Homework 1

STAT 451: Machine Learning (Fall 2020)
Instructor: Sebastian Raschka (sraschka@wisc.edu)

Course website: http://pages.stat.wisc.edu/~sraschka/teaching/stat451-fs2020/ (http://pages.stat.wisc.edu/~sraschka/teaching/stat451-fs2020/)

Due: Oct 05, (before 3:00 pm).

How to submit

You need to send the `.ipynb` file with your answers plus an `.html` file, which will serve as a backup for us in case the `.ipynb` file cannot be opened on my or the TA's computer. In addition, you may also export the notebook as PDF and upload it as well.

The homework solution should be uploaded on Canvas. You can submit it as often as you like before the deadline.

Important

- Please make sure that you provide an answer in each cell and place that contains the **[your answer]** tags.
- The places that require your code answer are marked with `"# YOUR CODE"` comments.
- Note that you may use 1 or more line of code for replacing each `"# YOUR CODE"` comment.

For example, imagine there is a question asking you to implement a threshold function that should return 1 if the input `x` is greater than 0.5 and otherwise. This could appear as follows in the the exercise:

```
def threshold_func(x):
    # YOUR CODE
```

A valid answer could be

```
def threshold_func(x):
    if x > 0.5:
        return 1
    else:
        return 0
```

Another valid solution could be

```
def threshold_func(x):
    return int(x > 0.5)
```

```
In [1]: %load_ext watermark
        %watermark  -d -u -a 'Roshan' -v -p numpy,scipy,matplotlib,sklearn
```

```
Roshan
last updated: 2020-10-05

CPython 3.8.3
IPython 7.16.1

numpy 1.18.5
scipy 1.5.0
matplotlib 3.2.2
sklearn 0.23.1
```

The watermark package that is being used in the next code cell provides a helper function of the same name, `%watermark` for showing information about your computational environment. This is useful for keeping track of what software versions are/were being used. If you encounter issues with the code, please make sure that your software package have the same version as the the ones shown in the pre-executed watermark cell.

Before you execute the watermark cell, you need to install watermark first. If you have not done this yet. To install the watermark package, simply run

```
!pip install watermark
```

or

```
!conda install watermark -c conda-forge
```

in the a new code cell. Alternatively, you can run either of the two commands (the latter only if you have installed Anaconda or Miniconda) in your command line terminal (e.g., a Linux shell, the Terminal app on macOS, or Cygwin, Putty, etc. on Windows).

For more information installing Python, please refer to the previous lectures and ask the TA for help.

E 1) [2 pts]

Choose 2 machine learning application examples from the first lecture (see section 1.2 in the lecture notes) and answer the following questions:

- What is the overall goal?
- How would an appropriate dataset look like?
- Which general machine learning category (supervised, unsupervised, reinforcement learning) does this problem fit in?
- How would you evaluate the performance of your model (in very general, non technical terms)

Example -- Email Spam classification:

- **Goal.** A potential goal would be to learn how to classify emails as spam or non-spam.
- **Dataset.** The dataset is a set consisting of emails as text data and their spam and non-spam labels.
- **Category.** Since we are working with class labels (spam, non-spam), this is a supervised learning problem.
- **Measure Performance.** Predict class labels in the test dataset and count the number of correct predictions to asses the prediction accuracy.

Example 1: Face Detection

- **Goal.** A potential goal would be to learn one/a small set of face(s), and be able to recognize if a new face (in the example of unlocking a device with facial recognition) matches the set of learned/authorized faces.
- **Dataset.** Face data provided to the device when opting to use face unlock.
- **Category.** Since we are not working with class labels, this is an unsupervised learning problem.
- **Measure Performance.** Face detection accuracy and speed.

Example 2: Product Recommendations

- **Goal.** A potential goal would be to recommend (target) products to customers that they are likely to buy based on what they are purchasing (Amazon product recommendations).
- **Dataset.** Dataset that contains information on what items are frequently bought together.
- **Category.** There are no labels here (just observing trends in the data being gathered), so this is an unsupervised learning problem.
- **Measure Performance.** More sales.

