POLITECNICO
DI TORINO

Data Science Lab

Scikit-learn

DataBase and Data Mining Group

Andrea Pasini, Elena Baralis

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

Classification

- Classifiers follow the **fit/predict pattern**
- Example: Decision tree

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

- Parameters:
  - *max_depth*: maximum tree height
  - *min_impurity_decrease*: split nodes only if impurity decrease above threshold

Classification

- Fit training data

```
clf.fit(X, y)
```

  - X is the 2D Numpy array with input features
  - y is the target vector

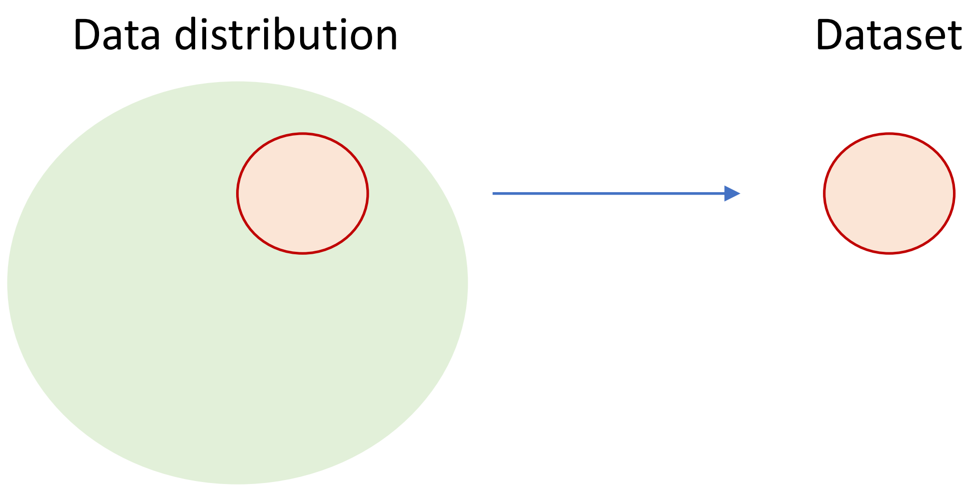
- Make predictions on new data (i.e. test set)

```
y_test_pred = clf.predict(X_test)
```

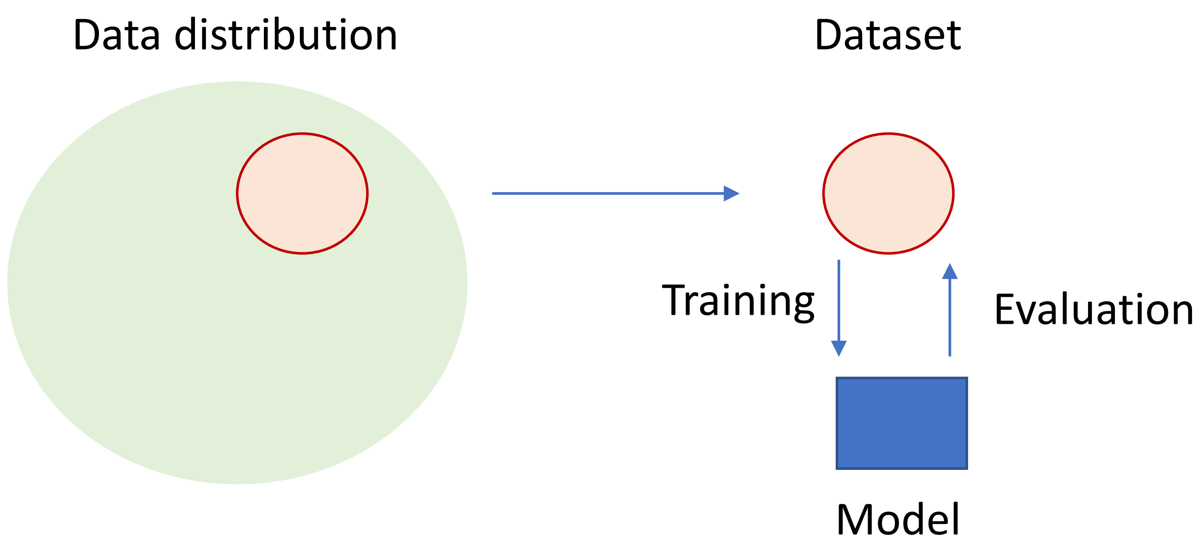
- To **choose** the most appropriate machine learning model for your data you have to **evaluate** its performances

