Grading

The final score that you will receive for your programming assignment is generated in relation to the total points set in your programming assignment item—not the total point value in the nbgrader notebook. When calculating the final score shown to learners, the programming assignment takes the percentage of earned points vs. the total points provided by nbgrader and returns a score matching the equivalent percentage of the point value for the programming assignment.
DO NOT CHANGE VARIABLE OR METHOD SIGNATURES The autograder will not work properly if your change the variable or method signatures.

WARNING

Please refrain from using **print statements/anything that dumps large outputs(>500 lines) to STDOUT** to avoid running to into **memory issues**. Doing so requires your entire lab to be reset which may also result in loss of progress and you will be required to reach out to Coursera for assistance with this. This process usually takes time causing delays to your submission.

Validate Button

Please note that this assignment uses nbgrader to facilitate grading. You will see a **validate button** at the top of your Jupyter notebook. If you hit this button, it will run tests cases for the lab that aren't hidden. It is good to use the validate button before submitting the lab. Do know that the labs in the course contain hidden test cases. The validate button will not let you know whether these test cases pass. After submitting your lab, you can see more information about these hidden test cases in the Grader Output. ***Cells with longer execution times will cause the validate button to time out and freeze. Please know that if you run into Validate time-outs, it will not affect the final submission grading.***

Module 4: K-nearest neighbors

Run the cell below to ensure that the required packages are imported.

In [1]:
```
import math
import pickle
import gzip
import numpy as np
import matplotlib.pylab as plt
%matplotlib inline

# importing all the required libraries

from math import exp
import numpy as np
import pandas as pd
import sklearn
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
%matplotlib inline

from sklearn.metrics import roc_auc_score
from sklearn.metrics import roc_curve
```

Problem 1 : Building a K- Nearest neighbours classifier for handwritten digit recognition [15 pts, Peer Review]

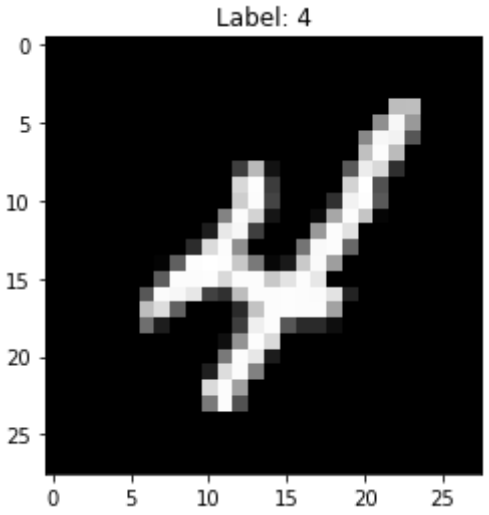
In this problem you will complete some code to build a k-nearest neighbour classifier to classify images of handwritten digits (0-9). For this purpose we will use a famous open-source dataset of handwritten digits called the MNIST that is ubiquitously used for testing a number of classification algorithms in machine learning.

In [2]:
```
# This cell sets up the MNIST dataset

class MNIST_import:
    """
    sets up MNIST dataset from OpenML
    """
    def __init__(self):

        df = pd.read_csv("data/mnist_784.csv")

        # Create arrays for the features and the response variable
        # store for use later
        y = df['class'].values
        X = df.drop('class', axis=1).values

        # Convert the labels to numeric labels
        y = np.array(pd.to_numeric(y))

        # create training and validation sets
        self.train_x, self.train_y = X[:5000,:], y[:5000]
        self.val_x, self.val_y = X[5000:6000,:], y[5000:6000]

data = MNIST_import()
```

In [3]:
```
def view_digit(x, label=None):
    fig = plt.figure(figsize=(3,3))
    plt.imshow(x.reshape(28,28), cmap='gray');
    plt.xticks([]); plt.yticks([]);
    if label: plt.xlabel("true: {}".format(label), fontsize=16)
```

Display a particular digit using the above function:

In [4]:
```
# Display a particular digit
def display_digit(data, index):
    digit = data.train_x[index].reshape(28, 28)
    plt.imshow(digit, cmap='gray')
    plt.title(f'Label: {data.train_y[index]}')
    plt.show()


# Display the digit at index 9
training_index = 9
display_digit(data, training_index)
```



Part 1 [5 points] Fill in the code in the following cell to determine the following quantities:

- Number of pixels in each image
- Number of examples in the training set
- Number of examples in the test set

In [5]:
```
# Calculate the required quantities
num_training_examples = data.train_x.shape[0]
num_test_examples = data.val_x.shape[0]
pixels_per_image = data.train_x.shape[1]

num_training_examples
num_test_examples
pixels_per_image
```

Out[5]: 784

In [6]:
```
# tests num_training_exampls, num_test_examples and pixels_per_image
```

Now that we have our MNIST data in the right form, let us move on to building our KNN classifier.