E 2) [2 pts]

If you think about the task of training a machine learning classifier that detects skin cancer based on some features derived from images. After model training, you find that your classifier predicts skin cancer with 95% accuracy. Do you think that the classification accuracy is a good evaluation metric for judging how useful classifier is? What are some potential pitfalls for using this classifier on new patients? (Hint: think about false positives and false negatives).

[your answer] I don't think that classification accuracy is a good evaluation metric for this classifier's usefulness. In the medical field, I believe that these types of classifiers would be used to 'cast a wide net' and check for things that could be skin cancer, then a doctor would take over. For this reason, I belive that maximizing classification accuracy while minimizing false negatives would be best (or if there are plenty of doctors around to deal with false positives, just minimize false negatives). If the above model with 95% accuracy has most of it's error as false negative, that could possibly be harmful to patients.

E 3) [2 pts]

In the example E 2), cancer classification is a supervised machine learning problem. List 2 examples of unsupervised learning tasks that would fall into the category of clustering. In one or more sentences, explain why you would describe these examples as clustering tasks and not supervised learning tasks. Select examples that are not already that are in the "Lecture note list" from E 1).

[your answer] K-means clustering can be used in medical trials/studies to split the people participating in the trial/study into k groups based on how similar individuals are to each other. This is not supervised learning (is unsupervised clustering) because it may not be known what group each person should be in.

Clustering can also be used for document analysis. The contents of the documents is analyzed and the documents can be grouped with other documents with similar contents. This is unsupervised because the documents do not already have labels for categories they belong to.

E 4) [2 pts]

In the *k*-nearest neighbor (*k*-NN) algorithm, what computation happens at training and what computation happens at test time? Explain your answer in 1-2 sentences.

[your answer] The training phase for k-NN is simply to remember the training dataset. When testing the k-NN algorithm, we compute the classification for each item in the testing dataset based on the k-NN, and we compute the test set accuracy/error.

E 5) [2 pts]

Assume that you are using a *k*-NN classifier on an image dataset with has images with 200x200*3=120,000 features. Would you expect that this classifier would perform better or worse (in terms of prediction accuracy) when you reduce the number of features by downscaling the image (assuming the number of training examples is fixed)? Explain your reasoning.

[your answer] With a fixed number of training examples, reducing the number of features usually increases the classifier performance because the model will be less likely to overfit the training data. In order to map things in higher dimensions (more features), we need more training examples.

E 6) [2 pts]

If your dataset contains several noisy examples (or outliers), is it better to increase or decrease *k*? Explain your reasoning.

[your answer] If the dataset has several outliers, it is better to increase k. If k=1, and the new example we see is mapped closest to an outlier, that example will likely be misclassified. For k=7, that same new example will still be close to an outlier, by there are 6 other examples that the algorithm considers to classify the new example.

E 7) [2 pts]

Implement the Kronecker Delta function in Python,

$$\delta(i, j) = \begin{cases} 1, & \text{if } i = j, \\ 0, & \text{if } i \neq j. \end{cases}$$

The `assert` statements are here to help you: They will raise an `AssertionError` if your function returns unexpected results based on the test cases.

In [2]:
```
# This is an example implementing a Dirac Delta Function
# There is no need to edit anything here.

def dirac_delta(x):
    if x > 0.5:
        return 1
    else:
        return 0


assert dirac_delta(1) == 1
assert dirac_delta(2) == 1
assert dirac_delta(-1) == 0
assert dirac_delta(0.5) == 0
```

In [3]:
```
def kronecker_delta(i, j):
    # YOUR CODE BELOW
    if i == j:
        return 1
    else:
        return 0



# DO NOT EDIT THE LINES BELOW
assert kronecker_delta(1, 0) == 0
assert kronecker_delta(2, 2) == 1
assert kronecker_delta(-1, 1) == 0
assert kronecker_delta(0.5, 0.1) == 0
```

E 8) [2 pts]

Suppose `y_true` is a list that contains true class labels, and `y_pred` is an array with predicted class labels from some machine learning task. Calculate the prediction error **in percent** (**without** using any external libraries like NumPy or scikit-learn).

