HACETTEPE UNIVERSITY

DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING

ELE489 – Fundamentals of Machine Learning

– Homework I–

Murat DOĞAN

2200357075

GitHub

03.04.2025

1) Data Loading and Feature Visualization

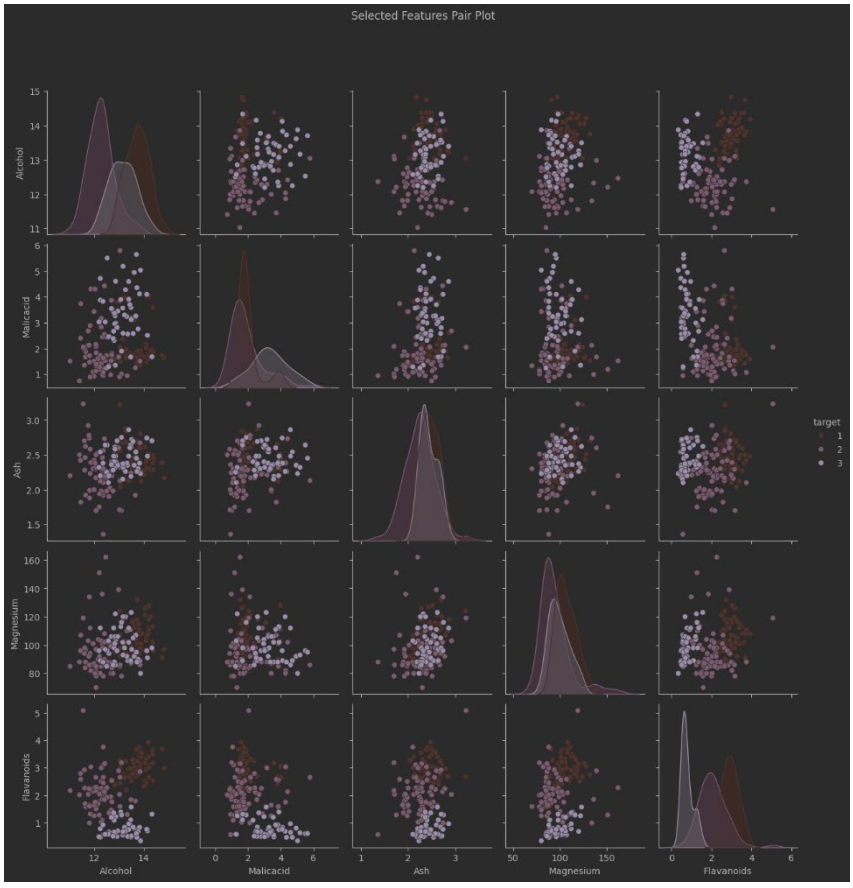
I used all the data while processing the data, but when I visualized all of them, a clear observation could not be made because there were too many features. So I selected a few features and visualized them. You can see the data plotted according to the selected features and its visualized form below.

```
[17]: selected_features = ['Alcohol', 'Malicacid', 'Ash','Magnesium','Flavanoids', 'target']
      df_selected = df[selected_features]
      before=sns.pairplot(df_selected,hue='target')
      before.fig.suptitle('Selected Features Pair Plot', y=1.08) # plotting graph of selected features
      df_selected
```

[17]:

	Alcohol	Malicacid	Ash	Magnesium	Flavanoids	target
0	14.23	1.71	2.43	127	3.06	1
1	13.20	1.78	2.14	100	2.76	1
2	13.16	2.36	2.67	101	3.24	1
3	14.37	1.95	2.50	113	3.49	1
4	13.24	2.59	2.87	118	2.69	1
...	...	...	...	...	...	...
173	13.71	5.65	2.45	95	0.61	3
174	13.40	3.91	2.48	102	0.75	3
175	13.27	4.28	2.26	120	0.69	3
176	13.17	2.59	2.37	120	0.68	3
177	14.13	4.10	2.74	96	0.76	3

178 rows × 6 columns



We see in these graphs, some classes are easier to separate, like class 1, which stands out with higher flavanoids and alcohol levels. However, features like malic acid, ash, and magnesium overlap significantly between the classes, making it harder to distinguish them. Overall, flavanoids and alcohol are the most useful for separating the classes, while the others aren't as effective.

2) Data Normalization and Dataset Splitting

In this part, I performed data normalization to scale the features and ensure that each feature contributes equally to the distance calculations in the k-NN algorithm. Normalization is crucial because features with larger numerical ranges could dominate the distance metrics, leading to biased results.

Before normalization, I proceeded to split the dataset into training and testing sets. The data was divided using the `train_test_split` function, which ensures that the model is trained on one portion of the data and evaluated on another. To ensure reproducibility, I used the parameters `shuffle=True` and `random_state=0`. The `shuffle=True` ensures that the data is randomly shuffled before splitting, avoiding any ordering biases in the dataset. I used `random_state=0` to ensure that the split would be consistent across different runs.

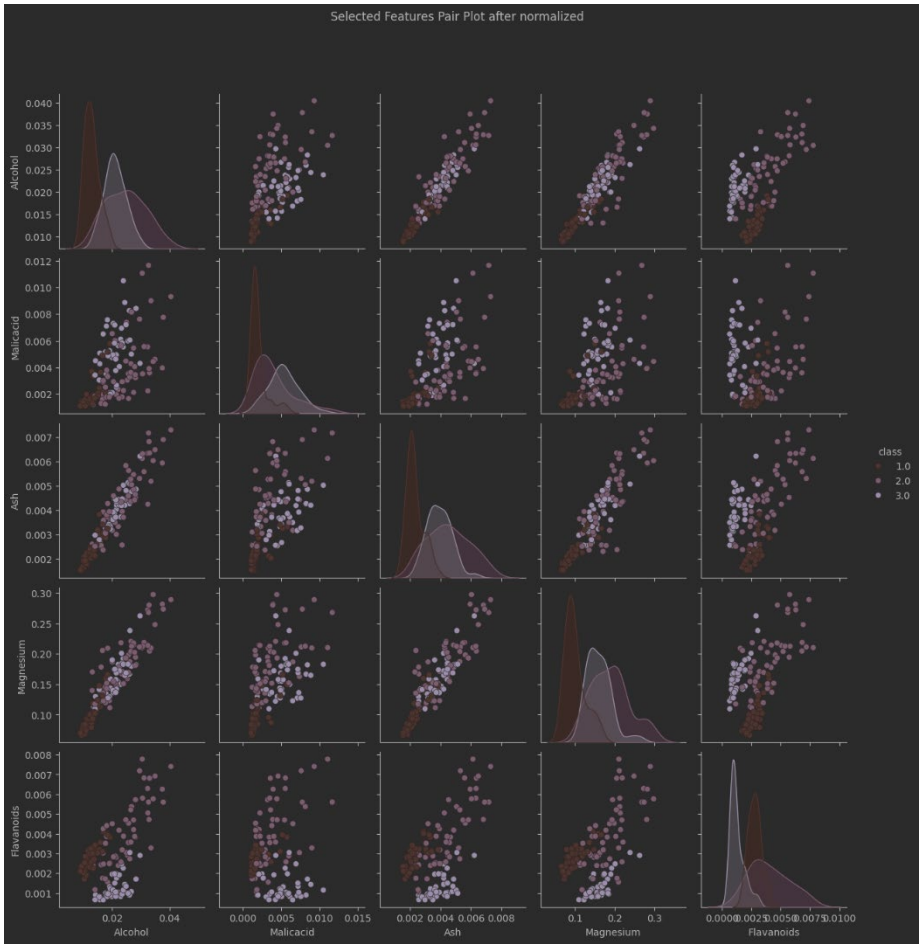
```
x_train, x_test, y_train, y_test= train_test_split(a, b,
                                                   test_size= 0.2,
                                                   shuffle= True, #shuffle the data to avoid bias
                                                   random_state= 0)
#x_train= np.asarray(x_train)
#y_train= np.asarray(y_train) # split the data into train and test sets
```

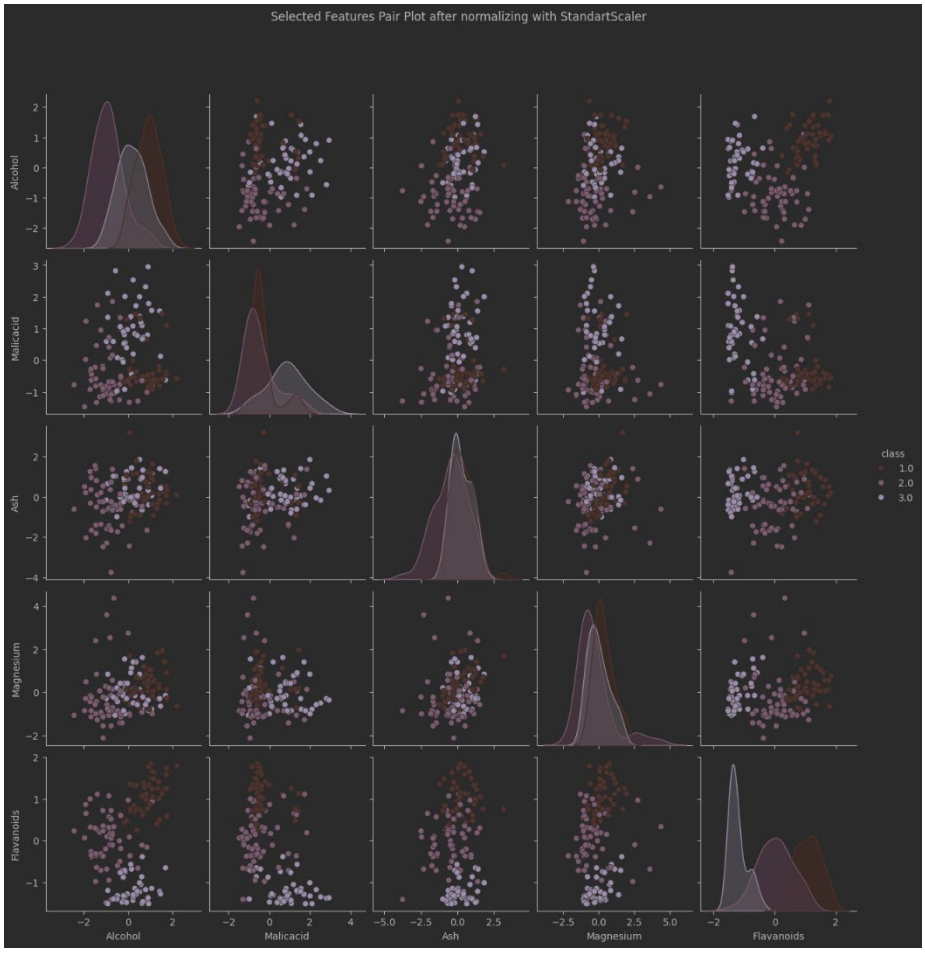
While normalizing the data, I used two different normalization functions: Normalizer and StandardScaler. Based on the graphs, StandardScaler appears to be more suitable for KNN applications. The reason for this is that StandardScaler centers the data and scales it to unit variance, ensuring that all features contribute equally to the distance calculations. In contrast, Normalizer scales each data point independently, which can result in less distinct class separations, as seen in the plots. Therefore, StandardScaler provides clearer class separations and more reliable distance-based decisions for KNN.

```
#normalize data set
scaler= Normalizer().fit(x_train) # the  normalize scaler is fitted to the training set
normalized_x_train= scaler.transform(x_train) # the scaler is applied to the training set
normalized_x_test= scaler.transform(x_test) # the scaler is applied to the test set
Sc=StandardScaler()
normalized_x_train1= Sc.fit_transform(x_train) # the standartscaler is applied to the training set
normalized_x_test1= Sc.fit_transform(x_test) # the standartscaler is applied to the test set
```

```
target_column_name = y.columns[0]
selected_features2 = ['Alcohol', 'Malicacid', 'Ash', 'Magnesium', 'Flavanoids']
df_2=pd.DataFrame(data= np.c_[normalized_x_train, y_train],columns=list(x.columns) + [target_column_name]) # Creating pandas data frame of normalizing da
df_selected2 = df_2[selected_features2 + [target_column_name]]
after=sns.pairplot(df_selected2,hue=target_column_name)
after.fig.suptitle('Selected Features Pair Plot after normalizing', y=1.08) # Plotting graph of selected features after normalizing data set
```

After normalizing the dataset, I visualized the normalized data to observe how the feature distributions change. The visualizations help to confirm that all features are now on a similar scale, which is important for the effectiveness of the k-NN model.

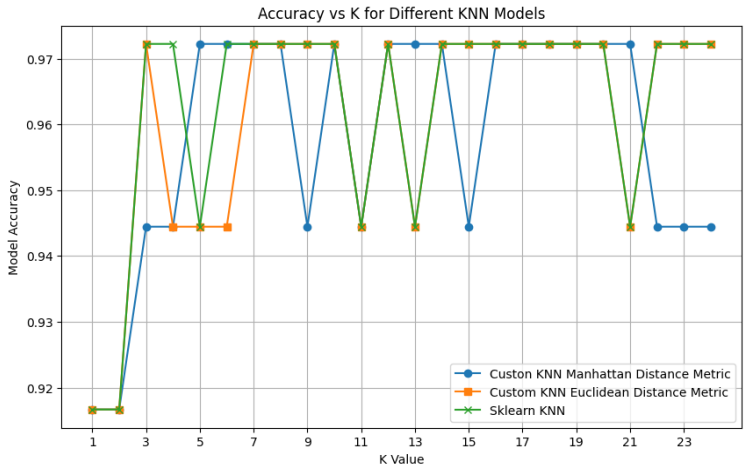

Selected Features Pair Plot after normalized
Alcohol
Malicacid
Ash
Magnesium
Flavanoids
class
1.0
2.0
3.0


Selected Features Pair Plot after normalizing with StandartScaler
Alcohol
Malicacid
Ash
Magnesium
Flavanoids
class
1.0
2.0
3.0

3) k-NN Algorithm Implementation and Comparison

In this section, I compared the results of my custom k-NN algorithm with those obtained using the sklearn KNeighborsClassifier. I evaluated the performance of different distance metrics (Euclidean and Manhattan) and plotted accuracy against various K values to identify the optimal number of neighbors.

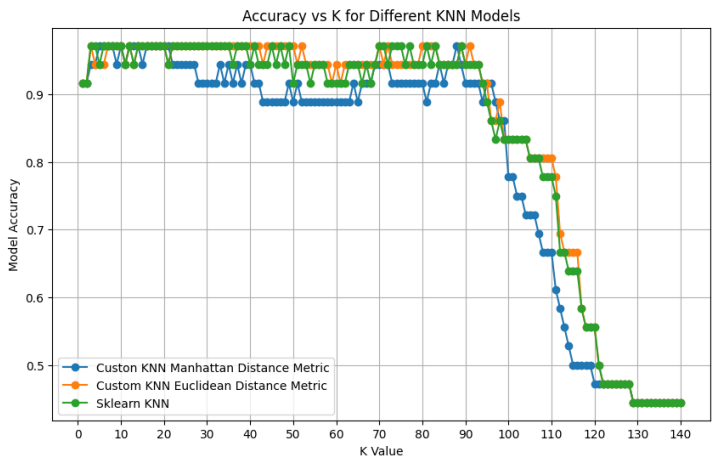
```
[32]: plt.figure(figsize=(10, 6))
plt.plot(k_range, scores3, label="Custon KNN Manhattan Distance Metric", marker='o')
plt.plot(k_range, scores2, label="Custom KNN Euclidean Distance Metric", marker='s')
plt.plot(k_range, scores1, label="Sklearn KNN", marker='x')

plt.xlabel("K Value")
plt.ylabel("Model Accuracy")
plt.title("Accuracy vs K for Different KNN Models")
plt.legend()
plt.grid(True)
plt.xticks(range(1, 25, 2))  # Adjust the frequency of ticks on the x-axis
plt.show()
```



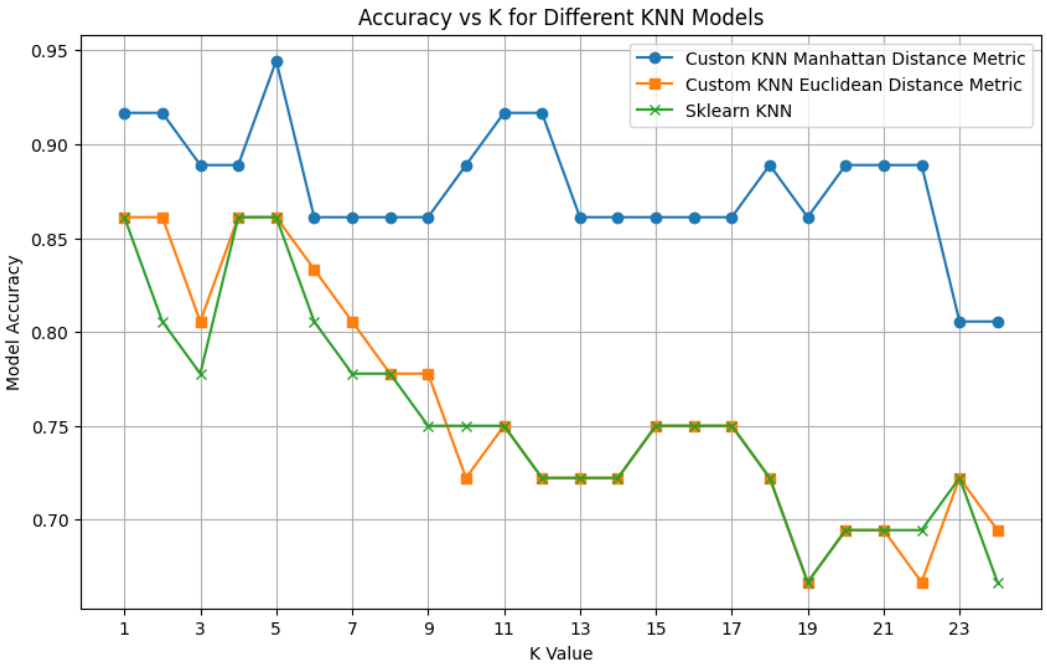
When we look at the comparison in these graphs, we see that the accuracy of the models we use in the data we normalized using StandardScaler for small K values is very similar to each other and very high.

However, when we increase the K value, the performance of the models decreases. Since the number of neighbors increases too much, even if it belongs to another class, since there is more data from the other class, our test data will appear to be in a place that does not belong to it, and this will decrease accuracy. We can observe this from our graphs.

```
[24]: plt.figure(figsize=(10, 6))
plt.plot(k_range, fscores3, label="Custon KNN Manhattan Distance Metric", marker='o')
plt.plot(k_range, fscores2, label="Custom KNN Euclidean Distance Metric", marker='o')
plt.plot(k_range, fscores1, label="Sklearn KNN", marker='o')

plt.xlabel("K Value")
plt.ylabel("Model Accuracy")
plt.title("Accuracy vs K for Different KNN Models")
plt.legend()
plt.grid(True)
plt.xticks(range(0, 141, 10))  # Adjust the frequency of ticks on the x-axis
plt.show()
```



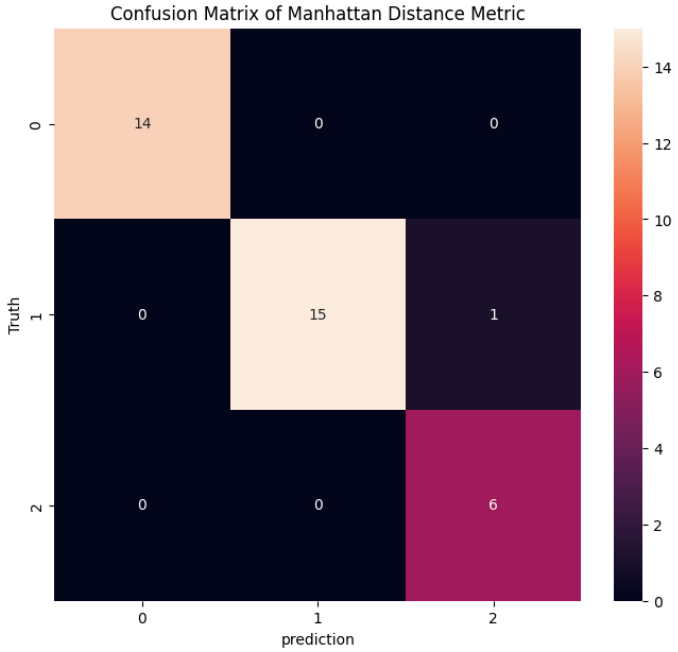
In this graph, we observe the accuracy performance of the data we normalized with the Normalizer function. It performs lower than the other normalization functions. The model using the Manhattan distance metric gave a better performance here.



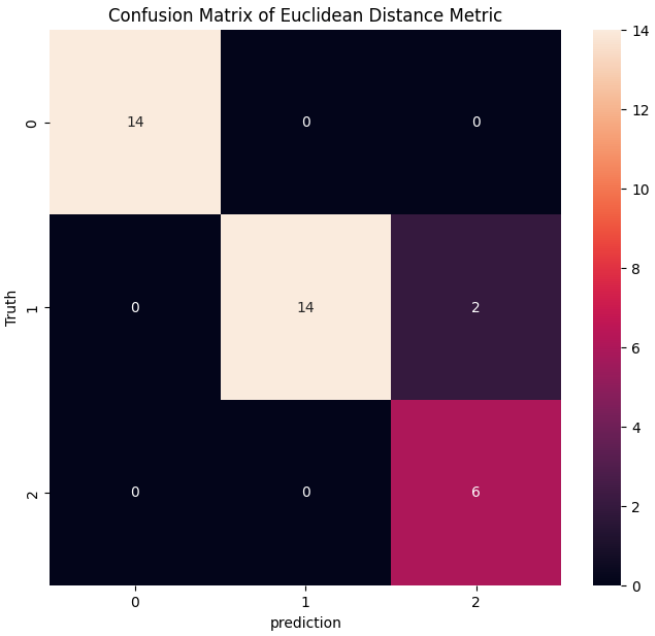
Confusion Matrix, Classification Report and Conclusion

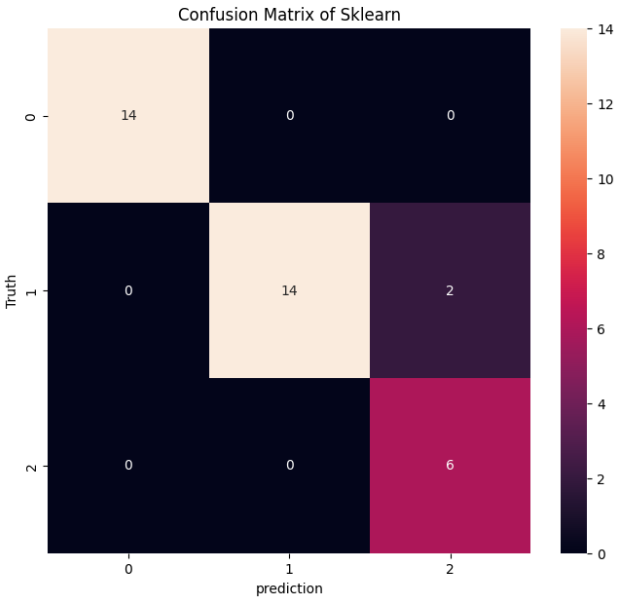
The confusion matrix offers a detailed view of the model's performance by presenting the counts of true positives, true negatives, false positives, and false negatives. This analysis is crucial for understanding specific areas where the model excels and where it may be misclassifying instances.

[20]:
```
# Train KNN models with K=5 for different distance metrics (Manhattan, Euclidean, Sklearn) and compute confusion matrices
knnhandmade3=KNN(k=5,distance_metric="manhattan")
knnhandmade3.fit(normalized_x_train1,y_train.values)
ypred=knnhandmade3.predict(normalized_x_test1)

knnhandmade2=KNN(k=5,distance_metric="euclidean")
knnhandmade2.fit(normalized_x_train1,y_train.values)
ypred1=knnhandmade2.predict(normalized_x_test1)

knn2=KNeighborsClassifier(n_neighbors=5)
knn2.fit(normalized_x_train1,y_train)
ypred2=knn2.predict(normalized_x_test1)
# Create confusion matrices for the predictions of each model
cm1=confusion_matrix(y_test,ypred)
cm2=confusion_matrix(y_test,ypred1)
cm3=confusion_matrix(y_test,ypred2)
```



I created the Confusion Matrix by choosing one of the small K values that showed high performance (5) and observed the performance of my models there and observed consistent results as in the graphs.





```
Classification Report for Manhattan Distance Metric:
              precision    recall  f1-score   support

           1       1.00      1.00      1.00        14
           2       1.00      0.94      0.97        16
           3       0.86      1.00      0.92         6

    accuracy                           0.97        36
   macro avg       0.95      0.98      0.96        36
weighted avg       0.98      0.97      0.97        36

Classification Report for Euclidean Distance Metric:
              precision    recall  f1-score   support

           1       1.00      1.00      1.00        14
           2       1.00      0.88      0.93        16
           3       0.75      1.00      0.86         6

    accuracy                           0.94        36
   macro avg       0.92      0.96      0.93        36
weighted avg       0.96      0.94      0.95        36

Classification Report for Sklearn KNN:
              precision    recall  f1-score   support

           1       1.00      1.00      1.00        14
           2       1.00      0.88      0.93        16
           3       0.75      1.00      0.86         6

    accuracy                           0.94        36
   macro avg       0.92      0.96      0.93        36
weighted avg       0.96      0.94      0.95        36
```

If we evaluate the Classification report values, we can observe them from here.

To briefly summarize the parameters here: Precision shows how accurate the positive predictions are, while Recall indicates how well the model identifies actual positive instances. The F1-score balances Precision and Recall, providing a more comprehensive performance measure. Accuracy reflects the overall correct predictions, and Support indicates the number of actual occurrences of each class. Together, these metrics help evaluate the model's performance both overall and for each individual class.

All three models, Manhattan distance, Euclidean distance, and Sklearn KNN, showed strong performance with high precision and recall values. Class 1 was consistently predicted perfectly by all models with 100% precision and recall. For Class 2, both the Manhattan and Sklearn models achieved a precision of 1.00, but with slightly lower recall for Euclidean (0.88), indicating some instances were missed. Class 3 had the lowest precision (0.75) across all models, but the recall for Class 3 was 1.00, meaning all relevant instances were captured. When we look at it in general, we can say that the model performances are similar to our other analyses and that we have verified each other in this way.

In this assignment, I implemented and analyzed the k-NN algorithm using different models and distance metrics. The comparison between my own k-NN implementation and sklearn's version yielded consistent outcomes. Through the confusion matrix, I confirmed that the model performs well overall, although some misclassifications still occur, especially with overlapping features. This project highlighted the importance of preprocessing and parameter selection in building an effective k-NN model.