- 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

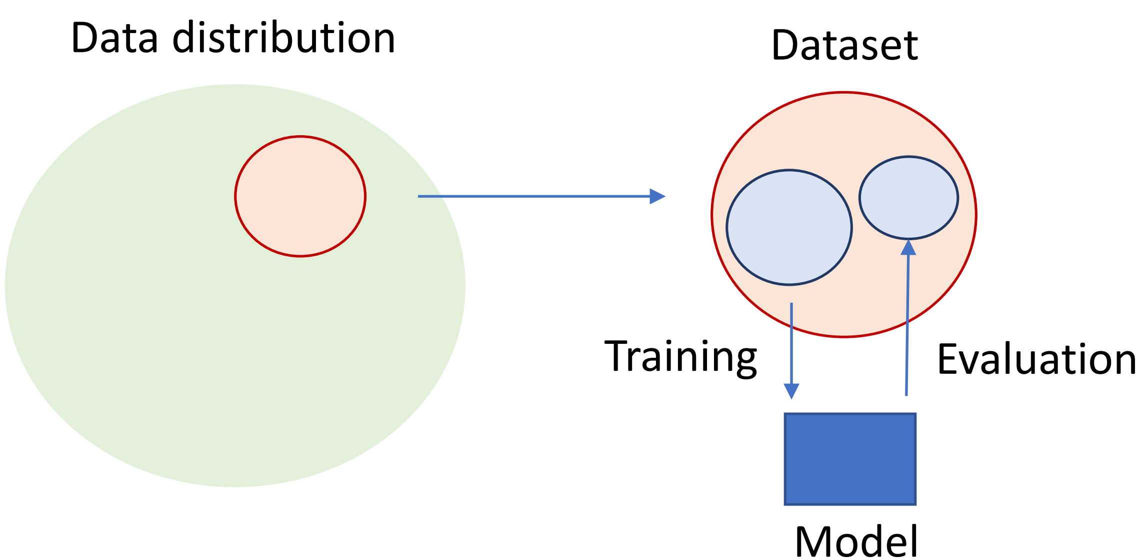


- If you choose the best model for your dataset, it may not perform so well for **new data**
  - This risk is called **overfitting**



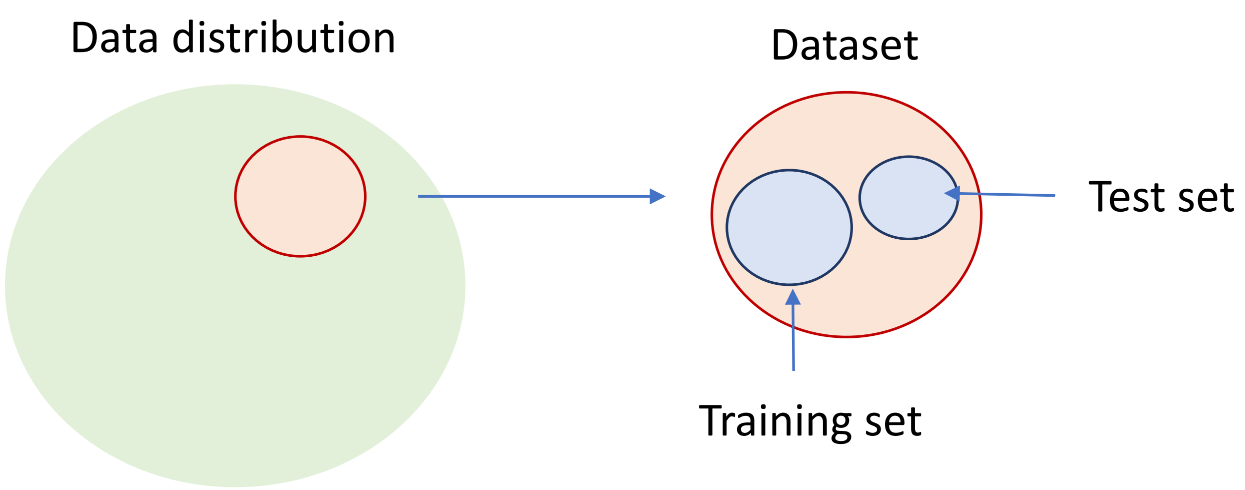
Classification

- 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



Classification

- This technique is called **hold-out**
  - Training set is typically 80/90% of your data

