Computers, Materials & Continua CMC, vol.64, no.1, pp.273-295, 2020

Programming Logic Modeling and Cross-Program Defect Detection Method for Object-Oriented Code

Yan Liu[1], Wenyuan Fang[1], Qiang Wei[1,*], Yuan Zhao[1] and Liang Wang[2]

Abstract: Code defects can lead to software vulnerability and even produce vulnerability risks. Existing research shows that the code detection technology with text analysis can judge whether object-oriented code files are defective to some extent. However, these detection techniques are mainly based on text features and have weak detection capabilities across programs. Compared with the uncertainty of the code and text caused by the developer's personalization, the programming language has a stricter logical specification, which reflects the rules and requirements of the language itself and the developer's potential way of thinking. This article replaces text analysis with programming logic modeling, breaks through the limitation of code text analysis solely relying on the probability of sentence/word occurrence in the code, and proposes an object-oriented language programming logic construction method based on method constraint relationships, selecting features through hypothesis testing ideas, and construct support vector machine classifier to detect class files with defects and reduce the impact of personalized programming on detection methods. In the experiment, some representative Android applications were selected to test and compare the proposed methods. In terms of the accuracy of code defect detection, through cross validation, the proposed method and the existing leading methods all reach an average of more than 90%. In the aspect of cross program detection, the method proposed in this paper is superior to the other two leading methods in accuracy, recall and F1 value.

Keywords: Method constraint relationship, programming logic, code defect, hypothesis test.

1 Introduction

With the continuous development of software technology, software security is be-coming more and more important for enterprises and individual users. But no organization can fundamentally eliminate the emergence of vulnerabilities. Bielak et al. [Bielak and Biffl (2003)] showed that programmers generate 100 to 150 errors per thousand lines of code during development, and these errors tend to become a vulnerability exploited by

[1] State Key Laboratory of Mathematical Engineering and Advanced Computing, PLA Strategic Support Force Information Engineering University, Zhengzhou, 450001, China.

[2] The School of Computing, Newcastle University, Newcastle upon Tyne, NE4 5TG, UK.

[*] Corresponding Author: Qiang Wei. Email: funnywei@163.com.

Received: 18 January 2020; Accepted: 01 March 2020.

attackers. IEEE/ISO/IEC 24765-2010 ISO/IEC/IEEE International Standard—Systems and software engineering-Vocabulary defines code defects as code problems that can cause an application to fail or produce incorrect results if not corrected. [ISO/IEC/IEEE International Standard (2017)] Detecting and fixing all possible code defects is key to ensuring the security of the underlying software system. However, how to use artificial intelligence methods to guide software testers to detect potential software defects in a targeted manner is a field worth studying. Therefore, cross program code defect detection has become a research hotspot in recent years [Yao, Huang, Feng et al. (2019); Gharehyazie, Ray, Keshani et al. (2019)].

Existing research shows that text analysis techniques can be used to detect code defects and achieve good results [Scandariato, Walden, Hovsepyan et al. (2014)]. Traditional text analysis often mines appropriate placards as feature words in source code, constructs vector representations or statistical models by the frequency of occurrence of different feature words and serves as a standard for classification. This kind of method has two problems in the analysis process: Firstly, text-based feature words are greatly influenced by the developer's programming style (such as different naming methods), which results in better performance when the training set and test set are from the same developer's software. Conversely the analysis of software defects across programs will be significantly reduced; Secondly, the static text analysis of the source code often forms a high-dimensional feature word set, which also brings huge computing resource consumption. Compared with the code text uncertainty due to developer personalization, programming languages have stricter logic specifications. Developers often follow similar programming logic to implement the same method function. These programming logics embody the rules of the language itself and the potential thinking of the developer. Making full use of these programming logics can make the code defect detection method based on text analysis break through the limitation of the probability of occurrence of sentences/words in the code and reduce the influence of personalized programming on the detection method. Unfortunately, in the current research, the code defect detection method based on text analysis often lacks the research and utilization of these programming logics, so that the accuracy of detection still has room for improvement.

In view of the above problems, this paper studies the modeling of programming logic and implements code defect detection based on programming logic. The main contributions of this article are:

1) Propose the method-based constraint-based programming logic model CPMMC (Code-Predicting Model based on Method Constraints) to overcome the limitations of text analysis on programming logic expression;

2) Propose an object-oriented code defect detection method based on programming logic OCDDM (Object-Oriented Code Defect Detection Method) to improve the detection capability of cross-language code defects.

The rest of the paper is organized as follows: Section 2 introduces the research status of code defect detection; Section 3 presents the concept of code defect detection based on programming logic and describes the problems to be solved by this method; Section 4 introduces the programming logic model; Section 5 introduces the code defect detection method based on the programming logic model; Section 6 introduces the experimental

methods and results; Section 7 is conclusion.

2 Related work

With the increasing importance of software security, developers are expected to fix vulnerabilities in software systems as much as possible. but for massive code files, comprehensive detection and patching of potential vulnerabilities are time consuming and expensive. Therefore, how to use data mining and other methods to detect code defects and guide software testers to detect and repair potential vulnerabilities has become a research hotspot focused by scholars at home and abroad. Code defect detection typically takes the analysis and extraction of features from the source code of the program (advanced source code or binary machine code) and uses machine learning algorithms (mainly supervised learning) to automatically learn the flawed modules in the code. According to different analysis methods, it can be roughly divided into general vulnerability code analysis, data flow and control flow analysis, code text mining and so on.

2.1 General vulnerability code parsing

Neuhaus et al. [Neuhaus, Zimmermann, Holler et al. (2007)] focused on the correlation between vulnerabilities and successfully used machine learning techniques to predict vulnerabilities in the snapshot environment of Mozilla projects (including Firefox and Thunderbird). The authors reported an average accuracy of 70% and a recall rate of 45%. Zimmermann et al. [Zimmermann, Nagappan and Williams (2010)] found weak correlations between vulnerabilities and various metrics, including code churn, code complexity, dependencies, and organizational measures. In the Windows Vista environment, they built two different predictors. The first is based on traditional metrics (code churn metrics, code complexity metrics, dependency metrics, code cover-age metrics, and organizational metrics) with an experimental accuracy of 66.7% and a recall rate of 20%. The second prediction model is based on the dependency between binary files with an accuracy of 60% but a recall rate of 40%. Yamaguchi et al. [Yamaguchi, Lindner and Rieck (2011); Yamaguchi, Lottmann, Rieck (2012)] proposed a method to assist in the discovery of vulnerabilities by introducing the concept of "vulnerability extrapolation". This method is designed to identify unknown vulnerabilities based on the programming patterns observed in known security vulnerabilities. The motivation for vulnerability extrapolation is based on the observation that security analysts search the code base for similar vulnerabilities that have recently been discovered. To this end, the author proposes four steps: Firstly, the abstract syntax tree (AST) for each function in the C and C++ code is extracted; Secondly, the unrelated nodes in the AST are discarded and each function is represented as a vector of the contained subtree, embedding the AST of the function into a vector space; Semantic analysis is then used on the vector representation to identify the corresponding code structure pattern; Finally, the vulnerability extrapolation is performed by calculating the distance. The authors experimented on four open source projects, LibTIFF, Pidgin, FFmpeg, and Asterisk, using some of the latest re-ported vulnerabilities in each project as a seed for extrapolation, helping to discover several "0 day" vulnerabilities. Zhang et al. [Zhang, Li, Li et al. (2019)] summarized the characteristics of Android application software

vulnerabilities, established a model to assess the security level of the vulnerability, and designed and implemented a secure container for the CM-Droid, a password abuse vulnerability in Android application software. Features are extracted using only about 8 packets. This enables MalDetect to determine if a stream was generated by malware before taking illegal actions, which can better protect network users from malware.

