Problem Set 1 - K-Nearest Neighbor and NaiveBayes classifiers

CSCI 4622 - Spring 2022

Kyle Moe

Only submit this notebook to canvas (no zip files).

For today's assignment, we will be implementing our own K-Nearest Neighbors classifier (KNNClassifier) algorithm and a Naive Bayes classifier.

But Professor Quigley, hasn't someone else already written KNN before?

Yes, you are not the first to implement KNN, or basically any algorithm we'll work with in this class. But

1. I'll know that you know what's really going on
2. You'll know you can do it, because
   A. someday you might have to implement some machine learning algorithm from scratch - maybe for a new platform (do you need to run python on your SmartToaster just to get it to learn how users like their toast?), maybe because you want to tweak the algorithm (there's always a better approach...),
   B. maybe because you're working on something important, and you need to control exactly what's on there (should you really be running anaconda on your secret spy plane?).

That said - we're not going to implement *everything*. We'll start by importing a few helper functions

In [3]:
```
import numpy as np
import matplotlib.pyplot as plt
import sklearn.neighbors
import sklearn.metrics
import data
import tests
%matplotlib inline
```

Wait a minute - didn't we just import Scikit-learn (sklearn)? The package with baked-in machine learning tools? Yes - but it also has a ton of helper functions that we'll be using later.

You will be guided through the different questions and you'll be expected to complete the classes and the functions following the provided signatures. Sometimes at the end of a question we would provide a difficulty estimate using the average scored by students who attempted the question (or a similar one) in previous assignments.

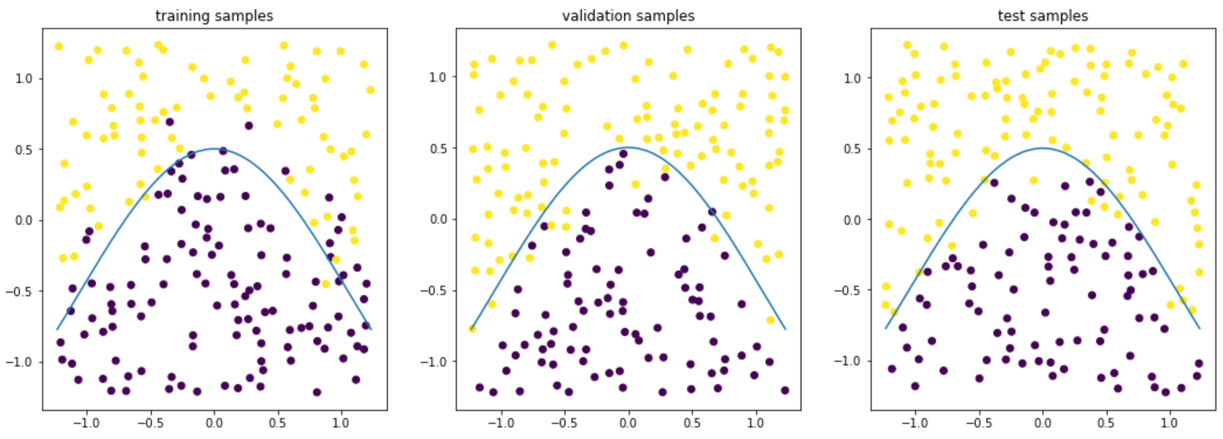
Remember to avoid adding positional arguments and make sure the returned values have the correct format (this applies to all assignments). The alternative is that your solution might be rejected by the auto-grader (we won't be debugging your code). We will provide some basic sanity checks. They're in no means exhaustive and passing them does not imply your solution is 100% correct.

For example, you're required to complete the method `compute_something` of class A. We provide examples of acceptable solutions.

In [4]:
```
# This cell can be removed from the submitted notebook
class A:
    def compute_something(self, X):
        """
        :return: numpy array of zeros, with shape (4,)
        """
        # BEGIN
        answer = None
        # END
        return answer


class A1:  # Acceptable solution:
    # - the added y is an optional argument and omitting it does not affect the s
    # - the returned object has the expected structure.
    def compute_something(self, X, y=None):
        # BEGIN
        return np.zeros((4,))
        # END


class A2:  # Wrong format:
    # - your solution requires a new positional argument y!
    # - the returned object does not have the expected format!
    # - solution outside the delimiters # BEGIN # END
    def compute_something(self, X, y):
        return [0, 0, 0, 0]


class A3:  # Acceptable solution:
    # You're free to add your own helper functions/methods
    def compute_something(self, X):
        # BEGIN
        return self.get_zeros(4)
        # END

    def get_zeros(self, i):
        return np.zeros((i,))
```

First, let's also load a dataset to play with and start working to build out our own classifier.

