Assignment 1: K-Nearest Neighbors and Decision Trees

Part 1: Implement KNN from scratch. For the Pima Indian Diabetes Dataset, train 5 different nearest neighbors regressors using the following number of neighbors: 3, 5, 10, 20, and 25 on the training set; report the F1 score on the test set. F1 score is defined as the harmonic mean of precision and recall.

Python filename on github: knn.py

```
import time
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.neighbors import BallTree
from collections import Counter

# define F Score function
def f1_score(y_true, y_pred, epsilon=1e-6):
    """
    Function for calculating the F1 score

    Params
    ------
    y_true  : the true labels shaped (N, C),
              N is the number of datapoints
              C is the number of classes
    y_pred  : the predicted labels, same shape
              as y_true

    Return
    ------
    score   : the F1 score, shaped (N,)

    """
    #calculate TP, TN, FP, FN
    true_positives = sum((y_true == 1) & (y_pred ==1))
    true_negatives = sum((y_true == 0) & (y_pred ==0))
    false_positives = sum((y_true == 0) & (y_pred ==1))
    false_negatives = sum((y_true == 1) & (y_pred ==0))

    #calcualte accuracy
    accuracy = ((true_positives+true_negatives)/(true_positives+true_negatives+false_positives+false_negatives))

    #calculate precision, recall, f1_score
    precision = true_positives/(true_positives+false_positives)
    recall = true_positives/(true_positives+false_negatives)
    f1_score = 2*((precision*recall)/(precision+recall))

    return f1_score, accuracy

class KNN(object):
    """
    The KNN classifier
    """
    #init initializes the class for a specific object
    def __init__(self, n_neighbors):
        #Initialize the KNN with n_neighbors
        self.n_neighbors = n_neighbors
        return

    def fit(self, x_train, y_train):
        #print("start fitting")
        """
        Fitting the KNN classifier

        Hint:   Build a tree to get neighbors
                faster at test time
        """
        #set the leaf size to 26 because at most we search for 25 neighbors
        self.y_train = y_train
        self.tree = BallTree(x_train, leaf_size = 100)
        #print("stop fitting")
```

```
        return

    def getKNeighbors(self, x_instance):
        #print('getting kNeighbors')
        """
        Locating the K nearest neighbors of
        the instance and return
        """
        neighbors = self.tree.query(x_instance, k=self.n_neighbors, return_distance=False, sort_results=True)
        return neighbors

    def getResponse(self, neighbors):
        #print("getting Response")
        """
        Helper function to count/vote neighbors' labels
        """
        return Counter(neighbors).most_common(1)[0][0]

    def predict(self, x_test):
        """
        Predicting the test data
        Hint:   Get the K-Neighbors, then generate
                predictions using the labels of the
                neighbors
        """
        #rows,__ = x_test.shape
        y_pred = []
        #print('begin predicting')
        #for i in range(rows):
            #pull first row of test data set to use as query in tree class
            #test_point = x_test[i-1,:]
            #nearest_neighbors = self.getKNeighbors(test_point)
        nearest_neighbors = self.getKNeighbors(x_test)
        length = len(nearest_neighbors)

        for rows in range(length):
            X_indices = nearest_neighbors[rows]
            instance_labels = []

            for i in X_indices:
                current_label = int(self.y_train[i])
                instance_labels.append(current_label)

            pred_value = self.getResponse(instance_labels)
            #store the predicted value in the y_pred vector
            y_pred.append(pred_value)

        #print('done predicting')
        return np.asarray(y_pred)


def load_data(data_dir='Data'):

    """
    Function for loading the dataset from data_dir
        default: data_dir='Data'
    """
    # Load the data
    X_train = pd.read_csv(f'{data_dir}/X_Train.csv', header=None).values
    y_train = pd.read_csv(f'{data_dir}/Y_Train.csv', header=None).values[:,0]

    X_test = pd.read_csv(f'{data_dir}/X_Test.csv', header=None).values
    y_test = pd.read_csv(f'{data_dir}/Y_Test.csv', header=None).values[:,0]


    return {"train": dict(X=X_train, y=y_train),
            "test": dict(X=X_test, y=y_test)}


def main():
    # Set up the neighbors list
    n_neighbors_list = [3,5,10,20,25]
    time_per_neighbor = []
    metrics = ['accuracy', 'f1_score', 'time']
    results = dict([(key, []) for key in metrics])

    print("\n\n")
    print('Running knn.py')

    # Load in data
    data = load_data('/Users/rossbrancati/Documents/Fall_2021_Classes/cs_589/assignments/hw1/Data')
```

```
    X_train = data['train']['X']
    y_train = data['train']['y']
    print(f"Training data loaded: X shape: {X_train.shape}, y shape: {y_train.shape}")

    X_test = data['test']['X']
    y_test = data['test']['y']
    print(f"Test data loaded: X shape: {X_train.shape}, y shape: {y_train.shape}")

    print(f"\t Class [0/1] ratio: Train [{(y_train==0).sum()}/{(y_train==1).sum()}]")
    print(f"\t Class [0/1] ratio: Test [{(y_test==0).sum()}/{(y_test==1).sum()}]")

    # loop over neighbors_list
    for n_neighbors in n_neighbors_list:
        print("n_neighbors: ", n_neighbors)

        # start timer
        t0 = time.time()

        # instantiate KNN instance
        knn = KNN(n_neighbors=n_neighbors)

        # fit the KNN model with X_train, y_train
        knn.fit(X_train, y_train)

        # generate predictions on the test set X_test
        y_pred = knn.predict(X_test)

        # compute metrics F1_score on the test set with y_test
        F1_score, Accuracy = f1_score(y_test, y_pred, epsilon=1e-6)

        # stop timer
        t1 = time.time()

        # record time and report metric
        total_time = (t1-t0)*1000

        time_per_neighbor.append(total_time)
        results['accuracy'].append(Accuracy)
        results['f1_score'].append(F1_score)
        results['time'].append(total_time)

    print(results)
    print("\n Completed!")

    #Plot the time used
    plt.plot(n_neighbors_list, time_per_neighbor)
    plt.scatter(n_neighbors_list, time_per_neighbor)
    plt.xlabel("K: Number of Neighbors (count)")
    plt.ylabel("Time (milliseconds)")
    plt.title("Time to Complete KNN Algorithm at\nVarious Neighbor Values")
    plt.show()
    #or plt.savefig()


if __name__ == '__main__':
    main()
```

Part 2: Decision Trees: For the Pima Indians Diabetes Dataset, train 5 different decision trees with the following maximum depths: 2, 3, 5, 7, and 9 on the training set. Report each model's F1 score on the test set. In addition, measure the time (in milliseconds) that it takes to complete the classification task with each model and report the results using a line graph.

Python file on github: decision_tree.py

```
import time
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

"""
Basic idea:
1. We have a dataset, we want to create a tree-like class to classify an input based on the given dataset
2. What we do is to split the dataset
```

```
3. Each time we split the dataset, we choose a column index, find a threshold value, move all rows with
   the target column value smaller than the threshold to the left child, move all the remaining rows to
   the right child
4. We do it recursively until we reach max depth
5. We then label the terminal node
"""

def f1_score(y_true, y_pred):
    """
    Function for calculating the F1 score

    Params
    ------
    y_true  : the true labels shaped (N, C),
              N is the number of datapoints
              C is the number of classes
    y_pred  : the predicted labels, same shape
              as y_true

    Return
    ------
    score   : the F1 score, shaped (N,)

    """
    #calculate TP, TN, FP, FN
    true_positives = sum((y_true == 1) & (y_pred ==1))
    true_negatives = sum((y_true == 0) & (y_pred ==0))
    false_positives = sum((y_true == 0) & (y_pred ==1))
    false_negatives = sum((y_true == 1) & (y_pred ==0))

    #calcualte accuracy
    accuracy = ((true_positives+true_negatives)/(true_positives+true_negatives+false_positives+false_negatives))

    #calculate precision, recall, f1_score
    precision = true_positives/(true_positives+false_positives)
    recall = true_positives/(true_positives+false_negatives)
    f1_score = 2*((precision*recall)/(precision+recall))

    return f1_score, accuracy


def gini_index(groups, classes):
    """
    Function for calculating the gini index
        -- The goodness of the split

    Params
    ------
    groups  : A list containing two groups of samples
                resulted by the split
    classes : The classes in the classification problem
                e.g. [0,1]

    Return
    ------
    gini    : the gini index of the split
    """
    #reset the gini index every time you loop over the groups
    gini = float(0)

    #loop over each group in the groups list
    for group in groups:

        length = float(len(group))
        total_length = len(groups[0])+len(groups[1])

        if length == 0:
            gini = 1
        else:
            #calcualte the probability that one group belongs to a particular class
            positives = (group.count(classes[1]))/length
            negatives = (group.count(classes[0]))/length
            #calculate the gini index, weighted by the number of observations in this group
            #gini index is a weighted sum of classes of child nodes
            gini = gini + ((1 - ((positives**2) + (negatives**2))) * (length/total_length))
    return gini


def get_split(x_train, y_train):
    """
    Function to generate the split which results in
```

```
    the best gini_index

Params
------
x_train : the input data (n_samples, n_features)
y_train : the input label (n_samples,)

Return
------
{gini_index, split_index, split_value}
"""
#create an empty dictionary to store gini index, split index, and split values
gini_dict = {"Gini Index":[], "Split Index":[], "Split Value":[]}

#calculate the number of rows and columns in x_train array
rows, cols = x_train.shape

for i in range(cols):
    #assign the current column of training data
    current_col = x_train[:,i]
    #assign the split index to the current column
    split_index = i
    #initialize some arbitratily large values of gini and value
    gini = 1000
    value = 1000
    for r in range(rows):
        test_value = current_col[r]
        #reset left and right decision lists on each iteration
        left_decision = list()
        right_decision = list()

        for n in range(rows):
            #test_value = current_col[n]
            current_value = current_col[n]
            if current_value < test_value:
                left_decision.append(y_train[n])
            else:
                right_decision.append(y_train[n])
        #create a global list of splits for gini_index function
        groups = [left_decision, right_decision]

        #assign classes to [0,1] list
        classes = [0,1]

        #calcualte the gini_index
        gini_test = gini_index(groups, classes)

        #test if new gini value is less than gini
        if gini_test < gini:
            gini = gini_test
            #and then assign the new value to value
            value = test_value

    #append the dictionary of values
    gini_dict['Gini Index'].append(gini)
    gini_dict['Split Index'].append(split_index)
    gini_dict['Split Value'].append(value)


#Now we need to calculate the index and threshold to split the data on

#order the gini_indices in ascending order
sorted_gini_indices = np.argsort(np.array(gini_dict['Gini Index']))

#reset the left and right node arrays after iterating over every column
left_node_data = np.empty([0, cols])
right_node_data = np.empty([0, cols])
left_node_labels = np.array([])
right_node_labels = np.array([])


#get the feature with the smallest gini index
feature = sorted_gini_indices[0]

#get the value of the split index for the feature of interest
split_value = gini_dict['Split Value'][feature]

for row in range(rows):
    if x_train[row,feature] < split_value:
        left_node_data=np.append(left_node_data,[x_train[row]], axis=0)
        left_node_labels=np.append(left_node_labels,y_train[row])
```

```
        else:
            right_node_data=np.append(right_node_data,[x_train[row]], axis=0)
            right_node_labels=np.append(right_node_labels,y_train[row])

    data = {'left node data':left_node_data, 'right node data':right_node_data, 'left node
labels':left_node_labels, 'right node labels':right_node_labels, 'split index':feature, 'split
value':split_value}


    return data


class DecisionTree(object):
    """
    The Decision Tree classifier
    """
    def __init__(self, max_depth, min_size):
        """
        Params
        ------
        max_depth   : the maximum depth of the decision tree
        min_size    : the minimum observation points on a
                        leaf/terminal node
        """
        self.max_depth = max_depth
        self.min_size = 5

        #hyperparameter testing
        #self.max_depth = 5
        #self.min_size = min_size


    def terminal_node(self, group):
        """
        Function called in split() to assign terminal node label
        """
        labels = list()
        for row in group:
            labels.append(row)
        node_label = max(set(labels), key=labels.count)
        return node_label

    def split(self, data, depth):
        """
        Function called recursively to split
            the data in order to build a decision
            tree.

