2016

Syntax errors identification from compiler error messages using ML techniques

Shubham K. Agrawal
Iowa State University

Syntax errors identification from compiler error messages using ML techniques

by

Shubham K Agrawal

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Jin Tian, Co-Major Professor
Wei Le, Co-Major Professor
Samik Basu

Iowa State University

Ames, Iowa

2016

TABLE OF CONTENTS

LIST OF FIGURES

Page

LIST OF TABLES

Page

NOMENCLATURE

SVM	Support Vector Machine
LDA	Latent Dirichlet Allocation
MLC	Multi Label Classification
LR	Label Ranking
v_u	Variable undeclared
c_c	Close curly
o_c	Open curly
c_p	Close parenthesis

ACKNOWLEDGMENTS

I would like to thank my major professors Dr. Jin Tian and Dr. Wei Le for their guidance and support throughout the course of this research. Their patience, valuable input and inspiring advice led me through the years of my Master's research. I would like to sincerely express my respect and heartfelt gratitude to them.

I would also like to thank my POS committee member Dr. Samik Basu for being very approachable and helpful throughout the course of this research and my Master's.

In addition, I would like to thank my friends, colleagues, the department faculty and staff for making my time at Iowa State University a wonderful experience. Finally, I would like to thank my brother and my family for their encouragement, advice, respect, and love.

ABSTRACT

Compiler error messages facilitate software development and debugging by providing cause and location of the error but due to various compiler bugs and inconsistencies it often fails its purpose and negatively affect performance of both novice and experienced programmers. An errant semicolon or brace can result in many errors reported throughout the program. This study tries to statistically analyze open source code base to predict real errors from different type of compiler error messages. It also tries to auto-fix these errors.

At the high level, this study handles two cases (1) when one error is present in code, (2) when two different errors are present in the code. We start with collecting different type of random error messages for both the cases by random error generation in C projects. We developed different models using document clustering, probabilistic topic modeling and multi-label classification algorithms for training and predicting real errors using collected error messages for both the cases.

Our empirical evaluation on open-source projects has shown that our model correctly predicts the real error in almost 95% cases, when only one error exists in program. In case of two errors, model correctly predicts at least one error in almost 91% cases and both the errors in almost 39% cases.

CHAPTER 1. INTRODUCTION

Compiler error message is the error message that compiler generates when code does not conform to the syntax of the programming language. It describes the cause of error and contains location of problematic part of code. Ideally these error messages should be enough to understand the cause and fix the code, but in many cases it fails to facilitate that. An errant semicolon or brace can result in many errors reported throughout the program. Following figure 1.1 shows four compiler error messages corresponding to one missing semicolon at different location. We can see that how inconsistent compiler error messages may get depending on the error location.

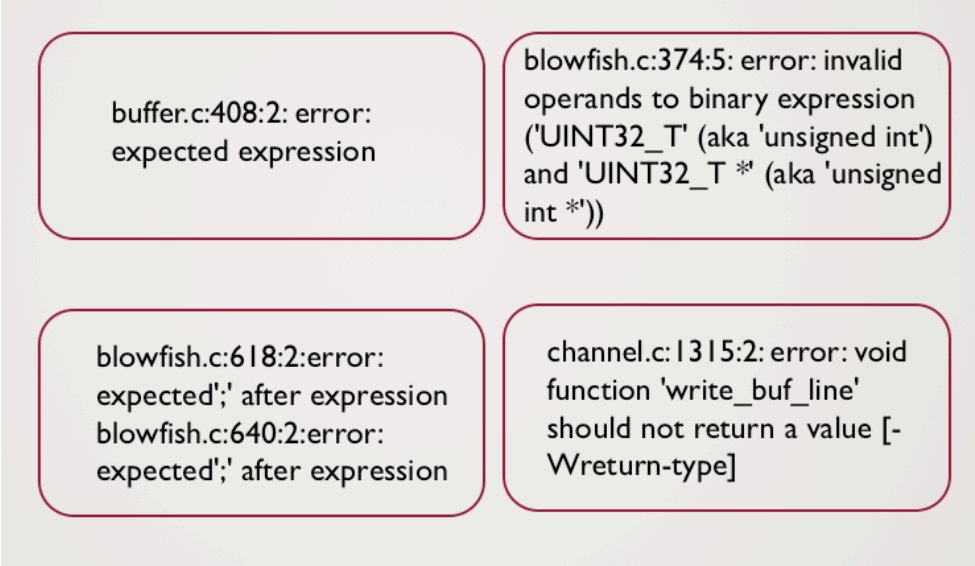

Figure 1.1. Compiler error messages for one missing semicolon error

This mainly happen because of bugs in compiler and its inconsistency in error reporting, this negatively affect the compiler's usability along with developer's productivity.

Syntax errors substantially affect new programmers learning rate [7]. Experienced programmers also get frustrated by compiler errors and waste their efforts to uselessly commenting out code to find out the cause of error [11]. Due to inconsistency in compiler error messages, it will be really helpful if there exist a model which can predict the correct causes from different compiler error messages and can be used to auto-fix these. If we can make such method then this will improve developer's productivity and will facilitate efficient programming for both novice and experienced programmers. High volume of publicly available open source code, advancement in the field of machine learning, data visualization and high performance computing can be used to improve this compiler error reporting. Our work primarily focuses on predicting real errors from different compiler error messages using statistical analysis and machine learning algorithms. Using open source projects, we can generate huge number of training data in the form of compiler error messages which can be used to train, improve and test our models.

There are some main challenges in predicting correct errors from compiler error messages. In some cases, compiler error in one line can get reported to different location making it hard to interpret the real cause. In some cases, compiler error message does not contain information of the fix for the error so it gets very hard to fix the program by just using the error messages e.g. Error message like "missing expression" does not give enough information of the real cause. In case of two or more errors it gets even more difficult to find out the original errors separately as generated compiler error message can be combination of error messages corresponding to individual errors when only one present at a time.

We propose an approach to apply machine learning techniques to improve the real cause detection from compiler error messages. We build our machine learning models, train

it with randomly generated compiler error messages from freely available open source code and then use it to predict error messages. This model can be used to find the real compiler errors given the compiler error messages. Furthermore, it can also auto-fix some of the errors. This will definitely help developer community to code efficiently and debug their programs faster.

1.1. Contribution

Our main contribution for this work can be divided into two parts. In first part we worked for the case of one error at a time, in this we made four main contributions. Firstly, we developed a tool to generate all random compiler error messages and classify them into one of the two types, type A and type B. Secondly, we performed document clustering on all of the type B training data from previous step. Thirdly, we used different machine learning algorithms to train our model with clusters created in previous step and perform classification for new test data. In fourth step we tried to auto-fix some of the above errors.

In next part we worked for the case of two errors at a time and made following main contributions. Firstly, we developed a tool to generate all random compiler errors with two random errors at a time. Secondly, we applied probabilistic topic modeling to find the most frequent words for each type of errors. In third step, we implemented slightly modified version of Latent Dirichlet Allocation(LDA) algorithm for probabilistic topic modeling using words distribution from previous step. In last step we tried finding two errors from test data. For second part we also worked with multi-label classification to improve the performance. In this we modified our error documents to ARFF and XML form and trained our MLC model to predict two major errors in new error documents. To summarize, our work is a

maiden attempt towards applying statistical analysis and ML techniques for improving compiler error detection and correction.

1.2. Outline of thesis

The reminder of the thesis is organized as follows. In chapter 2 we talk about related work in compiler error analysis and application of statistical methods. Chapter 3 talkss about some of the ML techniques that we used in this work. In chapter 4 we discuss our approach towards solving the problem. Chapter 5 illustrates the design of experiments and results. Finally, chapter 6 summarizes our contributions and future work.

