediction, while the feature FIX in the purpose dimension is unique to just-in-time defect prediction.

The features in diffusion dimension characterize the distribution of a change. Previous studies showed that a highly distributed change is more likely to be defective [19, 37, 40]. The features in size dimension characterize the size of a change, and a larger change is more likely to be defective since more code has to be changed or implemented [38]. The purpose dimension only consists of FIX, and it is believed that a defect-fixing change is more likely to introduce a new defect [14, 16, 44]. The features in history dimension demonstrate how developers interacted with different files in the past. As stated by Yang et al. [66], a change was more likely to be defective if the touched files have been modified by more developers [30], by more recent changes [14], or by more unique last changes [19]. The experience dimension measures a developer's experience based on the number of changes made by the developer in the past. In general, a change made by a more experienced developer is less likely to introduce defects [37].

1.2.3 Prediction Models and Results

Various prediction models can be built on these features to perform file-level or just-in-time defect prediction. Table 1.3 summaries the 31 supervised models, which are grouped into six categories, namely Function, Lazy, Rule, Bayes, Tree and Ensemble. These supervised models are commonly used in defect prediction studies [13, 18, 27, 34, 36]. And all of them were investigated in Yang et al.'s work [66], and most of them (except Random Forest) were revisited in Ghotra et al.'s work [13].

Function category contains regression models, neural networks and support vector machine, including EALR [24] (i.e., Effort-Aware Linear Regression), Simple Logistic (SL), Radial Basis Functions Network (RBFNet), and Sequential Minimal Optimization (SMO). The Lazy family represents lazy learning methods, and the K-Nearest Neighbour (IBk) method is used in this category. The Rule family represents models based on rules, including propositional rules (JRip) and ripple down rules (Ridor). Bayes family represents probability-based models, and the most popular one, namely Naive Bayes (NB) is included in this category. The Tree family represents decision tree based methods, including J48, Logistic Model Tree (LMT) and Random For-

TABLE 1.2: Summary of Features for JIT Defect Prediction.

Category	Feature	Definition
Diffusion	NS	Number of subsystems touched by the current change
	ND	Number of directories touched by the current change
	NF	Number of files touched by the current change
	Entropy	Distribution across the touched files
Size	LA	Lines of code added by the current change
	LD	Lines of code deleted by the current change
	LT	Lines of code in a file before the current change
Purpose	FIX	Whether or not the current change is a defect fix
History	NDEV	Number of developers that changed the files
	AGE	Average time interval between the last and current change
	NUC	Number of unique last changes to the files
Experience	EXP	Developers experience (number of files modified)
	REXP	Developers experience in recent years
	SEXP	Developer experience on a subsystem

est (RF). In the last family, four ensemble learning methods are investigated: Bagging, Adaboost, Rotation Forest and Rotation Subspace. Different from other models, ensemble learning models are built with multiple base classifiers.

Yang et al. compared the performance of different prediction models on just-in-time defect prediction, and they found that EALR showed the best performance than the other prediction models – it can detect 33% defective changes when inpsecting 20% LOC [66]. A similar results were found by Yan et al.'s study [63], and they found EALR achieved the best performance in file-level defect prediction – it can detect 34% defective files when inspecting 20% LOC.

TABLE 1.3: Summary of prediction models

Category	Model	Abbreviation
Function	Linear Regression	EALR
	Simple Logistic	SL
	Radial basis functions network	RBFNet
	Sequential Minimal Optimization	SMO
Lazy	K-Nearest Neighbour	IBk
Rule	Propositional rule	JRip
	Ripple down rules	Ridor
Bayes	Naive Bayes	NB
Tree	J48	J48
Ensemble	Logistic Model Tree	LMT
	Random Forest	RF
	Bagging	BG+LMT, BG+NB, BG+SL, BG+SMO, and BG+J48
	Adaboost	AB+LMT, AB+NB, AB+SL, AB+SMO, and AB+J48
	Rotation Forest	RF+LMT, RF+NB, RF+SL, RF+SMO, and RF+J48
	Random subspace	RS+LMT, RS+NB, RS+SL, RS+SMO, and RS+J48

1.3 Features for Crash Release Prediction for Apps

The quality of mobile applications has a vital impact on their users experience, ratings and ultimately overall success. Compared with traditional applications, mobile apps tend to have more releases. In many cases, mobile developers may release versions of the app that are of poor quality, e.g., crash releases which cause app to crash [61]. A crashing release is likely to cause users to uninstall an app, potentially giving it a negative rating, which in turn impacts the app's revenue. Thus, identifying crashing releases early on (e.g., before the release date), can help warn mobile app developers about a potential crashing version before it is released and reduce the number of crashing releases.