        Params
        ------
        data    : {left_node_data, right_node_data, left_node_labels, right_node_labels, split_index,
split_value}
        depth   : the current depth of the node in the decision tree

        Return
        ------
        """

        #check if we have reached the maximum depth of decision tree
        if depth >= self.max_depth:
            left_leaf_label = self.terminal_node(data['left node labels'])
            right_leaf_label =  self.terminal_node(data['right node labels'])
            data['left node'] = left_leaf_label
            data['right node'] = right_leaf_label
            return

        #check to see if we have reached min_size or is all labels are the same
        if len(data['left node data']) <= self.min_size or len(set(data['left node labels'])) == 1:
            left_leaf_label = self.terminal_node(data['left node labels'])
            data['left node'] = left_leaf_label

        else:
            #reassign the training and testing data to indices of left node
            x_train = data['left node data']
            y_train = data['left node labels']
            #delete the feature vector from x_train
            #np.delete(x_train, data['split index'])
            #get best split on new training and test set
            data['left node'] = get_split(x_train, y_train)
            #resplit the data and add counter to depth
            self.split(data['left node'], depth+1)
```

```
        #check to see if we have reached min_size or if all labels are the same
        if len(data['right node data']) <= self.min_size or len(set(data['right node labels'])) == 1:
            right_leaf_label = self.terminal_node(data['right node labels'])
            data['right node'] = right_leaf_label

        else:
            #reassign the training and testing data to indices of left node
            x_train = data['right node data']
            y_train = data['right node labels']
            #delete the feature vector from x_train
            #np.delete(x_train, data['split index'])
            #get best split on new training and test set
            data['right node'] = get_split(x_train, y_train)
            #resplit the data and add counter to depth
            self.split(data['right node'], depth+1)

        return


    def fit(self, x_train, y_train):
        """
        Fitting the KNN classifier

        Hint: Build the decision tree using
                splits recursively until a leaf
                node is reached

        """
        depth = 0

        tree = get_split(x_train, y_train)

        self.split(tree, depth)

        return tree

    def predict(self, x_test, node):
        """
        Predicting the test data

        Hint: Run the test data through the decision tree built
                during training (self.tree)
        """


        #check if the value of the feature is less than the split value
        if x_test[node['split index']] < node['split value']:
            #check if the node is a terminal node, which can be determined from the datatype
            if isinstance(node['left node'], dict):
                #reassign the node
                left_node = node['left node']
                #and recursively predict until you arrive at a terminal node
                prediction = self.predict(x_test, left_node)
                return prediction
            else:
                #once you arrive at a terminal node, assign the class to that data point
                predicted_class = node['left node']
                return predicted_class
        else:
            #check if the node is a terminal node which can be determined from the datatype
            if isinstance(node['right node'], dict):
                #reassign the node
                right_node = node['right node']
                #ans recursively split until you arrive at a terminal node
                prediction = self.predict(x_test, right_node)
