Lab8_Scikit-Learn_Classification_Solutions

May 29, 2025

1 Lab 8: Classification with Scikit-Learn

The objective of this notebook is to learn about the **Scikit-Learn** library (official documentation) and **classification models**.

Firstly, run the next cell to import useful libraries to complete this lab.

[1]:
```
import pandas as pd
import numpy as np
from sklearn.datasets import load_iris
from sklearn.svm import SVC
from sklearn.inspection import DecisionBoundaryDisplay
from sklearn import tree, neighbors
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, classification_report,␣
 ↪accuracy_score
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import f1_score

import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings("ignore")
```

1.1 1. Load dataset

We will use the **Iris** dataset as our first classification problem. *Iris* is a genus of flowering plants that contains several species, including **Iris setosa**, **Iris versicolor**, and **Iris virginica**.

The dataset consists of: - 150 samples - 3 labels: species of Iris (*Iris setosa*, *Iris versicolor*, and *Iris virginica*) - 4 features: *Sepal length*, *Sepal width*, *Petal length*, and *Petal width* in centimeters

Your objective is to build a **multiclass classifier** that can predict the target class (i.e., the species of Iris) given the feature values (i.e., Sepal length, Sepal width, Petal length, and Petal width).

You can find an **exploratory analysis of the dataset** here.

Scikit-Learn comes with built-in datasets for the **Iris** classification problem. The next cell loads the iris dataset from Scikit-Learn and stores it in a Pandas DataFrame.

[2]:
```python
iris = load_iris() # Load Data

df = pd.DataFrame(iris.data, columns = iris.feature_names) # Create a dataframe
df['target'] = iris.target
df['target name'] = df['target'].apply(lambda x: 'sentosa' if x == 0 else␣
 ↪('versicolor' if x == 1 else 'virginica'))
```

[3]:
```python
df
```

[3]:
```
     sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  \
0                  5.1               3.5                1.4               0.2
1                  4.9               3.0                1.4               0.2
2                  4.7               3.2                1.3               0.2
3                  4.6               3.1                1.5               0.2
4                  5.0               3.6                1.4               0.2
..                 …                …                 …                …
145                6.7               3.0                5.2               2.3
146                6.3               2.5                5.0               1.9
147                6.5               3.0                5.2               2.0
148                6.2               3.4                5.4               2.3
149                5.9               3.0                5.1               1.8

     target target name
0         0     sentosa
1         0     sentosa
2         0     sentosa
3         0     sentosa
4         0     sentosa
..      …         …
145       2   virginica
146       2   virginica
147       2   virginica
148       2   virginica
149       2   virginica

[150 rows x 6 columns]
```

[4]:
```python
n_labels = len(set(df['target']))
print(f'Number of labels: {n_labels}')
print(f"labels: {set(df['target name'])}")
```

```
Number of labels: 3
labels: {'versicolor', 'virginica', 'sentosa'}
```

[5]:
```python
labels = ["sentosa", "versicolor", "virginica"]
label2id = {"sentosa":0, "versicolor":1, "virginica":2}
```

The following cell describes the dataset by computing the *mean*, *std*, *min*, *max*, and *quantiles* of each column.

[6]:
```
df.describe()
```

[6]:
```
       sepal length (cm)  sepal width (cm)  petal length (cm)  \
count         150.000000        150.000000         150.000000
mean            5.843333          3.057333           3.758000
std             0.828066          0.435866           1.765298
min             4.300000          2.000000           1.000000
25%             5.100000          2.800000           1.600000
50%             5.800000          3.000000           4.350000
75%             6.400000          3.300000           5.100000
max             7.900000          4.400000           6.900000

       petal width (cm)      target
count        150.000000  150.000000
mean           1.199333    1.000000
std            0.762238    0.819232
min            0.100000    0.000000
25%            0.300000    0.000000
50%            1.300000    1.000000
75%            1.800000    2.000000
max            2.500000    2.000000
```

The following cell counts the number of *null* values in each column.

[7]:
```
nan_count = df.isna().sum()
print(nan_count )
```

```
sepal length (cm)    0
sepal width (cm)     0
petal length (cm)    0
petal width (cm)     0
target               0
target name          0
dtype: int64
```

As you can see, there are no *null* values (i.e., **missing values**) in the dataset.

