CrossPare: A Tool for Benchmarking Cross-Project Defect Predictions

Steffen Herbold
Institute of Computer Science
Georg-August-Universität Göttingen, Germany
Email: herbold@cs.uni-goettingen.de

***Abstract*—During the last decade, many papers on defect prediction were published. One still for the most part unresolved issue are cross-project defect predictions. Here, the aim is to predict the defects of a project, with data from other projects. Many approaches were suggested and evaluated in recent years. However, due to the usage of different implementations and data sets, the comparison between the work is a hard task. Within this paper, we present the tool CrossPare. CrossPare is designed to facilitate benchmarks for cross-project defect predictions. The tool already implements many techniques proposed within the current state of the art of cross-project defect predictions. Moreover, the tool is able to load different data sets that are commonly used for the evaluation of techniques and supports all major performance metrics. Through the usage of CrossPare other reseachers can improve the comparability of their results and possibly also reduce their implementation efforts for new cross-project defect prediction techniques by reusing features already offered by CrossPare.**

I. Introduction

Due to its potential for the support of software quality assurance, defect prediction has been under investigation for a long time. However, due to the hardness of the problem, parts are still unresolved. Especially the prediction of defects in a cross-project context, i.e., with data that is not from the project for which the prediction is being made. In a study by Zimmermann et al. [1], the authors determined that only 3.4% of their cross-project predictions achieved more than 0.75 recall, precision, and F-measure. While this exact value is influenced by the data of the case study and may be too low in general, it demonstrates how hard the problem of cross-project defect predictions is.

Over the years, researchers have proposed many approaches for the improvement of the performance of cross-project predictions. Most approaches apply the concept of transfer learning, i.e., the usage of information from the target product to either process the training data, select a subset of training data, or manipulate the defect prediction model directly in some other way with this knowledge. The approaches suggested were quite diverse. Turhan et al. [2] suggested a k nearest neighbor approach where only the closest entites to the target project are selected, Watanabe et al. [3] and Nam et al. [4] suggest approaches for the normalization of data, Ma et al. [5] suggest to use data weighting, and [6] suggest data transformation. Some other approaches focus rather on the selection of appropriate projects from a pool of candidates. Zimmerman et al. [1] themselves already suggest a decision tree based on context factors (e.g., programming languages and technologies used). Herbold [7] instead suggest to use the distribution of the data to select projects. He et al.[8] propose to use the separatebility between projects to select data.

While this is only a selection of the work produced on cross-project defect prediction, all of the above share one problem: they are all implemented independent of each other and use different data and performance measures for the evaluation of the performance. Some papers perform some baseline comparisons, e.g., to the cross-validation performance or to the k-nearest neighbor by Turhan et al. [2]. However, an overall comparison between the suggested approaches is still almost impossible which makes it difficult for researchers to compare their results among each other.

Within this paper, we want to present CrossPare, a tool created for the benchmarking of cross-project defect prediction techniques. CrossPare is developed as open-source project (https://crosspare.informatik.uni-goettingen.de/) and can be reused by other researchers who want to compare their techniques or contribute their techniques to CrossPare. The key features that CrossPare provides are the following.

- Support for multiple public defect prediction data sets that can be used for the comparison of techniques.
- Implementation of many cross-project defect prediction techniques proposed within the state of the art.
- Evaluation of the results with all commonly used performance metrics.
- Easily configurable benchmarks for the detailed evaluation of techniques.
- Extensibility with new loaders for data formats, defect prediction techniques, and performance measures.

With CrossPare, we provide researchers with a powerful tool they can use to benchmark their defect prediction techniques and compare their results with others. CrossPare removes the need to re-implement the state of the art over and over again in order to provide comparisons. Moreover, by extending CrossPare with their new techniques, researchers can make sure that others are able to compare their suggested techniques properly to better disseminate their results.

