Machine Learning Experiment Report

Experiment 1: KNN Algorithm for Iris Flower Classification

Student ID	24SF51099
Student Name	Ilkin Masimov
Experiment Date	09.05.2025

Please include screenshots of the code and experimental results in the report, accompanied by brief descriptions in the text.

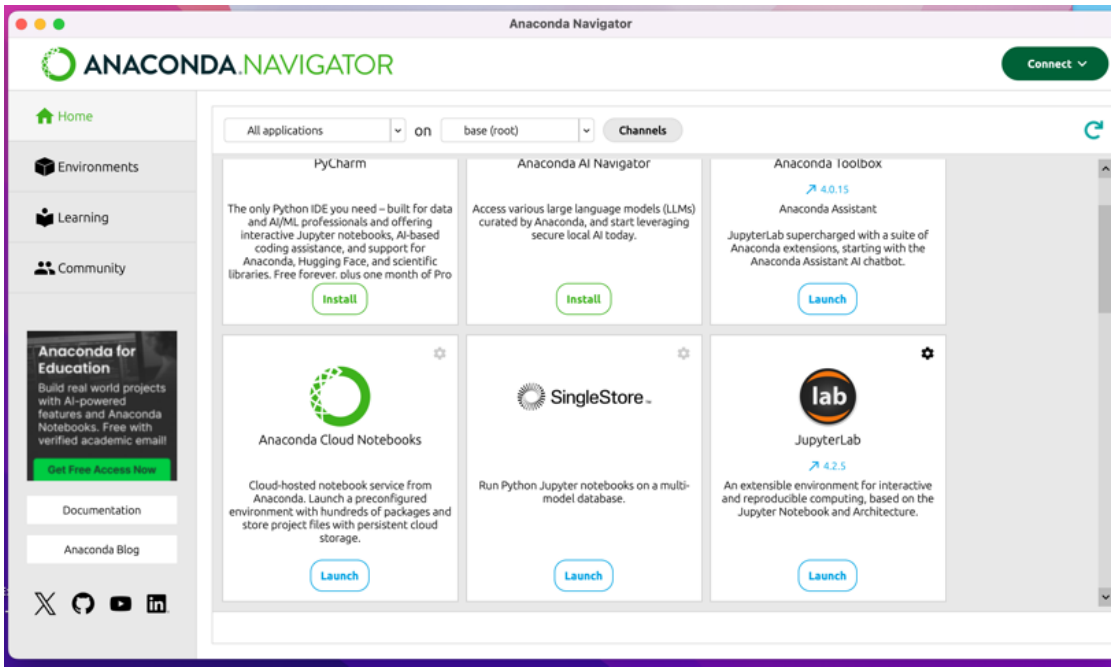
Harbin Institute of Technology, Shenzhen