The next cell **counts the number of examples for each class label**.

[8]:
```
df.value_counts("target")
```

[8]:
```
target
0    50
1    50
2    50
dtype: int64
```

As you can see, the labels are **equally represented** in the dataset. Therefore, it is a **balanced** dataset.

1.2 2. Classification with 2D input features

Now, you will perform the classification task (i.e., predict the species of Iris) using the first **two input features** of the dataset (i.e., *sepal length* and *sepal width*).

Exercise 2.1 Select the **first two columns** of the dataset and store them in a variable `X_2d`.

[9]:
```
y = df.target
y_names = df["target name"]

#### START CODE HERE (~1 line) ####
X_2d = df.iloc[:, :2]
#### END CODE HERE ####
```

[10]:
```
X_2d.head()
```

[10]:
```
   sepal length (cm)  sepal width (cm)
0                5.1               3.5
1                4.9               3.0
2                4.7               3.2
3                4.6               3.1
4                5.0               3.6
```

Expected output sepal length (cm) sepal width (cm) 0 5.1 3.5 1 4.9 3.0 2 4.7 3.2 3 4.6 3.1 4 5.0 3.6

[11]:
```
y
```

[11]:
```
0      0
1      0
2      0
3      0
4      0
      ..
145    2
146    2
147    2
148    2
149    2
Name: target, Length: 150, dtype: int64
```

[12]:
```
y_names
```

```
[12]: 0        sentosa
1        sentosa
2        sentosa
3        sentosa
4        sentosa
           …
145    virginica
146    virginica
147    virginica
148    virginica
149    virginica
Name: target name, Length: 150, dtype: object
```

Exercise 2.2 Split the dataset `X_2d` into **training** and **test** sets using the train_test_split function provided by the Scikit-Learn library. Store the features of the training set in `X_train_2d`, the features of the test `X_test_2d`, the labels of the training set in `y_train`, and the labels of the test set in `y_test`. Split the dataset with 80% of samples for training and 20% of samples for testing. **Shuffle** the data and set the **random state** to 42.

```
[13]: #### START CODE HERE (~1 line) ####
X_train_2d, X_test_2d, y_train, y_test = train_test_split(X_2d, y, test_size=0.
 ↪2, shuffle=True, random_state=42)
#### END CODE HERE ####
```

```
[14]: print(f"{len(X_train_2d)} training examples")
print(f"{len(X_test_2d)} test examples")
```

```
120 training examples
30 test examples
```

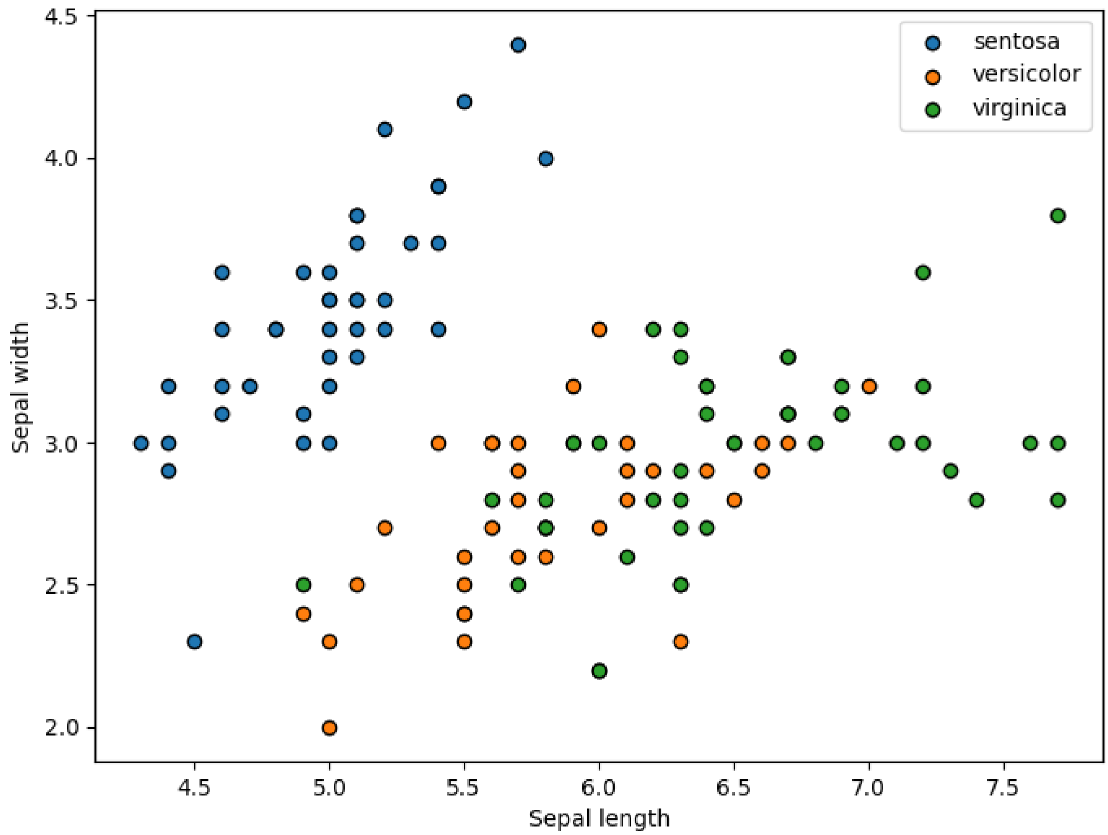
Expected output `120 training examples 30 test examples`