The remainder of this paper is structured as follows. First, we discuss the key aspects of the architecture and implementation of CrossPare in Section II. We introduce the interfaces of CrossPare against which defect prediction techniques are implemented. Then we show how the benchmark workflow of CrossPare works and explain how users can define their

own benchmark configurations. Afterwards, we give a detailed overview of the features of CrossPare, including a list of the already supported cross-project defect prediction techqniues in Section III. Finally, we conclude our paper and give an outlook on future work in Section IV.

II. Architecture and Implementation

CrossPare is implemented as a stand-alone Java application that is called via command line. Currently, CrossPare requires at least Java version 7. In this section, we consider three key aspects of the architecture and implementation: 1) the usage of WEKA [9] as machine learning core; 2) strict programming against interfaces in order to allow the components of CrossPare and by extension the defect prediction approaches to be interchangable; and 3) easy configuration of benchmark experiments.

A. Machine Learning

The first aspect of CrossPare is the usage of WEKA as machine learning core. WEKA provides many state of the art machine learning algorithms, as well as many powerful data processors which can be useful for the implementation of techniques for cross-project predictions. WEKA can either be used as a stand-alone Java application or as a library. Since CrossPare is implemented in Java, we decided on the latter, i.e., to use WEKA as a library and call its functionality directly within CrossPare. This way, we can communicate with WEKA directly using only the main memory without having to store information to the hard disk, which would degrade performance greatly.

In order to simplify the communication with the WEKA library, we decided to re-use the data model of WEKA internally in CrossPare. The data representation within WEKA consists of basic parts: `Instance` objects which are basically an array of doubles and represent values of the data and `Instances` objects which are a complete data set. CrossPare loads all data into WEKA `Instances` and can, therefore, easily call any functionality provided by WEKA from within CrossPare.

B. Interfaces

The second key aspect of CrossPare is that the architecture prescribes a set of interfaces, depicted in Figure 1. All functionality of CrossPare is hidden behind these interfaces. The core of CrossPare is programmed strictly against these interfaces. Thereby, any class that implements the interface, can be directly used within CrossPare. The interfaces are the following.

- `IVersionLoader`: import of data into CrossPare, e.g., for the defect prediction data set curated by Jureczko and Madeyski [10].
- `IVersionFilter`: responsible for removing software versions from the loaded projects that do not meet certain minimal criteria, e.g., regarding their size.
- `IProcessingStrategy`: applies a processor to the test and training data, e.g., normalization. The test and the training data are passed seperately, such that they can be treated differently.

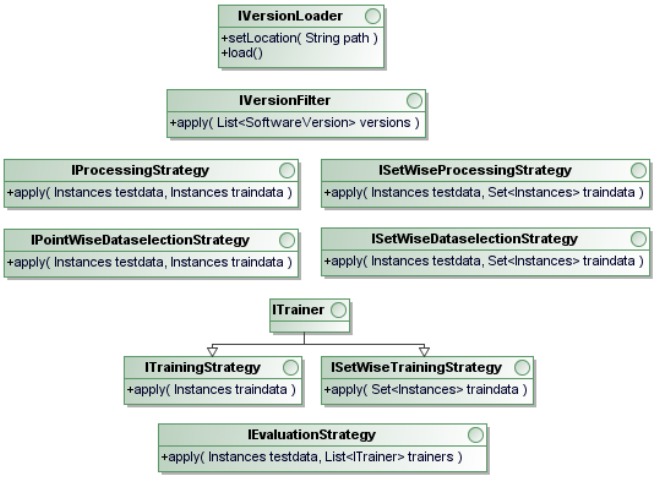

Fig. 1: Interfaces provided by CrossPare.

