修　士　論　文

題　目　　ソースコードにおけるトークン出現傾向の
不具合予測への応用に関する研究

主任指導教員　　水野　修　准教授

京都工芸繊維大学大学院 工芸科学研究科

情報工学専攻

学生番号　10622031

氏　　名　平田　幸直

平成24年2月10日提出

様式 2 号

学位論文の要旨（和文）

平成 24 年 2 月 10 日

京都工芸繊維大学大学院
工芸科学研究科長　殿

工芸科学研究科	情報工学専攻
平成 22 年入学	
学生番号	10622031
氏　　名	平田　幸直　㊞

（主任指導教員　　水野　修　㊞　）

本学学位規則第 4 条に基づき、下記のとおり学位論文内容の要旨を提出いたします。

1.　論文題目

ソースコードにおけるトークン出現傾向の不具合予測への応用に関する研究

2.　論文内容の要旨（400 字程度）

ソフトウェアの開発において不具合は品質とコストに悪影響を与える．そのため，不具合を予測するためのさまざまな手法が提案されている．現在，行われている研究のほとんどはプロジェクトの過去の情報を用いるため，新規開発のプロジェクトに対しては利用することができない．この問題に対して，他のプロジェクトの情報を利用することにより過去の情報がなくても予測が可能なプロジェクト横断型予測がある．本研究ではモジュール内のトークンを利用して不具合を予測する Fault-prone フィルタリングを用いてプロジェクト横断型予測と従来のプロジェクト内部型予測を 28 バージョン (8 プロジェクト) のデータに対して行う．これらの予測実験を通して，Fault-prone フィルタリングにおいて，予測モデルは多変数ベルヌーイモデルが適していることを確認した．また，モジュール内の全てのトークンを利用する場合が最も性能が良く，コードのみを利用した場合は適合率が良くなることを確認した．そして，プロジェクト横断型予測が可能であり，プロジェクト内部型予測よりも再現率が高くなることを示した．

Application of Trend of Tokens in Source Code Modules to Fault-prone Module Prediction

2012 10622031 Yukinao HIRATA

Abstract

In the software development, defects affect quality and cost in an adverse way. Therefore, various studies have been proposed defect prediction techniques. Most of current defect prediction approaches use past project data for building prediction models. That is, these approaches are difficult to apply new development projects without past data. In this study, we focus on the cross project prediction that can predict faults of target projects by using other projects. We use 28 versions of 8 projects to conduct experiments of the cross project prediction and intra-project prediction using the fault-prone filtering technique. Fault-prone filtering is a method that predicts faults using tokens from source code modules. Additionally, we try to find an appropriate prediction model in the fault-prone filtering, since there are several ways to calculate probabilities. From the results of experiments, first, we conclude that the multi-variate Bernoulli model is appropriate to the fault-prone filtering. Second, we show that using tokens extracted from all parts of modules is the best way to predict faults and using tokens extracted from code part of modules shows better precision. We also show that the results of the cross project predictions have better recall than the results of the intra-project predictions.

Contents

1. Introduction

In software development defects affect quality and cost in an adverse way. Therefore, various studies have been proposed the defect prediction techniques. Most of current studies that predict defects use past project data for building a prediction model. These prediction models can capture characteristics of projects. However, if there is no data about past projects, it is difficult to use these approaches. That is, these approaches are difficult to apply new development projects. There are studies [1–5] for solving such a problem. These studies have been aimed to predict defects in a project using other project data. We call such types of prediction using other project as the cross project prediction. The predictive ability of the cross project prediction depends on characteristics between training project and target project (i.e. similar characteristics lead good result). For this reason, the study [6] has been conducted to perform clustering on software projects in order to identify groups of software projects with similar characteristic.

These studies often use basic software metrics like the line of code (LOC), cyclomatic complexity, CK object oriented metrics [7] and Code Churn [8]. However, we use a method called fault-prone filtering based on text filtering for predicting defects in this study. This approach uses tokens extracted from modules as a metric for prediction. We can extract these tokens from kind of part in modules (e.g. code and comment). In addition, there are no steady way of selecting filter. Therefore, this approach needs to select a kind of tokens and a filter that we use to predict as well as other approaches select metrics and learning algorithms. In addition, the ability of our approach depend on tokens contained in modules. Hence, if we conduct fault-prone filtering in cross project prediction situation, it is a legitimate question whether we can predict faults.

We conduct experiments using 28 versions (8 projects) and investigate related to following issues:

- Which classification model is appropriate to fault-prone filtering?

  There are the multi-variate Bernoulli model and the multinomial model in naive Bayes classifier. In general, the multinomial model shows better results in document classification context. However, it is not known know that which model shows better results in fault-prone filtering.

- Which class of tokens show good prediction?

  We can extract classes of tokens from different part of modules. We guess these different class of tokens show different prediction trends.

- Can we conduct the cross project prediction using fault-prone filtering?

  Tokens are most important in the fault-prone filtering. Our approach needs tokens that are contained in both training and target projects to conduct cross project prediction.

Main findings of this study is summarized as follow:

- The multi-variate Bernoulli model is better than the multinomial model in fault-prone filtering.
- Using all tokens extracted from modules is the best way to predict faults from the viewpoint of F_1 value. If we need better precision, we can use tokens extracted from code and end-of-line comment. In addition, we show that we can predict faults using structures of lines.
- Intra-project prediction show better precision than cross project prediction. In contrast, recall is better in cross project prediction.

The remainder of this paper is structured as follows: In Section 2, we describe fault-prone filtering. In Section 3, we explain target projects and evaluation measures. In Section 4, we describe experiments and show results of predictions related to the intra-project prediction. In Section 5, we describe experiments and show results of predictions related to the cross project prediction. In Section 6, we describe related works. In Section 7, we discuss threats to validity. Conclusions are given in Section 8.

2. Fault-Prone Filtering

2.1 Basic Idea

The basic idea of fault-prone filtering [9] is inspired by the spam mail filtering. In the spam e-mail filtering, the spam filter first trains both spam and ham e-mail messages from the training data set. Then, an incoming e-mail is classified into either ham or spam by the spam filter.

This framework is based on the fact that spam e-mail usually includes particular patterns of words or sentences. From the viewpoint of source code, a similar situation usually occurs in faulty software modules. That is, similar faults may occur in similar contexts. We thus guessed that faulty software modules have similar patterns of words or sentences as spam e-mail messages have. To obtain such features, we adopted the spam filter in fault-prone module prediction.

Intuitively speaking, we try to introduce a new metric as a fault-prone predictor. The metric is "frequency of particular words". In detail, we do not treat a single word, but use combinations of words for the prediction. Thus, the frequency of a certain length of words is the only metric used in our approach.

We then try to apply a spam filter to identify fault-prone modules. We named this approach as "fault-prone filtering". That is, a learner first trains both faulty and non-faulty modules. Then, a new module can be classified into either fault-prone or not-fault-prone using a classifier.

2.2 Extraction of Tokens

We conduct experiments using the fault-prone filtering in this study. In these experiments, we use tokens extracted from specific parts of modules (e.g. code and comment). We explain the specific parts of modules and how to extract tokens from each part in this section.

2.2.1 Definition of Comment Lines

In order to investigate the ability of fault-prone prediction, we distinguish the contents of source code modules into two classes. Code lines describe a list of operations that developers would like to realize on computers. Comment lines include descriptions of code lines, usage of methods or modules. Code lines are written in a specific programming language, but comment lines are written in a free form.

Original implementation of fault-prone filtering did not distinguish the comment lines and code lines. The source code module is passed into text filter without any modification. However, since code lines and comment lines have different roles in source code modules, we need to consider such difference in the fault-prone filtering.

For example, comments are usually placed near the difficult codes. Therefore, learning the contents of comment lines may be useful to identify the bug-related part in modules. Actually, previous research [10] shows that the prediction using tokens in comments is better than the prediction of tokens in code.

In this study, we treat a java class file as a software module.Developers can write comments anywhere in the module. Those comments are classified into three types by the written form in Java [11]. First, there is a comment starting with "//". This type of comment is called the End-Of-Line Comments (EOL). EOL is treated as a comment from "//" to the end of line. Second, a comment that starts with "/*" and ends with "*/" is either the Block comment or the Single-Line comment or the Trailing comment (BLK). BLK treats the enclosed part as a comment. Therefore, it can comment out the part of line, or multiple lines. Finally, a comment used for the tool that is called javadoc is Documentation comment (DOC). DOC is a comment that starts with "/**" and ends with "*/". This comment is used to explain classes and methods. Comments mentioned above are summarized in Table 2.1. In this research, as shown in Table 2.1, comment is defined as a combination of three different comment types.

Table 2.1 **Definition of Comments**

Comment Type	Definition
COM	//, /* */, /** */
EOL	//
BLK	/* */
DOC	/** */

2.2.2 Structure of Line

Copied and pasted source code, which is called "code clone", affects to software quality. There are several studies [12, 13] that detect faults using code clone. We consider that these studies use structure of code clone to detect faulty code. Therefore, we guess that structure of code affect to faults. Code clones are parts of structure in module. In contrast, we use all of structure in modules to predict faults because we guess that all structures in a module affect to the trend of faults of the module.

Next, we explain how to extract structure from modules. In order to extract structure, we apply following steps to modules.

Step 1 Remove all types of comments, '{' and '}'.

Step 2 Replace identifiers include numbers, strings with double quotes and character with single quotes into I, S and C, separately.

