Empirical Evaluation of Fault Localisation Using Code and Change Metrics

Jeongju Sohn and Shin Yoo

Abstract—Fault localisation aims to reduce the debugging efforts of human developers by highlighting the program elements that are suspected to be the root cause of the observed failure. Spectrum Based Fault Localisation (SBFL), a coverage based approach, has been widely studied in many researches as a promising localisation technique. Recently, however, it has been proven that SBFL techniques have reached the limit of further improvement. To overcome the limitation, we extend SBFL with code and change metrics that have been mainly studied in defect prediction, such as size, age, and churn. FLUCCS, our fault learn-to-rank localisation technique, employs both existing SBFL formulæ and these metrics as input. We investigate the effect of employing code and change metrics for fault localisation using four different learn-to-rank techniques: Genetic Programming, Gaussian Process Modelling, Support Vector Machine, and Random Forest. We evaluate the performance of FLUCCS with 386 real world faults collected from `Defects4J` repository. The results show that FLUCCS with code and change metrics places 144 faults at the top and 304 faults within the top ten. This is a significant improvement over the state-of-art SBFL formulæ, which can locate 65 and 212 faults at the top and within the top ten, respectively. We also investigate the feasibility of cross-project transfer learning of fault localisation. The results show that, while there exist project-specific properties that can be exploited for better localisation per project, ranking models learnt from one project can be applied to others without significant loss of effectiveness.

Index Terms—Fault Localisation, SBSE, Genetic Programming

1 Introduction

As the scale of software systems grows more and more, the number of faults in them has increased dramatically, burdening human developers with a tremendous amount of time and effort demanded to debug them. Automated test generation [1], [2], [3] aims to relieve the burdens of debugging by reducing the cost of test generation. However, even with this approach, a large part of debugging is still left to human developers, taking up to 80% of total software cost [4]. Consequently, there is an urgent need to further automate the debugging process. Automated Program Repair (APR) [5], [6] has been proposed to reduce the debugging cost by generating patches for the detected faults automatically. Nevertheless, APR techniques still require the information regarding the location of faults in order to focus their effort in the large search space for patches. Fault Localisation is an act of identifying the locations of the faults. Manual inspection of all program elements is a tedious and time-consuming task [7]. Spectrum Based Fault Localisation (SBFL) is a branch of fault localisation techniques that has received much attention due to its effectiveness and simplicity [8], [9]. SBFL takes both coverage and pass/fail information of individual test cases, and uses a risk evaluation formula to compute the likelihood of each program element being responsible for the observed failure. The likelihood is often referred to as the *suspiciousness score*: the higher the score is, the more likely the program element is to be responsible for the observed failure. SBFL uses these scores to rank program elements in descending order. The expected use case is that, by examining program elements following the ranking order, human developers will discover the faulty element faster than when following the order dictated by the program structure [10].

SBFL has been in the spotlight for quite some time [11], [12], [13], [14], [15]. Recently, however, limitations of SBFL have been pointed out both empirically and theoretically. Parnin and Orso conducted empirical human evaluation of SBFL in practice. Their evaluations showed that the claimed helpfulness of the formulæ does not hold in practice [16]. In particular, Parnin and Orso pointed out the inadequacy of the Expense metric as an evaluation of the effectiveness of the formulæ. Expense measures the amount of effort wasted on inspecting non-faulty program elements, i.e., the number of examined non-faulty ones, before encountering the very first faulty one, in percentage. Since it is a percentage, it fails to accurately measure the effort as it does not reflect the size of the System Under Test (SUT). For example, a single digit value for Expense may appear impressive. However, when applied to a large projects with millions of program elements, even 1% Expense is practically useless. To overcome the ambiguity of the Expense metric, Parnin and Orso recommended the use absolute ranking, which directly reflects the actual amount of efforts required from the human developers. Theoretically, the hierarchy between different formulæ, as well as the maximality of some formulæ, have been proven [8]: a formula is maximal if it is guaranteed to be better or at least equivalent to other formulæ against arbitrary faults and test suites.

Recently, multiple techniques that use multiple existing SBFL formulæ instead of a single one have been pro-

- *Jeongju Sohn and Shin Yoo are with the School of Computing, Korea Advanced Institute of Science and Technology, Daejon, Republic of Korea E-mail: kasio555@kaist.ac.kr, shin.yoo@kaist.ac.kr*
- *This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (Grant No. NRF-2016R1C1B1011042).*

posed [17], [18]. This approach seeks to overcome the limitation of a single SBFL formula by combining multiple formulæ to rank program elements. Moreover, a technique to explore other distinctive patterns of faults other than coverage, such as difference in program invariants, has been suggested [18]. A subsequent proof showing that there is no single SBFL formula that can dominate all the other formulæ [19] strengthen the justification for this new research direction.

FLUCCS is a learn-to-rank fault localisation technique based on Genetic Programming (GP), designed to address the known limitations of SBFL [20]. Instead of manually designing formulæ, FLUCCS learns to rank program elements according to their likelihood of being the root cause of the observed failure using Genetic Programming as the default learning mechanism. To overcome the theoretical limitations of pure SBFL approaches, FLUCCS uses multiple SBFL formula scores as input while learning to rank program elements. Moreover, FLUCCS also uses code and change metrics, such as age, complexity, and churn. This is based on the insight that there exists a large overlap between Defect Prediction (DP) and Fault Localisation (FL): DP is essentially trying to locate yet-to-happen faults, whereas FL seeks to locate faults *a posteriori*. To overcome the practical limitations, FLUCCS introduces method level aggregation to improve the accuracy of method level fault localisation, and use absolute ranking based evaluation.

We empirically evaluate FLUCCS with 386 real world faults collected from `Defects4J` repository [21]. Method level localisation results from FLUCCS have been compared against existing state-of-the-art SBFL techniques. To assess the contributions made by the use of the code and change metrics, we compare the results of FLUCCS with code and change metrics to the results of FLUCCS without them, using multiple learn-to-rank algorithms. Out of 386 faults, FLUCCS with the code and change metrics places 144 faults at the top, significantly outperforming the state-of-the-art SBFL formulæ. Comparison between the results of FLUCCS with and without code and change metrics indicates that a large portion of improvement made by FLUCCS originates from the use of these metrics, regardless of the learning algorithm used.

The technical contributions of this paper are follows:

- We present FLUCCS, a fault localisation technique that learns to rank program elements using Genetic Programming. FLUCCS uses existing SBFL formulæ as well as code and change metrics as input [1]. The results show that code and change metrics, traditionally studied in the context of defect prediction, can effectively complement existing SBFL techniques for more accurate localisation.
- We empirically evaluate FLUCCS using 386 real world faults from `Defects4J`. FLUCCS ranks 37% of the studied faults at the top, and about 79% of the studied faults within the top ten 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 50% more faults at the top.

This paper is an extension of our conference paper [20] that introduced FLUCCS. It has been extended with the following contributions:

- We have increased the number of studied faults from 210 to 386, by incorporating the faults that have been newly added in the version 1.2.0 of `Defects4J` benchmark [21].
- We have added two more learn-to-rank algorithms, Gaussian Process and Random Forest, in addition to the algorithms studied in the original conference paper, Genetic Programming and linear Support Vector Machine.
- We have performed a quantitative analysis of the relative feature for SBFL scores as well as code and change metrics, using Random Forest and Support Vector Machine.
- We have conducted both the cross-project learning and gradual cross-project learning, to evaluate the generalisability of our learn-to-rank approach. With gradual cross-project learning, we start training our learn-to-rank models with faults from other projects, but gradually replace them with faults from the project of interest, reflecting a real world adoption scenario.

The rest of the paper is organized as follows: Section 2 formulates fault localisation as the learning to rank problem and introduces the features used in FLUCCS. 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 7 discusses the potential threats to validity. Section 8 presents the related work and Section 9 concludes.

2 Features for Learning-to-Rank Fault Localisation

Fig. 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 the 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 the 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.

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

1. FLUCCS and the data used for the empirical evaluation are made available at https://bitbucket.org/teamcoinse/fluccs.

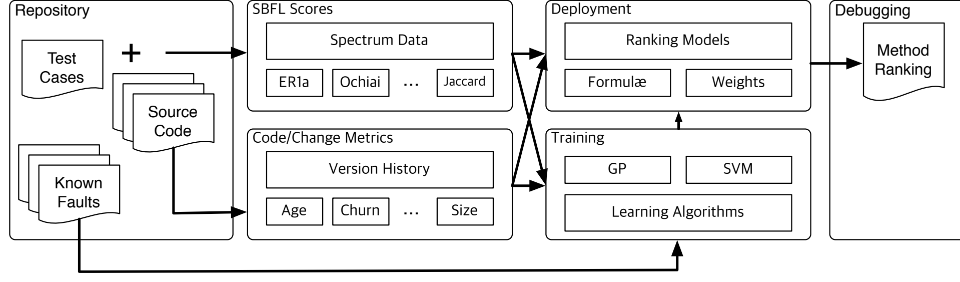

Fig. 1: Overall Architecture of FLUCCS

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 and failing test cases that do not execute the given structural element. SBFL formulæ tend to assign higher risk scores to elements with 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$_2$	$2(e_f + \sqrt{n_p}) + \sqrt{e_p}$
Ochiai	$\frac{e_f}{\sqrt{(e_f+n_f)(e_f+e_p)}}$	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$_{19}$	$e_f\sqrt{\lvert e_p - e_f + n_f - n_p\rvert}$
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	$\frac{2e_f+2n_p}{2e_f+2n_p+n_f+e_p}$
Wong2	$e_f - e_p$	Anderberg	$\frac{e_f}{e_f+2e_p+2n_f}$
Wong3	$e_f - h, h = \begin{cases} e_p & \text{if } e_p \leq 2 \\ 2 + 0.1(e_p - 2) & \text{if } 2 < e_p \leq 10 \\ 2.8 + 0.001(e_p - 10) & \text{if } e_p > 10 \end{cases}$		
Ochiai2	$\frac{e_f n_p}{\sqrt{(ef+e_p)(n_f+n_p)(e_f+n_f)(e_p+n_p)}}$		
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 [17], [18], while eleven formulæ have been proven to be maximal [22].

2.2 Code and Change Metrics

Many code and change metrics have been studied in relation to defect proneness [23], [24], [25], [26]. We expect these features to provide additional guidance towards the faulty program elements, and have adopted three categories of code and change metrics as features of our learning-to-rank approach: age, churn, and complexity. In total, FLUCCS uses the following six metrics, in their normalised form.

2.2.1 Age

Age simply measures how long a given program element has existed in the code base [24]. We consider two methods in two consecutive version to be the same method if their signatures match. Based on this definition of method identity, we define two types of method age:

- **min age**: the number of commits between the faulty version and the last commit that modified any part of the method under consideration.
- **max age**: the number of commits between the faulty version and the first commit that introduced the signature of the method under consideration.

Both age metrics aim to measure how mature the method under consideration is. The min_age is related to recent changes made to *how* the method operates, represented by the statements in the method body, whereas the max_age relates to *what* the method is supposed to do, which is represented by its method signature [27], [28].

2.2.2 Churn

Churn metric measures how frequently a given program element has been modified; it has been shown to be correlated with the fault density [23]. 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 version where the structural element first appeared. 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 [25]. 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 reflects the internal complexity.
- Size: this has been used by much of the defect prediction work in the literature as a surrogate for code complexity [25], [29], [30]. We use both LoC (Lines

of Code) and the number of compiled Java Bytecode instructions.

We do acknowledge that code complexity is difficult concept to measure: we deliberately chose metrics that can be simply and directly measured from the source code. Future study will investigate more sophisticated complexity metrics.

