Politecnico di Torino

1859

Data Science Lab

Scikit-learn

Classification

Andrea Pasini
Flavio Giobergia
Elena Baralis

DataBase and Data Mining Group

- Scikit-learn
  - Machine learning library built on **NumPy**, **SciPy** and **Matplotlib**
- What Scikit-learn can do
  - **Supervised** learning
    - Regression, classification
  - **Unsupervised** learning
    - Clustering
  - Data **preprocessing**
    - Feature extraction, feature selection, dimensionality reduction

- What Scikit-learn **cannot** do
  - Distributed computation on multiple computers
    - Only multi-core optimization
  - Deep learning
    - Use Keras and Tensorflow instead

- Scikit learn models work with structured data
  - Data must be in the form of **2D Numpy arrays**
    - Rows represent the **samples**
    - Columns represent the **attributes (or features)**
  - This table is called **features matrix**

shape = (3, 3)

	Price	Quantity	Liters
Sample 1	1.0	5	1.5
Sample 2	1.4	10	0.3
Sample 3	5.0	8	1

- Features can be
  - **Real** values
  - **Integer** values to represent categorical data
- If you have strings in your data, you first have to convert them to integers (**preprocessing**)

Input data

1.0	**January**	1.5
1.4	**February**	0.3
5.0	**March**	1

→

Features matrix

1.0	**0**	1.5
1.4	**1**	0.3
5.0	**2**	1

- Also **missing values** must be solved before applying any model
  - With imputation or by removing rows

Input data

1.0	0.5	1.5
1.4	**NaN**	0.3
5.0	0.5	1

→

Features matrix

1.0	0.5	1.5
1.4	**0.5**	0.3
5.0	0.5	1

Input data

1.0	0.5	1.5
1.4	**NaN**	0.3
5.0	0.5	1

→

Features matrix

1.0	0.5	1.5
5.0	0.5	1

- For **unsupervised** learning you only need the features matrix
- For **supervised** learning you also need a **target** array to train the model
  - It is typically one-dimensional, with length n_samples
    - May be 2-dimensional for multi-output models

Features matrix
shape = (n_samples, n_features)

1.0	5	1.5
1.4	10	0.3
5.0	8	1

Target array
shape = (n_samples,)

A
A
B

- The target array can contain
  - Integer values, each corresponding to a class label

Target labels		Target array
Dog	→	0
Dog		0
Cat		1

  - Real values for regression

Target array
0.4
1.8
-6.9

- Scikit-learn estimator API
  - All models are represented with Python classes
  - Their classes include
    - The values of the **hyperparameters** used to configure the model
    - The values of the **parameters** learned after training
      - By convention these attributes end with an underscore
    - The **methods** to train the model and make inference
  - Scikit-learn models are provided with sensible **defaults** for the hyperparameters

- Scikit learn models follow a simple, shared **pattern**

  1. **Import** the model that you need to use
  2. **Build** the model, setting its hyperparameters
  3. **Train** model parameters on your data
     - Using the fit() method
  4. **Use** the model to make predictions
     - Using the predict()/transform() methods

- Sometimes fit and predict/transform are implemented within the same class method

- **fit():** learn model parameters from input data
  - E.g. train a classifier
- **predict():** apply model parameters to make predictions on data
  - E.g. predict class labels
- **transform()**: transform data into a different representation
  - E.g. normalize test data
- **fit_predict():** fit model and make predictions
  - E.g. apply clustering to data
- **fit_transform():** fit model and transform data
  - E.g. apply PCA to transform data

- Classification:
  - Given a 2D features matrix X
    - X.shape = (n_samples, n_features)
  - The task consists of assigning a class label y_pred to each data sample
    - y_pred.shape = (n_samples)

1.0	5	1.5
1.4	10	0.3
…	…	…

X

A
B
B

y_pred

Classification

By following the estimator API pattern:

- Import a model

```
from sklearn.tree import DecisionTreeClassifier
```

- Build model object

```
clf = DecisionTreeClassifier()
```

- Important decision tree hyperparameters:

```
from sklearn.tree import DecisionTreeClassifier
clf = DecisionTreeClassifier(max_depth = 10,
                             min_impurity_decrease=0.01)
```

- Hyperparameters:
  - *max_depth*: maximum tree height
    - Default = None
  - *min_impurity_decrease*: split nodes only if impurity decrease above threshold
    - Default = 0.0

- Train model with ground-truth labels

```
In [1]: clf.fit(X_train, y_train)
```

- This operation builds the decision tree structure
  - X_train is the 2D Numpy array with input features (**features matrix**)
  - y_train is a 1D array with ground-truth labels

X_train

6.1	3.1	2
1.8	12	0.15
...	...	...

y_train

0
2
1

Classification

- Predict class labels for new data

```
In [1]: y_pred = clf.predict(X_test)
```

```
Out[1]: [3, 1, 1, 1, 2, 2, 0]
```

- This operation shows the capability of classifiers to make predictions for unseen data

