FLUCCS: Using Code and Change Metrics to Improve Fault Localisation

ABSTRACT

Fault localisation aims to support the debugging activities of human developers by highlighting the program elements that are suspected to be responsible for the observed failure. Spectrum Based Fault Localisation (SBFL), an existing localisation technique that only relies on the coverage and pass/fail results of executed test cases, has been widely studied but also criticised for the lack of precision and limited effort reduction. To overcome restrictions of techniques based purely on coverage, we extend SBFL with code and change metrics that have been studied in the context of defect prediction, such as size, age and code churn. Using suspiciousness values from existing SBFL formulæ and these source code metrics as features, we apply two learn-to-rank techniques, Genetic Programming (GP) and linear rank Support Vector Machines (SVMs). We evaluate our approach with a ten-fold cross validation of method level fault localisation, using 210 real world faults from the `Defects4J` repository. GP with additional source code metrics ranks the faulty method at the top for 106 faults, and within the top five for 173 faults. This is a significant improvement over the state-of-the-art SBFL formulæ, the best of which can rank 49 and 127 faults at the top and within the top five, respectively.

CCS CONCEPTS

•Software and its engineering →Search-based software engineering;

KEYWORDS

Fault Localisation, SBSE, Genetic Programming

ACM Reference format:
. 2016. FLUCCS: Using Code and Change Metrics to Improve Fault Localisation. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 11 pages.
DOI: 10.475/123_4

1 INTRODUCTION

As software systems grow larger and more complex, they are beset with an increasingly large number of faults. While automated test data generation [11, 12, 25] may reduce the cost of test creation and eventually lead to easier detection of faults in the System Under Test (SUT), the task of debugging itself is still largely left to human developers, taking up to 80% of total software cost for some projects [35]. Consequently there is an urgent need for automated support for human debugging. Automated patching [9, 13] has been proposed as an alternative to human debugging, but it also requires automated guidance on *where* in the SUT to modify in order to remove the observed fault, further adding to the need for automated debugging support.

Fault localisation is a problem that, given results of testing, asks to identify the location of the fault in the SUT [36]. A particular branch of localisation techniques that has been widely studied is Spectrum Based Fault Localisation (SBFL) [34, 38], which takes the coverage and pass/fail information of individual test cases and assigns to each program element in SUT what is called a *suspiciousness score*. The score of a given program element is expected to be correlated with likelihood of it containing the fault. The expected use case is that the human developer can inspect program elements following the ranking based on these scores, thereby reaching the faulty program element faster than when following the original order given by the source code structure [33].

While SBFL has received much attention [2, 6, 16, 22, 37], its limits have also been pointed out both empirically and theoretically. Parnin and Orso conducted an empirical human evaluation of the effectiveness of SBFL techniques and reported that previous claims on helpfulness did not hold in practice [30]. Furthermore, they pointed out that the traditional evaluation metric for SBFL, 'Expense', is misleading. Expense measures the wasted effort (i.e. number of program elements that are ranked above the faulty one) as a percentage of the size of SUT. While a single digit, or even fractional Expense value may appear impressive, the accuracy of localisation may still be impractical if the SUT is sufficiently large. Theoretically, it has been recently proved that there does not exist a single SBFL risk evaluation formula that is guaranteed to outperform all others [43], following analyses on maximality and hierarchy between different formulæ [2, 29, 38].

This paper presents FLUCCS (Fault Localisation Using Code and Change Metrics), a fault localisation technique that learns to rank program elements based on both existing SBFL techniques and source code metrics. FLUCCS makes a series of critical design choices based on findings in the literature. Instead of designing a single formula or a fixed technique, it opts to *learn* how to rank from a given training data set. Since there does not exist a single greatest SBFL formula [43], the best SBFL formula has to be adaptively learnt rather than declared. We use Genetic Programming as the learning mechanism: it not only can deal with non-linear models, but also has been proven effective at learning SBFL formulæ [39, 41]. Instead of learning from raw spectrum data, we use existing SBFL formulæ as features for learning, as the theoretical analysis shows that different formulæ are already maximal against different classes of faults [38]. Finally, and most importantly, we use a number of source code metrics previously studied for defect prediction as additional features for learning. This is to improve the accuracy of localisation measured by the *absolute ranking*, following the guidelines by Parnin and Orso [30]. We posit that the same set of features can be effective for both defect prediction and fault localisation: defect prediction can be interpreted as aiming to localise faults *a priori* (i.e. before testing and actual detection), whereas fault localisation simply does so *post hoc*.

We empirically evaluate FLUCCS using 210 real world faults from `Defects4J` repository [19]. The method level localisation results obtained by FLUCCS have been compared to those from existing SBFL baselines, FLUCCS with different learning mechanism, as well as FLUCCS without the additional source code metric features. FLUCCS with Genetic Programming convincingly outperforms all the other approaches, placing the faulty method at the top of the ranking for 106 faults out of 210. The final result shows that source code metrics that are relatively easy to collect may effectively augment existing SBFL techniques for higher accuracy.

The technical contribution of this paper can be summarised as follows:

- We present FLUCCS, a fault localisation technique that learns to rank program elements using Genetic Programming, existing SBFL techniques, and source code metrics[1].
- We empirically evaluate FLUCCS using 210 real world faults from `Defects4J`. FLUCCS ranks 50% of the studied faults at the top, and about 82% of the studied faults within the top 5 of the ranking.
- We introduce a new way of computing method level SBFL scores called method level aggregation. Empirical evaluation of this technique applied to existing state-of-the-art SBFL formulæ shows that formulæ with method level aggregation can rank about 42% more faults at the top.
- We show that simple source code metrics can effectively augment existing SBFL techniques for more accurate localisation, prompting further study of the connection between defect prediction and fault localisation.

The rest of the paper is organised as follows: Section 2 formulates fault localisation as a learning to rank problem and introduces the features FLUCCS uses. Section 3 describes the learning algorithms that we use in the paper. Section 4 presents the set-up for the empirical evaluation, the results of which are discussed in Section 5. Section 6 discusses the potential threats to validity. Section 7 presents the related work and Section 8 concludes.

2 FEATURES USED BY FLUCCS

Figure 1 shows the overall architecture of FLUCCS. FLUCCS extracts two sets of features from a source code repository. The first is a set of SBFL scores using different SBFL formulæ: this requires test execution on source code instrumented for structural coverage. The second is a set of code and change metrics: this requires lightweight static analysis and version mining. In training phase, these features, along with locations of known faults, are fed into learning algorithms, which produce ranking models that rank the faulty method as high as possible. In deployment phase, these learnt models take the features from source code with unknown faults, and produce rankings of methods according to their likelihood of being faulty. In this section, we describe the features used by FLUCCS, as well as how these features are extracted and processed.

[1]FLUCCS and the data used for the empirical evaluation are made available at http://Redacted.For.Double.Blind.Review

2.1 SBFL Scores

SBFL formulæ take program spectrum data as input and return risk scores (also known as suspiciousness scores). For a structural program element (such as a statement or a method), the spectrum data consists of four variables that are aggregated from test coverage and pass/fail results: (e_p, e_f, n_p, n_f): e_p and e_f represent the number of passing and failing test cases that execute the given structural element, respectively. Similarly, n_p and n_f represent the number of passing that failing test cases that do not execute the given structural element. SBFL formulæ tend to assign higher risk scores to elements wigh higher e_f and n_p values, which suggest executing those elements tend to result in failing test executions, while not executing them tend to result in passing test executions.