In [4]:
```
y_true = [1, 2, 0, 1, 1, 2, 3, 1, 2, 1]
y_pred = [1, 2, 1, 1, 1, 0, 3, 1, 2, 1]


correct = 0
total_elements = 0
for i, j in zip(y_true, y_pred):
    # YOUR CODE BELOW
    total_elements+=1
    if i == j:
        correct+=1

error = (correct + 0.0) / (total_elements + 0.0)    # YOUR CODE

print('Error: %.2f%%' % (error))
```

```
Error: 0.80%
```

E 9) [2 pts]

Import the NumPy library to create a 3x3 matrix with values ranging 10-18. The expected output should look as follows:

```
array([[10, 11, 12],
       [13, 14, 15],
       [16, 17, 18]])
```

In [5]:
```
import numpy as np

A = np.arange(10,19).reshape(3, 3)

print(A)
```

```
[[10 11 12]
 [13 14 15]
 [16 17 18]]
```

E 10) [2 pts]

Use create a 2x2 NumPy array with random values drawn from a uniform distribution using the random seed `123` and show the results below.

In [6]:
```
rng = np.random.RandomState(123)
# YOUR CODE
randArr = rng.uniform(size=(2,2))

print(randArr)
```

```
[[0.69646919 0.28613933]
 [0.22685145 0.55131477]]
```

E 11) [2 pts]

Given an array `A`,

```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16]])
```

use the NumPy slicing syntax to only select the 2x2 lower-right corner of this matrix.

In [7]:
```
A = np.array([
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 10, 11, 12],
    [13, 14, 15, 16]])

print(A[2:,2:])
```

```
[[11 12]
 [15 16]]
```

E 12) [2 pts]

Given the array `A` below, find the least frequent integer in that array:

```
In [8]: rng = np.random.RandomState(123)
        A = rng.randint(0, 10, 200)
```

```
In [9]: # YOUR CODE
        print(A)
        print(str(np.bincount(A).argmin()) + ' was the least frequent integer in that
         array')
```

```
[2 2 6 1 3 9 6 1 0 1 9 0 0 9 3 4 0 0 4 1 7 3 2 4 7 2 4 8 0 7 9 3 4 6 1 5 6
 2 1 8 3 5 0 2 6 2 4 4 6 3 0 6 4 7 6 7 1 5 7 9 2 4 8 1 2 1 1 3 5 9 0 8 1 6
 3 3 5 9 7 9 2 3 3 3 8 6 9 7 6 3 9 6 6 6 1 3 4 3 1 0 5 8 6 8 9 1 0 3 1 3 4
 7 6 1 4 3 3 7 6 8 6 4 4 7 0 0 9 8 8 4 8 6 1 6 8 7 9 1 7 1 7 9 8 7 1 3 1 8
 7 5 1 2 5 2 2 9 3 2 6 7 9 1 3 8 3 7 9 9 3 3 5 6 0 8 7 7 4 4 5 0 8 9 2 5 1
 5 9 2 4 3 0 3 7 7 2 5 1 7 5 9]
5 was the least frequent integer in that array
```

E 13) [2 pts]

Complete the line of code below to read in the `'train_data.txt'` dataset, which consists of 3 columns: 2 feature columns and 1 class label column. The columns are separated via commas and show the first 5 lines of the DataFrame.

```
In [10]: import pandas as pd

         df_train = pd.read_csv('train_data.txt')

         df_train.head()
```

Out[10]:

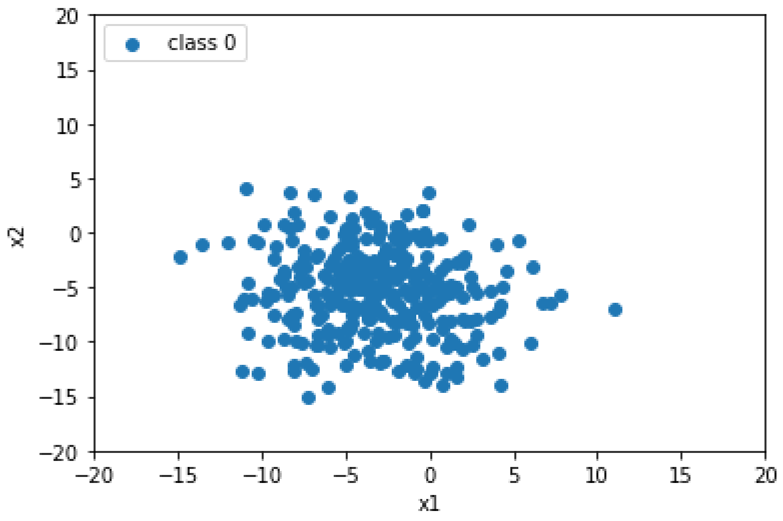
	x1	x2	y
0	-3.84	-4.40	0
1	16.36	6.54	1
2	-2.73	-5.13	0
3	4.83	7.22	1
4	3.66	-5.34	0

E 14) [2 pts]

Consider the following code below, which plots one the samples from class 0 in a 2D scatterplot using matplotlib:

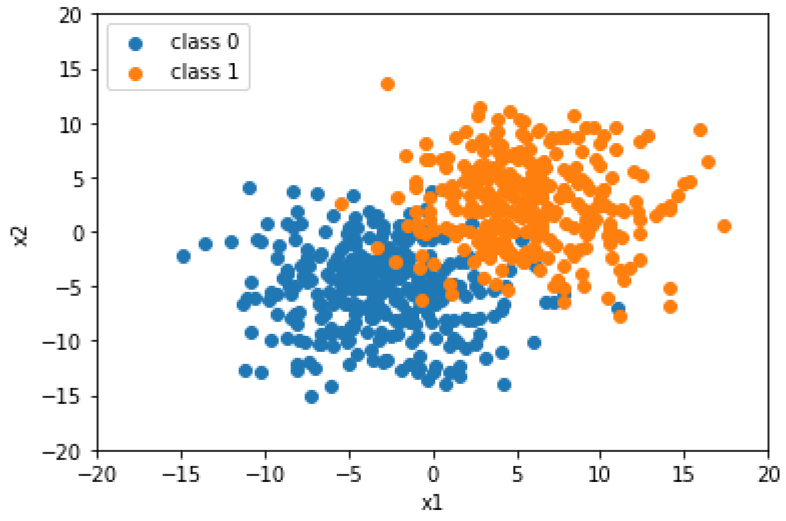
In [11]:
```
X_train = df_train[['x1', 'x2']].values
y_train = df_train['y'].values
```

In [12]:
```
%matplotlib inline
import matplotlib.pyplot as plt


plt.scatter(X_train[y_train == 0, 0],
            X_train[y_train == 0, 1],
            label='class 0',)

plt.xlabel('x1')
plt.ylabel('x2')
plt.xlim([-20, 20])
plt.ylim([-20, 20])
plt.legend(loc='upper left')
plt.show()
```



Now, the following code below is identical to the code in the previous code cell but contains partial code to also include the examples from the second class. Complete the second `plt.scatter` function to also plot the training examples from `class 1`.

In [13]:
```
plt.scatter(X_train[y_train == 0, 0],
            X_train[y_train == 0, 1],
            label='class 0',)

plt.scatter(X_train[y_train == 1, 0],
            X_train[y_train == 1, 1],
            label='class 1',)

plt.xlabel('x1')
plt.ylabel('x2')
plt.xlim([-20, 20])
plt.ylim([-20, 20])
plt.legend(loc='upper left')
plt.show()
```



E 15) [2 pts]