CHAPTER 2. RELATED WORK

There has been numerous work towards studying behavior of novice and experienced programmers to find out the importance and frequency of syntax errors in programs [5, 6, 7, 8]. The authors of [5] studied students' behavior of repeated editing and compiling code. This cycle often represents how students try to fix the program syntactically as opposed to semantically. As per the studies in [5, 6] novice programmers make almost 10%-20% missing semi-colons and misplaced braces errors. Undefined variable is also one of the most occurring type of error, causing almost 24% errors in the study [9]. The authors of [10] studied correlation between syntax errors and student's grades. They studied almost 120 students and found that lower the incidence of syntax error corresponds to higher grades. So syntax errors can negatively affect student's performance. The authors of [11] studied how programmers approach towards fixing syntax errors. They found that experienced programmers rely on some strategies to fix syntax errors, in case when these strategies fail these programmers also makes erratic changes in the program to fix the programs which is similar to novice programmers. These all studies show that syntax errors significantly affect performance of both novice and experienced programmers and waste lot of efforts and time.

Compiler warning and error messages indicates likely programming mistakes that developer makes. It contains cause of the error message and the location of it in the program. E.g. "adoc.c:333:14: error: expected ';' at end of declaration". Although these compiler diagnostics are widely used, it may contain bugs. These bug can negatively affect the debugging efforts. Study in [12] tries to test compilers' diagnostic support. This work firstly generates random errors into program leading to different compiler warnings. Secondly for

different compliers it parses generated warnings and align them. Lastly it checks the inconsistencies between error reporting for different compliers.

N-gram language models have been used to improve code auto completion performance [2, 3] and find syntax errors [4]. The authors of [4] try to improve compiler syntax error location reporting and find the actual source of syntax errors by relying on consistency of software, they have assumed that valid source code is very natural, repetitive and unsurprising. So this was exploited by N-gram language model of lexed source code tokens. Here authors trained a model with compilable source code token sequences and used this model to evaluate new code and see how frequently those sequences appear. Source code which does not compile should be surprising for the model. They have used existing software as a corpus of compilable and working software. This work tries to use the consistency of programs but does not evaluate how error messages are related and can be exploited to improve the error detection. Although it is possible to have some consistency between programs for a particular projects and its versions but it is less common to have such inconsistency between different projects by different users.

The authors of [1] surveyed and found that there exists a huge amount of freely available high quality programs in code repositories such as Github. Different statistical reasoning can be applied to study different properties of program. These program properties can be either sematic or syntactic. In this work they transform the training data and create conditional random field model [13] then in prediction phase, test program was converted to dependency network and Maximum a Posteriori (MAP) inference was applied to infer the unknown properties from the known properties.

Authors of [14] has presented a new code metric, charm. Charming code is code that attracts the error messages from other part of program to itself. So there exist some part of code which shows more error messages even though it does not contain error. This lead to wrong error location reporting. In this work random mutations have been done to generate different type of errors at different locations. This is interesting concept as it gives a strong reason for wrong error location reporting.

CHAPTER 3. BACKGROUND

In this chapter we will define some of the algorithms and machine learning techniques that we will use as part of this study. Section 3.1 talks about hierarchical clustering. Section 3.2 talks about support vector machines. Section 3.3 discusses about probabilistic topic modeling. Section 3.4 talks about MLC.

3.1. Hierarchical clustering

Clustering is grouping similar documents into one clusters and dissimilar document into separate cluster. It tries to build a tree-based hierarchical taxonomy (dendogram) from a set of documents. We do not need to provide number of clusters, k, for dendogram creation. We can cut the dendogram tree into separate required number of clusters once we have the dendogram. There are two main types of hierarchical clustering

1. Agglomerative – It is a bottom-up approach. In this algorithm first start with considering each documents as a separate cluster and merging two most similar clusters iteratively. It ends when there is only one cluster left and return one dendogram tree.
2. Divisive – It is a top-down approach. In this algorithm first start with considering all documents as part of one cluster and recursively split the clusters based on the dissimilarity between documents. It ends when there is no more splitting possible.

3.2. Support vector machines

SVM[27] is a supervised machine learning technique which perform classification by constructing an optimal separating hyperplane. It finds the support vectors first, which are the

nearest training data points. Secondly it tries to find a separating hyperplane by maximizing the distance between support vectors and the plane. Larger the distance between support vectors and plane, the lower the classification error. It trains the classifier using training data and find the separating hyperplane. Once we have the plane it tries to map the test data and based on the sides it decides classes for test data.

3.3. Probabilistic topic modeling

The main objective of probabilistic topic modeling is to find hidden topics across a collection of documents. It is a statistical method that analyze the words of the original text documents to discover the themes and the relation between them. Author of [15, 26] talks about probabilistic topic models and presents simplest topic modeling algorithm, latent dirichlet allocation (LDA). The main idea behind LDA is that documents exhibit multiple topics. It randomly chooses distribution over topics within documents and reassigns words to topic based on the probabilities across all the documents. It gives us word distribution of each topic using which we can decide what are the key topics that belongs to each document. We will talk more about topic modeling in next chapters.

3.4. Multi-label classification

MLC[19, 22] is a machine learning technique which tries to assign multiple target labels to each document into a corpus of documents. There exist two different approaches for MLC. First is problem transformation methods which try to transform the multi-label classification into binary or multiclass classification problems. Second method, algorithm adaptation, try to adapt multiclass algorithms so they can be applied directly to the problem.

CHAPTER 4. ERROR DETECTION

In this chapter, we present our approach to predict the real errors using compiler error messages by applying statistical methods on open-source projects. In the following sections we will discuss the main steps towards achieving this goal. Section 4.1 talks about our terminology for classifying errors into one of the two types. In section 4.2 we define document term matrix. Section 4.3 discusses document clustering on error documents. Section 4.4 talks about how we analyze clusters which come as a result of clustering. Section 4.5 talks about training machine learning models with the clusters and predicting clusters for new error documents. In section 4.6 we discuss our second part of study, 2 errors at a time. Section 4.7 talks about finding word distribution for common error types. In the end section 4.8 talks about probabilistic topic modeling.

4.1. Type of error

Compiler error messages are very diverse and may contain multiple useful information related to error-fix. Based on the information provided by compiler error messages we can divide the them into two type -

Type A - Compiler error messages in which first error message can be parsed to get the correct fix for the error along with the location. exp – “aof.c:1130:37: error: expected ';' after goto statement”

Type B – Compiler error messages which does not have enough information which can be find by directly looking at the error messages itself and the can be used to fix for the errors. For example, if we take C code provided into following snippet and remove the

second last close curly bracket, it generates compiler error message, as shown in figure 4.1. It contains multiple errors even though there is only one error present. So we need further analysis of such error messages to correctly predict error type.

```
for (i = 0; i < bn; i++)
{
bf_key_init(&state, (char_u *)(bf_test_data[i].password),
        bf_test_data[i].salt,
        (int)STRLEN(bf_test_data[i].salt));
if (!bf_check_tables(state.pax, state.sbx, bf_test_data[i].keysum))
    err++;

/* Don't modify bf_test_data[i].plaintxt, self test is idempotent. */
memcpy(bk.uc, bf_test_data[i].plaintxt, 8);
bf_e_cblock(&state, bk.uc);
if (memcmp(bk.uc, bf_test_data[i].cryptxt, 8) != 0)
{
    if (err == 0 && memcmp(bk.uc, bf_test_data[i].badcryptxt, 8) == 0)
    EMSG(_("E817: Blowfish big/little endian use wrong"));
    err++;
}
}
```

```
1 blowfish.c:573:1: error: function definition is not allowed here
2 blowfish.c:610:1: error: function definition is not allowed here
3 blowfish.c:633:1: error: function definition is not allowed here
4 blowfish.c:654:1: error: function definition is not allowed here
5 blowfish.c:676:1: error: function definition is not allowed here
6 blowfish.c:690:24: error: expected '}'
```

Figure 4.1. Compiler error message when one close curly bracket missing

So based on above classification we can skip type A error documents as they don't need any other processing to predict the real error. We will focus on type B errors only in this study, and next steps will be applied on these errors.

