An Instance Based Reasoning for Empirical Transfer Learning

Rahul Krishna
North Carolina State University, USA
rkrish11@ncsu.edu

Tim Menzies
North Carolina State University, USA
tim.menzies@gmail.com

***Abstract*—**

***Index Terms*—defect prediction, CART**

I. Introduction

II. Motivating Example

III. Background

IV. Instance Based Learning

Assessing the quality of solutions to a real-world problem can be extremely hard. In several applications, researchers have models that can emulate the problem. Using these models it is possible to examine several scenarios in a short period of time, and this can be done in a reproducible manner. However, models aren't always the solution, as we shall see.

There exist several problems where models are hard to obtain, or the input and output are related by complex connections that simply cannot be modeled in a reliable manner, or generation of reliable models take prohibitively long. Software defect prediction is an excellent example of such a case. Models that incorporate all the intricate issues that may lead defects in a product is extremely hard to come by. Moreover, it has been shown that models for different regions within the same data can have very different properties [1].

In this paper we propose the use of an instance based approach in place of the conventional model based approach. Instead of generating solutions and going back to a model to see if the solution has any value, we

A. Spectral Learning using WHERE

The algorithm uses WHERE to recursively cluster the input data to identify subsets in the training data that a test instance can learn from. WHERE is a spectral learner which uses the FastMap heuristic to estimate the first principal component. It then recursively partitions the data into two halves along the median point of the projection on the first principal component, terminating when a half has less than $\sqrt{N}$ items.

B. Planning changes using WHAT

We propose the WHAT algorithm as a tool to plan changes in the original test data. Figure 1 highlights the procedure by which WHAT operates. It begins by creating clusters which are generated using the WHERE algorithm discussed above. Following this, a nearest neighbour scheme is applied to identify pairs of nearby clusters. A projective plane is constructed with these clusters at the vertices to characterize the cluster pairs.

The mutation policy works by projecting the test instance onto the projective planes and identifying the the plane on which the test instance has the largest scalar projection. The planner then reflects over the vertices of the chosen hyperplane, identifying the *better* vertex among the two. Now, a new instance is generated by mutating the attributes of the test instance towards the better vertex and away from the worse vertex. This process is repeated for all the test instances that are considered defective by the defect prediction scheme, discussed below.

C. Defect Prediction

To validate the treatments that have been suggested by our planner, we need to have defect predictors that capable of identifying if a certain module may (or may not) have a defect. A recent IEEE TSE paper by Lessmann et al. [5] compared 21 different leaners for software defect prediction:

- *Statistical classifiers:* Linear discriminant analysis, Quadratic discriminant analysis, Logistic regression, Naive Bayes, Bayesian networks, Least-angle regression, Relevance vector machine,
- *Nearest neighbor methods:* k-nearest neighbor, K-Star
- *Neural networks:* Multi-Layer Perceptron, Radial bias functions,
- *Support vector machine-based classifiers:* Support vector machine, Lagrangian SVM Least squares SVM, Linear programming, Voted perceptron,
- *Decision-tree approaches:* C4.5, CART, Alternating DTs
- *Ensemble methods:* **Random Forest**, Logistic Model Tree.

WHAT begins by *clustering* the training data into similar subsets using WHERE. WHERE works as follows:

1) Find two distant points in that population; call them the *east* and *west* poles.
2) Draw an axis of length c between the poles.
3) Let each point be at distance a, b to the *east,west* poles. Using the cosine rule, project each point onto the axis at $x = (a^2 + c^2 - b^2)/(2c)$.
4) Using the median x value, divide the population.
5) For each half that is larger than $\sqrt{N}$ of the original population, go to step 1.

There are couple of things to note: firstly, the above algorithm requires a distance measure between sets of decisions, for this WHERE uses case-based reasoning measure defined by Aha et al. [2]; secondly, in step-1 a linear time heuristic called FASTMAP [3] is used to find the two most distant points, this procedure is listed below:

- Pick a point at random;
- Let *east* be the point furthest from that point;
- Then *west* be the point furthest from *east*.

The final clusters that are found by WHERE are then used by WHAT as follows:

- For each cluster find its nearest neighbor.
- Construct a projective plane for each cluster pair. Label the cluster pairs as *Good* and *Bad*.
- For a test instance, project it onto the planes generated above and pick the one on which the test case has the largest projection.
- Mutate the attributes of the test case towards the *Good Cluster* in that projective plane.

Formally, WHAT can be considered as an instance based learning scheme. And that's beacuse it builds a set of recommendations based on a given test case based on the clusters formed at the training stage. This is unlike a model based approach, which would have used a model to do the same [4].