1.0	5	1.5
1.4	10	0.3
…	…	…

X_test

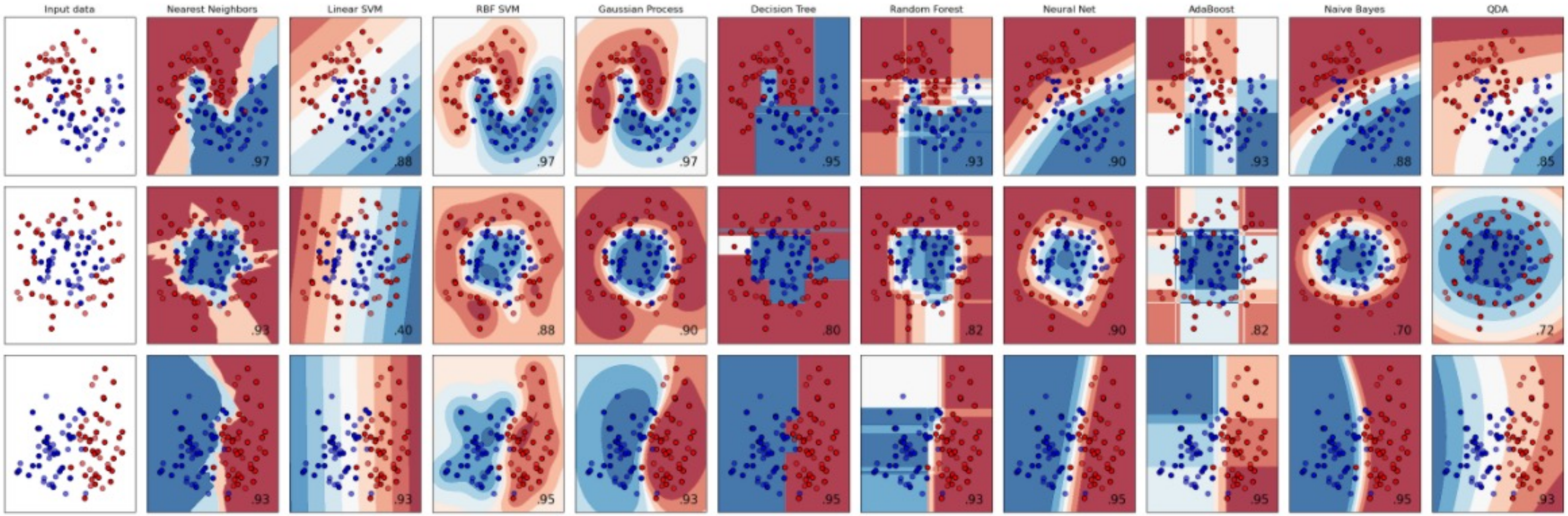
→

1
3
3

y_pred

Classification

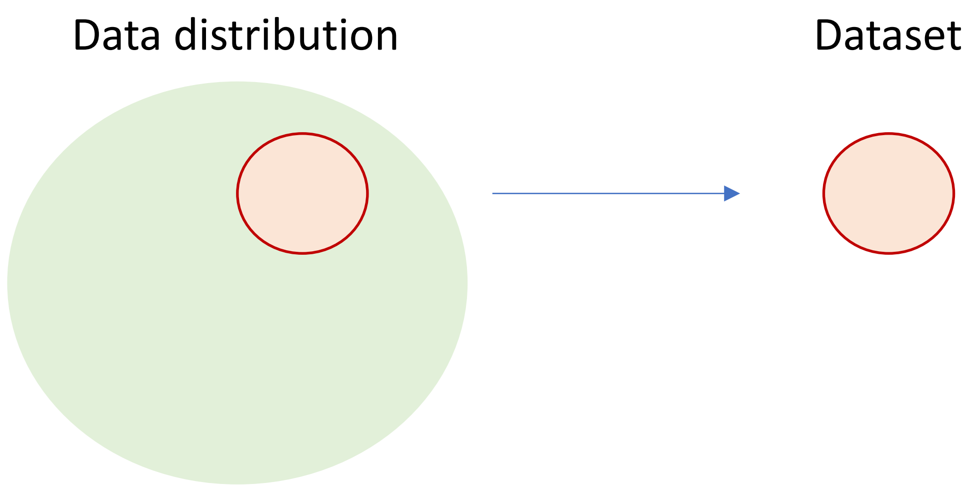
- Take a look at all the other models in the scikit-learn documentation
  - https://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html



