A Comparison of Transfer Learning Algorithms for Defect and Vulnerability Detection

Ashton Webster
University Of Maryland, College Park
ashton.webster@gmail.com

ABSTRACT

Machine learning techniques for defect and vulnerability detection have the potential to quickly direct developers' attention to software components with faulty implementations. Effective application of such defect prediction methods in practical software development environments requires transfer learning algorithms so that models built using existing projects can recognize defects as they emerge in a new project. Up until this study, comparing the efficacy of transfer learning algorithms was challenging because previous studies used differing data sets, baselines and performance metrics. By providing open source implementations and baseline performance metrics for several transfer learning algorithms on two different data sets, our project offers software engineers the tools to objectively compare methods and readily identify top performing transfer learning algorithms in the domain of both vulnerability and defect prediction.

1. INTRODUCTION

Detecting vulnerabilities and defects in software programs is as important as it is expensive. The IBM System Science Institute found that the relative cost of fixing a bug during testing was 15 times higher than during design, while fixing a bug during maintenance costs 100 times more than fixing it during project design [3]. Similarly, it is clear that the longer a vulnerability remains undetected, the more likely it is to be exploited by an attacker. The most common method of preventing defects and vulnerabilities is currently testing and code reviews. While these methods provide many benefits, they rarely detect all code issues, and are usually time and resource expensive. Therefore, many researchers have considered the possibility of using software metrics or language specific tokens along with machine learning techniques to identify specific code components likely to contain bugs. This technique can be used in conjunction with code reviews, so that the models created "suggest" files for the developers to consider during review. The benefit is often considerable, as more of the time taken by reviewers can be spent on code that is actually defective, instead of searching randomly without guidance. For example, Stuckman has found that nearly 80% of vulnerabilities in a code base can be identified in this manner while only inspecting 20% of the total files [21].

However, this approach still has its drawbacks. The main challenge is that in order to use the machine learning approach, one needs to have access to a (preferably large) set of programs with labeled vulnerabilities and defects. While labeled datasets exist, they may be from projects in completely different domains than the project being developed. It is not clear whether this "cross-project" data will be relevant for labeling the target project. This is where a technique called *transfer learning* can be used: by taking a set of potentially unrelated, labeled "source" domains and creating a model which can predict on a new, unlabeled "target" domain. This method is more general than identifying defects or vulnerabilities across projects; in general, transfer learning algorithms are domain agnostic.

This technique has significant value, as being able to use a previously curated dataset from unrelated projects, efficiently find the relevant data, and predict the vulnerabilities or defects on a new project has time and effort saving value for developers. Unfortunately, it has not been clear *which* transfer learning algorithms are best suited for this task. Additionally, many transfer learning algorithms are explored and compared in various papers, but rarely have open source implementations that can be easily used in practice [9, 14, 16, 20, 22]. The goal of the current study is to rectify these previous limitations by providing open source implementations of several transfer learning algorithms and comparing their performance on two public datasets.

The rest of this paper is organized as follows: first, section 2 provides a background on various approaches to solving this problem. Next, in section 3, a brief summary of the selected transfer learning algorithms is provided (section 3.1), along with the baselines, and performance metrics (section 3.2). Then, the experiment is defined and conducted (section 4). Finally, discussion (section 5), conclusions (section 6), and future work are outlined (section 7).

2. BACKGROUND

There are three main approaches for transfer learning which rely on different fundamental assumptions about the "problem" of predicting on a target domain. If we assume that there is only a small subset of the existing source domain that is similar to the target domain, then we might use the *filtering* approach, which seeks to eliminate irrelevant source domain data from the training set. Instead, if we believe that the source domain and the target domain are not similar, but can be made more similar by various transformations to the feature space, then this leads us to the idea of *normalization*. Finally, if we assume that each instance of the target domain might benefit from prediction based on a different subset of the available training source data, then *ensemble methods* are a logical choice, because multiple classifiers are used depending on the given test instance to be classified. Additional details and examples of

these approaches are provided in the following sections.

2.1 Filtering Methods

Several popular methods use a filtering methodology, using only "relevant" training examples. Turhan proposed the "Burak" (Nearest Neighbors) algorithm [22], which finds the closest training instances (from any project) to each test instance. In [20], this method is compared to the "Peters" filter, which flips the idea and focuses on finding the closest test instance to each training instance. Although it remains rare to see studies analyzing the performance of vulnerability detection, Yamaguchi, Lindner, and Rieck demonstrate that PCA can be used to effectively find the instances most similar to vulnerable files and flag these for inspection [24]. Another idea, proposed by Fukushima, Kamei, and McIntosh, is is to filter away *projects* (instead of individual isntances) that are not similar to the test project based on correlation of predictors and dependent variables [6]. The common thread in these methods is the decision to completely ignore certain training instances or entire projects that are determined to be irrelevant to the prediction task at hand.

2.2 Normalization/Weighting Methods

Normalization methods can be used to bring test and training projects into a similarly distributed feature space. For example, Nam, Pan, and Kim use a method called TCA+ (Improved Transfer Component Analysis) to project the target and source domain data to a common feature space where classification can be performed. Similarly, Minku, Sarro, Mendes, and Ferrucci suggest a method called Dycom which is used which to scale effort estimations from one project and apply them to other projects [16]. Another technique, somewhat related to normalization, is weighting training instances based on similarity to test instances. Ma, Luo, Zeng, and Chena found this method helpful in creating priors for the Naive Bayes classifier from other source projects and using them to predict on target projects [13].

2.3 Ensemble methods

Ensemble methods are one popular idea in the field of transfer learning for defect and vulnerability detection. These methods create multiple "weak learners" from a subset of the training data or features and use different voting, averaging, or clustering methods to produce predictions. One method is the "Cluster-Then-Label" idea, which involves creating groups of data around test instances, training classifiers within each group, and then labeling the instances using the multiple created models. A simple implementation of this method using K-means clustering has already already been used in intrusion detection with high success levels [10]. For example, Menzies et al. propose a two-step process consisting of a clustering function named WHERE and a rule-learning function named WHICH, focusing on human-readable and understandable output [14]. Additionally, Kamishima, Hamasaki, and Akaho use a modified bagging algorithm called TrBagg, which creates many weak learners by repeatedly resampling the dataset using a technique known as "Bagging" [9]. Several methods are proposed on how to recombine the predictions of the individual classifiers generated by TrBagg to produce very accurate predictions in different contexts.

