Write-Up

Ibraheem Khan and Matthew Alighchi

September 26, 2019

1 DT_train_binary Implementation Details

DT_train_binary takes in our globally available X and Y along with some max-depth and returns a binary tree of said depth based on the information gain algorithm. It does so by first creating a root node which is a class object as defined in the BinaryNode class and feeds the root node all of the data. It then cycles through all the possible features in feature_list which is a list of all features for which the algorithm has not yet split on, pretends to split on it and the calculates the information gain. It then sets the best_IG to this information gain. As it loops through the best_IG is updated to the actual maximum information gain along with the associated best_index. The function then reorganizes this data and places the left and right label sets and features corresponding to the maximal information gain split into the left and right nodes. DT_train_aux then iterates this function for a specified max_depth until the tree is fully complete up to max_depth without recreating a root node.

The information gain function works by relying on the entropy function which is a standard implementation of the mathematical formula for entropy. The information gain implementation, too, is standard conversion from math to code, however we test edge cases for when our code may not work. For example, if our label sets are empty then our information gain must be 0.

2 DT_test_binary Implemenation Details

DT_test_binary mainly relies on the output of DT_test_binary_aux which iterates through our decision tree's label prediction parameter (which is defined in the class and computed in DT_train_binary and DT_train_binary_aux) and increases the accuracy output should the label set correspond to our prediction. It initially checks to see if all we have is a single node with no children and computes its accuracy. Then it goes through its children and assesses them recursively. The output of DT_test_binary_aux then has to be correctly resized (to take into account the number of total correct predictions) in DT_test_binary by a factor of 1 over the number of samples.

3 DT_train_binary_best Implementation Details

DT_train_binary_best works by first finding the maxDepth possible for our given DT. This is computed by maxDepth, which runs recursively until the tree is no more. DT_train_binary_best. The function then iterates through all possible depths and creates decision trees of those depths for the given data and assesses their accuracy via DT_test_binary against the validation data. It then returns the decision tree with the best accuracy.

4 Testing DT_train_binary, DT_test_binary, and DT_train_binary_best

This is how we are implementing this section:

```
import numpy as np
import decision_trees as dt

#Training Set 1:
X = np.array([[0, 1], [0, 0], [1, 0]])
Y = np.array([[1], [0], [0]])
#Validation Set 1
X_val = np.array([[0, 0], [0, 1], [1, 0]])
Y_val = np.array([[0], [1], [0]])
#Testing Set 1
X_tst = np.array([[0, 0], [0, 1], [1, 0]])
Y_tst = np.array([[1], [1], [0]])
print("The accuracy is: ", dt.DT_test_binary(X_tst,Y_tst,dt.
    DT_train_binary(X,Y, 2)))
print("The best accuracy is: ", dt.DT_test_binary(X_tst, Y_tst, dt.
    DT_train_binary_best(X, Y, X_val, Y_val)))

```

Listing 1: Testing Implementation

There is an unsolved issue with our code where our data entries have to only contain a certain number of samples. 3 samples, for example, works so I have fed the data given after lopping off the last sample. The results we get are:

```

The accuracy is:  0.3333333333333333
The best accuracy is:  0.6666666666666666


```

Listing 2: Testing Results

5 A Forest of Decision Trees

I am feeding the same type of lopped off data in this section as well. I am just choosing the first 3 and last 3 here. Here is the implementation of our test:

```
import numpy as np
import decision_trees as dt

#Training Set 1:
X = np.array([[0, 1], [0, 0], [1, 0]])
Y = np.array([[1], [0], [0]])
#Validation Set 1
X_val = np.array([[0, 0], [0, 1], [1, 0]])
Y_val = np.array([[0], [1], [0]])
#Testing Set 1
X_tst = np.array([[0, 0], [0, 1], [1, 0]])
Y_tst = np.array([[1], [1], [0]])
print("The accuracy is: ", dt.DT_test_binary(X_tst,Y_tst,dt.
    DT_train_binary(X,Y, 2)))
print("The best accuracy is: ", dt.DT_test_binary(X_tst, Y_tst, dt.
    DT_train_binary_best(X, Y, X_val, Y_val)))

```

Listing 3: Testing Implementation

6 DT_train_real, DT_test_real, DT_train_real_best Implementations

DT_train_real works much in the same way as the binary version does with the exception that it should be splitting on a much larger feature_list. Unfortunately, we were not able to fully implement this part of the function however we made sure that it would compile and output results. Hypothetically, we would have implemented it in such a way that the only major difference between the binary version and the real is that the feature list would be much larger- it is now a 2D array of triples as we no longer split on just some feature index but now on some relation defined on the feature- that is for example asking the question is the feature given by feature index 1 less than (or less than or equal to) some real value. In this way, we could preserve most of our code and just add a few more cases and make sure to traverse our arrays correctly. The remaining functions work in practically the exact same way as their binary counterparts.

7 Testing DT_train_real

Since our implementation of DT_train_real does not work as intended we were not able to complete this section.