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 Fig. 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 $\frac{1}{1+0+2} = 0.333$. The method util and its line 12 share the same spectrum as well as scores, making it impossible to differentiate util and testMe. However, for line 4, the spectrum (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 }
```

Fig. 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.

3 Learning Algorithms

Learning to rank is a technique that uses machine learning to construct ranking models for an information retrieval system [31]. It intends 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. Learning-to-rank approaches can be categorized by their input representation and loss function: pointwise, pairwise, and listwise [31].

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, they aim 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 others, 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 pointwise and pairwise approaches. We consider the listwise approach to be inappropriate, because the ideal rankings in the training data are mostly all tied (i.e. zero for not faulty ones). For the pointwise approach, we choose Genetic Programming, which is the default learning mechanism of FLUCCS, Gaussian Process Modelling, and Random Forest. For the pairwise approach, we choose linear rankSVM.

3.1 Genetic Programming

We use Genetic Programming (GP) as a symbolic regression tool to learn the ranking models: it evolves a ranking function that takes features and produces 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 [32] and has the benefit of being able to generate non-linear ranking models.

3.2 Gaussian Process Modelling

Gaussian Process Modelling (GPM) is a collection of random variables, any finite number of which have a joint Gaussian distribution [33]. Let $x \in \mathbb{R}_d$ represents a set of d random variables, and f be a function that maps $x \in \mathbb{R}_d$ to $y \in \mathbb{R}$. GPM assumes that, for any finite number of x, $\{x_1, x_2, ..., x_n\}$, and the corresponding random variable set, $\{f(x_1), f(x_2), ..., f(x_n)\}$, respectively have joint Gaussian distributions. This allows us to learn f by learning the joint distributions that minimises the differences between $f(x_i)$ and y_i where $1 \leq i \leq n$. In the context of FLUCCS, each data

point in the training data is a feature vector (x) of a method, containing n_F features used by FLUCCS. GPM performs logistic regression on pairs of a feature vector ($x_i \in \mathbb{R}_{n_F}$) and the corresponding label (y_i) in the training data: y_i is the label that denotes whether the corresponding method is faulty or not.

For an unseen data point, x', its predicted label, y', will be computed based on the similarity of x' to previous data points via a covariance function defined over them: FLUCCS uses *radial basis function* for the covariance. We treat the estimated label value, $f(x')$, as the likelihood of the corresponding method being faulty: the closer to 1.0 $f(x)$ is, the more likely the method is to be faulty. Finally, methods are ranked based on their $f(x')$ values.

3.3 Support Vector Machine

Ranking SVM is a variant of Support Vector Machine [34] algorithm that performs pairwise learning to rank. The performance of Ranking SVM varies with the choice of a kernel type. Commonly used types for the kernel are *linear*, *polynomial*, and *radial basis function*. Among those kernel types, we choose *linear* due to the computational burden of using *polynomial* and *radial basis function*; unlike other kernel types whose time required for learning increases dramatically with the large-scale data, *linear* kernel allows users to generate ranking models in reasonable time.

FLUCCS uses linear ranking SVM intending to learn linear coefficients to input features that collectively produce the fewest ordinal inversions. We assign an arbitrary rank of 1 to all faulty methods and 0 to all non-faulty methods. We subsequently rank them based on their distance to the separating hyper-plane in descending order: the farther away a method is from the hyperplane, the more suspicious it is. While being orders of magnitudes faster than GP, linear ranking SVM is restricted by the linearity of the ranking model and the inherent imbalance in fault localisation training data. From this point, we will call linear ranking SVM simply as SVM.

3.4 Random Forest

Random Forest (RF) [35] is an ensemble learning method that aims to overcome the problem of over-fitting to the training data, which is one of the major weaknesses of decision trees. Bootstrap aggregating or bagging is used in RF with an additional layer of randomness [35], [36]. It randomly samples multiple sub-training datasets with replacement and builds a single decision tree for each of them. For test data, outcomes of these decision trees are aggregated, e.g. averaging the decision values, for prediction.

FLUCCS uses Random Forest to perform regression on methods, using feature vectors as independent variables and the given labels, faulty (1) or non-faulty (0), as the dependent variable. Each single tree is a single, independent regression model: for final prediction, outcomes of all trees are aggregated using average. Similar to GPM, we treat the aggregated label estimation, i.e., the averaged output of RF, as our suspiciousness score, and use it to rank the methods.

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 GP runs and generate 30 ranking models per training dataset; we choose models with the best and the median GP fitness values for evaluation. 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 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 the 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 metrics contribute to the fault localisation?

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: How much does the Method Level Aggregation contribute to the localisation of faults?

Method Level Aggregation can be applied to any metric as long as the metric's values can be computed at the statement level and the definition of the metric at method level does not fundamentally differ from the one at statement level. In this paper, we apply Method Level Aggregation to spectrum-based metrics, SBFL scores.

To evaluate the impact of Method Level Aggregation on SBFL scores, 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.

RQ4. Algorithm Dependency: Does the choice of learning algorithm affect the effectiveness of FLUCCS?

First two research questions, RQ1 and RQ2, investigate the effectiveness of FLUCCS, using GP as a learning algorithm. To further generalise our claim that the use of code and change metrics will aid SBFL by finding and exploiting previously unknown traits of faults, we inspect three other

well-known machine learning algorithms in the place of GP: GPM, SVM, and RF.

We evaluate the effectiveness of these substitute algorithms by comparing their answers to RQ1 and RQ2 with each other. Corresponding versions of GP^A_{min}, GP^A_{med}, GP^S_{min}, and GP^S_{med} for GPM, RF, and SVM are referred as (GPM^A_{min}, GPM^A_{med}, GPM^S_{min}, GPM^S_{med}), (SVM^A, SVM^S) and (RF^A_{min}, RF^A_{med}, RF^S_{min}, RF^S_{med}) hereafter.

RQ5. Existence of project-specific traits that are effective for fault localisation: Does each project have any unique traits that might affect the effectiveness of fault localisation?

Performance of trained models heavily depends on whether these models are able to catch features or traits that can differentiate faulty ones from the others. As the purpose of each software differs, some projects might have unique traits of faults that are specific to them. Suppose, there is a project that does not allow to use certain TimeZone data and is failed whenever this TimeZone has been used. For this project, detecting the use of this specific TimeZone will be crucial in localisation of faults. For other projects where use of specific TimeZone does not matter, this detection has no use.

To find out whether there is any project-specific traits and whether their influence holds statistical significance, we compare the performance of FLUCCS with three different configurations of the training dataset: *mixed*, *other*, and *self*.

- ***mixed***: this setting does not distinguish where the faults come from; it randomly divides the entire dataset into ten folds for cross-validation.
- ***other***: for each project, we use a training dataset consisting of faults from one of the other projects, and a test dataset consisting of its own faults. We denote a training dataset consisting of faults from the project P as $other_P$.
- ***self***: every project has its own training and test dataset, consisting only of its own faults. These faults are divided into five folds: we use four folds as the training data, and the remaining one as the test data.

The *self* and *other* configurations are used to confirm the existence of project-specific traits and their influence on the effectiveness. *mixed*, the default setting used by FLUCCS, has been included as the baseline. Vargha-Delaney A_{12} statistic and Mann-Whitney U-test are used for statistic analysis.

RQ6. *Gradual* Cross-Project Training: When should we transit from cross-project to *self* configuration?

While there may exist project-specific traits that can be best exploited by the *self* configuration, it may not always be a feasible option, for example when the target project is in the early stage of development: there simply may not be a sufficient number of faults to learn from. Consequently, until a sufficient number of own faults are collected, cross-project learning using faults from other projects may be the only feasible option. Under this use case scenario, the critical question now is when to switch to *self* configuration.

To answer this question, we perform gradual cross-project learning. Let us define *others* to be the set of faults collected from other, non-self projects. For gradual cross-project learning, we vary the ratio between *others* and *self* datapoints, by increasingly replacing *others* datapoints with *self* datapoints. Initially, the training dataset consists of as many *others* datapoints as the size of four *self* folds: we evaluate the ranking model trained with this dataset using one *self* fold. Subsequently, we replace one fold of *others* datapoints in the training dataset with a *self* fold, and continue to evaluate the ranking model trained with the updated dataset that contains more *self* datapoints, until we eventually replace all *others* datapoints with *self* datapoints. We observe how this transition from *others* to *self* affects the performance of the trained model. The intermediate mixtures of training datasets are denoted by $gradual_{i/4}$, in which i is the number of *others* folds in the dataset (e.g., $gradual_{3/4}$ contains three parts *others* and one part *self* datapoints).

RQ7. Feature Importance: Which features contribute the most to ranking models?

FLUCCS uses 40 features, 33 from SBFL score metrics and seven from code and change metrics. To identify which features are useful in FLUCCS, we analyse feature importance for FLUCCS using Random Forest, which is one of the standard filtering technique for feature selection. The relative feature importance can be estimated from the average reduction of variance when splitting decision tree nodes based on a specific feature. We report the top ten most important features using *self* configurations (i.e., for each project) as well as the *mixed* configuration (i.e., for all studied project combined).

Furthermore, we directly measure how each of the top ten features affects the performance of the corresponding models. We randomly permutate the values of each of the top features, one at a time, and observe the differences in model performance. We assume that that, the more important a feature is, the more severely the performance of the model will be affected when the values for that feature are permutated.

TABLE 2: Subject software systems and their faults

Project	# Faults	Loc	# Methods	# Test cases
Commons Lang	63	9059–11490	1953–2408	1540–2295
Commons Math	105	4726–41344	1049–6668	817–4429
Joda-Time	26	12732–13270	3628–3802	3749–4041
Closure Compiler	131	30438–50523	4848–8880	2595–8443
Jfreechart	25	41075–51523	6578–8281	1586–2193
Mockito	36	2110–4385	747–1476	695–1399

4.2 Subjects

We use real world faults from `Defects4J` repository [21] to evaluate FLUCCS. Table 2 lists the subject programs. Our previous work used version 0.2.0 of `Defects4J`; we now use version 1.1.0, which contains an additional project, Mockito. A few issues in the previous version of FLUCCS have been addressed to increase the number of faults studied. We have changed the coverage measurement tool from `JaCoCo` [37] to `Cobertura` [38]. This allowed us to cover faults that have been excluded in the previous work[2].