Step 3 Remove all white spaces and tabs.

Applying by these steps, we get structure of modules. In this study, we use each lines as tokens for fault-prone filtering. We define a class that consist of these tokens as TC_{LINE}.

2.2.3 Tokenization

In this study, we define 8 kinds of token class extracted from the source code: TC_{ALL}, TC_{CODE}, TC_{EOL}, $TC_{CODE+EOL}$, TC_{COM}, TC_{BLK}, TC_{DOC} and TC_{LINE}. Here, TC_{CODE} token class includes tokens extracted from code except comments in modules. TC_{ALL} token class includes TC_{CODE} and TC_{COM} token classes. That is, TC_{ALL} token class uses all part of a module. In similar way, $TC_{CODE+EOL}$ token class includes TC_{CODE} and TC_{EOL} token classes. The reason that we use this combination is TC_{EOL} shows that better prediction result in previous study [14].

We conduct to tokenize above all token classes in same way except TC_{LINE} token class (tokenization of TC_{LINE} class already explained in section 2.2.2). We tokenize contents into identifiers, numbers, escape sequence, keywords, operators in Java language.In this study, we admit that identifiers include dots. Therefore, `System.out.println` is a token (not three tokens). In addition, we treat double quote as separator. Therefore, `"Hello world"` separated into `Hello`, `world`. However, character include escape sequence with single quote as a token (e.g. `'c'` is a token).

2.3 Naive Bayes Classifier

The naive Bayes classifier is a traditional document classifier [15, 16]. In the naive Bayes, we can calculate $P(d|c)$ (the generation probability of document d in class c) under assumption of words are independent of each other words in a document. In general, this assumption is not correct, however, the naive Bayes classifier can classify documents well. Hence, this classifier is still often used.

2.3.1 Generative Models

There are two popular models, the multi-variate Bernoulli model and the multinomial model, in the naive Bayes classifier. Here, we explain the models and application to fault-prone filtering.

(1) Multi-variate Bernoulli Model

In the multi-variate Bernoulli model, a document is represented by a vector that shows each tokens occur or not in the document. That is, it is important that tokens exist or not in the document. Therefore, if a token does not exist in a document, we use the data about "the token does not exist". However, we use only binary data about tokens, hence, we lose the number of occurrence of a token in this model.

Next, we explain the application of the multi-variate Bernoulli model to fault-prone filtering. The application using the naive Bayes classifier consist of following two algorithms. (1) In training algorithm, we calculate the probability using tokens and modules extracted from training modules. (2) In classification algorithm, we use the probability to classify target modules.

Training Algorithm

In order to explain this algorithm, we define some symbols about modules. M_{train} and M_{target} are sets of training and target modules, respectively. $C = \{faulty, nonfaulty\}$ is a set of class labels. All modules belong to either $faulty$ or $nonfaulty$. c denotes a module includes faults or not and $c \in C$. If a module includes one or more faults, $c = faulty$; otherwise, $c = nonfaulty$.

First, we count following numbers using M_{train}.

- $N_{t,c}$: The number of modules that contain token t in class c.
- N_c: The number of modules in class c.

These numbers are counted as follow. We classify all modules in M_{train} based on its class c and count the number of modules in each class to obtain N_c. Then, we tokenize and count the number of modules containing token t for each class.

Next, We calculate the probability of a module containing token t in class c and the probability of a module in class c using $N_{t,c}$ and N_c. We denote these probability $p_{t,c}$ and p_c, respectively and define them as follows:

$$p_{t,c} = \frac{N_{t,c} + 1}{N_c + 2} \tag{2.1}$$

$$p_c = \frac{N_c + 1}{\sum_{c \in C} N_c + |C|} \tag{2.2}$$

<u>Classification Algorithm</u>

In classification algorithm, we tokenize $m_{target} \in M_{target}$ and calculate Equation (2.3) using tokens extracted from m_{target} for each class. P_c in this equation means the probability of m_{target} belonging to class c.

$$P_c = p_c \times \prod_{t \in m_{target}} p_{t,c} \times \prod_{t \notin m_{target}} (1 - p_{t,c}) \tag{2.3}$$

Next, we calculate Equation (2.4). If $P > 0.9$, m_{target} is classified faulty.Otherwise, the module is classified nonfaulty.

$$P = \frac{P_{faulty}}{P_{faulty} + P_{nonfaulty}} \tag{2.4}$$

We repeat this classification algorithm for all modules in M_{target}.

(2) Multinomial Model

In the multinomial model, a document is represented by a combination of occurrence of tokens in the document. Unlike in the case of the the multi-variate Bernoulli model, This model utilize the number of occurrences of tokens. However, tokens that are not exist in a target document do not used in this model. In general, the multinomial model is better than the multi-variate Bernoulli model in document classification [16].

Training Algorithm

First, we count $n_{t,c}$ defined as follow and N_c for all modules in M_{train}. Here, N_c is same in the multi-variate Bernoulli model.

- $n_{t,c}$: The number of token t in class c.

The way of counting is almost same in the multi-variate Bernoulli model. The difference is that we count the number of occurrence of token t for each class c in the multinomial model.

By using $n_{t,c}$ and N_c, we calculate the probability of occurrence of token t in class c and p_c. Here, T denote a set of tokens in training modules.

$$p_{t,c} = \frac{n_{t,c} + 1}{\sum\limits_{\acute{t} \in T} n_{\acute{t},c} + |T|} \tag{2.5}$$

$$p_c = \frac{N_c + 1}{\sum\limits_{c \in C} N_c + |C|} \tag{2.6}$$

Classification Algorithm

We tokenize and calculate Equation (2.7) to obtain the trends of module for each target module $m_{target} \in M_{target}$. Here, $n_{t,target}$ denotes the number of occurrence of token t in the module m_{target}. In this Equation (2.7), we use the data about how many and which token exist in module m_{target}.

$$P_c = p_c \times \prod_{t \in m_{target}} p_{t,c}^{n_{t,target}} \tag{2.7}$$

Next, we calculate the probability to be faulty about the m_{target} by Equation (2.4). As is the case with the multi-variate Bernoulli model, if $P > 0.9$, m_{target} is classified faulty.Otherwise, the module is classified nonfaulty.

3. Target Projects and Evaluation Measures

3.1 Target Projects

In this study, we use 8 projects, Apache Ant, Eclipse, jEdit, Apache Lucene, Apache POI, Apache Velocity, Apache Xalan and Apache Xerces, for our experiments. In order to do our experiments, we need to collect faulty information of modules in each projects. We adopt faulty data in PROMISE [17] repository. There are about one hundred data about faults. In our approach, we need information about modules and modules themselves. Therefore, we use data that contain modules name in order to satisfy our requirement. Consequently, we select 8 projects described above. Seven of 8 projects data except Eclipse are donated by Jureczko [6, 18] and Eclipse are donated by Zimmermann [19].

These datasets contain the number of faults for each modules. However, our approach determines existence or non-existence about faults. Therefore, we define a faulty module as a module that contain one or more faults and if a module does not contain any faults, we define the module as a non-faulty module. We show the number of faulty modules and non-faulty modules for each versions in Table 3.1. As shown in the Table 3.1, we use some versions for each projects. The number of modules in this table is fewer than the number of modules in the dataset because we only use data about modules that contained in each release versions. This happen, if the datasets include data about modules that was made between releases.

Next, we explain each project in brief.

- Ant[1] is a build tool for Java Language.
- Eclipse[2] is well known IDE (Integrated Development Environment).
- jEdit[3] is a text editor for programmers.
- Lucene[4] is a text search engine library.
- POI[5] is a library for reading and writing Microsoft Office files.

[1] http://ant.apache.org/
[2] http://eclipse.org/
[3] http://jedit.org/
[4] http://lucene.apache.org/
[5] http://poi.apache.org/

Table 3.1 Target Versions

project	faulty	non-faulty	ratio of faulty	project	faulty	non-faulty	ratio of faulty
Ant1.4	40	137	22.6 %	POI1.5	141	94	60.0 %
Ant1.5	32	260	11.0 %	POI2.0	37	272	12.0 %
Ant1.6	92	258	26.3 %	POI2.5	247	132	65.2 %
Ant1.7	166	575	22.4 %	POI3.0	281	157	64.2 %
Eclipse2.0	975	5754	14.5 %	Velocity1.4	147	48	75.4 %
Eclipse2.1	854	7034	10.8 %	Velocity1.5	142	72	66.4 %
Eclipse3.0	1568	9025	14.8 %	Velocity1.6	78	151	34.1 %
jEdit3.2	90	170	34.6 %	Xalan2.4	110	566	16.3 %
jEdit4.0	75	218	25.6 %	Xalan2.5	387	375	50.8 %
jEdit4.1	79	221	26.3 %	Xalan2.6	411	464	47.0 %
jEdit4.2	48	307	13.5 %	Xerces1.2	71	368	16.2 %
jEdit4.3	11	476	2.3 %	Xerces1.3	69	383	15.3 %
Lucene2.0	91	95	48.9 %	Xerces1.4	210	118	64.0 %
Lucene2.2	143	91	61.1 %				
Lucene2.4	203	127	61.5 %				

- Velocity[6] is a templating engine that provide a simple template language to reference objects defined in Java code.
- Xalan[7] is an XSLT processor for transforming XML documents into other XML document types.
- Xerces[8] is a processor for parsing, validating, serializing and manipulating XML.