The following cell **plots** the examples of the **training set** in the **plane**, with a different color based on the target label.

```
[15]: plt.figure(2, figsize=(8, 6))
plt.clf()

for label_id, label in enumerate(labels):
    X_temp = X_train_2d.loc[y_train == label_id]
    plt.scatter(X_temp.iloc[:, 0], X_temp.iloc[:, 1], cmap=plt.cm.Set1,␣
 ↪edgecolor="k", label=label)

plt.xlabel("Sepal length")
plt.ylabel("Sepal width")

plt.xticks()
```

```
plt.yticks()
plt.legend()
```

[15]: <matplotlib.legend.Legend at 0x7fca6352fa30>



1.2.1 Train a Support Vector Machine SVM classifier

Here, you will train a **Support Vector Machine SVM** Classifier using the *Scikit-Learn* library. You can learn more about **Support Vector Machines** here. You can find the official Scikit-Learn documentation for **SVM** for classification here. For this exercise, you can use the **SVC** implementation here.

Exercise 2.3 Create an **SVC object** with the following parameters `gamma=0.1`, `kernel="rbf"`, `probability=True` in a variable `svm_model`.

Feel free to change the parameters and see how it affects the results.

[16]:
```
#### START CODE HERE (~1 line) ####
svm_model = SVC(gamma=0.1, kernel="rbf", probability=True)
#### END CODE HERE ####
```

Exercise 2.4 **Fit** (i.e., train) the `svm_model` with the training data. You should pass the input features and the targets of the training set. Please refer to the documentation for the parameters of the **fit()** method.

[17]:
```
#### START CODE HERE (~1 line) ####
svm_model.fit(X_train_2d, y_train)
#### END CODE HERE ####
```

[17]:
```
SVC(gamma=0.1, probability=True)
```

Exercise 2.5 **Predict** the labels for the **test dataset** and store them in a variable `y_test_pred_svm`. You should pass the input features of the test data. Please refer to the documentation for the parameters of the **predict()** method.

[18]:
```
#### START CODE HERE (~1 line) ####
y_test_pred_svm = svm_model.predict(X_test_2d)
#### END CODE HERE ####
```

1.2.2 Confusion Matrix

Exercise 2.6 Compute the **confusion matrix** for the predictions on the test set in a variable `cm`. You should pass the **real labels** (i.e., ground-truth labels) and the **predicted labels** by the classifier. Use the `confusion_matrix` function of the Scikit-Learn library. You can find the documentation here.