2. `Jacoco` misses some statements when the normal sequence of statements is disturbed, resulting a inserted probe to be remained not executed. The official FAQ (http://www.eclemma.org/jacoco/trunk/doc/faq.html)

Remaining faults of *Closure*, which have been excluded in previous version of FLUCCS, are now added. Faults of *Chart*, which we failed to handle due to the revision id problem[3], are included by migrating to *git* from *svn* using *git-svn*. Finally, we incorporate 36 faults from *Mockito* that have been newly introduced in version 1.1.0.

In total, among 395 faults available in `Defects4J`, we use 386 faults, excluding 9 faults. This exclusion is due to some issues that prevent us establishing the ground truth about input features and locations of the real faults. Our filtering criteria are:

- **Scope of Faults:** We focus only on the methods that are parts of the given system under testing (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 fault lies on a method that overrides another method external to Commons-Lang. (Lang:1)
- **Restrictions on targeted methods:** FLUCCS performs method-level fault localisation. We only target methods that can be invoked directly by JVM instructions; this restriction leads FLUCCS to be unable to handle faults on class initialisation methods. (Lang:1, Math:1, Time:1, Mockito:1)
- **Compilation Failures:** Instead of working on codebase that `Defects4J` provides, FLUCCS works directly on the codebase where an actual fault was made. While compiling these versions of codebase using compiling tools of `Defects4J`, some of them were failed, preventing FLUCCS from further process such as coverage metric computation. (Chart:1, Mockito:1)
- **Missed partially covered branch:** FLUCCS considers a given method to be covered if *hits* attribute in xml coverage report is larger than zero. As *hits* for a branch element remains zero despite the fact that some of its conditions are covered, methods where only these branch are executed will be regarded as uncovered. Combining with the exclusion of uncovered methods, some of faults are excluded. (Closure:1)
- **Missed class name:** FLUCCS parses results of *git diff* between fixed and faulty version of codebase to set the ground truth: the id of faulty method. During parsing process, FLUCCS missed one faulty method with nested structure of class name in attempt to avoid ambiguous class name, integer, assigned for the instance. (Closure:1)

From the 386 faulty versions that we study, any methods that are not executed at all by test cases have 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.

3. revision ids provided by `Defects4J` are not available in the codebase provided by `Defects4J`: the revision ids are for *svn* and the codebase are maintained by *git*.

4.3 Configuration

4.3.1 Genetic Programming (GP)

We use DEAP [39], a Python evolutionary computation framework, to implement the GP version of FLUCCS. We use 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 initialized by the ramping method [40]; the maximum tree depth is eight and the algorithm stops after 100 generations. As described in Section 3, GP uses the ranks of known faulty methods as the fitness. Table 3 lists GP operators used by FLUCCS; for terminal nodes, we use variables corresponding to 40 features described in Section 2 plus a constant 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 for fitness evaluation per generation if the training set contains more than 30 faults: the training dataset of *mixed* configuration consists of 347-351 faults and only randomly sampled 30 faults are used. If no more than 30 faults compose the training dataset, we use all of them. We also adopt elitism, preserving the best 8 individuals from the parent generation into the generation of offspring.

4.3.2 Gaussian Process Modelling (GPM)

Due to the limited scalability of Gaussian process (N^3 scaling for training with N number of training data points), we use Sparse Gaussian Process [41]. Sparse Gaussian Process parametrises the covariance of Gaussian Process regression model by the locations of M ($M << N$) pseudo-input points, which are learnt by a gradient based optimisation. Additionally, we employ Gaussian Process for batch data to further reduce the computational burden [42], [43]. We use `GPy` [44], a Gaussian Process framework from the Sheffield machine learning group, to implement the GPM version of FLUCCS. We perform hyper-parameter optimisation to the number of pseudo-inputs (M) and the number of maximum iteration ($iter_{max}$), using `hyperopt` [45]. The batch size has been manually tuned to 3,000.

We optimise M and $iter_{max}$ as follows. Since M can be a value between 1 and the number of data points in the training data (N), and $iter_{max}$ can be any integer greater than 1, we firstly reduce the sample space of M and $iter_{max}$. Using a single fold from our ten-fold cross-validation, we manually narrow down M to the range between 10 and 50, and $iter_{max}$ between 10 and 300. Table 4 shows the results of the manual investigation. Within this reduced ranges, we subsequently perform hyper-parameter optimisation to derive 140 for M and 18 for $iter_{max}$.

We select Broyden-Fletcher-Goldfarb-Shanno algorithm (BFGS), an iterative function used to solve unconstrained non-linear optimisation problems, as the optimiser, and the radial basis function as the covariance function.

With GP, the fitness is computed from the aggregated ranks of faulty methods in faulty programs. However, each rank is computed independently, considering only a single fault. In contrast, GPM learns from multiple faults simultaneously. GPM receives the entire training dataset as a pair of a single feature matrix of size [the number of data points] by [the number of features], and a matching array of labels. Here, each data point corresponds to a single method, which can be from any faulty program in the training dataset. As a result, GPM can learn from the differences between feature vectors from different faulty programs.

TABLE 4: Variance of performance over the number of pseudo-input points and the maximum number of iterations. M indicates the number of pseudo-input points and $iter_{max}$ indicates the number of max iterations.

$iter_{max}$ \ M	10 *acc@1,3*	25 *acc@1,3*	50 *acc@1,3*	100 *acc@1,3*	300 *acc@1,3*
5	1, 5	9, 19	12, 20	1, 5,	10, 20
10	7, 13	11, 23	10, 20	12, 21	12, 23
30	10, 21	10, 19	13, 22	11, 22	12, 22
50	10, 21	12, 20	13, 21	12, 22	11, 22
75	10, 19	14, 23	12, 22	12, 22	11, 22
100	11, 20	12, 22	12, 22	12, 22	11, 22

4.3.3 Support Vector Machine (SVM)

We use version 2.11 of large-scale rankSVM [46], which depends on `libSVM` [47] for linear ranking SVM, with out-of-the-box default parameters. We choose rankSVM because of its support on linear ranking SVM for large scale data. rankSVM uses L2-regularized L2-loss kernel in default, and has been used to learn ranking models for fault localisation in the literature [18]. SVM takes the entire training dataset as a single matrix, without discriminating feature vector entries from different faulty programs, similarly to GPM.

4.3.4 Random Forest (RF)

We use `RandomForestRegressor` from version 0.19.1 of `sklearn` [48] using the default setting in `sklearn`; e.g., the number of individual trees is ten and Mean Squared Error (MSE), which aims to minimise variance in final leaves, is used to measure the quality of splits. Like GPM and SVM, RF also takes a single matrix of feature vectors and a corresponding labels without using sampling.

4.4 Evaluation Metrics

We use three metrics to analyse the performance of GP with new features following existing work [17], [18]. In particular, the use of accuracy (*acc@n*) and wasted effort (*wef*) conforms to the guideline from Parnin and Orso [16], 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, and 10 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. Similar with *acc@n*, we only use the highest rank of faulty program element when there are multiple of them.

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:

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

Unlike *acc@n* and *wef*, AP caters for the existence of multiple faulty program elements. Mean Average Precision (MAP) is the mean of AP values computed for a set of faults. We calculate MAP for faults from the same project.

4.5 Validation

To maximise the size of training dataset and also to avoid overfitting of GP, we use ten-fold cross validation for RQ1 to RQ4. Given the set of 386 faults, we divide fault data into ten folds, each comprises approximately 39 faults; RQ5 requires a different approach, as the number of available faults can be too small to perform a ten-fold cross validation for the *self* configuration with some projects (for example, $Time$ has only 26 faults and would yield validation sets composed of only a couple of faults). Consequently, for the *self* configuration, we use five-fold cross validation instead. For the *cross-project* configuration, we simply use faults of one project as a validation set and all faults from others as a training set.

We iterate FLUCCS with GP 30 times, resulting in 30 different ranking models. The same goes for GPM and RF. SVM is deterministic algorithm and therefore, does not require repetition. To summarise and compare overall results, evaluation metrics of rankings are aggregated per `Defects4J` project. Since *acc@n* is a counting metric, 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.

TABLE 5: Metric values showing the effectiveness of FLUCCS

Technique	Project	Total Faults	*acc* @1	@3	@5	@10	*wef* mean	std	MAP
GP^{A}_{min}	Lang	63	30	43	47	57	3.10	5.98	0.5654
	Math	105	49	63	69	78	21.69	123.75	0.4673
	Time	26	10	14	17	20	8.62	13.57	0.4075
	Closure	131	32	65	76	93	20.38	64.19	0.3469
	Chart	25	14	20	22	24	1.52	2.90	0.5654
	Mockito	36	9	18	23	32	5.36	9.71	0.3499
	Overall	386	144	223	254	304	14.50	75.25	0.4338
GP^{A}_{med}	Lang	63	31	46	50	58	2.33	3.80	0.6025
	Math	105	47	64	71	79	57.66	481.00	0.4707
	Time	26	9	14	16	19	134.38	636.50	0.4063
	Closure	131	35	64	80	97	29.98	100.81	0.3665
	Chart	25	15	21	24	24	1.16	2.34	0.6108
	Mockito	36	10	18	22	31	7.97	20.30	0.3523
	Overall	386	147	227	263	308	36.13	308.00	0.4507

5 Results and analysis

5.1 RQ1. Effectiveness

Table 5 shows the performance of FLUCCS measured by using evaluation metrics described in Section 4.4. For GP^{A}_{min}, 144 faults (roughly 37% of all faults studied) are located at the top and 304 faults (79%) are placed within the top ten. For GP^{A}_{med}, 147 faults (38%) and 308 faults (80%) are placed at the top and within the top ten respectively. Although the result of GP^{A}_{med} has a slightly better result than GP^{A}_{min} for metric *acc@n*, GP^{A}_{min} outperforms GP^{A}_{med} for metric *wef*, which is used as the fitness function of FLUCCS. Recall that the fitness function is the average ranking of faults in the test datasets; *wef* directly reflects the relative fitness (higher ranking results in lower *wef*), whereas *acc@n* 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 less than 0.5. While *acc@n* 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 [18].

The results suggest that FLUCCS is more effective at localizing faults when compared to baseline SBFL formulæ. The right column ("Without Method Level Aggregation") in Tables 7 and 8 shows the results from the 11 baseline SBFL formulæ. The top six best performing SBFL formulæ are $ER1_a$, $ER1_b$, gp13, gp19, Ochiai, and Jaccard. Compared to these formulæ, GP^{A}_{min} places at least 31% and at most 47% more faults at the top (*acc@1*).In terms of average *wef*, GP^{A}_{min} has 14.5 while the average *wef* for these baseline formulæ ranges from 47.5 to 1094.0. MAP values for all top six formulæ do not exceed 0.4, which GP^{A}_{min} exceeds: the values are ranged from 0.31396 to 0.37622.

The violinplots in Fig. 3 present the overall values of *wef* from eleven baseline SBFL formulæ as well as GP^{A}_{med} and GP^{A}_{min} $they{-}axisisinlogscale$: FLUCCS outperforms all other baseline formulæ. This provides an answer for

Answer to **RQ1**: FLUCCS outperforms the existing SBFL formulæ by locating at least 31% and at most 47% more faults without any wasted effort.

TABLE 6: Code and Change Metric Contribution: Metric values for the results of FLUCCS without using Code and Change Metrics as features

Technique	Project	Total Faults	*acc* @1	@3	@5	@10	*wef* mean	std	MAP
GP^{S}_{min}	Lang	63	26	38	44	48	3.70	5.01	0.5064
	Math	105	24	49	55	62	29.44	82.56	0.3203
	Time	26	8	12	18	20	69.06	299.25	0.3411
	Closure	131	35	61	77	95	19.94	64.88	0.3628
	Chart	25	11	17	20	21	3.28	5.56	0.5322
	Mockito	36	13	20	23	30	8.95	18.75	0.3823
	Overall	386	117	197	237	276	21.08	98.06	0.3860
GP^{S}_{med}	Lang	63	28	40	45	46	3.76	5.32	0.5322
	Math	105	27	48	57	62	73.06	480.50	0.3364
	Time	26	7	12	14	18	139.00	636.00	0.3015
	Closure	131	34	64	78	99	26.67	88.44	0.3582
	Chart	25	11	18	23	23	2.24	4.46	0.5313