Table 1: SBFL formulæ used by FLUCCS as features

Name	Formula	Name	Formula
ER1$_a$	$\begin{cases} -1 & \text{if } n_f > 0 \\ n_p & \text{otherwise} \end{cases}$	ER1$_b$	$e_f - \frac{e_p}{e_p+n_p+1}$
ER5$_a$	$e_f - \frac{e_f}{e_p+n_p+1}$	ER5$_b$	$\frac{e_f}{e_f+n_f+e_p+n_p}$
ER5$_c$	$\begin{cases} 0 & \text{if } e_f < F \\ 1 & \text{otherwise} \end{cases}$	GP$_2$	$2(e_f+\sqrt{e_p+n_p})+\sqrt{e_p}$
Ochiai	$\frac{e_f}{(e_f+n_f)(e_f+e_p)}$	GP$_3$	$\sqrt{\lvert e_f^2-\sqrt{e_p}\rvert}$
Jaccard	$\frac{e_f}{e_f+n_f+e_p}$	GP$_{13}$	$e_f(1+\frac{1}{2e_p+e_f})$
AMPLE	$\lvert\frac{e_f}{F}-\frac{e_p}{P}\rvert$	GP$_{19}$	$e_f\sqrt{\lvert e_p-e_f+F-P\rvert}$
Hamann	$\frac{e_f+n_p-e_p-n_f}{P+F}$	Tarantula	$\frac{\frac{e_f}{e_f+n_f}}{\frac{e_f}{e_f+n_f}+\frac{e_p}{e_p+n_p}}$
Dice	$\frac{2e_f}{e_f+e_p+n_f}$	RusselRao	$\frac{e_f}{e_p+e_f+n_p+n_f}$
M1	$\frac{e_f+n_p}{n_f+e_p}$	SørensenDice	$\frac{2e_f}{2e_f+e_p+n_f}$
M2	$\frac{e_f}{e_f+n_p+2n_f+2e_p}$	Kulczynski1	$\frac{e_f}{n_f+e_p}$
Hamming	e_f+n_p	Kulczynski2	$\frac{1}{2}(\frac{e_f}{e_f+n_f}+\frac{e_f}{e_f+e_p})$
Goodman	$\frac{2e_f-n_f-e_p}{2e_f+n_f+e_p}$	SimpleMatching	$\frac{e_f+n_p}{e_p+e_f+n_p+n_f}$
Euclid	$\sqrt{e_f+n_p}$	RogersTanimoto	$\frac{e_f+n_p}{e_f+n_p+2n_f+2e_p}$
Wong1	e_f	Sokal	$\frac{2e_f+2n_p}{2_ef+2n_p+n_f+e_p}$
Wong2	e_f-e_p	Anderberg	$\frac{e_f}{e_f+2e_p+2n_f}$
Wong3	$e_f-h, h=\begin{cases} e_p & \text{if } e_p \leq 2 \\ 2+0.1(e_p-2) & \text{if } 2<e_p\leq 10 \\ 2.8+0.01(e_p-10) & \text{if } e_p>10 \end{cases}$		
Ochiai2	$\frac{e_f n_p}{\sqrt{(ef+e_p)(n_f+n_p)(e_f+n_p)(e_p+n_f)}}$		
Zoltar	$\frac{e_f}{e_f+e_p+n_f+\frac{10000 n_f e_p}{e_f}}$		

FLUCCS uses 33 SBFL formulæ to generate score metrics, which are listed in Table 1 We include both the state-of-the-art human generated SBFL formulæ and GP evolved SBFL formulæ. Of these, 25 formulæ have been used in combination with each other in previous work [3, 40], while eleven formulæ have been proven to be maximal [39].

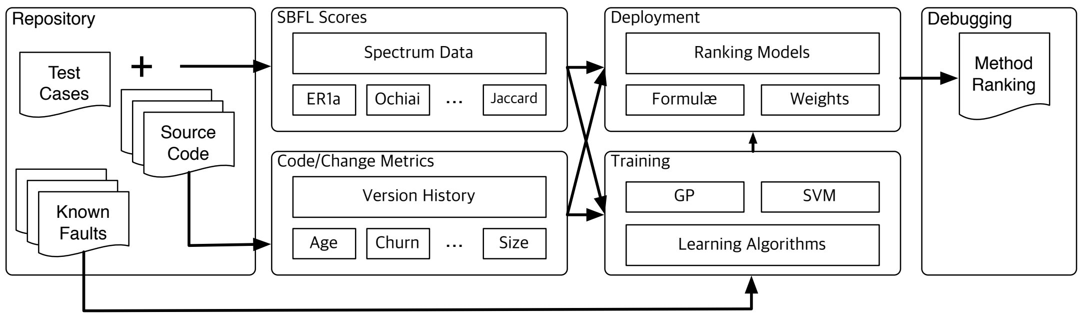

Figure 1: Overall Architecture of FLUCCS

2.2 Code and Change Metrics

There is a large number of code and change metrics that have been studied in relation to defect proneness [7, 26–28]. We adopt some of the widely studied code and change metrics as features to our learning, expecting that these features will provide additional guidance towards the faulty program elements. In total, we use 14 code and change metrics.

2.2.1 Age. Age simply measures how long a given program element has existed in the code base [27]. We calculate the age of a given statement as the number of consecutive versions from the faulty version backwards to the latest version containing a modification to the statement. Statement level age metric is aggregated into three different method ages: minimum, maximum, and mean ages of statements that consist the method. Min and max ages represent the ages of the youngest and the oldest statement in the method, whereas the mean age represents the average age of all statements in the method.

2.2.2 Churn. Churn metric measures the change frequency of a given program element, and has been shown to be correlated with the fault density [28]. Churn metric is calculated as the number of commits that have changed the structural element (such as methods) under consideration, divided by the total number of commits made to the repository up to the faulty version. A method is considered to be changed if any of its statements is changed.

2.2.3 Complexity Metrics. Code complexity and its impact on defect proneness has been widely studied [7]. Various types of code complexity metrics have been suggested in the literature, out of which we select the following, cheap to measure, metrics:

- Number of formal arguments: this indirectly reflects the internal complexity of the given method, as well as the degree of external coupling.
- Number of local variables: this indirectly measures the internal complexity.
- Size: this has been used by much of the defect prediction work in the literature as a surrogate for code complexity [7, 14, 23]. We use both LoC (Lines of Code) and the number of compiled Java Bytecode instructions.

2.3 Method Level Aggregation of SBFL Scores

Although FLUCCS performs method level localisation, it does not use method coverage to calculate the SBFL score features. Instead, we calculate SBFL scores for statements and aggregate them up to the method level by taking the highest score among those from statements that consist the method under consideration. While this adds to the cost of localisation (instrumentation for the statement coverage is more expensive than one for the method coverage), this has clear benefits.

Consider the code snippet in Figure 2, which is executed with three test cases: a = 1, 2, 3. Let us also assume that there exist two other test cases that do not execute this method. In total, there are five test cases: three execute testMe and one of them fails.

Method testMe is covered by three test cases: its spectrum tuple (e_p, e_f, n_p, n_f) is $(2, 1, 2, 0)$, resulting in Ochiai score of $\frac{1}{\sqrt{1(1+2)}} = 0.578$ and Jaccard score of $\frac{1}{1+0+2} = 0.333$. Method util and its line 12 share the same spectrum tuple as well as scores, making it impossible to differentiate util and testMe. However, for line 4, the spectrum tuple (e_p, e_f, n_p, n_f) becomes $(0, 1, 4, 0)$, resulting in both Ochiai and Jaccard score of 1.0, placing testMe above util.

```
1  public void testMe(int a){
2    util();
3    if (a % 3 == 0){
4      ... // faulty code
5    }
6    else{
7      ...
8    }
9  }
10
11 public void util(){
12   ...
13 }
```

Figure 2: Example code snippet showing the benefits of method level aggregation. With method coverage, `testMe` and `util` share the same SBFL scores; however, if we represent `testMe` with the highest SBFL score among those of its constituent statements, it is ranked higher than `util`.

In general, there are two drawbacks in using method coverage to calculate SBFL scores. First, methods on a single call chain can share the same spectrum tuple values, resulting in tied SBFL scores. Second, if there exist test cases that execute only the non-faulty parts of an actually faulty method, they will increase the e_p value at the method level. This is undesirable, because with most of the practically effective SBFL formulæ, higher e_p values tend to decrease the suspiciousness. Our method level aggregation approach is designed to overcome these two weaknesses.