- `ISetWiseProcessingStrategy`: same as `IProcessingStrategy`, except that if training data from multiple projects is available, they are each passed separatly in a set which facilitates different treatment of the data in each project.
- `IPointWiseDataselectionStrategy`: creates a subset of the training data. The test data is usually used as the foundation for this, e.g., in the k-nearest neighbor approach by Turhan et al. [2].
- `ISetWiseDataselectionStrategy`: reduces the training data by deleting complete projects from the training data. The test data is usually used as the foundation for this, e.g., in the strategy based on the distribution of the test and training data by Herbold [7].
- `ITrainer`: parent interface for all training algorithms for the actual defect prediction. Is not implemented directly, instead the child interfaces `ITrainingStrategy` and `ISetWiseTrainingStrategy` are implemented.
- `ITrainingStrategy`: interfaces the training of all defect prediction models, where the training data is taken as a single set. This is the usual case.
- `ISetWiseTrainingStrategy`: interface the training of ensemble models for defect prediction, where a separate prediction model is trained for each project from which training data is available. An example for this is the bagging strategy proposed by He et al. [8].
- `IEvaluationStrategy`: takes the test data and a list of defect prediction models that was previously trained and evaluates the performance of the models on the test data.

C. Benchmark Workflows

The third key part of CrossPare is the definition of workflows for the benchmarking. The main part of a workflow

Listing 1: Benchmark workflow for cross-project defect predictions in CrossPare

```
public void run() {
  final List<SoftwareVersion> versions = new LinkedList<>();
  for(IVersionLoader loader : config.getLoaders()) {
    versions.addAll(loader.load());
  }
  for( IVersionFilter filter : config.getVersionFilters() ) {
    filter.apply(versions);
  }
  for( SoftwareVersion testVersion : versions ) {
    // Setup testdata and training data
    Instances testdata = testVersion.getInstances();
    Set<Instances> traindataSet = setCandidateTrainingData();

    for( ISetWiseProcessingStrategy processor : config.getSetWisePreprocessors() ) {
      processor.apply(testdata, traindataSet);
    }
    for( ISetWiseDataselectionStrategy dataselector : config.getSetWiseSelectors() ) {
      dataselector.apply(testdata, traindataSet);
    }
    for( ISetWiseProcessingStrategy processor : config.getSetWisePostprocessors() ) {
      processor.apply(testdata, traindataSet);
    }
    for( ISetWiseTrainingStrategy setwiseTrainer : config.getSetWiseTrainers() ) {
      setwiseTrainer.apply(traindataSet);
    }
    Instances traindata = makeSingleTrainingSet(traindataSet);
    for( IProcessesingStrategy processor : config.getPreProcessors() ) {
      processor.apply(testdata, traindata);
    }
    for( IPointWiseDataselectionStrategy dataselector : config.getPointWiseSelectors() ) {
      traindata = dataselector.apply(testdata, traindata);
    }
    for( IProcessesingStrategy processor : config.getPostProcessors() ) {
      processor.apply(testdata, traindata);
    }
    for( ITrainingStrategy trainer : config.getTrainers() ) {
      trainer.apply(traindata);
    }
    for( IEvaluationStrategy evaluator : config.getEvaluators() ) {
      List<ITrainer> allTrainers = new LinkedList<>();
      for( ISetWiseTrainingStrategy setwiseTrainer : config.getSetWiseTrainers() ) {
        allTrainers.add(setwiseTrainer);
      }
      for( ITrainingStrategy trainer : config.getTrainers() ) {
        allTrainers.add(trainer);
      }
      evaluator.apply(testdata, traindata, allTrainers, writeHeader);
    }
  }
}
```

definition is the selection of CrossPare functions to be executed. The (simplified) Java code for the workflow execution is shown in Listing 1. First, the data is loaded by calling instances of `IVersionLoader`, then the data is filtered with `IVersionFilter` instances. Afterwards, the cross-project prediction for each data set that was loaded is performed. The other data serves as candidate training data.[1] Subsequently, we apply training-set-wise pre-processors that implement the `ISetWiseProcessingStrategy` interface, then apply all training-set-wise data selectors that implement the `ISetWiseDataselectionStrategy` interface, and, finally, we apply all training-set-wise post-processors that implement the `ISetWiseProcessingStrategy`. The seperation in pre- and post-processors allows us to manipulate the data before and/or after we perform data selection. Once all set-wise data manipulations are performed, we call set-wise training algorithms, i.e., essemble learners like bagging [8]. Then, we join the data of all training candidate projects into a single data set. Analogous to the set-wise approach, we then perform pre-processing, data selection, and post-processing, but this time only for the single set. Then, we train the defetect prediction models. Finally, we call the evaluators in order to determine the benchmark results.

