# Feature Subset Selection and Instance Filtering for Cross-project Defect Prediction - Classification and Ranking

**Faimison Porto and Adenilso Simao**
Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo
PO Box 668, 13560-970, São Carlos, SP, Brazil
{*faimison, adenilso*}*@icmc.usp.br*

**Abstract**

The defect prediction models can be a good tool on organizing the project's test resources. The models can be constructed with two main goals: 1) to classify the software parts - defective or not; or 2) to rank the most defective parts in a decreasing order. However, not all companies maintain an appropriate set of historical defect data. In this case, a company can build an appropriate dataset from known external projects - called Cross-project Defect Prediction (CPDP).

The CPDP models, however, present low prediction performances due to the heterogeneity of data. Recently, Instance Filtering methods were proposed in order to reduce this heterogeneity by selecting the most similar instances from the training dataset. Originally, the similarity is calculated based on all the available dataset features (or independent variables).

We propose that using only the most relevant features on the similarity calculation can result in more accurate filtered datasets and better prediction performances. In this study we extend our previous work. We analyse both prediction goals - Classification and Ranking. We present an empirical evaluation of 41 different methods by associating Instance Filtering methods with Feature Selection methods. We used 36 versions of 11 open source projects on experiments.

The results show similar evidences for both prediction goals. First, the defect prediction performance of CPDP models can be improved by associating Feature Selection and Instance Filtering. Second, no evaluated method presented general better performances. Indeed, the most appropriate method can vary according to the characteristics of the project being predicted.

**Keywords:** Software quality assurance, Cross-project defect prediction, Instance filtering, Feature selection, Classification, Ranking.

## 1 Introduction

Software testing activities are crucial for quality assurance in the software development process. Applying those activities, however, can be expensive and the available resources can be limited. The defect prediction models aim at predicting the likely defective parts of the software. Thus, the cost and efficiency of test can be improved by prioritizing the most critical software parts [1].

In an ideal defect prediction it would be possible to predict the exact number of defects for each software part. However, this goal is hard to achieve or even impossible due to the lack of good quality data in practice [2]. Thus, an alternative is to simplify the prediction goal and make it feasible for practical application. In the literature there are two main types of software defect prediction being studied. In the first, the goal is to classify whether a software part is defective or not [3, 4, 5]. In practice, this kind of prediction allows to distribute the test resources more efficiently, although does not differentiate the level of importance between the defective parts [6]. In the second type, the goal is to predict the instances more likely to contain the larger number of defects and rank them in a decreasing order [7, 6, 2]. In a context where resources are limited, this kind of prediction allows to direct the software quality assurance team to the most defective

parts first [8]. Both prediction goals have their own singularities and practical importance. An analysis involving the two prediction goals provides a comprehensive perspective in the current state-of-the-art of software defect prediction.

In another ideal context of defect prediction it would be possible to predict a software from a base of knowledge constructed from known external projects. Actually, this kind of prediction has already been studied in the literature. This approach is called Cross-project Defect Prediction (CPDP). On the one hand, this approach solves the lack of defect data commonly absent in software companies [9]. On the other, it introduces heterogeneity on data which can compromises the defect prediction performance, as discussed in [10] and [11].

Alternative methods were proposed in order to improve the performance of CPDP models. Among these methods, we can highlight the filtering methods proposed by Turhan et al. [12] and Peters et al. [13]. These methods aim at building an accurate filtered dataset by selecting the most similar instances from the CPDP training dataset. Both methods make use of the Euclidean distance on measuring the similarity between instances. This similarity measure, though, originally considers the entire set of features of the dataset.

We propose that using only the most relevant features on measuring the similarity between instances can improve the performance of the mentioned filtering methods. In order to select the most relevant features of a dataset we refer to the Feature Selection (FS) methods, widely investigated and adopted on the data mining literature [14]. We evaluate four distinct FS methods: Information Gain [15], Relief [16], CFS [14], and Sparcl [17]. We also compare the use of two specific subset of features: code metrics [18] and network metrics [4]. We call the evaluated methods as *Instance Filtering methods based on Feature subset Selection* (IFFS).

In our previous study [19], we present an empirical evaluation of the mentioned IFFS methods in the Classification context. In this paper we extend this experimentation for both prediction contexts - Classification and Ranking. The two experimentation contexts are similar on their structure although differ in relation to the datasets, the prediction model, and the performance measures, as discussed in the text.

We aim to answer the following research questions for both prediction contexts:

- RQ1: Can IFFS methods lead to better performances on CPDP models?

  The results indicate a positive answer in both contexts - Classification and Ranking. For all evaluated projects, at least one IFFS method presented better performance in relation to the absent of filtering methods, with statistical significance. In the Classification context, the IFFS methods improved the percentage of models considered successful for practical use. In the Ranking context, the IFFS methods presented significant better performance in relation to the absent of filtering methods and even better performances if compared with random ordering.

- RQ2: Which IFFS methods present better performances on CPDP?

  The results show similarities in both prediction contexts. The results do not reveal a unique method with general good performance. Instead, the most appropriate method depends on the project characteristics.

This paper is organized as follows. Section 2 provides the necessary background of the related subjects as well as the related works proposed in the literature. Section 3 shows the experimental setup and methodology. Section 4 presents the obtained results and the answers for the research questions. Section 5 presents the threats to validity. Finally, Section 6 concludes the paper and sketches future work.

## 2 Background and Related Works

### 2.1 Software Defect Prediction

Constructing a good defect prediction model in both contexts (Classification and Ranking) encompasses two main issues: 1) building an accurate training dataset and 2) applying an appropriate machine learning algorithm. The training dataset consists of a table of elements (software parts or instances) associated with their respective independent variables (software characteristics or attributes) and the dependent variable (target attribute). In classification problems, the dependent variable corresponds to a binary class: defective or not-defective [4]. In ranking problems, the dependent variable represents the number of defects associated with each instance [7].

In both prediction contexts, several machine learning algorithms have been studied. In [20], the authors compared the prediction performance of 22 classifiers over 10 public domain datasets from the NASA Metrics Data repository. Their results indicate a superior performance of Random Forest in relation to others algorithms. In addition, they found no statistically significant difference among the top 17 classifiers.

Weyuker et al. [6] compared four regression algorithms in the ranking context over three large industrial software systems. Their results also indicate the good performance of Random Forest in relation to others algorithms. Usually, the ranking task is conducted in two steps [21, 7, 6]: first, a regression model is build and applied in order to predict the number of defects for each instance; then, the instances are ordered based on their predicted number of defects. Yang et al. [2] propose a learning-to-rank approach in which the model is constructed aiming to directly optimize the ranking performance, instead of the two step approach. The proposed model leads to good performances in datasets with few attributes. However, in datasets with a large number of attributes, the proposed method performed worse than Random Forest.

### 2.2 Software Metrics

Some studies in the literature were proposed in order to identify the most relevant independent variables, i.e., the software properties most closely associated with the defective instances or the instances with highest defect density. Basili et al. [22] presented seminal studies in this context by using object-oriented metrics. Ostrand et al. [3] proposed prediction models for industrial large-scale systems by using code metrics and historical defect information. Zimmermann and Nagappan [4] proposed the use of social network analysis metrics (also called network metrics) extracted from the software dependency graph. Further, other metrics were investigated such as developer related metrics [23], organizational metrics [24], process metrics [25], change related metrics [5], and antipatterns related metrics [26]. On this study, we use both code metrics and network metrics since they can be automatically extracted just from the source code. Thus, no historical or additional data is required. Furthermore, some works in the literature indicate that better prediction performances can be obtained by combining these two metrics sets [5, 4].

### 2.3 Cross-project Defect Prediction

The dependent variable information can be extracted from historical defect data. When available, those data can be mined and associated with the respective defective software parts [27, 18]. However, in practice, not all software companies maintain clear records about defects or do not have sufficient data from previous projects. In this case, the training dataset can be composed by external projects with known defect information. In the literature, this approach is called Cross-project Defect Prediction (CPDP). On this approach, available projects from different application domains and with different characteristics can be agglomerated in a heterogeneous dataset. However, the heterogeneity of data may compromises the efficiency obtained from the defect prediction models [10].

Some alternative approaches were proposed in the literature in order to improve the performance obtained from CPDP models. Representative approaches on this context are Metric Compensation [28], Nearest Neighbour Filter [12], Transfer Naive Bayes [29], TCA+ [30], and Clustering Filter [13]. Among these approaches we highlight the filtering methods proposed by Turhan et al. [12] and Peters et al. [13] since they consist of simple and effective methods. Both filtering methods aim to select the most similar (and relevant) instances from the CPDP training dataset, as shown below.

### 2.4 Instance Filtering methods

Given a cross-project training dataset and a test dataset to be predicted, the filtering methods aim at selecting only the most relevant training instances. The resulting filtered dataset may improve the efficiency obtained from CPDP models.

Turhan et al. [12] proposed a filtering method based on the nearest neighbor filter. On this method, here called Burak filter, each test instance is compared with the entire training dataset. The comparison is based on the Euclidean distance and considers the entire set of independent variables. Then, for each test instance, the $k$ most similar training instances are selected to compose the filtered dataset. This method is illustrated in Figure 1(b), where for each test instance the two most similar training instances are selected. As illustrated, the same training instance can be similar with more than one test instance. The instance labeled with 'L' is not included in the filtered dataset.

The method proposed by Peters et al. [13] selects the most similar instances from a different perspective. First, the training instances are clustered considering all the test instances as centroids. The clustering process makes use of the K-means algorithm with Euclidean distance [31]. As a result, each training instance is associated with its most similar centroid (test instance). This first step is illustrated in Figure 1(c). Note that a test instance can be not selected (label 'L'). Second, for each test instance is selected the most similar training instance of the respective cluster. This second step is illustrated in Figure 1(d). The instances labeled with '1' and '2' represent the most similar instances and the resulting filtered dataset.

