Exercise 1 – Environment preparation

Step 1: check if python is already installed on your computer, otherwise download it.

```
$ python3 --version
```

If no, install python from here.
If yes, upgrade python to the latest version.

```
$ sudo apt update -y
$ sudo apt install python3.9
```

Step 2: install or upgrade pip

Pip is a package management system which we will use to install and manage Python software packages. Normally, pip is already installed with python.

Follow these instructions to check if that is the case. Also, if you have it already installed, make sure to use the latest version by upgrading it.

Step 3: install libraries which we will use in our projects

```
$ pip3 install ipython
$ pip3 install jupyter
```

Jupyter and ipython will allow us to have a user-friendly, interactive interface to write python code. Read more about it here.

```
$ pip3 install pandas
$ pip3 install scikit-learn
```

Pandas is a software library for data manipulation and analysis in python.
Sklearn (scikit-learn) is a software library containing machine learning algorithms, written for python.

Exercise 2 - Decision trees

In the following example, we will use the iris flower data set, which contains 150 instances belonging to 3 classes: setosa (class id=0), versicolor (class id=1), virginica (class id=2).

Each instance has 4 features: sepal length (cm), sepal width (cm), petal length (cm), petal width (cm).

Step 0: create a Jupyter notebook

```
$ jupyter notebook
```

This will open a new tab in your web browser that shows the Notebook Dashboard, used for managing your Jupyter notebooks.

In order to create a new notebook, choose New in the upper right corner, then Python3. Change the title from “Untitled” to your choice.

Next, follow this beginner’s tutorial about Jupyter starting with the section “Creating Your First Notebook” .

We will solve the following exercises in a notebook (every time you are required to write code, select a code cell). Also, you will most likely prepare your course projects in this kind of environment as it allows both code and descriptions. (it’s not an obligation, but highly advisable).

Step 1: import libraries.

Exercise 1 - Decision trees

Step 1: import libraries

```
In [3]: import pandas as pd
        from sklearn.datasets import load_iris
        from sklearn.tree import DecisionTreeClassifier
```

Usually, data scientists do not code machine learning algorithms from scratch, but re-use existing implementations available in different libraries. This is the case in our exercise today when we will use the DecisionTreeClassifier from scikit-learn. Additionally, the iris dataset is also available in the scikit-learn library.

Step 2: read the dataset, prepare the features and classes data frames.

Step 2: read the dataset, prepare the features and classes data frames

```
In [5]: iris = load_iris()
        X = pd.DataFrame(iris.data[:, :], columns = iris.feature_names[:])
        y = pd.DataFrame(iris.target, columns =["Species"])
```

A data frame resembles a table. These objects allow us to manipulate data in an easier manner. (if you are familiar with SQL or Excel / Spreadsheets tables, then a data frame is a similar abstraction and enables similar data manipulation).

Step 3: visualize the data: the instances and their corresponding classes.

Step 3: visualize the data: the instances and their corresponding classes.

In [6]:
```
print(X)
print(y)
```

```
     sepal length (cm)  sepal width (cm)  petal length (cm)  petal wid
th (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
..                 ...               ...                ...
```

Step 4: instantiate a decision tree and train it.

Step 4: instantiate a decision tree and train it.

In [7]:
```
tree = DecisionTreeClassifier(max_depth = 2)
tree.fit(X,y)
```

Out[7]:
```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=
2,
                       max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=N
one,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort=False,
                       random_state=None, splitter='best')
```

A DecisionTreeClassifier class has multiple parameters (e.g. max_depth). If you want to find out more about them consult the documentation or the reference at the end of this exercise.

Step 5: visualize the decision tree.