4.2. Document term matrix

Document term matrix (DTM) is a numerical matrix which contains frequency of all the different terms across a group of documents. In the first part of study we work with the

type B error message documents corresponding to one error at a time. We first create a corpus of all the documents that need clustering. We then apply pre-processing on the corpus. Later a DTM is created corresponding the corpus which contain all the terms as the columns, documents as the row and values are the frequency of term in the document. We also normalized the frequency value across the corpus. We used Term frequency–Inverse document frequency, tfidf for this. It is a statistical measure used to evaluate how important a term t is to a document d in a collection or corpus D

$$\mathbf{tfidf(t,d,D) = tf(t,d)*idf(t,D)}$$

where,

tf(t,d), term frequency – It is frequency of term t in document d

and

idf(t,D), inverse document frequency – It is information provided by term t across all the documents D, it shows how common or rare term t is across corpus D. It can be obtained by taking logarithm quotient of division of the total number of documents by the number of documents containing the term t.

Figure 4.2 shows a sample DTM where each rows represents all the documents in the corpus and columns represent unique terms across corpus. Each value is represents tfidf calculated as mentioned before.

	aka	and	assignable	binary	called	closeparenthesis	expected	expression
red_type2_1450890184926.txt	0.000000	0.2643856	0.0000000	0.2643856	0.0000000	0.0000000	0.0000000	0.2643856
red_type2_1450890190073.txt	0.386988	0.2203213	0.0000000	0.2203213	0.0000000	0.0000000	0.0000000	0.2203213
red_type2_1450890248618.txt	0.000000	0.2643856	0.0000000	0.2643856	0.0000000	0.0000000	0.0000000	0.2643856
red_type2_1450923199764.txt	0.290241	0.0000000	0.0000000	0.0000000	0.0921207	0.0000000	0.0000000	0.0000000
red_type2_1450923278365.txt	0.000000	0.1888469	0.1581871	0.1888469	0.0000000	0.1581871	0.1581871	0.1888469
red_type2_1450923283590.txt	0.000000	0.0000000	0.0000000	0.0000000	0.1052808	0.0000000	0.0000000	0.0000000
red_type2_1450923288772.txt	0.000000	0.0000000	0.0000000	0.0000000	0.1052808	0.0000000	0.0000000	0.0000000
red_type2_1450923293946.txt	0.000000	0.0000000	0.0000000	0.0000000	0.1052808	0.0000000	0.0000000	0.0000000
red_type2_1450934316576.txt	0.000000	0.0000000	0.0000000	0.0000000	0.1052808	0.0000000	0.0000000	0.0000000
red_type2_1450935168837.txt	0.000000	0.0000000	0.0000000	0.0000000	0.1052808	0.0000000	0.0000000	0.0000000

Figure 4.2. Sample DTM

4.3. Document clustering

Once we have done pre-processing and DTM is created, we can perform document clustering on the corpus. Document clustering [25] is unsupervised learning method which group similar documents together into separate cluster. There are many algorithms available for document clustering but we found hierarchical clustering to be best for our experiments. It always gives same resultant clusters every time we run clustering on data, this was one of the main reason behind using hierarchical clustering algorithm. So for clustering first we try to find optimal number of clusters k that can be make from given corpus and then apply hierarchical clustering algorithm on it. Main reason for clustering is to find sub-types of errors based on the similarity of error messages for each sub-type.

We have used R package, NbClust, for finding k. It tries to vary different number of clusters, distance measure and clustering algorithm to find the optimal number of clusters. It also uses Silhouette of the data to check how closely data instances are matched within cluster and how loosely with instances in neighboring clusters.

Once we have cluster count k, we cut the dendogram tree into k clusters. This clustering gives us all the sub-types of that particular error. We repeat this clustering for all of the five main error types and calculate individual sub-types.

4.4. Cluster analysis

Once we have all the document grouped into different sub-type clusters, we manually try to find what can be the cause of errors for the documents in one cluster. Once we finalize one error for each cluster we decide the fix for the error, if possible. In the end we have written one Java program which contains all the possible fixes for different errors that we

found after cluster analysis. So we can use this for fixing the program based on the cluster information of new error document. For fixing, our auto-fix Java program uses cluster information and based on that it calls the appropriate fixing module and make changes in the original file. It parses approximate error location and file name from the original error message. Suppose for one sub-type most of the error messages were similar to error message "*called object type is not a function or function pointer"* and were generated when two function are called one after another and ';' is missing after first call. So in this case, our auto-fix program takes error message and sub-type information as input and based on the error details in error document and sub-type information it tries to insert ';' at correct location to fix the program. If we could not find proper fix for any sub-type, we did not make any changes in file and displayed the sub-type information without fixing it.

4.5. Training model and prediction

After we have enough cluster information for each sub-type of compiler error, we can train a prediction model with error message documents as training data and use this model for predicting clusters for new test error documents. We tried different classification algorithms and finally found Support Vector Machine (SVM) [27] to be the best and that is why used SVM for training the model. SVM is a supervised machine learning technique which perform classification by constructing an optimal separating hyperplane. We randomly divided the data into two parts, train and test dataset. Once we trained the model on the train dataset we used test dataset to predict the clusters. From this model we were able to predict correct cluster and probability of the assignment. We repeated this process multiple times by shuffling the data and creating training and test data-set. Once we have the cluster

information for new error document, we used java program written in section 4.4 to auto fix the error, as many as possible.

4.6. Handling multiple error

It is very common to have more than one error in the program. In this case when we compile the program, compiler error can be a combination of error messages corresponding to individual error. It makes it hard to predict cause from the compiler error messages alone. A sample error message document will look like as figure 4.3. It contains multiple errors and locations.

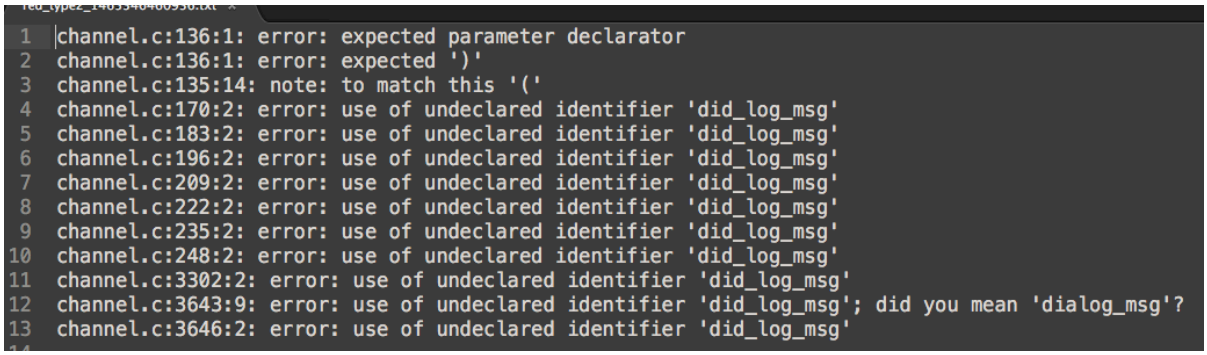

Figure 4.3. Sample error messages for 2 errors present at a time

To handle the case when we have two errors in the program we applied probabilistic topic modeling to our error messages. If we consider two errors as two topics and use corresponding word distribution to find topics, then it can help us finding the errors from error documents.