In [5]:
```
binary_data = data.BinaryData()
fig, axs = plt.subplots(1, 3)
fig.set_figheight(6), fig.set_figwidth(18)
for i, name in enumerate(["training", "validation", "test"]):
    axs[i].plot(*binary_data.boundary())
    axs[i].set_title("%s samples" % name)
axs[0].scatter(binary_data.X_train[:, 0], binary_data.X_train[:, 1], c=binary_dat
axs[1].scatter(binary_data.X_valid[:, 0], binary_data.X_valid[:, 1], c=binary_dat
axs[2].scatter(binary_data.X_test[:, 0], binary_data.X_test[:, 1], c=binary_data.
plt.show()
```



We have data! The `binary_data` instance has the following attributes:

- a training set (`X_train, y_train`): to train the model and on which the prediction is based
- a validation set (`X_valid, y_valid`): to select the best **hyper-parameters** of the model
- a test set (`X_test, y_test`): to evaluate the performance of the model on unseen data

Problem 1: Complete our KNN Classifier - 40 Points

The KNNClassifier class we're implementing will have similar design to the K-Nearest Neighbors classifier class from *scikit-learn*:

- Initialize the classifier with corresponding parameters (number of neighbors k)
- A `fit` method that uses the training data
- A `predict` method that returns the predicted labels given data `X`

We've written out a lot of the structure for you for consistency across different parts of the assignment and so you can focus on the "important" stuff that actually relates to the machine learning itself.

In [37]:
```
class KNNClassifier:

    def __init__(self, k=5):
        """
        Initialize our custom KNN classifier
        :param k: the number of nearest neighbors to consider for classification
        """
        self._k = k
        self._ball_tree = None
        self._y = None
        self.label_to_index = None
        self.index_to_label = None

    def fit(self, X, y):
        """
        Fit the model using the provided data
        :param X: 2-D np.array of shape (number training samples, number of featu
        :param y: 1-D np.array of shape (number training samples,)
        :return: self
        """
        self._ball_tree = sklearn.neighbors.BallTree(X)  # See documentation of B
        self._y = y
        # Should be used to map the classes to {0,1,..C-1} if needed (C is the nu
        # We can assume that the training data contains samples from all the poss
        classes = np.unique(y)
        self.label_to_index = dict(zip(classes, range(classes.shape[0])))
        self.index_to_label = dict(zip(range(classes.shape[0]), classes))

        return self

    def majority_vote(self, indices_nearest_k, distances_nearest_k=None):
        """
        Given indices of the nearest k neighbors for each point, report the major
        :param indices_nearest_k: np.array containing the indices of training nei
        :param distances_nearest_k: np.array containing the corresponding distanc
        :return: The majority label for each row of indices, shape (M,)
        """

        # Workspace 1.1
        # TODO: Determine majority for each row of indices_nearest_k
        # TODO: if there is a tie, remove the farthest neighbor until the tie is
        #BEGIN
        voted_labels = np.empty(indices_nearest_k.shape[0])
        #for point in indices_nearest_k:
        for idx, item in enumerate(indices_nearest_k):
            # array to hold labels of nearest neighbors
            labels = []
            for index in item:
                labels.append(self._y[index])
            # Find which class has most votes
            max_vote = max([labels.count(i) for i in set(labels)])
            # find if tie exists
            max_vote_alts = [i for i in set(labels) if labels.count(i)==max_vote]
            # pop furthest neighbor
            while(len(max_vote_alts) > 1):
                labels.pop()
                max_vote = max([labels.count(i) for i in set(labels)])
                max_vote_alts = [i for i in set(labels) if labels.count(i)==max_v
            voted_labels[idx] = max_vote_alts[0]
        # code here
        #END
        return voted_labels

    def predict(self, X):
        """
        Given new data points, classify them according to the training data provi
        You should use BallTree to get the distances and indices of the nearest k
        :param X: feature vectors (num_points, num_features)
        :return: 1-D np.array of predicted classes of shape (num_points,)
        """
        # Workspace 1.2
        #BEGIN
        distances_nearest_k, indices_nearest_k = self._ball_tree.query(X, k=self.
        #END
        return self.majority_vote(indices_nearest_k, distances_nearest_k)

    def confusion_matrix(self, X, y):
        """
        Generate the confusion matrix for the given data
        :param X: an np.array of feature vectors of points, shape (N, n_features)
        :param y: the corresponding correct classes of our set, shape (N,)
        :return: a C*C np.array of counts, where C is the number of classes in ou
        """
        # The rows of the confusion matrix correspond to the counts from the true
        # Workspace 1.3
        # TODO: Run classification for the test set X, compare to test answers y,
        c_matrix = np.zeros((len(self.label_to_index), len(self.label_to_index)),
        predictions = self.predict(X)
        for i, pred in enumerate(predictions):
            c_matrix[self.label_to_index[y[i]]][self.label_to_index[pred]] += 1
        return c_matrix
        '''return sklearn.metrics.confusion_matrix(y, predictions)
        for i, pred in enumerate(predictions):
            if (pred == y[i]) and (pred == -1):
                c_matrix[0][0] += 1
            elif (pred == y[i]) and (pred == 1):
                c_matrix[1][1] += 1
            elif (pred != y[i]) and (pred == -1):
                c_matrix[1][0] += 1
            elif (pred != y[i]) and (pred == 1):
                c_matrix[0][1] += 1
        return c_matrix'''

    def accuracy(self, X, y):
        """
        Return the accuracy of the classifier on the data (X_test, y_test)
        :param X: np.array of shape (m, number_features)
        :param y: np.array of shape (m,)
        :return: accuracy score [float in (0,1)]
        """
        # Workspace 1.4
        correct = 0
        size = len(X)
        c_matrix = self.confusion_matrix(X, y)
        for i in range(len(c_matrix)):
            correct += c_matrix[i][i]
        score = correct / size
        return score
```

