Better Cross Company Defect Prediction

Fayola Peters, Tim Menzies
Lane Department of CS & EE,
West Virginia University, USA
fayolapeters@gmail.com, tim@menzies.us

Andrian Marcus
Computer Science,
Wayne State University, USA
amarcus@wayne.edu

***Abstract*—How can we find data for quality prediction? Early in the life cycle, projects may lack the data needed to build such predictors. Prior work assumed that relevant training data was found *nearest to the local project*. But is this the best approach?**

This paper introduces the Peters filter which is based on the following conjecture: When local data is scarce, more information exists in *other projects*. Accordingly, this filter selects training data via the structure of *other projects*.

To assess the performance of the Peters filter, we compare it with two other approaches for quality prediction. Within-company learning and cross-company learning with the Burak filter (the state-of-the-art relevancy filter). This paper finds that: 1) within-company predictors are weak for small data-sets; 2) the Peters filter+cross-company builds better predictors than both within-company and the Burak filter+cross-company; and 3) the Peters filter builds 64% more useful predictors than both within-company and the Burak filter+cross-company approaches. Hence, we recommend the Peters filter for cross-company learning.

***Index Terms*—Cross company; defect prediction; data mining**

I. Introduction

Defect prediction is a method for predicting the number of defects in software. It is valuable for organizing a project's test resources [1]. For example, given limited resources for software inspection, defect predictors can focus test engineers on the modules most likely to be defective [2].

Zimmermann et al. [3] warn that *defect prediction works well within projects as long as there is a sufficient data to train models*. That is, to build defect predictors, we need access to historical data. If the data is missing, what can we do?

Cross Company Defect Prediction (CCDP) is the art of using data from other companies to build defect predictors. CCDP lets software companies with small unlabeled data-sets use data from other companies to build their quality predictors. Multiple recent studies have certified the utility of this approach for defect prediction [2], [4]–[6] (as well as effort estimation [7]). For example given the right *relevancy filtering* (described below and illustrated in Figure 1), Tosun et al. [8] used data from NASA systems to predict for defects in software for Turkish domestic appliances (and vice versa)[1].

A major issue in CCDP is *how to find the right training data in a software repository*. There is much data, freely available, on Software Engineering (SE) projects (e.g. this study uses 56 defect data sets from the PROMISE repository [10]). Rodriguez et al. document 18 repositories, including PROMISE,

[1]Elsewhere we have explained this surprising result by a consideration of clusters built from eigenvectors of the data [9].

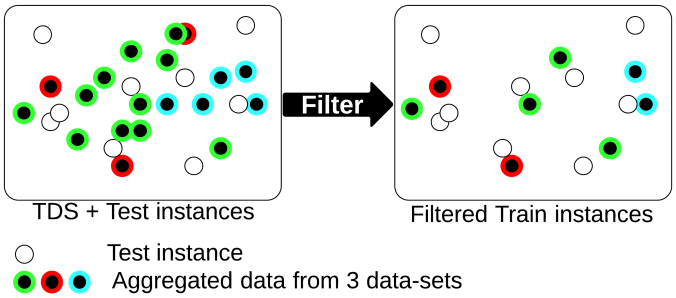


Fig. 1: Aggregate the data from the repository into a *TDS*. Using a filter with the Test instances find $FilteredTDS \subset TDS$.

that offer software project data [11]. However much of this data is irrelevant to specific projects. Turhan et al. showed that if we use *all* the data from a *Training Data Set (TDS)* - an aggregate of multiple data-sets, then the resulting defect predictor will have excessive false alarms [2]. A more recent study by Peters et al. [12] demonstrated that if we used *all* the data from a *TDS*, then false alarms *and* recall would be low.

When reasoning about new problems, it is wise to carefully reflect about the old data. Before we can find defects in local data, we must filter the *TDS* to select the most useful *Filtered TDS*. One such filtering method, shown in Figure 1, is the *Burak filter* [2] that returns the nearest *TDS* instance for each test instance. The core idea of the this filter is to use *Test* instances to guide the selection of the *Filtered TDS*.

The Burak filter must be repeated each time new test data arrives. But is that the right way to do the filtering? Is there any advantage to learning and caching some strutures in the training data *before* reflecting over the test data? The following speculation argues that such an advantage might exist in the form of the *Peters filter*:

- When one company wants to use data from many other companies, the expected case is that the *Test* data (from this company) is much less than the the *TDS* (the training data set) from all the other companies;
- When *Test* is smaller than *TDS* then there should be *more* information about defects in the *TDS* than in *Test*.
- Hence, when selecting relevant data, it might be better to guide that search using the structure of the *TDS* training data rather than the *Test* data.

The rest of this paper checks the above speculation. We propose the *Peters filter* that selects data by focusing more on the structure of *TDS* than of *Test*:

- The Burak filter uses the *Test* instances to finds its *Filtered TDS* (10 instances for each *Test* instance).
- The Peters filter lets the instances in the *TDS* find their nearest *Test* instance, and the ones nearest to their *Test* instances are selected for the final *Filtered TDS*.

The difference with the Peters filter is that not all *Test* instances will have a nearest *TDS* instance (we show an example of that, later in this paper, in Figure 2). To assess the Peters filter, this paper explores the following research questions.

- RQ1: Is there a need for cross-company learning?
- RQ2: Can the Peters filter produce better defect predictors than the Burak filter?
- RQ3: Does the Peters filter generate practical (or useful) defect predictors?

The contributions of this work are:

- A new method, called the Peters filter, for CCDP.
- A certification experiment that shows the Peters filter outperforming prior work on the Burak filter.
- A demonstration that CCDP can be applied very early in a project's lifecycle. In one extreme case, we demonstrate success with a test set as small as just six classes.

The rest of this paper is structured as follows: In Section II we summarize related work. In Section III, we expand on the experimental procedures followed in this work, the data-sets, the filters, the prediction models, and performance measures. Next, the experiments and results are discussed in Section IV and Section V respectively. Lastly, we conclude the paper with threats to validity and some notes on future work in Section VI, and the conclusion in Section VII.

II. Related Work

A. Defect Prediction Economics

Many researchers have documented the economical value of early defect detection [13]–[15]. Once we accept that it is economically effective to find bugs earlier, then the next question becomes "how do we find the bugs earlier?". Defect prediction learned from static code measures allows software companies to take advantage of early defect detection [16]. Models made for defect prediction are usually built with local or within-company data-sets using common machine learners. The data-sets are comprised of independent variables such as the *code metrics* used in this work and one dependent variable or prediction target with values (labels) to indicate if defects are present. However, this local data may not always be available for defect prediction. In this situation, rather than waiting until enough local data is collected for within-company defect prediction (WCDP), we propose the use of CCDP for *earlier* bug or defect detection.

