CS 334: Homework #2 (David Lee)

1. Decision Tree Implementation

(a) Train function of decision tree

```
class Node():
    def __init__(self):
        #splitting feature
        self.feature = None
        #splitting value
        self.value = None
        #left node
        self.left = None
        #right node
        self.right = None
        #depth of current node
        self.depth = 0
        #prediction
        self.predict = None
```

Node class for the decision tree. Each Node will contain information including the splitting feature, splitting value, left node that has the dataset with less than the splitting value at the splitting feature, right node that has the dataset with greater than the splitting value at the splitting feature, depth of the current depth, and if it is at the stopping condition, it will have the prediction value.

There are two helper functions for the decision tree implementation.

```
#Helper function for Attribute selection measure
def attribute_selection_measure(y, criterion):

    #The total number of entries in the given case
    total = len(y)

    #Get the unique values of the labels
    labels = np.unique(y)

    #List of probably of each unique values
    prob_labels = []

    #Calculate the probability of each label ocurring in the case
    for i in range(len(labels)):
        prob_labels.append(float(np.sum(y == labels[i]))/total)

    #if using gini as the measure
    if criterion == "gini":
        #Initialize gini index as 1
        gini = 1
        #Subtract the square of the probability of each labels to get gini index
        for i in range(len(prob_labels)):
            gini -= prob_labels[i]**2

        return gini

    #if using entropy as the measure
    elif criterion == "entropy":
        #Initialize entropy as 0
        entropy = 0
        #Subtract the p log p of each labels to get the entropy
        for i in range(len(prob_labels)):
            entropy -= prob_labels[i] * np.log2(prob_labels[i])

        return entropy
```

The first function is the attribute_selection_measure() function. This function measures the gini or entropy score.

```python
#Helper function for finding the best feature and the best value to split in the decision tree
def find_best_split(xFeat, y, criterion, minLeafSample):

    #initialize current best feature to split as the first feature
    best_feature = xFeat.columns[0]

    #initialize current best value to split as the first value
    best_val = xFeat.iloc[0, 0]

    #initialize best gain as 0
    best_gain = 0

    #iterate all the features
    for curr_feature in xFeat.columns:
        #sort the xFeat in order of the value in the current feature
        sorted_index = np.argsort(xFeat[curr_feature])
        sorted_xFeat = xFeat.iloc[sorted_index]
        #sort the y correspondingly
        sorted_y = y[sorted_index]

        #get the original gini/entropy
        measure_df = attribute_selection_measure(y, criterion)


        for i in range(minLeafSample, len(xFeat)-minLeafSample):
            curr_val = sorted_xFeat.iloc[i][curr_feature]
            less_than = sorted_xFeat[curr_feature] <= curr_val
            greater_than = sorted_xFeat[curr_feature] > curr_val
            y_less = sorted_y[less_than]
            y_greater = sorted_y[greater_than]
            #if gini, gini impurity of y_less
            #if entropy, entropy of y_less
            measure_less = attribute_selection_measure(y_less, criterion)
            #if gini, gini impurity of y_greater
            #if entropy, entropy of y_greater
            measure_greater = attribute_selection_measure(y_greater, criterion)
            #if gini, impurity of total
            #if entropy, entropy of total
            measure_total = float(len(y_less)/len(y)) * measure_less + float(len(y_greater)/len(y)) * measure_greater
            #if gini, gini gain
            #if entropy, information gain
            information_gain = measure_df - measure_total
            if information_gain > best_gain:
                best_feature = curr_feature
                best_val = curr_val
                best_gain = information_gain
    return best_feature, best_val
```

The second function is the find_best_split() function. This function loops through the sorted values of each feature to find the best splitting feature and best splitting value at the current node.

```python
#stopping criteria
def is_stopping_criteria(self, left_y, right_y, depth):
    #if the left or right node of the current node has less than the minimum samples in a leaf
    if len(left_y) < self.minLeafSample or len(right_y) < self.minLeafSample:
        return True
    #if the current node has depth that is greater than the maxDepth
    elif depth >= self.maxDepth:
        return True
    #not a stopping criteria
    else:
        return False
```

