9 Random Forests

Lab Objective: *Understand how to build and use a classification tree and a random forest.*

Classification Trees

Decision Classification trees are a class of decision trees used in a wide variety of settings where labeled training data is available. The desired outcome is a model that can accurately assign labels to unlabeled data. Decision trees are widely used because they have a fast run time, low computation cost, and can handle irrelevant, missing, and noisy data easily.

We begin with a data set of samples, such as information about customers from a certain store. Each sample contains a variety of features, such as if the individual is married or has children. The sample also has a classification label, such as whether or not the person made a specific purchase.

A classification tree is composed of many *nodes*, which ask a question (i.e. "Is income $>=$ 85?") and then split the data based on the answers. If the response is `True`, then the sample is "pushed" down the tree to the left child node. If the response is `False`, then the sample is "pushed" down the tree to the right child node. A *leaf* node is a node that has no child node. Upon arrival at a leaf, an unlabeled sample is labeled with the classification that matches the majority of labeled samples at that leaf. Table 9.1 includes information about 10 individuals and then an indicator of whether or not they made a certain purchase. To simplify construction of the tree, all data is numeric, so 1=Yes and 0=No for yes/no questions.

Suppose we wanted to guess whether a single college student making under $30,000 would purchase this item. Starting at the top of the tree, we compare our sample to the question and first choose the right branch, and then we compare with the second question and choose the right branch again. Now we reach a leaf with the dictionary `{0:1}`. The key `0` corresponds to the label, and the value `1` means one of our original samples is at this leaf with that label. Since 100% of samples at this leaf are labeled with `0`, our new sample college student will be predicted to share the label `0`.

If we arrived instead at a leaf with the dictionary `{0:1, 1:4}`, then one of our original samples at this leaf would be labeled `0` and four would be labeled `1`, so the majority vote would assign the label `1` to our new sample.

Married (Y/N)	Children	Income ($1000)	Purchased (Y/N)
0	5	125	0
1	0	100	0
0	0	70	0
1	3	120	0
0	0	95	1
1	0	60	0
0	2	220	1
0	0	85	1
1	0	75	0
0	0	90	1

Table 9.1: Customer data with 3 features (Married, Children, Income) and a label (Purchase) indicating whether or not the customer bought the item.

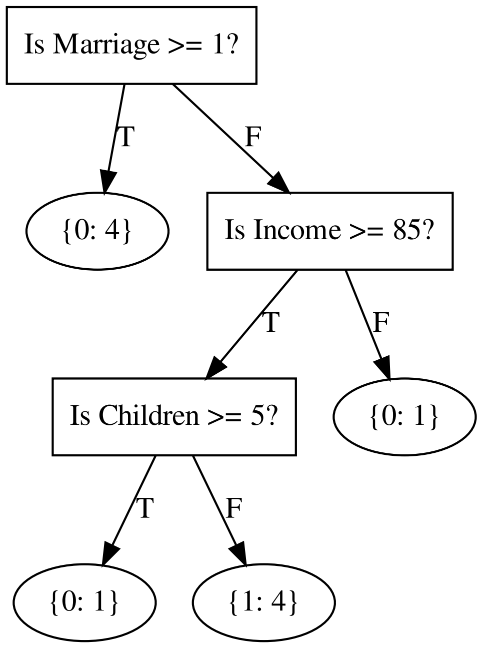

Figure 9.1: A classification tree built using Table 9.1. Each leaf includes a dictionary of the label (0 or 1) and how many individuals from the data match the classification. In this example, each leaf contains individuals with only one label.

Problem 1. At each node in a classification tree, a question indicates which branch a sample belongs to. Write a `Match` method for the class `Question` that accepts a sample and returns `True` or `False` depending on how the sample's features compare to the question. For example, in the example above, a single college student making $20,000 would be a sample represented by the array `[0, 0, 20]`.

Next, write a `partition` function that partitions a data set for a given question. Return the left and right regions of the partition in that order. If one region is empty, return it as `None`.

Measures

To use the `partition` function from Problem 1, we need to know which question to ask at each node. Usually, the question is determined by the split that maximizes either the Gini impurity or the information gain. Gini impurity measures how often a sample would be mislabeled based on the distribution of labels. It is a measure of homogeneity of labels, so it is 0 when all samples at a node have the same label.