3.2 Evaluation Measures

Table 3.2 shows a classification result matrix. True negative (TN) shows the number of modules that are classified as non-fault-prone, and are actually non-faulty. False positive (FP) shows the number of modules that are classified as fault-prone, but are actually non-faulty. On the contrary, false negative (FN) shows the number of modules that are classified as non-fault-prone, but are actually faulty. Finally, true positive (TP) shows the number of modules that are classified as fault-prone which are actually faulty.

In order to evaluate the results, we prepare three measures: recall, precision, and accuracy. Recall is the ratio of modules correctly classified as fault-prone to the number of entire faulty modules. Recall is defined as by Equation (3.1).

$$\text{Recall} = \frac{TP}{TP + FN} \tag{3.1}$$

Precision is the ratio of modules correctly classified as fault-prone to the number of entire modules classified fault-prone. Precision is defined as by Equation (3.2).

$$\text{Precision} = \frac{TP}{TP + FP} \tag{3.2}$$

Accuracy is the ratio of correctly classified modules to the entire modules. Accuracy is defined as by Equation (3.3).

$$\text{Accuracy} = \frac{TP + TN}{TN + TP + FP + FN} \tag{3.3}$$

[6] http://velocity.apache.org/
[7] http://xml.apache.org/xalan-j/
[8] http://xerces.apache.org/xerces-j/

Table 3.2 Classification Result Matrix

		Classified	
		non-fault-prone	fault-prone
Actual	non-faulty	True negative (TN)	False positive (FP)
	faulty	False negative (FN)	True positive (TP)

Since recall and precision are in the trade-off, F_1-measure is used to combine recall and precision. F_1-measure is defined as by Equation (3.4).

$$F_1 = \frac{2 \times \text{Recall} \times \text{Precision}}{\text{Recall} + \text{Precision}} \tag{3.4}$$

In this definition, recall and precision are evenly weighed.

In addition, we introduce an evaluation measure related to precision. Precision defined by Equation (3.2) is affected by ratio of faulty modules (see Table 3.1). For example, if ratio of faulty modules of the target version is high, precision also tends to be high in general. In order to make a new evaluation measure that eliminate the effect of ratio of faulty modules, first, we define the ratio of faulty using TN, TP, FP, FN as follow:

$$F_r = \frac{TP + FN}{TN + TP + FP + FN} \tag{3.5}$$

Next, we define a new evaluation measure by Equation (3.6) using Precision and F_r. This measure reduces the impact of ratio of faulty modules.

$$PF_r = \text{Precision} - F_r \tag{3.6}$$

4. Intra-Project Prediction

In this section, we conduct experiments related to intra-project prediction using fault-prone filtering. Intra-project prediction is a prediction that selects versions from same project as training and target data. That is, we predict newer version using previous version of same project. If we have one version except target versions, we can adopt this approach. Hence, when it is compared to the prediction that uses a part of target project as training data, intra-project prediction can be applied early in the development.

4.1 Ex. 1: Comparing Multi-variate Bernoulli Model and Multinomial Model

As already mentioned in subsection 2.3, there are two famous models, the multi-variate Bernoulli model and the multinomial model, for document classification in the naive Bayes. In general, it is said that the multinomial model shows better results in document classification [16]. However, we don't know that which model is better in fault prediction context. In ex. 1, we conduct comparative experiments using these two models in order to investigate which model is suitable for fault-prone filtering.

4.1.1 Experiment Method

We conduct comparative experiments using the multi-variate Bernoulli model and the multinomial model. When we apply fault-prone filtering to predict faults, we need to determine three things. 1) how to select training and target data, 2) kind of token classes we use, 3) kind of filters we use. All experiments in this section are intra-project prediction. Therefore, we decide training and target data from same project. In addition, we use only one version as a training data because this condition is very simple and likely to happen from a practical point of view.

First of all, we describe how to select training and target data. We use newer version as target data and older version as training data in the same project. we make combinations of new and old versions for 8 target projects. We show all combinations as an example in Ant project. Here, we represent training and target versions using a pair like (training, target). There are 6 combinations like (Ant1.4, Ant1.5), (Ant1.4, Ant1.6), (Ant1.4, Ant1.7), (Ant1.5, Ant1.6), (Ant1.5, Ant1.7), (Ant1.6, Ant1.7) in Ant project. Next, we use TC_{ALL} token class extracted from modules. That is, we use all code lines and all comment lines in modules. Finally, we conduct experiments using the multi-variate Bernoulli model and the multinomial model in above condition.

4.1.2 Results and Discussions

Figure 4.1 shows the result of ex. 1. From Figure 4.1, we find that F_1 value is almost the same in both models. However, recall and precision have different trends. The multi-variate Bernoulli model shows better precision than the multinomial model. In contrast, the multinomial model shows better recall than the multi-variate Bernoulli model. Therefore, we guess that there are no significant difference in prediction ability between both models from the viewpoint of F_1 values. However, from the viewpoint of precision and recall, when the multi-variate Bernoulli model predicts a module as fault prone, the module seems more likely to include actual faults, and the multinomial model can find more faults. These trends could be due to the major difference between two models, that is, how to use the number of occurrence of each tokens in module. The probability of each token ($p_{t,c}$) is implicit in fault prone filtering because features of modules are also implicit. When we use the multinomial model for prediction, the result is significantly affected by tokens that occur frequently in module. Therefore, if the probability of tokens of frequent occurrence is wrong, the result will be wrong. In particular, faulty modules tend to be large than non-faulty modules. This might cause unfair bias to frequent tokens.

We guess that it is a situation-dependent issue to decide which model is better. Prediction results using the multi-variate Bernoulli model show higher precision and lower recall than prediction results using the multinomial model. These results means that the multi-variate Bernoulli model tends to predict modules as non-fault-prone and the multinomial model tends to predict modules as fault-prone. That is, if we use the multinomial model, we obtain more fault-prone modules. As the fault-prone modules increase, it will be difficult to allocate the resource for tests. It is said that the result of high recall and

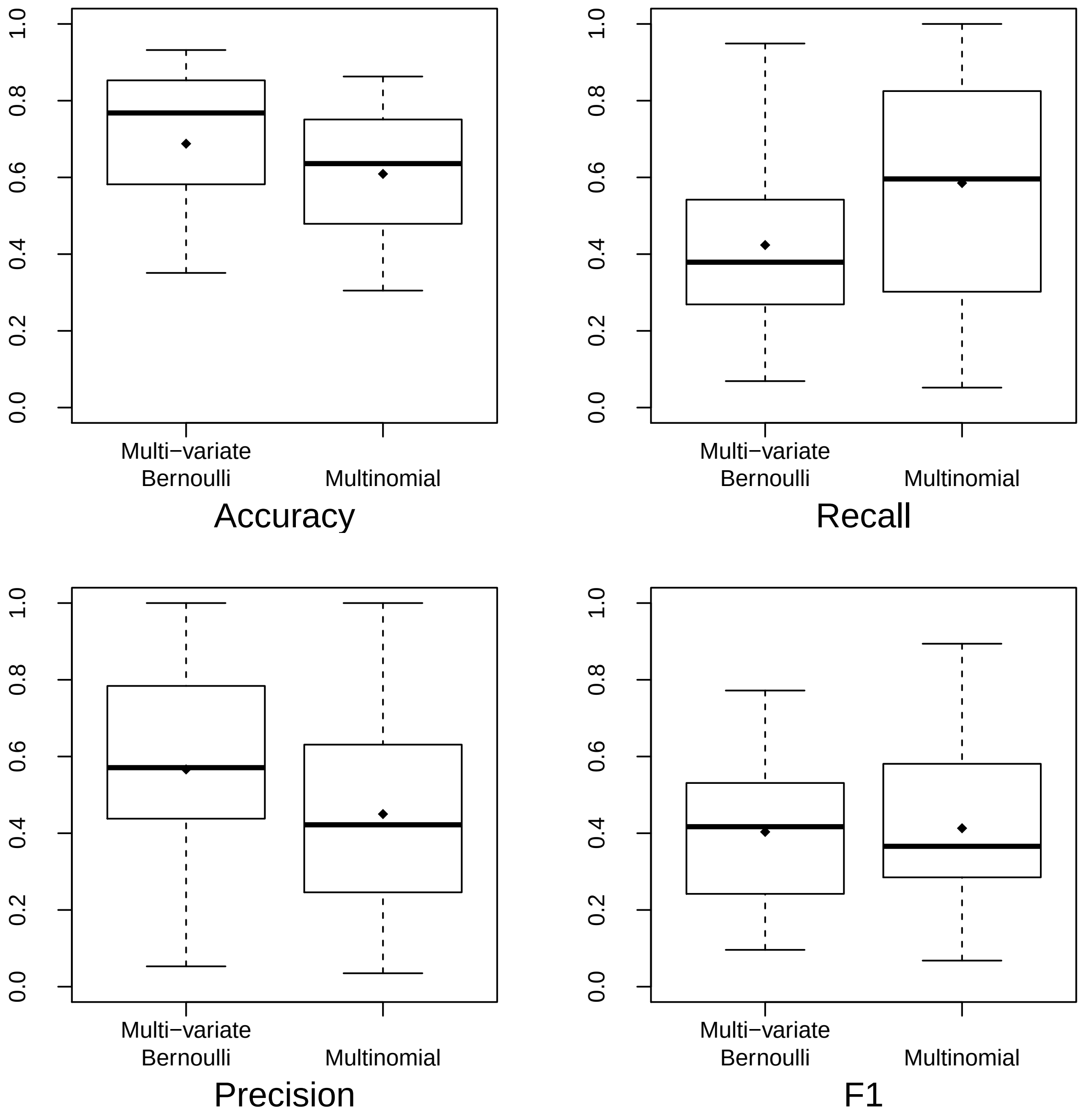

Figure 4.1 Comparison of Prediction Result of Multi-variate Bernoulli Model and Multinomial Model

low precision is also useful in some studies [20, 21]. However, we think that developers prefer to be better precision in a practical situation. Therefore, we use the multi-variate Bernoulli model in all experiments in the successive sections.