2.4 Limitations

Despite the considerable body of work available on various different transfer learning algorithms, several challenges remain. While many options are available, it is not clear which transfer learning algorithms are the best because they are not compared with consistent performance metrics. Additionally, many of these algorithms were created for the domain of defect prediction, and it is not clear whether they can also be useful for vulnerability detection. Unfortunately, practical usage of these algorithms is currently difficult because few of the previously mentioned papers provide an implementation of their prescribed methods. This work seeks to resolve some of these shortcomings by comparing several transfer learning algorithms on the same dataset with a variety of performance metrics. Furthermore, based on the descriptions from these papers, the algorithms are implemented and provided as open-source code. These contributions will benefit researchers and practitioners alike; researchers will benefit from a set of uniformly compared baselines while developers will benefit from easily accessible off-the-shelf algorithms and recommendations of the highest performers.

3. EXPERIMENT DESIGN

The main goals of this paper are threefold: (a) to contribute an open-source, well-documented implementation of several transfer learning algorithms proposed in other papers; (b) to provide baseline benchmarks for transfer learning algorithms on multiple datasets; and (c) to analyze the results of the benchmarking and determine the best performing algorithm. The primary question being answered is *whether the performance of any transfer learning methods significantly outperforms the baselines*, where baselines will be defined as simple, non-transfer learning methods of prediction.

To achieve these goals and answer the research question, we define the following experiment design. Each experiment consists of assigning one project as the target project and the others as source projects. This is analogous to the situation where a developer has no available training data within the project he or she is working on, but has an existing labeled dataset from other projects. Each transfer learning algorithm is applied and evaluated on this test project. For the PHP Vulnerability Dataset, both metrics and token features are evaluated separately. This is repeated once with every project as the target project. So, for example, if we have labeled projects A, B, and C, we first take A and B as the source projects and treat C as the target project. After each transfer learning algorithm and baseline is applied and evaluated, B, followed by A, are used as the target projects, with the others as source projects. Note that because the training set is exclusively from the source projects, the terms "source projects" and "training set" are equivalent in this work, but *not* in general, as the target project might have a subset of previously labeled files that is also used for training. Similarly, "target project" and "test set" are equivalent in this work, but not in general.

To perform this experiment several components are needed; namely transfer learning algorithms and baseline techniques for comparison, datasets, and performance metrics. Transfer learning algorithms were implemented manually as an initial step. Five transfer learning algo-

rithms (Burak, Peters, Cluster-Then-Label, TrBagg, and Gravity Weighting) along with two baselines techniques (Inspection Baseline and Source Baseline) were selected and implemented in Python, with the source made available at an open source repository [1]. For the datasets, one dataset (PROMISE) was selected for defect prediction, and another (PHP vulnerability dataset) was selected for vulnerabilities. Finally, performance metrics were defined for the purpose of comparing the transfer learning algorithms. The following sections describe these experimental components and how they were developed.

3.1 Algorithms Implemented

Five transfer learning algorithms are implemented and tested in this study and compared to two baselines. The selection of transfer learning algorithms was based on some intuitively beneficial aspects. Algorithms with a history of success in defect or vulnerability detection were preferred. Additionally, an attempt was made to select algorithms using disparate techniques. For example, we have filtering algorithms (Burak, Peters, and Cluster-Then-Label), ensemble methods (TrBagg), and weighting/normalization methods (Gravity Weight).

Random Forest is used for all classification tasks based on its high performance in a variety of tasks as explained in [2], and specifically for its high performance in in defect prediction [12]. All transfer learning algorithms were implemented in Python, and the scikit-learn [19] library was used for its implementation of Random Forest and several other convenience functions. A brief explanation of the details and implementation for each algorithm follows.

3.1.1 Burak

This simple filtering algorithm was first proposed by Turhan [22]. For every target test instance, the closest k instances from any other project are selected for the training data. In line with the original experiment [22] and the replication [20], we set $k = 10$ for our experiment. This method is also known as the "Nearest Neighbors" method.

Because this method requires computing the distance from every instance to every other instance, it can be computationally expensive. An approximation proposed by Menzies (and used in this study) is to first run k-means to create "batches" of instances close to one another and then the Burak Method within each batch [20]. The benefit of this method is that the number of distance calculations is reduced from N^2 to $\sum_i^K n_i^2$, where N is the size of the training set, K is the number of clusters, and n_i is the number of instances in the ith cluster. For this study, the number of batches was set at $N/100$, such that each cluster should have approximately 100 instances.

3.1.2 Peters

Building off of the Burak filter, [20] proposes the "Peters" filter and demonstrates the benefits of letting training data drive the process of training set selection. Instead of filtering based on the training instances closest to a given test instance, this method first groups by closest *test* instance to each *training* instance and then filters away all but the closest training instance for each test instance. In summary, the process is:

1) For each train instance, find closest test instance.
2) For a test instance x_i, let the set of training instances be F_{x_i} such that x_i is the closest test instance for each $f \in F_{x_i}$.
3) For each test instance x_i, find the closest training instance f_i' in F_{x_i} and include this in the training set.
4) The final set of $F' = \{f_0 ... f_n\}$ is the filtered training set.

Similar to above, the batching approximation method is used for efficiency, but this time the number of batches is $N/30$, such that the number of instances in each cluster is approximately 30.

3.1.3 Cluster-Then-Label

Another simple approach to transfer learning is to use the semi-supervised learning approach of "Cluster-Then-Label." The idea is to first cluster using an unsupervised machine learning algorithm and then, within each cluster, train a classifier within each cluster. New instances are then labeled by first identifying the closest cluster (e.g. euclidean distance to the metric vector) and classifying based on the model trained on that cluster. The proposed benefit of this method in the context of the fully supervised task of identifying software defects and vulnerabilities is that the closest training instances (potentially from several different source domains) to the given test instance will provide the greatest benefit to model performance. For our study, the K-means algorithm was used with $k = 8$ (i.e. 8 cluster centers were used). This parameter selection was arbitrary, but additional research could provide empirical evidence to guide better parameter selection.