In [38]:
```
# Test cell, uncomment to run the tests
tests.testKNN(KNNClassifier)
```

```
Question 1.1: [PASS]
Question 1.2: [PASS]
Question 1.3: [PASS]
Question 1.4: [PASS]
```

But professor, this code isn't complete!

1.1 [10 points] Complete the `majority_vote` function to determine the majority class of a series of neighbors. If there is a tie, then you should remove the farthest element until the tie is broken. (Avg **9.6**)

1.2 [5 points] Complete the `predict` function to capture the predicted class of a new datapoint (Avg **4.9**)

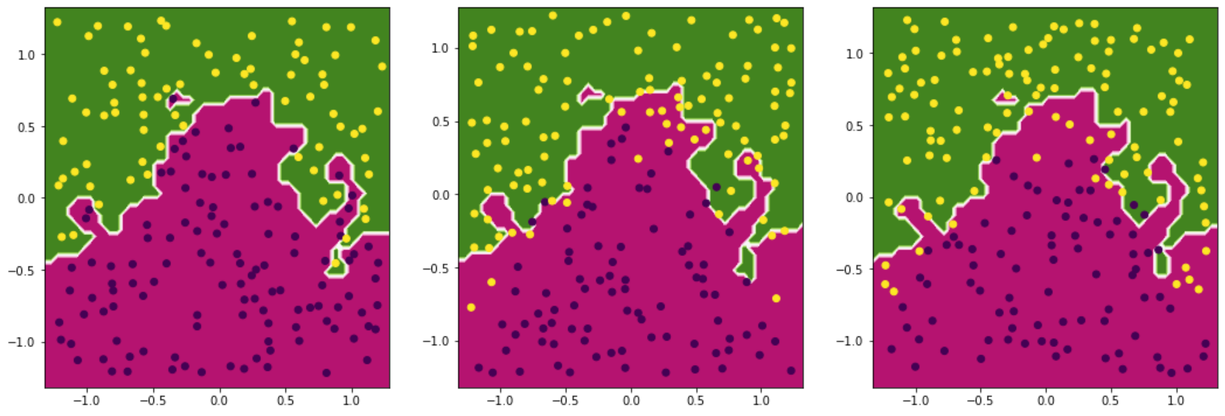
- HINT: Use the BallTree documentation to determine how to retrieve neighbors from the model (https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.BallTree.html#sklearn.neighbors.BallTree (https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.BallTree.html#sklearn.neighbors.BallTree

1.3 [5 points] Complete the `confusion_matrix` function to reveal the results of classification (Avg **5**)

1.4 [5 points] Complete the `accuracy` function to get accuracy of the classifier based on a given test data (Avg **5**)