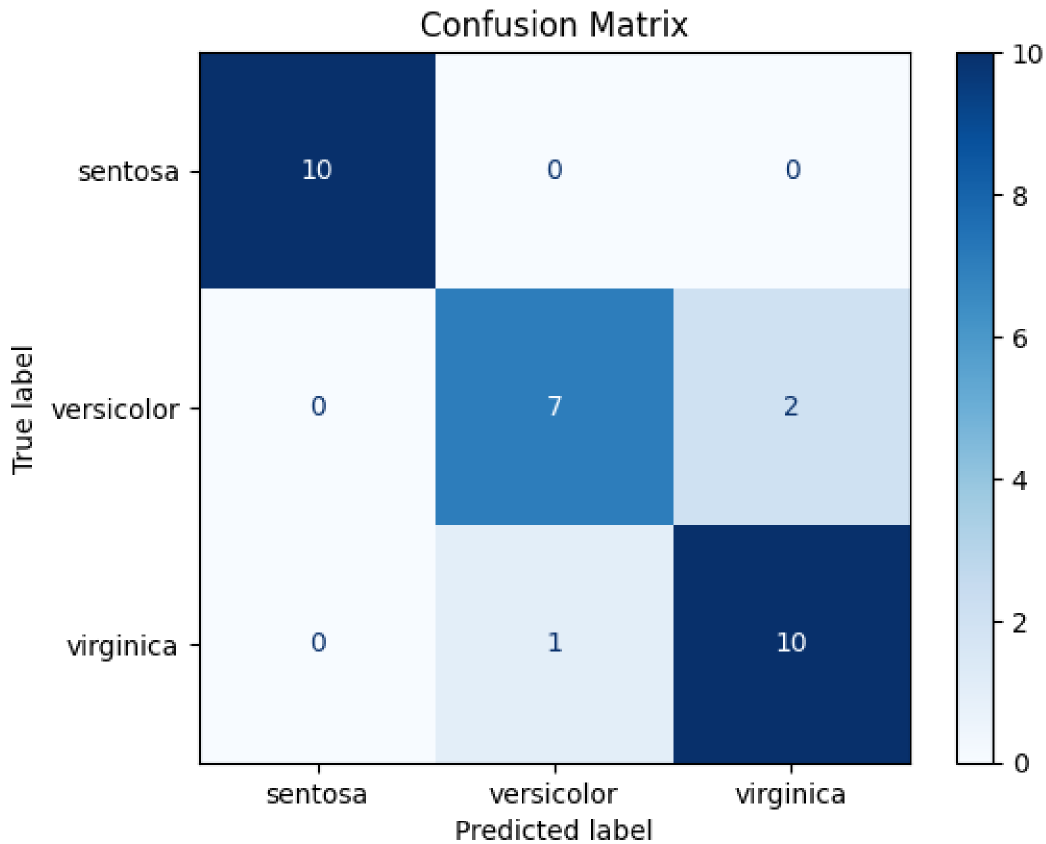
You can learn how to interpret a **confusion matrix** here.

[19]:
```
#### START CODE HERE (~1 line) ####
cm = confusion_matrix(y_test, y_test_pred_svm)
#### END CODE HERE ####
print(cm)
```

```
[[10  0  0]
 [ 0  7  2]
 [ 0  1 10]]
```

The following cell **plots the confusion matrix**.

[20]:
```
from sklearn.metrics import ConfusionMatrixDisplay

cmd = ConfusionMatrixDisplay.from_predictions(y_test, y_test_pred_svm, cmap=plt.
 ↪cm.Blues)
ax = cmd.ax_
ax.set_title('Confusion Matrix')
ax.set_xticklabels(labels)
ax.set_yticklabels(labels)
plt.show()
```



1.2.3 Accuracy and F1 Score

Exercise 2.7 Compute the **accuracy** and the **F1 score** for the predictions on the **test set**, and store the results in the variables `acc_svm` and `f1_svm`, respectively. To compute the **accuracy**, you can use the accuracy_score function of the Scikit-Learn library. Instead, to compute the **F1 score**, you can use the f1_score function of the Scikit-Learn library. For the **F1**, compute the `macro` score (you can specify it in the parameters).