Various features can be extracted to predict crashing releases. Our previous study proposed 20 features which were grouped into six unique dimensions: complexity, time, code, diffusion, commit, and text [61]. All of these features are derived from the source control repository data of a mobile application. Table 1.4 presents a summary of the features.

TABLE 1.4: Features used to identify crashing releases

Dimension	Name	Definition
Complexity	Cyclomatic	The number of branching paths within code in all the source code files in a release.
Time	PreDays	The number of days since the previous release.
Code	LA	Number of lines added in a release.
	LD	Number of lines deleted in a release
	SIZE	Total number of lines of code in the current release
	SAME	Number of source code files that are modified by both the current and the previous release.
	CUR_file	Number of source code files in the current release
	PREV_file	Number of modified source code files in the previous release
Diffusion	Top_NS	Number of unique subsystems changed between two releases
	Bottom_NS	Number of unique subsystems changed between two releases
	NF	Number of unique files that have changed between two releases
	File_entropy	Distribution of modified files across the release
	Churn_entropy	Distribution of modified code across the application
Commit	NC	Number of commits
	NFC	Number of commits which fix bugs
Text	Fuzzy_score	Fuzzy set scores of commit logs
	NB_score	Naive Bayes scores of commit logs
	NBM_score	Naive Bayes Multinomial scores of commit logs
	DMN_score	Discriminative naive Bayes Multinomial scores of commit logs
	COMP_score	Complement naive Bayes scores of commit logs

1.3.1 Complexity Dimension

If source code in a release is too complex (e.g., high number of data or control flows in an applications), the code will be harder to write and maintenance, which may increase the chance of a crashing release. Also, prior stud-

ies showed complexity (e.g., cyclomatic complexity) was a good predictor of defect-prone modules [32,40,51]. As shown in Table 1.4, McCabe's cyclomatic complexity is used in the complexity dimension, which is measured directly from the source code in the current release using a standard static analysis tool.

1.3.2 Time Dimension

If the time period between the two releases is short, the current release may have a high chance to be a non-crash release since it may fix the crash bugs which appears in the previous release. Based on this, the number of days since the previous release (PreDays) is used as a feature to predict crashing release. It is computed by counting the number of days between the previous and current release.

1.3.3 Code Dimension

If many lines of code have been added/deleted/modified between the current and previous release, the current release is more likely to be a crash release. This is the case since many new features may have been added which increases the likelihood of a feature malfunction that causes a crash [38]. Also, if the current release modifies many of the same source code files as the previous release, which may indicate that many repairs have been done, and in turn indicate that the current release is a good release. As shown in Table 1.4, six features make up this dimension; they can be extracted from the source control repository by comparing the difference between two releases.

1.3.4 Diffusion Dimension

Intuitively, if too many different source code files are changed during a release, this release might be more difficult to understand, and requires more work to inspect all the locations that are changed. In defect prediction literature, prior studies found that the number of subsystems touched is an indicator of defects [37], and scattered changes was a good indicator of defects [19]. Also, the more functionalities there are in a release, the more prone it is to fail. Thus, subsystems are used as proxies to features. Releases that contain many modifications at the subsystem level are more likely to be crash releases. In Table 1.4, five features that make up the diffusion dimension.

Top directory name and bottom directory name as the subsystem name are used to measure Top_NS and Bottom_NS, respectively. For example, if a commit changes a file in the path "src/app/token/main.java", then its top directory name is "src/", and the bottom directory name is "src/app/token/". For the i^{th} release, the set of top and bottom directory names are denoted as Top_i and $Bottom_i$, respectively. Then, for two consecutive releases (i^{th} and

$(i+1)^{th}$ releases), Top_NS= $|Top_i \cap Top_{i+1}|$, and Bottom_NS =$|Bottom_i \cap Bottom_{i+1}|$.

