Joseph Barr, Sunil Green, Ashley Konger

Team 38

CS373

16 April 2021

Final Report

**Definition of the problem:**

Over the course of the semester, we studied the problem of “pork barrel” legislation. Pork barrel legislation is wasteful legislation that misappropriates federal resources, often for local or special interests.

**Data Preprocessing**

In order to simplify our data, we focused our analysis on legislation proposed in the House of Representatives alone. This was done because the House of Representatives has a higher likelihood of passing pork barrel bills (since members represent smaller districts, rather than entire states). We hoped that, by considering the features below, we could develop a model capable of predicting whether novel legislation is “pork barreled”. Our data, in addition to the key column bill\_id and the label column is\_pork, includes:

* cosponsors: the number of representatives who cosponsored the bill
* sponsor\_party: the party affiliation of the representative who introduced the bill
* sponsor\_state: the home state of the representative who introduced the bill
* committee\_codes: a list of the committees to which the bill was referred
* subcommittee\_codes: a list of the subcommittees to which the bill was referred
* primary\_subject: the main subject that the bill addresses (e.g., ‘Health’)

## *Acquisition*

We aggregated our data from a few different sources. First, we used [OpenSecrets.org](https://www.opensecrets.org/earmarks/index.php) to find information on which Representatives received the most contributions from lobbying, PACs, and individuals as a fraction of the amount of funding that went to the interests represented by the lobbyists, PACs, and individuals. Essentially, we considered the bills that produced the best “return on investment” for Representatives to be the most pork barreled, and thus labeled these bills as pork barreled. Next, we used the [Citizens Against Government Waste Earmark Database](https://www.cagw.org/reporting/earmarks) to get more detailed data on specific pork barrel bills. Finally, we used the bill search available on [congress.gov](https://www.congress.gov/) to find bills that are not considered pork barreled. Bills we identified as not being pork barreled did not contain any Congressional spending. Bills from the years 2008-2021 were used in our data set. We used the [ProPublica API](https://projects.propublica.org/api-docs/congress-api/bills/) to gather data from all the bills identified (both as pork barreled and not pork barreled) to complete our dataset. A final note, much of the pork barreling that goes on in government occurs when Congress passes its yearly budget through a series of appropriations bills. As such, a large amount of appropriation bills constitute the identified pork barreled bills in our dataset.

## 

## *Transformation of Non-Atomic Features*

The final two features above—committee\_codes and subcommittee\_codes— contain lists of values, rather than atomic values. In order to resolve this, we transformed them by adding a binary column for each distinct value found in any list. For example, the HSFA column contains a “1” for any bill submitted under the HSFA committee, and a “0” for all other bills. After this modification, we were able to remove the original columns.

*Encoding of Categorical Features*

One problem we encountered in our data was the abundance of categorical features, which are not supported by scikit. A naïve solution to this would have been to encode them by incrementally assigning each possible value a corresponding integer key (akin to a C-style enum), but this has the potential issue of implying some quantitative significance. Instead, we used [‘One-Hot Encoding’](https://towardsdatascience.com/categorical-encoding-using-label-encoding-and-one-hot-encoder-911ef77fb5bd) (which is, in practice, identical to the transformation given in the previous section), which does not assign a significance to the numerical values. Each possible value for a categorical variable is transformed into a column (feature) in our dataset and given a value of “0” or “1”.

After performing both of these preprocessing steps, our data has a total of 168 features. These features are based on the 6 main features we identified outlined above, and the reason our number of features is so high is because of One-Hot Encoding’s transformation of our categorical variables and the topic’s reliance on categorical data.