[21]:
```
#### START CODE HERE (~2 lines) ####
acc_svm = accuracy_score(y_test, y_test_pred_svm)
f1_svm = f1_score(y_test, y_test_pred_svm, average='macro')
#### END CODE HERE ####
```

[22]:
```
print(f"Accuracy: {acc_svm:.2}")
print(f"F1: {f1_svm:.2}")
```

```
Accuracy: 0.9
F1: 0.9
```

Congratulations. You have trained a very good classifier! It predicts the correct class 9 times out of 10!

1.2.4 Train a Decision Tree Classifier

Here, you will train a **Decision Tree DT** Classifier using the *Scikit-Learn* library. You can learn more about **Decision Trees** here. You can find the official Scikit-Learn documentation for **Decision Tree** here. For this exercise, you should use the **DT Classifier** here.

Exercise 2.6

- Create an `DecisionTreeClassifier` object with the following parameters `max_depth=4` in a variable `dt_model`.
- Fit (i.e., train) the `dt_model` with the training data. You should pass the input features and the targets. Please refer to the documentation.
- Predict the labels for the test dataset and store them in a variable `y_test_pred_dt`. You should pass the input features of the test data.

Feel free to change the parameters and see how it affects the results.

[23]:
```
#### START CODE HERE (~3 lines) ####
dt_model = DecisionTreeClassifier(max_depth=4)
dt_model.fit(X_train_2d, y_train)
y_test_pred_dt = dt_model.predict(X_test_2d)
#### END CODE HERE ####
```

1.2.5 Train a K-Nearest-Neighbors Classifier

Here, you will train a **K-Nearest-Neighbors** Classifier using the *Scikit-Learn* library. You can learn more about **K-Nearest-Neighbors** here. You can find the official Scikit-Learn documentation for **K-Nearest-Neighbors** here. For this exercise, you should use the **KNeighborsClassifier** here.

Exercise 2.7

- Create an `KNeighborsClassifier` object with the following parameters `n_neighbors=7` in a variable `knn_model`.
- Fit (i.e., train) the `knn_model` with the training data. You should pass the input features and the targets. Please refer to the documentation.
- Predict the labels for the test dataset and store them in a variable `y_test_pred_knn`. You should pass the input features of the test data.

Feel free to change the parameters and see how it affects the results.

[24]:
```
#### START CODE HERE (~3 lines) ####
knn_model = KNeighborsClassifier(n_neighbors=7)
knn_model.fit(X_train_2d, y_train)
y_test_pred_knn = knn_model.predict(X_test_2d)
#### END CODE HERE ####
```

1.2.6 Train a Random Forest Classifier

Here, you will train a **Random Forest** Classifier using the *Scikit-Learn* library. You can learn more about **Random Forests** here. You can find the official Scikit-Learn documentation for **Random Forest Classifiers** here. For this exercise, you should use the **Random Forest Classifier** here.

Exercise 2.8

- Create an `RandomForestClassifier` object with the following parameters `max_depth=2` in a variable `rf_model`.
- Fit (i.e., train) the `rf_model` with the training data. You should pass the input features and the targets. Please refer to the documentation.
- Predict the labels for the test dataset and store them in a variable `y_test_pred_rf`. You should pass the input features of the test data.