Fig. 1: Inside WHAT

They concluded that Random Forrest was the best method, CART being the worst.

Random Forest is an ensemble learning scheme that constructs a number of decision trees at the training time, for a test instance it outputs the mode of the classes of individual tree. It's patent from how random forest operates that the prediction will suffer if there is an imbalance in classes during the training. Unfortunately, the data sets explored here do suffer from severe skewness, as highlighted in **??**. A study conducted by Pelayo and Dick [6] inspected this issue. They showed that the SMOTE technique [7] can be used to improve recognition of defect-prone modules.

In short, SMOTE works by under-sampling the majority class and oversampling all the minority classes in the training data. We use a similar approach with one minor addition to the original algorithm. In our implementation of SMOTE we have introduced an additional step called *resampling*, wherein we ensure that after we do the over/under sampling, the new training data does not have any of the original rows. The new training data merely resembles the original data. This is done

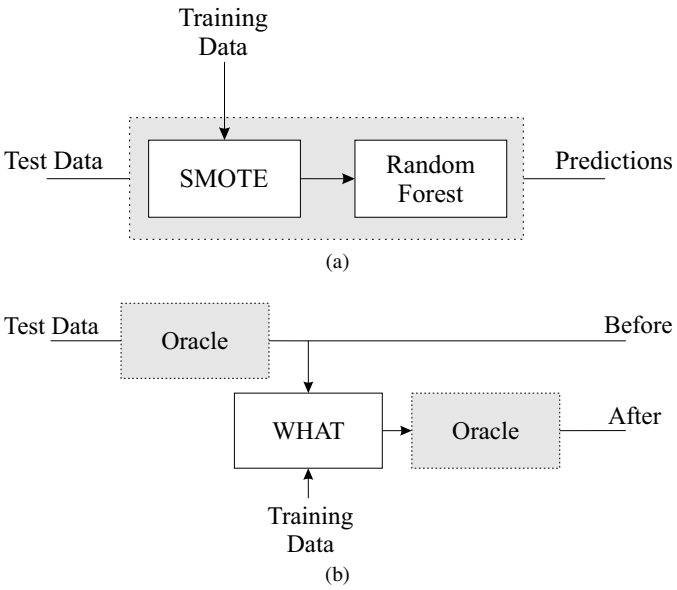

Fig. 3: (a) The defect prediction oracle, (b) The Planning scheme

as a precautionary measure, so as not to have WHAT and Random Forest train on the same training data.

V. Experimental Design

The following section describes the experimental rig and the experiments used to measure the performance of WHAT on 10 defect data sets and 6 performance prediction data sets.

A. Data sets

The data was obtained from the PROMISE repository. For the defect data we investigated 32 releases from 11 open source Java projects defined by the metrics highlighted in Fig. 2: *Apache Ant* (1.5 – 1.7), *Apache Camel* (1.2 – 1.6), *Apache Ivy* (1.1 – 2.0), *JEdit* (4.1 – 4.3), *Apache Log4j* (1.0 – 1.2), *Apache Lucene* (2.0 – 2.2), *PBeans* (1.0 and 2.0), *Apache POI* (2.0 – 3.0), *Apache Synapse* (1.0 – 1.2), *Apache Velocity* (1.4 – 1.6), and *Apache Xalan-Java* (2.5 – 2.7). Given the empirical nature of the data, it is important to design an experiment such that the planning phase uses only the *past* data to learn trends which can then be applied to the *future* data. Thus for our experiment we use data sets that have at least two consecutive releases.

- To generate recommendations for a release i, the planner uses releases releases $(i-1)$ and $(i-2)$.
- The predictor also uses releases $(i-1)$ and $(i-2)$. However, we use SMOTE with re-sampling in order to handle the class imbalance in the data and to prevent the predictor from using the same training data as the planner.

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

Fig. 2: OO measures used in our defect data sets. Last line is the dependent attribute (whether a defect is reported to a post-release bug-tracking system).

The performance prediction data set was obtained from []. The data set contains six examples of real world configurable systems: *Apache*, *LLVM*, *x264*, *Berkeley DB (written in C and Java*, and *SQLite*. The systems have different characteristics, different implementation languages, and different configuration mechanisms. The data set is a collection of all possible configurations (with SQLITE being a exception with only 4653 configurations). We performed a 5-fold cross validation study on this data set.

B. The Rig

The experimental rig is shown in Figure 3. It uses an *oracle* to determine wether a certain test case is defective or not. If the oracle suggests that an instance is defective, then we use WHAT to apply the recommendations to that test instance. The oracle is used to check if the recommendation helped subdue the defect.