4.7. Finding word distribution

Any error can be represented by some of the key words that frequently occur in error messages when that particular error is present in the code. So first of all, we try to find word

distribution or most common words for each main error type. For this we use our error documents of one error at a time from previous steps and apply Latent Dirichlet Allocation (LDA) on corpus for probabilistic topic modeling. It tries to find what words are most probable for given errors across all the documents. This gives us top k words for each main error type.

4.8. Topic modeling

Once we have top k words for each individual error type, we can use this to find two errors into 2-error documents. We have implemented modified version of latent Dirichlet allocation algorithm. In this each error type corresponds to one topic and we are using these topics, for initializing the words in error document, based on the topic distribution. First we train our model with training data-set. For corpus of train data-set documents D, let d is a document and t is a term. We first initialize each word with a topic based on the topic distribution. After this we calculate following probabilities

- p(topic t | document d) - proportion of words in document d that are currently assigned to topic t
- p(word w | topic t) - proportion of assignments to topic t over all documents that come from this word w

Based on the resultant probability of both probabilities, we reassign words to most probable topics. For this assignment, algorithm uses Gibbs sampling [20]. We repeat this process multiple time. After large number of iterations, once no more change in allocation we stop the reallocation.

For testing and evaluation on test data-set. We start with initializing words with corresponding topics based on the topic distribution. Later we calculate both of the above mentioned probabilities for words in test documents and re-allocate words to topics based on the resultant probability. After large number of iterations, once no more change in allocation we stop the reallocation and try to calculate two most occurring topics in each document. For this calculation we count the words in one document and check which two topics have most number of words mapped to it. Those two topic correspond to most probable two errors that generated the given compiler error document.

CHAPTER 5. EXPERIMENTAL DESIGN AND RESULTS

In the following sub-sections, we will discuss our experiments and major findings. Section 5.1 gives details about data and tools used for this study. Section 5.2 talks about pre-processing steps taken to clean the data. Section 5.3 gives details about our approach for handling one error at a time cases. Section 5.4 talks about how we handle case when two different errors are present in the program, using topic modeling. Section 5.5 talks about our approach for previous two error cases using MLC. In the end section 5.6 discusses limitation of this study.

5.1. Data and tools

Our study focuses on projects in C language. We have taken open-source projects from Github for this study. We have also picked 5 most common errors based on study in [9]. All the preprocessing and data collection programs are written in Java. R programming language is used for statistical analysis and model creation. Data collection for this study is done in two parts. In first part we have developed a tool which take a C project and type of error and then randomly generate one error, compile the program and collect all of the compiler error messages in one text document. To divide the errors into two types our programs also try to parse and fix the error using first compiler error message only. If it is fixable error, then it is marked as type A error otherwise type B error. Only type B error documents are used for further processing. We used tm package [23] in R for all of the text mining and clustering. For building the model, training and evaluation mainly we have used RTextTools package [24] in R.

In second part, we have developed another program which takes a C project and two different errors. In this program both the errors are randomly generated in one of the file in the project. After this, project is compiled and all the compiler error messages are collected in separate text document. These documents are used for further processing. We used Java and R for topic modeling. To work with MLC, we have used Java based open source tool, Mulan [21].

5.2. Pre-processing

Data pre-processing is important step towards statistical analysis. In our study each error message contains many unnecessary terms and information that would not help in document clustering as it does not add value towards classification. So to clean this noise we performed some common pre-processing and some domain specific pre-processing. Main steps involved in pre-processing are as follows –

- Removing file name and location – As it does not provide any information about the error and cannot be use for classification
- Converting to lower case – To avoid duplicate counting of same words e.g. “Text” is equal to “text”
- Removing all the numbers – No information about errors and common across different text documents
- Removing punctuation and whitespaces - No information about errors and common across different text documents
- Removing some domain specific frequently occurring words e.g. error, ‘int’ - No information about errors and frequently occur across different text documents

- Changing some symbols to words based on importance e.g. '{' to closecurly – So that English string can contribute towards classification

5.3. One error at a time

In this part we focus on the case when we have single error present in the program. We randomly add one compiler error and try to find that error using compiler error message generated. We have used document clustering and SVM for this part of study.

5.3.1. Experimental design

In this we first tried to collect error documents corresponding to both the types for 5 main errors - semicolon missing, undeclared variable, close curly bracket missing, open curly bracket missing, closing parenthesis missing, as we discussed in section 4.1. Each error documents contains all the compiler error messages generated by the compiler for one error. Table 5.1 shows count of type A and type B errors for each of the five errors for both the projects.

Table 5.1. Count of error documents

Error Type	Subtype	Count of error documents (project 1)	Count of error documents (project 2)
Semicolon missing	Type A	9610	10582
	Type B	460	624
Undeclared Variable	Type A	0	0
	Type B	507	277

Table 5.1. (continued)

Close curly bracket missing	Type A	0	48
	Type B	2219	8357
Open curly bracket missing	Type A	0	0
	Type B	4461	4583
Closing parenthesis missing	Type A	7161	5945
	Type B	1078	1209

Once we have enough data points, we disregard type A error messages as no further processing require for fixing these errors. In next step, we applied pre-processing on type B errors, as we discussed in section 5.2. Following is the example of an error message before and after pre-processing –

“aof.c:238:5: error: called object type 'int' is not a function or function pointer” → “called object type is not a function or function pointer”

As discussed in section 4.2, once our data is cleaned we convert it to DTM for further processing. Figure 5.1 shows summary of one of the DTM

```
<<DocumentTermMatrix (documents: 460, terms: 60)>>
Non-/sparse entries: 1790/25810
Sparsity           : 94%
Maximal term length: 16
Weighting          : term frequency (tf)
```

Figure 5.1. Summary of a DTM

Once DTM is created, we applied hierarchical clustering to both of the project individually using *hclust* package to find all the sub-types of errors, as discussed in section

4.3. We calculated optimal number of clusters as per section 4.3 and used this to cut the dendogram tree into separate clusters. Each sub-type represents one kind of error messages that can be generated using one particular compiler error.

We manually checked each cluster and tried to find if there is a consistency between all the error documents in one cluster. Also if there exist some pattern between the real compiler errors corresponding to error documents in one cluster. If we found a pattern, we decided a fix for the errors within each cluster. If no pattern can be found, then we marked that cluster as cannot be auto-fixed cluster. If we take example of one cluster that we found as a sub-type of semicolon missing error.

e.g. cluster 1 – It contains compiler errors with same pattern where ';' is missing in the line just before value pointer as shown in the following code. Here if we have ';' from the end of the first line then the corresponding error message "invalid operands to binary expression" belong to cluster 1.

```
sec = when_sec;
*ms = when_ms;
```

Once we have the pattern, we can decide the fix for this cluster and auto-fix the error by adding missing semicolon in the line just before the value pointer in the end of first line.

Similarly, we divided all of the error documents into different clusters and based on the pattern in the compiler error messages within each clusters we marked them if they can be auto-fix or not.

Figure 5.2 shows dendogram tree for semicolon type errors. We created similar dendogram for all of the five error types and divided them into separate clusters.

