DataScientest • com

Introduction to Machine Learning with Scikit-learn

Part II: Simple classification models

For this second part of an introduction to the `scikit-learn` module, we will focus on the second type of problem in Machine Learning: the **classification** problem.

The objective of this introduction is to:

- Introduce the classification problem.
- Learn to use the `scikit-learn` module to build a classification model, also known as a "classifier".
- Introduce metrics adapted to the evaluation of classification models.

Introduction to classifcation

Objective of classification

In supervised learning, the objective is to predict the value of a target variable from explanatory variables.

- In a **regression** problem, the target variable takes **continuous values**. These values are numerical: price of a house, quantity of oxygen in the air of a city, etc. The target variable can therefore take an **infinity of values**.

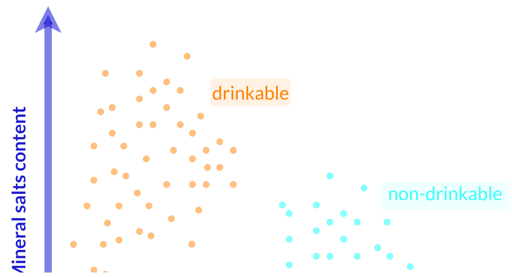
- In a **classification** problem, the target variable takes **discrete values**. These values can be numeric or literal, but in both cases the target variable takes a **finite** number of values.

The different values taken by the target variable are called **classes**.

The objective of classification therefore consists in predicting the class of an observation from its explanatory variables.

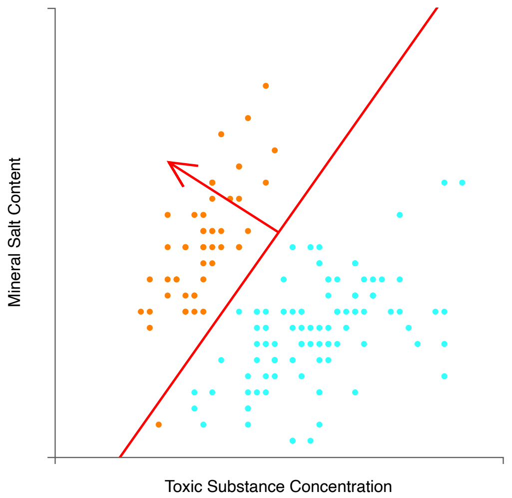
We will look at a problem of a **binary** classification, i.e. where there are **two** classes. We are trying to determine whether the water in a stream is drinkable or not depending on its concentration of toxic substances and its mineral salts content.

The two classes are therefore **'drinkable'** and **'non-drinkable'**.



In [1]:
```
from classification_widgets import linear_classification

linear_classification()
```



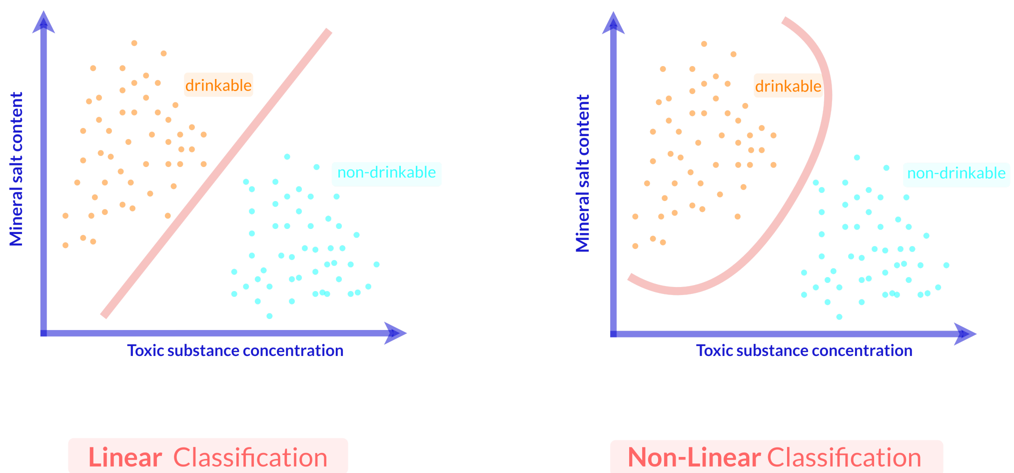
w1 −1.47

w2 0.84

The classification we just performed is of **linear** type, that is to say that we used a flat linear plane to separate our classes.