3.1.4 TrBagg

TrBagg was first introduced by [9]. The main idea is to take advantage of a machine learning technique known as *bagging*. Training data is repeatedly sampled without replacement to train many *weak classifiers*, which can then be combined in different ways to produce a classification. The main extension on weak classification contributed by Kamishima is that the weak classifiers are only incorporated to the ensemble if there is evidence to suggest that adding the weak classifier will improve the ability of the overall ensemble to classify instances of the specific target class. Several heuristic methods were implemented, but we will use the simplest form, which is simply standard bagging from the entire available training set.

3.1.5 Gravity Weighting

Ma, Luo, Zeng, and Chena propose a "Transfer Naive Bayes" algorithm which relies on gravitational weighting [13]. That is, training instances are weighted inversely proportional to their distance from the test instances, based on measure of similarity defined in the paper.

Specifically, Ma defines an indicator function h:

$$h(a_{ij}) = \begin{cases} 1 \text{ if } min_j \leq a_{ij} \leq max_j \\ 0 \text{ else} \end{cases}$$

where min_j and max_j are the minimum and maximum value for feature j across all test instances, respectively, and a_{ij} is the value of feature j for training instance

[1] All implemented transfer learning algorithms are available with documentation at https://github.com/ashtonwebster/tl_algs

i. Intuitively, h is an indicator function which determines whether a_{ij} is within the range of values of the test instances (min_j is the bottom of this range for the feature, max_j is the top). Then, using this h function, the number of "similar" features for instance i, denoted s_i, can be computed:

$$s_i = \sum_{j=1}^{k} h(a_{ij})$$

Finally, the weighting measure is computed:

$$w_i = \frac{s_i}{k - s_i + 1}$$

Where k is the total number of features. This gives a weight on $[0, 1]$, with higher weights assigned to instances with more features in the range of values of the test features.

Although this weighting is proposed for creating a prior distribution for the Naive Bayes classifier, in this study, it is used in as an instance weight parameter for Random Forest.

3.1.6 Baselines

There are two baselines which will be used for comparison:

- "Inspection Baseline": If we assume a uniform distribution of vulnerabilities throughout the code base, it is expected that looking at $X\%$ of the code will reveal approximately $X\%$ of the vulnerabilities. This can be thought of as the random approach, where the code reviewer simply inspects files at random. No machine learning is used, making this the most naïve option.
- "Source Baseline": All available source project data is used, with no additional filtering, weighting, or other special techniques.

Returning to our example, let's suppose we have two source projects A, B, and a single target project C. For the Inspection Baseline, we ignore A and B completely and just consider C. We then take in the *inspection ratio* as a parameter. This parameter represents the maximum proportion of the dataset to be inspected. Based on this parameter, the inspection ratio simply randomly samples the given proportion of C and flags these files as vulnerable or defective. In expectation, this should correctly identify a proportional number of vulnerabilities or defects. Obviously this is a very naive baseline, but it represents the performance floor of random guessing. For the Source Baseline, we build a classifier using just the data from A and B, with no other special transfer learning methods, and simply predict on C. This baseline is far less trivial to outperform, as demonstrated by the results of this study.

3.2 Datasets

Two datasets were used for this study. First, we examine the dataset publicly available from the PROMISE repository [8], which contains 34 projects. This dataset included 20 different software engineering metrics per file (such as Lack of Cohesion and Average Cyclomatic Complexity) in addition to the number of defects in the file. The number of defects in the file was changed to be a binary field such that the value was zero if no defects were present and one otherwise. In all, this dataset contains 87,399 files of which 14,623 contained at least one defect. The purpose of the PROMISE dataset was to evaluate the transfer learning algorithms on a well known defect dataset used in many previous studies, including [1, 7, 11, 13, 20].

Project	Total Files	Vulnerable Files	Percent Vulnerable
Drupal	200	62	31.0%
Moodle	2942	24	0.82%
phpMyAdmin	321	27	8.41%

Table 1: Comparison of number of vulnerabilities in PHP projects

The second dataset used is a publicly available dataset created by Walden et al. from three web-based PHP projects [23]. This dataset identifies the number of vulnerabilities per file based on security announcements from the National Vulnerability Database [18] and from project specific pages. Three projects were considered: Moodle, phpMyAdmin, and Drupal, all of which are open source and web based. Both software metrics (though not all the same as the PROMISE dataset) and tokens (vectors indicating which PHP language tokens were present) were available for each file. In total, there were 3,465 files with 113 vulnerabilities. Note that the distribution of vulnerabilities across the projects was far from uniform. In particular, Moodle had the lowest frequency of vulnerabilities, while Drupal had the highest. See Table 1 for more details.

3.3 Performance Metrics

With the datasets and the methods defined, it is important to consider how we will go about comparing the performance of the methods on the datasets. There are many available metrics for defect prediction which may extend to vulnerability detection. Accuracy, although simple, is not frequently used due to class skew (i.e. the number of vulnerable or defective instances is far outnumbered by the number of non-vulnerable instances). For example, simply labeling all Moodle instances as benign (no vulnerabilities) will result in an accuracy of 99.16% percent, because only 0.84% of the total files are vulnerable. Clearly, this is not a good estimate of the performance of the classifier.

Recall is on of the most popular performance metric selections for defect and vulnerability detection; that is, the proportion correctly identified positive instances (i.e. defects or vulnerabilities) detected out of all positive instances. The complimentary measure, *precision*, is often used to assess the proportion of identified positive instances that are actually positive instances. F-measure (also known as F1 Score or F Score) is also frequently used, which incorporates the geometric mean of precision and sensitivity. However, Menzies suggests that when the number of positive instances is relatively small compared to total instances, the variability of precision measures results in instabilities in the measure, and can often make the precision (and in turn, F-measure) performance metrics less useful [15]. In [20], the F-measure is therefore replaced with the G-measure, which removes the dependence on precision by instead incorporating the false positive rate.

