On the relative value of cross-company and within-company data for defect prediction

**Burak Turhan · Tim Menzies · Ayşe B. Bener ·
Justin Di Stefano**

Abstract We propose a practical defect prediction approach for companies that do not track defect related data. Specifically, we investigate the applicability of cross-company (CC) data for building localized defect predictors using static code features. Firstly, we analyze the conditions, where CC data can be used as is. These conditions turn out to be quite few. Then we apply principles of analogy-based learning (i.e. nearest neighbor (NN) filtering) to CC data, in order to fine tune these models for localization. We compare the performance of these models with that of defect predictors learned from within-company (WC) data. As expected, we observe that defect predictors learned from WC data outperform the ones learned from CC data. However, our analyses also yield defect predictors learned from NN-filtered CC data, with performance close to, but still not better than, WC data. Therefore, we perform a final analysis for determining the minimum number of local defect reports in order to learn WC defect predictors. We demonstrate in this paper that the minimum number of data samples required to build effective defect predictors can be quite small and can be collected quickly within a few months. Hence, for companies with no local defect data, we recommend a two-phase approach that allows them to employ the defect prediction process instantaneously. In phase one, companies should use

B. Turhan (✉) · A. B. Bener
Department of Computer Engineering, Bogazici University, Istanbul, Turkey
e-mail: turhanb@boun.edu.tr

A. B. Bener
e-mail: bener@boun.edu.tr

T. Menzies · J. Di Stefano
Lane Department of Computer Science and Electrical Engineering, Morgantown,
West Virginia, USA

T. Menzies
e-mail: tim@menzies.us

J. Di Stefano
e-mail: jdistefano@ismwv.com

NN-filtered CC data to initiate the defect prediction process and simultaneously start collecting WC (local) data. Once enough WC data is collected (i.e. after a few months), organizations should switch to phase two and use predictors learned from WC data.

1 Introduction

Defect prediction studies usually focus on building models with available local data (i.e. within company predictors). To employ these models, a company should have a data repository, where project metrics and defect information from past projects are stored. However, few companies apply this practice. We suspect that a common reason for not using defect predictors in practice, is the lack of local data repositories. Constructing such a repository requires keeping track of project metrics together with related defect information. When automated tools are not used, manual effort is inevitable to maintain these repositories.

On the other hand, there are public data repositories including projects from companies such as NASA (Boetticher et al. 2007). In this context, we investigate whether these public project data can be helpful for other companies for building localized defect predictors[1], especially for those with limited or no defect data repository.

Defect prediction literature contains many examples where predictors are learned from within company data (Arisholm and Briand 2006a; Bell et al. 2006; Ostrand et al. 2007; Menzies et al. 2007a; Chen et al. 2005; Dekhtyar et al. 2004; Jiang et al. 2007; Nagappan and Ball 2005a). However, there exists no case that attempts to harness cross company data. In this paper, we focus on *binary defect prediction* and perform analyses to check if we can reach a conclusion in favor of either CC or WC data. Specifically, this paper assesses the relative merits of cross-company (CC) vs within-company (WC) data for defect prediction.

We perform a series of analyses on 10 publicly available project data from different companies. Our analyses use the Naive Bayes data miner and are based on static code features.[2]

Our analyses aim at answering the following research questions:

– *Are CC data ever useful for organizations?*:
  Our goal is to identify the conditions under which cross-company data should be preferred to within-company for the purposes of structuring software inspections using outcomes of defect predictors. In the first analysis, we compare the performance of defect predictors learned from WC data to those learned from CC data.

---

[1]Throughout the paper, the following notation is used: a defect predictor (or predictor) means a binary classification method that categorizes software modules as either defective or defect-free; data refers to MxN matrices of raw measurements of N metrics from M software modules; these N metrics are referred to as features.

[2]Therefore, throughout the paper, the term "data" refers to static code features.

– *How can companies filter CC data for local tuning?*:
  CC data includes information from many diverse projects and are heterogeneous compared to WC data. The goal of the second analysis is to select a subset of the available CC data that is similar to WC data and to investigate the effect of data homogeneity on the defect prediction performance. We apply a simple nearest neighbor (NN) filtering to CC data for automatically constructing a locally tuned repository. We use the Euclidean distance between static code features of WC and CC data for measuring similarity and automated selection of neighbors.[3]
– *What is the smallest amount of local data needed for constructing a model?*:
  In the third analysis, we focus on WC models and determine the smallest amount of local data needed for constructing a model. We employ an incremental learning approach to WC data in order determine the number of samples in local repositories for building defect prediction models.
– *Can our results be generalized?*:
  We initially use only NASA projects in our three analyses to answer the above questions. In order to check the external validity of our results, our last analysis replicates all three analyses on projects from a company that has no ties with NASA: specifically, a Turkish company writing software controllers for home appliances.

The contributions of this paper are relevant for both academia and practice. Prior to this paper, no study was performed to investigate the relative merits of using cross company or within company data for constructing defect predictors. In our analysis, in favor of the common belief, we empirically show that within company data is better for defect prediction problem. Further, on the contrary to the common belief, we empirically show that, the required local data for constructing defect predictors can be easily and quickly collected. For our analysis, we have collected data from industry and made it publicly available for the use of other researchers and practitioners. As a practical contribution, our analysis of cross company data and the proposed methodology allow the construction of defect predictors even for companies with no local defect data. In summary, we observe that the defect detectors learned from site A are found to be useful at site B, at least for the datasets we analyzed. Furthermore, it is a result of major significance that other project's data can be used locally (after applying a very simple filter). Nevertheless, there may be some reluctance on the part of programmers and managers to use data collected at other sites. Such caution is to be expected amongst professionals striving to deliver quality products to their clients. In this paper, we suggest that it can be quite beneficial to use data from other sites, particularly when no local data is available. Professional developers should apply our advise cautiously—perhaps via an initial case study on a small part of their system.

This paper is organized as follows: The next section motivates our work with examples of industrial groups that use these detectors. Then, in Section 3, we explain our methodology by describing the data sources, data mining algorithm and the performance evaluation criteria. In Sections 4, 5 and 6, we describe our analyses, results and discussions for the first three research questions stated above.

---

[3]We should carefully note that we do not make use of any conceptual similarities, since our analysis is based on static code features. As to the issue of conceptual connections within the code, we refer the reader to the concept location and cohesion work of Marcus et al. (2008).

In Section 7, we replicate all three analyses in order to check the external validity of our results. Sections 8, 9 and 10, discusses the related work, practical implications and threats to validity respectively. We conclude our work in Section 11.

2 Motivation

2.1 Why Study WC vs CC?

The genesis of this work was several frustrating years working at NASA trying to collect data on new projects. This proved difficult to do, for a variety of reasons (including feuding NASA centers and contractors trying to maintain some control over the information flow to the client). Several times in that process, we asked "Is it necessary to collect new data? Can't we just apply old data?".

In the middle of 2007, we had an opportunity to evaluate CC data usage on a Turkish software development company. That company worked in a highly competitive market and a decrease in their bug rate of even 1% had an economic benefit to them. However, that company had no historical data with which to train its defect detectors. We therefore trained a defect detector on NASA data and applied it to the Turkish software. Much to our surprise, that detector could "adequately" predict errors within Turkish software ("adequate" in the sense that the Turkish client looked at the code that triggered our detectors, found bugs, and was subsequently interested enough to request more studies of this nature).

Based on that small informal experience, we saw:

– The business benefits in using CC data: no need to wait for local data collection;
– The adequacy of the learned detectors;

A single case study is not enough to justify a general principle. Accordingly, we conducted the analyses described in this paper.

2.2 Why Study Defect Predictors?

Defect predictors are subtly different to other quality assurance practices. This section highlights those differences and explains why we elect to study them.

It is insightful to contrast defect prediction from Musa-style software reliability engineering (SRE) (Musa et al. 1987). In standard SRE, there is a track record of post-release failures and the goal is to predict the probability of failure on demand in the next call on the software. One of the major results of the SRE literature is that as the software is developed and bugs are removed, this probability forms a reliability growth curve with predictable properties.

The standard defect prediction task is different to standard SRE. Our task is to predict defective modules during development and before delivery. Such early lifecycle detection is important since, as Boehm reports, repair cost increases exponentially, the longer a bug remains in the system (Boehm and Papaccio 1988). Hence, we learn defect predictors from logs of pre-release defects found by a verification and validation (V&V) team. Such defects may be seen in code from systems that is not yet complete. Hence, many of the features used in this kind of analysis are static code features that can be extracted from as-yet-un-executing code.

The patterns found in pre-release defects or post-release failures can be used in different ways. Reliability growth curves can be used to predict how long before the running software will reach some required reliability of, i.e. one failure in 10,000 calls. Defect predictors, on the other hand, can be used to bias the ordering of modules to be inspected by V&V teams:

– In the case where there is insufficient resources to inspect all code (which is a very common situation in industrial developments), defect predictors can be used to increase the odds that the inspected code will have more defects.
– In the case where all the code is to be inspected, but that inspection process will take weeks to months to complete, defect predictors can be used to increase the chances that defective modules will be inspected earlier. This is useful since it gives the development team earlier notification of what modules require rework, hence giving them more time to complete that rework prior to delivery.

The above two cases are quite common. The authors of this paper are from the United States and Turkey and only began exploring defect detectors when they realized that their user community was making extensive use of these detectors.

Furthermore, these detectors are widely used despite the fact that they are quite "coarse-grained". In the following study, our only knowledge of modules will be that they are defective or defect-free. This work makes no use of other knowledge about a module such as its criticality or the dates on which it failed.

– *Regarding failure occurence logs:* When working with large teams of contractors, it is often difficult to access precise logs on when failures occured since such information is often corporate critical. The NASA coarse-grained information used in this study, for example, took years of careful negotiation to acquire. All the data passed to us was highly sanitized; e.g. this research team could not even access project or module names.
– *Regarding criticality knowledge:* Our goal is to predict defective modules during development and before delivery. Goseva and Hamill (2007) report that pre-delivery perceptions of module criticality can be widely misleading. In their work, they trace errors back to the modules that caused runtime faults. They find that a remarkably small set of modules are implicated in all faults and that set can change, radically, if the input space to the program alters.

For further notes on related work on defect predictors from static code attributes, see Section 8.2.

3 Methodology

3.1 Data

The analyses of this paper use the static code features of 10 projects tabulated in Table 1 and Table 2, which are downloaded from the PROMISE repository.[4] Available static code features are shown in Table 3. An advantage of static code

---

[4]http://promisedata.org/repository.

Table 1 Descriptions of ten software projects used in the paper

Source	Project	Language	Description
NASA	pc1	C++	Flight software for earth orbiting satellite
NASA	kc1	C++	Storage management for ground data
NASA	kc2	C++	Storage management for ground data
NASA	cm1	C++	Spacecraft instrument
NASA	kc3	JAVA	Storage management for ground data
NASA	mw1	C++	A zero gravity experiment related to combustion
SOFTLAB	ar4	C	Embedded controller for white-goods
SOFTLAB	ar3	C	Embedded controller for white-goods
NASA	mc2	C++	Video guidance system
SOFTLAB	ar5	C	Embedded controller for white-goods

The rows labeled "NASA" come from NASA aerospace projects while the rows labeled "SOFT-LAB" come from a Turkish software company writing applications for domestic appliances

features is that they can be quickly and automatically collected from the source code, even if no other information is available.

Our project data are taken from software developed in different geographical locations across North America (NASA) and Turkey (SOFTLAB). Therefore, the static code features that are available for each project vary. While Table 3 shows the features available for each project, Table 4 shows the common features for both sources. While NASA and SOFTLAB are single *sources* of data, there are several projects within each source. For example, NASA is really an umbrella organization used to co-ordinate and fund a large and diverse set of projects:

– The NASA data was collected from across the United States over a a period of five years from numerous NASA contractors working at different geographical centers.
– These projects represent a wide array of projects, including satellite instrumentation, ground control systems and partial flight control modules (i.e. Altitude Control).
– The data sets also represent a wide range of code reuse: some of the projects are 100% new, and some are modifications to previously deployed code.

