### 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.

### 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()
```

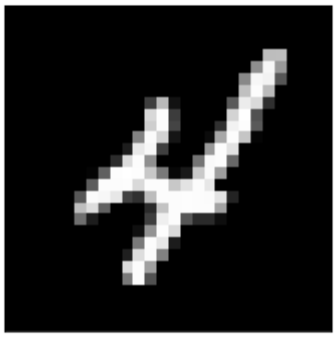
In [3]:
```
def view_digit(x, label=None):
    fig = plt.figure(figsize=(3,3))
    plt.imshow(x.reshape(28,28), cmap='gray');
    plt.xticks([]); plt.yticks([]);
    if label: plt.xlabel("true: {}".format(label), fontsize=16)
```

Display a particular digit using the above function:

In [4]:
```
training_index = 9
# your code here
xtrain = data.train_x
ytrain = data.train_y
xval = data.val_x
yval = data.val_y

#view_digit(xtrain[training_index])
view_digit(xtrain[training_index])
print(ytrain[training_index])
```

```
#view_digit(xval[training_index])
#print(xval[training_index])
```

4



**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]:
```
# Here are the numbers you need to provide here:
num_training_examples = 5000
num_test_examples = 1000
pixels_per_image = 784

# your code here
print(xtrain.shape)
print(xval.shape)
print(np.sqrt(784))
print(num_training_examples)
print(num_test_examples)
print(pixels_per_image)
```

```
(5000, 784)
(1000, 784)
28.0
5000
1000
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 object. You will find that its implemented 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.

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 classificati
on
        """

        # 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 maj
ority label.
        Break ties by considering 1 fewer neighbor until a clear winner is
 found.

        :param neighbor_indices: The indices of the K nearest neighbors in
 self.X_train
        :param neighbor_distances: Corresponding distances from query poin
t to K nearest neighbors.
        """

        # your code here
        K = self.K
        label, freq = np.unique(self.y_train[neighbor_indices[:K]], return
```

```
_counts = True)
        winner = np.argwhere(freq == np.amax(freq))
        m = winner.size

        if m > 1:
            K -= 1
            while K > 0:
                #print("There is a tie, let's try one less neighbor, count
: {} and winners' indices are:{}".format(m, winner),
                #  "\nwith values:{}".format(label))
                #print("The nearest {} neighbors are {}".format(K, self.y_
train[neighbor_indices[:,:K]]))
                #print("Considering one few neighbor... K = {}".format(K))
                label, freq = np.unique(self.y_train[neighbor_indices[:,:K
]], return_counts = True)
                winner = np.argwhere(freq == np.amax(freq))

                m = winner.size
                #print("Re-run complete. Now we have K = {}, and number of
 winners {}".format(K, m))

                K -= 1
            return label[np.argmax(freq)]

        return label[np.argmax(freq)]




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

        :param x: a query point stored as an ndarray
        """
        # your code here

        self.neighbor_distances, self.neighbor_indices = self.balltree.que
ry(x.reshape(1, -1), k=self.K)
        #self.majority(self.neighbor_indices, self.neighbor_distances)
        return self.majority(self.neighbor_indices, self.neighbor_distance
s)

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

        :param X: an (m x p) dimension ndarray of points to predict labels
 for
        """
        # your code here
        #pred_label = np.ndarray(shape = X.shape)
        pred_label = []

        for i, point in enumerate(X):
```

```
            x = X[i]
            pred_label.append(self.classify(x))
        yhat = np.array(pred_label).reshape([-1, ])
        print(yhat.shape)
        return yhat #np.array(pred_label).reshape(X.shape[0],)
```

In [8]:
```
# 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 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.