For this study, we use a measure proposed by Stuckman in [21], who suggests the measure Area Under Cost Efficiency Curve (AUCEC). The curve is given as follows:

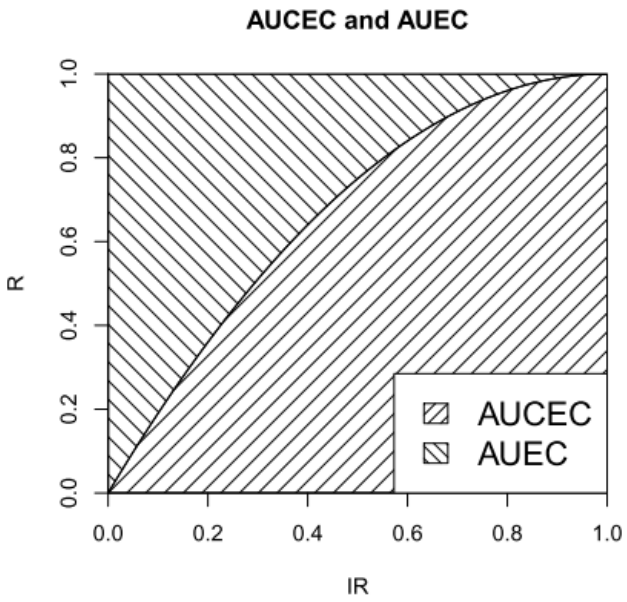

Figure 1: Diagram illustrating the AUCEC measure. Replicated from Figure 4.1 in [21].

Where recall is given as:

$$R = \frac{TP}{TP + FN}$$

"Inspection Ratio" is given as:

$$IR = \frac{TP + FP}{TP + FP + FN + TN}$$

Where TP is True Positives, FN is False Negatives, FP is False Positives, and TN is True Negatives. Intuitively, IR represents the proportion of total files in the project that would have to be inspected in order to find the positive instances identified by the model.

Often, the AUCEC50 measure is used instead of AUCEC, which measures only the area under the curve up to 50% inspection ratio. This captures the most "useful" part of the classification, because most companies are not interested in a method that flags more than 50% of the codebase for inspection in order to find vulnerabilities. As in [21], we follow this precedent.

AUCEC50 primarily focuses on the number of vulnerabilities found per file inspected, which is the goal we are trying to optimize. It is also resistant to class skew, which is present in this field especially (vulnerable files tend to be considerably outnumbered by non-vulnerable files, with a similar pattern exhibited for defects). Furthermore, AUCEC50 incorporates the fact that most classifiers produce an intermediate, continuous confidence level which is converted into a binary prediction. Instead of only observing the prediction itself, AUCEC50 implicitly varies the prediction confidence threshold from 0 to 1 to obtain inspection ratio's between 1 and 0, respectively. Specifically, if the confidence threshold is 0, then everything will be classified as vulnerable, and everything will be inspected. Meanwhile, if the threshold is set at 1, then only the subset of files with the highest likelihood of vulnerability will be inspected (which should be very small). Practitioners can pick a confidence value that maximizes recall and minimizes the number of files inspected in this manner. Based on these benefits, AUCEC50 is used throughout the rest of this paper as the primary metric for comparison of algorithms. However, in order to allow for comparison with results of other, more frequently used metrics, we include several other supplemental metrics, including F-measure, G-measure, Recall, and Inspection Ratio.

4. RESULTS

In order to determine if the difference between transfer learning algorithm performance is greater than what would be expected due to chance, we use the Friedman test [5]. The Friedman test uses the null hypothesis that the observed difference between the classification methods is simply due to randomness, similar to the ANOVA test. However, ANOVA relies on assumptions of normality, homogeneity, and sphericity of variance, which do not hold in practice in this domain [4]. The Friedman test avoids the normality assumption by using relative rankings. The Friedman test is proposed as a good test for comparison of multiple machine learning methods on a fixed dataset, and is the primary test used in a similar study performed by Lessman et al. [12]. If the Friedman test reveals that the difference between performance of the classifiers is not due to random chance, then the post-hoc Nemenyi test can be used to identify the subset of classifiers which significantly outperform the others [17].

One chief concern when using machine learning algorithms (and similarly for transfer learning algorithms), is to avoid *overfitting*, where hyperparameters are tuned with respect to the test set to give artificially high performance at the expense of not generalizing well to new datasets. To allay this concern, a validation dataset was used which consisted of a 20% random sample of the Drupal dataset in order to tune several hyperparameters for the Random Forest classifier and the transfer learning algorithms. Tests on this validation set (which was *not* used in any of the actual experiment test or training sets) revealed that setting the number of trees to 100 (from the default of 20) lead to significant improvement in the Random Forest classification accuracy.

Interestingly, validation set experimentation with the TrBagg algorithm revealed that using the simplest method (no filtering of weak learners) produced the best results for the Drupal dataset. Often, using the filtering approaches when bagging resulted in training sets that were too small to be useful. Therefore, the simplest method explained above was used. Further research is needed to determine which TrBagg heuristic method performs the best over a larger number of contexts, but from now on in the paper, the reader should assume "TrBagg" refers to the simplistic bootstrapping technique with no additional voting.

For the actual experiments, we show three different figures which show the main comparisons of different methods:

- A chart (of the type first proposed by [4]) showing the pairwise comparisons of the Nemenyi post-hoc test for average rank (used to demonstrate which classifiers are significantly outperforming the others).

- An example of a single projects AUCEC50 curve, showing the trade off of IR (inspection ratio) and R (recall).
- A table of the top ten classifiers ranked by descending AUCEC50 values, with popular performance metrics such as F-measure, G-measure, recall, precision, accuracy, and several others.

We begin with a comparison of transfer learning methods in the context of defect prediction using the public PROMISE dataset. We then move on to a comparison of transfer learning methods for vulnerability detection on the PHP Vulnerability Dataset. For the PHP Vulnerability Dataset, we consider the two different input formats for each experiment (metrics and tokens) separately.