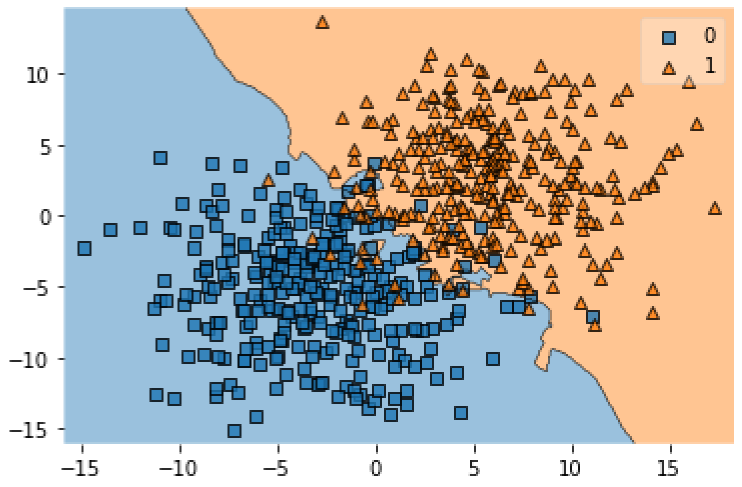
Consider the we trained a 3-nearest neighbor classifier using scikit-learn on the previous training dataset:

In [14]:
```
from sklearn.neighbors import KNeighborsClassifier


knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)
```

Out[14]:
```
KNeighborsClassifier(n_neighbors=3)
```

In [15]:
```python
from mlxtend.plotting import plot_decision_regions

plot_decision_regions(X_train, y_train, knn)
```

Out[15]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x20aa88f4760>
```



Compute *the number of* misclassifications of the 3-NN classifier on the training set. The number of errors should be a count, i.e., a positive integer.

In [16]:
```python
print('Errors:', (sum(knn.predict(X_train) != y_train)))
```

```
Errors: 33
```

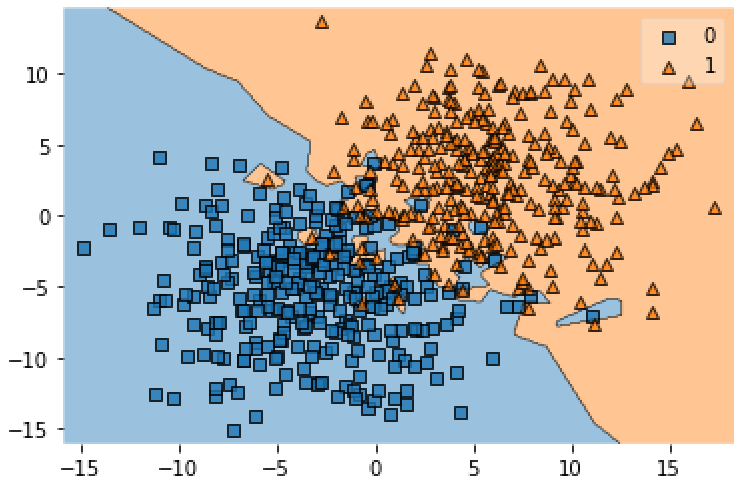
E 16) [2 pts]

Use the code from E 15) to

- also visualize the decision boundaries of *k*-nearest neighbor classifiers with k=1, k=5, k=7, k=9
- compute the prediction error on the training set for the *k*-nearest neighbor classifiers with k=1, k=5, k=7, k=9

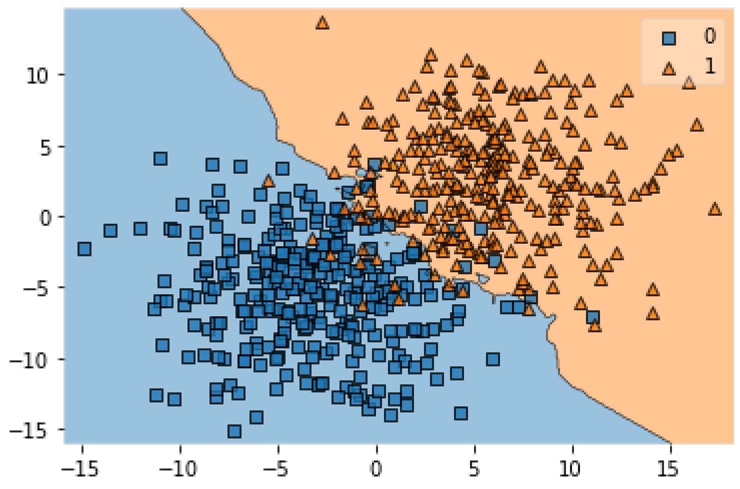
In [17]:
```
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train, y_train)
plot_decision_regions(X_train, y_train, knn)
print('Errors:', (sum(knn.predict(X_train) != y_train)))
```