As the source code of the benchmark algorithm shows, CrossPare iterates over lists of instances of each interface. These lists can be empty. In that case, no instance of the related interface is called during a benchmark. To create the lists of data loaders, processors, selectors, trainers, etc., we use an XML configuration file. The concrete CrossPare functions are selected by the name of the implementing class. To instantiate the classes, we use the Java's Reflection mechanism to dynamically create the instances at runtime. Within the XML file, the element names represent the type of interface that is implemented by a class. The concrete structure of the configuration files is best explained using an example.

Listing 2 shows a CrossPare workflow configuration file. In line three of the example, we define how the data is loaded. `CSVFolderLoader` implements the interface `IVersionLoader` and can handle data from the data set by Jureczko and Madeyski [10]. The data is located in the relative subfolder `experiment/data`. In line four, we define a version filter called `MinClassNumberFilter` that implements the interface `IVersionFilter` with a parameter value of 5. The parameter construct can be used with all of the above interfaces and it allows the passing of a string to the implementing classes. How the parameters are handled depends solely on the class. In this case, the value of 5 means that there must be at least five defect prone and five non-defect-prone instances in a project, in order to be part of the experiments. In line five of the example, the data is normalized per project, i.e., the values are transformed to the interval [0,1] by calling the `Normalization` class which implements the `ISetWiseProcessor`. Then, in line six the k-nearest neighbor stragety from Turhan et al. [2] is selected for data selection. In line seven, we define that undersampling should be applied as a postprocessor after the data selection. In lines eight to eleven, we define that two defect prediction models shall be trained. The parameter passed to the `WekaTrainer` is the same as a command line call of WEKA itself. We, thereby, support to train any model that is supported by WEKA. Within a workflow, an arbitrary number of trainers can be selected and trained. The trainers are independet of each other. The usage of multiple trainers at once saves runtime in case of expensive processing of the data and, moreover, facilitates comparisons between different machines learning algorithms. Finally, in lines twelve and thirteen, we define how the model is evaluated. The location of the results is defined in line twelve. In line thirteen, we define that a normal evaluator for models trained with Weka is used. The performance metrics that are evaluated here are listed in Section III.

All parts of the above configurations are interchangable and can be combined freely. The only restriction is that some techniques themselves might not be compatible with each other. For example, two data weighting techniques cannot be used together, because the second technique will overwrite the weigths of the first technique.

III. Supported Features

With CrossPare, we already support a wide array of techniques, data sets, and performance measures.

A. Cross-project Techniques

Most of the techniques for CrossProject defect prediction proposed in the literature are implemented within CrossPare. Only techniques based on the project context are missing, e.g., Zimmermanns et al.'s [1] decision tree technique or the approach for universal predictors by Zhang et al. [11]. Concretely, we implemented the approaches proposed in the following papers.

- The k-nearest neighbor data selection by Turhan et al. [2].
- The data selection, attribute selection, and bagging approach proposed by He et al. [8].
- The project-wise data selection approaches proposed by Herbold [7], including the data weighting scheme to handle prediction bias.
- The gravity-based approach for data weighting by Ma et al. [5].
- The local predictors proposed by Menzies et al. [12]. We only support the detection of local areas with WHERE clustering with CrossPare. The WHICH algorithm for the creation of rules is not supported.
- The standardization methods proposed by Watanabe et al. [3]
- The power transformation and median-based standardization proposed by Camargo Cruz and Ochimizu [6].
- The normalization techniques based on min/max and Z-score standardization proposed by Nam et al. [4].
- Undersampling, oversampling, and resampling to treat bias in the training data.

[1] In the complete source code of the workflow, a filter is implemented such that only projects with a different name than the current target project are selected for the training data. This is to ensure that we are actually in a cross-project setting and to exclude all data from the same context.

Listing 2: Example of a CrossPare configuration file.