B. CCDP = Cross Company Defect Prediction

When data can be shared between organizations, defect predictors from one organization can generalize to another. However, initial experiments with cross-company learning were either very negative [3] or inconclusive [17]. Zimmermann et. al. [3] observed, defect prediction via local data is not always available to many software companies as

- The companies may be too small.
- The product might be in its first release and so there is no past data.

Kitchenham et al. [17] also saw problems with relying on *within-company* data-sets. They noted that the time required to collect enough data on past projects from a single company may be prohibitive. Additionally, collecting *within-company* data may take so long that technologies used by the company would have changed and therefore older projects may no longer represent current practices.

Recently, we have had more success using better selection tools for training data [2], [18] but this success was only possible if the learner had unrestricted access to all the data. For example, defect predictors developed at NASA [16] have also been used in software development companies outside the US (in Turkey). When the inspection teams focused on the modules that triggered the defect predictors, they found up to 70% of the defects using just 40% of their QA effort (measured in staff hours) [19]. Work by Rahman et al. [6], also show the success of CCDP. Their focus was on cost sensitive prediction where only the top $n\%$ of reported defect prone lines or files were used in CCDP and WCDP experiments.

The particular focus of this paper is CCDP in the situation where the test set is much smaller than the examples available in other data sets (the *TDS*). Two recent research results lead us to this focus.

Firstly, in their systematic literature review, "Cross versus Within-Company Cost Estimation Studies", Kitchenham et al. [17] stated that the main aim of their work was to assist software companies with *small* data-sets in deciding whether or not to use an estimation model obtained from a benchmarking data-set. Although their work is based in effort estimation, we maintain that this distinction holds for defect prediction based on the cross company learning performed in their work.

Secondly, in order to motivate their work, the authors of Zimmermann et al. [3] comment on data sets from Firefox and Internet Explorer (IE). These web browsers were used in their CCDP experiments and the results showed that although Firefox data could predict for IE defects very well, IE data could not do the same for Firefox. To explain this curious asymmetry, Zimmermann et al. noted that their Firefox data had more files than IE and that "building a model from a small population to predict a larger one is likely more difficult than the reverse direction".

C. Measuring the Feasibility of CCDP

Beyond the use of small test sets for CCDP, we contend that the feasibility of CCDP should not depend on its comparison to WCDP. Instead, since recent studies have shown that the success of CCDP depends on selecting or creating the right Filtered TDS, the feasibility of CCDP should depend on the number of test sets able to access a Filtered TDS + predictor

combination that will build a defect model whose performance meet user criteria [3], [4].

More recent work in CCDP, have avoided the within company comparison for judging the success of CCDP. Instead CCDP success is based on perceived user criteria and conclusions are made based on these results. For instance, Zimmermann et al. [3] built 622 cross-company predictions and found that only 3.4% met the criteria they used to determine if a project was a strong predictor for another project (accuracy, precision and recall were above 75%). From these results, they concluded that CCDP remained a challenge. However, they also studied the factors which influence the the success of CCDP and used these factors to derive decision trees that provided estimates for precision, recall, and accuracy before a prediction was attempted. In others words careful selection of training data can determine the success of CCDP.

Similar to the work done by Zimmermann et al. [3], He et al. [4] checked when cross defect predictions that were successful. In their work, defect predictors were considered strong if recall was above 70% and precision above 50%. Although their criteria was different and less stringent than the Zimmermann et al. study [3], their results were similar, ranging from 0.32% to 4.67% of cross-company predictions that met their criteria over 5 different learners. However, the authors did not use this result as a measure of the success of CCDP, instead they based their measure on the results of selecting the best train data for a test set. On average there results met their criteria. Out of 34 test sets 18 were considered as strong defect predictors. From those results, they concluded that CCDP was feasible as long as it involved the careful selection of training data.

D. Transfer Learning

In the studies mentioned so far, work on selecting the best data-sets for CCDP has focused on decision trees [3], [4]. Besides this, filtering training data has also been used in the Burak filter [2] with some success. In their work each instance of the test set finds 10 of it's nearest neighbors from a *TDS*. This results in a Filtered TDS more similar to the test set.

Another method of improving CCDP through use of the test set, is Transfer Learning. Ma et al. [5] expressed concerns with the possible information loss associated with the filtering method used in the Burak filter. They transfered estimates of the distribution of the test set to the train data by using the information to weight the train instances. It is these weighted instances that are used as the train data. The work is benchmarked against the Burak filter [2]. Note that the improvements reported by Ma et al. are very small while the improvements shown below from the Peters filter are very large. Hence, in terms of experimental comparisons, we will compare the Peters filter to the Burak filter.

To summarize, we can find in the literature a clear justification for our experimental design (testing on small data-sets using data filtered from a larger *TDS* of aggregated of data-sets from a repository). It is important to note that the above argues that the success of CCDP should not be based on comparisons with within-company prediction or the ratio of success of multiple cross predictions. Instead it should be based on the ability to find or create for each test set, the best train data that produces results that meet user defined criteria. In this work we benchmark against Burak et al. [2] and show better CCDP results. Also we measure the success of CCDP in terms of the number of test sets that meet the criteria used in this work.

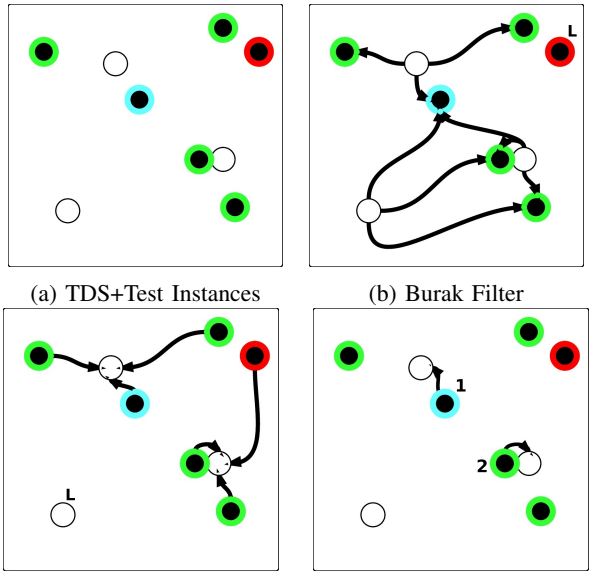
III. Methodology

A. The Filters

Figure 2 illustrates the steps involved in the Burak and Peters filters. The white circles are the test instances and the color-bordered black circles are the instances that make up the *TDS*. For the Burak filter, using k-nearest neighbor where *k=3*, each test instance will select three nearest *TDS* instances. This is made clear by the direction of the arrows of the Burak filter illustration of Figure 2. Also, note the lone *TDS* instance labeled L not selected by any of the test instances.

