OSCLUS: Combining Clustering and Component-Sensitive Algorithm for Cross-Project Software Fault Prediction

Inbal Roshanski, Meir Kalech, Amir Elmishali

OSCLUS: Combining Clustering and Component-Sensitive Algorithm for Cross-Project Software Fault Prediction

Inbal Roshanski, Meir Kalech and Amir Elmishali
Ben Gurion University of the Negev
Be'er Sheva, Israel
inbalros@post.bgu.ac.il , kalech@bgu.ac.il , amir9979@gmail.com

Abstract

Today, software is an indispensable part of our daily life. Unfortunately, the more advanced and complicated software becomes, the more likely it will malfunction. Predicting the probability of faulty software components can assist in maintaining program effectiveness. The volume and quality of historical data collected by the project's version control and issue tracker technologies is an essential factor for the accuracy of the prediction. However, for new projects, for example, there is no historical data from which to learn. This is referred to as cross-project software fault prediction. Previous work presented cross-project software fault prediction models, in which fault prediction models from other projects are used to determine whether or not the components of a new project are faulty. In this paper, we propose a novel approach named OSCLUS. OSCLUS is a hybrid algorithm that combines the strengths of two well-known state-of-the-art algorithms for cross-project software fault prediction approaches: clustering and component-sensitive. The prediction of OSCLUS is more accurate than the basic algorithms on their own, according to an evaluation conducted on a large-scale dataset containing 25 software projects.

1 Introduction

Software is an essential part of our lives today. The more complicated and complex the software, the more prone it is to faults. More importantly, the more significant the software in our lives, the more critical and dangerous the fault becomes. Finding and fixing software bugs is one of the challenges for software developers, and many companies employ professionals whose job is to perform the debugging process optimally. Software fault prediction is an automatic tool that can help to isolate the faulty software component. Software fault prediction is a binary classification problem where, for each program component, a label is to be determined that tells whether or not the component is at fault [1; 2]. One of the significant problems with this tool is that it requires a lot of historical data to be collected from the software. This is not a realistic assumption for all software. This is a well-known problem in the machine learning field, called **the cold-start problem**. The most familiar cold-start phenomenon occurs in recommendation systems. [3].

In this paper, we present a novel approach that combines two successful and well-known methods to deal with the cold problem, thus improving the prediction results, and get a hybrid method that is better than any of these methods on its own. This paper presents a novel approach that combines two successful and well-known strategies for dealing with this problem. Our hybrid method improves the prediction results, and it outperforms each of the basic strategies on its own. The first method is presented in [4]. Herbold suggests a clustering strategy – choosing the training set from the projects in the same cluster based on the projects' characteristic vectors. An alternative strategy is K-Nearest Neighbors. The prediction model is built based on the k most similar projects to the new project. The second method, the component-sensitive cross-project software fault prediction approach (OSCAR), is presented in our preview work [5]. The basis of this approach is to have a set of available projects, and each has a prediction model. The approach for a new project is twofold. First, a multi-class classifier is used to assign each of the new project components to a particular project from the available collection of projects. In the second phase, we go over all existing projects and utilize their prediction model for the components that have been associated with the same project. As a result, each component receives a prediction about whether it is faulty.

In this paper, we present OSCLUS. A hybrid method that combines OSCAR and Clustering methods. Clustering is the task of grouping a set of objects so that objects in the same group are more similar to each other than to those in other groups. The clusters allow us to narrow the available projects for OSCAR. This way, OSCAR can better target the "most suitable" model for each component.

We suggest two different types of clustering for this purpose:

1. Projects-based clustering - create clusters of similar projects. We propose two variations. (1) given a new project, we find the most similar cluster to the project and use its projects for OSCAR. (2) for each cluster, we merge the projects' records that belong to the cluster and build a fault prediction model for each cluster. Then we use the clusters as projects for OSCAR.
2. Components-based clustering - in this type of clustering, we consider the components rather than the projects by creating clusters of similar components. Then we use the clusters as projects for OSCAR.

To evaluate OSCLUS, we compare all the above variations' accuracy against previous cross-project clustering ap-

proaches from the literature that do not use OSCAR and against the basic OSCAR without clustering.

We use the HAWAII dataset that includes 25 real-world apacHe jAva softWAre projects collected from gIt and Bugzilla. The results show that the accuracy of the OSCLUS approach outperforms the basic methods on their own, in some variations with statistical significance.

The rest of the paper is organized as follows: In the next section, we present a general background to the software fault prediction problem and define this problem in the context of cross-project fault prediction. In Section 3 we introduce OSCLUS as a novel hybrid method to solve cross-project software fault prediction problem. In Section 4 we present the research questions and the experiments we conducted. In Section 5 we discuss the difficulties and challenges we cope with in the experiments. In Section 6 we present a summary of the literature published in the domain of software fault prediction problem. In Section 7 we conclude and present future work.