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
  <loader name="CSVFolderLoader" datalocation="experiment/data"/>
  <versionfilter name="MinClassNumberFilter" param="5" />
  <setwisepreprocessor name="Normalization" param=""/>
  <pointwiseselector name="TurhanFilter" param="10" />
  <postprocessor name="Undersampling" param="" />
  <trainer name="WekaTraining"
           param="RandomForest weka.classifiers.trees.RandomForest -CVPARAM I 5 25 5"/>
  <trainer name="WekaTraining"
           param="C4.5-DTree weka.classifiers.trees.J48 -CVPARAM C 0.1 0.3 5"/>
  <resultspath path="experiment/results"/>
  <eval name="NormalWekaEvaluation" param="" />
</config>
```

B. Data Sets

Currently, we can import data from the following data sets into CrossPare.

- The Java defect prediction data donated by Jureczko and Madeyski [10], hosted in the tera-PROMISE repository [13].
- The preprocessed version of the NASA MDP data set provided by [14] also hosted in the tera-PROMISE repository.
- The automotive data set collected at Audi [15].
- The Eclipse metric set donated by D'Ambros et al. [16].
- The defect prediction data set based on the data mined by Mockus [17] prepared for defect prediction by Zhang et al. [11].
- Defect data mined with the DECENT model-based software mining approach [18].

Note, that we currently cannot recommend to use data from multiple sets within the same benchmark. This is due to the difference in the metric sets used.

C. Performance Measures

CrossPare supports a wide array of performance measures that are already used in the defect prediction literature: error, precision, recall, F-measure, G-measure, true positive rate, true negative rate, AUC, AUCEC according to [19]. Additionally, CrossPare reports the confusion matrix, i.e., true positives, true negatives, false positives, and false negatives.

D. Additional Features

Another notable feature of CrossPare is that the benchmarking workflow itself can be changed, i.e., other workflows then the one presented in Listing 1 are possible. The benchmarking workflow itself is from CrossPare's perspective also hidden behind an interface, the `IExecutionStrategy` interface. By default, the workflow we described above is choosen. However, through an additional class that implements the `IExecutionStrategy` and an appropriate entry to the configuartion file, other workflows are also possible.

Moreover, CrossPare also supports within-project prediction with 10x10 cross validation. To this aim, CrossPare provides a separate evaluator, the `CVWekaEvaluation`. We evaluate the same performance metrics as for the cross-project defect predictions.

Furthermore, we provide filtering mechanisms for projects that are allowed within a case study. This way, a complete data set can be loaded, but only a subset that meets certain criteria is included in the actual benchmark. We currently support filtering based on the number of instances available for a project, the number of instances per class (i.e., defect-prone and non-defect-prone), and the bias of the data. With the latter, we allow, e.g., the exclusion of projects, where less than 5% of the instances are defect prone.

Additionally, CrossPare provides some rudimentary support for the parallel execution of tasks. Each experiment configuration is executed in its own thread. This way, multiple benchmarks can be performed at the same time to fully utilize the computational power of a machine.

IV. Conclusion

Within this paper, we present our tool CrossPare for the benchmarking of cross-project defect predictions. CrossPare provides a flexible and extensible framework within which cross-project defect prediction approaches can be implemented and compared to each other. A standard benchmarking workflow is already implemented and can be used by the definition of XML configuration files with the information which techniques shall be executed and evaluated. Furthermore, CrossPare supports a large array of machine learning algorithms for the creation of the defect prediction models by using WEKA as machine learning core. Moreover, many different data sets can already be imported into CrossPare.

Due to its nature as open source software, other researchers can extend CrossPare on their own or contact us to give them access to the main development trunk of CrossPare to integrate their contributions.