2.4 Call Graph Propagation

While a newly created fault may be directly committed into a code repository, a *regression* fault can be caused at an unchanged location, different from the latest change that is, in itself, completely valid [42]. Changes in method interface or expected semantic behaviour can cause such regression faults. Consequently, if a change metric is an effective indicator of fault proneness, we argue that its impact should be propagated through the dependency graph.

We propagate age and churn metrics through method level call graph, which is extracted using Apache Bytecode Engineering Library (BCEL [2]). With the basic age metrics, we include three additional Call Graph Propagated (CGP) versions of age metrics: CGP min age, CGP max age, and CGP mean age. For a given method, its CGP min/max age is defined recursively as the smallest/largest value among its own min/max age, and min/max CGP age values of all its callees; its CGP mean age is the mean of its own mean age as well as the mean CGP ages of all of its callees. For methods without callees, their CGP age values are the same as their own age values.

Similarly to the way age metrics are propagated, we include three additional churn metrics: CGP min churn, CGP max churn, and CGP mean churn, based on the churn metrics of the method in consideration plus those of all its callees.

3 LEARNING ALGORITHMS

Learning to rank is a technique that uses machine learning to construct ranking models for an information retrieval system [24]. It aims to learn how to produce a permutation of unseen lists of items in *some* way that is similar to ones that have been provided as training data. There are three different approaches to learning to rank: pointwise, pairwise, and listwise. Pointwise approaches approximate learning to rank problems as regression problems for the ordinal scores in the training data. Pairwise approaches transform learning to rank problems as classification problems for pairs of items: by classifying pairs according to their ordinal relationships, it aims to minimise ordinal inversions. Listwise approaches attempt to produce ranking models that minimise the dissimilarity to rankings in the training data.

With FLUCCS, the objective for learning is to construct ranking models that rank faulty program elements as high as possible, based on features described in Section 2. Fault localisation is a unique learning to rank problem, as our interest is limited solely to the rank of the faulty program elements, and not those of the other, non-faulty ones. The labels in training data are binary: one for faulty elements, and zero otherwise. Even with multi-location faults, there will be significantly more zeros than ones.

In this paper, we evaluate a pointwise and a pairwise approach. We consider the listwise approach to be inappropriate, because the rankings in the training data are mostly all tied (i.e. zero for not faulty). For the pointwise approach, we choose Genetic Programming; for the pairwise approach, we choose rankSVM [21].

[2]https://commons.apache.org/proper/commons-bcel/

3.1 Genetic Programming

We use GP as a symbolic regression tool to learn the ranking models: it evolves a ranking function that takes features and produce ordinal scores. Instead of evolving a function that reproduces the original binary labels (i.e. 'faulty' or not 'faulty') as closely as possible, our fitness function is simply the average ranking of the faulty program element (the one that is ranked highest, if multiple elements are marked to consist a single fault), calculated from all faults that are considered for fitness evaluation. GP has been successfully applied to evolving SBFL formulæ from raw spectrum data [41] and has the benefit of being able to generate non-linear ranking models.

3.2 Support Vector Machine

Ranking SVM is a variant of Support Vector Machine [5] algorithm that performs pairwise learning to rank. We use rankSVM [21], an implementation of linear ranking SVM. It learns the linear weights to features that produce ordinal scores with the fewest ordinal inversions. While being orders of magnitudes faster than GP, linear ranking SVMs are restricted by the linearity of the ranking model and the inherent imbalance in fault localisation training data.

4 EXPERIMENTAL SETUP

4.1 Research Questions

We investigate the following research questions to evaluate the effectiveness of FLUCCS.

RQ1. Effectiveness: How effective is FLUCCS at localizing the studied faults?

We evaluate the effectiveness of the GP version of FLUCCS that uses all features (referred to as GP^A hereafter), by computing the evaluation metrics described in Section 4.4. Due to the stochastic nature of GP, we evaluate 30 runs of GP and generate 30 ranking models per training data set: when evaluating these ranking models with test data sets, we pick models with the best and median performance. The best performance model, GP^A_{min}, (i.e. the one that produces the best fitness out of the 30 runs) represents the best use case scenario, in which the user of FLUCCS completes multiple GP runs and picks the best model. The median performance model, GP^A_{med}, is the one that corresponds to the median fitness value from multiple runs; it is included to show the variance in the models produced by the GP version of FLUCCS. These evaluation results are then compared with evaluation results of 11 state-of-art SBFL formulæ, including both human designed and GP evolved ones, using the same evaluation metrics.

RQ2. Code and Change Metric Contribution: How much do the code and change metric features contribute to the localisation of faults?

To confirm that the code and change metric features contribute positively to localisation, we evaluate FLUCCS using only SBFL score features, leaving other settings for GP untouched. Resulting models, GP^S_{min} and GP^S_{med}, are compared with GP^A_{min} and GP^A_{med}.

RQ3. Method Level Aggregation and Call Graph Propagation: How much do the method level aggregation and the call graph propagation contribute to the localisation of faults respectively?

Method level aggregation can be applied to any spectrum-based technique, whereas the use of call graph propagation is unique to FLUCCS due to its use of code and change metrics. Consequently, we evaluate the contribution of the method level aggregation and the call graph propagation separately. To evaluate the impact of method level aggregation, we simply compare two sets of SBFL scores from 11 state-of-art SBFL formulæ, with and without method level aggregation. This will evaluate whether method level aggregation can be generally useful to any spectrum-based techniques. Since these formulæ provide SBFL scores as features for FLUCCS, we posit that improvements in their scores will result in improvements in FLUCCS as well.

For the evaluation of the call graph propagation, we compare results of FLUCCS with and without call graph propagation, leaving all other factors the same. Results with call graph propagation are named $GPCG_{min}$ and $GPCG_{med}$, using min and median model from multiple GP runs, respectively.

RQ4. Learning Algorithm: Is GP a suitable approach to learn the ranking?

To evaluate the effectiveness of GP as a learning mechanism, we compare GP^A_{min}, GP^A_{med}, GP^S_{min}, and GP^S_{med} to corresponding versions of FLUCCS that uses rankSVM as the learning mechanism: SVM^A and SVM^S.

Table 2: Subject software systems and their faults

Project	# Faults	Loc	# Methods	# Test cases
Commons Lang	60	9343–11813	1794–2335	1585–2295
Joda-Time	27	12986–13604	3338–3510	3749–4041
Commons Math	96	4771–42408	897–5905	817–4429
Closure Compiler	27	43809–45151	6884–7187	7514–7911

4.2 Subjects

We use real world faults from `Defects4J` repository [19] to evaluate FLUCCS. Table 2 lists the subject programs. The version of `Defects4J` we use is 0.2.0, which contains 357 faults; we use 210 faults due to issues that prevent us establishing the ground truth about input features and locations of the real faults. Our filtering criteria are:

- **Missing Revision IDs:** To extract code and change metrics, FLUCCS requires the revision id of the faulty version, so that it can process the original faulty version of SUT in the context of consecutive commits to establish the ground truth. We exclude JFreeChart in `Defects4J` because its revision ids in `Defects4J` repository do not align with those in its own repository.
- **Scope of Faults:** We focus only on the methods that are parts of the given SUT. If the faulty method does not originate from the subject, it is considered out of scope. For example, fault 23 from Commons-Lang in `Defects4J` has been excluded as the location of the fault is a method that overrides another method external to Commons-Lang.
- **Limitations of JaCoCo:** We use `JaCoCo` [1] to collect coverage data. For some faulty methods, we noted that `JaCoCo` fails to record coverage when some test cases do actually execute them and reveal faults. This is a known limitation of `JaCoCo`[3]. We filter out a total of 17 faults due to missing coverage (Commons-Lang:4, Commons-Math: 10, Closure: 3)

From the 210 faulty versions that we study, any method that are not executed at all has been excluded from analysis, as they cannot cause any observable failures. `Defects4J` provides the location of faults in the form of patches that fix them. Consequently, we take the methods that are patched as the ground truth for the location of the fault.