4.2 Ex. 2: Effect of Each Content Extracted from Modules in Fault Prone Filtering

We can divide contents of modules into some kinds of parts like code lines and comment lines. In previous studies [10, 14, 22], we conduct fault-prone filtering using some kinds of contents in modules and find that different kinds of contents show different kind of results. However, we use only one or two projects as target project in these studies. Therefore, we conduct fault-prone filtering using more projects and more contents of modules to investigate the trends of results in this experiment.

4.2.1 Experiment Method

In ex. 2, we conduct experiments using tokens extracted from different parts of module. Specifically, we use TC_{CODE}, TC_{COM}, TC_{EOL}, TC_{BLK}, TC_{DOC}, $TC_{CODE+EOL}$ and TC_{LINE} for predicting faults about modules. That is, we used all tokens extracted from module in ex. 1, however, if we use TC_{CODE} to conduct experiments in ex. 2, we extract code from modules in training and target versions. Ex. 2 largely similar to ex. 1 except what token classes we use. In short, we use same combination of training and target versions in ex. 1, and select the multi-variate Bernoulli model from the results of ex. 1.

4.2.2 Results and Discussions

Figure 4.2 shows statistical summary of recall and precision using different kinds of token classes. The result in Figure 4.2 shows that TC_{ALL}, TC_{CODE}, TC_{EOL}, $TC_{CODE+EOL}$ and TC_{LINE} are better in view point of precision and TC_{COM}, TC_{BLK} and TC_{DOC} are better in view point of recall. However, F_1 value of TC_{COM}, TC_{BLK} and TC_{DOC} are low. Therefore, we should select TC_{ALL}, TC_{CODE}, TC_{EOL}, $TC_{CODE+EOL}$ and TC_{LINE} to predict faults, if we use fault-prone filtering. Table 4.1 shows average PF_r in each experi-

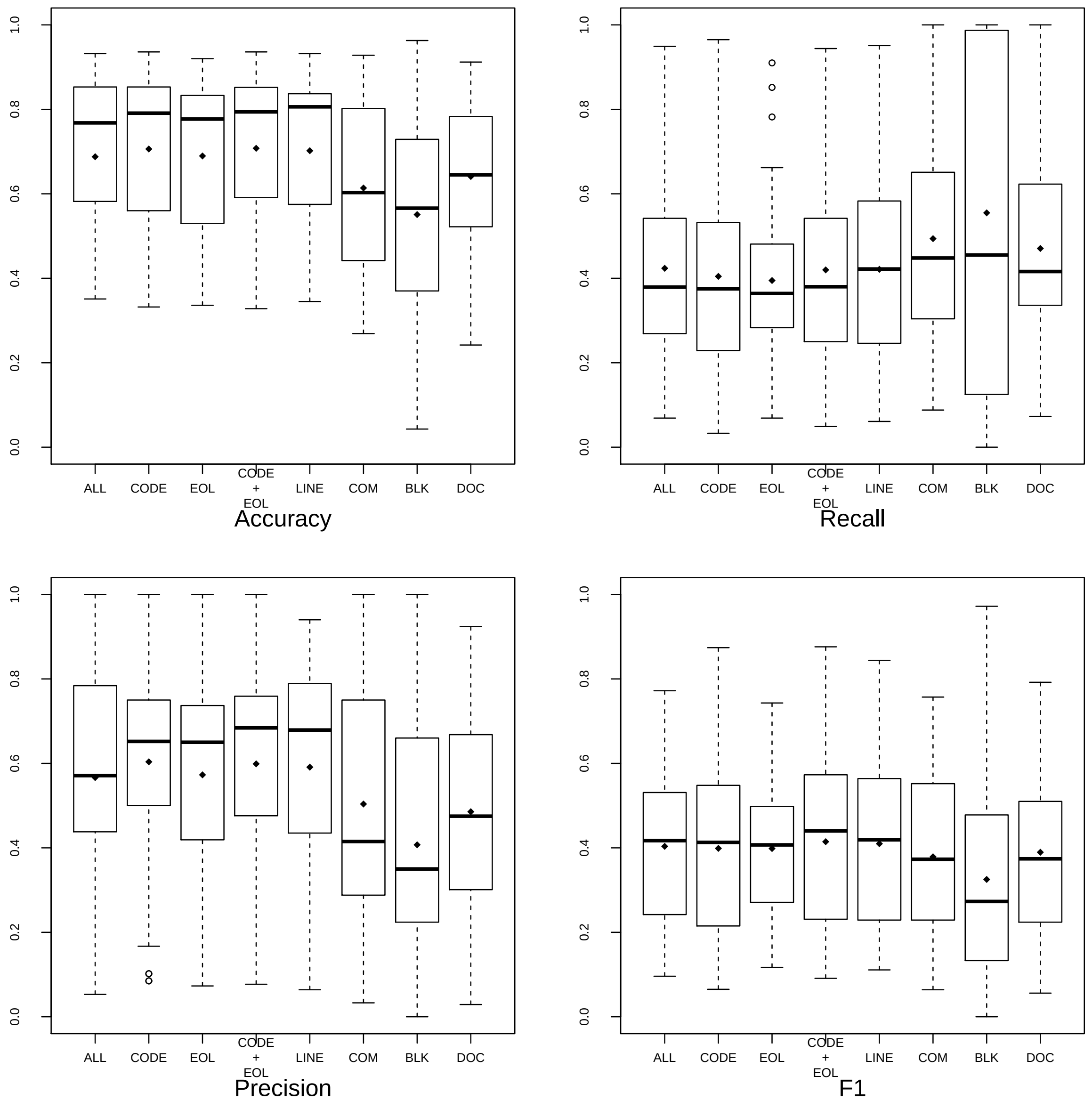

Figure 4.2 Results Using Different Kinds of Tokens

Table 4.1 Average PF_r Values in Intra-Project Prediction

	TC_{ALL}	TC_{CODE}	TC_{EOL}	$TC_{CODE+EOL}$	TC_{LINE}	TC_{COM}	TC_{BLK}	TC_{DOC}
average PF_r	0.231	0.268	0.237	0.264	0.256	0.168	0.072	0.150

ments. This table also shows that we should use TC_{ALL}, TC_{CODE}, TC_{EOL}, $TC_{CODE+EOL}$ and TC_{LINE}.

The results using each content extracted from module have trends (see Figure 4.2). From this figure, we find that if we combine token classes, the result by combining token classes inherit characteristics from each token class. For example, if we use TC_{COM} token class, the result inherit trends of TC_{EOL}, TC_{BLK} and TC_{DOC}. Next, we consider the reasons of these trends. To begin with, we can classify these trends into three groups. The first group is the better precision group. TC_{CODE}, TC_{EOL}, $TC_{CODE+EOL}$ and TC_{LINE} belong to this group. The second group is the better recall group. TC_{BLK} and TC_{DOC} belong to this group. The third group is average precision and recall group. This group is yielded by combining token classes of the first and the second group. We guess that the results of better precision group are not out of the way because these token classes do not have specific feature against token classes of better recall group. The better recall group has relativity small size of tokens and includes fixed tokens. Here, fixed tokens means that are included in most of modules. Examples of these tokens are header comments and tags. If these tokens are made up of a majority of a target module, the result was decided by these tokens. By considering from Equation (2.1), when ratio of faulty of training modules is low and fixed tokens are few in target module, the target module will be classified as fault-prone. In contrast, ratio of faulty of training modules is high, target modules will be classified non-fault-prone. From Table 3.1, there are many versions that ratio of faulty is low in our dataset. Therefore, token classes in better recall group show high recall.

4.3 Ex. 3: Combination of Older Versions

We use one version as training data to conduct fault-prone filtering in ex. 1 and ex. 2 because this is the easiest condition in intra-project prediction. However, if we can use two or more versions, what results do we obtain using these versions? In this experiment, we investigate the effects of combination of older versions.

4.3.1 Experiment Method

In ex. 3, we use some versions as training data for predicting one target version. We can select some combination about training data. However, we combine all versions that were developed before target version. In other words, we predict target version by combining all older versions. For example, when we use Ant project, there are 2 combinations about training and target version, that is, (Ant1.4+Ant1.5, Ant1.6), (Ant1.4+Ant1.5+Ant1.6, Ant1.7). Here, we don't conduct experiments that have only one training data (e.g. (Ant1.4, Ant1.5)) because these experiments are already conducted in ex. 1 or ex. 2. We make these combinations for each project and conduct experiments. Next, we describe about tokens. From the result of ex. 2, we guess $TC_{CODE+EOL}$ shows better result. Therefore, we use $TC_{CODE+EOL}$ in this experiment.

