Enhancing Fault Localization with Cost Awareness

Kanagaraj Nachimuthu Nallasamy

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science and Applications

Francisco Servant, Chair

Na Meng

Aditya Prakash

May 15, 2019

Blacksburg, Virginia

Keywords: fault localization, automated debugging, source code line features, cost-aware fault localization

Enhancing Fault Localization with Cost Awareness

Kanagaraj Nachimuthu Nallasamy

(ABSTRACT)

Debugging is a challenging and time-consuming process in software life-cycle. The focus of the thesis is to improve the accuracy of existing fault localization (FL) techniques. We experimented with several source code line level features such as line commit size, line recency, and line length to arrive at a new fault localization technique. Based on our experiments, we propose a novel enhanced cost-aware fault localization (ECFL) technique by combining line length with the existing selected baseline fault localization techniques. ECFL improves the accuracy of DStar (Baseline 1), CombineFastestFL (Baseline 2), and CombineFL (Baseline 3) by locating 81%, 58%, and 30% more real faults respectively in Top-1 evaluation metric. In comparison with the baseline techniques, ECFL requires a marginal additional time (on an average, 5 seconds per bug) and data while providing a significant improvement in accuracy. The source code line features also improve the baseline fault localization techniques when "learning to rank" SVM machine learning approach is used to combine the features. We also provide an infrastructure to facilitate future research on combining new source code line features with other fault localization techniques.

Enhancing Fault Localization with Cost Awareness

Kanagaraj Nachimuthu Nallasamy

(GENERAL AUDIENCE ABSTRACT)

Software debugging involves locating and fixing faults (or bugs) in software. It is a challenging and time-consuming process in software life-cycle. Fault localization (FL) techniques help software developers to locate faults by providing a ranked set of program elements. The focus of the thesis is to improve the accuracy of existing fault localization techniques. We experimented with several source code line level features such as line commit size, line recency, and line length to arrive at a new fault localization technique. Based on our experiments, we propose a novel enhanced cost-aware fault localization (ECFL) technique by combining line length with the existing selected baseline fault localization techniques. ECFL improves the accuracy of DStar (Baseline 1), CombineFastestFL (Baseline 2), and CombineFL (Baseline 3) by locating 81%, 58%, and 30% more real faults respectively in Top-1 evaluation metric. In comparison with the baseline techniques, ECFL requires a marginal additional time (on an average, 5 seconds per bug) and data while providing a significant improvement in accuracy. The source code line features also improve the baseline fault localization techniques when machine learning approach is used to combine the features. We also provide an infrastructure to facilitate future research on combining new source code line features with other fault localization techniques.

Dedication

To my parents, teachers, advisor, guru, and Almighty for their infinite love and support.

Acknowledgments

I would like to thank my thesis and graduate advisor Dr. Francisco Servant. He inspired me to improve my problem solving and research skills. He always guided me towards the correct solution whenever I was stuck on a problem. I would also like to thank my committee members Dr. Na Meng and Dr. Aditya Prakash for their support and guidance. Special thanks to my lab mates especially to Khadijah who was always there to help me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and Problem

Software debugging, a challenging and time-consuming process, involves locating and fixing bugs. Software fault localization (FL) focuses on the former step to identify faulty elements. Since software has large size and complexity, manual fault localization is tedious, time-consuming and expensive [25]. Besides, the effectiveness of manual fault localization would mainly rely on the expertise of the software developers in identifying and locating faulty code or bugs. Automated fault localization addresses these challenges by locating faults automatically to reduce human time and effort. Automated fault localization expedites the debugging process, which in turn will increase the productivity of software developers. Moreover, automated fault localization enables automatic program repair to locate the faults automatically before fixing them.

1.2 User Studies and Existing solutions

Several recent studies [27], [20], [28], [10] on the applicability of automated fault localization techniques found that practitioners need accurate and fast fault localization techniques to adopt automated fault localization in practice. Our focus of this research is to improve the accuracy of the existing fault localization techniques that provide different trade-offs

between data needed, accuracy, and run time. We aim to improve accuracy while keeping the additional overhead (both data needed and time) as low as possible. We believe that such a novel fault localization has a tremendous potential for widespread adoption.

1.3 Proposed Solution

We propose a novel enhanced cost-aware fault localization (ECFL) that improves the accuracy of the selected existing baseline fault localization techniques. From the existing fault localization techniques, we identified three different baselines, i.e. DStar (Baseline 1), CombineFastestFL (Baseline 2), and CombineFL (Baseline 3). Each baseline technique provides different trade-offs between data needed, time, and accuracy. DStar achieves high accuracy with low runtime and low requirement of data. CombineFastestFL takes low runtime and provides high accuracy. CombineFL is the slowest technique but provides the highest accuracy. To improve the accuracy, ECFL combines source code line level features with the selected best fault localization baselines using a weighted sum formula. By making use of source code line features that would take a low extraction cost, the additional overhead (both data needed and time) will be minimal.

1.4 Experiments

We performed three experiments to design the ECFL technique that improves the accuracy of the selected best fault localization techniques in the literature. First, we experimented combining various source code line level characteristics (features) such as line commit size, line recency, and line length with the selected baselines. By experimenting with multiple weight pair configurations, we determined the best weight pair for each of our features. We

arrived at the best feature that provides the highest accuracy. Second, we used machine learning technique to combine our source code line features with the baselines instead of weighted sum approach. Then, we compared the ECFL (weighted sum) results with the machine learning results and provided a recommendation of the best FL techniques for each baseline (scenario). Third, we analyzed the accuracy improvement of the proposed FL technique across five different subjects.

1.5 Results

Evaluation of ECFL shows that it improves the accuracy of DStar (Baseline 1), Combine-FastestFL (Baseline 2), and CombineFL (Baseline 3) by locating 81%, 58%, and 30% more real faults respectively in Top-1 evaluation metric. The source code line features used in ECFL are easy to extract. ECFL requires a marginal additional time (on an average, 5 seconds per bug) and data. Our results show that ECFL (weighted sum) provides better overall accuracy than machine learning technique for all three baselines. Since ECFL is designed at the line level, we are not comparing our results with other method level fault localization techniques.

1.6 Infrastructure

We provide our implementation as an infrastructure (ECFL) to enable future research on combining various source code line features with other fault localization techniques. Using our ECFL infrastructure, researchers could build new fault localization techniques by extracting various source code line level features and combine them with other fault localization techniques.

1.7 Contributions

The contributions of the thesis include:

- We propose a novel enhanced cost-aware fault localization technique (ECFL) that improves the accuracy of the state-of-the-art fault localization techniques significantly. ECFL adds only a marginal additional time (an average of 5 seconds per bug).
- We compare the contributions of different source code line level features such as line commit size, line recency, and line length in improving the fault localization accuracy.
- We show that ECFL (weighted sum) performs better than learning to rank (linear SVM) machine learning approach.
- We provide an infrastructure (ECFL) to enable future research on combining different source code line features with other fault localization techniques.

1.8 Overview of Thesis

The rest of the thesis is organized as follows: Chapter 2 provides a background of different families of fault localization we study. Chapter 3 gives a case study for choosing baseline fault localization techniques. Chapter 4 describes the approach of our technique. Chapter 5 gives the research questions, subjects, ground truth and evaluation metrics. Chapter 6 describes the experiment for research question 1 and its results. Similarly, Chapter 7 describes the experiment for research question 2 and its results. Chapter 8 explains the experiment for the final research question 3 and its results. Chapter 9 discuses the recommendation of fault localization techniques for different usage scenarios and the ECFL infrastructure. Chapter

10 presents related work. Chapter 11 concludes the thesis by summarizing the results and contributions.

Chapter 2

Background

This chapter provides an overview on various types of fault localization techniques such as spectrum-based fault localization (SBFL), mutation-based fault localization (MBFL), slicing-based fault localization, stack trace-based fault localization, and predicate switching. We explain these line level fault localization techniques in this chapter since our goal is to focus on the line level techniques and not on method level fault localization techniques.

2.1 Spectrum-based Fault Localization (SBFL)

In 2001, Jones et al. proposed a spectrum-based fault localization (SBFL) technique called Tarantula [7] that uses coverage and test case execution results to compute the suspiciousness score for each executed program element. In 2006, Ochiai similarity coefficient-based technique [3] was proposed, and it was shown to perform better than Tarantula. Later in 2014, Wong et al. [26] proposed a fault localization technique called DStar that outperformed all the previously proposed 31 similarity coefficient-based SBFL techniques.

2.2 Mutation-based Fault Localization (MBFL)

Mutation-based fault localization (MBFL) changes a statement and checks whether the execution of the modified program with the changed statement (mutant) affects the result of

the previously run test case. MBFL uses execution information from the mutation analysis as the input to compute suspiciousness scores. Papadakis and Traon [18] used mutation testing for fault localization. The two state-of-the-art MBFL techniques are MUSE [15] and Metallaxis [19].

2.3 Slicing-based Fault Localization

Program slicing helps to reduce the set of program elements to look for while debugging by considering only a subset of program elements that might affect a specific variable under consideration. The term slice is used to denote a subset of program elements that might affect the variable under consideration. Dynamic slicing [4] considers only the executed statements for one execution of a single test case by making a slice of the program.

2.4 Stack trace-based Fault Localization

During the execution of a program, each function call is mapped as a stack frame internally by the system. When a program crashes, developers typically use the stack frame information such as last called function stack while debugging. Stack trace based fault localization uses stack frames information to locate faults [32].

2.5 Predicate Switching

Predicate switching [31] uses changing or switching the executed predicates (conditional expressions) in a program to check whether there is any change in the test result. When switching a particular predicate produces a passed test result from a failed test result, then

such predicates might be a reason for the fault. Such predicate switching information is used to perform fault localization.

Chapter 3

Case Study

This chapter presents a case study we conducted to determine the fault localization baseline techniques from the existing fault localization techniques. We use the baselines found in this chapter for the rest of the thesis.

3.1 Two Classes of Existing Fault Localization Families

Figure 3.1, Figure 3.2, Figure 3.3, and Figure 3.4 show runtime and accuracy at Top-1, Top-3, Top-5 and Top-10 respectively of existing fault localization families such as SBFL (DStar), Slicing, Stack trace, Predicate Switching, MBFL (Metallaxis) as presented in the Chapter 2. CombineFL is a FL technique that combines all other five families of fault localization techniques using learning to rank machine learning approach. We make use of the suspiciousness statements and suspiciousness scores from the dataset provided by Zou et al. [32]. Since we use source code lines in all our evaluations, we convert suspicious statements to lines and compute accuracy at Top-1, Top-3, Top-5 and Top-10 in our implementation. Based on the Figure 3.1, Figure 3.2, Figure 3.3, and Figure 3.4, we classify the existing fault localization families into two classes or clusters: fault localization techniques that have run time less than or equal to 1000 seconds and fault localization techniques that have run time greater than 1000 seconds.

Each class has its pros and cons. The first class of fault localization contains techniques

that take low run time (less than or equal to 1000 seconds). Developers who care about runtime can choose a fault localization technique from the first class. The second class of fault localization takes runtime greater than 1000 seconds. For instance, combineFL technique that combines all other standalone fault localization techniques takes the highest runtime of 5700 seconds (1 hour 35 minutes per bug) but provides the highest accuracy. So, CombineFL can be chosen if someone needs the highest accuracy but does not care about runtime.

From the existing fault localization techniques as shown in Figure 3.1, Figure 3.2, Figure 3.3, and Figure 3.4, we selected two baselines such as DStar and CombineFL. DStar has the highest accuracy in class 1 while CombineFL has the highest accuracy in class 2.

3.2 Combination of Fast FL techniques

We tried all possible combinations of DStar with other FL techniques in class 1 to find a technique that is the best among all combinations in class 1 using machine learning. We tried seven combinations such as DStar + Stack trace, DStar + Slicing, DStar + Predicate Switching, DStar + Stack trace + Predicate Switching, DStar + Slicing (freq) + Stack trace, DStar + Slicing + Predicate Switching, and DStar + Slicing + Stack-Trace + Predicate Switching. Each combination is plotted as a small x-mark symbol as shown in 3.5, Figure 3.6, Figure 3.7, and Figure 3.8. From all the combinations, we found DStar + Slicing + Stack-Trace + Predicate Switching as the best combination (named as CombineFastestFL) giving the highest overall accuracy (sum of accuracies at Top-1, Top-3, Top-5 and Top-10). We chose CombineFastestFL as another baseline.

3.3 Selection of 3 Baseline FL Techniques

We plotted all three baselines such as DStar (Baseline 1), CombineFastestFL (Baseline 2), and CombineFL (Baseline 3) along with the standalone FL techniques in the Figure 3.9, Figure 3.10, Figure 3.11, and Figure 3.12. The baseline FL techniques are highlighted in blue. DStar (Baseline 1) takes low runtime and low data while providing high accuracy. CombineFastestFL (Baseline 2) takes low runtime and provides high accuracy. CombineFL (Baseline 3) is the slowest technique but provides the highest accuracy. DStar and CombineFastestFL belong to the first class while CombineFL belongs to the second class.