4.3 Configuration

4.3.1 Genetic Programming. We use DEAP [10], a Python evolutionary computation framework, to implement the GP version of FLUCCS. FLUCCS uses a tree-based GP with single-point crossover with rate of 1.0 and subtree mutation with rate of 0.1. The population contains 40 individuals and is initialised by the ramping method [31]; the maximum tree depth is 8 and the algorithm stops after 100 generations. As described in Section 3, GP uses the rank of a known faulty program element as the fitness. Table 3 lists types of operator nodes used by GP; for terminal nodes, we use variables corresponding to 47 features described in Section 2 plus a constant one (1.0).

Table 3: List of GP operators

Operator Node	Definition
`gp_add(a, b)`	`a + b`
`gp_sub(a, b)`	`a - b`
`gp_mul(a, b)`	`ab`
`gp_div(a, b)`	1 if $b = 0$, $\frac{a}{b}$ otherwise
`gp_unarymin(a)`	$-a$
`gp_sqrt(a)`	$\sqrt{\lvert a \rvert}$

To avoid overfitting of GP, we randomly sample 30 faults from the training data set, which consists of 189 faults, for fitness evaluation in each generation in GP. We also adopt elitism, preserving the best 8 individuals from the parent generation into the generation of offspring; these individuals are reevaluated with the new sample at each generation.

4.3.2 RankSVM. We use version 3.20 of `rankSVM`, which depends on `libSVM` [4], with out-of-the-box default parameters. It uses the deterministic trust region Newton method to minimise the loss function and has been used to learn ranking models for fault localisation in the literature [3].

[3]The official FAQ (http://www.eclemma.org/jacoco/trunk/doc/faq.html) states that, if the normal sequence of statement execution is disturbed (by, for example, exceptions), a probe inserted by `JaCoCo` may not be executed, resulting in a failure to record coverage of any statements executed between the previous probe and the missed one.

4.4 Evaluation Metrics

We use three metrics to analyse the performance of GP with new features following existing work [3, 40]. In particular, the use of accuracy (*acc@n*) and wasted effort (wef) conforms to the guideline from Parnin and Orso [30], as these metrics are based on an absolute count of program elements, rather than percentage values.

4.4.1 Accuracy (acc@n). *acc@n* counts the number of faults that have been localised within top *n* places of the ranking. We use 1, 3, 5 for number *n*, and count the number of corresponding faults per project and also overall. When there are multiple faulty program elements, we assume the fault is localised if any of them are ranked within top *n* places.

4.4.2 Wasted Effort (wef). wef measures the amount of effort wasted looking at non-faulty program elements. Essentially, wef can be interpreted as the absolute count version of the traditional Expense metric.

4.4.3 Mean Average Precision (MAP). MAP is an evaluation metric for ranking, used in Information Retrieval; it is the mean of the average precision of all faults. First, we define the precision of localisation at each rank i, $P(i)$:

$$P(i) = \frac{\text{number of faulty methods in top } i \text{ ranks}}{i} \quad (1)$$

Average precision (AP) for a given ranking is the average precision for faulty program elements:

$$AP = \sum_{i=1}^{M} \frac{P(i) \times isFaulty(i)}{\text{number of faulty methods}} \quad (2)$$

Mean Average Precision (MAP) is the mean of AP values computed for a set of faults. We calculate MAP for faults belonging to the same project.

Table 4: Metric values showing the effectiveness of FLUCCS

Technique	Project	Total	*acc*			wef		MAP
		Faults	@1	@3	@5	mean	std	
GP^A_{min}	Lang	60	34	51	56	1.1833	2.3345	0.6794
	Time	27	11	15	19	4.2222	7.2231	0.4060
	Math	96	54	72	82	4.2396	16.2075	0.5884
	Closure	27	7	15	16	37.5185	98.1821	0.3460
	Overall	210	106	153	173	11.7909	45.1638	0.5050
GP^A_{med}	Lang	60	34	49	54	1.3500	2.5088	0.6865
	Time	27	10	16	17	19.3704	58.6137	0.4001
	Math	96	55	76	82	4.3333	20.8185	0.6401
	Closure	27	9	14	17	92.5185	284.0684	0.3768
	Overall	210	108	155	170	29.3931	130.4855	0.5259

4.5 Validation

To maximize the size of training data set and also to avoid overfitting of GP, we use ten-fold cross validation. Given the set of 210 faults, we divides fault data set into ten different sets, each comprises 21 faults. These sets are used as test data sets. Each test data set is paired with remaining faults as matching training data set.

We execute 30 runs of FLUCCS with GP, resulting in 30 different ranking models. To summarize and compare overall results, evaluation metrics of rankings are aggregated per `Defects4J` project. Since *acc@n* is a counting metric (i.e. it is a count of faults for which a localisation method ranks the fault at the top), we simply count the number of faults, in a `Defects4J` project, for which the given ranking model placed the faulty method at the top. On the other hand, both wef and MAP values can be computed for localisation of a single fault. Therefore, unlike *acc@n*, wef and MAP for each `Defects4J` project is the average of all wef and MAP values over the faults belonging to the same project.

4.6 Tie-breaking

Ranking models generated by both FLUCCS and the eleven baseline SBFL formulæ produce ordinal scores. However, when converting these scores into rankings, ties often take place. To break these ties, We use *max* tie-breaker that ranks all tied elements with the lowest ranking. We use the `rankdata` function from `scipy.stats`, a Python module for statistical functions as well as probability distribution, to implement *max* tie breaker.

5 RESULTS AND ANALYSIS

5.1 RQ1. Effectiveness

Table 4 shows the performance of FLUCCS measured by using evaluation metrics described in Section 4.4. For GP^A_{min}, 106 faults (roughly 50% of all faults studied) are located at the top and 173 faults (82%) are placed within the top five. For GP^A_{med}, 108 faults (51%) and 170 faults (80%) are placed at the top and within the top five respectively. Although the result of GP^A_{med} has a slightly better result than GP^A_{min} for metric *acc*@1, GP^A_{min} outperforms GP^A_{med} for metric wef. Recall that the fitness function is the average ranking of faults in the test data sets; wef directly reflects the relative fitness (higher ranking results in lower wef), whereas *acc*@1 counts specific cases of produced rankings. Consequently, improved wef by GP^A_{min} can still result in worse *acc*@1.

The overall MAP values for both GP^A_{min} and GP^A_{med} are higher than 0.5. While *acc*@1 and wef focus on the method that has the highest rank, MAP concerns all faulty methods that consist a single `Defects4J` fault, communicating more complete views on the rankings of constituent methods. The observed overall MAP values are higher than those reported in fault localisation literature [3].

The results suggest that FLUCCS is more effective at localising faults when compared to baseline SBFL formulæ. The right column ("Without Method Level Aggregation") in Table 6 shows the results from the 11 baseline SBFL formulæ. The top six best performing SBFL formulæ are $ER1_a$, $ER1_b$, gp03, gp19, Ochiai, and Jaccard. Compared to these formulæ, GP^A_{min} places at least 49% and at most 54% more faults at the top (*acc*@1) than these formulæ. In terms of wef, GP^A_{min} has 12.8 while wef values for these baseline formulæ are ranged from 103.9 to 731.5. In terms of MAP metric, MAP values for all top six formulæ do not exceed 0.5, which GP^A_{min} exceeds: the values are ranged from 0.4005 to 0.4303.

The boxplots in Figure 3 present the overall wef results from eleven baseline SBFL formulæ as well as GP^A_{med} and GP^A_{min} (the y-axis is in log scale): FLUCCS outperforms all other baseline formulæ.