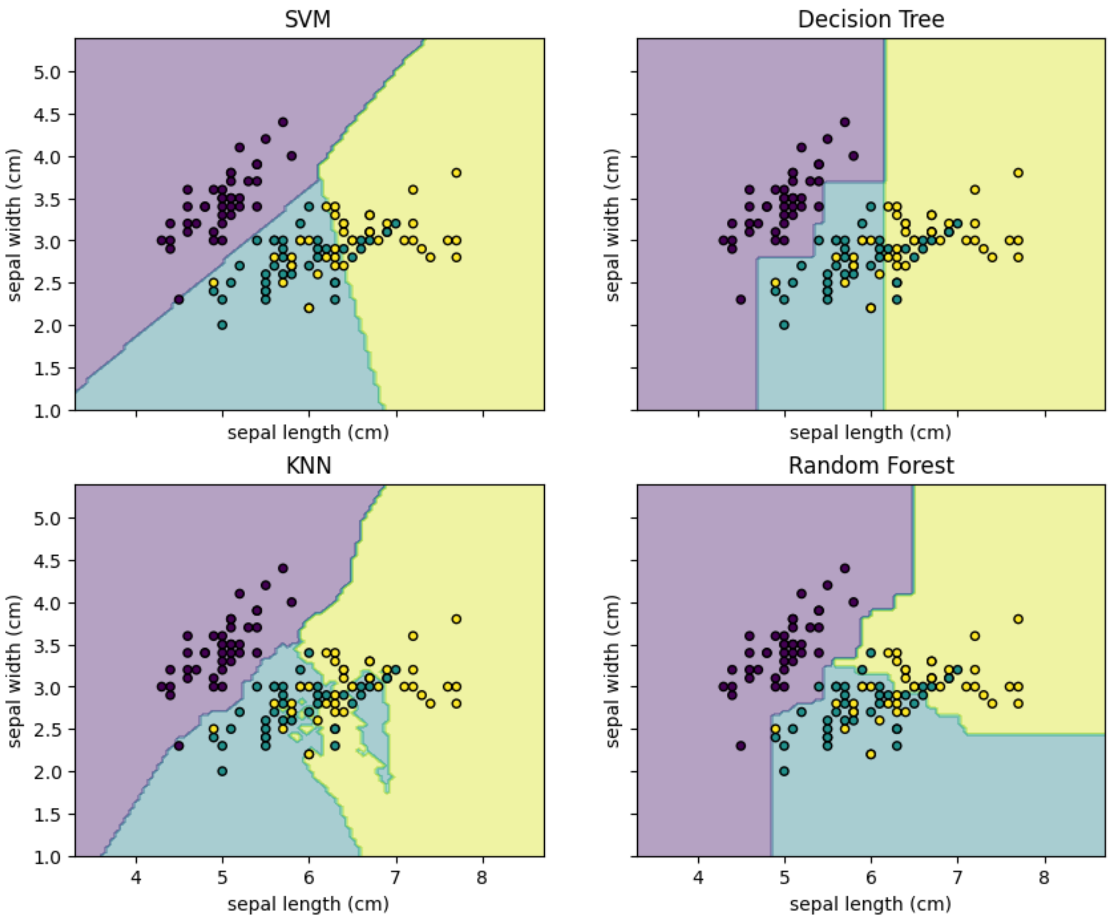
Feel free to change the parameters and see how it affects the results.

[25]:
```
#### START CODE HERE (~3 lines) ####
rf_model = RandomForestClassifier(max_depth=2)
rf_model.fit(X_train_2d, y_train)
y_test_pred_rf = rf_model.predict(X_test_2d)
#### END CODE HERE ####
```

1.2.7 Plot Decision Boundaries

The next cell plots the **decision boundaries** for the SVM, the Decision Tree, the KNN, and the Random Forest. To run this cell, ensure that you correctly named the variables `svm_model`, `dt_model`, `knn_model`, and `rf_model`. You can learn more about **decision boundaries** here.

[26]:
```
# Plotting decision regions
from itertools import product

f, axarr = plt.subplots(2, 2, sharex="col", sharey="row", figsize=(10, 8))
for idx, clf, tt in zip(product([0, 1], [0, 1]),
                        [svm_model, dt_model, knn_model, rf_model],
                        ["SVM", "Decision Tree", "KNN", "Random Forest"]):
    DecisionBoundaryDisplay.from_estimator(
        clf, X_train_2d, alpha=0.4, ax=axarr[idx[0], idx[1]],␣
 ↪response_method="predict"
    )
    axarr[idx[0], idx[1]].scatter(X_train_2d.iloc[:, 0], X_train_2d.iloc[:, 1],␣
 ↪c=y_train, s=20, edgecolor="k")
    axarr[idx[0], idx[1]].set_title(tt)

plt.show()
```



1.2.8 Compare the Classifiers with Quantitative Evaluation Metrics

So far, you have trained 4 different classifiers on the same training data. To assess which performs better, you will calculate **quantitative evaluation** metrics such as **F1**, **Precision**, and **Recall**. Metrics will be calculated either separately for each class or aggregated as a whole.

You can learn more about such quantitative metrics here and here.