This plane was parametrized by the vector w. Thus, **the objective of linear classification models is to find the vector w allowing the best possible separation of the different classes**. Each model of linear type has its own technique to find this vector.

There are also **non-linear** classification models, which we will see later.



We will now introduce the main tools of the `scikit-learn` module for solving a classification problem.

In this exercise we will use the [Congressional Voting Records (https://archive.ics.uci.edu/ml/datasets/congressional+voting+records)](https://archive.ics.uci.edu/ml/datasets/congressional+voting+records) dataset, containing a number of votes cast by members of Congress of the United States House of Representatives.

The objective of our classification problem will be to **predict the political party** ("Democrat" or "Republican") of the members of the House of Representatives according to their votes on subjects like the education, health, budget, etc.

The explanatory variables will therefore be the votes on various subjects and the target variable will be the "democrat" or "republican" political party.

To solve this problem we will use:

- A non-linear classification model: **K-Nearest Neighbors**.
- A linear classification model: **Logistic Regression**.

Data preparation

- **(a)** Run the following cell to import the `pandas` and `numpy` modules needed for the exercise.

In [2]:
```
import pandas as pd
import numpy as np
%matplotlib inline
```

- **(b)** Load the data contained in the file `'votes.csv'` into a `DataFrame` named `votes`.

In [3]:
```
# Insert your code here

votes = pd.read_csv('votes.csv')
votes.head(3)
```

Out[3]:

	party	infants	water	budget	physician	salvador	religious	satellite	aid	missile	immigration	synfuels	education	superfund	crime	duty_free_exports	eaa_rsa
0	republican	n	y	n	y	y	y	n	n	n	y	n	y	y	y	n	y
1	republican	n	y	n	y	y	y	n	n	n	n	n	y	y	y	n	n
2	democrat	n	y	y	n	y	y	n	n	n	n	y	n	y	y	n	n

Hide solution

In [4]:
```
votes = pd.read_csv('votes.csv')
```

In order to briefly visualize our data:

- **(c)** Display the number of rows and columns of `votes`.
- **(d)** Show a preview of the first 20 rows of `votes`.

In [5]:
```
# Insert your code here
print(votes.shape)

#votes.head(20)
votes.head()
```

```
(435, 17)
```

Out[5]:

	party	infants	water	budget	physician	salvador	religious	satellite	aid	missile	immigration	synfuels	education	superfund	crime	duty_free_exports	eaa_rsa
0	republican	n	y	n	y	y	y	n	n	n	y	n	y	y	y	n	y
1	republican	n	y	n	y	y	y	n	n	n	n	n	y	y	y	n	n
2	democrat	n	y	y	n	y	y	n	n	n	n	y	n	y	y	n	n
3	democrat	n	y	y	n	n	y	n	n	n	n	y	n	y	n	n	y
4	democrat	y	y	y	n	y	y	n	n	n	n	y	n	y	y	y	y

Show solution

the target variable.

- The following **16** columns contain the votes of each member of Congress on legislative proposals:
  - `'y'` indicates that the elected member voted **for** the bill.
  - `'n'` indicates that the elected member voted **against** the bill.

In order to use the data in a classification model, it is necessary to transform these columns into binary **numeric** values, i.e. either 0 or 1.

- **(e)** For each of the columns 1 to 16 (column 0 being our target variable), replace the values `'y'` by 1 and `'n'` by 0. To do so, we can use the **`replace`** method from the `DataFrame` class.
- **(f)** Display the first 10 rows of the modified `DataFrame` .