Table 5: Code and Change Metric Contribution: Metric values for the results of FLUCCS without using Code and Change Metrics as features

Technique	Project	Total	*acc*			*wef*		MAP
		Faults	@1	@3	@5	mean	std	
GP^S_{min}	Lang	60	29	50	55	1.8000	3.7408	0.6424
	Time	27	8	13	19	16.1852	32.2778	0.3234
	Math	96	29	62	74	31.8854	202.8204	0.4583
	Closure	27	9	15	17	79.4815	306.9720	0.3872
	Overall	210	75	140	165	32.3380	143.6969	0.4528
GP^S_{med}	Lang	60	30	51	56	2.0333	5.0232	0.6521
	Time	27	8	14	19	131.1481	584.0116	0.3398
	Math	96	39	67	79	61.5625	490.5831	0.5267
	Closure	27	10	17	18	107.7037	418.6214	0.4379
	Overall	210	87	149	172	75.6119	255.4931	0.4891

This provides an answer for **RQ1**: FLUCCS can be significantly more effective at ranking faulty methods at the top than existing SBFL formulæ.

5.2 RQ2. Code and Change Metric Contribution

To investigate the impact of using code and change metrics for localisation, we compare the results of FLUCCS with and without code and change metrics to each other, leaving other factors the same. Table 5 shows results of FLUCCS with only SBFL scores as features, named GP^S_{min} and GP^S_{med}. Compared with Table 4, which describes results using all features, GP^A_{min} and GP^A_{med} outperform GP^S_{min} and GP^S_{med} respectively by placing 41% and 24% more faults at the top rank (*acc*@1). In terms of wef, GP^A_{min} and GP^A_{med} reduce wasted effort by 61% and 60% respectively; MAP values of both GP^S_{min} and GP^S_{med} do not exceed 0.5.

Boxplots in Figure 4 show overall wef metric values of GP^A_{min}, GP^A_{med}, GP^S_{min}, and GP^S_{med}. For all projects except Closure-Compiler, GP^A_{min} and GP^A_{med} 4 place more faults at the top rank. Answer to **RQ2**: code and change metrics can make positive contribution to the effectiveness of fault localisation.

5.3 RQ3. Method Level Aggregation and Call Graph Propagation

Table 6 shows the impact of using method level aggregation for the 11 baseline SBFL formulæ. Among 11 baseline formulæ, the top six best performing formulæ are $ER1_a$, $ER1_b$, gp03, gp19, Ochiai, Jaccard. Method level aggregation can improve *acc*@1 values of these formulæ by 40% to 43%. However, for the other 5 formulæ, which place less than 3% of faults at the top, method level aggregation does not result in any improvement at all. These results indicate that method level aggregation can improve the accuracy of existing SBFL formulæ in some cases, but it cannot overcome the inherent limits of given SBFL formulæ. Answer for **RQ3**: method level aggregation can augment the accuracy of SBFL formulæ.

Table 7 shows the results of FLUCCS using 6 additional CGP versions of the change metric features; the boxplots in Figure 5 show the distribution of wef resulting from GP^A_{min}, GP^A_{med}, GP^{CGP}_{min}, and GP^{CGP}_{med}. Using the CGP version of change metric features allows us to place more faults at the top, but the improvement is not significant: 2% and 1% additional faults at the top for $GPCG_{min}$ and $GPCG_{med}$, respectively. The results also show that there is little improvements in wef and MAP, which show mixed trends. Answer for **RQ3**: we find only limited supporting evidence for our assumption about CGP in Section 2.4.

5.4 RQ4. Learning Algorithm

Table 8 presents the results of FLUCCS using linear rankSVM as the learning algorithm: SVM^A and SVM^S indicate the results of linear rankSVM with and without code and change metrics, respectively. Considering that the linear rankSVM is deterministic and requires a single run, we compare its results to those from the median performance GP models: GP^A_{med} and GP^S_{med}.

When all features are used, GP^A_{med} locates 9% more faults at the top and reduces wef by 73% compare to SVM^A. For MAP, GP^A_{med} produces an average over 0.5, SVM^A reports MAP of 0.4466. When code and change metrics are excluded, GP^S_{med} places 13% more faults at the top and reduces wef by 45% compared to SVM^S. However MAP values for both GP^S_{med} and SVM^S do not exceed 0.5, at 0.4891 and 0.4298 respectively.

Overall distributions of wef for both GP and linear rankSVM versions of FLUCCS are shown in boxplots in Figure 6. FLUCCS with GP ranks faulty methods higher than FLUCCS with linear rankSVM for all subject project except for Closure-Compiler. Answer for **RQ4**: GP as a learning-to-rank algorithm can outperform linear support vector machines.

6 THREATS TO VALIDITY

Threats to internal validity includes the extent to which the results of the empirical evaluation warrants the claims, such as the data integrity of training and test data we use, as well as the correctness of the tools. The spectrum data is collected using one of the most widely used coverage instrumentation tool, `JaCoCo`; similarly, both of the learning techniques used by FLUCCS are existing open source frameworks [4, 10, 21] that withstood public inspection and have been used in a variety of applications. We perform our statistical analysis using GNU R [32].

Threats to external validity includes factors that may affect how well the conclusions generalise. While the fact that `Defects4J` provides real world faults from open source projects may at least partially alleviate the risk of over-generalisation, our conclusions may be limited by the choice of language (Java), as well as factors in the studied projects that we failed to take note. The study is also limited by the factors that prevented us from establishing accurate ground truth regarding some faults in `Defects4J` as mentioned in Section 4.2.

Threats to construct validity includes how well the measurements we take are actually correlated to what they claim to measure. The use of absolute metrics, instead of the percentage based ones, helps communicating more realistic readings of the effort reduction possible with fault localisation.

7 RELATED WORK

Spectrum Based Fault Localisation has been one of the most widely studied technique for automated debugging [36]. While one of the

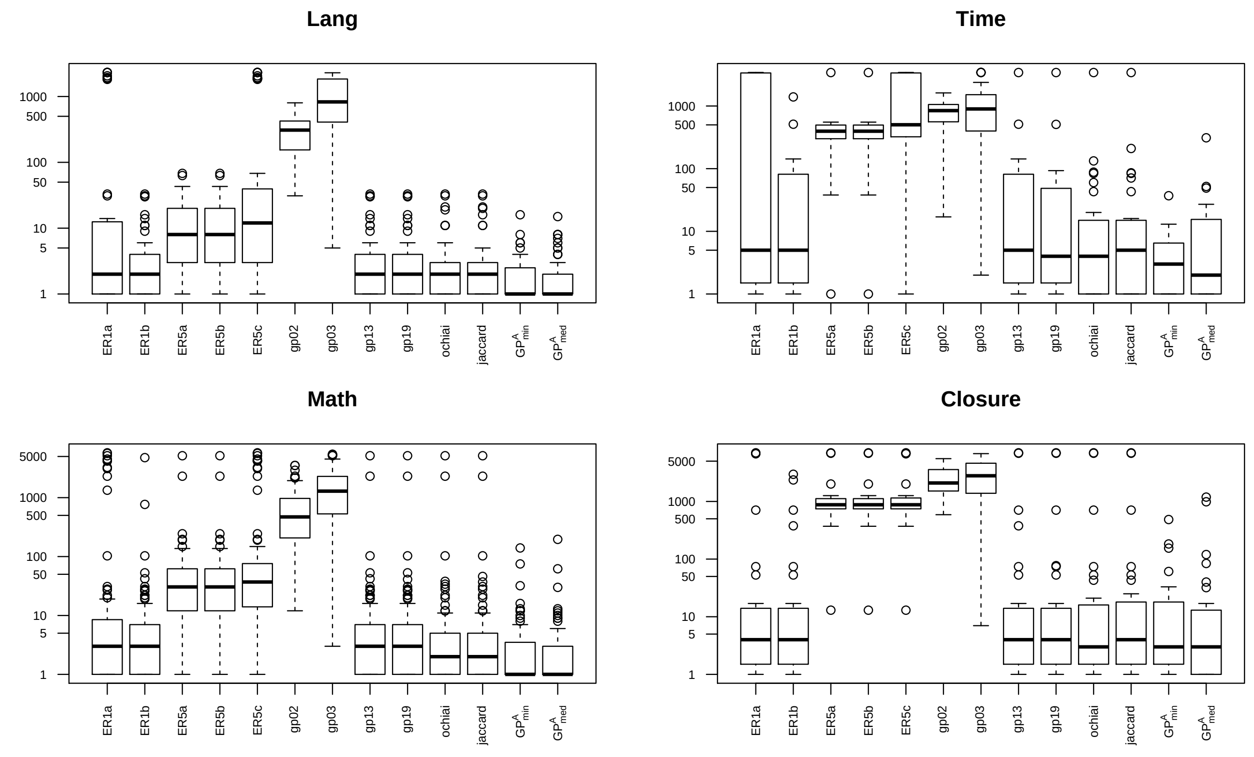

Figure 3: Boxplots of wef metric values from the 11 base SBFL formulæ as well as the minimum and the median wef from FLUCCS (GP^A_{min} and GP^A_{med}) that uses all the features. FLUCCS outperforms all baseline SBFL formulæ across all subject projects.