Part 2 [10 points]: Modify the class above to implement a KNN classifier. There are three methods that you need to complete:

- `predict` : Given an $m \times p$ matrix of validation data with m examples each with p features, return a length-m vector of predicted labels by calling the `classify` function on each example.
- `classify` : Given a single query example with p features, return its predicted class label as an integer using KNN by calling the `majority` function.
- `majority` : Given an array of indices into the training set corresponding to the K training examples that are nearest to the query point, return the majority label as an integer. If there is a tie for the majority label using K nearest neighbors, reduce K by 1 and try again. Continue reducing K until there is a winning label.

Notes:

- Don't even think about implementing nearest-neighbor search or any distance metrics yourself. Instead, go read the documentation for Scikit-Learn's BallTree (http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.BallTree.html) object. You will find that its implemented query (http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.BallTree.html#sklearn.neighbors.BallTree.query) method can do most of the heavy lifting for you.
- Do not use Scikit-Learn's KNeighborsClassifier in this problem. We're implementing this ourselves.
- Use the visible test cases to validate your code.

In [7]:
```
class KNN:
    """
    Class to store data for regression problems
    """
    def __init__(self, x_train, y_train, K=5):
        """
        Creates a kNN instance

        :param x_train: numpy array with shape (n_rows,1)- e.g. [[1,2],
[3,4]]
        :param y_train: numpy array with shape (n_rows,)- e.g. [1,-1]
        :param K: The number of nearest points to consider in classificat
ion
        """

        # Import and build the BallTree on training features
        from sklearn.neighbors import BallTree
        self.balltree = BallTree(x_train)

        # Cache training labels and parameter K
        self.y_train = y_train
        self.K = K


    def majority(self, neighbor_indices, neighbor_distances=None):
        """
        Given indices of nearest neighbors in training set, return the ma
jority label.
        Break ties by considering 1 fewer neighbor until a clear winner i
s found.

        :param neighbor_indices: The indices of the K nearest neighbors i
n self.X_train
        :param neighbor_distances: Corresponding distances from query poi
nt to K nearest neighbors.
        """

        while len(neighbor_indices) > 0:
            nearest_labels = self.y_train[neighbor_indices]
            labels, counts = np.unique(nearest_labels, return_counts=Tru
e)
            max_count = counts.max()
            majority_labels = labels[counts == max_count]
            if len(majority_labels) == 1:
                return majority_labels[0]
            neighbor_indices = neighbor_indices[:-1]  # Reduce K by 1



    def classify(self, x):
        """
        Given a query point, return the predicted label

        :param x: a query point stored as an ndarray
        """
        dist, ind = self.balltree.query([x], k=self.K)
        return self.majority(ind[0])
```

```
    def predict(self, X):
        """
        Given an ndarray of query points, return yhat, an ndarray of predictions

        :param X: an (m x p) dimension ndarray of points to predict labels for
        """
        return np.array([self.classify(x) for x in X])
```

In [8]:
```
# Sample tests for KNN class

import pytest
# set-up
X_train = np.array([[1,6], [6,4], [2,5], [1,3], [2,2], [3,1], [1,5], [2,3], [4,6], [3,5], [6,5], [0,4]])
y_train = np.array([+1, -1, +1, +1, -1, +1, +1, -1, +1, -1, +1, -1])
x = np.array([0,5])

# test k=2,
k2nn = KNN(X_train, y_train, K=2)
assert -1 == pytest.approx(k2nn.classify(x)), "KNN class doesn't perform as expected with two neighbors"


# test k=3
k3nn = KNN(X_train, y_train, K=3)
assert 1 == pytest.approx(k3nn.classify(x)), "KNN class doesn't perform as expected with three neighbors"


# test 3NN Predict
X = np.array([[2,5], [5,1]])
k3p = KNN(X_train, y_train, K=3)
yhat = k3p.predict(X)

# correct labels for the above two points(X).
ytrue = [1, -1]

for yh, yt in zip(yhat, ytrue):
    assert yh == yt, "Look at the predict function in the KNN class."
```

In [9]:
```
# tests KNN class
```

Part 3 : Checking how well your classifier does Use your `KNN` class to perform KNN on the validation data with $K = 3$ and do the following:

- **[Peer Review]** Create a **confusion matrix** (feel free to use the Scikit-Learn confusion_matrix (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html) function). Upload a screenshot or copy of your confusion matrix for this week's Peer Review assignment.
  Note: your code for this section may cause the Validate button to time out. If you want to run the Validate button prior to submitting, you could comment out the code in this section after completing the Peer Review.