As discussed below, the Peters filter labels each *TDS* instance with its nearest test instance (in Figure 2c) Note that it is possible for a test instance (in this case, L) to not be selected by any of the *TDS* instances. To build the Filtered TDS, each test instance X reports it's nearest *TDS* instance with label X. This is shown in Figure 2d where labels 1 and 2 indicate the test instances and the respective *TDS* instances chosen for the final Filtered TDS.

In the end, Figure 2 shows that the Filtered TDS for the Burak filter contains five instances, while the Peters filter selects two instances for its Filtered TDS.



(a) TDS+Test Instances (b) Burak Filter

(c) Peters Filter 1 (d) Peters Filter 2

Fig. 2: Illustration of the Burak and Peters filters. The white circles are the test instances and the black circles bordered by green, blue or red are the train instances from different projects. See text for details.

1) The Burak Filter: This filter selects instances from *TDS* to create a Filtered TDS similar to the test set. The Burak filter simply used k-nearest neighbor to measure the similarity (with Euclidean distance) between test and *TDS* instances. In their experiment *k=10* and so each test instance contributed 10 of its nearest *TDS* instances to the new Filtered TDS. Prior to using the Burak filter, in their first analysis, Turhan et al. [2] found that like the Zimmermann study, using *TDS* lead to poor results (i.e. high false alarm rates). After applying the filter to *TDS*, they found that these cross-company predictors were equivalent to those learned from within-company data.

2) The Peters Filter: The Burak filter assumes that the test data should drive the selection process. The Peters filter is different- it assumes that the *TDS*'s content-rich database contains more information about defects than the smaller test set. Therefore, instead of each instance in the test set finding 10 nearest instances from the *TDS*, *for each TDS instance we find it's nearest test instance*. In other words, the Peters filter is like a popularity contest where each test instance is surrounded by its supporters from the *TDS*. In the end the test instance chooses its greatest fan (the one closest to it) as a candidate for the Filtered TDS while rejecting all the others. In this scenario, it is possible for a test instance to have no fans at all and so it does not contribute a candidate to the Filtered TDS. We conjecture that this particular point helps to produce a Filtered TDS based on the most influential test instances (those with at least one fan), and those with no influence at all will produce inferior defect models and therefore weak prediction results.

On a side note, if a test instance's closest fan is a duplicate of itself, it is rejected and the next greatest fan is selected for the Filtered TDS. Implementation note: the Peters filter used in this paper first clusters the *TDS* with the test set using *k*-means [20]. Next clusters containing at least one test instance are kept and others are rejected. Finally the Peters filtering process described above is applied to each remaining cluster. For example, with a *TDS*+Test totaling *n*=10,000 instances, we use *k*=1,000 for *k*-means, i.e. one cluster per *r*=10 *TDS*+Test instances. We use this value for *r* since Turhan et al. [2] used *10*-nearest neighbors for each test instance. In future work we will explore other values of *r*.

B. Data

A total of 56 static code defect data-sets from the PROMISE data repository [10] are used in this study. These data-sets were collected by Jureczko and Madeyski [21], and Jureczko and Spinellis [22]. Each instance in a data-set represents a class and consists of two parts: 20 independent static code attributes and the dependent attribute labeled "defects" indicating the number of defects in the class. For our work, we refer to each class as an instance. Additionally, instances with no defects are labeled as 0, and instances with one or more defects are labeled as 1. Table I shows the names and details of our test data sets, the number of instances in each data-set, and the number and percent of defects. The bottom part of that table also shows the data-sets used in the *TDS*. The last row indicating that there

TABLE I: Characteristics of Defect Data-sets. *Tests* data shown at the top, data-sets that make up the *TDS* shown below. All rows are sorted by the number of instances (in these data-sets, each instance is one class). The last 2 rows of this table show the total number of instances in the *TDS* along with the number and percent of total defects before and after removing duplicate instances from the *TDS*.

		Tests		
Defect Data	**Symbol**	**# Instances**	**# Defects**	**% Defects**
forrest-0.6	for06	6	1	16.7
ckjm	ckjm	10	5	50.0
wspomaganiepi	wsp	18	12	66.7
sklebagd	skleb	20	12	60.0
szybkafucha	szy	25	14	56.0
pbeans1	pb1	26	6	23.1
intercafe	inter	27	4	14.8
kalkulator	kal	27	6	22.2
nieruchomosci	nier	27	10	37.0
forrest-0.7	for07	29	5	17.2
zuzel	zuzel	29	13	44.8
forrest-0.8	for08	32	2	6.3
workflow	work	39	20	51.3
termoproject	termo	42	13	31.0
berek	berek	43	16	37.2
serapion	sera	45	9	20.0
skarbonka	skar	45	9	20.0
pbeans2	pb2	51	10	19.6
pdftranslator	pdf	53	15	28.3
e-learning	elearn	64	5	7.8
systemdata	sys	65	9	13.8
		Data for the TDS		
Defect Data	**Symbol**	**# Instances**	**# Defects**	**% Defects**
log4j-1.1	log11	109	32	29.4
ivy-1.1	ivy11	111	63	56.8
ant-1.3	ant13	125	20	16.0
log4j-1.0	log10	135	34	25.2
synapse-1.0	syn10	157	16	10.2
xerces-init	xer	162	77	47.5
ant-1.4	ant14	178	40	22.5
lucene-2.0	luc20	195	91	46.7
velocity-1.4	vel14	196	49	25.0
velocity-1.5	vel15	214	72	33.6
synapse-1.1	syn11	222	60	27.0
velocity-1.6	vel16	229	78	34.1
poi-1.5	poi15	237	141	59.5
ivy-1.4	ivy14	241	16	6.6
lucene-2.2	luc22	247	144	58.3
synapse-1.2	syn12	256	86	33.6
jedit-3.2	jedit32	272	90	33.1
ant-1.5	ant15	293	32	10.9
jedit-4.0	jedit40	306	75	24.5
jedit-4.1	jedit41	312	79	25.3
poi-2.0	poi20	314	37	11.8
camel-1.0	cam10	339	13	3.8
ant-1.6	ant16	351	92	26.2
jedit-4.2	jedit42	367	48	13.1
poi-2.5	poi25	385	248	64.4
xerces-1.2	xer12	440	71	16.1
poi-3.0	poi30	442	281	63.6
xerces-1.3	xer13	453	69	15.2
xerces-1.4	xer14	588	437	74.3
camel-1.2	cam12	608	216	35.5
prop-6	prop6	660	66	10.0
xalan-2.4	xal24	723	110	15.2
xalan-2.5	xal25	803	387	48.2
camel-1.4	cam14	872	145	16.6
xalan-2.6	xal26	885	411	46.4
TOTAL		**12427**	**3926**	**31.6**
TOTAL UNIQUE		**9552**	**3710**	**38.8**

are a total of 9552 unique instances of which 3710 contain at least one defect. In addition, Table II describes the attributes of these data-sets.