2 Background and Problem Definition

Typically, a software project is developed in a manner that every few revisions a new version is declared. Let $\mathcal{S} = (s_1, ..., s_{cur})$ be a sequence of software versions of project $\mathcal{S}$, where s_1 stands for the first version and s_{cur} for the current version (last one). A version of a software project $\mathcal{S}$ is composed of a set of components $s_i = \{c_{i_1}, \ldots, c_{i_m}\}$.

For the ongoing discussion, we assume that a software component is a class that belongs to a software project. Nevertheless, our algorithm also holds for different component granularity levels, e.g., where each component is a function, a program statement, a file, etc. A software component could be either healthy or faulty. Software fault prediction is a classification problem: given a software component, the goal is to determine its state - either healthy or faulty. Supervised machine learning algorithms are commonly used to solve classification problems. They work as follows: They are given a *training set* as input, which includes pairs of instances and their labels, i.e., the class (status) of each instance. The instances are processed before they are inserted into the training set to extract features from them. In our case, instances are software components (later, we elaborate on the feature extraction), and the labels include whether each software component is healthy or not. Using the training set, a learning algorithm outputs a *classification model*, which predicts a label for a new unlabeled instance (in our case, whether it is healthy or faulty).

Many factors can influence the quality of the software fault prediction model [6]. Lessmann *et al.* [7] describe different classification algorithms that are used for software fault prediction and conclude that the classification algorithm does not influence much on the accuracy of the prediction model. On the other hand, the software metric set (features) choice dramatically impacts the model accuracy. Previous work dealt with studying and developing various software metric sets and dividing them into three categories:

The first category includes **traditional metrics** based on static code analysis [8] The second category is **Object-Oriented**. These metrics target the effect of components on the system, and the relationship between components [9; 10]. Both traditional metrics and Object-Oriented metrics analyze the current version of source files without addressing past changes in the code. The third category includes the **process metrics**, which aim to measure the influence of changes in the code [11; 12]. This category has been proved as a critical factor in the success of the prediction model [13]. The historical data of the project is used to generate a within-project fault prediction model, formally: **Within-project fault prediction** creates a prediction model that is built within the context of a specific software project. Recall $\mathcal{S} = (s_1, ..., s_{cur})$ is a sequence of software versions of project $\mathcal{S}$, Within-project fault prediction aims to predict, for each one of the software components in the last version of the project (s_{cur}), whether it is faulty:

Definition 1 (Within-project fault prediction)**.** *A within-project software fault prediction algorithm gets as input $\mathcal{S}' \subseteq \mathcal{S} \setminus \{s_{cur}\}$ and returns a prediction model $PM_{s_{cur}}$ such that given a component $c_i \in s_{cur}$ it determines whether is it healthy or faulty.*

Within-project fault prediction relies on features extracted from previous versions of the software project used for the training set. However, this is not feasible for all software projects due to different reasons. Most commonly, the software might be new. In this case, it does not have historical versions. An additional possible reason is that the software is not publicly documented or may be classified, so access to historical versions is not possible for outsiders. This situation defines a cold-start project:

Definition 2 (Cold-start project)**.** *A software project that has no recorded historical versions. Let $\mathcal{S}^{CS}$ denote a cold-start project, then $\mathcal{S}^{CS} = \{s_{cur}\}$.*

Given a cold-start project, extracting features from the third category (process metrics) is infeasible. This makes the fault prediction task even harder since the process metrics are very influential and significant for creating a high-quality prediction model. More importantly, there are no historical versions to learn from, and there are no labels for any instance. That means that practically it is impossible to construct a within-project fault prediction model from the information available. Cross-project fault prediction suggests relying on other software projects to get labeled instances for learning a qualitative classification model for the cold-start project.

To formalize cross-project software fault prediction, we define a set of software projects. Let $\mathbb{SP}$ be a set of software projects $\mathbb{SP} = \{\mathcal{S}^1, ..., \mathcal{S}^n\}$, and let $\mathcal{S}^{CS} \notin \mathbb{SP}$ be the cold-start project.

Definition 3 (Cross-project fault prediction)**.** *A cross-project software fault prediction algorithm gets as input $\mathbb{SP}$ and a cold-start project $\mathcal{S}^{CS} = \{s_{cur}\}$, and returns a prediction model $PM_{s_{cur}}$, such that given a component $c_i \in s_{cur}$ it determines whether c_i is healthy or faulty.*

OSCAR Previous work proposed cross-project fault prediction methods, which learn a prediction model for the cold-start project by generating a training set collected from other projects or applying a within-project prediction model of one of these projects. This approach raises a challenge. When using a training set to learn a fault prediction model, an implicit assumption is that the training set and the test set are taken from the same distribution. This assumption explains why we can predict the label of a software component in the test set by learning from the training set. However, previous cross-project fault prediction methods predict **all** of the components in the new project using a fault prediction model which has been trained with instances of **other**

projects (s). Thus, the training set used to generate the fault prediction model is not produced from the same project as the test set (the new project). This drawback motivated the creation of OSCAR [5]. OSCAR examining every component in the test set (the cold-start project) by itself (rather than considering all the components together) and predict whether it is faulty according to the prediction model of the most "suitable" project among a set of other software projects ($\mathbb{SP}$). To determine which project is most suitable for a component, a multi-class classification model (belongingness classifier) is trained to learn the belongingness of a software component to a software project.