# **CART**

# *Tuning Hyperparamters*

One of the algorithms we chose to implement was the CART classification tree algorithm. We used the scikit-learn python library to implement the CART algorithm, found [here](https://scikit-learn.org/stable/modules/tree.html). This implementation of the CART algorithm uses a hyperparameter called the gini threshold. The threshold is passed into the DecisionTreeClassifier function and we pass an argument in cart.train() to determine it. The different splits in the classification tree are constructed using a minimum gini threshold passed into the CART algorithm, and if no value is passed in, the threshold defaults to 0.0. The algorithm splits on the gini threshold it determines yields the highest information gain. We tuned the gini hyperparameter using 4-fold cross validation. Each iteration starts at 0.0 and passes in iteratively higher gini thresholds, and then the corresponding error is measured. We found that low values of gini resulted in smaller error values, and that it was very sensitive to small increases. The code for tuning can be found in model.tune\_cart().

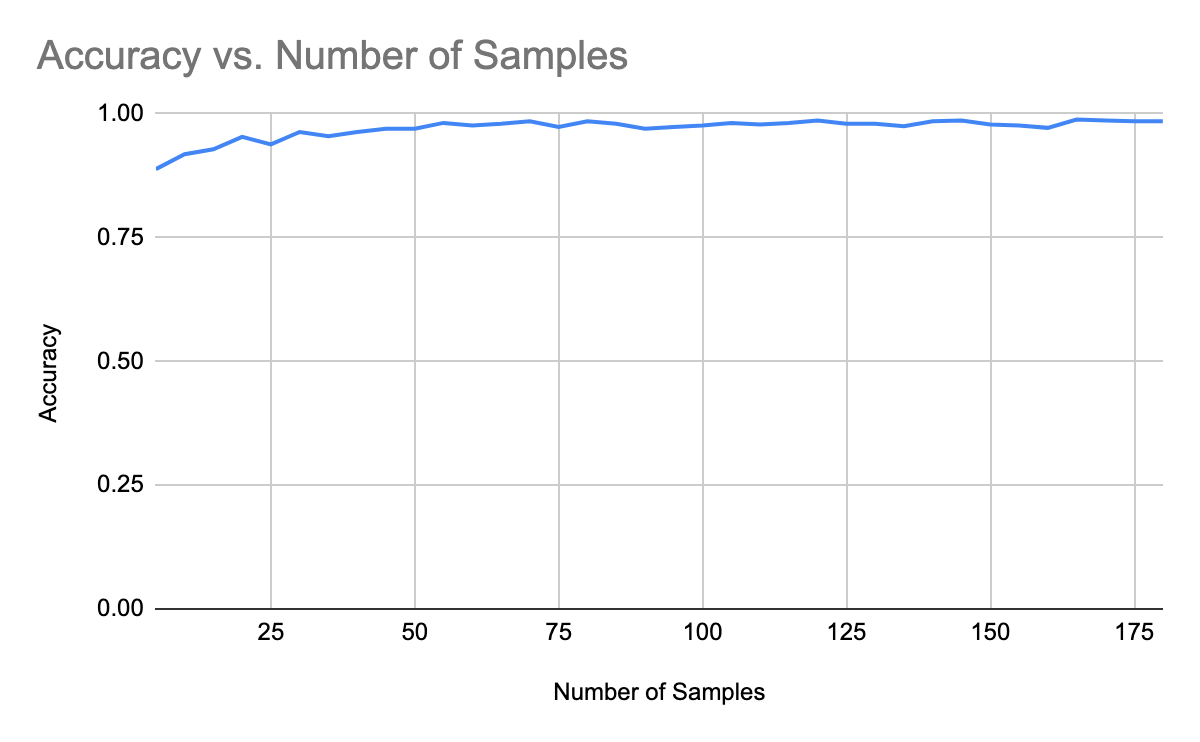
Summary of gini threshold hyperparameter tuning:

|  |  |  |
| --- | --- | --- |
| **Gini Threshold** | **Error Values** | **Mean Error** |
| 0.000 | 0.0, 0.0, 0.0, 0.06 | 0.015 |
| 0.001 | 0.0, 0.0, 0.0, 0.06 | 0.015 |
| 0.010 | 0.0, 0.0, 0.0, 0.06 | 0.015 |
| 0.020 | 0.0, 0.0, 0.0, 0.06 | 0.015 |
| 0.026 | 0.0, 0.0, 0.0, 0.06 | 0.015 |
| 0.027 | 0.0, 0.0, 0.0, 0.38 | 0.095 |
| 0.127 | 0.0, 0.0, 0.0, 0.38 | 0.095 |
| 0.182 | 0.0, 0.0, 0.0, 0.38 | 0.095 |
| 0.183 | 0.0, 0.0, 0.04, 0.38 | 0.105 |
| 0.229 | 0.0, 0.0, 0.04, 0.38 | 0.105 |