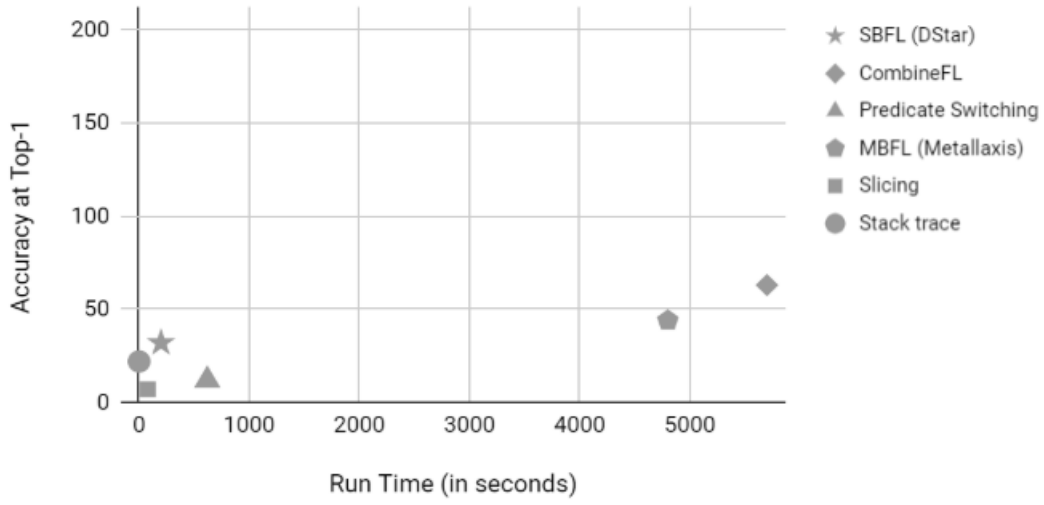

Figure 3.1: Runtime and Accuracy at Top-1 of Fault Localization Families

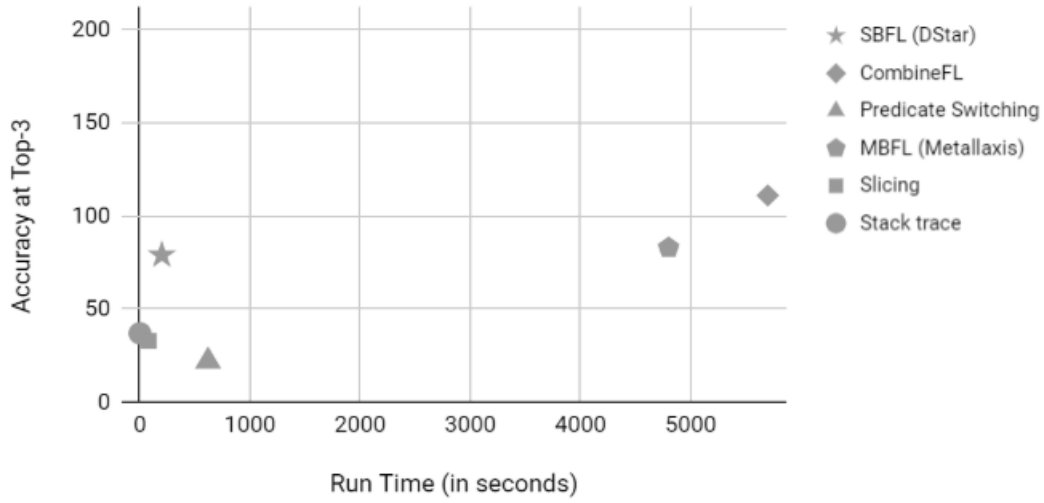

Figure 3.2: Runtime and Accuracy at Top-3 of Fault Localization Families

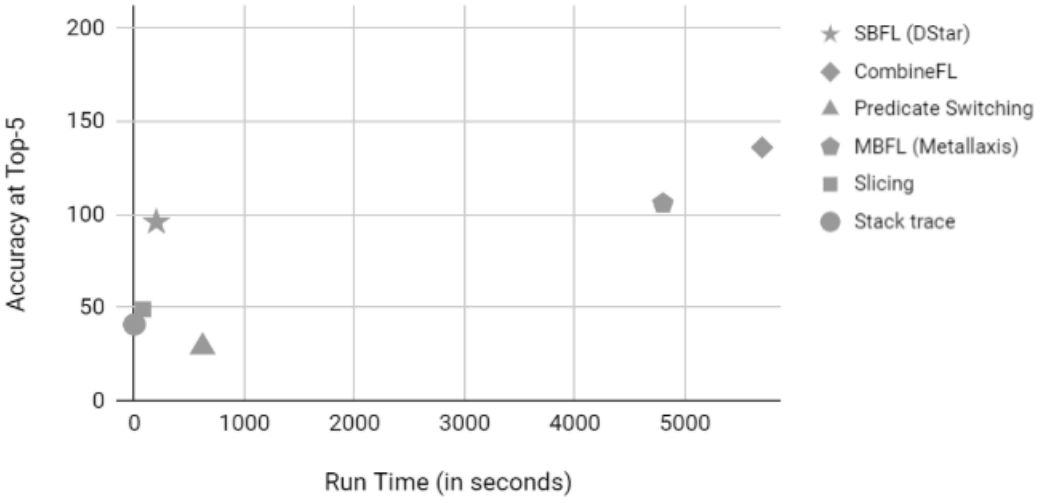

Figure 3.3: Runtime and Accuracy at Top-5 of Fault Localization Families

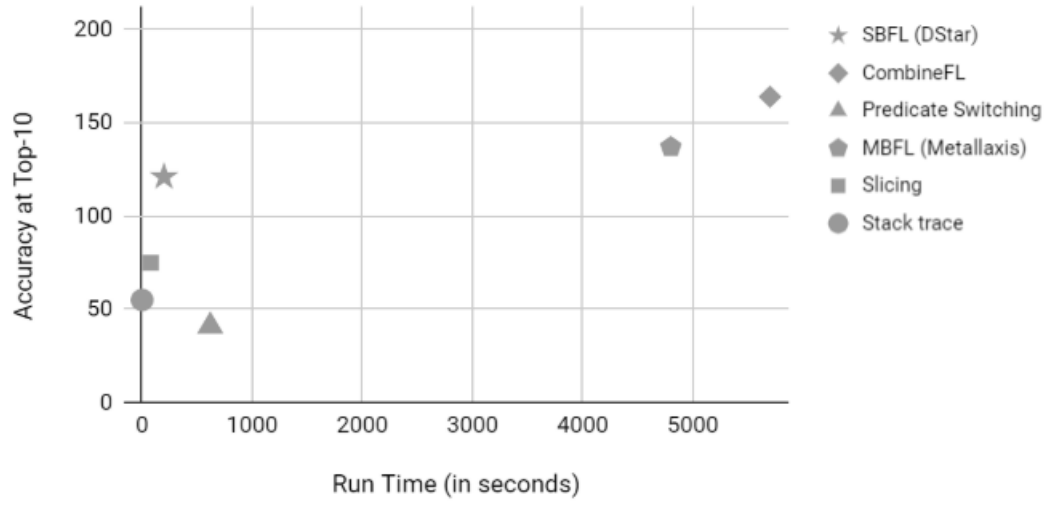

Figure 3.4: Runtime and Accuracy at Top-10 of Fault Localization Families

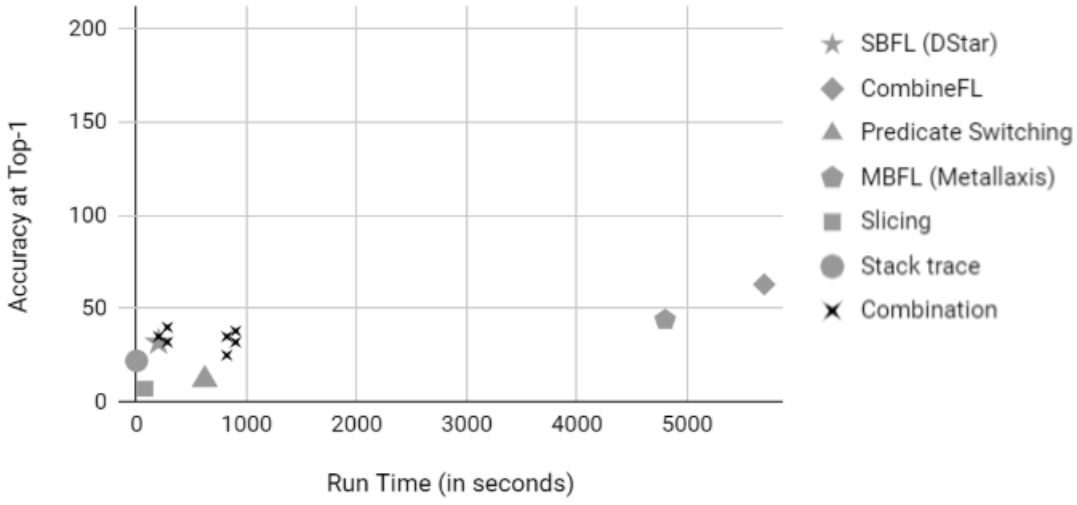

Figure 3.5: Runtime and Accuracy at Top-1 of Fault Localization Families with Combinations

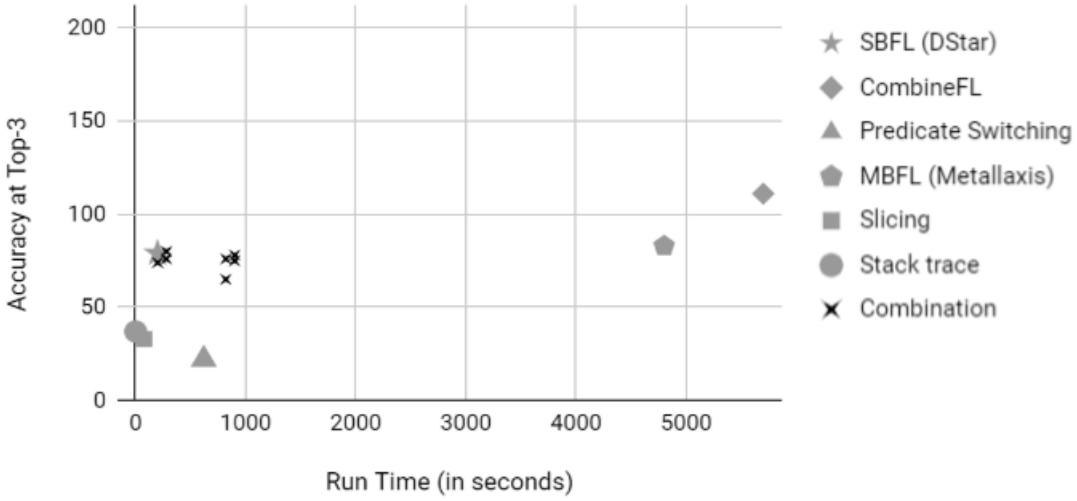

Figure 3.6: Runtime and Accuracy at Top-3 of Fault Localization Families with Combinations

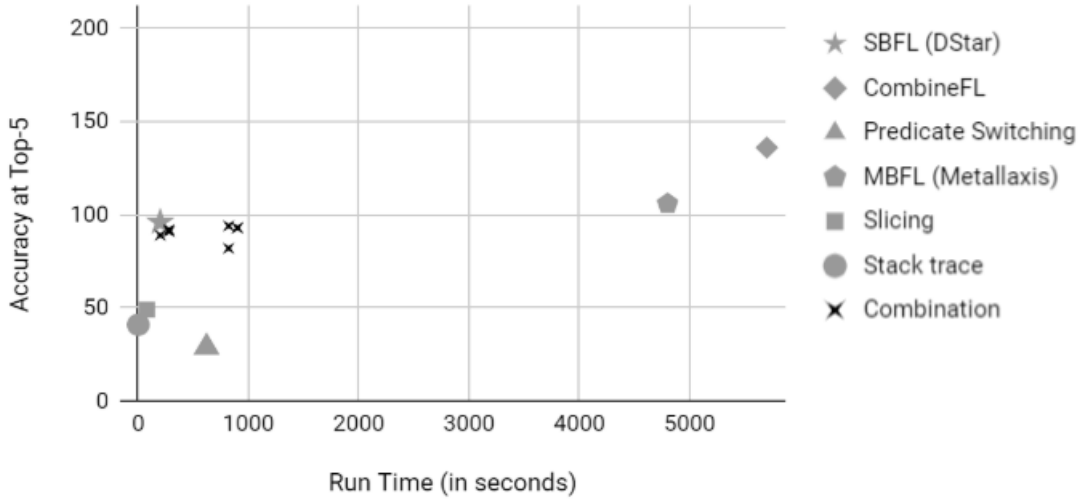

Figure 3.7: Runtime and Accuracy at Top-5 of Fault Localization Families with Combinations

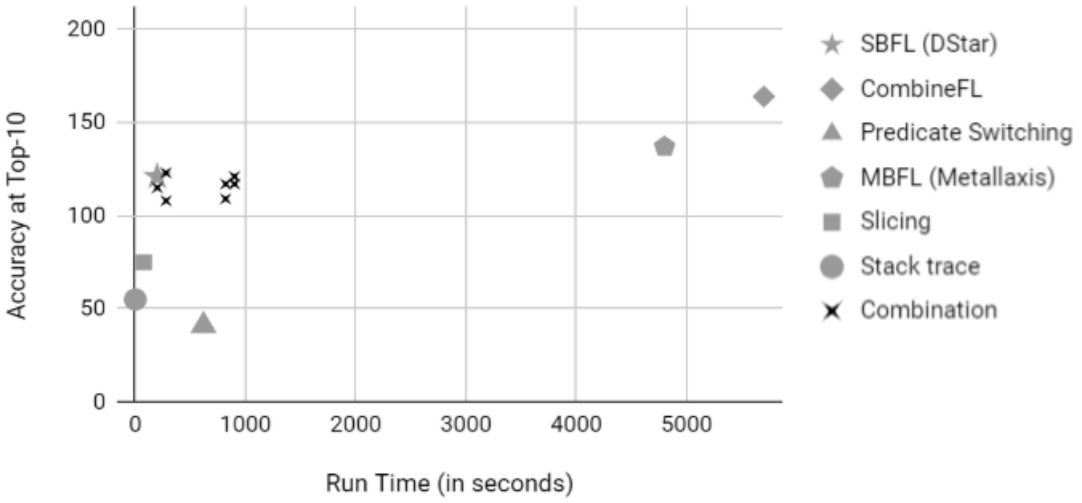

Figure 3.8: Runtime and Accuracy at Top-10 of Fault Localization Families with Combinations

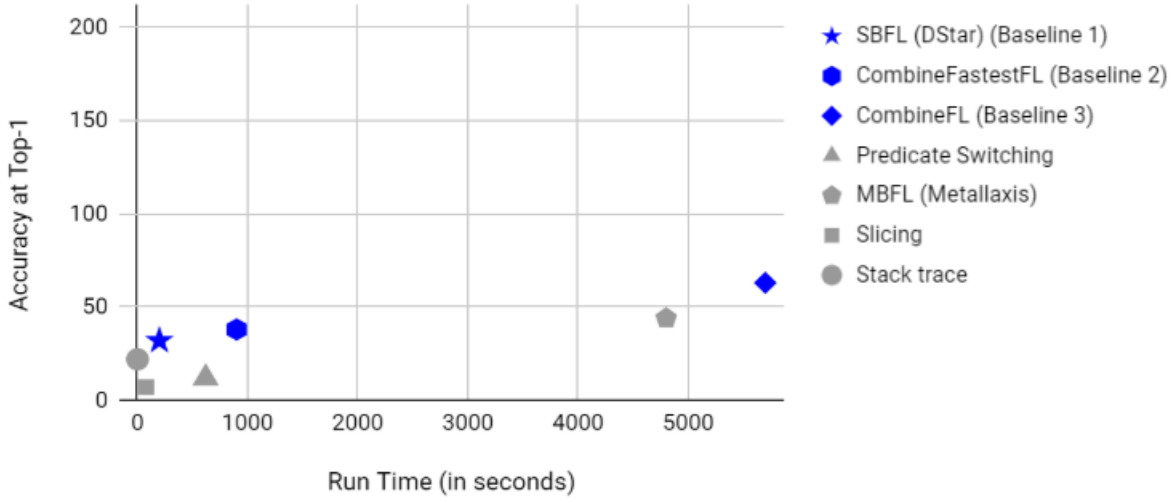

Figure 3.9: Runtime and Accuracy at Top-1 of Fault Localization Families with 3 Baselines

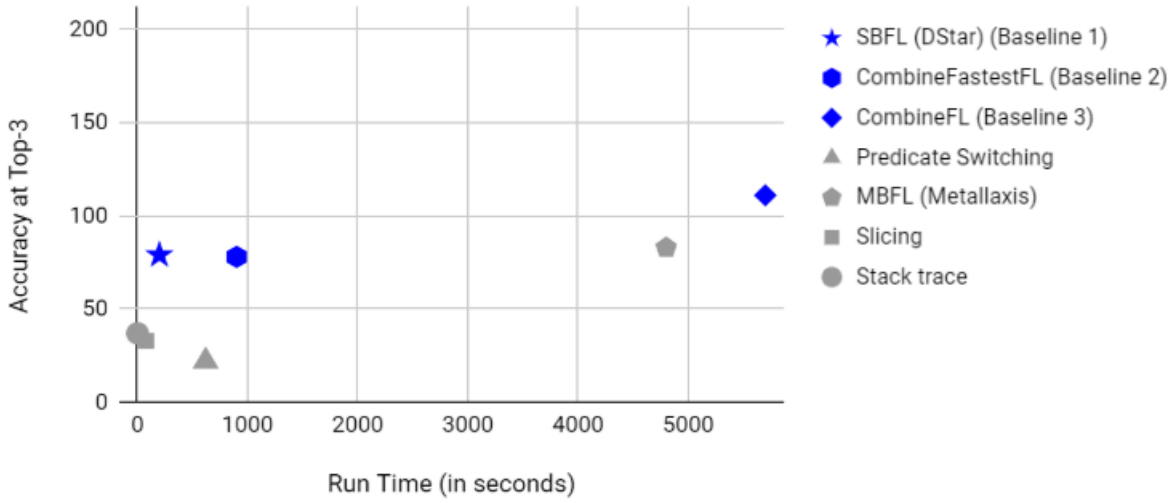

Figure 3.10: Runtime and Accuracy at Top-3 of Fault Localization Families with 3 Baselines

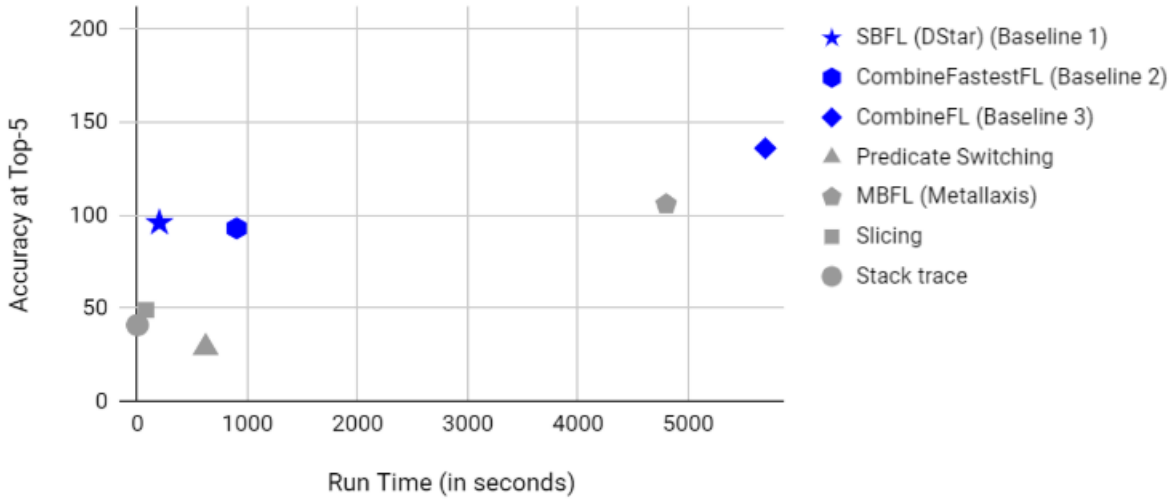

Figure 3.11: Runtime and Accuracy at Top-5 of Fault Localization Families with 3 Baselines

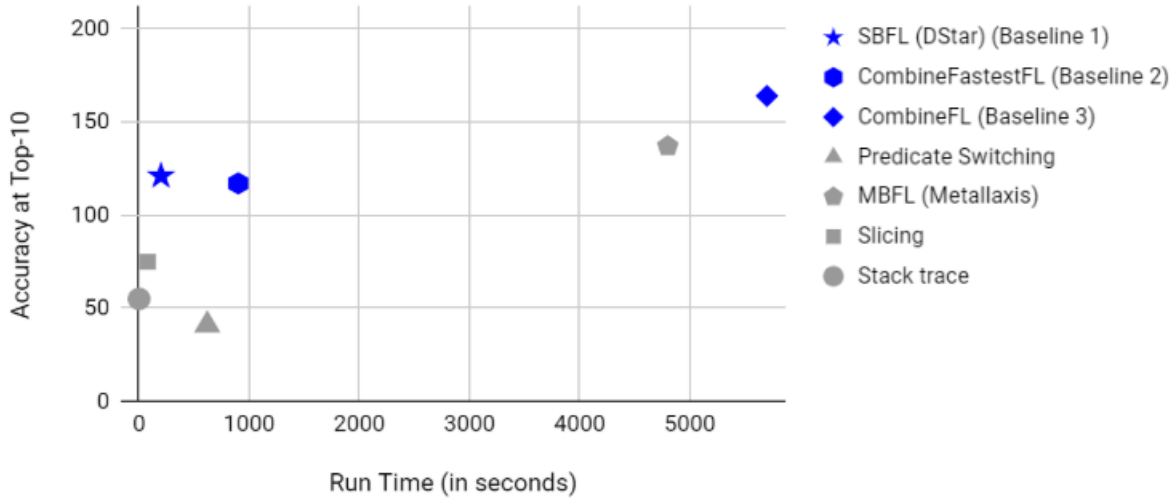

Figure 3.12: Runtime and Accuracy at Top-10 of Fault Localization Families with 3 Baselines

Chapter 4

Approach

We propose a novel cost-aware fault localization technique that improves the accuracy of baseline fault localization techniques identified in Chapter 3. Enhanced Cost-Aware Fault Localization (ECFL) involves four phases: run fault localization technique, extract source code features, combine by weighted sum and rank program elements, as shown in Figure 4.1. We explain each phase in detail.