Entropy aims to measure the distribution of a release across different files or the lines of code in the files. Releases with high entropy are more likely to be crash releases, since a developer needs to inspect large number of scattered changes across files. Two kinds of entropies were proposed, i.e., file and churn entropy [19]. Entropy is computed as: $H(P) = -\sum_{k=1}^{n}(p_k \times log_2\ p_k)$, where n is the number of files changed in the release, and $p_k \geq 0$ is the probability for a file k, and p_k satisfies $(\sum_{k=1}^{n} p_k) = 1$. To compute file entropy, p_k is computed as the the proportion that of commits between the current release and the previous release that include changes to file k. To compute churn entropy, p_k is computed as the the proportion that of number of lines of code between the current release and the previous release that include changes to file k.

1.3.5 Commit Dimension

If there are many commits between the current and previous release, the current release may have a high probability to be a crashing release. This is the case since more commits means more changes (e.g., bug fixes, new functionalities) to an app, which may introduce more problems (e.g., bugs) in the release. In Table 1.4, two features are proposed in this dimension: number of commits (NC), and number of bug fixing commits (NFC). NC is computed by counting the number of commits in the current release, and NFC is computed by counting the number of commits whose logs contain one of the following keywords: strings "fix", "'error", "fault", "crash", "issue", or "bug" [25, 46, 49].

1.3.6 Text Dimension

During the release of an app, many commits may be submitted to fix defects or implement new features. The textual features are first extracted from the commit messages by tokenization, stop-word removal, and stemming. The resulting textual tokens and count the number of times each token appears in the commit logs of a release are used to represent the textual features. Since there are a large number of unique words in commit logs, and to avoid that the text features to crowd out the other features, the words that appear in commit logs are converted into a small number of features. In total, five different features are proposed in the text dimension, which correspond to the textual scores outputted by five different classifiers.

To come up with the scores for the different textual features, the whole collected data are divided into a training set and a testing set. Then, the training data set is split into two training subsets by leveraging stratified random sampling, so that the distribution and number of non-crashing and crashing releases in both training subsets is the same [55]. A classifier is trained with the first training subset, and it is used to obtain the textual scores on the

second training subset. Besides, a second classifier is trained with the second training subset, and it is used to obtain the textual scores on the first training subset. In the prediction phase, for a new release, text mining classifiers that are built on all of the training releases to compute the values of the textual features. Different text mining classifiers can be used to build the textual features, our prior study use 5 types of textual classifiers to calculate the scores of the textual features, including fuzzy set classifier [67], naive Bayes classifier [33], naive Bayes multinomial classifier [33], discriminative naive Bayes multinomial classifier [50], and complement naive Bayes classifier [47].

1.3.7 Prediction Models and Results

Similar to defect prediction studies, various prediction models can be used to predict crash releases. In Xia et al.'s study [61], they built prediction models by using four different classification algorithms, namely Naive Bayes, decision tree, kNN, and Random Forest. They found Naive Bayes achieved the best performance, which corresponds to a F1 and AUC of 0.30 and 0.64, respectively. Considering that only a small number of releases are crash releases, predicting them accurately is a difficult problem.

1.4 Features from Mining Monthly Reports to Predict Developer Turnover

Developers are a key asset of an Information Technology (IT) company. Unfortunately, in an IT company, the influx and retreat of software developers, which are referred to as *turnover*, are often very frequent. Prior studies found the turnover rate in IT companies vary from 20% to 48% [22, 42, 57]. To help companies better manage developer turnover, it would be interesting to predict who are likely to leave.

Many companies require their employees to write monthly report, reporting their estimated number of working hours[2], and what they have done in the month. Table 1.5 present two example monthly reports. Although the structure of a monthly report is often simple, various features can be extracted from such reports.

