Problem Set 1

Due: AUGUST 27, before class (before 4:00 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 attach it to the email as well.

Please use the following subject header for sending in your homework, so that we can make sure that nothing gets lost:

Your id mentioned on offer letter: Your Name

.

Note No help will be provided bny instructor you have to do it on your own.

you will get certificates on basis its result

```
%reload_ext watermark
%watermark -d -u -a "Syed Junaid Jaffery" -v -p numpy,scipy,matplotlib,sklearn
```

```
Author: Syed Junaid Jaffery

Last updated: 2024-08-26

Python implementation: CPython
Python version       : 3.12.4
IPython version      : 8.26.0

numpy     : 1.26.4
scipy     : 1.14.0
matplotlib: 3.9.0
sklearn   : 1.5.1
```

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 usefult to keep track of what software versions are/were being used. If you should 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)

Pick 3 machine learning application examples from the first lecture (see section 1.2 in the lecture notes, shared in group 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.

E 1) ANSWER

Example -- Sports Predictions:

- **Goal.** The goal would be to predict the outcome of a sports event like a cricket match or a wrestling match. which team/competitor would win etc. This would be done using historical

data or other statistical data of each team/wrestler to find out who has a higher chance of winning

- **Dataset.** The dataset would have historical data of the teams and data about the players of each team and their past performance. If we are talking cricket specifically then we could also consider variables like the country the match is being held in or the weather conditions etc.
- **Category.** The dataset would consist of labeled data on which the model would be trained so this is Supervised Learning.
- **Measure Performance.** We would predict the data and compare those predicted results to the actual results and asses the prediction accuracy.

Example -- ATMs (reading cheques for instance):

- **Goal.** The goal would be to automatically process cheques, extract info from them like the amount of money or the date of the transaction etc.
- **Dataset.** The data would actually be images this time instead of just written data. there would be images of cheques and along with that we would have the accurate extracted data from those images.
- **Category.** Again, the data is labeled and the model is learning from that so this is also Supervised Learning
- **Measure Performance.** We are either correctly or incorrectly identifying the text on the cheques so this is also a classification task, so we can analyze how much of the text the model predicted correctly and how much it didn't and find out the accuracy from there.

Example -- Self-driving cars:

- **Goal.** The goal would be to make a car identify and then avoid obstacles and navigate its way to a specific destination
- **Dataset.** The data would consist of a lot of images, like pedestrians, street signs, sidewalks, and other cars. Regarding street signs, datasets like GTSRB, LISA, and BTSD are frequently used in traffic sign recognition systems. Those datasets contains images of traffic signs in varying weather conditions.
- **Category.** The goal is again classification, we want our system to correctly identify the objects and correctly identify the type of traffic sign (stop sign, parking sign, etc) so this is also Supervised Learning. Although the navigation of the vehicle, i.e: the action it takes depending on the object would involve reinforcement learning, I think.
- **Measure Performance.** Performance can be evaluated based on the prediction of the object as well as the navigation of the car, how safely it drives and does it follow the path prescribed perfectly or not. We can count and analyze the amount of specific turns and if they were correct or not and the action taken when identifying an object, like a stop sign or a traffic light etc.

E 2)

If you think about the task of spam classification more thoroughly, do you think that the classification accuracy or misclassification error is a good error metric of how good an email classifier is? What are potential pitfalls? (Hint: think about false positives [non-spam email classified as spam] and false negatives [spam email classified as non-spam]).

E 2) ANSWER

Accuracy can be sufficient in cases where we have a good dataset but if we have an imbalanced dataset where we have a disproportionate amount of spam mails compared to non spam then we could end up with false positives or false negatives, these two are the main pitfalls of using accuracy alone basically: False Positives: Non-spam emails wrongly classified as spam. This is problematic because important emails might be missed. False Negatives: Spam emails classified as non-spam. These can clutter the inbox with unwanted messages, reducing the system's reliability.

The issue is that accuracy is simple the ratio of the correct predictions to the total number of mails, as I said, this would suffice if the dataset wasn't imbalanced. This is where we would want to use other methods to evaluate classification tasks like precision, F1-score and recall.

E 3)

In the exercise example of E 1), email spam classification was listed as an example of 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).

E 3) ANSWER

So I have to provide two examples of clustering tasks using Unsupervised Learning that were not discussed in the lecture.

1. Networks and Traffic analysis: We can group traffic patterns in a network and group them together (grouping them so its clustering btw) and then, when a new pattern emerges that

doesn't fit in our groups, we can say with reasonable certainty that that is a breach/anomaly in the network. So, basically an anomaly detection system.

2. Classifying writing styles: We can group formally written documents like memorandums, reports, formal letters etc and then when a pattern is detected that doesn't fit in that group, we can flag that as an informal/casual writing style. This could be an application in technical writing labs where students are taught formal letter writing

E 4)

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.

E 4) ANSWER

Full disclaimer, haven't studied KNN in detail yet so I will be using AI and google for assistance in-order to understand it quickly.

In short, the KNN algorithm calculates the Euclidean distance between data points and then identifies the nearest data point (kth nearest neighbor). so, to answer the question:

1. There is no training part of the algorithm basically so little to no computation happens here, the data is just loaded.
2. This is the part where the euclidean distance comes in and the algorithm calculates the distance between the data points and finds the kth nearest neighbor to then make predictions.

E 5)

Does (*k*-NN) work better or worse if we add more information by adding more feature variables (assuming the number of training examples is fixed)? Explain your reasoning.

E 5) ANSWER

Generally, if we add more features without changing the dataset in any other way, KNN would work worse. This is because the distance between the data points would increase as the features increase, and since the distance increase, it becomes harder to find the kth nearest neighbors. However, if the features we add are relevant and they help us distinguish the labels/classes, then it would be better instead of worse. So, it basically depends on the quality of the feature that we add.

E 6)

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

E 6) ANSWER

If we have noisy data with incorrect labels and outliers in the data, then the smaller our K value is, the higher the impact of that neighbor, so if for instance K=1 and that 1 nearest neighbor is chosen as one of those outliers then that would significantly affect the model's performance. Choosing a larger number of K values would make the algorithm choose more nearest neighbor and hence the affect of the outliers would decrease on the overall model. So, increasing the K value makes more sense when we have unreliable, noisy data with outliers.

E 7)

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.

```
# This is an example implementing a Dirac Delta Function

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


def kronecker_delta(i, j):
    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)

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 accuracy in percent (without using any external libraries).

```
y_true = [1, 2, 0, 1, 1, 2, 3, 1, 2, 1]
y_pred = [1, 2, 1, 1, 1, 0, 3, 1, 2, 1]


# accuracy is the ratio of correct predictions to the total number of predictions:
total_predictions = len(y_pred)
correct_predictions = len([i for i in range(len(y_pred)) if y_pred[i] == y_true[i]])
accuracy = (correct_predictions / total_predictions) * 100

print('Accuracy: %.2f%%' % accuracy)
```

```
Accuracy: 80.00%
```

E 9)

Import the NumPy library to create a 3x3 matrix with values ranging 0-8. The expected output should look as follows:

```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

```
import numpy as np

x = np.arange(9).reshape(3, 3)

print(x)
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
```

E 10)

Use create a 2x2 NumPy array with random values drawn from a standard normal distribution using the random seed `123`:

If you are using the 123 random seed, the expected result should be:

```
array([[-1.0856306 ,  0.99734545],
       [ 0.2829785 , -1.50629471]])
```

```
np.random.seed(123)
x = np.random.randn(2, 2)
print(x)
```

```
[[-1.0856306   0.99734545]
 [ 0.2829785  -1.50629471]]
```

E 11)

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 center of this matrix, i.e., the subarray

```
array([[ 6,  7],
       [10, 11]]).
```

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

print(A[1:3, 1:3])
```

```
[[ 6  7]
 [10 11]]
```

E 12)

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

```
rng = np.random.RandomState(123)
A = rng.randint(0, 10, 200)

print(np.bincount(A).argmax())
```

```
3
```

E 13)

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 white spaces. If your implementation is correct, the last line should show a data array in below the code cell that has the following contents:

```
        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
```

```
import pandas as pd
```

```
df_train = pd.read_csv("train_data.txt", delimiter=' ', header=0)
```

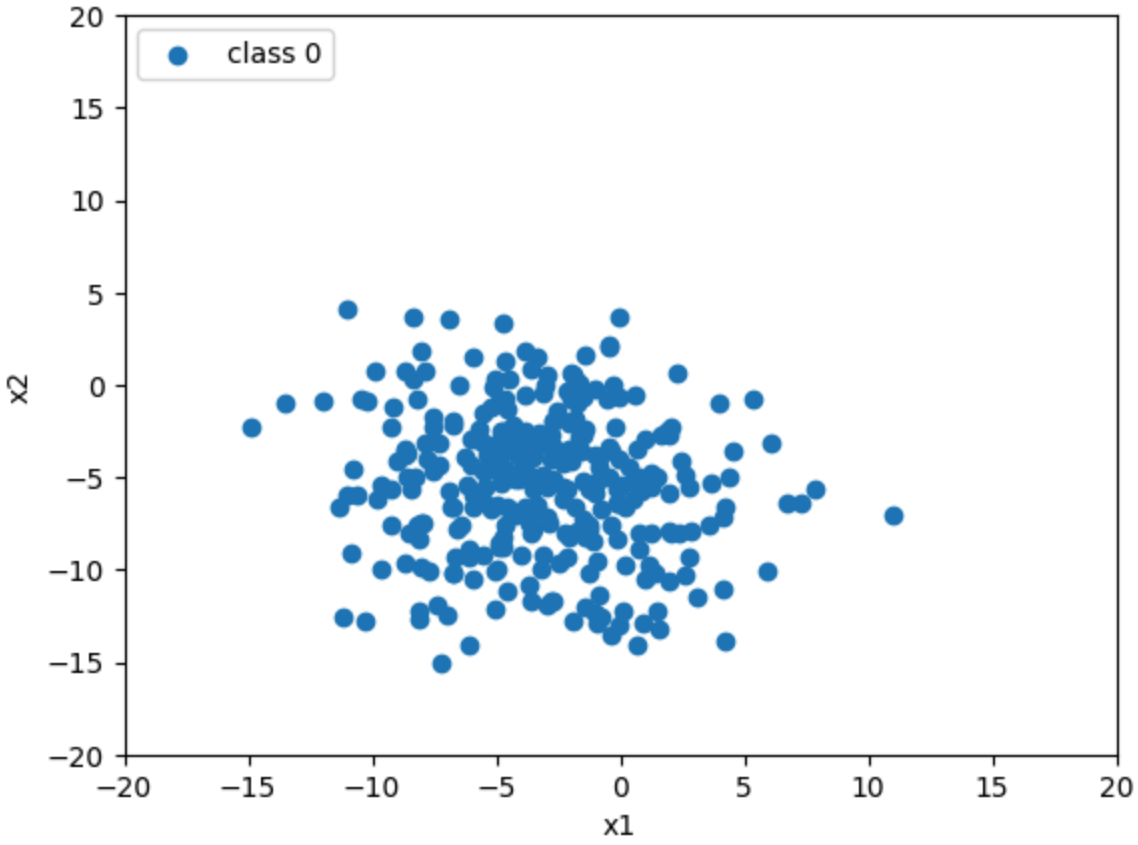
```
df_train.head()
```

	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)

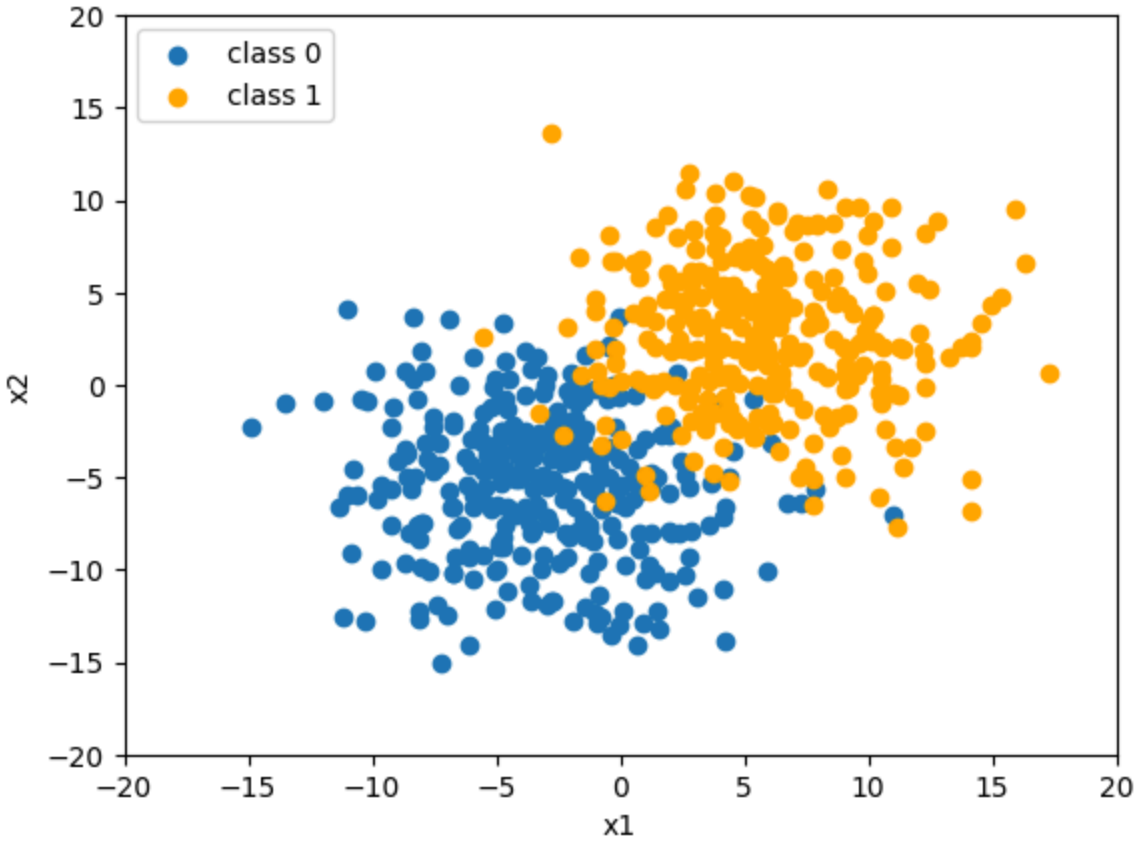
Consider the following code below, which plots one the samples from class 0 in a 2D scatterplot using matplotlib:

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


%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 trainign examples from `class 1`.