4.1 PROMISE

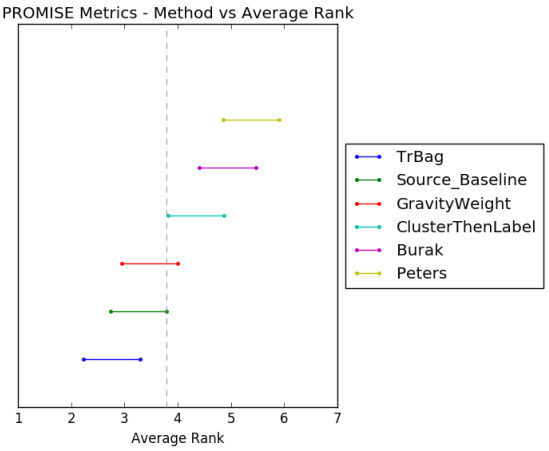

Figure 2: The average rank graph (illustrating the post-hoc Nemenyi test). Each line corresponds to a measure of the average rank for the algorithm. The leftmost point on each line is the actual average rank across all projects; the line itself represents the *critical difference*, which is the minimum amount by which other projects' average ranks must differ to be considered significantly different). The dotted grey is added to illustrates that projects to the left (TrBagg and Source Baseline) are significantly outperforming the other algorithms because the others have ranks higher (i.e. worse) than the critical difference.

First, we consider the table of top models by descending AUCEC50 score in Table 2. Notice that Source Baseline (the method of training on all available source projects with no transfer learning) and TrBagg (the method using resampling to create an ensemble of classifiers) make up most of the top 10, and tend to have very similar performance within the same projects. The top 10 classifiers serve as a baseline for several performance measures on this dataset. To better understand how the AUCEC50 values in this table are computed, the AUCEC curve for the Tomcat project is demonstrated in Figure 3. Each curve in this graph represents a transfer learning algorithm or a baseline. Each point on the curves gives information about the tradeoff of proportion

Performance Metric	Peters - Burak Difference
G Measure	-0.021
AUCEC50	-0.020
F-measure	-0.028
FP Rate	-0.024
Recall	-0.038
Precision	-0.016
Accuracy	0.000
Rank	1.000

Table 3: Median difference of several performance metrics for Burak and Peters methods. Positive values indicate that the Peters algorithm outperformed the Burak algorithm, while negative values indicate the opposite.

of code inspected and proportion of defects detected. For example, when 20% of the code base is "recommended" for review (i.e. classified as defective, seen as .2 on the inspection ratio axis), the Source Baseline correctly identifies approximately 60% of the total vulnerabilities. For this project, all methods appear to be outperforming the "inspection baseline", indicated by the straight diagonal line from the origin (which comes from the assumption that random inspection of N% of the files should detect approximately N% of the defects). The AUCEC50 score (representing the area under this curve from 0 to 50% inspection) for the Tomcat project for the source baseline is 0.566, which is more than two times higher than the AUCEC50 score for the inspection ratio, which is always .250.

By applying the Friedman test, we conclude that the differences in transfer learning methods for the PROMISE dataset are not solely due to chance because the p-value is less than .0001. The average rank graph in Figure 2, demonstrating the Nemenyi post-hoc test, shows that TrBagg and the Source Baseline methods outperform the Cluster-Then-Label, Burak, and Peters algorithms (but not the Gravity-Weight algorithm).

As a short aside, we note that these results seem to be different from those found in [20], which assert that the Burak filter outperforms the Peters filter by a median of 40% using the G-measure on the PROMISE dataset using metrics. However, Table 3 reveals a -2% median difference between Peters and Burak in AUCEC50 (that is, Peters has an AUCEC50 .02 *lower* than the Burak filter). The Peters Filter also outperforms the Burak filter in other measures of success; for example, the median F-measureure for the Peters filter is 2% lower and the AUCEC50 score is about 3 percent lower than the respective scores for the Burak filter. Furthermore, the Burak *and* Peters filter are both significantly outperformed by the Source Baseline method (which simply uses all available training data without filtering at all). While [20] did use the same dataset (PROMISE), one major difference in the implementation of the experiment was that projects with more than 100 instances were used for training and projects with less than 100 instances were used for testing. This experiment does not make that distinction. Additionally, different batching values were used (Peters used batches of size approximately 10, while we use larger batches, which should give more accurate distance fil-

Function Name	Test Proj. Name	IR	Recall	Prec.	Accuracy	F-measure	AUCEC50	G
TrBagg	E-Learning	0.188	0.600	0.250	0.828	0.353	0.637	0.386
Source Baseline	E-Learning	0.188	0.600	0.250	0.828	0.353	0.600	0.386
TrBagg	Berek	0.186	0.438	0.875	0.767	0.583	0.576	0.917
TrBagg	Tomcat	0.199	0.610	0.275	0.821	0.379	0.572	0.414
Gravity Weight	Tomcat	0.199	0.610	0.275	0.821	0.379	0.570	0.414
Source Baseline	Tomcat	0.199	0.558	0.251	0.811	0.347	0.566	0.387
TrBagg	Serapion	0.200	0.556	0.556	0.822	0.556	0.563	0.684
Burak	Serapion	0.200	0.444	0.444	0.778	0.444	0.561	0.586
Gravity Weight	InterCafe	0.185	0.500	0.400	0.815	0.444	0.556	0.548
Burak	InterCafe	0.185	0.500	0.400	0.815	0.444	0.556	0.548

Table 2: Top 10 Classifiers by Descending AUCEC50 score with selected performance metrics for the PROMISE dataset

tering). Further research is required to determine if other variables intervened to result in inconsistent results between our studies.

4.2 PHP Vulnerabilities

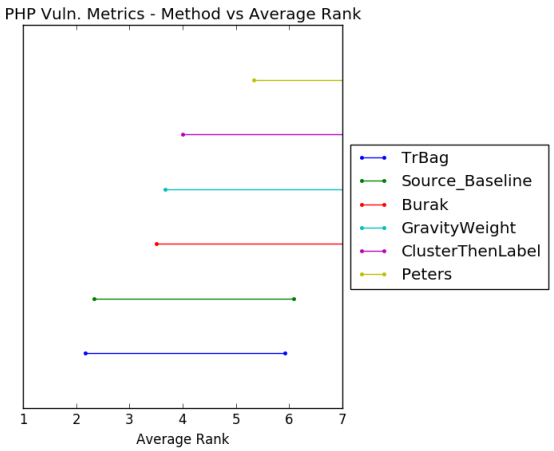

Figure 5: PHP Metrics - average rank

For the PHP Vulnerability Dataset, the top 10 classifiers by AUCEC50 are again provided in table 4. The top 10 classifiers were almost entirely created on the Moodle dataset, which might be explained by the fact that Moodle had the smallest proportion of vulnerable files at 0.84%. An example AUCEC50 curve is provided as an example for the Moodle project in Figure 4.