That is, even if we explore just the NASA data sets, we can still examine issues of cross- vs within- company data use. Nevertheless, using our connections with the

Table 2 Summary of data from ten software projects of Table 1, sorted in order of number of functional units

Source	Project	(# modules) examples	Features	loc	%defective
NASA	pc1	1,109	21	25,924	6.94
NASA	kc1	845	21	42,965	15.45
NASA	kc2	522	21	19,259	20.49
NASA	cm1	498	21	14,763	9.83
NASA	kc3	458	39	7749	9.38
NASA	mw1	403	37	8341	7.69
SOFTLAB	ar4	107	29	9196	18.69
SOFTLAB	ar3	63	29	5624	12.70
NASA	mc2	61	39	6134	32.29
SOFTLAB	ar5	36	29	2732	22.22
				4,102	

Table 3 Static code features available in Table 2 projects

#	Feature	pc1	kc1	kc2	cm1	kc3	mw1	ar4	ar3	mc2	ar5
1	*branchcount*	X	X	X	X	X	X	X	X	X	X
2	*codeandcommentloc*	X	X	X	X	X	X	X	X	X	X
3	*commentloc*	X	X	X	X	X	X	X	X	X	X
4	*cyclomaticcomplexity*	X	X	X	X	X	X	X	X	X	X
5	*designcomplexity*	X	X	X	X	X	X	X	X	X	X
6	*halsteaddifficulty*	X	X	X	X	X	X	X	X	X	X
7	*halsteadeffort*	X	X	X	X	X	X	X	X	X	X
8	*halsteaderror*	X	X	X	X	X	X	X	X	X	X
9	*halsteadlength*	X	X	X	X	X	X	X	X	X	X
10	*halsteadtime*	X	X	X	X	X	X	X	X	X	X
11	*halsteadvolume*	X	X	X	X	X	X	X	X	X	X
12	*totaloperands*	X	X	X	X	X	X	X	X	X	X
13	*totaloperators*	X	X	X	X	X	X	X	X	X	X
14	*uniqueoperands*	X	X	X	X	X	X	X	X	X	X
15	*uniqueoperators*	X	X	X	X	X	X	X	X	X	X
16	*executableloc*	X	X	X	X	X	X	X	X	X	X
17	*totalloc*	X	X	X	X	X	X	X	X	X	X
18	*halsteadcontent*	X	X	X	X	X	X			X	
19	*essentialcomplexity*	X	X	X	X	X	X			X	
20	*halsteadvocabulary*	X	X	X	X			X	X		X
21	*blankloc*	X	X	X	X	X	X	X	X	X	X
22	*callpairs*					X	X	X	X	X	X
23	*conditioncount*					X	X	X	X	X	X
24	*cyclomaticdensity*					X	X	X	X	X	X
25	*decisioncount*					X	X	X	X	X	X
26	*decisiondensity*					X	X	X	X	X	X
27	*halsteadlevel*					X	X	X	X	X	X
28	*multipleconditioncount*					X	X	X	X	X	X
29	*designdensity*					X	X	X	X	X	X
30	*normcyclomaticcomp.*					X	X	X	X	X	X
31	*formalparameters*					X	X	X	X	X	X
32	*modifiedconditioncount*					X	X			X	
33	*maintenanceseverity*					X	X			X	
34	*edgecount*					X	X			X	
35	*nodecount*					X	X			X	
36	*essentialdensity*					X	X			X	
37	*globaldatacomplexity*					X				X	
38	*globaldatadensity*					X				X	
39	*percentcomment*					X	X			X	
40	*numberoflines*					X	X			X	
	Total	21	21	21	21	39	37	29	29	39	29

Turkish software industry, we collected new data sets in the format of Table 3 from a Turkish white- goods manufacturer. The SOFTLAB datasets ({ar3, ar4, ar5}) in Table 1, are the controller software for:

– A washing machine;
– A dishwasher;
– And a refrigerator.

Table 4 Static code features shared by NASA and SOFTLAB projects

#	Feature	NASA shared	All shared
1	*branchcount*	X	X
2	*codeandcommentloc*	X	X
3	*commentloc*	X	X
4	*cyclomaticcomplexity*	X	X
5	*designcomplexity*	X	X
6	*halsteaddifficulty*	X	X
7	*halsteadeffort*	X	X
8	*halsteaderror*	X	X
9	*halsteadlength*	X	X
10	*halsteadtime*	X	X
11	*halsteadvolume*	X	X
12	*totaloperands*	X	X
13	*totaloperators*	X	X
14	*uniqueoperands*	X	X
15	*uniqueoperators*	X	X
16	*executableloc*	X	X
17	*totalloc*	X	X
18	*halsteadcontent*	X	
19	*essentialcomplexity*	X	
	Total	19	17

In summary, 7 datasets are from NASA projects developed at different sites by different teams, hence we treat each of them as if they were from 7 different companies. Remaining 3 datasets are from a Turkish company collected from software for domestic home appliances. Therefore we use 10 projects from 8 different companies.

In all analyses, we used all available features to learn defect predictors. One question is whether it might be better to focus on just some special subset of those features. In previous work (Menzies et al. 2007b), we have found that there is no best single set of features. In fact, this *best* set is highly unstable and changes dramatically from data set to data set. That study is summarized, below. The main message of that study is that it is "best" to never assume a *best* set of features. Rather, give them all to a learner and let it work out which ones are best to use or ignore for a particular application.

Feature subset selection is a technique that explores subsets of the available features. The simplest and fastest subsetting method is to rank attributes from the most informative to least informative. After discretizing numeric data[5] then if F is a set of features, the number of bits required to encode a class after observing a feature is:

$$H(C|F) = -\sum_{f \in F} p(f) \sum_{c \in C} p(c|f) log_2(p(c|f)) \tag{1}$$

The highest ranked feature F_i is the one with the largest *information gain*; i.e the one that most reduces the encoding required for the data *after* using that feature; i.e.

$$InfoGain(F_i) = H(C) - H(C|F_i) \tag{2}$$

---

[5]E.g. given an attribute's minimum and maximum values, replace a particular value n with $(n - min)/((max - min)/10)$. For more on discretization, see Dougherty et al. (1995).

data	N	% pd	% pf	selected attributes (seeFigure 2)	selection method
pc1	100	48	17	3, 35, 37	exhaustive subsetting
mw1	100	52	15	23, 31, 35	iterative subsetting
kc3	100	69	28	16, 24, 26	iterative subsetting
cm1	100	71	27	5, 35, 36	iterative subsetting
pc2	100	72	14	5, 39	iterative subsetting
kc4	100	79	32	3, 13, 31	iterative subsetting
pc3	100	80	35	1, 20, 37	iterative subsetting
pc4	100	98	29	1, 4, 39	iterative subsetting
all	800	71	25		

Fig. 1 Best defect predictors learned in Menzies et al. (2007b). Mean results from Naive Bayes after a 10 repeats of (i) randomize the order of the data; (ii) divide that data into ten 90%:10% splits for training:test. Prior to learning, all numerics where replaced with logarithms. InfoGain was then used to select the best two or three attributes shown in the right-hand column (and if "three" performed as well as "two", then this table shows the results using "two")

where $H(C)$ comes from Eq. 1. In *iterative InfoGain subsetting*, predictors are learned using the $i = 1, 2..., N$-th top-ranked attributes. Subsetting terminates when $i + 1$ attributes perform no better than i. In *exhaustive InfoGain subsetting*, the attributes are first ranked using iterative subsetting. Next, predictors are built using all subsets of the top j ranked attributes. For both iterative and exhaustive subsetting, the process is repeated 10 times using 90% of the data (randomly selected). Iterative subsetting takes time linear on the number of attributes N while exhaustive subsetting takes time 2^j (so it is only practical for small $j \leq N$).

Figure 1 shows the feature subset selection results reported previously (Menzies et al. 2007b). Please note that, that study was run on many of the same NASA data sets used in this study. The right-hand-side column of Fig. 1 reports which attributes used in that study were selected by feature subset selection. In a result consistent with many of the static code measures being highly correlated, note that most of the features were removed and using 3 features was often as good as using many more.

ID	frequency in Figure 1	what	type
1	2	loc_blanks	locs
3	2	call_pairs	misc
4	1	loc_code_and_command	locs
5	2	loc_comments	locs
13	1	edge_count	misc
16	1	loc_executable	locs
20	1	I	H (derived Halstead)
23	1	B	H (derived Halstead)
24	1	L	H (derived Halstead)
26	1	T	H (derived Halstead)
31	2	node_count	misc
35	3	μ_2	h (raw Halstead)
36	1	μ_1	h (raw Halstead)
37	2	number_of_lines	locs
39	2	percent_comments	misc

Fig. 2 Attributes used in Fig. 1

Fig. 3 InfoGain for KC3 attributes. Calculated from Eq. 2. *Lines* show means and *t-bars* show standard deviations after 10 trials on 90% of the training data (randomly selected)
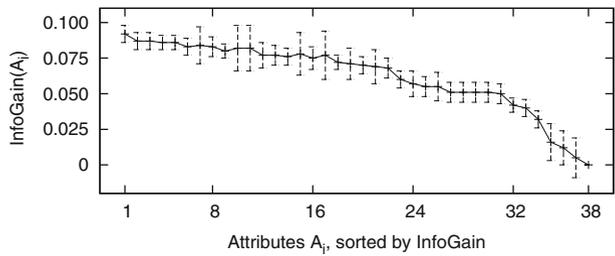
In terms of "which features are best to use?", the key feature of Fig. 1 is that different data sets selected very different "best" features. Figure 2 tallies the number of times each feature was selected: no feature was selected all the time and many of the features were never selected at all. This aspect can be explained by Fig. 3 which shows the InfoGain of all the featues in one of our data sets (KC3). Note how the highest ranked attributes (those on the left-hand-side) offer very similar information. That is, there are no clear winners, so minor changes in the training sample (the 90% sub-sampling used in subsetting or a cross-validation study) can result in the selection of very different "best" attributes.

The pattern of InfoGain values of Fig. 3 (where there are many alternative "best" features) repeats in all our data sets. This pattern explains a prior observation of Shepperd & Ince who found 18 publications where an equal number of studies reporting that the McCabe cyclomatic complexity is the same, is better, or is worse than lines of code in predicting defects (Shepperd and Ince 1994). Figure 3 motivates the following principles:

– Do not seek "best" subsets of static code attributes.
– Rather, seek for learning methods that can combine multiple partial defect indicators like the statistical methods of Naive Bayes.

Therefore, we do not perform feature subset selection, instead we use all available features in our analyses.

3.2 Naive Bayes Classifier

Each project in Table 2 contains information from many modules; the smallest unit of functionality.[6] To learn defect predictors, the project data are augmented with one dependent variable holding boolean values for "defects detected". The data mining task is to find combinations of static code features that predict the boolean value of the dependent variable.

In prior work we have explored a range of data mining methods for defect prediction and found that classifiers based on Bayes theorem yields better performance than rule based methods (i.e. decision trees, oneR), for the Table 2 NASA data[7] (Menzies et al. 2007b). An extensive study by Lessmann et.al also shows that Naive Bayes performs equivalently well with 15 other methods (Lessmann et al. 2008).

---

[6]In other languages, modules may be called "function" or "method".

[7]SOFTLAB data were not available at that time.

Therefore, we find Naive Bayes to be a viable choice as a classifier to use in our analysis.

A Naive Bayes classifier (Duda et al. 1976) is based on Bayes' Theorem. Informally, the theorem says *next* = *old* ∗ *new* i.e. what we'll believe *next* comes from how *new* evidence effects *old* beliefs. More formally:

$$P(H|E) = \frac{P(H)}{P(E)} \prod_i P(E_i|H) \tag{3}$$

i.e. given fragments of evidence E_i and a prior probability for a class $P(H)$, the theorem lets us calculate a posterior probability $P(H|E)$. For example, in our data sets, there are two hypotheses: modules are either defective or not: $H \in$ {*defective*, *non Defective*}. Also, if a particular module has *number of Symbols* = 27 and $LOC = 40$ and was previously classified as "defective" then

$$E_1 : number\ Of\ Symbols = 27$$
$$E_2 : LOC = 40$$
$$H \ : defective$$

When building defect predictors, the prior probability of each class ("defective" or "defect-free") is calculated, given the defect content of each module. So, if a data set has 100 modules and 25 of them are faulty, then:

$$P(defective) = 0.25$$

When testing new data, a module is assigned to the class with the higher posterior probability, calculated from Eq. 3.

Naive Bayes classifiers are called "naive" since they assume independence of each feature. While this assumption simplifies the implementation (only the frequency counts are required for each feature), it is possible that correlated events are missed by this "naive" approach. Domingos and Pazzani show theoretically that the independence assumption is a problem in a vanishingly small percent of cases (Domingos and Pazzani 1997). This explains the repeated empirical result that, on average, Naive Bayes classifiers perform as well as other seemingly more sophisticated schemes.[8]

Equation 3 offers a simple method for handling missing values. Generating a posterior probability means of tuning a prior probability to new evidence. If that evidence is missing, then no tuning is needed. In this case Eq. 3 sets $P(E_i|H) = 1$ which, in effect, makes no change to $P(H)$.