	Mockito	36	13	20	22	28	9.50	22.78	0.3940
	Overall	386	120	202	239	276	39.94	306.75	0.3914

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 6 shows results of FLUCCS with only SBFL scores as features, named GP^{S}_{min} and GP^{S}_{med}. Compared to Table 5, which describes results using all features, GP^{A}_{min} and GP^{A}_{med} excel GP^{S}_{min} and GP^{S}_{med} respectively by placing 23% and 22.5% more faults at the first place (*acc@1*). In terms of *wef*, GP^{A}_{min} reduces the average by 45% while GP^{A}_{med} saves 10.5% in terms of average;MAP values of both GP^{S}_{min} and GP^{S}_{med} are less than 0.4. Violinplots in Fig. 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 and Mockito, GP^{A}_{min} and GP^{A}_{med} place more faults at the top rank.

Answer to **RQ2**: Code and change metrics improve the effectiveness of fault localisation, allowing to localise at least 22% more faults correctly and reduce *wef* at least 10.5% and at most 45%.

5.3 RQ3. Method Level Aggregation

Tables 7 and 8 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 50% to 68%. For other five formulæ, which place less than 3% of faults at the top, only marginal benefits are achieved; underlying performance of these formulæ does not change: poorly performed formulæ remain as they are. These results imply 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. Therefore, this technique can be only used as an additional tool for fault localisation.

Answer for **RQ3**: Some SBFL formulæ can benefit from using Method Level Aggregation, localizing 50% to 68% more faults, however, their fundamental shortcoming, performance barrier originated from the formulæ themselves, cannot be overcome by Method Level Aggregation alone.

5.4 RQ4. Algorithm Dependency

To inspect the effect of learning algorithm over the effectiveness of FLUCCS, we revisit RQ1 and RQ2 for three new

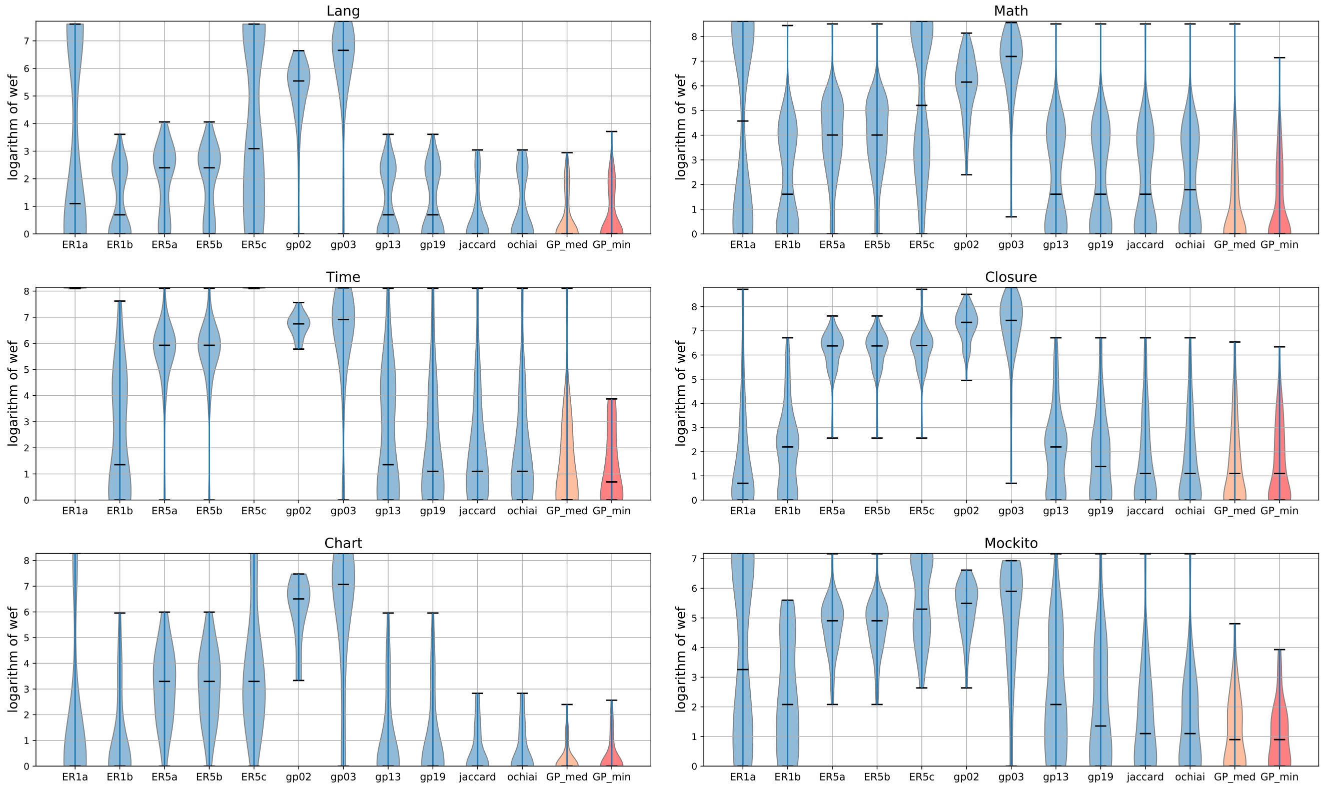

Fig. 3: Violinplots of *wef* values from the 11 base SBFL formulæ as well as the minimum and the median *wef* from FLUCCS (GP^A_{min} and GP^A_{med}) that uses all the features. FLUCCS outperforms all baseline SBFL formulæ across all subjects.

algorithms: GPM, SVM, and RF. Table 9 and violinplots in Fig. 5 present the results of them.

Let us first consider the effectiveness of localisation. Compared to the baseline results from SBFL formulæ in Tables 7 and 8, GPM and RF place 40% and 24.5% more faults at the top respectively, while SVM locates 10% more faults at the top. Considering the observed *acc@1* values, we contribute the increased *wef* observed in GPM and RF results to missed faults being ranked lower, thereby increasing the average *wef*. MAP for all three algorithms exceeds 0.4 while MAP for the baseline remains below 0.4. The benefit of code and change metrics is also clearly shown in Table 9: the left half, containing the results without code and coverage metric, shows lower *acc@n* and MAP as well as higher *wef* in general.

A closer look at the results reveals that relative performance of algorithms differs depending on the evaluation metric. For *acc@1*, GPM performs the best (GPM_{med}: 156), followed by GP (GP_{med}: 147), RF (RF_{med}: 134), and SVM (121). In contrast, when evaluated using the mean *wef*, the order is SVM (24.641), GP (GP_{med}: 36.125), GPM (GPM_{med}: 258.5), and RF (RF_{med}: 1125.0), placing the worst performer by *acc@1*, SVM, at the top. Violinplots in Fig. 5 further explain this phenomena: despite the high number of localised methods near the top of rankings, violinplots of GPM and RF tend to be longer and thicker at the bottom than other algorithms, suggesting missed faulty methods being ranked relatively lower.

We conjecture that these mixed results are due to the types of learn-to-rank algorithms: pointwise or pairwise. Fault localisation through GPM and RF are both pointwise approaches, as both focus on classifying *each* point's label correctly; GPM tries to reduce the variance of each point's predicted output, ordinal score, to the true label and RF attempts to minimise the homogeneity of the final leaves, where the state of a data point, 0 or 1, is determined based on the proximity of its ordinal score to each state value: to the closer one. On the other hand, SVM (or ranking SVM) is a pairwise approach, which aims to restore ordinal associations between data points; here, ordinal association between data points represents the relative likelihood to be faulty. It uses Kendall's tau to define its loss term [46]. Both Kendall's tau and *wef* take into account of ordinal relations, explaining SVM's preference on *wef*. GP uses average *wef* as fitness, which describes why average *wef* is smaller than any other methods for GP (GP_{min}) even though it is a pointwise approach.

Answer to **RQ4**: All three algorithms outperform the baseline, locating 10% to 40% more faults, and share universal benefits of employing code and change metrics. The comparison between four algorithms implies that depending on different aspect of accuracy, certain type of learning algorithm can be more fitting.

5.5 RQ5. Existence of project-specific traits that are effective for fault localisation

To analyse the results of the cross-project learning, we train seven different models using *mixed*, $other_{lang}$, $other_{math}$, $other_{time}$, $other_{closure}$, $other_{chart}$, and $other_{mockito}$: one of the *other* models, $other_P$, can double as *self* for project P. For all

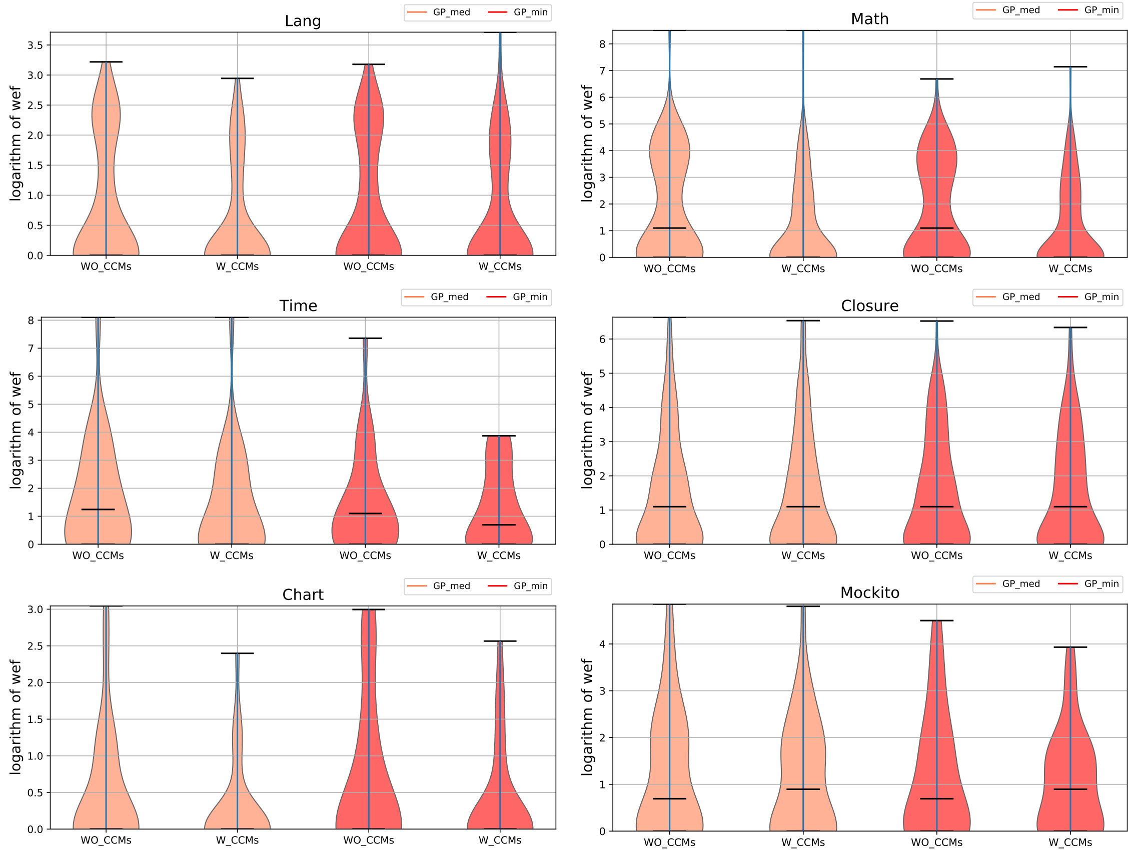

Fig. 4: Violinplots of *wef* values from FLUCCS with (GP^A_{min} and GP^A_{med}) and without (GP^S_{min} and GP^S_{med}) the code and change metric features. x-axis denotes whether code and change metrics have been used: with (W_CCMS) or without (WO_CCMS) code and change metrics. The use of code and change metric features does improve the *wef* values.

pairs between seven models, we perform Mann-Whitney U-test and measure Vargha-Delaney A_{12} effect size to compare the *wef* values of the pair: the *wef* values are computed from ranking models with the best fitness such as GP^A_{min}. Table 10 and Table 11 present the summaries of A_{12} statistic and U test results respectively.

The number of comparisons made is different for each pair of configurations. For example, between *mixed* and *self*, 24 comparisons exist: *mixed* against six different *self* configurations, multiplied by four learn-to-rank algorithms. We denote this as n_{total}. Table 10 and Table 11 show, out of n_{total}, how many times the configuration corresponding to the row was *better* than the other corresponding to the column. For A_{12} statistic, we consider the row configuration to be better than the column configuration if the effect size is greater than 0.5. For Mann-Whitney U-test, we consider the row configuration to be better than the column configuration if we can reject the null hypothesis, which is that the column configuration produces lower *wef* than the row configuration. The significance level is 0.05.

In general, *self* and *mixed* configurations tend to perform better than the *other* configurations. In terms of A_{12} statistic, *self* outperforms *other* configurations at least 16 out of 20 comparisons. The *mixed* configuration shows a similar pattern, also outperforming *other* configurations at least 16 times. Between *self* and *mixed*, *self* outperforms *mixed* configuration 16 times out of 24 comparisons. Table 11 shows that there is only one case of *self* outperforming *mixed* with statistical significance. In other cases where both *self* and *mixed* are concerned, the differences in their performances are not statistically significant. Both *self* and *mixed*, however, outperform *other* configurations with statistical significance.

Table 12 and Table 13 show the detailed results of A_{12} statistic and U-test for Time and Closure, respectively[4]. In both projects, *self* and *mixed* generally outperform *other* configurations, although in some cases such as $other_{mockito}$ with SVM, *other* succeeds to outperform *self* and *mixed*. Nevertheless, none of the *other* configurations outperforms *self* and *mixed* with statistical significance (U-test). Furthermore, while A_{12} statistic shows that there exists a performance difference between *self* and *mixed*, the p-values of U-test show that the differences are not statistically significant. Fig. 6 visualises the distributions of *wef* using different configurations.

Answer to **RQ5**: While the results of A_{12} suggest that *self* configuration outperforms *mixed*, the difference between their performance does not hold any statistical significance, which is shown by U-test. Based on this, we argue that, although there exist project-specific traits that may improve the localisation performance, it is possible to learn general ranking models that are as effective as the one that exploits the project-specific traits.

4. Results for other projects have been left out due to space restrictions: see https://coinse.kaist.ac.kr/projects/fluccs/journal_extension for full results.

TABLE 7: Baseline metric values from SBFL formulæ, with and without Method Level Aggregation

Tech	Project	Total Faults	With Method Level Aggregation *acc* @1	@3	@5	@10	*wef* mean	std	MAP	Without Method Level Aggregation *acc* @1	@3	@5	@10	*wef* mean	std	MAP
ER1a	Lang	63	24	30	33	34	726.50	894.50	0.4114	18	27	28	32	728.00	893.50	0.3489
	Math	105	19	37	44	49	2,146.00	2,218.00	0.2421	12	28	36	45	2,146.00	2,218.00	0.1897
	Time	26	0	0	0	0	3,370.00	46.47	0.0004	0	0	0	0	3,370.00	46.47	0.0004
	Closure	131	38	66	84	94	231.00	1,015.50	0.3833	16	31	45	60	264.75	1,020.50	0.2079
	Chart	25	11	17	18	20	548.50	1,259.00	0.4480	7	13	14	20	711.00	1,623.00	0.2932
	Mockito	36	6	8	12	16	545.00	583.50	0.2356	5	9	10	13	552.50	577.00	0.1912
	Overall	386	98	158	191	213	1,094.00	1,711.00	0.3140	58	108	133	170	1,117.00	1,722.00	0.2159
ER1b	Lang	63	25	34	37	38	5.57	7.09	0.4761	19	31	32	36	7.00	9.27	0.4143
	Math	105	20	41	49	55	77.69	450.75	0.2800	13	31	40	51	85.44	440.75	0.2252
	Time	26	7	11	13	15	126.75	394.25	0.2974	6	10	11	13	118.88	348.50	0.2460
	Closure	131	27	44	57	69	44.34	122.38	0.2844	13	24	34	48	78.00	194.88	0.1713
	Chart	25	11	17	18	20	23.84	76.38	0.4729	7	13	14	20	36.97	113.75	0.3149
	Mockito	36	8	10	14	21	47.38	73.88	0.2908	7	11	12	17	64.13	107.75	0.2456
	Overall	386	98	157	188	218	51.59	269.50	0.3281	65	120	143	185	67.25	277.00	0.2468
ER5a	Lang	63	6	19	23	26	12.51	12.36	0.2411	6	19	23	26	12.51	12.36	0.2411
	Math	105	3	8	10	16	140.62	480.75	0.0776	3	8	10	16	140.62	480.75	0.0776
	Time	26	1	1	1	1	458.50	590.50	0.0237	1	1	1	1	458.50	590.50	0.0237
	Closure	131	0	0	0	0	601.00	368.25	0.0043	0	0	0	0	601.00	368.25	0.0043
	Chart	25	1	2	3	9	57.91	87.38	0.1229	1	2	3	6	77.19	125.50	0.1027
	Mockito	36	0	0	0	1	166.62	205.75	0.0162	0	0	0	1	166.62	205.75	0.0162
	Overall	386	11	30	37	53	294.50	441.75	0.0730	11	30	37	50	295.75	441.75	0.0717
ER5b	Lang	63	6	19	23	26	12.51	12.36	0.2411	6	19	23	26	12.51	12.36	0.2411
	Math	105	3	8	10	16	140.62	480.75	0.0776	3	8	10	16	140.62	480.75	0.0776
	Time	26	1	1	1	1	458.50	590.50	0.0237	1	1	1	1	458.50	590.50	0.0237
	Closure	131	0	0	0	0	601.00	368.25	0.0043	0	0	0	0	601.00	368.25	0.0043
	Chart	25	1	2	3	9	57.91	87.38	0.1229	1	2	3	6	77.19	125.50	0.1027
	Mockito	36	0	0	0	1	166.62	205.75	0.0162	0	0	0	1	166.62	205.75	0.0162
	Overall	386	11	30	37	53	294.50	441.75	0.0730	11	30	37	50	295.75	441.75	0.0717
ER5c	Lang	63	5	17	19	22	732.50	890.00	0.1859	5	17	19	22	732.50	890.00	0.1859
	Math	105	2	7	9	14	2,158.00	2,206.00	0.0614	2	7	9	14	2,158.00	2,206.00	0.0614
	Time	26	0	0	0	0	3,370.00	46.47	0.0004	0	0	0	0	3,370.00	46.47	0.0004
	Closure	131	0	0	0	0	785.50	971.00	0.0037	0	0	0	0	785.50	971.00	0.0037
	Chart	25	1	2	3	9	581.50	1,246.00	0.1036	1	2	3	6	750.00	1,608.00	0.0865
	Mockito	36	0	0	0	0	599.00	535.50	0.0102	0	0	0	0	599.00	535.50	0.0102
	Overall	386	8	26	31	45	1,294.00	1,616.00	0.0560	8	26	31	42	1,305.00	1,632.00	0.0549
gp02	Lang	63	1	1	1	2	284.25	200.25	0.0236	1	1	1	2	284.25	200.25	0.0236
	Math	105	0	0	0	0	717.50	705.00	0.0051	0	0	0	0	717.50	705.00	0.0051
	Time	26	0	0	0	0	878.00	372.25	0.0016	0	0	0	0	878.00	372.25	0.0016
	Closure	131	0	0	0	0	1,700.00	1,014.00	0.0011	0	0	0	0	1,700.00	1,014.00	0.0011
	Chart	25	0	0	0	0	688.00	487.75	0.0056	0	0	0	0	923.00	658.50	0.0043
	Mockito	36	0	0	0	0	269.75	172.38	0.0095	0	0	0	0	269.75	172.38	0.0095
	Overall	386	1	1	1	2	947.50	918.50	0.0069	1	1	1	2	962.50	923.00	0.0069
gp03	Lang	63	2	2	3	3	932.00	688.50	0.0309	2	3	3	3	1,174.00	757.50	0.0336
	Math	105	0	2	2	2	1,663.00	1,381.00	0.0080	0	2	2	2	1,777.00	1,404.00	0.0077
	Time	26	0	2	2	2	1,181.00	925.50	0.0402	1	2	2	2	1,339.00	1,066.00	0.0593
	Closure	131	0	1	2	3	2,128.00	1,614.00	0.0083	0	1	1	1	2,286.00	1,737.00	0.0063
	Chart	25	3	3	3	3	1,504.00	1,284.00	0.0711	1	2	2	3	2,031.00	1,718.00	0.0246
	Mockito	36	2	2	2	4	356.25	293.50	0.0511	1	1	1	2	372.00	299.75	0.0412
	Overall	386	7	12	14	17	1,537.00	1,405.00	0.0221	5	11	11	13	1,707.00	1,508.00	0.0191