Definition 9.1. *Let D be a data set with K different class labels and N different samples. Let N_k be the number of samples labeled class k for each $1 \leq k \leq K$, and let $f_k = \frac{N_k}{N}$. We define the* Gini impurity *to be*

$$G(D) = 1 - \sum_{k=1}^{K} f_k^2.$$

Information gain is based on the concept of Information Theory entropy. It measures the difference between two probability distributions. If the distributions are equal, then the information gain is 0. We will use a modified version of information gain for simplicity.

Definition 9.2. *Let $s_D(p, x) = D_1, D_2$ be a partition of data D. We define the* information gain *of this partition to be*

$$I(s_D(p, x)) = G(D) - \sum_{i=1}^{2} \frac{|D_i|}{|D|} \cdot G(D_i)$$

where $|D|$ represents the number of samples (or rows) in D.

Problem 2. Write a function `gini()` that computes the Gini impurity of an array of data with the class labels in the last column. Write another function `info_gain()` that computes the information gain for a given split of data. Make sure these functions account for the case of the data array containing only a single sample.

The file `animals.csv` contains information about 7 features for 100 animals. The last column, the class labels, indicates whether or not an animal lives in the ocean. You may use this file to test your functions. To test your functions, your values should match those below.

```
>>> import numpy as np
# Load in the data
>>> animals = np.loadtxt('animals.csv', delimiter=',')
# Load in feature names
>>> features = np.loadtxt('animal_features.csv', delimiter=',', dtype=str,
...                       comments=None)
# Load in sample names
>>> names = np.loadtxt('animal_names.csv', delimiter=',', dtype=str)
```

```
# Test your functions
>>> gini(animals)
0.4758
# split animals into two sets with fifty animals in each
>>> info_gain(animals[:50], animals[50:], gini(animals))
0.14579999999999999
```

Optimal Split

The optimal split of a data set can be chosen by maximizes either the Gini impurity or the information gain. We will optimize the information gain, so the optimal split is

$$s_D^* = s_D(p^*, x^*),$$

where

$$p^*, x^* = \text{argmax}_{p,x} I(s_D(p, x)).$$

Sometimes the partition to split on may separate the data into very small subsets with only a few samples each. This can make the classification tree vulnerable to overfitting and noisy data. For this reason, classification trees include an argument to specify the smallest allowable leaf size, or the minimum number of samples at the node. This number depends on the size of the whole data set; for example, data with 10,000 samples would have a larger minimum leaf than our first example using data with only 10 samples.

Problem 3. Write a function `find_best_split()` that computes the optimal split of a data set by checking through all possible `Questions` associated with the data (all values present for each feature (column)). Recall that the final column has the class label and will have no possible questions associated with it. Include a minimum leaf argument defaulting to 5. Do not allow the best split to include a leaf smaller than this size. Return the information gain and question associated with the best split. If two splits have the same information gain, choose the first split.

The output for the animals data set should be `(0.12259833679833688, Is # legs/tentacles >= 2.0?)`.

Building the Tree

Once the optimal split is determined, the node is defined to be a Leaf node or a Decision node. As described earlier, leaf nodes have no children nodes and is where the classification for a sample is made. If the optimal split returns a left and right tree, then the node is a decision node and has a question associated with it to determine which path a sample should follow. The next two problems will walk through building a classification tree using the functions and classes from the previous problems.

Problem 4. Write the class `Leaf`. It should have an attribute `prediction` that is the dictionary of how many samples at the leaf belong to each label, as shown in the leaves of Figure

9.1.

Next, write the class Decision_Node. This should have three attributes: an associated Question, a left branch, and a right branch. The branches will be Leaf or Decision_Node objects. Name these three attributes question, left, and right.

In addition to having a minimum leaf size, it's also important to have a maximum depth for trees. Without restricting the depth, the tree can become very large; if there is no minimum leaf size, it can be one less than the number of training samples. Limiting the depth can stop the tree from having too many splits, preventing it from becoming too complex and overfitting the training data. It's also important to not have too shallow of a tree because then the tree will underfit the data.

Problem 5. Write a function build_trees() that uses your previous functions to build a classification tree. Include a minimum leaf argument defaulting to 5 and a maximum depth argument defaulting to 4. Start counting depth at 0. For comparison, the tree in Figure 9.1 has depth 3.

You will probably want to build this tree recursively. If the remaining data has too few samples, if the depth is too much, or if the information gain is 0, make a Leaf. Otherwise, make a partition and build a new tree for each branch, returning those as Decision_Nodes.