```
Errors: 0
```



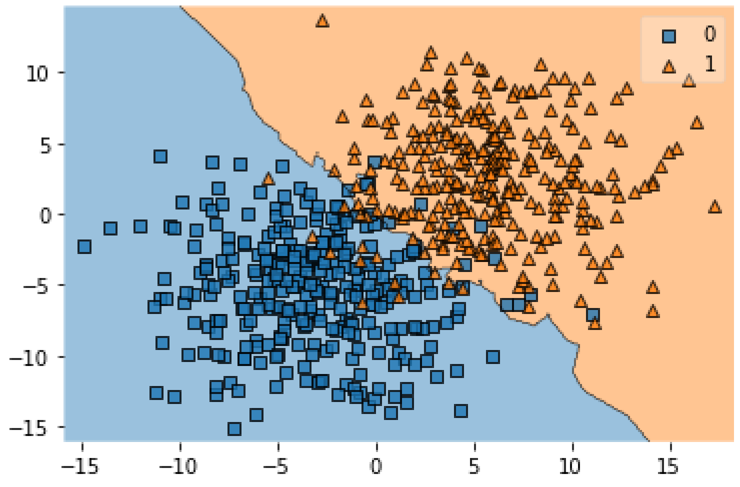
In [18]:
```
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)
plot_decision_regions(X_train, y_train, knn)
print('Errors:', (sum(knn.predict(X_train) != y_train)))
```

```
Errors: 32
```



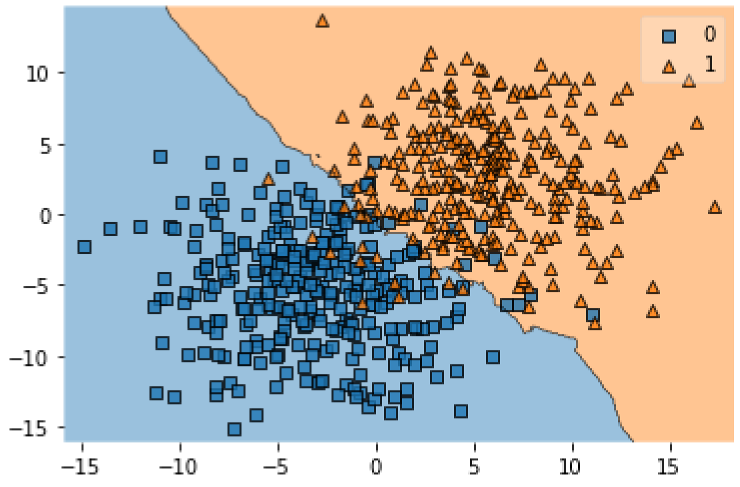
In [19]:
```
knn = KNeighborsClassifier(n_neighbors=7)
knn.fit(X_train, y_train)
plot_decision_regions(X_train, y_train, knn)
print('Errors:', (sum(knn.predict(X_train) != y_train)))
```

```
Errors: 29
```



In [20]:
```
knn = KNeighborsClassifier(n_neighbors=9)
knn.fit(X_train, y_train)
plot_decision_regions(X_train, y_train, knn)
print('Errors:', (sum(knn.predict(X_train) != y_train)))
```

```
Errors: 30
```



E 17) [2 pts]

Using a similar approach you used in E 13), now load the `test_data.txt` file into a pandas array. However, note that the dataset now has whitespaces separating the columns instead of commas.

In [21]:
```
df_test = pd.read_csv('test_data.txt', delim_whitespace=True)

df_test.head()
```

Out[21]:

	x1	x2	y
0	-5.75	-6.83	0
1	5.51	3.67	1
2	5.11	5.32	1
3	0.85	-4.11	0
4	-0.50	-0.45	1

In [22]:
```
X_test = df_test[['x1', 'x2']].values
y_test = df_test['y'].values
```

E 18) [2 pts]

Use the `train_test_split` function from scikit-learn to divide the training dataset further into a training subset and a validation set. The validation set should be 20% of the training dataset size, and the training subset should be 80% of the training dataset size.

For you reference, the `train_test_split` function is documented at http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html (http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html).

In [23]:
```
import math
from sklearn.model_selection import train_test_split
testing_size = math.floor(len(X_train)*.8)