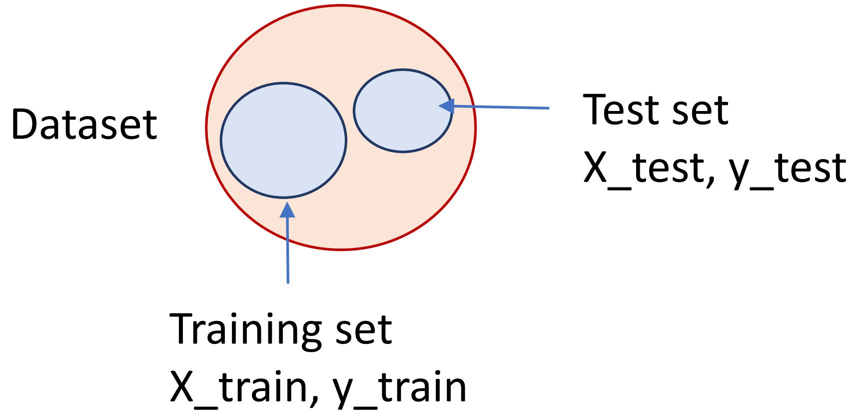


- Hold-out with Scikit-learn

```
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%)



- 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
  - F1(c): harmonic mean between precision and recall

- Evaluation metrics with Scikit-learn

```
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)
```

Classification

```python
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

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

- **Macro average** scores vs **Micro average** scores
  - **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 micro-averaged score is affected with the same weight as another with more samples

Micro average scores

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

- Micro average scores are computed by collecting all the TP, FP, TN, FN **independently of the class**
  - micro-p = (total_TP) / (total_TP + total_FP)
  - micro-r = (total_TP) / (total_TP + total_FN)
  - micro-f1 = micro-p = micro-r
- Classes with higher **cardinality** have **higher impact** on these metrics