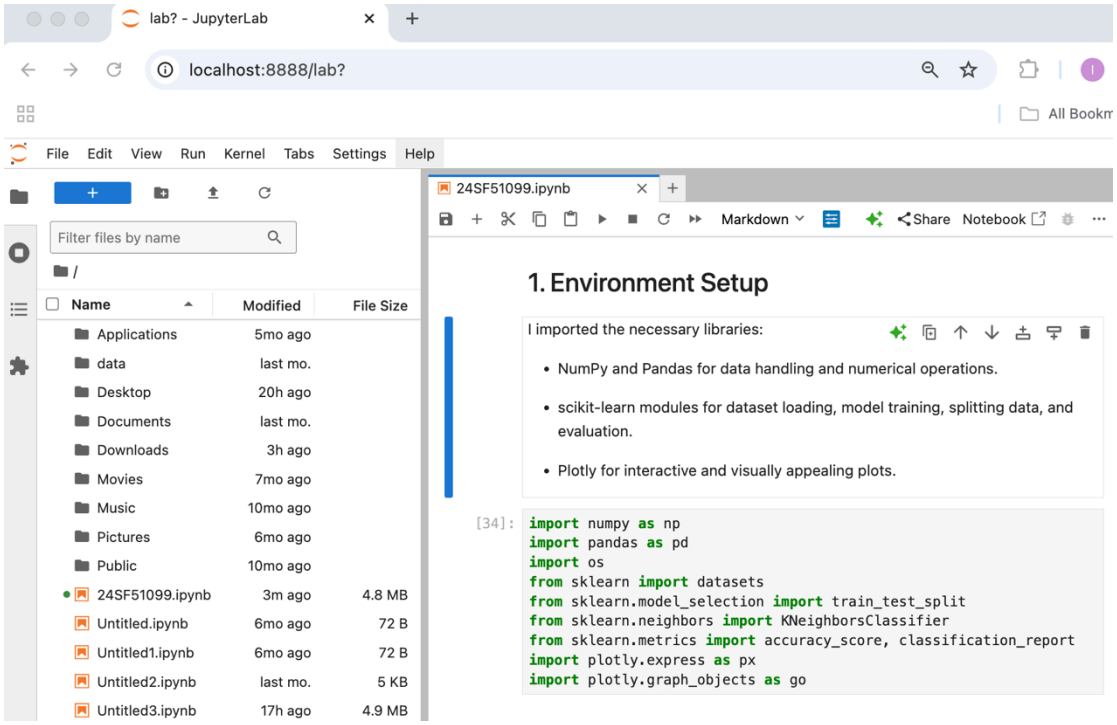
Content

1) Environment Setup

- Describe the process of installing Anaconda and Jupyter Notebook and ensure the environment is operational.

I already had Anaconda installed on my Mac and used Jupyter before, so the environment was operational.





2) Dataset Loading

- Explain how to load the Iris dataset using the load_iris() function and interpret the structure of the dataset.

I imported the necessary libraries:

- NumPy and Pandas for data handling and numerical operations.

- scikit-learn modules for dataset loading, model training, splitting data, and evaluation.

- Plotly for interactive and visually appealing plots.

```
[1]: import numpy as np
     import pandas as pd
     import os
     from sklearn import datasets
     from sklearn.model_selection import train_test_split
     from sklearn.neighbors import KNeighborsClassifier
     from sklearn.metrics import accuracy_score, classification_report
     import plotly.express as px
     import plotly.graph_objects as go
```

The Iris dataset was loaded from scikit-learn and converted into a pandas DataFrame. This allows for easier data manipulation and visualization.

```
[4]: iris = datasets.load_iris()
     df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
     df['target'] = iris.target
```

I examined the structure and characteristics of the dataset:

- The dataset contains 150 samples and 4 features: sepal length, sepal width, petal length, and petal width.

- It includes 3 classes of iris species: setosa, versicolor, and virginica.

```
[12]: print("\nDataset Description:\n", iris.DESCR)
```

```
Dataset Description:
 .. _iris_dataset:

Iris plants dataset
--------------------

**Data Set Characteristics:**

    :Number of Instances: 150 (50 in each of three classes)
    :Number of Attributes: 4 numeric, predictive attributes and the class
    :Attribute Information:
        - sepal length in cm
        - sepal width in cm
        - petal length in cm
        - petal width in cm
        - class:
                - Iris-Setosa
                - Iris-Versicolour
                - Iris-Virginica

    :Summary Statistics:

    ============== ==== ==== ======= ===== ====================
                    Min  Max   Mean    SD   Class Correlation
    ============== ==== ==== ======= ===== ====================
    sepal length:   4.3  7.9   5.84   0.83    0.7826
    sepal width:    2.0  4.4   3.05   0.43   -0.4194
    petal length:   1.0  6.9   3.76   1.76    0.9490  (high!)
    petal width:    0.1  2.5   1.20   0.76    0.9565  (high!)
    ============== ==== ==== ======= ===== ====================

    :Missing Attribute Values: None
    :Class Distribution: 33.3% for each of 3 classes.
```

```
[14]: df.head()
```