The selection criteria for the Test and *TDS* data-sets are as follows:

- The data-sets used are publicly available for reproducibility purposes and each are chosen arbitrarily.
- Each data-set has the same static code attributes (see Menzies et al. [16] for information on the value of static code attributes as defect predictors).
- Each test set is small, i.e. much smaller than the combined data sets in the *TDS*. For the sample we had available, this meant using data with less that 100 instances.
- None of the data-sets that make up the *TDS* are used as test sets in this work.

C. Prediction Models

For our work we chose four prediction models which represent a wide range of research in defect prediction [23]. First, Random Forest (RF) is used based on the results of a benchmarking study done by Lessmann et al. [23] which showed RF to be significantly better than 21 other predictors. Second, it is reported by Menzies et al [16] and Lessmann et al. [23] that Naive Bayes (NB) performs well compared to more complex predictors. Third, Logistic Regression (LR) is favored by Zimmermann, and Weyuker et al. [3], [24]. Last, the simplistic $K = 1$NN learner is used to find a lower bound on predictor performance.

We do not implement these ourselves, instead we use their Weka [25] implementations with the default values. Note, when using LR, the default threshold value is 0.5. This treats the *false negatives* and *false positives* equally. More details for the prediction models are provided below.

1) Random Forest, RF: Breiman [26] describes RF as a combination of tree predictors such that each tree depends on the values of a random vector sampled independently and with the same distribution for all trees in the forest. In other words, RF is a collection of trees, where each tree is grown from a bootstrap sample (randomly sampling the data with replacement). Additionally, the attributes used to find the best split at each node is a randomly chosen subset of the total number of attributes. Each tree in the collection is used to classify a new instance. The forest then selects a classification by choosing the majority result.

2) Naive Bayes, NB: A statistical learning scheme that assumes that attributes are equally important and statistically independent. Lewis [27] describes NB as a classifier based on Baye's rule shown below:

$$P(c_k|x) = P(c_k) \times \frac{P(x|c_k)}{P(x)}$$

where c_k is a member of the set of values for the dependent attribute. In our case c_k could be 0 or 1. Also, x represents a test instance or unknown instance. So, to classify a test instance, NB finds the conditional probability of that instance being labeled c_k. The c_k with the highest probability is chosen as the label for x.

3) Logistic Regression, LR: Afzal [28] recommend LR when the dependent variable is dichotomous (e.g. either fault-prone or non-fault-prone). The method avoids the Gaussian assumption used in standard Naive Bayes. The form of the logistic regression model is:

$$log(\frac{p}{1-p}) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_k X_k$$

where p is the probability that the fault was found in the module and $X_1, X_2, ..., X_k$ are the independent variables. $\beta_0, \beta_1, ..., \beta_k$ are the regression coefficients estimated using maximum likelihood.

4) K-Nearest Neighbor, KNN: Cover and Hart [29] describes KNN as a simple non-parametric decision procedure which classifies x, an unknown instance in the category of its nearest neighbor. KNN is one of the simplest defect predictors that can be used. It is therefore used as a baseline for the more complicated methods described above.

D. Performance Measures

The performance measures used for the defect predictors described above are shown in Table III and summarized below.

- Accuracy measures the percentage of correctly classified instances of both the defective and non-defective classes.
- Recall or *pd* is equal to how much of the target (defective instances) are found. The higher the pd, the fewer the false negative results.
- Probability of false alarm or *pf* measures how many of the instances that triggered the detector actually did not contained the target (defects) concept. Like pd, the highest pf is 100% however its optimal result is 0%.
- Precision measures how many predicted as defects are actually defects.
- *f*-measure is a dual assessment of both recall and precision. It has the property that if either precision or recall is low, then the *f*-measure is decreased. We refrain from reporting *f*-measures in this work based on the the study done by Menzies et al. [30] which shows that when data-sets contain a low percentage of defects, precision can be unstable. If we look at our test sets at the top of Table I, we see that defects are rare in most cases.
- *g*-measure (harmonic mean of pd and 1-pf): Instead of the *f*-measure, we report on the *g*-measure. The 1-pf represents *Specificity* (not predicting instances without defects as defective. *Specificity* (1-pf) is used together with *pd* to form the *G-mean*$_2$ measure seen in Jiang et al. [31]. It is the geometric mean of the pd's for both the majority and the minority class. In our case, we use these to form the *g*-measure which is the harmonic mean of pd and 1-pf.

In this work, for the CCDP experiments we report only on *g*-measures. However, for the reader of this work who wishes to use different performance measures, we include the true positive (TP), true negative (TN), false positive (FP), and false negative (FN) for these results. However for the WCDP results in Table IV, since results show the median of a 3-way

TABLE II: The C-K metrics of the data-sets used in this work (see Table I). The last row is the dependent variable. Jureczko et al. [21] provide more information on these metrics.

Attributes	Symbols	Description
average method complexity	amc	e.g., number of JAVA byte codes
average McCabe	avg_cc	average McCabe's cyclomatic complexity seen in class
afferent couplings	ca	how many other classes use the specific class
cohesion amongst classes	cam	summation of number of different types of method parameters in every method divided by a multiplication of number of different method parameter types in whole class and number of methods
coupling between methods	cbm	total number of new/redefined methods to which all the inherited methods are coupled
coupling between objects	cbo	increased when the methods of one class access services of another
efferent couplings	ce	how many other classes is used by the specific class
data access	dam	ratio of the number of private (protected) attributes to the total number of attributes
depth of inheritance tree	dit	provides the position of the class in the inheritance tree
inheritance coupling	ic	number of parent classes to which a given class is coupled (includes counts of methods and variables inherited)
lack of cohesion in methods	lcom	number of pairs of methods that do not share a reference to an instance variable
another lack of cohesion measure	locm3	if m, a are the number of $methods, attributes$ in a class number and $\mu(a)$ is the number of methods accessing an attribute, then $lcom3 = ((\frac{1}{a}\sum_j^a \mu(a_j)) - m)/(1-m)$
lines of code	loc	measures the volume of code
maximum McCabe	max_cc	maximum McCabe's cyclomatic complexity seen in class
functional abstraction	mfa	number of methods inherited by a class plus number of methods accessible by member methods of the class
aggregation	moa	count of the number of data declarations (class fields) whose types are user defined classes
number of children	noc	measures the number of immediate descendants of the class.
number of public methods	npm	counts all the methods in a class that are declared as public. The metric is known also as Class Interface Size (CIS)
response for a class	rfc	number of methods invoked in response to a message to the object
weighted methods per class	wmc	the number of methods in the class (assuming unity weights for all methods).
defects	defects	number of defects per class, seen in post-release bug-tracking systems

TABLE III: Some popular measures used in software defect prediction work.