Our prior work extracted 67 features from monthly reports that developers write in the first six months of them joining a company [6], which were divided into three categories: working hours task report, and project. The details of the features in these categories are presented in the following sections.

[2]This is especially true for outsourcing companies that charge clients based on the number of hours their developers spent on a project.

TABLE 1.5: Examples of Monthly Reports

	Example 1	**Example 2**
Month	2013-03	2015-08
Employee ID	1	11
Employee Name	D1	D2
Project Name	P1	P2
Tasks	1. fix bugs on UI 2. implement 5 new functionalities (ID XXX)	Write unit tests on the model XXX
Working Hours	168	128

1.4.1 Working Hours

In this category, eleven features corresponding to the working hours of developers in each of the first six months are collected. Working hours are related to a developer's workload. Software developers might be asked to take heavy workload or have tight deadlines. Heavy workload might cause a developer to leave a company. On the other hand, if a developer's working hours are less than normal, he/she might not be interesting in the job, which is an indicator of his/her eventual departure. Thus, the working hours of a developer in the first six month are used as the first six features in this category.

Summary statistics based on a developer's first six months' working hours are also collected. Another five features are proposed in this category, i.e., *sum*, *mean*, *median*, *standard deviation* and *maximum* of a developer's working hours for the six months.

1.4.2 Task Report

In this category, features which are based on the text information of task report written by the developers are collected. Since the written style of task report could be different for different developers, which is related to a developer's character and working attitude. For example, a developer, who writes the monthly report in much detail, is usually very conscientious. Otherwise, a simple task report might imply that the developer does not focus on his/her work or is dissatisfied with the work. One kind of these features is based on the length of length of text of task report for each monthly report, and calculate five kinds of statistics of task report, including the *sum*, *mean*, *median*, *standard deviation* and *maximum* of length of text of task report for each developer. Sometimes, some "lazy" developers copy the text of previous task reports or write similar task reports. Thus, after tokenizing and steming the text of task report, the *sum*, *mean*, *median*, *standard deviation* and *maximum* of number of tokens in the monthly report for each developer are calculated. In total, 11 features are proposed in this category:

1. *task_len_sum*: the sum of length of text of task reports.
2. *task_len_mean*: the mean of length of text of task reports.
3. *task_len_median*: the median of length of text of task reports.
4. *task_len_std*: the standard deviation of length of text of task reports.
5. *task_len_max*: the maximum of length of text of task reports.
6. *task_zero*: the number of monthly report whose length of task is 0.
7. *token_sum*: the sum of the token number of task reports.
8. *token_mean*: the mean of the token number of task reports.
9. *token_median*: the median of the token number of task reports.
10. *token_std*: the standard deviation of the token number of task reports.
11. *token_max*: the maximum of the token number of task reports.

Readability features, which refers to the ease with which a reader can understand the task report, are collected. The readability of a text is measured by the number of syllables per word and the length of sentences. Readability measures can be used to tell how many years of education a reader should have before reading the text without difficulties [11,12]. Amazon.com uses readability measures to inform customers about the difficulty of books. Readability features of task report are used as a complementary of statistics features of task report since readability could also be an indicator of a developer's working attitude. Following the prior study on the state-of-the-art on readabilty, nine readability features are used, i.e., Flesh [12], SMOG (simple measure of gobbledygook) [31], Kincaid [26], Coleman-Liau [10], Automated Readability Index [48], Dale-Chall [11], difficult words [11], Linsear Write [2], Fog [15]. These readability features can be extracted by using a python package named *textstat*[3].

1.4.3 Project

In this category, these features represent the information of a project which a developer is working on for each month. The working environment and other members in the project might have very important effect on a developer's working experience. For example, the good collaboration with other members in the project can improve a developer's work efficiency and experience. For each month, the following measures of the project which the developer is working for are calculated: the number of project members, the sum, mean and standard deviation of working hours of project members, and the number

[3]https://pypi.python.org/pypi/textstat

of changed developers. The number of project members is an indicator of project size. Small project size usually means more workload to each individual in the project. The working hours of project members could reflect the overall workload in the project. And the number of changed developers might indicate the stability of the project. The developers often prefer stay at a stable project. In total, 30 features are proposed in this dimension:

1. $p\{N\}\_person$: the number of persons in the project that the developer is working for in N^{th} month, where N is from 1 to 6.
2. $p\{N\}\_hour\_mean$: the mean of working hours of project members in N^{th} month.
3. $p\{N\}\_hour\_sum$: the sum of working hours of project members in N^{th} month.
4. $p\{N\}\_hour\_std$: the standard deviation of working hours of project members in N^{th} month.
5. $p\{N\}\_person\_change$: the number of changed person compared with the previous month in N^{th} month.