5.6 RQ6. Gradual Cross-Project Training

Table 14 and Table 15 present the results of gradual cross-project learning on Time and Closure, respectively. Boxplots in Fig. 7 visualise the overall distribution of *wef* values. We further show the details of how *wef* values are distributed with scatter-plots of *wef* values[5]. In Time, the *self* configuration tends to produce better ranking models. Both *acc@n* and *wef* values improve as more Time faults replace the faults from the other projects in the training data; the boxplots of Time in Fig. 7 show patterns of decreasing *wef* with all learning algorithms.

The results of Closure, however, shows a pattern that is different from what we have observed from Time. While using more *self* faults improves the performance, training with both *others* and *self* faults can also produce the best ranking models (see Table 15 and Fig. 7). We posit that, by including the *others* faults, the diversity in training data increases, which in turn lowers the risk of overfitting and results in better performance against unseen faults.

Answer to **RQ6**: When there are not enough historical faults in the SUT to learn from, developers can start learning from faults of the other projects, preferably more than one project, and gradually augment the training data with the own faults from the SUT. The results also show that a certain level of diversity in training data, owed to faults from other projects, can help avoiding overfitting.

5.7 RQ7. Feature Importance

Table 16 presents the top ten features obtained by RF based analysis. It shows the normalised average reduction in variance due to split by each feature. Table 17 shows the direct impact of these top ten features on the effectiveness of a given ranking model in percentage decrease for Math and Time[6]. The effectiveness of ranking models are evaluated using *acc@1,3,5*. The value x in each cell denotes $x\%$ decrease in *acc@1,3,5* when the input values for the feature corresponding to that row have been randomly permutated among datapoints: negative values means *increases* in performance.

We note that the majority of code and change metrics are among the top ten, whereas only a relatively small number of SBFL scores feature in the top ten. We suspect that some of the SBFL scores exhibit multicollinearity, i.e., they are

5. Full results for all six projects are available from https://coinse.kaist.ac.kr/projects/fluccs/journal_extension.

6. For full results, please refer to our website: https://coinse.kaist.ac.kr/projects/fluccs/journal_extension

TABLE 8: Baseline metric values from SBFL formulæ, with and without Method Level Aggregation

Tech	Project	Total Faults	With Method Level Aggregation							Without Method Level Aggregation						
			acc				*wef*		MAP	*acc*				*wef*		MAP
			@1	@3	@5	@10	mean	std		@1	@3	@5	@10	mean	std	
gp13	Lang	63	25	34	37	38	5.57	7.09	0.4761	19	31	32	36	7.00	9.27	0.4143
	Math	105	20	41	49	55	80.63	480.25	0.2798	13	31	40	51	89.38	480.50	0.2251
	Time	26	7	11	13	15	176.00	635.50	0.2974	6	10	11	13	177.50	635.50	0.2460
	Closure	131	27	44	57	69	44.34	122.38	0.2844	13	24	34	48	78.19	196.38	0.1713
	Chart	25	11	17	18	20	23.84	76.38	0.4727	7	13	14	20	36.97	113.75	0.3145
	Mockito	36	8	10	14	21	75.75	214.88	0.2903	7	11	12	17	84.38	214.75	0.2450
	Overall	386	98	157	188	218	58.34	318.50	0.3281	65	120	143	185	74.19	331.50	0.2467
gp19	Lang	63	25	34	37	38	5.57	7.09	0.4761	19	31	32	36	7.00	9.27	0.4143
	Math	105	20	41	49	55	80.38	480.25	0.2800	13	31	40	51	89.13	480.50	0.2252
	Time	26	7	12	14	18	161.25	638.00	0.3208	6	11	12	16	162.75	638.00	0.2698
	Closure	131	31	48	67	80	40.22	126.31	0.3167	13	27	39	54	74.06	197.62	0.1814
	Chart	25	11	17	18	20	23.84	76.38	0.4727	7	13	14	20	36.97	113.75	0.3145
	Mockito	36	11	13	18	22	60.28	212.75	0.3525	8	12	15	18	68.94	213.12	0.2795
	Overall	386	105	165	203	233	54.44	318.75	0.3464	66	125	152	195	70.31	331.50	0.2551
ochiai	Lang	63	25	39	42	45	4.54	5.98	0.5015	24	34	35	40	6.19	8.73	0.4622
	Math	105	20	41	49	56	75.44	480.25	0.2871	13	30	39	49	86.56	480.25	0.2234
	Time	26	8	11	15	19	144.00	635.50	0.3298	5	10	13	17	147.62	635.00	0.2627
	Closure	131	33	64	81	92	36.13	116.25	0.3540	13	28	43	58	74.13	192.50	0.1881
	Chart	25	12	17	20	22	2.68	4.57	0.5425	7	13	14	20	8.12	15.98	0.3565
	Mockito	36	12	17	20	24	43.84	211.00	0.3584	9	13	15	19	59.78	212.88	0.2710
	Overall	386	110	189	227	258	47.50	316.25	0.3709	71	128	159	203	65.75	328.75	0.2661
jaccard	Lang	63	24	46	50	52	3.46	5.70	0.5288	19	39	41	50	5.19	8.59	0.4519
	Math	105	20	43	51	56	75.25	480.25	0.2896	13	31	40	50	86.25	480.25	0.2231
	Time	26	8	11	14	18	147.12	635.50	0.3281	5	9	12	17	150.75	634.50	0.2573
	Closure	131	32	64	81	92	40.06	125.56	0.3499	11	25	40	58	80.63	199.00	0.1787
	Chart	25	12	17	20	22	2.68	4.57	0.5425	7	13	14	19	8.60	17.20	0.3562
	Mockito	36	12	16	21	24	45.28	211.00	0.3770	9	12	16	18	60.50	211.88	0.2861
	Overall	386	108	197	237	264	48.94	317.50	0.3762	64	129	163	212	68.00	330.25	0.2622

TABLE 9: Metric values from FLUCCS using other learning algorithms - GPM, SVM, and RF.

Tech	Project	Total Faults	With Code and Change metrics							Without Code and Change metrics						
			acc				*wef*		MAP	*acc*				*wef*		MAP
			@1	@3	@5	@10	mean	std		@1	@3	@5	@10	mean	std	
GPM_{min}	Lang	63	41	52	57	59	62.19	315.00	0.7202	40	53	59	59	2.21	6.83	0.7026
	Math	105	37	54	59	65	213.00	962.50	0.3889	25	44	52	57	46.00	169.12	0.3091
	Time	26	9	16	19	20	126.31	584.50	0.4141	6	10	14	18	173.00	652.50	0.3057
	Closure	131	47	73	84	95	155.62	715.00	0.4209	43	72	83	97	65.00	231.62	0.4036
	Chart	25	16	22	23	24	136.62	666.50	0.6314	13	18	21	22	5.24	15.70	0.5845
	Mockito	36	6	19	25	28	36.66	181.12	0.3240	8	16	19	23	28.56	79.69	0.3196
	Overall	386	156	236	267	291	141.62	707.00	0.4651	135	213	248	276	49.59	238.62	0.4241
GPM_{med}	Lang	63	41	53	58	59	125.00	479.25	0.7212	40	52	59	59	9.05	54.88	0.6890
	Math	105	38	52	58	66	299.50	1,105.00	0.3853	25	44	52	57	123.06	670.00	0.3069
	Time	26	10	15	19	20	132.25	525.50	0.4089	5	9	14	18	458.25	1,089.00	0.2988
	Closure	131	48	71	82	96	398.50	1,309.00	0.4163	42	70	83	100	303.25	1,186.00	0.4004
	Chart	25	15	22	23	24	136.38	664.50	0.6148	13	18	20	22	6.64	21.58	0.5742
	Mockito	36	8	22	24	29	40.34	205.25	0.3555	9	16	21	24	79.56	239.12	0.3240
	Overall	386	160	235	264	294	258.50	1,009.50	0.4644	134	209	249	280	176.50	839.00	0.4194
SVM	Lang	63	34	48	52	60	2.38	5.16	0.6494	25	38	50	59	2.95	4.79	0.5269
	Math	105	31	46	49	56	27.41	49.94	0.3386	21	43	49	56	34.78	113.56	0.3003
	Time	26	10	17	19	19	16.27	35.97	0.4302	8	13	16	19	84.38	377.25	0.3464
	Closure	131	27	64	76	94	36.66	123.94	0.3320	38	60	76	91	42.00	154.38	0.3718
	Chart	25	11	21	21	23	2.08	3.53	0.5005	10	17	19	22	3.92	6.91	0.4981
	Mockito	36	8	21	23	27	33.59	124.62	0.3237	11	18	21	25	35.78	134.25	0.3333
	Overall	386	121	217	240	279	24.64	87.25	0.4023	113	189	231	272	33.47	152.50	0.3806
RF_{min}	Lang	63	40	50	53	58	60.75	323.25	0.6685	37	52	54	58	101.56	448.75	0.6509
	Math	105	35	58	65	75	844.00	1,724.00	0.3789	28	50	61	64	1,087.00	1,888.00	0.3311
	Time	26	9	13	16	19	389.50	1,057.00	0.3987	7	11	15	17	772.50	1,399.00	0.3225
	Closure	131	41	64	73	89	1,269.00	2,392.00	0.3601	38	53	64	80	1,453.00	2,510.00	0.3345
	Chart	25	10	17	19	20	139.00	666.00	0.4260	12	17	20	21	289.00	957.50	0.4507
	Mockito	36	7	10	13	21	166.50	386.75	0.2366	11	15	18	24	192.88	427.50	0.2629
	Overall	386	142	212	239	282	721.00	1,766.00	0.4109	133	198	232	264	894.00	1,907.00	0.3853
RF_{med}	Lang	63	35	47	51	56	92.94	407.50	0.6240	36	48	49	54	182.00	556.50	0.6284
	Math	105	33	54	59	63	1,380.00	2,026.00	0.3433	25	46	51	55	1,421.00	2,037.00	0.2783
	Time	26	8	15	19	19	511.50	1,190.00	0.3906	6	10	13	15	1,033.00	1,535.00	0.2800
	Closure	131	40	57	64	74	1,986.00	2,768.00	0.3355	33	48	59	67	2,019.00	2,764.00	0.2971
	Chart	25	9	16	19	21	139.75	666.00	0.4348	13	19	21	22	315.50	900.50	0.4834
	Mockito	36	9	14	15	20	187.12	396.50	0.2446	10	14	15	21	321.75	485.25	0.2492
	Overall	386	134	203	227	253	1,125.00	2,120.00	0.3862	123	185	208	234	1,221.00	2,130.00	0.3525

correlated with each other. Since they are all designed to rank the faulty statement higher, and computed from the same test and coverage results, it is not unnatural that there exists some correlation between them. Once RF chooses one of the SBFL scores for a split, the chance of other SBFL scores contributing significantly to subsequent splits may be small. In contrast, code and change metrics do not resemble each other as much as SBFL scores do, as they are computed from multiple data sources: change metrics from the code development history and complexity metrics from the source code itself.

Complexity metric *b_length* and *loc* appear to be important consistently across all six projects. These metrics are also present in the top ten features of the model trained using *mixed* configuration. Looking at Table 17, *acc@1* decreases by 27% and by 24% when values of *b_length* and *loc* are randomly shuffled among datapoints for the *mixed* configuration. This suggests that *b_length* and *loc* may be particularly useful features for fault localisation.

Although six projects have common important features, the change metrics, particularly *churn* and *min_age*, have more influence on Time than on other projects. In Time,

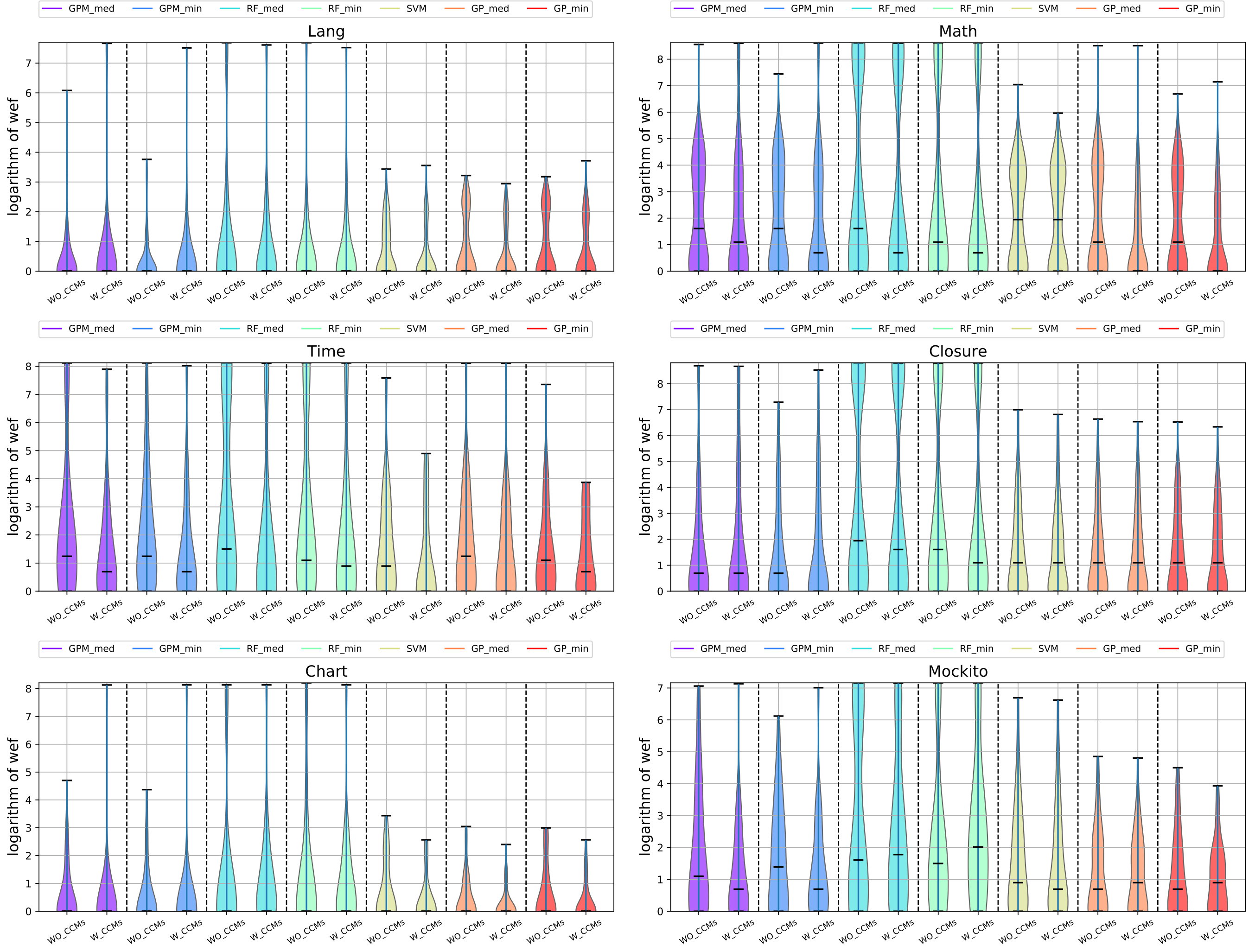

Fig. 5: Violinplots of *wef* values from FLUCCS using GP (GP^A_{min}, GP^S_{min}, GP^A_{med}, GP^S_{med}) and three other learning algorithms: GPM (GPM^A_{min}, GPM^S_{min}, GPM^A_{med}, GPM^S_{med}), SVM (SVM^A, SVM^S), RF (RF^A_{min}, RF^S_{min}, RF^A_{med}, RF^S_{med}). Overall, results of GP and GPM tend to be better than those of SVM and RF

TABLE 10: The effect sizes of Vargha-Delaney A_{12} statistic between the pairs of training data configurations. $n_{win}(n_{tied})/n_{total}$ in a cell located at (row i, column j) implies the row i configuration outperforms and ties with the column j configuration n_{win} and n_{tied} times out of n_{total} comparisons, respectively.

	mixed	*self*	*other lang*	*other math*	*other time*	*other closure*	*other chart*	*other mockito*
mixed	–	7(1)/24	16(0)/20	17(0)/20	19(0)/20	16(0)/20	20(0)/20	16(0)/20
self	16(1)/24	–	17(0)/20	18(0)/20	20(0)/20	16(0)/20	18(0)/20	17(0)/20
$other_{lang}$	4(0)/20	3(0)/20	–	9(0)/16	12(0)/16	9(0)/16	7(0)/16	10(0)/16
$other_{math}$	3(0)/20	2(0)/20	7(0)/16	–	13(0)/16	7(0)/16	6(0)/16	7(0)/16
$other_{time}$	1(0)/20	0(0)/20	4(0)/16	3(0)/16	–	2(0)/16	3(0)/16	3(0)/16
$other_{closure}$	4(0)/20	4(0)/20	7(0)/16	9(0)/16	14(0)/16	–	8(0)/16	11(0)/16
$other_{chart}$	0(0)/20	2(0)/20	9(0)/16	10(0)/16	13(0)/16	8(0)/16	–	8(0)/16
$other_{mockito}$	4(0)/20	3(0)/20	6(0)/16	9(0)/16	13(0)/16	5(0)/16	8(0)/16	–

TABLE 11: The p-values of Mann-Whitney $U-test$ between the pairs of training data configurations. n_{reject}/n_{total} in a cell located at (row i, column j) implies the null-hypothesis of row i configuration producing lower *wef* values than column j configuration is rejected n_{reject} times out of n_{total}.

	mixed	*self*	*other lang*	*other math*	*other time*	*other closure*	*other chart*	*other mockito*
mixed	–	0/24	5/20	8/20	14/20	4/20	3/20	5/20
self	1/24	–	11/20	12/20	16/20	8/20	7/20	8/20
$other_{lang}$	0/20	0/20	–	2/16	8/16	1/16	0/16	2/16
$other_{math}$	0/20	0/20	1/16	–	8/16	0/16	1/16	4/16
$other_{time}$	0/20	0/20	1/16	1/16	–	0/16	0/16	0/16
$other_{closure}$	0/20	1/20	2/16	3/16	9/16	–	0/16	1/16
$other_{chart}$	0/20	0/20	3/16	6/16	10/16	0/16	–	4/16
$other_{mockito}$	0/20	0/20	3/16	4/16	9/16	1/16	1/16	–

random permutation of *churn* and *min_age* decreases *acc@1* by 36% and 67% respectively. In *mixed*, however, the permutation of these metrics decreases *acc@1* only by 2% and 3% respectively. To explain this, we have investigated the distribution of change metrics per project, which is shown in Table 18. Time has the largest *churn* and the smallest *min_age*; the methods of Time are the most frequently changed, but relatively young, when compared to other projects. Based on this, we conjecture that, for projects going through intensive period of frequent changes, code and change metrics can become important features for fault localisation. This seems to confirm the existing results in defect prediction, which suggests that higher code churn is a good indicator of defect proneness [23].

Answer to **RQ7**: The results of feature importance analysis suggest that some of the SBFL scores may be redundant features due to multicollinearity, whereas the majority of code and change metrics are important features. Among the code and change metrics, *b_length* and *loc* have significant

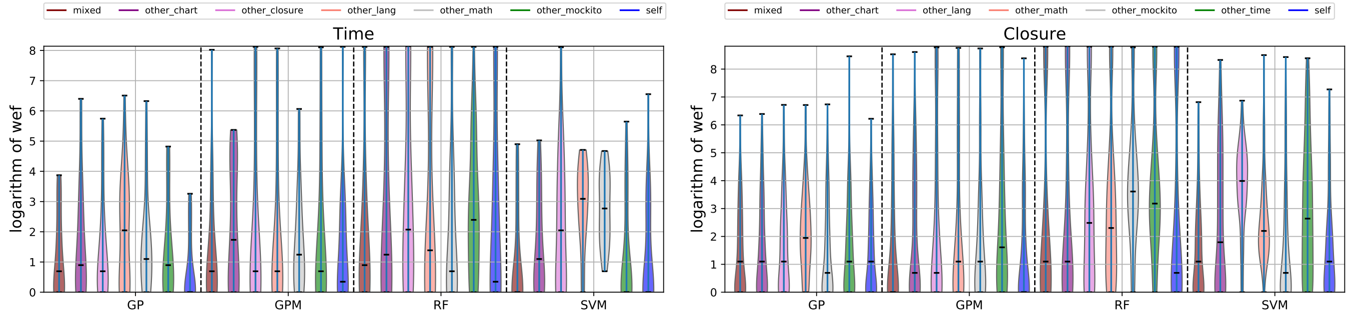

Fig. 6: Violin plots of *wef* metric values from min model of four different learning algorithms: GP, GPM, SVM, and RF. The color of the plots differentiates the configuration that has been used for generating ranking models.

TABLE 12: Effect sizes of Vargha-Delaney A_{12} statistic and p-values of Mann-Whitney U-test on *wef* values for $Time$ dataset. Training configurations on row i and column j are compared.