		Actual	
		yes	**no**
Predicted	**yes**	TP	FP
	no	FN	TN
Accuracy	$\frac{TP+TN}{TP+TN+FP+FN}$		
Recall (pd)	$\frac{TP}{TP+FN}$		
pf	$\frac{FP}{FP+TN}$		
Precision (prec)	$\frac{TP}{TP+FP}$		
f-measure	$\frac{2*pd*prec}{pd+prec}$		
***g*-measure**	$\frac{\textbf{2*pd*(100-pf)}}{\textbf{pd+(100-pf)}}$		

experiment, we report five performance measures (accuracy, pd, precision, pf, and g-measure).

IV. Experimentation

All our experiments are designed around the three research questions from Section I. However, before we elaborate on these it is important to explain how the data-sets are selected and used in our experiments. From the PROMISE repository, we arbitrarily choose 56 defect data-sets for our experiments. We then group these data-sets into two categories: *tests* and *data*. Data-sets qualify for the former category if they contain less than 100 instances (small data-sets) and data-sets qualify for the latter category if they don't fall into the first category. It is important to note however that data-sets of any size can be a part of *data*. As a result of these criteria, there are 21 data-sets in *tests* and 35 data-sets in *data*. These are shown in Figure 3 and in more detail in Table I.

To check *if there is a need for cross-company learning* (RQ1), we conduct a WCDP experiment, i.e. a cross-validation experiment. This is a standard evaluation approach in Machine Learning where an experiment is repeated *n* times on *m* random subsamples of data. In other words, *n*-times, m_{all}-m_i is treated as the training set and m_{all} - *training set* is the test set. We use a 3-way cross-validation for each test set where n and m are both 3. We used a 3-way since the data-sets are too small to support the standard 10-way cross-validation. This experiment was repeated once for each learner.

To find out if *the Peters filter can produce better defect predictors than the Burak filter* (RQ2), we used the Filtered TDS, to create defect predictors with RF, NB, LR and KNN (see Section III-C for explanation of the filters). Figure 3 shows the experiment for the Peters filter, however the same procedure is followed for the Burak filter. To begin, we first bind the data-sets in *data* together to form what we call a *TDS*. We then make sure that the *TDS* only contains unique instances using the **ExtractUnique** function. Next, the inner for-loop iterates through each data-set (test) in *tests* and the filter is used to create a cross-company Filtered TDS. This Filtered TDS is used to build a defect predictor which is then evaluated on *test* to return performance measures of TP, TN, FP, and FN. Finally, the outer for-loop indicates that the above occurs for each of the four learners used in this work.

We compared the performance of our defect predictors for each filter using g-measures described in Section III-D. From the difference in the g-measures between these treatments, we

```
data = [log11, ivy11, ant13, log10, syn10, ant14,
        luc20, vel14, vel15, syn11, vel16, poi15,
        ivy14,luc22, syn12, jedit32, ant15, jedit40,
        jedit41, poi20, cam10, ant16, jedit42, poi25,xer,
        xer12, poi30, xer13, xer14, cam12, prop6, xal24,
        xal25,cam14, xal26]
tests = [for06, ckjm, wsp, skleb, szy, pb1, inter, kal,
         nier, for07, zuzel, for08, work, termo, berek,
         sera, skar, pb2, pdf, elearn, sys]
filter = [Peters]
learners = [NB, RF, LR, NN]  // defect predictors

// Create training data
TDS =  ExtractUnique(BindRows(data))

FOR EACH learner in learners
    // Apply filter to TDS for each test.
    FOR EACH test in tests
        // Get filtered cross company training data.
        Filtered TDS = filter(TDS, test)
        // Apply learner to each Filtered TDS
        // to create defect predictors.
        predictor = learner(Filtered TDS)
        // Evaluate predictor on test
        [TP, TN, FP, FN] = predictor(test)
    END FOR
END FOR
```

Fig. 3: Pseudo code for experimental design. Here only the Peters filter is shown, however the same procedure is followed for the Burak filter.

find the Δg between the performance of the Burak and the Peters filter on the same test data:

- If Δg is negative, then Burak performed *better* than the Peters filter;
- Otherwise, the Peters filter was as good, or better, than the Burak filter.

Finally, to determine if *the Peters filter could generate practical (or useful) defect predictors* (RQ3), we counted the number of test-sets which had defect predictors with *g*-measures above 75% and 60%. We chose 75% based on the work of Zimmermann et al. [3] which indicated that a strong predictor was one where precision, recall, and accuracy were each greater than 75%. We add 60% because it is the lower bound surpassed by all test sets using the Peters filter.

V. Results

We organize our results around the three research questions.

RQ1: Is there a need for cross-company learning? Table IV shows the within-company defect prediction (WCDP) 3-way cross-validation results from the test data. For each learner, the data is sorted top-to-bottom from largest to smallest *g*-measure. Observe how small data sets can generate weak defect predictors. While the median result for some learners seems adequate (e.g. the 63% for Naive Bayes), if we look beyond the median results, we see a large number *g=0* results for all learners (even for Random Forests, which Lessmann et al. [23] declares to be the current state-of-the-art defect predictor).

From these results, we conclude that cross-company learning is required when organizations need to make predictions about small data sets. For such data sets, it is not enough to apply standard learners to build defect predictors. Data sets this small need to be augmented with training data taken from some repository. Our other results, shown below, illustrate what are the best methods for selecting that data.

RQ2: Can the Peters filter produce better defect predictors than the Burak filter? Table V shows the results for Peters filter executing with the 4 defect predictors used in this work for 21 test sets. For each learner, the results are sorted on the Δg measures; i.e. by the difference performance measure seen using the Burak filter and the Peters filter (due to page constraints, we omitted the table of results for the Burak filter).

For all learners, the median Δg results were always positive; i.e. the Peters filter selects much better Filtered TDS than the Burak filter. Note, that in the minority of cases, the Burak filter outperforms the Peters filter (e.g. *skar, inter, for08* and *sys* for Random Forests in Table V). In future work we will investigate these projects further. Also, the better the learner, the greater the improvement. Note how the median Δg for Random Forest was very large (40%). In addition, Naive Bayes performs relatively better than our baseline Nearest Neighbor with a median Δg of 20% while Logistic Regression is comparable with a Δg of 14%.

