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 [].

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. The Planning Algorithm

The planning scheme makes use of the clusters 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.

C. Defect Prediction

V. Experimental Design

The following section describes 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 **??**: *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.

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).

- 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.

WHAT begins by *clustering* the training data into similar subsets using WHERE. It 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.

Note that the above requires a distance measure between sets of decisions: GALE uses the standard case-based reasoning measure defined by Aha et al. [**?**]. Note also that GALE implements step1 via the FASTMAP [**?**] linear-time heuristic:

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

These final sub-divisions found by this process are the *neighborhoods* that GALE will *perturb* as follows:

- Find the objective scores of the *east,west* poles in each neighborhood.
- Using the continuous domination predicate of moea, find the *better* pole.
- Perturb all points in that neighborhood by pushing them towards the better pole, by a distance $c/2$ (recall that c is the distance between the poles).
- Let generation $i+1$ be the combination of all pushed points from all neighborhoods.

From a formal perspective, GALE is an active learner [**?**] that builds a piecewise linear approximation to the Pareto frontier [**?**]. For each piece, it then pushes the neighborhood up the local gradient. This approximation is built in the reduced dimensional space found the FASTMAP Nyström approximation to the first component of PCA [**?**].

Fig. 1: Inside GALE

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. 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.

VI. Experimental Results

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

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 [] 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. In our paper, we use the A12 test and a non-parametric bootstrap sampling for hypothesis testing.

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