The training set to create such a belongingness classifier is assembled from all the software components from all the projects with version history ($\mathbb{SP}$). Each component's features for the belongingness training set include the same features extracted for the components in the within-project prediction model's training set. This component's class is the name of the project that it belongs to (rather than healthy/faulty in the within-project fault prediction model). Using this classification model, OSCAR can classify each component in the cold-start project to a project among the set of available projects $\mathbb{SP}$, and then use that project's within-project fault prediction model to determine whether the component of the cold-start project is faulty or not.

Based on OSCAR's belongingness classifier characteristics, there are two possible ways to improve the accuracy of OSCAR. 1) Improving the homogeneity of all the available projects. Doing so will make it easier for the belongingness classifier to determine the classification of each component classification. 2) Focusing the set of projects on a subset of the most suitable projects. The belongingness classifier will become a much more accurate expert if it learns from fewer, more appropriate projects to the specific cold-start project.

These two options for improving OSCAR accuracy motivated us to create OSCLUS hybrid-approach, where each one of the techniques we used is aimed to focus on one of the possible ways to improve the accuracy of OSCAR.

3 Method Description

In this paper, we present OSCLUS. This hybrid method combines the strength of two known methods in the software fault prediction domain. OSCLUS suggests improvement of the OSCAR method using a clustering algorithm. We believe that narrowing down the projects available for OSCAR and getting OSCAR to focus on the most suitable projects OSCAR will yield better accuracy results. Our process consists of two steps. First, we create the clusters, Then use those clusters combined with OSCAR in different variations. We suggest two ways to create clusters: by projects and by components.

3.1 Projects-based clustering

In this type of clustering, each cluster is comprised of similar projects. To cluster similar projects, we consider two types of characteristic vectors for a project:

1. **Meta-Features:** we extract meta-features based on the features of the projects' components in the training sets. These meta-features include the mean, std, min, and max values of the numeric features and the unique values, frequency, and count values for the categorical features. We do not include the label (faulty or not) in this process.
2. **Project Features:** we use the source code of the project to extract features representing the project. Each project is represented by a profile (vector of characteristics) including 21 known features: Halstead Cumulative Length and Volume, Halstead Difficulty, Halstead Bugs, Halstead Cumulative Bugs, Halstead Effort, Halstead Vocabulary, Halstead Volume, # Methods, # Comments,# Statements,# Packages, # classes, Loc, average Loc, max cc, average cc,tcc, cumulative Number Of Comments, maintainability Index and maintainability Index NC.

The K-means algorithm is then used to cluster the projects, either by the Meta-Features or by the project Features characteristic vector. Once the projects' clusters have been created, there are two variants to use OSCAR combined with the clusters.

OSCAR *Inside Cluster* In this variant, we reduce the software projects set $\mathbb{SP}$ to include only the cluster projects rather than all the projects. Thus, given a cold-start project, we create a characteristic vector for that project (either by Meta-Features or Project Features). Then we use a similarity function to match this project to one of the clusters in $\mathbb{SP}$, and use the projects in this cluster as the training set $\mathbb{SP}$ for OSCAR.

OSCAR *With Clusters* In this variant, we introduce the software projects set $\mathbb{SP}$ to include the clusters as projects. In this way, each project in $\mathbb{SP}$ actually includes a cluster of projects. To transform a cluster of projects into a single project, we merge all the projects' components from the same cluster and create a within-project fault prediction model based on the merged projects. Then, given a cold-start project, we use the clusters in $\mathbb{SP}$ as the training set for OSCAR.

To summarize, there are two ways of creating clusters and two variations for using OSCAR combined with the clusters. These lead to a total of four combinations: (1) Meta-Features+**OSCAR *Inside Cluster***, (2) Meta-features+**OSCAR *With Clusters***, (3) Project Features +**OSCAR *Inside Cluster***, and (4) Project Features +OSCAR with clusters.

Projects-based clustering consists of two steps. First, we create the clusters. To be able to create the clusters, the algorithm obtains the clustering method - in our case - projects or components, and the characteristic vector extraction Method, in our case - Meta-Features or Projects Features as explained in Section 3.1). This allows us to measure the similarity between projects or components. Then in the second stage, we combined those clusters with OSCAR. To combine OSCAR in different ways, the algorithm obtains the available project's method that uses the clusters to create $\mathbb{SP}$ - the set of software projects available for OSCAR. We present the methods used in Sections 3.1 and 4.1.

