University of New Mexico

Project 1: Random Forests

CS 529: Introduction to Machine Learning
Professor Trilce Piedras-Estrada

Prasanth Guvvala

Thomas Fisher

9 March 2024

Analysis

1. What is the difference between the resulting trees that each IG method produced? Compare and contrast in terms of accuracy and structure.

We ran two workloads, one each with the hyper parameters being constant among the workloads. The results are:

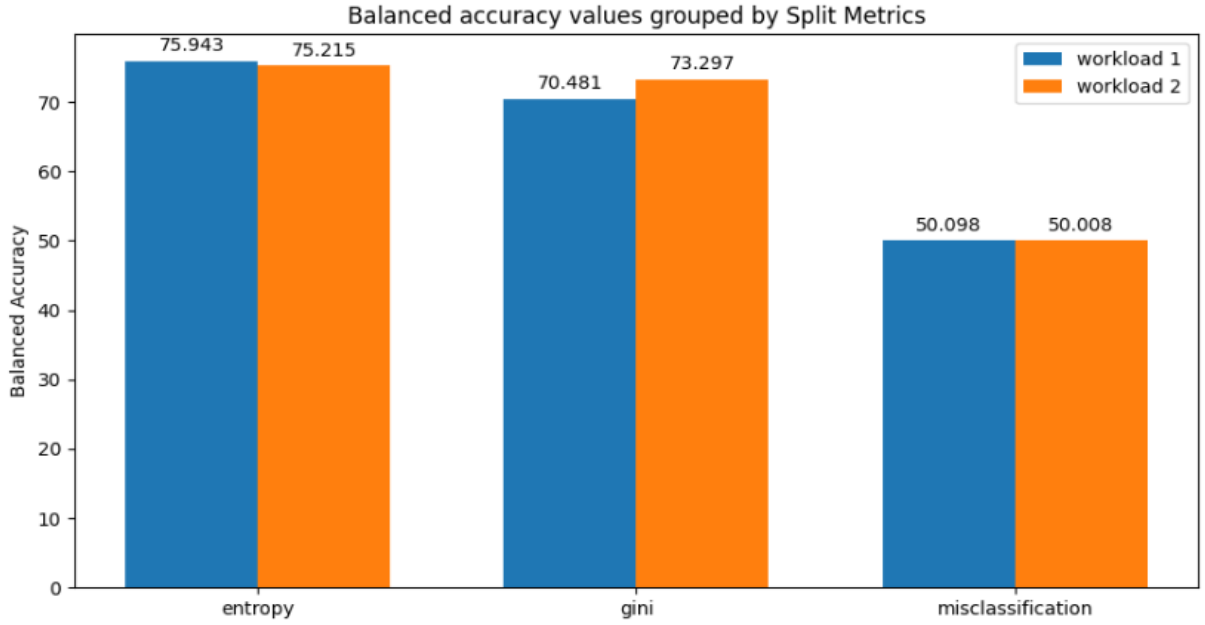
Workload 1:

Split Metric	Runtime (seconds)	Accuracy (%)
Entropy	8497.48	75.94
Gini	7165.29	70.48
Misclassification	26273.64	50.10

Workload 2:

Split Metric	Runtime (seconds)	Accuracy
Entropy	40031.77	75.22
Gini	23551.78	73.30
Misclassification	27290.05	50.01

The split metrics are plotted against the accuracy scores using the bar plot as shown in the figure below:



Since we talk about the forest, the structure of the trees in the forest is being represented using the average metric among the trees.

Workload 1:

Split Metric	Average node Count	Average Depth
Entropy	9100	19.7
Gini	19749.4	20
Misclassification	119.6	20

Workload 2:

Split Metric	Average node Count	Average Depth
Entropy	11445.3	19.6
Gini	14595.4	20
Misclassification	3698	20