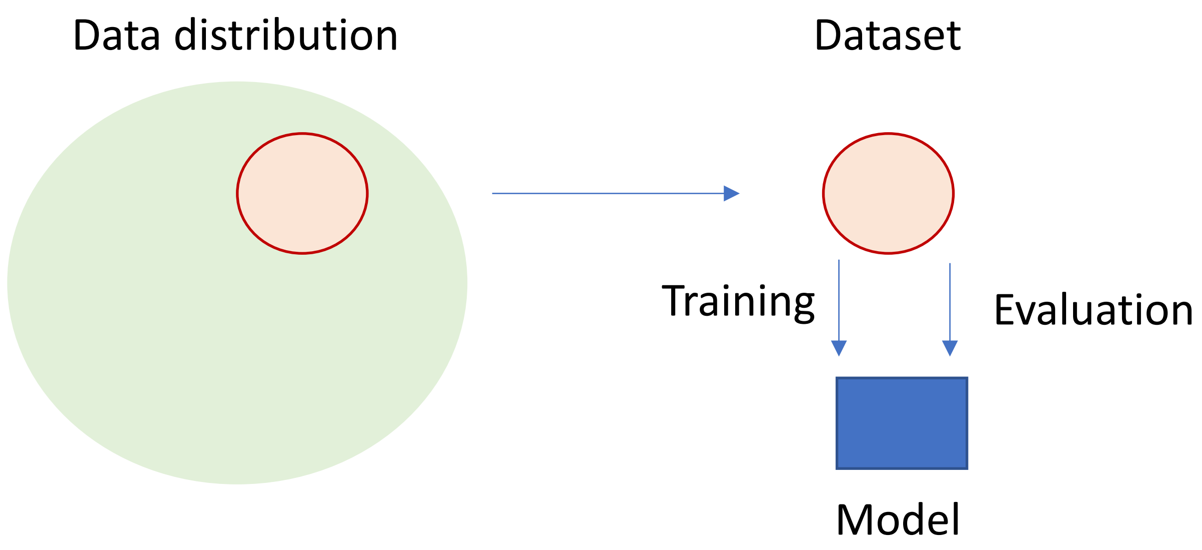
- To **choose** the most appropriate machine learning model for your data you have to **evaluate** its performance

- Evaluation can be performed according to a **metric (scoring function)**
  - E.g. accuracy, precision, recall

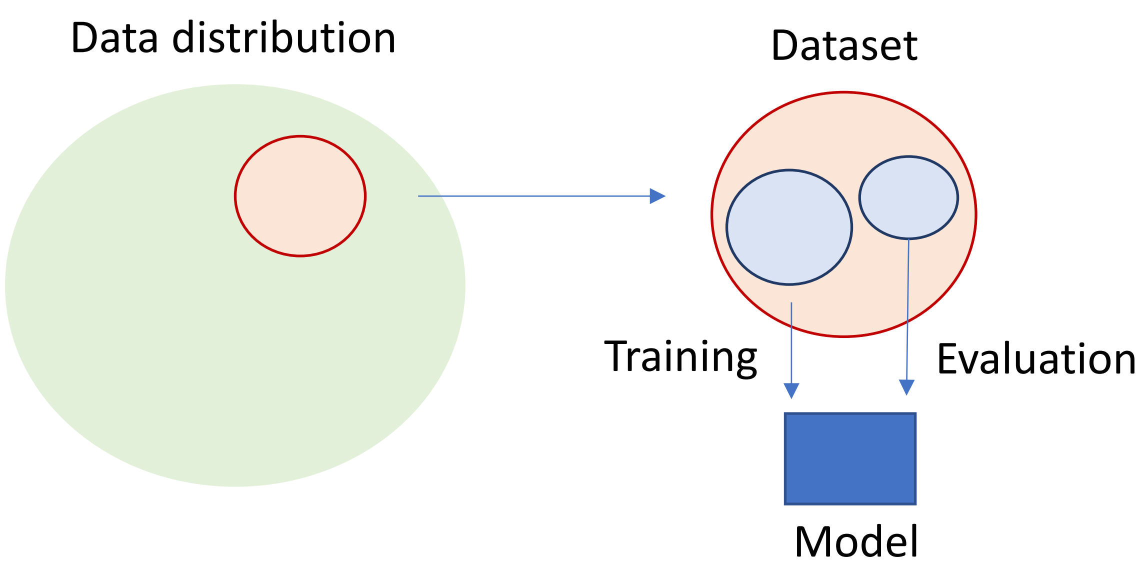
- The data that you have in a dataset is only a **sample extracted** from the distribution of real world data



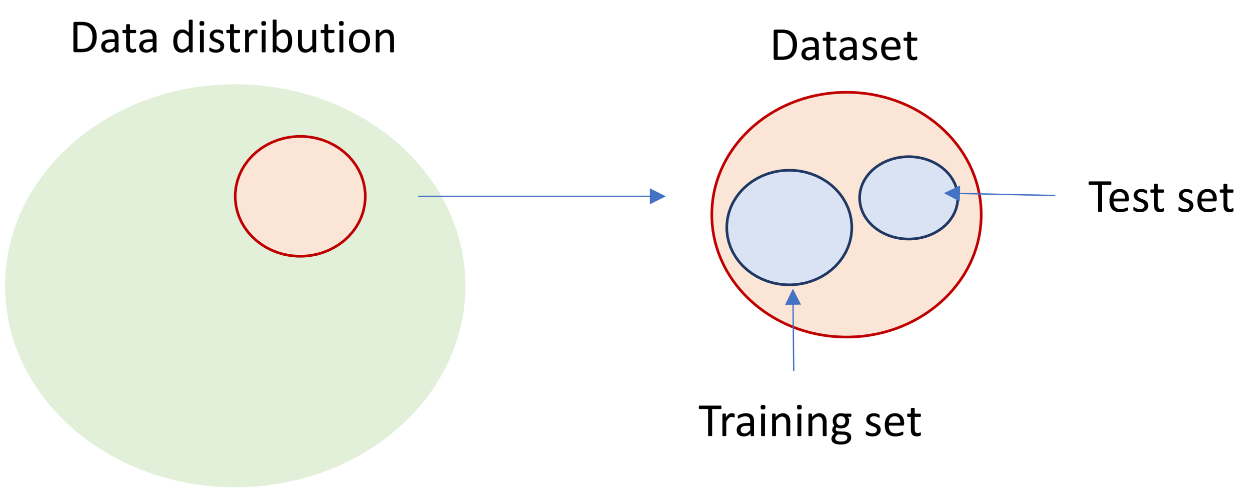
- If you choose the best model for your dataset, it may not perform so well for **new data**
  - This risk is called **overfitting**