```
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', color='orar


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



E 15)

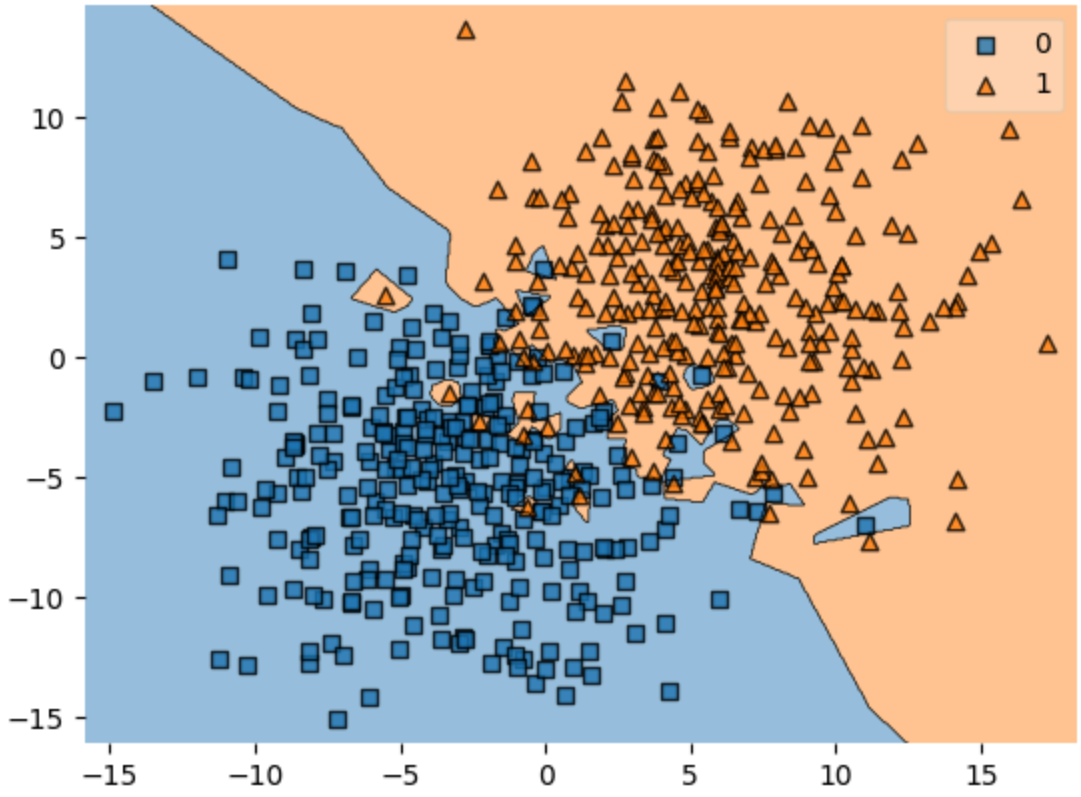
Consider the we trained a 1-nearest neighbor classifier using scikit-learn on the previous training dataset:

```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train, y_train)
```

```
KNeighborsClassifier
KNeighborsClassifier(n_neighbors=1)
```

```
from mlxtend.plotting import plot_decision_regions

plot_decision_regions(X_train, y_train, knn)
```

```
<Axes: >
```



Compute the misclassification error of the 1-NN classifier on the training set:

```
from sklearn.metrics import accuracy_score

y_pred = knn.predict(X_train)

error = 1 - accuracy_score(y_train, y_pred)
print(f'Misclassification Error: {error}')
```

```
Misclassification Error: 0.0
```

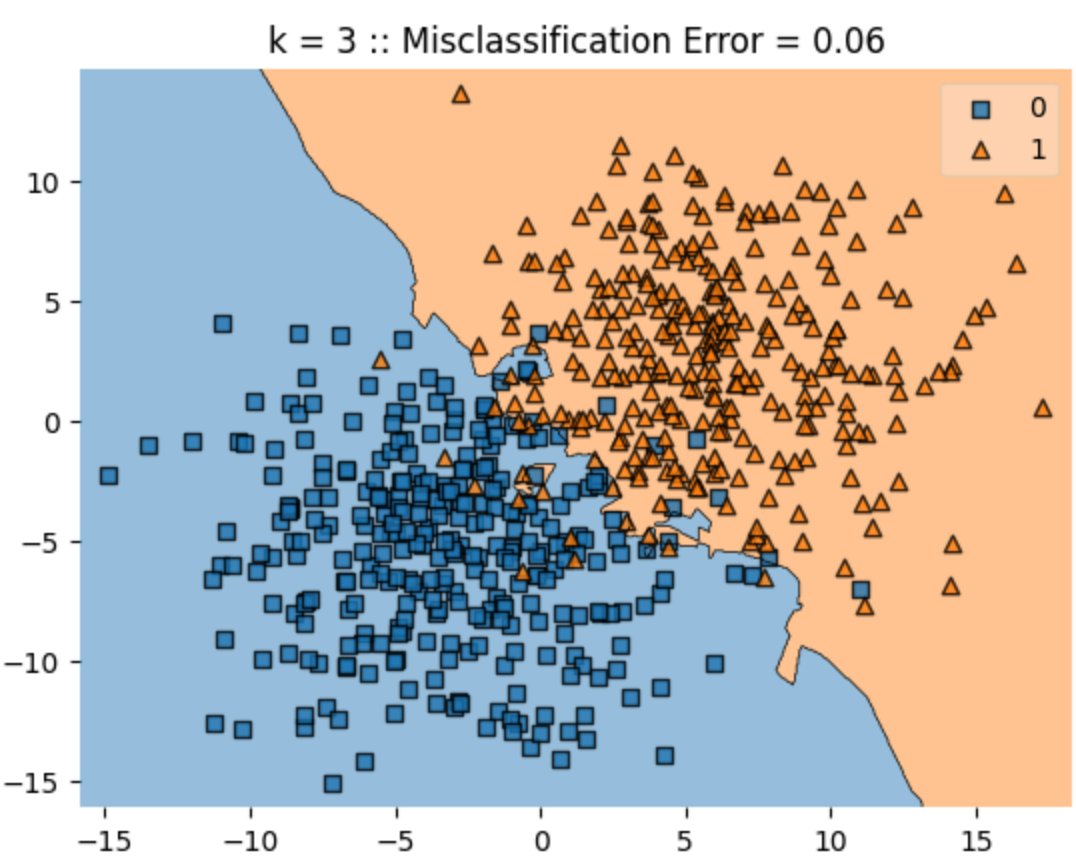
E 16)