2. How did the value of alpha affect Chi Square termination? Compare and contrast in terms of accuracy and structure.

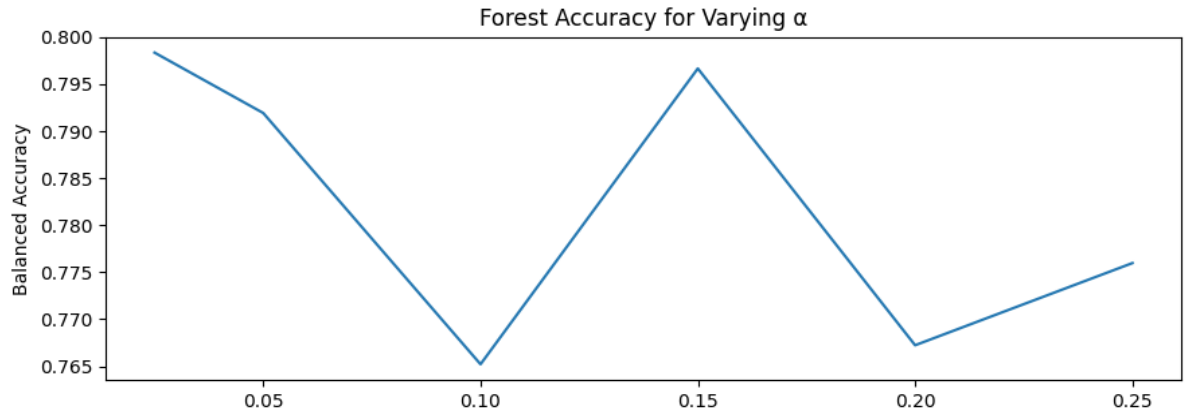
We found the α value used in our Chi Square termination criteria to have a significant impact on the accuracy and speed of our forest training. Below are shown runtime and accuracy numbers for different values of α when training a forest of 10 trees.

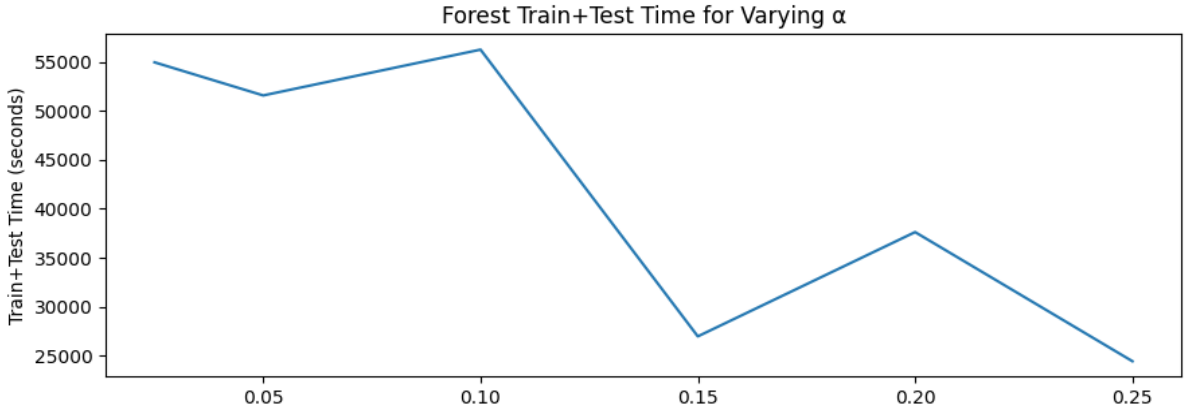
α Value	Runtime (seconds)	Accuracy (%)
0.25	24428.71	77.60
0.20	37620.87	76.72
0.15	26967.30	79.67
0.1	56262.95	76.52
0.05	51581.64	79.19
0.025	54964.20	79.84

We can see that increasing the α value from 0.025 (the smallest value we tested) generally lowers the accuracy of the model when all other hyperparameters are held constant. The change isn't large as a percentage of the total, but a clear trend is visible.

Increasing the value of α was also observed to decrease the runtime (training + validation time) required to generate the forest. This isn't consistent with our understanding of the role of the α value, since we'd expect a higher alpha to allow more splits and thus create larger trees taking more time. However, we identified several reasons why our timings may not be reflective of our algorithm alone. The CS department machines we used to run our training were in high demand by other students in the class, and there would often be imbalanced loads between machines running different parameter set trainings. Also, as previously mentioned, a larger number of continuous features resulted in longer runtimes, meaning that the random feature bagging for a particular run could heavily influence the amount of time it would take to complete.

See below for graphs of accuracy and runtime for varying values of α.





Our chosen value of α also influenced the structure of the trees themselves. The table below shows some structural analysis performed across 10-tree forests. Contrary to what we expected, larger α values actually caused the generated trees to have a lower depth. This could be caused by the features selected for each particular runs, where features that are more grouped by class in the training data naturally lead to later Chi Square test terminations.