Interestingly, the lower gini thresholds had higher accuracy. A few inflection points in the data were found. At a gini threshold of 0.027 and up the mean error increased by 0.08. At a gini threshold of 0.183 and higher, there was another mean error increase of 0.01. Values in between the ranges of 0.0 to 0.026, 0.027 to 0.183, and 0.183 and above remained constant.

# 

Below we have charted the accuracy of our model against the number of samples used in training. The details are spelled out in model.py, but broadly, we iteratively took randomized subsets of *n* = 5, 10, 15, …, 180 samples from our dataset and trained the cart model on them. We also took 20 more random samples to use as test data (taking care to ensure no training data was used as test data as well), and used our model to predict whether these samples are pork barreled. For each n, we repeated this process 30 times, and took the average accuracy (where accuracy is the proportion of samples classified correctly). That average accuracy is plotted on the graph, against the relevant subset size *n*.



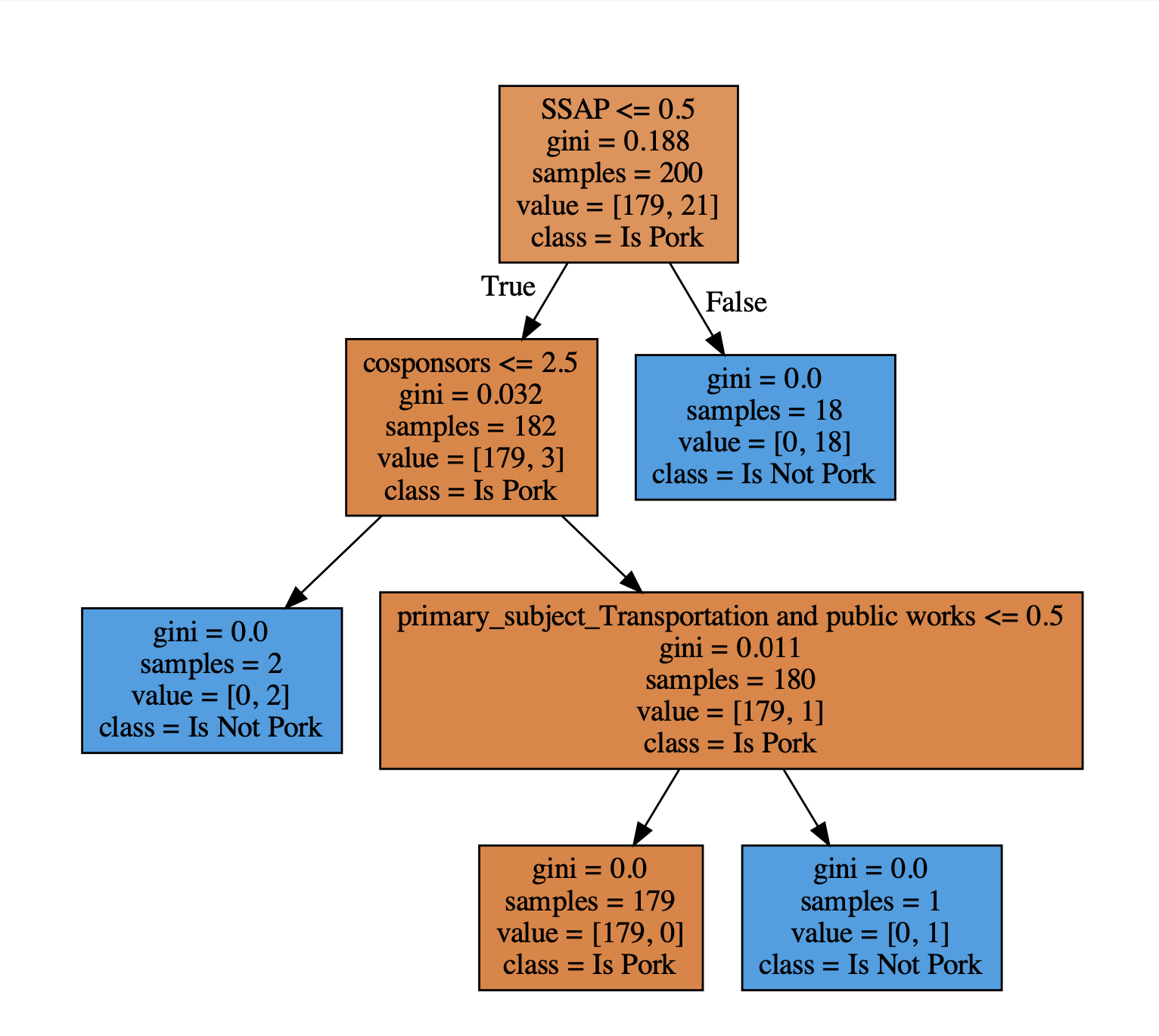
The accuracy measured starts plateaus at 100% around n=60. As such, we determine 60 to be the appropriate number of samples to use in cart.py. The data is generated by calling model.subsets(False). Note that the process is randomized, so re-running this function will generally provide slightly different results.