Although the general vulnerability code parsing method can find the code vulnerability more directly, it has limited ability to detect vulnerabilities that are different from the known vulnerability patterns. More importantly, it requires researchers with a wealth of expertise in the field of vulnerability mining and a very high-quality training sample of a certain scale, which poses a huge challenge to cross-disciplinary re-searchers.

2.2 Data flow and control flow analysis

Shar et al. [Shar and Tan (2012, 2013)] presented a set of static code attributes based on data flow analysis of PHP web applications, which can be used to predict program statements that are vulnerable to SQL injection (SQLI) and cross-site scripting (XSS) attacks. The authors present a total of 20 static code attributes that reflect the different data flow aspects of the code segment. For example: the source of input (HTTP re-quest, file, database, etc.), the types of input data, the number of different output statements (database query, HTML output, etc.), and the different input validations etc. To evaluate the validity of the attribute, the author developed a prototype tool called PhpMinerI and experimented with eight open source web applications based on PHP. PhpMinerI extracts the control flow (CFG) and data flow graph (DFG) for a given PHP program and performs a backward data flow analysis on the target receive statement. Using this backward analysis, the above attributes are calculated and expressed as 20-dimensional vectors. Experiments show that the method has a recall rate of 93% for statements that are susceptible to SQL injection, and a false positive rate of 11%. For statements that are vulnerable to XSS attacks, the recall rate is 78% and the false positive rate is 6%. Shar et al. [Shar, Briand and Tan (2015)] further expanded their previous work in 2015. Firstly, in addition to SQLI and XSS, they extended the scope of the vulnerability by adding Remote Code Execution (RCE) and File Containment (FI) Web Vulnerabilities; Secondly, they used static reverse program sharding and use control dependency information to extract different execution paths from program shards; thirdly, they extend static attributes to 10 static attributes and 22 dynamic attributes. The experiment achieved an average recall rate of 77% and a false positive rate of 5%.

2.3 Code text analysis

Hata et al. used text features and spam filtering algorithms to predict defects in the software. In their early work [Mizuno, Ikami and Nakaichi (2007)], this method was used to predict defects in ArgoUML and Eclipse BIRT software with an accuracy of 72%-75% and a recall rate of 70%-72%. In the follow up work [Hata, Mizuno and Kikuno (2010)], they tried 5 open source Eclipse projects with an accuracy rate and a recall rate of 40% and 80% respectively. Gruska et al. [Gruska, Wasylkowski and Zeller (2010)] conducted large-scale data mining by mining more than 6,000 open source Linux projects, and obtained 16 million attributes reflecting the normal interface usage. Based on these attributes, new

project anomalies were checked. The authors validated the method on 20 project samples and found actual defects in the top 25% of the anomalies. Scandariato et al. [Scandariato, Walden, Hovsepyan et al. (2014)] constructed a vector representation of source code files by the frequency of different texts in the source code based on Bag-of-words, and constructed naive Bayesian and random forest classification models. The author experimented with code in 10 Android apps. The experimental results showed that the accuracy and recall rate of file-level defect detection in 8 applications can reach more than 80%, and the detection accuracy of 5 applications is even more than 90%.

The code text analysis method treats the code as plain text with low computation-al complexity and is suitable for large-scale data sets. But there are still some problems. Firstly, it is impossible to judge the type of defect; secondly, the detection result has a large granularity, it is difficult to accurately find the position of the defect code; Thirdly, because the programming habits of developers vary from person to person (for example, some people prefer Hungarian nomenclature, some may prefer camel nomenclature, etc.) at the text level, and it is difficult to ensure accuracy in cross-program detection; The last one is that the a large number of attributes or features generated (such as the 16 million attributes discovered by Gruska et al. [Gruska, Wasylkowski and Zeller (2010)] result in data sparseness or even dimensionality disaster.

In summary, although the existing code models have different fields of application, the balance between the logic information of the code (such as syntax, structure, etc.) and the complexity of the model is still a problem, and it does not meet the requirement to extract developer programming logic modeling in large-scale open source code. However, in the existing code defect detection method, the conventional vulnerability code parsing method requires a large number of known vulnerability samples, and the data stream and the control flow analysis method are very expensive to calculate, which makes it difficult to analyze the large-scale network open source code; Although the code text analysis method is suitable for the research field of large-scale open source code defect detection, it still needs to be improved in terms of the granularity of detection results and the applicability of cross-program detection.

3 Problem definition

Code-based text analysis has been proven to be successfully applied to code defect detection, but the code text is more affected by the developer's personalization. The programming styles and naming conventions of different developers are different, which makes the code defect detection based on text analysis not effective in dealing with cross-program detection. Compared to the uncertainty of the code text, developers often follow similar programming logic to achieve a certain function, not only because of the reuse of code modules brought by the engineering of software development, but also in order to avoid software vulnerabilities, experienced developers follow a standardized, mature program structure and method call sequence. We understand programming logic as an orderly code abstraction that expresses the programmer's thinking logic. This paper hopes to make better use of these programming logics when detecting code defects and improve the effect of cross-program detection. Usually, the code in the program does not exist independently. Several lines of code before and after it determine the appearance and use of

the code. The closer the two lines of code are, the more influence each other has.

Based on the above analysis, the object-oriented code defect detection problem in this paper can be understood as trying to refine the programming logic from the source code written by multiple developers, construct a statistical model that reflects the developer's programming thinking logic, and use the programming logic model to achieve the detection of code defects.

In this section, we will present a conceptual programming logic model of the constraint relationship of methods.

Definition 1 (Method constraint relationship) Method constraint relationship is defined as the effect of usage mode between methods in an object-oriented programming language. In an object-oriented programming language l, for any method m_1 and m_2, if one of the methods is used by another method, there is a constraint relationship between m_1 and m_2.

The constraint relationship of the method can be divided into two categories, the internal constraint relationship of the method and the external constraint relationship of the method.

Definition 2 (Internal constraint relationship of the method) Internal constraint relationship of a method is defined as the constraint relationship associated with a single object. In an object-oriented programming language l, for any method m_1 and m_2, if the usage of m_2 is affected by m_1, and there is an object o satisfies the condition that $m_1, m_2 \in o$, there is an internal constraint relationship between m_1 and m_2.

Definition 3 (External constraint relationship of the method) External constraint relationship of a method is defined as a constraint relationship that is associated across objects. In an object-oriented programming language l, for any method m_1 and m_2, if the usage of m_2 is affected by m_1, and there is no object o Satisfies the condition that $m_1, m_2 \in o$, then there is an external constraint relationship between m_1 and m_2.

Since the subject of the constraint relationship is the method call of the object, it is especially important to represent a method in the model. Here we give the concept of code feature words.

Definition 4 (Code feature words) Code feature word is defined as a feature segment that expresses the grammatical structure of a method call statement, denoted as $T = \{t_1, t_2 \cdots t_n\}$.

This paper builds a programming logic model based on the constraint relationship of the method, and on this basis do code defect detection.