α Value	Avg. Node Count	Average Tree Depth
0.25	7857.9	59.2
0.20	15679.2	63.9
0.15	18036.1	68.3

3. Which Information Gain Criteria method did you choose for your Final (kaggle submission) Random Forest? Why?

For our final submission to kaggle, we chose to use a random forest which was trained using the entropy information gain criteria methods. To arrive at this conclusion, we trained forests with different information gain criteria methods while keeping all other hyperparameters constant. The table below shows the difference in accuracies we observed for this type of test.

Split Metric	Accuracy (%)
Entropy	75.94
Gini	70.48
Misclassification	50.10

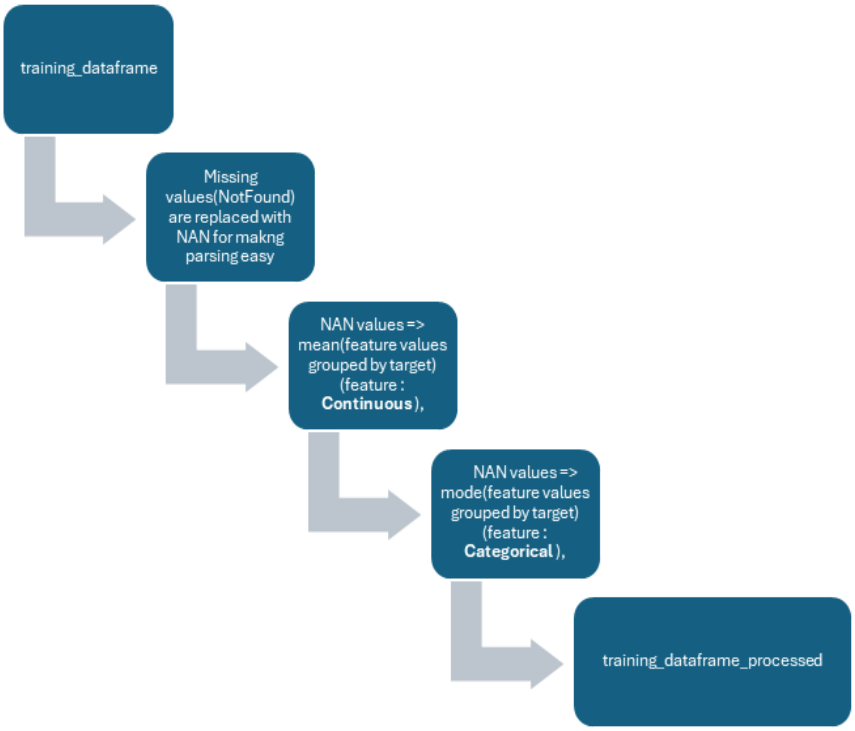
Clearly, entropy performs best, while Gini index is a near second and misclassification is a distant third.

4. What did you do to account for missing data?

The handling of missing values takes place as explained below. The features are divided into two groups categorical and continuous features.

1) For categorical features, the missing values are replaced with the mode value of the feature according to the target i.e., given a target what is the mode of the feature values that contribute to the specific value of target .
2) For continuous features, the missing values are being replaced with the mean given that all the feature values in a group contribute to the same target.
3) Missing targets in the predictions can also occur. In our implementation, if the tree search hasn't found a way to get to the leaf nodes for a decision, we kind of treat that immaturity as a label of value 2 i.e. the target that has value 2. These (2 values in the targets) are basically the targets for which the decision tree wouldn't decide what the target class pertaining to the given set of feature values.

Algorithm:



By calculating the mean of continuous feature values grouped by a target class enables the algorithms to maintain or preserve the underlying relationship between the target class and feature values. When replaced the replaced values represent the class that they contribute to given target information and feature values.

We found this method to be more efficient than simply replacing the mean of the values of the feature without considering the target class.

Calculating the mode of categorical feature values grouped by a target class enables the algorithms to maintain or preserve the underlying relationship between the target class and feature values. When replaced the majority of the replaced values represent categorical class that they contribute to given target information and feature values.