4.1 Phase 1: Run Fault Localization Technique

After knowing the existing best fault localization baseline technique, take the suspiciousness scores of the program elements. Typically, the output of any fault localization technique is a set of ranked program elements (statement, line, method or files) ordered by suspiciousness scores. We use line level granularity throughout this work. Next, the suspiciousness scores are normalized using min-max. Further, the normalized suspiciousness scores are combined with the normalized features in phase 3.

4.2 Phase 2: Extract Source Code Features

In phase 2, top 100 suspicious lines from the results of phase 1 are chosen. Next, extract line level source code features that would incur a low runtime for all the top 100 lines. Normalize

the feature values using min-max normalization so that they fall in the range between 0 and 1 (including both 0 and 1). Normalization is done to perform a weighted sum in the next phase as different features might have a different range of values.

4.3 Phase 3: Combine by Weighted Sum

Once the normalized suspiciousness score and the normalized feature score for all top 100 suspiciousness lines are collected, combine them using the following weighted sum formula.

$$wt\_susp = (s\_wt * s\_score) + (f\_wt * f\_score)$$

where *wt_susp* stands for weighted suspiciousness, *s_wt* and *f_wt* stand for suspiciousness weight and feature weight respectively, *s_score* and *f_score* stand for suspiciousness score and feature score respectively.

We experiment different weights for susp_weight and feature_weight in Section 6.2 to find the best weights.

4.4 Phase 4: Rank Program Elements

Based on the newly computed weighted suspiciousness scores from the previous phase, sort the lines to generate the final output list which is used by the developers for debugging. For each source code line level feature extracted, results are computed.

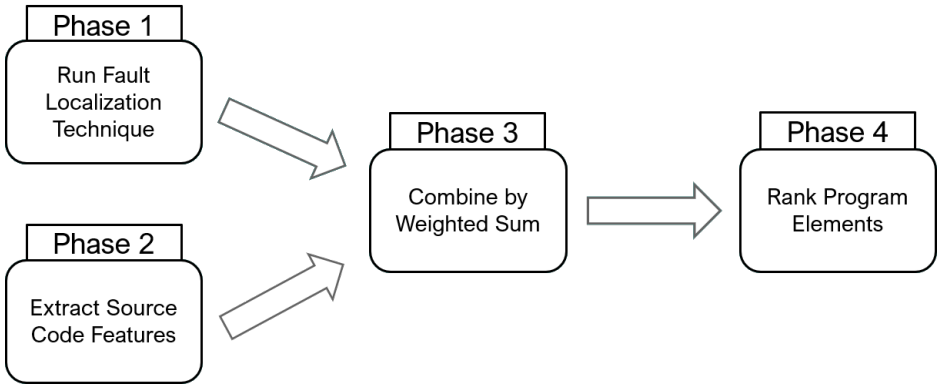

Figure 4.1: Four phases of ECFL technique

Chapter 5

Experimental Methodology

This chapter contains three research questions and common details about our experiments. We describe our experiments for each research question in their respective chapters.

5.1 Research Questions (RQ)

RQ-1: What feature of the ECFL technique would improve our selected fault localization baselines the highest?

This question helps to find a feature that will provide the best improvement in terms of accuracy among a set of features we study. We study three different line level features: line commit size, line recency, and line length. We want to find the best weights for each feature using the weighted sum ECFL approach. We chose these four features since they take a low runtime for extraction.

RQ-2: Would an ML approach improve the accuracy of our ECFL approach?

This question aims to compare the weighted sum (ECFL) approach with the ML approach to find which one is performing better in terms of accuracy.

RQ-3: Are the improvements in accuracy by the proposed FL technique consistent

across all subjects?

For this question, we aim to study how the improvement in accuracy varies across five different subjects as each subject might have different characteristics.

5.2 Subjects

We use Defects4J (Version 1.1.0) [8] dataset in our experiments. Defects4J is a database of real software bugs found in large open-source Java projects. The Defects4J dataset is used in many of the recent studies of fault localization techniques [32], [24], [21], [30]. We use 357 real bugs from five different Defects4J subjects, namely Google Closure Compiler, Apache Commons Lang, JFreeChart, Apache Commons Math, and Joda-Time, as shown in Table 5.1. Each bug in the Defects4J dataset has specific properties. For instance, bugs are fixed in a single commit and also fixed only by modifying the source code instead of changing documentation, test files, or configuration files.

Table 5.1: Defects4J Subjects.

Subjects	Identifier	Number of Bugs
Google Closure Compiler	Closure	133
Apache Commons Lang	Lang	65
JFreeChart	Chart	26
Apache Commons Math	Math	106
Joda-Time	Time	27
All	All	357

5.3 Ground Truth

For evaluation, we use the ground truth from the fault localization dataset provided by Pearson et al. [21]. The dataset contains actual faulty lines for each of the 357 bugs. Some of the faults are multi-line faults. Such faults are fixed by introducing changes in more than one line. We consider a fault as localized if at least one of the faulty lines is localized [21], [24], [32]. Some of the other faults are due to faults of omission. Faults of omission arise when there is no faulty line in the program, but the developer fixes the faults by adding new lines that were missing. For such faults, the developer's patch will contain only one possible location of placing those missing lines. However, there can be multiple correct locations to add the lines. In such cases, we consider all the possible candidate locations for that fault as correct locations as done by Pearson et al. [21].

5.4 Evaluation Metrics

We use top-N as the evaluation metric to evaluate ECFL technique. Absolute metrics are recommended for evaluation of fault localization techniques based on a user study by Parnin and Orso [20]. Further, recent research papers [32], [24], [21], [30] on fault localization use top-N as the evaluation metric, and we follow the same approach. Top-N means the fault is located in Nth rank of the fault localization output list. For example, Top-1 means how many faults out of total faults are located at rank 1. We use accuracy at Top-1, Top-3, Top-5, and Top-10 since 98% of the practitioners consider only top-10 lines of the fault localization technique when they debug as per the finding of a study by Kochhar et al. [10].

Chapter 6

Research Question 1 (RQ-1)

In this section, we explain the experimental method and the results for research question 1 (RQ-1) to devise an ECFL technique with a feature that gives the highest improvement in accuracy among the other considered features.

6.1 Method

We designed our technique to improve the accuracy of the existing baseline fault localization techniques with an addition of marginal cost (runtime and data). So, we decided to make use of source code line level metrics such as line commit size, line recency, and line length that would incur a low extraction cost in runtime. Prior research [6], [14], [16], [17] has shown correlation between source code metrics and defect prediction. We make use of source code line level features that are used in defect prediction for fault localization. We study how each feature contributes towards improving the accuracy of fault localization.

We make use of the data from the Fault localization data repository [1], [32] and [2] to get the suspiciousness lines for all the three baseline fault localization techniques such as DStar (Baseline), CombineFastestFL (Baseline 2) and CombineFL (Baseline 3). We consider only the top X (X = 100) suspiciousness lines based on our experiment results.

The following are the three features we used:

Feature 1: Line Commit Size (LineCommitSize):

Meaning: Size of the commit where the line was last modified.

Intuition: We hypothesize that if a line was added as part of a big commit, the likelihood of it being buggy might be more compared to a line that was added as a part of a smaller commit. The intuition behind this hypothesis is that humans sometimes tend to make mistakes while pushing large amount of code (large commit).

Computation: For each line, we use git commands to identify the commit id of the line and then find the size of the commit as measured by the number of additions made. We then normalize the values to 0-1 range using min-max normalization.

Feature 2: Line Recency (LineRecency):

Meaning: Recency denotes how recent the line was added or updated.

Intuition: We hypothesize that if a line was added newly or updated recently, the likelihood of it being buggy might be more compared to a line that was not updated recently. The intuition behind this hypothesis is that the probability of recent lines being buggy is more than old lines.

Computation: For each suspicious line, we found the date of last update or creation in the buggy version by using git blame command. From the extracted date of all the suspiciousness lines, we calculated the range of the dates for each bug (newest date and the oldest date). We mapped this range to a recency metric ranging between 0 and 1 for each bug using min-max normalization. A high value of recency (closer to 1) indicates that the line is recently added or updated while low recency denotes that the line is not recently updated.

Feature 3: Line Length (LineLength):

Meaning: Length of the line measured as the number of characters.

Intuition: We hypothesize that length of the line can be related to the complexity of the line. Many long lines have complex parts including long, complex formula, method calls

with a long list of arguments, a long list of predicates in a conditional line. On the other hand, many small lines have simple parts such as increment, decrement, simple if condition, and simple assignment. If a line is long, the likelihood of being faulty is more due to the associated complexity.

Computation: For each line, we compute the length of the line by measuring the number of characters in that line. Finally, the values are normalized using min-max normalization to 0-1 range.

As discussed in Chapter 4, for each feature, we combined the normalized suspiciousness scores of each baseline with the normalized feature scores to compute new weighted suspiciousness scores using the following formula.

$$wt\_susp = (s\_wt * s\_score) + (f\_wt * f\_score)$$

For finding the best feature and best weights, we experimented with the following weight pairs: (susp_weight, feature_weight) = (0.9, 0.1), (0.8, 0.2), (0.7, 0.3), (0.6, 0.4), (0.5, 0.5), (0.4, 0.6), (0.3, 0.7), (0.2, 0.8), and (0.1, 0.9). For each weight pair, we computed the accuracy at Top-1, Top-3, Top-5, and Top-10 for each feature using 177 random bugs. We chose 50% of random bugs from each of the five Defects4J subjects totaling 177 random bugs. For each baseline, we chose a different set of 177 random bugs. We used 50% of random bugs to show that our recommendation of best weights and best feature generalize to other datasets. Appendix A contains the RQ-1 results for all 357 bugs. To find a single best weight and best feature for each baseline, we picked the weight and the feature that provided the highest overall accuracy (sum of the accuracy of Top-1, Top-3, Top-5, and Top-10).

6.2 Results

For baseline 1 (DStar), the results of RQ-1 are shown in Figures 6.1, 6.2, 6.3 and 6.4. We show only the feature_weight in x-axis instead of showing weights pair (susp_weight, feature_weight). In the figures, we plotted the baseline 1 (DStar) as a straight dotted line in blue for reference to compare the results. From the mentioned Figures for baseline 1 results, we find that LineLength is the best feature across all Top-1, Top-3, Top-5 and Top-10 evaluation metrics. The best weight of LineLength is found as 0.2 for baseline 1.

For baseline 2 (CombineFastestFL), the results of RQ-1 are shown in Figures 6.5, 6.6, 6.7 and 6.8. We show only the feature_weight in x-axis instead of showing weights pair (susp_weight, feature_weight). In the figures, we plotted the baseline 2 (CombineFastestFL) as a straight dotted line in blue for reference to compare the results. From the mentioned Figures for baseline 2 results, we again find that LineLength is the best feature across all Top-1, Top-3, Top-5 and Top-10 evaluation metrics. The best weight of LineLength is found as 0.1 for baseline 2.

For baseline 3 (CombineFL), the results of RQ-1 are shown in Figures 6.9, 6.10, 6.11 and 6.12. We show only the feature_weight in x-axis instead of showing weights pair (susp_weight, feature_weight). In the figures, we plotted the baseline 3 (CombineFL) as a straight dotted line in blue for reference to compare the results. From the mentioned Figures for baseline 3 results, we again find that LineLength is the best feature across all Top-1, Top-3, Top-5 and Top-10 evaluation metrics. The best weight of LineLength is found as 0.1 for baseline 3.

We measured the run time of our ECFL technique in our server (32 GB RAM, 2.1 GHz CPU processor, single core) as 4.71 seconds per bug (rounded as 5 seconds per bug). The low run time shows that our ECFL technique adds only a little overhead while providing

improvement in accuracy of all the three baselines.