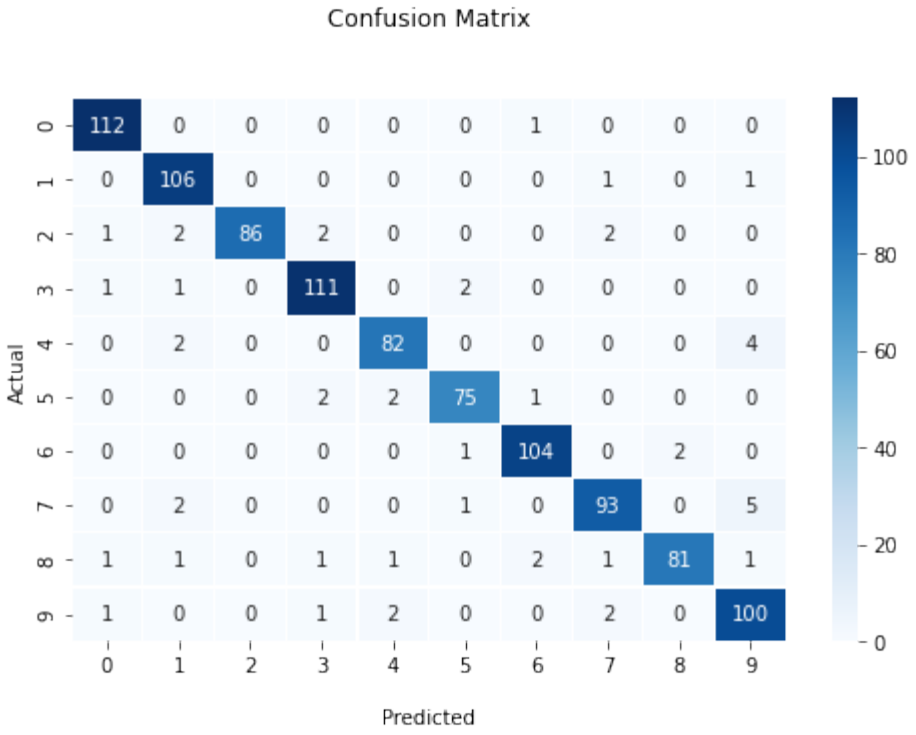
In [9]:
```
# use your KNN class to perform KNN on the validation data with K = 3
knn = KNN(xtrain, ytrain, K = 3)
val_yhat = knn.predict(xval)

# create a confusion matrix
# your code here



from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.metrics import confusion_matrix
y_true = yval
conf_matrix = confusion_matrix(y_true, val_yhat)
```

```
(1000,)
```

In [10]:
```
import seaborn as sns
print(conf_matrix)

plt.figure(figsize = (8, 5))
confm = sns.heatmap(conf_matrix, annot=True, cmap = 'Blues', fmt='g', line
widths = 0.1)
confm.set_title('Confusion Matrix\n\n');
confm.set_xlabel('\nPredicted')
confm.set_ylabel('Actual');
plt.show()
```

```
[[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]]
```




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.

**Accuracy Plot [Peer Review]**: Create a plot of the accuracy of the KNN on the test set on the same set of axes for =1,2,…,20 (feel free to go out to =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 [11]:
```
acc = []
wacc = []
allks = range(1,30)

# your code here
#a = confusion_matrix(y_true, val_yhat).ravel()
from sklearn.metrics import accuracy_score
#accur = accuracy_score(y_true, val_yhat)
#print(accur)
#-------------------------------------------------
#for i in allks:
#    knn = KNN(xtrain, ytrain, K = i)
#    y_pred = knn.predict(xval)
#    acc.append(accuracy_score(y_true, y_pred))

# you can use this code to create your plot
#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 neighbors", fontsize=16)
#ax.set_ylabel("accuracy", fontsize=16)
#plt.xticks(range(1,31,2))
#ax.grid(alpha=0.25)
```