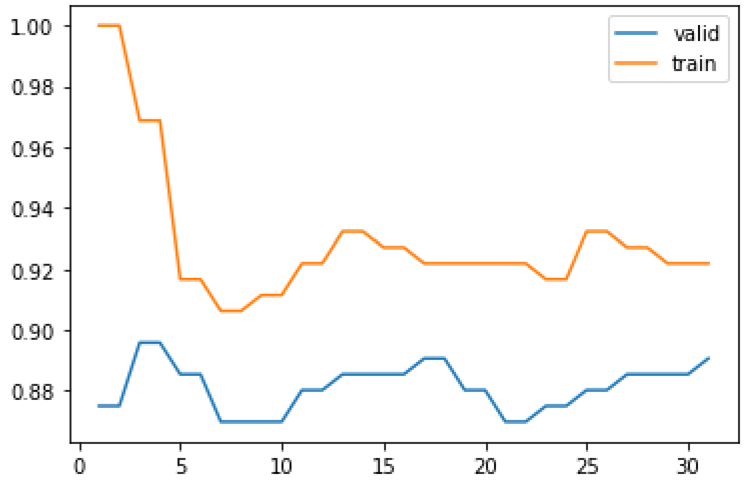
Below, we'll be using our KNNClassifier (sent in as "model") to show how we would predict any points in space given the input data.

In [39]:
```
# This cell is to show the decision surface of the classifier
# You can change k to visualize KNN behavior
knn = KNNClassifier(2).fit(binary_data.X_train, binary_data.y_train)
fig, axs = plt.subplots(1, 3)
fig.set_figheight(6), fig.set_figwidth(18)
tests.show_decision_surface(knn, binary_data.X_train, binary_data.y_train, axs[0]
tests.show_decision_surface(knn, binary_data.X_valid, binary_data.y_valid, axs[1]
tests.show_decision_surface(knn, binary_data.X_test, binary_data.y_test, axs[2])
plt.show()
```



1.5 [10 points] For each k in the range [1,32], fit a KNNClassifier on the training set and plot the accuracies on training and validation data versus k. What's the value of k that yields the best accuracy on the training set? on the validation set? Which one should we choose? (Avg **9.8**)

In [40]:
```
# Workspace 1.5.a
#TODO: Try different Ks
ks = list(range(1, 32))
accuracies_train = []
accuracies_valid = []
#print(binary_data.X_train, binary_data.y_train)
for k in ks:
    knnTrain = KNNClassifier(k).fit(binary_data.X_train, binary_data.y_train)
    accuracies_train.append(knnTrain.accuracy(binary_data.X_train, binary_data.y_
    accuracies_valid.append(knnTrain.accuracy(binary_data.X_valid, binary_data.y_

plt.plot(ks, accuracies_valid, label="valid")
plt.plot(ks, accuracies_train, label="train")
plt.legend()
plt.show()
```



In [41]:
```
print("Best k for test set: " + str(accuracies_train.index(max(accuracies_train))
print("Best k for validation set: " + str(accuracies_valid.index(max(accuracies_v
```

```
Best k for test set: 1
Best k for validation set: 3
```

# Workspace 1.5.b

% Write up: best k for training and validation and which one should we choose

The best k for the test set is k=1 and the best k for the validation set is k=3. We should use the k=3 given by the validation set because the validation set is the one used primarily to tune fine parameters such as k. We would not want to use k=1 from the training set because the the training set was used to define the K-nearest neighbors fit and would be heavily biased.

1.6 [5 points] Report the accuracy and the confusion matrix on the test set using the value of k chosen in 1.5 (Avg **5**)

In [42]:
```python
# Workspace 1.6
# TODO: compute and print the accuracy on the test set using k from 1.5
#BEGIN
# code here
best_k = 3

knn = KNNClassifier(best_k).fit(binary_data.X_train, binary_data.y_train)
print("Accuracy for k=3 on test set: " + str(knn.accuracy(binary_data.X_test, bi
print("Confusion matrix: ")
print(knn.confusion_matrix(binary_data.X_test, binary_data.y_test))
```

```
Accuracy for k=3 on test set: 0.890625
Confusion matrix:
[[81  2]
 [19 90]]
```

Bonus (for the avid machine learner) (4 Points)

1.7.a [2 point] A **consistent classifier** (https://proceedings.neurips.cc/paper/1996/file/7bb060764a818184ebb1cc0d43d382aa-Paper.pdf) on the training data is defined as a classifier that reaches 100% accuracy on the training set. For which values of k is KNNClassifier Consistent? (Avg **1**)

1.7.b [2 points] Edit your `KNNClassifier` so that it's consistent for all k (we made sure that the change does not affect the sanity checks) (Avg **1**)

Write-up for the bonus

Workspace 1.7.a

a) The KNNClassifier fits this definition of consistent for both the values of k=1 and k=2. This can be seen in the plot for 1.5.