In [10]:
```
# use your KNN class to perform KNN on the validation data with K = 3
# knn =
# val_yhat =

# create a confusion matrix

from sklearn.metrics import accuracy_score, classification_report, confus
ion_matrix
from sklearn.neighbors import BallTree

# Perform KNN with K=3
knn_classifier = KNN(data.train_x, data.train_y, K=3)

# Predict on validation data
val_yhat = knn_classifier.predict(data.val_x)

# Create confusion matrix
conf_matrix = confusion_matrix(data.val_y, val_yhat)
print('Confusion Matrix:')
print(conf_matrix)
```

```
Confusion Matrix:
[[112   0   0   0   0   0   1   0   0   0]
 [  0 106   0   0   0   0   0   1   0   1]
 [  1   2  86   2   0   0   0   2   0   0]
 [  1   1   0 111   0   2   0   0   0   0]
 [  0   2   0   0  82   0   0   0   0   4]
 [  0   0   0   2   2  75   1   0   0   0]
 [  0   0   0   0   0   1 104   0   2   0]
 [  0   2   0   0   0   1   0  93   0   5]
 [  1   1   0   1   1   0   2   1  81   1]
 [  1   0   0   1   2   0   0   2   0 100]]
```

In [11]:
```
print(data.val_x.shape)
```

```
(1000, 784)
```

Based on your confusion matrix, which digits seem to get confused with other digits the most? Put your answer in this week's Peer Review assignment.

Based on the provided confusion matrix, we can analyze which digits are most often confused with others. Here is a detailed breakdown:

```
Confusion Matrix:
[[112   0   0   0   0   0   1   0   0   0]
 [  0 106   0   0   0   0   0   1   0   1]
 [  1   2  86   2   0   0   0   2   0   0]
 [  1   1   0 111   0   2   0   0   0   0]
 [  0   2   0   0  82   0   0   0   0   4]
 [  0   0   0   2   2  75   1   0   0   0]
 [  0   0   0   0   0   1 104   0   2   0]
 [  0   2   0   0   0   1   0  93   0   5]
 [  1   1   0   1   1   0   2   1  81   1]
 [  1   0   0   1   2   0   0   2   0 100]]
```

Analysis:

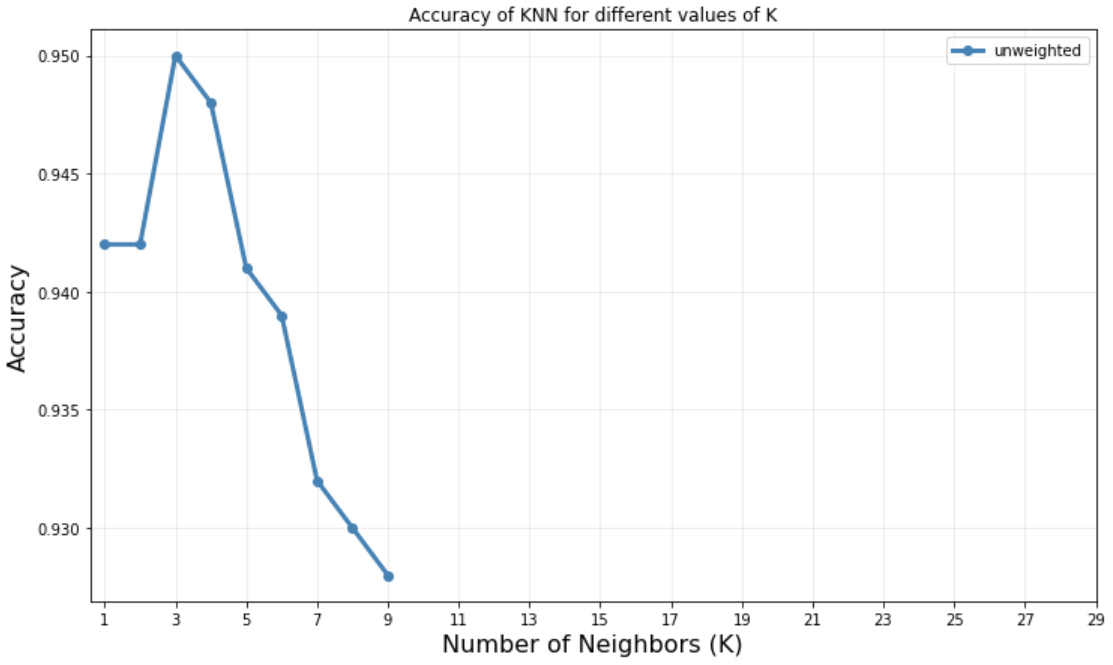
- **Digit 0**: Rarely confused, with only 1 instance being misclassified as digit 6.
- **Digit 1**: Mostly correctly classified, with very few confusions.
- **Digit 2**: Some confusion with digits 1 and 3.
- **Digit 3**: Few confusions with digits 1 and 5.
- **Digit 4**: Confused with digit 9.
- **Digit 5**: Confused with digits 3 and 4.
- **Digit 6**: Confused with digits 5 and 8.
- **Digit 7**: Confused with digits 1 and 9.
- **Digit 8**: Confused with digits 0, 1, 3, 4, 6, and 9.
- **Digit 9**: Confused with digits 4, 7, and 8.

Common Confusions:

- **Digit 2** with digits 1 and 3.
- **Digit 4** with digit 9.
- **Digit 5** with digits 3 and 4.
- **Digit 8** with several other digits, indicating it has more ambiguity.

Accuracy Plot [Peer Review]: Create a plot of the accuracy of the KNN on the test set on the same set of axes for K=1,2,…,20 (feel free to go out to K=30 if your implementation is efficient enough to allow it). Upload a copy or screenshot of the plot for this week's Peer Review assignment.
Note: your code for this section may cause the Validate button to time out. If you want to run the Validate button prior to submitting, you could comment out the code in this section after completing the Peer Review.

In [12]:
```
# Plot accuracy for K=1, 2, ..., 30
acc = []
allks = range(1, 10)

for k in allks:
    knn_classifier = KNN(data.train_x, data.train_y, K=k)
    val_yhat = knn_classifier.predict(data.val_x)
    acc.append(accuracy_score(data.val_y, val_yhat))

# Plot the results
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(12, 7))
ax.plot(allks, acc, marker="o", color="steelblue", lw=3, label="unweighte
d")
ax.set_xlabel("Number of Neighbors (K)", fontsize=16)
ax.set_ylabel("Accuracy", fontsize=16)
plt.xticks(range(1, 31, 2))
ax.grid(alpha=0.25)
plt.legend()
plt.title("Accuracy of KNN for different values of K")
plt.show()

# Print the value of K that results in highest accuracy
best_k = allks[np.argmax(acc)]
print(f"The value of K that results in the highest accuracy is: {best_
k}")
```




```
The value of K that results in the highest accuracy is: 3
```

Based on the plot, which value of K results in highest accuracy? Answer this question in this week's Peer Review assignment.

The value of K that results in the highest accuracy is: 3

Problem 2: Decision Tree, post-pruning and cost complexity parameter using sklearn 0.22 [10 points, Peer Review]

We will use a pre-processed natural language dataset in the CSV file "spamdata.csv" to classify emails as spam or not. Each row contains the word frequency for 54 words plus statistics on the longest "run" of captial letters.

Word frequency is given by:

$$f_i = m_i / N$$

Where f_i is the frequency for word i, m_i is the number of times word i appears in the email, and N is the total number of words in the email.

We will use decision trees to classify the emails.

Part A [5 points]: Complete the function `get_spam_dataset` to read in values from the dataset and split the data into train and test sets.