In [6]:

```
# Insert your code here
lst = ['y', 'n']
df = votes.copy()

df = df.iloc[: ,1:].replace('y', 1).replace('n', 0)
# df.replace(('y', 'n'), (1,0)) # better
df.head()
```

Out[6]:

	infants	water	budget	physician	salvador	religious	satellite	aid	missile	immigration	synfuels	education	superfund	crime	duty_free_exports	eaa_rsa
0	0	1	0	1	1	1	0	0	0	1	0	1	1	1	0	1
1	0	1	0	1	1	1	0	0	0	0	0	1	1	1	0	0
2	0	1	1	0	1	1	0	0	0	0	1	0	1	1	0	0
3	0	1	1	0	0	1	0	0	0	0	1	0	1	0	0	1
4	1	1	1	0	1	1	0	0	0	0	1	0	1	1	1	1

Hide solution

In [7]:

```
# Replacing the values
votes = votes.replace(('y', 'n'), (1, 0))

# Display the first 10 rows of the DataFrame
votes.head(10)
```

Out[7]:

	party	infants	water	budget	physician	salvador	religious	satellite	aid	missile	immigration	synfuels	education	superfund	crime	duty_free_exports	eaa_rsa
0	republican	0	1	0	1	1	1	0	0	0	1	0	1	1	1	0	1
1	republican	0	1	0	1	1	1	0	0	0	0	0	1	1	1	0	0
2	democrat	0	1	1	0	1	1	0	0	0	0	1	0	1	1	0	0
3	democrat	0	1	1	0	0	1	0	0	0	0	1	0	1	0	0	1
4	democrat	1	1	1	0	1	1	0	0	0	0	1	0	1	1	1	1
5	democrat	0	1	1	0	1	1	0	0	0	0	0	0	1	1	1	1
6	democrat	0	1	0	1	1	1	0	0	0	0	0	0	0	1	1	1
7	republican	0	1	0	1	1	1	0	0	0	0	0	0	1	1	0	1
8	republican	0	1	0	1	1	1	0	0	0	0	0	1	1	1	0	1
9	democrat	1	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0

- **(g)** In a `DataFrame` named `X` , store the **explanatory** variables of the dataset (all columns except `'party'`). For this, you can use the **`drop`** method of a `DataFrame` .
- **(h)** In a `DataFrame` named `y` , store the **target variable** (`'party'`).

In [8]:

```
# Insert your code here

X = votes.drop(['party'], axis=1)

y = votes['party']
```

Hide solution

In [9]:

```
# Separation of the variables

X = votes.drop(['party'], axis = 1)
y = votes['party']
```

> - The training set is used to **train the classification** model, that is to say find the parameters of the model which best separate the classes.
> - The test set is used to **evaluate** the model on data that it has never seen. This evaluation will allow us to judge the **generalizability** of the model.