For both filtering methods, the resulting filtered dataset is then used as the training dataset.

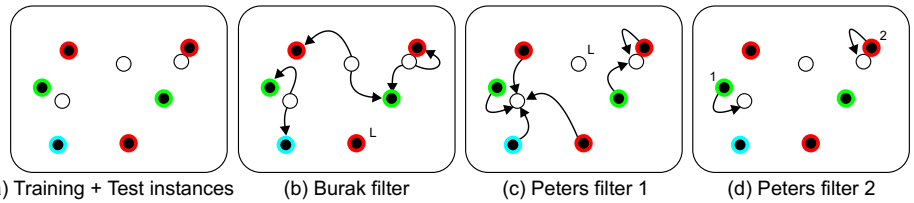


Figure 1: Illustration of the procedure executed for each filtering method evaluated. On Figure (a), the colored instances represent the cross-project training data and the white instances represent the test data. See text for more details (Adapted from Peters et al. [13])

### 2.5 Feature Subset Selection

Both instance filtering methods presented above make use of the Euclidean distance on measuring the similarity between instances. The set of independent variables for an instance $m$ can be represented by a vector $X_m = \{x_1^m, x_2^m, ..., x_n^m\}$, where $|X_m| = n$ is the number of variables (or features). The Euclidean distance between two instances $i$ and $j$ can be defined as

$$d(X_i, X_j) = \sqrt{\sum_{k=1}^{n}(x_k^i - x_k^j)^2}. \quad (1)$$

Let $S$ be a subset of features composed of the most relevant features in $X$. We propose that using only the most relevant features on the Euclidean distance (i.e., $d(S_i, S_j)$ on Equation 1) can lead to a more accurate filtered dataset and then to a more efficient defect model. In order to select the most relevant features $S \subset X$, we evaluate different Feature Selection (FS) methods.

The FS methods have been widely investigated in the data mining literature [14]. Traditionally, the FS methods are applied in high dimensional data in order to reduce the data dimensionality by removing the irrelevant and the redundant features [14]. We focused on a specific category of FS methods called *filter model* since they are not dependent on the data mining algorithm. We evaluate three supervised methods: Information Gain [15], Relief [16], and CFS [14]; and one unsupervised method: Sparcl [17]. The supervised methods select the most relevant features based on the target attribute. The unsupervised methods, however, select the most relevant features based on their characteristics only, independently of the target attribute.

These different methods represent part of the most important FS methods known in the literature. A specialized feature selection text can be found in [14, 17].

## 3 Experimental Setup

We conducted an *in silico* experimentation in order to evaluate the performance of IFFS methods in both contexts: Classification and Ranking. For each experimentation context, we evaluate two filtering methods - Burak Filter [12] and Peters Filter [13] - combined with feature subset selection. We analysed either Feature Selection (FS) methods and specific Software metric Subsets (SS). All experiments and data analysis were made by using the R Project Platform [32].

### 3.1 Data Collection

*Software Projects*

The experiments on this study were conducted based on 36 versions of 11 Java open source projects, available in the PROMISE repository[1]. The collection and preparation of defect data were made and provided by Jureczko and Madeyski [18]. The authors provide a link[2] with detailed information about the software projects and the construction of each dataset. Each instance in a dataset represents a Java object-oriented class (OO class). Originally, the dependent variable corresponds to the number of defects found for each OO class.

For the ranking analysis we considered the datasets in their original forms. Appendix Table A2 presents the distribution and density of defects for each dataset. For the classification analysis we converted the dependent variable to a binary classification problem (1, for number of defects $> 0$; or 0, otherwise). Appendix Table A1 list the number of instances, number of defects, and defect rate for each of the analysed datasets.

[1] `http://openscience.us/repo/`
[2] `http://purl.org/MarianJureczko/MetricsRepo/`

We evaluate two distinct metrics sets as the independent variables: code metrics (CODE) and network metrics (NET). These metrics are numerical and can be automatically extracted directly from the source code files.

*Code Metrics*

The code metrics set is composed of 19 metrics and include complexity metrics, C&K (*Chidamber and Kemerer*) metrics, and structural code metrics. This metric set has been reported as good quality indicators on literature, as discussed in [18]. These metrics can be extracted by using the Ckjm[3] tool. Further details can be found in [18].

*Network Metrics*

The network metrics set was extracted in two steps. First, it was constructed the dependency graph for each analysed software version. Each OO class file is represented by a vertex and any dependency relationship between two vertices is represented by an edge on the graph. This process was made by using the PF-CDA tool[4]. The inner classes vertices of a same parent class was treated as a unique vertex. For example, vertices with names "A", "A$B" and "A$C" were all contracted to the parent vertex "A".

Table 1: The analysed network metrics. The metrics were computed for incoming (In), outgoing (Out), and/or undirected (All) dependencies (Descriptions adapted from Zimmermann and Nagappan [4] and Herzig et al. [5])

| **Ego Network Metrics** | |
|---|---|
| **Metric Name** | **Description** |
| $\text{Size}^{In\ Out\ All}$ | # nodes connected to the ego network |
| $\text{Ties}^{In\ Out\ All}$ | # directed ties corresponds to the number of edges |
| $\text{Pairs}^{In\ Out\ All}$ | # ordered pairs is the maximal number of directed ties |
| $\text{Density}^{In\ Out\ All}$ | % of possible ties that are actually present |
| $\text{WeakComp}^{In\ Out\ All}$ | # weak components in neighborhood |
| $\text{nWeakComp}^{In\ Out\ All}$ | # weak components normalized by size |
| $\text{TwoStepReach}^{In\ Out\ All}$ | % nodes that are two steps away |
| $\text{ReachEfficiency}^{In\ Out\ All}$ | TwoStepReach normalized by size |
| $\text{EgoBetween}^{In\ Out\ All}$ | % shortest paths between neighbors through ego |
| $\text{nEgoBetween}^{In\ Out\ All}$ | EgoBetween normalized by the size |
| $\text{EffSize}^{All}$ | size of the ego network minus the average number of ties between their nodes |

| **Global Network Metrics** | |
|---|---|
| **Metric Name** | **Description** |
| $\text{Degree}^{In\ Out\ All}$ | # dependencies for a node |
| $\text{Closeness}^{In\ Out\ All}$ | Total length of the shortest paths from a node (or to a node) to all other nodes |
| $\text{Reach}^{In\ Out\ All}$ | How far a node can receive or send information in the network |
| $\text{Spread}^{In\ Out\ All}$ | How wide a node can receive or send information in the network |
| $\text{Radiality}^{In\ Out\ All}$ | Radial centrality of a node |
| $\text{Eigenvector}^{All}$ | Measure the influence of an node in a network |
| $\text{Betweenness}^{All}$ | Measure for a node in how many shortest paths between other nodes it occurs |
| $\text{Power}^{All}$ | Bonacich Power centrality measure |
| $\text{Clustering}^{All}$ | % of adjacent vertices to a vertex be connected |
| $\text{Page Rank}^{All}$ | Google's PageRank value. Generalization of Eigenvector metric |
| $\text{Eccentricity}^{All}$ | Distance of the shortest path from a vertex to the farthest vertex on the network |
| $\text{Hub}^{All}$ | Kleinberg's Hub measure. Generalization of Eigenvector metric |
| $\text{Authority}^{All}$ | Kleinberg's Authority measure. Generalization of Eigenvector metric |

Once created the dependency graph, we extracted the network metrics data. Table 1 briefly describes the metrics analysed in this study. The metrics set can be grouped in two categories: the ego metrics and the global metrics. The ego metrics refers to the metrics extracted from the *ego network*. Given a vertex $v$, its ego network contains all vertices directly connected to $v$ and their respective connections. The global metrics refers to the metrics extracted from the original entire graph. In addition, some metrics can be extracted from three different kinds of networks according to the direction of edges: incoming edges (In), outgoing edges (Out), or undirected edges (All). We analysed 24 different network metrics. From these 24 metrics, considering all their variations (In, Out, All), we extracted a total of 54 network metrics. We computed these metrics by using the igraph package [33]. The analysed metrics were selected according to availability at igraph package or feasibility to implement.

[3] `http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm`
[4] `http://www.dependency-analyzer.org/`

### 3.2 Feature Subset Selection

We analysed the FS methods: CFS (FS_CFS) [14], Information Gain (FS_IG) [15], Relief (FS_RLF) [16], and Sparcl (FS_SPC) [17]. The first three are supervised methods. Thus, the feature selection is performed from the training data, in which the target attribute is known. The Sparcl method is unsupervised. Thus, can be performed from both training (FS_SPC_Tr) and test data (FS_SPC_Te) since no target attribute information is required. The methods FS_IG, FS_RLF, and FS_SPC provides a rank of weights that measures the relevance of each feature, according to some criterion. Deciding the best $k$ features to compose the feature subset is not a trivial task [14]. Thus, for each method we analysed four different values of $k = \{5, 10, 15, 20\}$ in order to approximate the best $k$ configuration. Considering all methods and their variations of $k$, we analysed 17 FS methods for each filtering method. We used the R packages 'FSelector' [34] and 'sparcl' [17] with the implemented FS methods. All the analysed FS methods and their respective implementations permit the application of both types of datasets with continuous (for ranking) and discrete (for classification) values in the dependent variable.

