Data Mining Programming Assignment 3 Report

Gregory Star -- starx013 -- 12/14/2018

Implementation

First data is read in and stored: the class labels are stored in an array, and the values are stored in a double array of type float (this is to ensure generality, since rep2 does not contain integers). The goal of the algorithm is to construct a decision tree, and my decision tree is represented as a binary tree consisting of nodes. Each node has an attribute called set (among other attributes, like label, id, etc.), which is the set of points that reached this point in the decision tree (more on how this set is used later). The set is stored as an ArrayList of indices which reference the actual dataset. This saves a lot of time and space, because the actual dimensions of the data points do not need to be stored in more than one place; it could probably be made even more efficient by using arrays rather than ArrayLists to represent these sets.

A root node is created for the decision tree and its set is initialized to all indices, then training begins. createDecisionTree() first checks the stopping conditions for the given node. If any of them are satisfied, then the label of the node is computed, the set of the node is wiped since it's no longer needed, and the function returns. Otherwise, the Gini index for every possible split (which hasn't already been used) is computed. For each feature, bestSplitForFeature() is called, which finds the optimal threshold for the given feature and computes the associated Gini index.

bestSplitForFeature() first sorts the splitting column, but maintains the true indices of the sorted list (i.e. if the data point at index 3 gets moved to index 10 in the sorted list, we want to keep track of its original index). Then, every point is chosen as a threshold in order. Sorting the column prior to computing splits allows us to skip points which are the same, which is especially relevant in rep1 where there are only 256 unique values for 60000 points. The threshold constitutes a split, and the Gini index of each such split is computed. To make computing the Gini index fast, the class count matrix for the over threshold set and under threshold set is initialized at the beginning of the loop, and instead of recomputing the whole thing, elements are decremented or incremented depending on the new threshold. At the end, the best Gini index is stored, and the best over/under split is stored (the over under split is stored as a Boolean array, which is a fast way to represent set membership).

Once the best split feature and corresponding threshold are determined, createDecisionTree() creates two sets based on the split, wipes its own set (now that those two sets exist, its own set is unnecessary), creates a new node, assigns the under threshold set to the new node, adds the split feature to a set of used features, and then recursively calls itself on that node. It then does the exact same thing but for the over threshold set. Note that by the time the right node (over threshold set) is computed, the set in the left node will have been wiped, so there's less memory usage. Finally, the feature that was used to split on is removed from the usedFeatures set, so that sibling nodes in the tree will still be able to use that feature (usedFeatures is global).

Once the decision tree is constructed, it must be stored in a file. I used a well-known motif wherein you store both the preorder and inorder traversal of the tree; this gives enough information to later reconstruct the tree without much difficulty. The file is formatted such that the first line of the file is the preorder traversal, and the second line is the inorder traversal (obviously leading to quite long lines). The nodes in the traversal are delimited by spaces, and the data within each node is delimited by commas. Every non-leaf node follows the format:

```
id,label,feature,threshold
```

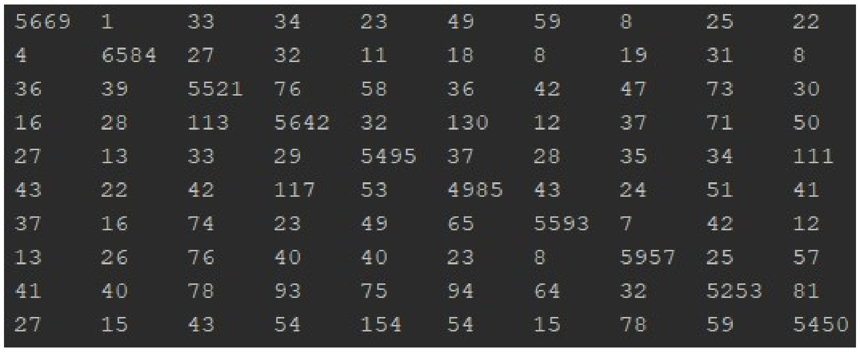
And every leaf node follows the format:

```
id,label
```

The id is used to uniquely identify each node, so that a standard tree-reconstruction algorithm can be used. The label of every non-leaf node is -1. Note that for the prediction file, we are told that the file should consist of two columns, with the first column being the true labels and the second column being the predicted labels; it was not specified how the columns should be delimited though, so I chose to use spaces (as opposed to commas).