3.2 Components-based clustering

In this type of clustering, each cluster consists of similar components. To cluster the components, we first combine all the components from all the available project's training sets ($\mathbb{SP}$). The K-means algorithm is then used to reorganize the components into clusters forming new software projects set $\mathbb{SP}$ in the size of the number of required clusters. Then, given a cold-start project, we use the clusters in $\mathbb{SP}$ as the training set for OSCAR.

4 Evaluation

This section describes a series of experiments conducted to evaluate OSCLUS against other existing cross-projects software fault prediction algorithms. In particular, we address the following research questions:

RQ1. *Does combining clustering methods with* OSCAR *improve* OSCAR*'s accuracy?*

> RQ1 asks whether reducing the software projects set $\mathbb{SP}$ by using clustering methods rather than using all the available projects improves OSCAR's accuracy.

RQ2. *Which clustering configuration performs the best?*

> In Section 3 we presented different configurations for clustering. RQ2 asks which configuration receives the best prediction results.

In Section 4.1 we describe the entire experimental setup. In Section 4.2 we present the results and analyze them.

4.1 Experimental Setup

In this section we describe the data-set used for the experiments, OSCLUS implementation, competing algorithms, the experimental process and the evaluation metrics.

Data-sets:

We evaluated the algorithms with a large-scale data-set that we collected. Creating this data-set is one of the paper's contributions. The standard known data-set used for cross-project defect prediction, PROMISE, is relatively small and outdated. This led us to create the following data-set.

HAWAII*:* [1]

To be able to use clustering in a meaningful way, a sufficient amount of projects and features are required. PROMISE [14], a well-known data-set for cross-project defect prediction, includes only ten projects. Also, APRIL [5], a state-of-the-art data-set, includes only eight projects. This small number of projects could be too small to evaluate the clustering-based methods. Thus we created a new data-set called HAWAII. HAWAII includes 25 real-world Java projects. These projects belong to the APACHE software foundation. Apache, the world's largest open-source foundation, has hundreds of programmers and projects. Apache projects are independent and managed independently[2]. The projects we used are listed in Table 1. Creating this large data-set is an additional contribution to this paper. In HAWAII, a software component granularity level is a class. The training set includes a set of classes' features, where each component is labeled whether it is faulty or not. Manually labeling the root cause of past bugs is not scalable. Instead, we automatically extract the training set from the project's *issue tracking* and *version control system*, as described below. Most projects use an issue tracking system, such as Jira and Bugzilla, and a version control system, such as Git and Mercurial. Issue tracking systems record all reported bugs and track changes in their status. They associate each bug with a unique issue ID. Version control systems track modifications – commits – done to the source files. Commonly, a commit contains only the required changes to resolve a specific task. A best practice in software development, usually enforced, is to add a modification description to each commit. In particular, when a commit fixes a bug, the bug's issue ID should be written in the commit's description. We use this information to match fixed bugs to the commit that fixed them. For a bug X, let $\Phi(X)$ denote the set of software components modified to fix X, as mentioned in the commit. In the absence of manual labeling of faulty software components, we assume that all components in $\Phi(X)$ are blamed as faulty.

To create the within-project software fault prediction, we chose five versions and their reported bugs. We used the first four versions as the training set and the last one as the test set. Let $(s_1, s_2, .., s_5)$ be the selected five versions of project $\mathcal{S}$. A version control system records for every version s_i its release date, and an issue tracking system records for every bug, the date it was reported, and its status. We associate with version s_i the entire set of bugs reported between the release of version s_i and the release of version s_{i+1}. To ensure a reasonable amount of bugs and changes associated with each version, we chose versions that the time between versions is at least six months.

To summarize, we gathered 125 versions from real-world-up-to-date java software projects, which left us with 25 available projects to use. Each project consists of the same number of versions, making the data-set fair and appropriate to the field. We chose versions with approximately the same percentage of bugs. Moreover, We tried to minimize the difference between the number of bugs in the training set and each project's test set. We extracted even more information about these versions, which enabled us to include sophisticated features, such as statistical features regarding methods belonging to the class. In detail, for each feature we collected regarding the methods that belong to the class (the component), we computed the mean, std, min, and max values of each numeric feature. We computed the unique, frequency, and count values for categorical features. We used only non-process features to imitate the cold-start problem for the features derived from the components. The complete list of features can be found in Github repository mining[3].

OSCLUS implementation:

Implementing OSCLUS includes the implementation of all clustering variations and OSCAR. For the within-project software fault prediction models used by OSCAR, we used the same set-up and method used by [5] - Balanced Random Forest learning algorithm with 1000 estimators. For the clustering algorithm, we used the K-means algorithm (sklearn.cluster python library). We tried different numbers of clusters: 2,3,4, and 5. We found that using k=3 as the number of clusters yields the best result in a preliminary comparison. Decreasing the number of clusters or increasing the number to 5 led to worse performance. Increasing the number to 4 did not significantly affect the accuracy of the classification model.