In addition, we investigated two distinct metric subsets: code metrics (SS_CODE) and network metrics (SS_NET). For each metric subset, we applied a Pearson correlation test in order to disregard the metrics considered redundant, with correlation greater than 0.90 [35]. We also analysed the performance with the original set of features (Orig), as a comparative reference.

### 3.3 Experiment Design

The experiment design is presented in Figure 2. The structure is quite similar for both experimentation contexts - Classification and Ranking. The differences are concentrated in three points: the dataset (Section 3.1); the prediction model (Section 3.4); and the performance measures (Section 3.5), as discussed individually in the text.

First, we joined all datasets in one unique Cross Project Dataset (CPD). Then, we conducted a cross project analysis. Consider a project $P$ with $V(P)$ versions and a specific version $v_i$ of $V(P)$. For each $v_i \in V(P)$, we used $v_i$ as test and $CPD - V(P)$ as training. In this way, we can analyse the prediction performance disregarding any bias from the different versions of a same project. The training data was filtered by applying the Burak and Peters filters combined with the studied feature subsets presented above. The filtered training data is then used to construct the defect prediction models.

```
projects = [ant, forrest, ivy, log4j, lucene, pbeans, poi, synapse, velocity, xalan, xerces]
FOR EACH project IN projects
    versions[project] = getAllVersions(project)
END FOR
CPD = joinAllProjects(versions) //all cross-project data

//evaluated FS methods and metric subsets
IFFS = [FS_CFS, FS_IG, FS_RLF, FS_SPC_Tr, FS_SPC_Te, SS_CODE, SS_NET, ORIG]
filters = [Burak, Peters] //instance filtering methods
learner = [RF] //Random Forest defect predictor

FOR EACH project IN projects
    CPTrain = CPD - versions[project] //all datasets except project versions
    FOR EACH version IN versions[project]
        FOR EACH filter IN filters
            FOR EACH feature_subset IN IFFS
                filtered_train = filter(CPTrain, feature_subset)
                IFFS_predictor = learner(RF, filtered_train)
                performance_measures = IFFS_predictor(version) //evaluate predictor on test
            END FOR
            //prediction with no filter as a comparative reference
            noFilter_predictor = learner(RF, CPTrain)
            performance_measures = noFilter _predictor(version)
        END FOR
    END FOR
END FOR
```

Figure 2: Pseudocode with the experiment design for both Classification and Ranking

### 3.4 Prediction Models

The models were constructed by using the Random Forest Algorithm [36]. This algorithm has shown good performances compared with others algorithms in both contexts - Classification [20, 37] and Ranking [6, 2]. Lessmann et al. [20] argue that Random Forest is the current state-of-the-art defect predictor. Random Forest is an ensemble of models based on decision trees. At the training time it is constructed a multitude of decision trees from which is calculated and outputted the mode of the classes (for classification) or the mean

prediction (for regression) of the individual trees [36]. The Random Forest algorithm presents robustness to redundant and irrelevant attributes although it can produce overfitting models.

Differently of the classification models, the ranking models are constructed in two steps: first, it is constructed a regression model and then the instances are ranked accordingly to their predicted number of defects. For the experiments we used the R package 'randomForest' [38]. It is important to note that some internal processes of Random Forest, such as sampling and bagging, includes randomness which can lead to different models in each execution. Thus, we constructed 30 models for each evaluated method. The results are presented considering the performance mean.

In [4] and [5] the authors argue that network metrics can lead to better prediction performance when combined with code metrics. Thus, we considered both metric sets CODE and NET as predictors on experiments. Again, the features with high Pearson correlation ($> 0.90$) were disregarded. The remaining set with non redundant features is composed of 20 code metrics and 35 network metrics.

### 3.5 Performance Measures

*Classification*

We analysed three performance measures for classification: recall, probability of false alarm (pf), and g-measure [39]. Both measures (recall and pf) have the focus on the defective class and are efficient for evaluating datasets with a small number of defective examples [40]. These metrics are also used to evaluate the performance of the two analysed filters in their original works [12, 13]. Table 2 presents the confusion matrix and the analysed measures.

The *recall* measures how much of the actual defects were found. The *pf* can be described as the probability of an instance predicted as defective to not be a real defective instance. The optimal pf is 0% and the worst result is 100%. The *g-measure* is the harmonic mean between recall and 1-pf (the specificity).

Table 2: Performance measures for Classification

(a) Confusion matrix

| | | **Actual** | |
|---|---|---|---|
| | | yes | no |
| **Predicted** | yes | TP | FP |
| | no | FN | TN |

(b) Analysed measures

| | |
|---|---|
| recall (pd) | $\frac{TP}{FN+TP}$ |
| pf | $\frac{FP}{TN+FP}$ |
| g-measure | $\frac{2*pd*(1-pf)}{pd+(1-pf)}$ |

*Ranking*

For the ranking task we analysed two performance measures: the percentage of defects ($\alpha$) in the former ($\beta$) instances of the ranking [8, 6]; and the fault-percentile-average (FPA) [6, 2].

The first measure is widely used in the literature since it is practical and simple. Weyuker et al. [6] studied datasets in which 20% ($\beta = 20\%$) of the most defective instances contains 80% ($\alpha = 80\%$) or more of the reported defects. This information can be used to analyse how many defects are present in the first 20% instances of the predicted ranking, in relation to the 80% already known. However, this cutoff $\beta = 20\%$ can varies for each project. In our study, for each project, we fixed $\alpha \approx 80\%$ with $\beta$ ranging between 5% and 74%, as we can see in the Appendix Table A2.

Although the first measure can be efficient for some purposes, it disregards the ordering of the remaining instances in the ranking. Furthermore, it is sensitive to the arbitrary cutoff. Weyuker et al. [6] proposed a general measure, called FPA, that takes into account the whole ranking performance. Consider $K$ instances $i_1, i_2, ..., i_K$, listed in a increasing order of predicted number of defects, being $i_K$ the most defective instance. Let $n_k$ be the actual number of defects for the instance $k$, and $N = n_1 + ... + n_K$ be the total number of defects in the entire list of instances. For any $m$ in $1, ..., K$, the proportion of actual defects in the top $m$ predicted files is

$$P_m = \frac{1}{N} \sum_{k=K-m+1}^{K} n_k. \tag{2}$$

The FPA measure [6] is then the average of the $P_m$. It is defined as:

$$\frac{1}{K} \sum_{m=1}^{K} \frac{1}{N} \sum_{k=K-m+1}^{K} n_k. \tag{3}$$

In other words, FPA is the average of the percentage of actual defects contained in the top $m$ instances ($m = 1, 2, ..., K$). It means that, if the prediction is good, the most defective instances will occur at or near to the top and then will be counted for most of the terms that contribute with the average.

## 4 Results

In this section we discuss the results for the respective research questions in both contexts - Classification and Ranking. We compare the performance of the evaluated IFFS methods with the performance of *noFilter*. In order to verify the statistical difference of performance, we use the non-parametric statistical test Wilcoxon's Signed-Rank ($p$-value $< 0.05$). The symbols $\ominus$ and $\oplus$ represent, respectively, lower and higher performance accordingly to the mean value, with statistical significance. The absence of these symbols means no statistical difference between the compared performances.

***RQ1: Can IFFS methods lead to better performances on CPDP models?***

*Classification*

Table 3 shows the prediction performances obtained for each project. The performance on the *noFilter* column are reported by the g-measure ($g$). All other columns are reported by the relative value $\Delta g = g - g_{noFilter}$, i.e., negative values show lower performance and positive values show higher performance in relation to *noFilter*. To better visualization we multiply the values by 100. The bold values indicates the higher $\Delta g$ obtained for the respective project row. The projects are presented in decreasing order considering the higher $\Delta g$ value. The last row presents the mean value for each column. The columns *Best FS* present the best performance among all the evaluated FS methods. The columns *NET* and *CODE* present the performance obtained by considering the feature subsets SS_NET and SS_CODE.

Table 3: Performance obtained for the analysed IFFS methods for both Burak and Peters Filters for Classification. All the performances are presented in relation to *noFilter*. Positive values means better performance and negative values means lower performance. The bold values indicate the highest performance for a project row. The symbols $^{\oplus}$ and $^{\ominus}$ represent the columns with significant difference in relation to *noFilter*