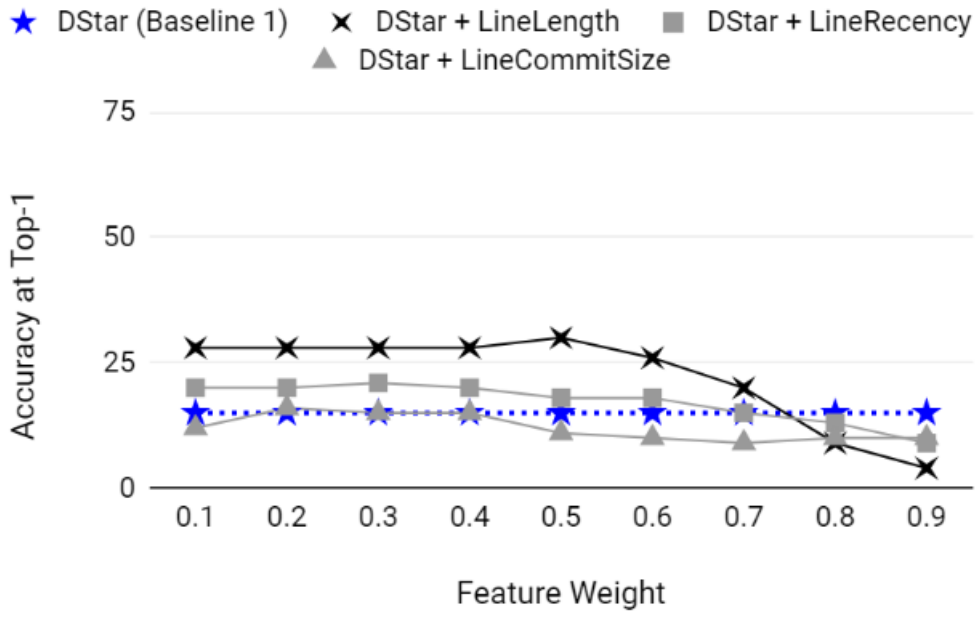

Figure 6.1: Comparison of Features with Baseline 1 using ECFL: Accuracy at Top-1

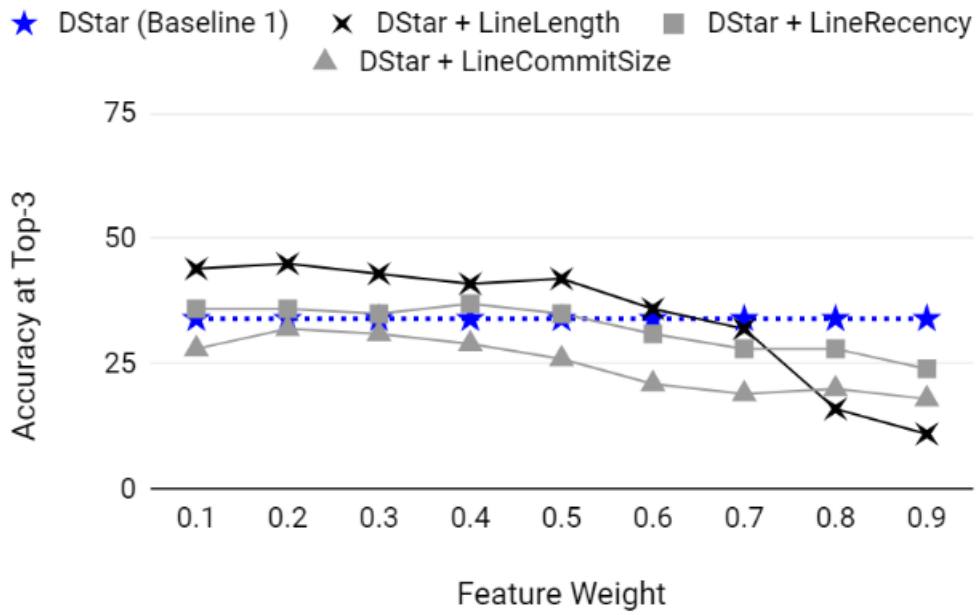

Figure 6.2: Comparison of Features with Baseline 1 using ECFL: Accuracy at Top-3

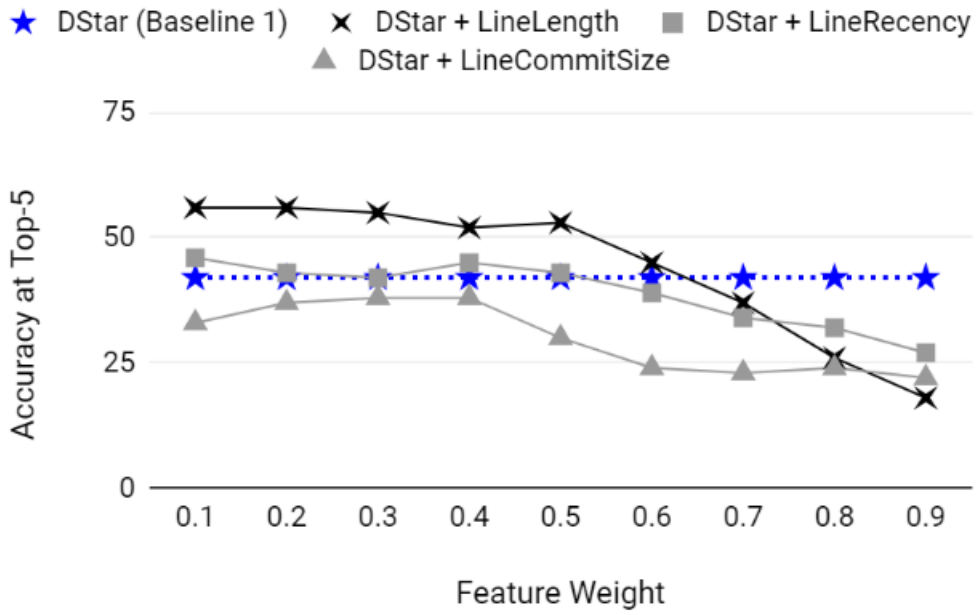

Figure 6.3: Comparison of Features with Baseline 1 using ECFL: Accuracy at Top-5

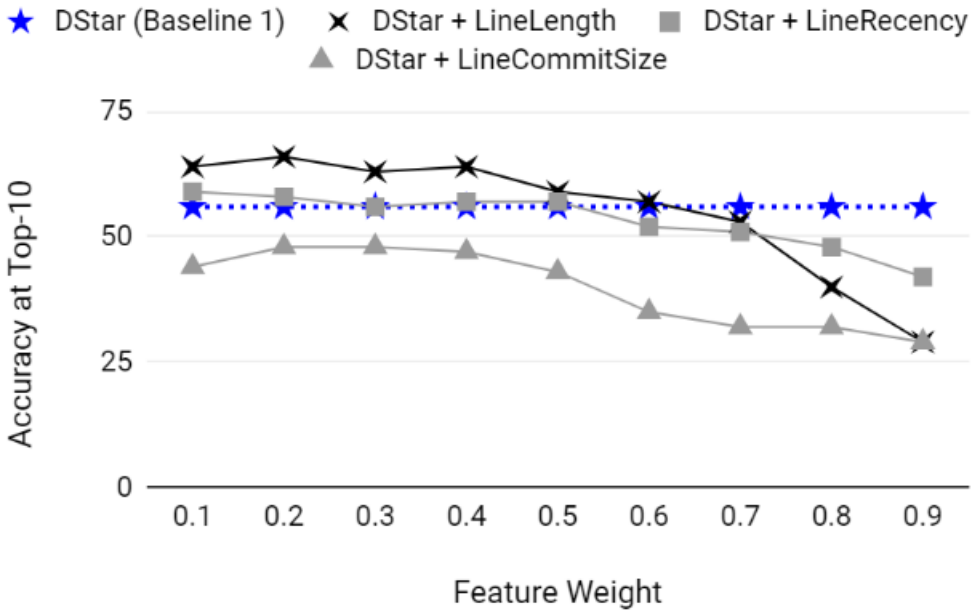

Figure 6.4: Comparison of Features with Baseline 1 using ECFL: Accuracy at Top-10

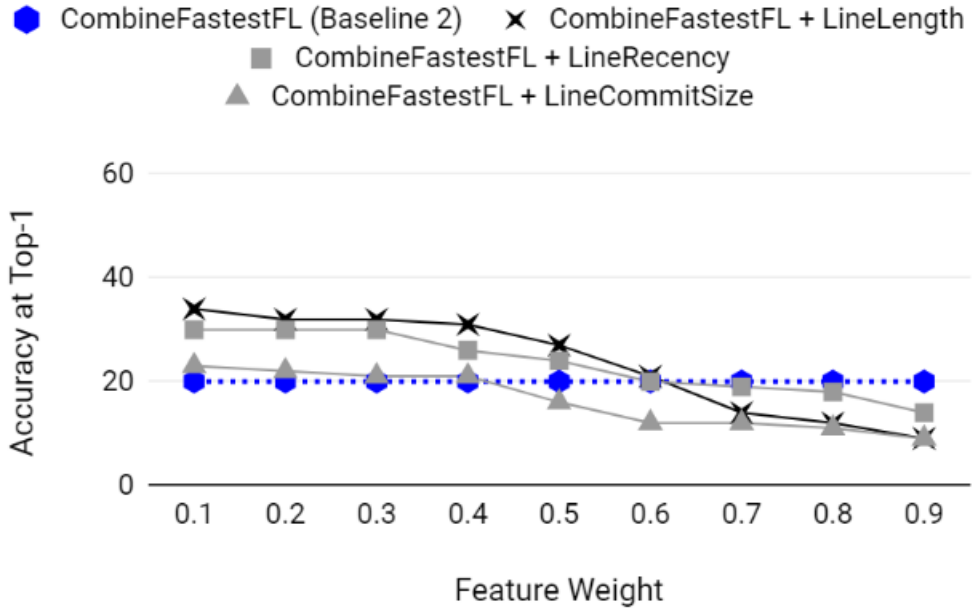

Figure 6.5: Comparison of Features with Baseline 2 using ECFL: Accuracy at Top-1

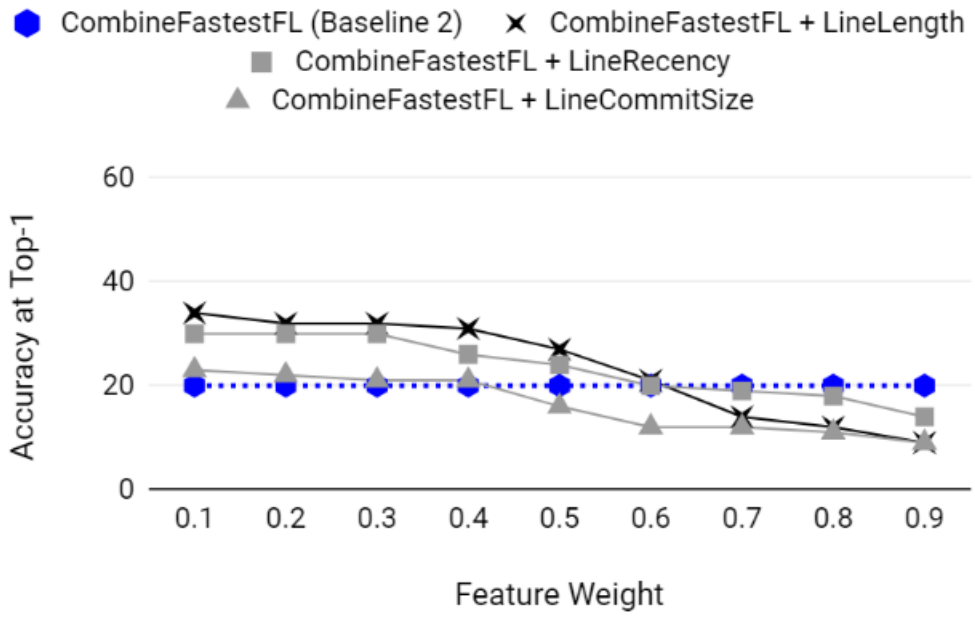

Figure 6.6: Comparison of Features with Baseline 2 using ECFL: Accuracy at Top-3

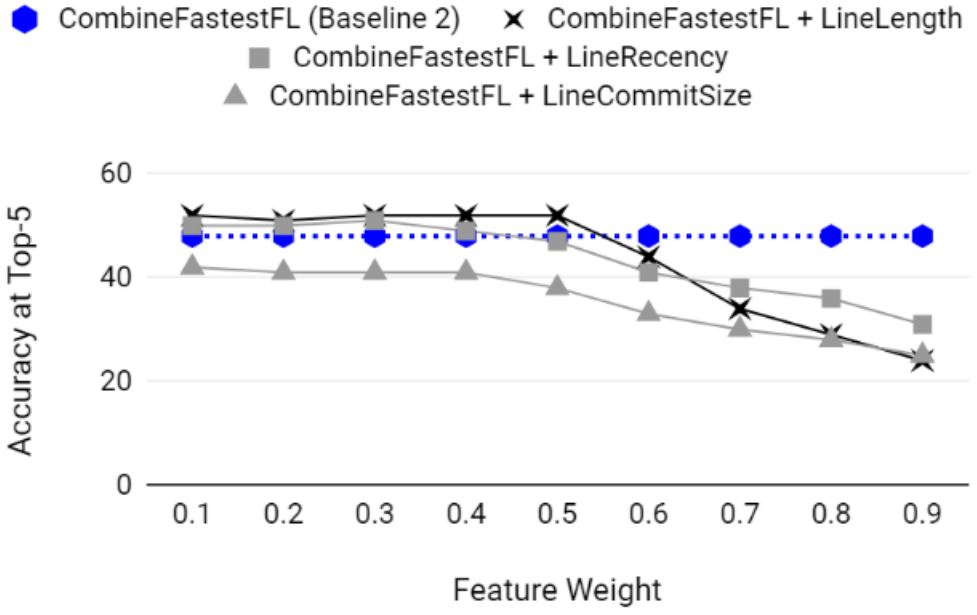

Figure 6.7: Comparison of Features with Baseline 2 using ECFL: Accuracy at Top-5

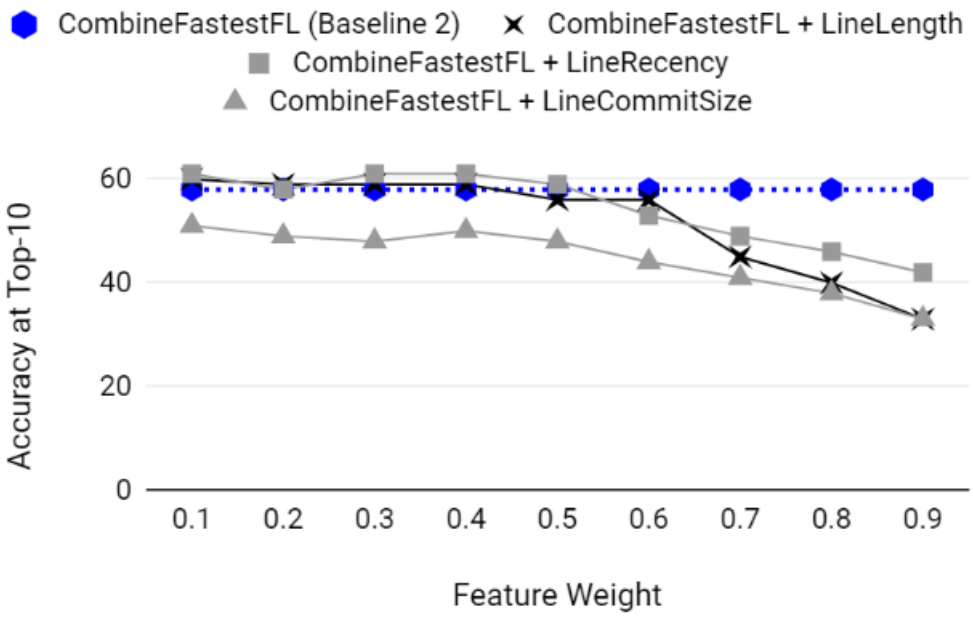

Figure 6.8: Comparison of Features with Baseline 2 using ECFL: Accuracy at Top-10

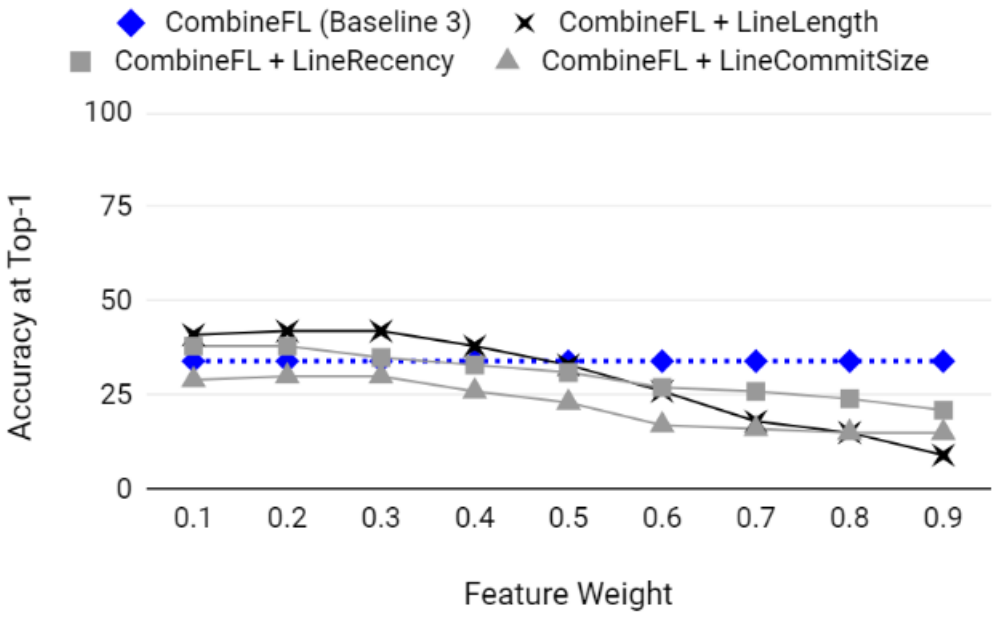

Figure 6.9: Comparison of Features with Baseline 3 using ECFL: Accuracy at Top-1

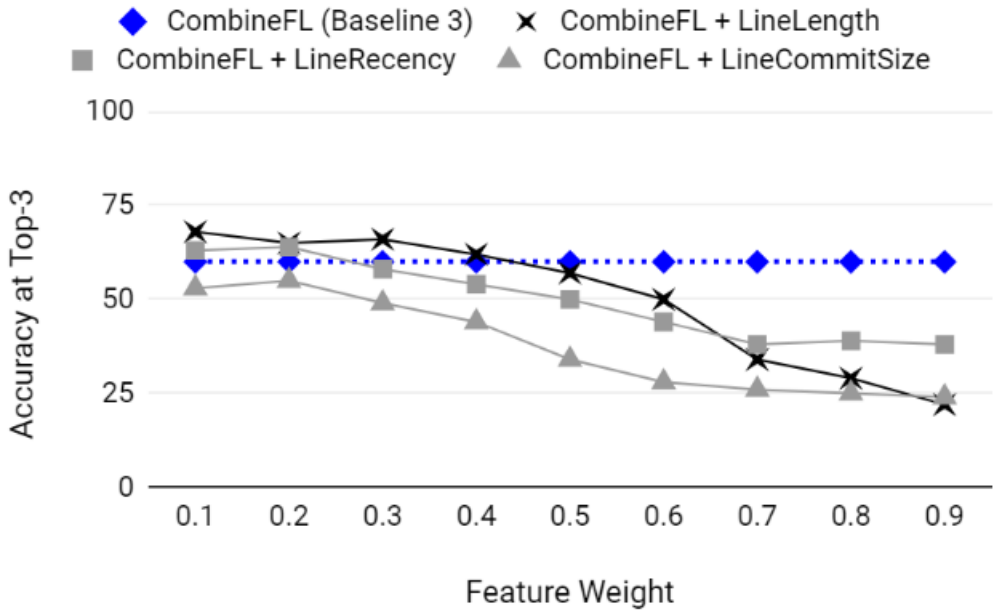

Figure 6.10: Comparison of Features with Baseline 3 using ECFL: Accuracy at Top-3

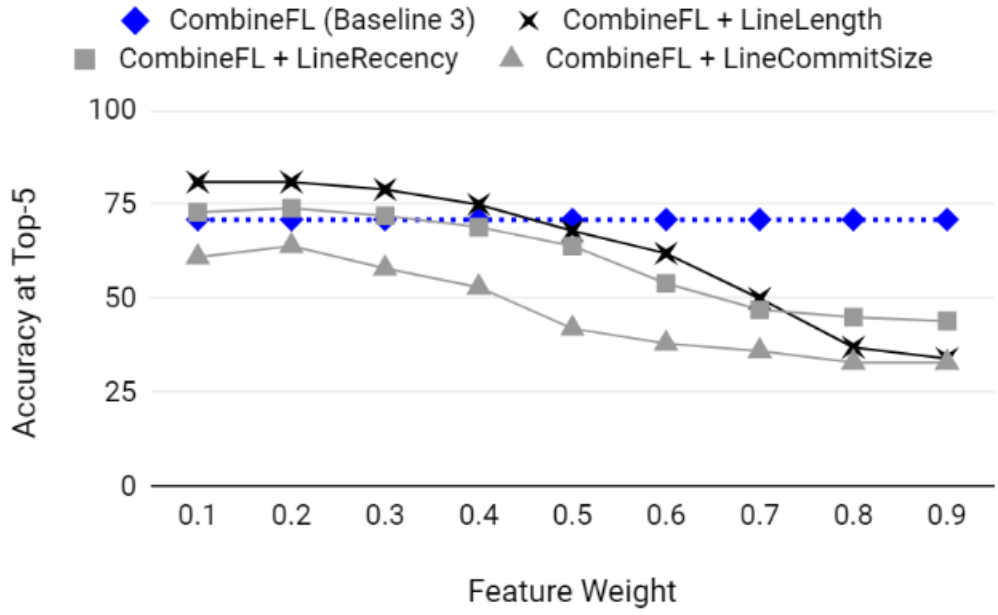

Figure 6.11: Comparison of Features with Baseline 3 using ECFL: Accuracy at Top-5