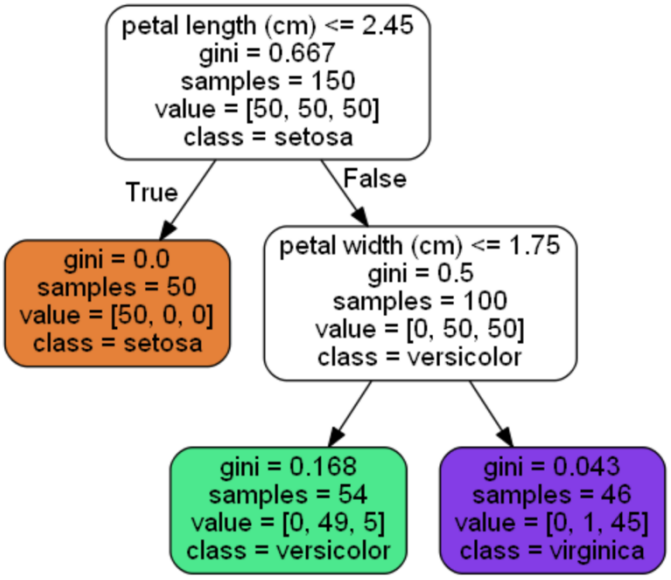
Step 5: visualize the decision tree.

In [8]:
```
from sklearn.tree import export_graphviz

# Creates dot file named tree.dot
export_graphviz(
            tree,
            out_file =  "myTree.dot",
            feature_names = list(X.columns),
            class_names = iris.target_names,
            filled = True,
            rounded = True)
```

This should create a file myTree.dot. Convert this file into an image using this online converter. You should obtain something like in the image below. Let us notice that the

decision nodes are white while the leaves nodes are coloured differently, depending on the class.



You can notice that each node displays multiple characteristics. We will discuss them in an example, starting with the content of the top most node, referred to as the root node. The root node has a depth of zero in the tree.

We can imagine that at each decision node a question is asked. The answer to this question divides the data into smaller datasets. The first question corresponding to the root node is if the petal length is smaller or equal to 2.45cm.

gini = 0.667: gini score is a metric that quantifies the purity of a node. A gini score greater than 0 implies that the node contains instances belonging to different classes. A gini score of zero means that the node contains instances belonging to a single class, thus the node is pure. Let us notice in the case of the root, the gini score is greater than zero; therefore, we know that the instances contained within the root node belong to different classes.

samples = 150: it is the number of instances contained in this node. Since the root node contains all dataset and we know that the iris flower data set contains 150 samples, this value is 150.

value = [50, 50, 50]: this tells us how many instances of this node fall into each class. The first element of the list shows the number of instances with the class id 0 (setosa), the second element with the class id 1 (versicolor) and the third element with the class id 2 (virginica). Remember, this node was not pure and here we see clearly how many instances belonging to different classes exist.

class = setosa: this value shows the prediction this node will make for a new instance. In fact, the class that will occur most frequently will be chosen. If our decision tree had a depth of 1 only, basically it would predict that all 150 instances belonged to the setosa class, which is of course a wrong prediction. Moreover, it appears that the decision tree is programmed to choose the first class of the list if there is an equal number of instances for each class.

Step 6: make predictions for new instances.

Step 6: make predictions for new instances.

In [11]:
```
sample_one_pred = int(tree.predict([[5, 5, 1, 3]]))
sample_two_pred = int(tree.predict([[5, 5, 2.6, 1.5]]))
print(f"The first sample most likely belongs a \
{iris.target_names[sample_one_pred]} flower.")
print(f"The second sample most likely belongs a \
{iris.target_names[sample_two_pred]} flower.")
```

```
The first sample most likely belongs a virginica flower.
The second sample most likely belongs a versicolor flower.
```

The input is a list in this order [sepal length, sepal width, petal length, petal width]. However, remember that for the tree we just built, the sepal length and sepal width do not affect the prediction; only the petal length and petal width were chosen.

Next, we will evaluate the decision tree predictor.

Step 7: split the dataset in a train and test datasets.