- To avoid overfitting evaluation must be performed on data that is not used for training the model
  - Divide your dataset into **training** and **test** set to simulate two different samples in the data distribution



- This technique is called **hold-out**
  - Training set is typically 70/90% of your data

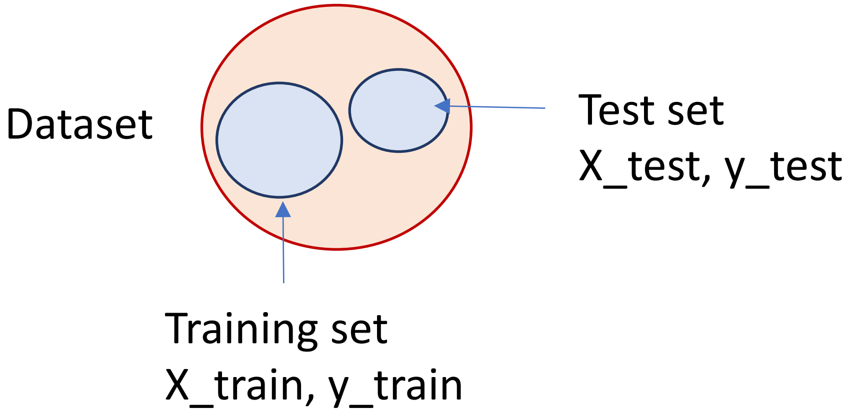


- Hold-out with Scikit-learn

```
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)
```

  - Default test_set size is 0.25 (25%)



Classification

- Evaluation = compare the following two vectors
  - y_test (y): the expected result (**ground truth**)
  - y_test_pred ($\hat{y}$): the prediction made by your model

- Main evaluation metrics for classification:
  - Accuracy: % of correct samples
  - Precision(c): % of correct samples among those predicted with class c
  - Recall(c); % of correct samples among those that belong to class c in ground truth
  - F_1 score(c): harmonic mean between precision and recall

- Evaluation metrics with Scikit-learn
  - With `precision_score()`, `recall_score()`, `f1_score()`, …
  - Or, `precision_recall_fscore_support()`
    - Returns those metrics together

```
from sklearn.metrics import accuracy_score,
                            precision_recall_fscore_support

acc = accuracy_score(y_test, y_test_pred)
p, r, f1, s = precision_recall_fscore_support(y_test, y_test_pred)
```

```
p, r, f1, s = precision_recall_fscore_support(y_test, y_test_pred)
```

- p, r, f1, s are 1D Numpy arrays with the scores computed separately for each class
  - Example

	class 0	class 1	class 2
p =	0.99	0.99	0.5
r =	0.77	0.97	0.99

many samples of class 2 are recognized, but model is not precise with this class

- **Macro average** scores vs **Weighted average** scores

```
p, r, f1, s = precision_recall_fscore_support(y_test, y_test_pred,
                                              average='macro')
```

  - **Macro** average f1:

```
macro_f1 = f1.mean()
```

  - Macro average gives the **same importance** to all classes, even if they are unbalanced
    - If a class with few elements gets a low f1, the macro-averaged score is affected with the same weight as another with more samples

Weighted average scores

```
p, r, f1, s = precision_recall_fscore_support(y_test, y_test_pred,
                                               average = 'weighted')
```

- Weighted average scores are by assigning each score a different weight, based on class cardinality
- Classes with higher **cardinality** have **higher impact** on these metrics

PoliTo DBMG

Confusion matrix

- Useful tool when you want to inspect with more details the classification results

In [1]:
```
from sklearn.metrics import confusion_matrix

conf_mat = confusion_matrix(y_test, y_test_pred)
print(conf_mat)
```