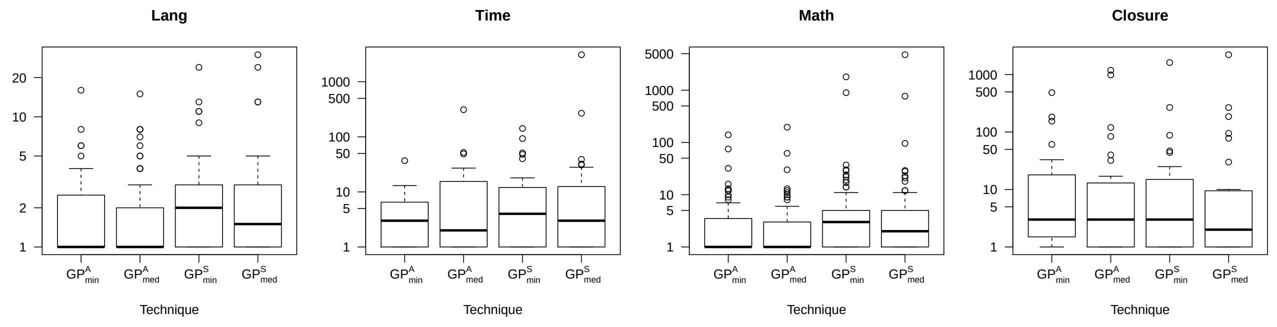

Figure 4: Boxplots of wef metric values from FLUCCS with (GP^A_{min} and GP^A_{med}) and without (GP^S_{min} and GP^S_{med}) the code and change metric features. The use of code and change metric features does improve the wef metric values.

earliest technique, Tarantula [17, 18], has been developed as a visual aid, it had been quickly applied as a ranking technique by which program elements are ranked according to their likelihood of being faulty. Many formulæ have been developed [6, 15, 29, 37] in order to empirically reduce the Expense metric, which measures the wasted effort (see Section 4.4) in percentage of the size of SUT. Later, it was pointed out that a percentage based evaluation metric can be unrealistic [30] when the SUT is sufficiently large. Consequently, recent work have adopted absolute measures, such as the accuracy ($acc@n$) or absolute wasted effort, for evaluation [3, 40], a trend which we follow in this paper.

Fault localisation has been approached as a learning problem in the literature. Yoo applied Genetic Programming to evolve SBFL formulæ from a set of known faults [41]. While evolving SBFL formulæ produced previously unknown maximal formulæ [39], there are also theoretically proven restrictions to what a single SBFL formula can achieve [43]. Instead of learning a complicated ranking model from raw spectrum data, FLUCCS takes existing SBFL scores as features, thereby accelerating the pace of learning.

Table 6: Baseline metric values from SBFL formulæ, with and without method level aggregation

Tech	Project	Total	With Method Level Aggregation						Without Method Level Aggregation					
		Faults	*acc*			*wef*		MAP	*acc*			*wef*		MAP
			@1	@3	@5	mean	std		@1	@3	@5	mean	std	
ER1a	Lang	60	27	39	43	410.1500	820.5376	0.5483	24	38	40	411.2833	819.1319	0.5102
	Time	27	7	11	14	1257.1111	1636.1855	0.5483	6	10	14	1257.4444	1634.3260	0.2450
	Math	96	28	54	66	460.5208	1341.5428	0.4079	18	45	52	473.2083	1364.5826	0.3224
	Closure	27	7	13	17	794.1852	2145.0155	0.3665	1	4	8	834.5185	2120.9147	0.1396
	Overall	210	69	117	140	730.4918	553.9450	0.4005	49	97	114	744.1137	543.0059	0.3043
ER1b	Lang	60	27	43	51	3.3167	6.9534	0.5986	24	41	47	4.8500	10.2515	0.5552
	Time	27	7	11	14	100.0370	273.2900	0.2825	6	10	14	134.5556	434.4126	0.2481
	Math	96	28	55	68	63.9896	490.0560	0.4309	18	46	53	93.8438	559.6703	0.3431
	Closure	27	7	13	17	244.2593	703.8727	0.3667	1	4	8	421.6296	1248.2000	0.1398
	Overall	210	69	122	150	102.9006	298.3463	0.4197	49	101	122	163.7197	513.6716	0.3215
ER5a	Lang	60	2	16	25	12.8333	14.7853	0.2259	2	17	26	12.4000	14.4271	0.2313
	Time	27	1	1	1	470.8889	598.3875	0.0242	1	1	1	470.5185	598.1112	0.0243
	Math	96	3	7	9	120.3021	566.7435	0.1043	3	7	9	162.4896	699.2417	0.1043
	Closure	27	0	0	0	1327.0741	1618.9638	0.0030	0	0	0	1293.0000	1614.1084	0.0031
	Overall	210	6	24	35	482.7746	668.8536	0.0894	6	25	36	484.6020	661.3180	0.0907
ER5b	Lang	60	2	16	25	12.8333	14.7853	0.2259	2	17	26	12.4000	14.4271	0.2313
	Time	27	1	1	1	470.8889	598.3875	0.0242	1	1	1	470.5185	598.1112	0.0243
	Math	96	3	7	9	120.3021	566.7435	0.1043	3	7	9	162.4896	699.2417	0.1043
	Closure	27	0	0	0	1327.0741	1618.9638	0.0030	0	0	0	1293.0000	1614.1084	0.0031
	Overall	210	6	24	35	482.7746	668.8536	0.0894	6	25	36	484.6020	661.3180	0.0907
ER5c	Lang	60	2	16	23	418.4667	816.5019	0.1916	2	17	24	417.7000	815.9973	0.1963
	Time	27	1	1	1	1465.6667	1483.1107	0.0232	1	1	1	1464.6667	1482.2877	0.0233
	Math	96	3	7	9	494.8438	1330.5444	0.0944	3	7	9	504.9375	1354.3736	0.0944
	Closure	27	0	0	0	1540.6667	1908.1414	0.0029	0	0	0	1506.0370	1903.4222	0.0030
	Overall	210	6	24	33	979.9109	450.7244	0.0780	6	25	34	973.3353	448.2796	0.0792
gp02	Lang	60	0	0	0	324.8000	204.7810	0.0065	0	0	0	321.7333	204.1228	0.0067
	Time	27	0	0	0	830.6667	410.4455	0.0033	0	0	0	829.8519	409.8967	0.0033
	Math	96	0	0	0	728.5104	753.5455	0.0059	0	0	0	741.0521	751.5603	0.0057
	Closure	27	0	0	0	2534.6667	1400.8204	7e-04	0	0	0	2480.4444	1356.4094	7e-04
	Overall	210	0	0	0	1104.6609	523.7267	0.0041	0	0	0	1093.2704	504.0054	0.0041
gp03	Lang	60	0	0	1	984.0667	678.0185	0.0090	0	1	1	1227.8500	748.0572	0.0129
	Time	27	0	2	2	1163.8148	1025.0582	0.0395	1	2	2	1310.0370	1156.2839	0.0578
	Math	96	0	2	2	1662.9583	1438.4292	0.0083	0	2	2	1805.8542	1461.9089	0.0080
	Closure	27	0	0	0	3053.4444	1906.5481	0.0059	0	0	0	3181.5556	2024.2213	0.0014
	Overall	210	0	4	5	1716.0711	530.3286	0.0157	1	5	5	1881.3242	537.5675	0.0201
gp13	Lang	60	27	43	51	3.3167	6.9534	0.5829	24	41	47	4.8500	10.2515	0.5551
	Time	27	7	11	14	175.0000	644.1596	0.2820	6	10	14	176.3333	643.5138	0.2481
	Math	96	28	55	68	83.9479	570.1467	0.4149	18	46	53	129.6042	703.6969	0.3430
	Closure	27	7	13	17	559.8148	1805.1642	0.3435	1	4	8	600.5556	1787.1087	0.1397
	Overall	210	69	122	150	205.5198	754.7328	0.4058	49	101	122	227.8358	737.3557	0.3215
gp19	Lang	60	27	43	51	3.3167	6.9534	0.5829	24	41	47	4.8500	10.2515	0.5551
	Time	27	7	12	15	161.1481	646.1446	0.2974	6	11	15	162.4815	645.5012	0.2630
	Math	96	28	55	68	83.9375	570.1475	0.4149	18	46	53	129.5938	703.6982	0.3430
	Closure	27	7	13	17	548.6667	1807.1871	0.3439	1	4	8	590.0741	1789.3828	0.1403
	Overall	210	69	123	151	199.2672	755.5713	0.4098	49	102	123	221.7498	738.2570	0.3215
ochiai	Lang	60	27	47	52	2.9500	6.5024	0.6054	25	42	48	4.4500	10.0057	0.5577
	Time	27	8	11	16	144.1481	643.0917	0.3120	5	10	16	147.2222	642.2084	0.2612
	Math	96	29	60	75	82.5729	570.3079	0.4610	19	50	58	129.0521	703.7914	0.3689
	Closure	27	7	14	17	547.5556	1807.4845	0.3430	1	4	7	595.0000	1787.9405	0.1365
	Overall	210	71	132	160	194.3067	755.9968	0.4303	50	106	129	218.9311	737.8993	0.3311
jaccard	Lang	60	26	46	53	3.0667	6.6505	0.5958	24	42	49	4.5333	10.0805	0.5487
	Time	27	8	11	14	147.6667	643.0826	0.3073	5	9	14	151.0000	642.0334	0.2527
	Math	96	29	60	74	82.7188	570.2949	0.4597	19	50	57	128.7292	703.8327	0.3673
	Closure	27	7	13	16	547.8519	1807.3970	0.3323	1	4	7	596.8889	1787.4509	0.1354
	Overall	210	70	130	157	195.3260	755.9088	0.4238	49	105	127	220.2878	737.6613	0.3260