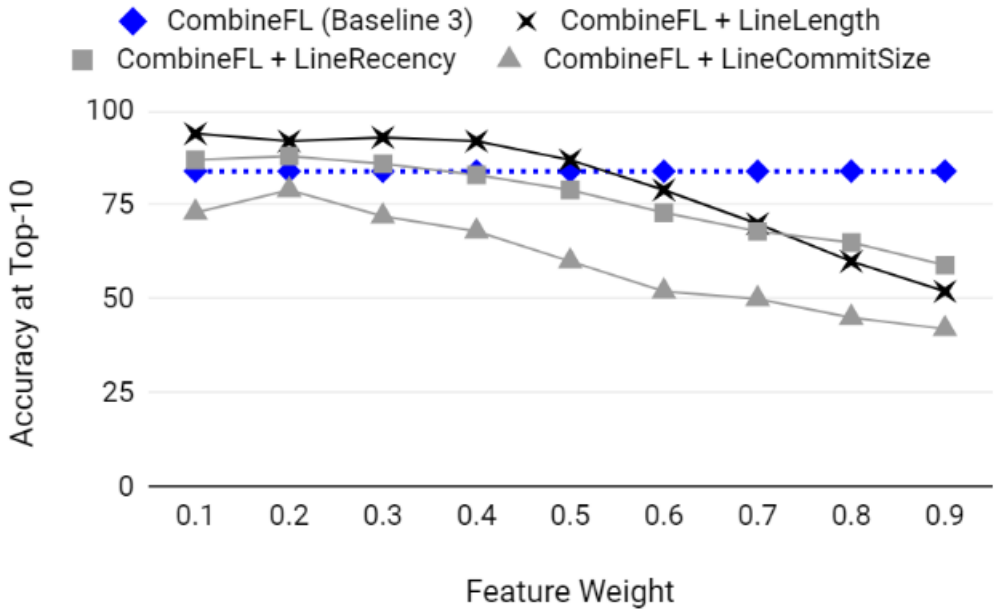

Figure 6.12: Comparison of Features with Baseline 3 using ECFL: Accuracy at Top-10

Chapter 7

Research Question 2 (RQ-2)

In this chapter, we explain the experimental method and the results for research question 2 (RQ-2) to compare whether the ECFL approach is better than the Machine Learning (ML) approach.

7.1 Method

To answer RQ-2, we combine each source code line level feature as a new feature with the three baseline techniques using the learning to rank machine learning approach. We make use of the combineFL machine learning infrastructure provided by Zou et al. [32], [2]. Since we use source code lines in ECFL evaluations, we convert suspicious statements to lines and compute accuracy at Top-1, Top-3, Top-5, and Top-10 using our implementation.

7.2 Results

Table 7.1 shows the run time (per bug in seconds) along with the Accuracy at Top-1, Top-3, Top-5, Top-10 and sum of Top-1, Top-3, Top-5 and Top-10. The source code line level features such as LineLength, LineRecency, LineCommitSize are combined with each baseline using the ML approach. We also included the best ECFL (weighted sum) approach found in Research Question RQ-2 6.2 for comparison with the ML results. Based on the last column

	Fault Localization Technique	Run Time per bug (in seconds)	Acc @ Top-1	Acc @ Top-3	Acc @ Top-5	Acc @ Top-10	Sum of Top-1, 3, 5, 10
Baseline 1	DStar	200	32	79	96	121	328
	DStar + LineLength (ML)	205	46	80	95	111	332
	DStar + LineRecency (ML)	205	38	71	90	112	311
	DStar + LineCommitSize (ML)	205	29	57	65	87	238
	DStar + 3 Features (ML)	205	46	77	95	112	330
	DStar + LineLength (Weighted Sum)	205	58	95	116	135	404
Baseline 2	CombineFastestFL (ML)	901	38	78	93	117	906
	CombineFastestFL + LineLength (ML)	906	52	87	104	131	374
	CombineFastestFL + LineRecency (ML)	906	52	82	103	128	365
	CombineFastestFL + LineCommitSize (ML)	906	52	84	105	135	376
	CombineFastestFL + 3 Features (ML)	906	54	87	106	127	374
	CombineFastestFL + LineLength (Weighted Sum)	906	60	91	107	129	387
Baseline 3	CombineFL (ML)	5700	63	111	136	164	474
	CombineFL + LineLength (ML)	5705	71	119	148	179	517
	CombineFL + LineRecency (ML)	5705	70	117	148	178	513
	CombineFL + LineCommitSize (ML)	5705	75	126	154	184	539
	CombineFL + 3 Features (ML)	5705	73	124	154	184	535
	CombineFL + LineLength (Weighted Sum)	5705	82	127	153	180	542

Table 7.1: Comparison of ML and ECFL results

of the Table (Sum of Top-1, Top-3, Top-5, and Top-10), Weighted sum (ECFL) performs better than ML combinations in terms of overall accuracy. Considering only ML results, for baseline 1, DStar + LineLength (ML) has the highest accuracy. For baseline 2 and baseline 3, DStar + LineCommitSize (ML) has the highest accuracy. The best fault localization techniques are bolded in Table 7.1 for reference.

Figures 7.1, 7.2, 7.3 and 7.4 shows the accuracy at Top-1, Top-3, Top-5 and Top-10 respectively for all the FL techniques provided in the Table 7.1. Extraction of source code features (LineLength, LineRecency, and LineCommitSize) take the same run time (5 seconds per bug). But, 100 seconds additional time (jitter) is added in the Figures to make it visible for the readers. Baseline FL techniques are colored as blue to differentiate from other FL techniques. The best weighted sum technique is also included for comparison. ML approach performs better than the weighted sum approach in a few cases. For instance, the ML technique has higher accuracy than the weighted sum for baseline 2 and baseline 3 in Top-10, as shown in Figure 7.4. From all the four Figures, we find that the ECFL (weighted sum) approach has the best accuracy in terms of the overall sum of the accuracy.

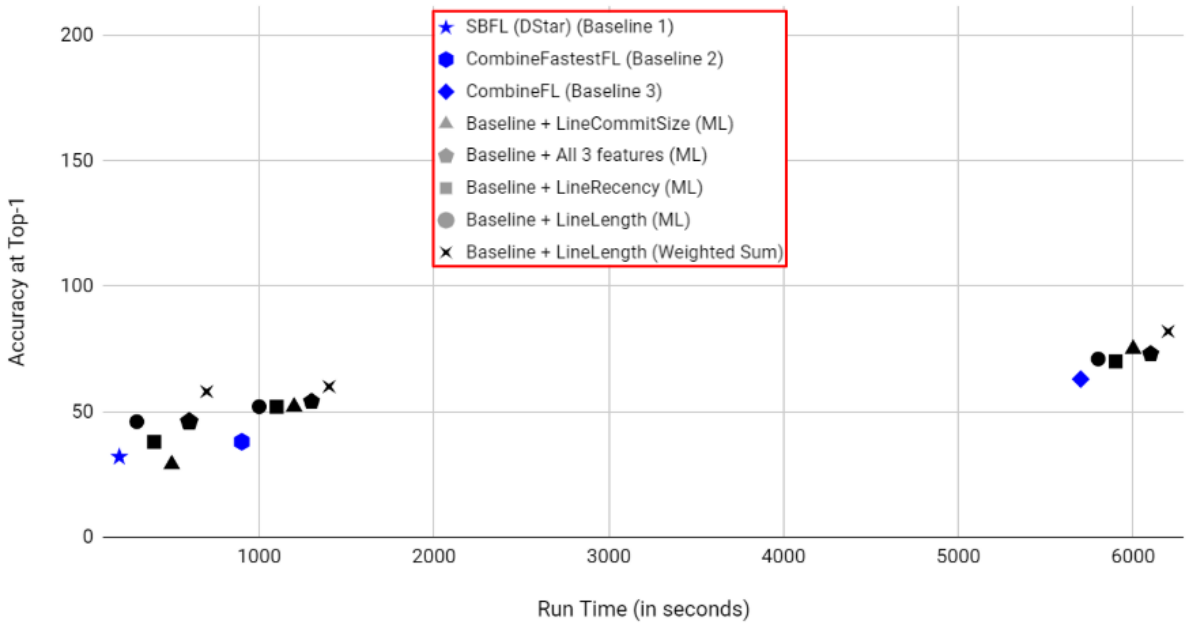

Figure 7.1: Comparison of Features with Baselines using ML: Accuracy at Top-1

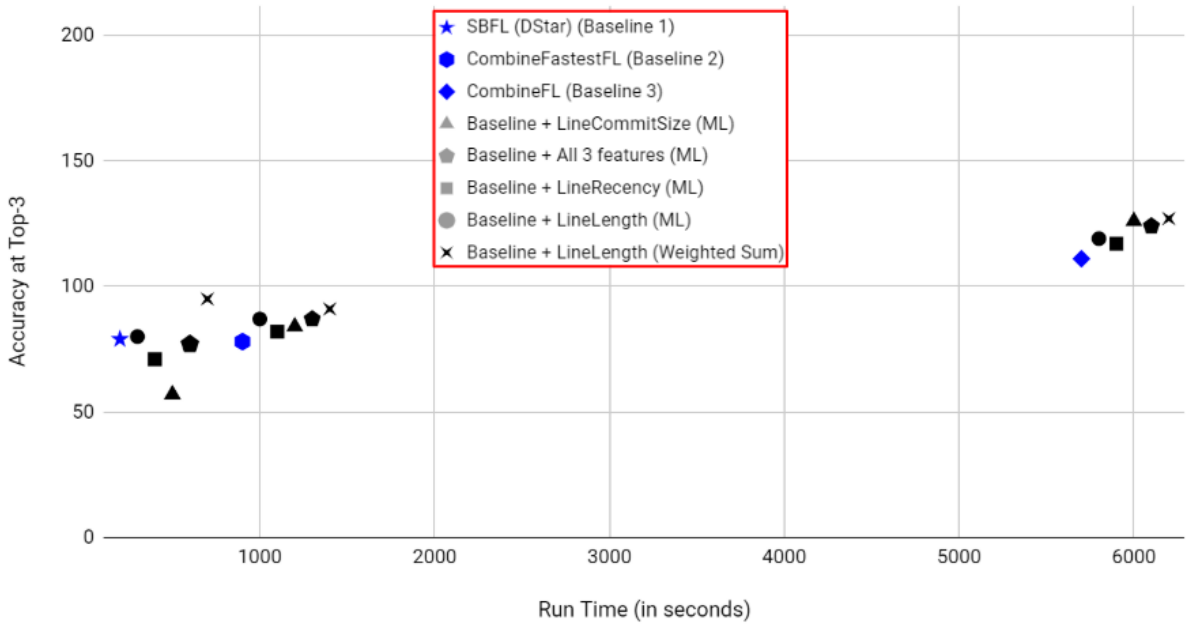

Figure 7.2: Comparison of Features with Baselines using ML: Accuracy at Top-3

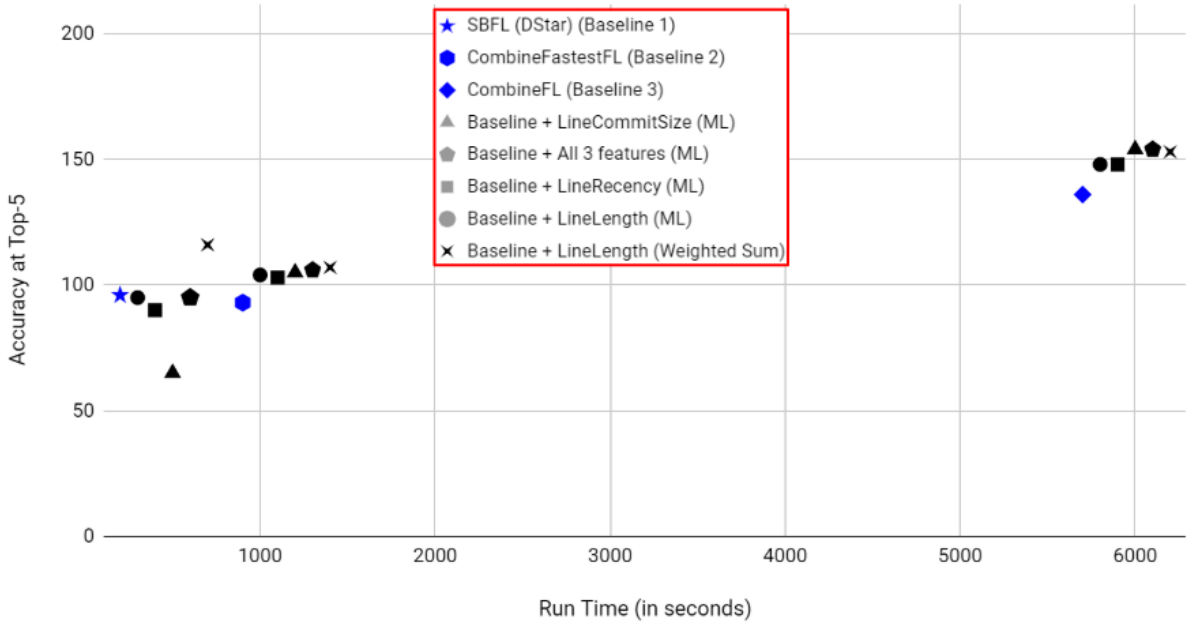

Figure 7.3: Comparison of Features with Baselines using ML: Accuracy at Top-5

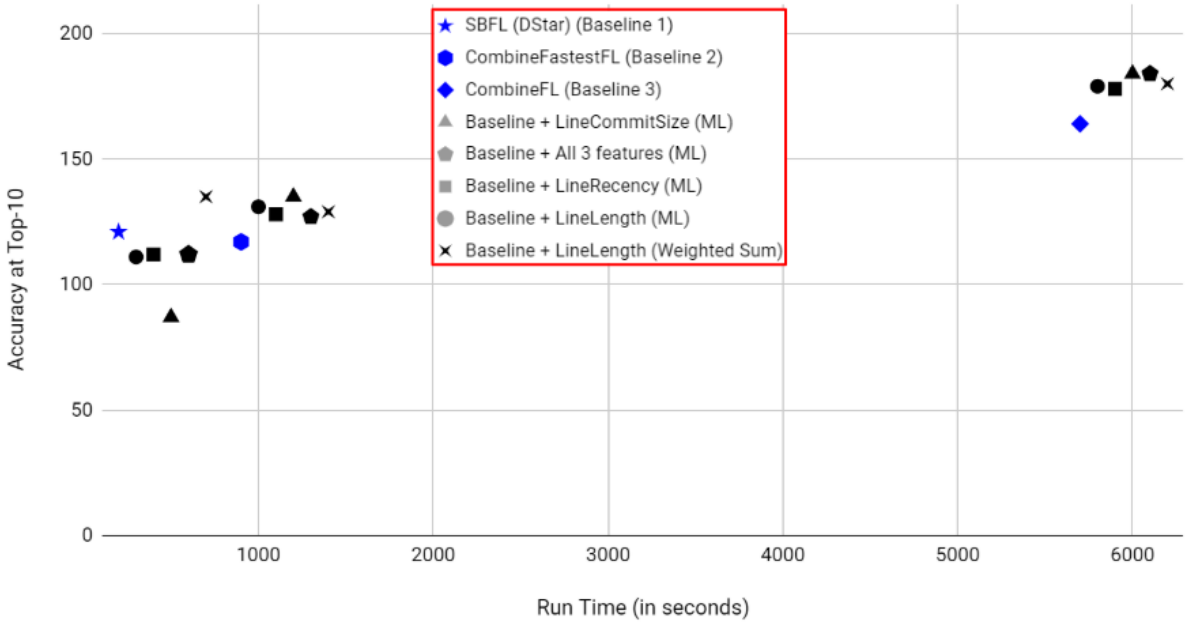

Figure 7.4: Comparison of Features with Baselines using ML: Accuracy at Top-10

Chapter 8

Research Question 3 (RQ-3)

In this chapter, we present the experimental method and the results for research question 3 (RQ-3) to find how the improvements of ECFL weighted sum and ML FL techniques vary over Defects4J subjects.

8.1 Method

To answer RQ-3, we computed accuracy at Top-1, Top-3, Top-5, and Top-10 for each of the five Defects4J subjects for the best ECFL (weighted sum) and ML technique for all three baselines. The five Defects4J subjects are Closure, Lang, Chart, Math, and Time as shown in Table 5.1 in Chapter 5.