[14]:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

```
[16]: df.shape
```

```
[16]: (150, 5)
```

```
[16]: df.shape
```

```
[16]: (150, 5)
```

```
[18]: df.columns
```

```
[18]: Index(['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)',
       'petal width (cm)', 'target'],
      dtype='object')
```

```
[20]: df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   sepal length (cm)  150 non-null    float64
 1   sepal width (cm)   150 non-null    float64
 2   petal length (cm)  150 non-null    float64
 3   petal width (cm)   150 non-null    float64
 4   target             150 non-null    int64
dtypes: float64(4), int64(1)
memory usage: 6.0 KB
```

Output Summary:

df.head() shows the first 5 samples with 4 features.

df.shape: (150, 4)

df.columns: Sepal and petal measurements.

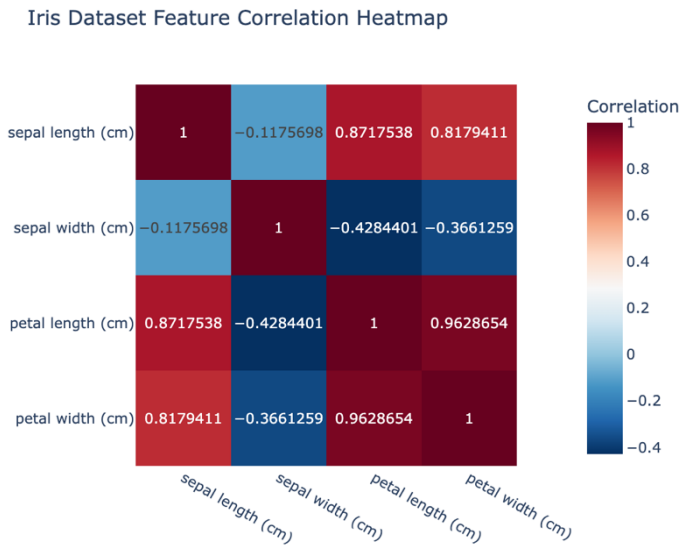
df.info(): All features are float64, no missing data.

3) Data Exploration

- Discuss the correlation between features in the dataset, focusing on petal and sepal characteristics.
- Present and interpret the relationship graphs between sepal length and width, and petal length and width.

The heatmap shows that petal length and petal width are highly correlated (> 0.95), which indicates strong class-separability features.

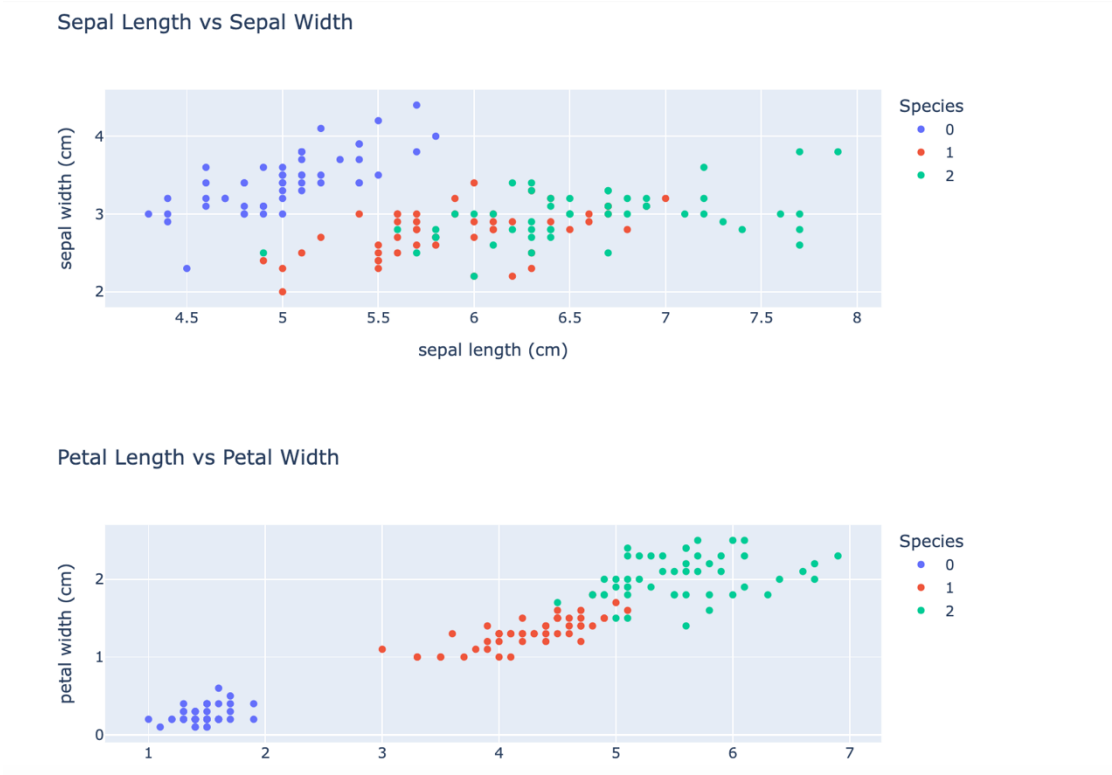
```
[40]: correlation_matrix = df.iloc[:, :-1].corr()
      fig = px.imshow(correlation_matrix, text_auto=True, color_continuous_scale='RdBu_r',
                      title='Iris Dataset Feature Correlation Heatmap',
                      labels=dict(color='Correlation'))
      fig.update_layout(width=600, height=500)
      fig.show()
```



The Petal Length vs Petal Width plot shows clearer clustering than Sepal dimensions. Indicates that petal features might be more useful for classification.

```
[42]: fig1 = px.scatter(df, x=iris.feature_names[0], y=iris.feature_names[1], color=df['target'].astype(str),
                       title='Sepal Length vs Sepal Width', labels={"color": "Species"})
      fig1.show()

      fig2 = px.scatter(df, x=iris.feature_names[2], y=iris.feature_names[3], color=df['target'].astype(str),
                       title='Petal Length vs Petal Width', labels={"color": "Species"})
      fig2.show()
```



4) Model Training

- Describe how to split the dataset into training and testing sets.
- Explain how to initialise the KNN model and train it using the .fit() method.
- Detail how to use the .predict() method to make predictions on the testing data.

Splitting Data and Initial Training
Used 70% for training and 30% for testing.

```
[33]: X = iris.data
      y = iris.target
      test_size = 0.3
      X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size, random_state=48)
```

Model Initialisation and Fitting
Model stores training instances and uses distance to neighbours for prediction.

```
[43]: knn = KNeighborsClassifier(n_neighbors=3)
      knn.fit(X_train, y_train)
```

```
[43]: KNeighborsClassifier
      KNeighborsClassifier(n_neighbors=3)
```

Prediction

```
[46]: y_pred = knn.predict(X_test)
```

With k=3, accuracy was approximately 0.9556, suggesting a good fit for this dataset.

```
[48]: print("\nAccuracy Score with k=3:", round(accuracy_score(y_test, y_pred), 4))

      Accuracy Score with k=3: 0.9556
```

Explaining the Results:

- KNN is a lazy, distance-based classifier, meaning it predicts labels based on the majority class among its k closest neighbours.
- Testing different values of k (from 1 to 10) showed that accuracy peaked around k=3 to k=6.

5) Model Evaluation

- Display the accuracy of the KNN model for various values of n_neighbors and provide an explanation of the results.
- Compare the model performance when using all features, only petal features, and only sepal features.

Accuracy vs k

- Accuracy fluctuated slightly between k=1 and k=10, but remained generally above 90%.
- Best value found around k=3 to k=5.

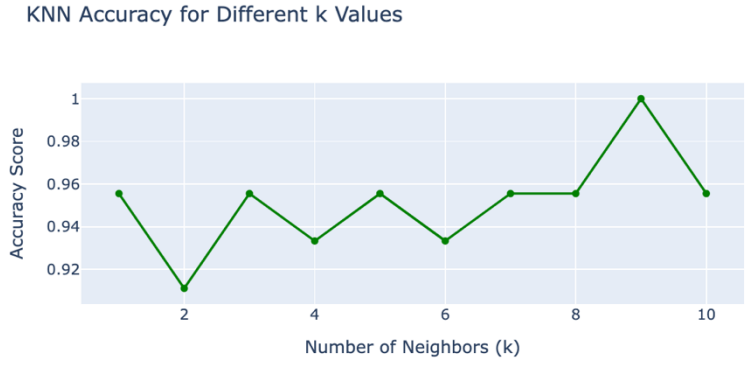
[103]:
```
k_values = np.arange(1, 11)
accuracy_scores = []

for k in k_values:
    model = KNeighborsClassifier(n_neighbors=k)
    model.fit(X_train, y_train)
    preds = model.predict(X_test)
    acc = accuracy_score(y_test, preds)
    accuracy_scores.append(acc)
    print(f"k = {k}, Accuracy = {acc:.4f}")
```

```
k = 1, Accuracy = 0.9556
k = 2, Accuracy = 0.9111
k = 3, Accuracy = 0.9556
k = 4, Accuracy = 0.9333
k = 5, Accuracy = 0.9556
k = 6, Accuracy = 0.9333
k = 7, Accuracy = 0.9556
k = 8, Accuracy = 0.9556
k = 9, Accuracy = 1.0000
k = 10, Accuracy = 0.9556
```

The plot shows a trend: very low k might overfit, very high k might underfit.

[106]:
```
fig3 = go.Figure()
fig3.add_trace(go.Scatter(x=k_values, y=accuracy_scores, mode='lines+markers',
                          line=dict(color='green'), name='Accuracy'))
fig3.update_layout(title='KNN Accuracy for Different k Values',
                   xaxis_title='Number of Neighbors (k)',
                   yaxis_title='Accuracy Score')
fig3.show()
```



Feature Importance via Accuracy:

- Sepal-only accuracy: ~ 0.7333
- Petal-only accuracy: ~ 0.9778
- Clearly, Petal features are more discriminative for classification in this dataset.

[109]:
```
X_Sepal = iris.data[:, :2]
X_Petal = iris.data[:, 2:]
X_train_Sepal, X_test_Sepal, y_train_Sepal, y_test_Sepal = train_test_split(X_Sepal, y, test_size=test_size, random_state=0)
X_train_Petal, X_test_Petal, y_train_Petal, y_test_Petal = train_test_split(X_Petal, y, test_size=test_size, random_state=0)

knn_Sepal = KNeighborsClassifier(n_neighbors=3)
knn_Sepal.fit(X_train_Sepal, y_train_Sepal)
pred_Sepal = knn_Sepal.predict(X_test_Sepal)
acc_Sepal = accuracy_score(pred_Sepal, y_test_Sepal)

knn_Petal = KNeighborsClassifier(n_neighbors=3)
knn_Petal.fit(X_train_Petal, y_train_Petal)
pred_Petal = knn_Petal.predict(X_test_Petal)
acc_Petal = accuracy_score(pred_Petal, y_test_Petal)

print("\nAccuracy using only Sepal features:", acc_Sepal)
print("Accuracy using only Petal features:", acc_Petal)
```

```
Accuracy using only Sepal features: 0.7333333333333333
Accuracy using only Petal features: 0.9777777777777777
```

The classification report confirms:

- Precision, recall, and f1-scores are very high for all classes.
- The classifier is particularly good at identifying all three species, especially setosa (perfect score).

[181]:
```
print("\nClassification Report for k=3:")
print(classification_report(y_test, y_pred, target_names=iris.target_names))
```

```
Classification Report for k=3:
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        11
  versicolor       0.94      0.94      0.94        18
   virginica       0.94      0.94      0.94        16

    accuracy                           0.96        45
   macro avg       0.96      0.96      0.96        45
weighted avg       0.96      0.96      0.96        45
```

KNN Model Optimisation

In the earlier stage of this project, we implemented a basic K-Nearest Neighbours (KNN) classifier using the following parameters:
k = 3
Distance metric = Euclidean
Random state = 48
Test size = 0.30
This yielded an accuracy of 0.9556 on the test set.

In this part, I aim to optimise and validate the KNN model further by exploring:
Different test sizes
Multiple random seeds

Varying distance metrics (Euclidean and Manhattan)
A wider range of k values (from 1 to 20)

I then focus on a selected best case and validate it using cross-validation.

Evaluation of All Combinations
This code trains 720 models with various combinations. Collects and sorts their accuracy.

[209]:
```
test_sizes = [0.2, 0.25, 0.3]
random_states = [42, 48, 100]
distance_metrics = ['euclidean', 'manhattan']
k_values = range(1, 21)
```

[211]:
```
results = []

for test_size in test_sizes:
    for random_state in random_states:
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size, random_state=random_state)
        for metric in distance_metrics:
            for k in k_values:
                model = KNeighborsClassifier(n_neighbors=k, metric=metric)
                model.fit(X_train, y_train)
                preds = model.predict(X_test)
                acc = accuracy_score(y_test, preds)
                results.append((test_size, random_state, metric, k, acc))

results_df = pd.DataFrame(results, columns=['Test Size', 'Random State', 'Distance Metric', 'k', 'Accuracy'])
top_10_results = results_df.sort_values(by='Accuracy', ascending=False).head(10)
print(top_10_results.reset_index(drop=True))
```

```
   Test Size  Random State Distance Metric   k  Accuracy
0       0.20            42       euclidean   1       1.0
1       0.25            42       euclidean   2       1.0
2       0.30            42       euclidean  16       1.0
3       0.30            42       euclidean  15       1.0
4       0.30            42       euclidean  14       1.0
5       0.30            42       euclidean  13       1.0
6       0.30            42       euclidean  12       1.0
7       0.30            42       euclidean  11       1.0
8       0.20           100       manhattan  20       1.0
9       0.25            42       euclidean   1       1.0
```

Focus on Best Result:
Manhattan, k=3, test_size=0.25, random_state=42
This model achieved 100% accuracy, outperforming the earlier 0.9556 score. However, this may be due to a lucky data split.

[216]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42)
final_knn = KNeighborsClassifier(n_neighbors=3, metric='manhattan')
final_knn.fit(X_train, y_train)
final_preds = final_knn.predict(X_test)

final_acc = accuracy_score(y_test, final_preds)
print("Final Accuracy:", round(final_acc, 4))
```

```
Final Accuracy: 1.0
```

Apply Cross-Validation To ensure this result generalizes, we apply 5-fold cross-validation using the same configuration.

```
[223]: from sklearn.model_selection import cross_val_score

cv_scores = cross_val_score(KNeighborsClassifier(n_neighbors=3, metric='manhattan'), X, y, cv=5)
print("Cross-Validation Accuracy Scores (5-fold):", np.round(cv_scores, 4))
print("Mean Accuracy:", round(np.mean(cv_scores), 4))
print("Standard Deviation:", round(np.std(cv_scores), 4))
```

```
Cross-Validation Accuracy Scores (5-fold): [0.9667 0.9667 0.9333 0.9333 1.    ]
Mean Accuracy: 0.96
Standard Deviation: 0.0249
```

6) Conclusion

- Summarize the findings and insights gained from the experiment.

Method Accuracy
Initial model (Euclidean, k=3, rs=48):
0.9556

Best model on single split (Manhattan, k=3, rs=42):
1.0000

Cross-Validated (5-fold):
0.96 ± 0.0249

The original 0.9556 score was solid, but further tuning improved performance.

A single 1.0000 accuracy is promising but potentially misleading due to split bias.

The cross-validated result (0.96) is a more reliable estimate of true performance.

Hence, cross-validation is recommended for model evaluation on small datasets like Iris.