Following a similar iterative and random sampling approach, we tested a number of different gini thresholds and measured the resulting accuracy. As apparent by the graph, the accuracy is almost constant irrespective of gini threshold used. The highest accuracy thresholds are 0.0, and 0.985, and 0.015 which had accuracies of 0.99. From this, our takeaway is that the default value of 0.0 should work just fine. The data is generated by calling model.thresholds(False)

# **Chart**

# *Visual representation of classification trees*

By using graphviz, we are able to visually display a resulting tree from cart.py. This tree is constructed using the default gini threshold of 0.0.



**C4.5**

Since scikit-learn does not offer an implementation of the C4.5 algorithm out of the box, we used an implementation that follows the same input and output paradigms as scikit’s CART algorithm designed by [RaczeQ](https://github.com/RaczeQ/scikit-learn-C4.5-tree-classifier). However, this implementation did not allow for the tuning of the information gain threshold, so we had to make some minor tweaks to the source code. Namely, we had to change the grow\_tree method so that, when deciding when to split, it compares against an information gain threshold that we provide as an argument, rather than the threshold it calculates itself.

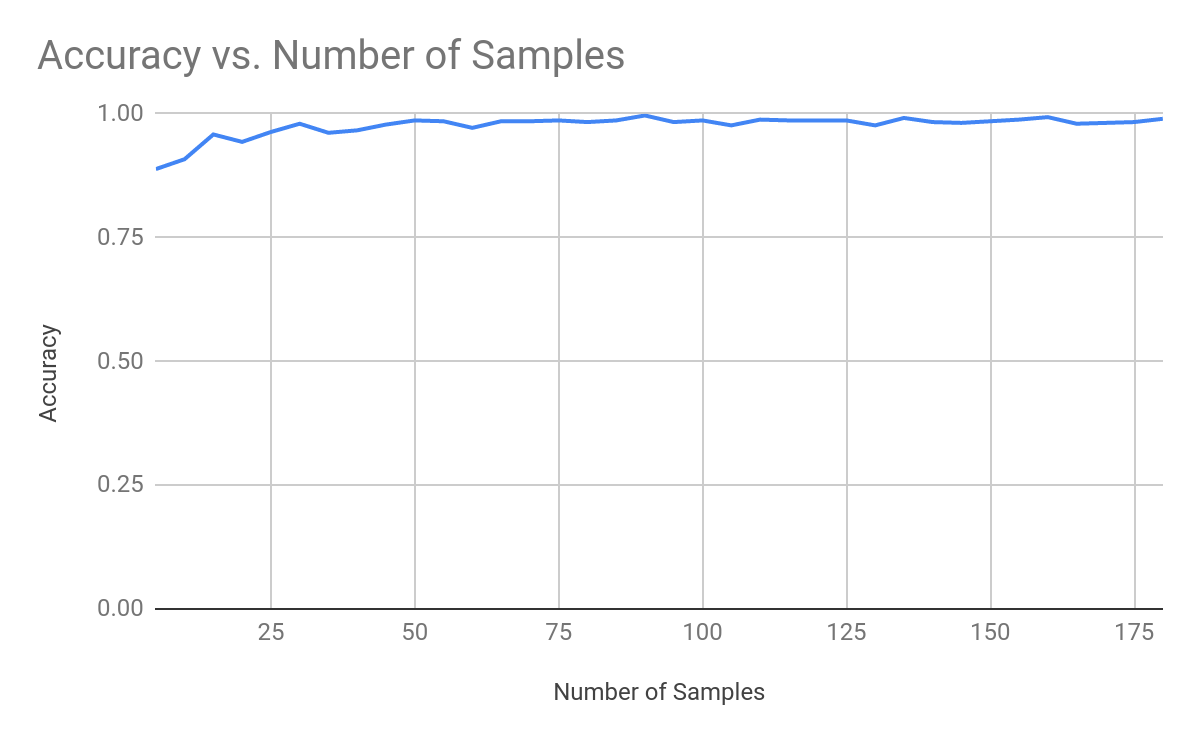
*Tuning Hyperparameters*

We tune the hyperparameter “information gain threshold” by running 4-fold cross-validation with different threshold values. By injecting print statements, we determined that the thresholds were always integer values. Thus, to tune our hyperparameter, we can simply iterate through a sequence of integers until we find a stable error value, and then determine which corresponds to the lowest error when running 4-fold cross-validation, and that threshold value will be our optimum. This is implemented in model.tune\_C45().

A summary of the findings from tuning the hyperparameters is given here:

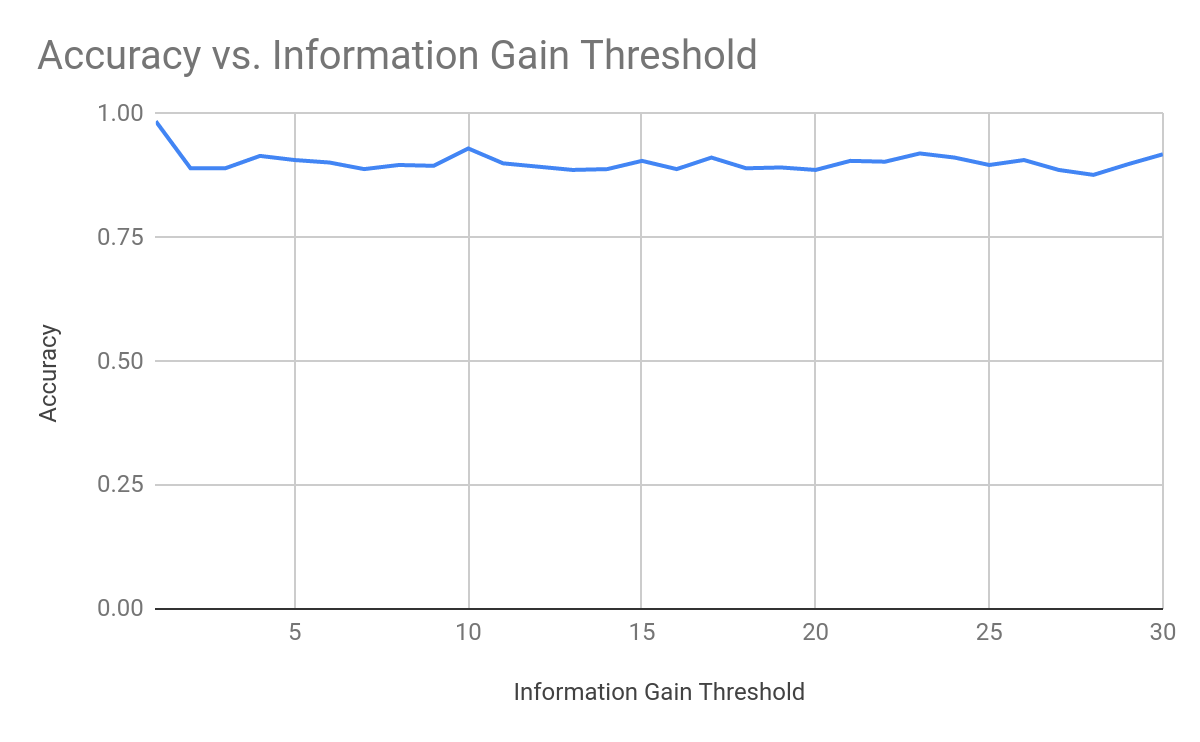
|  |  |  |
| --- | --- | --- |
| **Threshold** | **Error Values** | **Mean Error** |
| 1.0 | 0.0, 0.0, 0.0, 0.06 | 0.015 |
| 2.0 | 0.0, 0.0, 0.04, 0.38 | 0.105 |
| 3.0 | 0.0, 0.0, 0.04, 0.38 | 0.105 |
| 4.0 | 0.0, 0.0, 0.04, 0.38 | 0.105 |
| 5.0 | 0.0, 0.0, 0.04, 0.38 | 0.105 |
| 6.0 | 0.0, 0.0, 0.04, 0.38 | 0.105 |
| 7.0 | 0.0, 0.0, 0.04, 0.38 | 0.105 |
| 8.0 | 0.0, 0.0, 0.04, 0.38 | 0.105 |
| 9.0 | 0.0, 0.0, 0.04, 0.38 | 0.105 |
| 10.0 | 0.0, 0.0, 0.04, 0.38 | 0.105 |

Empirically, we see that the error stabilizes quite quickly, and that the errors are uniformly higher for thresholds greater than 1.0 than they are when the threshold equals 1.0. So, we should use 1.0 as our information gain threshold.

Here we followed the same steps used in plotting Accuracy vs. Number of Samples as used previously for Cart. This time, however, instead of training on Cart we trained on the C45 algorithm. By iteratively increasing the number of randomly selected samples and training and testing on separate subsets of the data, we plotted our Accuracy vs. Number of Samples below. 

As one can see, the accuracy starts fairly high and stays high. It is worth noting that it reaches a plateau near 100% accuracy at about 50 samples, and the accuracy is generally lower for smaller subsets. From this data, we can conclude that training the C4.5 model with at least 50 samples gives as high an accuracy as we can reasonably expect. So, using significantly more than 50 samples is wasteful of resources, but using fewer than 50 gives a less than optimal model. The data is generated by calling model.subsets(True). Note that the process is randomized, so re-running this function will generally provide slightly different results.

Following a similar procedure as above, we test the model’s accuracy against different values of the hyperparameter “Information Gain Threshold”



It would appear the only threshold that gives a significantly different accuracy is in fact 1.0. All the other thresholds give lower accuracies, and the trend seems to hold steady. So, from this data, we can conclude that 1.0 would be the best information gain threshold to choose. The data is generated by calling model.thresholds(True).

**Additional Notes**

The C45 algorithm did not allow us to graphically chart the resulting classification tree as Cart did. In addition, we attribute our relatively high accuracies to bias in data selection, and time constraints. In Congress’ actual functioning, many thousand more bills are proposed than the ones analyzed, and as a result there is likely a lower concentration of pork barreled bills among those. A more extensive analysis and a lot more data would be required to simulate “real world” conditions.