

# 02\_numerical\_pipeline\_introduction

May 29, 2021

## 1 First model with scikit-learn

In this notebook, we present how to build predictive models on tabular datasets, with only numerical features.

In particular we will highlight:

- the scikit-learn API: `.fit(X, y)/.predict(X)/.score(X, y)`;
- how to evaluate the statistical performance of a model with a train-test split.

### 1.1 Loading the dataset with Pandas

We will use the same dataset “adult\_census” described in the previous notebook. For more details about the dataset see <http://www.openml.org/d/1590>.

Numerical data is the most natural type of data used in machine learning and can (almost) directly be fed into predictive models. We will load a subset of the original data with only the numerical columns.

```
[1]: import pandas as pd

adult_census = pd.read_csv("../datasets/adult-census-numeric.csv")
```

Let's have a look at the first records of this dataframe:

```
[2]: adult_census.head()
```

```
[2]:
```

	age	capital-gain	capital-loss	hours-per-week	class
0	41	0	0	92	<=50K
1	48	0	0	40	<=50K
2	60	0	0	25	<=50K
3	37	0	0	45	<=50K
4	73	3273	0	40	<=50K

We see that this CSV file contains all information: the target that we would like to predict (i.e. "class") and the data that we want to use to train our predictive model (i.e. the remaining columns). The first step is to separate columns to get on one side the target and on the other side the data.

## 1.2 Separate the data and the target

```
[3]: target_name = "class"
     target = adult_census[target_name]
     target
```

```
[3]: 0      <=50K
     1      <=50K
     2      <=50K
     3      <=50K
     4      <=50K
     ...
    39068    <=50K
    39069    <=50K
    39070     >50K
    39071    <=50K
    39072     >50K
     Name: class, Length: 39073, dtype: object
```

```
[4]: data = adult_census.drop(columns=[target_name, ])
     data.head()
```

```
[4]:   age  capital-gain  capital-loss  hours-per-week
0   41             0             0             92
1   48             0             0             40
2   60             0             0             25
3   37             0             0             45
4   73          3273             0             40
```

We can now linger on the variables, also denominated features, that we will use to build our predictive model. In addition, we can also check how many samples are available in our dataset.

```
[5]: data.columns
```

```
[5]: Index(['age', 'capital-gain', 'capital-loss', 'hours-per-week'], dtype='object')
```

```
[6]: print(f"The dataset contains {data.shape[0]} samples and "
     f"{data.shape[1]} features")
```

The dataset contains 39073 samples and 4 features

## 1.3 Fit a model and make predictions

We will build a classification model using the “K-nearest neighbors” strategy. To predict the target of a new sample, a k-nearest neighbors takes into account its k closest samples in the training set and predicts the majority target of these samples.

Caution!

We use a K-nearest neighbors here. However, be aware that it is seldom useful in practice. We use it because it is an intuitive algorithm. In the next notebook, we will introduce better models.

The `fit` method is called to train the model from the input (features) and target data.

```
[7]: # to display nice model diagram
from sklearn import set_config
set_config(display='diagram')
```

```
[8]: from sklearn.neighbors import KNeighborsClassifier

model = KNeighborsClassifier()
model.fit(data, target)
```

```
[8]: KNeighborsClassifier()
```

Learning can be represented as follows:

The method `fit` is composed of two elements: (i) a **learning algorithm** and (ii) some **model states**. The learning algorithm takes the training data and training target as input and sets the model states. These model states will be used later to either predict (for classifiers and regressors) or transform data (for transformers).

Both the learning algorithm and the type of model states are specific to each type of model.

Note

Here and later, we use the name `data` and `target` to be explicit. In scikit-learn documentation, `data` is commonly named `X` and `target` is commonly called `y`.

Let's use our model to make some predictions using the same dataset.

```
[9]: target_predicted = model.predict(data)
```

We can illustrate the prediction mechanism as follows:

To predict, a model uses a **prediction function** that will use the input data together with the model states. As for the learning algorithm and the model states, the prediction function is specific for each type of model.

Let's now have a look at the computed predictions. For the sake of simplicity, we will look at the five first predicted targets.

```
[10]: target_predicted[:5]
```

```
[10]: array([' >50K', ' <=50K', ' <=50K', ' <=50K', ' <=50K'], dtype=object)
```

Indeed, we can compare these predictions to the actual data...

```
[11]: target[:5]
```

```
[11]: 0    <=50K
      1    <=50K
      2    <=50K
      3    <=50K
      4    <=50K
      Name: class, dtype: object
```

...and we could even check if the predictions agree with the real targets:

```
[12]: target[:5] == target_predicted[:5]
```

```
[12]: 0    False
      1     True
      2     True
      3     True
      4     True
      Name: class, dtype: bool
```

```
[13]: print(f"Number of correct prediction: "
          f"{(target[:5] == target_predicted[:5]).sum()} / 5")
```

Number of correct prediction: 4 / 5

Here, we see that our model makes a mistake when predicting for the first sample.

To get a better assessment, we can compute the average success rate.

```
[14]: (target == target_predicted).mean()
```

```
[14]: 0.8224349294909529
```

But, can this evaluation be trusted, or is it too good to be true?

## 1.4 Train-test data split

When building a machine learning model, it is important to evaluate the trained model on data that was not used to fit it, as generalization is more than memorization (meaning we want a rule that generalizes to new data, without comparing to data we memorized). It is harder to conclude on never-seen instances than on already seen ones.

Correct evaluation is easily done by leaving out a subset of the data when training the model and using it afterwards for model evaluation. The data used to fit a model is called training data while the data used to assess a model is called testing data.

We can load more data, which was actually left-out from the original data set.

```
[15]: adult_census_test = pd.read_csv('../datasets/adult-census-numeric-test.csv')
```

From this new data, we separate out input features and the target to predict, as in the beginning of this notebook.

```
[16]: target_test = adult_census_test[target_name]
      data_test = adult_census_test.drop(columns=[target_name, ])
```

We can check the number of features and samples available in this new set.

```
[17]: print(f"The testing dataset contains {data_test.shape[0]} samples and "
          f"{data_test.shape[1]} features")
```

The testing dataset contains 9769 samples and 4 features

Instead of computing the prediction and manually computing the average success rate, we can use the method `score`. When dealing with classifiers this method returns their performance metric.

```
[18]: accuracy = model.score(data_test, target_test)
      model_name = model.__class__.__name__

      print(f"The test accuracy using a {model_name} is "
            f"{accuracy:.3f}")
```

The test accuracy using a `KNeighborsClassifier` is 0.807

Let's check the underlying mechanism when the `score` method is called:

To compute the score, the predictor first computes the predictions (using the `predict` method) and then uses a scoring function to compare the true target `y` and the predictions. Finally, the score is returned.

If we compare with the accuracy obtained by wrongly evaluating the model on the training set, we find that this evaluation was indeed optimistic compared to the score obtained on an held-out test set.

It shows the importance to always testing the statistical performance of predictive models on a different set than the one used to train these models. We will discuss later in more details how predictive models should be evaluated.

Note

In this MOOC, we will refer to statistical performance of a model when referring to the test score or test error obtained by comparing the prediction of a model and the true targets. Equivalent terms for statistical performance are predictive performance and generalization performance. We will refer to computational performance of a predictive model when accessing the computational costs of training a predictive model or using it to make predictions.

In this notebook we:

- fitted a **k-nearest neighbors** model on a training dataset;
- evaluated its statistical performance on the testing data;
- introduced the scikit-learn API `.fit(X, y)` (to train a model), `.predict(X)` (to make predictions) and `.score(X, y)` (to evaluate a model).