While there was a significant difference between transfer learning methods in the PROMISE dataset, the Friedman test determines that the methods applied to the PHP Vulnerability Dataset did not have a statistically significant difference in performance for either metrics or tokens (the p values were 0.1746 and 0.3992 for metrics and tokens, respectively). This is confirmed in Figures 5 and 6 (Nemenyi post-hoc tests), where we see that all of the horizontal lines are overlapping and therefore none of the average ranks among the projects is greater than the critical difference necessary for statistical significance. Note that the Nemenyi post-hoc test is not actually appropriate here because the Friedman test did not return a positive result, but the diagram is provided to aid the reader in understanding the negative result

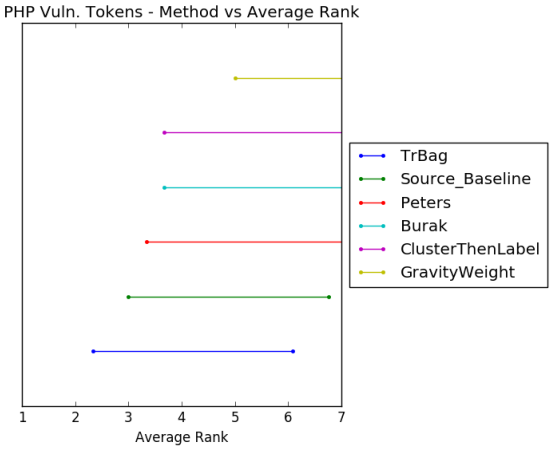

Figure 6: PHP Tokens - Nemenyi post-hoc test. In all cases, we see that none of the average ranks for the methods is significantly better than the others.

of the Friedman test.

5. DISCUSSION

We can now return to the question posed in the introduction: *Does the performance of any transfer learning method significantly outperform the baslines?*. With few exceptions, the Source Baseline method (simply using all of the data available from other projects with no filtering) appears to perform at least as well as other methods. In most cases, TrBagg and Cluster-Then-Label perform at a similar level to the Source Baseline as well. This appears to be the case for both defect detection and vulnerability detection. Based on these facts and the results of the Friedman and Nemenyi tests, we can conclude that none of the implemented transfer learning methods significantly outperform the Source Baseline for either vulnerability (as seen in the PHP Vulnerability Dataset) or defect prediction (as seen in the PROMISE dataset).

At first glance, this is an unsatisfying conclusion, but two caveats are worth mentioning. First, there may exist some transfer learning algorithm that was not tested on this dataset that outperforms this baseline, and this study does not prove that it is impossible to outperform the Source

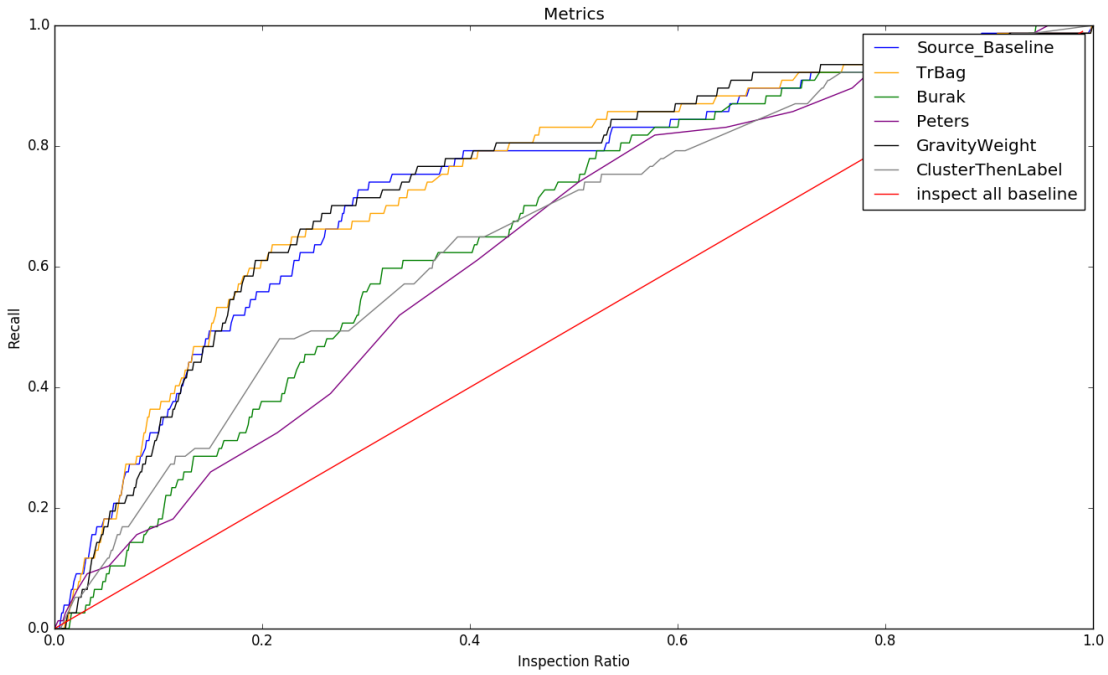

Figure 3: An example of the AUCEC50 curve for the Tomcat project in the PROMISE dataset.