Summary statistics based on projects that a developer join in their first six months are also collected. In total, six features are proposed, i.e., project_num, multi_project, avg_person_change, less_zero, equal_zero, and larger_zero. Project_num refers to The number of project in the first six months for each developer, and multi_project refers to that whether a developer take part in more than one project in a month, these two features are proposed since the experience of working for multiple projects is different from that of working for only one project and multiple projects might mean higher workload. The number of developer changed in the project which a developer works for (avg_person_change, less_zero, equal_zero, larger_zero) are also counted, since the stability of the project might have impact on the working experience of a developer.

1.4.4 Prediction Models and Results

Based on the features described in the previous subsections, Bao et al. used five different classification algorithm to build prediction models [6], namely Naive Bayes, Support Vector Machine (SVM), decision tree, kNN, and Random Forest. They performed experiments on monthly reports collected from two companies in China, and random forest achieved the best performance, which corresponded to F1-scores for retained and not-retained developers of 0.86 and 0.65, respectively.

1.5 Summary

In this chapter, we present three case studies, to demonstrate how features can be generated from different software artifacts for different software engineering problems. The generated features can be used as input to a machine learning engine (e.g., a classification algorithm) to automate some software tasks or better manage projects. We hope our chapter can inspire more researchers and developers to dig into software artifacts to generate more powerful features, to further improve the performance of existing software analytics solutions or build new automated solutions that address pain points of software developers.

Nowadays, the performance of many predictive models developed to improve software engineering tasks is highly dependent manually on the constructed features. However, significant expert knowledge is required to identify domain-specific features. It would be interesting to investigate methods to automatically generate features from raw data. Deep learning is a promising direction that can be used to automatically learn advanced features from the multitude of raw data available in software repositories, APIs, blog posts, etc. Some of recent studies have showed the potential of deep learning to solve many software analystic problems (e.g., defect prediction [56, 65], similar bug detection [64], and linkable knowledge detection [62]) with promising results. Thus, it would be interesting to use deep learning techniques to relieve the heavy workload involved in manually crafting domain-specific features for various software engineering tasks and applications.

Bibliography

[1] Celebrating nine years of github with an anniversary sale. *https://goo.gl/4tXxUu, Retrieved on May 30, 2017*.

[2] Linsear write. `http://www.csun.edu/~vcecn006/read1.html#Linsear`.

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

[4] Erik Arisholm, Lionel C Briand, and Eivind B Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, 83(1):2–17, 2010.

[5] Jagdish Bansiya and Carl G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Trans. Software Eng.*, 28(1):4–17, 2002.

[6] Lingfeng Bao, Zhenchang Xing, Xin Xia, David Lo, and Shanping Li. Who will leave the company?: a large-scale industry study of developer turnover by mining monthly work report. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 170–181. IEEE Press, 2017.

[7] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code!: examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 4–14. ACM, 2011.

[8] Cagatay Catal, Banu Diri, and Bulent Ozumut. An artificial immune system approach for fault prediction in object-oriented software. In *Dependability of Computer Systems, 2007. DepCoS-RELCOMEX'07. 2nd International Conference on*, pages 238–245. IEEE, 2007.

[9] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Software Eng.*, 20(6):476–493, 1994.

[10] Meri Coleman and Ta Lin Liau. A computer readability formula designed for machine scoring. *Journal of Applied Psychology*, 60(2):283, 1975.

[11] Edgar Dale and Jeanne S Chall. A formula for predicting readability: Instructions. *Educational research bulletin*, pages 37–54, 1948.