X_train_sub, X_val, y_train_sub, y_val = train_test_split(X_train, y_train,
                                                          test_size = .2,
                                                          random_state = 123,
                                                          stratify=y_train)
```

E 19) [2 pts]

Write a for loop to evaluate different *k* nn models with k=1 to k=14. In particular, fit the `KNeighborsClassifier` on the training subset, then evaluate it on the training subset, validation subset, and test subset. Report the respective classification error or accuracy.

```
In [24]: for k in range(1, 15):
             knn = KNeighborsClassifier(n_neighbors=k)
             knn.fit(X_train_sub, y_train_sub)
             error_count = sum(knn.predict(X_val) != y_val)
             accuracy = (len(y_val)-error_count)/len(y_val)
             print("For k =", k, "the number of errors was:", error_count, "(accuracy:"
         , accuracy, ")")
```

```
For k = 1 the number of errors was: 14 (accuracy: 0.8833333333333333 )
For k = 2 the number of errors was: 12 (accuracy: 0.9 )
For k = 3 the number of errors was: 11 (accuracy: 0.9083333333333333 )
For k = 4 the number of errors was: 11 (accuracy: 0.9083333333333333 )
For k = 5 the number of errors was: 11 (accuracy: 0.9083333333333333 )
For k = 6 the number of errors was: 10 (accuracy: 0.9166666666666666 )
For k = 7 the number of errors was: 9 (accuracy: 0.925 )
For k = 8 the number of errors was: 8 (accuracy: 0.9333333333333333 )
For k = 9 the number of errors was: 9 (accuracy: 0.925 )
For k = 10 the number of errors was: 9 (accuracy: 0.925 )
For k = 11 the number of errors was: 9 (accuracy: 0.925 )
For k = 12 the number of errors was: 9 (accuracy: 0.925 )
For k = 13 the number of errors was: 9 (accuracy: 0.925 )
For k = 14 the number of errors was: 9 (accuracy: 0.925 )
```

E 20) [2 pts]

Consider the following code cell, where I implemented *k*-nearest neighbor classification algorithm following the the scikit-learn API

In [25]:
```
import numpy as np


class KNNClassifier(object):
    def __init__(self, k, dist_fn=None):
        self.k = k
        if dist_fn is None:
            self.dist_fn = self._euclidean_dist

    def _euclidean_dist(self, a, b):
        dist = 0.
        for ele_i, ele_j in zip(a, b):
            dist += ((ele_i - ele_j)**2)
        dist = dist**0.5
        return dist

    def _find_nearest(self, x):
        dist_idx_pairs = []
        for j in range(self.dataset_.shape[0]):
            d = self.dist_fn(x, self.dataset_[j])
            dist_idx_pairs.append((d, j))

        sorted_dist_idx_pairs = sorted(dist_idx_pairs)

        return sorted_dist_idx_pairs

    def fit(self, X, y):
        self.dataset_ = X.copy()
        self.labels_ = y.copy()
        self.possible_labels_ = np.unique(y)

    def predict(self, X):
        predictions = np.zeros(X.shape[0], dtype=int)
        for i in range(X.shape[0]):
            k_nearest = self._find_nearest(X[i])[:self.k]
            indices = [entry[1] for entry in k_nearest]
            k_labels = self.labels_[indices]
            counts = np.bincount(k_labels,
                                 minlength=self.possible_labels_.shape[0])
            pred_label = np.argmax(counts)
            predictions[i] = pred_label
        return predictions
```

In [26]:
```
five_test_inputs = X_train[:5]
five_test_labels = y_train[:5]

knn = KNNClassifier(k=1)
knn.fit(five_test_inputs, five_test_labels)
print('True labels:', five_test_labels)
print('Pred labels:', knn.predict(five_test_inputs))
```