In [10]:
```
from sklearn.model_selection import train_test_split
X_trn, X_test, y_trn, y_test = train_test_split(X,
                                                y,
                                                test_size=0.333,
                                                random_state=0,
                                                stratify=y)
print("X_trn.shape = {}, X_test.shape = {}".format(
    X_trn.shape, X_test.shape))
```

```
X_trn.shape = (100, 4), X_test.shape = (50, 4)
```

In the parameters of train_test_split we specify that the test size is about 0.333 of the data. Also, the stratify parameter means that the split considers the distribution of the classes and maintains this exact distribution between the train and test datasets.
With other words, as each class represented about one third of the data in the full dataset, we could expect that in the test dataset, each class will also appear approximately in the same proportion, 1/3.

Step 8: retrain the tree classifier, this time only on the train dataset. Evaluate how good this classifier is on the train dataset.

```
In [24]: from sklearn.metrics import classification_report
         tree.fit(X_trn, y_trn)
         y_trn_pred = tree.predict(X_trn)
         print(classification_report(
             y_trn, y_trn_pred,
             target_names=['setosa', 'versicolor', 'virginica']))
```

```
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        33
  versicolor       0.94      0.97      0.96        33
   virginica       0.97      0.94      0.96        34

    accuracy                           0.97       100
   macro avg       0.97      0.97      0.97       100
weighted avg       0.97      0.97      0.97       100
```

The classification report will show us the precision, the recall, the f1-score for each class (the first 3 lines). The last two lines show us the same scores but computed on the overall dataset (if you want to know more about the difference between macro and weighted check this). We also see the accuracy reported.

Be aware, reporting the evaluation metrics on the train dataset does not show us how the classifier will perform in reality on new data. However, we can see these results as an upper limit. If the evaluation on the train dataset is poor, then we know that more data is required for the classifier to learn or new features should be defined or a new classifier algorithm may be more suitable for this task.

Step 9: make predictions on the test dataset and evaluate.

```
In [14]: y_test_pred = tree.predict(X_test)
         print(classification_report(
             y_test, y_test_pred,
             target_names=['setosa', 'versicolor', 'virginica']))
```

```
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        17
  versicolor       0.89      0.94      0.91        17
   virginica       0.93      0.88      0.90        16

    accuracy                           0.94        50
   macro avg       0.94      0.94      0.94        50
weighted avg       0.94      0.94      0.94        50
```

We can see that the scores drop for 2 classes: versicolor and virginica, and the overall dataset accuracy changes from 0.97 to 0.94.

If you want to read more about the decision tree parameters, please consults this tutorial (which was used as an inspiration for our current exercise too):

Finally, in data science projects, we rarely use a single decision tree. Instead, we use an ensemble of them (a forest) and the prediction is made by taking into consideration the

output of each decision tree in the forest. One of the most used machine learning classifiers of this type is Random Forest.

Step 10: train a Random Forest classifier instead and compare the results with those of a single decision tree.

The documentation including examples of a Random Forest classifier is here.

Exercise 3- K-nearest neighbours

Step 1: train K-nearest neighbours on the iris dataset and compare the results.

The documentation including examples of a KNeighborsClassifier is here.

Step 2: load and split a new dataset.

Choose another toy dataset from here, load it and prepare the train and test datasets similar to how we did in Exercise 2.

Step 3: train a Decision tree classifier and a K-nn classifier on this new dataset.

Compare the results on the test dataset.

Repeat the experiments on a different dataset split (train 50%- test 50%) and with different numbers of nearest neighbours (parameter n_neighbors).

Consider using only a part of the features (hint: X[List_feature_names] selects a sub-table of X with only the columns given by List_feature_names).

Exercise 4- K-means

For this exercise we will need to install some extra libraries, namely numpy and matplotlib, which will help us to generate some random data and visualize it.

```
$ pip3 install numpy
$ pip3 install matplotlib
```

Step 1: import libraries.