Use the code from E 15) to

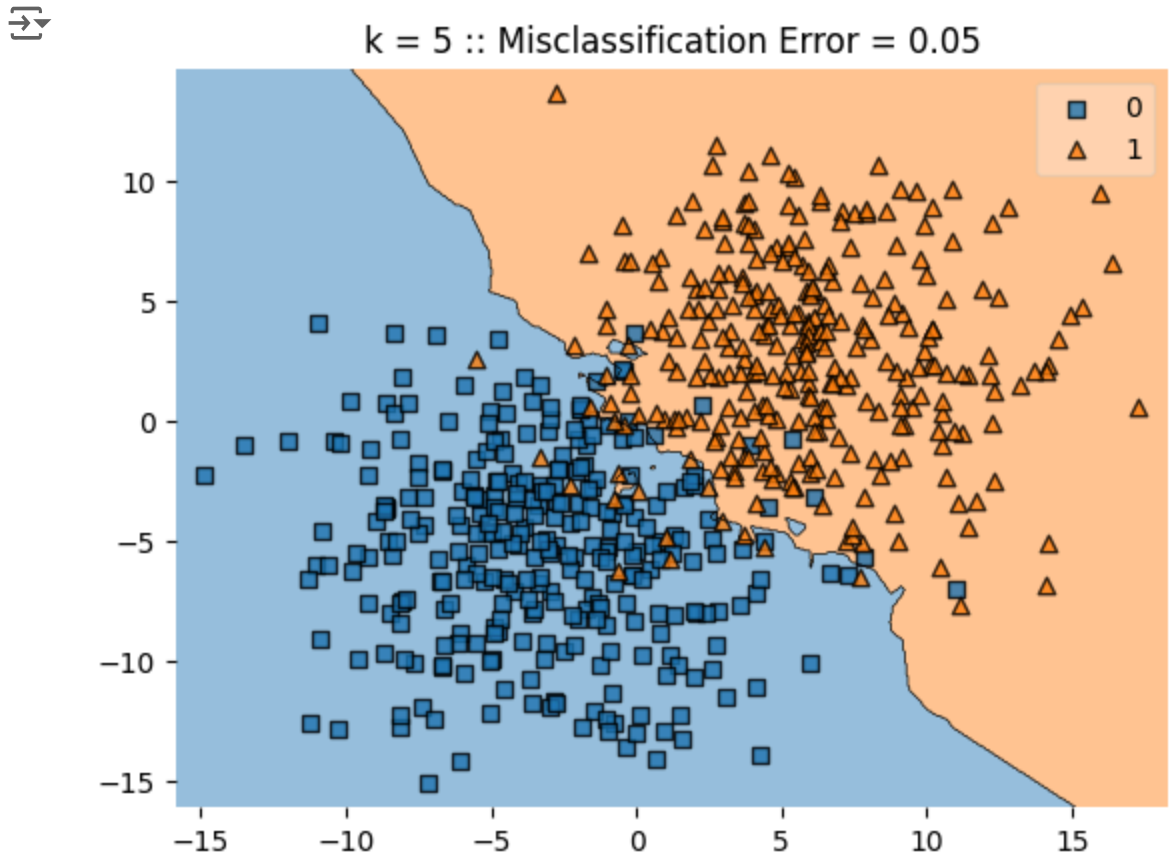
- also visualize the decision boundaries of *k*-nearest neighbor classifiers with k=3, k=5, k=7, k=9
- compute the prediction error on the training set for the *k*-nearest neighbor classifiers with k=3, k=5, k=7, k=9

```
# for k=3
k = 3
```

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

y_pred = knn.predict(X_train)

error = 1 - accuracy_score(y_train, y_pred)


plt.figure()
plot_decision_regions(X_train, y_train, clf=knn)
plt.title(f"k = {k} :: Misclassification Error = {error:.2f}")
plt.show()
```

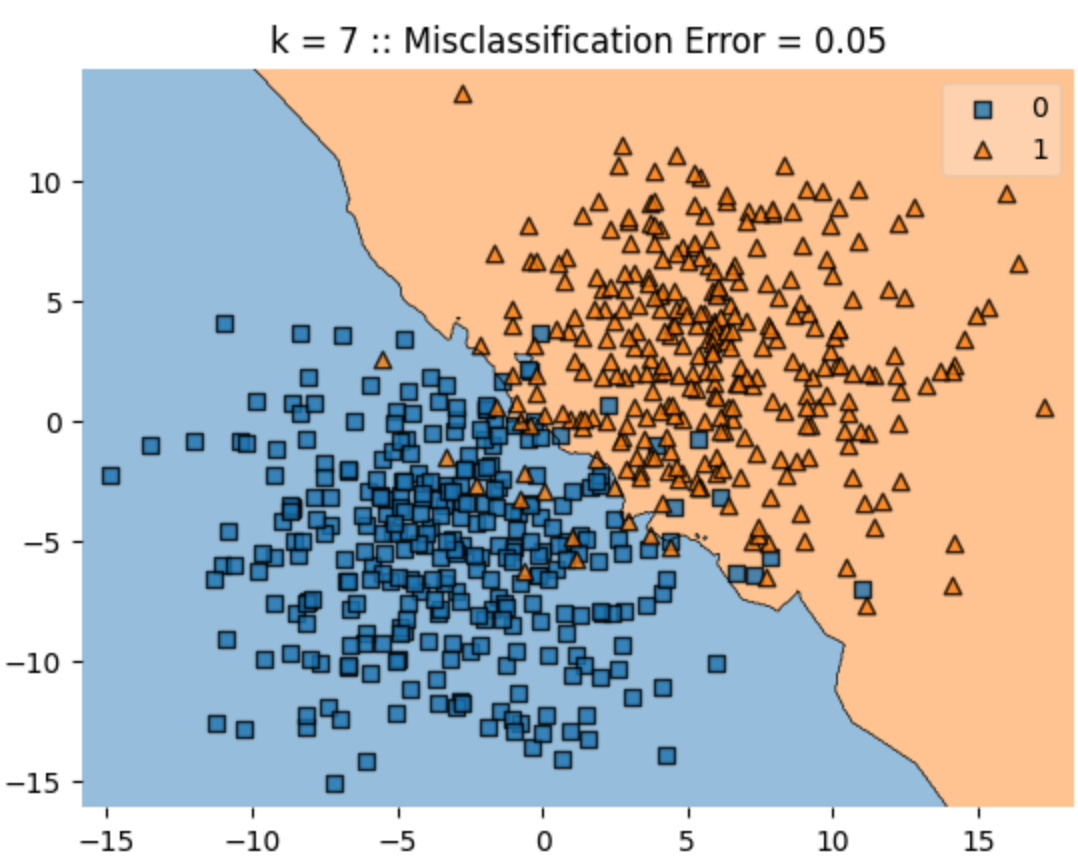


