



ELE 489

Fundamentals of Machine Learning

Homework 1

2220357137

Ali Özyüksel

03.04.2025

[GitHub](#)

Q1)

In the first question, we will be loading the data and visualizing it to understand what kind of features and labels we are dealing with.

```
import pandas as pd

# Let us read the data and give the corresponding column names.

data_file = "wine.data"
column_names = ["Class", "Alcohol", "Malic acid", "Ash", "Alcalinity of ash",
                "Magnesium", "Total phenols", "Flavanoids", "Nonflavanoid phenols", "Proanthocyanins",
                "Color intensity", "Hue", "OD280/OD315 of diluted wines", "Proline"]

df = pd.read_csv(data_file, names=column_names)
print(df.head(n=5))
```

	Class	Alcohol	Malic acid	Ash	Alcalinity of ash	Magnesium \
0	1	14.23	1.71	2.43	15.6	127
1	1	13.20	1.78	2.14	11.2	100
2	1	13.16	2.36	2.67	18.6	101
3	1	14.37	1.95	2.50	16.8	113
4	1	13.24	2.59	2.87	21.0	118

	Total phenols	Flavanoids	Nonflavanoid phenols	Proanthocyanins \
0	2.80	3.06	0.28	2.29
1	2.65	2.76	0.26	1.28
2	2.80	3.24	0.30	2.81
3	3.85	3.49	0.24	2.18
4	2.80	2.69	0.39	1.82

	Color intensity	Hue	OD280/OD315 of diluted wines	Proline
0	5.64	1.04	3.92	1065
1	4.38	1.05	3.40	1050
2	5.68	1.03	3.17	1185
3	7.80	0.86	3.45	1480
4	4.32	1.04	2.93	735

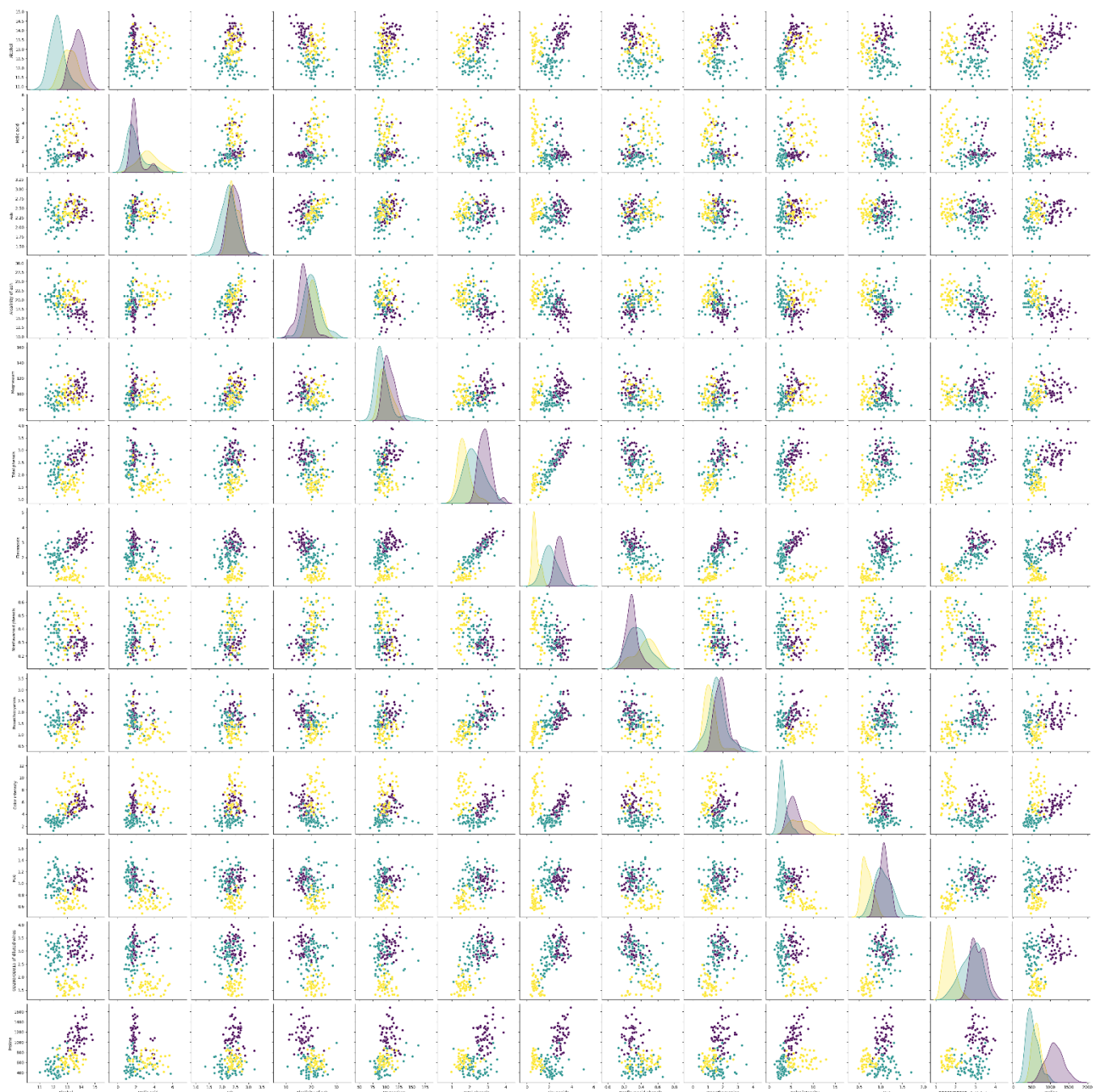
After loading the data, the very first thing we see is that values are not close to each other. Another saying, when proline is 1065, the ash is 2.43. So, this means that we will be needed to normalize the data later on.

Let us take a brief look at the data as a 'big picture'.

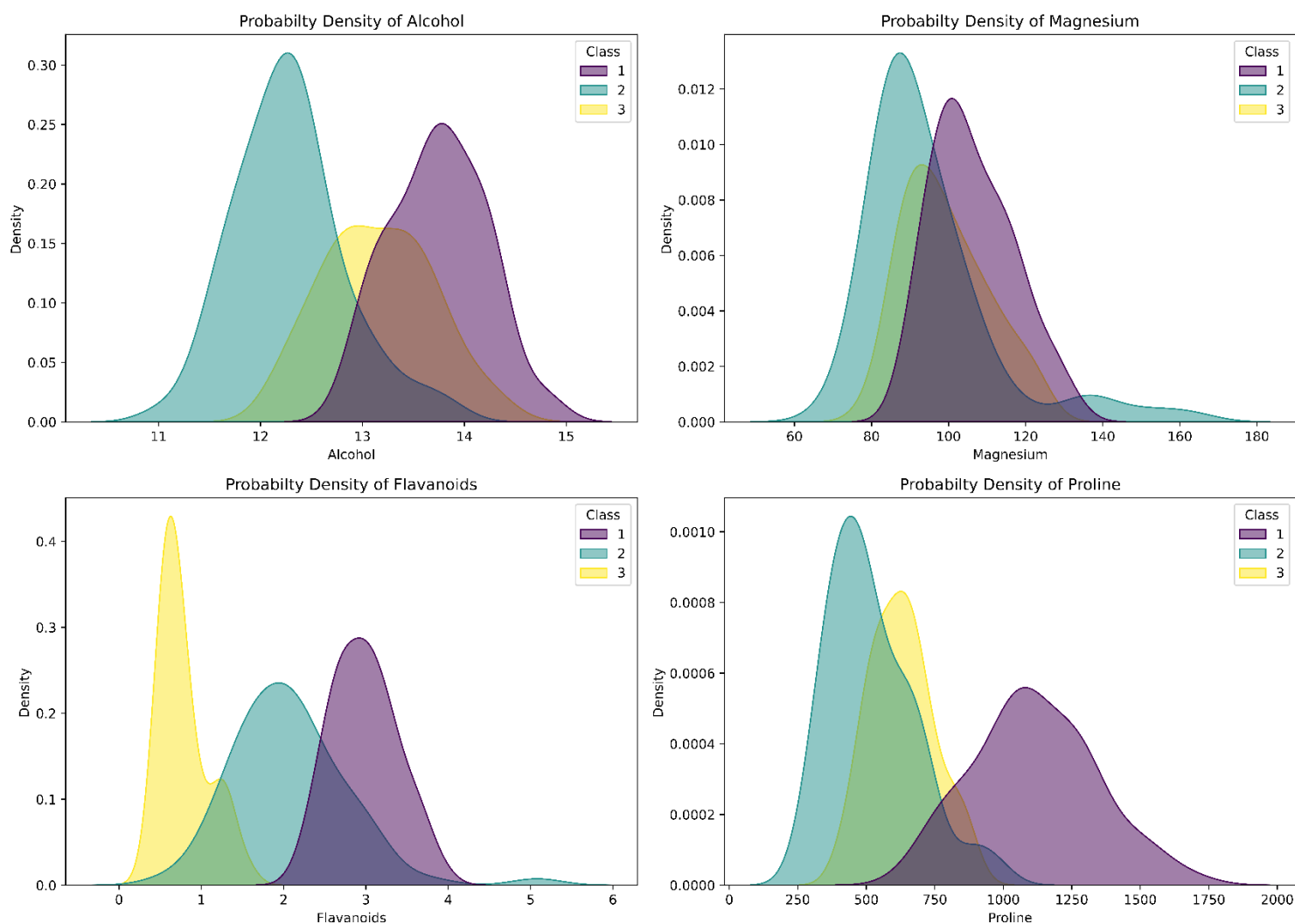
```
import seaborn as sns
import matplotlib.pyplot as plt

# At first, we can take a look at the big picture.
# By means of the pairplot function, we can see the each relationship between each pair of features.
# It takes time to run it but it is showing us what we are dealing with.

sns.pairplot(df, hue="Class", height=3, palette='viridis', diag_kind='kde')
plt.show()
```

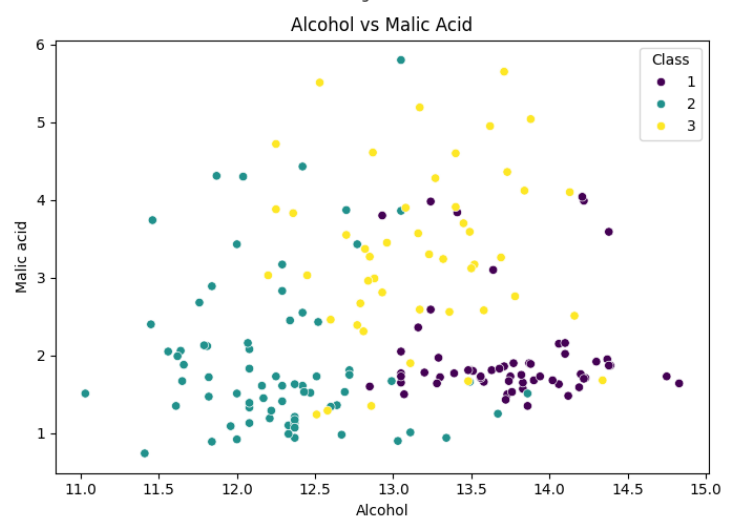
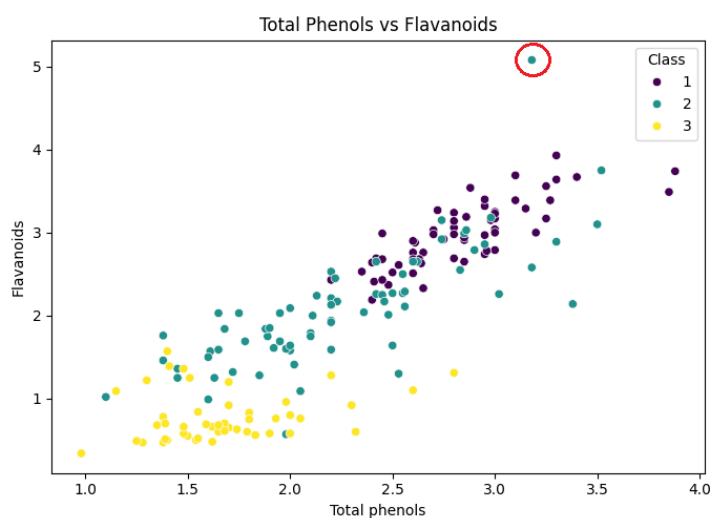
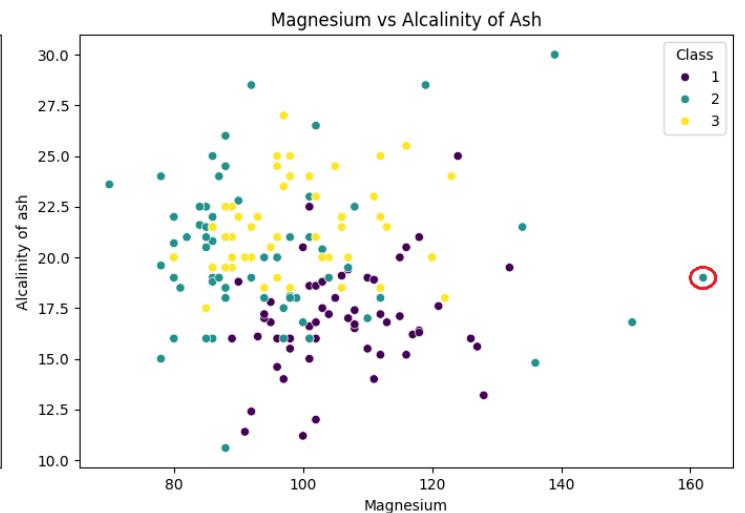
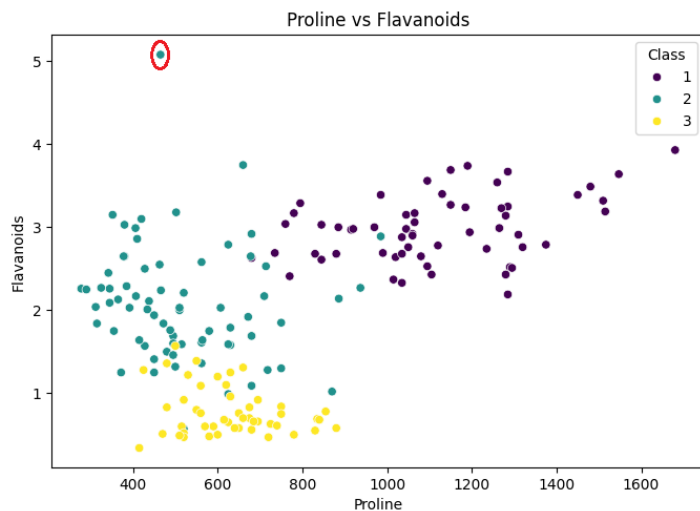


The diagonal graphs show the probability density of each feature. There are 3 colors representing 3 class. On top of this, other graphs (scatter plots) showing feature-feature characteristics such as correlation, overlapping and outliers. Although it seems confusing, we can get the insight of where to look. For example, alcohol, magnesium, color intensity and proline are worth to take a look as a probability density plot. Let us use that insight to delve deeper into our data.



It is easier to see overlapping from those graphs. For alcohol level of 13, there can be no judgment made. Let's say someone gave us a wine with an alcohol level of 13 and we want to label it as 1, 2 or 3. There is a strong possibility for it to be class 3 but class 1 and 2 are nearly strong as class 3. Or we may have detected the magnesium level as 100 so that probability of each class is equal to each other.

What we might want actually is that each class's probability density ranges different than one other. This would eliminate overlapping. For example, it would be better to class 2 to take value between alcohol levels between 11 and 12, class 1 to 12 and 13, finally class 3 to 13 and 14. By this way, if we measured the alcohol level as 11.5, we would have known it is class 2. It is nearly impossible to achieve in such a way that each class's probability density will be defined in distinct ranges. So, we are using numerous features to eliminate it.



We can look overlapping from scatter plots as well. In proline vs flavonoids graph, it is nearly possible to separate each class by simply encircling them. It is harder to do the same thing in magnesium vs alkalinity of ash graph.

Also, it is possible to see correlation of features. In total, phenols vs flavonoids graph, when a feature is tended to be increasing the other one does the same thing. So, we can come up with the conclusion that they are positively correlated. This can be useful in question 3. One of the distance metric will use correlation.

Furthermore, some of the outliers in the data are circulated in red as an example.

Q2)

In the second question, we will be preprocessing the data such as normalizing and handling missing values. Then we will be splitting it into training and test subsets.

```
print(df.isnull().sum(), "\n")
print(df.isna().sum())

#if there is any null or na values, we can drop them. but it is not necessary in this case. so that line is commented out.

#df.dropna(inplace=True)
✓ 0.0s
```

Alcohol	0
Malic acid	0
Ash	0
Alcalinity of ash	0
Magnesium	0
Total phenols	0
Flavanoids	0
Nonflavanoid phenols	0
Proanthocyanins	0
Color intensity	0
Hue	0
OD280/OD315 of diluted wines	0
Proline	0
dtype: int64	
Class	0
Alcohol	0
Malic acid	0
Ash	0
Alcalinity of ash	0
Magnesium	0
Total phenols	0
Flavanoids	0
Nonflavanoid phenols	0
Proanthocyanins	0
Color intensity	0
Hue	0
OD280/OD315 of diluted wines	0
Proline	0
dtype: int64	

It seems there is no missing value whatsoever. Then we are ready to normalize it.

```
#we are normalizing the data to make sure that all features are on the same scale.
#and using min-max normalization to scale the data between 0 and 1.
df_first_column = df.iloc[:, 0]
df_normalized = (df - df.min()) / (df.max() - df.min())

#we are keeping the first column as it is because they are just labels. we dont need to normalize them.
df_normalized.iloc[:, 0] = df_first_column
print(df_normalized.head())
print(df_normalized.min())
print(df_normalized.max())
```

✓ 0.0s

	Class	Alcohol	Malic acid	Ash	Alcalinity of ash	Magnesium \
0	1.0	0.842105	0.191700	0.572193	0.257732	0.619565
1	1.0	0.571053	0.205534	0.417112	0.030928	0.326087
2	1.0	0.560526	0.320158	0.700535	0.412371	0.336957
3	1.0	0.878947	0.239130	0.609626	0.319588	0.467391
4	1.0	0.581579	0.365613	0.807487	0.536082	0.521739

	Total phenols	Flavanoids	Nonflavanoid phenols	Proanthocyanins \
0	0.627586	0.573840	0.283019	0.593060
1	0.575862	0.510549	0.245283	0.274448
2	0.627586	0.611814	0.320755	0.757098
3	0.989655	0.664557	0.207547	0.558360
4	0.627586	0.495781	0.490566	0.444795

	Color intensity	Hue	OD280/OD315 of diluted wines	Proline
0	0.372014	0.455285	0.970696	0.561341
1	0.264505	0.463415	0.780220	0.550642
2	0.375427	0.447154	0.695971	0.646933
3	0.556314	0.308943	0.798535	0.857347
4	0.259386	0.455285	0.608059	0.325963

Class	1.0
Alcohol	0.0
Malic acid	0.0
Ash	0.0
Alcalinity of ash	0.0
...	
Hue	1.0
OD280/OD315 of diluted wines	1.0
Proline	1.0

dtype: float64

As it seems, all the values are between 0 to 1. If we would have kept the values as it is, it could result in misclassification of some instances.

```

from sklearn.model_selection import train_test_split

#we are splitting the data into training and testing sets.
#X is the features and Y is the target variable.
X = df.drop('Class', axis=1)
y = df['Class']
#we are using 80% of the data for training and 20% for testing.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
df.shape, X_train.shape, X_test.shape, y_train.shape, y_test.shape
✓ 0.0s
((178, 14), (142, 13), (36, 13), (142,), (36,))

```

We divided features and labels into test and training data with a ratio of 20% and 80%, respectively. Random state is a shuffle seed which is 42 for this example so that each time code runs, it splits the data in the same way.

Q3)

In question 3, we will implement the k-NN algorithm from scratch.

```

import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

class KNN:
> def __init__(self, k=3, metric='euclidean'): ...
> def fit(self, X_train, y_train): ...
    # distance functions.
> def euclidean_distance(self, x1, x2): ...
> def manhattan_distance(self, x1, x2): ...
> def mahalanobis_distance(self, x1, x2): ...
> def compute_distance(self, x1, x2): ...
    # prediction function. it is basically a loop that iterates over the test data and finds the k nearest neighbors by cheking the distance.
> def predict(self, X): ...
> def evaluate(self, X_test, y_test): ...
    #self written confusion matrix function.
> def confusion_matrix(self, x_test, y_test): ...
    #plotting confusion matrix as a heatmap to visualize the performance of the model.
> def plotting(self, x_test, y_test): ...

```

To use it, we create an KNN object with a given k and metric. We are fitting train data to it so when we use other methods inside of it, it will use the training data which is saved into memory. Distance functions are just a mathematical expression. Implemented k-NN algorithm is behind the predict function.


```
# prediction function. it is basically a loop that iterates over the test data and finds the k nearest neighbors by checking the distance.
def predict(self, X):
    X = np.array(X)
    predictions = []
    for x in X:
        distances = [self.compute_distance(x, x_train) for x_train in self.X_train]
        k_indices = np.argsort(distances)[:self.k]
        k_nearest_labels = [self.y_train[i] for i in k_indices]
        unique_labels, counts = np.unique(k_nearest_labels, return_counts=True)
        most_common = unique_labels[np.argmax(counts)]
        predictions.append(most_common)
    return np.array(predictions)
```

To use the predict function, we will give it the data frame of the features as well as given training features and training labels in fit function. It will take an element of given features and compute the distance between each training element. In this process, we can use different distance functions for our needs. They will be examined in the next section.

Then we sort the distance from shortest to longest and take only k-element. We know the labels corresponding to each of them (as we are keeping them in y_train). So, the most repeated label becomes our prediction.

In return, this function gives us labels for given features.

```
# distance functions.
def euclidean_distance(self, x1, x2):
    return np.sqrt(np.sum((x1 - x2) ** 2))

def manhattan_distance(self, x1, x2):
    return np.sum(np.abs(x1 - x2))

def mahalanobis_distance(self, x1, x2):
    diff = x1 - x2
    cov_matrix = np.cov(self.X_train, rowvar=False)
    cov_matrix_inv = np.linalg.inv(cov_matrix)
    return np.sqrt(np.dot(np.dot(diff.T, cov_matrix_inv), diff))


def compute_distance(self, x1, x2):
    if self.metric == 'euclidean':
        return self.euclidean_distance(x1, x2)
    elif self.metric == 'manhattan':
        return self.manhattan_distance(x1, x2)
    elif self.metric == 'mahalanobis':
        return self.mahalanobis_distance(x1, x2)
    else:
        raise ValueError("Unsupported metric. Use 'euclidean', 'mahalanobis' or 'manhattan'.")
```

As it is stated, most of the processes in k-NN algorithms are to measure the distance between two elements. The first two distance functions are well known: Euclidian and Manhattan. The third one is called Mahalanobis distance. Which takes into account the correlation between the features. Their performance will be given later on.

```

import knn
# we can create a KNN object and fit it to the training data with given k value and distance metric.
k1 = knn.KNN(3, "euclidean")
k1.fit(X_train, y_train)
y_pred=k1.predict(X_test)
print(y_pred)
# we can check the accuracy of the model.
acc = k1.evaluate(X_test, y_test)
print(acc)
# we can see the confusion matrix of the model by looking it as heatmap.
k1.plotting(X_test, y_test)
# also we can see the confusion matrix in a tabular form.
conf_matrix = k1.confusion_matrix(X_test, y_test)
print(conf_matrix)

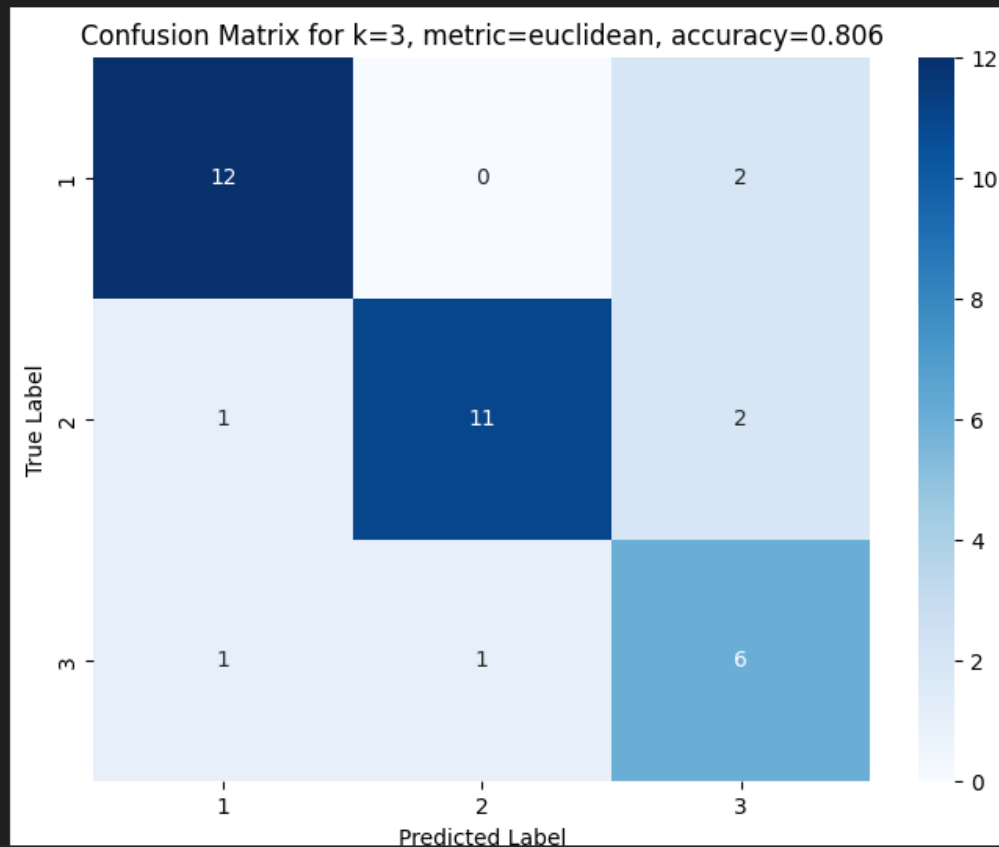
```

✓ 0.4s  Open 'conf_matrix' in Data Wrangler

```

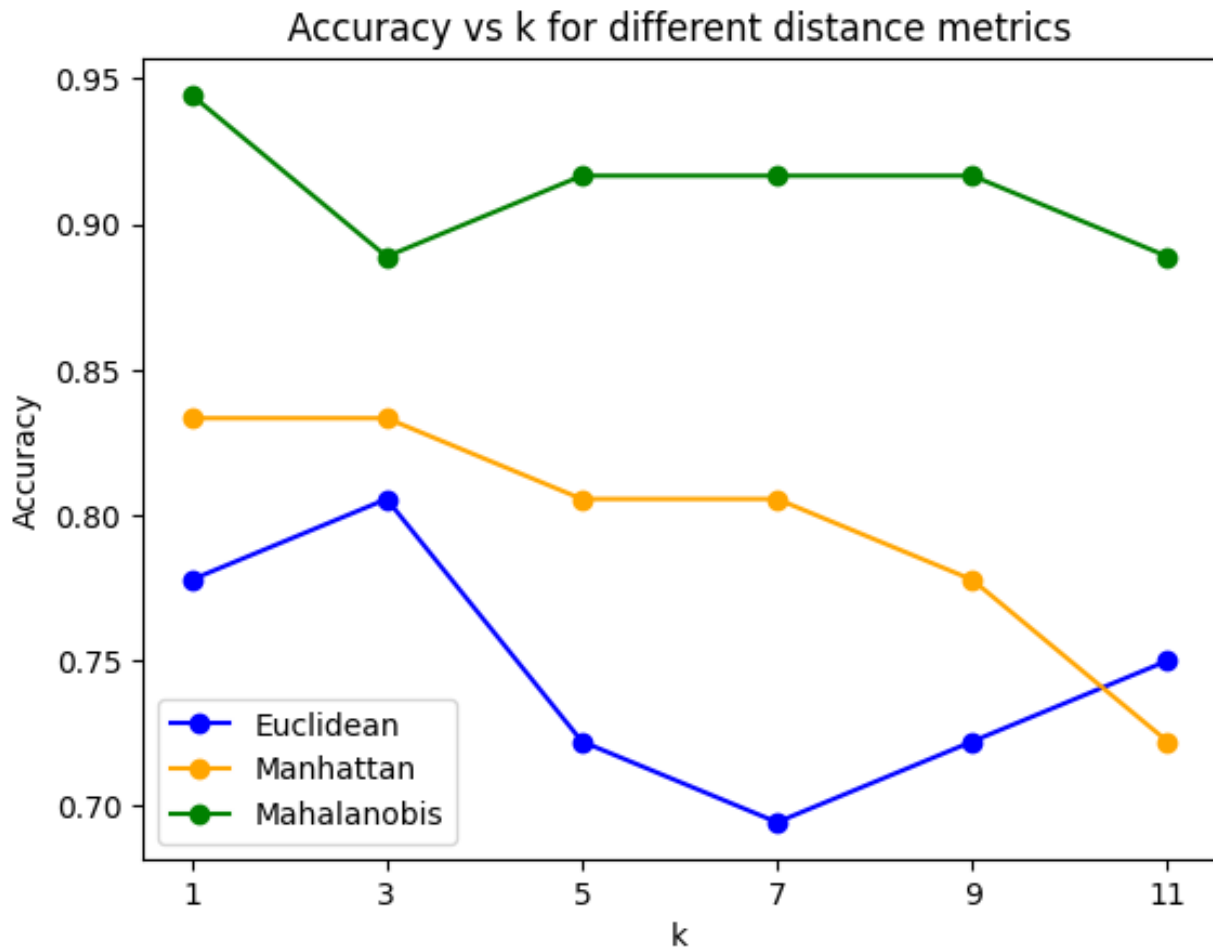
[3 1 3 1 2 1 2 3 1 1 3 3 1 2 1 2 2 2 1 2 1 2 3 3 2 3 2 3 2 1 1 2 3 1 1]
0.8055555555555556

```



Example of usage of the k-NN algorithm. The functions evaluate, plotting, confusion matrix are self-written functions.

In the next step we can plot accuracy versus k. To make it in a more scientific way, as we keep the metric distance, we will change the k. We will do it for each metric.

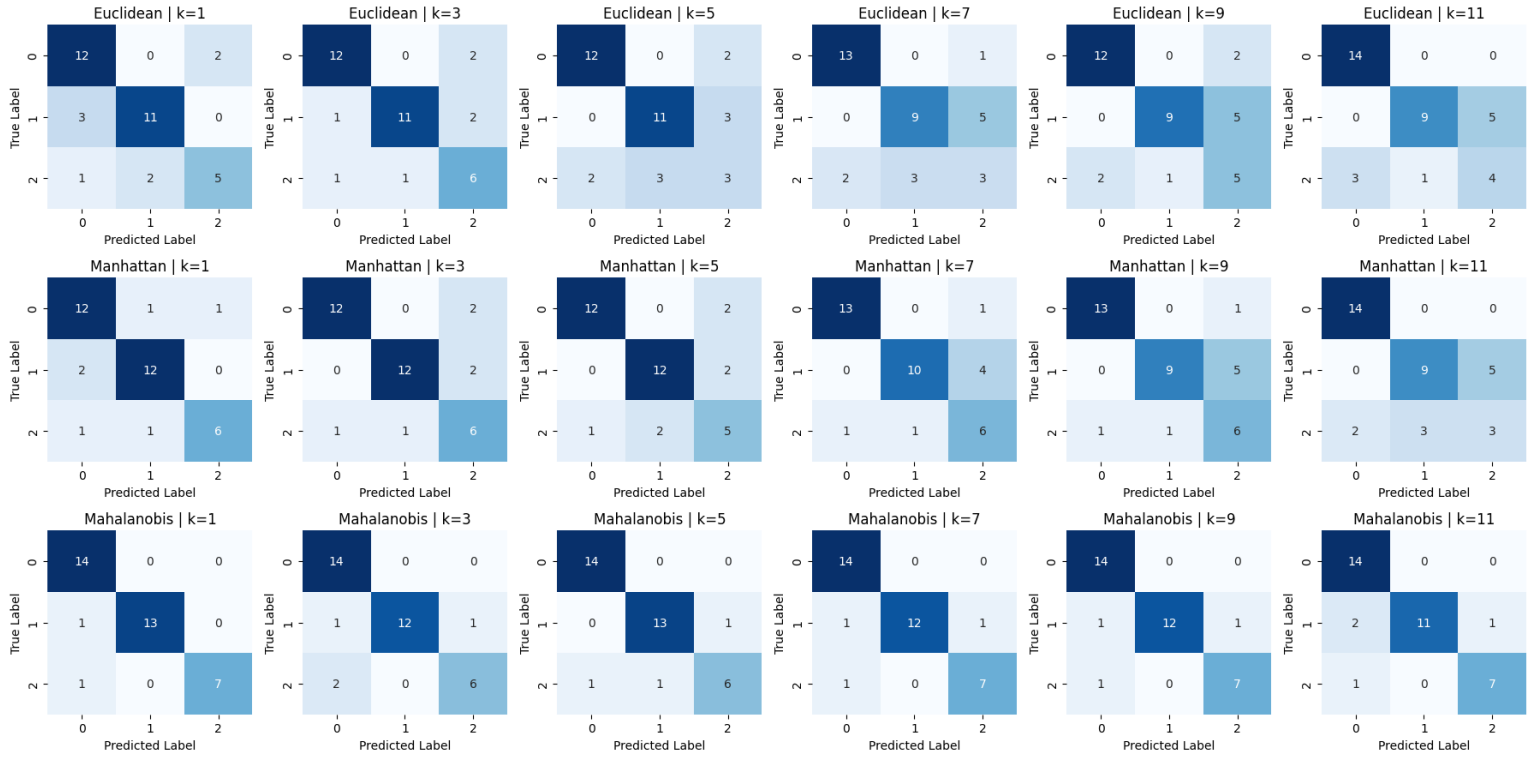


Accuracy becomes better when we use Mahalanobis distance. No other metric can get even closer to it. And it seems it gives the best accuracy when $k=1$.

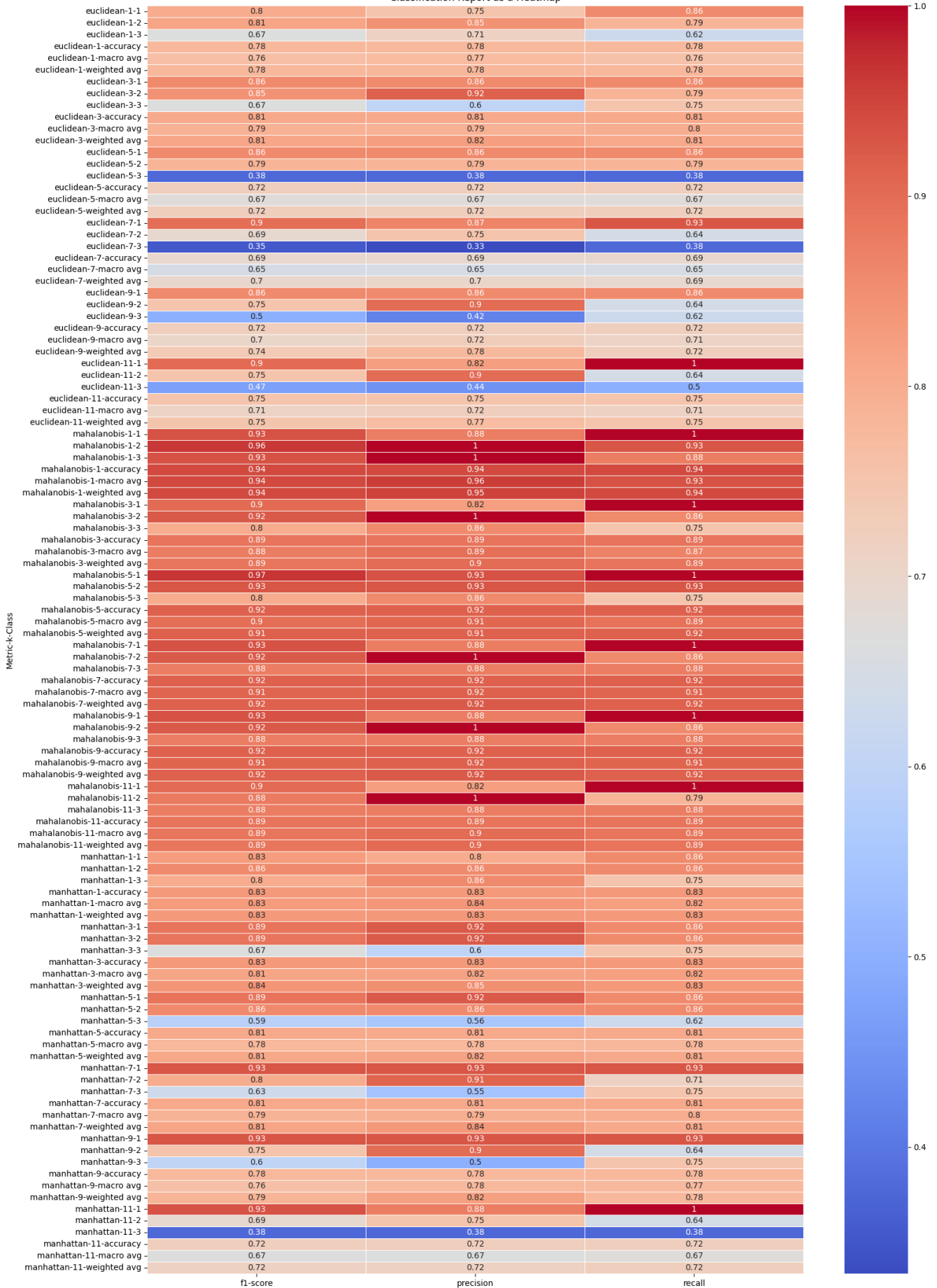
Manhattan seems to outperform Euclidean metric. The tides only change when $k=11$. For other values, Manhattan is simply better than Euclidean. Also, it gives the best accuracy for $k=1$ and $k=3$.

Euclidean is visibly the worst amongst 3 metrics. It gives the best accuracy for $k=3$.

We can do the similar way when it comes to confusion matrix and classification report. However, it is easier to examine classification report from the notebook file because there are better display options on it. Nevertheless, a suitable graph is given below.



Classification Report as a Heatmap



Although it seems confusing, the more readable classification report is on the notebook file.

As a conclusion, we have done k-NN algorithm from scratch. We saw how changing metric and changing k affects classification. We processed the data before we begin to classification.