Exercise 2.9 Compute the **quantitative metrics** for the **SVM** model in a variable `classification_report_svm`, for the **Decision Tree** in a variable `classification_report_dt`, for the **KNN** in a variable `classification_report_knn`, and for the **Random Forest** in a variable `classification_report_rf`. Use the classification_report function of the Scikit-Learn library. It computes all the metrics for each class and overal at once. Remember that the names of your target labels are stored in the variable `labels`.

[27]:
```
print(labels)
```

```
['sentosa', 'versicolor', 'virginica']
```

```
[28]: #### START CODE HERE (~4 lines) ####
classification_report_svm = classification_report(y_test, y_test_pred_svm,␣
 ↪target_names=labels)
classification_report_dt = classification_report(y_test, y_test_pred_dt,␣
 ↪target_names=labels)
classification_report_knn = classification_report(y_test, y_test_pred_knn,␣
 ↪target_names=labels)
classification_report_rf = classification_report(y_test, y_test_pred_rf,␣
 ↪target_names=labels)
#### END CODE HERE ####
```

```
[29]: print("SVM")
print(classification_report_svm)

print("\n\nDecision Tree")
print(classification_report_dt)

print("\n\nKNN")
print(classification_report_knn)

print("\n\nRandom Forest")
print(classification_report_rf)
```

```
SVM
              precision    recall  f1-score   support

     sentosa       1.00      1.00      1.00        10
  versicolor       0.88      0.78      0.82         9
   virginica       0.83      0.91      0.87        11

    accuracy                           0.90        30
   macro avg       0.90      0.90      0.90        30
weighted avg       0.90      0.90      0.90        30



Decision Tree
              precision    recall  f1-score   support

     sentosa       1.00      0.90      0.95        10
  versicolor       0.75      0.67      0.71         9
   virginica       0.77      0.91      0.83        11

    accuracy                           0.83        30
   macro avg       0.84      0.83      0.83        30
weighted avg       0.84      0.83      0.83        30
```

```
KNN
              precision    recall  f1-score   support

     sentosa       1.00      1.00      1.00        10
  versicolor       0.60      0.67      0.63         9
   virginica       0.70      0.64      0.67        11

    accuracy                           0.77        30
   macro avg       0.77      0.77      0.77        30
weighted avg       0.77      0.77      0.77        30



Random Forest
              precision    recall  f1-score   support

     sentosa       1.00      1.00      1.00        10
  versicolor       0.64      0.78      0.70         9
   virginica       0.78      0.64      0.70        11

    accuracy                           0.80        30
   macro avg       0.80      0.80      0.80        30
weighted avg       0.81      0.80      0.80        30
```

What do you think is the best classifier? Why?

1.3 3. Classification with all features

Now you will perform the same procedure but using **all the features** in the dataset. Remember that the original dataset contains 4 features but in the previous exercise you used only 2 features.

Exercise 3.1 Select **all the feature columns** of the dataset and store them in a variable `X`. The features are stored in the first 4 columns of the DataFrame `df` (i.e., *sepal length (cm)*, *sepal width (cm)*, *petal length (cm)*, and *petal width (cm)*).

[30]:
```
y = df.target
y_names = df["target name"]

#### START CODE HERE (~1 line) ####
X = df.iloc[:, :4]
#### END CODE HERE ####
```

[31]:
```
X.head()
```

[31]:

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

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

This time the input array have 4 features. Therefore you can't visualize it in the plane.

The following cell **splits the dataset into train and test**.

[32]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪shuffle=True)
```

[33]:
```
print(f"Number of training examples {len(X_train)}")
print(f"Number of test examples {len(X_test)}")
```

```
Number of training examples 120
Number of test examples 30
```

Exercise 3.2 Now **train different classifiers using all input features** `X`. You can also use other classifiers in the *Scikit-Learn* library and different hyperparameters. Can you outperform the best model obtained using only 2 input features?

You can find the list of all implemented classification models here.

Remember that the steps are always the same: 1. **Instantiate the model object** you want to use. 2. **Train the model** on the training data using the **fit()** method. 3. **Predict labels for test data** using the **predict()** method. 4. Repeat training and testing **for different models** (and also different hyperparameters of the models). 5. Compute **quantitative evaluation metrics** to identify the best model.

[ ]:
```
```