OK - now we've demonstrated that our KNN classifier works, let's think about our problem space!

Our Dataset - Identifying Digits from Images

It's a pretty common problem - just imagine working at the post office, and you're handed a hand-written check, and you have to identify exactly what it says. Did they pay 500 or 600 dollars? Is the letter going to 80309 (campus) or 30309 (Atlanta)?

Problem 2: Improving KNN on Digits dataset - 25 Points

2.1 [7 points] `report` the number of examples different partitions of the digit dataset adn the number of pixels in the images (Avg **6.8**)

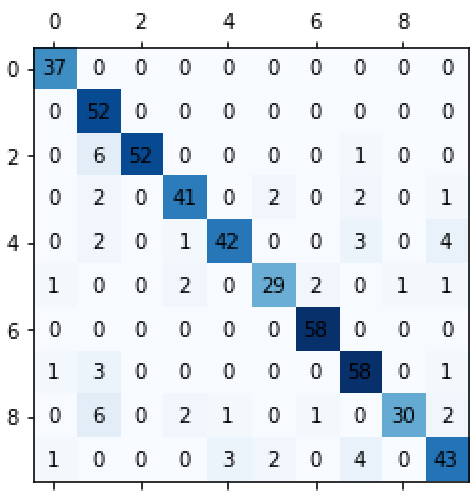
2.2 [8 points] complete the `evaluate` to perform the same evaluation we did in 1.5:

- For k in range (1, 20):
    - initialize the classifier for k and train in on the training set
    - Compute the accuracy on the validation set and save it
- Choose k with the best accuracy on the validation set
- Report the accuracy and the confusion matrix on the test set (use `display_confusion` for a cleaner output)

In [43]:
```
class Numbers:
    def __init__(self):
        self.data = data.DigitData() # it has the same structure as binary_data

    def report(self):
        """
        Report information about the dataset using the print() function
        """
        # Workspace 2.1
        #TODO: Create print-outs for reporting the size of each set and the size
        print("Total size of the training dataset is: " + str(len(self.data.X_tra
        print("Total size of the validation dataset is: " + str(len(self.data.X_v
        print("Total size of the test dataset is: " + str(len(self.data.X_test))

        print("Total pixel count of each image is: " + str(len(self.data.X_train[

    def evaluate(self, classifier_class):
        """
        valuates instances of the classifier class for different values of k and
        :param classifier_class: Classifier class (either KNNClassifier or Weight
        """

        # Workspace 2.2

        ks = list(range(1, 20))
        accuracies_valid = []
        #BEGIN
        # code here (anything between BEGIN and END is yours to edit if needed)
        best_valid_k = None
        confusion_matrix = None
        accuracy = 0
        ks = list(range(1, 20))
        accuracies_valid = []
        for k in ks:
            knn = classifier_class(k).fit(self.data.X_train, self.data.y_train)
            accuracies_valid.append(knn.accuracy(self.data.X_valid, self.data.y_v
        best_valid_k = accuracies_valid.index(max(accuracies_valid))+1
        best_classifier = classifier_class(best_valid_k).fit(self.data.X_train, s
        accuracy = best_classifier.accuracy(self.data.X_test, self.data.y_test)
        confusion_matrix = best_classifier.confusion_matrix(self.data.X_test, sel
        #END
        print("best k:", best_valid_k)
        print("Accuracy on test set:", accuracy)
        self.display_confusion(confusion_matrix)

    def view_digit(self, index, partition):
        """
        Display a digit given its index and partition
        :param index: index of the digit image
        :param partition: partition from which the digit is retrieved, either "tr
        """
        image = {"train": self.data.X_train, "valid": self.data.X_valid, "test":
        label = {"train": self.data.y_train, "valid": self.data.y_valid, "test":
        image = image.reshape(28, 28)
        plt.figure()
        plt.matshow(image)
        plt.title("Digit %i" % label)
        plt.show()

    @staticmethod
    def display_confusion(c_matrix):
        """
        Displays the confusion matrix using matshow
        :param c_matrix: square confusion matrix, shape (num_classes, num_classes
        """
        _, ax = plt.subplots()
        ax.matshow(c_matrix, cmap=plt.cm.Blues)
        for i in range(c_matrix.shape[0]):
            for j in range(c_matrix.shape[0]):
                ax.text(i, j, str(c_matrix[j, i]), va='center', ha='center')
        plt.show()
```