Out[1]:

actual \ predicted	0	1	2
0	45	0	1
1	0	43	0
2	0	3	42

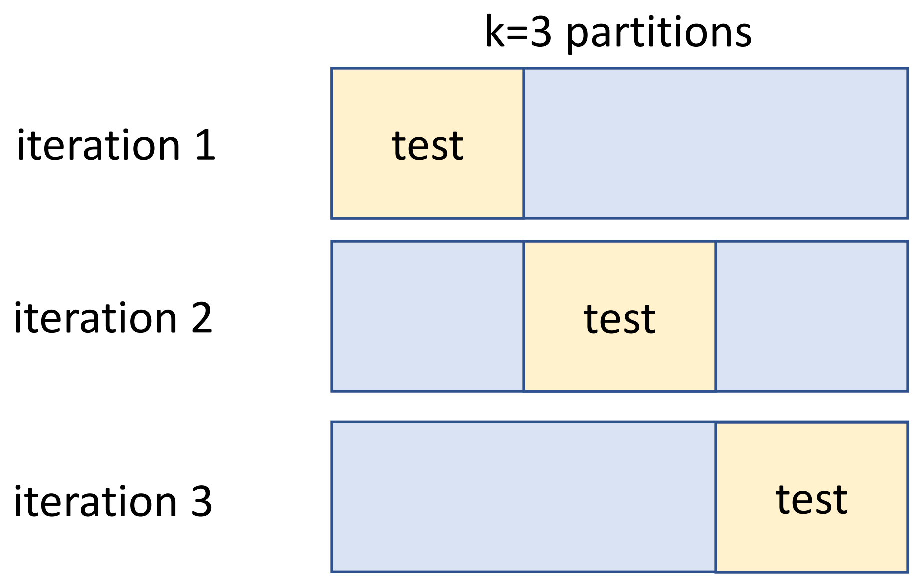
```
[[45, 0, 1],
 [0, 43, 0],
 [0, 3, 42]]
```

Notebook Examples

- **4a-Scikitlearn-Classification.ipynb**
  - **1. Classification and hold out**

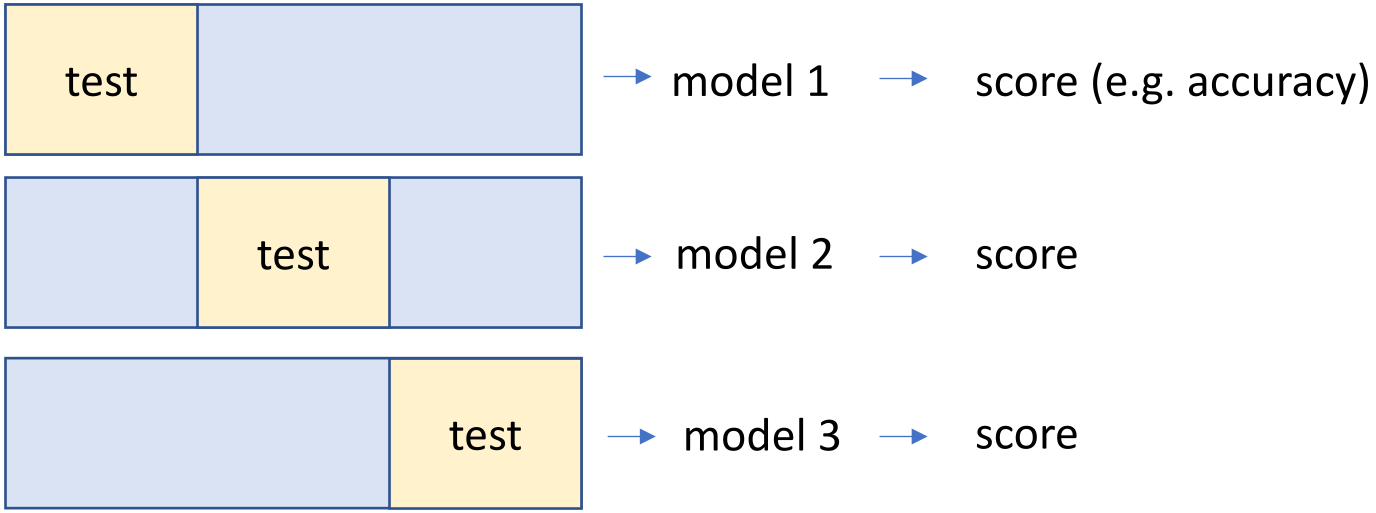
Cross-validation

- Divide your dataset into **k** partitions
- At each iteration select a partition to be used as **test** set and the others will be the **training** set