8.2 Results

Figure 8.1 shows the results for Closure subject. All four evaluation results, such as Top-1, Top-3, Top-5, and Top-10, are shown in the same Figure. For each baseline, there are three bars: one for baseline, one for the best Weighted Sum (ECFL) and one for ML FL technique as shown in the Figure. There are three baselines and totally nine bars for each Top-X (X = 1, 3, 5, and 10). Based on overall accuracy, both Weighted Sum and ML techniques outperform baseline 2 and 3 for Closure subject. Weighted sum outperforms baseline 1, while

the ML technique did not outperform baseline 1.

For subjects such as Lang (Figure 8.2), Chart (Figure 8.3), and Math (Figure 8.4), both Weighted Sum and ML techniques outperform all three baselines. For Time Subject, as shown in Figure 8.5, both weighted sum and ML techniques outperform baseline 2 and 3. Both the weighted sum and ML technique did not exceed baseline 1 (DStar) for Time subject, as shown in Figure 8.5.

In summary, the ECFL (weighted sum) technique outperformed all three baselines in four out of five subjects, while the ML technique outperformed all three baselines in three out of five subjects.

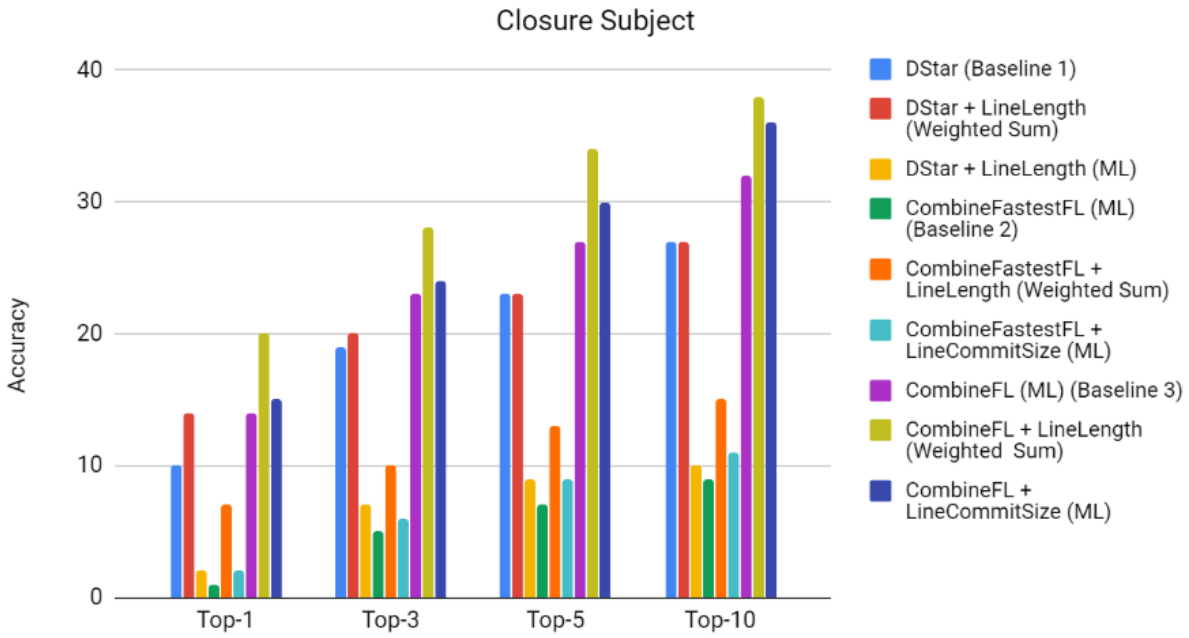

Figure 8.1: Accuracy of Defects4J Subject : Closure

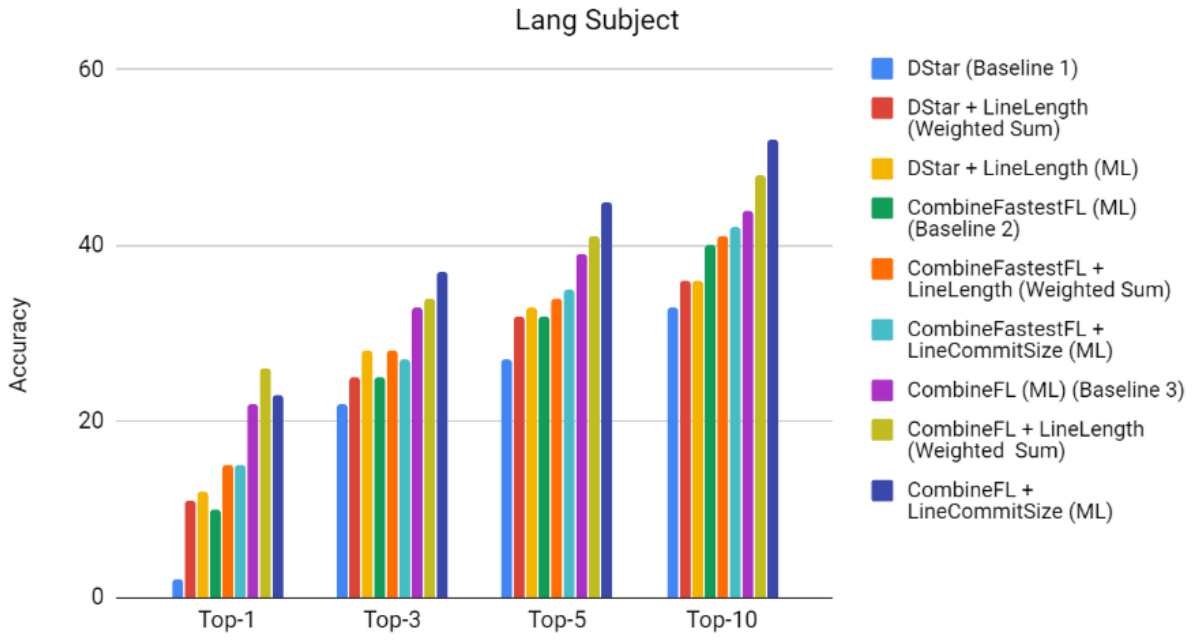

Figure 8.2: Accuracy of Defects4J Subject : Lang

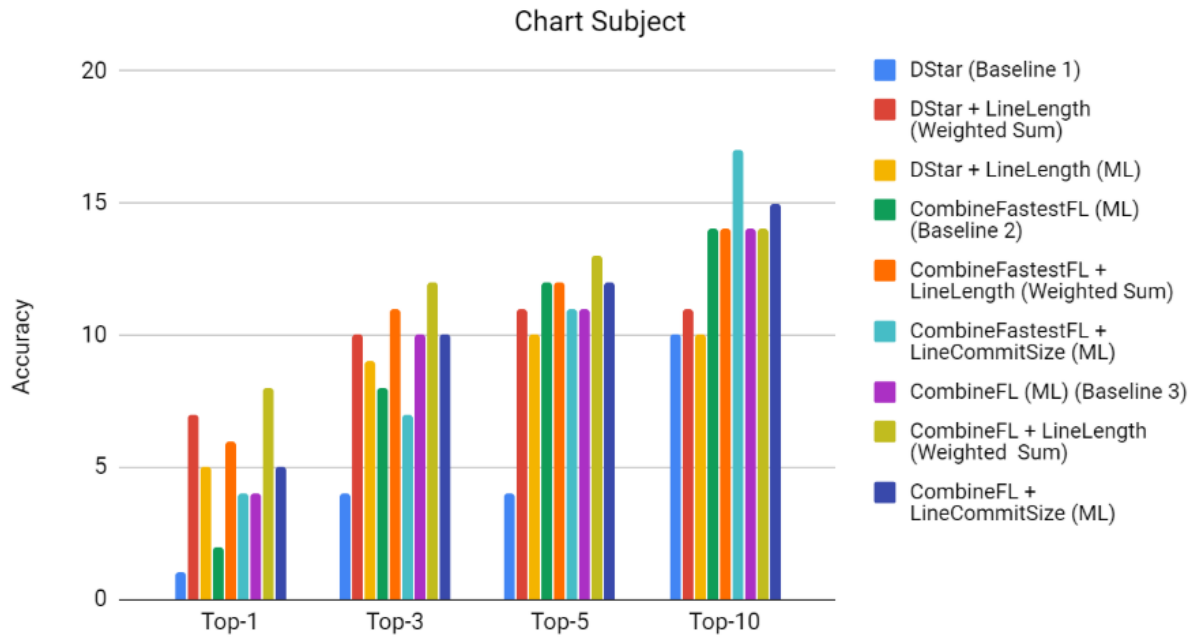

Figure 8.3: Accuracy of Defects4J Subject : Chart

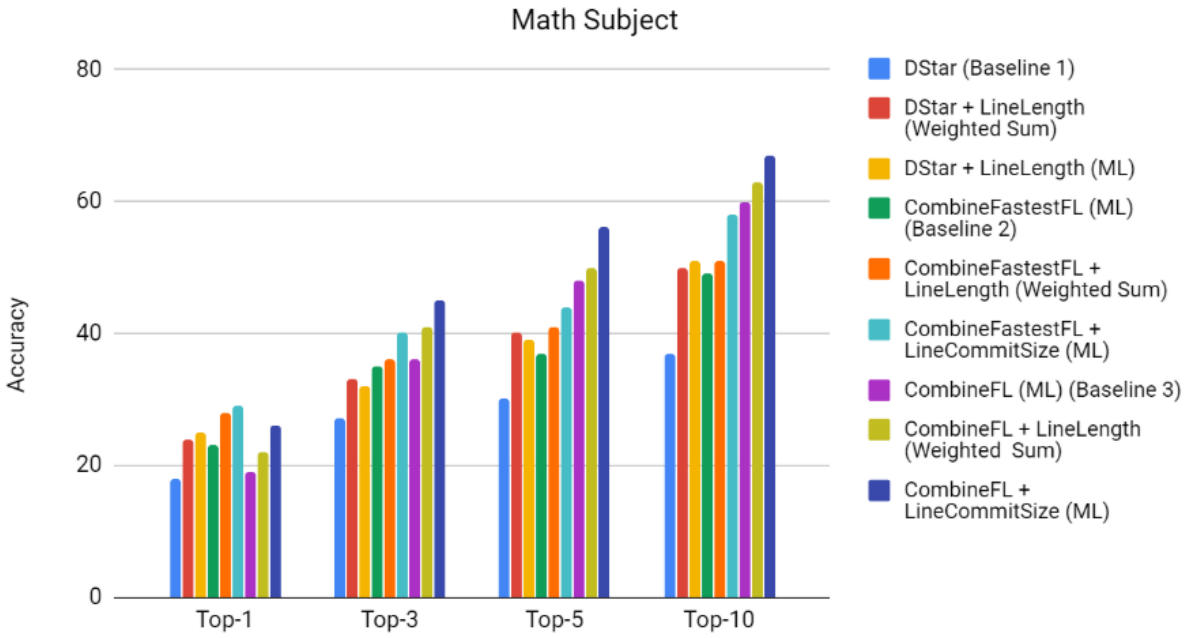

Figure 8.4: Accuracy of Defects4J Subject : Math

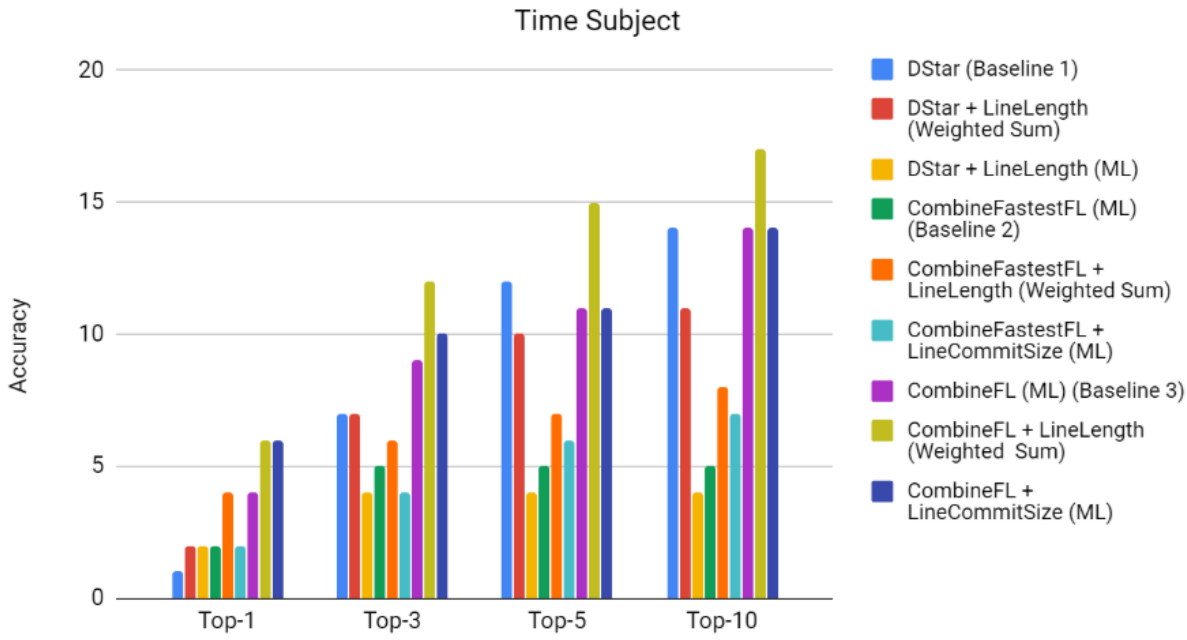

Figure 8.5: Accuracy of Defects4J Subject : Time

Chapter 9

Discussion

9.1 Results and Recommendations

Table 9.1 presents the recommendation of Fault Localization techniques based on different usage scenarios considering the trade-off between data needed, run time, and accuracy. If an emphasis is laid on data required and total runtime, DStar + LineLength (weighted sum) technique could be chosen since it requires only line length feature and test execution information. If runtime and accuracy is more important than data required, CombineFastestFL + LineLength (weighted sum) technique could be chosen. CombineFastestFL + LineLength (weighted sum) combines all the fast FL techniques (run time $<$ 1000 seconds per bug) along with LineLength and provides higher overall accuracy than DStar + LineLength but requires more data than DStar + LineLength.

On the other hand, if only accuracy is crucial, CombineFL + LineLength (weighted sum) technique could be selected instead. CombineFL + LineLength (weighted sum) combines all the five FL techniques along with LineLength and provides the highest overall accuracy among all other FL techniques.

What do you care?	Top-N	Best Fault Localization Technique
Data needed (low) + Time (low)	Top-1	DStar + LineLength (Weighted Sum : 0.8, 0.2)
Data needed (low) + Time (low)	Top-3	DStar + LineLength (Weighted Sum : 0.8, 0.2)
Data needed (low) + Time (low)	Top-5	DStar + LineLength (Weighted Sum : 0.8, 0.2)
Data needed (low) + Time (low)	Top-10	DStar + LineLength (Weighted Sum : 0.8, 0.2)
Data needed (low) + Time (low)	Overall	DStar + LineLength (Weighted Sum : 0.8, 0.2)
Time (low) + Accuracy (high)	Top-1	CombineFastestFL + LineLength (Weighted Sum: 0.9, 0.1)
Time (low) + Accuracy (high)	Top-3	CombineFastestFL + LineLength (Weighted Sum: 0.9, 0.1)
Time (low) + Accuracy (high)	Top-5	CombineFastestFL + LineLength (Weighted Sum: 0.9, 0.1)
Time (low) + Accuracy (high)	Top-10	CombineFastestFL + LineCommitSize (ML)
Time (low) + Accuracy (high)	Overall	CombineFastestFL + LineLength (Weighted Sum: 0.9, 0.1)
Accuracy (highest)	Top-1	CombineFL + LineLength (Weighted Sum: 0.9, 0.1)
Accuracy (highest)	Top-3	CombineFL + LineLength (Weighted Sum: 0.9, 0.1)
Accuracy (highest)	Top-5	CombineFL + LineCommitSize (ML)
Accuracy (highest)	Top-10	CombineFL + LineCommitSize (ML)
Accuracy (highest)	Overall	CombineFL + LineLength (Weighted Sum: 0.9, 0.1)

Table 9.1: Recommendation of FL techniques for different usage scenarios

9.2 Infrastructure for Future Research

To facilitate future research on fault localization, infrastructure resources and ECFL implementation is provided. Using our ECFL infrastructure, researchers can extract different source code line level features and combine them with other fault localization techniques. In addition to source code line features, source code history features can also be extracted since our implementation contains a history slicing algorithm [23]. History slicing traverses back in history one commit at a time. The ECFL infrastructure is designed to be modular and it is well documented; researchers can extend the work on ECFL to build and refine new techniques.

Chapter 10

Related Work

10.1 Standalone Fault Localization Techniques

Spectrum-based fault localization (SBFL) is a widely studied fault localization technique. Many SBFL techniques have been proposed including Tarantula [7], Ochiai [3], and DStar [26]. Mutation-based fault localization include MUSE [15] and Metallaxis [19]. Other stand alone fault localization techniques include dynamic slicing [4], stack trace [32], predicate switching [31], information retrieval based fault localization [22] and history-based fault localization [9].

10.2 Combination of Fault Localization Techniques

Laghari et al. [11] proposed a technique named Program Spectrum Analysis (PSA), a variant of spectrum-based fault localization technique that uses patterns of method calls to improve the fault localization for continuous integration scenario. Laghari et al. proposed another similar variant named Sequenced Spectrum Analysis (SSA) [12] that leverages the sequence mining to make use of the dependencies between methods that cause the fault.