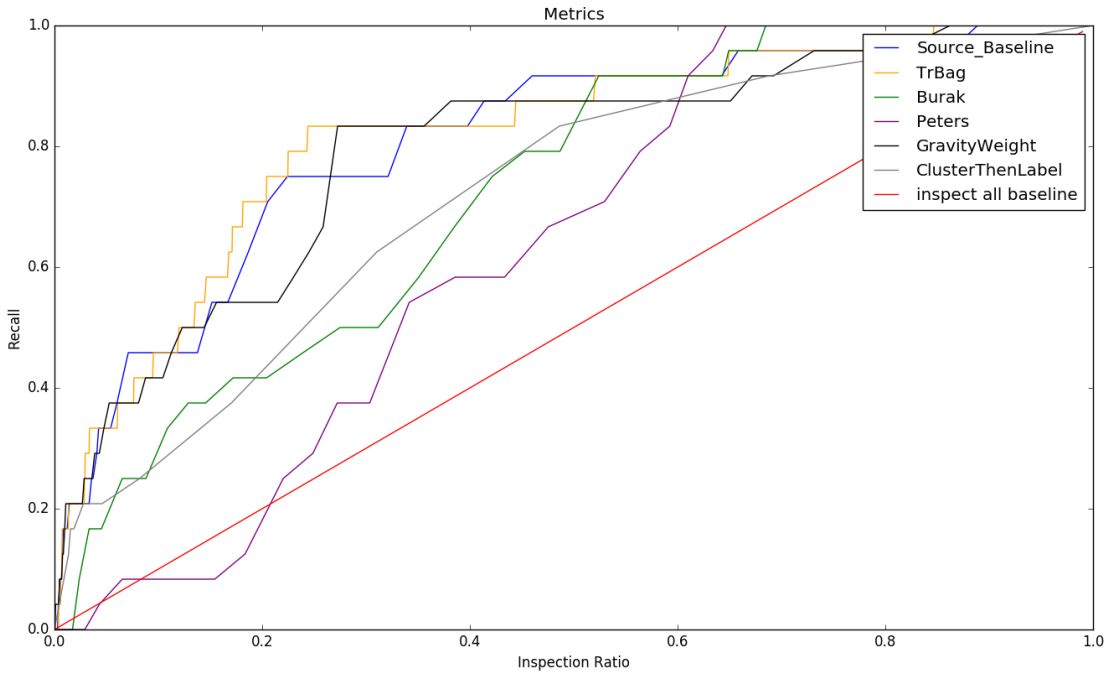

Figure 4: Here we have an example AUCEC50 curve, which compares the proportion of code inspected and the proportion of total vulnerabilities found. The methods are closely clustered but all outperform the "inspect all" baseline, which represents code reviews of X% of the code revealing an expected X% of the vulnerabilities.

Format Type	Function Name	Test Name	Proj.	IR	Recall	Prec.	Accuracy	F-measure	AUCEC50	G
Metrics	TrBagg	Moodle		0.201	0.708	0.029	0.803	0.055	0.675	0.056
Metrics	Source Baseline	Moodle		0.187	0.625	0.027	0.815	0.052	0.658	0.053
metrics	GravityWeight	Moodle		0.199	0.542	0.022	0.801	0.043	0.642	0.043
Tokens	Burak	Moodle		0.192	0.583	0.025	0.809	0.048	0.602	0.048
Tokens	TrBagg	Moodle		0.200	0.625	0.026	0.802	0.049	0.590	0.049
Tokens	GravityWeight	Moodle		0.199	0.583	0.024	0.803	0.046	0.566	0.047
Tokens	Source Baseline	Moodle		0.190	0.583	0.025	0.811	0.048	0.561	0.049
Metrics	Cluster-Then-Label	Moodle		0.170	0.375	0.018	0.455	0.034	0.506	0.035
Metrics	Burak	Moodle		0.172	0.417	0.020	0.827	0.038	0.475	0.039
Tokens	Burak	phpMyAdmin		0.193	0.407	0.177	0.792	0.247	0.464	0.292

Table 4: Top 10 Classifiers by Descending AUCEC50 score with selected performance metrics for the PHP Vulnerability Dataset

Baseline. Second, the Source Baseline has high performance in absolute terms and is not a trivial baseline (compared to, say, the inspection baseline, which is significantly outperformed in all cases). In other words, it is still possible to produce high performing classifiers without complex transfer learning algorithms by simply using the Source Baseline in this case.

One hypothesis as to why this may have occurred lies in the tradeoff inherent in transfer learning: the idea is that "irrelevant" training data is filtered away or discounted during weighting; however, if the "irrelevant" training data turns out to be relevant, this method is effectively reducing the size of the training set, which will have negative implications on the performance of the algorithm. The best performing transfer learning algorithms and the Source Baseline tend to use all of the data to some extent, while lower performing methods tend to filter more aggressively. For example, the Burak and Peters filters have filtered training sets that are, at a maximum, the same size as the test set, and as the test set is usually considerably smaller (a single project), this has significant impact on the effectiveness of the classifier. At least for the selected datasets, it appears the benefit of transfer learning algorithms filtering, weighting, or clustering on the training data was outweighed by the cost of reducing the effective size of the training set.

It is interesting to note that while the PROMISE dataset showed statistically significant differences between the performance of the transfer learning algorithms, no significant differences were discovered in the PHP Vulnerability Dataset. There are two possible conclusions based on the results of the Friedman test: either (a) the difference between the methods is purely due to chance *or* (b) there exists a difference between the methods, but the Friedman test lacks the statistical power necessary to detect the difference. Support for the (b) can be found in the considerably smaller number of blocks (projects) in the PHP Vulnerability Dataset, which would drive test power down. Furthermore, because the Friedman test uses relative rankings instead of the normality assumption, it inherently has less power in general than ANOVA tests. It is worth noting that the number of blocks (projects to test on) in the PHP Vulnerability Dataset is considerably smaller than the number of projects in the PROMISE dataset, which makes the power even lower. In [25], Zimmerman explains that the Asymptotic Relative Efficiency (ARE, a measure of relative power) reveals that the Friedman test has relatively low power compared to other techniques. Therefore, more labeled projects are needed to conclude which method is best for vulnerability detection. Until then, Source Baseline is a good option based on its simplicity and similar performance to the other options.

The results from this study can now guide selection of algorithm for cross project prediction and provide a baseline for future transfer learning algorithms. For example, a practitioner with access to a dataset mapping file features to labels (vulnerable/not vulnerable or defective/not defective) can now compute the same features on his or her (unlabeled) project and use this study's results to determine which transfer learning algorithm to use (specifically, Source Baseline or TrBagg are a good choice). Alternatively, a researcher who has developed a new transfer learning algorithm can test the performance of the algorithm on this dataset, repeat the statistical tests, and easily determine whether the new transfer learning algorithm significantly outperforms the ones outlined in this study. Thus, this work has several practical benefits.