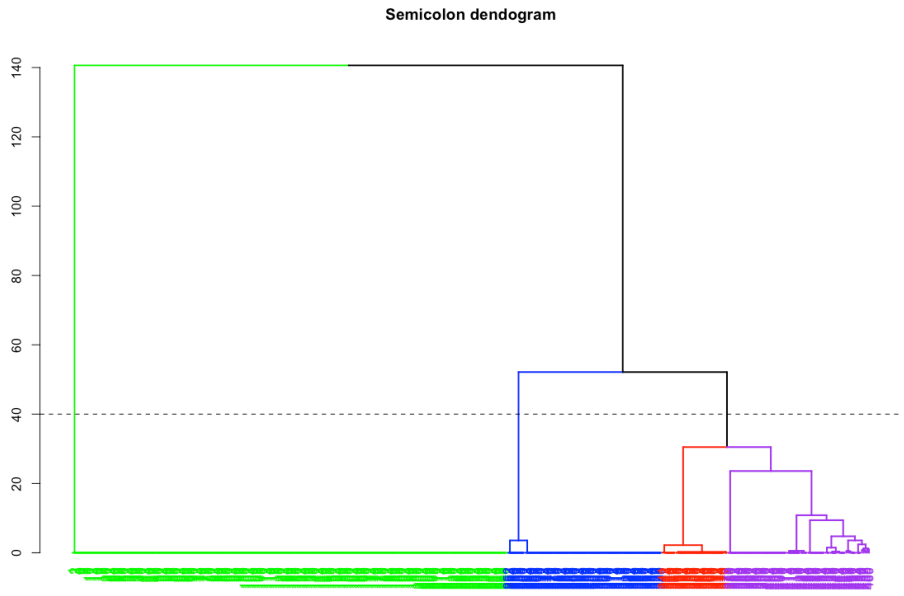

Figure 5.2. Dendogram for semicolon type-B errors with 4 clusters

Table 5.2 shows count of error documents in each sub-types for type-B errors of all 5 main type of errors. It also shows if any sub-type is auto-fixable or not. We created auto-fix modules for sub-types of 2 main types of errors. For remaining 3 type of errors we marked them as could not be auto-fix based on the sub-type information, as we could not find significant pattern within one sub-type.

Table 5.2. Count of error documents in each sub-type

Error Type	Subtype	Count of error documents (project 1)	Count of error documents (project 2)	Auto-fixable
Semicolon missing	1	38	122	Yes

Table 5.2. (continued)

	2	89	163	Yes
	3	82	220	Yes
	4	251	119	Yes
Close curly bracket missing	5	2148	8357	No
Open curly bracket missing	6	2727	4583	No
Closing parenthesis missing	7	415	228	Yes
	8	466	655	Yes
	9	197	326	Yes
Undeclared Variable	10	36	77	No
	11	456	598	No
	12	15	51	No

So once all the sub-types of main error-types were found, we divided our error documents into two sets with 9:1 ratio and used them as training and test dataset respectively, we used training dataset to train our model, as discussed in section 4.5. SVM was used to build the classification model using all of the subtypes with RTextTools package from R. Once we have the trained model, we used it to predict sub-types for new error documents in the test dataset. Once we predicted sub-types, we tried auto fixing the errors by passing the sub-type information and error document to our program for auto-fixing. If error is auto-fixable then it fixes the error and make changes into the file otherwise it prints the error sub-type for the error document, as discussed in section 4.4.

5.3.2. Findings

In this part of study for one error documents, we worked with total five main errors and made 12 sub-types of these errors. After training the model with combined data we tested our model with test data, which was also combination of all 12 sub-types.

- test data set size – 1550

In this figure 5.5 we can see number of test documents mapped to different clusters corresponding to different subtypes. The green points show the real cluster and count of documents belonging to that cluster. The red points show the predicted clusters and count of documents belonging to that cluster. We have created total 12 sub-types.

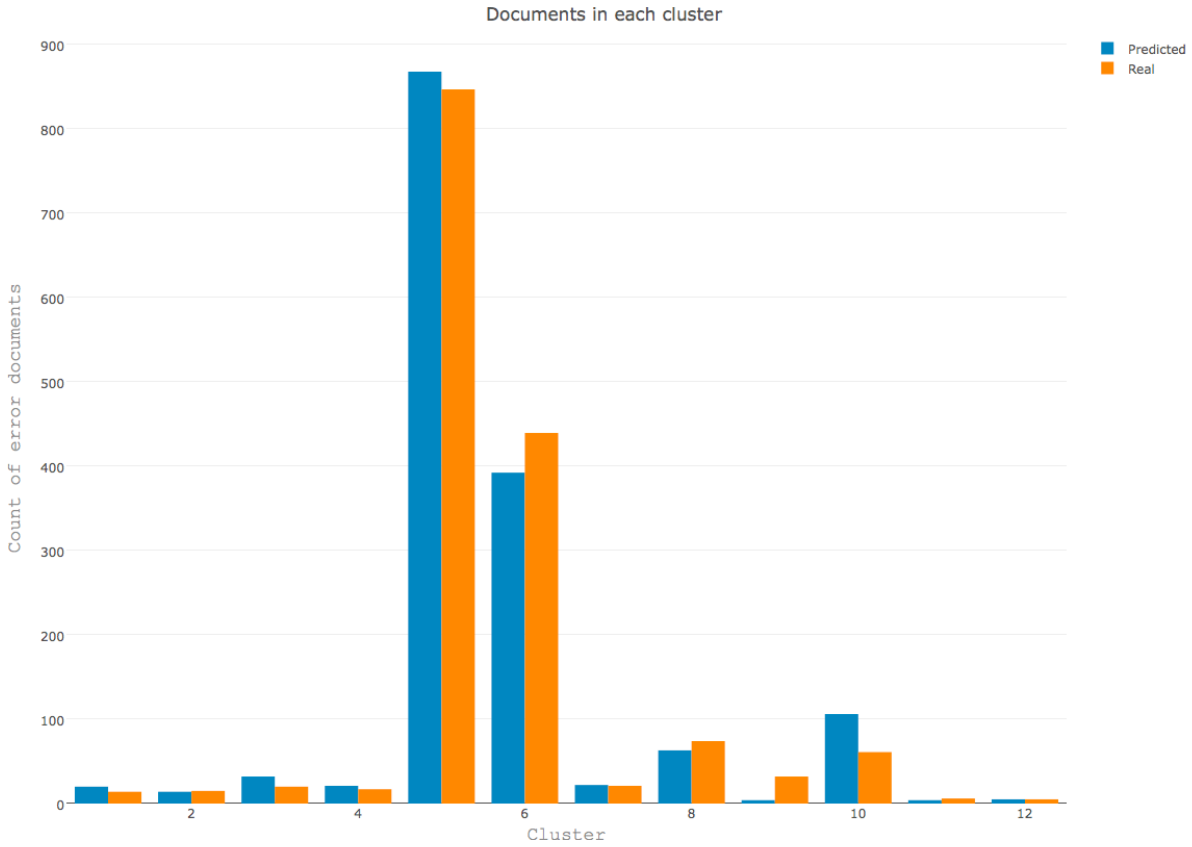

Figure 5.5. Prediction outcome for 1-error test data

Once we predicted sub-type clusters we checked how many error documents are correctly matched so we compared the predicted and actual sub-type for each of the test document. Following plot 5.6 shows correctly predicted vs incorrectly predicted sub-type counts for both of the projects. Orange shows percentage of correctly predicted error documents and blue shows percentage of incorrectly matched error documents.