- **(i)** Import the `train_test_split` function from the `sklearn.model_selection` submodule. Remember that this function is used as follows:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2)
```

- **(j)** Split the data into a training set `(X_train, y_train)` and a test set `(X_test, y_test)` keeping 20% of the data for the test set.

> To **eliminate the randomness** of the `train _test_split` function, you can use the **`random_state`** parameter with an integer value (for example `random_state = 2`). This will make it so every time you use the function with the argument `random_state = 2` , the datasets produced will be the same.

In [10]:
```python
# Insert your code here
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2)
```

Hide solution

In [11]:
```python
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 2)
```

Non-linear classification: K-Nearest Neighbors model

> In order to assign a class to an observation, the K-Nearest Neighbors algorithm considers, as its name suggests, the **K nearest neighbors** of the observation and determines **the most represented class** among these neighbors.
>
> Concretely, the algorithm is as follows:
>
> > - Suppose that **K = 5**.
> > - For an observation that we want to classify, we will look at the 5 points of the training set that are closest to our observation. The distance metric used is often the **euclidian norm**.
> > - If among the 5 neighbors, the majority is "democrat", then the observation will be classified "democrat".
>
> To train a K-Nearest Neighbors model for our problem, we'll use the **`KNeighborsClassifer`** class from the `neighbors` submodule of `scikit-learn` .
>
> The number **K** of neighbors to consider is entered using the parameter **`n_neighbors`** of the **`KNeighborsClassifer`** constructor.

- **(k)** Run the following cell to import the `KNeighborsClassifer` class.

In [12]:
```python
from sklearn.neighbors import KNeighborsClassifier
```

- **(l)** Instantiate a `KNeighborsClassifier` model named **`knn`** which will consider the **6 nearest neighbors** for classification.
- **(m)** Using the **`fit`** method, train the model `knn` on the training dataset.
- **(n)** Using the **`predict`** method, perform a prediction on the **test** dataset. Store these predictions in **`y_pred_test_knn`** and display the first 10 predictions.

In [13]:
```python
# Insert your code here
knn = KNeighborsClassifier(n_neighbors = 6)
knn.fit(X_train, y_train)
y_pred_test_knn = knn.predict(X_test)
y_pred_test_knn[:10]
```

Out[13]:
```
array(['democrat', 'democrat', 'democrat', 'democrat', 'democrat',
       'democrat', 'democrat', 'democrat', 'republican', 'democrat'],
      dtype=object)
```

Hide solution

```
knn = KNeighborsClassifier(n_neighbors = 6)

# Entraînement du modèle sur le jeu d'entraînement
knn.fit(X_train, y_train)

# Prédiction sur les données de test
y_pred_test_knn = knn.predict(X_test)

# Affichage des 5 premières prédictions
print(y_pred_test_knn[:10])
```

```
['democrat' 'democrat' 'democrat' 'democrat' 'democrat' 'democrat'
 'democrat' 'democrat' 'republican' 'democrat']
```

In the last part of this exercise, we will see how to evaluate the performance of our model using these predictions.

Linear Classification: Logistic Regression

The logistic regression model is closely related to the **linear regression** model seen in the previous notebook.

They should not be confused since **they do not solve the same types of problems**:

- **Logistic** Regression is used for **classification** (predict classes).
- **Linear** regression is used for **regression** (predict a quantitative variable).

The linear regression model was defined with the following formula:

$$y \approx \beta_0 + \sum_{j=1}^{p} \beta_j x_j$$

Logistic regression no longer estimates y directly but the **probability** that y is equal to 0 or 1. Thus, the model is defined by the formula:

$$P(y = 1) = f(\beta_0 + \sum_{j=1}^{p} \beta_j x_j)$$

Where

$$f(x) = \frac{1}{1 + e^{-x}}$$

The f function, often called **sigmoid** or **logistic function**, transforms the linear combination $\beta_0 + \sum_{j=1}^{p} \beta_j x_j$ into a value between 0 and 1 that can be interpreted as a **probability**:

- If $\beta_0 + \sum_{j=1}^{p} \beta_j x_j$ is **positive**, then $P(y = 1) > 0.5$, so the predicted class of the observation will be **1**.
- If $\beta_0 + \sum_{j=1}^{p} \beta_j x_j$ is **negative**, then $P(y = 1) < 0.5$, i.e. $P(y = 0) > 0.5$, so the predicted class of the observation will be **0**.

- **(o)** Import the `LogisticRegression` class from the `linear_model` submodule of `scikit-learn`.
- **(p)** Instantiate a `LogisticRegression` model named **`logreg`** without specifying constructor arguments.
- **(q)** Train the model on the training dataset.
- **(r)** Make a prediction on the **test** dataset. Store these predictions in **`y_pred_test_logreg`** and display the first 10 predictions.

In [15]:

```
# Insert your code here

from sklearn.linear_model import LogisticRegression

logreg = LogisticRegression()

logreg.fit(X_train, y_train)

y_pred_test_logreg = logreg.predict(X_test)

y_pred_test_logreg[:10]
```

Out[15]:

```
array(['democrat', 'republican', 'democrat', 'democrat', 'democrat',
       'democrat', 'democrat', 'democrat', 'republican', 'democrat'],
      dtype=object)
```

Hide solution

```python
from sklearn.linear_model import LogisticRegression

# Instanciation du modèle
logreg = LogisticRegression()

# Entraînement du modèle sur le jeu d'entraînement
logreg.fit(X_train, y_train)

# Prédiction sur les données de test
y_pred_test_logreg = logreg.predict(X_test)

# Affichage des 5 premières prédictions
print(y_pred_test_logreg[:10])
```

```
['democrat' 'republican' 'democrat' 'democrat' 'democrat' 'democrat'
 'democrat' 'democrat' 'republican' 'democrat']
```

There are different metrics to evaluate the performance of classification models such as:

- The **accuracy**.
- The **precision and the recall**.
- The **F1-score**.

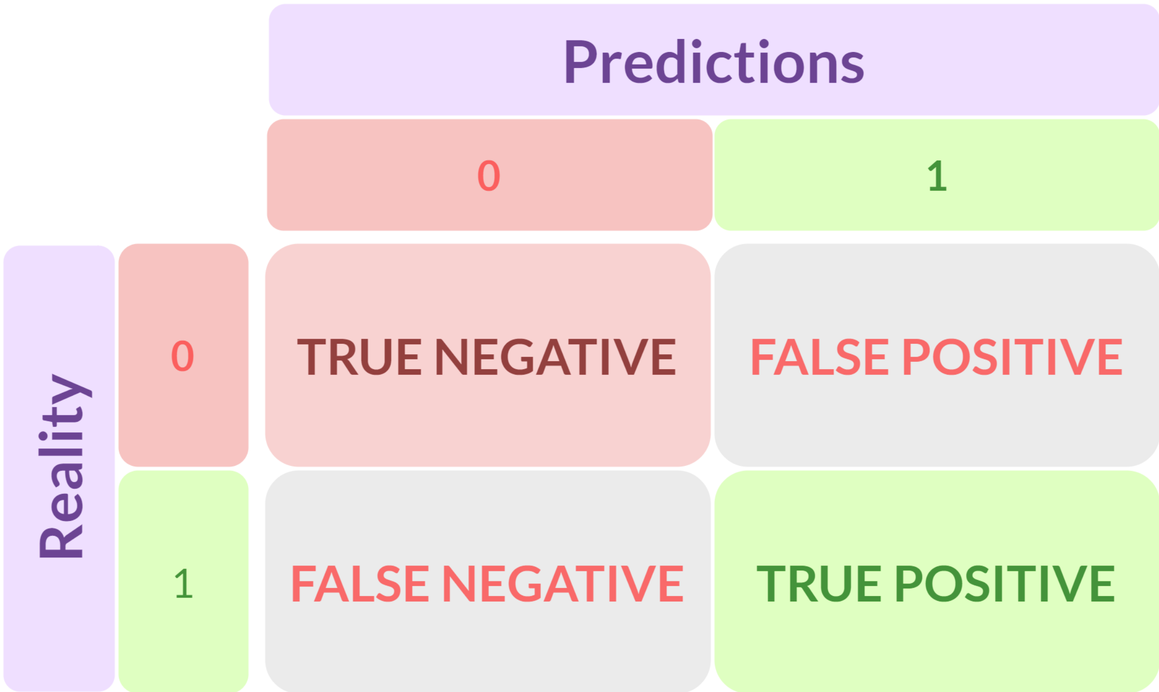
Each metric assesses the performance of the model with a different approach.

In order to explain these concepts, we will introduce 4 very important terms.

Arbitrarily, we will choose that the class **'republican' will be the positive class** (1) and **'democrat' will be the negative class** (0).

Thus, we will call:

- **True Positive (TP)** an observation classified as **positive** ('republican') by the model which is indeed **positive** ('republican').
- **False Positive (FP)** an observation classified as **positive** ('republican') by the model which was actually **negative** ('democrat').
- **True Negative (TN)** an observation classified as **negative** ('democrat') by the model and which is indeed **negative**('democrat').
- **False Negative (FN)** an observation classified as **negative** ('democrat') by the model which was actually **positive** ('republican').



The **accuracy** is the most common metric used to evaluate a model. It simply corresponds to the rate of **correct** predictions made by the model.

We suppose that we have n observations. We denote by TP the number of True Positives and TN the number of True Negatives. Then the accuracy is given by:

$$\text{accuracy} = \frac{\mathrm{TP} + \mathrm{TN}}{n}$$

The **precision** is a metric which answers the question: **Among all the positive predictions of the model, how many are true positives?** If we denote by FP the number of False Positives of the model, then the precision is given by:

$$\text{precision} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}$$

A high precision score tells us the model does not blindly classify everyone as positive.

The **recall** is a metric that quantifies the proportion of truly positive observations that were correctly classified as positive by the model.

If we write FN as the number of False Negatives, then the callback is given by:

$$\text{recall} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}$$

A high recall score tells us the model is able to properly detect the truly positive observations.

The **confusion matrix** counts the values of TP, TN, FP and FN for a set of predictions, which allows us to calculate the three previous metrics:

$$\text{ConfusionMatrix} = \begin{bmatrix} \mathrm{TN} & \mathrm{FP} \\ \mathrm{FN} & \mathrm{TP} \end{bmatrix}$$

The **`confusion_matrix`** function of the `sklearn.metrics` submodule generates the confusion matrix from the **predictions** of a model:

```
confusion_matrix (y_true, y_pred)
```

As a reminder:

- **`y_pred`** contains the values of y **predicted** by the model.

- **(a)** Import the **`confusion_matrix`** function from `sklearn.metrics`.
- **(b)** Calculate the confusion matrix of the predictions produced by the model **`knn`**. These predictions were stored in **`y_pred_test_knn`**.
- **(c)** Display the confusion matrix. How many false positives have occurred? The positive class corresponds to **`'republican'`**.
- **(d)** Using the formulas given above, calculate the **accuracy**, **precision**, and **recall** scores of the `knn` model on the test set. You can use **tuple assignment** to deconstruct the confusion matrix:

```
(TN, FP), (FN, TP) = confusion_matrix(y_true, y_pred)
```

In [17]:

```python
# Insert your code here

# NOTE
# 'republican' = 1
# 'democrat' = 0
from sklearn.metrics import confusion_matrix


y_pred_test_knn = knn.predict(X_test)

cm = confusion_matrix(y_test, y_pred_test_knn)
print('confusion_matrix : \n', cm)
print(cm[0])
print(cm[0,0])

(TN, FP), (FN, TP) = confusion_matrix(y_test, y_pred_test_knn)
accuracy = (TP + TN) / len(y_test)
precision = TP / (TP + FP)
recall = TP / (TP + FN)

print('Accuracy :', accuracy, '\nPrecision :', precision, '\nRecall :', recall)
```

```
confusion_matrix :
 [[48  5]
 [ 3 31]]
[48  5]
48
Accuracy : 0.9080459770114943
Precision : 0.8611111111111112
Recall : 0.9117647058823529
```

Hide solution

In [18]:

```python
from sklearn.metrics import confusion_matrix

# Computation and display of the confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred_test_knn)
print("Confusion Matrix:\n",  conf_matrix)

print("\nThe knn model made", conf_matrix[0,1], "False Positives.")

# Computation of the accuracy, precision and recall
(TN, FP), (FN, TP) = confusion_matrix(y_test, y_pred_test_knn)
n = len(y_test)

print("\nKNN Accuracy:", (TP + TN) / n)

print("\nKNN Precision:", TP / (TP + FP))

print("\nKNN Recall:", TP / (TP + FN))
```

```
Confusion Matrix:
 [[48  5]
 [ 3 31]]

The knn model made 5 False Positives.