```
# for k=5
k = 5
knn = KNeighborsClassifier(n_neighbors=k)
knn.fit(X_train, y_train)

y_pred = knn.predict(X_train)

error = 1 - accuracy_score(y_train, y_pred)


plt.figure()
plot_decision_regions(X_train, y_train, clf=knn)
plt.title(f"k = {k} :: Misclassification Error = {error:.2f}")
plt.show()
```

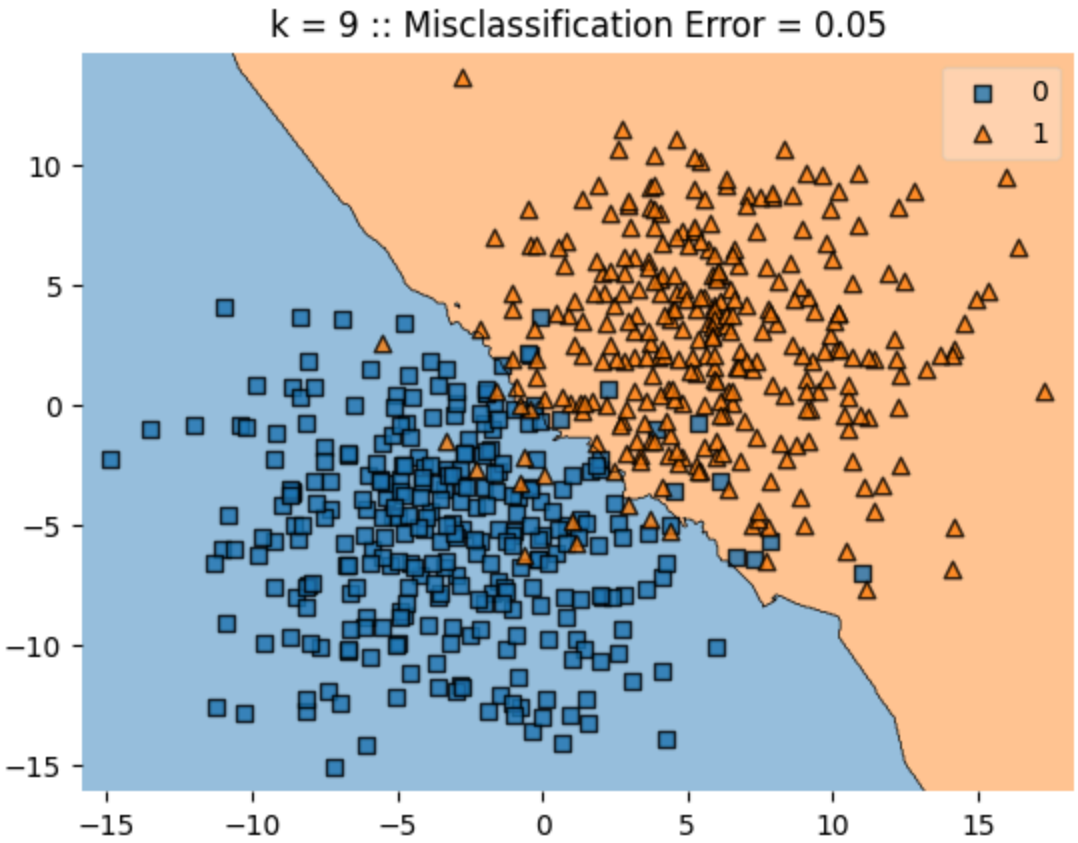