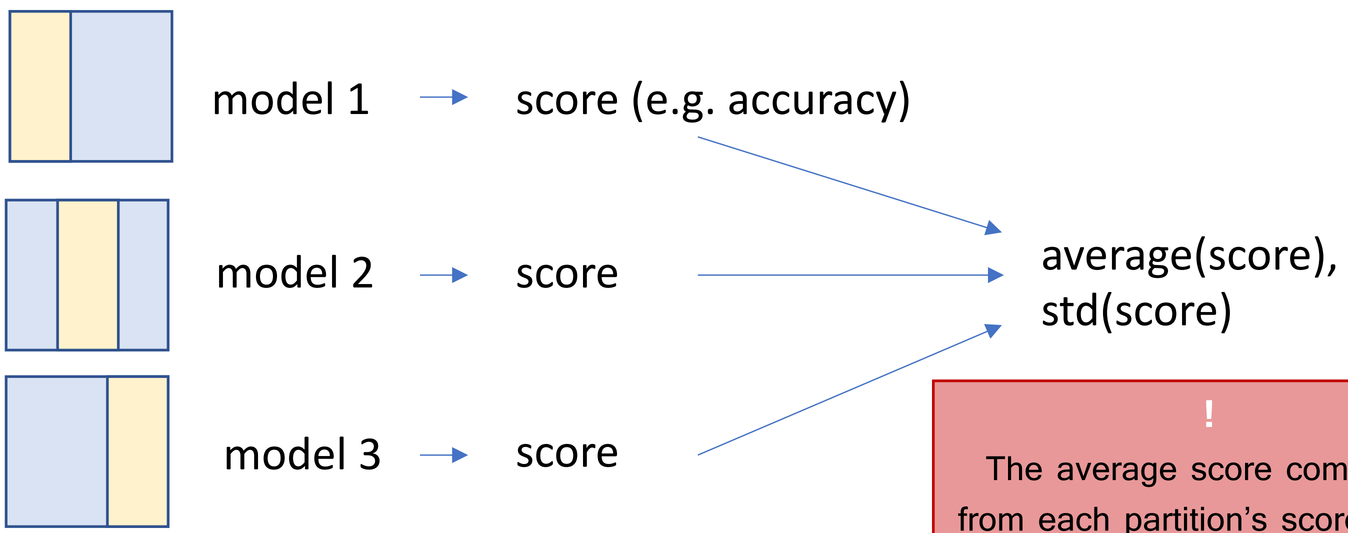


Cross-validation

- At each iteration a **different model** is trained
- After training a model compute a **scoring** metric to the predictions for the test set



- At the end you can compute **statistics** on the obtained scores



!

The average score computed from each partition's score is not necessarily the same as the overall accuracy.
We should weight the average by the # of samples in each partition

- Method 1: iterate across partitions

```
from sklearn.model_selection import KFold


# K-Fold with 5 splits
kfold = KFold(n_splits=5, shuffle=True)


for train_indices, test_indices in kfold.split(X, y):
        ... executed 5 times, 1 for each k-fold iteration ...
```

- Shuffle specifies to shuffle data before creating the k partitions (default is False)

- Method 1: iterate across partitions

```
...
for train_indices, test_indices in kfold.split(X, y):
        ... executed 5 times, 1 for each k-fold iteration ...
```

- kfold.split() returns at each **iteration** a tuple with two **arrays**:
  - train_indices: array of the **indices** (row number) of the training samples
  - test_indices: array of the indices of the test samples

Cross-validation

- Method 1: iterate across partitions

```
...
for train_indices, test_indices in kfold.split(X, y):
        train model on X[train_indices], y[train_indices]
        test model on X[test_indices]
        compute an evaluation score for this partition
```

- At each iteration you can use **fancy indexing** to select the samples from X and y
- Then you can train a model and compute its performances on the test set

- Method 2: use cross_val_score()

```
from sklearn.model_selection import cross_val_score


clf = DecisionTreeClassifier()
acc = cross_val_score(clf, X, y, cv=5, scoring='accuracy')
```

- Parameters:
  - clf = the model that you want to be trained
  - X, y = your dataset, where cross-validation will be performed
- Important: this method **does not shuffle** data
  - Manually shuffle them when necessary (suggested)

Cross-validation

- Method 2: use cross_val_score()

```
from sklearn.model_selection import cross_val_score


clf = DecisionTreeClassifier()
acc = cross_val_score(clf, X, y, cv=5, scoring='accuracy')
```

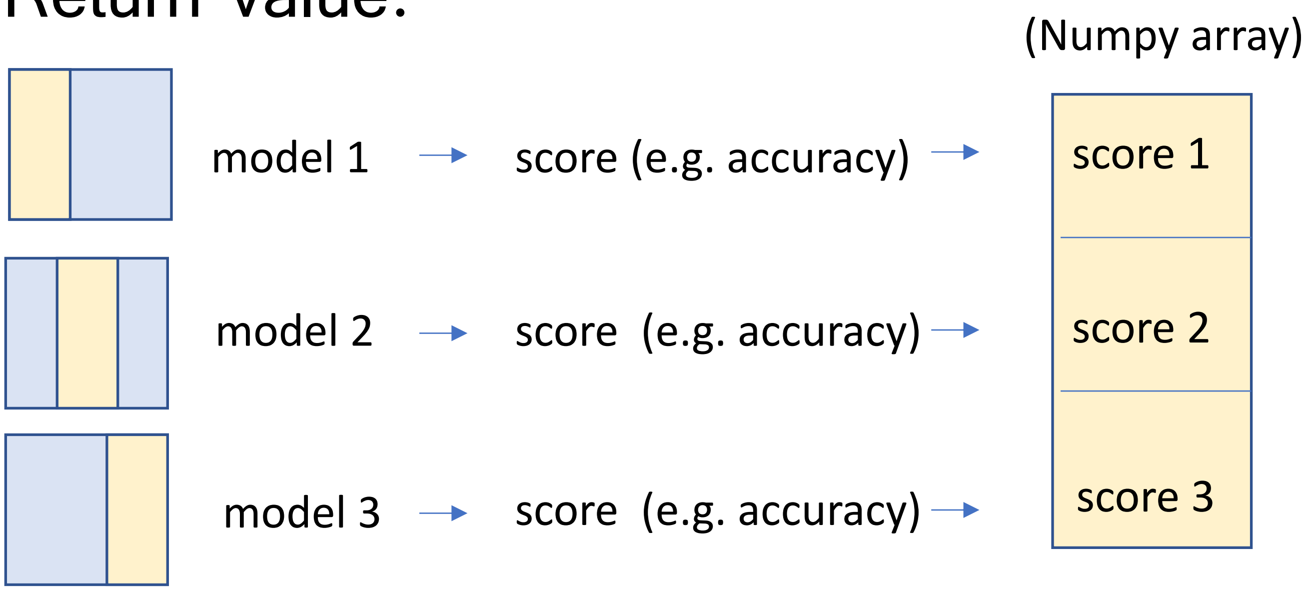
- Parameters:
  - cv = number of partitions for cross-validation
  - scoring = scoring function for the evaluation
    - E.g. 'f1_macro', 'f1_micro', 'accuracy', 'precision_macro'