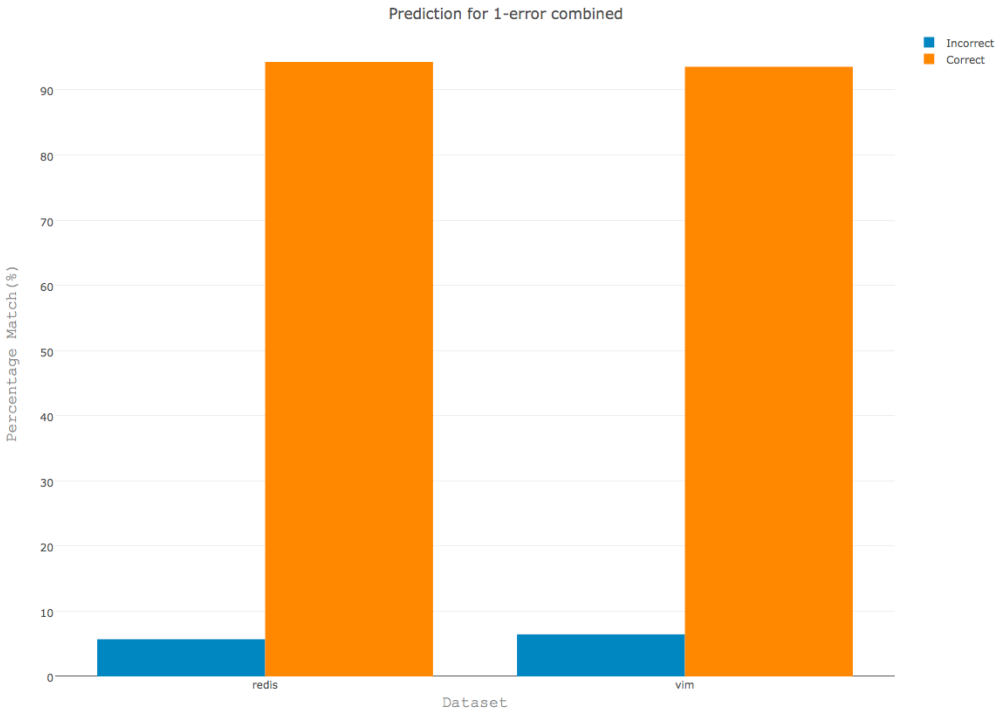

Figure 5.6. Prediction results

In last step we tried auto-fixing errors using predicted sub-type and error document for a sub-set of documents for each sub-type, where we correctly predicted sub-type. Following table 5.3 shows results for our auto-fixing tool. In this we used a small number of test documents for each sub-type and corresponding correctly predicted sub-type information to auto-fix the program, as per section 4.4. If error is auto-fixable then table shows the

percentage of correctly auto-fixed error documents otherwise it shows that error cannot be auto-fix using sub-type information.

Table 5.3. Count of error documents in each

Error Type	Subtype	Count of error documents (project 1)	Auto-fixable(%) / Not auto-fixable	Count of error documents (project 2)	Auto-fixable(%) / Not auto-fixable
Semicolon missing	1	30	83.3	30	76.7
	2	30	76.7	30	80
	3	30	76.7	30	80
	4	30	80.0	30	86.7
Close curly bracket missing	5	100	Not auto-fixable	100	Not auto-fixable
Open curly bracket missing	6	100	Not auto-fixable	100	Not auto-fixable
Closing parenthesis missing	7	30	66.7	30	76.7
	8	30	86.7	30	80
	9	30	76.7	30	66.7
Undeclared Variable	10	30	Not auto-fixable	30	Not auto-fixable
	11	100		100	
	12	15		15	

So to conclude our finding for the case when we have one error at a time in the program –

- We divided our random error messages into two types and as type-A error messages can be auto fix without any further analysis, we discarded those as part of this study
- For type-B error documents first we tried predicting sub-type information and our prediction results are almost 95% correct. So we correctly predicted sub-type for 95% of test data.
- Based on the correctly predicted sub-type information we tried auto-fixing errors in the program, we implemented auto-fixing module for sub-types of two main errors and fixed programs correctly in almost 80% of test error documents. For rest 3 errors we could not find a fix based on the sub-type information.

5.4. Two errors at a time with topic modeling

In this part we focus on the case when two different compiler errors are present at the same time in the program. We add two compiler errors at random locations and try to find those errors using compiler error message generated. We have used probabilistic topic modeling for this part of study.

5.4.1. Experimental design

Firstly, we tried finding word distribution corresponding to main four error types. As discussed in section 4.7, we used type-B single error documents from first part of study and applied probabilistic topic modeling to find most common k-words for each error type. Table 4.3 shows word distribution for each main error type. We ignored "semicolon missing" error in this part of study because count of type-B errors for semicolon missing type was relatively very low as compared to count of type-A errors and for finding word distribution we were using only type-B error messages. Table 5.4 shows word distribution for each of the four error types.

Table 5.4. Top 10 words for each main type of errors

Open curly bracket missing	**Close parentheses missing**	**Variable undeclared**	**Close curly bracket missing**
identifier	closingbracket	undeclared	definition
types	macro	identifier	closingbracket
closing	expression	use	function
function	invocation	member	allowed
extraneous	function	type	expression

Table 5.4. (continued)

type	functionlike	invalid	labels
brace	allowed	named	default
declarator	unterminated	increment	aka
use	definition	expression	switch
parameter	closeparentheses	declaration	type

Once we got the word distribution for all of the 4 main type errors we made 6 combinations of error pair from these 4 errors. First we generate random error messages corresponding to each of the 6 pairs. Table 5.5 shows count of each 2-error combination documents, generated from one project. Later we implemented modified Latent Dirichlet Algorithm, as discussed in section 4.8, for further probabilistic topic modeling. In this we used our individual error word distribution and two error documents from table 5.5 to train the model.

Table 5.5. Count of each 2-error combination documents used for training

Error Combination	**Count**
Close Curly, Open Curly	600
Close Curly, Close Parenthesis	2792
Close Curly, Variable undeclared	3200

Table 5.5. (continued)

Open Curly, Close Parenthesis	2786
Open Curly, Variable undeclared	3197
Variable undeclared, Close Parenthesis	3200

5.4.2. Findings

We created a test data set for each of the 6 pairs. We used 600 total number of test documents of each type of pair for evaluating our model once it is trained with training dataset. We count the words in each test document and check which two topics have most number of words mapped to it. Those two topic correspond to most probable two errors that generated the given compiler error document, as we explained in section 4.8.

Figure 5.7 shows results for percentage of correct predictions for each of the six pair. It shows percentage values with different colors for different matching e.g. green color shows we correctly predicted both the errors in each document etc.