4.3.2 Results and Discussions

We show the result of ex. 2 and ex. 3 using $TC_{CODE+EOL}$ token class in Table 4.2. The numbers that are written by bold font in this table means that the result is the best in the target. From this table, we find that the results using some version as training version show better recall in many cases than using one version as training data. In contrast, precision is not better in the result by combination of older versions. Altogether, we can say that combining some projects as training data don't always improve prediction result.

According to Equation (2.1), higher ratio of faulty of training versions creates a strong tendency for target modules to classify into faulty. We can see this trend in Table 3.1 and Table 4.2. Hence, if we combine versions with high ratio of faulty module and the ones with low ratio of faulty modules, recall is averaged. Combining projects that ratio of faulty is almost same, in most case, recall is improved. We guess that these results are caused by extensive training data. If we use some versions for training data, classifier can be trained about more faults.

Table 4.2 Comparison of Results in ex. 2 and ex. 3

Training	Target	Accuracy	Recall	Precision	F_1
Ant1.4	Ant1.6	0.766	0.250	0.639	0.359
Ant1.5		0.780	0.239	**0.759**	0.364
Ant1.4+Ant1.5		**0.786**	**0.348**	0.681	**0.460**
Ant1.4	Ant1.7	0.794	0.139	0.697	0.231
Ant1.5		0.800	0.169	**0.737**	0.275
Ant1.6		**0.823**	0.380	0.692	0.490
Ant1.4+Ant1.5+Ant1.6		0.821	**0.464**	0.636	**0.537**
Eclipse2.0	Eclipse3.0	**0.853**	0.318	**0.508**	0.391
Eclipse2.1		**0.853**	0.247	0.507	0.332
Eclipse2.0+Eclipse2.1		0.845	**0.374**	0.470	**0.416**
jEdit3.2	jEdit4.1	0.820	0.532	0.712	0.609
jEdit4.0		**0.840**	0.456	**0.878**	0.600
jEdit3.2+jEdit4.0		**0.840**	**0.570**	0.763	**0.652**
jEdit3.2	jEdit4.2	0.839	0.562	0.429	0.486
jEdit4.0		**0.876**	0.562	**0.540**	0.551
jEdit4.1		0.873	0.542	0.531	0.536
jEdit3.2+jEdit4.0+jEdit4.1		0.845	**0.750**	0.456	**0.567**
jEdit3.2	jEdit4.3	0.852	**0.545**	0.082	0.143
jEdit4.0		0.887	0.455	0.093	0.154
jEdit4.1		0.887	0.364	0.077	0.127
jEdit4.2		**0.936**	0.364	**0.143**	**0.205**
jEdit3.2+jEdit4.0+jEdit4.1+jEdit4.2		0.789	**0.545**	0.058	0.104
Lucene2.0	Lucene2.4	0.564	0.365	**0.831**	0.507
Lucene2.2		**0.591**	**0.498**	0.754	**0.599**
Lucene2.0+Lucene2.2		0.567	0.389	0.806	0.525
POI1.5	POI2.5	**0.847**	**0.826**	**0.932**	**0.876**
POI2.0		0.369	0.049	0.750	0.091
POI1.5+POI2.0		0.420	0.142	0.814	0.241
POI1.5	POI3.0	**0.662**	0.651	0.785	0.712
POI2.0		0.395	0.068	0.864	0.125
POI2.5		0.651	**0.683**	0.750	**0.715**
POI1.5+POI2.0+POI2.5		0.546	0.331	**0.894**	0.483
Velocity1.4	Velocity1.6	0.328	0.910	0.326	0.480
Velocity1.5		**0.629**	0.872	**0.476**	**0.615**
Velocity1.4+Velocity1.5		0.376	**0.949**	0.347	0.509
Xalan2.4	Xalan2.6	0.642	0.299	**0.831**	0.440
Xalan2.5		0.663	**0.482**	0.707	**0.573**
Xalan2.4+Xalan2.5		**0.667**	0.409	0.778	0.536
Xerces1.2	Xerces1.4	**0.412**	0.081	**1.000**	0.150
Xerces1.3		0.399	0.067	0.933	0.124
Xerces1.2+Xerces1.3		**0.412**	**0.090**	0.905	**0.165**

5. Cross Project Prediction

In this section, we conduct the cross project predictions using fault-prone filtering. In cross project prediction, we use one or more projects as training data to predict other projects. We need at least one version of target project in intra-project prediction. Hence, we can only apply intra-project prediction after the second version. However, we need no older versions of target project in cross project prediction. Therefore, if we have only one version of target project, we can apply the project to the cross project prediction.

5.1 Ex. 4: Prediction Using Single Project

We consider that cross project prediction is more difficult task than intra-project prediction. In this experiment, we conduct simple condition cross project prediction, that is, using single project as training data, in order to investigate whether fault-prone filtering work in cross project prediction.

5.1.1 Experiment Method

In ex. 4, we conduct the cross project prediction using single project as training data. In particular, we use all of versions as training data in a project, and predict using the other projects. For example, when we use Ant project as training data, other projects, Eclipse, jEdit, Lucene, POI, Velocity, Xalan and Xerces, are target data. That is, training and target tuples are (All of Ant versions, Eclipse2.0), ..., (All of Ant versions, Eclipse3.0), (All of Ant versions, jEdit3.2), ..., (All of Ant versions, Xerces1.6). In this experiment, we use TC_{ALL} token class extracted from modules.

5.1.2 Results and Discussions

We show the results of prediction in Table 5.1. In this table, if the value of PF_r 0 or fewer, that is, if the values of precision are worse than the value of prediction by random prediction, we write '×'. '-' means that we do not conduct the combination of prediction because training and target data are same in these combinations. We find that most

Table 5.1 Combinations of Trainings and Targets that Predict well

target \ training	Ant	Eclipse	jEdit	Lucene	POI	Velocity	Xalan	Xerces
Ant1.4	-			×	×	×	×	
Ant1.5	-				×	×		
Ant1.6	-				×			
Ant1.7	-				×			
Eclipse2.0		-				×		
Eclipse2.1		-		×		×		
Eclipse3.0		-				×		
jEdit3.2			-	×		×		
jEdit4.0			-			×		
jEdit4.1			-					
jEdit4.2			-			×		
jEdit4.3			-			×		
Lucene2.0				-		×		
Lucene2.2				-				
Lucene2.4				-				
POI1.5				×	-	×		
POI2.0					-	×		
POI2.5				×	-	×		
POI3.0					-	×		
Velocity1.4			×			-		×
Velocity1.5						-		
Velocity1.6						-		
Xalan2.4						×	-	
Xalan2.5					×		-	
Xalan2.6					×		-	
Xerces1.2					×	×		-
Xerces1.3					×	×		-
Xerces1.4					×	×		-

projects work well as training data. However, some projects can't use for predicting other projects from this table. Therefore, if we use single project as training data, the result is based on the project.

Table 5.2 shows the average values of results in each project. From this results, Eclipse is the best project for predicting other projects. We guess that this results are caused the number of modules in Eclipse project. Eclipse is more than ten times larger than other projects in the viewpoint of the number of modules. We guess that Eclipse can adjust and predict various projects because of enough number of modules.

5.2 Ex. 5: Prediction Using All Other Project

In this experiment, we investigate three things in cross project prediction. 1) do the results of prediction using all projects show better than the results using single project, 2) trends of results using some kinds of token classes, 3) comparison of the results of intra-project prediction and cross project prediction.

5.2.1 Experiment Method

In this experiment, we predict a target project using all other training projects. That is, when we select a version as target data, we use all versions for training except versions that belong to target project. For example, when we use Ant as target project, training and target tuples are (All versions except Ant's, Ant1.4), (All versions except Ant's, Ant1.5), (All versions except Ant's, Ant1.6), (All versions except Ant's, Ant1.7). We predict all versions based on above rule. In addition, we conduct theses prediction using TC_{ALL}, TC_{CODE}, TC_{COM}, TC_{EOL}, TC_{BLK}, TC_{DOC}, $TC_{CODE+EOL}$ and TC_{LINE} token classe classes.

5.2.2 Results and Discussions

Table 5.3 shows the comparison of the results using single project (Eclipse) and all projects except target project (this experiment). From this table, we find that the results are improved in the values of recall and F_1. Therefore, we concluded that combining all

Table 5.2 Average Results Using Single Project in Cross Project Predicition

	Accuracy	Recall	Precision	F_1
Ant	0.651	0.323	0.598	0.356
Eclipse	0.621	**0.529**	0.555	**0.447**
jEdit	0.636	0.201	**0.647**	0.258
Lucene	0.628	0.233	0.425	0.231
POI	**0.683**	0.399	0.404	0.307
Velocity	0.619	0.090	0.242	0.055
Xalan	0.671	0.251	0.596	0.305
Xerces	0.636	0.307	0.615	0.285

Table 5.3 Average Results Using Single Project and All Projects

	Accuracy	Recall	Precision	F_1
Eclipse	0.621	0.529	**0.555**	0.447
All Projects	**0.625**	**0.595**	0.504	**0.471**

projects improve the results.