In the future, we plan to work on the execution backend of CrossPare in order to allow massive parrallelization of

experiments in cloud infrastructures. We already tested a simple Map/Reduce approach to distribute tasks to the cloud to scale the execution of experiments [20]. We plan to build on this foundation in order to allow large-scale benchmarks in a short amount of time. Moreover, we are always interested in adding new techniques for cross-project defect prediction, as well as support for more data sets to CrossPare.

REFERENCES

[1] 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. the 7th Joint Meeting European Softw. Eng. Conf. (ESEC) and the ACM SIGSOFT Symp. on the Foundations of Softw. Eng. (FSE)*, 2009.

[2] B. Turhan, T. Menzies, A. Bener, and J. Di Stefano, "On the relative value of cross-company and within-company data for defect prediction," *Empirical Softw. Eng.*, vol. 14, pp. 540–578, 2009.

[3] S. Watanabe, H. Kaiya, and K. Kaijiri, "Adapting a fault prediction model to allow inter language reuse," in *Proc. 4th Int. Workshop on Predictor Models in Softw. Eng. (PROMISE)*. ACM, 2008.

[4] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proc. 35th Int. Conf. on Softw. Eng. (ICSE)*, 2013.

[5] Y. Ma, G. Luo, X. Zeng, and A. Chen, "Transfer learning for cross-company software defect prediction," *Inf. Softw. Technology*, vol. 54, no. 3, pp. 248 – 256, 2012.

[6] A. E. Camargo Cruz and K. Ochimizu, "Towards logistic regression models for predicting fault-prone code across software projects," in *Proc. 3rd Int. Symp. on Empirical Softw. Eng. and Measurement (ESEM)*. IEEE Computer Society, 2009.

[7] S. Herbold, "Training data selection for cross-project defect prediction," in *Proc. 9th Int. Conf. on Predictive Models in Softw. Eng. (PROMISE)*. ACM, 2013.

[8] Z. He, F. Peters, T. Menzies, and Y. Yang, "Learning from open-source projects: An empirical study on defect prediction," in *Proc. 7th Int. Symp. on Empirical Softw. Eng. and Measurement (ESEM)*, 2013.

[9] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: an update," *ACM SIGKDD Explorations Newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[10] M. Jureczko and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction," in *Proc. 6th Int. Conf. on Predictive Models in Softw. Eng. (PROMISE)*. ACM, 2010.

[11] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou, "Towards building a universal defect prediction model," in *Proc. 11th Working Conf. on Mining Softw. Repositories (MSR)*, 2014.

[12] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, and T. Zimmermann, "Local versus Global Lessons for Defect Prediction and Effort Estimation," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 822–834, June 2013.

[13] T. Menzies, C. Pape, and C. Steele, "tera-promise," http://openscience.us/repo/, 2014.

[14] M. Shepperd, Q. Song, Z. Sun, and C. Mair, "Data Quality: Some Comments on the NASA Software Defect Datasets," *IEEE Transactions on Software Engineering*, vol. 39, no. 9, pp. 1208–1215, 2013.

[15] H. Altinger, S. Siegl, Y. Dajsuren, and F. Wotawa, "A novel industry grade dataset for fault prediction based on model-driven developed automotive embedded software," in *Proc. 12th Working Conf. on Mining Softw. Repositories (MSR)*. Florence, Italy: IEEE, 2015.

[16] M. D'Ambros, M. Lanza, and R. Robbes, "An Extensive Comparison of Bug Prediction Approaches," in *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE Computer Society, 2010.

[17] A. Mockus, "Amassing and indexing a large sample of version control systems: Towards the census of public source code history," in *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, ser. MSR '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 11–20. [Online]. Available: http://dx.doi.org/10.1109/MSR.2009.5069476

[18] P. Makedonski, F. Sudau, and J. Grabowski, "Towards a model-based software mining infrastructure," *SIGSOFT Softw. Eng. Notes*, vol. 40, no. 1, pp. 1–8, Feb. 2015. [Online]. Available: http://doi.acm.org/10.1145/2693208.2693224

[19] F. Rahman, D. Posnett, and P. Devanbu, "Recalling the "imprecision" of cross-project defect prediction," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2012.

[20] M. Göttsche, F. Glaser, S. Herbold, and J. Grabowski, "Automated Deployment and Parallel Execution of Legacy Applications in Cloud Environments," in *The 8th IEEE International Conference on Service Oriented Computing & Applications (SOCA)*, 2015.