```
# for k=7
k = 7
knn = KNeighborsClassifier(n_neighbors=k)
knn.fit(X_train, y_train)

y_pred = knn.predict(X_train)

error = 1 - accuracy_score(y_train, y_pred)


plt.figure()
plot_decision_regions(X_train, y_train, clf=knn)
```

```
plt.title(f"k = {k} :: Misclassification Error = {error:.2f}")
plt.show()
```




```
# for k=9
k = 9
knn = KNeighborsClassifier(n_neighbors=k)
knn.fit(X_train, y_train)

y_pred = knn.predict(X_train)

error = 1 - accuracy_score(y_train, y_pred)


plt.figure()
plot_decision_regions(X_train, y_train, clf=knn)
plt.title(f"k = {k} :: Misclassification Error = {error:.2f}")
plt.show()
```



E 17)

Using the same approach you used in E 13), now load the `test_data.txt` file into a pandas array.

```
df_test = pd.read_csv("test_data.txt", delimiter=' ', header=0)

df_test.head()
```

	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

Assign the features to `X_test` and the class labels to `y_test` (similar to E 13):

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

E 18)

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 30% of the training dataset size, and the training subset should be 70% 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.

```
from sklearn.model_selection import train_test_split


X_train_sub, X_val, y_train_sub, y_val = train_test_split(X_train, y_train,
                                                          test_size= 0.3,
                                                          train_size= 0.7,
                                                          random_state=123,
                                                          stratify=y_train)
```

E 19)

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.

```
for k in range(1, 15):
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train_sub, y_train_sub)

    # evaluate the model on the train, validation, and test subsets
    y_pred = knn.predict(X_train_sub)
    train_error = 1 - accuracy_score(y_train_sub, y_pred)

    y_pred = knn.predict(X_val)
    val_error = 1 - accuracy_score(y_val, y_pred)

    y_pred = knn.predict(X_test)
    test_error = 1 - accuracy_score(y_test, y_pred)
```

```
    print(f'k = {k} :: test error = {test_error:.2f} :: validation error = {val_error:.2f} :
```

```
k = 1 :: test error = 0.09 :: validation error = 0.11 :: train error = 0.00
k = 2 :: test error = 0.08 :: validation error = 0.09 :: train error = 0.04
k = 3 :: test error = 0.06 :: validation error = 0.08 :: train error = 0.04
k = 4 :: test error = 0.06 :: validation error = 0.09 :: train error = 0.03
k = 5 :: test error = 0.05 :: validation error = 0.07 :: train error = 0.03
k = 6 :: test error = 0.05 :: validation error = 0.07 :: train error = 0.04
k = 7 :: test error = 0.06 :: validation error = 0.07 :: train error = 0.04
k = 8 :: test error = 0.05 :: validation error = 0.07 :: train error = 0.04
k = 9 :: test error = 0.05 :: validation error = 0.07 :: train error = 0.05
k = 10 :: test error = 0.04 :: validation error = 0.07 :: train error = 0.05
k = 11 :: test error = 0.05 :: validation error = 0.07 :: train error = 0.05
k = 12 :: test error = 0.04 :: validation error = 0.07 :: train error = 0.05
k = 13 :: test error = 0.05 :: validation error = 0.07 :: train error = 0.05
k = 14 :: test error = 0.04 :: validation error = 0.07 :: train error = 0.05
```

E 20)

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

```
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


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:

```
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:

```
rng = np.random.RandomState(123)

a = rng.rand(100)
b = rng.rand(100)


%timeit euclidean_dist(a, b)
```

```
19.3 µs ± 459 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

E 21)

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 22). Did you make the function faster? Yes or No? Explain why, in 1-2 sentences.

```
def euclidean_dist_np(a, b):
    return np.sqrt(np.sum((a - b) ** 2))


%timeit euclidean_dist_np(a, b)
```

```
3.58 µs ± 657 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

```
def euclidean_dist_norm(a, b):
    return np.linalg.norm(a - b)


%timeit euclidean_dist_norm(a, b)
```

```
1.75 µs ± 45.2 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

E 22)

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.

```
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:

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

```
[6.783831227508141e-05, 8.188761366767494e-05, 0.0001201014889748997]
```