Tech	Config	Vargha-Delaney A_{12} statistic							Mann-Whitney U-test						
		mixed	*self*	$other_{lang}$	$other_{closure}$	$other_{mockito}$	$other_{chart}$	$other_{math}$	*mixed*	*self*	$other_{lang}$	$other_{closure}$	$other_{mockito}$	$other_{chart}$	$other_{math}$
GP	*mixed*	**0.5**	0.4128	**0.6289**	**0.5420**	**0.5400**	**0.5488**	**0.5923**	**0.0**	0.8711	0.0533	0.2979	0.3066	0.2686	0.1235
	self	**0.5874**	**0.5**	**0.7095**	**0.6318**	**0.6196**	**0.6304**	**0.6821**	0.1290	**0.0**	**0.0040**	**0.0470**	0.0623	**0.0476**	**0.0108**
	$other_{lang}$	0.3713	0.2908	**0.5**	0.4082	0.4082	0.4253	0.4497	0.9468	0.9961	**0.0**	0.8735	0.8740	0.8247	0.7339
	$other_{closure}$	0.4578	0.3684	**0.5918**	**0.5**	**0.5024**	**0.5156**	**0.5708**	0.7021	0.9531	0.1263	**0.0**	0.4890	0.4229	0.1880
	$other_{mockito}$	0.4600	0.3801	**0.5918**	0.4978	**0.5**	**0.5112**	**0.5693**	0.6934	0.9375	0.1261	0.5112	**0.0**	0.4446	0.1927
	$other_{chart}$	0.4512	0.3699	**0.5747**	0.4844	0.4890	**0.5**	**0.5444**	0.7314	0.9521	0.1754	0.5771	0.5557	**0.0**	0.2900
	$other_{math}$	0.4075	0.3181	**0.5503**	0.4290	0.4304	0.4556	**0.5**	0.8765	0.9893	0.2659	0.8120	0.8071	0.7100	**0.0**
GPM	*mixed*	**0.5**	0.4851	**0.5259**	**0.5503**	**0.5254**	**0.6108**	**0.5830**	**0.0**	0.5752	0.3713	0.2627	0.3750	0.0809	0.1487
	self	**0.5146**	**0.5**	**0.5332**	**0.5576**	**0.5347**	**0.6167**	**0.5801**	0.4248	**0.0**	0.3347	0.2310	0.3281	0.0682	0.1553
	$other_{lang}$	0.4741	0.4668	**0.5**	**0.5259**	**0.5015**	**0.5649**	**0.5576**	0.6289	0.6650	**0.0**	0.3718	0.4924	0.2057	0.2338
	$other_{closure}$	0.4497	0.4424	0.4741	**0.5**	0.4771	**0.5400**	**0.5269**	0.7373	0.7690	0.6284	**0.0**	0.6138	0.3079	0.3691
	$other_{mockito}$	0.4749	0.4653	0.4985	**0.5229**	**0.5**	**0.5620**	**0.5488**	0.6250	0.6719	0.5073	0.3862	**0.0**	0.2168	0.2698
	$other_{chart}$	0.3892	0.3831	0.4348	0.4600	0.4380	**0.5**	0.4482	0.9189	0.9316	0.7944	0.6924	0.7832	**0.0**	0.7417
	$other_{math}$	0.4172	0.4202	0.4424	0.4734	0.4512	**0.5518**	**0.5**	0.8511	0.8447	0.7661	0.6309	0.7305	0.2583	**0.0**
SVM	*mixed*	**0.5**	0.4927	**0.7568**	**0.6206**	0.4705	**0.5659**	**0.8003**	**0.0**	0.5381	**0.0007**	0.0633	0.6504	0.2013	**0.0001**
	self	**0.5073**	**0.5**	**0.7539**	**0.6094**	0.4763	**0.5630**	**0.7773**	0.4619	**0.0**	**0.0007**	0.0815	0.6230	0.2103	**0.0003**
	$other_{lang}$	0.2433	0.2463	**0.5**	0.3921	0.2404	0.2737	0.4614	0.9995	0.9990	**0.0**	0.9102	0.9995	0.9976	0.6831
	$other_{closure}$	0.3794	0.3906	**0.6079**	**0.5**	0.3684	0.4246	**0.6138**	0.9365	0.9185	0.0897	**0.0**	0.9551	0.8291	0.0789
	$other_{mockito}$	**0.5298**	**0.5234**	**0.7598**	**0.6318**	**0.5**	**0.5820**	**0.7944**	0.3499	0.3772	**0.0005**	**0.0451**	**0.0**	0.1437	**0.0001**
	$other_{chart}$	0.4341	0.4370	**0.7266**	**0.5757**	0.4180	**0.5**	**0.7417**	0.7988	0.7896	**0.0024**	0.1709	0.8564	**0.0**	**0.0013**
	$other_{math}$	0.1997	0.2227	**0.5386**	0.3860	0.2056	0.2581	**0.5**	1.0000	0.9995	0.3169	0.9209	1.0000	0.9985	**0.0**
RF	*mixed*	**0.5**	**0.5161**	**0.5518**	**0.6089**	**0.6294**	**0.5723**	**0.5068**	**0.0**	0.4175	0.2556	0.0854	0.0523	0.1804	0.4663
	self	0.4836	**0.5**	**0.5332**	**0.5708**	**0.5601**	**0.5557**	0.4883	0.5825	**0.0**	0.3342	0.1829	0.2245	0.2399	0.5605
	$other_{lang}$	0.4482	0.4668	**0.5**	**0.5430**	**0.5376**	**0.5132**	0.4475	0.7446	0.6660	**0.0**	0.2939	0.3179	0.4331	0.7485
	$other_{closure}$	0.3914	0.4290	0.4570	**0.5**	0.4941	0.4690	0.3972	0.9146	0.8174	0.7061	**0.0**	0.5293	0.6519	0.9033
	$other_{mockito}$	0.3706	0.4402	0.4622	**0.5059**	**0.5**	0.4785	0.3899	0.9478	0.7754	0.6821	0.4705	**0.0**	0.6055	0.9170
	$other_{chart}$	0.4275	0.4446	0.4866	**0.5312**	**0.5215**	**0.5**	0.4275	0.8193	0.7603	0.5669	0.3481	0.3943	**0.0**	0.8203
	$other_{math}$	0.4934	**0.5117**	**0.5527**	**0.6030**	**0.6104**	**0.5723**	**0.5**	0.5337	0.4395	0.2517	0.0969	0.0831	0.1797	**0.0**

impact on model performance compared to other features. The result also seem to confirm the existing findings that suggest frequent changes may lead to defect proneness.

6 Discussion

6.1 Selection of the learning algorithm of FLUCCS

Section 5.4 shows the discrepancies between evaluation metrics observed from a single algorithm: e.g., SVM performs the worst according to *acc@1*, but performs the best according to *wef*. Section 5.4 subsequently attributed these discrepancies to the different modes that learn-to-rank algorithms operate: pointwise and pairwise. This section focuses on the more practical question of which algorithm to use for each of the two common use cases of fault localisation: human developers debugging a fault, and an Automatic Program Repair (APR) technique generating a patch.

The two evaluation metrics we used, *acc@n* and *wef*, directly measure the effort required to localise a given fault: *acc@n* approximates the probability of locating a fault from inspecting the top n elements, and the mean *wef* shows the average number of non-faulty program elements to examine before encountering the first faulty one. The primary difference between these two metrics is whether outliers matter (*wef*) or not (*acc@n*). Outliers, i.e., faults that FLUCCS completely fails to localise, can increase the mean *wef* significantly but will not affect *acc@n* as long as they are placed outside the first n places.

If developers have a limited debugging budget and do not necessarily want to locate all faults, it is better to be guided by *acc@n*, as algorithms with higher *acc@n* are likely to lead the developers to more faults within limited time. In contrast, if developers can afford time to debug all faults, they should focus on *wef*, as *wef* measures the total effort required to localise all faults considered. Based on the results in Table 5 and Table 9, for developers with a limited amount of resource and allowed to miss a fault, GP (*acc@1* = 147) and GPM (*acc@1* = 160) would be the ideal choice. On the contrary, if developers have more spared resource, and want to localise all faults, either SVM (*wef* = 24.64) or GP (*wef* = 36.13) could be the answer.

It has been shown that fault localisation techniques that produce a higher ranking for the faulty problem element are not always the best choice for APR techniques [49]. APR techniques do not necessarily benefit from high rankings, as they tend to use the suspiciousness scores as a weight for program mutation. Consequently, for APR techniques, assigning lower scores to non-faulty program elements is

TABLE 13: Effect sizes of Vargha-Delaney A_{12} statistic and p-values of Mann-Whitney U-test on *wef* values for $Closure$ dataset. Training configurations on row i and column j are compared.

Tech	Config	Vargha-Delaney A_{12} statistic							Mann-Whitney U-test						
		mixed	*self*	$other_{lang}$	$other_{mockito}$	$other_{chart}$	$other_{math}$	$other_{time}$	*mixed*	*self*	$other_{lang}$	$other_{mockito}$	$other_{chart}$	$other_{math}$	$other_{time}$
GP	*mixed*	**0.5**	0.4827	**0.5493**	0.4897	**0.5166**	**0.6440**	**0.5093**	**0.0**	0.6875	0.0831	0.6147	0.3196	**0.0000**	0.3953
	self	**0.5171**	**0.5**	**0.5664**	**0.5068**	**0.5337**	**0.6597**	**0.5264**	0.3127	**0.0**	**0.0304**	0.4250	0.1698	**3.695e-06**	0.2285
	$other_{lang}$	0.4509	0.4336	**0.5**	0.4404	0.4690	**0.6025**	0.4617	0.9170	0.9697	**0.0**	0.9536	0.8091	**0.0020**	0.8594
	$other_{mockito}$	**0.5103**	0.4934	**0.5596**	**0.5**	**0.5259**	**0.6509**	**0.5205**	0.3853	0.5752	**0.0462**	**0.0**	0.2307	**0.0000**	0.2803
	$other_{chart}$	0.4834	0.4663	**0.5312**	0.4739	**0.5**	**0.6240**	0.4941	0.6807	0.8301	0.1909	0.7690	**0.0**	**0.0003**	0.5669
	$other_{math}$	0.3560	0.3403	0.3975	0.3491	0.3762	**0.5**	0.3665	1.0000	1.0000	0.9980	1.0000	0.9995	**0.0**	1.0000
	$other_{time}$	0.4905	0.4736	**0.5381**	0.4795	**0.5059**	**0.6333**	**0.5**	0.6050	0.7715	0.1406	0.7197	0.4331	**0.0001**	**0.0**
GPM	*mixed*	**0.5**	0.4866	**0.5459**	**0.5405**	**0.5396**	**0.5508**	**0.6167**	**0.0**	0.6504	0.0941	0.1249	0.1301	0.0740	**0.0005**
	self	**0.5132**	**0.5**	**0.5591**	**0.5518**	**0.5537**	**0.5654**	**0.6274**	0.3499	**0.0**	**0.0452**	0.0704	0.0625	**0.0306**	**0.0001**
	$other_{lang}$	0.4541	0.4412	**0.5**	0.4915	0.4858	**0.5020**	**0.5630**	0.9058	0.9546	**0.0**	0.5967	0.6562	0.4773	**0.0374**
	$other_{mockito}$	0.4597	0.4485	**0.5088**	**0.5**	**0.5024**	**0.5083**	**0.5825**	0.8750	0.9297	0.4033	**0.0**	0.4719	0.4084	**0.0099**
	$other_{chart}$	0.4607	0.4465	**0.5142**	0.4976	**0.5**	**0.5146**	**0.5762**	0.8696	0.9375	0.3438	0.5278	**0.0**	0.3386	**0.0153**
	$other_{math}$	0.4492	0.4343	0.4980	0.4919	0.4854	**0.5**	**0.5679**	0.9258	0.9692	0.5229	0.5913	0.6616	**0.0**	**0.0278**
	$other_{time}$	0.3833	0.3723	0.4373	0.4175	0.4236	0.4321	**0.5**	0.9995	1.0000	0.9624	0.9902	0.9849	0.9722	**0.0**
SVM	*mixed*	**0.5**	**0.5044**	**0.8218**	0.4656	**0.5747**	**0.7075**	**0.6733**	**0.0**	0.4519	**0.0**	0.8345	**0.0172**	**0.0**	**5.364e-07**
	self	0.4956	**0.5**	**0.8037**	0.4587	**0.5669**	**0.7070**	**0.6650**	0.5479	**0.0**	**0.0**	0.8789	**0.0297**	**0.0**	**1.729e-06**
	$other_{lang}$	0.1781	0.1965	**0.5**	0.1976	0.3342	0.2798	0.3823	1.0000	1.0000	**0.0**	1.0000	1.0000	1.0000	0.9995
	$other_{mockito}$	**0.5342**	**0.5415**	**0.8022**	**0.5**	**0.5879**	**0.7212**	**0.6826**	0.1654	0.1210	**0.0**	**0.0**	**0.0063**	**0.0**	**1.192e-07**
	$other_{chart}$	0.4250	0.4331	**0.6660**	0.4121	**0.5**	**0.5620**	**0.5718**	0.9829	0.9702	**1.669e-06**	0.9937	**0.0**	**0.0405**	**0.0219**
	$other_{math}$	0.2925	0.2932	**0.7202**	0.2791	0.4377	**0.5**	**0.5259**	1.0000	1.0000	**0.0**	1.0000	0.9595	**0.0**	0.2323
	$other_{time}$	0.3264	0.3350	**0.6177**	0.3174	0.4282	0.4739	**0.5**	1.0000	1.0000	**0.0005**	1.0000	0.9780	0.7676	**0.0**
RF	*mixed*	**0.5**	**0.5024**	**0.5742**	**0.6914**	**0.5317**	**0.6064**	**0.6846**	**0.0**	0.4717	**0.0179**	**5.960e-08**	0.1847	**0.0014**	**1.192e-07**
	self	0.4976	**0.5**	**0.5605**	**0.6650**	**0.5273**	**0.5913**	**0.6611**	0.5283	**0.0**	**0.0439**	**1.788e-06**	0.2177	**0.0051**	**3.040e-06**
	$other_{lang}$	0.4258	0.4397	**0.5**	**0.6255**	0.4651	**0.5176**	**0.6040**	0.9819	0.9561	**0.0**	**0.0002**	0.8369	0.3120	**0.0018**
	$other_{mockito}$	0.3083	0.3352	0.3745	**0.5**	0.3491	0.3628	0.4695	1.0000	1.0000	1.0000	**0.0**	1.0000	1.0000	0.8042
	$other_{chart}$	0.4683	0.4724	**0.5347**	**0.6509**	**0.5**	**0.5645**	**0.6470**	0.8154	0.7822	0.1631	**0.0000**	**0.0**	**0.0347**	**0.0000**
	$other_{math}$	0.3936	0.4087	0.4824	**0.6372**	0.4353	**0.5**	**0.6128**	0.9985	0.9946	0.6880	**0.0001**	0.9653	**0.0**	**0.0008**
	$other_{time}$	0.3157	0.3391	0.3960	**0.5308**	0.3530	0.3872	**0.5**	1.0000	1.0000	0.9980	0.1959	1.0000	0.9990	**0.0**

TABLE 14: Time (6): The results of gradual cross-project learning on Time. The performance of the trained model improves as more faults from Time are included in training data. The total number of faults in test data is 6, which is approximately one fifth of the total number of faults in Time (26).

Algorithm	Train data Conf	*acc* @1	@3	@5	≥ 6	*wef* mean	std	MAP
GP	$gradual_{4/4}$	1	4	4	2	2.83	3.18	0.43
	$gradual_{3/4}$	1	3	**6**	**0**	2.33	1.49	0.39
	$gradual_{2/4}$	1	4	**6**	**0**	1.83	1.34	0.41
	$gradual_{1/4}$	**2**	3	5	1	2.33	2.05	0.39
	self	1	**5**	**6**	**0**	1.67	1.25	0.37
GPM	$gradual_{4/4}$	1	2	3	3	1,348.83	1,516.43	0.26
	$gradual_{3/4}$	0	2	4	2	9.33	14.24	0.26
	$gradual_{2/4}$	0	3	5	1	5.33	7.54	0.33
	$gradual_{1/4}$	1	**4**	5	1	2.67	2.56	0.39
	self	**2**	**4**	**6**	**0**	1.50	1.26	0.44
SVM	$gradual_{4/4}$	1	1	1	5	14.00	11.69	0.23
	$gradual_{3/4}$	1	2	**5**	**1**	3.17	2.27	0.36
	$gradual_{2/4}$	1	2	**5**	**1**	3.00	2.24	0.37
	$gradual_{1/4}$	2	4	**5**	**1**	61.50	135.29	0.49
	self	**4**	**5**	**5**	**1**	3.17	6.23	0.63
RF	$gradual_{4/4}$	0	2	2	4	570.33	1,251.64	0.15
	$gradual_{3/4}$	0	1	2	4	1,141.00	1,597.83	0.09
	$gradual_{2/4}$	1	**3**	**4**	**2**	1,125.67	1,589.81	0.29
	$gradual_{1/4}$	0	**3**	3	3	565.83	1,253.62	0.17
	self	**2**	**3**	3	3	1,125.67	1,589.82	0.43

TABLE 15: Closure (27): The results of gradual cross-project learning on Closure. For Closure, learning more from its own faults does not always result in better performance. The total number of faults in test data is 27, which is approximately one fifth of the total number of faults in Closure (131).

Algorithm	Train data Conf	*acc* @1	@3	@5	≥ 6	*wef* mean	std	MAP
GP	$gradual_{4/4}$	**9**	14	17	10	30.74	87.64	0.36
	$gradual_{3/4}$	**9**	14	16	11	28.67	79.47	0.35
	$gradual_{2/4}$	**9**	14	16	11	31.15	95.36	0.35
	$gradual_{1/4}$	**9**	14	**19**	**8**	14.26	26.02	0.36
	self	**9**	**15**	17	10	18.11	34.03	0.36
GPM	$gradual_{4/4}$	7	12	13	14	1,742.81	2,585.02	0.25
	$gradual_{3/4}$	**11**	**16**	**18**	**9**	839.37	1,974.15	0.38
	$gradual_{2/4}$	**11**	**16**	**18**	**9**	740.30	1,699.99	0.40
	$gradual_{1/4}$	**11**	15	**18**	**9**	406.07	1,224.17	0.41
	self	**11**	**16**	**18**	**9**	199.81	899.56	0.41
SVM	$gradual_{4/4}$	5	9	14	13	51.70	115.43	0.24
	$gradual_{3/4}$	5	11	15	12	54.81	166.86	0.26
	$gradual_{2/4}$	6	**12**	15	12	57.41	185.67	0.29