Competitive Algorithms

To estimate the improvement of our hybrid approach OSCLUS, we compared all the OSCLUS variations with OSCAR without clustering and with clustering methods without OSCAR. To answer RQ2, we used two baseline clustering methods that do not use OSCAR. The first variation is called "Majority-Voting." It proceeds as follows:

1. Cluster the projects in $\mathbb{SP}$ into clusters either by Meta-Features or Project Features.

[1] https://github.com/inbalros/HAWAII

[2] https://httpd.apache.org/.

[3] https://github.com/amir9979/repository_mining/

Project name	Train components	Train bugs	Train bugs-components ratio	Test components	Test bugs	Test bugs-components ratio
Archiva	2747	255	9.28%	643	50	7.78%
Cassandra	1282	93	7.25%	336	16	4.76%
CommonsCodec	275	96	34.91%	103	16	15.53%
CommonsDBCP	381	18	4.72%	98	27	27.55%
CommonsIO	608	122	20.07%	224	58	25.89%
CommonsJexl	491	60	12.22%	201	16	7.96%
Commonslang	980	238	24.29%	291	67	23.02%
CommonsValidator	524	34	6.49%	137	6	4.38%
Continuum	1792	256	14.29%	554	78	14.08%
Crunch	1835	258	14.06%	531	61	11.49%
DirectoryServer	1857	114	6.14%	1218	93	7.64%
Helix	2630	242	9.20%	640	21	3.28%
Juneau	2332	324	13.89%	873	25	2.86%
Knox	2357	345	14.64%	847	66	7.79%
Metron	2203	287	13.03%	744	86	11.56%
MyFaces	1270	53	4.17%	418	18	4.31%
MyfacesTobago	2388	298	12.48%	709	112	15.80%
Nutch	1320	229	17.35%	481	55	11.43%
Parquet	1956	284	14.52%	561	48	8.56%
QpidJMS	2067	251	12.14%	594	105	17.68%
Samza	1463	265	18.11%	849	130	15.31%
Struts	2975	143	4.81%	731	18	2.46%
Surefire	510	94	18.43%	212	12	5.66%
Tapestry-5	4738	1942	40.99%	1433	108	7.54%
Tika	1378	282	20.46%	469	113	24.09%
AVERAGE	1694	263	14.72%	556	56	11.54%

Table 1: Basic information about HAWAII's projects.

2. Given a cold-start project, match that project to the most similar cluster C_i, based on the characteristic vector.
3. Use the within-project fault prediction models of the projects in cluster C_i as an ensemble of models. The prediction for each component of the cold-start project will be chosen by majority voting between these models.

Note that steps 1 and 2 are similar to the "**OSCAR *Inside Cluster***" variation. The third step is different. While **OSCAR *Inside cluster*** uses OSCAR for the prediction method, the Majority-Voting baseline uses the majority voting prediction method.

The second variation is called the "Best-Cluster-Model". A variation of this method has been presented by [15]. It proceeds as follows:

1. Cluster the projects in $\mathbb{SP}$ into clusters either by Meta-Features or Project Features.
2. Given a cold-start project, match that project to the most similar cluster C_i, based on the characteristic vector.
3. Create a within-project fault prediction model for C_i based on a merged set of all project components belonging to C_i. Use this model to predict the cold-start project's components.

To summarize, there are four combinations for the competitive algorithms. Two baseline methods, each uses two clustering methods: (1) Meta-Features+Majority-voting, (2) Meta-Features+Best-Cluster-Model, (3) Project Features +Majority-voting, and (4) Project Features +Best-Cluster-Model. Table 2 summarizes the clustering configurations we proposed as well as the baselines.

Experiment process:

The experiment was carried out in rounds, where at each round, one project was simulated as the cold-start project by using only its last version - the test set. The other projects were included in $\mathbb{SP}$, and used as the projects that have within-project fault prediction models. The results of the experiments are the average scores of the collected metrics results over all the rounds.

Metrics:

Software Fault prediction is a binary classification task. Binary classifiers accept an instance to classify (in our case, a software component), and output one of two classes: positive (in our case, a faulty software component) or negative (a healthy software component).

We define four possible outcomes of the classifier as follows:

1. TP (True Positive): A component that belongs to class faulty was classified as faulty.
2. FP (False Positive): A component that belongs to class healthy was classified as faulty.
3. TN (True Negative): A component that belongs to the class healthy was classified as healthy.
4. FN (False Negative): A component that belongs to the class faulty was classified as healthy.