We show the result of experiment in Figure 5.1. From this figure, we find that trends of recall and precision in each token class are the same in intra-project prediction. From the viewpoint of F_1 value, TC_{ALL}, TC_{COM}, TC_{BLK} and TC_{DOC} show better results in cross project prediction. However, as shown in Table 5.4 TC_{COM}, TC_{BLK} and TC_{DOC} show low average PF_r. Therefore, We guess that predictions using TC_{COM}, TC_{BLK} and TC_{DOC} are impractical. By comparing TC_{ALL}, TC_{CODE}, TC_{EOL}, $TC_{CODE+EOL}$ and TC_{LINE}, obviously, TC_{ALL} shows better F_1 value than TC_{CODE}, TC_{EOL}, $TC_{CODE+EOL}$ and TC_{LINE} in cross project prediction. As a result, using TC_{ALL} token class is the best way of prediction in the cross project prediction. However, if we need better precision, we can use TC_{CODE}, $TC_{CODE+EOL}$ or TC_{LINE}.

Table 5.5 shows comparison of intra-project and cross project prediction results. Results of intra-project prediction show better precision than results of cross project prediction. In contrast, the results of cross project prediction show better recall. Such trend is also shown in the reference [2].

Table 5.6 shows the number of kinds of tokens extracted from TC_{ALL}. The row of "All Projects" in this table related to the cross project prediction. Therefore, tokens are used for predicting faults in two or more projects. Other rows like "Ant Project" are related to the intra-project prediction. Therefore, tokens are used for predicting faults in two or more versions. From this table, we find that kinds of tokens are used in the cross project prediction is low (see row of "ALL Projects"). However, we can predict faults by these tokens (see Figure 5.1 and Table 5.5). Therefore, we can consider that tokens used in several projects are important for predicting faults. In other words, these common tokens between several projects characterize either faulty and non-faulty modules beyond projects. From this result, tokens (structures) extracted from TC_{ALL}, TC_{CODE}, TC_{EOL}, $TC_{CODE+EOL}$ and TC_{LINE} are potentially related to generic fault-proneness.

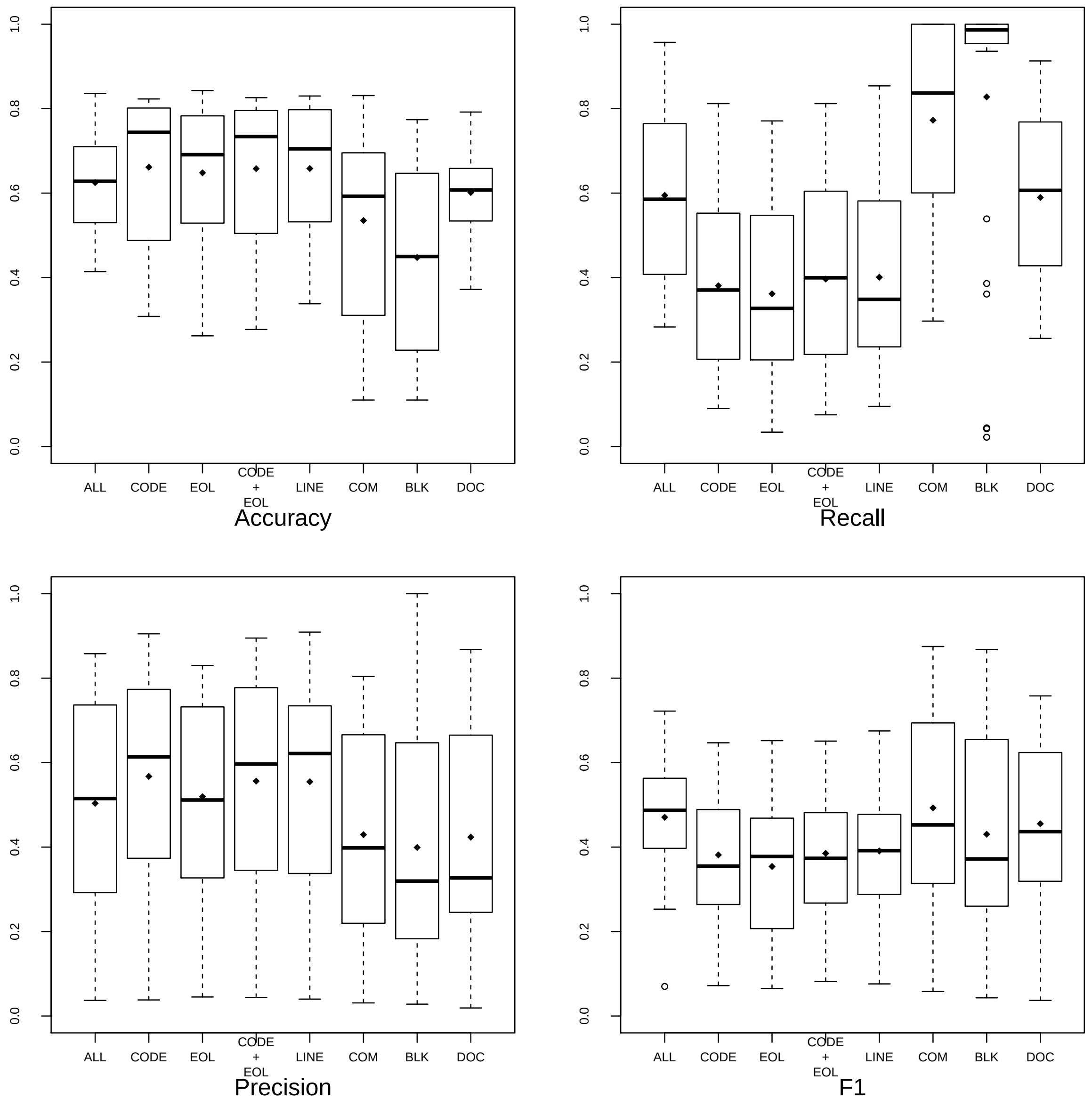

Figure 5.1 Comparison of Results Using Different Tokens in Cross Project Prediction

Table 5.4 Average PF_r Values in Cross Project Prediction

	TC_{ALL}	TC_{CODE}	TC_{EOL}	$TC_{CODE+EOL}$	TC_{LINE}	TC_{COM}	TC_{BLK}	TC_{DOC}
average PF_r	0.152	0.216	0.168	0.205	0.204	0.078	0.048	0.072

Table 5.5 Comparison of Intra and Cross Project Predictions

Training	Target	Accuracy	Recall	Precision	F_1
Ant1.4	Ant1.5	**0.877**	0.438	**0.438**	**0.438**
All projects except Ant		0.414	**0.906**	0.147	0.253
Ant1.4	Ant1.6	**0.760**	0.239	0.611	0.344
Ant1.5		0.774	0.228	**0.724**	0.347
All projects except Ant		0.571	**0.957**	0.376	**0.540**
Ant1.4	Ant1.7	0.779	0.277	0.511	0.359
Ant1.5		**0.799**	0.410	**0.571**	0.477
Ant1.6		0.781	0.602	0.510	**0.552**
All projects except Ant		0.667	**0.777**	0.381	0.511
Eclipse2.0	Eclipse2.1	**0.857**	0.362	**0.345**	**0.353**
All projects except Eclipse		0.829	**0.411**	0.293	0.342
Eclipse2.0	Eclipse3.0	**0.853**	0.356	0.503	**0.417**
Eclipse2.1		**0.853**	0.269	**0.504**	0.351
All projects except Eclipse		0.826	**0.411**	0.411	0.411
jEdit3.2	jEdit4.0	**0.788**	**0.640**	**0.578**	**0.608**
All projects except jEdit		0.751	0.627	0.511	0.563
jEdit3.2	jEdit4.1	0.793	0.671	0.596	0.631
jEdit4.0		**0.853**	0.506	**0.889**	**0.645**
All projects except jEdit		0.750	**0.684**	0.519	0.590
jEdit3.2	jEdit4.2	0.806	0.750	0.387	0.511
jEdit4.0		0.856	0.542	0.473	0.505
jEdit4.1		**0.870**	0.542	**0.520**	**0.531**
All projects except jEdit		0.707	**0.812**	0.291	0.429
jEdit3.2	jEdit4.3	0.768	0.545	0.053	0.096
jEdit4.0		0.873	0.455	0.082	0.139
jEdit4.1		0.883	0.364	0.074	0.123
jEdit4.2		**0.932**	0.364	**0.133**	**0.195**
All projects except jEdit		0.620	**0.636**	0.037	0.070
Lucene2.0	Lucene2.2	0.590	0.420	**0.822**	0.556
All projects except Lucene		**0.607**	**0.462**	0.815	**0.589**
Lucene2.0	Lucene2.4	0.582	0.379	**0.865**	0.527
Lucene2.1		0.600	**0.483**	0.784	0.598
All projects except Lucene		**0.630**	0.478	0.858	**0.614**
POI1.5	POI2.0	0.424	0.514	0.106	0.176
All projects except POI		**0.469**	**0.784**	**0.157**	**0.261**
POI1.5	POI2.5	**0.567**	**0.336**	**1.000**	**0.503**
POI2.0		0.375	0.069	0.708	0.125
All projects except POI		0.472	0.283	0.753	0.412
POI1.5	POI3.0	0.521	0.263	**0.961**	0.413
POI2.0		0.404	0.096	0.794	0.171
POI2.5		**0.644**	**0.665**	0.751	**0.706**
All projects except POI		0.559	0.391	0.833	0.533
Velocity1.4	Velocity1.5	**0.636**	**0.930**	0.660	**0.772**
All projects except Velocity		0.542	0.401	**0.814**	0.538
Velocity1.4	Velocity1.6	0.362	**0.949**	0.343	0.503
Velocity1.5		0.620	0.872	0.469	**0.610**
All projects except Velocity		**0.686**	0.500	**0.542**	0.520
Xalan2.4	Xalan2.5	0.537	0.194	**0.647**	0.298
All projects except Xalan		**0.612**	**0.749**	0.593	**0.662**
Xalan2.4	Xalan2.6	0.667	0.365	0.833	0.508
Xalan2.5		0.591	0.139	**0.934**	0.242
All projects except Xalan		**0.728**	**0.752**	0.694	**0.722**
Xerces1.2	Xerces1.3	0.819	0.290	**0.377**	0.328
All projects except Xerces		0.712	**0.826**	0.326	**0.467**
Xerces1.2	Xerces1.4	0.351	0.076	0.457	0.131
Xerces1.3		0.405	0.076	**0.941**	0.141
All projects except Xerces		**0.488**	**0.324**	0.723	**0.447**

Table 5.6 Kinds of Tokens Extracted from TC_{ALL}

extracted from	in one proejct	in two or more projects	sum	used in prediction
All Projects	401,903	28,227	430,130	6.6 %
extracted from	in one version	in two or more versions	sum	used in prediction
Ant Project	14,447	18,796	33,243	56.5 %
Eclipse Project	114,922	201,746	316,668	63.7 %
jEdit Project	9,229	24,796	34,025	72.9 %
Lucene Project	5,385	9,688	15,073	64.3 %
POI Project	2,778	17,287	20,065	86.2 %
Velocity Project	2,517	8,777	11,294	77.7 %
Xalan Project	10,430	34,282	44,712	76.7 %
Xerces Project	2,040	19,110	211,50	90.4 %

5.3 Ex. 6: Prediction Using All Other Project and Older Versions

We find that the results of intra-project prediction show better precision and the results of cross project prediction show better recall. Therefore, in this experiment, we investigate whether the results are improved by combining intra-project prediction and cross project prediction.

5.3.1 Experiment Method

In this experiment, we predict a target project using all other training projects and older versions. In brief, we combine ex. 3 with ex. 5. Therefore, when we predict a target version, we use all versions in other project and its older versions. For example, if we use Ant1.7 as target version, training versions are all versions of projects except Ant and Ant1.4+Ant1.5+Ant1.6. We use TC_{ALL} token class, because TC_{ALL} token class show the best F_1 value in ex. 5.

5.3.2 Results and Discussions

We show the comparison of ex. 5 and ex. 6 in Figure 5.2. We can see that results are almost the same from this figure. Therefore, we conclude that using older versions do not improve results in cross project prediction.

Table 5.7 shows comparison of ex. 3, ex. 5, and ex. 6. "ex. 3+ex. 5" in the table means that we combine training data written after "ex. 3" and "ex. 5". That is, the lines of "ex. 3+ex. 5" show the results of this experiment. In order to compare the result using same type of tokens, we re-conduct ex. 3 using TC_{ALL} token class. We can say that this experiment is combination of intra-project and cross project prediction. Precision is better in order of ex. 3, ex. 6 and ex. 5. Recall is better in order of ex. 5 ,ex. 6 and ex. 3. Therefore, if we need better precision, we should conduct intra-project prediction and we need better recall, we should conduct cross project prediction.

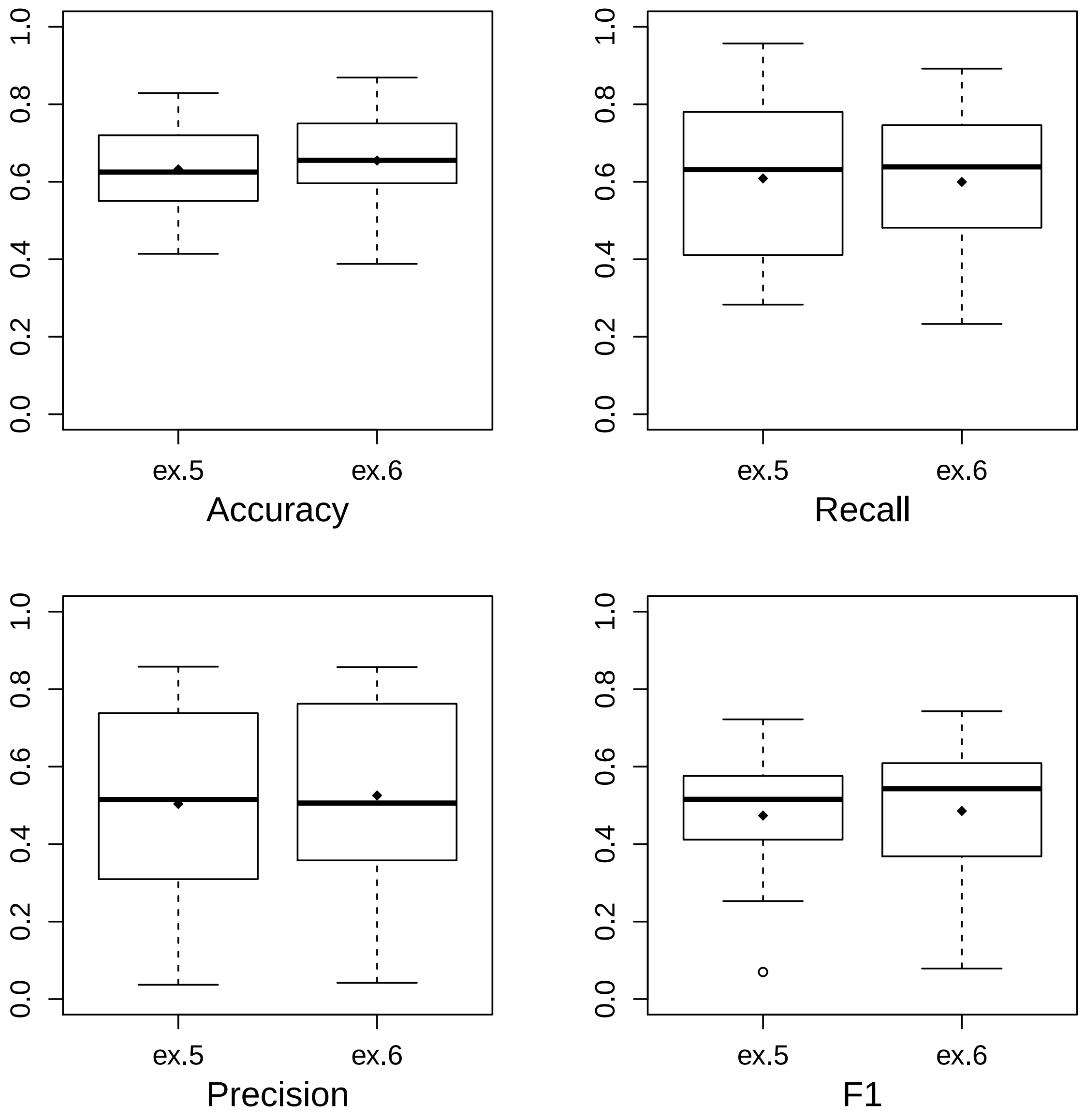

Figure 5.2 Comparison of ex. 5 and ex. 6

Table 5.7 Comparison of ex. 3, ex. 5 and ex. 6

Training	Target	Accuracy	Recall	Precision	F_1
ex. 3: Ant1.4+Ant1.5	Ant1.6	**0.791**	0.402	**0.673**	0.503
ex. 5: All projects except Ant		0.571	**0.957**	0.376	0.540
ex. 6: ex. 3+ex. 5		0.703	0.880	0.466	**0.609**
ex. 3: Ant1.4+Ant1.5+Ant1.6	Ant1.7	**0.725**	0.693	**0.429**	**0.530**
ex. 5: All projects except Ant		0.667	**0.777**	0.381	0.511
ex. 6: ex. 3+ex. 5		0.715	0.729	0.422	0.534
ex. 3: Eclipse2.0+Eclipse2.1	Eclipse3.0	0.841	0.381	0.457	**0.416**
ex. 5: All projects except Eclipse		0.826	**0.411**	0.411	0.411
ex. 6: ex. 3+ex. 5		**0.852**	0.304	**0.502**	0.379
ex. 3: jEdit3.2+jEdit4.0	jEdit4.1	**0.813**	0.684	**0.635**	**0.659**
ex. 5: All projects except jEdit		0.750	0.684	0.519	0.590
ex. 6: ex. 3+ex. 5		0.770	**0.747**	0.546	0.631
ex. 3: jEdit3.2+jEdit4.0+jEdit4.1	jEdit4.2	**0.814**	0.792	**0.404**	**0.535**
ex. 5: All projects except jEdit		0.707	0.812	0.291	0.429
ex. 6: ex. 3+ex. 5		0.735	**0.854**	0.320	0.466
ex. 3: jEdit3.2+jEdit4.0+jEdit4.1+jEdit4.2	jEdit4.3	**0.764**	0.545	**0.052**	**0.094**
ex. 5: All projects except jEdit		0.620	**0.636**	0.037	0.070
ex. 6: ex. 3+ex. 5		0.663	**0.636**	0.042	0.079
ex. 3: Lucene2.0+Lucene2.2	Lucene2.4	0.591	0.433	0.815	0.566
ex. 5: All projects except Lucene		0.630	0.478	**0.858**	0.614
ex. 6: ex. 3+ex. 5		**0.648**	**0.537**	0.832	**0.653**
ex. 3: POI1.5+POI2.0	POI2.5	0.372	0.049	**0.800**	0.092
ex. 5: All projects except POI		**0.472**	**0.283**	0.753	**0.412**
ex. 6: ex. 3+ex. 5		0.462	0.255	0.759	0.382
ex. 3: POI1.5+POI2.0+POI2.5	POI3.0	**0.616**	**0.936**	0.637	**0.758**
ex. 5: All projects except POI		0.559	0.391	0.833	0.533
ex. 6: ex. 3+ex. 5		0.610	0.470	**0.857**	0.607
ex. 3: Velocity1.4+Velocity1.5	Velocity1.6	0.437	**0.936**	0.371	0.531
ex. 5: All projects except Velocity		**0.686**	0.500	**0.542**	0.520
ex. 6: ex. 3+ex. 5		0.646	0.641	0.485	**0.552**
ex. 3: Xalan2.4+Xalan2.5	Xalan2.6	0.649	0.275	**0.926**	0.424
ex. 5: All projects except Xalan		0.728	**0.752**	0.694	0.722
ex. 6: ex. 3+ex. 5		**0.758**	0.745	0.741	**0.743**
ex. 3: Xerces1.2+Xerces1.3	Xerces1.4	0.409	0.095	**0.833**	0.171
ex. 5: All projects except Xerces		**0.488**	**0.324**	0.723	**0.447**
ex. 6: ex. 3+ex. 5		0.463	0.233	0.766	0.358

6. Related Works

In this section, we describe related works that conducted cross project prediction.

Turhan et al. [2] conducted cross-company and within-company defect predictions using 10 project data from 8 different companies. They showed that cross-company predictions increase probability of detection (pd) and decrease probability of false (pf). In order to improve pf value, they proposed an approach that selects training data using k-nearest neighbor. However, the results of within-company is better than the results of cross-company with k-nearest neighbor. Therefore, they concluded that if there are no within-company data, we can use cross-company prediction with k-nearest neighbor and start to collect within-company data. After a few hundred examples are available, it can be switched to use within-company data to predict faults.

Zimmermann et al. [1] conducted 622 cross project prediction and showed that only 21 prediction (3.4%) satisfy their criteria (accuracy, recall and precision are greater than 0.75). They investigate relation between factors for predicting and the results and conclude that projects in the same domain do not work to build accurate prediction models.

He et al. [5] showed that the results of cross project prediction using suitable training data are better than the results of intra-project prediction. They proposed an approach that selects training data properly. They showed that their approach can find proper training dataset for 24 of 34 versions.

Watanabe et al. [3] conducted inter project prediction using same domain (text editor) but different languages (Java and C++) projects. They proposed an approach called metrics compensation that normalizes metrics between different datasets based on average value of each metric. They showed that the results of inter project prediction are improved by their approach. In study [4], Watanabe et al. conducted experiments that use several projects as training data and several machine learning algorithms to predict faults. They showed that the results of majority vote of some machine learning algorithm is better than the results of each machine learning algorithm.

7. Threats to Validity

In this research, we use 8 projects, Ant, Eclipse, jEdit, Lucene, POI, Velocity, Xalan and Xerces, including 28 versions. We guess that the number of projects and versions is sufficiently large. However, all of these projects are open source software. Therefore, if we use projects that were developed by company, results might show other trends. In addition, 7 of 8 projects that we select to use in this research are developed by Apache community. We can think such conditions are most project developed by one company. Hence, these projects might similar in some way. We have possibility that these affinity affect to our results.

We use the data of faulty modules in PROMISE database. These data that we use are not collected by developer. These are collected by third person using tool. Therefore, These data about faults are not perfectly correct because there has not yet been any complete way to extract bug information from bug tracking system and version control system. Accordingly, our results include a certain level of error.

8. Conclusions

In this research, first of all, we conduct comparison of experiment using the multi-variate Bernoulli model and the multinomial model. From the result, we conclude that the multi-variate Bernoulli model is better model in fault-prone filtering. From the results of intra-project prediction and cross project prediction using some kinds of tokens. TC_{ALL}, TC_{CODE}, TC_{EOL}, $TC_{CODE+EOL}$ and TC_{LINE} can use in fault-prone filtering. Especially, TC_{ALL} show the best result in cross project prediction. From the results of prediction using TC_{LINE}, we can say that structures of modules related to faulty. From the results of cross prediction, we guess that there are tokens and structures that are related to fault-proneness potentially.

Acknowledgements

I received a lot of help from many individuals during this work. Especially, I would like to thank Associate Professor Osamu Mizuno. I would not have completed this work without his continuous support and encouragement. I am also very grateful to the members of the software engineering laboratory in Kyoto Institute of Technology. I also thank to my friends and my family.

References

[1] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, “Cross-project defect prediction: a large scale experiment on data vs. domain vs. process,” Proc. of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pp.91–100, Amsterdam, The Netherlands, 2009.

[2] B. Turhan, T. Menzies, A.B. Bener, and J. Di Stefano, “On the relative value of cross-company and within-company data for defect prediction,” Empirical Softw. Engg., vol.14, pp.540–578, Oct. 2009.

[3] S. Watanabe, H. Kaiya, and K. Kaijiri, “Adapting a fault prediction model to allow inter languagereuse,” Proc. of the 4th international workshop on Predictor models in software engineering, pp.19–24, Leipzig, Germany, 2008.

[4] S. Watanabe, H. Kaiya, and K. Kaijiri, “Reuse of the estimation model of error-prone module,” Technical report of IEICE. KBSE, vol.109, no.41, pp.55–60, May 2009. (In Japanese).

[5] Z. He, F. Shu, Y. Yang, M. Li, and Q. Wang, “An investigation on the feasibility of cross-project defect prediction,” Automated Software Engineering, pp.1–33, 2011.

[6] M. Jureczko and L. Madeyski, “Towards identifying software project clusters with regard to defect prediction,” Proc. of the 6th International Conference on Predictive Models in Software Engineering, pp.1–10, Timişoara, Romania, 2010.

[7] S.R. Chidamber and C.F. Kemerer, “A metrics suite for object oriented design,” IEEE Trans. on Software Engineering, vol.20, no.6, pp.476–493, 1994.

[8] N. Nagappan and T. Ball, “Use of relative code churn measures to predict system defect density,” Proc. of 27th International Conference on Software Engineering, pp.284–292, St. Louis, MO, USA, 2005.

[9] O. Mizuno and T. Kikuno, “Training on errors experiment to detect fault-prone software modules by spam filter,” Proc. of 6th joint meeting of the European software

engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering, pp.405–414, Dubrovnik, Croatia, 2007.

[10] O. Mizuno and Y. Hirata, "Fault-prone module prediction using contents of comment lines," Proc. of International Workshop on Empirical Software Engineering in Practice 2010 (IWESEP2010), pp.39–44, Dec. 2010. NAIST, Nara, Japan.

[11] Sun Microsystems, "5-comments". http://java.sun.com/docs/codeconv/html/CodeConventions.doc4.html, Accessed Feb 3, 2012.

[12] K. Sawa, Y. Higo, and S. Kusumoto, "Proposal and evaluation of an approach to find bugs using difference information of code clone detection tools," Technical report of IEICE. SS, vol.108, no.173, pp.67–72, Jul. 2008. (In Japanese).

[13] S. Morisaki, N. Yoshida, Y. Higo, S. Kusumoto, K. Inoue, K. Sasaki, K. Murakami, and K. Matsui, "Empirical evaluation of similar defect detection by code clone search," The IEICE Trans. Inf. & Syst. (Japanese edition), vol.91, no.10, pp.2466–2477, Oct. 2008. (In Japanese).

[14] Y. Hirata and O. Mizuno, "Investigating effects of tokens on detecting fault-prone modules by text filtering," Proc. of 22nd International Symposium on Software Reliability Engineering (ISSRE2011), Supplemental proceedings, no.3-2, Nov. 2011. Hiroshima, Japan.

[15] H. Takamura, Introduction to Machine Learning for Natural Language Processing, CORONA PUBLISHING CO., LTD., 2010. (In Japanese).

[16] A. McCallum and K. Nigam, "A comparison of event models for naive bayes text classification," Dimension Contemporary German Arts And Letters, vol.752, pp.41–48, 1998.

[17] G. Boetticher, T. Menzies, and T. Ostrand "Promise repository of empirical software engineering data repository,", http://promisedata.org/, West Virginia University, Department of Computer Science, 2007.

[18] M. Jureczko and D. Spinellis, "Using object-oriented design metrics to predict soft-

ware defects," Models and Methodology of System Dependability. Proc. of RELCOMEX 2010: Fifth International Conference on Dependability of Computer Systems DepCoS, pp.69–81, Wrocław, Poland, 2010.

[19] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," Proc. of the Third International Workshop on Predictor Models in Software Engineering, p.9, May 2007.

[20] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," IEEE Trans. on Software Engineering, vol.33, no.1, pp.2–13, Jan. 2007.

[21] T. Menzies, A. Dekhtyar, J. Distefano, and J. Greenwald, "Problems with precision: A response to "comments on 'data mining static code attributes to learn defect predictors'"," IEEE Trans. on Software Engineering, vol.33, pp.637–640, 2007.

[22] Y. Hirata and O. Mizuno, "Do comments explain codes adequately? – investigation by text filtering –," Proc. of 8th Working Conference on Mining Software Repositories (MSR2011), pp.242–245, May 2011. Honolulu, HI, USA.