In [44]:
```
numbers = Numbers()
numbers.report()
numbers.evaluate(KNNClassifier)
```

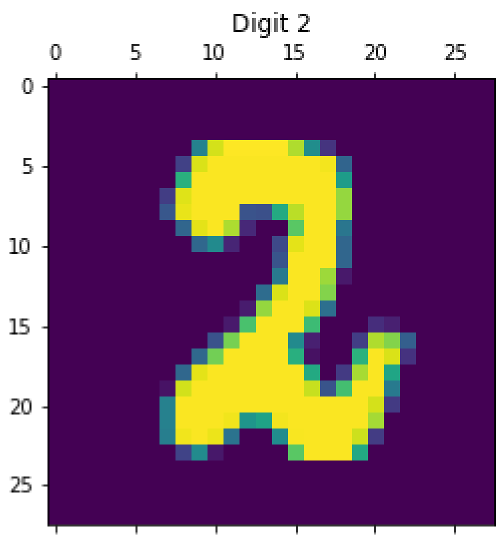
```
Total size of the training dataset is: 1000
Total size of the validation dataset is: 500
Total size of the test dataset is: 500
Total pixel count of each image is: 784 (28^2)
best k: 4
Accuracy on test set: 0.884
```



2.3 [10 points] Determine which classes are most often confused (from our confusion matrix above), inspect some examples of these digits (using the `view_digit` function in our Numbers class), and write a brief (4 - 5 sentences) description of why you think these particular numbers may be misclassified. (Avg **9.1**)

```
In [45]: # Workspace 2.3.a
         #TODO: Print out problem class images
         for i in range(50):
             numbers.view_digit(i,"test")
```

```
<Figure size 432x288 with 0 Axes>
```



```
<Figure size 432x288 with 0 Axes>
```

Workspace 2.3.b

For the most part, our KNN Classifier succeeds in classifying these hand-written numbers and this can be seen in both the accuracy of 88.4% and the diagonality of the confusion matrix. From the confusion matrix, we can see that the most common predictions that the algorithm makes that are wrong are that the number is a 1, a 9, or a 7 for this test data. This makes some sense if we look at the above examples. Many of the numbers exhibit the same slanted line that characterizes some of the 1s such as in 2 and the "second-half" of 8. 9 and 4 appear to get confused often which is unsurpirsing considering they may have similar structure depending on a person's handwriting style (closed or open top of 4). No 0 nor 1 from the test data was misidentified which suggests that these two contain similar structure between different writers and even though 1s may be slanted, there were enough slanted in the training to be able to distinguish these.

Problem 3 - Naive Bayes [35 points]

Consider the problem of predicting whether a person has a college degree based on age, salary, and Colorado residency. The dataset looks like the following.

Age	Salary	Colorado Resident	Has Siblings	College degree
37	44,000	Yes	No	Yes
61	52,000	Yes	No	No
23	44,000	No	No	Yes
39	38,000	No	Yes	Yes
48	49,000	No	No	Yes

Age	Salary	Colorado Resident	Has Siblings	College degree
57	92,000	No	Yes	No
38	41,000	No	Yes	Yes
27	35,000	Yes	No	No
23	26,000	Yes	No	No
38	45,000	No	No	No
32	50,000	No	No	No
25	52,000	Yes	No	No

In [46]:
```
features = np.array([
    [37, 44000, 1, 0],
    [61, 52000, 1, 0],
    [23, 44000, 0, 0],
    [39, 38000, 0, 1],
    [48, 49000, 0, 0],
    [57, 92000, 0, 1],
    [38, 41000, 0, 1],
    [27, 35000, 1, 0],
    [23, 26000, 1, 0],
    [38, 45000, 0, 0],
    [32, 50000, 0, 0],
    [25, 52000, 1, 0]
])
labels = np.array([1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1])
```

3.1 [5 points] Complete `threshold_features` to convert age and salary features to binary ones using the threshold arguments. (Avg **4.9**)