Xuan and Montperrus combined 25 different SBFL formulæ, by taking a linear weighted sum of formulæ that score above learnt threshold values [40]. More recently, Le et al. added changes made to invariants (extracted using Daikon [8]) as an additional feature to the same set of 25 SBFL formulæ [3], and used linear rankSVM to learn the ranking model. SBFL has also been augmented by Information Retrieval based localisation techniques, using the linear weighted sum approach [20].

While FLUCCS also learns its ranking model from multiple SBFL formulæ as well as additional features, its use of Genetic Programming as the learning mechanism allows non-linear models. It should

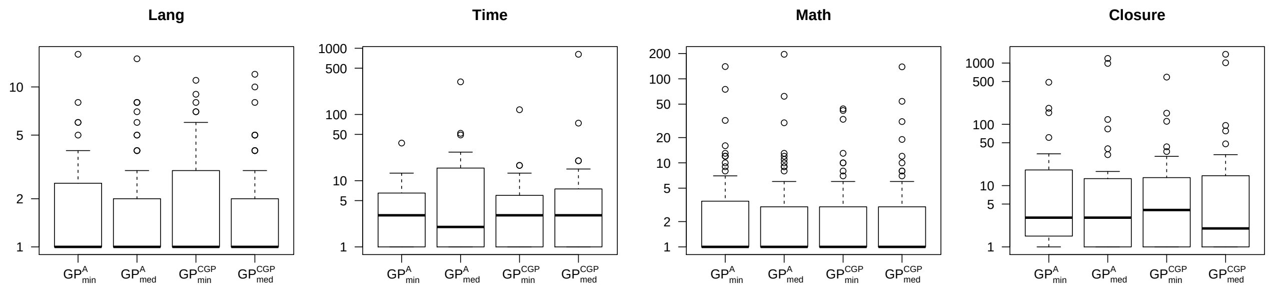

Figure 5: Boxplots of wef metric values from FLUCCS with (GP^{CGP}_{min} and GP^{CGP}_{med}) and without (GP^A_{min} and GP^A_{med}) the Call Graph Propagation. The use of CGP versions of features does not have significant impact on the effectiveness of FLUCCS.

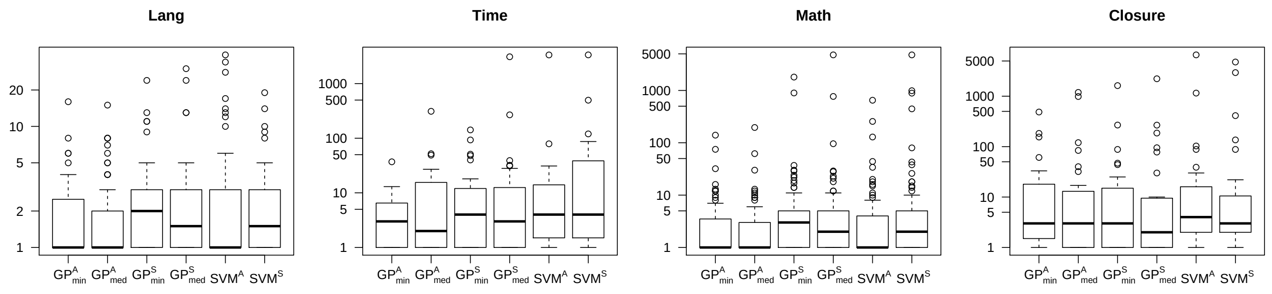

Figure 6: Boxplots of wef metric values from FLUCCS using GP (GP^A_{med} and GP^S_{med}) and linear rankSVM (SVM^A and SVM^S). Overall, results obtained using Genetic Programming tend to be better than those obtained using linear rankSVM.

Table 7: Metric values from FLUCCS with Call Graph Propagation. Compared to the results without CGP in Table 4, there is little improvement.

Technique	Project	Total	*acc*			*wef*		MAP
		Faults	@1	@3	@5	mean	std	
$GPCG_{min}$	Lang	60	36	51	54	1.2667	2.1975	0.6784
	Time	27	9	15	18	7.8889	21.9027	0.3950
	Math	96	54	74	83	2.5000	6.9267	0.6085
	Closure	27	9	13	16	37.6296	113.5560	0.3748
	Overall	210	108	153	171	12.3213	52.2859	0.5142
$GPCG_{med}$	Lang	60	35	51	57	1.1500	2.1512	0.6979
	Time	27	9	15	19	36.3704	152.3775	0.3838
	Math	96	55	76	84	3.6562	15.2581	0.6239
	Closure	27	10	17	18	99.7407	316.1949	0.4304
	Overall	210	109	159	178	35.2293	146.5049	0.5340

Table 8: Metric values from FLUCCS using linear rankSVM as learning to rank algorithm. Compared to the results in Table 4 (GP^A_{med}) and Table 5 (GP^S_{med}), results from FLUCCS with rankSVM are outperformed.

Technique	Project	Total	*acc*			*wef*		MAP
		Faults	@1	@3	@5	mean	std	
SVM^A	Lang	60	35	47	49	3.2667	7.6482	0.6593
	Time	27	7	13	16	132.9630	630.7210	0.2956
	Math	96	55	69	79	13.4062	71.4409	0.5911
	Closure	27	2	11	16	300.0370	1255.6878	0.2406
	Overall	210	99	140	160	112.4182	581.3569	0.4466
SVM^S	Lang	60	30	48	55	1.7333	3.2397	0.6432
	Time	27	7	11	15	158.8519	634.1114	0.2919
	Math	96	35	58	73	78.1250	502.3535	0.4786
	Closure	27	5	14	15	311.8148	1035.6120	0.3055
	Overall	210	77	131	158	137.6313	425.8163	0.4298

also be noted that, while invariant change feature is capable of capturing changes in program semantic, its extraction is much more expensive than the code and change metrics FLUCCS uses. FLUCCS also benefits from the method level aggregation of SBFL scores (described in Section 2.3).