- Method 2: use cross_val_score()

```
In [1]: cross_val_score(clf, X, y, cv=3, scoring='accuracy')

Out[1]: array([0.85, 0.86, 0.833])
```

- Return value:



- Method 3: use cross_val_predict()

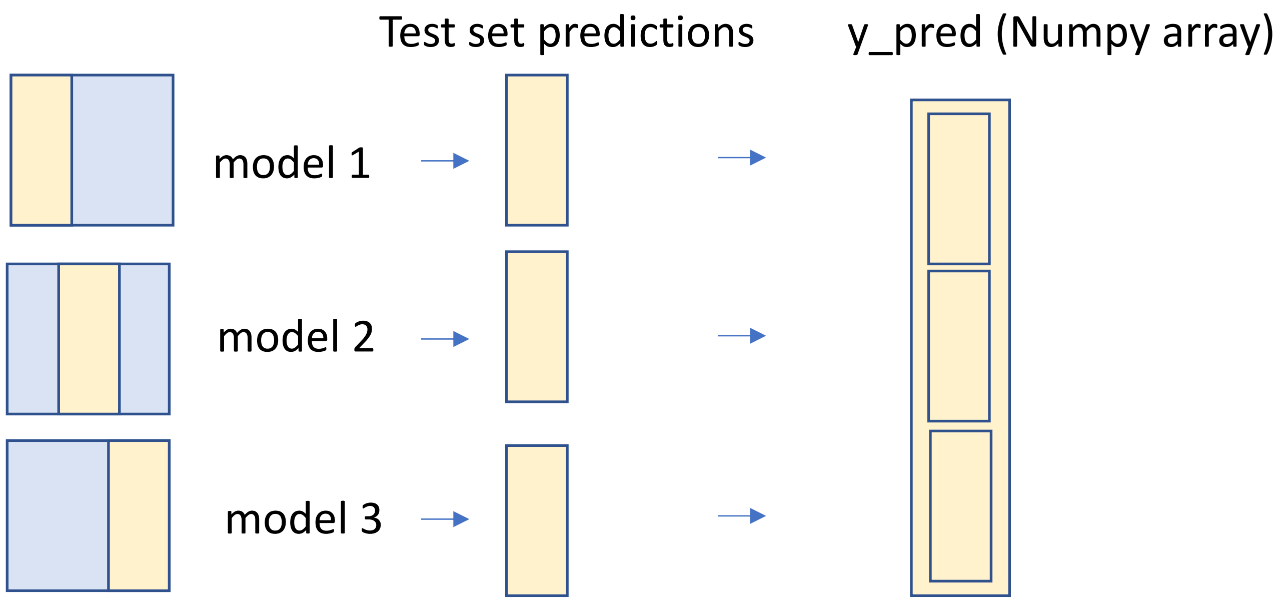
```
from sklearn.model_selection import cross_val_predict
y_pred = cross_val_predict(clf, X, y, cv=3)
```

- This method returns a Numpy array with the predictions of the *cv* models trained during cross validation
- Data is **not shuffled**

Cross-validation

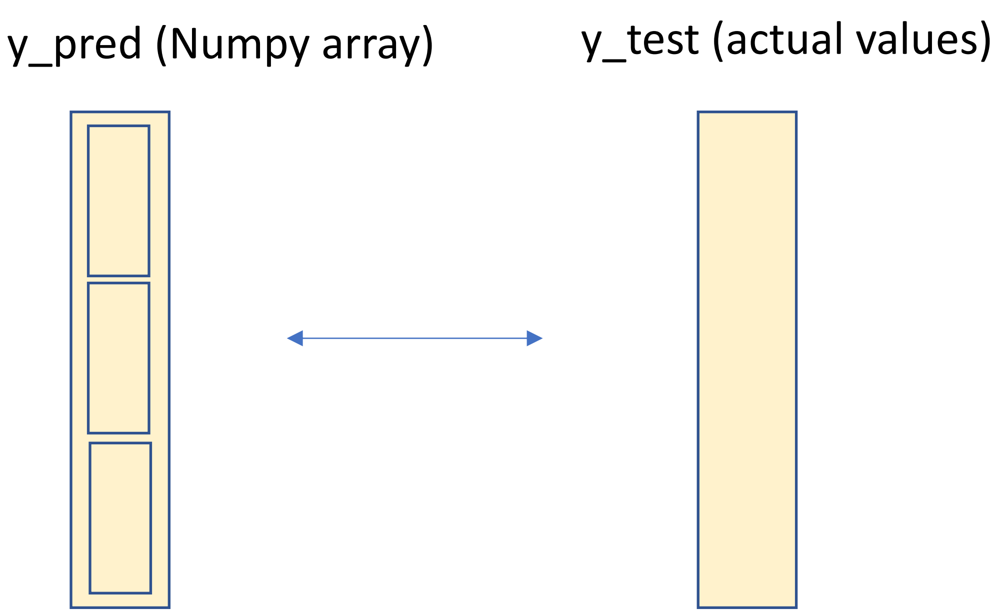
- Method 3: use cross_val_predict()

```
from sklearn.model_selection import cross_val_predict
y_pred = cross_val_predict(clf, X, y, cv=3)
```