KNN Accuracy: 0.9080459770114943

KNN Precision: 0.8611111111111112

KNN Recall: 0.9117647058823529
```

```
pd.crosstab(y_test, y_pred_test_knn, rownames = ['Reality'], colnames = ['Prediction'])
```

Which in our case will produce the following `DataFrame` :

Prediction	democrat	republican
Reality		
democrat	48	5
republican	2	32

For this dataset, the KNN model performs quite well. When the classes are **balanced**, i.e. there are about as many positives as there are negatives in the dataset, accuracy is a good enough metric to assess the performance.

However, as you will see later, when **a class is dominant**, precision and recall are much more relevant metrics.

If you think you cannot remember the formulas for the metrics of accuracy, precision and recall, do not worry! The **`sklearn.metrics`** submodule contains functions to calculate them quickly:

```
accuracy_score(y_test, y_pred_test_knn)
>>> 0.9195402298850575
```

- **(e)** Import the **`accuracy_score`** , **`precision_score`** and **`recall_score`** functions from the `sklearn.metrics` submodule.
- **(f)** Display the confusion matrix of the predictions made by the **`logreg`** model using **`pd.crosstab`** .
- **(g)** Calculate the accuracy, precision and recall of model predictions **`logreg`** . To use the `precision_score` and `recall_score` metrics, you will need to fill in the argument **`pos_label = 'republican'`** in order to specify that the `'republican'` class is the positive class.

In [19]:

```
# Insert your code here

from sklearn.metrics import accuracy_score, precision_score, recall_score


knn_crosstab = pd.crosstab(y_test, y_pred_test_knn, rownames = ['Reality'], colnames = ['Prediction'])
print(knn_crosstab)


accuracy_score = accuracy_score(y_test, y_pred_test_knn)
precision_score = precision_score(y_test, y_pred_test_knn, pos_label = 'republican')
recall_score = recall_score(y_test, y_pred_test_knn, pos_label = 'republican')

print('Accuracy Score : ', accuracy_score, '\nPrecison Score : ', precision_score, '\nRecall Score   : ', recall_score )
```

```
Prediction  democrat  republican
Reality
democrat          48           5
republican         3          31
Accuracy Score :  0.9080459770114943
Precison Score :  0.8611111111111112
Recall Score   :  0.9117647058823529
```

Hide solution

In [20]:

```
from sklearn.metrics import accuracy_score, precision_score, recall_score

# Computation and display of the confusion matrix
pd.crosstab(y_test, y_pred_test_logreg, rownames=['Reality'], colnames=['Prediction'])

# Computation of the accuracy, precision and recall
print("\nLogReg Accuracy:", accuracy_score(y_test, y_pred_test_logreg))

print("\nLogReg Precision:", precision_score(y_test, y_pred_test_logreg, pos_label = 'republican'))

print("\nLogReg Recall:", recall_score(y_test, y_pred_test_logreg, pos_label = 'republican'))
```

```
LogReg Accuracy: 0.9310344827586207

LogReg Precision: 0.8888888888888888

LogReg Recall: 0.9411764705882353
```

The **`classification_report`** function of the `sklearn.metrics` submodule displays all these metrics for each class.

- **(h)** Import the **`classification_report`** function from the `sklearn.metrics` submodule.
- **(i)** Display using the `print` and `classification_report` functions the classification reports of the models **`logreg`** and **`knn`** on the test set.

```
from sklearn.metrics import classification_report

print('Logistic Regresion Report: \n', classification_report(y_test, y_pred_test_logreg))

print('\nKNN Report: \n', classification_report(y_test, y_pred_test_knn))
```

```
Logistic Regresion Report:
               precision    recall  f1-score   support

    democrat       0.96      0.92      0.94        53
  republican       0.89      0.94      0.91        34

    accuracy                           0.93        87
   macro avg       0.92      0.93      0.93        87
weighted avg       0.93      0.93      0.93        87


