Prediction is Hard (Especially About the Future)

Rahul Krishna
CS, NcState, USA
i.m.ralk@gmail.com

Tim Menzies
CS, NcState, USA
tim.menzies@gmail.com

ABSTRACT

This paper provides ...

1 Introduction

2 Cluster and Contrast

A recurring data analysis pattern in this paper is $cluster{+}contrast$. The data is distilled into a few clusters using a clustering scheme. Then lessons are inferred by studying the differences between these clusters. These lessons are used to generate rules that can be applied in any context.

2.1 Clustering

There are a wide variety of clustering methods to choose from. A study by Ganesan [1] explored different clustering methods for software engineering data using the effort and defect data from the PROMISE repository [2]. In that study methods such as WHERE, K-Means, mini-batch-K-Means, DBScan, EM, and Ward were investigated. The results of the study showed that the size and number of clusters is more important that the specifics of the techniques used.

For this purposes of this work, we have chosen WHERE, a clustering scheme which is capable of generating at least $\sqrt{N}$ clusters given N instances. In addition to this, WHERE has been shown to run fast while ignoring spurious dimensions [3]. This is particularly useful, for much of the SE data is noisy, they contain a information not associated with the target variable.

2.2 Finding Contrasts

All the following methods use clustering in the form of WHERE, a top-down clustering method which recursively splits the data in two along a dimension that represents the highest variability, shown as Step-1 in figure 1.

Clustering is followed generating *contrast sets*. These contrast sets represent recommendations on what could be altered to better improve an outcome. In this work we have explored several algorithms as possible tools to identify contrast between clusters. These fall into three broad categories: a) Case Based Reasoning techniques (Nearest Neighbors), b) A Gradient base planner (called HOW), and c) Decision Trees. These techniques are discussed below. It is worth noting that they are organized in such a way that each technique seeks to address certain fallacies in the ones that precede it.

2.2.1 Case Based Reasoning

Case-based reasoning seeks to find solutions to problems by emulating human recollection and adaptation from past experiences. It has found extensive usage in Artificial Intelligence because it offers several advantages. However, one of the most important benefits that CBR has to offer is that it works on a "case-by-case" basis. Therefore it's advise is tailored to be specific to the particular case being considered. Several paper in SE have applied this technique, most of all for effort estimation [4–8].

A classic example of a CBR is a nearest neighbor approach. The nearest neighbor approach is rather straight forward and has been developed as a "straw-man"; i.e., a simple baseline tool to act as a benchmark used to evaluate other methods.

Our approach is rather straight forward, we divide the N training samples into $\sqrt{N}$ clusters and compute the centroid of these clusters. For every test case, we identify a cluster from the training set that most closely resembles it. Following this, we find the nearest cluster with a better performance score. The differences in the attributes between these two clusters constitute the *"contrast set"*. These contrast sets acts as plans that can be used to reflect over the test cases to improve them.

In most SE applications, not all features contribute equally to a problem. With this in mind, we opine that it would be beneficial if the above method is extended to include some form of feature weighting, thus enabling the tools to recommend changes to only the most informative features.

```
<Demrits of knn here and lead to the next
section>
```

2.2.2 HOW

HOW is very different compared to other CBR planners in that it explores the gradient between pairs of nearby clusters instead of studying the clusters themselves. HOW works by clustering the data during training using WHERE and then drawing `slopes` between the centroids of pairs of nearby clusters. Assuming the cluster pairs are labeled X and Y, with X having slightly better performance score than Y, the `slope` between X and Y acts as an indicator pointing to a direction to displace the data; i.e. away from Y and towards X.

While testing, HOW finds the nearest slope to every test case. The slope provides the exact magnitude and direction of displacements. Contrast sets are derived from these displacements. HOW offers a distinct advantage over CBR planners by limiting the displacements to very small regions (the displacements are never more than the separation between two clusters).

Step1: Top down clustering using WHERE
The data is recursively divided in clusters using WHERE as follows:
- Find two distance cases, X, Y by picking any case W at random, then setting X to its most distant case, then setting Y to the case most distant from X (this requires only $O(2N)$ comparisons of N cases).
- Project each case Z onto a `Slope` that runs between X, Y using the cosine rule.
- Split the data at the median X value of all cases and recurses on each half (stopping when one half has less than $\sqrt{N}$ of the original population).

Step2: Distinguish between clusters using decision trees
Call each leaf from WHERE a "class". Use an entropy-based decision tree (DT) learner to learn what attributes select for each "class". To limit tree size:
- Only use the top $\alpha = 33\%$ of the features, as determined by their information gain [9].
- Only build the trees down to max depth of $\beta = 10$.
- Only build subtrees if it contains at least $N^{\gamma=0.5}$ examples (where N is the size of the training set).

Score DT leaf nodes via the mean score of its majority cluster.