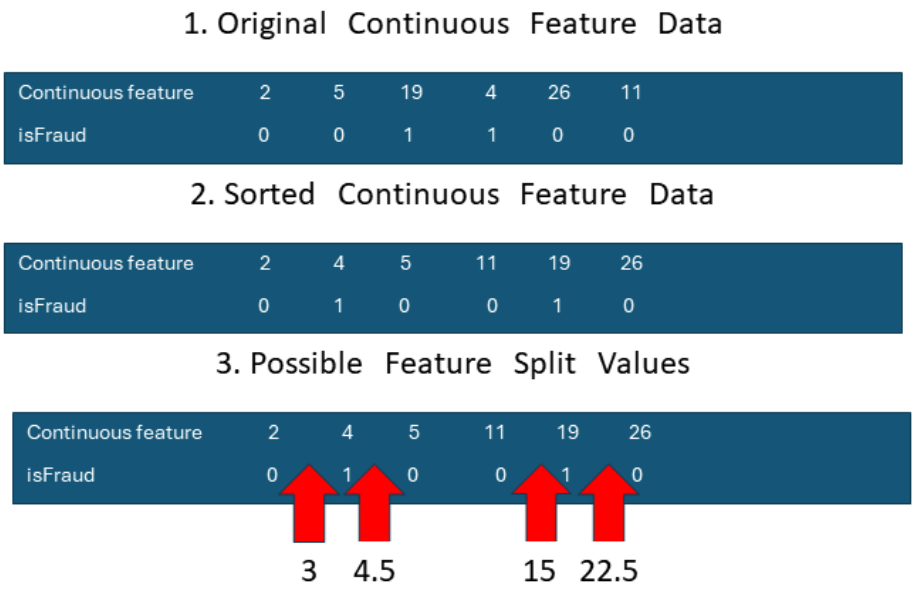
We found this method to be more efficient than simply replacing the mean of the values of the feature without considering the target class.

5. What did you do to account for numerical features?

Computing the information gain for numerical features, especially at low tree depths near the root, was the performance bottleneck for our decision tree and random forest training algorithm. For this reason, we were very intentional in how we implemented this aspect of our project.

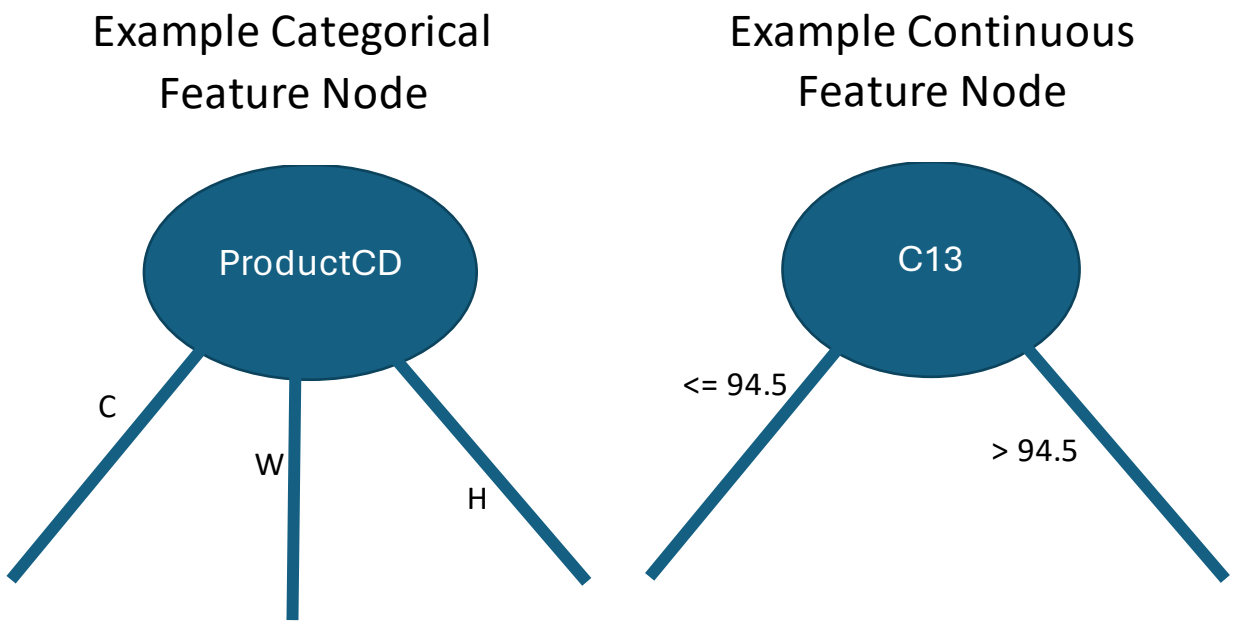
To calculate the information gain of continuous features, we employed the method described in pages 72-73 of Mitchell's *Machine Learning*. First, the values of the continuous feature are sorted, and the corresponding class of each feature value is noted. Next, each pair of adjacent sorted feature values which belong to dissimilar classes are chosen as possible points between which to split the feature, and the average of the two values is calculated. The information gain from splitting the feature using the average is recorded, and the greatest information gain found using these adjacent pairs is used to determine which value upon which to split the feature. This process is shown visually in the figure below.