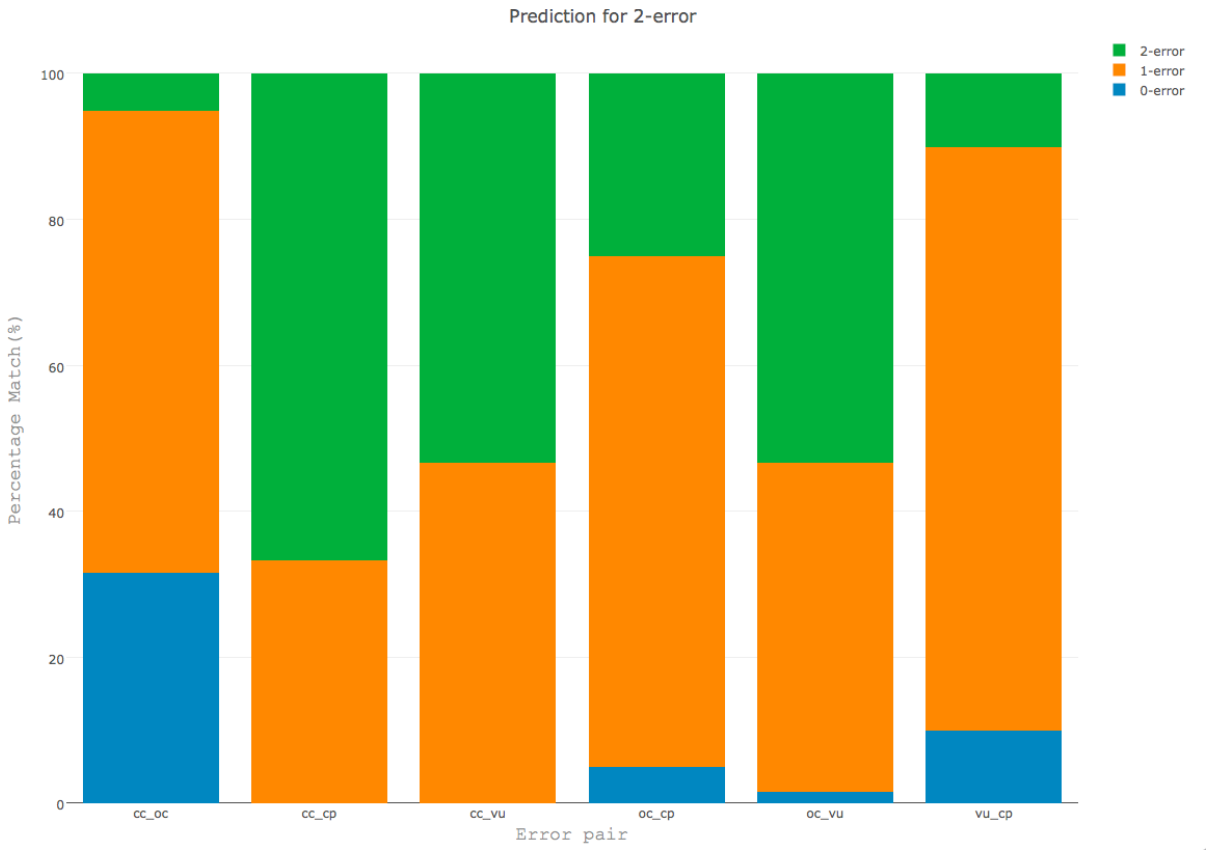

Figure 5.7. Prediction results for 2-errors data

So to conclude our findings for topic modeling for the case when we have two different errors at the time in the program –

- We predicted at least one error correctly in almost 91.3% of documents.
- We predicted both the errors correctly in almost 35.5% of documents.
- We could not predict any of the error out of 2 correctly in almost 8.7% documents. As some error messages can be common between different compiler errors so this affect our performance while predicting separate errors using error messages.

5.5. Two errors at a time with MLC

In this part also we focus on the case when two different compiler errors are present at the same time in the program. We randomly add two compiler errors and try to find those errors using compiler error message generated. We have used Multi-Label Classification for this part of study.

5.5.1. Experimental design

In this part, we implemented Multi-Label Classification on the two-error documents and for this we used same training data as mentioned in table 5.5. For this, first we created a training dataset which included combined pre-processed 2-error messages and 4 labels corresponding to 4 errors with values (0,1). Labels has value 1 when corresponding error is present in the code. So two of the labels has values 1 and remaining two has values 0. We used Mulan [21], a Java based open-source software devoted to multi-label data mining, for our study. Mulan requires two text files for the specification of a multi-label dataset first is XML file specifying all the labels and second is ARFF file which contains actual data that needs classification along with the labels which should be specified as nominal attributes with two values "0" and "1" indicating absence or existence of the label respectively. So we transformed our two error training documents into required ARFF and XML datasets using mldr package from R and applied MLC. Figure 5.8 shows a sample XML file for the training dataset. Figure 5.9 shows one of the document in ARFF file, here last 4 values represent 4 labels and values of labels shows which 2 errors are present.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <labels xmlns="http://mulan.sourceforge.net/labels">
3  <label name="v_u"></label>
4  <label name="c_c"></label>
5  <label name="o_c"></label>
6  <label name="c_p"></label>
7  </labels>
```

Figure 5.8. Sample XML file