| | | Burak Filter ($\Delta g$) | | | | Peters Filter ($\Delta g$) | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Project** | **noFilter (g)** | **Orig** | **NET** | **CODE** | **Best FS** | **Orig** | **NET** | **CODE** | **Best FS** |
| forrest-0.6 | 0 | 0 | 0 | 0 | **94.29** | 0 | 8.33 | 58.07 | 58.52 |
| velocity-1.4 | 7.84 | 6.49 | 2.1 | -7.58 | **49.78** | -1.74 | -0.73 | 11.75 | 45.5 |
| xerces-1.4 | 33.27 | -2.6 | -5.61 | 3.88 | **45.15** | 20.97 | 25.28 | 15.31 | 38.34 |
| velocity-1.5 | 9.21 | 7.15 | 6.61 | 0.43 | 27.13 | -2.22 | 5.85 | 10.49 | **41.34** |
| log4j-1.1 | 45.49 | 4.37 | 31.98 | 25.79 | **39.86** | 11.81 | 16.91 | 22.89 | 30.8 |
| pbeans-1.0 | 33.96 | -0.42 | 12.96 | -10.41 | 20.38 | 0 | 0 | 15.85 | **33.68** |
| lucene-2.4 | 15.55 | -4.57 | -4.09 | 10.85 | 10.56 | 3.9 | -0.53 | 24.79 | **31.49** |
| log4j-1.0 | 43.61 | 21.68 | 20.39 | 8.39 | **30.64** | 24.14 | 18.31 | 18.91 | 27.84 |
| velocity-1.6 | 18.37 | -6.3 | -2.43 | -8.12 | 17.26 | -0.92 | 0.18 | 13.64 | **28.99** |
| log4j-1.2 | 26.12 | -4.36 | -4.25 | 3.45 | 2.56 | -18.23 | -17.01 | 5.56 | **28.4** |
| poi-2.5 | 15.26 | 7.58 | 0.44 | 19.99 | 16.93 | 17.95 | -0.05 | 17.93 | **27.58** |
| poi-3.0 | 26.12 | 3.42 | 5.42 | 4.77 | 12.45 | 12.03 | -9.85 | 16.29 | **22.55** |
| lucene-2.2 | 31.38 | -18.4 | -3.31 | -0.75 | 5.16 | -4.41 | **20.09** | 13.47 | 19.74 |
| xerces-1.3 | 46.53 | -3.15 | -5.26 | 6.48 | 19.21 | 4.21 | 15.9 | 14.23 | **19.97** |
| synapse-1.1 | 35.81 | -0.41 | -20.41 | -13.93 | 11.51 | -11.9 | 2.78 | 6.25 | **17.57** |
| synapse-1.2 | 45.3 | 2.79 | 3.07 | -3 | **13.61** | -5.54 | -32.53 | 6.89 | 11.04 |
| xerces-1.2 | 36.68 | -3.07 | -3.76 | 3.65 | **13.5** | 1.39 | 8.3 | 6.78 | 11.82 |
| synapse-1.0 | 56 | -3.27 | -1.49 | -4.51 | **13.23** | -8.98 | 6.78 | -0.46 | 11.83 |
| lucene-2.0 | 54.66 | 2.07 | 4.89 | 10.28 | **12.69** | -7.06 | 3.3 | -0.46 | 5.65 |
| forrest-0.8 | 58.6 | -45.08 | -58.6 | -0.69 | **12.55** | -53.25 | -36.66 | 11.68 | -3.7 |
| poi-2.0 | 59.14 | 5.87 | -5.9 | 7.93 | **11.7** | 5.91 | -5.89 | 7.39 | 7.03 |
| ant-1.3 | 64.94 | -0.73 | -12.64 | 4.9 | **11.21** | -6.74 | -27.9 | -5.32 | 3.31 |
| ivy-1.1 | 63.18 | -0.01 | 0.15 | -15.3 | **11.03** | 4.92 | -4.58 | -17.42 | 7.42 |
| ant-1.6 | 64.24 | 3.69 | 3.18 | -12.68 | **10.48** | 1.91 | -5.43 | -0.55 | 9.27 |
| ant-1.5 | 63.68 | 2.14 | 5.67 | -9.85 | **9.19** | 2.2 | 7.97 | -16.96 | 9.04 |
| xalan-2.4 | 65.74 | 2.11 | 4.61 | -1.34 | **8.8** | 1.01 | -3.24 | -0.55 | 6.52 |
| xalan-2.7 | 42.05 | -9.23 | -3.89 | -2.48 | **6.18** | -4.64 | -3.57 | -10.38 | 4.91 |
| ivy-1.4 | 72.38 | -2.6 | 2.42 | -5.85 | **5.71** | 1.59 | -2.52 | 0.6 | 3.37 |
| ant-1.7 | 73.43 | -0.13 | 1.16 | -0.93 | **4.69** | -4.36 | -8.46 | -4.67 | -0.31 |
| ant-1.4 | 58.93 | -5.59 | -7.83 | -2.37 | 1.76 | -5.54 | -16.63 | -9.81 | **4.68** |
| poi-1.5 | 59.72 | -5.19 | -10.94 | -8.48 | **4.54** | -18.99 | -13.8 | -14.99 | -2.58 |
| forrest-0.7 | 56.55 | -23.86 | -23.64 | -38.32 | **4.2** | -32.16 | -39.48 | -19.81 | 3.72 |
| pbeans-2.0 | 60.74 | 0.8 | -0.31 | -1.86 | **3.88** | -21.42 | -42.9 | -13.2 | -3.66 |
| xalan-2.6 | 55.15 | -8.71 | -2.28 | -5.45 | **2.79** | -7.48 | -6.98 | -6.69 | 0.97 |
| xalan-2.5 | 46.76 | -4.64 | -1.22 | -3.23 | **2.71** | -11.75 | -2.23 | -8.63 | 2.27 |
| ivy-2.0 | 76.73 | -1.62 | -5.97 | **0.66** | 0.46 | -5.75 | -17.21 | -2.27 | -1.41 |
| **Mean** | **45.09** | **-2.39** | **-2.25** | **-1.27** | **15.77**$^{\oplus}$ | **-3.4** | **-4.52** | **4.63** | **15.65**$^{\oplus}$ |

We can analyse this table in different perspectives. First, we can notice that the columns *Best FS* presented better performance than *Orig* in all cases for both filtering methods. It means that at least one of the evaluated FS methods performs better than the original method. The FS methods presented the highest performances in most projects, as highlighted by the bold values. These improvements in performance can be crucial to define if a prediction model is appropriate or not for practical use. Zimmermann et al. [10] present a large experiment on cross-prediction feasibility. Their results show that only 3.4% of the studied models presented a prediction performance considered appropriate to practical use. They considered a successful defect model when accuracy, precision, and recall presents performance greater than 75%. Peters et al. [13] considered both performance thresholds of 60% and 75% for the g-measure. In Figure 3 we compare the percentage of projects with performance greater than 60% and 75% for the methods *noFilter*, original Burak, original Peters, Burak FS and Peters FS (with reference to the *Best FS* column). The original methods perform slightly better than *noFilter*. The highest performances are presented by the filtering methods associated with FS. Considering $g \geq 60$, Peters FS and Burak FS presented successful performance for more than 50% of projects. For the threshold $g \geq 75$, the method Burak FS presents practical use for almost 20% of projects.

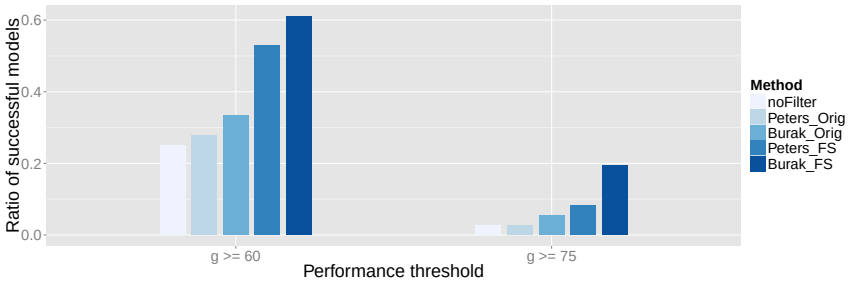


Figure 3: Ratio of successful models for practical use, considering the thresholds $g \geq 60$ and $g \geq 75$

*Ranking*

Table 4 shows the performance obtained for each IFFS method considering the percentage of actual defects present in the top instances of the ranking. The respective cutoffs $\alpha$ and $\beta$ for each project can be viewed in the Appendix Table A2. Table 4 shares the same visual structure as shown in Table 3. The results are relative to *noFilter*, where $\Delta\alpha = \alpha - \alpha_{noFilter}$.

The column *Random* can be used as a comparative reference. In theory, as discussed in [2], supposing a context where the test resources are limited and allow to test only 20% of the software, the testers will spend it in the first 20% instances. In a random resource allocation, it is expected to cover only 20% of the total amount of defects in the first 20% instances. However, by prioritizing the most defective software parts it is possible to increase the percentage of covered defects. By comparing the random allocation shown in Table 4 with the $\beta$ cutoff shown in Appendix Table A2, we can see coincident values (including the mean values of 25%) which corroborates the expected theory. In addition, we can observe a strong learning of the *noFilter* models in relation to the *Random* models in which the performances are higher for all projects and the mean value is twice higher.

The column *Best FS* presented higher performance than *noFilter* with statistical significance - for both filters Burak and Peters. The original methods, however, presented lower performance than *noFilter* also with statistical significance. Peters filter presented a general performance slightly better than Burak filter if we consider the mean value of *Best FS*.

It is important to note that here we ignored all versions of the projects Forrest and Pbeans since their $\beta$ cutoff (see Appendix Table A2) shown to be very small for this kind of analysis. This sensitivity to the cutoff is one of the drawbacks of this performance measure. Furthermore, the ordering beyond the $\beta$ cutoff is ignored by this measure. In order to provide a more accurate analysis we also considered the FPA performance measure.

Table 5 shows the relative performances obtained for each evaluated method. The columns *Max* and *Random* are comparative references. *Max* is the reference for the actual number of defects and perfect ranking with the highest possible FPA. Again, the column *Best FS* presents better performances for both filters in relation to *noFilter*, with statistical significance. The results presented in Table 5 show similar patterns of performances that corroborate for a positive answer for RQ1.