Due to the amount of time our algorithm spent on the process above, we implemented a performance optimization which is described in Part 7 of our analysis.



Continuous features also required a slight modification to our custom tree classes written in Python. One of our classes represented a tree node and consisted of a node label, which was populated with the feature name, and a list of branch objects, which contained the distinct feature values in the case of a categorical feature. For continuous features, however, each node required 2 branches only, one which represented "less than" the split point of the feature, and another which represented "greater than." To implement this, the appropriate inequality operator (">" or "<=") was inserted before the split feature value,

and this string was used as the branch label. The figure below shows a visual example of the difference between tree node structures for categorical and continuous features.



A last difference in how our handling of continuous features differed from that of categorical features is whether we allowed a feature to be repeatedly used in any path of the tree. Our recursive calls to our build tree function included a set argument which contained all the categorical features previously used to split the dataset in the tree path so that they would not be considered again. Continuous features, however, were not added to the set argument, allowing them to be selected multiple times in the construction of tree paths.

6. *What did you do to account for class imbalance?*

There are two ways in which our random forest implementation handles the fact that our provided dataset is heavily imbalanced.

Firstly, the way our data is sampled for each tree is performed in a way to give greater weight to the minority class. If a fraction $0.0 \leq f \leq 1.0$ of the total training rows are to be used on a particular tree, first all the positive instances are included, and then the

remaining rows needed to reach f are sampled with replacement from the negative instances. This ensures that each tree sees the entirety of positive samples and lessens the imbalance between the classes somewhat.

Secondly, we maintain a record of the imbalance in our initial training data used for each tree and use that value to weight counts of each class when calculating information gain and when performing the Chi Square test. Each time the number of instances associated with a particular class are totaled, the imbalance weight (of the original tree data) is multiplied by the number of the minority class to avoid the negative instances overwhelming all mode calculations.

7. What other optimizations did you implement to improve either accuracy or performance?

We implemented several optimizations to improve the accuracy and performance of our random forests beyond what we achieved using the general algorithm shown in class.

Firstly, we combined forest models together to form larger ensemble models which benefit from the greater number of total trees. In lecture, we learned that, if tree balanced accuracies are greater than 50%, adding more trees to the forest should only increase the overall accuracy. We took advantage of this fact by combining our smaller models into a larger one to produce our best accuracy.

Another accuracy optimization implemented by our team was how we handled missing values in our training data. Simple methods of handling missing values exist, such as removing all instances with missing values or using the mean over the entire feature. We chose a more sophisticated approach which involved determining the mean or mode (for continuous or categorical features respectively) of each feature for each class and using this value to replace missing feature values.

A performance optimization which greatly improved the runtime of our forest training routine was to identify the slowest part of our algorithm (information gain of continuous features) and use a Numba JIT decorator to compile the code into a faster format. The JIT decorator also allows for specifying a CUDA target for running the included code on a GPU. Using the CS Department machines with GPUs helped greatly in decreasing our training runtime.

We added another accuracy optimization in how we kept a record of our initial dataset's imbalance throughout the process of training each tree. We used this imbalance factor when calculating counts of class instances per feature for information gain and Chi Square test. Weighting the counts of class instances based on the proportions of each in the dataset prior to performing training improved our accuracy as before each mode/mean value was dominated by the majority class.

Lastly, we employed a random-based accuracy technique to improve our kaggle score. The CSV files we generated with our predictions for submission sometimes contained missing values. This was caused by the trees in the forest not being able to classify a testing instance due to it containing a feature value which was not seen during training. To maximize our score, we filled the gaps in our file with 0's and 1's (a separate file for each) and submitted both to kaggle.