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

## 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 [19]:
```
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 split

    Returns:
        X_train, X_test, y_train, y_test, feature_names

        (in that order)
        first four are  np.ndarray

    '''

    # your code here
    df = pd.read_csv(filepath, sep = ' ')
    feature_names = df.columns
    y = df['isSPAM'].values
    X = df.drop('isSPAM', axis = 1).values
```

```
    y = np.array(pd.to_numeric(y))

    #print(feature_names)
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=te
st_split, random_state=1)

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

In [27]:
```
# TO-DO: import the data set into five variables: X_train, X_test, y_train
, 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 tha
t this variable is used as an argument to your function
# your code here
filepath = "data/spamdata.csv"
xtr, xte, ytr, yte, label_names = get_spam_dataset(filepath = filepath, te
st_split = test_split)
print(xtr.shape, xte.shape, ytr.shape, yte.shape)
clf = DecisionTreeClassifier()
clf = clf.fit(xtr, ytr)
y_pred = clf.predict(xte)
print(y_pred)
print(sklearn.metrics.precision_score(yte,y_pred))
```

```
(4140, 57) (461, 57) (4140,) (461,)
[1 0 1 0 0 1 1 1 1 1 0 1 1 1 0 0 0 1 1 0 0 0 0 1 1 1 1 0 0 1 1 1 0 0 1 0 0
 0 0 0 0 1 1 0 0 0 1 0 0 1 0 0 1 1 1 0 1 0 0 0 1 0 0 0 0 0 1 0 0 0 1 1 0 0
 0 1 1 1 1 1 0 1 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 1 1 1 0 1 0 0 0 0 1 0 0 1
 0 0 0 1 1 0 0 1 1 1 1 0 0 0 0 1 0 0 0 1 0 1 0 0 1 0 1 1 0 1 0 0 1 0 0 0 0
 0 0 0 0 1 0 0 0 1 1 0 0 0 0 1 1 0 0 1 0 1 0 0 1 0 0 0 1 0 0 0 1 1 1 1 0 0
 0 1 0 0 1 0 1 1 0 0 0 0 1 1 1 0 1 1 0 0 0 0 0 1 0 1 0 1 0 1 1 1 0 0 0 1 0
 0 1 0 1 0 0 0 0 1 0 0 0 0 0 1 0 1 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
 1 1 1 0 0 0 0 0 0 1 0 1 1 1 0 0 0 0 0 1 0 1 0 1 1 0 0 1 0 0 1 1 1 0 1 1 0
 0 1 1 1 0 0 0 0 1 0 0 0 1 1 0 1 0 1 1 1 0 0 0 1 0 1 1 0 0 1 0 0 0 1 0 0 1
 0 0 0 1 0 0 1 0 0 0 1 1 0 1 1 0 0 0 0 0 1 0 1 0 1 0 1 0 1 1 1 1 1 1 1 0
 0 0 1 0 0 1 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 1 1 0 0 1 0 1 0 0 0 0 0 0 1
 0 1 1 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 1 0 1 1 0 0 0 0 1 0 0 0 1 0 1 1 0 0 0
 0 1 1 0 1 0 0 0 1 0 1 1 0 0 0 0 0]
0.8833333333333333
```

In [14]:
```
# 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 [39]:
```
def build_dt(data_X, data_y, max_depth = None, max_leaf_nodes =None):
    '''
    This function builds the decision tree classifier and
    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 max
imum
                depth the tree can reach.

    Returns:
        A trained DecisionTreeClassifier
    '''

    # your code here
    clf = DecisionTreeClassifier(max_depth=max_depth, max_leaf_nodes=max_l
eaf_nodes)
    clf = clf.fit(data_X, data_y)

    return clf
```

In [40]:
```
# 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 [41]:
```
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 taken
        to be the non-positive class for the binary classifier.

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



    # your code here
    TP, FP, TN, FN = 0, 0, 0, 0
    for i in range(len(y_pred)):
        if (y_pred[i] == 1) & (y_true[i] == 1):
            TP += 1
        elif (y_pred[i] == 1) & (y_true[i] == 0):
            FP += 1
        elif (y_pred[i] == 0) & (y_true[i] == 0):
            TN += 1
        else:
            FN += 1
    precision = TP / (TP + FP)
    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 taken
        to be the non-positive class for the binary classifier.