	$gradual_{1/4}$	**9**	**12**	**16**	**11**	34.41	112.47	0.33
	self	7	**12**	15	12	38.70	84.75	0.30
RF	$gradual_{4/4}$	5	10	10	17	1,578.26	2,654.20	0.21
	$gradual_{3/4}$	6	10	12	15	2,164.63	2,831.59	0.24
	$gradual_{2/4}$	**10**	**13**	**13**	**14**	2,010.48	2,838.96	0.31
	$gradual_{1/4}$	8	11	**13**	**14**	2,620.96	2,936.84	0.26
	self	7	10	11	16	2,644.89	2,959.05	0.23

as important as assigning higher scores to faulty program elements. However, this property is captured neither by *acc@n* nor *wef*.

We instead use *Locality Information Loss (LIL)*, an evaluation metric for fault localisation based on information theory [50]. *LIL* regards the distribution of suspiciousness scores as a probability distribution ($\mathcal{P}$), and computes the cross-entropy between the ideal distribution ($\mathcal{L}$) obtained from the ground-truth of fault location, and the distribution computed from given scores, using Kullback-Leibler divergence. Kullback-Leibler divergence of 0 means that two distributions are identical. Consequently, a learning algorithm with a smaller *LIL* is likely to be more suitable for APR techniques than others, as it will guide APR techniques to modify the actually faulty program element more frequently.

Fig. 8 presents the overall distribution of *LIL* scores produced by different learning algorithms for faults in six projects, as well as all faults. It shows that *LIL* scores of GPM and SVM have much smaller variances than those of GP and RF, despite having higher median values. This implies that GPM and SVM will perform more consistently for APR techniques, when compared to the other two algorithms whose performance may vary significantly depending on the fault being localised. For an unseen arbitrary fault, the results suggest that GPM and SVM are better choices of algorithms to perform fault localisation for APR techniques.

6.2 The Influence of Code and Change Metrics

In Section 5.2 and Section 5.4, we concluded that using code and change metrics improve the effectiveness of fault

TABLE 16: RF with top 10 features: Each cell contains a feature importance value for the feature. Among the 40 features used in FLUCCS, the table shows only the feature importance values of the top ten features.

Feature	Lang	Math	Time	Closure	Chart	Mockito	mixed
ochiai	-	-	-	-	-	-	-
jaccard	-	-	-	-	-	-	-
gp13	-	-	-	-	-	-	-
wong1	-	-	-	-	-	-	-
wong2	-	-	-	-	-	-	-
wong3	-	-	-	-	-	-	-
tarantula	0.26	0.03	0.12	-	0.08	0.08	-
ample	-	-	-	-	-	-	-
RussellRao	-	-	-	-	-	-	-
SorensenDice	-	-	-	-	-	-	-
Kulczynski1	-	-	-	-	-	-	-
SimpleMatching	-	-	-	-	-	-	-
M1	-	-	-	-	-	-	-
RogersTanimoto	-	-	-	-	-	-	-
Hamming	-	-	-	-	-	-	-
Ochiai2	-	-	-	-	0.04	0.03	-
Hamann	-	-	-	-	-	-	-
dice	-	-	-	-	-	-	-
Kulczynski2	-	-	-	-	-	-	-
Sokal	-	-	-	-	-	-	-
M2	-	-	0.04	-	-	-	-
Goodman	-	-	-	-	-	-	-
Euclid	-	-	-	-	-	-	-
Anderberg	-	-	-	-	-	-	-
Zoltar	-	-	-	0.05	-	-	-
ER1a	0.03	0.09	-	0.08	-	-	0.09
ER1b	0.03	-	0.06	-	0.05	-	-
ER5a	-	-	-	-	-	-	-
ER5b	-	-	-	-	-	-	-
ER5c	-	-	-	-	-	-	-
gp02	0.05	0.05	0.03	0.06	0.04	-	0.05
gp03	0.05	0.04	-	0.05	0.06	0.03	0.05
gp19	-	-	0.05	-	-	0.04	-
churn	0.05	0.06	0.08	0.06	-	0.10	0.06
max_age	-	0.03	-	0.05	0.03	0.11	0.04
min_age	0.03	0.06	0.10	0.05	0.04	0.14	0.06
num_args	-	-	-	-	-	-	0.03
num_vars	0.05	0.07	0.04	0.08	0.05	0.04	0.07
b_length	0.11	0.11	0.08	0.07	0.12	0.05	0.09
loc	0.05	0.11	0.07	0.05	0.05	0.03	0.05

TABLE 17: Performance differences caused by the random permutation of each feature in the top ten for RF: each value in the cell shows the percentage decrease by the permutation of the feature.

Feature	Math			Time			mixed		
	acc@1	*acc@3*	*acc@5*	*acc@1*	*acc@3*	*acc@5*	*acc@1*	*acc@3*	*acc@5*
tarantula	46.90	44.93	41.56	57.46	23.02	25.64	-	-	-
M2	-	-	-	21.05	16.15	18.23			
ER1a	13.43	8.19	4.82	-	-	-	41.09	22.58	17.91
ER1b	-	-	-	2.19	4.81	6.84	-	-	-
gp02	0.61	6.70	6.23	7.89	3.44	6.27	24.19	13.73	10.13
gp03	4.17	4.35	3.27	-	-	-	1.91	1.37	1.19
gp19	-	-	-	18.86	25.77	27.92	-	-	-
churn	1.53	4.41	3.37	35.96	37.46	38.75	2.45	3.00	2.27
max_age	2.44	1.09	1.01	-	-	-	0.19	0.29	0.09
min_age	3.66	7.78	6.43	66.67	59.11	54.42	2.76	2.89	1.91
num_args	-	-	-	-	-	-	1.74	1.60	0.87
num_vars	23.09	19.63	13.37	−1.75	0.69	0.85	16.18	12.93	11.89
b_length	29.30	20.84	17.64	7.89	4.12	3.42	27.43	21.39	18.21
loc	31.74	29.71	26.48	14.47	10.31	10.54	23.59	22.39	19.58

TABLE 18: Distribution of *churn* and *min_age* metrics: the largest mean and median values of *churn* and the smallest mean and median values of *min_age* are written in bold. Compared to the other projects, change are the most frequent in Time

Project	churn (0.0 – 1.0)					min_age (1 –)				
	min	max	mean	std	median	min	max	mean	std	median
Lang	0.00	1.00	0.02	0.04	0.01	1	1684	96.18	111.26	56.00
Math	0.00	1.00	0.01	0.03	0.01	1	4625	202.60	224.04	131.00
Time	0.00	1.00	**0.03**	0.02	**0.03**	1	227	**57.17**	61.60	**39.00**
Closure	0.00	1.00	0.02	0.03	0.01	1	2396	168.15	221.38	88.00
Chart	0.00	1.00	0.01	0.02	0.01	1	911	187.59	199.45	137.00
Mockito	0.00	1.00	**0.03**	0.04	0.01	1	1476	269.54	282.03	167.00

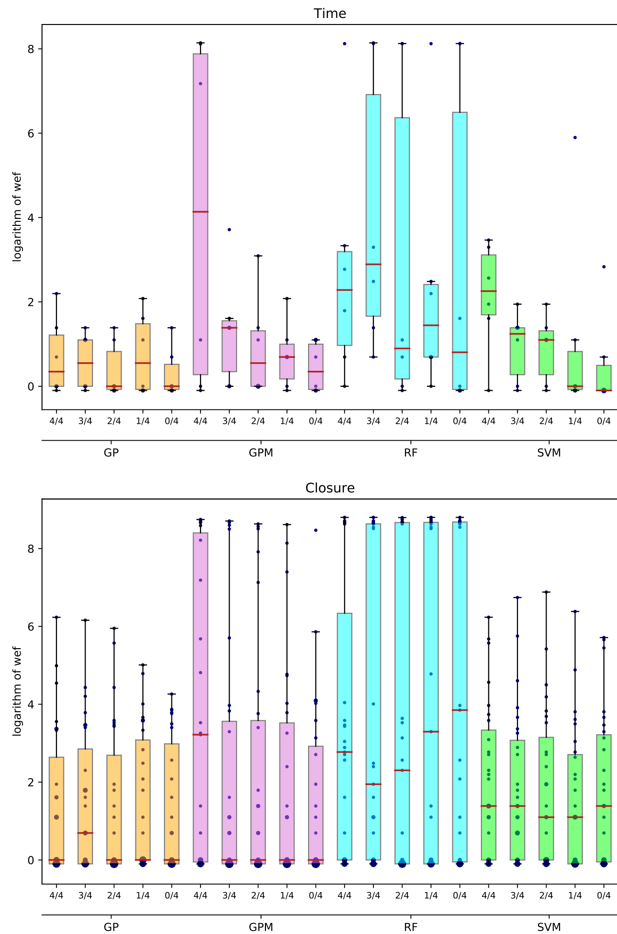

Fig. 7: Boxplots and scatterplots(blue) of *wef* values from gradual cross-project learning with four learning algorithms on Time and Closure. The size of a datapoint in the scatterplots is proportional to the number of data points having the same value. In Time, the performance of a trained model gradually improves as more of its own faults are included in training data, while in Closure, learning from its own faults does not always improve the localisation performance.

localisation. Nevertheless, there are cases for which employing code and change metrics leads to worse performance especially in terms of *acc@1*.

Compared to other projects, the benefit of using code and change metrics is unclear on Mockito. In Mockito, *acc@1* values decrease whenever code and change metrics are used. However, *acc@n* values with $n > 1$ increases, for example, *acc@3* increases from 16 to 22 at GPM_{med} when adding code and change metrics, with all algorithms except RF for which omitting code and change metrics often leads to better performance for Mockito. Overall, the results suggest that using code and change metrics may reduce the number of faults ranked at the top, but can increase the number of faults located near the top.

To explain why code and change metrics are less effective against Mockito, we check whether the feature values of faulty program elements differ from those of non-faulty program elements per project. We first test the normality of the code and change metrics using Shapiro-Wilk test, failing to confirm normal distribution in all seven code and change metrics FLUCCS uses. Consequently, we choose Mann-Whitney U-test to test whether the distributions code and

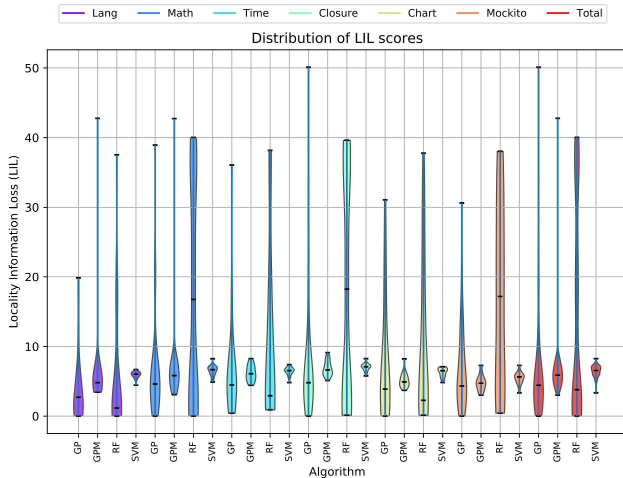

Fig. 8: Overall Distribution of *LIL*: GPM and SVM have smaller variances compared to GP and SVM

change metric feature values are different between faulty and non-faulty program elements. The null hypothesis is that they are from the same distribution.

Table 19 shows the result of the U-test. U-test on a metric *loc* fails to reject the null-hypothesis on Mockito: the p-value for *loc* is 0.55. On the contrary, the null-hypothesis is rejected on the other five projects for *loc*. Since the *loc* has been identified as one of the most important features in Section 5.7, we conjecture that the use of code and change metrics is less effective against faults of Mockito because *loc*, a feature that works well for the other projects, does not contribute to the classification of faulty and non-faulty program elements.

TABLE 19: p-values of Mann-Whitney U-test between code and change metrics of non-faulty and faulty program elements.

Feature	Lang pval	Math pval	Time pval	Closure pval	Chart pval	Mockito pval
churn	0.00	0.00	0.00	0.00	0.00	0.00
max_age	0.00	0.00	0.84	0.12	0.00	0.02
min_age	0.03	0.00	0.00	0.00	0.00	0.00
num_args	0.00	0.08	0.00	0.00	0.00	0.01
num_vars	0.00	0.00	0.01	0.00	0.00	0.00
b_length	0.00	0.00	0.00	0.00	0.00	0.00
loc	0.00	0.00	0.00	0.00	0.00	0.55

7 Threats to Validity

Internal validity concerns the degree to which the results of empirical evaluations warrants the claims; data integrity of training and test data and the correctness of tools are included here. We use `Cobertura`, a widely used coverage tool, to collect the spectrum data; Four learning technique used by FLUCCS are from open source frameworks [39], [44], [46], [47], [51] that withstood public inspection and have been applied in various purpose. We use scipy [51], a python-based open source software, for our statistical analysis. For the implementation of both age and churn metric, we identify methods by their signature. While tracking changes in method signature might be necessary to obtain a complete change history of a method, we limit our implementation of FLUCCS to follow only the changes in the body of a method: we consider any method whose signature has changed as a completely different one, since method signature is often used to describe semantic behaviour of the method [27].

External validity are about factors related to the generality of the conclusions. Despite the use of faults from open source projects, provided by `Defects4J`, our conclusions might be restricted to the programming language (Java) used in these subjects, as well as some unknown factors that we missed to contemplate. In addition, the study is limited by the factors that prevented as stated in Section 4.2, the study is limited by the factors that prevented us from setting accurate ground truth for some faults. We plan to handle these factors in future.

Threats to construct validity includes the correlation between what we actually measured and what they claim to measure. The use of absolute metrics provides more realistic readings of reduction in effort with fault localisation.

8 Related Work

Spectrum Based Fault Localisation has been accepted as the one of the most widely studied technique [7]. Tarantula [52], [53], one of the earliest technique, has been originally developed as a visual aid for debugging, however, it had been quickly applied as a ranking technique that orders program elements based on their likelihood of being faulty. At the early stage of SBFL, most of the formulæ have been developed to reduce the Expense metric, which measures the wasted effort (see Section 4.4) in percentage of the size of SUT. Later, the weakness of Expense metric, becoming unrealistic when the size of SUT is sufficiently large, has been pointed out [16]. Absolute metrics, such as *acc@n* or absolute wasted effort, have been suggested to replace percentage-based metrics and most of recent works have embraced these absolute measures [17], [18]; this paper also follows this trend.

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 [32]. While evolving SBFL formulæ produced previously unknown maximal formulæ [22], there are also theoretically proven restrictions to what a single SBFL formula can achieve [54]. Instead of learning a complicated ranking model from raw spectrum data, FLUCCS takes existing SBFL scores as features, thereby accelerating the pace of learning.

More recently, to overcome the limitation of a single formula, various approaches have been suggested. Xuan and Montperrus combined 25 different SBFL formulæ, by taking a linear weighted sum of formulæ that score above learnt threshold values [17]. Le et al. included changes made to invariants (extracted using Daikon [55]) as an additional feature to the same set of 25 SBFL formulæ [18]; linear rankSVM has been used in learning ranking models. SBFL has also been augmented by Information Retrieval based localisation techniques, using the linear weighted sum approach [56]. Zhang et al. pointed out that both program source code entities and *test* should be differentiated: PRFL, a suggested technique, leverages *pagerank* algorithm to distinguish *test* and recompute the spectrum information based on the contribution of respective test [57].

Some of these approaches combine fault localisation with another domain. Papadakis and Traon approached fault localisation with mutation analysis. The basic intuition behind this approach is that when two elements show same behavior on a same test, they are likely to be related. From this, the technique links mutants killed by failed tests to the real faults and deems places from which these mutants have been originated suspicious [58]. Mutation based fault localisation are further expanded to leverage call graph information to inference sub-graphs related to faults [59]. Maammar et al. tackled a fault localisation problem as a frequent itemset mining problem, finding patterns that satisfying a set of constraints modelling the most suspicious elements; Constraint Programming is used in this approach [60].