[12] Rudolf Franz Flesch. *How to write plain English: A book for lawyers and consumers*. Harpercollins, 1979.

[13] Baljinder Ghotra, Shane McIntosh, and Ahmed E Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. In *ICSE*, pages 789–800. IEEE Press, 2015.

[14] Todd L Graves, Alan F Karr, James S Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Transactions on software engineering*, 26(7):653–661, 2000.

[15] Robert Gunning. {The Technique of Clear Writing}. 1952.

[16] Philip J Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 1, pages 495–504. IEEE, 2010.

[17] Tibor Gyimothy, Rudolf Ferenc, and Istvan Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software engineering*, 31(10):897–910, 2005.

[18] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. A systematic literature review on fault prediction performance in software engineering. *TSE*, 38(6):1276–1304, 2012.

[19] Ahmed E Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 78–88. IEEE Computer Society, 2009.

[20] B. Henderson-Sellers. *Object-Oriented Metrics, Measures of Complexity*. Prentice Hall, 1996.

[21] Qiao Huang, Xin Xia, and David Lo. Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction. In *Proceedings of the 33nd International Conference on Software Maintenance and Evolution*. IEEE, 2017, to appear.

[22] James J Jiang and Gary Klein. Supervisor support and career anchor impact on the career satisfaction of the entry-level information systems professional. *Journal of management information systems*, 16(3):219–240, 1999.

[23] Marian Jureczko and Lech Madeyski. Towards identifying software project clusters with regard to defect prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, page 9. ACM, 2010.

[24] 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, 2013.

[25] Sunghun Kim, Thomas Zimmermann, E James Whitehead Jr, and Andreas Zeller. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering*, pages 489–498. IEEE Computer Society, 2007.

[26] J Peter Kincaid, Robert P Fishburne Jr, Richard L Rogers, and Brad S Chissom. Derivation of new readability formulas (automated readability index, fog count and flesch reading ease formula) for navy enlisted personnel. Technical report, DTIC Document, 1975.

[27] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, 34(4):485–496, 2008.

[28] Paul Luo Li, James Herbsleb, Mary Shaw, and Brian Robinson. Experiences and results from initiating field defect prediction and product test prioritization efforts at abb inc. In *Proceedings of the 28th international conference on Software engineering*, pages 413–422. ACM, 2006.

[29] R. Martin. Oo design quality metrics - an analysis of dependencies. *IEEE Trans. Software Eng.*, 20(6):476–493, 1994.

[30] Shinsuke Matsumoto, Yasutaka Kamei, Akito Monden, Ken-ichi Matsumoto, and Masahide Nakamura. An analysis of developer metrics for fault prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, page 18. ACM, 2010.

[31] G Harry Mc Laughlin. Smog grading-a new readability formula. *Journal of reading*, 12(8):639–646, 1969.

[32] T.J. McCabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4):308–320, 1976.

[33] Andrew McCallum, Kamal Nigam, et al. A comparison of event models for naive bayes text classification. In *AAAI-98 workshop*.

[34] Thilo Mende and Rainer Koschke. Revisiting the evaluation of defect prediction models. In *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, page 7. ACM, 2009.

[35] Tim Menzies, Andrew Butcher, David Cok, Andrian Marcus, Lucas Layman, Forrest Shull, Burak Turhan, and Thomas Zimmermann. Local versus global lessons for defect prediction and effort estimation. *IEEE Transactions on software engineering*, 39(6):822–834, 2013.

[36] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE transactions on software engineering*, 33(1):2–13, 2007.

[37] Audris Mockus and David M Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000.

[38] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th international conference on Software engineering*, pages 181–190. ACM, 2008.

[39] John C. Munson and Taghi M. Khoshgoftaar. The detection of fault-prone programs. *IEEE Transactions on Software Engineering*, 18(5):423–433, 1992.

[40] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*, pages 452–461. ACM, 2006.

[41] Hector M Olague, Letha H Etzkorn, Sampson Gholston, and Stephen Quattlebaum. Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes. *IEEE Transactions on software Engineering*, 33(6), 2007.

[42] Nancy Pekala. Holding on to top talent. *Journal of Property management*, 66(5):22–22, 2001.