Exercise 4

Step 1: import libraries

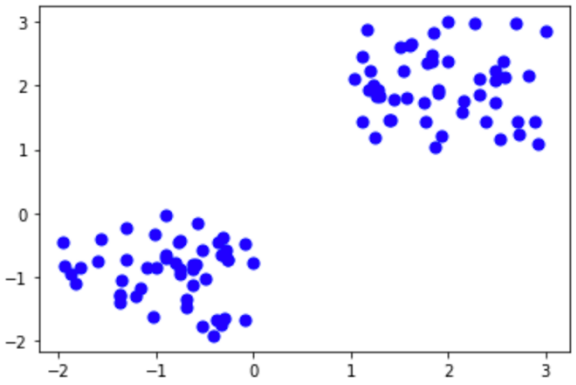
In [16]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
```

Step 2: generate some random data and visualize it.

We generate 100 random data points in a two-dimensional space, divided into two groups, each containing 50 instances.

Step 2: generate some random data and visualize it.

In [18]:
```
X= -2 * np.random.rand(100,2)
X1 = 1 + 2 * np.random.rand(50,2)
X[50:100, :] = X1
plt.scatter(X[ : , 0], X[ :, 1], s = 50, c = 'b')
plt.show()
```



Step 3: fit a K-means classifier.

In [19]:
```
Kmean = KMeans(n_clusters=2)
Kmean.fit(X)
```

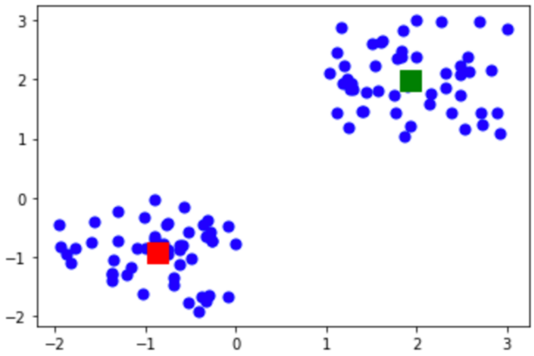
Out[19]:
```
KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
       n_clusters=2, n_init=10, n_jobs=None, precompute_distances='aut
o',
       random_state=None, tol=0.0001, verbose=0)
```

Step 4: visualize the cluster centroids.

First, we print the centroids and then we display them in the same plot as the data.

```
In [22]: centroids = Kmean.cluster_centers_
print(centroids)
plt.scatter(X[ : , 0], X[ : , 1], s =50, c='b')
plt.scatter(centroids[0][0], centroids[0][1], s=200, c='g', marker='s')
plt.scatter(centroids[1][0], centroids[1][1], s=200, c='r', marker='s')
plt.show()
```

```
[[ 1.92319496  1.99744452]
 [-0.86042449 -0.9134136 ]]
```



Step 5: test the clustering algorithms.

Let’s start by visualizing the labels attributed by the algorithm to the 100 data points. We can notice that the first half of them are in a class with id 1, while the second half in a class with id 0 (or it may be vice versa in your experiment).

Step 5: test the clustering algorithms.

```
In [23]: print(Kmean.labels_)
```

```
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

Finally, we can predict for new data points as follows:

```
In [25]: sample_test=np.array([-3.0,-3.0])
second_test=sample_test.reshape(1, -1)
predicted = Kmean.predict(second_test)
print(predicted)
```

```
[1]
```

As expected this new data point is part of the lower left cluster as it is the closest to the other data points with negative coordinates.

Step 6: choose the numbers of clusters.

In this exercise we knew that we had two clusters as we were the ones generating the data. However, in real-world scenarios, we may not have this information.

For this reason, data scientists often use the Elbow method, which helps them to decide on the optimal number of clusters.

To find out more about the Elbow method, I suggest this tutorial.

Step 7: image compression with k-means (optional)

This is an optional exercise but quite fun!
We will use k-means for image compression. Follow the steps from this tutorial.