Zhang et al. [30] proposed a technique to boost the SBFL using the PageRank algorithm. The idea behind their approach is to consider the contribution of different test cases and

assign different weights using the PageRank algorithm. Our technique ECFL is complementary to the technique proposed by Zhang et al. We believe that combining both techniques will give better accuracy.

Pearson et al. [21] combined SBFL and MBFL to improve the fault localization. They proposed a hybrid technique called MCBFL (Mutant and Coverage Based Fault Localization) that takes the average of MBFL suspiciousness scores for different mutants and the SBFL suspiciousness score. Our technique ECFL incurs only a very low-overhead of 5 seconds whereas the MBFL technique alone takes 4800 seconds [32]. Li and Zhang [13] proposed a technique named TraPT that improved MBFL by using mutation information collected from test code and messages. However, TraPT takes a long run time since it uses MBFL.

Xuan and Monperrus [29] developed a machine learning based learning to rank approach to combine 25 different SBFL formulae. A technique named Savant was proposed by B. Le et al. [5] that extended SBFL with invariants as an additional feature to improve SBFL.

Sohn and Yoo [24] used source code and change metrics along with 33 spectrum based formulae to improve fault localization (FLUCCS) in a Genetic programming model. Our technique ECFL is different from FLUCCS in following aspects. First, we focus on line level that is more fine-grained compared to method level used in FLUCCS. Second, we show how each of the four source code line level features contributes to improving the three baseline FL techniques. Third, our technique ECFL is a lightweight approach and takes a low run time (5 seconds per bug) unlike genetic programming machine learning approach.

Zou et al. [32] used "learning to rank" machine learning approach (linear SVM) and showed that combining multiple fault localization techniques improved the accuracy significantly. They computed the accuracy of each standalone technique using Defect4J benchmark dataset along with the run time of each technique. We make use of their data and infras-

tructure to show that combining source code line features using both weighted sum and ML approach improves state-of-the-art fault localization techniques.

Chapter 11

Conclusion

We proposed a novel enhanced cost-aware fault localization (ECFL) technique by combining line length with the selected baseline fault localization techniques. We experimented with different line level source code features such as line commit size, line recency, and line length to arrive at a new technique that improves the selected baseline state-of-the-art FL techniques. ECFL evaluation results showed that it improved the accuracy of DStar (Baseline 1), CombineFastestFL (Baseline 2), and CombineFL (Baseline 3) by locating 81%, 58%, and 30% more real faults respectively in Top-1 evaluation metric. ECFL added only a little additional time (5 seconds per bug) while providing a significant improvement over the accuracy of the baselines. We further showed that our source code line level features improve the baseline fault localization techniques when combined with them using learning to rank (SVM) machine learning approach. We also provided our implementation of ECFL as an infrastructure to facilitate future research in fault localization.

Bibliography

[1] Fault localization dataset, 2016. URL http://fault-localization.cs.washington.edu/.

[2] Fault localization combinefl ml infrastructure, 2019. URL https://combinefl.github.io/.

[3] R. Abreu, P. Zoeteweij, and A. J. c Van Gemund. An Evaluation of Similarity Coefficients for Software Fault Localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, pages 39–46, December 2006. doi: 10.1109/PRDC.2006.18.

[4] Hiralal Agrawal and Joseph R. Horgan. Dynamic Program Slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 246–256, New York, NY, USA, 1990. ACM. ISBN 978-0-89791-364-5. doi: 10.1145/93542.93576. URL http://doi.acm.org/10.1145/93542.93576. event-place: White Plains, New York, USA.

[5] Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. A Learning-to-rank Based Fault Localization Approach Using Likely Invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 177–188, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4390-9. doi: 10.1145/2931037.2931049. URL http://doi.acm.org/10.1145/2931037.2931049. event-place: Saarbrücken, Germany.

[6] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction

approaches. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 31–41, May 2010. doi: 10.1109/MSR.2010.5463279.

[7] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 467–477, May 2002. doi: 10.1145/581396.581397.

[8] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 437–440, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2645-2. doi: 10.1145/2610384.2628055. URL http://doi.acm.org/10.1145/2610384.2628055. event-place: San Jose, CA, USA.

[9] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller. Predicting Faults from Cached History. In *29th International Conference on Software Engineering (ICSE'07)*, pages 489–498, May 2007. doi: 10.1109/ICSE.2007.66.

[10] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. Practitioners' Expectations on Automated Fault Localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 165–176, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4390-9. doi: 10.1145/2931037.2931051. URL http://doi.acm.org/10.1145/2931037.2931051. event-place: Saarbrücken, Germany.

[11] G. Laghari, A. Murgia, and S. Demeyer. Fine-tuning spectrum based fault localisation with frequent method item sets. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 274–285, September 2016.

[12] Gulsher Laghari and Serge Demeyer. On the Use of Sequence Mining Within Spectrum Based Fault Localisation. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, pages 1916–1924, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5191-1. doi: 10.1145/3167132.3167337. URL http://doi.acm.org/10.1145/3167132.3167337. event-place: Pau, France.

[13] Xia Li and Lingming Zhang. Transforming Programs and Tests in Tandem for Fault Localization. *Proc. ACM Program. Lang.*, 1(OOPSLA):92:1–92:30, October 2017. ISSN 2475-1421. doi: 10.1145/3133916. URL http://doi.acm.org/10.1145/3133916.

[14] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayşe Bener. Defect Prediction from Static Code Features: Current Results, Limitations, New Approaches. *Automated Software Engg.*, 17(4):375–407, December 2010. ISSN 0928-8910. doi: 10.1007/s10515-010-0069-5. URL http://dx.doi.org/10.1007/s10515-010-0069-5.

[15] S. Moon, Y. Kim, M. Kim, and S. Yoo. Ask the Mutants: Mutating Faulty Programs for Fault Localization. In *Verification and Validation 2014 IEEE Seventh International Conference on Software Testing*, pages 153–162, March 2014. doi: 10.1109/ICST.2014.28.

[16] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 181–190, May 2008. doi: 10.1145/1368088.1368114.

[17] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 284–292, May 2005. doi: 10.1109/ICSE.2005.1553571.

[18] M. Papadakis and Y. Le Traon. Using Mutants to Locate "Unknown" Faults. In *Verification and Validation 2012 IEEE Fifth International Conference on Software Testing*, pages 691–700, April 2012. doi: 10.1109/ICST.2012.159.

[19] Mike Papadakis and Yves Le Traon. Metallaxis-FL: Mutation-based Fault Localization. *Softw. Test. Verif. Reliab.*, 25(5-7):605–628, August 2015. ISSN 0960-0833. doi: 10.1002/stvr.1509. URL http://dx.doi.org/10.1002/stvr.1509.

[20] Chris Parnin and Alessandro Orso. Are Automated Debugging Techniques Actually Helping Programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 199–209, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0562-4. doi: 10.1145/2001420.2001445. URL http://doi.acm.org/10.1145/2001420.2001445. event-place: Toronto, Ontario, Canada.

[21] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller. Evaluating and Improving Fault Localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 609–620, May 2017. doi: 10.1109/ICSE.2017.62.

[22] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry. Improving bug localization using structured information retrieval. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 345–355, November 2013. doi: 10.1109/ASE.2013.6693093.

[23] Francisco Servant and James A. Jones. History Slicing: Assisting Code-evolution Tasks. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 43:1–43:11, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1614-9. doi: 10.1145/2393596.2393646. URL http://doi.acm.org/10.1145/2393596.2393646. event-place: Cary, North Carolina.

[24] Jeongju Sohn and Shin Yoo. FLUCCS: Using Code and Change Metrics to Improve Fault Localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 273–283, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5076-1. doi: 10.1145/3092703.3092717. URL http://doi.acm.org/10.1145/3092703.3092717. event-place: Santa Barbara, CA, USA.

[25] I. Vessey. Expertise in Debugging Computer Programs: An Analysis of the Content of Verbal Protocols. *IEEE Transactions on Systems, Man, and Cybernetics*, 16(5): 621–637, September 1986. ISSN 0018-9472. doi: 10.1109/TSMC.1986.289308.

[26] W. E. Wong, V. Debroy, R. Gao, and Y. Li. The DStar Method for Effective Software Fault Localization. *IEEE Transactions on Reliability*, 63(1):290–308, March 2014. ISSN 0018-9529. doi: 10.1109/TR.2013.2285319.

[27] X. Xia, L. Bao, D. Lo, and S. Li. "Automated Debugging Considered Harmful" Considered Harmful: A User Study Revisiting the Usefulness of Spectra-Based Fault Localization Techniques with Professionals Using Real Bugs from Large Systems. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 267–278, October 2016. doi: 10.1109/ICSME.2016.67.

[28] X. Xie, Z. Liu, S. Song, Z. Chen, J. Xuan, and B. Xu. Revisit of Automatic Debugging via Human Focus-Tracking Analysis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 808–819, May 2016. doi: 10.1145/2884781.2884834.

[29] J. Xuan and M. Monperrus. Learning to Combine Multiple Ranking Metrics for Fault Localization. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 191–200, September 2014. doi: 10.1109/ICSME.2014.41.

[30] Mengshi Zhang, Xia Li, Lingming Zhang, and Sarfraz Khurshid. Boosting Spectrum-based Fault Localization Using PageRank. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 261–272, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5076-1. doi: 10.1145/3092703.3092731. URL http://doi.acm.org/10.1145/3092703.3092731. event-place: Santa Barbara, CA, USA.

[31] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Locating Faults Through Automated Predicate Switching. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 272–281, New York, NY, USA, 2006. ACM. ISBN 978-1-59593-375-1. doi: 10.1145/1134285.1134324. URL http://doi.acm.org/10.1145/1134285.1134324. event-place: Shanghai, China.

[32] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang. An Empirical Study of Fault Localization Families and Their Combinations. *IEEE Transactions on Software Engineering*, pages 1–1, 2019. ISSN 0098-5589. doi: 10.1109/TSE.2019.2892102.

Appendices

Appendix A

ECFL Results

A.1 ECFL Results for 100% of Defects4J bugs

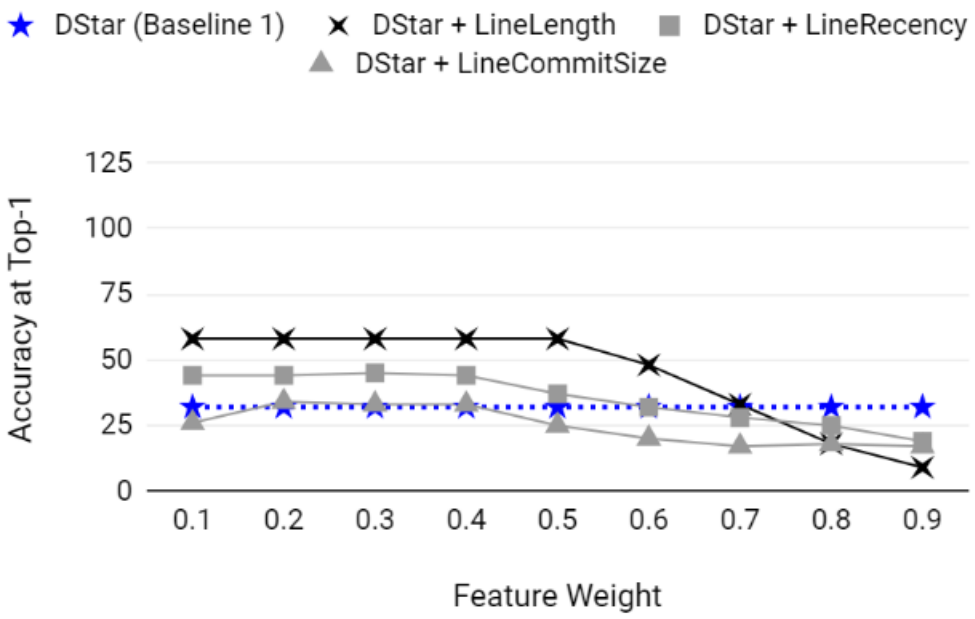

Figure A.1: Comparison of Features with Baseline 1 using ECFL: Accuracy at Top-1

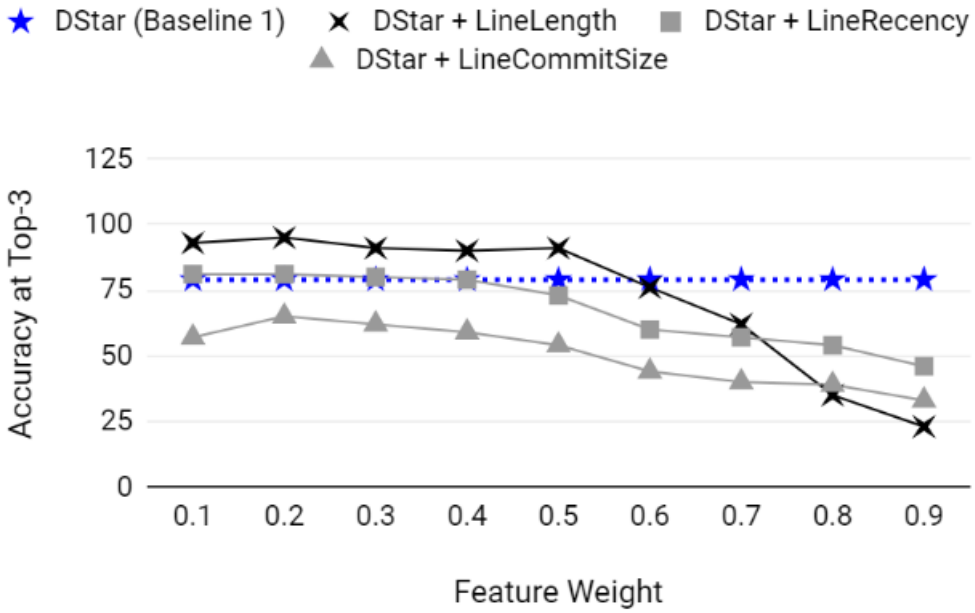

Figure A.2: Comparison of Features with Baseline 1 using ECFL: Accuracy at Top-3

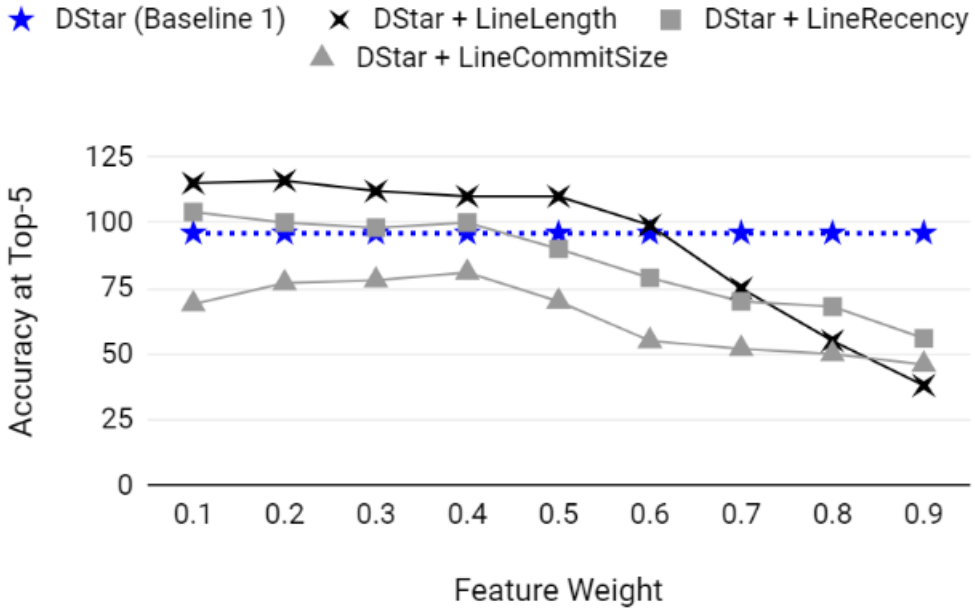

Figure A.3: Comparison of Features with Baseline 1 using ECFL: Accuracy at Top-5

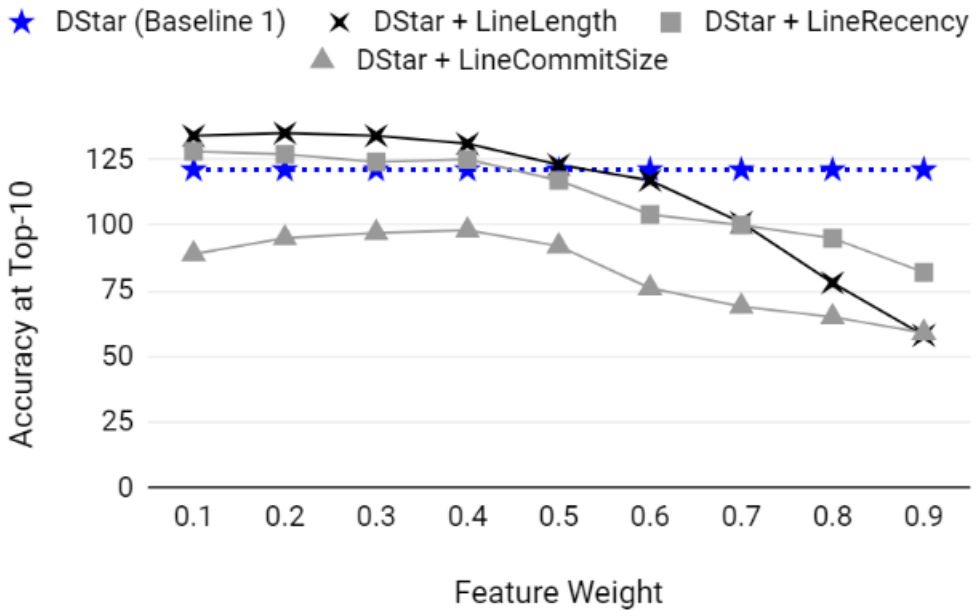

Figure A.4: Comparison of Features with Baseline 1 using ECFL: Accuracy at Top-10