RQ3: Does the Peters filter generate practical (or useful) defect predictors? Figure 4, summarizes the actual number of test sets that meet the 60% and 75% criteria for all learners. For example, *pb2* has a *g*-measure of 46% with NB and 45% for both LR and KNN, however for RF its *g*-measure is 66%. We therefore count this as a strong defect predictor for *pb2* when the criteria is g>60% and not when it is g>75%. For each labeled g>60, and g>75, the Peters filter produces strong predictors for approximately twice the number of test sets than the Burak filter.

We have included the 3×3 WCDP results in Figure 4. These results are not directly comparable to the Burak and Peters results since they come from a cross-validation experiment. However, informally, we say that these results support the conclusions of Turhan et al. [2] that CCDP+Burak filter performs just as well as WCDP. Also, they highlight the dramatic result of CCDP+Peters filter.

VI. Discussion

A. External Validity

Measured in terms of external validity, these results are stronger than many other papers in this area of research. The size of our study is 56 data-sets which is around 5 to 10 times larger than most papers in this field. That said, there is a clear bias in our sample: open-source Java systems. As more data becomes available, we plan to repeat this study.

Additionally, another source of bias in this study are the learners used for the defect prediction studies. Data mining is a large and active field and any single study can only use a small subset of the known data mining algorithms. In this work, results for Naive Bayes, Random Forests, Logistical Regression, and K-nearest Neighbor are published.

TABLE IV: Results for a 3×3 within cross-validation experiment for WCDP. Tables show results for Naive Bayes, Random Forest, Logistic Regression and K-nearest neighbor. Each row shows the medians of 5 performance measures (accuracy, pd, precision, pf, and g-measure).

WCDP	Naive Bayes					Random Forest					Logistic Regression					K-Nearest Neighbor				
Data	**acc**	**pd**	**prec**	**pf**	**g**	**Data**	**acc**	**pd**	**prec**	**pf**	**g**	**Data**	**acc**	**pd**	**prec**	**pf**	**g**	**Data**	**acc**	**pd**	**prec**	**pf**	**g**
berek	87	83	100	0	**88**	berek	93	83	80	9	**89**	berek	93	83	86	9	**87**	berek	86	80	75	11	**83**
sera	80	67	50	9	**76**	pdf	82	80	100	0	**80**	pdf	73	80	75	25	**73**	pdf	73	60	75	20	**70**
zuzel	78	75	86	14	**75**	nier	78	75	75	0	**73**	pb1	78	86	86	50	**63**	term	71	67	60	18	**69**
pdf	73	60	71	22	**70**	work	69	60	71	33	**69**	sera	73	50	50	18	**62**	kal	78	50	67	13	**67**
sys	86	60	29	5	**69**	zuzel	70	67	75	20	**69**	kal	67	50	33	25	**60**	pb1	75	86	86	50	**63**
nier	78	50	67	14	**67**	kal	78	50	50	13	**63**	nier	67	67	67	17	**60**	wsp	75	75	75	50	**60**
szy	67	67	75	20	**67**	pb1	78	86	86	50	**63**	wsp	67	75	75	50	**60**	zuzel	70	63	67	25	**60**
wsp	83	100	80	50	**67**	szy	63	67	67	40	**63**	zuzel	60	63	67	29	**60**	work	62	67	63	50	**56**
term	79	57	67	10	**63**	term	79	50	75	9	**60**	work	62	67	71	38	**59**	szy	50	67	60	50	**50**
kal	67	50	33	29	**63**	wsp	67	75	75	33	**60**	szy	63	57	67	25	**53**	pb2	76	33	33	15	**48**
pb1	67	75	83	50	**63**	skleb	57	80	71	67	**46**	elearn	90	33	25	5	**50**	sys	81	33	25	6	**48**
skleb	57	50	75	50	**50**	sera	80	25	50	0	**40**	term	71	33	50	14	**49**	nier	67	50	50	20	**44**
skar	73	33	33	17	**48**	pb2	76	20	20	8	**33**	for07	70	33	25	20	**48**	skleb	57	80	75	67	**44**
work	62	44	75	13	**47**	ckjm	50	50	50	33	**0**	skar	67	33	25	25	**46**	sera	73	25	33	8	**40**
pb2	76	20	50	8	**33**	elearn	86	0	0	0	**0**	skleb	57	80	60	67	**40**	ckjm	33	33	25	100	**0**
ckjm	33	50	33	67	**0**	for06	100	0	0	0	**0**	sys	76	20	20	17	**33**	elearn	86	0	0	0	**0**
elearn	86	0	0	10	**0**	for07	70	0	0	0	**0**	pb2	71	20	33	19	**31**	for06	50	0	0	0	**0**
for06	100	0	0	0	**0**	for08	91	0	0	0	**0**	ckjm	33	50	50	67	**0**	for07	60	0	0	0	**0**
for07	80	0	0	0	**0**	inter	78	0	0	0	**0**	for06	50	0	0	0	**0**	for08	90	0	0	0	**0**
for08	100	0	0	0	**0**	skar	73	0	0	8	**0**	for08	82	0	0	10	**0**	inter	78	0	0	0	**0**
inter	67	0	0	0	**0**	sys	77	0	0	0	**0**	inter	78	0	0	13	**0**	skar	73	0	0	15	**0**
MEDIAN					**63**						**46**						**50**						**48**

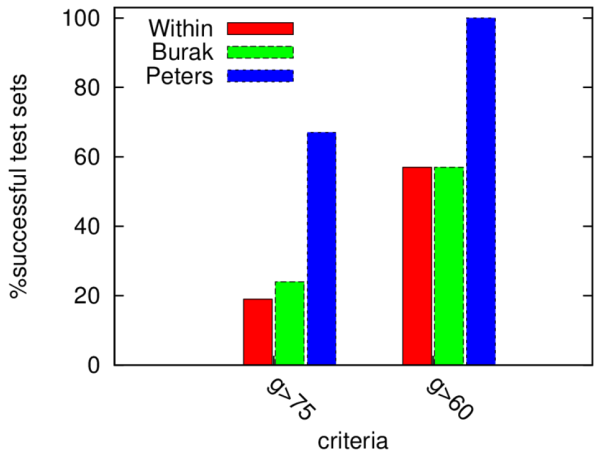


Fig. 4: Measuring the success of CCDP by the percent of test sets whose defects were predicted and met the criteria for g-measures greater than 60% and greater than 75%. For all criteria except $g > 60$, the Peters filter builds twice as many successful prediction models than WCDP, and the Burak filter. When the criteria is $g > 60$ the Peters filter has successful prediction models for *all* test sets.

Questions of validity also arise in terms of how the projects (data-sets) are chosen for our experiments. All 56 data-sets are chosen arbitrarily with the only stipulation that they all have the same attributes for ease in experimentation. Other data-sets which met this stipulation were not used in order to keep our experiment manageable. Therefore it is not clear if our results would generalize to those unused data-sets.

Finally, all the data used in this work are from open source projects. Although our results seem stable across different learners, we are not sure of how well they would generalize to closed source projects.

B. Future Work

Clearly, the experiments of this paper should be repeated on more data and we hope our results encourages more cross-company experiments (and more work on building SE models that are general to large classes of systems).

An open issue with our approach is the cost of the nearest neighbor methods used by the Peters filter. In this work we use k-means [20] to optimize the processing of each instance finding its nearest test instance. For future work we would explore other methods such as canopy clustering [32] or mini-batch k-means [33].

Also, for a minority of projects, the Burak filter has relatively better g-measures than the Peters filter. We will investigate these projects in future work.

Another open issue is our use of clusters of size ten. We used that size since prior work on the Burak filter showed that this was a useful division of the data. However, once we have faster clustering methods, it would be practical and insightful to explore a much wider range of cluster sizes.

VII. Conclusion

A decade ago, it was not known if cross-company data mining was possible. Preliminary results were either inconclusive [17] or negative [3].

In 2009, the Burak filter was the first clear demonstration that data miners could take defect data from one project and successfully apply them to another. One of the accomplishments of that work was to show *why* prior results were so inconclusive or negative. Not all data from other companies is relevant to the local company. Some *relevancy filter* must be applied to select the right training data.

Subsequent work on the Burak filter offered modest improvements [4], [5]. But until this work, the core premise of the Burak filter remained unchallenged (that test data was the best guide for selecting training data). In this work, we show that if we guide data selection via the *TDS* data, then major

TABLE V: Results for the Peters algorithm. Tables show results for Naive Bayes, Random Forest, Logistic Regression and K-nearest neighbor. The Δg column represents the deltas between the g of the Burak result and the Peters result. The positive numbers for Δg indicate how many times the Peters result was better than the Burak result.

Peters			**Naive Bayes**			
Data	**tp**	**tn**	**fp**	**fn**	**g**	Δg
inter	2	22	1	2	66	**-20**
elearn	2	56	3	3	56	**-16**
sera	5	35	1	4	71	**-1**
pb2	3	39	2	7	46	0
sys	4	53	3	5	60	1
for08	1	28	2	1	65	6
for07	3	18	6	2	67	12
ckjm	3	4	1	2	69	15
skar	5	27	9	4	64	17
work	7	18	1	13	51	18
pdf	9	15	3	6	70	20
skleb	7	7	1	5	70	20
nier	8	14	3	2	81	24
kal	3	12	9	3	53	25
szy	10	10	1	4	80	30
term	7	27	2	6	68	31
berek	13	26	1	3	88	34
pb1	10	6	0	10	67	41
wsp	10	5	1	2	83	55
zuzel	10	16	0	3	87	61
for06	1	5	0	0	100	100
MEDIAN					**68**	**20**

			Random Forest			
Data	**tp**	**tn**	**fp**	**fn**	**g**	Δg
skar	1	33	3	8	20	**-29**
inter	1	22	1	3	40	**-23**
for08	1	30	0	1	67	**-18**
sys	3	52	4	6	49	**-17**
for07	0	24	0	5	0	0
elearn	1	59	0	4	33	0
pb2	5	40	1	5	66	15
sera	5	35	1	4	71	22
skleb	7	8	0	5	74	24
berek	13	27	0	3	90	24
ckjm	5	4	1	0	89	36
nier	10	13	4	0	87	43
pdf	13	17	1	2	90	43
term	8	29	0	5	76	50
work	12	16	3	8	70	52
zuzel	9	16	0	4	82	56
kal	5	19	2	1	87	59
pb1	19	5	1	1	89	63
szy	13	9	2	1	87	74
wsp	12	5	1	0	91	76
for06	1	5	0	0	100	100
MEDIAN					**76**	**40**

Peters			**Logistic Regression**			
Data	**tp**	**tn**	**fp**	**fn**	**g**	Δg
term	7	21	8	6	62	**-24**
berek	13	26	1	3	88	**-10**
inter	1	20	3	3	39	**-1**
elearn	1	56	3	4	33	0
for06	1	4	1	0	89	0
skar	3	29	7	6	47	0
work	8	13	6	12	50	1
pb2	3	37	4	7	45	1
wsp	5	5	1	7	56	6
skleb	8	5	3	4	65	9
sera	6	31	5	3	75	14
kal	3	19	2	3	64	15
nier	9	8	9	1	62	16
sys	5	54	2	4	70	21
for07	1	15	9	4	30	30
pdf	10	12	6	5	67	33
szy	5	9	2	9	50	36
ckjm	5	5	0	0	100	47
pb1	19	4	2	1	78	52
zuzel	7	14	2	6	67	67
for08	2	25	5	0	91	91
MEDIAN					**64**	**14**

			Nearest Neighbor			
Data	**tp**	**tn**	**fp**	**fn**	**g**	Δg
for08	0	29	1	2	0	**-89**
inter	1	22	1	3	40	**-39**
kal	0	18	3	6	0	**-26**
pb2	3	36	5	7	45	**-26**
sys	3	50	6	6	49	**-24**
pdf	7	11	7	8	53	**-12**
skar	4	32	4	5	59	0
for07	3	17	7	2	65	0
work	8	17	2	12	55	2
sera	2	34	2	7	36	3
ckjm	3	5	0	2	75	15
elearn	3	52	7	2	71	17
wsp	12	3	3	0	67	19
skleb	8	7	1	4	76	22
berek	11	27	0	5	81	25
nier	6	11	6	4	62	30
term	8	24	5	5	71	34
pb1	18	5	1	2	87	40
szy	11	9	2	3	80	45
zuzel	11	14	2	2	86	63
for06	1	4	1	0	89	89
MEDIAN					**65**	**16**

improvements in performance can be expected. The difference between the Peters and Burak filter seems very small- just some details on what controls the first pass selection of train data. However, that seemingly small difference can lead to major performance improvement (recall the +40% difference in the g-measures with the Peters filter and Random Forests).

Our conclusions from this work are two-fold. Firstly, this work has strengthened the business case for cross-company learning in software engineering. As reported by Rodriguez et al. [11], there are now numerous repositories where projects can gather on-line data about prior SE projects. When programming teams need to assess the quality of their projects, if they lack local data for that task, then it is possible to use data from those on-line repositories.