Table 4: Performance comparison of the $\alpha$ percentage of defects in the top $\beta$ instances of Ranking, obtained for each evaluated method. The performances (except Random) are presented in relation to *noFilter*. Positive values means better performance and negative values means lower performance. The bold values indicate the highest performance for a project row. The symbols $^{\oplus}$ and $^{\ominus}$ represent the columns with significant difference in relation to *noFilter*

| | ($\alpha$) | | Burak Filter ($\Delta\alpha$) | | | | Peters Filter ($\Delta\alpha$) | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Project** | **Random** | **noFilter** | **Orig** | **NET** | **CODE** | **Best FS** | **Orig** | **NET** | **CODE** | **Best FS** |
| ivy-1.4 | 5.37 | 32.59 | -0.56 | -2.59 | 6.67 | 12.59 | -4.81 | -2.96 | -1.85 | **13.52** |
| synapse-1.2 | 22.05 | 38.3 | 2.51 | 2.07 | 8.14 | **12.57** | 6.85 | -3.93 | 8.78 | 10.94 |
| velocity-1.6 | 17.82 | 42.98 | 1.88 | 0.82 | -0.21 | 9.33 | -1.84 | 0.21 | 2.68 | **10.57** |
| ant-1.4 | 16.3 | 26.67 | 0.07 | 3.84 | 8.48 | **10.29** | -4.49 | -1.45 | -3.12 | 8.55 |
| synapse-1.1 | 18.42 | 43.1 | 0.88 | 4.11 | 2.49 | 7.74 | 6.57 | 6.06 | -2.15 | **10.27** |
| ant-1.7 | 16.06 | 53.74 | 4.1 | 3.81 | 0.8 | 5.68 | -5.23 | 1.4 | -2.54 | **8.08** |
| lucene-2.2 | 29.41 | 53.05 | -6.37 | -10.28 | -4.33 | 3.71 | -1.81 | -12.8 | 0.87 | **8** |
| poi-2.5 | 39.05 | 55.6 | -4.29 | 3.76 | 0.87 | 4.96 | 0.28 | -3.41 | -7.87 | **6.23** |
| velocity-1.4 | 51.56 | 59.56 | -1.58 | -2.16 | -10.26 | 3.64 | 2.98 | 0.78 | -9.91 | **6.16** |
| ant-1.5 | 10.67 | 36.67 | -6.48 | -0.48 | -0.86 | **5.9** | -7.14 | 1.9 | -13.81 | 2.57 |
| ivy-2.0 | 7.98 | 47.86 | -1.55 | -0.18 | -1.31 | 4.82 | 4.82 | 1.73 | -3.57 | **5.54** |
| lucene-2.4 | 30.53 | 55.15 | -0.4 | 3.05 | 0.5 | **5.18** | 2.33 | 2.46 | 3.38 | 4.65 |
| xerces-1.2 | 14.61 | 22.43 | -0.2 | -2.81 | 3.42 | 4.09 | 2.14 | -0.61 | 3.19 | **5.1** |
| log4j-1.0 | 25.91 | 63.04 | -6.08 | -12.75 | 2.98 | **4.5** | -9.3 | -12.46 | -0.53 | 2.51 |
| velocity-1.5 | 38.33 | 61.57 | -4.88 | -2.05 | -5.17 | **4.16** | -1.08 | -1.25 | -4.74 | 4.1 |
| ant-1.6 | 15.45 | 52.46 | 1.59 | -0.78 | 1.36 | **4.13** | -0.24 | -0.67 | -4.13 | 2.14 |
| poi-2.0 | 10.09 | 30.26 | 2.02 | -2.28 | -2.89 | **3.77** | 1.05 | -1.58 | 2.19 | 2.19 |
| xalan-2.4 | 12.86 | 40.7 | -2.13 | -2.24 | -4.51 | 1.62 | -0.2 | -4.85 | -2.18 | **3.67** |
| xerces-1.4 | 33.63 | 66.04 | -7.56 | -3.06 | -3.96 | 1.77 | -7.52 | -3.11 | -1.15 | **3.66** |
| ant-1.3 | 12.22 | 38.08 | -1.92 | -0.2 | -6.77 | **3.54** | 3.23 | -2.63 | -4.95 | -0.3 |
| xerces-1.3 | 11.97 | 46.55 | 0.1 | 1.14 | 1.62 | 1.62 | 0.78 | -0.48 | -0.31 | **3.25** |
| synapse-1.0 | 8.1 | 40.63 | -15.56 | -11.59 | -13.33 | -0.63 | -14.92 | -7.46 | 3.02 | **3.17** |
| xalan-2.6 | 28 | 51.74 | -2.17 | -0.74 | -3.14 | 2.66 | -0.56 | -1.75 | -1.3 | **3.05** |
| lucene-2.0 | 22.79 | 59.7 | -7.3 | -8.98 | -3.18 | 0.82 | -5.63 | -12.77 | -2.54 | **2.01** |
| xalan-2.5 | 35.78 | 52.2 | -1.1 | -0.65 | -1.72 | **1.85** | -2.58 | -2.46 | 1.04 | 1.04 |
| poi-3.0 | 40.03 | 64.91 | 1.09 | -2.54 | -0.38 | **1.59** | -5.89 | -4.68 | -1.82 | 1.11 |
| ivy-1.1 | 26.09 | 72.09 | -1.97 | -0.77 | 0.37 | **0.99** | -2.76 | -4.96 | -8.5 | -1.46 |
| log4j-1.2 | 65.26 | 74.46 | -1 | -1.68 | 0.03 | **0.81** | -4.38 | -8.06 | -1.74 | -0.89 |
| xalan-2.7 | 74.01 | 79.58 | -0.39 | -0.03 | 0.02 | **0.13** | -0.3 | -1.15 | -0.05 | -0.02 |
| log4j-1.1 | 23.21 | 70.42 | -22.19 | -14.51 | -5.15 | -3.12 | -2.57 | -2.11 | **0.08** | 0.08 |
| poi-1.5 | 29.36 | 52.96 | -0.6 | -1.67 | -1.32 | -0.43 | -8.7 | -6.36 | -1.02 | **-0.4** |
| **Mean** | **25.58** | **51.13** | **-2.65**$^{\ominus}$ | **-2.01** | **-0.99** | **3.88**$^{\oplus}$ | **-1.97**$^{\ominus}$ | **-2.88**$^{\ominus}$ | **-1.76**$^{\ominus}$ | **4.16**$^{\oplus}$ |

***RQ2: Which IFFS methods present better performances on CPDP?***

For both contexts (Classification and Ranking), the best performances obtained for each project are not dominated by one or just a few IFFS methods. Instead, each project has exclusive characteristics that promotes one or other IFFS method. Thus, we analysed the best performances by counting the frequency in which a method is among the top 5 best performances for a project. We consider all the 41 analysed methods, $((17\,FS + 2\,SS + Orig) * 2\,filters) + noFilter$. By this approach we can show the results more clearly. The methods with higher frequency leads to better performances for a larger amount of software projects.

Tables 6 and 7 show the 30 methods with highest frequencies among the best performances for Classification and Ranking, respectively. The Ranking frequencies are based on the FPA measure since it covers the performance of all projects and also has shown to be more consistent in this context.

In the Classification context, the method with the highest frequency (BURAK_FS_IG_10, where the postfix 10 represents the $k$ configuration used on FS_IG) covers almost 30% of all projects, followed by PETERS_SS_CODE with frequency 10. These two methods also appear between the 10 most frequent methods for Ranking. Indeed, we can observe some similarities by comparing the two contexts. For example, 6 of the 10 most frequent methods for Classification are also between the 10 most frequent methods for Ranking. The originals filter methods (with no feature subset selection) presented frequencies lower than *noFilter* in both prediction contexts. Actually, the performance of *noFilter* appears between the 15 and 5 highest frequencies for Classification and Ranking, respectively. This result partially contradicts the results presented by Turhan et al. [12] and Peters et al. [13], in which the instance filtering improves the prediction in relation to *noFilter*. However, some differences can be highlighted between their works and ours. First, we assumed that both original methods consider all features on filtering instances. However, in their works the experiments are conducted considering only the CODE metric set since only this set is available (i.e., the NET metric set is not available for analyses). Here, we can observe that SS_CODE presented good performances since it appears between the 10 most frequent methods for both contexts and filters.

In addition, the experiments of Turhan et al. [12] are on the cross-company context and the analysed

Table 5: Performance comparison based on the ranking measure FPA. The columns Max and Random are comparative references. All other performances are presented in relation to *noFilter*. Positive values means better performance and negative values means lower performance. The symbols $^{\oplus}$ and $^{\ominus}$ represent the columns with significant difference in relation to *noFilter*