Function is_stopping_criteria checks if the current node is a stopping criteria.

```python
def decision_tree(self, xFeat, y, curr_node):


    #find the best feature and splitting value
    best_feature, best_val = find_best_split(xFeat, y, self.criterion, self.minLeafSample)

    #save the splitted datas of x and y
    left_index = xFeat[best_feature] <= best_val
    right_index = xFeat[best_feature] > best_val
    left_x = xFeat.loc[left_index]
    right_x = xFeat.loc[right_index]
    left_y = y[left_index]
    right_y = y[right_index]

    #check for stopping criteria first
    if self.is_stopping_criteria(left_y, right_y, curr_node.depth):
        #if at stopping criteria, add the mode of y in the node's value
        #prediction
        curr_node.predict = stats.mode(y, keepdims = True)[0]

    else:
        #Save feature, value to the current node
        curr_node.feature = best_feature
        curr_node.value = best_val
        #Create the left and right node with depth increased by 1
        curr_node.left = Node()
        curr_node.right = Node()
        curr_node.left.depth = curr_node.depth + 1
        curr_node.right.depth = curr_node.depth + 1
        #Recursion to create whole decision tree
        self.decision_tree(left_x, left_y, curr_node.left)
        self.decision_tree(right_x, right_y, curr_node.right)
```

Function decision_tree creates the decision tree using the functions above. By finding the best splitting feature and value for each node, each time the current node is checked if it is a stopping criteria. If it is a stopping criteria, the current node with the mode of the y value of the dataset becomes the prediction. If not, there is a recursion to the function to create the left and right node of the current node.

```python
def train(self, xFeat, y):
    """
    Train the decision tree model.

    Parameters
    ----------
    xFeat : nd-array with shape n x d
        Training data
    y : 1d array with shape n
        Array of labels associated with training data.

    Returns
    -------
    self : object
    """
    # TODO do whatever you need
    self.decision_tree(xFeat, y.to_numpy(), self.head)
    return self
```

The train function creates the decision tree using the training data.

<u>(b) Predict function of decision tree</u>

```python
def predict_sample(self, node, x):
    # If there is a child node
    if node.left != None:
        # If the sample's value is greater than the split value
        if x[node.feature] > node.value:
            # Move to the right node and recursively call the function
            newNode = node.right
            return self.predict_sample(newNode, x)
        # If the sample's value is less than or equal to the split value
        else:
            # Move to the left node and recursively call the function
            newNode = node.left
            return self.predict_sample(newNode, x)
    # If the current node is the prediction node, return the predicted value
    else:
        return node.predict
```

The predict_sample function is used to predict a single row.

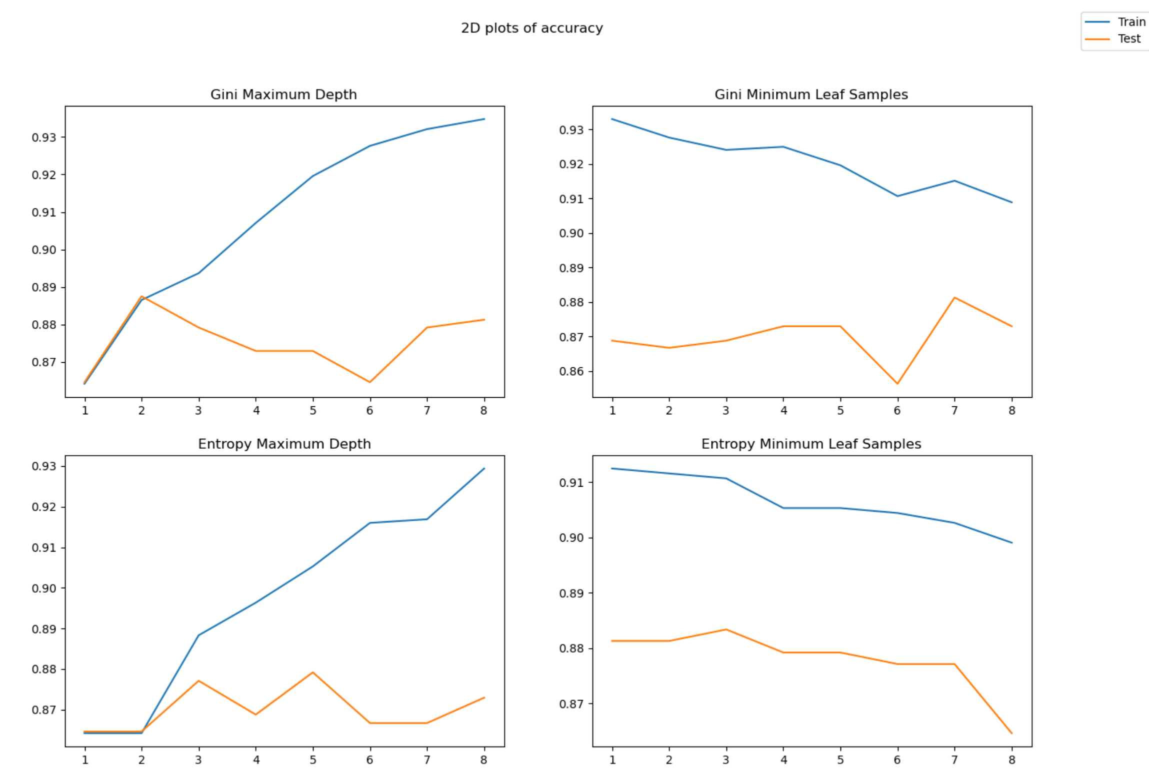
```python
def predict(self, xFeat):
    """
    Given the feature set xFeat, predict
    what class the values will have.

    Parameters
    ----------
    xFeat : nd-array with shape m x d
        The data to predict.

    Returns
    -------
    yHat : 1d array or list with shape m
        Predicted class label per sample
    """
    yHat = [] # feature to store the estimated class label
    # Iterate over all the rows of the provided xFeat
    for i in range(xFeat.shape[0]):
        # Append the predicted label to yHat
        yHat.append(self.predict_sample(self.head, xFeat.iloc[i, :]))
    return yHat
```

The predict function uses the predict_smaple function to create yHat, which is a list that stores the estimated class labels for each of the rows.

<u>(c) Training accuracy and test accuracy of the data for different values of maxdepth and minimum number of samples in a leaf</u>



The training accuracy increased as the maximum depth increased for both gini and entropy. The training accuracy decreased as the minimum leaf samples increased for both gini and entropy. However, the test accuracy for all four cases seemed to fluctuate.
Png file saved as 1(c).png

<u>(d) Computational complexity with training size (n), the number of features (d), and the maximum depth (p).</u>

The computational complexity of the train function is O(pdnlogn). For each feature, all the possible splits are calculated -> dn and for each of this process, the rows are sorted. -> dnlogn. If you add this this becomes O(dnlogn). And this process is done for each node in the tree which makes O(p(dnlogn)) which is just O(pdnlogn). The computational complexity of the predict function is O(p). THe predict function is only traversing the tree which is just the maximum depth.

2. Exploring Model Assessment Strategies

(a) Implement the holdout technique

```python
def holdout(model, xFeat, y, testSize):
    """
    Split xFeat into random train and test based on the testSize and
    return the model performance on the training and test set.

    Parameters
    ----------
    model : sktree.DecisionTreeClassifier
        Decision tree model
    xFeat : nd-array with shape n x d
        Features of the dataset
    y : 1-array with shape n x 1
        Labels of the dataset
    testSize : float
        Portion of the dataset to serve as a holdout.

    Returns
    -------
    trainAuc : float
        Average AUC of the model on the training dataset
    testAuc : float
        Average AUC of the model on the validation dataset
    timeElapsed: float
        Time it took to run this function
    """
    trainAuc = 0
    testAuc = 0
    timeElapsed = 0
    # TODO fill int

    start = time.time()

    xTrain, xTest, yTrain, yTest = train_test_split(xFeat, y, test_size = testSize)

    trainAuc, testAuc = sktree_train_test(model, xTrain, yTrain, xTest, yTest)
    end = time.time()
    timeElapsed = end - start
    return trainAuc, testAuc, timeElapsed
```

Used the train_test_split from sklearn.model_selection.
Measured time by using time module.

(b) Implement k-fold cross-validation approach for a model.

```
def kfold_cv(model, xFeat, y, k):

    trainAuc = 0
    testAuc = 0
    timeElapsed = 0
    # TODO FILL IN
    start = time.time()

    train_acc = []
    test_acc = []

    kFold = KFold(n_splits = k)
    for trainIdx, testIdx in kFold.split(xFeat):
        xTrain = xFeat.iloc[trainIdx]
        xTest = xFeat.iloc[testIdx]
        yTrain = y.iloc[trainIdx]
        yTest = y.iloc[testIdx]
        curr_trainAuc, curr_testAuc = sktree_train_test(model, xTrain, yTrain, xTest, yTest)

        train_acc.append(curr_trainAuc)
        test_acc.append(curr_testAuc)

    trainAuc = np.sum(train_acc)/k
    testAuc = np.sum(test_acc)/k
    end = time.time()
    timeElapsed = end - start

    return trainAuc, testAuc, timeElapsed
```

Used the KFold from sklearn.model_selection to get the splitted train and test dataset index
Measured time by using time module.

(c) Implement Monte Carlo Cross-validation approach

```
def mc_cv(model, xFeat, y, testSize, s):

    trainAuc = 0
    testAuc = 0
    timeElapsed = 0
    # TODO FILL IN
    start = time.time()

    train_acc = []
    test_acc = []

    for i in range(s):
        curr_trainAuc, curr_testAuc, curr_timeElapsed = holdout(model, xFeat, y, testSize)
        train_acc.append(curr_trainAuc)
        test_acc.append(curr_testAuc)
        timeElapsed += curr_timeElapsed

    trainAuc = np.sum(train_acc)/s
    testAuc = np.sum(test_acc)/s

    end = time.time()
    timeElapsed = end - start

    return trainAuc, testAuc, timeElapsed
```

Used the holdout function to get the trainAuc, testAuc, and timeElapsed each time.
Measured time by using time module.

