# HOW NOT

Rahul Krishna, Tim Menzies
Computer Science, North Carolina State University, USA
{i.m.ralk, tim.menzies}gmail.com

***Abstract*—**

***Index Terms*—Defect prediction, configuration, prediction, planning, case-based reasoning.**

## I. Introduction

## II. Contrast Trees

Contrast trees are decision trees that can be used to learn decision rules from a large data set. The use of decision trees for this purpose makes it easy to visualize the data and also makes interpretation of the recommended policies fairly straight forward. In addition to this, contrast tree generate ranges of decisions that allow for better feasibility.

### *A. How does a contrast tree work?*

Contrast trees works by building a simple decision tree based on entropy, as in the classification and regression trees. Contrast tree uses the WHERE clustering algorithm, which in-turn uses FASTMAP [] to compute clusters that contain instances that are spatially close. Each cluster is assigned a unique cluster id. Then, a simple decision tree is built using a standard classification and regression tree (CART). The tree is built with the following tunable features:

- *Entropy* is used as criteria for generating optimum cuts while building the decision trees. These were shown to be consistent in giving high performance across most data.
- *Minimum samples per leaf* defines the minimum number of instances in a leaf of the decision tree.
- *Max depth* defines the number of levels of recursive splits that are allowed in the tree.

Decision trees can be used to generate a decision rule, which when applied to solution sets must theoretically improve performance. To generate rules – a tree is constructed with some training data, and for every instance in a the test data, we find a branch in the tree that best matches the test data. All instances that needs improvement in the test data constitute the "worst" set W. We find the nearest branch, "B", in the tree with better performance. The difference between W and B constitutes the decision rules.

A contrast set, "C" is simply a collection of all the decision rules. It can be obtained traversing through the tree from "W" to "B" and tracking all the branch variables along this path. Note that the branch variables use decision ranges instead of single point solutions, making implementation of the rules in real life feasible.

**Figure 1a: top down clustering:**
Data can be rapidly and recursively divided 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). Next, HOW projects each case $Z$ onto a `Slope` that runs between $X, Y$ using the cosine rule of Figure 1b. After that, 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). For more details. Figure **??**, line 84,

**Figure 1b: finding distances:**
In the Figure 1a, to compute distances, we use the Euclidean measure recommended for case-based reasoning by Aha et al. [**?**]; i.e. $\sqrt{\sum_i w_i(X_i - Y_i)^2}$ where $X_i, Y_i$ where values are normalized 0..1 for the range min..max and $w_i$ is some importance weight given to attribute $i$. Usually, $w_i = 1$ but it can be adjusted using, say, the feature weighting methods of Figure 1d.

**Figure 1c: 3 point geometry:**
Let $X, Y$ be two points joined by a *Slope* of length $c$. A third point, $Z$, has distances $a = dist(Z, X)$ and $b = dist(Z, Y)$ to $\overline{X, Y}$, respectively. According to the cosine rule, $Z$ projects onto $\overline{XY}$ at distance $x = (a^2 + c^2 - b^2)/(2c)$ from $X$. Further, according to Pythagoras' theorem, $Z$ stands at a distance $y = \sqrt{a^2 - x^2}$ from the line $\overline{XY}$. For more details, see Figure **??**, line 52.

**Figure 1d: feature weighting:**
HERE's feature weighting algorithm comes from the CART regression tree learner [**?**]. It sorts independent variables according to how well they can reduce the variability of a numeric objective. If we split some column $i$ of independent numeric data into $N = n_1 + n_2 + ..$ splits, then the expected value of the objective value of the split is $w_i = \sum_i v_i n_i / N$ where $v$ is the standard deviation of the objective. A recursive procedure can find those divisions $n_1, n_2, ...$ by (a) sorting a numeric column, then split at $\arg\min_i w_i$, then recursing into each half. For more details, see Figure **??**, line 113.

Fig. 1: Sub-routines within HOW.

### *B. Why would it work?*

In theory, contrast set is an ideal tool for generating decision rules. Some of the key advantages are listed below:

- They allow the user to exert a fine grain control over the parameters that needs to be changed, while providing a way to visualize the recommended changes.
- The changes suggested by the contrast sets are inherently local in nature, making the changes practical and potentially easy to implement.
- The contrast tress require a worst case time complexity of $O(n)$, which is a function in linear time.

### C. *Reasons for failure*

Although the idea seemed ideal, initial implementation of the contrast tree provided results that we most discouraging. One of the major problems with using contrast trees was that the trees were usually rather large. The initial motivation was that the trees could serve as a medium for experts to identify and explore solution spaced that were local to the problem. The size of the decision tree jeopardized the readability of the solutions by increasing the complexity. In our efforts to reduce the size of the tree, we devised a pruning method, that prunes away irrelevant branches that do not contribute better solutions. The smaller trees, while being easier to understand, posed another problem: the test instances were too dissimilar to the leaves in the tree. This meant that the relevance of the contrast set could not be justified.

The depth of the tree also influences the performance of contrast sets. The depth is represented by levels of splits in the tree, and this parameter determines the number of features that require changes. If the tree is too shallow, the changes suggested by the planner are too little to make any difference. On the other hand, deeper trees leads to over-fitting and therefore suggests too many changes when not required.

A study was conducted on numerous examples of the Jureczko object-oriented static code data sets []: Ant, Camel, Ivy, Jedit, Log4j, Lucene, Poi, Synapse, Velocity, Xalan, Xerces [1]. On these data sets, the plans generated by contrast tree advise how to change static code attributes in order to reduce defects in Java classes. The challenges with contrast trees had a profound impact on the performance, as shown in figure **??**.

### D. *What worked?*

Contrast tree is the second generation of our efforts to use contrast set learners for planning case based reasoning. W2 represents the first generation of our work. W2 is a CBR planner that reflected over the delta of raw attributes []. However, it frequently suffered from an optimization failure. When its plans were applied, performance improved in only $\frac{1}{3}$rd of test cases.

Generation two of our work resulted in the development of contrast trees. It uses a recursive clustering method discussed above followed by summarizing these recursive divisions into a tree structure. Although the initial results with contrast tree were weak, they were somewhat positive. We tried to extend and improve the contrast tree prototype, however the results were not encouraging. The tree-based approach suffered from one key issue: the local changes were highly ineffective in reducing the defects, see figure **??**. In addition, it also faced the optimization failure problem, which was also seen with W2.

As a continuation of the contrast tree work, we built HOW as a simpler approach that was to provide a baseline result, above which contrast tree was meant to do better. However, HOWs results were so good that we threw away months of work on tree-based planning with contrast tree.

[1]Available from the object-oriented defects section of the PROMISE respository openscience.us/repo/defect/ck.

## III. Future Work

Now, we strongly recommend HOW over contrast tree (and W2) since, as shown by the following results, HOWs plans never lead to performance getting worse. Also, when HOW did improve the expected values of the performance, those performance improvements were an order of magnitude larger than those seen with contrast tree.