True-positive rate (TPR) and false-positive rate (FPR) are two primary metrics used to evaluate binary classifiers: TPR is the proportion of correctly identified positives, and FPR is the proportion of negatives wrongly identified.

Another known metric is precision (also called positive predictive value), which is the fraction of positive identifications among the retrieved instances and calculated as $\frac{TP}{TP+FP}$. Recall (also known as sensitivity) is the fraction of actual positives that have been retrieved over the total amount of positives instances and calculated as $\frac{TP}{TP+FN}$.
To evaluate the results, we use the following metrics:

1. **F score**: a weighted average of precision and recall.
2. **F2 score**: In this metric, recall has a higher weight than precision. In the software domain, this feature is significant since classifying components as healthy incorrectly is worse than classifying valid components as

Using OSCAR or Baseline	Clustering Type	Characteristic Vector	Cross-Projects Prediction
Using OSCAR	**Components-based clustering**	Available Features	OSCAR with Clusters
Using OSCAR	**Projects-based clustering**	**Meta-Features**	OSCAR Inside Cluster
			OSCAR with Clusters
		Project Features	OSCAR Inside Cluster
			OSCAR with Clusters
Baseline	**Projects-based clustering**	**Meta-Features**	Majority-voting
			Best-Cluster-Model
		Project Features	Majority-voting
			Best-Cluster-Model

Table 2: A summary of the proposed clustering-based configurations, as well as the baseline methods.

faulty

$$F2 = 5 \cdot \frac{precision \cdot recall}{4 \cdot precision + recall}$$

3. **Area Under the Curve (AUC)**: A commonly used metric for evaluating binary classifiers [16]. The AUC metric is calculated as the area under the receiver operating characteristic (ROC) curve. The ROC curve plots the FPR as a function of the TPR.
4. **Precision Recall Curve (PRC)**: A precision-recall curve shows the relationship between precision and recall for different thresholds. The main difference between ROC curves and precision-recall curves is that the number of true-negative results is not used for PRC, making the PRC a better metric for imbalanced data.

4.2 Results

To answer RQ1 and RQ2, we performed experiments on HAWAII data-set with the configurations presented in Table 2. We present the experiments in the following order: First, we discuss the results of the Projects-based clustering configurations. Then we compare the results achieved by using Projects-based clustering against the Components-based clustering. Finally, we examine the improvement the using clustering to the basic OSCAR.

Project-based clustering

Tables 3 and 4 present the average scores by comparing **OSCAR *Inside Cluster*** and **OSCAR *With Clusters*** with the baseline methods for experiments with Meta-Features and Project Features, correspondingly. Every row represents an algorithm for both tables, and the columns represent the metrics. The highest average in each column is bolded. It is easy to see that **OSCAR *With Clusters*** method achieves the highest average along with all metrics in both clustering similarity methods (Meta-Features and Projects Features). Although the improvement is relatively small, it is consistently higher, along with all the metrics. In this domain, even a small improvement is significant.

Although, on average, using the "Meta-Features" presentation with **OSCAR *With Clusters*** yields the best results in both clustering similarity methods, we did not find these results statistically better than **OSCAR *Inside Cluster***.

Project-based clustering VS components-based clustering

In the next experiment, we compare the performance of projects-based clustering against components-based clustering. Table 5 presents the Components-based clustering results with 5 and 7 clusters against the best-performed algorithm we achieved while using the project-based clustering with Meta-Features+**OSCAR *With Clusters***. For the components-based clustering method, we tried different numbers of clusters and found in preliminary comparison that 5 and 7 yield the best result.

Algorithm	F	F2	AUC	PRC
OSCAR Inside Cluster	0.270	0.407	0.639	0.459
Majority-voting	0.309	0.427	0.672	0.453
Best-Cluster-Model	0.316	0.439	0.680	0.463
OSCAR With Clusters	**0.319**	**0.442**	**0.688**	**0.466**

Table 3: Project-based clustering, k=3 using Meta-Features. Average metrics over all the projects.

Algorithm	F	F2	AUC	PRC
OSCAR Inside Cluster	0.276	0.423	0.643	**0.464**
Majority-voting	0.311	0.420	0.673	0.444
Best-Cluster-Model	0.309	0.432	0.677	0.462
OSCAR With Clusters	**0.312**	**0.441**	**0.686**	**0.464**

Table 4: Project-based clustering, k=3 using Project Features. Average metrics over all the projects.

We can see that, on average, the best results are achieved by project-based clustering with Meta-Features+**OSCAR *With Clusters***. However, this result is not statistically significant.

OSCAR VS clustering

Next, we compare the performance of combining OSCAR with clustering to the basic version of OSCAR.

Table 6 presents the results of basic OSCAR compared to the best-performed algorithm we achieved while using the project-based clustering and the component-based clustering.

It is clear that all clustering-based algorithms perform better, on average than basic OSCAR. Nevertheless, only the AUC gap was approved as statistically significant (with a significance level of 5%).