| | (FPA) | | | Burak Filter (ΔFPA) | | | | Peters Filter (ΔFPA) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Project** | **Max** | **Random** | **noFilter** | **Orig** | **NET** | **CODE** | **Best FS** | **Orig** | **NET** | **CODE** | **Best FS** |
| forrest-0.6 | 100 | 56.11 | 16.67 | 16.11 | 16.67 | 0 | 62.22 | **83.33** | 10.56 | 70.56 | 83.33 |
| forrest-0.7 | 96.78 | 46.98 | 69.23 | 9.73 | 2.35 | 13.51 | **22.96** | 12.86 | 13.54 | 8.28 | 18.68 |
| forrest-0.8 | 98.96 | 51.01 | 71.42 | -11.63 | -19.79 | 6.25 | 19.58 | 13.37 | 5.87 | **20** | 20 |
| xerces-1.2 | 92.46 | 49.82 | 49 | -1.01 | 0.73 | 0.73 | 6.38 | 5.68 | 3.35 | 2.57 | **10.95** |
| poi-2.0 | 94.29 | 50.43 | 65.77 | 1.65 | -4.02 | 0.72 | 1.47 | -0.97 | -8.91 | -7.82 | **6.55** |
| synapse-1.2 | 88.19 | 50.27 | 62.73 | -0.22 | -1.36 | 2.84 | **6.3** | 1.66 | -4.31 | 4.28 | 4.28 |
| xerces-1.3 | 95.37 | 49.64 | 73.97 | 1.83 | 2.3 | 2.11 | 5.63 | 4.63 | 5.45 | 2.79 | **6.13** |
| velocity-1.4 | 72.37 | 50.38 | 53.93 | -0.62 | -1.72 | -6.68 | 1.78 | 1.84 | 0.56 | -5.64 | **4.42** |
| xerces-1.4 | 84.01 | 50.25 | 70.66 | -1.54 | -1.56 | 0.49 | 1.96 | -4.69 | 0.84 | 1.23 | **3.9** |
| poi-2.5 | 77.21 | 49.82 | 62.37 | -1.63 | 1.95 | -1.11 | 1.96 | 0.34 | -1.2 | -3.15 | **3.42** |
| synapse-1.0 | 96.22 | 50.46 | 76.73 | -4.96 | -2.52 | -1.65 | **3.15** | -9.92 | -2.41 | 1.02 | 3.07 |
| ant-1.4 | 90.76 | 49.9 | 61.34 | -1.45 | 0.93 | 1.92 | **3.08** | -1.87 | -0.54 | -3.07 | 2.73 |
| log4j-1.0 | 88.66 | 50.1 | 75.68 | -4.21 | -10.77 | **2.95** | 2.95 | -0.91 | -7.02 | 2.19 | 2.19 |
| velocity-1.5 | 79.67 | 49.65 | 65.65 | -2.22 | -0.61 | -2.66 | 2.07 | 0.13 | 1.8 | -0.48 | **2.82** |
| lucene-2.4 | 83.11 | 50.64 | 68.13 | -0.32 | 1.03 | 0.33 | 1.54 | 1.08 | -1.05 | **2.65** | 2.65 |
| ant-1.7 | 91.11 | 49.48 | 79.05 | 0.86 | 0.73 | -0.92 | 1.16 | -2.81 | 0.62 | -1.8 | **2.64** |
| ant-1.5 | 95.1 | 48.64 | 77.43 | -1.93 | 0.72 | -2.14 | **2.41** | -4.52 | -0.04 | -3.03 | 1.22 |
| lucene-2.2 | 84.77 | 52.13 | 67.98 | -3.47 | -5.01 | -1.82 | 1.15 | -1.46 | -6.85 | -0.88 | **2.35** |
| poi-1.5 | 84.01 | 51.91 | 64.85 | -1.61 | -0.08 | 0.27 | **2.06** | -4.3 | -3.36 | 0.63 | 0.67 |
| synapse-1.1 | 90.72 | 49.65 | 71.96 | -0.18 | -0.41 | 0.36 | 1.95 | -1.66 | -2.92 | -4.04 | **2.04** |
| pbeans-1.0 | 74.89 | 51.94 | 69.79 | -1.58 | -3.4 | -6.44 | 0.84 | -5.54 | -3 | -6.14 | **2.03** |
| ant-1.6 | 91.35 | 49.68 | 79.32 | 1.3 | 1.11 | -0.03 | **1.87** | -0.27 | 0.02 | -3.13 | 1.19 |
| ivy-1.4 | 97.21 | 49.33 | 86.29 | -0.2 | 0.84 | -0.3 | 1.4 | -6.59 | -1.19 | -3.12 | **1.54** |
| log4j-1.1 | 87.64 | 49.48 | 79.22 | -5.28 | -2.68 | 0.31 | 0.53 | 1.27 | 1.06 | **1.31** | 1.31 |
| xalan-2.6 | 84.8 | 49.59 | 66.73 | -0.71 | -0.11 | -0.2 | **1.28** | -0.04 | -0.12 | -0.35 | 0.74 |
| log4j-1.2 | 64.41 | 50.85 | 56.2 | -0.63 | -1.11 | 0.49 | 1.05 | -0.18 | -5.01 | **1.27** | 1.27 |
| velocity-1.6 | 89.78 | 50.13 | 71.02 | -2.05 | -1.26 | -4.32 | 1.1 | -3.55 | -3.75 | -3.96 | **1.25** |
| ant-1.3 | 94.38 | 49.8 | 77.88 | -8.09 | -2.23 | -2.73 | **1.16** | 1.05 | -5.2 | -2.93 | -1.43 |
| pbeans-2.0 | 94.22 | 49.72 | 78.15 | -0.7 | -2.99 | -6.36 | **0.9** | -7.16 | -4.32 | -0.35 | 0.31 |
| xalan-2.4 | 92.94 | 51.38 | 76.23 | -1.37 | -1.1 | -2.29 | **0.75** | -1.42 | -2.64 | -1.54 | -0.29 |
| ivy-2.0 | 95.67 | 50.79 | 83.88 | -0.67 | -0.75 | -0.1 | **0.63** | -1.04 | -0.61 | -1.93 | -0.4 |
| poi-3.0 | 80.64 | 50.2 | 68.63 | 0.21 | -0.49 | -0.78 | **0.45** | -3.01 | -2.29 | -2.44 | 0.27 |
| xalan-2.5 | 80.94 | 50.15 | 61.65 | -0.24 | -0.36 | -0.35 | **0.42** | -1.93 | -2.37 | -0.63 | 0.41 |
| lucene-2.0 | 88.13 | 50.47 | 74.56 | -1.49 | -1.7 | -0.34 | **0.09** | -1.97 | -5.1 | -0.79 | -0.02 |
| xalan-2.7 | 59.36 | 50.45 | 56.1 | -0.64 | -0.1 | 0 | **0.08** | -0.37 | -0.76 | 0 | 0 |
| ivy-1.1 | 86.69 | 49.37 | 80.17 | -0.73 | -0.25 | -0.8 | **0.05** | -1.03 | -2.16 | -3.34 | -1.08 |
| **Mean** | **87.41** | **50.29** | **68.62** | **-0.82**$^{\ominus}$ | **-1.03** | **-0.24**$^{\ominus}$ | **4.57**$^{\oplus}$ | **1.67** | **-0.93** | **1.62**$^{\oplus}$ | **5.31**$^{\oplus}$ |

Table 6: Frequency in which a IFFS method performs among the top 5 prediction performances for a project (Classification)

| Method | Freq. | Method | Freq. | Method | Freq. |
|---|---|---|---|---|---|
| BURAK_FS_IG_10 | 11 | PETERS_FS_RLF_5 | 6 | BURAK_FS_IG_20 | 4 |
| PETERS_SS_CODE | 10 | PETERS_FS_IG_5 | 6 | BURAK_SS_NET | 4 |
| BURAK_FS_IG_5 | 9 | PETERS_FS_RLF_15 | 6 | PETERS_FS_SPC_Te_5 | 4 |
| PETERS_FS_SPC_Tr_5 | 9 | PETERS_FS_IG_10 | 5 | BURAK_FS_RLF_15 | 4 |
| BURAK_FS_SPC_Te_5 | 8 | NO_FILTER | 5 | PETERS_FS_RLF_10 | 4 |
| BURAK_FS_RLF_10 | 7 | PETERS_FS_IG_15 | 5 | PETERS_FS_IG_20 | 3 |
| BURAK_FS_SPC_Tr_10 | 7 | BURAK_FS_SPC_Te_20 | 5 | BURAK_FS_RLF_5 | 3 |
| BURAK_FS_SPC_Te_10 | 7 | BURAK_FS_SPC_Te_15 | 5 | PETERS_SS_NET | 3 |
| BURAK_SS_CODE | 6 | PETERS_FS_SPC_Te_10 | 5 | BURAK_FS_CFS | 3 |
| BURAK_FS_SPC_Tr_5 | 6 | BURAK_FS_SPC_Tr_15 | 4 | PETERS_FS_SPC_Te_20 | 3 |

Table 7: Frequency in which a IFFS method performs among the top 5 prediction performances for a project (Ranking)

| Method | Freq. | Method | Freq. | Method | Freq. |
|---|---|---|---|---|---|
| BURAK_FS_SPC_Tr_10 | 10 | BURAK_FS_RLF_5 | 5 | BURAK_FS_IG_20 | 4 |
| BURAK_FS_SPC_Tr_5 | 9 | PETERS_FS_IG_10 | 5 | BURAK_ORIG | 4 |
| BURAK_FS_IG_10 | 9 | BURAK_FS_IG_15 | 5 | PETERS_FS_IG_20 | 4 |
| PETERS_SS_CODE | 9 | PETERS_FS_SPC_Te_5 | 5 | BURAK_FS_SPC_Te_15 | 4 |
| NO_FILTER | 8 | PETERS_FS_SPC_Tr_10 | 5 | PETERS_SS_NET | 4 |
| BURAK_SS_CODE | 8 | PETERS_FS_CFS | 5 | PETERS_FS_SPC_Tr_20 | 4 |
| BURAK_FS_SPC_Te_10 | 8 | BURAK_FS_SPC_Te_5 | 4 | PETERS_FS_SPC_Te_20 | 4 |
| PETERS_FS_IG_15 | 6 | BURAK_FS_CFS | 4 | BURAK_FS_SPC_Te_20 | 3 |
| PETERS_FS_SPC_Tr_15 | 6 | PETERS_FS_IG_5 | 4 | PETERS_FS_RLF_10 | 3 |
| PETERS_ORIG | 5 | BURAK_FS_IG_5 | 4 | BURAK_SS_NET | 3 |

projects were developed by different companies. The projects considered on this study were generally developed by the Apache Foundation (except for the Pbeans project). This aspect can influence on the level of heterogeneity of data and then improve the performance of prediction by using *noFilter*. Lastly, the authors of Peters et al. [13] restrict the testing domain to software projects with less than 100 instances, differently of us (see Appendix Tables A1 and A2).

By observing the filters separately, we can note that the Burak filter is present in 5 of the first 7 most frequent methods in both contexts. Those frequencies, however, are not mutually exclusive and may count the same project for several methods. For example, in the classification context it is necessary 10 of the first most frequent methods to cover all the 36 projects.