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)
```

predicted

0 1 2

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

actual

0

1

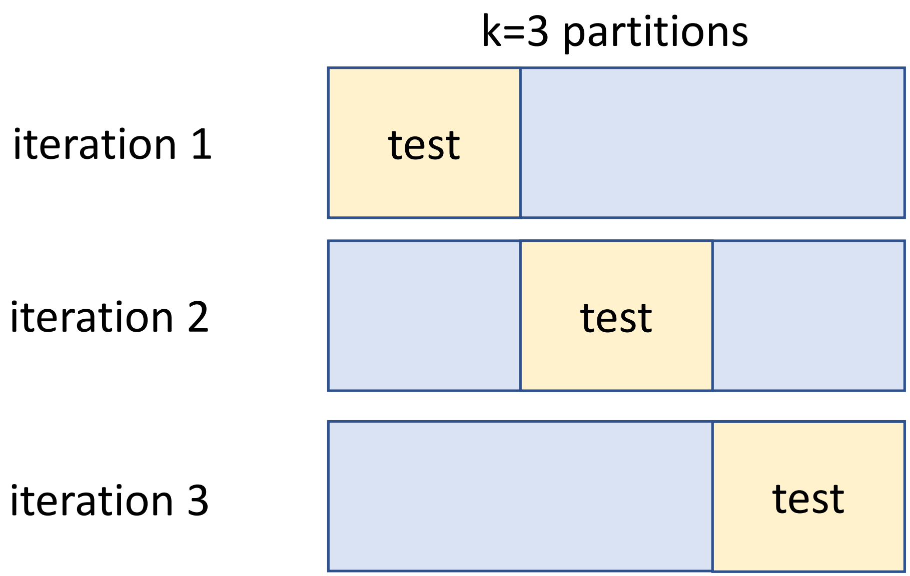
2

Notebook Examples

- **3b-Scikitlearn-Classification.ipynb**
  - **1. Classification and hold out**

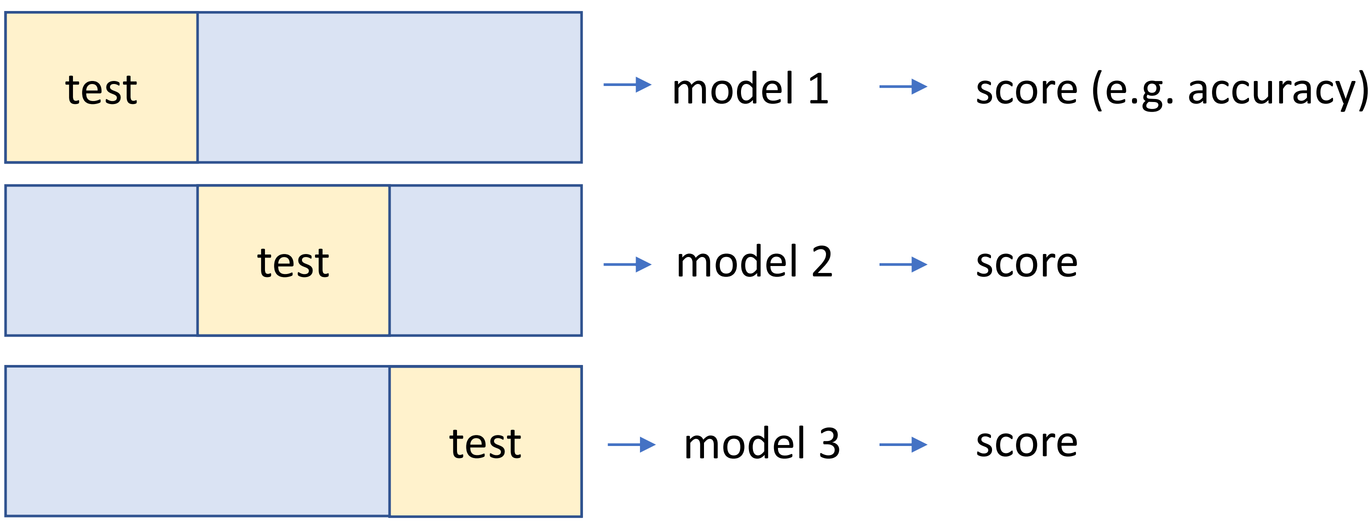
Cross-validation

- 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

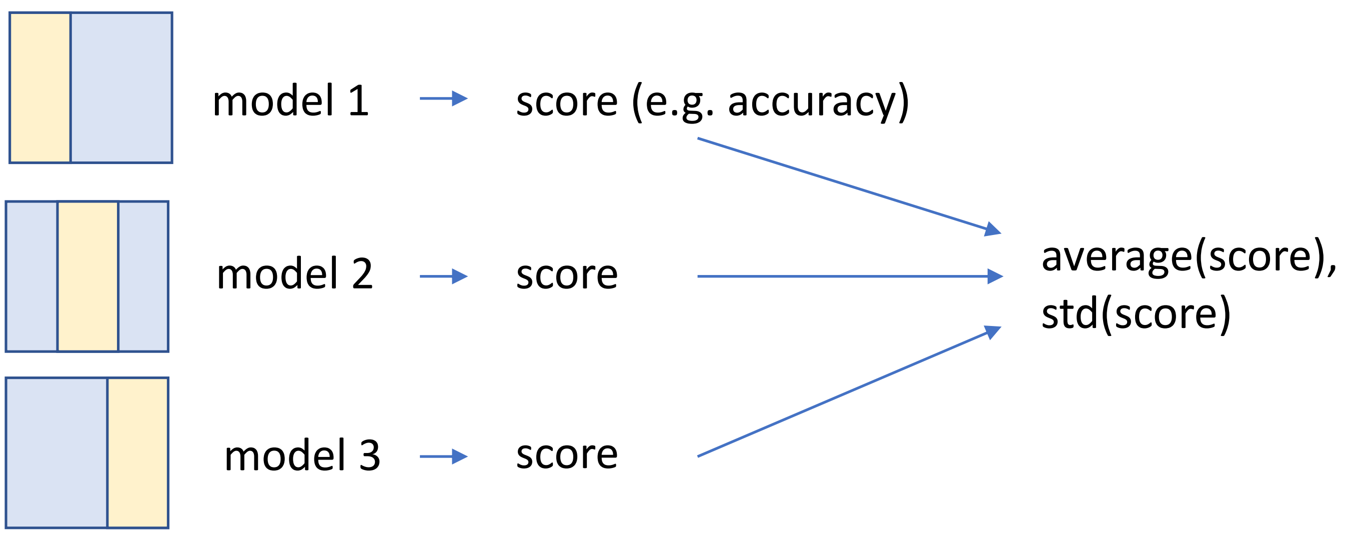


Cross-validation

- 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



Cross-validation

- 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

- 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 **lists**:
  - train_indices: list of the **indices** (row number) of the training samples
  - test_indices: list of the indices of the test samples

- 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

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:
  - clf = the model that you want to be trained
  - X, y = your dataset, where cross-validation will be performed

- 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')
```