6. FUTURE WORK

In the present work, it appears that the Peters algorithm did not significantly outperform the Burak algorithm in any case, which differs from the results found in [20]. Experiments are now being conducted to see if the previous experiment can be replicated exactly, and in what cases the Peters and Burak algorithms have significantly different performance.

Future work will also consider the addition of different types of features which may contribute to better performance. While many metrics and tokens are available, it is worth considering more subjective features that underlie the code components. For example, adding labels for different data structures or coding patterns used by files could have performance benefits for the transfer learning algorithms. The idea is that conceptually similar components of the program can be identified and used to improve classification of vulnerabilities within those sections.

Finally, deconstructing the most useful features used by the algorithms could provide insight into what aspects of code contribute to faults in security and behavior. The

most useful features might be derived using feature subset selection algorithms, which are available by design in many machine learning algorithms. These insights could then be provided to developers in order to encourage coding practices that minimize defects and vulnerability.

7. CONCLUSIONS

In this paper, open source implementations of state of the art transfer learning algorithms were provided and compared on two public datasets. The results demonstrate that among the transfer learning algorithms, TrBagg and Cluster-Then-Label are the best performers for both vulnerability and defect prediction, but no algorithm significantly outperforms the Source Baseline. This suggests that even seemingly unrelated projects may be useful for cross-project defect and vulnerability prediction. In other words, the Source Baseline is still a high performing option in practice, even if it theoretically does not perform well on certain datasets where projects are vastly different.

Baselines provided by this paper can now be used to compare new transfer learning approaches in a consistent and easily understandable manner. These baselines can also guide the selection of transfer learning algorithms for defect and vulnerability detection in enterprise settings. It is hoped these tools will benefit both developers and practitioners seeking to more quickly and easily identify vulnerabilities and defects.

8. ACKNOWLEDGMENTS

I would like to thank my advisor Dr. James Purtilo for his considerable assistance with this project. I would also like to thank Ryan Eckenrod for his many suggestions and improvements to the contents of this paper.

9. REFERENCES

[1] N. Bettenburg, M. Nagappan, and A. E. Hassan. Think locally, act globally: improving defect and effort prediction models. pages 60–69, jun 2012.

[2] R. Caruana and A. Niculescu-Mizil. An empirical comparison of supervised learning algorithms. In *Proceedings of the 23rd international conference on Machine learning - ICML '06*, pages 161–168, New York, New York, USA, jun 2006. ACM Press.

[3] M. Dawson, D. N. Burrell, E. Rahim, and S. Brewster. Integrating software assurance into the software development life cycle (SDLC). *Journal of Information Systems Technology & Planning*, 3(6):49–53, 2010.

[4] J. Demšar. Statistical Comparisons of Classifiers over Multiple Data Sets. *The Journal of Machine Learning Research*, 7:1–30, 2006.

[5] M. Friedman. The Use of Ranks to Avoid the Assumption of Normality Implicit in the Analysis of Variance. *Journal of the American Statistical Association*, 32(200):675–701, 1937.

[6] T. Fukushima, Y. Kamei, S. McIntosh, K. Yamashita, and N. Ubayashi. An empirical study of just-in-time defect prediction using cross-project models. *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, pages 172–181, 2014.

[7] Z. He, F. Shu, Y. Yang, M. Li, and Q. Wang. An investigation on the feasibility of cross-project defect prediction. *Automated Software Engineering*, 19(2):167–199, 2012.

[8] M. Jureczko and L. Madeyski. Towards identifying software project clusters with regard to defect prediction. *Proceedings of the 6th International Conference on Predictive Models in Software Engineering - PROMISE '10*, page 1, 2010.

[9] T. Kamishima, M. Hamasaki, and S. Akaho. TrBagg: A simple transfer learning method and its application to personalization in collaborative tagging. *Proceedings - IEEE International Conference on Data Mining, ICDM*, pages 219–228, 2009.

[10] B. Klimkowski. *Analysis of a Semi-Supervised Learning Approach to Intrusion Detection*. PhD thesis, University of Maryland, 2014.

[11] E. Kocaguneli, G. Gay, T. Menzies, Y. Yang, and J. W. Keung. When to use data from other projects for effort estimation. In *Proceedings of the IEEE/ACM international conference on Automated software engineering - ASE '10*, page 321, New York, New York, USA, 2010. ACM Press.

[12] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings. *IEEE Transactions on Software Engineering*, 34(4):485–496, jul 2008.

[13] Y. Ma, G. Luo, X. Zeng, and A. Chen. Transfer learning for cross-company software defect prediction. *Information and Software Technology*, 54(3):248–256, 2012.

[14] 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*, 39(6):822–834, jun 2013.

[15] T. Menzies, J. Greenwald, 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 '. 33(November 2015):7–10, 2007.

[16] L. Minku, F. Sarro, E. Mendes, and F. Ferrucci. How to Make Best Use of Cross-Company Data for Web Effort Estimation ? *ESEM*, (3), 2015.

[17] P. Nemenyi. *Distribution-Free Multiple Comparisons*. PhD thesis, 1963.

[18] NIST. National Vulnerability Database (NVD).

[19] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[20] F. Peters, T. Menzies, and A. Marcus. Better cross company defect prediction. In *IEEE International Working Conference on Mining Software Repositories*, pages 409–418, 2013.

[21] J. Stuckman. *Continuous , Effort-Aware Prediction of Software Security Defects*. PhD thesis, University of Maryland, College Park, 2015.

[22] 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. *Empirical Software Engineering*, 14(5):540–578, 2009.

[23] J. Walden, J. Stuckman, and R. Scandariato. Predicting Vulnerable Components: Software Metrics vs Text Mining. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 23–33. IEEE, nov 2014.

[24] F. Yamaguchi, F. Lindner, and K. Rieck. Vulnerability Extrapolation: Assisted Discovery of Vulnerabilities Using Machine Learning. *Proceedings of the 5th USENIX Conference on Offensive Technologies*, page 13, 2011.

[25] D. W. Zimmerman and B. D. Zumbo. Relative Power of the Wilcoxon Test, the Friedman Test, and Repeated-Measures ANOVA on Ranks. *Journal of Experimental Education*, 62(1):11, 1993.