When estimating the prior probability of hypothesis H, it is common practice (Yang and Webb 2003) to use an *M-estimate* as follows. Given that the total number of hypothesis is C, the total number of training instances is I, and $N(H)$ is the frequency the hypothesis H within I, then

$$P(H) = \frac{N(H) + m}{I + m \cdot C} \tag{4}$$

---

[8]Details of this issue are out of the scope of this paper. For more, please see Table 1 in (Domingos and Pazzani 1997).

Here m is a small non-zero constant (often, $m = 1$). Three special cases of Eq. 4 are:

– For high frequency hypothesis in large training sets, $N(H)$ and I are much larger than m and $m \cdot C$, so Eq. 4 simplifies to $P(H) = \frac{N(H)}{I}$, as one might expect.
– For low frequency classes in large training sets, $N(H)$ is small, I is large, and the prior probability for a rare class is never less than $\frac{1}{I}$; i.e. the inverse of the number of instances. If this were not true, rare classes would never appear in predictions.
– For very small data sets, I is small and $N(H)$ is even smaller. In this case, Eq. 4 approaches the inverse of the number of classes; i.e. $\frac{1}{C}$. This is a useful approximation when learning from very small data sets when all the data relating to a certain class has not yet been seen.

The prior probability calculated in Eq. 4 is a useful lower bound for $P(E_i|H)$. If some value v is seen $N(f = v|H)$ times in feature f's observations for hypothesis H, then

$$P(E_i|H) = \frac{N(f = v|H) + l \cdot P(H)}{N(H) + l} \qquad (5)$$

Here, l is the *L-estimate* and is set to a small constant (Yang and Webb (2003) recommend $l = 2$). Two special cases of are:

– A common situation is when there are many examples of an hypothesis and numerous observations have been made for a particular value. In that situation, $N(H)$ and $N(f = v|H)$ are large and Eq. 5 approaches $\frac{N(f=v|H)}{N(H)}$, as one might expect.
– In the case of very little evidence for a rare hypothesis, $N(f = v|H)$ and $N(H)$ are small and Eq. 5 approaches $\frac{l \cdot P(H)}{l}$; i.e. the default frequency of an observation in a hypothesis is a fraction of the probability of that hypothesis. This is a useful approximation when very little data is available.

For numeric features it is common practice for Naive Bayes classifiers to use the Gaussian probability density function (Witten and Frank 2005):

$$g(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \qquad (6)$$

where $\{\mu, \sigma\}$ are the feature's {mean,standard deviation}, respectively. To be precise, the probability of a continuous feature having exactly the value x is zero, but the probability that it lies within a small region, say $x \pm \epsilon/2$, is $\epsilon \times g(x)$. Since ϵ is a constant that weighs across all possibilities, it cancels out and needs not be computed.

All the static code features of Table 3 are numeric and are highly skewed. Therefore, we replace all numeric values with a "log-filter", i.e. N with $ln(N)$. This spreads out skewed curves more evenly across the space from the minimum to maximum values (to avoid numerical errors with $ln(0)$, all numbers under 0.000001 are replaced with $ln(0.000001)$). This "spreading" can significantly improve the effectiveness of data mining, since the distribution of log-filtered feature values fits better to the normal distribution assumption (Menzies et al. 2007b).

3.3 Performance Evaluation

Data mining effectiveness was measured using *probability of detection* (*pd*), *probability of false alarm* (*pf*) and *balance* (*bal*) (Menzies et al. 2007a, b). If $\{A, B, C, D\}$

are the true negatives, false negatives, false positives, and true positives (respectively) found by a defect predictor, then:

$$pd = recall = D/(B + D) \tag{7}$$

$$pf = C/(A + C) \tag{8}$$

$$bal = balance = 1 - \frac{\sqrt{(0 - pf)^2 + (1 - pd)^2}}{\sqrt{2}} \tag{9}$$

All these values range zero to one. Better and *larger* balances fall *closer* to the desired zone of no false alarms ($pf = 0$) and 100% detection ($pd = 1$).

Other measures such as *accuracy* and *precision* were not used since, as shown in Table 2, the percent of defective examples in our projects was usually very small (median value around 13%). Accuracy and precision are poor indicators of performance for data where the target class (i.e. defective) is so rare (for more on this issue, see Menzies et al. 2007a, b).

The results were visualized using *quartile charts*. To generate these charts, the performance measures for an analysis are sorted to isolate the median and the lower and upper quartile of numbers. For example:

$$\{\overbrace{4, 7, 15}^{q1}, 20, 31, \overbrace{40}^{median}, 52, 64, \overbrace{70, 81, 90}^{q4}\}$$

In our quartile charts, the upper and lower quartiles are marked with black lines; the median is marked with a black dot; and vertical bars are added to mark the 50% percentile value. The above numbers would therefore be drawn as follows:

$$0\% | \!\!-\!\! \quad \bullet \; | \quad\quad -\!\!- \quad |100\%$$

The Mann-Whitney U test (Mann and Whitney 1947) was used to test for statistical difference between results. This non-parametric test replaces performance measure values (i.e. pd, pf, bal) with their rank inside the population of all sorted values. Such non-parametric tests are recommended in data mining since many of the performance distributions are non-Gaussian (Demsar 2006).

4 Analysis #1: Are CC Data Ever Useful for Organizations?

4.1 Design

Our goal is to determine whether using cross company data is beneficial for constructing defect predictors and to identify the conditions under which cross-company data should be preferred to within-company data. Our first WC-vs-CC analysis follows the pseudo code given in Table 5, for all 7 NASA projects of Table 1. For each project,

Table 5 Pseudocode for Analysis 1

```
DATA = [PC1, KC1, KC2, CM1, KC3, MW1, MC2]       // all available data
LEARNER = [Naive Bayes]                          //  defect predictor

C_FEATURES <- Find common features IN DATA
FOR EACH data IN DATA
        data = Select C_FEATURES in data         // use common features
END
REPEAT 20 TIMES
        FOR EACH data in DATA
                CC_TRAIN = DATA - data           // cross company training data
                WC_TRAIN = random 90% of data    // within company training data
                TEST = data - WC_TRAIN           // shared test data

                //construct predictor from CC data
                CC_PREDICTOR = Train LEARNER with CC_TRAIN
                // construct predictor from WC data
                WC_PREDICTOR = Train LEARNER with WC_TRAIN
                //Evaluate both predictors on the same test data
                [cc_pd, cc_pf, cc_bal] = CC_PREDICTOR on TEST
                [wc_pd, wc_pf, wc_bal] = WC_PREDICTOR on TEST
        END
END
```

test sets were built from 10% of the data, selected at random. Defect predictors were then learned from:

– CC data: all data from the other 6 projects.
– WC data: remaining 90% data of that project;

Most of the Table 1 data come from systems written in "C/C++" but at least one of the systems was written in JAVA. For cross-company data, an industrial practitioner may not have access to detailed meta-knowledge (e.g. whether it was developed in "C" or JAVA). They may only be aware that data, from an unknown source, are available for download from a certain url. To replicate that scenario, we will make no use of our meta-knowledge about Table 1.

In order to control for *order effects* (where the learned theory is unduly affected by the order of the examples) our procedure was repeated 20 times, randomizing the order of data in each project each time. In all, we ran 280 tests to compare WC-vs-CC:

$$(2 \text{ data sources}) * (20 \text{ randomized orderings}) * (7 \text{ projects})$$

The projects' data come from different sources and, hence, have different features. For this analysis, only the features that are common in all NASA projects are used (a total of 19 features). These features are marked in "NASA Shared" column of Table 4.

4.2 Results from Analysis #1

Figure 4 shows the $\{pd, pf\}$ quartile charts for CC vs WC data averaged over seven NASA projects. The pattern is very clear: CC data dramatically increases *both* the
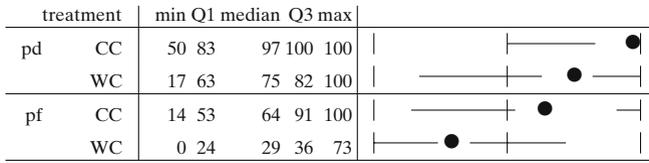
treatment		min	Q1	median	Q3	max	
pd	CC	50	83	97	100	100	
	WC	17	63	75	82	100	
pf	CC	14	53	64	91	100	
	WC	0	24	29	36	73	

Fig. 4 Analysis #1 results averaged over seven NASA tables. Numeric results on *left*; quartile charts on *right*. *Q1* and *Q3* denote the 25% and 75% percentile points (respectively). The upper quartile of the first row is not visible since it runs from 100% to 100%; i.e. it has zero length

probability of detection and the probability of false alarms. The *pd* results are particularly striking.

For cross-company data:

– 50% of the pd values are at or above 97%
– 75% of the pd values are at or above 83%;
– And all the pd values are at or over 50%.

To the best of our knowledge, Fig. 4 are the largest *pd* values ever reported from these data. However, these very high *pd* values come at some considerable cost. We observe in Fig. 4 that the median false alarm rate has changed from 29% (with WC) to 64% (with CC) and the maximum *pf* rate now reaches 100%. Note that a 100% *pf* rate means that all defect-free modules are classified as defective, which yields inspection of all these modules unnecessarily and contradicts with the purpose of defect prediction. However, it is not right to assess the general behavior of the CC defect predictors with such an extreme case. We mention this issue in order to clarify that high false alarm rates may be prohibitive in the practical application of defect predictors.

We explain these increases in *pd, pf* with the *extraneous factors* in CC data. More specifically, using a large training set (e.g. seven projects in Table 2) informs of not only the sources of errors, but also numerous irrelevancies. For example, the defect characteristics of software modules with different complexity levels or sizes may differ (Koru et al. 2008). In this context, a complicated search algorithm's metrics are irrelevant to the defective behavior of a simple *sum* function. Since there are few modules with extreme characteristics (i.e. complexity, size) in a single project, their effect on the overall model are limited. However, when data from multiple companies are combined, the number of these extreme cases, and hence their cumulative effect on the overall model increase significantly. Therefore, factors such as programming language constructs (i.e. object oriented vs. procedural) and project specific requirements (i.e. availability, speed) that have impacts on the module characteristics can be considered as extraneous factors.

Table 6 Summary of U-test results (95% confidence): moving from WC to CC

Group	$WC \overset{pd}{\to} CC$	$WC \overset{pf}{\to} CC$	Tables	$|Tables|$
a	Increased	Increased	CM1 KC1 KC2 MC2 MW1 PC1	6
b	Same	Same	KC3	1

For all projects' results, see Fig. 5

Table (cm1)

treatment		min	Q1	median	Q3	max
pd	CC	80	100	100	100	100
	WC	40	60	80	100	100
pf	CC	87	91	96	96	98
	WC	24	27	33	38	47

Table (kc1)

treatment		min	Q1	median	Q3	max
pd	CC	82	88	94	94	100
	WC	64	73	82	85	97
pf	CC	47	49	51	53	57
	WC	27	34	36	38	40

Table (kc2)

treatment		min	Q1	median	Q3	max
pd	CC	82	91	91	100	100
	WC	55	73	82	91	100
pf	CC	57	62	64	74	81
	WC	14	24	31	33	45

Table (kc3)

treatment		min	Q1	median	Q3	max
pd	CC	60	80	80	80	100
	WC	40	60	80	80	100
pf	CC	14	19	24	31	38
	WC	10	17	21	26	36

Table (mc2)

treatment		min	Q1	median	Q3	max
pd	CC	50	67	83	100	100
	WC	17	33	67	67	83
pf	CC	55	64	73	73	100
	WC	0	27	36	45	73

Table (mw1)

treatment		min	Q1	median	Q3	max
pd	CC	75	75	100	100	100
	WC	25	50	75	75	100
pf	CC	50	55	63	66	82
	WC	13	18	21	29	37

Table (pc1)