Step3: Generating contrast sets from DT branches
- Find the *current* cluster: take each test instance, run it down to a leaf in the DT tree.
- Find the *desired* cluster:
  - Starting at *current*, ascend the tree $lvl \in \{0, 1, 2...\}$ levels;
  - Identify *sibling* clusters; i.e. leaf clusters that can be reached from level lvl that are not *current*
  - Using the *score* defined above, find the *better* siblings; i.e. those with a *score* less than $\epsilon = 0.5$ times the mean score of *current*. If none found, then repeat for $lvl += 1$
  - Return the *closest* better sibling where distance is measured between the mean centroids of that sibling and *current*
- Find the *delta*; i.e. the set difference between conditions in the DT branch to *desired* and *current*. To find that delta:
  - For discrete attributes, return the value from *desired*.
  - For numerics, return the numeric difference.
  - For numerics into ranges, return a random number selected from the low and high boundaries of the that range.

Figure 1: CROSSTREES. Controlled by the parameters $\{\alpha, \beta, \gamma, \delta, \epsilon\}$ (set via engineering judgement).

2.2.3 Decision Trees

3 CROSSTREES

3.1 Design

3.2 Assessment

4 Experiments

4.1 Data

4.2 When not to Plan?

4.3 Evaluation

4.4 Results

5 Threats to Validity

6 Conclusion

Acknowledgements

7 References

[1] Divya Ganesan. Exploring essential content of defect prediction and effort estimation through data reduction. Master's thesis, Computer Science, West Virginia University, 2014.

[2] The Promise Repository of Empirical Software Engineering Data, 2015. http://openscience.us/repo. North Carolina State University, Department of Computer Science.

[3] Tim Menzies, Andrew Butcher, David Cok, Andrian Marcus, Lucas Layman, Forrest Shull, Burak Turhan, and Thomas Zimmermann. Local versus global lessons for defect prediction and effort estimation. *IEEE Transactions on Software Engineering*, 39(6):822–834, 2013.

[4] Jacky Wai Keung, Barbara A Kitchenham, and D Ross Jeffery. Analogy-x: providing statistical inference to analogy-based software cost estimation. *Software Engineering, IEEE Transactions on*, 34(4):471–484, 2008.

[5] T. Menzies, A. Brady, J. Keung, J. Hihn, S. Williams, O. El-Rawas, P. Green, and B. Boehm. Learning project management decisions: A case study with case-based reasoning versus data farming. *Software Engineering, IEEE Transactions on*, 39(12):1698–1713, Dec 2013.

[6] Fiona Walkerden and Ross Jeffery. An empirical study of analogy-based software effort estimation. *Empirical software engineering*, 4(2):135–158, 1999.

[7] Martin Shepperd and Chris Schofield. Estimating software project effort using analogies. *Software Engineering, IEEE Transactions on*, 23(11):736–743, 1997.

[8] Ekrem Kocaguneli, Gregory Gay, Tim Menzies, Ye Yang, and Jacky W Keung. When to use data from other projects for effort estimation. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 321–324. ACM, 2010.

[9] KB Irani and UM Fayyad. Multi-Interval Discretization of Continuous-Valued Attributes for Classification learning. *Proceedings of the National Academy of Sciences of the United States of America*, pages 1022–1027, 1993.

	Data set properties				
	training		testing		
data set	versions	cases	versions	cases	% defective
jedit	3.2, 4.0, 4.1, 4.2	1257	4.3	492	2
ivy	1.1, 1.4	352	2.0	352	11
camel	1.0, 1.2, 1.4	1819	1.6	965	19
ant	1.3, 1.4, 1.5, 1.6	947	1.7	745	22
synapse	1.0, 1.1	379	1.2	256	34
velocity	1.4, 1.5	410	1.6	229	34
lucene	2.0, 2.2	442	2.4	340	59
poi	1.5, 2, 2.5	936	3.0	442	64
xerces	1.0, 1.2, 1.3	1055	1.4	588	74
log4j	1.0, 1.1	244	1.2	205	92
xalan	2.4, 2.5, 2.6	2411	2.7	909	99

Results from learning									
untuned			tuned			change			
pd	pf	good?	pd	pf	good?	pd	pf		
55	29		64	29	y	9	0		$\star$
65	35	y	65	28	y	0	-7		$\star$
49	31		56	37		5	6		
49	13	y	63	16	y	14	3		$\star$
45	19		47	15		2	-4		
78	60		76	60		-2	0		
56	25		60	25	y	4	0		
56	31		60	10	y	4	-21		$\star$
30	31		40	29		10	-2		$\times$
32	6		30	6		-2	0		$\times$
38	9		47	9		9	0		$\times$

Figure 2: Training and test *data set properties* for the Jureczko data sets, sorted in ascending order of % defective examples. On the right-hand-side, we show the *results from learning*. Data is "good" if it has recall over 60% and false alarm under 40% (and note that, after tuning, there are more "good" than before). Data marked with "$\star$" show large improvements in performance, after tuning. Data marked with "$\times$" are "not good" since their test suites have so few non-defective examples (less than 5% of the total sample) that it becomes harder to find better data towards which we can displace test data.