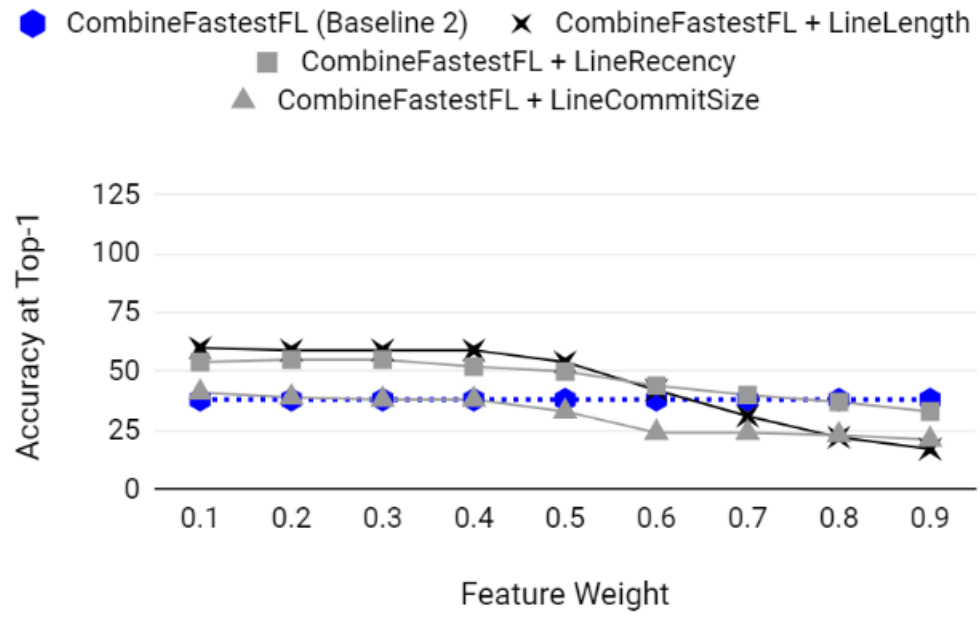

Figure A.5: Comparison of Features with Baseline 2 using ECFL: Accuracy at Top-1

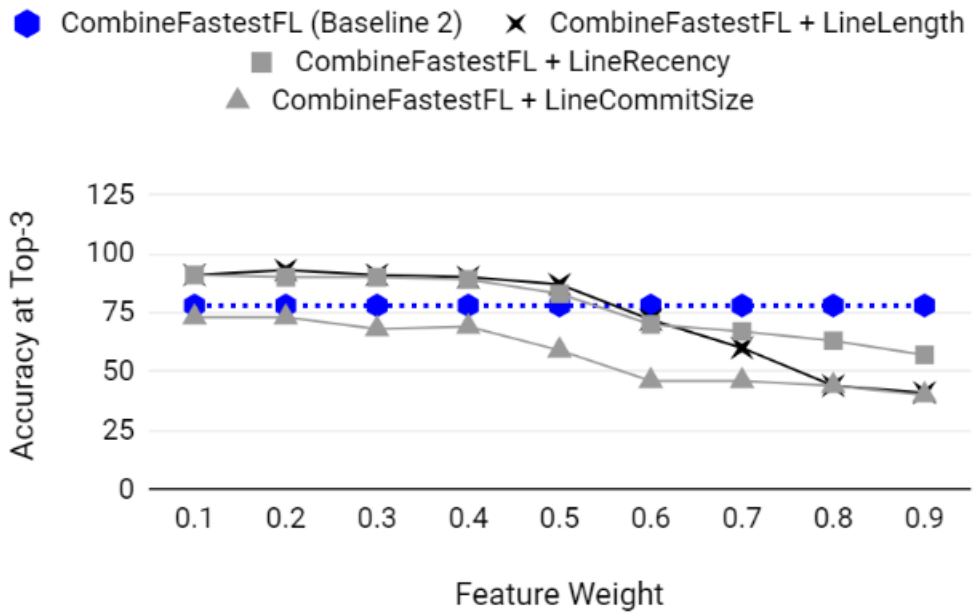

Figure A.6: Comparison of Features with Baseline 2 using ECFL: Accuracy at Top-3

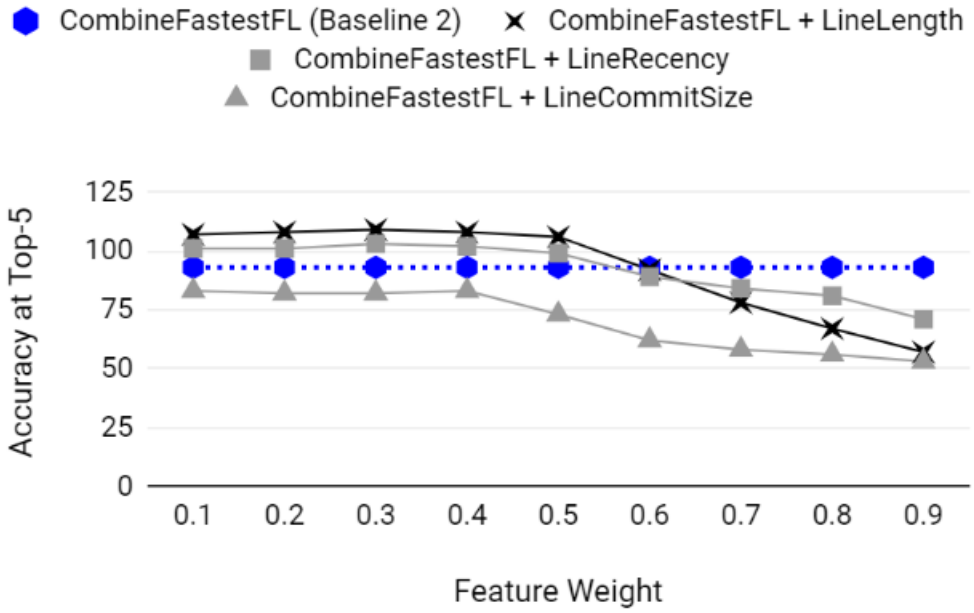

Figure A.7: Comparison of Features with Baseline 2 using ECFL: Accuracy at Top-5

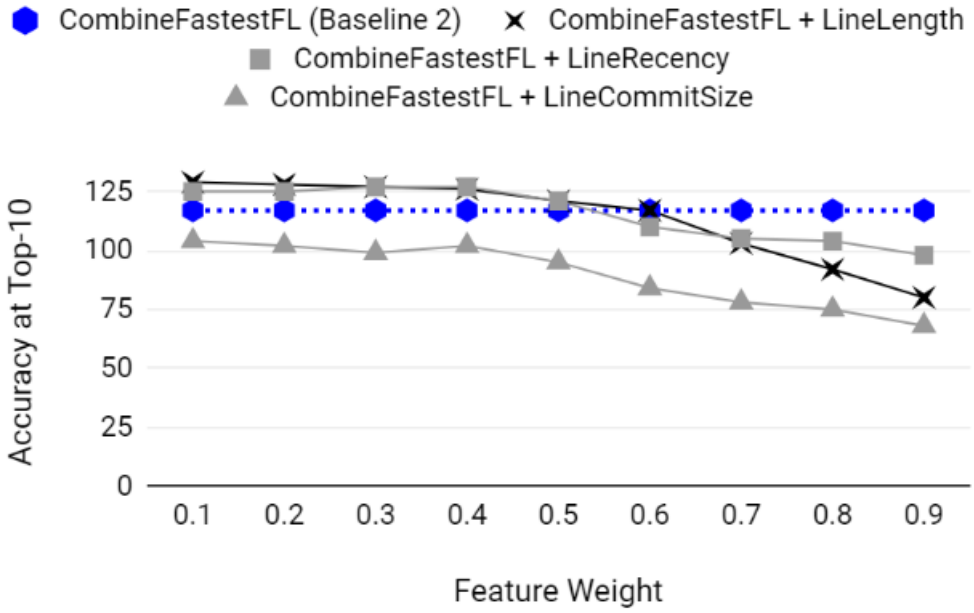

Figure A.8: Comparison of Features with Baseline 2 using ECFL: Accuracy at Top-10

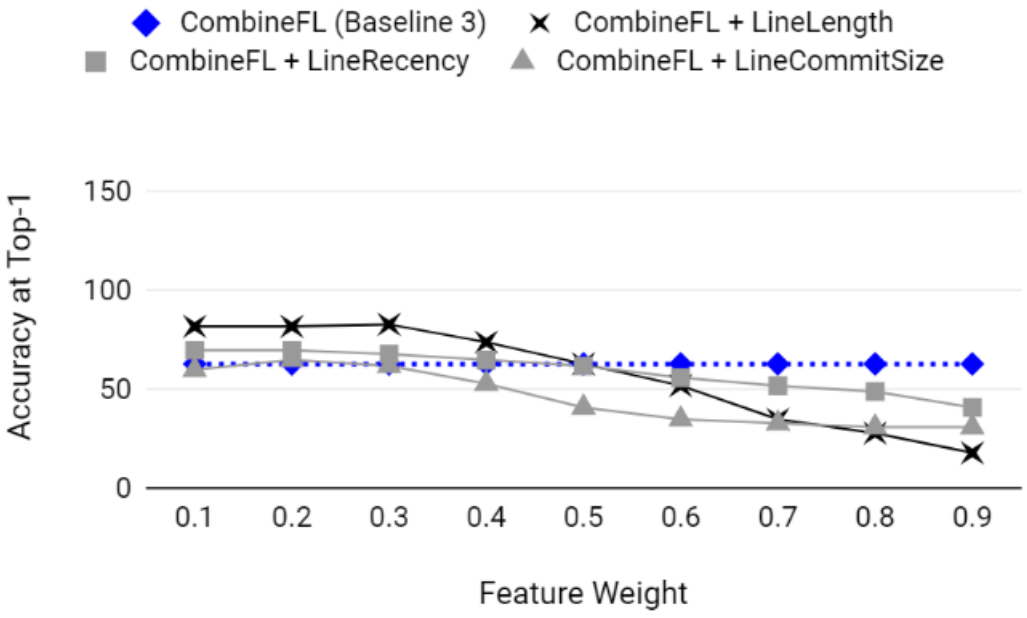

Figure A.9: Comparison of Features with Baseline 3 using ECFL: Accuracy at Top-1

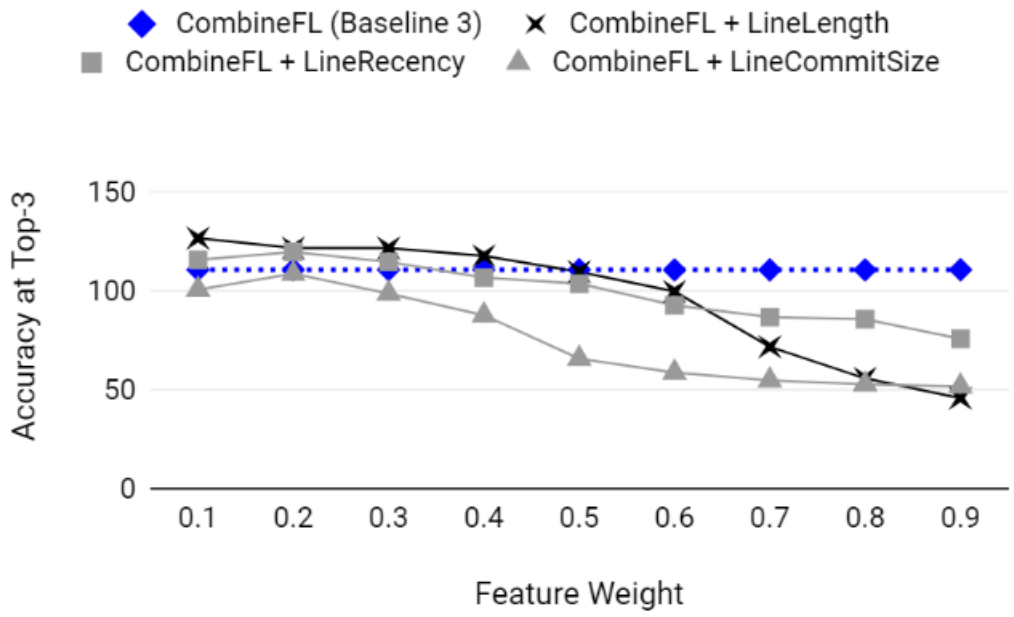

Figure A.10: Comparison of Features with Baseline 3 using ECFL: Accuracy at Top-3

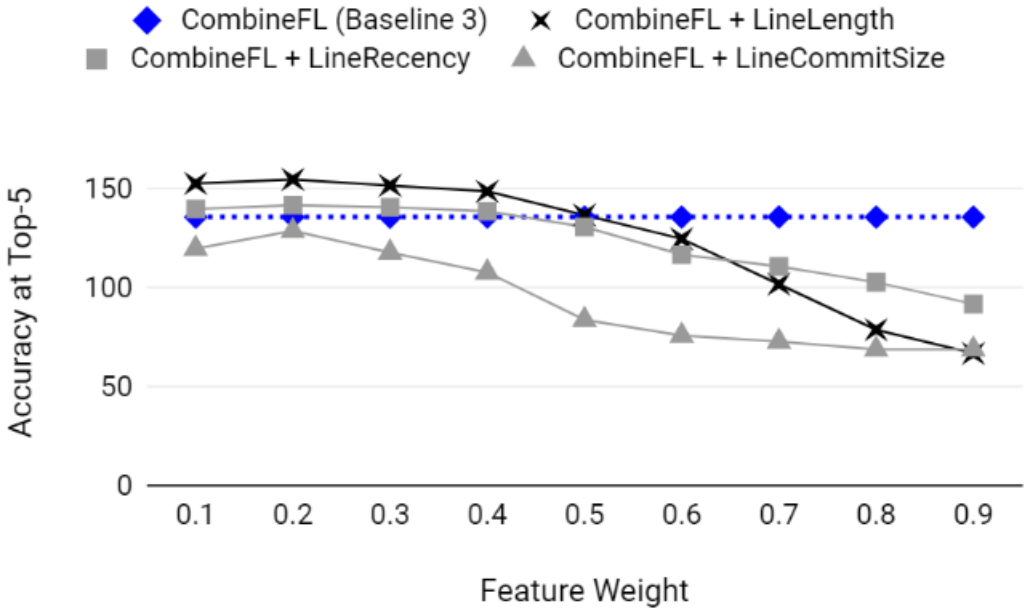

Figure A.11: Comparison of Features with Baseline 3 using ECFL: Accuracy at Top-5

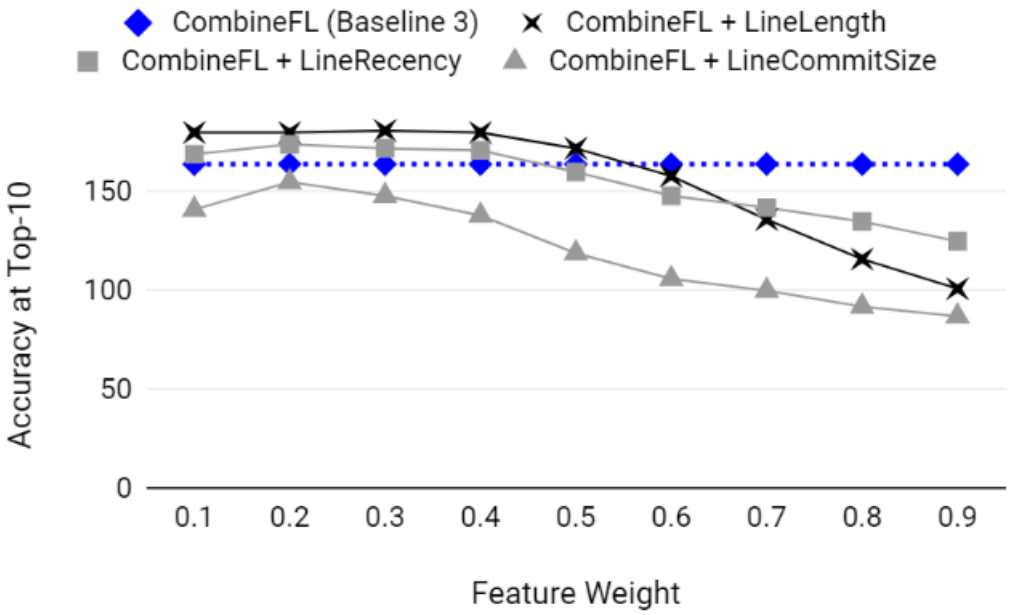

Figure A.12: Comparison of Features with Baseline 3 using ECFL: Accuracy at Top-10