In [47]:
```
def threshold_features(features, age_threshold, salary_threshold):
    """
    Transform age and salary to binary
    :param features: data array of shape (m, n_features) where features[:,0] for
    :param age_threshold: used to "binarize" the data, 1 if age > age_threshold a
    :param salary_threshold: used to "binarize" the data, 1 if salary > salary_th
    :return: binary features matrix
    """
    binary_features = features * 1  #This row just creates a "hard copy" of the X

    for data in binary_features:
        data[0] = 1 if (data[0] > age_threshold) else 0
        data[1] = 1 if (data[1] > salary_threshold) else 0

    return binary_features
```

In [48]:
```
# Test cell, uncomment to run the tests
tests.test_threshold(threshold_features)
```

```
Question 3.1: [PASS]
```

3.2 [3 points] If we were to use only one binary feature (age >40, salary > 40000, colorado resident, has siblings), then what's the highest accuracy we could achieve? Which feature should we use?

```
In [49]: accuracy = [0,0,0,0]

         for i in range(len(features)):
             if (features[i][0] > 40 and labels[i] == 1):
                 accuracy[0] += 1
             if (features[i][1] > 40000 and labels[i] == 1):
                 accuracy[1] += 1
             if (features[i][2] == 1 and labels[i] == 1):
                 accuracy[2] += 1
             if (features[i][3] == 1 and labels[i] == 1):
                 accuracy[3] += 1

         for i in range(4):
             accuracy[i] = accuracy[i]/12
         print(accuracy)
```

```
[0.08333333333333333, 0.5, 0.16666666666666666, 0.16666666666666666]
```

Workspace 3.2

From the above code block, we can see that if we use the binary features age>40, salary>40000, is a colorado resident, and has siblings, we get accuracies of 0.083, 0.5, 0.167, and 0.167 respectively. This tells us that we should use salary as a feature as it has the highest accuracy in predicting whether someone has a college degree at 0.5.

How to implement NaiveBayes

As seen during the class, given a row (x_1, x_2, x_3), the naive Bayes classifier should assign the label y that maximizes:

$$\log[p(y)\prod_i p(x_i|y)] = \log p(y) + \sum_i \log p(x_i|y)$$

$p(y)$ and $p(x_i|y)$ are computed using the training set (during `fit` call).

For this, we need two attributes to store $\log p(y)$ and $\log p(x_i|y)$ for different features i.

Let's assume we're working with binary classes $\{0, 1\}$ and all features have discrete supports. Then we will store `classes_log_probability` as an array of shape `(2,)` that contains:

$$\left[\log p(y=0), \log p(y=1)\right]$$

.

If feature i has 2 possible values $\{0, 1\}$, then $\log p(x_i|y)$ would be stored as a 2×2 matrix:

$$A_i = \begin{bmatrix} \log p(x_i = 0|y=0) & \log p(x_i = 1|y=0) \\ \log p(x_i = 0|y=1) & \log p(x_i = 1|y=1) \end{bmatrix}$$

`features_log_likelihood` should then store such matrix for each feature.

We have defined $p(x_i | y)$ as :

$$p(x_i | y) = \frac{N_{y,x_i}}{N_y}$$

where $N_{y,i}$ is the number of rows where y and x_i occur together, and $N_y = \sum_i N_{y,x_i}$.

3.3 [5 points] Complete the method `compute_classes` and store the log prior in `classes_log_probability` (Avg **5**)

3.4 [7 points] Complete the method `compute_features` by storing the matrices A_i in `self.features_log_likelihood` (Avg **6.3**)

3.5 [5 points] Complete the method `join_log_likelihood` that computes the likelihood quantities $[\sum_i \log p(x_i | y = 0), \sum_i \log p(x_i | y = 1)]$ for each observation