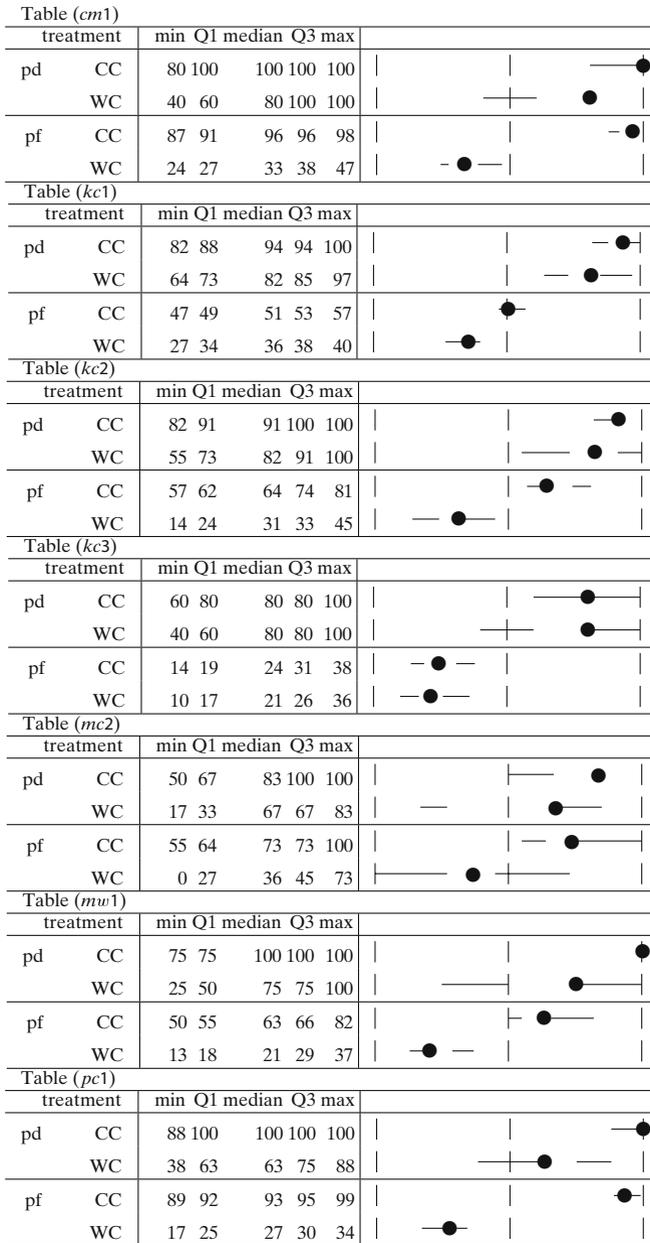| treatment		min	Q1	median	Q3	max
pd	CC	88	100	100	100	100
	WC	38	63	63	75	88
pf	CC	89	92	93	95	99
	WC	17	25	27	30	34

Fig. 5 Project-wise Analysis #1 results for NASA projects

Hence, large training sets increase the error detection rates (i.e. *pd*) since there are more known sources of errors. However, they also increase the probability of false alarms (*pf*) since there are more *extraneous factors* introduced to the analysis. We will test the validity of this claim in the next analysis.

4.3 Checking the Analysis #1 Results

Once a *general result* is defined (e.g. defect predictors learned from CC projects dramatically increase both *pf* and *pd*), it is good practice to check for specific exceptions to that pattern. Table 6 shows a summary of results when U tests with $\alpha = 0.05$ were applied to test results from each of the 7 projects *in isolation* and Fig. 5 shows the {*pd*, *pf*} quartile charts for Analysis #1 for each NASA project:

– For six projects, the general result holds (i.e. both (*pd*, *pf*) increases if defect predictors learned from CC projects are used rather than defect predictors learned from WC projects. See group *a* in Table 6).
– For one project, there is no difference in the performances of the defect predictors learned from CC and WC projects (see group *b* in Table 6).

4.4 Discussion of Analysis #1

When practitioners use defect predictors with high false alarm rates (e.g.the 64% reported above), they must allocate a large portion of their debugging budget to the unfruitful exploration of erroneous alarms.

In our industrial work, we have sometimes seen several situations where detectors with high false alarms are useful:

– *When the cost of missing the target is prohibitively expensive.* In mission critical or security applications, the goal of 100% detection may be demanded in all situations, regardless of the cost of chasing false alarms.
– *When only a small fraction of the data is returned.* Hayes, Dekhtyar, & Sundaram call this fraction *selectivity* and offer an extensive discussion of the merits of this measure (Hayes et al. 2006).
– *When there is little or no cost in checking false alarms.*

Having said that, we have shown these results to five of our user groups in the United States and Turkey and none of them could accept false alarm rates as high as 64%. Therefore, the conditions under which the benefits of CC data (high probabilities of detection) outweigh their high costs (high false alarm rates) are not common.

In summary, for most software applications, very high *pf* rates like the CC results of Fig. 4 make the predictors impractical to use.

5 Analysis #2: How Can Companies Filter CC Data for Local Tuning?

The results of the first analysis restricts the use of CC data to a limited domain (i.e. mission critical) and may look discouraging at first glance. In the first analysis we explained our observations with the *extraneous factors*, that the results are affected by the irrelevant factors in CC data. In this section we hypothesize this claim and test for its validity.

Table 7 Pseudocode for Analysis 2

```
DATA = [PC1, KC1, KC2, CM1, KC3, MW1, MC2]
LEARNER = [Naive Bayes]

C_FEATURES <- Find common features IN DATA
FOR EACH data IN DATA
        data = Select C_FEATURES in data
END
REPEAT 100 TIMES
        FOR EACH data in DATA
                WC_TRAIN = random 90% of data
                TEST = data - WC_TRAIN
                CC_TRAIN = DATA - data

        //NN-Filtering: Select k=10 neighbors for each validation data

                FOR EACH test IN TEST
                        dist = L2_DISTANCE{test, CC_TRAIN}
                        NNCC_TRAIN <- 10 Samples in CC_TRAIN with min{dist}
                END

        //Use only unique samples from the union of all selected neighbors

                NNCC_TRAIN = UNIQUE(NNCC_TRAIN)

                NNCC_PREDICTOR = Train LEARNER with NNCC_TRAIN
                CC_PREDICTOR = Train LEARNER with CC_TRAIN
                WC_PREDICTOR = Train LEARNER with WC_TRAIN
                [nncc_pd, nncc_pf, nncc_bal] = NNCC_PREDICTOR on TEST
                [cc_pd, cc_pf, cc_bal] = CC_PREDICTOR on TEST
                [wc_pd, wc_pf, wc_bal] = WC_PREDICTOR on TEST
        END
END
```
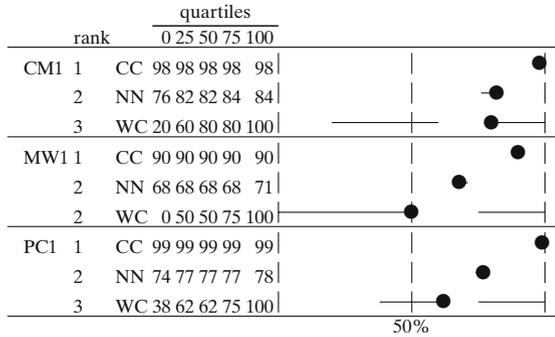
5.1 Design

In this analysis, we try to construct more homogeneous *defect* datasets from CC data in an automated manner. For this purpose we use a simple filtering method (i.e. NN) which is to be described next. The analysis design is given in Table 7. For this analysis, we use all common features available in NASA projects which is a total of 19 features. These features are marked in NASA Shared column of Table 4.

5.1.1 Nearest Neighbor (NN) Filtering

Our idea behind filtering is to collect similar instances together in order to construct a learning set that is homogeneous with the validation set. More formally, we try to introduce a bias in the model by using training data that are similar to the validation data characteristics. While a bias is not desired in general, it is what we seek on purpose, since in our case we can control the bias against removing the noise in CC data (i.e. extraneous factors). We simply use the k-Nearest Neighbor (k-NN) method to measure the similarity between the validation set and the training *candidates*. The similarity measure is the Euclidean distance between the static code features of validation and training candidate sets. As mentioned before, we do not perform any feature selection and use all available common features for filtering. The expected outcome of the filtering part is to obtain a subset of available CC data that shows similar characteristics to the local code culture.

Fig. 6 Analysis #2 PD results where $NN_{pd} \geq WC_{pd}$. Rankings computed via Mann-Whitney (95% confidence) comparing each row to all other rows

	rank		quartiles 0 25 50 75 100	
CM1	1	CC	98 98 98 98 98	
	2	NN	76 82 82 84 84	
	3	WC	20 60 80 80 100	
MW1	1	CC	90 90 90 90 90	
	2	NN	68 68 68 68 71	
	2	WC	0 50 50 75 100	
PC1	1	CC	99 99 99 99 99	
	2	NN	74 77 77 77 78	
	3	WC	38 62 62 75 100	

50%

The details of NN filter are as follows: We calculate the pairwise Euclidean distances between the validation set and the *candidate* training set (i.e. all CC data). Let N be the number of validation set size. For each validation data, we pick its $k = 10$ nearest neighbors from candidate training set.[9] Then we come up with a total of $10 \times N$ similar data points (i.e. module feature vectors). These $10 \times N$ samples may not be unique (i.e. a single data sample can be a nearest neighbor of many data samples in the validation set). Using only unique ones, we form the training set and use a random 90% of it for training a predictor.

Please note that we do not use the class information (i.e. a module is defective or defect-free) while measuring similarity and selecting neighbors. This corresponds to a real life case, where the development of some modules are completed and they are ready for testing: there is no defect related information available, however static code features are collected with automated tools.
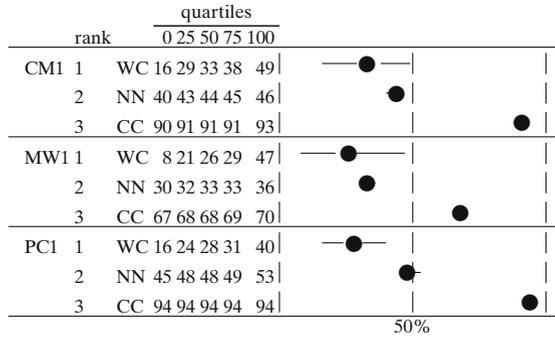
5.2 Results

If the results of Analysis #1 are due to the extraneous factors in CC data (as we suspect), then we would expect lower pf's in NN results than CC results. Therefore, for pd, we define the null hypothesis as:

$$H_0 : NN_{pf} \geq CC_{pf}$$

H_0 is rejected by the U test with $\alpha = 0.05$. Therefore, using NN filtered CC data significantly decreases the false alarms compared to CC data. Yet, we observe that pd's have also decreased. However, false alarm rates are more dramatically decreased than detection rates as seen in Figs. 6, 7, 8 and 9. For example, in $CM1$ project, the median false alarm rate decreases nearly one third, from 91% to 33%, whereas the median detection rate slightly decreases from 94% to 82%. In all cases, predictors learned from NN data dramatically reduce the high false alarm rates

---

[9]Caveat: We did not optimize the value of k for each project. We simply used a constant $k = 10$. We consider it as a future work to dynamically set the value of k for a given project (Baker 2007).

Fig. 7 Analysis #2 PD results where $NN_{pd} < WC_{pd}$

	rank		quartiles 0 25 50 75 100
KC1	1	CC 94 94 94 94 94	
	2	WC 64 76 82 85 94	
	3	NN 60 64 65 66 69	
KC2	1	CC 94 94 94 94 94	
	2	WC 45 73 82 91 100	
	2	NN 77 78 79 79 80	
KC3	1	CC 81 81 81 84 84	
	2	WC 20 60 80 100 100	
	3	NN 60 63 65 67 70	
MC2	1	CC 83 83 83 83 85	
	2	WC 17 50 67 83 100	
	3	NN 56 56 56 56 58	

50%

associated with the use of cross-company data. Often that reduction halves the false alarm rate. For example, in MW1, the median false alarm rate drops from 68% (CC) to 33% (NN).

Showing that defect predictors learned from NN-filtered CC data are significantly better than the ones learned from CC data, the ideal result would be that models based on NN-filtered can be used as a surrogate for WC data based models. This, in turn, would mean that developers could avoid the tedious and expensive work of local data collection. We now investigate the relation between the NN-filtered CC data and the WC data based defect predictors.

If defect predictors learned from NN-filtered CC data out-performed the ones learned from WC data then we would expect to notice two observations:

– *Observation1:* NN would have *pd* values above or equal to WC's *pd*. The examples displaying *Observation*1 are shown in Fig. 6.
– *Observation2:* NN would have *pf* values below or equal to WC's *pf*. The examples displaying *Observation*2 are shown in Fig. 8.

Fig. 8 Analysis s #2 PF results where $NN_{pf} \leq WC_{pf}$

	rank		quartiles 0 25 50 75 100
KC1	1	NN 22 23 24 25 27	
	2	WC 26 32 35 37 43	
	3	CC 59 60 60 60 60	
KC2	1	NN 24 25 25 25 27	
	1	WC 10 21 26 31 40	
	2	CC 67 67 67 67 67	
KC3	1	NN 17 18 18 19 20	
	2	WC 7 17 21 26 31	
	3	CC 26 27 27 27 27	
MC2	1	NN 29 30 31 32 35	
	2	WC 0 27 36 45 73	
	3	CC 71 71 71 71 71	

50%

Fig. 9 Analysis #2 PF results where $NN_{pf} > WC_{pf}$

	rank		quartiles 0 25 50 75 100	
CM1	1	WC	16 29 33 38	49
	2	NN	40 43 44 45	46
	3	CC	90 91 91 91	93
MW1	1	WC	8 21 26 29	47
	2	NN	30 32 33 33	36
	3	CC	67 68 68 69	70
PC1	1	WC	16 24 28 31	40
	2	NN	45 48 48 49	53
	3	CC	94 94 94 94	94

50%

Please note that the conjunction of *Observation*1 and *Observation*2 is uncommon. In fact, our results suggest that *Observation*1 and *Observation*2 are somewhat mutually exclusive:

– As shown in Figs. 6 and 9, the examples where *NN* increases the probability of detection are also those where it increases the probability of false alarms.

Hence, we cannot recommend NN as a replacement for WC. Nevertheless, if local WC data are not available, then we would recommend processing foreign CC data with NN.

5.3 Discussions

In Analysis #1 we have used random samples of CC data and observed that the false alarm rates substantially increased compared to the WC models. Our new analysis shows that NN filtering CC data removes the increased false alarm rates. Now we argue that, using NN filtering instead of using all available CC data, helps choosing training examples that are similar to problem at hand. Thus, the irrelevant information in non-similar examples is avoided. However, this also removes the rich sample base and yields a slight decrease in detection rates. Mann-Whitney tests reveal that NN filtering is

– far better than random sampling cross company data,
– and still worse than using within company data.

The performances of defect predictors based on the NN-filtered CC vs. WC data do not give necessary empirical evidence to make a strong conclusion. Sometimes NN data based models may perform better than WC data based models. A possible reason may be hidden in the processes that are implemented those projects. Perhaps, a group of new developers were working together for the first time and corresponding WC data included more heterogeneity, which is reduced by NN. Perhaps the development methodology changed during the project, producing a different code

culture. However, we do not have access to the internals of these projects that allows a discussion of these observations.

6 Analysis #3: What is the Smallest Amount of Local Data Needed for Constructing a Model?

Our results of Analyses #1 and #2 reveal that WC data models are better if data are available. In this section, we will show that defect predictors can be learned from very small samples of WC data.

6.1 Design

An important aspect of the Analyses #1 and #2 results is that defect predictors were learned using only a handful of defective modules. For example, consider a 90%/10% train/test split on $pc1$ with 1,109 modules, only 6.94% of which are defective. On average, the training set will only contain $1109 * 0.9 * 6.94/100 = 69$ defective modules. Despite this, $pc1$ yields an adequate median $\{pd, pf\}$ results of $\{63, 27\}\%$.

Analysis #3 was therefore designed to check the smallest amount of data needed to learn defect predictors. The design is given in Table 8. Analysis #3 is essentially the same as the first analysis, but without the cross-company study. Instead, analysis #3 took the 7 NASA projects of Table 1 and learned predictors using:

– reduced WC data: a randomly selected subset of up to 90% of each project data.

After randomizing the order of the data, training sets were built using just the first 100, 200, 300, . . . data samples in the project. After training the defect predictor, its performance is tested on the remaining data samples not used in training.

Analysis #1 only used the features found in all NASA projects. For this analysis, we imposed no such restrictions and used whatever features were available in each data set.

Table 8 Pseudocode for Analysis 3

```
DATA = [PC1, KC1, KC2, CM1, KC3, MW1, MC2]
LEARNER = [Naive Bayes]

REPEAT 100 TIMES
 FOR EACH data IN DATA
    WC_TRAIN = random 90% of data
        TEST = data - WC_TRAIN
        FOR i IN {100, 200, 300, ...}
            WC_TRAIN_INCREMENTAL <- Random i Examples from WC_TRAIN
            WC_INC_PREDICTOR = Train LEARNER with WC_TRAIN_INCREMENTAL
                [iwc_pd, iwc_pf, iwc_bal] = WC_INC_PREDICTOR on TEST
        END
  END
END
```

6.2 Results from Analysis #3

Equation 9 defined "balance" to be a combination of $\{pd, pf\}$ that decreases if pd decreases or pf increases. As shown in Fig. 10, there was very little change in balanced performance after learning from 100,200,300,... examples. Indeed, there is some evidence that learning from larger training sets had detrimental effects: the more training data, the larger the variance in the performance of the learned predictor. Observe how, in $kc1$ and $pc1$, as the training set size increases (moving right along the x-axis) the dots showing the balance performance start spreading out. We conjecture that this effect is due to the presence of occasional larger outliers in the training data (the probability of discovering these increases as the training set grows inside and, if the learner trains on them, then the resulting theory has more variance.)
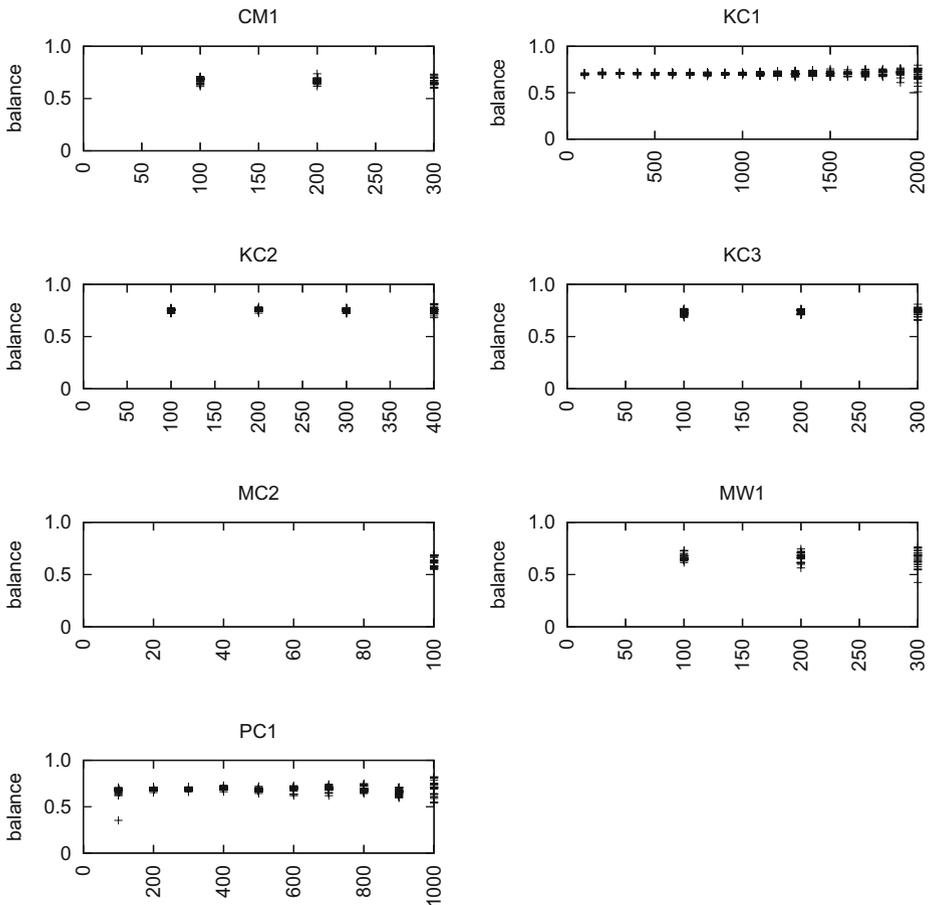

Fig. 10 Results from Analysis #3. Training set size grows in units of 100 examples, moving left to right over the x-axis. The MC2 results only appear at the maximum x-value since MC2 has less than 200 examples

The Mann-Whitney U test was applied to check the visual trends seen in Fig. 10. For each project, all results from training sets of size 100,200,300... were compared to all other results from the same project. The issue was "how much data are enough?" i.e. what is the *min*imum training set size that never lost to other training set of a larger size. Usually, that *min* value was quite small:

–   In five tables {$cm1$, $kc2$, $kc3$, $mc2$, $mw1$}, $min = 100$;
–   In {$kc1$, $pc1$}, $min = \{200, 300\}$ instances, respectively.

We explain the analysis #3 results as follows. This analysis used simplistic static code features such as lines of code, number of unique symbols in the module, etc. Such simplistic static code features are hardly a complete characterization of the internals of a function. We would characterize such static code features as having *limited information content* (Menzies et al. 2008). Limited content is soon exhausted by repeated sampling. Hence, such simple features reveal all they can reveal after a small sample.

6.3 Checking the Analysis #3 Results

There is also some evidence that the results of Analysis #3 (that performance improvements stop after a few hundred examples) have been seen previously in the data mining literature. To the best of our knowledge, this is the first report of this effect in the defect prediction literature.

–   In their discussion on how to best handle numeric features, Langley and John offer plots of the accuracy of Naive Bayes classifiers after learning on 10,20,40,..200 examples. In those plots, there is little change in performance after 100 instances (John and Langley 1995).
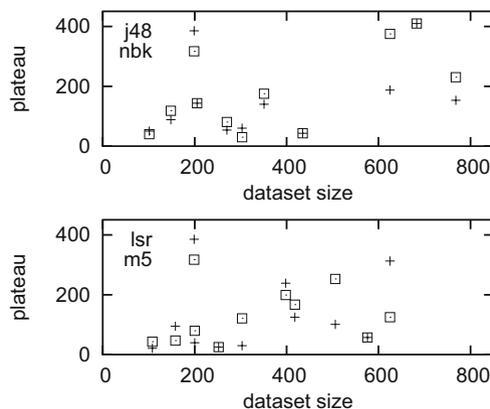

Fig. 11 Y-axis shows plateau point after learning from data sets that have up to *X* examples (from Orrego 2004). The *left plot* shows results from using Naive Bayes (nbk) or a decision tree learner (j48) (Quinlan 1992a) to predict for discrete classes. *Right plot* shows results from using linear regression (lsr) or model trees (m5) (Quinlan 1992b) to learn predictors for continuous classes. In this study, data sets were drawn from the UC Irvine data repository (Blake and Merz 1998)

Table 9 An estimate of the effort required to build and test 100 modules

100 modules may take as little as two to four person months to construct. This estimate was
generated as follows:

– In the *cm*1 data base, the median module size is 17 lines. 100 randomly selected modules would
  therefore use 1700 LOC.
– To generate an effort estimate for these modules, we used the on-line COCOMO (Boehm 2000)
  effort estimator (http://sunset.usc.edu/research/COCOMOII/expert_cocomo/expert_cocomo2000.
  html). Estimates were generated assuming 1700 LOC and the required reliability varying from
  very low to very high.
– The resulting estimates ranged from between 2.4 and 3.7 person months to build and test those
  modules.

– Orrego (2004) applied four data miners (including Naive Bayes) to 20 data sets to
  find the *plateau point*: i.e. the point after which there was little net change in the
  performance of the data miner. To find the plateau point, Oreggo used t-tests to
  compare the results of learning from Y or $Y + \Delta$ examples. If, in a 10-way cross-
  validation, there was no statistical difference between Y and $Y + \Delta$, the plateau
  point was set to Y. As shown in Fig. 11, many of those plateaus were found at
  $Y \leq 100$ and most were found at $Y \leq 200$. Please note that these plateau sizes
  are consistent with the results of Analysis #3.

6.4 Discussion of Analysis #3

In the majority of cases, predictors learned from as little as one hundred examples
perform as well as predictors learned from many more examples. This suggests that
the effort associated with learning defect predictors from within-company data may
not be overly large. For example, Table 9 estimates that the effort required to
build and test 100 modules may be as little as 2.4 to 3.7 person months. However,
practitioners should use this approach cautiously. The populations of one hundred
examples in our experiments are randomly selected from completed projects *with*
stratification. Therefore, in practice, any one hundred sample may not necessarily
reflect the company characteristics and constructing this initial set may take longer
than expected.

7 Replication: Can Our Results be Generalized?

Analyses # 1 to # 3 were based on NASA projects. For the external validity of the
conclusions of those Analyses, we replicate the same analyses on SOFTLAB projects
of Table 1.

   For each SOFTLAB project, we follow the same procedure as in Analyses # 1 and
#2; i.e. 10% of the rows of each data set are selected at random for constructing test
sets. Then three different types of defect predictors are constructed. First type are
defect predictors trained with cross company data (i.e. all 7 NASA projects). Second
type of defect predictors are trained with within company data (i.e. random 90%
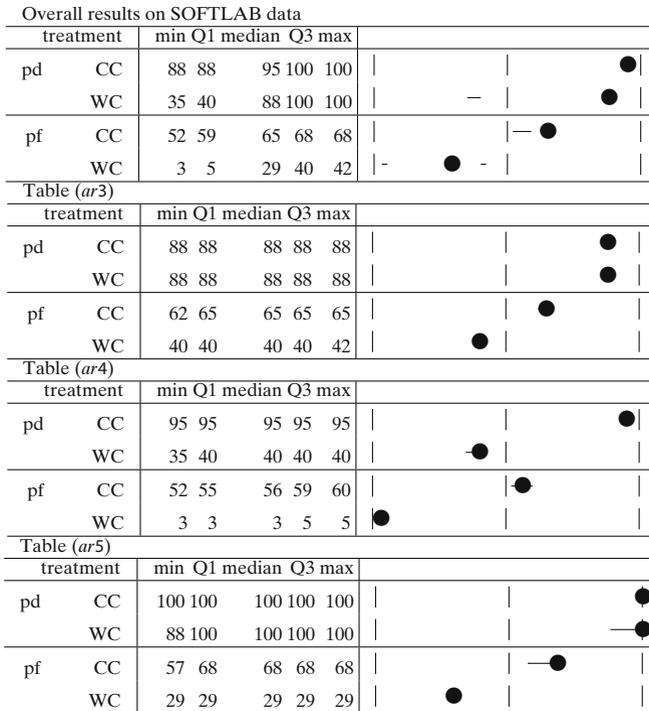
Overall results on SOFTLAB data

treatment		min	Q1	median	Q3	max	
pd	CC	88	88	95	100	100	
	WC	35	40	88	100	100	
pf	CC	52	59	65	68	68	
	WC	3	5	29	40	42	

Table (ar3)

treatment		min	Q1	median	Q3	max	
pd	CC	88	88	88	88	88	
	WC	88	88	88	88	88	
pf	CC	62	65	65	65	65	
	WC	40	40	40	40	42	

Table (ar4)

treatment		min	Q1	median	Q3	max	
pd	CC	95	95	95	95	95	
	WC	35	40	40	40	40	
pf	CC	52	55	56	59	60	
	WC	3	3	3	5	5	

Table (ar5)

treatment		min	Q1	median	Q3	max	
pd	CC	100	100	100	100	100	
	WC	88	100	100	100	100	
pf	CC	57	68	68	68	68	
	WC	29	29	29	29	29	

Fig. 12 Analysis #1 results for the SOFTLAB projects. Overall and individual results are shown respectively

rows of remaining SOFTLAB projects[10]). Finally, the third type are defect predictors trained with nearest neighbor filtered cross company data (i.e. *similar* rows from 7 NASA tables).
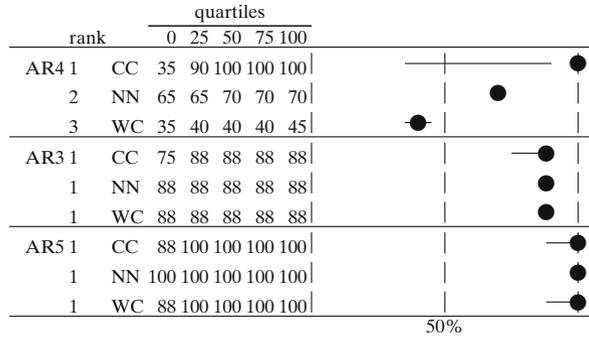
The SOFTLAB projects include 29 static code features, 17 of which are common with the NASA projects. In order to simplify the comparison between these new projects and Analyses #1 and #2, we used only these shared attributes in our CC analyses. On the other hand we use all available features in WC analyses for SOFTLAB projects. In the following external validity analysis, we treated each NASA project as cross- company data for SOFTLAB projects.

Figure 12 shows the results:

– The *pd* values for CC data increase compared to WC data with the cost of increased *pf*.
– CC data shifts {Q1, median} of *pf* from {5, 29} to {59, 65}.
– For CC data:

---

[10]In order to reflect the use in practice, we do not use the remaining 90% of the same project for training, we rather use a random 90% of data from other projects. Please note that all WC analysis in this paper reflects within-company, not within project simulations. Since SOFTLAB data are collected from a single company, learning a predictor on some projects and to test it on a different one does not violate within company simulation.

Fig. 13 Analysis #2 PD results for the SOFTLAB projects where $NN_{pd} \geq WC_{pd}$. Rankings computed via Mann-Whitney (95% confidence) comparing each row to all other rows
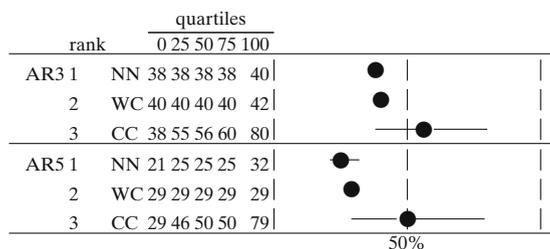
	rank		quartiles 0 25 50 75 100					
AR4	1	CC	35	90	100	100	100	
	2	NN	65	65	70	70	70	
	3	WC	35	40	40	40	45	
AR3	1	CC	75	88	88	88	88	
	1	NN	88	88	88	88	88	
	1	WC	88	88	88	88	88	
AR5	1	CC	88	100	100	100	100	
	1	NN	100	100	100	100	100	
	1	WC	88	100	100	100	100	

50%

- 25% of the *pd* values are at 100%.
- 50% of the *pd* values are above 95%
- And all the *pd* values are at or over 88%.

These results also provide evidence for the validity of our conclusions for Analysis #3. In Analysis #3, we conclude that the minimum number of instances for training a defect predictor is around $100 - 200$ data samples. Please note that three SOFTLAB projects *ar3, ar4* and *ar5* have {63,107, 36} modules respectively. Thus, the minimum number of training samples occurs when a predictor is trained from *(ar3 + ar5)* projects. In this case, only $(63 + 36) * 0.90 = 90$ training samples are used to construct a defect predictor for the *ar4* project. Similarly, the maximum number of training samples occurs when a predictor is trained from *(ar3 + ar4)* projects. Then $(63 + 107) * 0.90 = 153$ training samples are used to construct a defect predictor for the *ar5* project. Therefore, the WC results in Fig. 12 are achieved using a minimum of 90 and a maximum of 153 training samples.

Figures 13, 14 and 15 shows *Observation1* (i.e. $NN_{pd} \geq WC_{pd}$) and *Observation2* (i.e. $NN_{pf} \leq WC_{pf}$) for SOFTLAB projects. Please recall that these observations were mutually exclusive for NASA data. The pattern is similar in SOFTLAB projects:

- for *ar4* mutual exclusiveness hold: $NN_{pd} \geq WC_{pd}$ and $NN_{pf} > WC_{pf}$
- for *ar3* and *ar5*: $NN_{pf} \leq WC_{pf}$. If the observations were mutually exclusive, we would expect $NN_{pd} < WC_{pd}$. While this is not the case for *pd*, we observe that $NN_{pd} = WC_{pd}$ and $NN_{pd} \not> WC_{pd}$ (see Fig. 13).

Fig. 14 Analysis #2 PF results for the SOFTLAB projects where $NN_{pf} \leq WC_{pf}$

	rank		quartiles 0 25 50 75 100					
AR3	1	NN	38	38	38	38	40	
	2	WC	40	40	40	40	42	
	3	CC	38	55	56	60	80	
AR5	1	NN	21	25	25	25	32	
	2	WC	29	29	29	29	29	
	3	CC	29	46	50	50	79	

50%

Fig. 15 Analysis #2 PF results for the SOFTLAB projects where $NN_{pf} > WC_{pf}$

In summary, the WC, CC and NN patterns found in American NASA rocket software are also observed in software controllers of Turkish domestic appliances. While this is not the definitive proof of the external validity of our results, we find it very compelling that they are reproducable in different companies.

8 Related Work

8.1 Neighborhood Reasoning

Our use of nearest neighbor is somewhat similar to *k-NN classifier* or *reasoning by analogy*. A *k-NN classifier* returns the majority class in the "k" nearest neighbors to the test instance. Lessman et.al. benchmark k-NN against 19 other learners and found that it was one of the four worst methods for defect prediction (Lessmann et al. 2008).

In *reasoning by analogy*, inference about some new case is made by partial match to prior cases, then applying some repair or generalization procedure to the matched cases. Like k-NN, reasoning by analogy makes use of some distance metric that finds old cases similiar (i.e. nearer) to the new case. While reasoning by analogy is a common effort estimation method (e.g. Shepperd and Schofield 1997), it is far less common to see it applied to defect prediction. One reason for this is the typical size of the training set: effort estimation typically deals with dozens of prior examples while defect training sets can have thousands of examples. We conjecture that as the data set grows in size, simple induction over many instances may be more powerful than complex analogical reasoning over a handful of examples.

Our work is most similar to that of Khoshgoftaar and Seliya (2003) who augmented k-NN with a counting procedure that was a slightly more sophisticated decision procedure than "return the majority class". Our approach differs to theirs in that we bundle together the k-nearest neighbors, then pass that bundle from a full-fledged classifier (Naive Bayes).

8.2 On Defect Prediction Using Static Code Features

We study defect predictors learned from static code attributes since they are *useful*, *easy to collect*, and *widely-used*.

Useful: Defect predictors are considered useful, if they provide a prediction performance that is comparable to or better than manual reviews (Moser et al. 2008). This paper finds defect predictors with a probability of detection of 80%, or higher. This is higher than currently-used industrial methods such as manual code reviews:

– A *IEEE Metrics 2002* (Shull et al. 2002) panel concluded that manual software reviews can find ≈60% of defects[11]

– In 2004, Raffo (personnel communication) reports that the defect detection capability of industrial review methods can vary from probability of detection: $pd = TR(35, 50, 65)\%$[12] for full Fagan inspections (Fagan 1976), to $pd = TR(13, 21, 30)$ for less-structured inspections.

Further, defect predictors based on static code features are considered as static analysis, since they do not require the execution of code. Zheng et.al. compares automated static analysis (ASA) with manual inspections on Nortel software and conclude that "*our results indicate that ASA is an economical complement to other verification and validation techniques.*"

Easy to collect: Employing defect predictors in practice should not take too much time (Moser et al. 2008) for data collection and constructing the models themselves. An advantage of static code features is that they can be quickly and automatically collected from the source code, even if no other information is available. Static code attributes like lines of code and the McCabe/Halstead attributes can be automatically and cheaply collected, even for very large systems (Nagappan and Ball 2005b). By contrast, other methods such as manual code reviews are labor-intensive; e.g. each member of a review team can inspect 8 to 20 LOC/minute (Menzies et al. 2002). Furthermore, other features (e.g. number of developers, the software development practices used to develop the code) may be unavailable or hard to characterize.

Widely used: Many researchers use static features to guide software quality predictions (see Nagappan and Ball 2005b; Menzies et al. 2002, 2004; Halstead 1977; Mccabe 1976; Chapman and Solomon 2002; Polyspace 2008; Hall and Munson 2000; Nikora and Munson 2003; Khoshgoftaar and Seliya 2004; Tang and Khoshgoftaar 2004; Porter and Selby 1990; Tian and Zelkowitz 1995; Srinivasan and Fisher 1995). Verification and validation (V&V) textbooks advise using static code features to select modules worthy of manual inspections (Rakitin 2001).

Nevertheless, there are many reasons to doubt the value of static code attributes for defect prediction. Static code attributes are hardly a complete characterization of the internals of a function. Fenton offers an insightful example where *the same* functionality is achieved using *different* programming language constructs resulting in *different* static measurements for that module (Fenton and Pfleeger 1995). Fenton uses this example to argue the uselessness of static code attributes. Further, Fenton & Pfleeger note that the main McCabe's attribute (cyclomatic complexity, or $v(g)$) is highly correlated with lines of code (Fenton and Pfleeger 1995). Shepperd and Ince repeated that result, commenting that "for a large class of software it (cyclomatic complexity) is no more than a proxy for, and in many cases outperformed by, lines of code" (Shepperd and Ince 1994).

However, if the above criticisms are correct then we would predict that, in general, the performance of a defect predictor learned by a data miner should be very poor.

---

[11]That panel supported neither Fagan's claim (Fagan 1986) that inspections can find 95% of defects before testing nor Shull's claim that specialized directed inspection methods can catch 35% more defects that other methods (Shull et al. 2000).

[12]$TR(a, b, c)$ is a triangular distribution with min/mode/max of a, b, c.

Table 10 Some representative *pd*s and *pf*s for prediction problems from the UC Irvine machine learning database (Blake and Merz 1998)

Data	Probability of	
	Detection	False alarm
Pima diabetes	60	19
Sonar	71	29
Horse-colic	71	7
Heart-statlog	73	21
Rangeseg	76	30
Credit rating	88	16
Sick	88	1
Hepatitis	94	56
Vote	95	3
Ionosphere	96	18
Mean	81	20

These values were generated using the standard settings of a state-of-art decision tree learner (J48). For each data set, ten experiments were conducted, where a decision tree was learned on 90% of the data, then tests are done of the remaining 10%. The numbers shown here are the average results across ten such experiments

More specifically, the supposedly better static code attributes such as Halstead and Mccabe should perform no better than just simple thresholds on lines of code.

Neither of these predictions are true, at least for the data sets used in this study. The defect predictors learned from static code attributes perform surprisingly well. Formally, learning a defect predictor is a *binary prediction problem* where each module in a project has been labeled as "defect-free" or "defective". The learning problem is to build some predictor which guesses the labels for as-yet-unseen modules. In this paper we find predictors (i.e. for SOFTLAB projects) with a probability of detection (pd) and probability of false alarm (pf) of

$$median(pd) = 88; median(pf) = 29$$

Table 10 lets us compare our new results against standard binary prediction results from the UC Irvine machine learning repository of standard test sets for data miners (Blake and Merz 1998). Our *median*(*pd*, *pf*) are very close to the standard results of *mean*(*pd*, *pf*)=(81%,20%) which is noteworthy in two ways:

1. It is unexpected. If static code attributes capture so little about source code (as argued by Shepherd, Ince, Fenton and Pfleeger), then we would expect lower probabilities of detection and much higher false alarm rates.
2. These (*pd*, *pf*) results are better than currently used industrial methods such as the *pd*≈60% reported at the 2002 IEEE Metrics panel or the *median*(*pd*) = [21..50] reported by Raffo.[13]

---

[13]Please note that we can only compare the defect *detection* properties of automatic vs manual methods. Unlike automatic defect prediction via data mining, the above manual inspection methods don't just report "true,false" on a module, Rather, the manual methods also provide specific debugging information. Hence, a complete comparison of automatic vs manual defect prediction would have to include both an analysis of the time to *detect* potential defects *and* the time required to *fix* them. Manual methods might score higher to automatic methods since they can offer more clues

9 Practical Implications

While Table 10 shows that our defect detectors work nearly as well as standard data mining methods, it does not necessarily demonstrate that false alarm rates of around 29% are useful in an industrial context. For example, Arisholm and Briand have certain concerns on the practical usage of defect predictors (Arisholm and Briand 2006b). They argue that if X% of the modules are predicted to be faulty and if those modules contain less than X% of the defects, then the costs of generating the defect predictor is not worth the effort.

Let us analyze the testing efforts on MW1 project from Arisholm and Briand's point of view. For MW1, there are a total of 403 modules with 31 defective and 372 defect-free ones. CC model yields 90% pd and 68% pf, and one should examine 280 modules, which is around a 31% reduction in inspection efforts compared to examining all modules. Yet, we argue that 68% pf rate is quite high and using NN we are able to reduce it to 33% along with 68% pd. This corresponds to examining 144 modules, a reduction of 47% compared to exhaustive testing (and we assume an exhaustive test should examine 274 modules for detecting 68% defects, as Arisholm and Briand suggests).

This analysis can be extended for all projects used in this paper. For instance, the company from which SOFTLAB data in Table 1 are collected is keen to use our detectors, arguing that they operate in a highly competitive market segment where profit margins are very tight. Therefore reducing the cost of the product even by 1% can make a major difference both in market share and profits. Their applications are embedded systems where, over the last decade, the software components have taken precedence over the hardware. Hence their problem is a software engineering problem. According to Brooks (1995), half the cost of software development is in unit and systems testing. The company also believes that their main challenge is the testing phase and they seek predictors that indicate where the defects might exist *before* they start testing. Their expectation from the predictor is not to detect all defects, but to guide them to the problematic modules so that they can detect more defects in shorter times. Hence, any reduction in their testing efforts allows them to efficiently use their scarce resources.

Considering the results of our analyses, if a company lacks local data, we would suggest a two-phase approach. In phase one, that organization uses imported CC data, filtered via NN. Also, during phase one, the organization should start a data collection program to collect static code attributes. Phase two commences when there is *enough* local WC data to learn defect predictors. During phase two, the organization would switch to new defect predictors learned from the WC data.

An important issue worth more mentioning is the concern about the time required for setting up a metric program (i.e. in order to collect data for building actual defect predictors). Our incremental WC results suggest that, in the case of defect prediction, this concern may be less than previously believed. Kitchenham et al. (2007) argue

---

back to the developer about what is wrong with the method. However, such an analysis is beyond the scope of this paper. Here, we focus only on the relative merits of different methods for *predicting* error prone modules.

that organizations use cross-company data since within-company data can be so hard to collect:

– The time required to collect enough data on past projects from within a company may be prohibitive.
– Collecting within-company data may take so long that technologies change and older projects do not represent current practice.

In our analysis we observe that as few as 100 modules are enough to learn adequate defect predictors. When so few examples are enough, it is possible that projects can learn local defect predictors that are relevant to their current technology in just a few months.

Further, our experiences with our industry partners show that data collection is not necessarily a major concern. Static code attributes can be automatically and quickly collected with relatively little effort. We have found that when there is high level management commitment, it becomes a relatively simple process. For the three projects of SOFTLAB data, neither the static code attributes, nor the mapping of defects to software modules were available when the authors attempted to collect these data. Since these were smaller scale projects, it was sufficient to spend some time with the developers and going through defect reports. Although not all projects have 100 modules individually, the company has a growing repository from several projects and enough data to perform defect prediction.

We also have experience with a large scale telecommunication company, where a long-term metric program for monitoring complex projects (around 750,000 lines of code) requires introducing automated processes. Again with high level management support, it was possible to employ appropriate tool support and these new processes were introduced easily and invisibly to the staff. For that project, we have now a growing repository of defects mapped with source code (around 25 defects per month). Software in that project are being developed for more than 10 years and have very low defect rates. We have obtained the first results in the 8th month of the project and it is planned to be completed in 12 months. In summary setting up a metric program for defect prediction can be done more quickly than it is perceived.

10 Threats to Validity

In prior work we have explored a range of data mining methods for defect prediction and found that classifiers based on Bayes theorem work best for the Table 1 data (Menzies et al. 2007b). Since that study, we have tried to find better data mining algorithms for defect prediction. To date, we have failed. Other researchers have also failed to improve our results. For example, Lessmann et al. investigate the statistical difference of the results between 19 learners, including naive Bayes, on the NASA projects (Lessmann et al. 2008). Figure 16 shows that the simple Bayesian method discussed above ties in first place along with 15 other methods.

Our analyses do not involve a value-based approach. We do not consider that fixing defects in a module may be more critical than fixing defects in another one. One reason is that our data sources do not consistently include necessary data (i.e. the severity level of defects). If and when data are available, same kind of analysis should be repeated considering this issue.

Fig. 16 x-axis shows the rank
of the data miners given in the
y-axis. All the methods whose
top ranks are 4 to 12 are
statistically insignificantly
different. From Lessmann
et al. (2008)
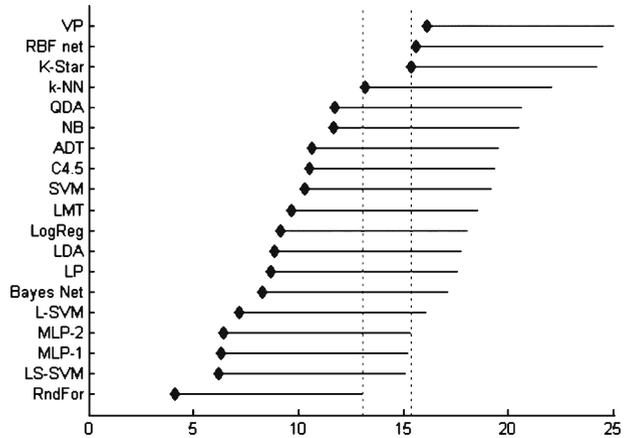
Another point is that descriptions of software modules *only* in terms of static code
attributes can overlook some important aspects of software including: the type of
application domain; the skill level of the individual programmers involved in sys-
tem development; contractor development practices; the variation in measurement
practices; and the validation of the measurements and instruments used to collect the
data. For this reason some researchers augment, or replace static code measures with
other information such as the history of past faults or changes to code or number of
developers who have worked on the code (Graves et al. 2000). Yet again, we have
used all available data in our data sources, which are the static code features of the
projects.

The external validity of generalizing from NASA examples has been discussed
elsewhere (Menzies et al. 2007b). In summary, NASA uses contractors who are
contractually obliged to demonstrate their understanding and usage of current in-
dustrial best practices. These contractors service many other industries; for example,
Rockwell-Collins builds systems for many government and commercial organiza-
tions. For these reasons, other noted researchers such as Basili et al. (2002) have
argued that conclusions from NASA data are relevant to the general software
engineering industry.

Nevertheless, it is always wise to test claims of external validity. Hence:

– The SOFTLAB data were, initially, in reserve. Our first three analyses are based
   solely on the aerospace applications found in the NASA data.
– Our last analysis checked if the SOFTLAB data exhibit the same pattern as the
   NASA data.

SOFTLAB data sets were deliberately chosen to be as far removed as possible
from American aerospace software applications (i.e. Turkish home appliances con-
troller software). Please note that this software was developed via methods that
are both culturally and organizationally different to NASA aerospace software.
Turkish domestic appliances company software are developed by a small team
of 2-3 people. The development is carried out in an ad-hoc, informal way rather
than formal, process oriented approach in NASA. Furthermore, the company is
a profit and revenue driven commercial organization, whereas NASA is a cost

driven government entity. This implies that our approach is widely applicable among different development practices. More precisely, our approach is independent of the processes that yield the final product, at least for the wide range of projects that we have analyzed.

11 Conclusion

In this study, we have analyzed defect predictors of static code features, constructed by cross company vs. within company data and found clear, unambiguous conclusions:

– CC-data dramatically increase the probability of detecting defective modules (i.e. from median value 75 to 92);
– But CC-data also dramatically increase the false alarm rate (i.e. from median value 29 to 64).
– NN-filtering CC data avoids the high false alarm rates by removing irrelevancies in CC data (i.e. from median value 64 to 32). This removal takes place by automatically selecting similar project data in terms of available static code features and discarding non-similar ones.
– Yet WC-data models are still the best and they can be constructed with small amounts of data (i.e. 100 examples).

Interpreting these in terms of the posed questions in the introduction, we conclude that:

– CC data are useful in extreme cases such as mission critical projects, where the cost of false alarms can be afforded. Therefore, CC data should be used only when WC data are not available.
– Pruning CC data with NN-filter allows the use of CC data for constructing practical defect predictors in other domains. NN-filtered CC data yields much better results than raw CC data, yet closer but worse results than WC data.
– The best option of using WC data requires the collection of a mere hundred examples from within a company and can be done in a short time (i.e. a few months).
– We observe the same patterns not only in aerospace software from NASA, but also in software from a completely different company located in another country. While this is a strong evidence of generality, we take great care *not to* interpret it as a proof of external validity.

We conclude our findings by proposing a two-phase approach for initiating defect prediction process:

– *Phase1:* Use NN filtered CC data to make local predictions and start to collect WC data.
– *Phase2:* After a few hundred examples are available in the local repository (usually a few months), discard the predictor learned on CC data and switch to those learned from WC data.

References

Arisholm E, Briand L (2006a) Predicting fault-prone components in a java legacy system. In: ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on international symposium on empirical software engineering, September 2006. http://portal.acm.org/citation.cfm?id=1159733.1159738

Arisholm E, Briand L (2006b) Predicting fault-prone components in a java legacy system. In: 5th ACM-IEEE international symposium on empirical software engineering (ISESE), Rio de Janeiro, Brazil, September 21–22. http://simula.no/research/engineering/publications/Arisholm.2006.4

Baker D (2007) A hybrid approach to expert and model-based effort estimation. Master's thesis, Lane Department of Computer Science and Electrical Engineering, West Virginia University. https://eidr.wvu.edu/etd/documentdata.eTD?documentid=5443

Basili V, McGarry F, Pajerski R, Zelkowitz M (2002) Lessons learned from 25 years of process improvement: the rise and fall of the NASA software engineering laboratory. In: Proceedings of the 24th international conference on software engineering (ICSE) 2002, Orlando, Florida. http://www.cs.umd.edu/projects/SoftEng/ESEG/papers/83.88.pdf

Bell R, Ostrand T, Weyuker E (2006) Looking for bugs in all the right places. In: ISSTA '06: Proceedings of the 2006 international symposium on software testing and analysis, July 2006. http://portal.acm.org/citation.cfm?id=1146238.1146246

Blake C, Merz C (1998) UCI repository of machine learning databases. http://www.ics.uci.edu/~mlearn/MLRepository.html

Boehm B, Papaccio P (1988) Understanding and controlling software costs. IEEE Trans Softw Eng 14(10):1462–1477, October 1988

Boehm B (2000) Safe and simple software cost analysis. IEEE Software, pp 14–17, September/October 2000. http://www.computer.org/certification/beta/Boehm_Safe.pdf

Boetticher G, Menzies T, Ostrand T (2007) The PROMISE repository of empirical software engineering data. http://promisedata.org/repository

Brooks FP (1995) The mythical man-month, Anniversary edn. Addison-Wesley, Reading

Chapman M, Solomon D (2002) The relationship of cyclomatic complexity, essential complexity and error rates. In: Proceedings of the NASA software assurance symposium, Coolfont Resort and Conference Center in Berkley Springs, West Virginia. http://www.ivv.nasa.gov/business/research/osmasas/conclusion2002/Mike_Chapman_The_Relationship_of_Cyclomatic_Complexity_Essential_Complexity_and_Error_Rates.ppt

Chen Z, Menzies T, Port D (2005) Feature subset selection can improve software cost estimation. In: PROMISE'05. http://menzies.us/pdf/05/fsscocomo.pdf

Dekhtyar A, Hayes JH, Menzies T (2004) Text is software too. In: International workshop on mining software repositories. http://menzies.us/pdf/04msrtext.pdf

Demsar J (2006) Statistical comparisons of clasifiers over multiple data sets. J Mach Learn Res 7:1–30. http://jmlr.csail.mit.edu/papers/v7/demsar06a.html

Domingos P, Pazzani MJ (1997) On the optimality of the simple bayesian classifier under zero-one loss. Mach Learn 29(2–3):103–130 citeseer.ist.psu.edu/domingos97optimality.html

Dougherty J, Kohavi R, Sahami M (1995) Supervised and unsupervised discretization of continuous features. In: International conference on machine learning, pp 194–202. http://www.cs.pdx.edu/~timm/dm/dougherty95supervised.pdf

Duda R, Hart P, Nilsson N (1976) Subjective bayesian methods for rule-based inference systems. In: Technical Report 124, Artificial Intelligence Center, SRI International

Fagan M (1976) Design and code inspections to reduce errors in program development. IBM Syst J 15(3):182–211

Fagan M (1986) Advances in software inspections. IEEE Trans Softw Eng 12:744–751, July 1986

Fenton NE, Pfleeger S (1995) Software metrics: a rigorous & practical approach, 2nd edn. International Thompson, London

Goseva K, Hamill M (2007) Architecture-based software reliability: why only a few parameters matter? In: 31st annual IEEE international computer software and applications conference (COMPSAC 2007), Beijing, July 2007

Graves TL, Karr AF, Marron JS, Siy HP (2000) Predicting fault incidence using software change history. IEEE Trans Softw Eng 26(7):653–661. www.niss.org/technicalreports/tr80.pdf

Hall G, Munson J (2000) Software evolution: code delta and code churn. J Syst Softw 54(2):111–118

Halstead M (1977) Elements of software science. Elsevier, Amsterdam

Hayes JH, Dekhtyar A, Sundaram SK (2006) Advancing candidate link generation for require-ments tracing: the study of methods. IEEE Trans Softw Eng 32(1):4–19. http://doi.ieeecomputersociety.org/10.1109/TSE.2006.3

Jiang Y, Cukic B, Menzies T (2007) Fault prediction using early lifecycle data. In: ISSRE'07. http://menzies.us/pdf/07issre.pdf

John G, Langley P (1995) Estimating continuous distributions in bayesian classifiers. In: Proceedings of the eleventh conference on uncertainty in artificial intelligence. Morgan Kaufmann, Montreal, pp 338–345. http://citeseer.ist.psu.edu/john95estimating.html

Khoshgoftaar TM, Seliya N (2003) Analogy-based practical classification rules for software quality estimation. Empirical Softw Eng 8(4):325–350

Khoshgoftaar T, Seliya N (2004) Comparative assessment of software quality classification tech-niques: an empirical case study. Empirical Softw Eng 9(3):229–257

Kitchenham BA, Mendes E, Travassos GH (2007) Cross- vs. within-company cost estimation studies: a systematic review. IEEE Trans Softw Eng 33:316–329, May 2007

Koru AG, Emam KE, Zhang D, Liu H, Mathew D (2008) Theory of relative defect proneness. Empirical Softw Eng 13(5):473–498

Lessmann S, Baesens B, Mues C, Pietsch S (2008) Benchmarking classification models for software defect prediction: a proposed framework and novel findings. IEEE Trans Softw Eng 34(4): 485–496

Mann HB, Whitney DR (1947) On a test of whether one of two random variables is stochastically larger than the other. Ann Math Stat 18(1):50–60. http://projecteuclid.org/DPubS?service=UI&version=1.0&verb=Display&handle=euclid.aoms/1177730491

Marcus A, Poshyvanyk D, Ferenc R (2008) Using the conceptual cohesion of classes for fault prediction in object-oriented systems. IEEE Trans Softw Eng 34(2):287–300, March–April 2008

McCabe T (1976) A complexity measure. IEEE Trans Softw Eng 2(4):308–320, December 1976

Menzies T, Raffo D, Setamanit S, Hu Y, Tootoonian S (2002) Model-based tests of truisms. In: Proceedings of IEEE ASE 2002. http://menzies.us/pdf/02truisms.pdf

Menzies T, DiStefano J, Orrego A, Chapman R (2004) Assessing predictors of software defects. In: Proceedings, workshop on predictive software models, Chicago. http://menzies.us/pdf/04psm.pdf.

Menzies T, Dekhtyar A, Distefano J, Greenwald J (2007a) Problems with precision. IEEE Trans Softw Eng 33:637–640. http://menzies.us/pdf/07precision.pdf

Menzies T, Greenwald J, Frank A (2007b) Data mining static code attributes to learn defect predic-tors. IEEE Trans Softw Eng 33:2–13 http://menzies.us/pdf/06learnPredict.pdf

Menzies T, Turhan B, Bener A, Gay G, Cukic B, Jiang Y (2008) Implications of ceiling effects in defect predictors. In: Proceedings of PROMISE 2008 workshop (ICSE). http://menzies.us/pdf/08ceiling.pdf

Moser R, Pedrycz W, Succi G (2008) A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: ICSE '08: Proceedings of the 30th international conference on software engineering, May 2008. http://portal.acm.org/citation.cfm?id=1368088.1368114

Musa J, Iannino A, Okumoto K (1987) Software reliability: measurement, prediction, application. McGraw Hill, New York

Nagappan N, Ball T (2005a) Static analysis tools as early indicators of pre-release defect density. In: ICSE 2005, St. Louis

Nagappan N, Ball T (2005b) Static analysis tools as early indicators of pre-release defect density. In: ICSE, pp 580–586. http://doi.acm.org/10.1145/1062558

Nikora A, Munson J (2003) Developing fault predictors for evolving software systems. In: Ninth international software metrics symposium (METRICS'03)

Orrego A (2004) Sawtooth: learning from huge amounts of data. Master's thesis, Computer Science, West Virginia University

Ostrand T, Weyuker E, Bell R (2007) Automating algorithms for the identification of fault-prone files. In: ISSTA '07: Proceedings of the 2007 international symposium on software testing and analysis, July 2007. http://portal.acm.org/citation.cfm?id=1273463.1273493

Polyspace (2008) Polyspace verifier®. http://www.di.ens.fr/~cousot/projects/DAEDALUS/synthetic_summary/POLYSPACE/polyspace-daedalus.htm

Porter A, Selby R (1990) Empirically guided software development using metric-based classification trees. IEEE Softw 7:46–54, March

Quinlan R (1992a) C4.5: programs for machine learning. Morgan Kaufman, San Francisco, iSBN: 1558602380

Quinlan JR (1992b) Learning with continuous classes. In: 5th Australian joint conference on artificial intelligence, pp 343–348. http://citeseer.nj.nec.com/quinlan92learning.html

Rakitin S (2001) Software verification and validation for practitioners and managers, 2nd edn. Artech House, Cormano

Shepperd M, Ince D (1994) A critique of three metrics. J Syst Softw 26(3):197–210, September 1994

Shepperd M, Schofield C (1997) Estimating software project effort using analogies. IEEE Trans Softw Eng 23(12), November 1997. http://www.utdallas.edu/~rbanker/SE_XII.pdf

Shull F, Basili V, Boehm B, Brown A, Costa P, Lindvall M, Port D, Rus I, Tesoriero R, Zelkowitz M (2002) What we have learned about fighting defects. In: Proceedings of 8th international software metrics symposium, Ottawa, Canada, pp 249–258. http://fc-md.umd.edu/fcmd/Papers/shull_defects.ps

Shull F, Rus I, Basili V (2000) How perspective-based reading can improve requirements inspections. IEEE Comput 33(7):73–79. http://www.cs.umd.edu/projects/SoftEng/ESEG/papers/82.77.pdf

Srinivasan K, Fisher D (1995) Machine learning approaches to estimating software development effort. IEEE Trans. Softw Eng 21(2):126–137, February 1995

Tang W, Khoshgoftaar TM (2004) Noise identification with the k-means algorithm. In: ICTAI 2004, pp 373–378. http://doi.ieeecomputersociety.org/10.1109/ICTAI.2004.93

Tian J, Zelkowitz M (1995) Complexity measure evaluation and selection. IEEE Trans Softw Eng 21(8):641–649, August 1995

Witten IH, Frank E (2005) Data mining, 2nd edn. Morgan Kaufmann, Los Altos

Yang Y, Webb G (2003) Weighted proportional k-interval discretization for naive-bayes classifiers. In: Proceedings of the 7th Pacific-Asia conference on knowledge discovery and data mining (PAKDD 2003). http://www.csse.monash.edu/~webb/Files/YangWebb03.pdf

Burak Turhan received his PhD degree from the department of Computer Engineering at Bogazici University. He recently joined in NRC-Canada IIT-SEG as a Research Associate after six years of research assistant experience in Bogazici University. His research interests include all aspects of software quality and are focused on software defect prediction models. He is a member of IEEE, IEEE Computer Society and ACM SIGSOFT.

Tim Menzies (tim@menzies.us) has been working on advanced modeling, software engineering, and AI since 1986. He received his PhD from the University of New South Wales, Sydney, Australia and is the author of over 160 refereeed papers. A former research chair for NASA, Dr. Menzies is now a associate professor at the West Virginia University's Lane Department of Computer Science and Electrical Engineering. For more information, visit his web page at http://menzies.us.

Ayşe B. Bener is an assistant professor and a full time faculty member in the Department of Computer Engineering at Bogazici University. Her research interests are software defect prediction, process improvement and software economics. Bener has a PhD in information systems from the London School of Economics. She is a member of the IEEE, the IEEE Computer Society and the ACM.

Justin Di Stefano is currently the Software Technical Lead for Delcan, Inc. in Vienna, Virginia, specializing in transportation management and planning. He earned his Master's degree in Electrical Engineering (with a specialty area of Software Engineering) from West Virginia University in 2007. Prior to his current employment he worked as a researcher for the WVU/NASA Space Grant program where he helped to develop a spin-off product based upon research into static code metrics and error prone code prediction. His undergraduate degrees are in Electrical Engineering and Computer Engineering, both from West Virginia University, earned in the fall of 2002. He has numerous publications on software error prediction, static code analysis and various machine learning algorithms.