C. Performance Assessment

In order to assess the performance of the planner for the defect data set, we use the Cliffs Delta score to measure the probability that the number of bugs in test data before applying the planner is larger than after doing so. In other words, we use the delta score to measure the ability of the planner to effective reduce the number of defects. Given the untreated test instance labeled *Before* of length *M* and the treated instances labeled *After*, the delta score is obtained as follows:

$$\delta = \frac{\#(Before > After) - \#(Before < After)}{M^2} \quad (1)$$

In the context of our application, the δ attains a value of 1 if the number of defects after applying the treatments has reduced to zero, and 0 if there is no change.

For the performance prediction data set on the other hand, Cliff's delta measure is not directly applicable. We therefore assess the performance by measuring the gain, given by $\frac{Before}{After}$. The gain takes a value larger than 1 if the run times have been reduced, a value of 1 if there is no change, and a value less than 1 if run times have increased after applying the recommended changes.

In addition to the above, we rank the different variants of the planning scheme to identify the best approach. We make use of the Scott-Knott procedure, recommended by Mittas & Angelis in their 2013 IEEE TSE paper [8], to compute the ranks. It works as follows: A list of treatments *l* is sorted by the median score. The list *l* is then split into sub-lists m, n in order to maximize the expected value of the differences in the observed performance before and after division. A statistical hypothesis test *H* is applied on the splits *m, n* to check if they are statistically different. If so, Skott-Knott then recurses on each division.

The research conducted by Shepperd and MacDonell [], Kampenes [] and Kocaguenli et al. [], highlighted that an "effect size" in lieu of a mere hypothesis test is required in order to verfiy if two populations are "significantly" different. An ICSE'11 paper by Arcuri [] endorsed the use of Vargha and Delaney's A12 effect size for reporting results in software engineering. Thus, for hypothesis testing H in Skott-Knott, we use the A12 test and a non-parametric bootstrap sampling [].

VI. Experimental Results

References

[1] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, and T. Zimmermann, "Local versus global lessons for defect prediction and effort estimation," *Software Engineering, IEEE Transactions on*, vol. 39, no. 6, pp. 822–834, June 2013.

[2] D. W. Aha, D. Kibler, and M. K. Albert, "Instance-based learning algorithms," *Machine learning*, vol. 6, no. 1, pp. 37–66, 1991.

Ant:

Rank	Treatment	Med	IQR
1	Baseline	0.03	0.0
2	$\alpha = 0.25$, Prune=50%	0.15	0.09
3	$\alpha = 0.5$, Prune=50%	0.25	0.1
4	$\alpha = 0.25$	0.34	0.05
4	$\alpha = 0.25$, weight	0.36	0.08
4	$\alpha = 0.75$, Prune=50%	0.41	0.13
5	$\alpha = 0.5$, weight	0.46	0.18
5	$\alpha = 0.5$	0.53	0.15
6	$\alpha = 0.75$	0.64	0.17
6	$\alpha = 0.75$, weight	0.73	0.17

Camel

Rank	Treatment	Med	IQR
1	Baseline	0.16	0.03
1	$\alpha = 0.25$	0.17	0.06
1	$\alpha = 0.25$, weight	0.17	0.06
2	$\alpha = 0.25$, Prune=50%	0.22	0.06
3	$\alpha = 0.5$	0.25	0.1
3	$\alpha = 0.5$, weight	0.26	0.14
4	$\alpha = 0.5$, Prune=50%	0.32	0.16
5	$\alpha = 0.75$, weight	0.38	0.19
5	$\alpha = 0.75$	0.4	0.12
6	$\alpha = 0.75$, Prune=50%	0.47	0.15

Ivy

Rank	Treatment	Med	IQR
1	Baseline	0.06	0.0
2	$\alpha = 0.25$, Prune=50%	0.14	0.04
3	$\alpha = 0.5$, Prune=50%	0.2	0.06
4	$\alpha = 0.75$, Prune=50%	0.27	0.14
5	$\alpha = 0.25$	0.41	0.06
5	$\alpha = 0.25$, weight	0.42	0.06
6	$\alpha = 0.5$, weight	0.55	0.11
6	$\alpha = 0.5$	0.57	0.09
7	$\alpha = 0.75$, weight	0.61	0.08
7	$\alpha = 0.75$	0.62	0.04

Jedit

Rank	Treatment	Med	IQR
1	Baseline	0.05	0.0
2	$\alpha = 0.25$, Prune=50%	0.11	0.05
3	$\alpha = 0.5$, Prune=50%	0.16	0.1
4	$\alpha = 0.25$	0.2	0.08
4	$\alpha = 0.25$, weight	0.21	0.1
5	$\alpha = 0.75$, Prune=50%	0.32	0.15
5	$\alpha = 0.5$, weight	0.33	0.05
5	$\alpha = 0.5$	0.34	0.18
6	$\alpha = 0.75$	0.56	0.2
6	$\alpha = 0.75$, weight	0.56	0.26

Log4j

Rank	Treatment	Med	IQR
1	$\alpha = 0.5$, Prune=50%	0.1	0.06
1	$\alpha = 0.25$, Prune=50%	0.11	0.04
1	$\alpha = 0.25$, weight	0.1	0.03
1	$\alpha = 0.5$, weight	0.1	0.13
1	Baseline	0.11	0.04
1	$\alpha = 0.25$	0.12	0.06
2	$\alpha = 0.5$	0.15	0.09
2	$\alpha = 0.75$, Prune=50%	0.15	0.05
2	$\alpha = 0.75$, weight	0.17	0.22
2	$\alpha = 0.75$	0.22	0.16

Lucene

Rank	Treatment	Med	IQR
1	$\alpha = 0.25$, weight	0.03	0.03
1	$\alpha = 0.25$	0.05	0.04
2	$\alpha = 0.5$	0.09	0.09
2	$\alpha = 0.25$, Prune=50%	0.09	0.03
2	Baseline	0.1	0.01
2	$\alpha = 0.5$, weight	0.14	0.06
3	$\alpha = 0.5$, Prune=50%	0.19	0.08
4	$\alpha = 0.75$, weight	0.23	0.15
4	$\alpha = 0.75$	0.26	0.1
4	$\alpha = 0.75$, Prune=50%	0.28	0.13

Poi

Rank	Treatment	Med	IQR
1	$\alpha = 0.25$, Prune=50%	0.09	0.15
1	Baseline	0.09	0.05
1	$\alpha = 0.5$, Prune=50%	0.12	0.13
2	$\alpha = 0.25$	0.24	0.3
2	$\alpha = 0.25$, weight	0.25	0.32
3	$\alpha = 0.75$, Prune=50%	0.37	0.45
3	$\alpha = 0.5$, weight	0.49	0.5
4	$\alpha = 0.75$	0.53	0.47
4	$\alpha = 0.5$	0.53	0.28
4	$\alpha = 0.75$, weight	0.61	0.43

Velocity

Rank	Treatment	Med	IQR
1	$\alpha = 0.25$	0.08	0.03
1	$\alpha = 0.25$, weight	0.09	0.05
2	Baseline	0.13	0.04
3	$\alpha = 0.5$	0.16	0.09
3	$\alpha = 0.25$, Prune=50%	0.17	0.04
3	$\alpha = 0.5$, weight	0.19	0.09
4	$\alpha = 0.5$, Prune=50%	0.22	0.11
4	$\alpha = 0.75$, weight	0.25	0.1
4	$\alpha = 0.75$	0.28	0.2
4	$\alpha = 0.75$, Prune=50%	0.28	0.09

Xalan

Rank	Treatment	Med	IQR
1	$\alpha = 0.75$, weight	0.31	0.14
1	$\alpha = 0.25$, weight	0.32	0.1
1	$\alpha = 0.25$	0.32	0.06
2	$\alpha = 0.5$	0.34	0.11
2	Baseline	0.36	0.13
2	$\alpha = 0.5$, weight	0.35	0.12
2	$\alpha = 0.5$, Prune=50%	0.36	0.12
2	$\alpha = 0.25$, Prune=50%	0.38	0.1
2	$\alpha = 0.75$, Prune=50%	0.37	0.21
2	$\alpha = 0.75$	0.38	0.17

[3] C. Faloutsos and K.-I. Lin, *FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets*. ACM, 1995, vol. 24, no. 2.

[4] T. Menzies, "Xomo: Understanding development options for autonomy."

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

[6] L. Pelayo and S. Dick, "Applying novel resampling strategies to software defect prediction," in *Fuzzy Information Processing Society, 2007. NAFIPS '07. Annual Meeting of the North American*, June 2007, pp. 69–72.

[7] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, no. 1, pp. 321–357, 2002.

[8] N. Mittas and L. Angelis, "Ranking and clustering software cost estimation models through a multiple comparisons algorithm," *Software Engineering, IEEE Transactions on*, vol. 39, no. 4, pp. 537–551, 2013.