In [13]:
```
# def get_spam_dataset(filepath="data/spamdata.csv", test_split=0.1):
#     '''
#     get_spam_dataset

#     Loads csv file located at "filepath". Shuffles the data and splits
#     it so that the you have (1-test_split)*100% training examples and
#     (test_split)*100% testing examples.

#     Args:
#         filepath: location of the csv file
#         test_split: percentage/100 of the data should be the testing sp
lit

#     Returns:
#         X_train, X_test, y_train, y_test, feature_names
#         Note: feature_names is a list of all column names including isS
pam.

#         (in that order)
#         first four are  np.ndarray

#     '''

#     # your code here


#     return 0
```

In [14]:
```
import pandas as pd
from sklearn.model_selection import train_test_split

def get_spam_dataset(filepath="data/spamdata.csv", test_split=0.1):
    '''
    get_spam_dataset

    Loads csv file located at "filepath". Shuffles the data and splits
    it so that you have (1-test_split)*100% training examples and
    (test_split)*100% testing examples.

    Args:
        filepath: location of the csv file
        test_split: percentage/100 of the data should be the testing spli
t

    Returns:
        X_train, X_test, y_train, y_test, feature_names
        Note: feature_names is a list of all column names including isSpa
m.

        (in that order)
        first four are  np.ndarray

    '''

    # Load the dataset (assuming it's a comma-separated CSV)
    data = pd.read_csv(filepath, sep=" ")

    # Shuffle the data
    data = data.sample(frac=1, random_state=42).reset_index(drop=True)

    # Split into features and labels
    X = data.drop(columns=["isSPAM"]).values
    y = data["isSPAM"].values

    # Get the feature names (including the label "isSPAM")
    feature_names = data.columns.tolist()

    # Split the data into train and test sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=t
est_split, random_state=42)

    return X_train, X_test, y_train, y_test, feature_names
```

In [15]:
```
# TO-DO: import the data set into five variables: X_train, X_test, y_trai
n, y_test, label_names
# Uncomment and edit the line below to complete this task.

test_split = 0.1 # default test_split; change it if you'd like; ensure th
at this variable is used as an argument to your function
# your code here

X_train, X_test, y_train, y_test, label_names = get_spam_dataset()
```

In [16]:
```
X_train.shape, X_test.shape, y_train.shape, y_test.shape, label_names
```

```
Out[16]: ((4140, 57),
          (461, 57),
          (4140,),
          (461,),
          ['word_freq_make',
           'word_freq_address',
           'word_freq_all',
           'word_freq_3d',
           'word_freq_our',
           'word_freq_over',
           'word_freq_remove',
           'word_freq_internet',
           'word_freq_order',
           'word_freq_mail',
           'word_freq_receive',
           'word_freq_will',
           'word_freq_people',
           'word_freq_report',
           'word_freq_addresses',
           'word_freq_free',
           'word_freq_business',
           'word_freq_email',
           'word_freq_you',
           'word_freq_credit',
           'word_freq_your',
           'word_freq_font',
           'word_freq_000',
           'word_freq_money',
           'word_freq_hp',
           'word_freq_hpl',
           'word_freq_george',
           'word_freq_650',
           'word_freq_lab',
           'word_freq_labs',
           'word_freq_telnet',
           'word_freq_857',
           'word_freq_data',
           'word_freq_415',
           'word_freq_85',
           'word_freq_technology',
           'word_freq_1999',
           'word_freq_parts',
           'word_freq_pm',
           'word_freq_direct',
           'word_freq_cs',
           'word_freq_meeting',
           'word_freq_original',
           'word_freq_project',
           'word_freq_re',
           'word_freq_edu',
           'word_freq_table',
           'word_freq_conference',
           'char_freq_;',
           'char_freq_(',
           'char_freq_[',
           'char_freq_!',
           'char_freq_$',
           'char_freq_#',
           'capital_run_length_average',
           'capital_run_length_longest',
```

```
            'capital_run_length_total',
            'isSPAM'])
```

In [17]:
```
# tests X_train, X_test, y_train, y_test, and label_names
```

Part B[5 points] : Build a decision tree classifier using the sklearn toolbox. Then compute metrics for performance like precision and recall. This is a binary classification problem, therefore we can label all points as either positive (SPAM) or negative (NOT SPAM).

In [18]:
```
def build_dt(data_X, data_y, max_depth = None, max_leaf_nodes =None):
    '''
    This function does the following:
    1. Builds the decision tree classifier using sklearn
    2. Fits it to the provided data.


    Arguments
        data_X - a np.ndarray
        data_y - np.ndarray
        max_depth - None if unrestricted, otherwise an integer for the ma
ximum
                depth the tree can reach.

    Returns:
        A trained DecisionTreeClassifier
    '''

    # Initialize the DecisionTreeClassifier with the specified parameters
    clf = DecisionTreeClassifier(max_depth=max_depth, max_leaf_nodes=max_
leaf_nodes, random_state=42)

    # Fit the classifier to the data
    clf.fit(data_X, data_y)

    return clf
```

In [19]:
```
# tests build_dt
```

Part C [Peer Review]: Here we are going to use `calculate_precision` and `calculate_recall` functions to see how these metrics change when parameters of the tree are changed.