[43] Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. Ecological inference in empirical software engineering. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 362–371. IEEE Computer Society, 2011.

[44] Ranjith Purushothaman and Dewayne E Perry. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, 31(6):511–526, 2005.

[45] Foyzur Rahman and Premkumar Devanbu. How, and why, process metrics are better. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 432–441. IEEE Press, 2013.

[46] Foyzur Rahman, Daryl Posnett, Abram Hindle, Earl Barr, and Premkumar Devanbu. Bugcache for inspections: hit or miss? In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 322–331. ACM, 2011.

[47] Jason D Rennie, Lawrence Shih, Jaime Teevan, David R Karger, et al. Tackling the poor assumptions of naive bayes text classifiers. In *ICML*, 2003.

[48] RJ Senter and Edgar A Smith. Automated readability index. Technical report, DTIC Document, 1967.

[49] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *ACM sigsoft software engineering notes*, volume 30, pages 1–5. ACM, 2005.

[50] Jiang Su, Harry Zhang, Charles X Ling, and Stan Matwin. Discriminative parameter learning for bayesian networks. In *ICML*, 2008.

[51] Ramanath Subramanyam and Mayuram S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on software engineering*, 29(4):297–310, 2003.

[52] Mei-Huei Tang, Ming-Hung Kao, and Mei-Hwa Chen. An empirical study on object-oriented metrics. In *METRICS*, pages 242–249, 1999.

[53] Yuan Tian, David Lo, Chengnian Sun, and Xin XIA. Automated prediction of bug report priority using multi-factor analysis. *Empirical Software Engineering*, 20(5):1354, 2015.

[54] Burak Turhan, Tim Menzies, Ayşe B Bener, and Justin Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering*, 14(5):540–578, 2009.

[55] Harold Valdivia Garcia and Emad Shihab. Characterizing and predicting blocking bugs in open source projects. In *Proceedings of the 11th working conference on mining software repositories*, pages 72–81. ACM, 2014.

[56] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering*, pages 297–308. ACM, 2016.

[57] Aja Whitaker. What causes it workers to leave. *Management Review*, 88(9):8, 1999.

[58] Xin Xia, David Lo, Denzil Correa, Ashish Sureka, and Emad Shihab. It takes two to tango: Deleted stack overflow question prediction with text and meta features. In *Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual*, volume 1, pages 73–82. IEEE, 2016.

[59] Xin Xia, David Lo, Ying Ding, Jafar M Al-Kofahi, Tien N Nguyen, and Xinyu Wang. Improving automated bug triaging with specialized topic model. *IEEE Transactions on Software Engineering*, 43(3):272–297, 2017.

[60] Xin Xia, David Lo, Sinno Jialin Pan, Nachiappan Nagappan, and Xinyu Wang. Hydra: Massively compositional model for cross-project defect prediction. *IEEE Transactions on software Engineering*, 42(10):977–998, 2016.

[61] Xin Xia, Emad Shihab, Yasutaka Kamei, David Lo, and Xinyu Wang. Predicting crashing releases of mobile applications. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, page 29. ACM, 2016.

[62] Bowen Xu, Deheng Ye, Zhenchang Xing, Xin Xia, Guibin Chen, and Shanping Li. Predicting semantically linkable knowledge in developer online forums via convolutional neural network. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 51–62. ACM, 2016.

[63] Meng Yan, Yicheng Fang, David Lo, Xin Xia, and Xiaohong Zhang. File-level defect prediction: Unsupervised vs. supervised models. In *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2017, to appear.

[64] Xinli Yang, David Lo, Xin Xia, Lingfeng Bao, and Jianling Sun. Combining word embedding with information retrieval to recommend similar bug reports. In *Software Reliability Engineering (ISSRE), 2016 IEEE 27th International Symposium on*, pages 127–137. IEEE, 2016.

[65] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. Deep learning for just-in-time defect prediction. In *Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on*, pages 17–26. IEEE, 2015.

[66] Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 157–168. ACM, 2016.

[67] HJ Zimmermann. *Fuzzy Set Theory and Its Applications Second, Revised Edition*. 1992.

Index