Conclusions of the Results

- Regarding RQ4, we can conclude that combining clusters with OSCAR improves the cross-projects fault prediction performance. The basic OSCAR was beaten by the combined clusters methods on average in

Algorithm	F	F2	AUC	PRC
Components-based clustering k = 5	0.317	0.438	0.686	0.462
Components-based clustering k = 7	0.311	0.430	0.682	0.458
project-based clustering, Meta-Features+OSCAR With Clusters	**0.319**	**0.442**	**0.688**	**0.466**

Table 5: Average metrics over all the projects

Algorithm	F	F2	AUC	PRC
OSCAR	0.277	0.406	0.649	0.440
Components-based clustering k = 5	0.317	0.438	0.686	0.462
project-based clustering	**0.319**	**0.442**	**0.688**	**0.466**
Meta-Features+OSCAR With Clusters				

Table 6: Average metrics over all the projects

terms of F, F2, AUC, and PRC. The difference among most metrics is not statistically significant, except for the AUC metric.

- Regarding RQ5, using projects-based clustering and applying Meta-Features presentation with **OSCAR *With Clusters*** method configuration performs the best on average (in terms of F, F2, AUC, and PRC). Nevertheless, this result is not statistically significant.

5 Threats To Validity

While experimentation in software engineering is necessary, it is not easy to run experiments in software due to the specific environmental conditions of each project [17]. Moreover, it is challenging to conclude from experiments in the software domain since it is a vast domain with significant variance between projects. We tried to cope with these challenges by adding HAWAII data-set and increasing the number of features. Ideally, our study would be replicated with more projects from different domains, different metrics, and more project characteristics.

The process of data extraction for HAWAII has been done systematically using automated and proven data extractors. We got a vast number of features and chose a subset of them, adding statistical information regarding each class's methods. As shown before, the features we used for the OSCLUS classifier affected greatly on OSCAR accuracy. Hence, it should be mentioned that other researchers can choose different sub-set of features and receive different results.

6 Related Work

As discussed above, Cross-project software fault prediction uses a training set collected from different projects. The challenge in cross-project fault prediction is choosing the most suitable training set or prediction model for a new project among multiple projects.

Zimmermann *et al.* [18] experiment logistic regression prediction models to transfer from one project to another on real-world projects. They consider a project's prediction model as a strong predictor if all the measures that the prediction produced are above a certain threshold. Only 3.4% of the cross-project prediction models passed the threshold. They conclude that cross-project prediction is a challenging task, meaning that simply using projects from the same domain for training does not yield an accurate prediction model. Process, data, and domain need to be quantified, understood, and evaluated before prediction models are built and used. He *et al.* [15] use a brute force strategy that considers all the combinations between the project versions in PROMISE to find the best prediction model that can be created for the cold-start project. Obviously, the challenge is to find which combination of projects will yield a good prediction model in advance. To address this challenge, They investigate the relationship between the training set's distributional characteristics and the cross-project fault prediction accuracy. They found that distributional data characteristics are informative for training data selection. Herbold [4] suggests some strategies to select the training data for a project with no recorded data. These strategies are also based on distributional characteristics of the available data. He suggests a clustering strategy – choosing the training set from the projects in the same cluster based on characteristic vectors of the projects. Guo *et al.* [19] suggests two methods to deal with the cold-start problem in the analytic software field. The first algorithm finds the best overall performing fault prediction model. It is selected according to the average of the F2 metric. The second algorithm matches a cold-start project to the most similar project among the set $\mathbb{SP}$ (represented by a vector of characteristics), and uses its within-project fault prediction model. He *et al.* [20] cope with the problem of imbalanced feature sets between the source and target projects in cross-project fault prediction. They map the instances' characteristic vectors of all projects onto the same latent space. They evaluate their model on eleven projects, most taken from PROMISE. Zhang *et al.* [21] suggest two types of unsupervised classifiers of fault prediction. The first is a distance-based classifier (k-means), and the second is a connectivity-based classifier. They compare these two types of classifiers using PROMISE and another two less known and very small data-sets. Bal and Kumar [22] use the PROMISE data-set. The main contribution of this paper is by empirically showing that Extreme Learning Machine techniques are comparable to known predictors. Wen *et al.* [23] compare cross-project fault prediction combinations between source selection (features and projects) and transfer learning. They use the PROMISE data-set and use only logistic regression classifiers in their experiments.

Prior studies have mainly investigated how to select the training data from other projects while considering the projects as a whole. They focus on the characteristics of the project as a criterion to find the most similar projects that can be used to create the cross-project fault prediction model. The paper presented in [5] presents a different approach where the cold-start project's instances are distributed among other projects' prediction models. OSCAR is a component-sensitive cross-project approach that first separately classifies each component in the new project to its most similar project. Then, OSCAR uses the fault prediction model of that project to predict whether the component in the new project is faulty. Although this approach is unique and has managed to get better results than previous approaches, we believe that it has not reached its full potential. This is because the classification of each component is among all existing projects. As shown in this article, we believe that reducing and more intelligent selection of projects using clusters will allow greater success for the proposed algorithm. Moreover, this paper proposes HAWAII - a more up-to-date and large-scale data-set, which is another contribution of the paper.