It is mainly divided into three parts, the construction of feature vectors, the ordering of features and the detection of code defects, as shown in Fig. 1. In the construction part of the feature vector, the class file with label is first abstracted into a programming logic model based on the method constraint relationship (CPMMC model); Then, the code feature word sequence fragments in the model are used as code defect features, and the feature vectors of each class file are constructed by the frequency of occurrence. In the feature sorting part, the feature matrix is first constructed based on the feature vector of each class file in the training set, each column represents a feature, and each row represents a source file; Then the hypothesis test probability value of each feature is calculated by the idea of hypothesis testing; Finally, the features are sorted based on the

probability values, and the features with larger contributions are selected. In the detection part of the code defects, the second classifier is constructed based on the support vector machine, and the class files without tags are detected.

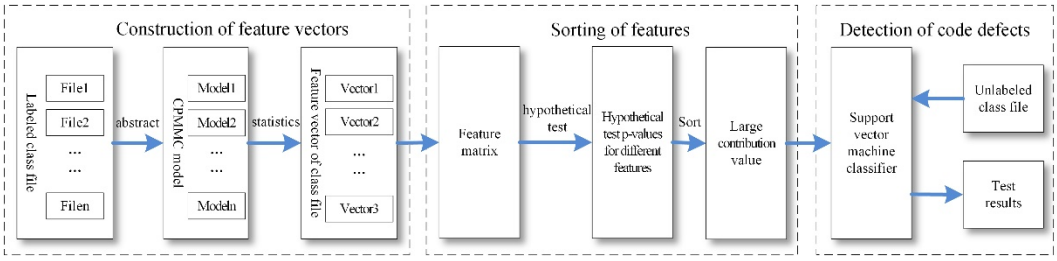

Figure 1: Flow chart of object-oriented code defect detection method based on CPMMC

There are three key issues to consider in the code defect detection process based on the programming logic model:

1) The difficult of Strong constraint relationship positioning. Not all methods have a constraint relationship, even if there is a constraint relationship, the strength of the relationship is not the same. How to determine the scope of this constraint relationship, more accurately, how to find other methods with a strong constraint relationship with a method is the first key issue in this paper.

2) The cross-program expression of the same method is quite different. Software development is a subjective process, and developers have personalization issues when naming objects. Compare two simple statements int len=str.length() and int l=s.length(). Although there are differences in expressions, essentially, there is no difference in the rules of the call and the functions implemented. How to minimize the negative impact of programming personalization on the model is the second key issue in this paper. 3)The sparse and dimensionality of feature data. How to use the relevant methods to analyze the influence degree of different features on code defects, and then select the features with greater influence to effectively reduce the dimension of features, reduce the impact of data sparseness on the model and avoid the occurrence of dimensionality disaster is the third key issue of this paper.

4 Programming logic model based on method constraint relationship

Code-Predicting Model based on Method Constraints CPMMC mainly includes two steps of programming logic abstraction and modeling. The overall modeling idea is shown in Fig. 2. In the abstract part of the programming logic. Firstly, by analyzing and expanding the constraint relationship of the method, the code feature words of the method call are constructed; Then the sequence of method calls involved in the object is converted into a sequence of code feature words. In the modeling part of the programming logic, based on the N-gram idea, the abstracted code feature word sequence is segmented by sliding, the frequency of different Nary segments is obtained statistically, and the programming logic model is constructed.

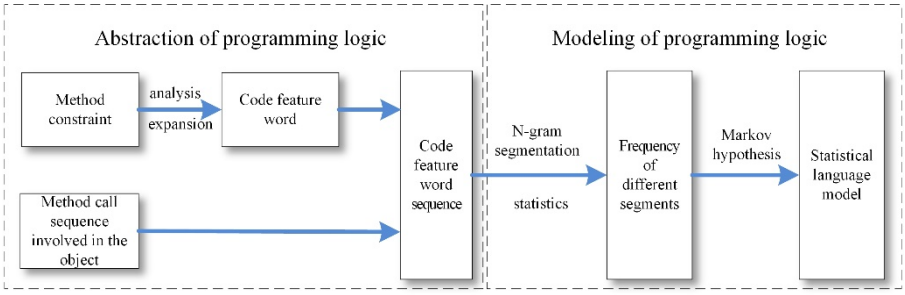

Figure 2: Programming logic modeling idea diagram based on method constraint relationship

4.1 Code abstraction

Code abstraction is the core of the model build, which determines the overall structure of the model and the representation of the method call in the model (code feature words).

4.1.1 Object hierarchical abstract structure

Before determining the overall structure of the model, we need to analyze the scope of the method constraint relationship. That is to find strong constraints as much as possible. Alan Kay analyzed the characteristics of object-oriented languages. He proposed that programs are collections of objects that send messages to tell each other what to do [Eckel (2005)]. Therefore, an object-oriented program can be thought of as a network with objects as nodes and method calls (messaging) as edges. Based on the basic characteristics of the network, this paper can qualitatively believe that there is a close relationship between the edges of the same node. That is to say, in general, there is a strong constraint relationship between method calls involved in the same object. Therefore, this paper limits the basic structure of the model to the sequence of meth-od calls involved in the same object. In this model, the source code is abstracted into three levels, formalized as:

$$Code = \{Class_1, Class_2, \cdots, Class_n\} \tag{1}$$

$$Class = \{Seq_{Object1}, Seq_{Object2}, \cdots, Seq_{Objectm}\} \tag{2}$$

$$Seq = Method_1 \cdot Method_2 \cdots Method_l \tag{3}$$

$Class_n$ represents the collection of code statements involved in the nth class, $Seq_{Objectm}$ represents the sequence of method calls involved in the mth object in the same class, and $Method_l$ represents the lth method call in the same method call sequence. This article converts source code into a collection of method call sequences Seq, each of which describes a method call statement that an object refers to.

4.1.2 Feature word selection based on method constraint relationship

On the basis of the previous section, this paper analyzes the method call statement Method from the three main levels of the constraint relationship, the expression form and the extension of the constraint relationship, and extracts the code feature words to describe it. As shown in Fig. 3.

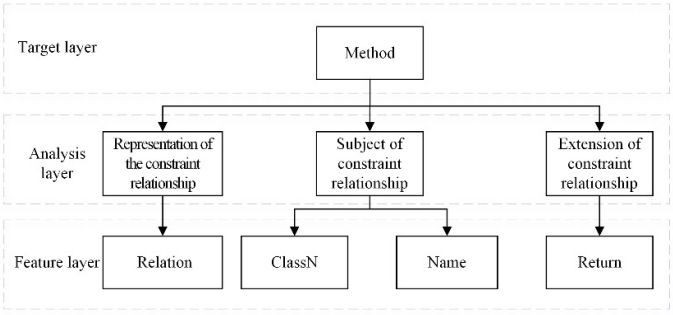

Figure 3: Code feature word analysis process of CPMMC

The Subject of the Constraint Relationship. The subject of the constraint relationship is the method call with the constraint relationship. This method call can come from the object itself or from other objects, so it needs to be represented by the object name $ObjectN$ of the method call. Reference [Eckel (2005)] defines a combination of a parameter list (including the number and type of parameters) and a method named as a "method signature", which uniquely identifies a method, and defined the method name $MethodN$ and parameter list $Parameter$ as candidate feature words. In order to solve the problem of personalized naming when developers name objects, this paper abstracts the object name $ObjectN$ as the class name $ClassN$. After experi-mental analysis, adding the parameter list directly to the code feature word does not improve the accuracy of the model, so the parameter list $Parameter$ is finally eliminated. Based on the above analysis, the feature representing the subject of the constraint relationship is determined to be $< ClassN, MethodN >$.