In [20]:
```
def calculate_precision(y_true, y_pred, pos_label_value=1.0):
    '''
    This function accepts the labels and the predictions, then
    calculates precision for a binary classifier.

    Args
        y_true: np.ndarray
        y_pred: np.ndarray

        pos_label_value: (float) the number which represents the postiive
        label in the y_true and y_pred arrays. Other numbers will be take
n
        to be the non-positive class for the binary classifier.

    Returns precision as a floating point number between 0.0 and 1.0
    '''


    true_positive = np.sum((y_pred == pos_label_value) & (y_true == pos_l
abel_value))
    predicted_positive = np.sum(y_pred == pos_label_value)

    if predicted_positive == 0:
        return 0.0

    precision = true_positive / predicted_positive

    return precision

def calculate_recall(y_true, y_pred, pos_label_value=1.0):
    '''
    This function accepts the labels and the predictions, then
    calculates recall for a binary classifier.

    Args
        y_true: np.ndarray
        y_pred: np.ndarray

        pos_label_value: (float) the number which represents the postiive
        label in the y_true and y_pred arrays. Other numbers will be take
n
        to be the non-positive class for the binary classifier.

    Returns precision as a floating point number between 0.0 and 1.0
    '''


    true_positive = np.sum((y_pred == pos_label_value) & (y_true == pos_l
abel_value))
    actual_positive = np.sum(y_true == pos_label_value)

    if actual_positive == 0:
        return 0.0

    recall = true_positive / actual_positive

    return recall
```

In [21]:
```
# Sample Test cell
ut_true = np.array([1.0, 1.0, 0.0, 1.0, 1.0, 0.0])
ut_pred = np.array([1.0, 1.0, 1.0, 1.0, 0.0, 1.0])
prec = calculate_precision(ut_true, ut_pred, 1.0)
recall = calculate_recall(ut_true, ut_pred, 1.0)
assert prec == 0.6, "Check the precision value returned from your calcula
te_precision function."
assert recall == 0.75, "Check the recall value returned from your calcula
te_recall function."
```

1. Modifying `max_depth` :

   - Create a model with a shallow `max_depth` of 2. Build the model on the training set.
   - Report precision/recall on the test set.
   - Report depth of the tree.

In [22]:
```
# TODO : Complete the first subtask for max_depth

dt = build_dt(X_train, y_train, max_depth=2, max_leaf_nodes=None)


y_test_hat = dt.predict(X_test)


prediction = calculate_precision(y_test, y_test_hat)


recall = calculate_recall(y_test, y_test_hat)

print("Precision: {:0.2f} Recall: {:0.2f} Tree Depth: {}".format(predicti
on, recall, dt.get_depth()))
```

```
Precision: 0.84 Recall: 0.71 Tree Depth: 2
```

Submit a screenshot of your code for this week's Peer Review assignment.

1. Modifying `max_leaf_nodes` :

   - Create a model with a shallow `max_leaf_nodes` of 4. Build the model on the training set.
   - Report precision/recall on the test set.
   - Report depth of the tree.

In [23]:
```
# TODO : Complete the second subtask for max_depth

clf_leaf_nodes_4 = build_dt(X_train, y_train, max_depth = None, max_leaf_
nodes=4)

y_test_hat = clf_leaf_nodes_4.predict(X_test)

prediction = calculate_precision(y_test, y_test_hat)


recall = calculate_recall(y_test, y_test_hat)

print("Precision: {:0.2f} Recall: {:0.2f} Tree Depth: {}".format(predicti
on, recall, dt.get_depth()))
```

```
Precision: 0.84 Recall: 0.71 Tree Depth: 2
```

In your Peer Review answer the following question:
How do precision and recall compare when you modify the max depth compared to the max number of leaf nodes? (Make sure to run your models a few times to get an idea).