8 CONCLUSION

We present FLUCCS, a fault localisation technique which learns how to rank program element based on existing SBFL formulæ and code and change metric. FLUCCS employs existing SBFL formulæ as features of the learning problem, instead of using raw spectrum data, reducing the effort to learn what is already known. FLUCCS is the first technique to use code and change metrics for fault localisation, connecting automated debugging to the field of defect prediction for the first time.

The empirical evaluation of FLUCCS, using real world faults and code history from `Defects4J` repository, shows that FLUCCS can be an effective fault localisation technique, placing 106 out of 210 faults at the top, and 173 out 210 faults within the top 5 places.

Our research on this paper adopts defect proneness prediction into fault localisation by extending SBFL with code and change metrics. Future work will investigate the use of various other learning algorithms as well as different sets of feature sets.

REFERENCES

[1] 2016. JaCoCo. http://www.eclemma.org/jacoco/. (2016). http://www.eclemma.org/jacoco/
[2] R. Abreu, P. Zoeteweij, and A.J.C. van Gemund. 2009. Spectrum-Based Multiple Fault Localization. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*. 88–99.
[3] Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. 2016. 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)*. ACM, New York, NY, USA, 177–188.
[4] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: A Library for Support Vector Machines. *ACM Trans. Intell. Syst. Technol.* 2, 3, Article 27 (May 2011), 27 pages.
[5] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Machine Learning* 20, 3 (1995), 273–297. http://dx.doi.org/10.1007/BF00994018
[6] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. 2005. Lightweight bug localization with AMPLE. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging (AADEBUG'05)*. ACM, New York, NY, USA, 99–104.
[7] Marco D'Ambros, Michele Lanza, and Romain Robbes. 2010. An extensive comparison of bug prediction approaches. In *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, 31–41.
[8] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 1999. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *Proceedings of the 21st International Conference on Software Engineering (ICSE-99)*. ACM Press, NY, 213–225.
[9] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. 2009. A Genetic Programming Approach to Automated Software Repair. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation (GECCO '09)*. ACM, New York, NY, USA, 947–954.
[10] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. 2012. DEAP: Evolutionary Algorithms Made Easy. *Journal of Machine Learning Research* 13 (July 2012), 2171–2175.
[11] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Trans. Softw. Eng.* 39, 2 (Feb. 2013), 276–291.
[12] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *PLDI*. 213–223.
[13] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 bugs for $8 Each. In *Proceedings of the 34th International Conference on Software Engineering*. 3–13.
[14] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. 2012. Bug Prediction Based on Fine-grained Module Histories. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 200–210.
[15] Tom Janssen, Rui Abreu, and Arjan J. C. van Gemund. 2009. Zoltar: A Toolset for Automatic Fault Localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*. IEEE Computer Society, Washington, DC, USA, 662–664.
[16] James A. Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th International Conference on Automated Software Engineering (ASE2005)*. ACM Press, 273–282.
[17] James A. Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*. ACM, New York, NY, USA, 467–477.
[18] James A. Jones, Mary Jean Harrold, and John T. Stasko. 2001. Visualization for Fault Localization. In *Proceedings of ICSE Workshop on Software Visualization*. 71–75.
[19] René Just, Darioush Jalali, and Michael D. Ernst. 2014. 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)*. ACM, New York, NY, USA, 437–440.
[20] Tien-Duy B. Le, Richard J. Oentaryo, and David Lo. 2015. Information Retrieval and Spectrum Based Bug Localization: Better Together. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 579–590.
[21] Ching-Pei Lee and Chih-Jen Lin. 2014. Large-scale Linear Ranksvm. *Neural Comput.* 26, 4 (April 2014), 781–817.
[22] Hua Jie Lee. 2011. *Software debugging using program spectra*. Ph.D. Dissertation. University of Melbourne.
[23] Taek Lee, Jaechang Nam, DongGyun Han, Sunghun Kim, and Hoh Peter In. 2011. Micro Interaction Metrics for Defect Prediction. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, New York, NY, USA, 311–321.
[24] Tie-Yan Liu. 2009. Learning to rank for information retrieval. *Foundations and Trends in Information Retrieval* 3, 3 (2009), 225–331.
[25] Philip McMinn. 2004. Search-based Software Test Data Generation: A Survey. *Software Testing, Verification and Reliability* 14, 2 (June 2004), 105–156.
[26] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayşe Bener. 2010. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering* 17, 4 (2010), 375–407.
[27] R. Moser, W. Pedrycz, and G. Succi. 2008. 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*. 181–190.
[28] Nachiappan Nagappan and Thomas Ball. 2005. Use of Relative Code Churn Measures to Predict System Defect Density. In *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*. ACM, New York, NY, USA, 284–292.
[29] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A model for spectra-based software diagnosis. *ACM Transactions on Software Engineering Methodology* 20, 3, Article 11 (August 2011), 32 pages.
[30] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers?. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA 2011)*. ACM, New York, NY, USA, 199–209.
[31] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. 2008. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`. (With contributions by J. R. Koza).
[32] R Core Team. 2015. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. https://www.R-project.org/
[33] M. Renieres and S.P. Reiss. 2003. Fault localization with nearest neighbor queries. In *Proceedings of the 18th International Conference on Automated Software Engineering*. 30 – 39.
[34] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. 2013. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*. ACM, New York, NY, USA, 314–324.
[35] G. Tassey. 2002. *The economic impacts of inadequate infrastructure for software testing*. Planning Report 02-3.2002. National Institute of Standards and Technology.
[36] W. E. Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (August 2016), 707.
[37] W. Eric Wong, Yu Qi, Lei Zhao, and Kai-Yuan Cai. 2007. Effective Fault Localization using Code Coverage. In *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 01 (COMPSAC '07)*. IEEE Computer Society, Washington, DC, USA, 449–456.
[38] Xiaoyuan Xie, Tsong Yueh Chen, Fei-Ching Kuo, and Baowen Xu. 2013. A Theoretical Analysis of the Risk Evaluation Formulas for Spectrum-based Fault Localization. *ACM Transactions on Software Engineering Methodology* 22, 4, Article 31 (October 2013), 40 pages.
[39] Xiaoyuan Xie, Fei-Ching Kuo, Tsong Yueh Chen, Shin Yoo, and Mark Harman. 2013. Provably Optimal and Human-Competitive Results in SBSE for Spectrum Based Fault Localisation. In *Search Based Software Engineering*, Günther Ruhe and Yuanyuan Zhang (Eds.). Lecture Notes in Computer Science, Vol. 8084. Springer Berlin Heidelberg, 224–238.
[40] Jifeng Xuan and M. Monperrus. 2014. Learning to Combine Multiple Ranking Metrics for Fault Localization. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME 2014)*. 191–200.
[41] Shin Yoo. 2012. Evolving Human Competitive Spectra-Based Fault Localisation Techniques. In *Search Based Software Engineering*, Gordon Fraser and Jerffeson Teixeira de Souza (Eds.). Lecture Notes in Computer Science, Vol. 7515. Springer Berlin Heidelberg, 244–258.
[42] Shin Yoo and Mark Harman. 2012. Regression Testing Minimisation, Selection and Prioritisation: A Survey. *Software Testing, Verification, and Reliability* 22, 2 (March 2012), 67–120.
[43] Shin Yoo, Xiaoyuan Xie, Fei-Ching Kuo, Tsong Yueh Chen, and Mark Harman. 2014. *No Pot of Gold at the End of Program Spectrum Rainbow: Greatest Risk Evaluation Formula Does Not Exist*. Technical Report RN/14/14. University College London.