Our second conclusion is more theoretical and speculative. Our results suggest that large *TDS*s created from repositories, contain useful structures that can help us guide and control new projects (we know this since the only difference between the Peters and Burak filters is in the former, the *TDS* has more controls over the selection of the nearest neighbors). These structures cannot be summarized as some simple single trite theory (e.g. the infamous $v(g) > 10$ defect predictor). Rather, it might be that when we look at a large enough sample (such as the 9552 unique instances studied in this work) then there appears hundreds of tiny micro-theories, each of which offers different, but powerful, guidance for quality improvement.

If so, then the challenge for the future is to understand this large space of multi-micro-theories. Note that our results

suggest that there are many of these micro-theories, but not an infinite number. If we could somehow visualize and navigate and exploit this large ensemble of micro-theories, then we might be able to better define the root causes of quality (or lack of quality) in software projects.

ACKNOWLEDGMENTS

The work was funded by NSF grant CCF:1017330 and the Qatar/West Virginia University grant NPRP 09-12-5-2-470.

REFERENCES

[1] A. Bener and T. Menzies, "Guest editorial: learning to organize testing," *Automated Software Engineering*, vol. 19, pp. 137–140, 2012.

[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 Software Engineering*, vol. 14, pp. 540–578, 2009.

[3] 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 *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 91–100.

[4] Z. He, F. Shu, Y. Yang, M. Li, and Q. Wang, "An investigation on the feasibility of cross-project defect prediction," *Automated Software Engineering*, vol. 19, pp. 167–199, 2012.

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

[6] 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*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 61:1–61:11.

[7] E. Kocaguneli and T. Menzies, "How to find relevant data for effort estimation?" in *Proceedings of the 2011 International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 255–264.

[8] A. Tosun, A. Bener, and R. Kale, "Ai-based software defect predictors: Applications and benefits in a case study," in *Twenty-Second IAAI Conference on Artificial Intelligence*, 2010, pp. 1748–1755.

[9] T. Menzies, A. Butcher, A. Marcus, T. Zimmermann, and D. Cok, "Local vs. global models for effort estimation and defect prediction," in *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, nov. 2011, pp. 343 –351.

[10] T. Menzies, B. Caglayan, E. Kocaguneli, J. Krall, F. Peters, and B. Turhan. (2012, June) The promise repository of empirical software engineering data. [Online]. Available: promisedata.googlecode.com

[11] D. Rodriguez, I. Herraiz, and R. Harrison, "On software engineering repositories and their open problems," in *Realizing Artificial Intelligence Synergies in Software Engineering (RAISE), 2012 First International Workshop on*, june 2012, pp. 52 –56.

[12] F. Peters, T. Menzies, L. Gong, and H. Zhang, "Balancing privacy and utility in cross-company defect prediction," *Software Engineering, IEEE Transactions on*, vol. PP, no. 99, p. 1, 2013.

[13] B. Boehm and P. Papaccio, "Understanding and controlling software costs," *IEEE Trans. on Software Engineering*, vol. 14, no. 10, pp. 1462–1477, October 1988.

[14] F. Shull, V. Basili, B. Boehm, A. W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz, "What we have learned about fighting defects," in *Proceedings of the 8th International Symposium on Software Metrics*, ser. METRICS '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 249–258.

[15] J. B. Dabney, G. Barber, and D. Ohi, "Predicting software defect function point ratios using a bayesian belief network," in *Proceedings of the PROMISE workshop*, 2006.

[16] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *Software Engineering, IEEE Transactions on*, vol. 33, no. 1, pp. 2 –13, jan. 2007.

[17] B. A. Kitchenham, E. Mendes, and G. H. Travassos, "Cross versus within-company cost estimation studies: A systematic review," *IEEE Trans. Softw. Eng.*, vol. 33, no. 5, pp. 316–329, May 2007.

[18] 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*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 321–324.

[19] A. Tosun, B. Turhan, and A. Bener, "Practical considerations in deploying ai for defect prediction: a case study within the turkish telecommunication industry," in *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, ser. PROMISE '09. New York, NY, USA: ACM, 2009, pp. 11:1–11:9.

[20] A. K. Jain, "Data clustering: 50 years beyond k-means," *Pattern Recognition Letters*, vol. 31, no. 8, pp. 651 – 666, 2010.

[21] M. Jureczko and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*. ACM, 2010, p. 9.

[22] M. Jureczko and D. Spinellis, "Using object-oriented design metrics to predict software defects," *Models and Methods of System Dependability. Oficyna Wydawnicza Politechniki Wrocławskiej*, pp. 69–81, 2010.

[23] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *Software Engineering, IEEE Transactions on*, vol. 34, no. 4, pp. 485 –496, july-aug. 2008.

[24] E. Weyuker, T. Ostrand, and R. Bell, "Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models," *Empirical Software Engineering*, vol. 13, pp. 539–559, 2008.

[25] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *SIGKDD Explor. Newsl.*, vol. 11, pp. 10–18, November 2009.

[26] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[27] D. Lewis, "Naive (bayes) at forty: The independence assumption in information retrieval," in *Machine Learning: ECML-98*, ser. Lecture Notes in Computer Science, C. Ndellec and C. Rouveirol, Eds. Springer Berlin / Heidelberg, 1998, vol. 1398, pp. 4–15.

[28] W. Afzal, "Using faults-slip-through metric as a predictor of fault-proneness," in *Proceedings of the 2010 Asia Pacific Software Engineering Conference*, ser. APSEC '10, 2010, pp. 414–422.

[29] T. Cover and P. Hart, "Nearest neighbor pattern classification," *Information Theory, IEEE Transactions on*, vol. 13, no. 1, pp. 21 –27, january 1967.

[30] T. Menzies, A. Dekhtyar, J. Distefano, and J. Greenwald, "Problems with precision: A response to "comments on 'data mining static code attributes to learn defect predictors'"," *IEEE Trans. Softw. Eng.*, vol. 33, no. 9, pp. 637–640, Sep. 2007.

[31] Y. Jiang, B. Cukic, and Y. Ma, "Techniques for evaluating fault prediction models," *Empirical Software Engineering*, vol. 13, pp. 561–595, 2008.

[32] A. McCallum, K. Nigam, and L. H. Ungar, "Efficient clustering of high-dimensional data sets with application to reference matching," in *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, ser. KDD, 2000, pp. 169–178.

[33] D. Sculley, "Web-scale k-means clustering," in *Proceedings of the 19th international conference on World wide web*, ser. WWW '10, 2010, pp. 1177–1178.