1. **Modifying `max_depth` :** The decision tree was limited to a maximum depth of 2, which yielded a precision of 0.84 and a recall of 0.71.
2. **Modifying `max_leaf_nodes` :** The decision tree was limited to a maximum of 4 leaf nodes, which also yielded the same precision of 0.84 and recall of 0.71.

Explanation:

- **Tree Depth (`max_depth`)**: This parameter controls the maximum number of levels in the tree. A smaller depth leads to a simpler model that may underfit the data. However, in this case, the depth of 2 provided sufficient complexity to achieve the reported precision and recall.
- **Number of Leaf Nodes (`max_leaf_nodes`)**: This parameter limits the total number of leaf nodes in the tree. Setting a limit on leaf nodes also results in a simpler model but focuses on controlling the number of final predictions (or outcomes). In this case, restricting the tree to 4 leaf nodes led to a tree that also achieved the same precision and recall as limiting the depth.

Comparison and Insights:

- **Same Precision and Recall:** The fact that both models produced the same precision and recall indicates that the configurations resulted in models of similar complexity, even though they were constrained by different parameters (`max_depth` vs. `max_leaf_nodes`). Both constraints led to a tree with 2 levels of depth and apparently the same overall structure.
- **Model Structure:** It suggests that, under the dataset and conditions you were working with, a maximum tree depth of 2 and a maximum of 4 leaf nodes may have led to very similar or identical decision trees. This can happen if, for example, the optimal splitting points in the tree naturally fit within both constraints.

Conclusion:

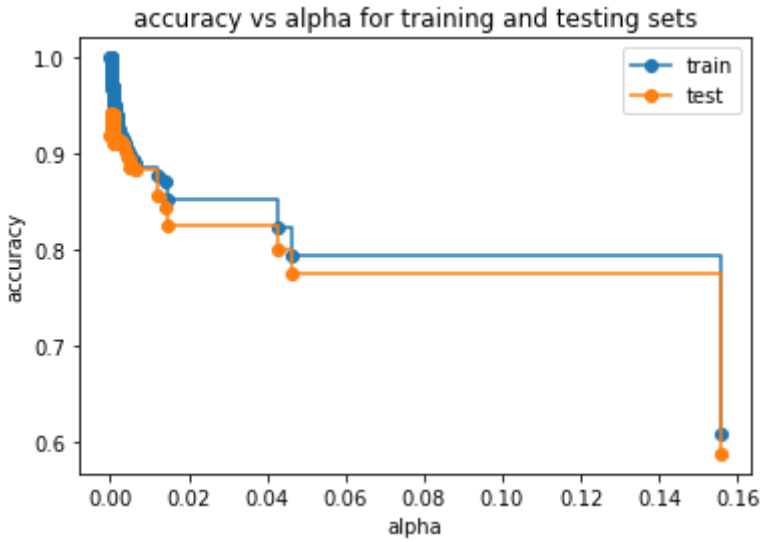
Modifying `max_depth` and `max_leaf_nodes` can sometimes lead to similar model performance if the constraints result in a tree with similar complexity. In your case, both constraints yielded the same precision and recall, indicating that the decision tree structures under these constraints were likely very similar, despite the different parameters used to achieve them.

Part D [Peer Review] : In class, we used gridsearchCV to do hyperparameter tuning to select the different parameters like `max_depth` to see how our tree grows and avoids overfitting. Here, we will use cost complexity pruning parameter α sklearn 0.22.1[https://scikit-learn.org/stable/user_guide.html (https://scikit-learn.org/stable/user_guide.html)] to prune our tree after training so as to improve accuracy on unseen data. In this exercise you will iterate over different `ccp_alpha` values and identify how performance is modulated by this parameter.
Note: your code for this section may cause the Validate button to time out. If you want to run the Validate button prior to submitting, you could comment out the code in this section after completing the Peer Review.

In [25]:

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt

# Assuming build_dt is a function that builds and returns a decision tree
classifier
# Replace this with the actual implementation of build_dt or use the foll
owing directly:
# dt = DecisionTreeClassifier(random_state=0)
# dt.fit(X_train, y_train)
dt = build_dt(X_train, y_train)

# Get the cost complexity pruning path
path = dt.cost_complexity_pruning_path(X_train, y_train)
ccp_alphas, impurities = path.ccp_alphas, path.impurities

clfs = []  # List to store classifiers for different alpha values

# Iterate over ccp_alpha values to train trees
for ccp_alpha in ccp_alphas:
    clf = DecisionTreeClassifier(random_state=0, ccp_alpha=ccp_alpha)
    clf.fit(X_train, y_train)
    clfs.append(clf)

print("Number of nodes in the last tree is: {} with ccp_alpha: {}".format
(
      clfs[-1].tree_.node_count, ccp_alphas[-1]))