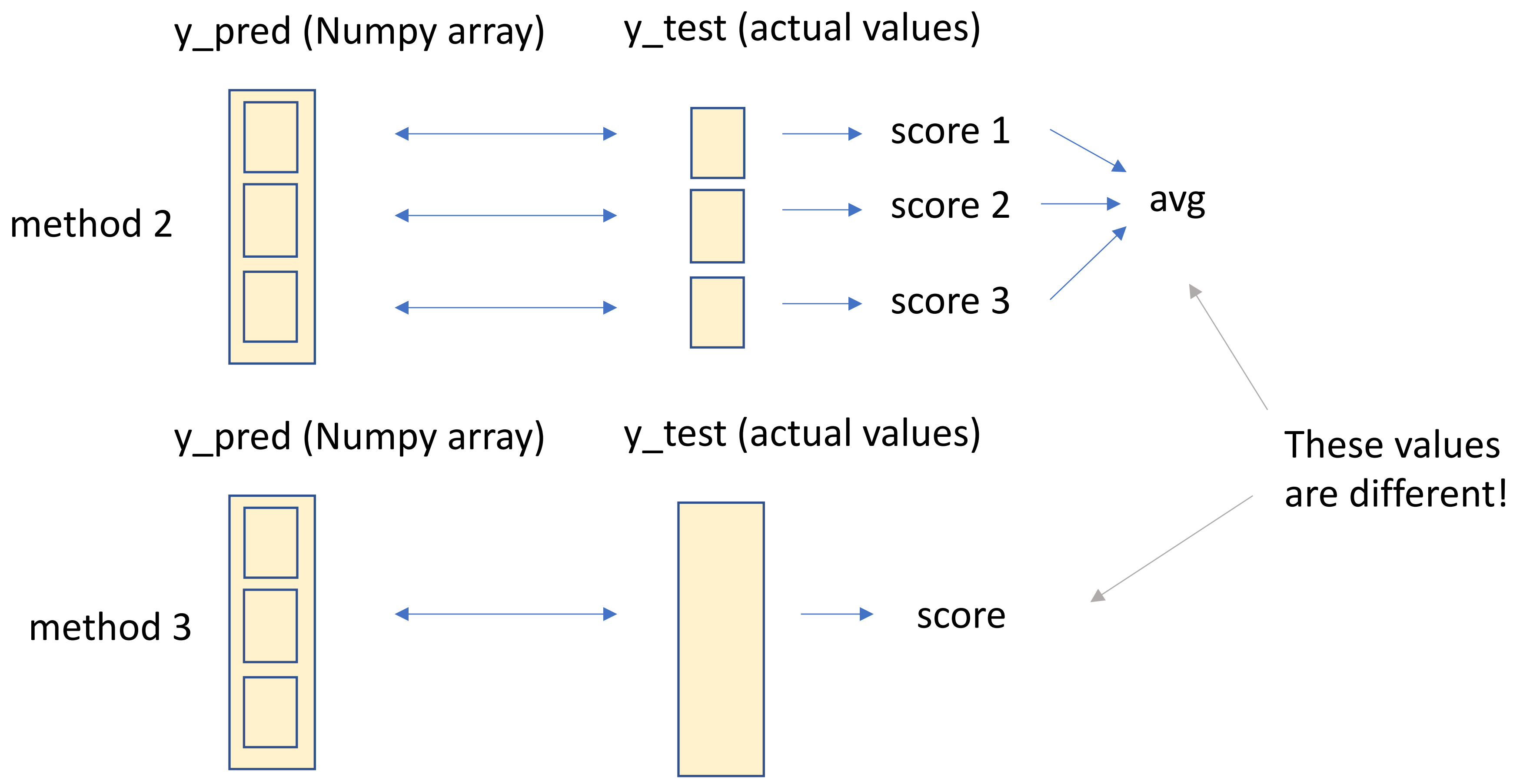


- Method 3: use cross_val_predict()
  - Finally you can evaluate the predictions

```
acc = accuracy_score(y_test, y_test_pred)
```



- Difference between method 2 and method 3



Notebook Examples

- **4a-Scikitlearn-Classification.ipynb**
  - **2. Cross validation**