```
True labels: [0 1 0 1 0]
Pred labels: [0 1 0 1 0]
```

Since this is a very simple implementation of *k*NN, it is relatively slow -- very slow compared to the scikit-learn implementation which uses data structures such as Ball-tree and KD-tree to find the nearest neighbors more efficiently, as discussed in the lecture.

While we won't implement advanced data structures in this class, there is already an obvious opportunity for improving the computational efficiency by replacing for-loops with vectorized NumPy code (as discussed in the lecture). In particular, consider the `_euclidean_dist` method in the `KNNClassifier` class above. Below, I have written is as a function (as opposed to a method), for simplicity:

In [27]:
```
def euclidean_dist(a, b):
    dist = 0.
    for ele_i, ele_j in zip(a, b):
        dist += ((ele_i - ele_j)**2)
    dist = dist**0.5
    return dist
```

Your task is now to benchmark this function using the `%timeit` magic command that we talked about in class using two random vectors, `a` and `b` as function inputs:

In [28]:
```
rng = np.random.RandomState(123)

a = rng.rand(100)
b = rng.rand(100)
print(euclidean_dist(a, b))
```

```
3.926547586774215
```

In [29]:
```
%timeit euclidean_dist(a, b)
```

```
111 µs ± 2.31 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

E 21) [2 pts]

Now, rewrite the Euclidean distance function from E 20) in NumPy using

- either using the `np.sqrt` and `np.sum` function
- or using the `np.linalg.norm` function

and benchmark it again using the `%timeit` magic command. Then, compare results with the results you got in E 20). Did you make the function faster? Yes or No? Explain why, in 1-2 sentences.

```
In [30]: def euclidean_dist(a, b):
             dist = 0.
             dist = np.sum((a - b)**2)
             dist = np.sqrt(dist)
             return dist
         print(euclidean_dist(a, b))
```

```
3.9265475867742152
```

```
In [31]: %timeit euclidean_dist(a, b)
```

```
10.2 µs ± 108 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

[Your answer] Yes, I made the function faster. This is because np.sqrt and np.sum use C code to efficiently perform computation.

E 22) [2 pts]

Another inefficient aspect of the `KNNClassifier` implementation is that it uses the sorted function to sort all values in the distance value array. Since we are only interested in the *k* nearest neighbors, sorting *all* neighbors is quite unnecessary.

Consider the array `c` :

```
In [32]: rng = np.random.RandomState(123)
         c = rng.rand(10000)
```

Call the sorted function to select the 3 smallest values in that array, we can do the following:

```
In [33]: sorted(c)[:3]
```

```
Out[33]: [6.783831227508141e-05, 8.188761366767494e-05, 0.00012010148897489997]
```

In the code cell below, use the `%timeit` magic command to benchmark the sorted command above:

```
In [34]: %timeit sorted(c)[:3]
```

```
4.88 ms ± 101 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

A more efficient way to select the *k* smallest values from an array is to use a priority queue, for example, implemented using a heap data structure. A convenient `nsmallest` function that does exactly that is available from Python's standard library:

```
In [35]: from heapq import nsmallest

         nsmallest(3, c)

Out[35]: [6.783831227508141e-05, 8.188761366767494e-05, 0.00012010148897489971]
```

In the code cell below, use the `%timeit` magic command to benchmark the `nsmallest` function:

```
In [36]: %timeit nsmallest(3, c)

         2.13 ms ± 77.9 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Summarize your findings in 1-3 sentences.

[Your Answer] It appears that using nsmallest is significantly faster than using sorted to select the 3 smallest values from the array. This makes sense because using a heap data structure for a priority queue will ensure that the smallest values are the fastest to access. Although, it appears that the functions below are even faster for this task than nsmallest (could be explained by the fact that it is more complex to insert things into a priority queue).

PS: There are many other options for performing the same task, for example, using NumPy:

```
In [37]: %timeit np.sort(c)[:3]

         643 µs ± 11.8 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

```
In [38]: %timeit c[np.argpartition(c, 3)[:3]]

         112 µs ± 1.75 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```