The Manifestation of the Constraint Relationship. The representation of the constraint relationship is reflected in the code as the relationship between the method and the $Objectn$ in the $Seq_{Objectn}$. This relationship is divided into three cases in the Java language: the object is the calling body of the method, the object is the parameter of the method, and the return value of the method is assigned to object. In order to de-scribe these three cases, this paper draws on the research results of Raychev et al. [Raychev, Vechev and Yahav (2014)], which is represented by a placeholder $Relation$.

The Expansion of the Constraint Relationship. There are also interactions between different code elements in the same line of code, where the assignment symbol represents the strongest relationship. In order to reflect this relationship, this paper extends the return value Return of the method as a complement to the method constraint relationship. The value is as follows:

$$Relation = \begin{cases} 0 & \text{The object is the calling body of the method} \\ n, n \in N & \text{The object is the nth argument of the method ret} \\ ret & \text{The return value of the method is assigned to the object} \end{cases} \quad (4)$$

In summary, a method call can be represented by a four-tuple, which is expressed as:

$$Method = < ClassN, Name, Return, Relation > \quad (5)$$

$Method$ is a method call to the object, $ClassN$ is the class name, $Name$ is the method name, $Return$ is the return type of the method, and $Relation$ is the relationship between

the object and the method.

In addition, two common situations can affect the extraction sequence $Seq_{Objectn}$. 1)When a conditional selection statement (such as if/else, switch/case) occurs, the program executes several complete and independent instructions due to different conditions, so it is necessary to construct several independent codes according to different conditions. sequence. 2)When the same object appears in different code structures, appropriate structural constraints become necessary to reduce complexity. Considering that the code in the program is closely related to the previous lines of code [Hindle, Barr, Gabel et al. (2016)], and the most closely coupled code statements are often in the same function. This paper defines the scope of the code sequence as the function body, and the same object appears in multiple function bodies. The situation is treated as multiple independent code sequences.

Fig. 4 is a piece of code fragment in the Stack overflow[2] website, which can be abstracted into a sequence of code feature words of three objects, as shown in Tab. 1.

```
SmsManager smsMgr = SmsManager.getDefault();
int length = message.length();
if (length > MAX_SMS_MESSAGE_LENGTH)
{
   ArrayList<String> msgList = smsMgr.divideMsg(message);
   smsMgr.sendMultipartTextMessage(msgList);
}
Else
{
   smsMgr.sendTextMessage(message);
}
```

Figure 1: Code fragment in Stack overflow

Table 1: Code feature word sequence of CPMMC

Class	Code feature word sequence
SmsManager	{(SmsManager,getDefault,SmsManager,ret)· (SmsManager,divideMsg,ArrayList<String>,0)· (SmsManager,sendMultipartTextMessage,void,0)}
	{(SmsManager,getDefault,SmsManager,ret)· (SmsManager,sendTextMessage,Void,0)}
String	{(String,length,Int,0)· (String,divideMsg,ArrayList<String>,1)}
	{(String,length,Int,0)· (String,sendTextMessage,Void,1)}
ArrayList <String>	{(ArrayList<String>,divideMsg,ArrayList<String>,ret)· (ArrayList<String>,sendMultipartTextMessage,Void,1)}

[2] Stack Overflow. https://stackoverflow.com/.

4.2 Construct feature vector

Based on the code abstraction, inspired by the bag of words, the N-gram method is used to segment the method call sequence of each object. CPMMC uses the trigram model to slide a code feature word sequence $Seq_{Objectn}$ of length l into $l-2$ consecutive sequences.

$$Seq_{Objectn} \mapsto \{Seq^t{}_1, Seq^t{}_2, \cdots, Seq^t{}_{l-2}\} \tag{6}$$

where $Seq^t{}_1$ represents a sequence of three consecutive code feature words

$$Seq^t{}_i = Method^i{}_1 \cdot Method^i{}_2 \cdot Method^i{}_3 \tag{7}$$

After the segmentation, the source $Code$ a becomes a set of n Seq^t

$$Code \mapsto \{Seq^t{}_1, Seq^t{}_2, \cdots, Seq^t{}_n\} \tag{8}$$

When using unigram (N-gram split, N is 1) feature, you can get a list of features, as shown in Tab. 2. A feature vector is created for each class file, the elements of which represent the number of times each feature appears in the list, i.e., the frequency. Because when the model in this paper considers the code logic, the conditional selection structure is treated as multiple independent method call sequences according to different conditions. Although the features (SmsManager, getDefault, SmsManager, ret) and (String, length, Int, 0) appear only once in the code, the feature vector of the file is 1, 1, 1, 1, 1, 1, because they exist in the two method call sequences. Similarly, when using the bigram (N-gram split, N is 2) feature, the feature list and appearance frequency are as shown in Tab. 3. The feature vector of the file is {2, 1, 1, 1, 2, 1, 1, 1, 1}. It is worth noting that when the program size increases, the dimensions of the bigram feature can be much higher than the trigram feature.

Table 2: Unigram feature vector

Feature list	Frequency
(SmsManager, getDefault, SmsManager, ret) (SmsManager, divideMsg, ArrayList <String>, 0)	1
(SmsManager, divideMsg, ArrayList <String>, 0) (SmsManager, sendMultipartTextMessage, void, 0)	1
(SmsManager, getDefault, SmsManager, ret) (SmsManager, sendTextMessage, void, 0)	1
(String, length, Int, 0) (String, divideMsg, ArrayList <String>, 1)	1
(String, length, Int, 0) (String, sendTextMessage, void, 1)	1
(ArrayList <String>, divideMsg, ArrayList <String>, ret) (ArrayList <String>, sendMultipartTextMessage, void, 1)	1

Table 3: Bigram feature vector

Feature list	Frequency
(SmsManager, getDefault, SmsManager, ret)	2
(SmsManager, divideMsg, ArrayList <String>, 0)	1
(SmsManager, sendMultipartTextMessage, void, 0)	1
(SmsManager, sendTextMessage, void, 0)	1
(String, length, Int, 0)	2
(String, divideMsg, ArrayList <String>, 1)	1

5 Object-oriented code defect detection based on programming logic

The essence of code defect detection is a binary Classification problem that deter-mines whether a code defect exists on a given code component. After the construction of the programming logic model, the data sparseness and dimensionality disasters have become the key issues to be solved in this paper. In this section, the feature selection and vector dimension reduction are completed by the idea of hypothesis testing, on the basis of which the classifier is constructed to implement code defect detection.

5.1 Code feature word sorting algorithm based on hypothesis testing

The feature selection algorithm aims to identify a subset of features that have a greater contribution to the classification. Feature selection is especially important in the case of more features and fewer samples. In this paper, the feature features based on hypothesis testing is used to sort features, thereby eliminating a large number of unrelated or less important features, and retaining a small number of features. The remaining features are discarded. This method has been applied to the fault location of graphics software by Xue et al. [Xue, Pang and Namin (2015)] and has achieved good results.

Hypothesis testing is an important method in statistical inference. The basic idea is the small probability counter method: the idea of small probability means that the small probability event ($P < 0.01$ or $P < 0.05$) does not basically occur in one experiment; the idea of counter-evidence is to make the hypothesis first (Test hypothesis H_0), and then use appropriate statistical methods to determine the probability of the hypothesis being established. If the probability is small, the hypothesis is not established. If the probability is large, the alternative hypothesis cannot be considered to established. More specifically, there are two competing assumptions: the null hypothesis and the alternative hypothesis. The null hypothesis H_0 assumes that there is no difference between the two sets of features; and the alternative hypothesis H_1 implies a statistically significant difference between the two sets of features. The probability value (value p) of the hypothesis test actually is the probability of error, that is, the original hypothesis H_0 is true, but we still assume that it is false and then reject it (Type I Error). So we expect the probability of making mistakes to be as small as possible. The greater the value p, the more cautious and the less likely it is to refuse H_0. In summary, the magnitude of the value p can be a measure of the strength of the null hypothesis H_0. We have developed the following null

hypotheses and alternative hypotheses, the purpose of which is to sort the features by the calculated value p .

$$H_0^{f_i}: \mu_{rob}^{f_i} = \mu_{def}^{f_i} \tag{9}$$

$$H_1^{f_i}: \mu_{rob}^{f_i} \neq \mu_{def}^{f_i} \tag{10}$$

where $H_0^{f_i}$ represents that the statistical result of feature fi in the robust code is not substantially different from the statistical result of feature fi in the defect code; $H_1^{f_i}$ means that there is an essential difference between the two in terms of statistics. Therefore, the smaller the value p, the greater the probability that there is an essential difference between the two.

Mann-Whitney-Wilcoxon is a non-parametric alternative to two-sample t-test, which does not rely on the assumption of data complying with any distribution [Siegal (1956)]. The Wilcoxon rank sum test observes the observations of the two samples and derives the shape and center of the two data sets based on the sum of the ranks observed for each sample. More specifically, the Wilcoxon rank sum test statistic is the sum of observations from one of the samples. Wilcoxon rank sum test has been widely used in bioinformatics, including tumor diagnosis by analyzing gene maps [Deng, Ma and Pei (2004)], cell type-specific labeling of microRNAs on target mRNA expression [Sood, Krek, Zavolan et al. (2006)], and asthma research on the development of allergic rhinitis [Möller, DreborgM, Ferdousi et al. (2002)].

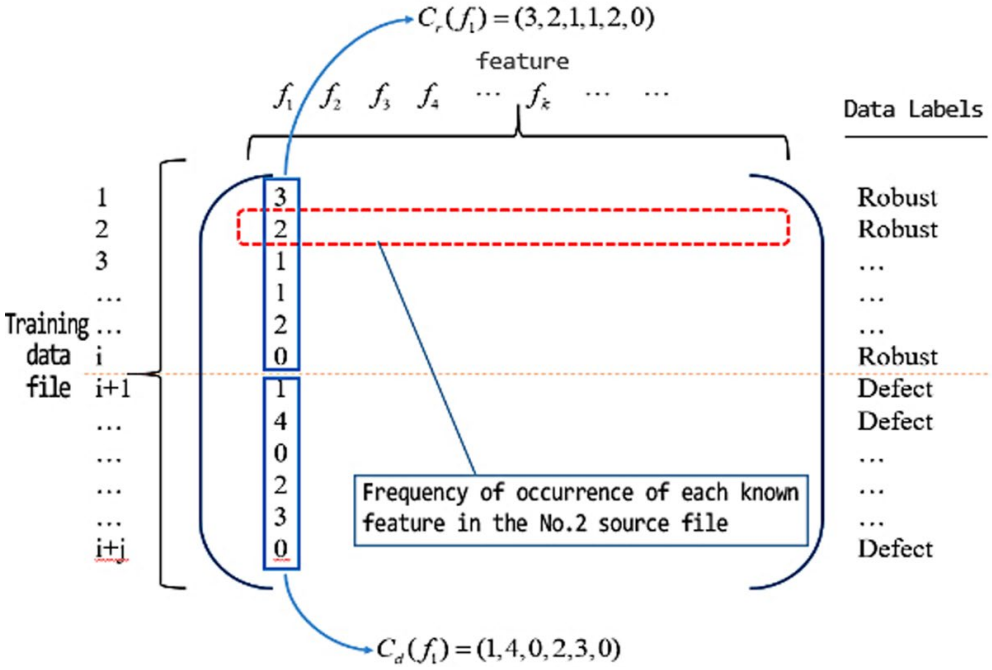

Figure 2: Thought chart of feature word sorting algorithm based on hypothesis test

Fig. 5 shows the idea of the algorithm in this paper. The data is organized in a matrix, with each column representing a feature, with each row representing a source file. The elements in the k column of the i row represent the frequency of the kth characteristic f_k appearing in the i file. Line 1 of the matrix to the i behavior robust code file, line $i+1$ to the $i+j$ behavior defect code file. For the first feature f_1 in Fig. 3, the frequency vector $C_r(f_1) = (3,2,1,1,2,0)$ in the robust code file can be obtained statistically, while the frequency vector $C_d(f_1) = (1,4,0,2,3,0)$ in the defect code file, based on $C_r(f_1)$ and $C_d(f_1)$, f1 can be calculated in the Wilcoxon rank sum test $p = 0.783$.

In this paper, we finally sort them according to the p value of each feature. The algorithm of feature sorting is shown in Tab. 4.

Table 4: Feature sorting algorithm

Algorithm: Ranking Features for OCDDM

input:

N-gram feature number N, feature set$F = \{f_1, f_2 \cdots f_N\}$,

Robust code source$SR = \{sr_1, sr_2 \cdots sr_I\}$

Defect code source$SD = \{sd_1, sd_2 \cdots sd_J\}$

output:

Feature sorting RF

1. define a list of features p-value: $P = \{p_1, p_2 \cdots p_N\}$
2. define a list of robust code vectors: $CR = \{C_r(f_1) \cdots C_r(f_I)\}$
3. define a list of defect code vectors: $CD = \{C_d(f_1) \cdots C_d(f_J)\}$
4. foreach p-value n=1 to N
5. initialize $p_n = 0$
6. for n=1 to N
7. for i=1 to I
8. $C_r(f_n) = C_r(f_n) \cup (sr_i.num(f_n))$
9. end for
10. for j=1 to J
11. $C_d(f_n) = C_d(f_n) \cup (sd_j.num(f_n))$
12. end for
13. $p_n = Wilcoxon's\ Test(C_r(f_n), C_d(f_n))$
14. end for
15. while $notNull(P)$ do
16. for n=1 to $P.Size$
17. if $isMin(p_n)$ then
18. $RF.add(f_n)$
19. $F.remove(f_n)$
20. $S.remove(s_n)$
21. end if
22. end for
23. end while

At this point, all features are ordered according to their strength in the hypothesis test.

The smaller the intensity, the smaller the value p, indicating that from a statistical point of view, the feature has a clear distinction between robust code and defect code, and can provide a greater contribution to the classifier.

5.2 Object-oriented code defect detection base on support vector machine

The essence of code defect detection is a binary classification that determines whether a code defect exists for a given code component. Different researchers often use different machine learning algorithms when classifying, such as support vector machines, logistic regression, neural networks, decision trees, random forests, and k-means. It is widely believed that this depends to a large extent on the characteristics of the training data and the categorical data. In the research of this paper, because the training data requires labeled source code, that is, which classes in the known code are robust and which are defective, the experimental sample size is relatively small. In addition, although feature selection has been made, the impact of high dimensional features on the classifier needs to be considered. This paper compares different classification algorithms and finally decides to use support vector machine (SVM) to solve object-oriented code defect analysis.

Support vector machine is a machine learning method based on statistical learning theory. Its biggest feature is to maximize the learning machine's generalization ability according to the Vapnik structure [Vapnik (1997)] risk minimization principle, that is, small errors obtained from a limited training set sample can still guarantee a small error for independent test sets [Rao, Dong and Yang (2003)]. Since the support vector algorithm is a convex optimization problem, the local optimal solution must be the global optimal solution, which is be-yond the reach of other machine learning algorithms [Rao, Dong and Yang (2003)]. In addition, the good adaptability of the support vector machine to high dimensional features also makes it have certain advantages in the problem to be solved in this paper.

6 Experimental implementation and results analysis

6.1 CPMMC model test

6.1.1 Experimental setup

According to the CPMMC model, the open source Java code is taken as a sample to predict the method calls in the code, verify the effect and performance of the CPMMC model, and analyze the applicable scope of the CPMMC model.

Data Collection. Validating the effect of the model in code prediction requires a large amount of object-oriented source code. The experiments in this article obtained 11 Java open source code scores above 1000 Stars on the GitHub website, with a total code of 139KLOC. In order to remove the interference of developers' custom classes on predictions, the experiment collected 4024 classes defined in the JavaTM Plat-form Standard Ed. 7 version, 51641 methods contained in these classes, and set it as the scope of experimental prediction.

Experimental Design. The experiment used 90% of the source code as the training set and the remaining 10% as the test set. we randomly delete 100 method calls in the test set, marked as the point P.P (Predicting Point) to be predicted, and ensure that the object

that calls P.P has at least involved the call of 2 methods.

The experiment is divided into a model training part and a code prediction part, as shown in Fig. 6. In the model training part, we analyze the training set, get the inter-mediate code. The call sequence of the method is extracted, and the obtained sequence is N-gram-segmented to construct a CPMMC model. In the code prediction section, we analyze the incomplete code fragments in the test set, and extract the method call sequence with P.P to build the CPMMC model. Then, according to the probability distribution of the sequence of feature words obtained by training, a recommendation list of P.P is given.

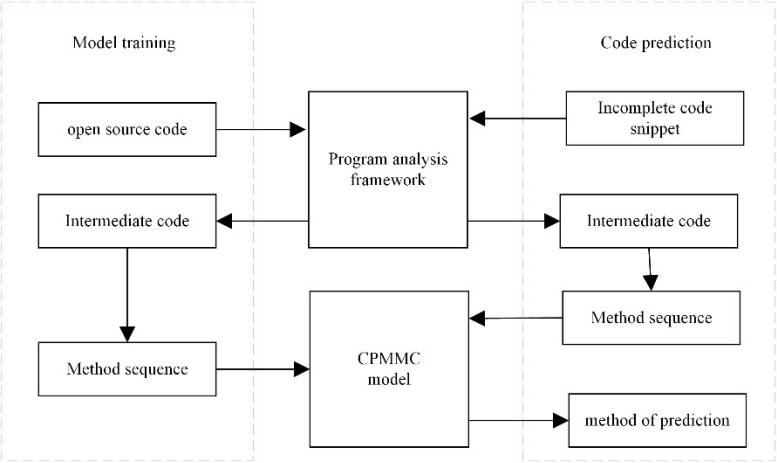

Figure 6: CPMMC model test experiment overall framework

In order to evaluate the accuracy of the prediction, the experiment designed three indicators, Top 1, Top 3 and Top 15. Top 1 indicates the probability that the first meth-od recommended by the model is the correct method. Top 3 indicates the probability that the first three methods in the recommendation list contain the correct method, and Top 15 indicates the probability that the first 15 methods in the recommendation list contain the correct method.

6.1.2 Comparative experiment and analysis

In order to evaluate the advanced nature of the CPMMC model, we compare the model of this paper with the SLANG (Statistical Language Model) model [Nicolai and Cheng (1981)] with the highest prediction accuracy. To verify the rationality of the model, we also built a CPMMC model with a list of parameters (represented by CPMMC_P). We used these three models to predict 100 identical P.P. The experimental results are shown in Fig. 7.

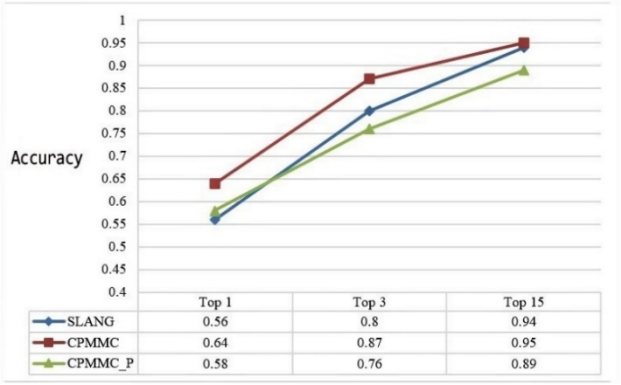

Figure 7: Experimental results of the three models

It can be seen that compared with the SLANG model, the accuracy of the CPMMC model is significantly improved, Top 1 is increased from 0.56 to 0.64, Top 3 is in-creased from 0.80 to 0.87, and Top 15 is increased from 0.94 to 0.95. This also verifies that the CPMMC model solves the problem of personalized naming of objects. And the addition of the constraint relationship of different elements in the same line of code significantly improves the accuracy of the prediction. The two models have little difference in Top 15 indicators, both of which are above 94%, mainly because both models can effectively capture the developer's rules for using Java methods as the indicators are relaxed.

By comparing the experimental results of the CPMMCP model and the CPMMC_P model, it can be found that the accuracy of the CPMMC_P model pre-diction is significantly reduced. On the surface, adding a parameter list will make the model data sparser. For example, the String class is common in source code. One of its methods indexOf has four methods of the same name indexOf(int), indexOf(string), indexOf(int, int), indexOf(string, int) after overloading. Although indexOf may get a higher probability value, if it is dispersed into 4 methods, the probability value of each method is not high. However, in essence, since the target of the prediction is the method name and does not go deep into the overload of the method, adding the parameter list will cause the over-fitting phenomenon, so the effect of the CPMMCP model is not good.

6.2 Code defect detection algorithm test

6.2.1 Experimental setup

Data Collection. In order to verify OCDDM's ability to detect object-oriented code defects, the experimental data selected Scandariato et al. [Scandariato, Walden, Hovsepyan et al. (2014)] provided four Android software source code with annotations in 2014, as shown in Tab. 5. These codes are available on the F-Droid and GitHub websites, which are of different types and have a large user base. Another reason for choosing these programs is that Scandariato et al. [Scandariato, Walden, Hovsepyan et al. (2014)] and Pang et al. [Pang, Xue, and Namin (2015)] and others have verified their proposed text-based analysis of code defect detection methods on this data set, and achieved good

results.

Table 5: Labeled experimental data

Software	Number of classes	Average size of the class
Anki-Android	255	18.95 K
Connectbot	235	6.41 K
CoolReader	94	13.18 K
k9Mail	519	11.96 K

Evaluation Index. In the field of statistics, in order to evaluate the performance of the classifier, it is often necessary to evaluate two important indicators of the classifier: Precision and Recall. The precision reflects the proportion of the true positive case in the positive case determined by the classifier; the recall reflects the proportion of the positive case correctly determined by the classifier in all true positive cases. Obviously the larger the two indicators, the better the performance of the classifier.

In order to calculate the above indicators, the experimental results are divided into four categories, the real example TP (true positive), false positive example FP (false positive), true counterexample TN (true negative), false counterexample FN (false negative). Precision and recall rates are calculated as follows:

$$Precision = \frac{TP}{TP+FP} \tag{11}$$

$$Recall = \frac{TP}{TP+FN} \tag{12}$$

Although in the ideal case, the better the precision and accuracy are higher, in reality, the two tend to have a negative correlation, while one increases and the other is likely to decrease. In order to better evaluate the effectiveness of our method, the value of F1 is used to comprehensively assess classification performance. The definition is as follows:

$$F1 = \frac{2\times Precision\times Recall}{Precision+Recall} \tag{13}$$

6.2.2 Comparative experiment and analysis

In order to verify the advanced nature of the method, the OCDDM method is com-pared with the method proposed by Scandariato et al. [Scandariato, Walden, Hovsepyan et al. (2014)] and Pang et al. [Pang, Xue, Namin et al. (2015)], which are in a leading state in the code defect detection method based on text analysis. For convenience of description, we will refer to these two methods as SM (Scandariato's Method) and PM (Pang's Method). In the experiment, we take all the features of N⩽3 in the N-gram segmentation, and take the top 5% to 20% of the features after sorting the features. We designed two experiments to compare these three methods.

Cross-Validation. This article uses a layered 5-fold cross-validation experiment for each application and provides support using the Weka [Holmes, Donkin and Witten (2002)]

tool. The tool randomly divides the file into five subsets, with the constraint that the ratio between the robust code file and the defect code file is the same in all subsets. Each subset is used as a test set in an iterative manner, and the remaining subsets are used as training sets. The experimental results are shown in Fig. 8.

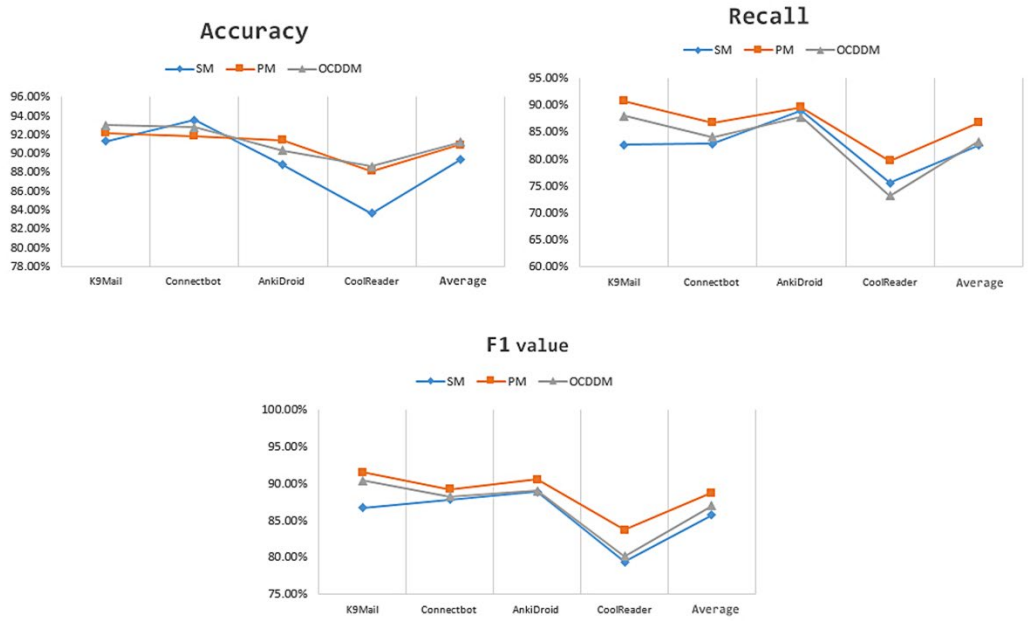

Figure 8: Cross-validation result

It can be seen from the experimental results that the PM method is higher than the SM method in terms of various indexes, which is the same as the text analysis theory, and the PM method is related to the improvement of the SM method.

Compared with the two, OCDDM has a better effect on detection accuracy, but the recall rate is lower than PM. This can explain two problems. One is that programming logic errors have a very high probability of causing code defects. Second, some code defects cannot be represented by the programming logic model (such as the overflow of the operation results).

It can be seen from the F1 value that the OCDDM method is superior to the SM method in overall performance and has little difference from the PM method. This shows that this method can effectively detect code defects in the same program detection.

Cross-Program Detection. Cross-program detection is a very important indicator of code defect detection. It embodies the versatility of the detection method, because it is difficult for the tester to label a part of the code file as a training set when testing each software. In order to verify the performance of each method in cross program detection, this paper uses all the Java files of each application as the training set, and the other three programs as the test set. The experimental results are shown in Fig. 9:

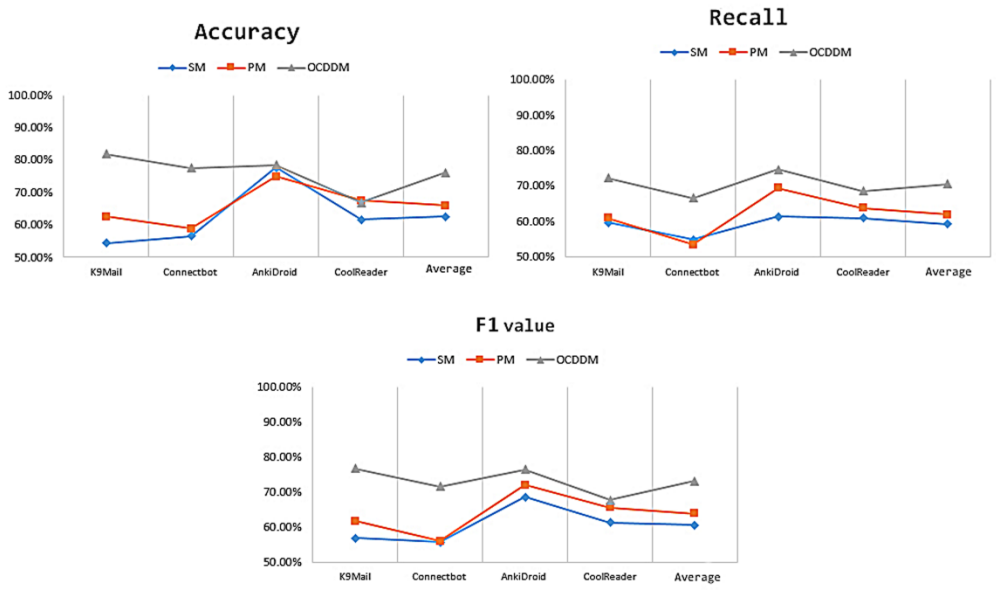

Figure 9: Cross-program test results

From the experimental results, in the cross-program detection, although compared with the experimental (1), the detection performance is difficult to avoid a large de-cline, but the OCDDM is better than the other two in the accuracy, recall and F1 value indicators. This is a delightful result, although different developers have different programming styles, but they still follow similar programming logic, thanks to the relatively standard way of using a large number of Java methods. At the same time, this set of experiments also shows that adding programming logic to text analysis can make code defect detection more versatile.

7 Conclusion

This paper takes open source code as the research object, and studies the general rules and personalization habits of human programming process, studies the object-oriented open source code programming logic modeling and application, and explores the potential logic and rules of human programming, aiming at code defects based on text analysis. The detection method is not effective in cross-program. A CPMMC based object-oriented code defect detection method is proposed, which uses programming logic instead of plain text analysis and N-gram technology to extract code features. The Wilcoxon rank sum test method is used to select the appropriate features and vectorize them, and finally realize the defect detection of the object-level code at the file level. Experiments show that this method can effectively detect the defect files in the object-oriented code, and the accuracy, recall rate and F1 value of the existing methods are improved in the cross-program detection.

The work of further research includes the following two aspects:

1. The object-oriented programming logic model constructed in this paper is at the

method level, which studies the logic of the developer when calling the method and does not analyze the finer granular programming behavior. In fact, in order to further implement technologies such as automated programming, it is necessary to extend the programming logic to the entire programming behavior, from the creation of more granular classes, the instantiation of objects, to the use of less granular keywords. Therefore, the next step will be to explore and model the developer's thinking logic at a fine-grained level.

2. The CPMMC-based object-oriented code defect detection method proposed in this paper improves the performance of the existing text-based defect detection technology in cross-program detection, but at present it can only effectively determine whether the file contains defects and it is difficult to determine the code. The specific location of the defect. Therefore, the next step will be to study the manifestation of code defects in programming logic, break the limitations of file detection, and conduct research on code body detection at the function body and fragment level.

Funding Statement: This work was supported by National Key RD Program of China under Grant 2017YFB0802901.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

Bielak, J; Biffl, S. (2003): Personal software process. *Computer Engineering & Applications*, vol. 13, no. 3, pp. 1-580.

Deng, L.; Ma, J.; Pei, J. (2004): Rank sum method for related gene selection and its application to tumor diagnosis. C*hinese Science Bulletin*, vol. 49, no. 15, pp. 1652-1657.

Eckel, B. (2005): Thinking in Java (Revision 4.0). http://www.mindview.net.

Gharehyazie, M.; Ray, B.; Keshani, M.; Zavosht, M. S.; Heydarnoori, A. et al. (2019). Cross-project code clones in GitHub. *Empirical Software Engineering*, vol. 24, no. 3, pp. 1538-1573.

Gruska, N.; Wasylkowski, A.; Zeller, A. (2010): Learning from 6,000 projects: lightweight cross-project anomaly detection. *Nineteenth International Symposium on Software Testing and Analysis*, pp. 119-130.

Hata, H.; Mizuno, O.; Kikuno, T. (2010): Fault-prone module detection using large-scale text features based on spam filtering. *Empirical Software Engineering*, vol. 15, no. 2, pp. 147-165.

Hindle, A.; Barr, E. T.; Gabel, M; Su, Z.; Devanbu, P. (2016): On the naturalness of software. *Communications of the ACM*, vol. 59, no. 5, pp. 122-131.

Holmes, G.; Donkin, A.; Witten, I. H. (2002): WEKA: a machine learning workbench. *Proceedings of ANZIIS '94-Australian New Zealnd Intelligent Information Systems Conference*, pp. 357-361.

ISO/IEC/IEEE International Standard (2017): ISO/IEC/IEEE international standard-systems and software engineering-vocabulary. *ISO/IEC/IEEE* 24765:2017(E).

Mizuno, O.; Ikami, S.; Nakaichi, S.; Kikuno, T. (2007): Spam filter based approach for finding fault-prone software modules. *International Workshop on Mining Software Repositories*, pp. 1-4.

Möller, C.; DreborgM, S.; Ferdousi, H. A.; Halken, S.; Høst, A. et al. (2002): Pollen immunotherapy reduces the development of asthma in children with seasonal rhinoconjunctivitis (the PAT-study). *Journal of Allergy and Clinical Immunology*, vol. 109, no. 2, pp. 251-256.

Neuhaus, S.; Zimmermann, T.; Holler, C.; Zeller, A. (2007): Predicting vulnerable software components. *ACM Conference on Computer and Communications Security*, pp. 529-540.

Nicolai, M.; Cheng, R. (1981): SLANG, a statistical language for descriptive time series analysis. *Computer Science and Statistics: Proceedings of the 13th Symposium on the Interface*, pp. 320-323.

Pang, Y.; Xue, X.; Namin, A. S. (2015): Predicting vulnerable software components through N-gram analysis and statistical feature selection. *IEEE International Conference on Machine Learning and Applications*, pp. 543-548.

Raychev, V.; Vechev, M.; Yahav, E. (2014): Code completion with statistical language models. *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, vol. 49, no. 6, pp. 419-428.

Rao, X.; Dong, C.; Yang, S. (2003): Statistic learning and intrusion detection. Wang, G., Liu, Q., Yao, Y., Skowron, A. (eds.) Rough sets, fuzzy sets, data mining, and granular computing. *Lecture Notes in Computer Science*, vol. 2639, pp. 652-659.

Scandariato, R.; Walden, J.; Hovsepyan, A.; Joosen, W. (2014): Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 993-1006.

Shar, L. K.; Tan, H. B. K. (2012): Predicting common web application vulnerabilities from input validation and sanitization code patterns. *IEEE/ACM International Conference on Automated Software Engineering*, pp. 310-313.

Shar, L. K.; Tan, H. B. K. (2013): Predicting SQL injection and cross site scripting vulnerabilities through mining input sanitization patterns. *Information & Software Technology*, vol. 55, no. 10, pp. 1767-1780.

Shar, L. K.; Briand, L. C.; Tan, H. B. K. (2015): Web application vulnerability prediction using hybrid program analysis and machine learning. *IEEE Transactions on Dependable & Secure Computing*, vol. 12, no. 6, pp. 688-707.

Siegal, S. (1988): *Nonparametric Statistics for the Behavioral Sciences, Second Edition.* McGraw-Hill Humanities.

Sood, P.; Krek, A.; Zavolan, M.; Macino, G.; Rajewsky, N. (2006): Cell-type-specific signatures of microRNAs on target mRNA expression. *Proceedings of the National Academy of Sciences of the United States of America*, vol. 103, no. 8, pp. 2746-2751.

Vapnik, V. N. (2000): The nature of statistical learning theory, second edition. *Statistics for Engineering and Information Science*, pp. 1-314

Xue, X.; Pang, Y.; Namin, A. S. (2015): Feature selections for effectively localizing faulty events in GUI applications. *International Conference on Machine Learning and Applications*, pp. 306-311.

Yamaguchi, F.; Lindner, F.; Rieck, K. (2011): Vulnerability extrapolation: assisted discovery of vulnerabilities using machine learning. *USENIX Workshop on Offensive Technologies*, pp. 13-13. San Francisco, CA, USA.

Yamaguchi, F.; Lottmann, M.; Rieck, K. (2012): Generalized vulnerability extrapolation using abstract syntax trees. *Computer Security Applications Conference*, pp. 359-368. Orlando, FL, USA.

Yao, Y.; Huang, S.; Feng, C.; Liu, C.; Xu, C. (2019): CD3T: cross-project dependency defect detection tool. *International Journal of Performability Engineering*, vol. 15, no. 9, pp. 2329-2337.

Zhang, W.; Li, K.; Li, T.; Niu, S.; Gao, Z. (2019): CM-Droid: secure container for android password misuse vulnerability, *Computers, Materials & Continua*, vol. 59, no. 1, pp. 181-198.

Zimmermann, T.; Nagappan, N; Williams, L. (2010): Searching for a needle in a haystack: predicting security vulnerabilities for windows vista. *International Conference on Software Testing*, pp. 421-428.