KNN Report:
               precision    recall  f1-score   support

    democrat       0.94      0.91      0.92        53
  republican       0.86      0.91      0.89        34

    accuracy                           0.91        87
   macro avg       0.90      0.91      0.90        87
weighted avg       0.91      0.91      0.91        87
```

Hide solution

In [22]:

```
from sklearn.metrics import classification_report

print("LogReg report:\n", classification_report(y_test, y_pred_test_logreg))

print("\n\n")

print("KNN report:\n", classification_report(y_test, y_pred_test_knn))
```

```
LogReg report:
               precision    recall  f1-score   support

    democrat       0.96      0.92      0.94        53
  republican       0.89      0.94      0.91        34

    accuracy                           0.93        87
   macro avg       0.92      0.93      0.93        87
weighted avg       0.93      0.93      0.93        87



KNN report:
               precision    recall  f1-score   support

    democrat       0.94      0.91      0.92        53
  republican       0.86      0.91      0.89        34

    accuracy                           0.91        87
   macro avg       0.90      0.91      0.90        87
weighted avg       0.91      0.91      0.91        87
```

> The classification report is a little more complete than what we have done so far. It contains an additional metric: the **F1-Score**.
>
> The F1-Score is a sort of average between precision and recall. The F1-Score adapts very well to classification problems with balanced or unbalanced classes.
>
> For most classification problems, the model with the highest F1-Score will be considered the model whose recall and precision **performances** are the most **balanced**, and is therefore **preferable** to others.

- **(j)** Import the **`f1_score`** function from submodule **`sklearn.metrics`**.
- **(k)** Compare the F1-Scores of the models **`knn`** and **`logreg`** on the test set. Which model has the best performance? As for the recall and the precision, it will be necessary to fill in the argument **`pos_label = 'republican'`**.

In [23]:

```
# Insert your code here

from sklearn.metrics import f1_score

print('Log. Reg. F1_score : ', f1_score(y_test, y_pred_test_logreg, pos_label = 'republican'))

print('KNN F1_score       : ', f1_score(y_test, y_pred_test_knn, pos_label = 'republican'))
```

```
Log. Reg. F1_score :  0.9142857142857143
KNN F1_score       :  0.8857142857142858
```

Hide solution

```
print("F1 KNN:", f1_score(y_test, y_pred_test_knn, pos_label = 'republican'))
print("F1 LogReg:", f1_score(y_test, y_pred_test_logreg, pos_label = 'republican'))
```

```
F1 KNN: 0.8857142857142858
F1 LogReg: 0.9142857142857143
```

Conclusion and recap

Scikit-learn offers many classification models that can be grouped into two families:

- Linear **models** like **`LogisticRegression`**.
- Non-linear **models** like **`KNeighborsClassifier`**.

The implementation of these models is done in the same way for **all** models of scikit-learn:

- **Instantiation** of the model.
- **Training** of the model: **`model.fit(X_train, y_train)`**.
- **Prediction**: **`model.predict(X_test)`**.

The prediction on the test set allows us to **evaluate** the performance of the model thanks to suitable **metrics**.

The metrics we have seen are used for **binary** classification and are calculated using 4 values:

- True Positives: Prediction = **+** | Reality = **+**
- True Negatives: Prediction = **-** | Reality =**-**
- False Positives: Prediction = **+** |Reality = **-**
- False Negatives: Prediction = **-**| Reality = **+**

All these values can be calculated using the **confusion matrix** generated by the **`confusion_matrix`** function of the `sklearn.metrics` submodule or by the **`pd.crosstab`** function.

Thanks to these values, we can calculate metrics like:

- **Accuracy**: The proportion of correctly classified observations.
- **Precision**: The proportion of true positives among all the positive predictions of the model.
- **Recall**: the proportion of truly positive observations that were correctly classified as positive by the model.

All these metrics can be obtained using the **`classification_report`** function of the **`sklearn.metrics`** submodule.

The **F1-Score** quantifies the **balance** between these metrics, which gives us a reliable **criterion** for **choosing** the model most suited to our problem.

Validate