7 Conclusions and Future Work

Models for software fault prediction can assist programmers in isolating faults and planning tests. To produce an accurate fault prediction model, significant size training data is required to learn from, which is unrealistic for cold-start projects. Previous work proposed learning from other existing projects. However, the training and test sets were

drawn from various distributions, resulting in non-accurate prediction models. In this paper, we propose OSCLUS, a novel hybrid strategy that combines OSCAR and clustering approaches to improve the accuracy of each method individually. We compared OSCLUS to the standard OSCAR and two other clustering variants. We used a novel data-set HAWAII to evaluate OSCLUS variants, which contains 25 real-world Java projects with 125 versions in total. The results demonstrated that by intelligently restricting the available relevant projects for OSCAR, it is possible to better focus each component on the "best suited" model. The OSCLUS methodology outperformed the baseline approaches, and OSCLUS increased OSCAR's performance by integrating clustering methods with OSCAR and constructing different types of clusters and data characteristic vectors. In the future, clusters will be combined in more sophisticated ways to boost OSCLUS performance even further.

References

[1] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2011.

[2] Amir Elmishali, Roni Stern, and Meir Kalech. An artificial intelligence paradigm for troubleshooting software bugs. *Engineering Applications of Artificial Intelligence*, 69:147–156, 2018.

[3] Cheng-Te Li, Chia-Tai Hsu, and Man-Kwan Shan. A cross-domain recommendation mechanism for cold-start users based on partial least squares regression. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 9(6):1–26, 2018.

[4] Steffen Herbold. Training data selection for cross-project defect prediction. In *Proceedings of the 9th International Conference on Predictive Models in Software Engineering*, page 6. ACM, 2013.

[5] Inbal Roshanski, Meir Kalech, Roni Stern, and Amir Elmishali. The cold start problem in software fault prediction. 2020.

[6] Susan A Sherer. Software fault prediction. *Journal of Systems and Software*, 29(2):97–105, 1995.

[7] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, 34(4):485–496, 2008.

[8] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[9] Fernando Brito Abreu and Rogério Carapuça. Object-oriented software engineering: Measuring and controlling the development process. In *Proceedings of the 4th international conference on software quality*, volume 186, pages 1–8, 1994.

[10] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[11] Todd L Graves, Alan F Karr, James S Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Transactions on software engineering*, 26(7):653–661, 2000.

[12] John C Munson and Sebastian G Elbaum. Code churn: A measure for estimating the impact of code change. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 24–31. IEEE, 1998.

[13] Peng He, Bing Li, Xiao Liu, Jun Chen, and Yutao Ma. An empirical study on software defect prediction with a simplified metric set. *Information and Software Technology*, 59:170–190, 2015.

[14] Tim Menzies, Bora Caglayan, Ekrem Kocaguneli, Joe Krall, Fayola Peters, and Burak Turhan. The promise repository of empirical software engineering data, 2012.

[15] Zhimin He, Fengdi Shu, Ye Yang, Mingshu Li, and Qing Wang. An investigation on the feasibility of cross-project defect prediction. *Automated Software Engineering*, 19(2):167–199, 2012.

[16] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[17] Victor R Basili, Forrest Shull, and Filippo Lanubile. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering*, 25(4):456–473, 1999.

[18] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 91–100. ACM, 2009.

[19] Jin Guo, Mona Rahimi, Jane Cleland-Huang, Alexander Rasin, Jane Huffman Hayes, and Michael Vierhauser. Cold-start software analytics. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 142–153. ACM, 2016.

[20] Peng He, Bing Li, and Yutao Ma. Towards cross-project defect prediction with imbalanced feature sets. *arXiv preprint arXiv:1411.4228*, 2014.

[21] Feng Zhang, Quan Zheng, Ying Zou, and Ahmed E Hassan. Cross-project defect prediction using a connectivity-based unsupervised classifier. In *Proceedings of the 38th International Conference on Software Engineering*, pages 309–320. ACM, 2016.

[22] Pravas Ranjan Bal and Sandeep Kumar. Cross project software defect prediction using extreme learning machine: An ensemble based study. In *ICSOFT*, pages 354–361, 2018.

[23] Wanzhi Wen, Bin Zhang, Xiang Gu, and Xiaolin Ju. An empirical study on combining source selection and transfer learning for cross-project defect prediction. In *2019 IEEE 1st International Workshop on Intelligent Bug Fixing (IBF)*, pages 29–38. IEEE, 2019.