3.6 [5 points] Complete the `predict` method

```
In [50]: class NaiveBayes(object):
             """
             NaiveBayes classifier for binary features and binary labels
             """

             def __init__(self, alpha=0.0):
                 self.alpha = alpha
                 self.classes_counts = None
                 self.classes_log_probability = np.empty((2,))
                 self.features_log_likelihood = []  # list of arrays where element i store

             def compute_classes(self, y):
                 """
                 Computes the log prior of binary classes and stores the result in self.cl
                 :param y: binary labels array, shape (m,)
                 """
                 # Workspace 3.3
                 # use np.unique as it sorts classes
                 vals, counts = np.unique(y, return_counts=True)
                 for i in range(len(vals)):
                     self.classes_log_probability[i] = np.log(counts[i]/len(y))

             def compute_features(self, X, y):
                 """
                 Computes the log likelihood matrices for different features and stores th
                 :param X: data matrix with binary features, shape (n_samples, n_features)
                 :param y: binary labels array, shape (n_samples,)
                 """
                 # Workspace 3.4
                 num_features = len(X[0])
                 #print(X)
                 for i in range(num_features):
                     vals = [row[i] for row in X]
                     counts = sklearn.metrics.confusion_matrix(y, vals).astype(float)
                     for j in range(2):
                         size = np.sum(counts[j])
                         for k in range(2):
                             counts[j][k] = np.log(counts[j][k] / size)
                     self.features_log_likelihood.append(counts)
                 #print(self.features_log_likelihood)

             def fit(self, X, y):
                 """
                 :param X: binary np.array of shape (n_samples, n_features) [values 0 or 1
                 :param y: corresponding binary labels of shape (n_samples,) [values 0 or
                 :return: Classifier
                 """
                 self.compute_classes(y)
                 self.compute_features(X, y)
                 return self

             def joint_log_likelihood(self, X):
                 """
                 Computes the joint log likelihood
                 :param X: binary np.array of shape (n_samples, n_features) [values 0 or 1
                 :return: joint log likelihood array jll of shape (n_samples, 2), where jl
```

```
        """
        # Workspace 3.5
        joint_log_likelihood = np.zeros((X.shape[0], 2))
        #print(joint_log_likelihood)
        for ob in range(len(X)):
            print(X[ob])
            for i in range(len(X[ob])):
                for y in range(2):
                    #print("on:  " + str(ob) + " i:  " + str(i) + " y:  "+ str(y))
                    joint_log_likelihood[ob][y] += self.features_log_likelihood[i

        return joint_log_likelihood

    def predict(self, X):
        """
        :param X:
        :return:
        """

        # Workspace 3.6
        # TODO: Find the corresponding labels using Naive bayes logic
        #BEGIN
        joint_log_likelihood = self.joint_log_likelihood(X)
        #print(joint_log_likelihood)
        #print(self.classes_log_probability)
        for i in range(len(joint_log_likelihood)):
            joint_log_likelihood[i][0] += self.classes_log_probability[0]
            joint_log_likelihood[i][1] += self.classes_log_probability[1]
        y_hat = np.zeros((X.shape[0],))
        for i in range(len(y_hat)):
            y_hat[i] = np.argmax(joint_log_likelihood[i])
        #END
        return y_hat
```

In [51]:
```
# Test cell, uncomment to run the tests
tests.test_NB(NaiveBayes)
```

```
Question 3.3: [PASS]
Question 3.4: [PASS]
[0 1 0 1]
[0 1 0 0]
[1 0 1 0]
[0 0 0 1]
[1 1 1 1]
Question 3.5: [PASS]
[0 1 0 1]
[0 1 0 0]
[1 0 1 0]
[0 0 0 1]
[1 1 1 1]
Question 3.6: [PASS]
```

3.7 [5 points] Using age 40 and salary 40,000 as thresholds, transform the features and evaluate (accuracy) the NaiveBayes classifier on the training data. Does it outperform our baseline (of using one feature)? (Avg **4.5**)

In [52]:
```
naive_bayes = NaiveBayes()
# Workspace 3.7
#TODO: Transform features to binary features, fit the classifier, report the accu

feat = threshold_features(features, 40, 40000)
#print(feat)
nb = NaiveBayes().fit(feat,labels)
res = nb.predict(feat)
correct = 0
for i in range(len(res)):
    if res[i] == labels[i]:
        correct += 1
print("Accuracy: " + str(correct/len(res)))
```

```
[0 1 1 0]
[1 1 1 0]
[0 1 0 0]
[0 0 0 1]
[1 1 0 0]
[1 1 0 1]
[0 1 0 1]
[0 0 1 0]
[0 0 1 0]
[0 1 0 0]
[0 1 0 0]
[0 1 1 0]
Accuracy: 0.8333333333333334
```

From this we can see that our accuracy is substantially improved upon the baseline of using just one feature (salary) in predicting whether a person attended college.

Bonus

3.8 [2 points] Use the attribute `alpha` of the NaiveBayes to convert it to smoothed NaiveBayes presented during the class. `alpha` defaults to 0, so editing the class should not affect NaiveBayes tests (Avg **1.8**)