Methodology

For both representations of the data set, 4 different minfreq values were tested (1, 5, 10, 20). Since there are 2 data representations and 4 minfreq values to test for each, 8 trials were performed. For each trial, the dtinduce program reads in a training set and generates the corresponding decision tree, which is then read by dtclassify to generate a prediction file for a test set, and finally that prediction file is read by showconfmatrix which outputs a confusion matrix and accuracy. Below is an example of a confusion matrix printed to the terminal by showconfmatrix:



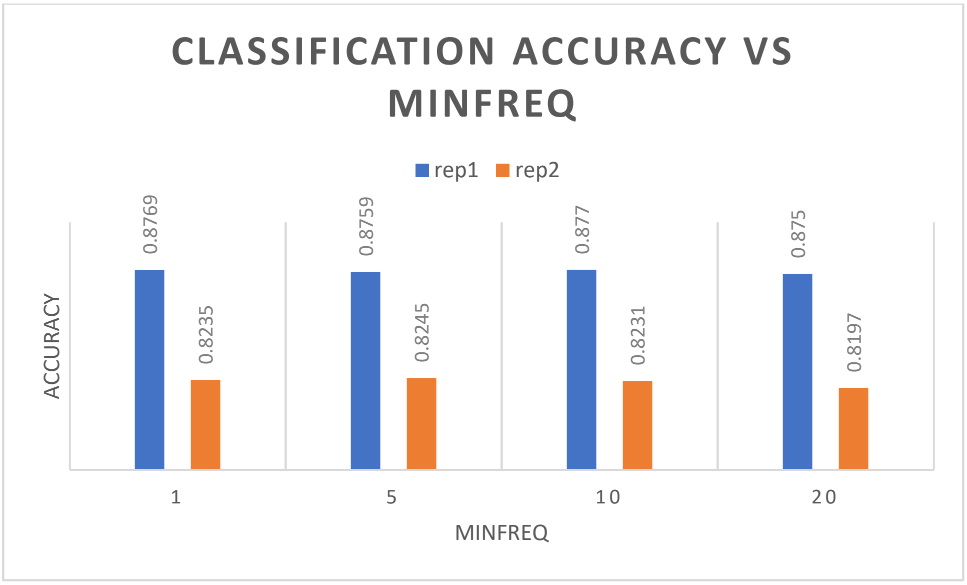
```
5669  1     33    34    23    49    59    8     25    22
4     6584  27    32    11    18    8     19    31    8
36    39    5521  76    58    36    42    47    73    30
16    28    113   5642  32    130   12    37    71    50
27    13    33    29    5495  37    28    35    34    111
43    22    42    117   53    4985  43    24    51    41
37    16    74    23    49    65    5593  7     42    12
13    26    76    40    40    23    8     5957  25    57
41    40    78    93    75    94    64    32    5253  81
27    15    43    54    154   54    15    78    59    5450
```

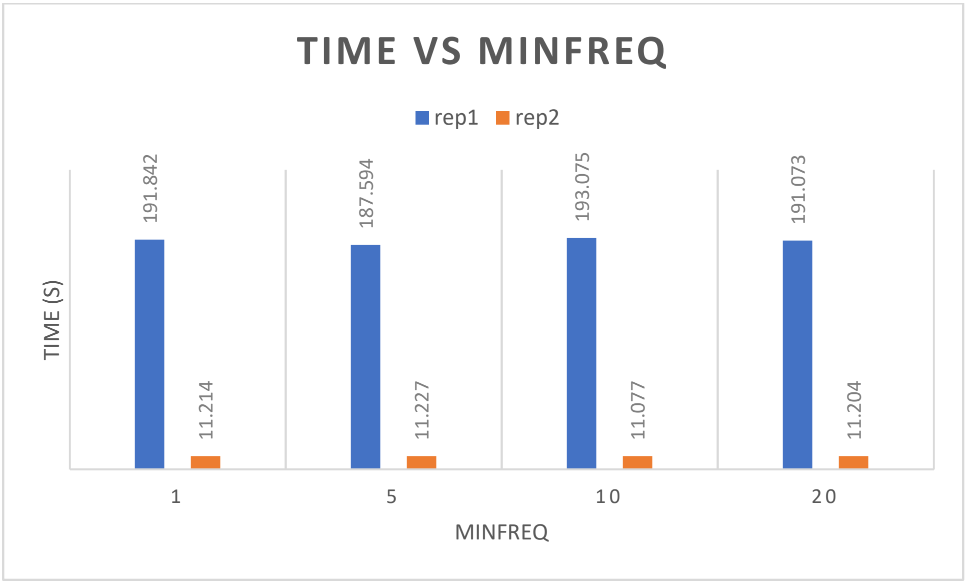
Note that index ij of this matrix represents the number of points whose real label is I - 1 and predicted label is j – 1 (the -1 is there because the digits being classified are 0-9 and matrix indices start at 1). This is not specified in the terminal output because the confusion matrix discussed in class is defined in the same manner, but I specify this here because in some texts i - 1 is the predicted label and j - 1 is the real label.

All tests were performed via vole.cse.umn.edu, which (to my knowledge) utilizes CSE lab machines. Curiously, runtimes appeared to be slightly faster on my personal machine, though not substantially so (my machine is markedly slower than a CSE lab machine). This is likely due to some quirk with vole; I expect run times to be faster when running these test on a CSE lab machine and being physically present.

Data

Below are bar charts which show the accuracies achieved on the two data representation for all minfreq values, as well as the time taken:





We can see that the decision tree is able to correctly classify the rep1 test set at about a 5% better rate than the rep2 test set. Better performance on the rep1 data set is to be expected, because it contains vastly more information; since the rep1 data set contains 784 dimensions and the rep2 data set only contains 20, the rep1 decision tree was trained on 39.2 times more data than the rep2 decision tree (*this* *isn't strictly true as the rep1 data set only has values ranging from 0 to 255 and could therefore be compressed, but since my program treats all values as floats for generality this can't be taken advantage of*). Of course, it could be argued that much of that extra data is not useful for classification, and indeed that is what we see; a 5% reduction in classification accuracy is not too high of a price to pay for such a great reduction in storage space and computation time, which shows that the first 20 principle components are a good representation of the data set. It should be noted that runtimes are only about 18 times worse for the rep1 data set than the rep2 data set, and this is likely because the rep1 data set has fewer unique values (and thus the training algorithm can skip many Gini index computations).

Interestingly, there appears to be no correlation between the minfreq and classification accuracy for the tested values. I expected that lower minfreq values could possibly lead to overfitting, which would reduce the accuracy on the test set. However, this does not appear to be the case. Since for both data sets we see little difference between the tested minfreq values, it makes sense to choose minfreq 10 or 20 which are the simpler models.

That being said, I tested a few larger values for minfreq which are not plotted here (such as 50 and 100) and found that classification accuracy started to decrease as minfreq gets large. Therefore, I am able to observe underfitting occurring when choosing too large of a minfreq value, but curiously am unable to observe overfitting occurring (model complexity is capped at minfreq 1, so the tests indicate overfitting can't happen in this setup). It is possible that this is the result of the chosen test set, but my hypothesis is that these results may be inherent to this data set: it seems that most sets with less than 20 elements in them tend to be pure anyways, so the possibility of further splitting with lower minfreq values has little effect.

Under the conditions I ran these tests, I was also unable to detect any correlation between minfreq values and runtimes. In theory, lower minfreq values should have higher run times since they have to run more splits; however, splitting 20 points is near instantaneous, and most sets of that size will be pure anyways. Other factors outside of my control are likely to influence runtime far more than the difference between minfreq 20 and minfreq 1, which appears to have happened in my tests. I did find some appreciable difference when going up to minfreq 100 and higher, but these values are not plotted here because they are outside the scope of the required tests.