The last column in the animals.csv file indicates whether or not the animal lives in the ocean; this is the class label for this data set. Test your classifier with this file and the function draw_tree. This will display and save a pdf of the graph. Examine the figure and test various parameters to check if your functions are working properly.

```
# How to draw a tree
>>> my_tree = build_tree(animals, features)
>>> draw_tree(my_tree)
```

Note

The function draw_tree relies on the Graphviz package, which you can download by typing conda install -c conda-forge python-graphviz if you have the Anaconda distribution. If draw_tree returns an error about pdf being an unrecognized file type, try typing dot -c in your terminal.

Predicting

It's important to test your tree to ensure that it predicts class labels fairly accurately and so that you can adjust the minimum leaf and maximum depth parameters as needed. It is customary to randomly assign some of your labeled data to a training set that you use to fit your tree and then use the rest of your data as a testing set to check accuracy.

Problem 6. Write a function `predict_tree` that returns the predicted class label for a new sample given a trained tree. You will probably have to make this recursive in order to traverse the branches and reach a `Leaf` node with prediction information.

Next, write a function `analyze_tree` that accepts a labeled data set (with the labels in the last column, as in `animals.csv`) and a trained classification tree and returns the proportion of samples that the tree labels correctly.

Test your function with the `animals.csv` file. Shuffle the data set with `np.random.shuffle()` and use 80 samples to train your classification tree. Use the other 20 samples as the test set to see how accurately your tree classifies them. Your tree should be able to classify this set with roughly 80% accuracy on average, given the default parameters.

Random Forest

As noted, one of the main issues with Decision Trees is their tendency to overfit Random forests are a way of mitigating overfitting that cannot be fixed by restricting the depth and leaf size. A *random forest* is just what it sounds like–a collection of trees. Each tree is trained randomly, meaning that at each node, only a small, random subset of the features is available by which to determine the next split. The size of this subset should be small relative to the total number of features present. Let n be the total number of features in the data set. One common method, and the one we will use here, is to split on $\sqrt{n}$ features, rounding down where applicable.

When predicting the label of a new sample, each trained tree in the forest casts a vote, determined as above, and the sample is labeled according to the majority vote of the trees.

Problem 7. Add an argument `random_subset` to `build_tree()` and `find_best_split()`, defaulting to `False`, that indicates whether or not the tree should be trained randomly. When `True`, each node should be restricted to a random combination of $\sqrt{n}$ features to use in its split, where n is the total number of features (note that class labels are not features).

Next, write a function `predict_forest()` that accepts a new sample and a trained forest (as a list of trees). It should return the assigned label, found by majority vote of the trees.

Finally, write a function `analyze_forest()` that accepts a labeled data set and a trained forest and analyzes the accuracy of the forest's predictions.

Test your functions out on the `animals.csv` file. Examine the graphs of the individual trees to see how they compare to the non-randomized versions.

Scikit-Learn

Next, we'll compare our implementation to scikit-learn's `RandomForestClassifier`. Rather than accepting all the data as a single array, as in our implementation, this package accepts the feature data as the first argument and all of the labels as the second argument.

```
>>> from sklearn.ensemble import RandomForestClassifier

# Create the forest with the appropriate arguments and 200 trees
>>> forest = RandomForestClassifier(n_estimators=200, max_depth=4,
...                                 min_samples_leaf=5)
```

```
# Shuffle the data
>>> shuffled = np.random.permutation(animals)
>>> train = shuffled[:80]
>>> test = shuffled[80:]

# Fit the model to your data, passing the labels in as the second argument
>>> forest.fit(train[:,:-1], train[:,-1])

# Test the accuracy with the testing set
>>> forest.score(test[:,:-1], test[:,-1])
0.85
```

Problem 8. The file `parkinsons.csv` contains annotated speech data from people with and without Parkinson's Disease. The first column is the subject ID, columns 2-27 are various features, and the last column is the label indicating whether or not the subject has Parkinson's. You will need to remove the first column so your forest doesn't use participant ID to predict class labels. Feature names are contained in the file `parkinsons_features.csv`.

Write a function to compare your forest implementation to the package from scikit-learn. Because of the size of this data set, we will only use a small portion of the samples and build a very simple forest. Randomly select 130 samples. Use 100 in training your forest and 30 more in testing it. Include 5 trees in the forest and use `min_samples_leaf=15`. Time how long it takes to train and analyze your forest.

Repeat this with scikit-learn's package, using the same 100 training samples and 30 test samples. Set `n_estimators=5` and `min_samples_leaf=15`.

Next, using scikit-learn's package, run the whole data set, using the default parameters. Use 80% of the data to train the forest and the other 20% to test it.

Return three tuples, where each tuple contains the accuracy and time for each variation.