# Generate train and test scores for different classifiers
train_scores = [accuracy_score(y_train, clf.predict(X_train)) for clf in
clfs]
test_scores = [accuracy_score(y_test, clf.predict(X_test)) for clf in clf
s]

# Plot the results
fig, ax = plt.subplots()
ax.set_xlabel("alpha")
ax.set_ylabel("accuracy")
ax.set_title("accuracy vs alpha for training and testing sets")
ax.plot(ccp_alphas, train_scores, marker='o', label="train", drawstyle="s
teps-post")
ax.plot(ccp_alphas, test_scores, marker='o', label="test", drawstyle="ste
ps-post")
ax.legend()
plt.show()
```

```
Number of nodes in the last tree is: 1 with ccp_alpha: 0.155646664668709
47
```



The parameter `ccp_alpha` in cost-complexity pruning is crucial in controlling the balance between underfitting and overfitting in decision trees. This figure depicts how accuracy changes as a function of the pruning parameter, `ccp_alpha` , for both the training and testing sets.

From your graph, here's a breakdown of how performance is modulated by the `ccp_alpha` parameter:

1. **Small `ccp_alpha` Values (Near 0):**

   - At very low `ccp_alpha` values, the tree is less pruned, and the number of nodes is high.
   - In this region, both the training and testing accuracies are initially high. However, as indicated by the slight gap between the two curves, the model might be overfitting: performing better on the training set than on the testing set.

1. **Increasing `ccp_alpha` :**

   - As `ccp_alpha` increases, the tree is pruned more aggressively, meaning nodes are removed, simplifying the tree.
   - Both training and testing accuracies drop, indicating that the model is becoming less complex and might start to underfit.

1. **Moderate `ccp_alpha` :**

   - At some moderate values of `ccp_alpha` , the training and testing accuracies are quite similar. This indicates a good balance between underfitting and overfitting. In your case, it appears around 0.02 to 0.06, where the curves are closely aligned. This is the region where pruning seems to be having the desired effect, improving generalization.

1. **High `ccp_alpha` Values:**

   - As `ccp_alpha` becomes too large, both training and testing accuracies drop significantly, indicating that the tree has become too simple (over-pruned). In this case, the model loses its ability to capture important patterns in the data, leading to underfitting.
   - The last point on the graph shows that the tree has only 1 node left when `ccp_alpha = 0.15564666466870947` . This results in a very low accuracy, as the tree has been pruned too aggressively, reducing its ability to classify correctly.

Overall Conclusion:

The performance of the decision tree is strongly modulated by the `ccp_alpha` parameter. A small value of `ccp_alpha` leads to an overly complex tree that may overfit, while a large value over-prunes the tree, leading to underfitting. The best performance is typically observed at moderate `ccp_alpha` values, where the training and testing accuracy are closely aligned, indicating good generalization to unseen data.

In [26]:
```
# dt = build_dt(X_train, y_train)

# path = dt.cost_complexity_pruning_path(X_train,y_train) #post pruning
# ccp_alphas, impurities = path.ccp_alphas, path.impurities

# clfs = [] # VECTOR CONTAINING CLASSIFIERS FOR DIFFERENT ALPHAS
# # TODO: iterate over ccp_alpha values
# # your code here


# print("Number of nodes in the last tree is: {} with ccp_alpha: {}".form
at(
#       clfs[-1].tree_.node_count, ccp_alphas[-1]))

# # TODO: next, generate the train and test scores and plot the variation
in these scores with increase in ccp_alpha
# # The code for plotting has been provided; edit the train_scores and te
st_scores variables for the right plot to be generated
# train_scores = []
# test_scores = []

# # your code here


# fig, ax = plt.subplots()
# ax.set_xlabel("alpha")
# ax.set_ylabel("accuracy")
# ax.set_title("accuracy vs alpha for training and testing sets")
# ax.plot(ccp_alphas, train_scores, marker='o', label="train",
#         drawstyle="steps-post")
# ax.plot(ccp_alphas, test_scores, marker='o', label="test",
#         drawstyle="steps-post")
# ax.legend()
# plt.show()
```

In [ ]: