UO Predictive Analytics

kNN Hands-on Task A: Iris Flower Classification

This notebook demonstrates the development of a kNN classifier for predicting Iris flower species.

Learning Objectives:

- Understand the k-Nearest Neighbors (kNN) algorithm.
- Learn how to preprocess and split a dataset.
- Train and evaluate a kNN model.
- Optimize k by testing multiple values.

> 💡 **Tip:** Always start by exploring the dataset before training a model!

Step 1: Import Required Libraries

We import the necessary libraries for data manipulation, model training, and evaluation.

In [3]:
```
# Import required libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt  # For visualization
import seaborn as sns # For visualization
```

Step 2: Load the Iris Dataset

The **Iris dataset** is a widely used dataset in machine learning and statistics, introduced by **Ronald Fisher** in 1936. It consists of **150 observations**, representing **three species** of iris flowers. A summary of the dataset is shown below:

Category	Details
Species	Setosa (Class 0), Versicolor (Class 1), Virginica (Class 2)
Features	Sepal Length, Sepal Width, Petal Length, Petal Width (in cm)

Figure 1: *Iris virginica*, *versicolor*, and *setosa*. The *virginica* and *versicolor* pictures are by courtesy of the Missouri Botanical Garden and the *setosa* picture is by Tiia Monto.

Image taken from Antony Unwin, Kim Kleinman, The Iris Data Set: In Search of the Source of Virginica, Significance, Volume 18, Issue 6, December 2021, Pages 26–29, DOI:10.1111/1740-9713.01589.

We load the dataset directly from `sklearn.datasets` to ensure consistency across runs.

> 💡 **Tip:** If you are using a CSV file, make sure the path is correct before loading it.

In [7]:
```
# Load dataset from sklearn
from sklearn.datasets import load_iris
iris = load_iris()

# Convert to pandas DataFrame
df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
df['species'] = iris.target  # Add species column

# Mapping numerical target to class names
species_map = {0: 'Setosa', 1: 'Versicolor', 2: 'Virginica'}
df['species_name'] = df['species'].map(species_map)

# Display first few rows
df.head()
```

Out[7]:

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

Step 3: Data Exploration

Before training a model, we should inspect the dataset:

- Check the shape (number of rows and columns).
- Check for missing values.
- Verify class distribution.

> 💡 **Tip:** Class imbalance can negatively affect model performance. Always check class distributions!

In [9]:
```
# Print dataset shape
print('Dataset Shape:', df.shape)

# Print dataset info
df.info()

# Check class distribution
df['species'].value_counts()
```

```
Dataset Shape: (150, 6)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 6 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   species            150 non-null    int32
 5   species_name       150 non-null    object
dtypes: float64(4), int32(1), object(1)
memory usage: 6.6+ KB
```

Out[9]:

```
 species
 0    50
 1    50
 2    50
 Name: count, dtype: int64
```

From the **Non-Null Count**, we can see that there are **no missing values** in the dataset. Missing values can adversely affect machine learning models. Later in the course, we will explore their impact and how to handle them effectively.

From the **Dtype (Data Type)** information, we observe that the data types are correctly formatted. We must ensure that data is properly identified for accurate analysis and modelling. If any data is in the wrong format, **we must convert it** to the appropriate type before proceeding.

From the **class** count we check whether we see that we have an **equal number of samples** for each target class. A dataset where the samples are **evenly distributed** across target classes is called a **balanced dataset**.

Imbalanced datasets are not ideal for training models, as they can lead to poor performance on underrepresented classes.
When a model is trained on an imbalanced dataset, it may struggle to generalise to minority class samples.

We will explore **imbalanced datasets** and techniques to handle them later in the course.

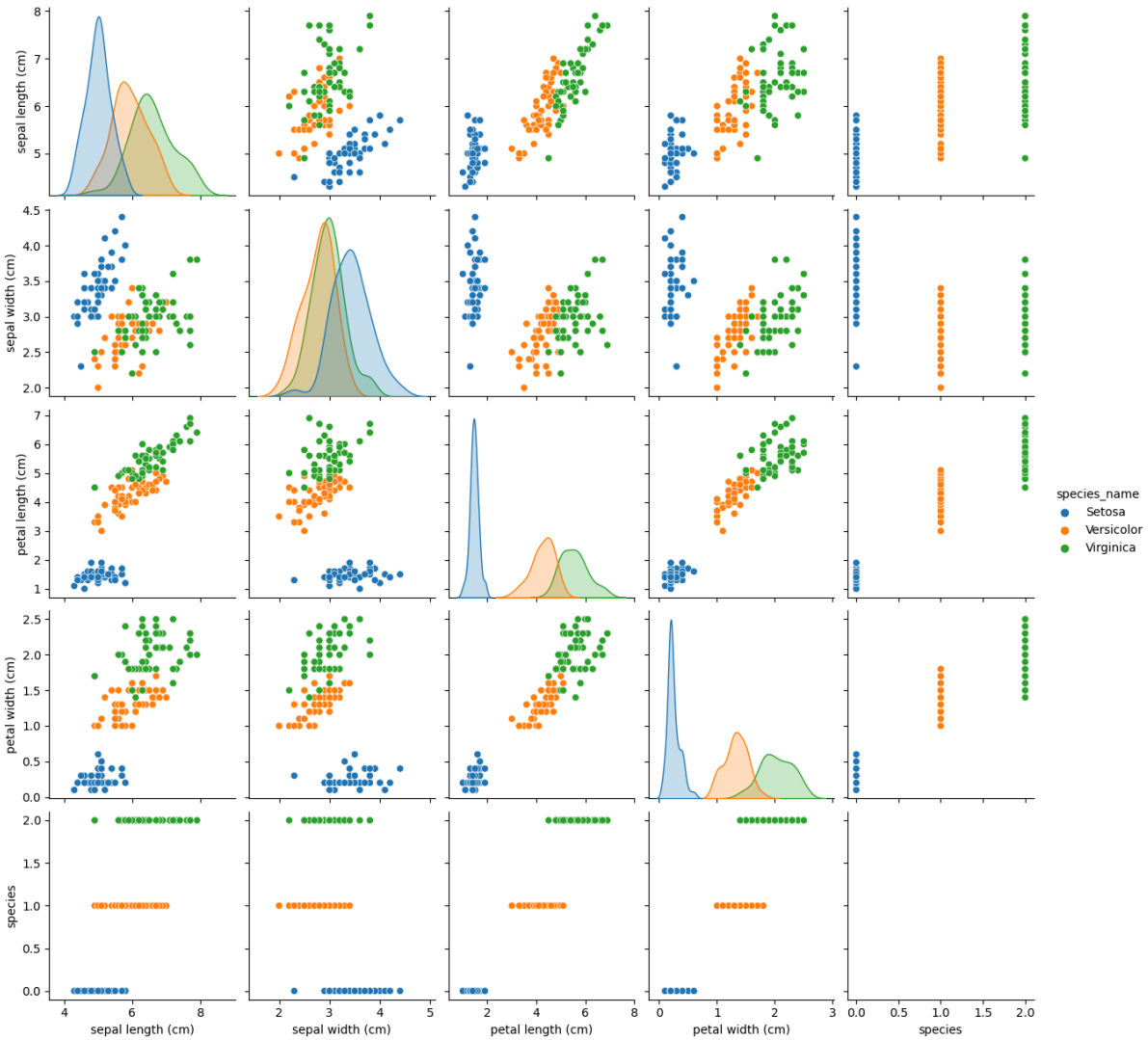
To further verify class distribution you can:

- Plot scatterplots of features to visualize class separation.
- Use pair plots or histograms to observe feature distributions.

In [12]:

```
# Plot Pairplot for feature visualization
sns.pairplot(df, hue="species_name", diag_kind="kde")
plt.show()
```

```
C:\Users\cifuenam\AppData\Local\anaconda3\Lib\site-packages\seaborn\axisgrid.py:118: UserWar
ning: The figure layout has changed to tight
  self._figure.tight_layout(*args, **kwargs)
```



Step 4: Split Dataset into Training and Test Sets

We use a **75/25 split** to ensure the model has enough training data. We also apply **stratified sampling** to maintain class balance in both sets.

Tip: Setting `random_state` ensures reproducibility across different runs.

In [14]:
```
# Define feature names
features = iris.feature_names

# Split dataset into training and test sets (stratified)
X_train, X_test, y_train, y_test = train_test_split(
    df[features], df['species'], test_size=0.25, random_state=42, stratify=df['species']
)

# Check test set class distribution
y_test.value_counts()
```

Out[14]:
```
species
1    13
2    13
0    12
Name: count, dtype: int64
```

The `train_test_split` function in **scikit-learn** is used to **randomly split a dataset** into **training and testing sets**.

This allows our machine learning models to **generalise well to unseen data**. You can find further info of this function at:

👉 **Official Documentation**

Step 5: Train kNN Model

The `KNeighborsClassifier` in **scikit-learn** is a simple yet powerful **machine learning algorithm** used for **classification tasks**.

- It classifies a new data point by looking at the **k nearest neighbours** in the dataset.
- The class with the most votes among the neighbours is the predicted class.
- Works well for small to medium-sized datasets, especially when patterns are **based on proximity**.

We initialize a kNN classifier with `k=5` and Euclidean distance as the metric.

> 💡 **Tip:** The choice of `k` affects model performance. A small `k` can make the model sensitive to noise, while a large `k` may oversimplify patterns.

Check more at: 👉**k-Nearest Neighbours (k-NN) Classification – Official Documentation**

In [17]:
```
# Initialize kNN classifier with k=5
k = 5
knn = KNeighborsClassifier(n_neighbors=k, metric='euclidean')

# Train the model on the training set
knn.fit(X_train, y_train)
```

Out[17]:

KNeighborsClassifier

```
KNeighborsClassifier(metric='euclidean')
```

Step 6: Model Evaluation

We test the model on the test set and calculate accuracy.

> 💡 **Tip:** Accuracy alone is not always a reliable metric, especially for imbalanced datasets.

In [19]:
```
# Predict the test set
y_pred = knn.predict(X_test)

# Calculate and display accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')
```

```
Accuracy: 0.97
```

Step 7: Finding the Optimal k

Instead of assuming `k=5` is the best choice, we systematically test different values of `k` to observe how accuracy changes.

In [21]:
```python
# Test different k values
lst_k = []
lst_acc = []

for k in range(3, 25, 2):  # Test odd values of k from 3 to 23
    knn = KNeighborsClassifier(n_neighbors=k, metric='euclidean')
    knn.fit(X_train, y_train)
    acc = accuracy_score(y_test, knn.predict(X_test))
    lst_k.append(k)
    lst_acc.append(acc)
    print(f'k = {k}, accuracy = {acc:.2f}')
```

```
k = 3, accuracy = 0.97
k = 5, accuracy = 0.97
k = 7, accuracy = 0.95
k = 9, accuracy = 0.97
k = 11, accuracy = 0.97
k = 13, accuracy = 0.95
k = 15, accuracy = 0.95
k = 17, accuracy = 0.95
k = 19, accuracy = 0.95
k = 21, accuracy = 0.95
k = 23, accuracy = 0.95
```

Step 8: Plot Accuracy vs. k Values

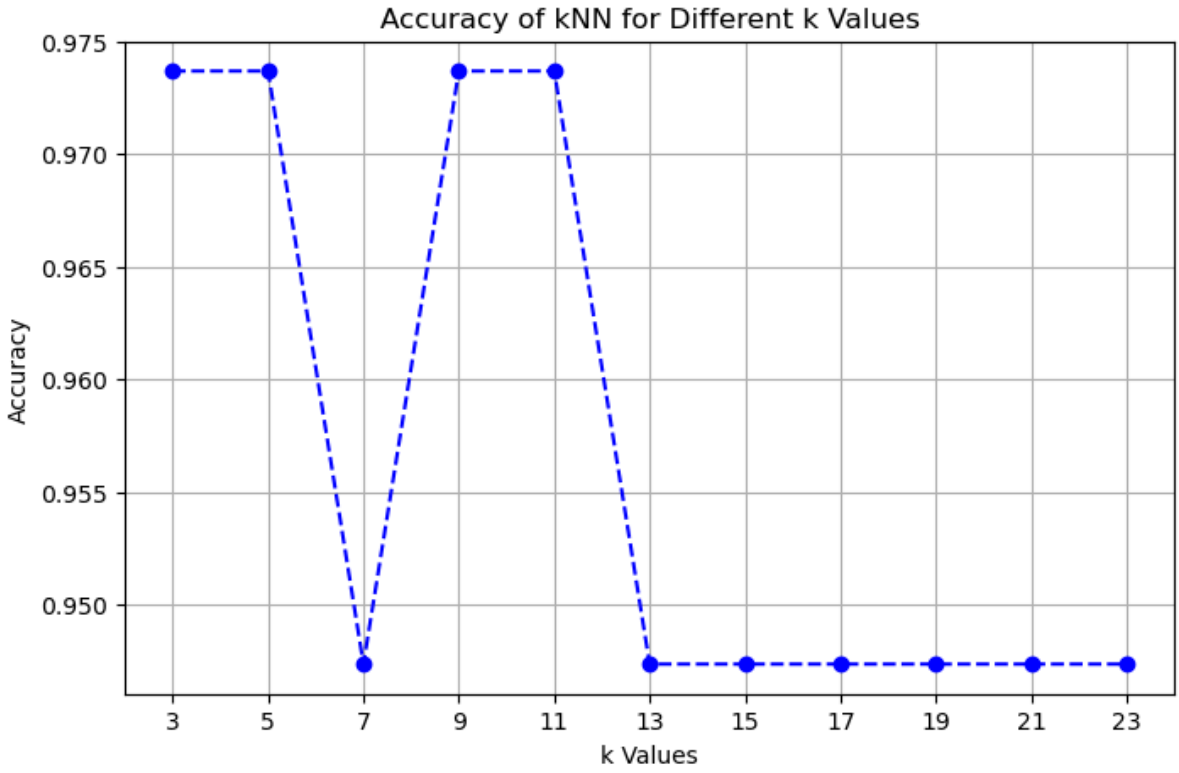
We visualize the accuracy for different values of `k` to find the optimal choice.

> 💡 **Tip:** Look for the `k` value where accuracy stabilizes or peaks.

In [23]:
```python
# Plot accuracy vs. k values
plt.figure(figsize=(8,5))
plt.plot(lst_k, lst_acc, marker='o', linestyle='--', color='b')

# Ensure x-axis only shows integer values as k can only be integers
plt.xticks(lst_k)  # Explicitly set x-ticks to k values

plt.xlabel('k Values')
plt.ylabel('Accuracy')
plt.title('Accuracy of kNN for Different k Values')
plt.grid()
plt.show()
```



The above graph shows that the best model (based on accuracy) is achieved at **k = 3**.
(You may get a different value when you run this yourself due to a different random split.)

Summary & Next Steps

Key Takeaways:

- We used the **Iris dataset** to train a kNN classifier.
- We split the data into **training (75%)** and **test (25%)** sets using **stratified sampling**.
- We searched for the **optimal k value** using accuracy trends.

Next Steps:

- Apply kNN to a different dataset.
- Experiment with **feature scaling** to see how it affects distance calculations.
- Try different **distance metrics** like Manhattan or Minkowski distance.