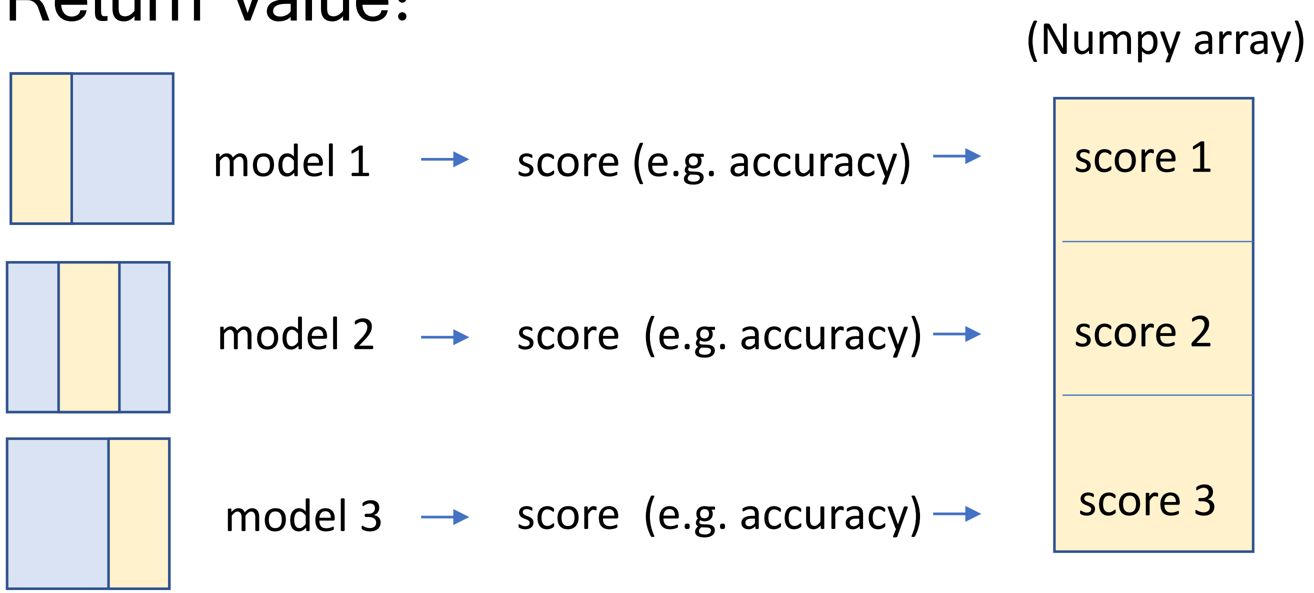
- Parameters:
  - cv = number of partitions for cross-validation
  - scoring = scoring function for the evaluation
    - E.g. ‘f1_macro’, 'f1_micro', ‘accuracy’, 'precision_macro'

- 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:



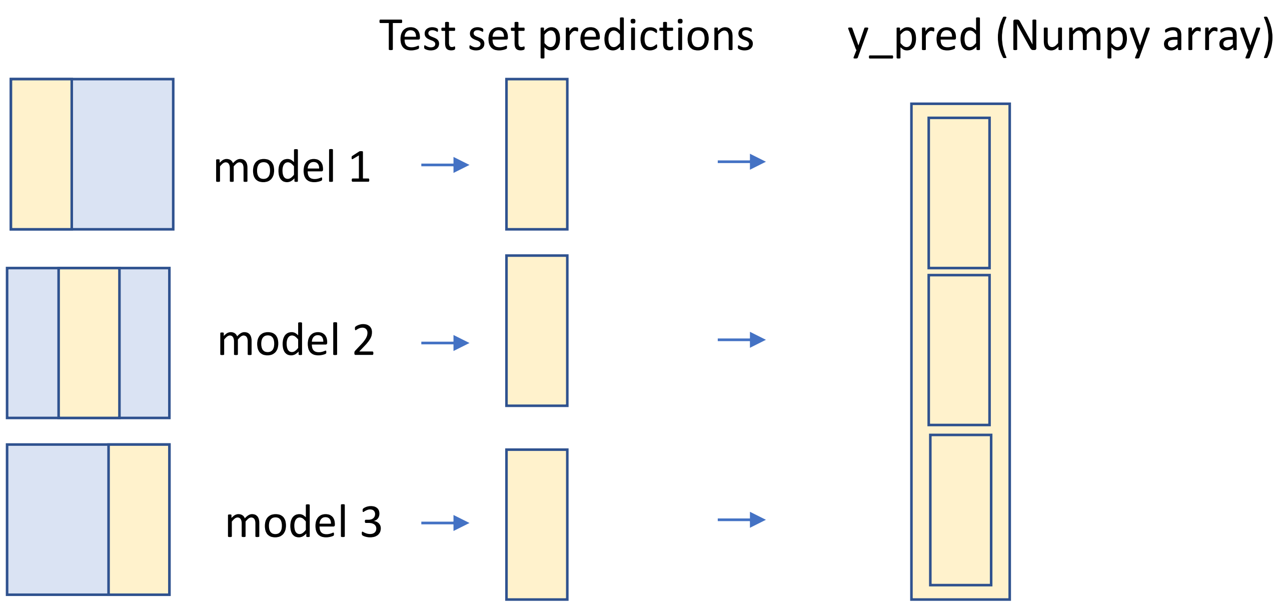
- Method 3: use cross_val_predict()

```
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

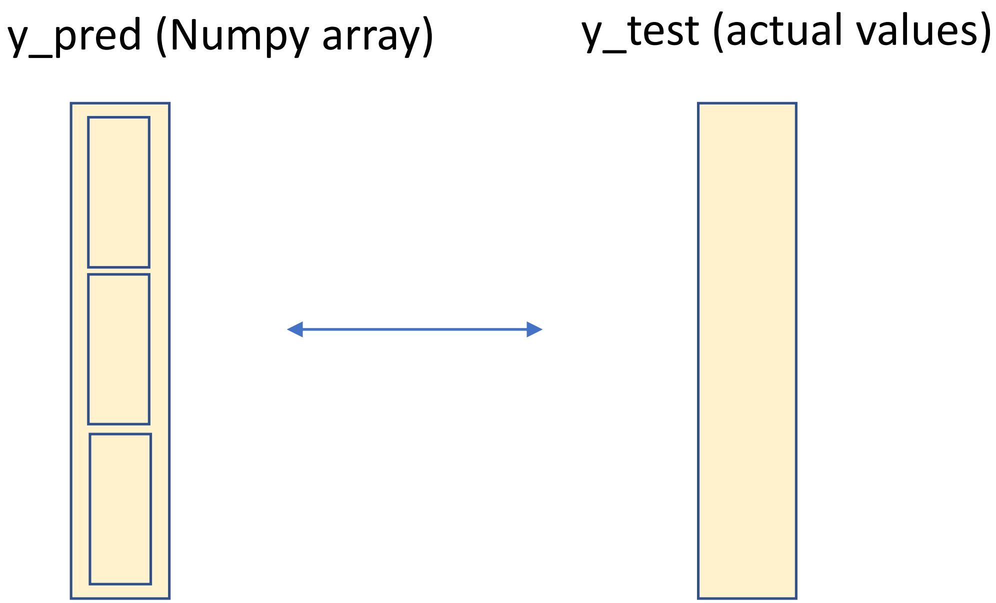
- Method 3: use cross_val_predict()

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

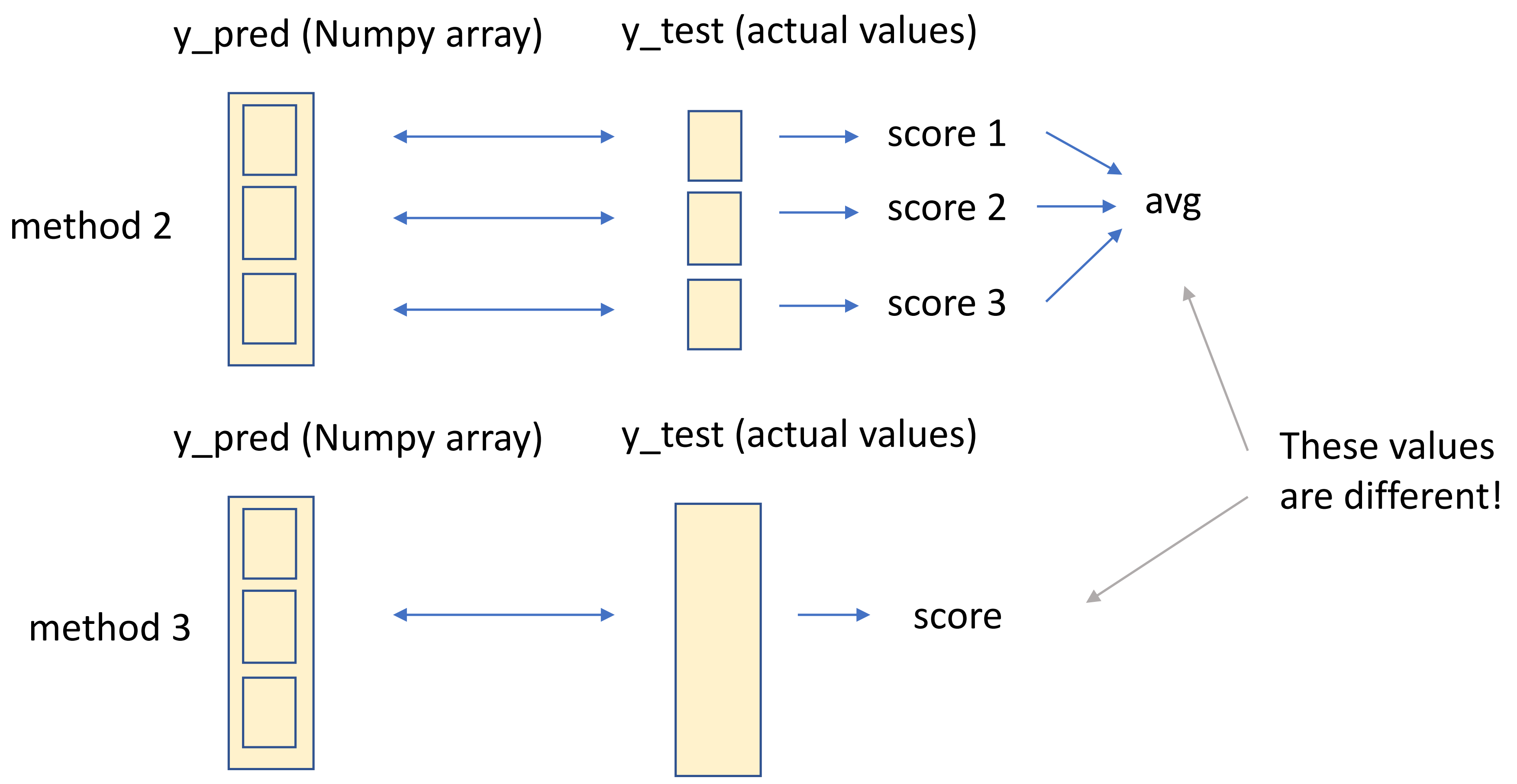


- Method 3: use cross_val_predict()
  - Finally you can evaluate the predictions

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



- Difference between method 2 and method 3



Notebook Examples

- **3b-Scikitlearn-Classification.ipynb**
  - **2. Cross validation**