## 5 Threats to Validity

The results indicate that the performance of CPDP models can be improved by applying the evaluated methods. However, some sources of bias can be highlighted from the conducted experiments. First, the evaluated dataset is composed only by Java open source projects. Also, the majority of the evaluated projects belongs to a unique company. These bias can reduce the heterogeneity of data and influence the experiment results. Other datasets with different sources can be used in order to improve the generality of results.

Another important threat to validity is that the number of defects found for each software part is an approximate estimation and does not represent the real number of existent bugs [18]. However, the information about the exact number of defects in a software is difficult if not impossible to acquire in a real project [2]. In Kitchenham et al. [9], the authors argue that software companies frequently do not keep proper historical information about defect data. When available, those information are commonly private or restricted for internal use only [4]. The datasets used in this work fulfill three important characteristics: 1) the datasets are open for reuse; 2) the defect information were extracted from open source projects, which allow us to extract new features from the source code (e.g. the dependency graph and the network metrics); and 3) the entire set is composed of different projects and versions, which enable us to conduct experiments in the cross-project context. Jureczko and Madeyski [18] followed a systematic process to acquire the defect information based on historical bug reports. This procedure is commonly used in the literature [27, 11]. The set of datasets provided by Jureczko and Madeyski [18] is also used in other works in the literature of defect prediction, such as [41], [42], [43] and [44].

Also, the models in this study were constructed by using only one machine learning algorithm, sustained by the performance results presented in [20], [37], [6], and [2]. However, the data mining is an active research field and other algorithms may lead to different results.

## 6 Conclusion

In this study we investigated the performance of CPDP models by applying filtering methods associated with feature subset selection in both prediction contexts - Classification and Ranking. We evaluated 19 methods derived from different configurations of four distinct Feature Selection methods and two metric subsets. The evidences obtained from the results are similar in both prediction contexts. The results show that for all the analysed projects at least one of the 19 evaluated methods presented better performance in relation to both the original filtering methods and the absent of filtering methods.

In the Classification context, the percentage of models considered successful for practical use was improved. In the Ranking context, besides to present better performances in relation to the comparative reference it was possible to observe the evident learning of defect patterns when compared with random ordering.

We also investigated which of the evaluated methods present better performance. We show a list of the evaluated methods ordered by the frequency in which a method is between the best top five performances for a project. The most frequent methods are present in only 11 (Classification) and 10 (Ranking) of the 36 analysed projects. It indicates that the most appropriate method can vary for each project.

For future works, we are investigating which characteristics better represent a software and how these information can be used in order to predict the most appropriate IFFS method to be applied.

## Acknowledgment

The authors acknowledge the support granted by FAPESP (grant 2013/01084-3).

## References


[1] A. Bener and T. Menzies, "Guest editorial: learning to organize testing," *Automated Software Engineering*, vol. 19, no. 2, pp. 137–140, 2012. [Online]. Available: http://dx.doi.org/10.1007/s10515-011-0095-y

[2] X. Yang, K. Tang, and X. Yao, "A learning-to-rank approach to software defect prediction," *IEEE Trans on Reliability*, vol. 64, pp. 234–246, 2015. [Online]. Available: http://dx.doi.org/10.1109/TR.2014.2370891

[3] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the bugs are," in *Proc Int Symp on Soft Test and Analy (ISSTA2004)*. ACM, 2004, pp. 86–96. [Online]. Available: http://dx.doi.org/10.1145/1007512.1007524

[4] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proc Int Conf on Soft Eng (ICSE2008)*, 2008, pp. 531–540. [Online]. Available: http://dx.doi.org/10.1145/1368088.1368161

[5] K. Herzig, S. Just, A. Rau, and A. Zeller, "Predicting defects using change genealogies," in *IEEE Int Symp on Soft Reliab Eng 2013*. IEEE, 2013. [Online]. Available: http://dx.doi.org/10.1109/ISSRE.2013.6698911

[6] E. J. Weyuker, T. J. Ostrand, and R. M. Bell, "Comparing the effectiveness of several modeling methods for fault prediction," *Emp Softw Eng*, vol. 15, pp. 277–295, 2010. [Online]. Available: http://dx.doi.org/10.1007/s10664-009-9111-2

[7] G. Denaro and M. Pezzè, "An empirical evaluation of fault-proneness models," in *Proc Int Conf on Soft Eng (ICSE2002)*, 2002, pp. 241–251. [Online]. Available: http://dx.doi.org/10.1145/581339.581371

[8] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *IEEE Trans on Soft Eng*, vol. 31, no. 4, pp. 340–355, 2005. [Online]. Available: http://dx.doi.org/10.1109/TSE.2005.49

[9] B. A. Kitchenham, E. Mendes, and G. H. Travassos, "Cross versus within-company cost estimation studies: A systematic review," *IEEE Trans Soft Eng*, vol. 33, no. 5, pp. 316–329, 2007. [Online]. Available: http://dx.doi.org/10.1109/TSE.2007.1001

[10] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: A large scale experiment on data vs. domain vs. process," in *Proc Int Symp on Found of Soft Eng (SIGSOFT/FSE2009)*. ACM, 2009, pp. 91–100. [Online]. Available: http://dx.doi.org/10.1145/1595696.1595713

[11] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou, "Towards building a universal defect prediction model," in *Proc Work Conf on Mining Soft Rep (MSR2014)*. ACM, 2014, pp. 182–191. [Online]. Available: http://dx.doi.org/10.1145/2597073.2597078

[12] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano, "On the relative value of cross-company and within-company data for defect prediction," *Emp Soft Eng*, vol. 14, no. 5, pp. 540–578, 2009. [Online]. Available: http://dx.doi.org/10.1007/s10664-008-9103-7

[13] F. Peters, T. Menzies, and A. Marcus, "Better cross company defect prediction," in *Proc Work Conf on Mining Soft Rep (MSR2013)*. IEEE, 2013, pp. 409–418. [Online]. Available: http://dx.doi.org/10.1109/MSR.2013.6624057

[14] Z. Zhao, F. Morstatter, S. Sharma, S. Alelyani, A. Anand, and H. Liu, "Advancing feature selection research - asu feature selection repository," Arizona State University, Tech. Rep., 2011.

[15] T. M. Cover and J. A. Thomas, *Elements of Information Theory*. New York: Wiley-Interscience, 1991.

[16] I. Kononenko, "Estimating attributes: Analysis and extensions of relief," in *Machine Learning: ECML-94*. Springer Berlin Heidelberg, 1994, pp. 171–182. [Online]. Available: http://dx.doi.org/10.1007/3-540-57868-4_57

[17] D. M. Witten and R. Tibshirani, "A framework for feature selection in clustering," *Jour Amer Statist Assoc*, vol. 105, no. 490, pp. 713–726, 2010. [Online]. Available: http://dx.doi.org/10.1198/jasa.2010.tm09415

[18] M. Jureczko and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction," in *Proc Int Conf on Pred Mod in Soft Eng (PROMISE2010)*. ACM, 2010, pp. 9:1–9:10. [Online]. Available: http://dx.doi.org/10.1145/1868328.1868342

[19] F. Porto and A. Simao, "Feature subset selection for instance filtering methods on cross-project defect prediction," in *Proc Ibero-American Conf on Soft Eng (CIBSE/ESELAW2016)*, Quito, 2016, pp. 171–184.

[20] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Trans Soft Eng*, vol. 34, no. 4, pp. 485–496, 2008. [Online]. Available: http://dx.doi.org/10.1109/TSE.2008.35

[21] K. Gao and T. M. Khoshgoftaar, "A comprehensive empirical study of count models for software fault prediction," *IEEE Trans on Reliability*, vol. 56, no. 2, pp. 223–236, 2007. [Online]. Available: http://dx.doi.org/10.1109/TR.2007.896761

[22] V. Basili, L. Briand, and W. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Trans Softw Eng*, vol. 22, pp. 751–761, 1996. [Online]. Available: http://dx.doi.org/10.1109/32.544352

[23] M. Pinzger, N. Nagappan, and B. Murphy, "Can developer-module networks predict failures?" in *Proc Int Symp on Found of Soft Eng (SIGSOFT/FSE2008)*. ACM, 2008, pp. 2–12. [Online]. Available: http://dx.doi.org/10.1145/1453101.1453105

[24] N. Nagappan, B. Murphy, and V. Basili, "The influence of organizational structure on software quality: An empirical case study," in *Proc Int Conf on Soft Eng (ICSE2008)*, 2008, pp. 521–530. [Online]. Available: http://dx.doi.org/10.1145/1368088.1368160

[25] A. E. Hassan, "Predicting faults using the complexity of code changes," in *IEEE Int Conf on Softw Eng*. IEEE, 2009, pp. 78–88. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070510

[26] S. E. S. Taba, F. Khomh, Y. Zou, A. E. Hassan, and M. Nagappan, "Predicting bugs using antipatterns," in *Proc IEEE Int Conf on Soft Maint (ICSM2013)*, 2013, pp. 270–279. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2013.38

[27] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Proc Int Conf on Pred Mod in Soft Eng (PROMISE2007)*. IEEE, 2007, pp. 9–. [Online]. Available: http://dx.doi.org/10.1109/PROMISE.2007.10

[28] S. Watanabe, H. Kaiya, and K. Kaijiri, "Adapting a fault prediction model to allow inter language reuse," in *Proc Int Conf on Pred Mod in Soft Eng (PROMISE2008)*. ACM, 2008, pp. 19–24. [Online]. Available: http://dx.doi.org/10.1145/1370788.1370794

[29] Y. Ma, G. Luo, X. Zeng, and A. Chen, "Transfer learning for cross-company software defect prediction," *Inf Soft Tech*, vol. 54, no. 3, pp. 248–256, 2012. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2011.09.007

[30] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proc Int Conf on Soft Eng (ICSE2013)*, 2013, pp. 382–391. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2013.6606584

[31] A. K. Jain, "Data clustering: 50 years beyond k-means," *Pattern Recognition Letters*, vol. 31, no. 8, pp. 651 – 666, 2010. [Online]. Available: http://dx.doi.org/10.1016/j.patrec.2009.09.011

[32] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2014.