                return prediction
            else:
                #once you arrive at a terminal node, assign the class to that data point
                predicted_class = node['right node']
                return predicted_class

        return predicted_class


def load_data(data_dir='Data'):

    """
    Function for loading the dataset from data_dir
        default: data_dir='Data'
    """
```

```
    # Load the data
    X_train = pd.read_csv(f'{data_dir}/X_Train.csv', header=None).values
    y_train = pd.read_csv(f'{data_dir}/Y_Train.csv', header=None).values[:,0]

    X_test = pd.read_csv(f'{data_dir}/X_Test.csv', header=None).values
    y_test = pd.read_csv(f'{data_dir}/Y_Test.csv', header=None).values[:,0]


    return {"train": dict(X=X_train, y=y_train),
            "test": dict(X=X_test, y=y_test)}

def main():

    # Set up the neighbors list
    max_depths = [2, 3, 5, 7, 9, 11]
    min_size = 5
    time_per_depth = []

    print("\n\n")
    print('Running decision_tree.py')

    #hyperparameter testing
    #testing new max_depths to find optimal hyperparameters
    #max_depths = [4, 5, 6, 7, 8]
    #testing some different min_sizes to find optimal min_size
    #min_sizes = [1, 2, 3, 4, 5, 6, 7]
    #max_depth = 5


    metrics = ['accuracy', 'f1_score', 'time']
    results = dict([(key, []) for key in metrics])

    # Load in data
    data = load_data('/Users/rossbrancati/Documents/Fall_2021_Classes/cs_589/assignments/hw1/Data')

    X_train = data['train']['X']
    y_train = data['train']['y']
    print(f"Training data loaded: X shape: {X_train.shape}, y shape: {y_train.shape}")

    X_test = data['test']['X']
    y_test = data['test']['y']
    print(f"Test data loaded: X shape: {X_test.shape}, y shape: {y_test.shape}")

    print(f"\t Class [0/1] ratio: Train [{(y_train==0).sum()}/{(y_train==1).sum()}]")
    print(f"\t Class [0/1] ratio: Test [{(y_test==0).sum()}/{(y_test==1).sum()}]")


    # Find/Record unique values for each feature to split on

    # Loop over depths
    for max_depth in max_depths:
        print("max_depth: ", max_depth)

    #testing some different min_sizes
    #loop over min terminal node sizes
    #for min_size in min_sizes:
        #print("min_size: ", min_size)

        #reset the empty prediction vector
        y_pred = np.array((), int)

        # start timer
        t0 = time.time()

        # instantiate DecisionTree instance
        dt = DecisionTree(max_depth=max_depth, min_size=min_size)

        # fit the model with X_train, y_train
        tree = dt.fit(X_train, y_train)

        # generate predictions on the test set X_test
        #loop over rows of X_test to avoid endless recursion
        for row in X_test:
            prediction = dt.predict(row, tree)
            y_pred = np.append(y_pred, int(prediction))

        #compute metrics F1_score on the test set with y_test
        F1_score, Accuracy = f1_score(y_test, y_pred)

        #stop timer
        t1 = time.time()
```

```
        # record time and report metric in milliseconds
        total_time = (t1-t0)*1000

        time_per_depth.append(total_time)
        results['accuracy'].append(Accuracy)
        results['f1_score'].append(F1_score)
        results['time'].append(total_time)


    print("\n Completed!")
    print(results)


    plt.plot(max_depths, time_per_depth)
    plt.scatter(max_depths, time_per_depth)

    #plt.plot(min_sizes, time_per_depth)
    #plt.scatter(min_sizes, time_per_depth)
    plt.title('Time to Complete Decision Tree\nat Specified Max Depth')
    plt.xlabel('Max Depths (count)')
    plt.ylabel('Time Per Depth (milliseconds)')
    #plt.title('Time to Complete Decision Tree\nat Specified Terminal Node Size')
    #plt.xlabel('Minimum Number of Observations in Terminal Node (count)')
    #plt.ylabel('Time Per Terminal Node Size (milliseconds)')


    plt.show()
    #plt.savefig()



if __name__ == '__main__':
    main()
```