(d) Table

	Strategy	TrainAUC	ValAUC	Time
0	Holdout	0.954378	0.790658	0.006006
1	2-fold	0.955713	0.769119	0.011010
2	5-fold	0.952979	0.797798	0.032068
3	10-fold	0.954251	0.794634	0.067062
4	MCCV w/ 5	0.938867	0.752263	0.024022
5	MCCV w/ 10	0.945142	0.774269	0.044084
6	True Test	0.952502	0.803077	0.000000

For MCCV, as the portion of the dataset to serve as a holdout increased, both the validation and the time elapsed increased. For k-fold CV, as the number of folds or groups increased, both the validation and the time elapsed increased.

3. Robustness of Decision Trees and K-NN

(a) Find optimal hyperparameters for k-nn and decision tree

```
# 3(a) / justification of k = 5 at the bottom
def gs_model(classifier, parameters, xTrain, yTrain):
    gs = GridSearchCV(estimator = classifier,
                      param_grid =  parameters,
                      cv = 5,
                      scoring = 'roc_auc')
    gs.fit(xTrain, yTrain)
    return gs
```

```
---------------------    KNN    ------------------------
{'n_neighbors': 14}
```

```
---------------------    DT    ------------------------
{'criterion': 'gini', 'max_depth': 4, 'min_samples_leaf': 7}
```

For k-nn, the optimal hyperparameter was {'n_neighbors': 14}. For decision tree, the optimal hyperparameters were {'criterion': 'gini', 'max_depth': 4, 'min_samples_leaf': 7} I chose k = 5 as the results from question 2 showed that k = 5 and 10 resulted in a relatively high score for test AUC and I wanted to minimize the computational time so I chose 5 over 10.

(b,c) train k-nn and decision tree with random removal of the original training data

```
def train_remove(model, xTrain, yTrain, xTest, yTest):
    train_size = len(xTrain)
    result = []
    for size_percentage in [1.00, 0.95, 0.90, 0.80]:
        removed_size = int(train_size*size_percentage)
        removed_index = np.random.choice(train_size, removed_size, replace=False)
        removed_xTrain = xTrain.iloc[removed_index, :]
        removed_yTrain = yTrain.loc[removed_index, 'label']
        removed_yTrain = removed_yTrain.to_numpy()
        subset = train_test(model.best_estimator_,
                            removed_xTrain,
                            removed_yTrain,
                            xTest,
                            yTest)
        result.append(subset)
    return result
```

```
---------------------   KNN   ------------------------
{'n_neighbors': 14}
               0%        5%       10%       20%
trainAuc  0.853936  0.790176  0.865952  0.862500
testAuc   0.856909  0.787006  0.870179  0.864583
trainAcc  0.857730  0.798647  0.868918  0.864583
testAcc   0.859569  0.790065  0.865922  0.864583
---------------------   DT    ------------------------
{'criterion': 'gini', 'max_depth': 4, 'min_samples_leaf': 7}
               0%        5%       10%       20%
trainAuc  0.885926  0.854866  0.905273  0.904167
testAuc   0.892731  0.862354  0.901223  0.887500
trainAcc  0.901643  0.857331  0.900695  0.881250
testAcc   0.878468  0.841372  0.901676  0.883333
```

(d) AUC and accuracy of the 8 different models

	Removed	trainAuc	testAuc	trainAcc	testAcc
k-nn	0%	0.8539364829	0.8570814915	0.8638308583	0.8494402067
k-nn	5%	0.790176089	0.7896014829	0.7903429101	0.7763855422
k-nn	10%	0.8659517426	0.8645343368	0.8679245283	0.8670391061
k-nn	20%	0.8625	0.8625	0.86875	0.8645833333
Decision tree	0%	0.8859263593	0.9082487003	0.8957546774	0.8943786277
Decision tree	5%	0.8403336423	0.8777757183	0.827673772	0.8481742354
Decision tree	10%	0.9052725648	0.9078080903	0.9026812314	0.9039106145
Decision tree	20%	0.875	0.9104166667	0.8729166667	0.9020833333

Table is saved as 3(d).csv. I expected that the testAuc and testAcc to decrease while the removed percentage increased. However, trainAuc, testAuc, trainAcc, testAcc were varied in the differences of removed percentage.