```
121  @data
122  0,0,0,0.187241434179997,0,0,0,0,0,0,0,0.0750030159917407,0,0,0,0,0.0750030159917407,0.
     0912329411138607,0,0,0,0,0,0,0,0,0.187715107419719,0,0,0,0,0,0,0.026540947452694,0,0.0750030159917407,0,0.
     156938357642535,0,0.0905389214103563,0,0,0,0,0,0,0,0,0,0,0,0,0,0.0977514756761985,0,0,0,0,0,0,0,0.
     493605460497335,0,0,0,0,0,0,0,0.104649294317446,0,0,0,0,0,0,0.495410771537417,0,0,0,0,0,0,0,0,0,0,0,0,0,0.
     495410771537417,0,0.216025190215573,0,0.0860579730198489,0.0864732189857589,0,0,0,0,0.206447902608636,0,0,0,0,0,0,0.
     108556241463902,0,0,1,1,0
```

Figure 5.9. Sample document in ARFF file

Once our training data is ready, we train a classifier model with it using RAkEL algorithm. After training the classifier, we performed prediction for new unlabeled 2-error data. From the labels returned from the classifier we decided the two most relevant labels for each document. Those 2 labels correspond to 2 type of errors present in program for each of the error document.

5.5.2. Findings

We used same test dataset, 6 pair combination out of 4 main errors, as previous section for evaluating our MLC model once it is trained with training dataset. We predicted two most probable labels in each test document. Those two labels correspond to most probable two errors that generated the given compiler error document. Figure 5.10 shows results for percentage of correct predictions for each of the six pair, individually. It shows percentage values with different colors for different matching e.g. green shows we correctly predicted both the errors etc.

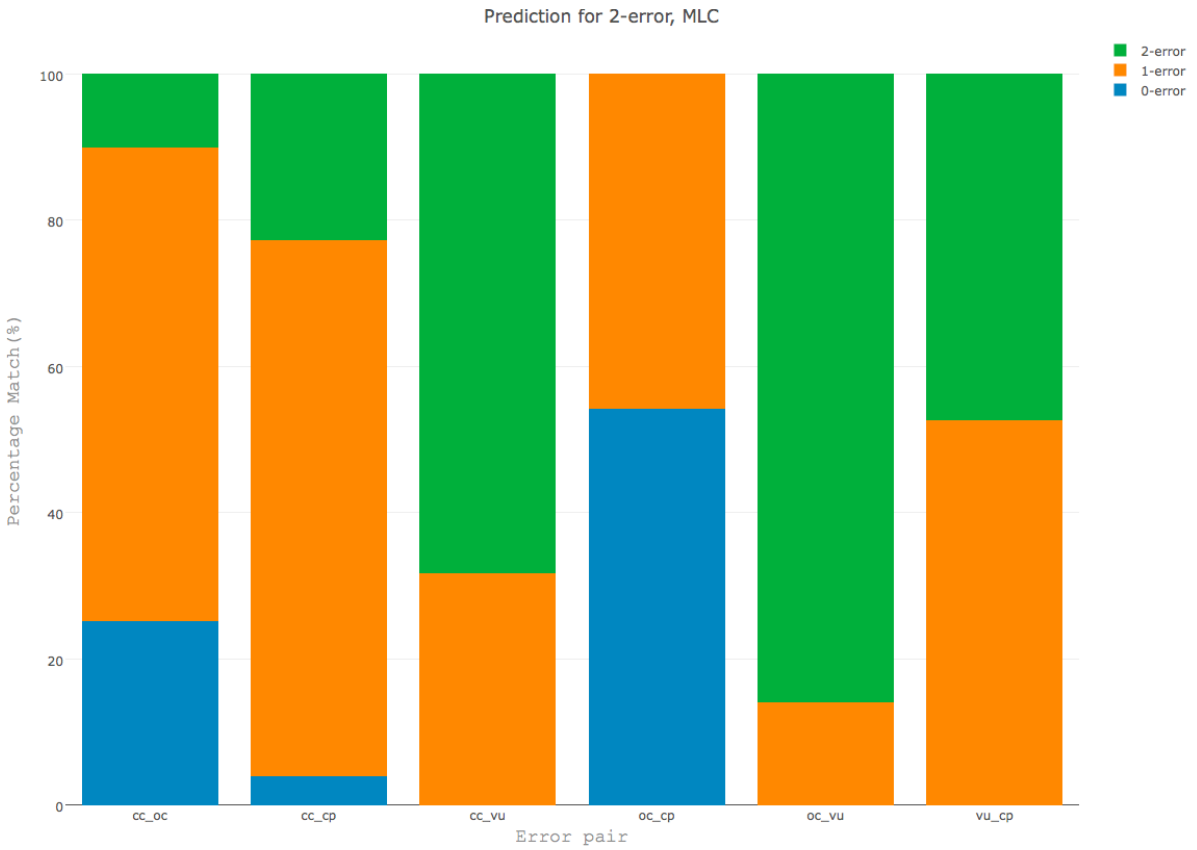

Figure 5.10. MLC model prediction results for 2-errors

So to conclude our findings for MLC for the case when we have two different errors at the same time in the program –

- We predicted at least one error correctly in almost 86.05% of documents.
- We predicted both the errors correctly in almost 39.02% of documents.
- We could not predict any of the error out of 2 correctly in almost 13.95% documents. As some error messages can be common between different compiler errors so this affect our performance while predicting separate errors using error messages.

5.6. Limitations of the study

This study uses five most common compiler errors and corresponding compiler error messages but it is possible that when we add more type of errors, performance of classifier and model may get impacted and results may get changed. This study focuses on C programming language and GCC compiler, so results may change for different programming language and compiler.

CHAPTER 6. CONCLUSIONS AND FUTURE WORK

Compiler error message reporting is not accurate enough and because of incorrect or ambiguous errors, developer waste significant efforts. We have described an approach to predict some common compiler errors, that developers usually make, using statistical analysis and machine learning techniques on large open-source code base. We have worked with five main errors and tried to handle the cases when one error or two errors out of these five are present in code. We have experimented with document clustering, probabilistic topic modeling and multi-label classification algorithms. Our empirical evaluation on open-source projects has shown that our model correctly predicted the real error in almost 95% cases, when only one error exists. In case of two errors, we correctly predicted at least one error in almost 91% cases and both the errors in almost 39% cases.

Our future work includes (1) work on improving the performance when predicting multiple errors, (2) working with more types of errors and (3) working with cases when number of errors present in the code is unknown.

REFERENCES

[1] Raychev, Veselin, Martin Vechev, and Andreas Krause. "Predicting program properties from big code." In ACM SIGPLAN Notices, vol. 50, no. 1, pp. 111-124. ACM, 2015.

[2] M. Allamanis and C. Sutton. Mining source code repositories at massive scale using language modeling. In Working Conference on Mining Software Repositories (MSR), 2013.

[3] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In International Conference on Software Engineering (ICSE), 2012.

[4] J. Campbell, A. Hindle, and J. N. Amaral. Syntax errors just aren't natural: Improving error reporting with language models. In Working Conference on Mining Software Repositories (MSR), 2014.

[5] Jadud, Matthew C. "Methods and tools for exploring novice compilation behaviour." In Proceedings of the second international workshop on Computing education research, pp. 73-84. ACM, 2006.

[6] Jadud, Matthew C. "A first look at novice compilation behaviour using BlueJ." Computer Science Education 15, no. 1 (2005): 25-40.

[7] L. McIver. The effect of programming language on error rates of novice programmers. In 12th Annual Workshop of the Psychology of Programming Interest Group, pages 181–192. Citeseer, 2000.

[8] Garner, Sandy, Patricia Haden, and Anthony Robins. "My program is correct but it doesn't run: a preliminary investigation of novice programmers' problems." In Proceedings of the 7th Australasian conference on Computing education-Volume 42, pp. 173-180. Australian Computer Society, Inc., 2005.

[9] Denny, Paul, Andrew Luxton-Reilly, and Ewan Tempero. "All syntax errors are not equal." In Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education, pp. 75-80. ACM, 2012.

[10] Tabanao, Emily S., Ma Mercedes T. Rodrigo, and Matthew C. Jadud. "Identifying at-risk novice java programmers through the analysis of online protocols." In Philippine Computing Science Congress, pp. 1-8. 2008.

[11] Kummerfeld, Sarah K., and Judy Kay. "The neglected battle fields of syntax errors." In Proceedings of the fifth Australasian conference on Computing education-Volume 20, pp. 105-111. Australian Computer Society, Inc., 2003.

[12] Sun, Chengnian, Vu Le, and Zhendong Su. "Finding and analyzing compiler warning defects." In Proceedings of the 38th International Conference on Software Engineering, pp. 203-213. ACM, 2016.

[13] Lafferty, John, Andrew McCallum, and Fernando CN Pereira. "Conditional random fields: Probabilistic models for segmenting and labeling sequence data." (2001).

[14] Campbell, Joshua C., and Abram Hindle. The charming code that error messages are talking about. No. e1388. PeerJ PrePrints, 2015.

[15] Blei, David M., Andrew Y. Ng, and Michael I. Jordan. "Latent dirichlet allocation." the Journal of machine Learning research 3 (2003): 993-1022.

[16] Singhal, Amit. "Modern information retrieval: A brief overview." IEEE Data Eng. Bull. 24, no. 4 (2001): 35-43.

[17] Redis project. Available: https://github.com/antirez/redis

[18] Vim project. Available: https://github.com/vim/vim

[19] Tsoumakas, G., Katakis, I., Vlahavas, I. (2010) "Mining Multi-label Data", Data Mining and Knowledge Discovery Handbook, O. Maimon, L. Rokach (Ed.), Springer, 2nd edition, 2010.
[20] Resnik, Philip, and Eric Hardisty. Gibbs sampling for the uninitiated. No. CS-TR-4956. MARYLAND UNIV COLLEGE PARK INST FOR ADVANCED COMPUTER STUDIES, 2010.
[21] Mulan tool. Available: http://sourceforge.net/projects/mulan/
[22] Tsoumakas, Grigorios, and Ioannis Katakis. "Multi-label classification: An overview." Dept. of Informatics, Aristotle University of Thessaloniki, Greece (2006).
[23] tm package. Available: http://CRAN.R-project.org/package=tm
[24] RTextTools. Available: https://cran.r-project.org/web/packages/RTextTools/index.html
[25] Steinbach, Michael, George Karypis, and Vipin Kumar. "A comparison of document clustering techniques." In KDD workshop on text mining, vol. 400, no. 1, pp. 525-526. 2000.
[26] Blei, David M. "Probabilistic topic models." Communications of the ACM 55, no. 4 (2012): 77-84.
[27] Wang, Lipo, ed. Support vector machines: theory and applications. Vol. 177. Springer Science & Business Media, 2005.