[33] G. Csardi and T. Nepusz, "The igraph software package for complex network research," *InterJournal*, vol. Complex Systems, p. 1695, 2006.

[34] P. Romanski and L. Kotthoff, *FSelector: Selecting attributes*, 2014, r package version 0.20. [Online]. Available: http://CRAN.R-project.org/package=FSelector

[35] L. Torgo, *Data Mining with R, learning with case studies*. Chapman and Hall/CRC, 2010.

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

[37] Y. Jiang, B. Cukic, and Y. Ma, "Techniques for evaluating fault prediction models," *Emp Softw Eng*, vol. 13, pp. 561–595, 2008. [Online]. Available: http://dx.doi.org/10.1007/s10664-008-9079-3

[38] 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/

[39] N. Japkowicz and M. Shah, *Evaluating Learning Algorithms: A Classification Perspective*. Cambridge University Press, 2011.

[40] T. Menzies, A. Dekhtyar, J. Distefano, and J. Greenwald, "Problems with precision: A response to "comments on 'data mining static code attributes to learn defect predictors'"," *IEEE Trans Soft Eng*, vol. 33, no. 9, pp. 637–640, 2007. [Online]. Available: http://dx.doi.org/10.1109/TSE.2007.70721

[41] M. Li, H. Zhang, R. Wu, and Z.-H. Zhou, "Sample-based software defect prediction with active and semi-supervised learning," *Automated Software Engineering*, vol. 19, no. 2, pp. 201–230, Jun. 2012. [Online]. Available: http://dx.doi.org/10.1007/s10515-011-0092-1

[42] Z. He, F. Peters, T. Menzies, and Y. Yang, "Learning from open-source projects: An empirical study on defect prediction." in *ESEM*. IEEE Computer Society, 2013, pp. 45–54. [Online]. Available: http://dx.doi.org/10.1109/ESEM.2013.20

[43] S. Prateek, A. Pasala, and L. Moreno Aracena, "Evaluating performance of network metrics for bug prediction in software," in *Asia-Pacific Soft Eng Conf (APSEC2013)*, vol. 1, 2013, pp. 124–131. [Online]. Available: http://dx.doi.org/10.1109/APSEC.2013.27

[44] F. Zhang, Q. Zheng, Y. Zou, and A. E. Hassan, "Cross-project defect prediction using a connectivity-based unsupervised classifier," in *Proc Int Conf on Soft Eng (ICSE2016)*. New York, NY, USA: ACM, 2016, pp. 309–320. [Online]. Available: http://dx.doi.org/10.1145/2884781.2884839

# Appendix

Table A1: Summary of the projects characteristics for Classification

| Project | No. of Classes | No. of Defective Classes | % of Defective Classes |
|---|---|---|---|
| ant-1.3 | 125 | 20 | 0.16 |
| ant-1.4 | 177 | 39 | 0.22 |
| ant-1.5 | 292 | 32 | 0.11 |
| ant-1.6 | 349 | 92 | 0.26 |
| ant-1.7 | 493 | 139 | 0.28 |
| forrest-0.6 | 6 | 1 | 0.17 |
| forrest-0.7 | 29 | 5 | 0.17 |
| forrest-0.8 | 32 | 2 | 0.06 |
| ivy-1.1 | 111 | 63 | 0.57 |
| ivy-1.4 | 239 | 16 | 0.07 |
| ivy-2.0 | 351 | 40 | 0.11 |
| log4j-1.0 | 82 | 30 | 0.37 |
| log4j-1.1 | 76 | 31 | 0.41 |
| log4j-1.2 | 123 | 117 | 0.95 |
| lucene-2.0 | 195 | 91 | 0.47 |
| lucene-2.2 | 245 | 142 | 0.58 |
| lucene-2.4 | 337 | 200 | 0.59 |
| pbeans-1.0 | 26 | 20 | 0.77 |
| pbeans-2.0 | 51 | 10 | 0.2 |
| poi-1.5 | 222 | 131 | 0.59 |
| poi-2.0 | 293 | 36 | 0.12 |
| poi-2.5 | 360 | 223 | 0.62 |
| poi-3.0 | 413 | 260 | 0.63 |
| synapse-1.0 | 155 | 16 | 0.1 |
| synapse-1.1 | 220 | 60 | 0.27 |
| synapse-1.2 | 252 | 86 | 0.34 |
| velocity-1.4 | 186 | 138 | 0.74 |
| velocity-1.5 | 204 | 137 | 0.67 |
| velocity-1.6 | 217 | 76 | 0.35 |
| xalan-2.4 | 409 | 79 | 0.19 |
| xalan-2.5 | 717 | 359 | 0.5 |
| xalan-2.6 | 776 | 328 | 0.42 |
| xalan-2.7 | 762 | 754 | 0.99 |
| xerces-1.2 | 385 | 71 | 0.18 |
| xerces-1.3 | 398 | 69 | 0.17 |
| xerces-1.4 | 275 | 168 | 0.61 |
| **Mean** | **266.19** | **113.36** | **0.39** |

Table A2: Summary of the projects characteristics for Ranking. We analysed a ranking performance measure that measures the percentage of defects ($\alpha$) in the former ($\beta$) instances of the ranking. Usually, in the literature, $\beta = 20\%$. In this study we fixed $\alpha \approx 80\%$ and the $\beta$ varies depending on the defect distribution for each project

| Project | No. of Classes | Sum of Defects | % $\beta$ | % $\alpha$ | No. of Classes in $\beta$ | No. of Defects in $\alpha$ |
|---|---|---|---|---|---|---|
| ant-1.3 | 125 | 33 | 0.11 | 0.82 | 14 | 27 |
| ant-1.4 | 177 | 46 | 0.17 | 0.8 | 30 | 37 |
| ant-1.5 | 291 | 35 | 0.09 | 0.8 | 25 | 28 |
| ant-1.6 | 348 | 184 | 0.16 | 0.8 | 56 | 148 |
| ant-1.7 | 493 | 284 | 0.17 | 0.8 | 83 | 228 |
| forrest-0.6 | 6 | 1 | 0.17 | 1 | 1 | 1 |
| forrest-0.7 | 29 | 15 | 0.07 | 0.73 | 2 | 11 |
| forrest-0.8 | 32 | 6 | 0.06 | 1 | 2 | 6 |
| ivy-1.1 | 111 | 233 | 0.27 | 0.81 | 30 | 188 |
| ivy-1.4 | 239 | 18 | 0.05 | 0.83 | 13 | 15 |
| ivy-2.0 | 351 | 56 | 0.08 | 0.8 | 29 | 45 |
| log4j-1.0 | 82 | 57 | 0.23 | 0.81 | 19 | 46 |
| log4j-1.1 | 76 | 79 | 0.24 | 0.82 | 18 | 65 |
| log4j-1.2 | 123 | 355 | 0.66 | 0.8 | 81 | 285 |
| lucene-2.0 | 195 | 268 | 0.23 | 0.8 | 44 | 215 |
| lucene-2.2 | 245 | 412 | 0.29 | 0.8 | 72 | 330 |
| lucene-2.4 | 337 | 629 | 0.31 | 0.8 | 105 | 505 |
| pbeans-1.0 | 26 | 36 | 0.5 | 0.81 | 13 | 29 |
| pbeans-2.0 | 51 | 19 | 0.14 | 0.84 | 7 | 16 |
| poi-1.5 | 222 | 326 | 0.3 | 0.8 | 66 | 261 |
| poi-2.0 | 291 | 38 | 0.1 | 0.82 | 29 | 31 |
| poi-2.5 | 360 | 454 | 0.4 | 0.8 | 144 | 365 |
| poi-3.0 | 410 | 479 | 0.4 | 0.8 | 165 | 384 |
| synapse-1.0 | 155 | 21 | 0.08 | 0.81 | 12 | 17 |
| synapse-1.1 | 220 | 99 | 0.19 | 0.81 | 41 | 80 |
| synapse-1.2 | 251 | 145 | 0.23 | 0.8 | 57 | 116 |
| velocity-1.4 | 182 | 196 | 0.52 | 0.8 | 95 | 157 |
| velocity-1.5 | 203 | 325 | 0.36 | 0.8 | 74 | 260 |
| velocity-1.6 | 216 | 188 | 0.19 | 0.81 | 42 | 152 |
| xalan-2.4 | 409 | 119 | 0.14 | 0.81 | 56 | 96 |
| xalan-2.5 | 714 | 502 | 0.36 | 0.8 | 258 | 402 |
| xalan-2.6 | 773 | 525 | 0.29 | 0.8 | 221 | 420 |
| xalan-2.7 | 763 | 952 | 0.74 | 0.8 | 565 | 762 |
| xerces-1.2 | 377 | 115 | 0.13 | 0.8 | 48 | 92 |
| xerces-1.3 | 391 | 193 | 0.09 | 0.81 | 37 | 156 |
| xerces-1.4 | 269 | 555 | 0.33 | 0.8 | 88 | 445 |
| **Mean** | **265.08** | **222.17** | **0.25** | **0.82** | **73.39** | **178.36** |