While these approaches beat traditional SBFL formulæ with ease and revealed a variety of new aspects of faults to contemplate, most of them are still confined to dynamic data or test execution data, leaving static side of faults barely touched. Although some of them used information extracted from call-graph analysis, concerning static side of the faults in some degree, this information is mainly used to complement dynamic data; they did not approach static data on same importance with dynamic one, only using them to enhance features of dynamic data. FLUCCS is not biased toward either type of data, placing both of them on an equal standing: both static and dynamic data are simply used as features. As a results, FLUCCS could overcome some of inherent limitations of dynamic data, e.g. dependency on quality of test suite, without any fundamental change in the general approach of learn-to-rank in fault localisation. FLUCCS also benefits from the method level aggregation of SBFL scores (described in Section 2.3).

9 Conclusion

We presented FLUCCS, a fault localisation technique that 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 144 out of 386 faults at the top, and 304 out 386 faults within the top ten places. Universal improvement of using code and change metrics are present in the results of all four learning algorithm of FLUCCS, further supporting the generality of our claim on leveraging code and change metrics in fault localisation. From the results of cross-project learning, we find out potential improvement by leveraging project-specific features, but, also the applicability of learnt models to unseen projects without loosing much accuracy.

Future works will regard using FLUCCS on other large-scale projects and going deeper into the details of the correlation between the similarity of projects and the transferability of learnt models.

References

[1] P. McMinn, "Search-based software test data generation: A survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, Jun. 2004.

[2] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Trans. Softw. Eng.*, vol. 39, no. 2, pp. 276–291, Feb. 2013.

[3] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *PLDI*, 2005, pp. 213–223.

[4] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," National Institute of Standards and Technology, Planning Report 02-3.2002, 2002.

[5] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues, "A genetic programming approach to automated software repair," in *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '09. New York, NY, USA: ACM, 2009, pp. 947–954.

[6] C. L. Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "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*, 2012, pp. 3–13.

[7] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, p. 707, August 2016.

[8] X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," *ACM Transactions on Software Engineering Methodology*, vol. 22, no. 4, pp. 31:1–31:40, October 2013.

[9] F. Steimann, M. Frenkel, and R. Abreu, "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*, ser. ISSTA 2013. New York, NY, USA: ACM, 2013, pp. 314–324.

[10] M. Renieres and S. Reiss, "Fault localization with nearest neighbor queries," in *Proceedings of the 18th International Conference on Automated Software Engineering*, October 2003, pp. 30 – 39.

[11] R. Abreu, P. Zoeteweij, and A. van Gemund, "Spectrum-based multiple fault localization," in *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2009, November 2009, pp. 88–99.

[12] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight bug localization with ample," in *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, ser. AADEBUG'05. New York, NY, USA: ACM, 2005, pp. 99–104.

[13] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th International Conference on Automated Software Engineering (ASE2005)*. ACM Press, 2005, pp. 273–282.

[14] H. J. Lee, "Software debugging using program spectra," Ph.D. dissertation, University of Melbourne, 2011.

[15] W. E. Wong, Y. Qi, L. Zhao, and K.-Y. Cai, "Effective fault localization using code coverage," in *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 01*, ser. COMPSAC '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 449–456.

[16] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA 2011. New York, NY, USA: ACM, 2011, pp. 199–209.

[17] J. Xuan and M. Monperrus, "Learning to combine multiple ranking metrics for fault localization," in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, ser. ICSME 2014, Sept 2014, pp. 191–200.

[18] T.-D. B. Le, D. Lo, C. Le Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016, pp. 177–188.

[19] S. Yoo, X. Xie, F.-C. Kuo, T. Y. Chen, and M. Harman, "Human competitiveness of genetic programming in sbfl: Theoretical and empirical analysis," *ACM Transactions on Software Engineering and Methodology*, vol. 26, no. 1, pp. 4:1–4:30, July 2017.

[20] J. Sohn and S. Yoo, "Fluccs: Using code and change metrics to improve fault localisation," in *Proceedings of the International Symposium on Software Testing and Analysis*, ser. ISSTA 2017, 2017, pp. 273–283.

[21] R. Just, D. Jalali, and M. 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*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 437–440.

[22] X. Xie, F.-C. Kuo, T. Y. Chen, S. Yoo, and M. Harman, "Provably optimal and human-competitive results in sbse for spectrum based fault localisation," in *Search Based Software Engineering*, ser. Lecture Notes in Computer Science, G. Ruhe and Y. Zhang, Eds. Springer Berlin Heidelberg, 2013, vol. 8084, pp. 224–238.

[23] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 284–292.

[24] 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*, May 2008, pp. 181–190.

[25] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*, ser. MSR 2010. IEEE, 2010, pp. 31–41.

[26] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, "Defect prediction from static code features: current results, limitations, new approaches," *Automated Software Engineering*, vol. 17, no. 4, pp. 375–407, 2010.

[27] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Automatically detecting and describing high level actions within methods," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 101–110. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985808

[28] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 38–49. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786849

[29] H. Hata, O. Mizuno, and T. Kikuno, "Bug prediction based on fine-grained module histories," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 200–210.

[30] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In, "Micro interaction metrics for defect prediction," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 311–321.

[31] T.-Y. Liu, "Learning to rank for information retrieval," *Foundations and Trends in Information Retrieval*, vol. 3, no. 3, pp. 225–331, 2009.

[32] S. Yoo, "Evolving human competitive spectra-based fault localisation techniques," in *Search Based Software Engineering*, ser. Lecture Notes in Computer Science, G. Fraser and J. Teixeira de Souza, Eds. Springer Berlin Heidelberg, 2012, vol. 7515, pp. 244–258.

[33] C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.

[34] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995. [Online]. Available: http://dx.doi.org/10.1007/BF00994018

[35] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct 2001. [Online]. Available: https://doi.org/10.1023/A:1010933404324

[36] A. Liaw and M. Wiener, "Classification and Regression by randomForest," *R News*, vol. 2, no. 3, pp. 18–22, 2002. [Online]. Available: http://CRAN.R-project.org/doc/Rnews/

[37] jacoco, "JaCoCo. http://www.eclemma.org/jacoco/," 2016. [Online]. Available: http://www.eclemma.org/jacoco/

[38] Cobertura, "Cobertura: A code coverage ut," http://cobertura.github.io/cobertura/, since 2007. [Online]. Available: http://cobertura.github.io/cobertura/

[39] F.-A. Fortin, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné, "DEAP: Evolutionary algorithms made easy," *Journal of Machine Learning Research*, vol. 13, pp. 2171–2175, July 2012.

[40] R. Poli, W. B. Langdon, and N. F. McPhee, *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008, (With contributions by J. R. Koza).

[41] E. Snelson and Z. Ghahramani, "Sparse gaussian processes using pseudo-inputs," in *Advances in Neural Information Processing Systems 18*, Y. Weiss, B. Schölkopf, and J. C. Platt, Eds. MIT Press, 2006, pp. 1257–1264. [Online]. Available: http://papers.nips.cc/paper/2857-sparse-gaussian-processes-using-pseudo-inputs.pdf

[42] J. Q. Shi, B. Wang, R. Murray-Smith, and D. M. Titterington, "Gaussian process functional regression modeling for batch data," *Biometrics*, vol. 63, no. 3, pp. 714–723, 2007. [Online]. Available: http://www.jstor.org/stable/4541403

[43] J. Hensman, N. Fusi, and N. D. Lawrence, "Gaussian processes for big data," *CoRR*, vol. abs/1309.6835, 2013. [Online]. Available: http://arxiv.org/abs/1309.6835

[44] GPy, "GPy: A gaussian process framework in python," http://github.com/SheffieldML/GPy, since 2012.

[45] J. Bergstra, "Hyperopt: Distributed asynchronous hyper-parameter optimization," 2013. [Online]. Available: https://github.com/hyperopt/hyperopt

[46] C.-P. Lee and C.-J. Lin, "Large-scale linear ranksvm," *Neural Comput.*, vol. 26, no. 4, pp. 781–817, Apr. 2014.

[47] C.-C. Chang and C.-J. Lin, "Libsvm: A library for support vector machines," *ACM Trans. Intell. Syst. Technol.*, vol. 2, no. 3, pp. 27:1–27:27, May 2011.

[48] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux, "API design for machine learning software: experiences from the scikit-learn project," in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 2013, pp. 108–122.

[49] Y. Qi, X. Mao, Y. Lei, and C. Wang, "Using automated program repair for evaluating the effectiveness of fault localization techniques," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ser. ISSTA 2013. New York, NY, USA: ACM, 2013, pp. 191–201.

[50] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *Proceedings of the 7th International Conference on Software Testing, Verification and Validation*, ser. ICST 2014, 2014, pp. 153–162.

[51] E. Jones, T. Oliphant, P. Peterson *et al.*, "SciPy: Open source scientific tools for Python," 2001–, [Online; accessed ¡today¿]. [Online]. Available: http://www.scipy.org/

[52] J. A. Jones, M. J. Harrold, and J. T. Stasko, "Visualization for fault localization," in *Proceedings of ICSE Workshop on Software Visualization*, 2001, pp. 71–75.

[53] 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*. New York, NY, USA: ACM, 2002, pp. 467–477.

[54] S. Yoo, X. Xie, F.-C. Kuo, T. Y. Chen, and M. Harman, "No pot of gold at the end of program spectrum rainbow: Greatest risk evaluation formula does not exist," University College London, Tech. Rep. RN/14/14, 2014.

[55] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," in *Proceedings of the 21st International Conference on Software Engineering (ICSE-99)*. NY: ACM Press, May 16–22 1999, pp. 213–225.

[56] T.-D. B. Le, R. J. Oentaryo, and D. Lo, "Information retrieval and spectrum based bug localization: Better together," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 579–590.

[57] M. Zhang, X. Li, L. Zhang, and S. Khurshid, "Boosting spectrum-based fault localization using pagerank," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: ACM, 2017, pp. 261–272. [Online]. Available: http://doi.acm.org/10.1145/3092703.3092731

[58] M. Papadakis and Y. L. Traon, "Metallaxis-fl: mutation-based fault localization," *Softw. Test., Verif. Reliab.*, vol. 25, no. 5-7, pp. 605–628, 2015. [Online]. Available: http://dx.doi.org/10.1002/stvr.1509

[59] T. A. D. Henderson and A. Podgurski, "Behavioral fault localization by sampling suspicious dynamic control flow subgraphs," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, April 2018, pp. 93–104.

[60] M. Maamar, N. Lazaar, S. Loudni, and Y. Lebbah, "Fault localization using itemset mining under constraints," *Automated Software Engg.*, vol. 24, no. 2, pp. 341–368, Jun. 2017. [Online]. Available: https://doi.org/10.1007/s10515-015-0189-z

Jeongju Sohn Jeongju Sohn is a PhD candidate at School of Computing, KAIST, in Republic of Korea. She received BSc in Computer Engineering from Ewha Womans University and MSc in Computer Science from KAIST. Her research interest includes Search Based Software Engineering, fault localisation, and genetic improvements.

Shin Yoo Shin You is an associate professor in School of Computing at KAIST, Republic of Korea. He received a PhD in Computer Science from King's College London, United Kingdom, and previously has been a lecturer at University College London, UK. His research interests include Search Based Software Engineering, i.e. the application of metaheuristic optimisation to software engineering problems, as well as evolutionary computation and genetic programming. He is the program co-chair of IEEE International Conference on Software Testing, Verification & Validation (ICST) 2018, and the Silver Medal recipient of ACM SIGEVO Human Competitiveness Award (HUMIES) 2017.