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

    # your code here
    TP, FP, TN, FN = 0, 0, 0, 0
    for i in range(len(y_pred)):
        if (y_pred[i] == 1) & (y_true[i] == 1):
            TP += 1
        elif (y_pred[i] == 1) & (y_true[i] == 0):
            FP += 1
        elif (y_pred[i] == 0) & (y_true[i] == 0):
            TN += 1
        else:
            FN += 1
    recall = TP / (TP + FN)

    return recall
```

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 [42]:
```
# TODO : Complete the first subtask for max_depth

# your code here
dtr = build_dt(xtr, ytr, max_depth = 2)
y_pred = dtr.predict(xte)
#print(y_test)

#print(y_pred)
p, r, d = calculate_precision(yte, y_pred), calculate_recall(yte, y_pred),
 dtr.get_depth()
print("Depth of tree is {}, \nRecall: {}, \nPrecision: {}".format(d, r, p)
)
```

```
Depth of tree is 2,
Recall: 0.7119565217391305,
Precision: 0.9492753623188406
```

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 [43]:
```
# TODO : Complete the second subtask for max_depth

# your code here
dtr = build_dt(xtr, ytr, max_depth = None, max_leaf_nodes = 4)
y_pred = dtr.predict(xte)
p, r, d = calculate_precision(yte, y_pred), calculate_recall(yte, y_pred),
 dtr.get_depth()
print("Depth of tree is {}, \nRecall: {}, \nPrecision: {}".format(d, r, p)
)
```

```
Depth of tree is 3,
Recall: 0.7934782608695652,
Precision: 0.8488372093023255
```

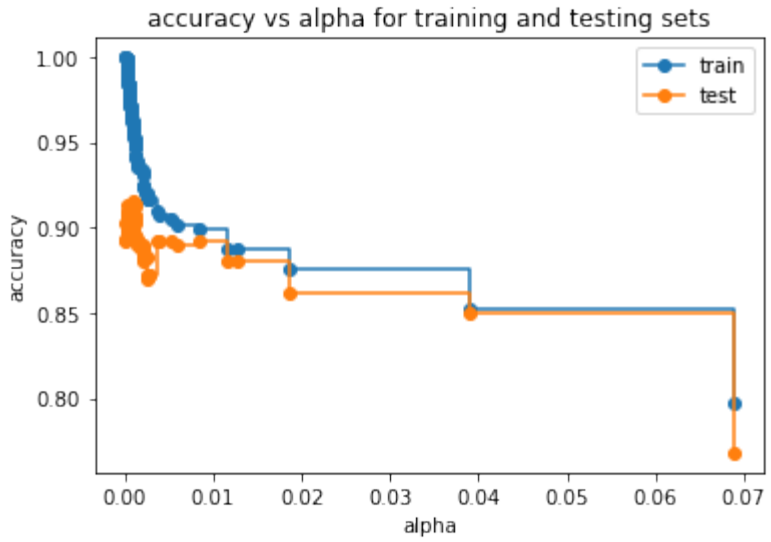
In your Peer Review answer the following question:
How do precision and recall compare when you modify the max depth compared to the max number of leaf nodes? (Make sure to run your models a few times to get an idea).

**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 $\alpha$ sklearn 0.22.1[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 [44]:
```
dt = build_dt(xtr, ytr)

path = dt.cost_complexity_pruning_path(xtr, ytr) #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
for ccp_alpha in ccp_alphas:
    clf = DecisionTreeClassifier(ccp_alpha = ccp_alpha)
    clf.fit(xtr, ytr)
    clfs.append(clf)



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

clfs = clfs[:-1]
```

```
ccp_alphas = ccp_alphas[:-1]
node_counts = [clf.tree_.node_count for clf in clfs]
depth = [clf.tree_.max_depth for clf in clfs]
# 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 test_
scores variables for the right plot to be generated
train_scores = [clf.score(xtr, ytr) for clf in clfs]
test_scores = [clf.score(xte, yte) for clf in clfs]

# 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()
```

```
Number of nodes in the last tree is: 1 with ccp_alpha: 0.1583583035325553
```




In [ ]: