OPENCOURSEWARE
ADVANCED PROGRAMMING
STATISTICS FOR DATA SCIENCE

Ricardo Aler

A Tutorial on Scikit Learn

What is Scikit Learn?

- It is the standard Python library for doing machine learning
  from sklearn import ...
- Collection of machine learning algorithms and tools in Python.
- BSD Licensed, used in academia and industry (Spotify, bit.ly, Evernote).
- ~20 core developers.
  - http://scikit-learn.org/stable/
- Other packages for Machine Learning in Python: Pylearn2, PyBrain, ...

The Machine Learning workflow

- Knowledge about the main ideas of Machine Learning / Statistical Learning is assumed
- The workflow:
  - Data preprocessing
  - Training:
    - Training the model
    - Hyper-parameter tuning
  - Model evaluation (holdout, crossvalidation)

The input: the dataset

- Datasets for sklearn are numpy numeric matrices:
  - This implies that categorical attributes/variables must be represented as:
    - Integers
    - One-hot-encoding / dummy variables
- However, there is a trend for integrating Pandas dataframes with scikit learn
- Missing values are represented as *np.nan*

Example of dataset: iris

- It is a dataset for classification of plants
  - Attributes / features:
    - ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
  - Response variable: type of plant:
    - ['setosa', 'versicolor', 'virginica']

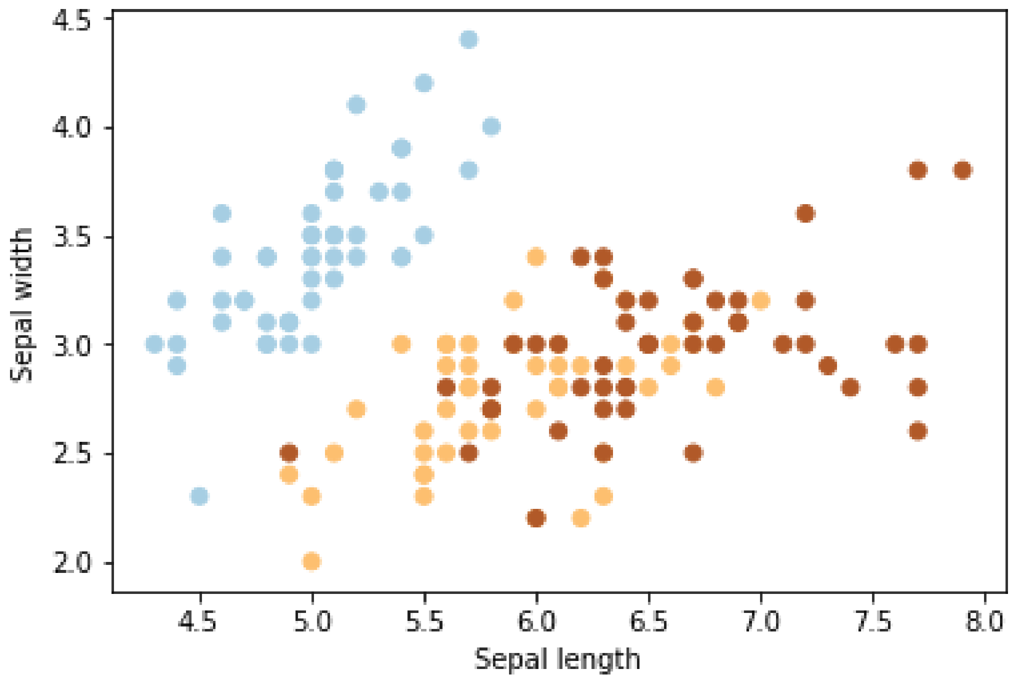
Example of dataset: iris

```
In [46]: # Sklearn already contains some
datasets
In [47]: from sklearn.datasets import load_iris
In [48]: iris = load_iris()
In [49]: print(iris.feature_names)
['sepal length (cm)', 'sepal width (cm)', 'petal
length (cm)', 'petal width (cm)']
In [50]: print(iris.target_names)
['setosa' 'versicolor' 'virginica']

In [51]: # The actual data is a numpy matrix
In [52]: X = iris.data
In [53]: y = iris.target

In [56]: print(type(X))
<class 'numpy.ndarray'>
In [59]: print(type(y))
<class 'numpy.ndarray'>
```

```
In [54]: # Those are the input attributes
In [55]: print(X[:10,])
[[5.1 3.5 1.4 0.2]
[4.9 3. 1.4 0.2]
[4.7 3.2 1.3 0.2]
[4.6 3.1 1.5 0.2]
[5. 3.6 1.4 0.2]
[5.4 3.9 1.7 0.4]
[4.6 3.4 1.4 0.3]
[5. 3.4 1.5 0.2]
[4.4 2.9 1.4 0.2]
[4.9 3.1 1.5 0.1]]

In [57]: #And this is the response variable column')
In [58]: print(y[:10,])
[0 0 0 0 0 0 0 0 0 0]
```

Example of dataset: iris

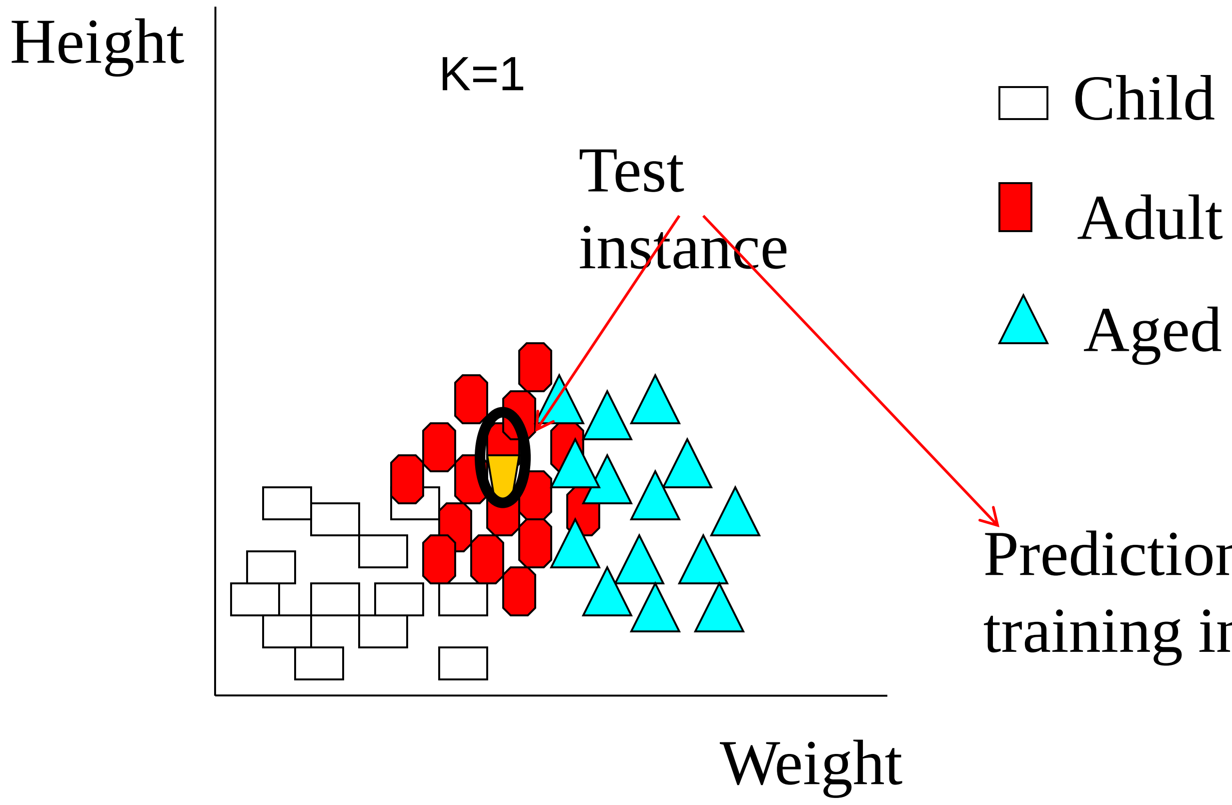
```
plt.scatter(X[:, 0], X[:, 1], c=y,
cmap=plt.cm.Paired)
plt.xlabel('Sepal length')
plt.ylabel('Sepal width')
plt.show()
```



Models

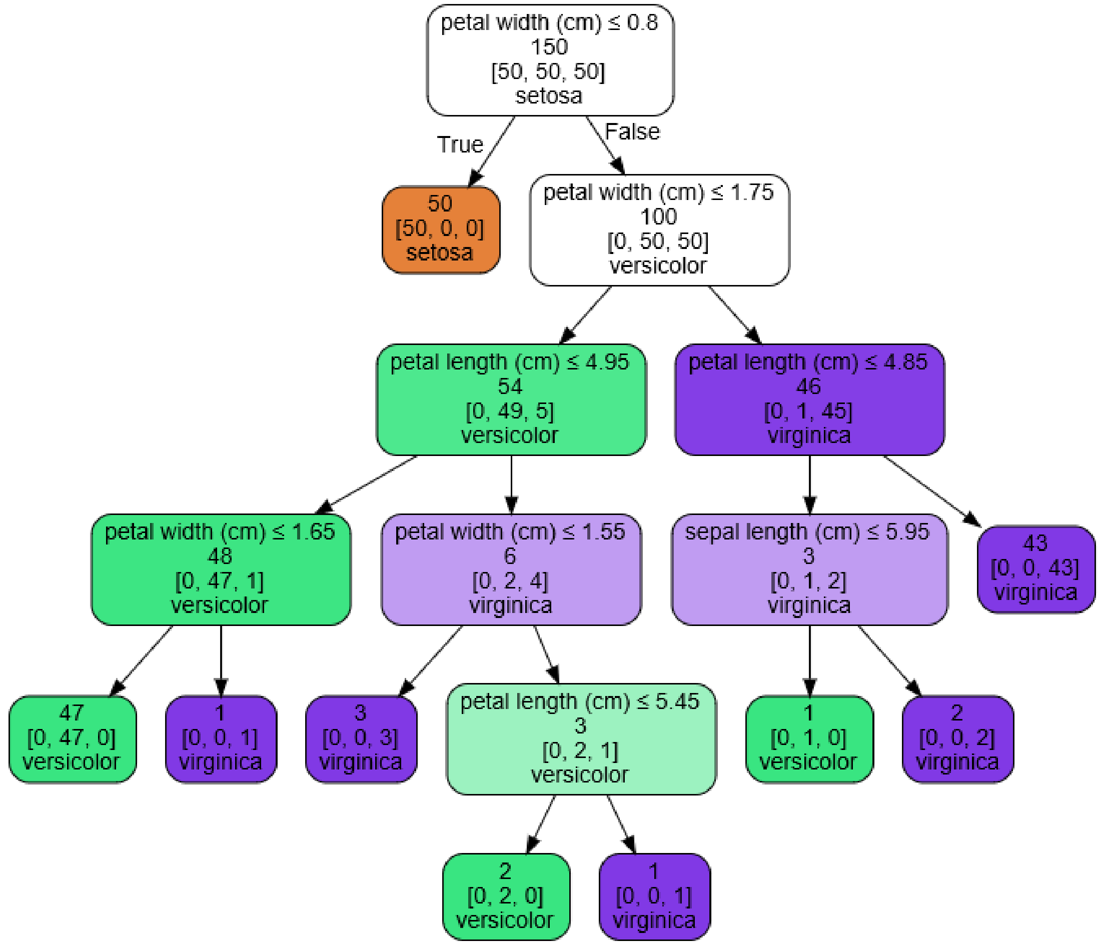
- There are many types of models
- We already know KNN (k-nearest neighbour)
- There are more:
  - Trees
  - Ensembles: bagging (random forests, gradient boosting, stacking)
  - Functions: neural networks, support vector machines, ...

Models: k-nearest neighbor



Prediction = Adult (closest training instance)

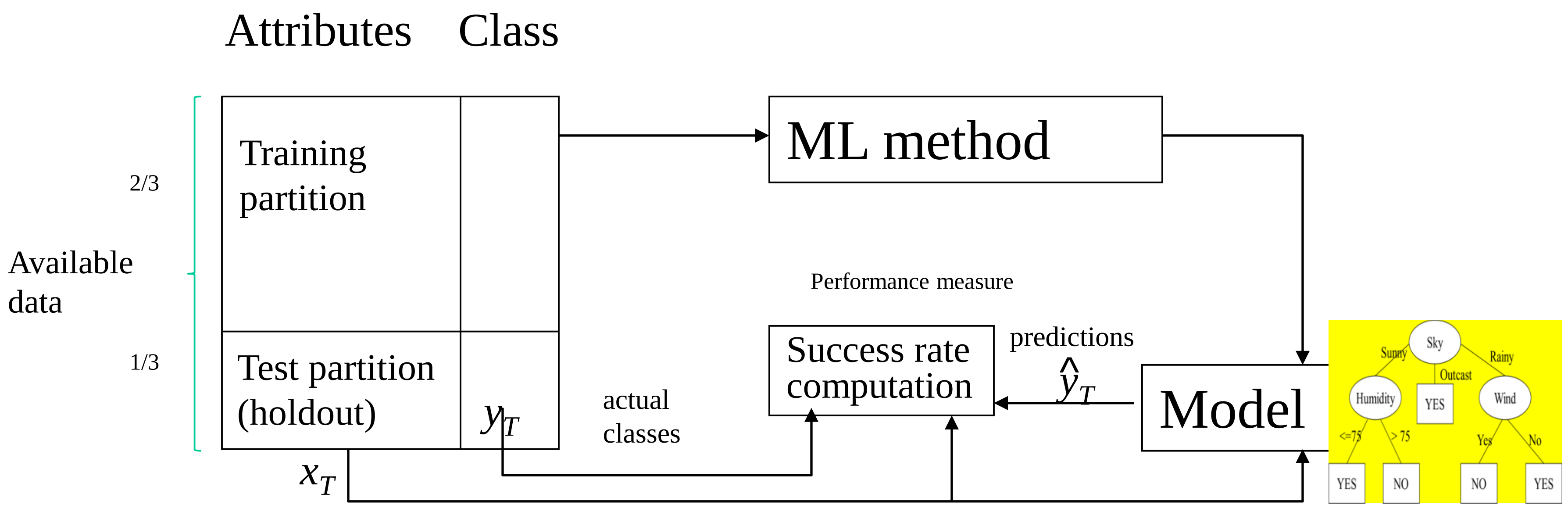
Models: decision tree



Training a decision tree

```
In [93]: from sklearn import tree
# Here, we define the type of training method (nothing happens yet)
In [94]: clf = tree.DecisionTreeClassifier()
# Now, we train (fit) the method on the (X,y) dataset
In [95]: clf = clf.fit(X, y)
# clf contains the trained model
In [96]: clf
Out[96]:
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
max_features=None, max_leaf_nodes=None,
min_impurity_decrease=0.0, min_impurity_split=None,
min_samples_leaf=1, min_samples_split=2,
min_weight_fraction_leaf=0.0, presort=False, random_state=None,
splitter='best')
```

Training and evaluating models with a test partition (holdout)



Rule: never evaluate a model with the same data used for training it

Training and evaluating models with a test partition (holdout)

- First, we create the train / test partitions

```
In []: from sklearn.model_selection import train_test_split
In []: from sklearn import preprocessing

# train_test_split creates the train and test partitions, respectively
# random_state = 33 is for reproducibility purposes
# 0.33 = 1/3 is the proportion of data for testing (67% = 2/3 for training)
In []: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33,
random_state=33)

In []: print(X_train.shape, y_train.shape)
(112, 4) (112,)
In []: print(X_test.shape, y_test.shape)
(38, 4) (38,)
```

Estimating performance (evaluation) with a test partition (holdout)

- Then, we train the model with fit, get predictions on the test set with predict, and compute the performance of the model

```
In []: from sklearn import metrics
In []: from sklearn import tree


In []: clf = tree.DecisionTreeClassifier()
# Making results reproducible
In []: np.random.seed(0)
In []: clf.fit(X_train, y_train)
In []: y_test_pred = clf.predict(X_test)
In []: print(y_test_pred)
[1 1 0 1 1 2 0 0 2 2 2 0 2 1 2 1 1 0 1 2 0 0 2 0 1 1 1 1 2 2 1 1 2 2 2 2 2 1]
In []: print(y_test)
[1 1 0 1 2 2 0 0 2 2 2 0 2 1 2 1 2 0 1 2 0 0 2 0 2 2 1 1 2 2 1 1 2 2 2 2 2 1]

In []: print(metrics.accuracy_score(y_test, y_test_pred))
0.8947368421052632
```

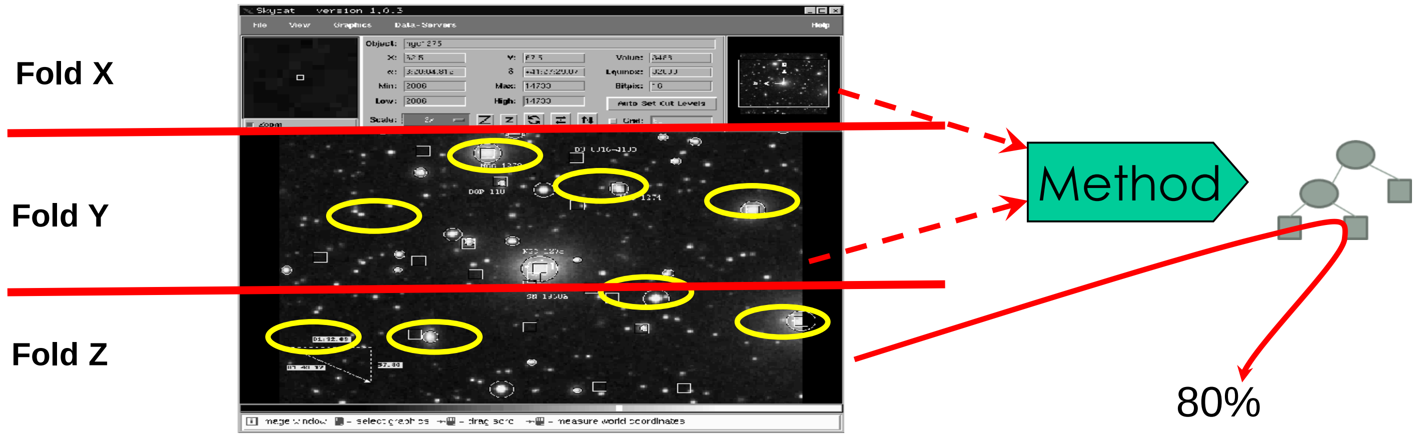
Crossvalidation

- The available data is divided into k folds (k partitions). With k=3, three partitions X, Y, and Z.
- The process has k steps (3 in this case):
  - Learn model with X, Y, and test it with Z (T1 = success rate on Z)
  - Learn model with X, Z, and test it with Y (T2 = success rate on Y)
  - Learn model with Y, Z and test it with X (T3 = success rate on X)
  - Success rate TX = (T1+T2+T3)/3
- The final classifier CF is learned **from the whole dataset (X, Y, Z)**. It is assumed that T is a good estimation of the success rate of CF
- k=10 is commonly used. K between 5 and 10 are recommended.

3-fold cross-validation evaluation

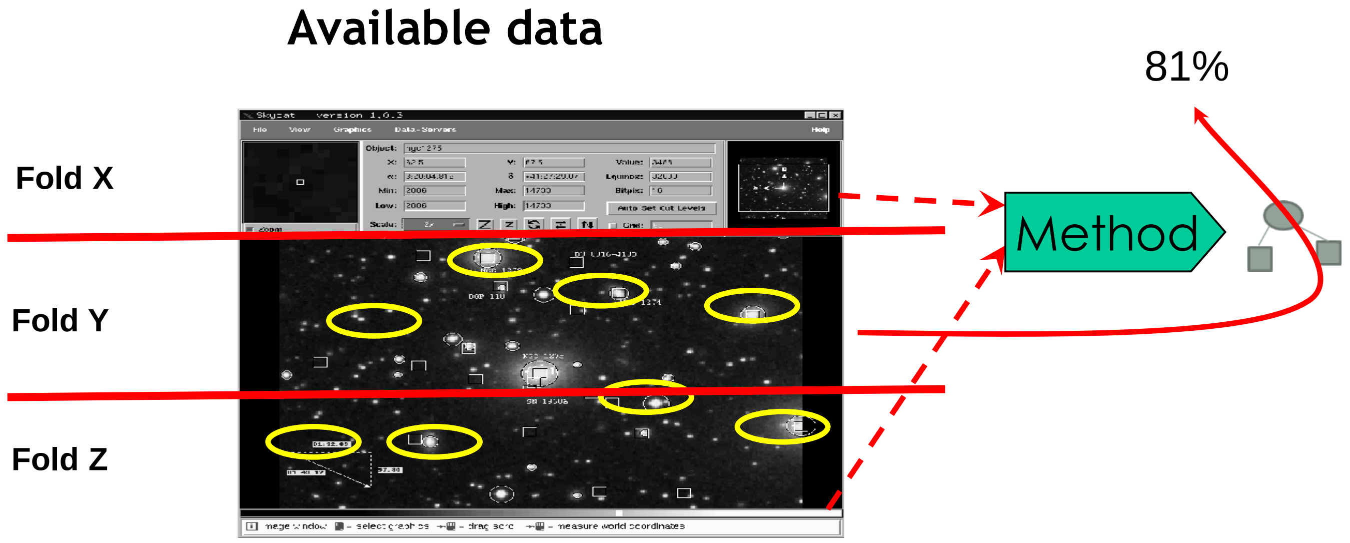
<u>Train with X and Y, evaluate with Z</u>

Available data



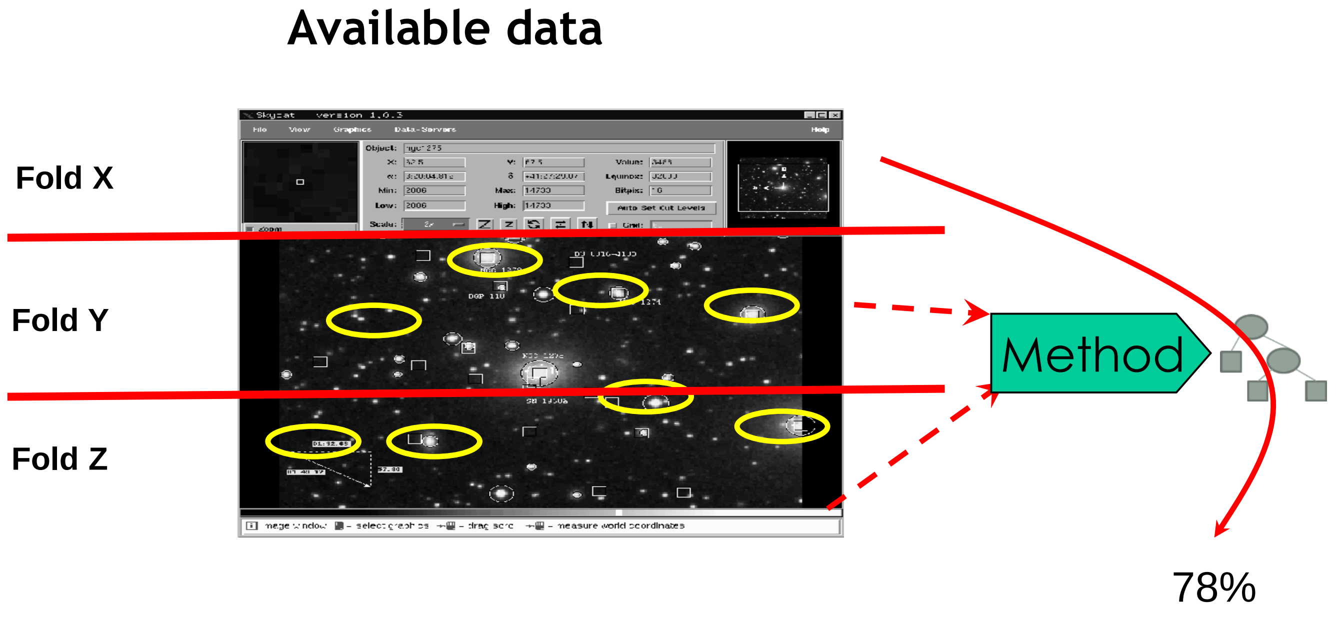
3-fold cross-validation evaluation

<u>Train with X, Z; evaluate with Y</u>



3-fold cross-validation evaluation

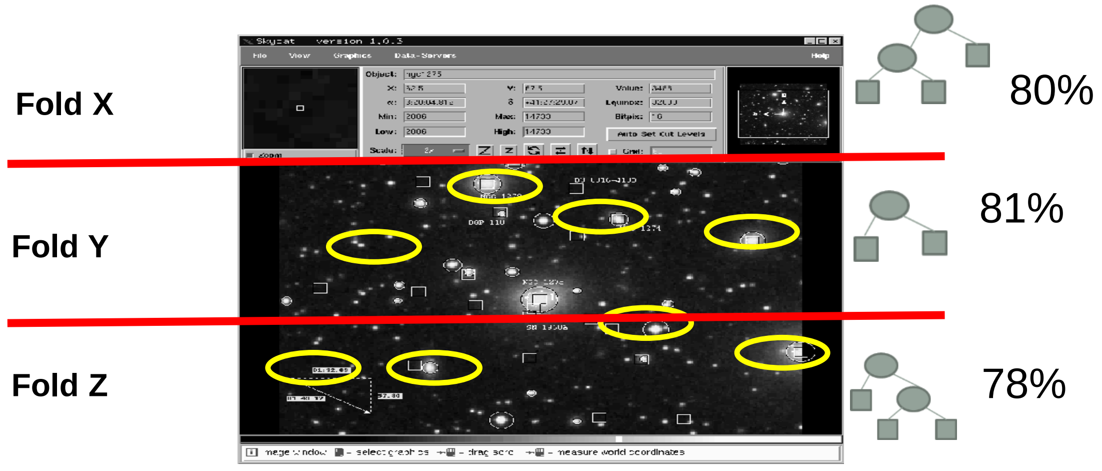
Train with Y, Z; evaluate with X



3-fold cross-validation evaluation

<u>The estimation of future performance T is the average of the three folds.</u>

Available data

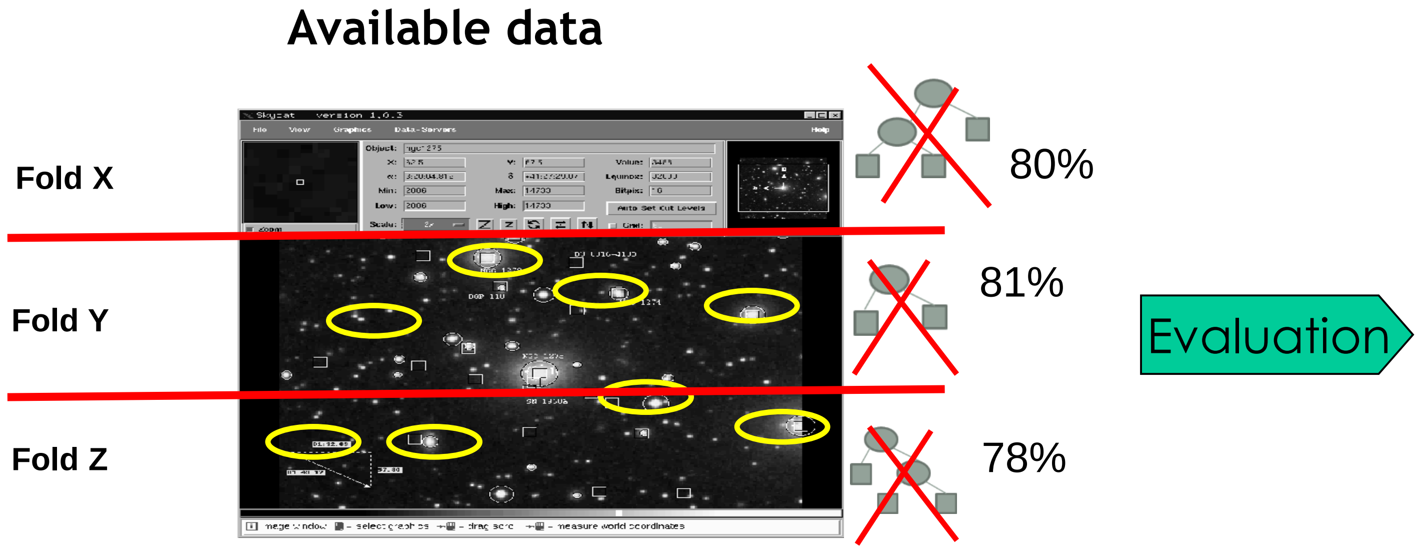


Evaluation

T=
(80%+81%+78%)/3
= 79.7%

3-fold cross-validation evaluation

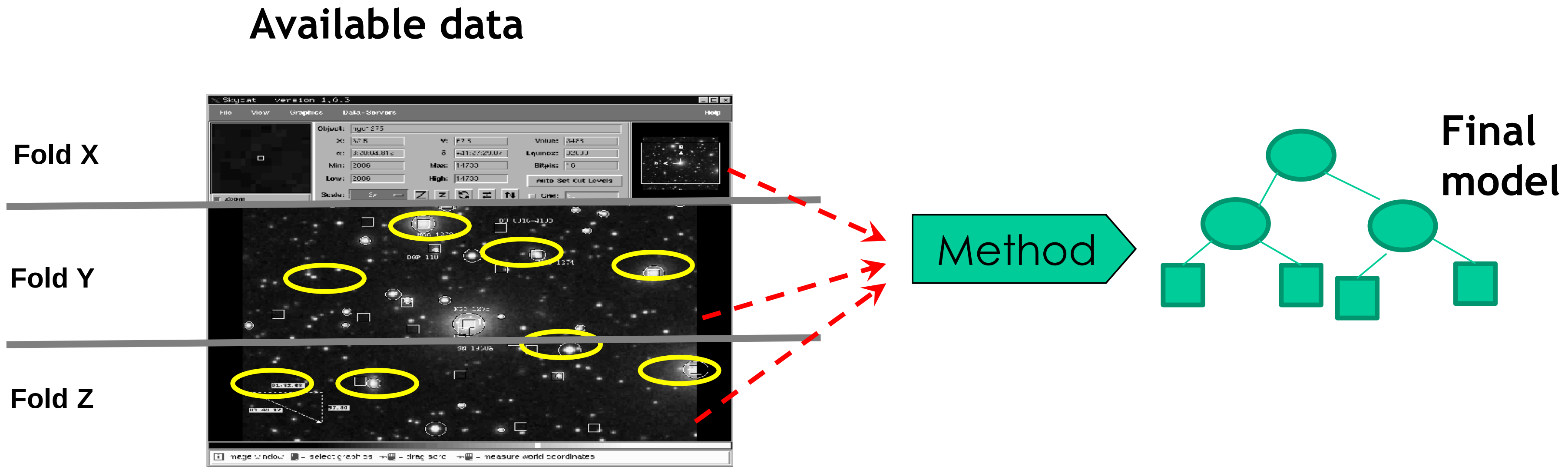
<u>Once T has been computed, the three models used to compute it are discarded and …</u>



T= (80%+81%+78%)/3 = 79.7%

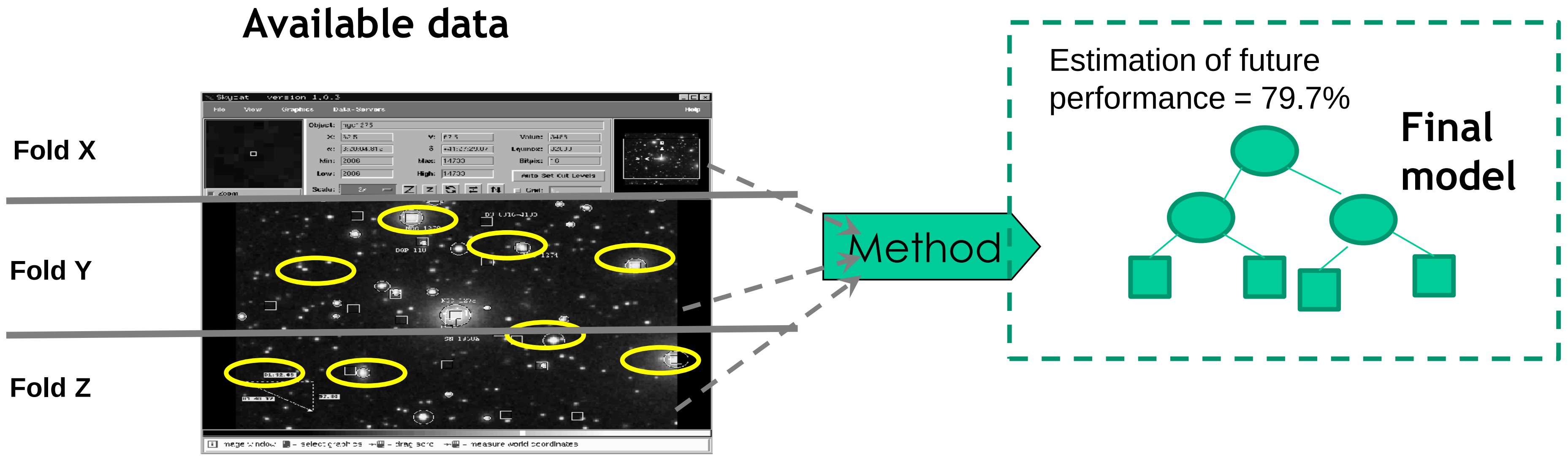
3-fold cross-validation evaluation

- A final model is trained with the complete dataset



3-fold cross-validation evaluation

- A final model is trained with the complete dataset
- The estimation of future performance computed previously is kept (79.7%)
- Again, this is considered a **pesimistic estimation**, because the data partitions used to compute it were smaller (2/3) than the dataset used to train the final model.

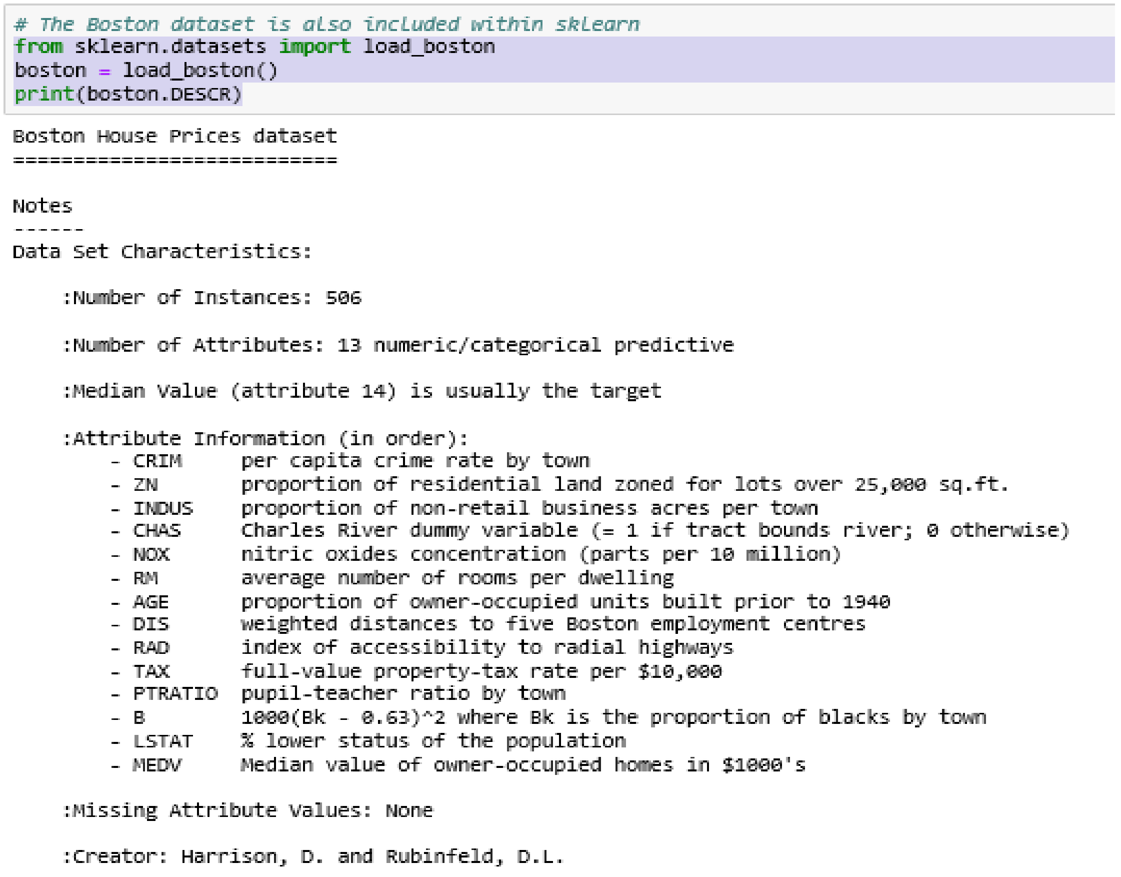


Estimating performance (evaluation) with crossvalidation

```
In []: from sklearn.model_selection import cross_val_score, KFold

# create a k-fold crossvalidation iterator of k=5 folds
# shuffle = True randomly rearranges the dataframe
# random_state = 0 is for making the folds reproducible
In []: cv = KFold(n_splits=5, shuffle=True, random_state=0)
In []: clf = tree.DecisionTreeClassifier()

# Making results reproducible
In []: np.random.seed(0)

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

# Printing the 10 scores
In []: print(scores)
[1.0 0.9 -1. -0.93333333 -0.93333333]

# Printing the average score and the standard deviation
In []: from scipy.stats import sem # Standard deviation
In []: print("Mean score: {0:.3f} (+/-{1:.3f})".format(scores.mean(), sem(scores)))
Mean score: -0.953 (+/-0.020)
```

Exercise: regression

- We are going to use the Boston dataset, about predicting house prices



```
# The Boston dataset is also included within sklearn
from sklearn.datasets import load_boston
boston = load_boston()
print(boston.DESCR)
```

```
Boston House Prices dataset
===========================

Notes
------
Data Set Characteristics:

    :Number of Instances: 506

    :Number of Attributes: 13 numeric/categorical predictive

    :Median Value (attribute 14) is usually the target

    :Attribute Information (in order):
        - CRIM     per capita crime rate by town
        - ZN       proportion of residential land zoned for lots over 25,000 sq.ft.
        - INDUS    proportion of non-retail business acres per town
        - CHAS     Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
        - NOX      nitric oxides concentration (parts per 10 million)
        - RM       average number of rooms per dwelling
        - AGE      proportion of owner-occupied units built prior to 1940
        - DIS      weighted distances to five Boston employment centres
        - RAD      index of accessibility to radial highways
        - TAX      full-value property-tax rate per $10,000
        - PTRATIO  pupil-teacher ratio by town
        - B        1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
        - LSTAT    % lower status of the population
        - MEDV     Median value of owner-occupied homes in $1000's

    :Missing Attribute Values: None

    :Creator: Harrison, D. and Rubinfeld, D.L.
```

```
# Getting the data
from sklearn.datasets import load_boston
boston = load_boston()
print(boston.DESCR)

X = boston.data
y = boston.target
```

Exercise: regression

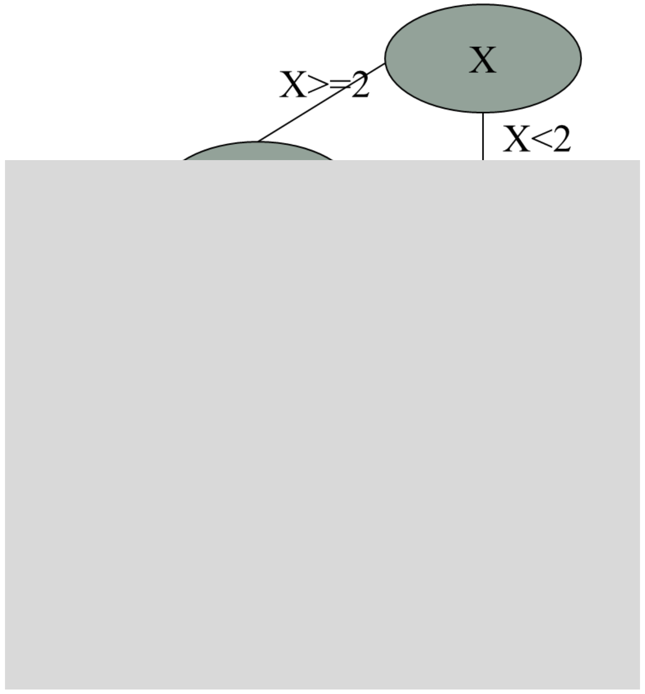
- Use train (75%)/test (25%) for training / evaluating a decision tree **regression** model:
  - tree.DecisionTreeRegressor()
  - metrics.mean_squared_error
- Do the same with KNN:
  - KNeighborsRegressor
  - find it yourself in the scikit docs (https://scikit-learn.org/)
- Now, do the evaluation with 5-fold crossvalidation:
  - scoring='neg_mean_squared_error',

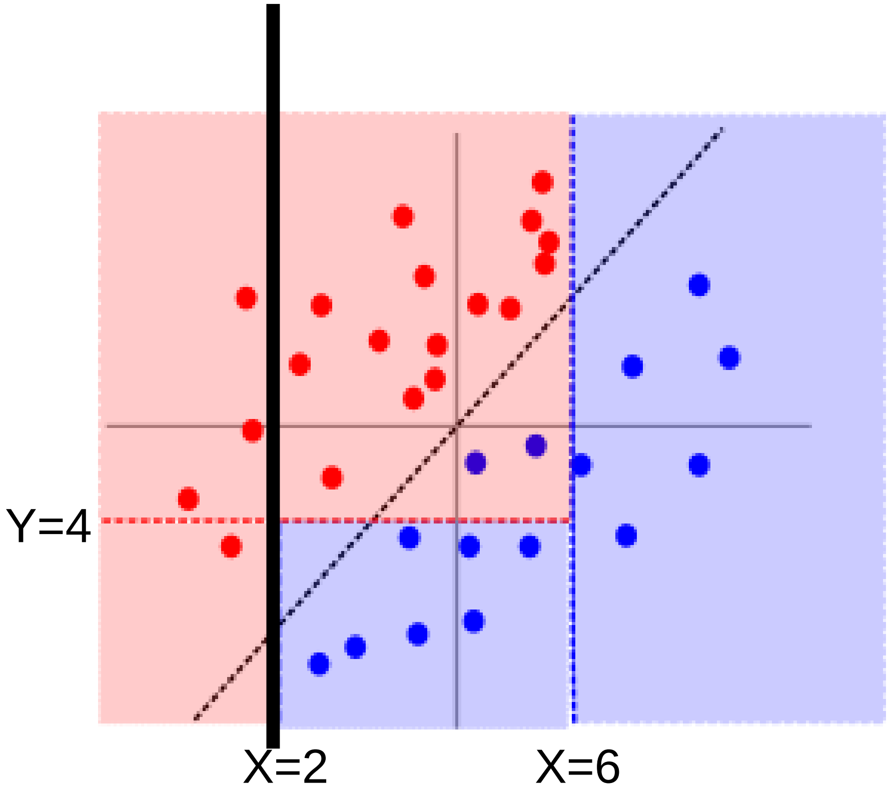
Hyper-parameters

- All machine learning methods have hyper-parameters that control their behavior
- For example, KNN has K = number of neighbors:
  - *n_neighbors*
- For example, decision trees have (at least):
  - *max_depth*
  - *min_samples_split*

MAX-DEPTH HYPER-PARAMETER FOR DECISION TREES

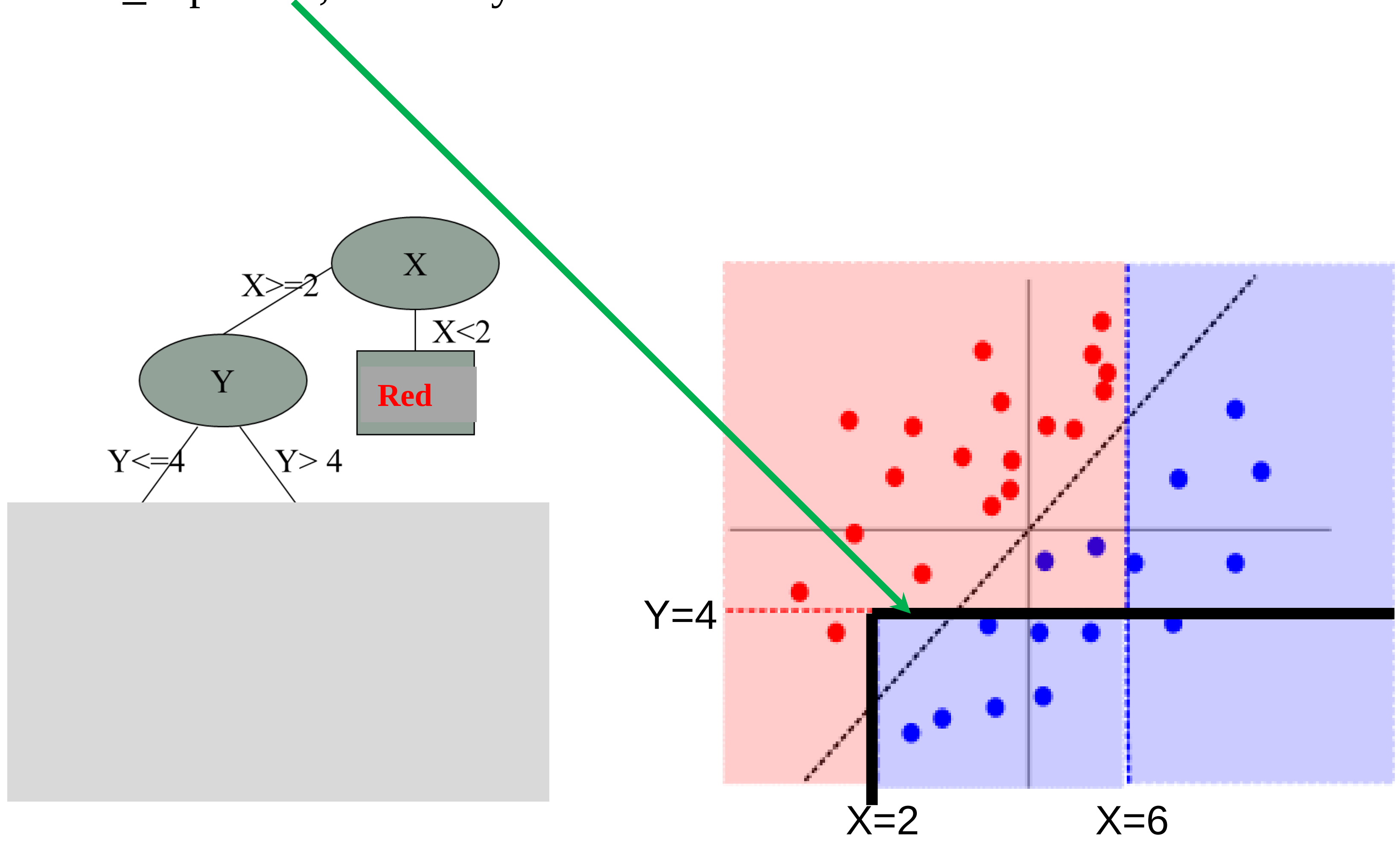
- With max_depth = 1, boundary is a line.





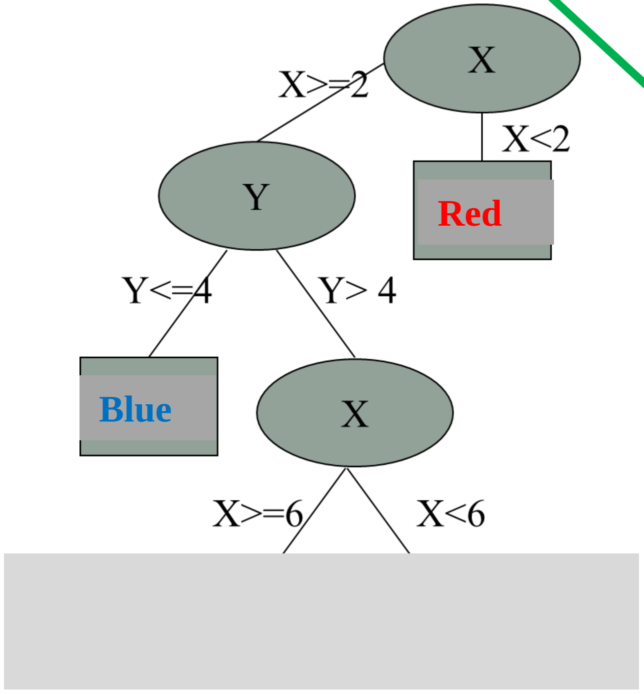
MAX-DEPTH HYPER-PARAMETER FOR DECISION TREES

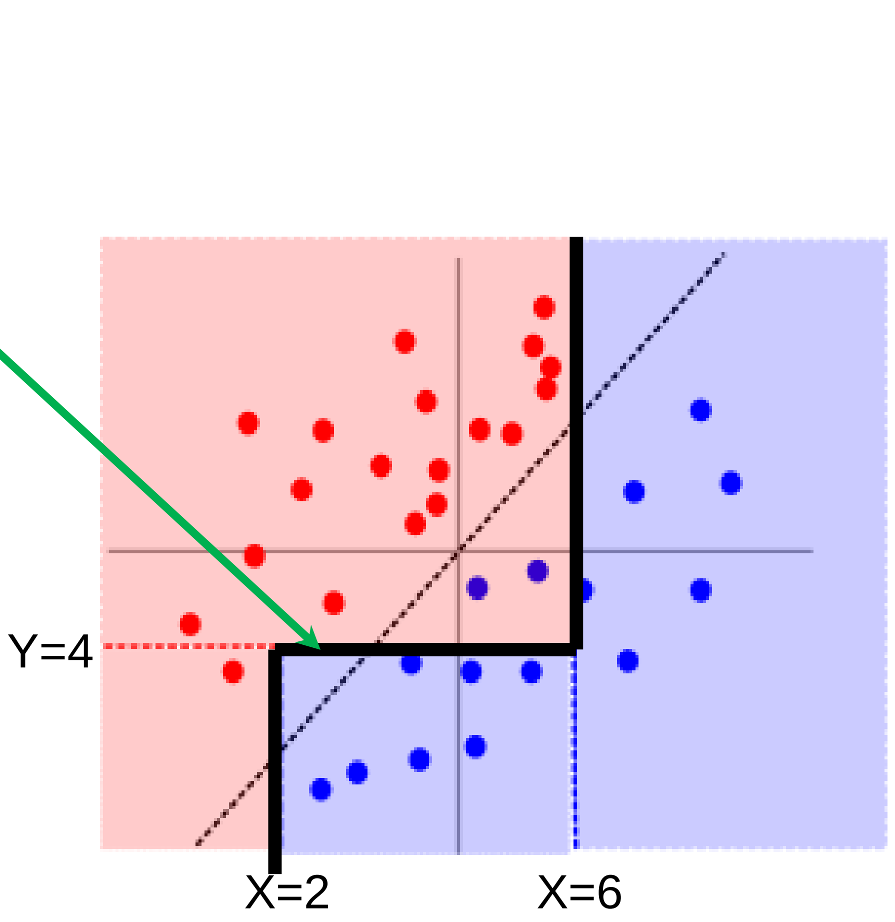
- With max_depth = 2, boundary is non-linear



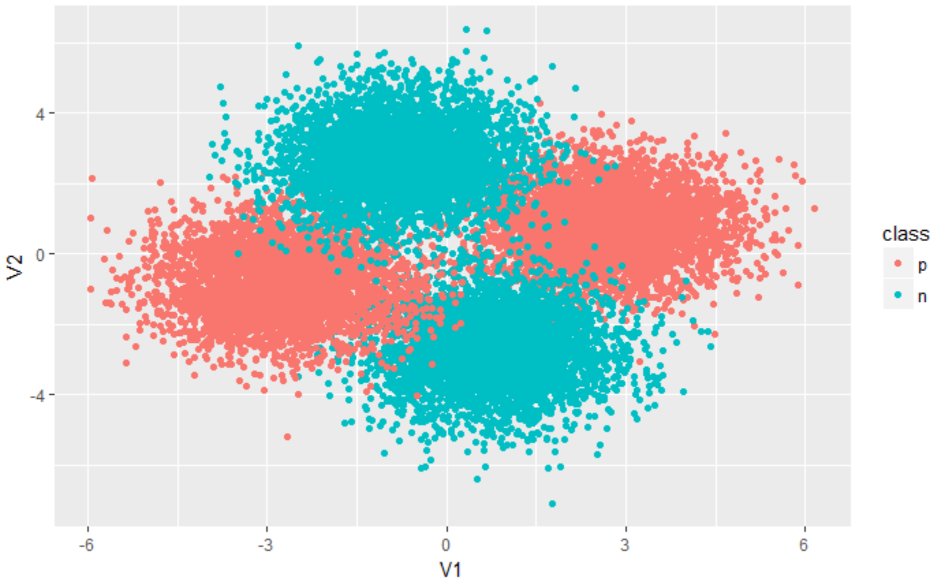
MAX-DEPTH HYPER-PARAMETER FOR DECISION TREES

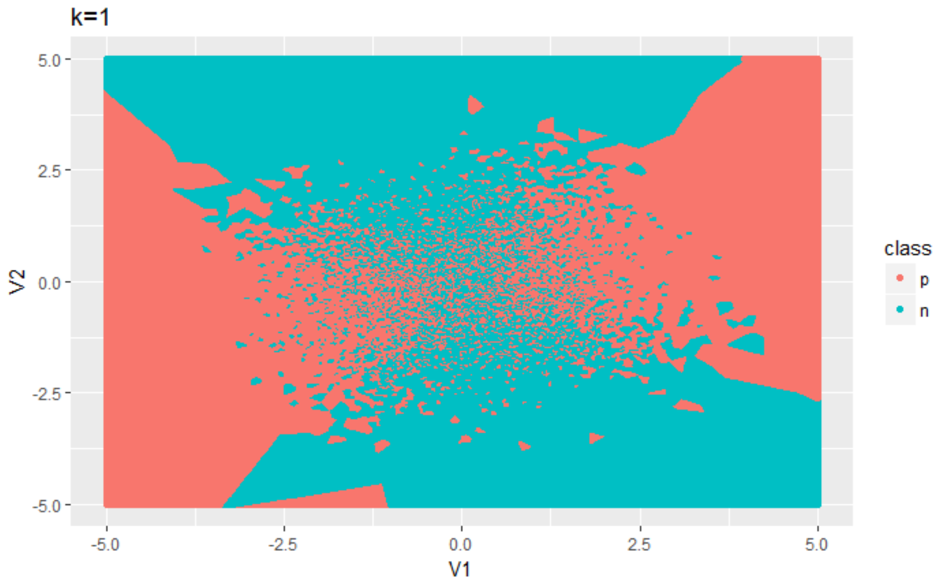
- With max_depth = 3, boundary is non-linear and more complex than with max_depth = 2
- And so on

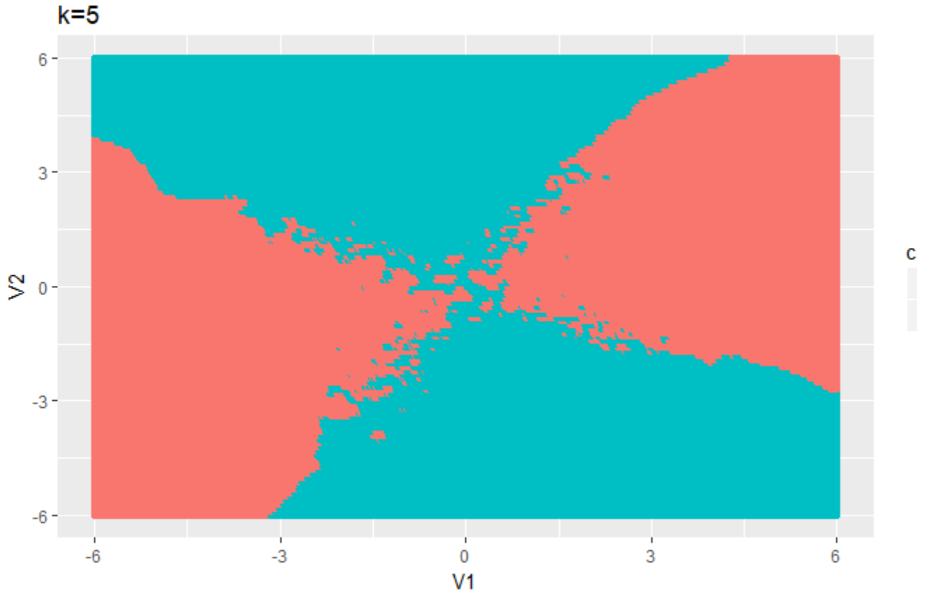


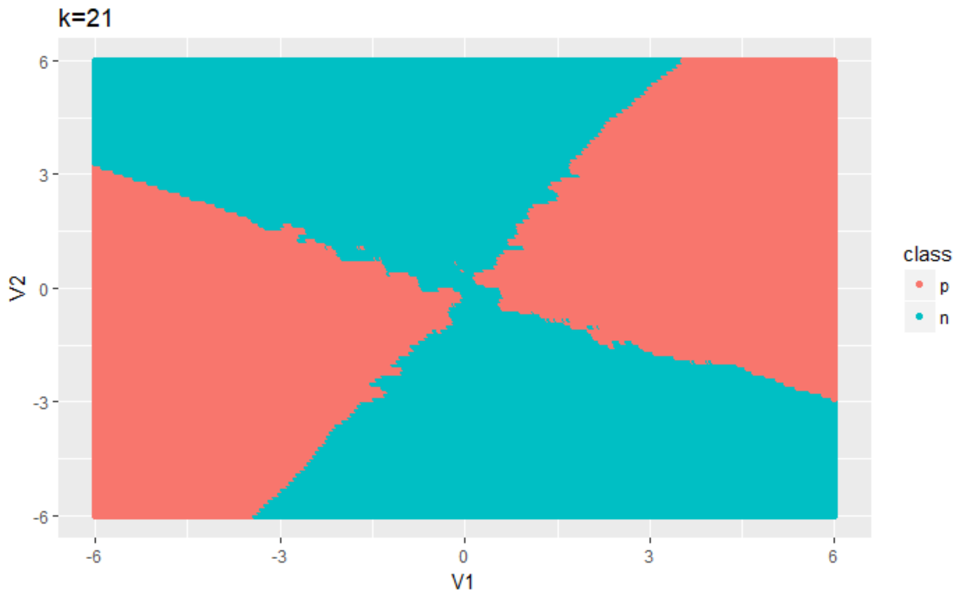


NUMBER OF NEIGHBORS IN KNN









Hyper-parameters

- It is possible to set them by hand when the method is defined:

```
In [191]: clf = tree.DecisionTreeClassifier()
In [192]: clf
Out[192]:
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None, max_features=None,
max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, min_samples_leaf=1,
min_samples_split=2, min_weight_fraction_leaf=0.0, presort=False, random_state=None,
splitter='best')

In [193]: clf = tree.DecisionTreeClassifier(max_depth=4)
In [194]: clf
Out[194]:
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=4, max_features=None,
max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, min_samples_leaf=1,
min_samples_split=2, min_weight_fraction_leaf=0.0, presort=False, random_state=None, splitter='best')
```

Automatic Hyper-parameter tuning

- If there is more than one hyper-parameter, **grid search** is typically used.
- All possible combinations of hyper-parameters is systematically evaluated.
- Computationally expensive.

Grid search

MAX_DEPTH	2	4	6	8
MIN_SAMPLES				
2	(2,2)	(2,4)	(2,6)	(2,8)
4	(4,2)	(4,4)	(4,6)	(4,8)
6	(6,2)	(6,4)	(6,6)	(6,8)

Grid search means: try all possible combinations of values for the two (or more) hyper-parameters. For each one, carry out a train/validation or a crossvalidation, and obtain the success rate.

MAX_DEPTH	2	4	6	8
MIN_SAMPLES				
2	70%	75%	76%	68%
4	72%	73%	**81%**	70%
6	68%	70%	71%	67%

Grid search

```
for(maxdepth in c(2,4,6,8)){
  for(minsplit in c(2,4,6)){
    model = train(train_set, maxdepth, minsplit)
    evaluation = "evaluate model with validation
set"
  }
}
"Return (maxdepth, minsplit) of model with best
evaluation"
```

Random search

maxdepth	2	4	6	8
minsplit				
2	(2,2)	(2,4)	(2,6)	(2,8)
4	(4,2)	(4,4)	(4,6)	(4,8)
6	(6,2)	(6,4)	(6,6)	(6,8)

Random search: test **randomly** only some of the combinations (Budget=4, in this case).

maxdepth	2	4	6	8
minsplit				
2	70%	**75%**	76%	68%
4	72%	73%	81%	70%
6	68%	70%	71%	67%

Random search

```
budget = 100 # budget is the maximum amount of hyper-parameter values to try
while(budget>0){
   budget = budget – 1 # Decrease budget
   (maxdepth, minsplit) = “get a random combination of hiper-parameter values”
    model = train(train_set, maxdepth, minsplit)
    evaluation <- "evaluate model with validation set"
  }}
"Return (maxdepth, minsplit) of model with best evaluation"
```

Automatic Hyper-parameter tuning

- In general, hyper-parameter tuning is a search in a parameter space for a particular **machine learning method** (or estimator). Therefore, it is necessary to define:
  - The **search space** (the hyper-parameters of the method and their allowed values)
  - The **search method**: so far, grid-search or random-search, but there are more (such as model based optimization)
  - The **evaluation method**: basically, validation set (holdout) or crossvalidation
  - The **performance measure** (or score function): missclassification error, balanced accuracy, RMSE, …

Defining the search space for grid-search

- For grid search, we must specify the list of actual values to be checked:

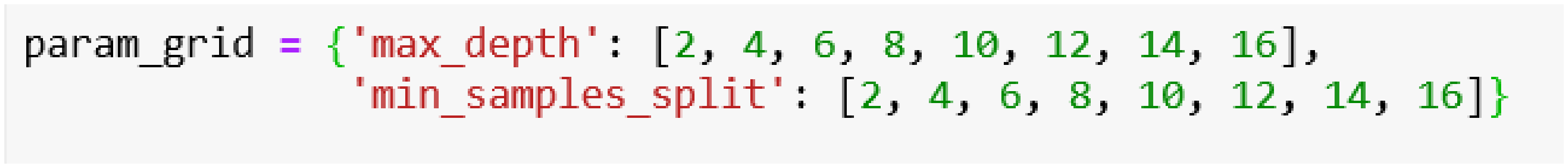
```
param_grid = {'max_depth': [2, 4, 6, 8, 10, 12, 14, 16],
              'min_samples_split': [2, 4, 6, 8, 10, 12, 14, 16]}
```

- Equivalently:

```
# Search space
param_grid = {'max_depth': list(range(2,16,2)),
              'min_samples_split': list(range(2,16,2))}
```

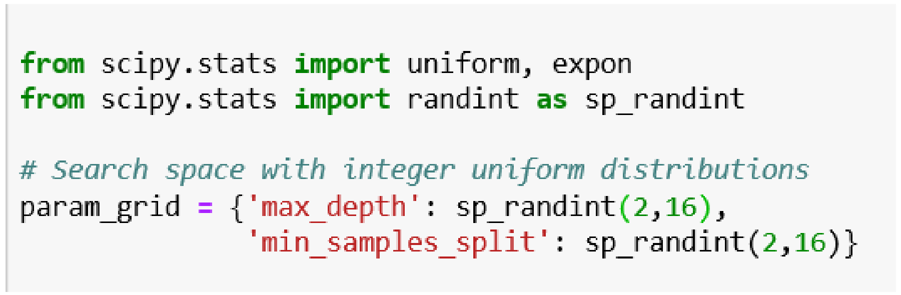
Defining the search space for random search

- For random search, we can also specify the list of values to be checked



```
param_grid = {'max_depth': [2, 4, 6, 8, 10, 12, 14, 16],
              'min_samples_split': [2, 4, 6, 8, 10, 12, 14, 16]}
```

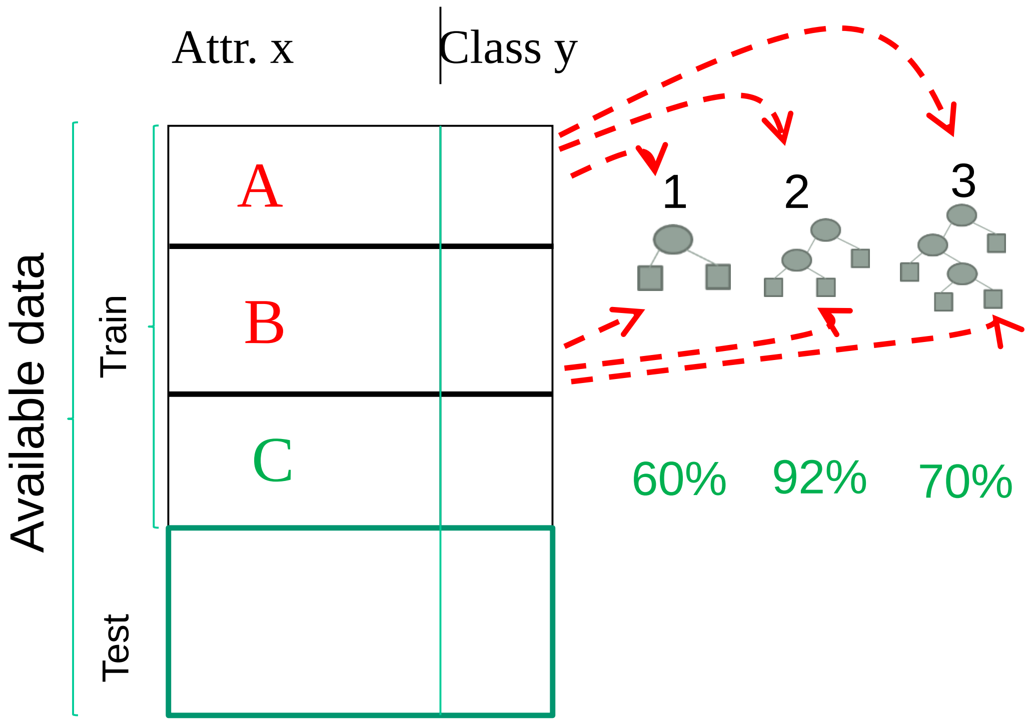
- But also, the statistical distribution out of which values can be sampled (this is preferred):



```
from scipy.stats import uniform, expon
from scipy.stats import randint as sp_randint

# Search space with integer uniform distributions
param_grid = {'max_depth': sp_randint(2,16),
              'min_samples_split': sp_randint(2,16)}
```

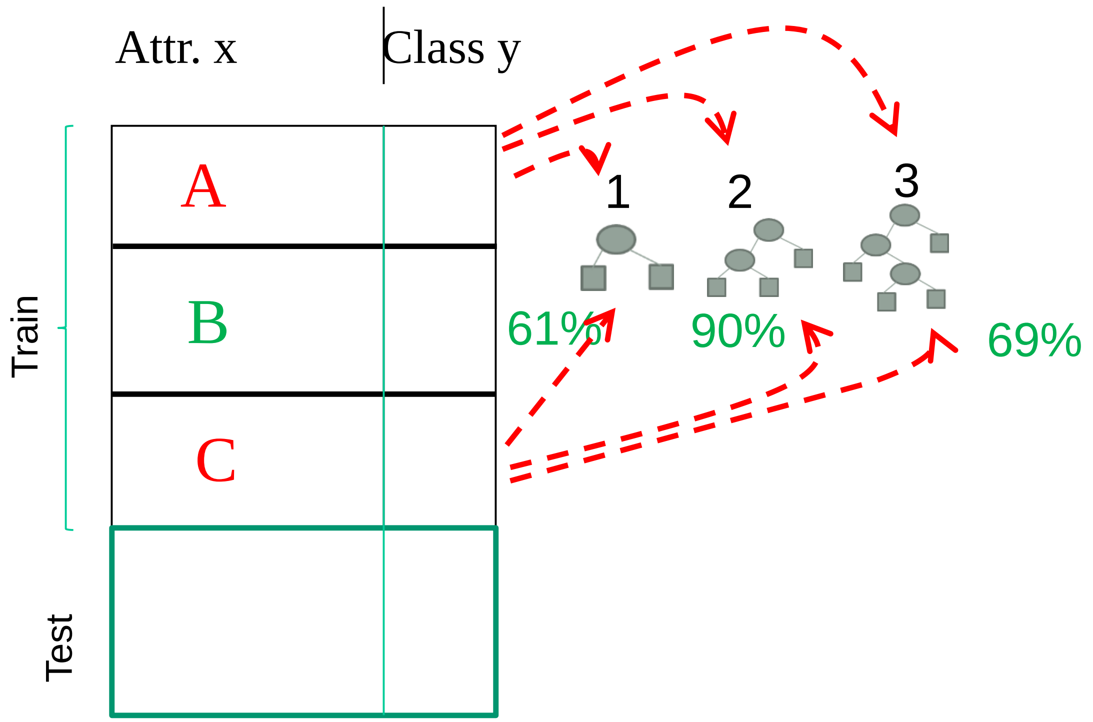
- *sp_randint* is a discrete uniform distribution. *uniform* and *expon* (gaussian) could be used for continous hyper-parameters

HYPER-PARAMETER tuning with crossvalidation



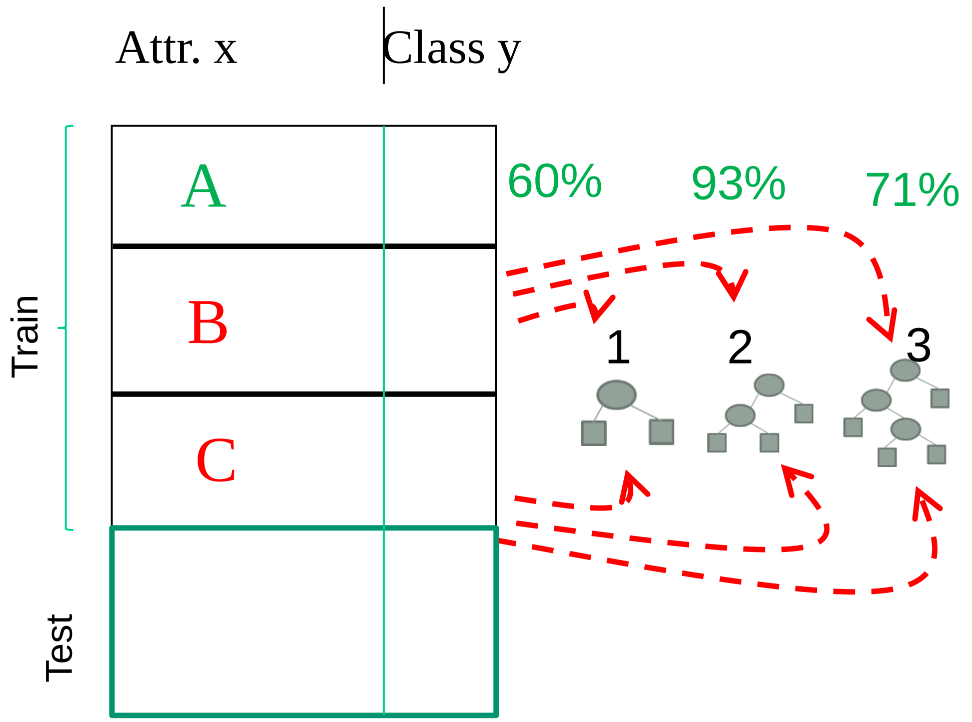
- Now, we are going to use 3-fold crossvalidation for hyper-parameter tuning, but train/test (holdout) for model evaluation (a.ka. estimation of future performance)
- First, we train with A and B, and validate with C

HYPER-PARAMETER tuning with crossvalidation



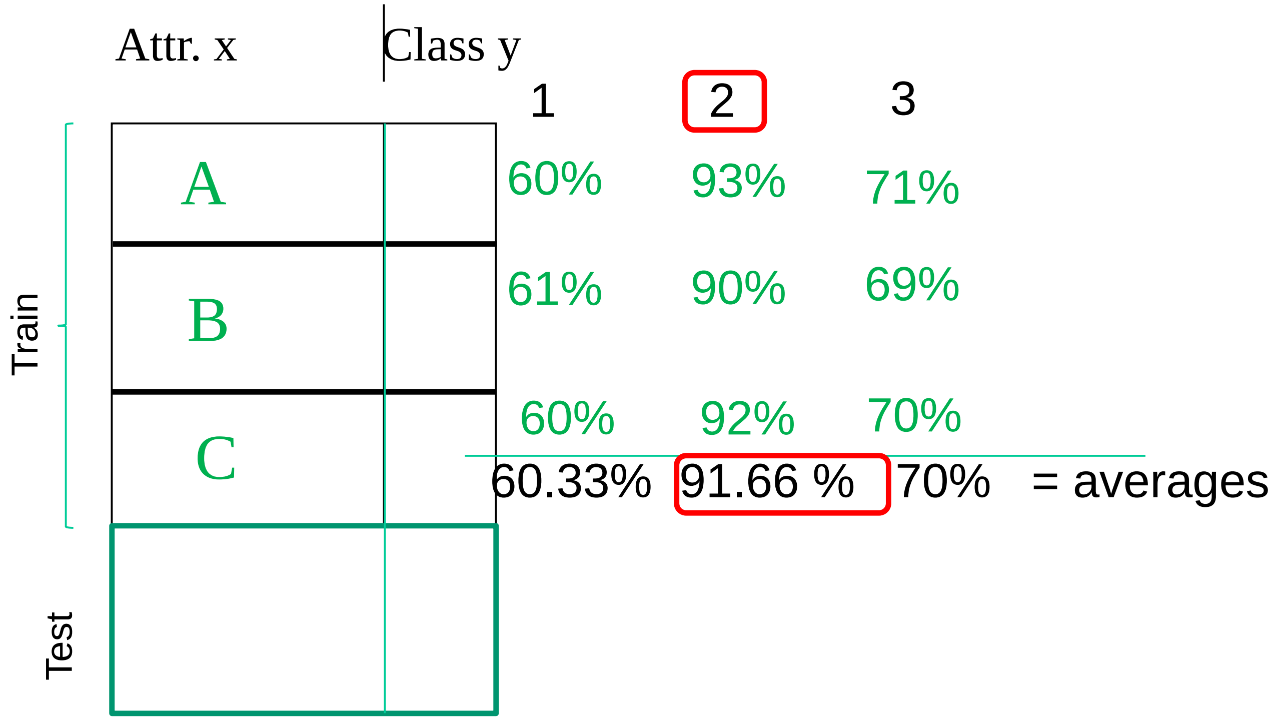
- Then, we train with A and C, and validate with B

HYPER-PARAMETER tuning with crossvalidation



- Finally, we train with B and C, and validate with A

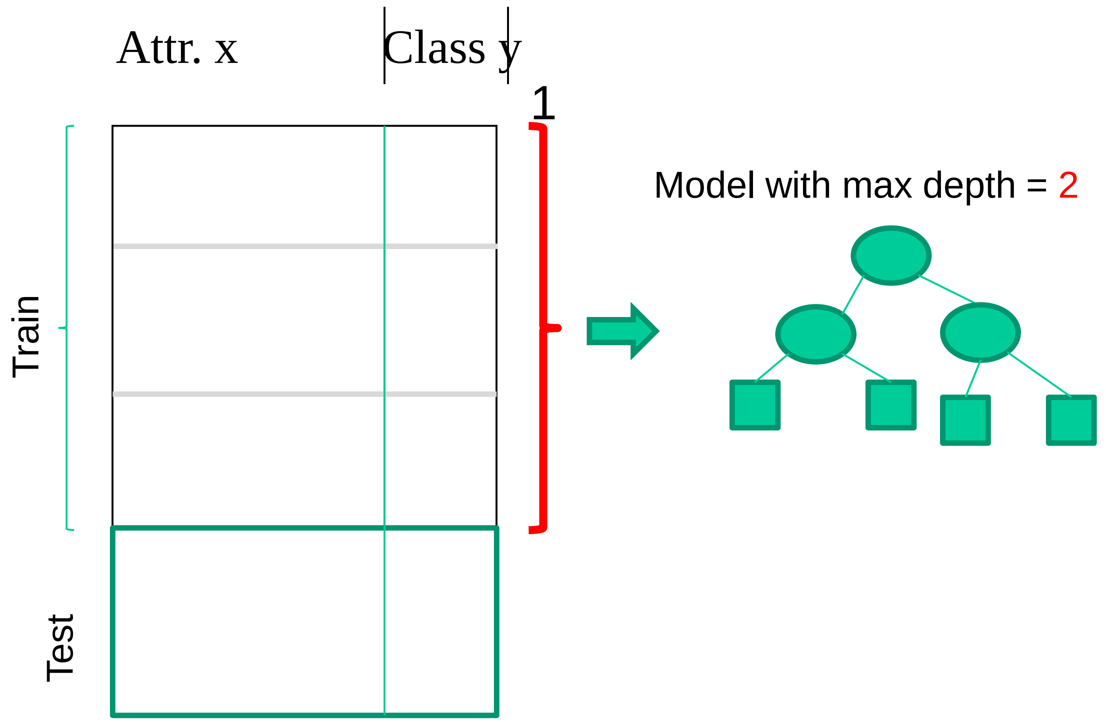
HYPER-PARAMETER tuning with crossvalidation



- Finally, each hyper-parameter value is evaluated by computing the average of the three folds.
- Max depth = 2 is the best.

HYPER-PARAMETER tuning with crossvalidation

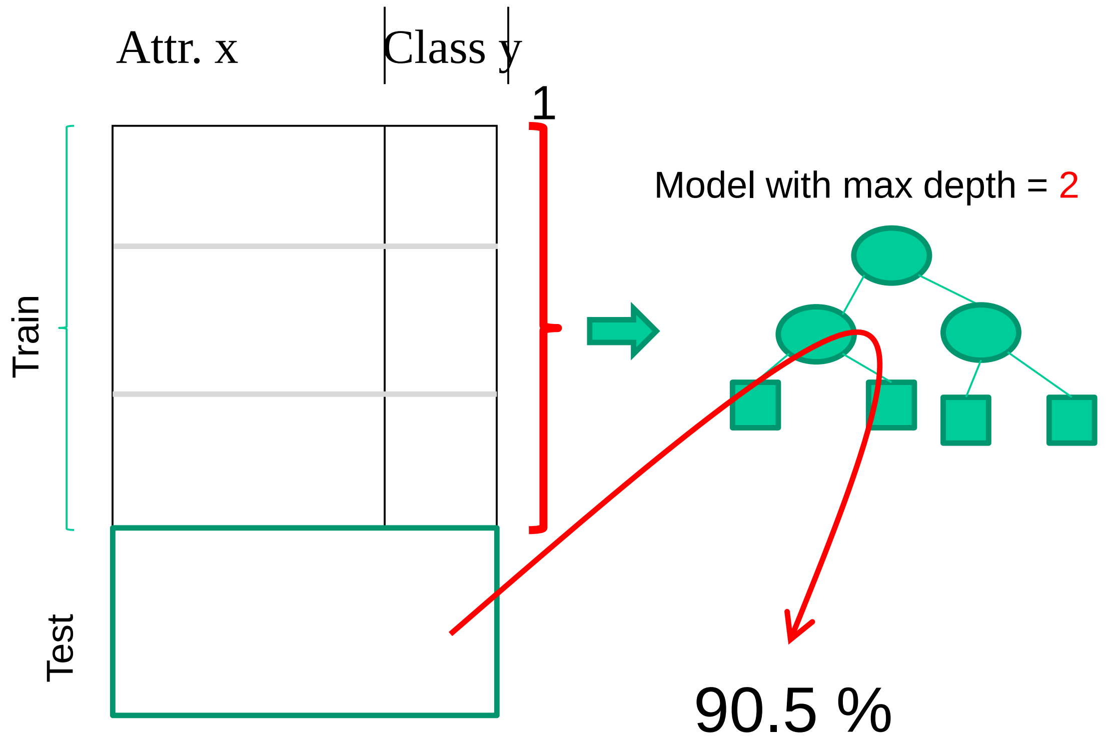
1	2	3
60%	93%	71%
61%	90%	69%
60%	92%	70%
60.33%	91.66 %	70%



- A model is trained with the whole train partition, with the best max depth.

HYPER-PARAMETER tuning with crossvalidation

1	2	3
60%	93%	71%
61%	90%	69%
60%	92%	70%
60.33%	91.66 %	70%



- And then it is evaluated with the test partition

Training with hyper-parameter tuning, then testing

- Training: grid-search with 5-fold crossvalidation
- Evaluation: testing partition

```
from sklearn.datasets import load_iris
from sklearn import tree
from sklearn.model_selection import train_test_split,
GridSearchCV
from sklearn import metrics

iris = load_iris()
X = iris.data
y = iris.target

# Defining the train/test partitions
# random_state is for reproducibility
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.25, random_state=0)
# Defining the method
clf = tree.DecisionTreeClassifier()
# Defining the Search space
param_grid = {'max_depth': range(2,16,2),
              'min_samples_split': range(2,34,2)}
```

```
# Defining a 5-fold crossvalidation grid-search
clf_grid = GridSearchCV(clf,
            param_grid,
            scoring='accuracy',
            cv=5 , n_jobs=1, verbose=1)

# Training the model with the grid-search
np.random.seed(0) # This is for reproducibility
clf_grid.fit(X_train, y_train)

# Making predictions on the testing partition
y_test_pred = clf_grid.predict(X_test)

# And finally computing the test accuracy
print(metrics.accuracy_score(y_test_pred, y_test))
```

```
Fitting 5 folds for each of 112 candidates, totalling 560 fits
0.9210526315789473
[Parallel(n_jobs=1)]: Done 560 out of 560 | elapsed:    0.3s finished
```

HYPER-PARAMETER tuning with train / validation

- Shuffled (i.e. randomly assigned to train and validation)

```
from sklearn.datasets import load_iris
from sklearn import tree
from sklearn.model_selection import train_test_split,
GridSearchCV
from sklearn import metrics

iris = load_iris()
X = iris.data
y = iris.target

# Defining the train/test partitions
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.25, random_state=33)
# Defining the method
clf = tree.DecisionTreeClassifier()
# Defining the Search space
param_grid = {'max_depth': range(2,16,2),
              'min_samples_split': range(2,34,2)}
```

```
from sklearn.model_selection import PredefinedSplit
import numpy as np
# Defining a fixed train/validation grid-search
# -1 means training, 0 means validation
validation_indices = np.zeros(X_train.shape[0])
validation_indices[:round(2/3*X_train.shape[0])] = -1
np.random.seed(0) # This is for reproducibility
validation_indices = np.random.permutation(validation_indices)
tr_val_partition = PredefinedSplit(validation_indices)

clf_grid = GridSearchCV(clf,
               param_grid,
               scoring='accuracy',
               cv=tr_val_partition,
               n_jobs=1, verbose=1)

# Training the model with the grid-search
np.random.seed(0) # This is for reproducibility
clf_grid.fit(X_train, y_train)

# Making predictions on the testing partition
y_test_pred = clf_grid.predict(X_test)

# And finally computing the test accuracy
print(metrics.accuracy_score(y_test_pred, y_test))
```

HYPER-PARAMETER tuning with train / validation

- Not shuffled

```
from sklearn.datasets import load_iris
from sklearn import tree
from sklearn.model_selection import train_test_split,
GridSearchCV
from sklearn import metrics

iris = load_iris()
X = iris.data
y = iris.target

# Defining the train/test partitions
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.25, random_state=33)
# Defining the method
clf = tree.DecisionTreeClassifier()
# Defining the Search space
param_grid = {'max_depth': range(2,16,2),
              'min_samples_split': range(2,34,2)}
```

```
from sklearn.model_selection import PredefinedSplit
import numpy as np
# Defining a fixed train/validation grid-search
# -1 means training, 0 means validation
validation_indices = np.zeros(X_train.shape[0])
validation_indices[:round(2/3*X_train.shape[0])] = -1
tr_val_partition = PredefinedSplit(validation_indices)

clf_grid = GridSearchCV(clf,
            param_grid,
            scoring='accuracy',
            cv=tr_val_partition,
            n_jobs=1, verbose=1)

# Training the model with the grid-search
np.random.seed(0) # This is for reproducibility
clf_grid.fit(X_train, y_train)

# Making predictions on the testing partition
y_test_pred = clf_grid.predict(X_test)

# And finally computing the test accuracy
print(metrics.accuracy_score(y_test_pred, y_test))
```

Exercise

- Would you be able to do this?
  - Training: grid-search with 3-fold crossvalidation
  - Evaluation: 5-fold crossvalidation

Standarization / Normalization

- Some machine learning methods require attributes to be in a similar range (e.g. KNN)
- In scikit-learn, this can be achieved using the standardScaler (standarization) or the minMaxScaler (normalization to 0-1)

Standarization / Normalization

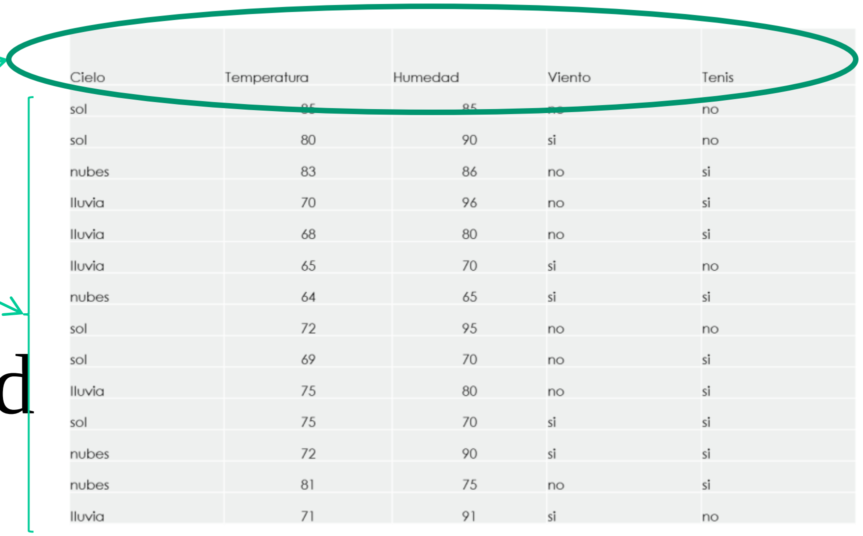
- It is important that all pre-processing (such as normalization) is done with information obtained from the training partition (.fit), and then applied to the testing partition (.transform).

```
import sklearn.preprocessing
X_train_minmax = min_max_scaler.fit_transform(X_train)
X_test_minmax = min_max_scaler.transform(X_test)
```

A Tutorial on Scikit Learn Pre-processing / Pipelines

Pipelines in Scikit Learn

- **Preprocessing:**
  - **Instances**
  - **Attributes**
- Pipelines are useful to combine pre-processing and training the model



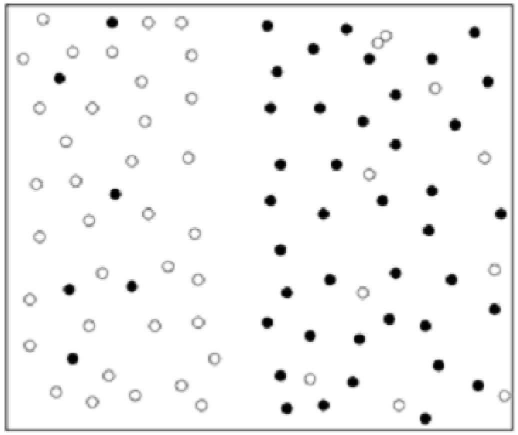
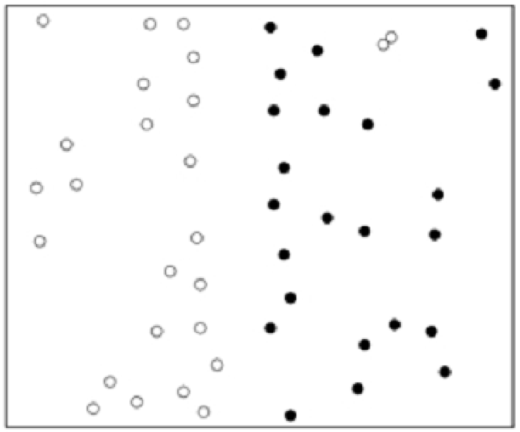
Cielo	Temperatura	Humedad	Viento	Tenis
sol	85	85	no	no
sol	80	90	si	no
nubes	83	86	no	si
lluvia	70	96	no	si
lluvia	68	80	no	si
lluvia	65	70	si	no
nubes	64	65	si	si
sol	72	95	no	no
sol	69	70	no	si
lluvia	75	80	no	si
sol	75	70	si	si
nubes	72	90	si	si
nubes	81	75	no	si
lluvia	71	91	si	no

Pre-processing

- Instances:
  - Removing outliers
  - Removing noisy instances (**Wilson editing rule**), mainly for KNN
  - Sampling in order to balance classes in imbalanced problems (such as **SMOTE** – Synthetic Minority Over-sampling Technique, …) or **ADASYN**
- Attributes:
  - Standarization / normalization (scaling to a range)
  - Imputation (what to do with missing values?)
  - Categorical attribute encoding into numbers
  - **Attribute selection**
  - Attribute transformation (PCA, ...)

Wilson editing rule

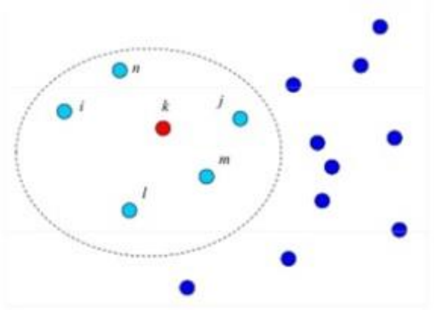
- Wilson editing rule: remove instance $\mathbf{x}_i$ if it is classified incorrectly by the majority class of its k neighbours:
  - It removes noisy instances inside a class region
  - It smooths boundaries
- It works well for KNN, but can be used for other methods too
- Example of repeated Wilson editing

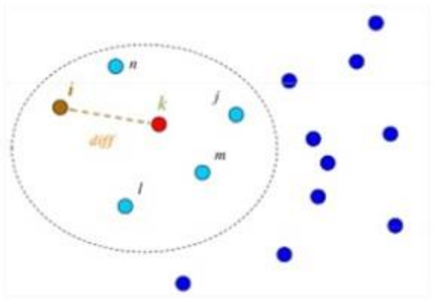
https://imbalanced-learn.readthedocs.io/

imblearn.under_sampling.EditedNearestNeighbours

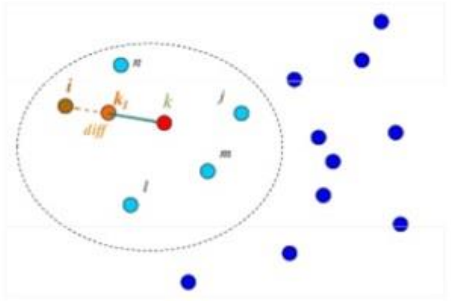
SMOTE (balance minority classes)



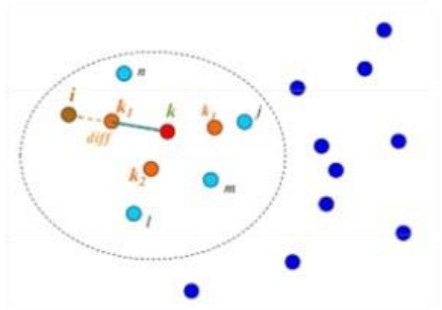
1. For each minority example k compute nearest minority class examples (i, j, l, n, m)



2. Randomly choose an example out of 5 closest points



3. Synthetically generate event k_1, such that k_1 lies between k and i



4. Dataset after applying SMOTE 3 times

https://imbalanced-learn.readthedocs.io/

Standarization and normalization to a range

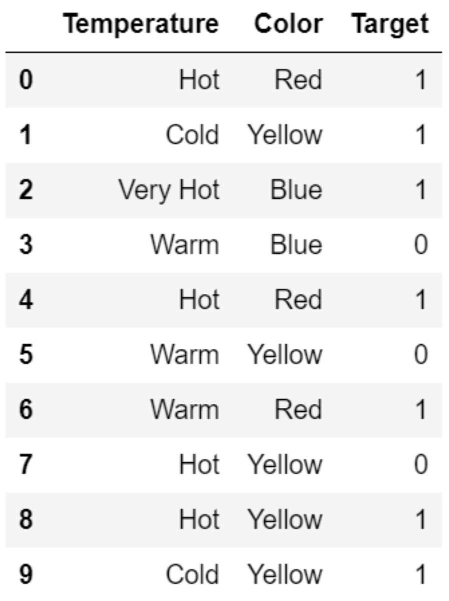
- Different attributes may have different ranges (e.g. height: 0m-2m, weight: 0kg-100kg, …)
- The aim is that all attributes have the same range or spread
- Important for some methods such as KNN, Support Vector Machines, and neural networks. Not important for Decision trees.
- If $\mathbf{x}_i$ is an attribute / feature (i.e. a column in a data matrix)
- Normalization: $\mathbf{x}_i = (\mathbf{x}_i\text{-min}(\mathbf{x}_i)) / (\max(\mathbf{x}_i) - \min(\mathbf{x}_i))$
  - New range = 0-1
- Standarization: $\mathbf{x}_i = (\mathbf{x}_i\text{-mean}(\mathbf{x}_i)) / \text{std}(\mathbf{x}_i)$

Imputation

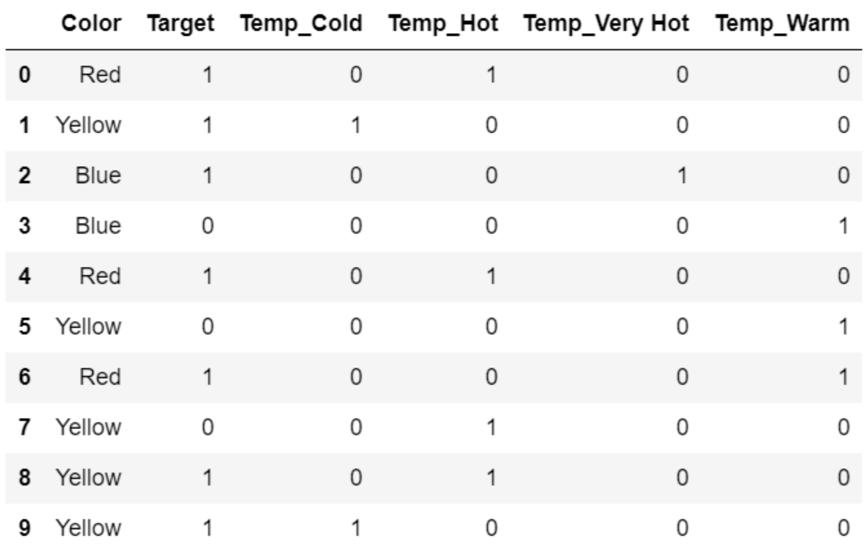
- Imputation = replacing missing values (np.nan)
- Some methods are able to deal with missing values (e.g. trees), but some methods aren't (e.g. KNN, SVM, ...)
- Strategies:
  - Remove instances with np.nan 's
  - Remove attributes with np.nan 's
  - Univariate: replace np.nan 's with mean, median, or mode (categorical attributes):
    - *sklearn.impute.SimpleImputer*
  - Multivariate: use a machine learning method to compute models of an attribute in terms of the other attributes. Use the model to impute each attribute, in turn.
    - *sklearn.impute.IterativeImputer*

Encoding categorical variables: one-hot-encoding (dummy variables)

- Some machine learning methods are not able to deal with categorical/discrete attributes
- Most commonly used: dummy variables or one-hot-encoding (typically, only N-1 columns are kept)



	Temperature	Color	Target
0	Hot	Red	1
1	Cold	Yellow	1
2	Very Hot	Blue	1
3	Warm	Blue	0
4	Hot	Red	1
5	Warm	Yellow	0
6	Warm	Red	1
7	Hot	Yellow	0
8	Hot	Yellow	1
9	Cold	Yellow	1

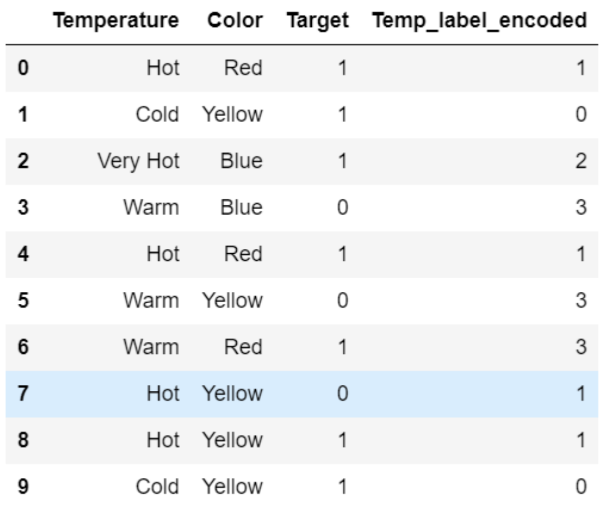


	Color	Target	Temp_Cold	Temp_Hot	Temp_Very Hot	Temp_Warm
0	Red	1	0	1	0	0
1	Yellow	1	1	0	0	0
2	Blue	1	0	0	1	0
3	Blue	0	0	0	0	1
4	Red	1	0	1	0	0
5	Yellow	0	0	0	0	1
6	Red	1	0	0	0	1
7	Yellow	0	0	1	0	0
8	Yellow	1	0	1	0	0
9	Yellow	1	1	0	0	0

Encoding categorical variables: frequency and integer

- However, one-hot-encoding generates too many columns for variables with many values.
- Alternatives: integer/label encoding
- Problem: an artificial (false) order is introduced

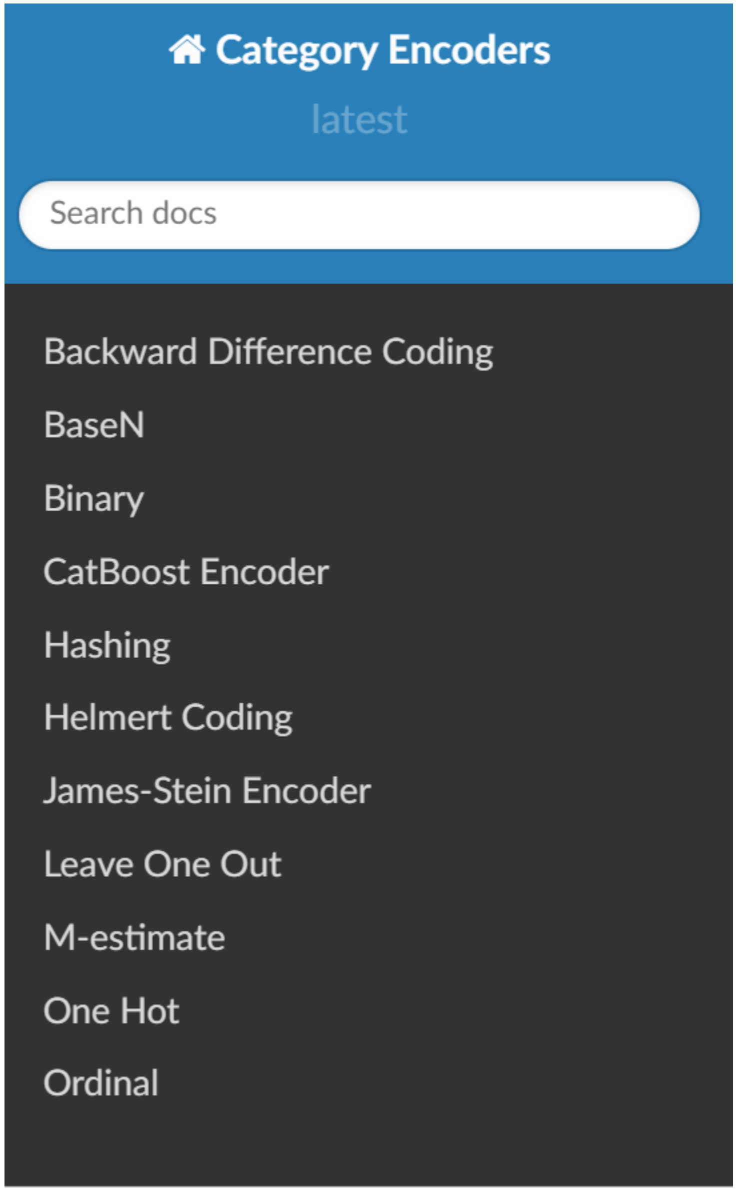
Label/integer encoding



	Temperature	Color	Target	Temp_label_encoded
0	Hot	Red	1	1
1	Cold	Yellow	1	0
2	Very Hot	Blue	1	2
3	Warm	Blue	0	3
4	Hot	Red	1	1
5	Warm	Yellow	0	3
6	Warm	Red	1	3
7	Hot	Yellow	0	1
8	Hot	Yellow	1	1
9	Cold	Yellow	1	0

Encoding categorical variables

- Target mean encoding (as in the assignment)
- https://contrib.scikit-learn.org/categorical-encoding/

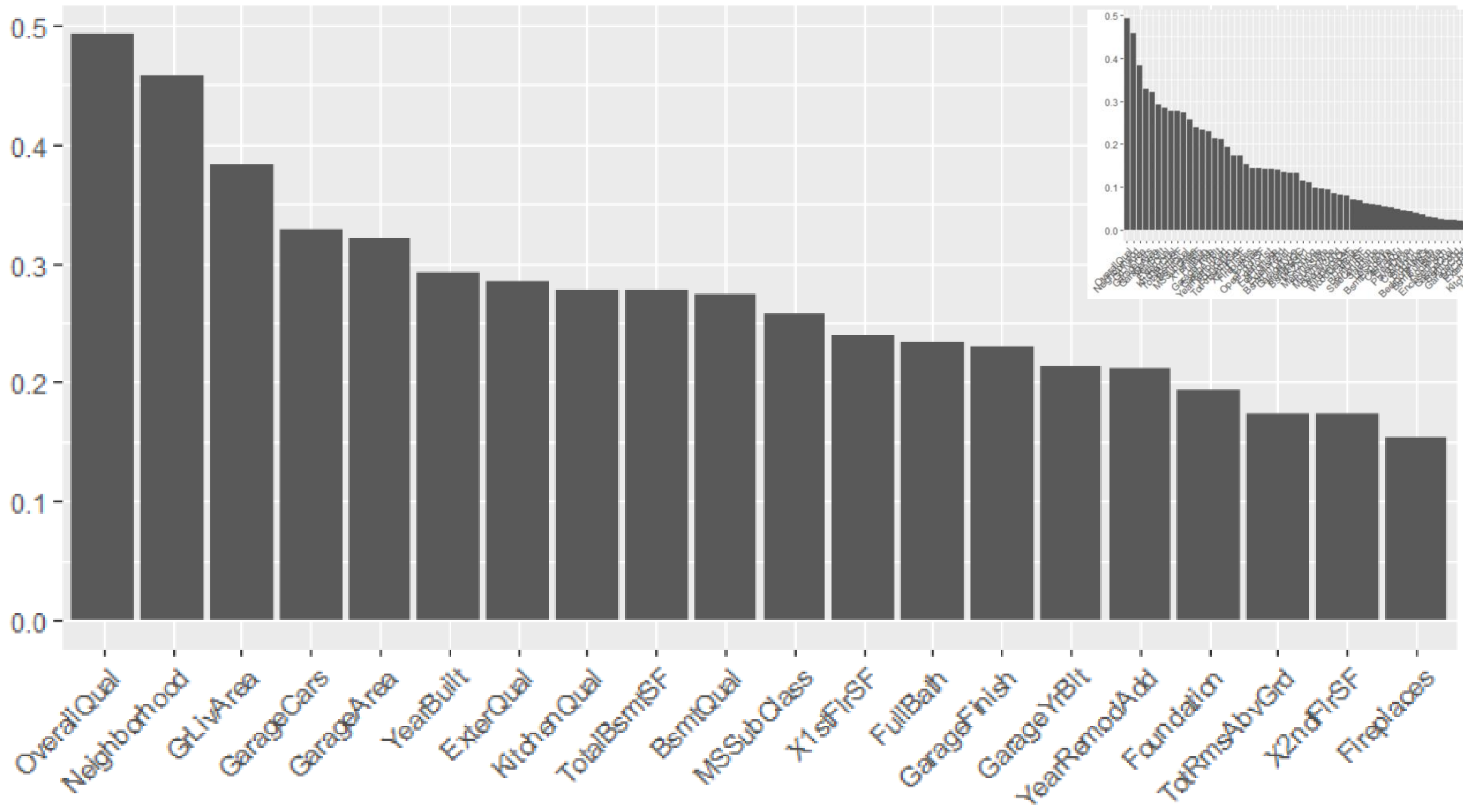


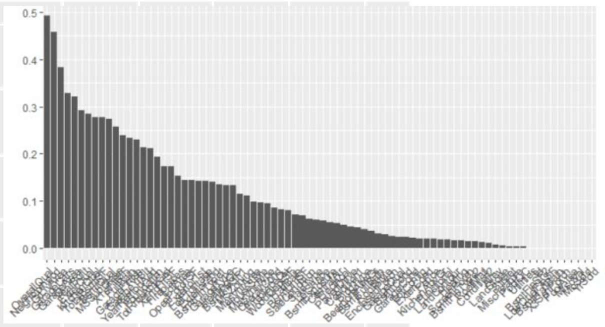
Attribute / Feature selection

- Given input attributes A_1, A_2, ..., A_n, each A_i is evaluated **individually**, computing its correlation or dependency with the class, independently of the rest of attributes (i.e. attributes are considered individually, rather than subsets)
- An attribute A_1 is correlated with the class, if knowing its value implies that the class can be predicted more accurately
  - For instance, car speed is correlated with having an accident. But the Social Security Number of the driver is not.
  - For instance, salary may be (inversely) correlated with credit default
- How to evaluate / rank attributes (attribute/class correlation):
  - Entropy (information gain), like in decision trees
  - Chi-square
  - Mutual information
  - …
- Once evaluated and ranked, the worst attributes can be removed (according to a threshold)

Example of filter ranking

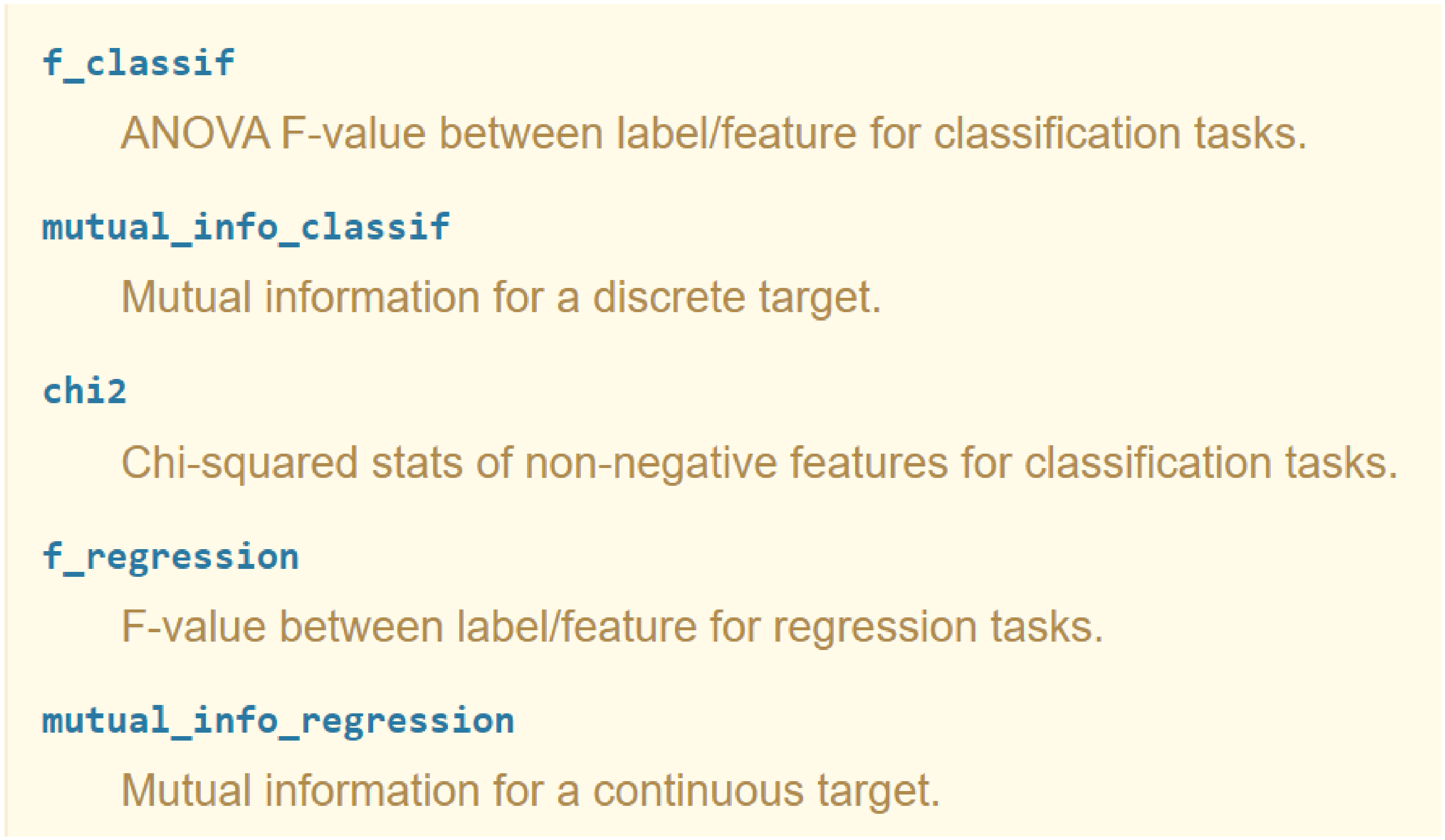
(Housing prices)





Attribute / Feature selection

- sklearn.feature_selection.SelectKBest



`f_classif`
: ANOVA F-value between label/feature for classification tasks.

`mutual_info_classif`
: Mutual information for a discrete target.

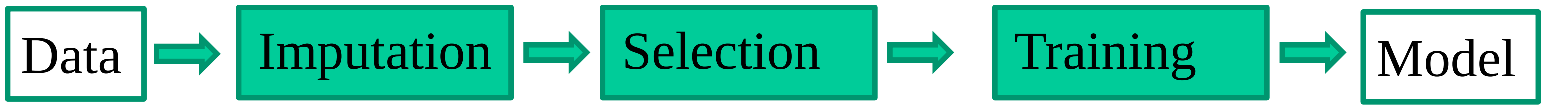
`chi2`
: Chi-squared stats of non-negative features for classification tasks.

`f_regression`
: F-value between label/feature for regression tasks.

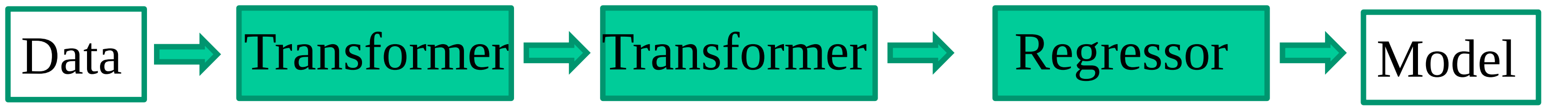
`mutual_info_regression`
: Mutual information for a continuous target.

Pipelines in Scikit Learn

- Sometimes training a model involves applying a sequence of methods, in most cases involving some preprocessing steps.
- For example, we might want to do:
  1. Imputation (to remove missing values)
  2. Attribute selection (to select the most relevant features)
  3. Model training



- Pipelines in sklearn are sequences of estimators: an **estimator** in sklearn is either a **transformer** (or pre-processing method) or a **classifier/regressor** (or training method)

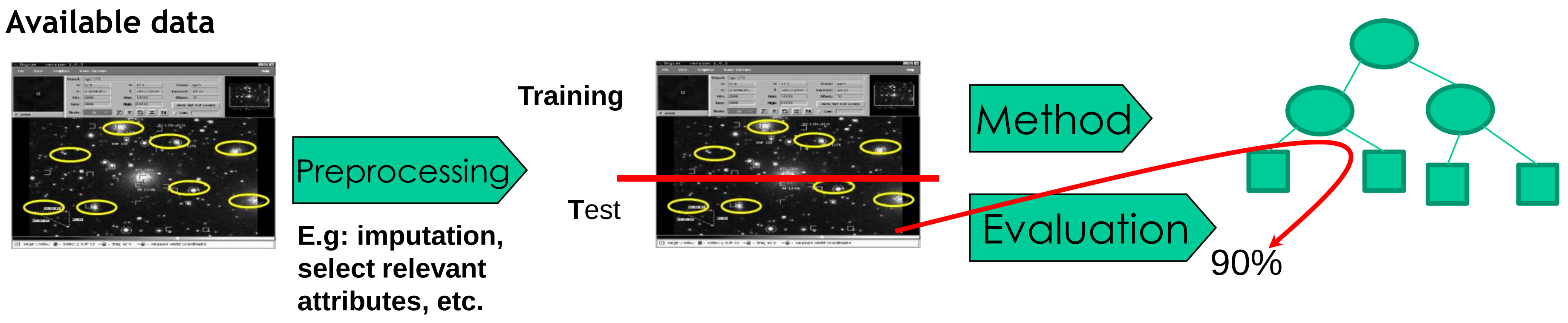


Why use pipelines?

1. Clear coding: a pipeline clearly states your preprocessing and training methods
2. Hyper-parameter tuning: each step in the pipeline has its own hyper-parameters. Pipelines make possible to tune all of them
3. Avoiding data leakage: test data should never be used for training, in any way

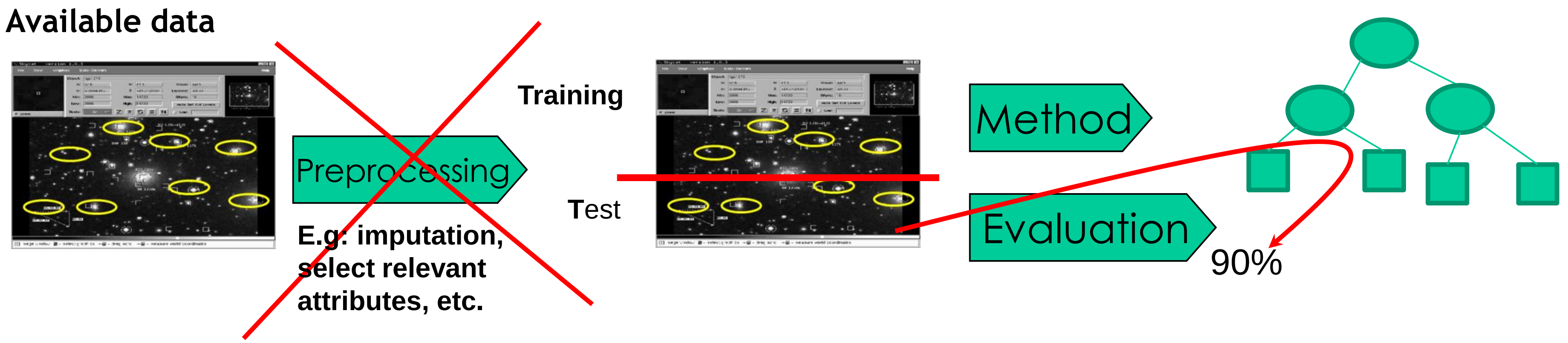
How to do preprocessing correctly?

- Two types of pre-processing:
  - Not data-dependent:
    - E.g. remove ID attribute because we know it is not useful for classification
    - We will do this no matter what the data matrix contains
  - Data-dependent:
    - E.g. remove attribute $\mathbf{x}_4$ because its values are not correlated with the class
- You may think the following workflow is correct, but the problem is that there might be some "data leakage" from the test partition to the training partition (i.e. the model will "know" a bit about the test partition)



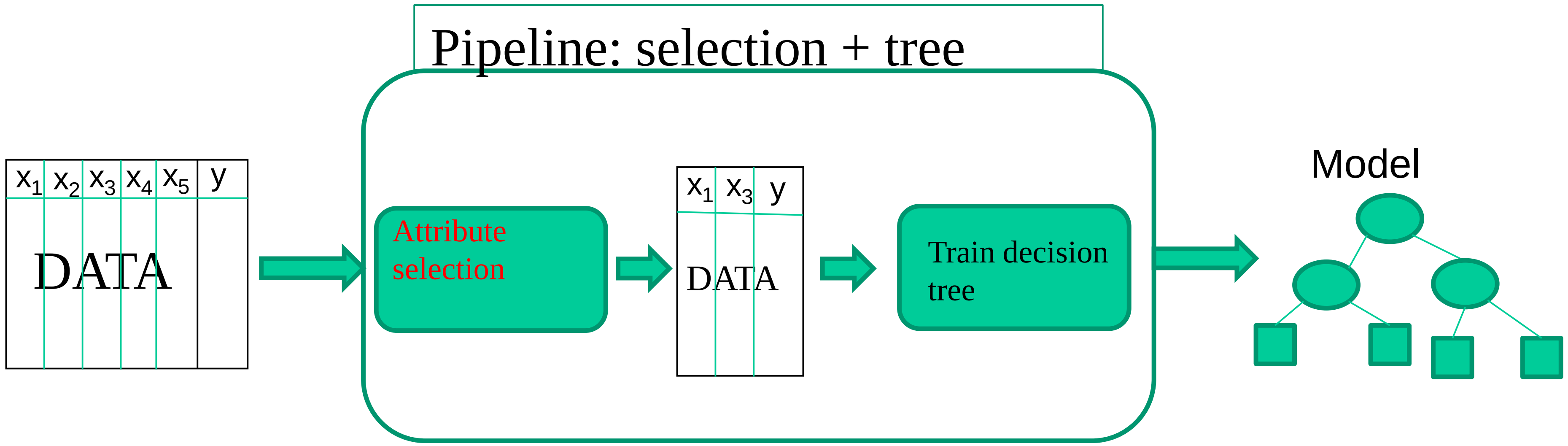
How to do preprocessing correctly?

- We shouldn't use test data for training the model, in any way

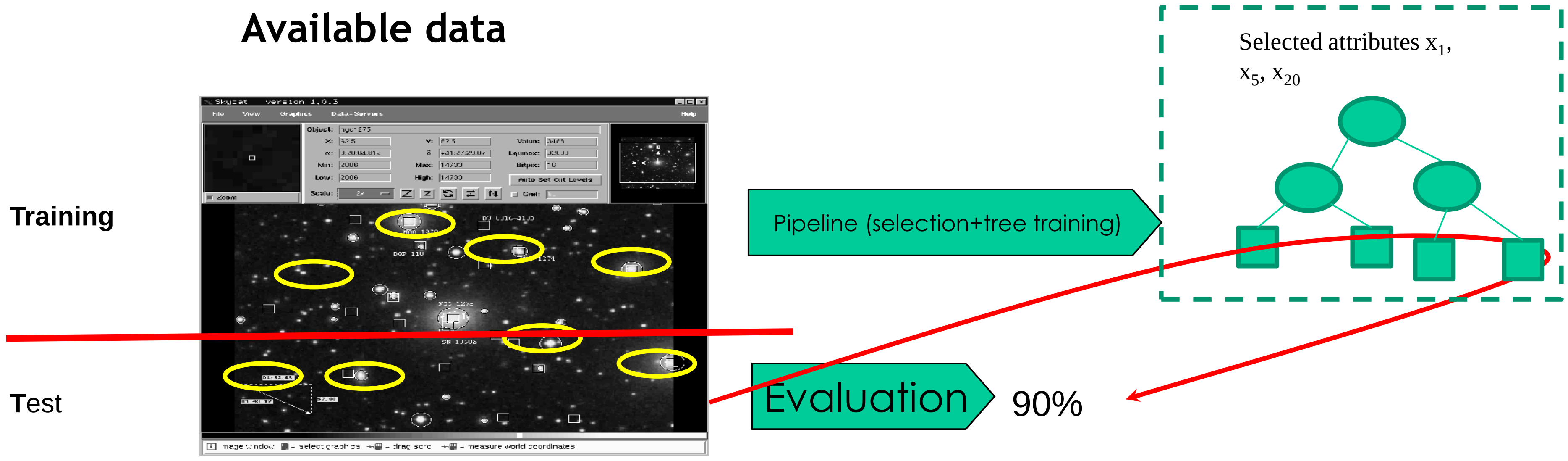


How to do preprocessing correctly?

- It is better to create a pipeline
- E.g. for attribute selection:

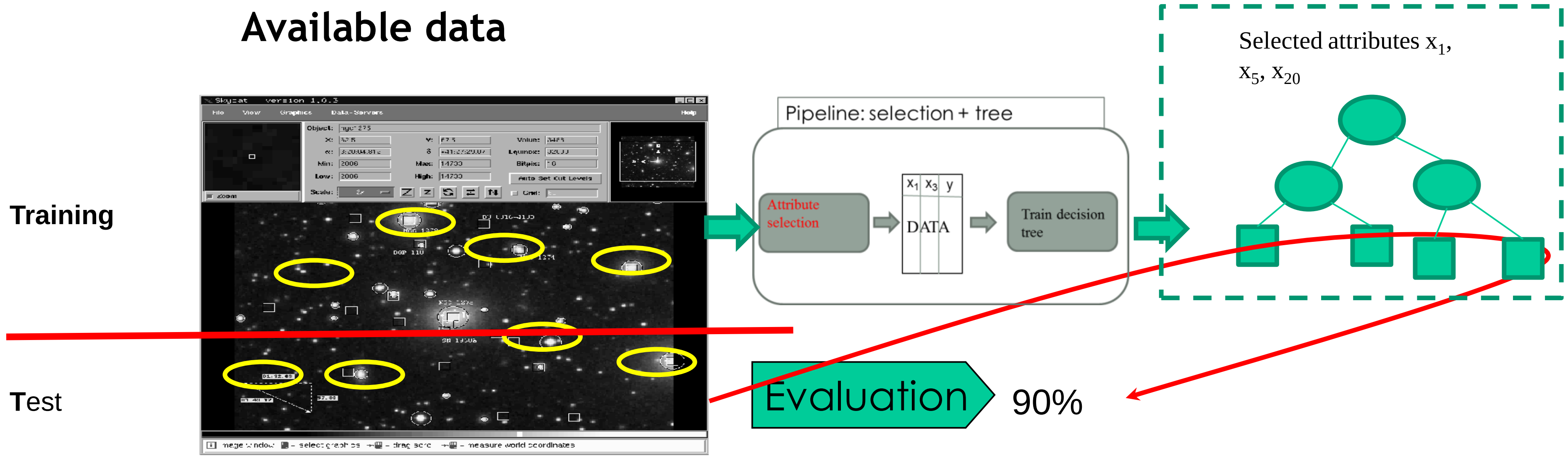


How to do preprocessing correctly?


Available data
Training
Test
Pipeline (selection+tree training)
Selected attributes x_1, x_5, x_{20}
Evaluation
90%

- Which attributes are selected is decided with the training partition only, and kept for use during testing
- The same thing is done for other preprocessing tasks:
  - For attribute normalization, max(xi), min(xi) are computed using training data only, and kept for use during testing
  - For imputation, mean(xi) is computed with training data, and used during testing.

How to do preprocessing correctly?

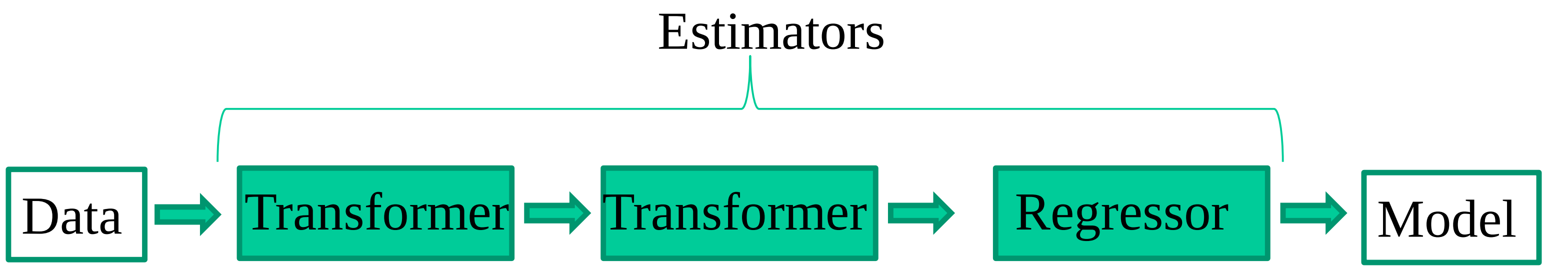


- Conclusion: pipelines can be used in two different contexts:
  - Training
  - Testing

https://scikit-learn.org/stable/data_transforms.html

Pipelines in Scikit Learn

- Pipeline: a sequence of estimators (transformers and classifier/regression)
- Transformer: feature selection, imputation, normalization, binarizer, ... They have two methods:
    - **.fit** (for training data)
    - **.transform** (typically, for testing data)
- Classifier / regressor: decision trees, knn, … Two methods:
    - **.fit** (for training data)
    - **.predict** (typically, for testing data)



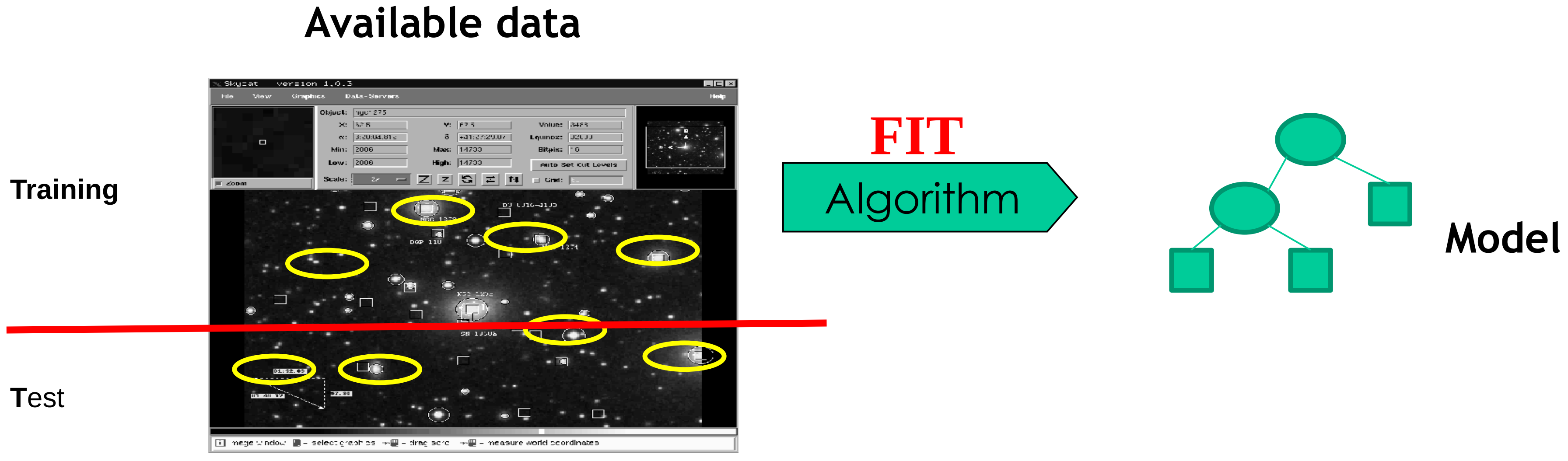
Let's see classifiers/regressors and transformers individually, and later, we will put them together into a pipeline.

But first, let's get some training and testing data:

```
# Getting the data
import numpy as np
from sklearn import datasets
from sklearn.model_selection import train_test_split

boston = datasets.load_boston()
X = boston.data
y = boston.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=33)
```

Classifier / regressor: fit

- The *fit* method trains a model

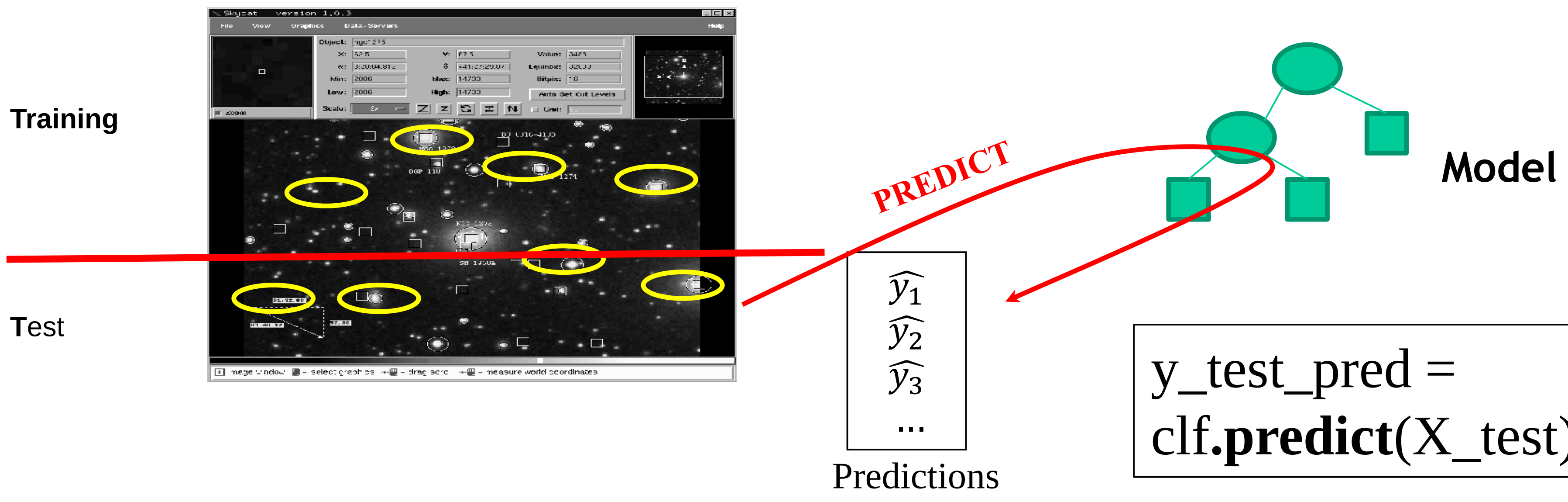


```
from sklearn.neighbors import KNeighborsRegressor
clf = KNeighborsRegressor()
clf.fit(X_train,y_train)
```

Classifier / regressor: predict

- The *predict* method obtains predictions from a model

Available data



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

Now, let's go to the transformers ...

- But let's put one nan for illustration purposes

```
X_train[1, 1] = np.nan
X_test[1, 1] = np.nan
X_train
X_test
```

```
In [63]: X_train
Out[63]:
array([
[2.9 e-01, 0.0 e+00, 6.2 e+00, ...],
[5.0 e-02, nan,      6.0 e+00, ...,],
[1.3 e+01, 0.0 e+00, 1.8 e+01, ...],
...,
[4.5 e-02, 0.0 e+00, 1.3 e+01, ...],
[5.2 e+00, 0.0 e+00, 1.8 e+01, ...],
[1.2 e+00, 0.0 e+00, 8.1 e+00, ...])

In [64]: X_test
Out[64]:
array([
[9.2 e-02, 0.0 e+00, 2.5 e+01, ...],
[2.5 e+01, nan,      1.8 e+01, ...],
[7.0 e+00, 0.0 e+00, 1.8 e+01, ...],
...,
[1.5 e+01, 0.0 e+00, 1.8 e+01, ...],
[2.0 e-01, 2.2 e+01, 5.8 e+00, ...],
[3.4 e-01, 0.0 e+00, 7.3 e+00, ...]])
```

Transformer: fit

```
In [63]: X_train
Out[63]:
array([
[2.9 e-01, 0.0 e+00, 6.2 e+00, ...],
[5.0 e-02, nan,       6.0 e+00, ...],
[1.3 e+01, 0.0 e+00, 1.8 e+01, ...],
...,
[4.5 e-02, 0.0 e+00, 1.3 e+01, ...],
[5.2 e+00, 0.0 e+00, 1.8 e+01, ...],
[1.2 e+00, 0.0 e+00, 8.1 e+00, ...]])
```

```
from sklearn.impute import SimpleImputer
trf = SimpleImputer(strategy='mean')
trf = trf.fit(X_train)
```

- trf.statistics_ contains the imputation fill value (the mean) for each feature (column):

```
trf.statistics_
Out[78]:
array([3.2 e+00, 1.1 e+01, 1.0 e+01, ...])
```

```
trf.statistics_
Out[78]:
array([3.2 e+00, 1.1 e+01, 1.0 e+01, 6.7 e-02, 5.4 e-01, 6.3 e+00, 6.8 e+01, 3.8 e+00, 8.7 e+00, 3.8 e+02, 1.8 e+01, 3.6 e+02, 1.2 e+01])
```

Transformer: transform

```
X_train = trf.transform(X_train)
X_test = trf.transform(X_test)
```

```
In [63]: X_train
Out[63]:
array([[2.9 e-01,  0.0 e+00,  6.2 e+00, ...],
       [ 5.0 e-02,  nan,       6.0 e+00, ...],
       [ 1.3 e+01, 0.0 e+00, 1.8 e+01, ...],
...,
      [ 4.5 e-02,  0.0 e+00,  1.3 e+01, ...],
      [ 5.2 e+00, 0.0 e+00,  1.8 e+01, ...],
      [ 1.2 e+00, 0.0 e+00,  8.1 e+00, ...]])
```

X_train = trf.transform(X_train)

```
array([[2.9 e-01, 0.0 e+00, 6.2 e+00, ...],
       [5.0 e-02, 1.1 e+01, 6.0 e+00, ...],
       [1.3 e+01, 0.0 e+00, 1.8 e+01, ...],
    ...,
       [4.5 e-02, 0.0 e+00, 1.3 e+01, ...],
       [5.2 e+00, 0.0 e+00, 1.8 e+01, ...],
       [1.2 e+00, 0.0 e+00, 8.1 e+00, ...]])
```

```
trf.statistics_
Out[78]:
array([3.2 e+00, 1.1 e+01, 1.0 e+01, 6.7 e-02, 5.4 e-01, 6.3 e+00, 6.8 e+01, 3.8 e+00, 8.7 e+00, 3.8 e+02, 1.8 e+01, 3.6 e+02, 1.2 e+01])
```

Transformer: transform

- Notice that the same transformation is applied to train and test

```
In [64]: X_test
Out[64]:
array([
[9.2 e-02,  0.0 e+00,  2.5 e+01, ...],
[2.5 e+01,  nan,       1.8 e+01, ...],
[7.0 e+00,  0.0 e+00,  1.8 e+01, ...],
...,
[1.5 e+01,  0.0 e+00,  1.8 e+01, ...],
[2.0 e-01,  2.2 e+01,  5.8 e+00, ...],
[3.4 e-01,  0.0 e+00,  7.3 e+00, ...]])
```

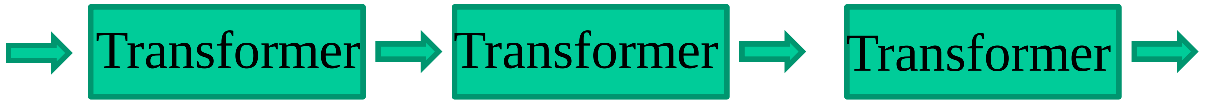
X_test = trf.transform(X_test)

```
array([
      [9.2 e-02, 0.0 e+00, 2.5 +01, ...],
      [2.5 e+01, 1.1 e+01, 1.8 e+01, ...],
      [7.0 e+00, 0.0 e+00, 1.8 e+01, ...],
      ...,
      [1.5 e+01, 0.0 e+00, 1.8 e+01, ...],
      [2.0 e-01, 2.2 e+01, 5.8 e+00, ...],
      [3.4 e-01, 0.0 e+00, 7.3 e+00, ...]])
```

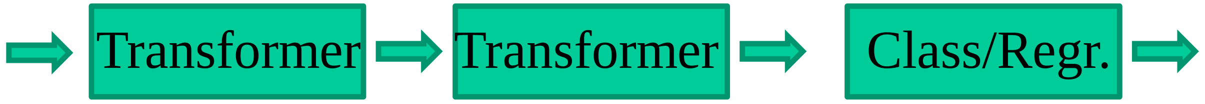
```
# Complete code
import numpy as np
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.impute import SimpleImputer

boston = datasets.load_boston()
X = boston.data
y = boston.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=33)

X_train[1, 1] = np.nan
X_test[1, 1] = np.nan
X_train
X_test

trf = SimpleImputer(strategy='mean')
trf = trf.fit(X_train)

trf.statistics_

X_train = trf.transform(X_train)
X_test = trf.transform(X_test)
```

Pipelines in Scikit Learn

- Let's put transfomers and class/regressors together: pipelines
- A sequence of transformers IS a transformer:
  - transformer + transformer + ... + transformer ≡ transformer
  - that means that it has the **.fit** and **.transform** methods



- A sequence of several transformers plus a classifier/regressor IS a classifier/regressor:
  - transformer + transformer + ... + class/regr ≡ class/regr
  - that means that it has the **.fit** and **.predict** methods



- All estimators in a pipeline except the last one, must be transformers

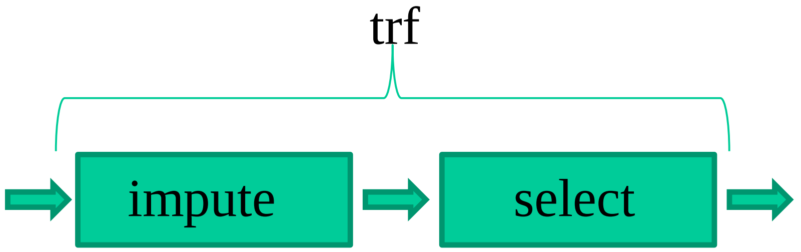
A transformer pipeline: trf=imputation + feature selection

```
import numpy as np
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.neighbors import KNeighborsRegressor
from sklearn.impute import SimpleImputer
from sklearn.feature_selection import SelectKBest, f_regression
```

In sklearn, pipelines are lists of tuples ('stepname', step)

```
imputer =  SimpleImputer(strategy='mean')
selector = SelectKBest(f_regression, k=3)

trf = Pipeline([
        ('impute', imputer),
        ('select', selector)])
```

trf is a sequence of transformers, therefore, trf IS a transformer (with **.fit,** and **.transform** methods).



A transformer pipeline: Accessing the individual steps

- We can **fit** the transformer pipeline and then access each step (tab completes the step names)

```
trf = trf.fit(X_train, y_train)
trf.named_steps['impute']
trf.named_steps['select']
```

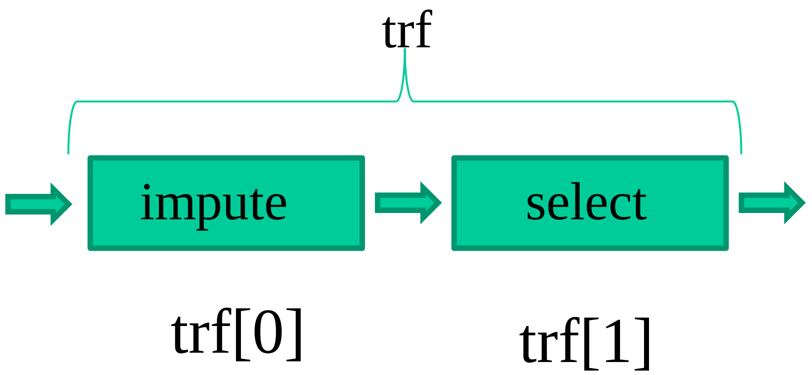
Shorter

```
trf['impute']
trf['select']
```

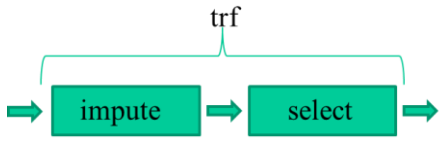
or by integer position

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

```
# The imputation step
In [36]: trf['impute']
Out[36]:
SimpleImputer(add_indicator=False, copy=True,
fill_value=None,
missing_values=nan, strategy='mean', verbose=0)

# The feature selection step
In [37]: trf['select']
Out[37]: SelectKBest(k=3, score_func=<function
f_regression at 0x0000018F017A3E58>)
```



A transformer pipeline:
Getting the properties of each individual steps



```
trf.named_steps['impute'].statistics_
trf.named_steps['select'].get_support()
```

```
# The imputation step
In [126]: trf.named_steps['impute'].statistics_
Out[126]:
array([3.2 e+00, 1.1 e+01, 1.0 e+01, ...])
```

These values will be used for imputation

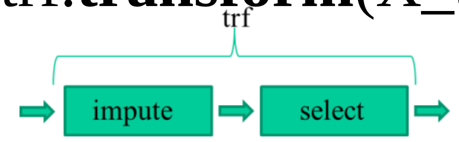
```
# The feature selection step
trf.named_steps['select'].get_support(True)
Out[128]: array([ 5, 10, 12], dtype=int64)
```

These attributes will be selected

A transformer pipeline: applying the transformation

```
[[2.9 e-01  0.0 e+00  6.2 e+00 ...  1.7 e+01  3.7 e+02  3.9 e+00]
 [5.0 e-02       nan  6.0 e+00 ...  1.6 e+01  3.9 e+02  1.2 e+01]
 [1.3 e+01  0.0 e+00  1.8 e+01 ...  2.0 e+01  1.3 e+02  1.3 e+01]
 ...
 [4.5 e-02  0.0 e+00  1.3 e+01 ...  1.6 e+01  3.9 e+02  1.3 e+01]
 [5.2 e+00  0.0 e+00  1.8 e+01 ...  2.0 e+01  3.7 e+02  1.8 e+01]
 [1.2 e+00  0.0 e+00  8.1 e+00 ...  2.1 e+01  3.7 e+02  2.1 e+01]]
```

X_train = trf.**transform**(X_train)



```
[[ 7.68  17.4   3.92 ]
 [  5.70  16.9  12.43 ]
 [  3.86  20.2  13.33 ]
 ...
 [ 5.88  16.4  13.51 ]
 [ 6.05  20.2  18.76 ]
 [ 5.57  21.    21.02 ]]
```

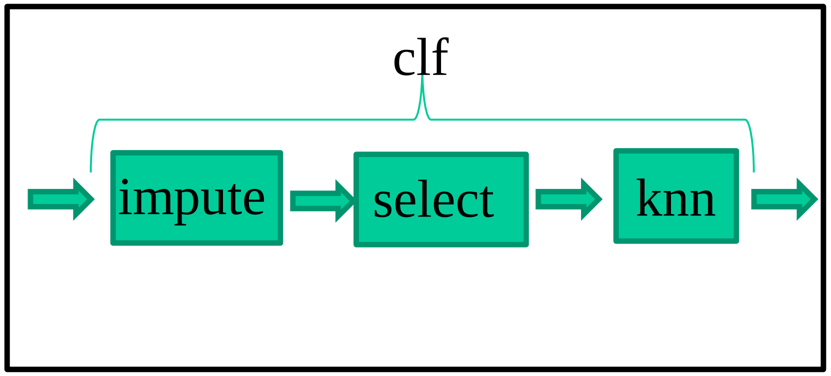
Attributes 5, 10, 12 have been selected, and the np.nan have been imputed

also X_test = trf.**transform**(X_test)

A classifier/regressor pipeline: transf + transf + ... + class/regr

- clf.fit(X_train, y_train):
  - impute.fit(X_train):
    - averages are computed using the train partition only
  - select.fit(X_train,y_train):
    - features are selected using the train partition only.
  - knn.fit(X_train,y_train):
    - model is trained on the imputed and feature-selected training data

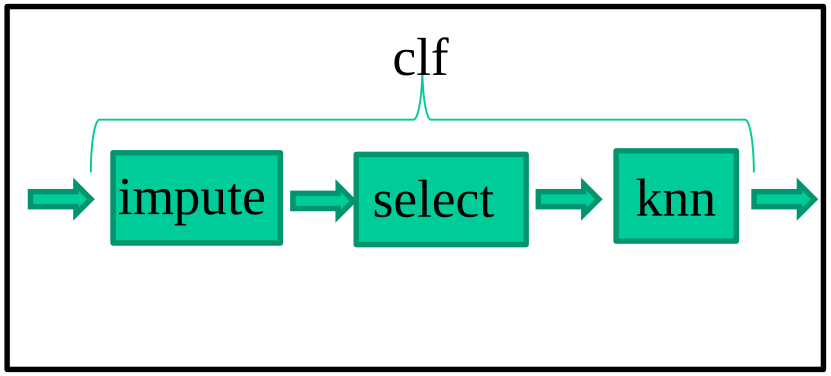
```
imputer =  SimpleImputer(strategy='mean')
selector = SelectKBest(f_regression, k=3)
knn = KNeighborsRegressor()

clf = Pipeline([
        ('impute', imputer),
        ('select', selector),
        ('knn_regression', knn)])

clf = clf.fit(X_train, y_train)
y_test_pred = clf.predict(X_test)
```



A classifier/regressor pipeline: transf + transf + ... + class/regr

- clf.fit(X_train, y_train):
  - impute.fit(X_train):
    - averages are computed using the train partition only
  - select.fit(X_train,y_train):
    - features are selected using the train partition only.
  - knn.fit(X_train,y_train):
    - model is trained on training data
- clf.predict(X_test):
  - impute.transform(X_test):
    - averages computed previously are used for imputation
  - select.transform(...):
    - features chosen previously are selected
  - knn.predict(...):
    - predictions are computed

```
imputer =  SimpleImputer(strategy='mean')
selector = SelectKBest(f_regression, k=3)
knn = KNeighborsRegressor()

clf = Pipeline([
        ('impute', imputer),
        ('select', selector),
        ('knn_regression', knn)])

clf = clf.fit(X_train, y_train)
y_test_pred = clf.predict(X_test)
```



Hyper-parameters of pipelines

- The hyper-parameters of a pipeline is the union of the hyper-parameters of each of the steps.
- The names of the hyper-parameters are: stepname__hyperparametername

```
imputer =  SimpleImputer(strategy='mean')
selector = SelectKBest(f_regression)
knn = KNeighborsRegressor()
clf = Pipeline([
        ('impute', imputer),
        ('select', selector),
        ('knn_regression', knn)])
```

select__k = how many features to select
knn_regression__n_neighbors = how many neighbors

Hyper-parameter tuning of pipelines

- Pipeline hyper-parameters can also be tuned

```
from sklearn.model_selection import GridSearchCV

imputer =  SimpleImputer(strategy='mean')
selector = SelectKBest(f_regression, k=3)
knn = KNeighborsRegressor()

# Defining the pipeline
clf = Pipeline([
        ('impute', imputer),
        ('select', selector),
        ('knn_regression', knn)])
```

Hyper-parameter tuning of pipelines

- Pipeline hyper-parameters can also be tuned

```
from sklearn.model_selection import GridSearchCV

imputer =  SimpleImputer(strategy='mean')
selector = SelectKBest(f_regression, k=3)
knn = KNeighborsRegressor()

# Defining the pipeline
clf = Pipeline([
        ('impute', imputer),
        ('select', selector),
        ('knn_regression', knn)])

# Defining hyper-parameter space
from sklearn.model_selection import GridSearchCV
param_grid = {
        'select__k': [2,3,4],
        'knn_regression__n_neighbors': [1,3,5]
        }
```

Hyper-parameter tuning of pipelines

- Pipeline hyper-parameters can also be tuned

```
from sklearn.model_selection import GridSearchCV

imputer =  SimpleImputer(strategy='mean')
selector = SelectKBest(f_regression, k=3)
knn = KNeighborsRegressor()

# Defining the pipeline
clf = Pipeline([
        ('impute', imputer),
        ('select', selector),
        ('knn_regression', knn)])

# Defining hyper-parameter space
from sklearn.model_selection import GridSearchCV
param_grid = {
        'select__k': [2,3,4],
        'knn_regression__n_neighbors': [1,3,5]
        }
```

```
# Defining a 5-fold crossvalidation grid-search
clf_grid = GridSearchCV(clf,
                   param_grid,
                   scoring=‘neg_mean_squared_error’,
                   cv=5 , n_jobs=1, verbose=1)

clf_grid  = clf_grid.fit(X_train, y_train)

# The tuned method can be used for making predictions,
just as any fit machine learning method
y_test_pred = clf_grid.predict(X_test)
```

Hyper-parameter tuning of pipelines

- Pipeline hyper-parameters can also be tuned

```
from sklearn.model_selection import GridSearchCV

imputer =  SimpleImputer(strategy='mean')
selector = SelectKBest(f_regression, k=3)
knn = KNeighborsRegressor()

# Defining the pipeline
clf = Pipeline([
        ('impute', imputer),
        ('select', selector),
        ('knn_regression', knn)])

# Defining hyper-parameter space
from sklearn.model_selection import GridSearchCV
param_grid = {
        'select__k': [2,3,4],
        'knn_regression__n_neighbors': [1,3,5]
        }
```

```
# Defining a 5-fold crossvalidation grid-search
clf_grid = GridSearchCV(clf,
                    param_grid,
                    scoring='neg_mean_squared_error',
                    cv=5 , n_jobs=1, verbose=1)

clf_grid  = clf_grid.fit(X_train, y_train)

# The tuned method can be used for making predictions,
just as any fit machine learning method
y_test_pred = clf_grid.predict(X_test)

# The best hyper-parameter values (and their scores)
can be accessed
clf_grid.best_params_
Out[]: {'knn_regression__n_neighbors': 5, 'select__k':
3}

clf_grid.best_score_
Out[]: -20.14685427728613
```

Hyper-parameter tuning of pipelines

- We can even get the optimized pipeline itself:

```
clf_grid.best_estimator_
Out[]:
Pipeline(memory=None,
        steps=[('impute',
                SimpleImputer(add_indicator=False, copy=True, fill_value=None,
                              missing_values=nan, strategy='mean',
                              verbose=0)),
               ('select',
                SelectKBest(k=3,
                            score_func=<function f_regression at 0x0000012D3D2FC798>)),
               ('knn_regression',
                KNeighborsRegressor(algorithm='auto', leaf_size=30,
                                    metric='minkowski', metric_params=None,
                                    n_jobs=None, n_neighbors=5, p=2,
                                    weights='uniform'))],
        verbose=False)
```

Hyper-parameter tuning of pipelines

- Note: if needed, all pipeline hyper-parameters can be obtained with method *.get_params()*

```
clf.get_params()


 'impute__add_indicator': False,
 'impute__copy': True,
 'impute__fill_value': None,
 'impute__missing_values': nan,
 'impute__strategy': 'mean',
 'impute__verbose': 0,
 'select__k': 10,
 'select__score_func': <function sklearn.feature_selection.univariate_selection.f_regression(X, y, center=True)>,
 'knn_regression__algorithm': 'auto',
 'knn_regression__leaf_size': 30,
 'knn_regression__metric': 'minkowski',
 'knn_regression__metric_params': None,
 'knn_regression__n_jobs': None,
 'knn_regression__n_neighbors': 5,
 'knn_regression__p': 2,
 'knn_regression__weights': 'uniform'}
```

Hyper-parameter tuning of pipelines

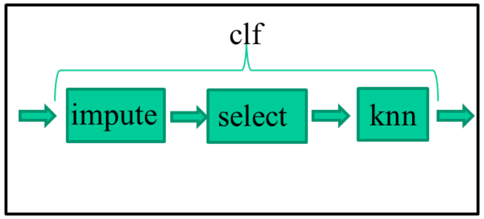
- and they can also be set with *.set_params*, like this:

```
clf = clf.set_params(**{'knn_regression__n_neighbors':10})
clf.get_params()


 'impute__add_indicator': False,
 'impute__copy': True,
 'impute__fill_value': None,
 'impute__missing_values': nan,
 'impute__strategy': 'mean',
 'impute__verbose': 0,
 'select__k': 10,
 'select__score_func': <function sklearn.feature_selection.univariate_selection.f_regression(X, y, center=True)>,
 'knn_regression__algorithm': 'auto',
 'knn_regression__leaf_size': 30,
 'knn_regression__metric': 'minkowski',
 'knn_regression__metric_params': None,
 'knn_regression__n_jobs': None,
 'knn_regression__n_neighbors': 10,
 'knn_regression__p': 2,
 'knn_regression__weights': 'uniform'}
```

Caching steps in a pipeline

- For hyper-parameter tuning, some of the transformers in the pipeline should be fitted just once
- For example, ordering the features should be done only once (in principle, the same ordering of features is going to be obtained everytime).

```
param_grid = {
        'select__k': [2,3,4],
        'knn_regression__n_neighbors': [1,3,5]
    }
```



- A cache can be used (however, notice that loading the cache from disk can be slow)

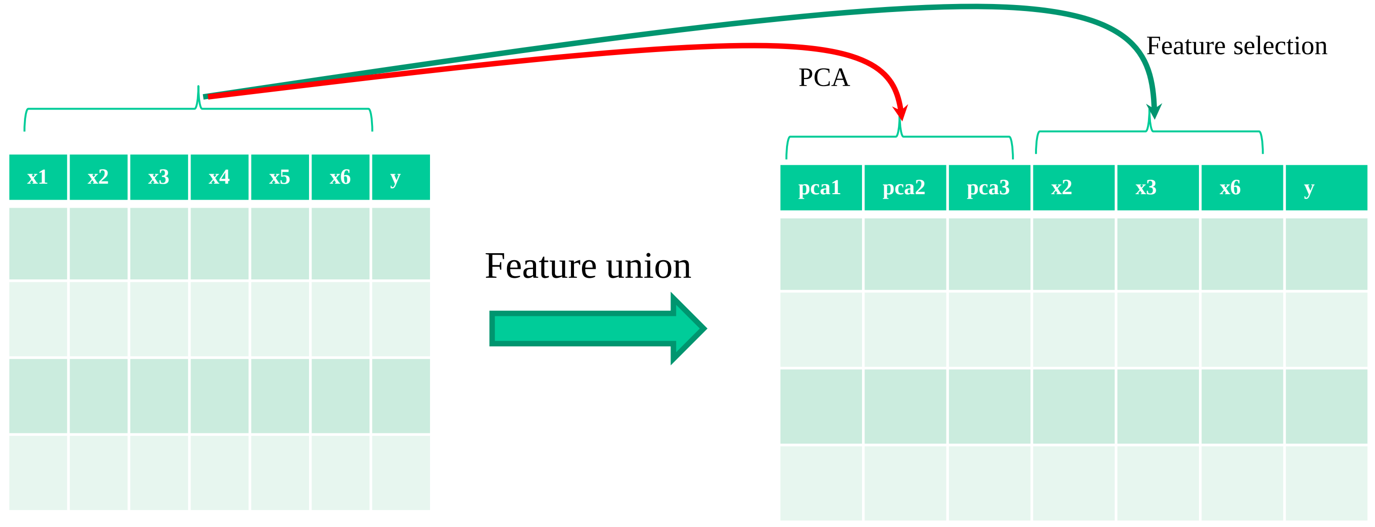
Caching steps in a pipeline

```
from sklearn.model_selection import GridSearchCV
from tempfile import mkdtemp
from shutil import rmtree
from joblib import Memory

imputer =  SimpleImputer(strategy='mean')
selector = SelectKBest(f_regression, k=3)
knn = KNeighborsRegressor()

cachedir = mkdtemp()
memory = Memory(location=cachedir, verbose=10)
memory = Memory(verbose=10)

# Select is going to be cached
clf = Pipeline([
        ('impute', imputer),
        ('select', selector),
        ('knn_regression', knn)],
      memory = memory)
```

```
# Defining hyper-parameter space
param_grid = {
        'select__k': [2,3,4],
        'knn_regression__k': [1,3,5]
        }

# Defining a 5-fold crossvalidation grid-search
clf_grid = GridSearchCV(clf,
                   param_grid,
                   scoring='neg_mean_squared_error',
                   cv=5 , n_jobs=1, verbose=1)

clf_grid = clf_grid.fit(X_train, y_train)
y_test_pred = clf_grid.predict(X_test)

# Delete the temporary cache before exiting
rmtree(cachedir)
```

Feature Unions

- Let's suppose that we want to use both PCA feature extraction/reduction and standard feature selection.
- Feature Unions allow to define a step in the pipeline that combines features (attributes) obtained from two different sources.

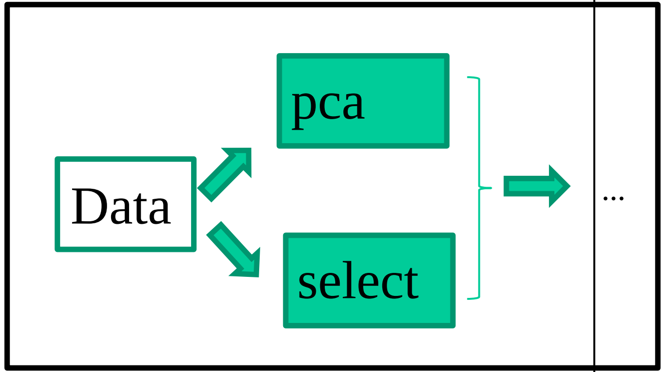


Feature Unions

```
# Just importing modules and preparing the data
from sklearn.pipeline import Pipeline, FeatureUnion
from sklearn.model_selection import GridSearchCV
from sklearn.neighbors import KNeighborsRegressor
from sklearn.datasets import load_iris
from sklearn.decomposition import PCA
from sklearn.feature_selection import SelectKBest
from sklearn.model_selection import train_test_split


iris = load_iris()
X, y = iris.data, iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=33)
```

Feature Unions

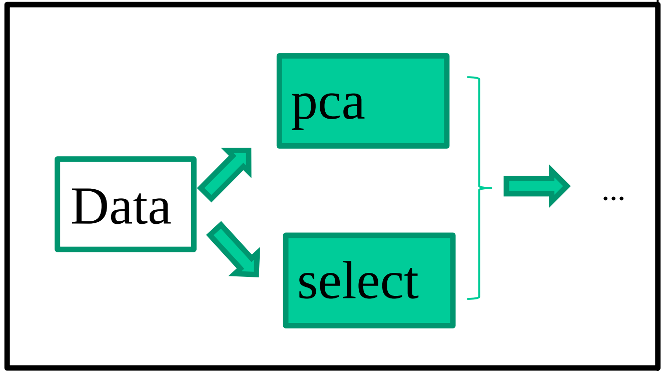
```
# Now, we prepare the two sources of features/attributes: PCA and Feature Selection
# We compute two features from each
pca = PCA(n_components=2)
selection = SelectKBest(k=2)

# Build estimator from PCA and selection:

combined_features = FeatureUnion([("pca",    pca),
                                  ("select", selection)])
```



Feature Unions

Feature Unions can be used as a standalone transformer. We fit it with the training data and use it to transform both training and test.

```
# ...
# Build estimator from PCA and selection:
combined_features = FeatureUnion([("pca",     pca),
                                  ("select", selection)])

combined_features = combined_features.fit(X, y)

X_train_new = combined_features.transform(X_train)
X_test_new = combined_features.transform(X_test)

print("Combined space has", X_train_new.shape[1], "features")
Combined space has 4 features
```



Feature Unions

Original dataset

```
array([[5.6, 2.9, 3.6, 1.3],
       [6.5, 3. , 5.5, 1.8],
       [5.4, 3.9, 1.7, 0.4],
       [7. , 3.2, 4.7, 1.4],
       [5.8, 2.8, 5.1, 2.4],
       [7.7, 2.6, 6.9, 2.3],
       [5.5, 2.5, 4. , 1.3],
       [5.9, 3.2, 4.8, 1.8],
       [4.9, 3.6, 1.4, 0.1],
```

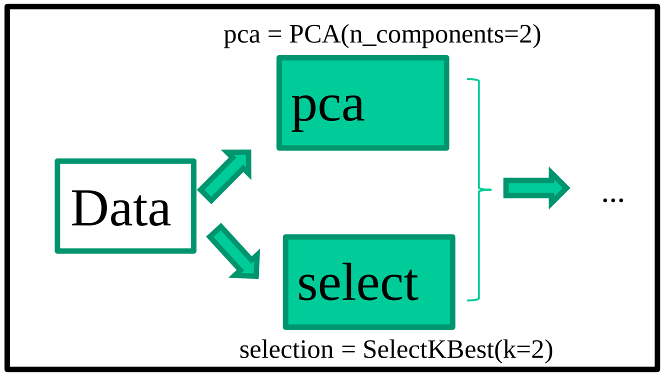
...

Transformed dataset

2 PCA's   2 selected features

```
array([[-0.17392537, -0.25485421,  3.6        ,  1.3        ],
       [ 1.94968906,  0.04194326,  5.5        ,  1.8        ],
       [-2.28085963,  0.74133045,  1.7        ,  0.4        ],
       [ 1.28482569,  0.68516047,  4.7        ,  1.4        ],
       [ 1.58592822, -0.53964071,  5.1        ,  2.4        ],
       [ 3.79564542,  0.25732297,  6.9        ,  2.3        ],
       [ 0.16641322, -0.68192672,  4.         ,  1.3        ],
       [ 1.11628318, -0.08461685,  4.8        ,  1.8        ],
       [-2.80068412,  0.26864374,  1.4        ,  0.1        ],
```

...

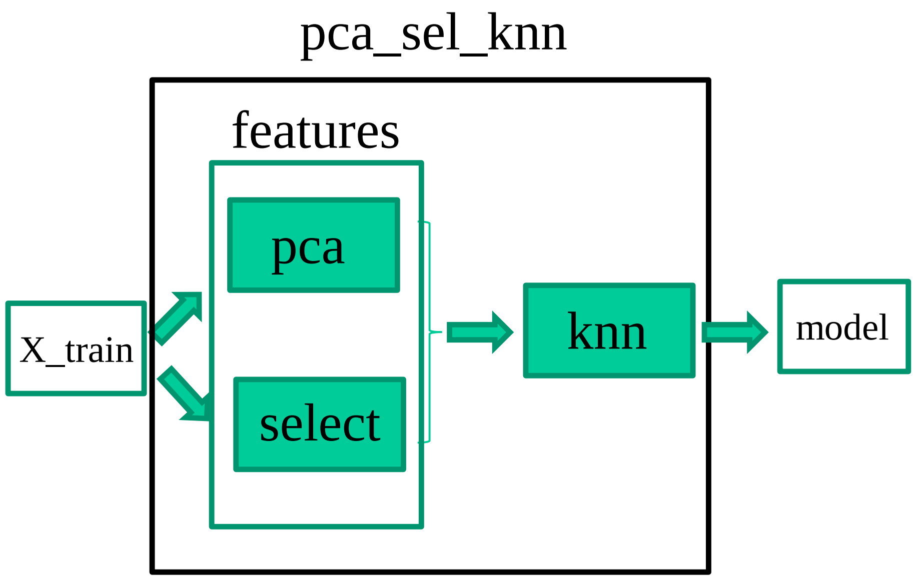


```
pca = PCA(n_components=2)
selection = SelectKBest(k=2)
```

Feature Unions

Feature Unions can also be used as a transformer step in a pipeline.

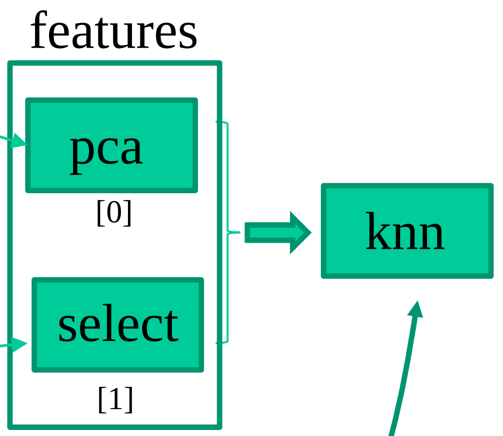
```
# ...
# Build estimator from PCA and selection:
combined_features =
  FeatureUnion([("pca",     pca),
                ("select", selection)])
knn = KNeighborsRegressor()
# Construct the pipeline of pca&select + knn
pca_sel_knn =
  Pipeline([("features", combined_features),
            ("knn", knn)])
# Fit it
pca_sel_knn = pca_sel_knn.fit(X_train, y_train)

# And use it for making predictions for the train and test datasets
pred_train = pca_sel_knn.predict(X_train)
pred_test = pca_sel_knn.predict(X_test)
```



Feature Unions

We can still Access each one of the steps in the pipeline

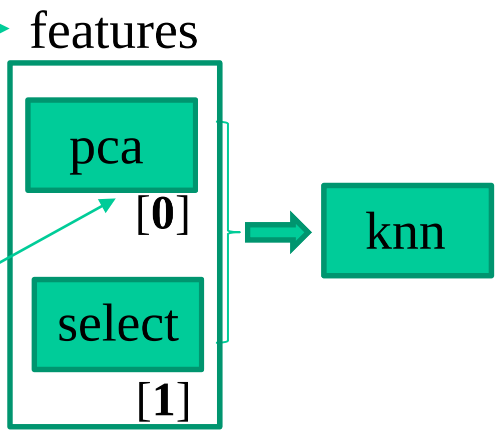
```
pca_sel_knn['features'].transformer_list[0]
Out[]: ('pca',
PCA(copy=True, iterated_power='auto', n_components=2,
random_state=None,
svd_solver='auto', tol=0.0, whiten=False))

pca_sel_knn['features'].transformer_list[1]
Out[]: ('select',
SelectKBest(k=2, score_func=<function f_classif at
0x0000012D3D2F7EE8>))

pca_sel_knn['knn']
Out[]: KNeighborsRegressor(algorithm='auto', leaf_size=30,
metric='minkowski',
metric_params=None, n_jobs=None, n_neighbors=5, p=2,
weights='uniform')
```



Feature Unions

... and use the individual steps to transform data!

```
X_train_transformed = pca_sel_knn['features'].transform(X_train)
print(X_train_transformed[:5,:])
Out[]:
[[-0.0 -0.2   3.6   1.3 ]
[ 2.0    0.0   5.5   1.8 ]
[-2.1    0.7   1.7   0.4 ]
[ 1.3    0.6   4.7   1.4 ]
[ 1.6   -0.5   5.1   2.4 ]]


X_train_transformed =
pca_sel_knn['features'].transformer_list[0][1].transform(X_train)
print(X_train_transformed[:5,:])
Out[]: [[-0.0 -0.2]
          [ 2.0 0.0]
          [-2.1 0.7]
          [ 1.3 0.6]
          [ 1.6 -0.5]]
```



Feature Unions: exercise

- Create a FeatureUnion that selects the first more relevant attribute according to three ranking methods:
  - f_classif
  - mutual_info_classif
  - chi2
- First, use it as standalone transformer and check that it works (that when used to transform a dataset (X_test, for instance), three features are created).
- And then use it into a pipeline together with knn. Fit the pipeline, and check that the three features are being created. You will need to access the FeatureUnion step in the pipeline and use it to transform a dataset (X_test, for instance), and see that three features are created.
- This transformer is not very useful, as the three methods will usually select the same attribute. Just for practising.

Feature Unions: exercise

```
from sklearn.pipeline import Pipeline, FeatureUnion
from sklearn.neighbors import KNeighborsRegressor
from sklearn.datasets import load_iris
from sklearn.feature_selection import SelectKBest, f_classif,
mutual_info_classif, chi2
from sklearn.model_selection import train_test_split

iris = load_iris()
X, y = iris.data, iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33,
random_state=33)

first_selector = SelectKBest(score_func=f_classif, k=1)
second_selector =  SelectKBest(score_func=mutual_info_classif, k=1)
third_selector = SelectKBest(score_func=chi2, k=1)

...
```

Feature Unions: exercise

```
...
# Combine the three features:
combined_features = FeatureUnion([("f1",     first_selector),
                                  ("f2", second_selector),
                                  ("f3", third_selector)])

# Here, we use combined_features as a standalone transformer
combined_features = combined_features.fit(X_train, y_train)
new_X_test = combined_features.transform(X_test)

# We see that three features have been created
new_X_test[:5,:]

Out[]:
array([[4.2, 1.3, 4.2],
       [4.4, 1.4, 4.4],
       [1.6, 0.2, 1.6],
       [4.6, 1.5, 4.6],
       [5.6, 1.4, 5.6]])
```

Feature Unions: exercise

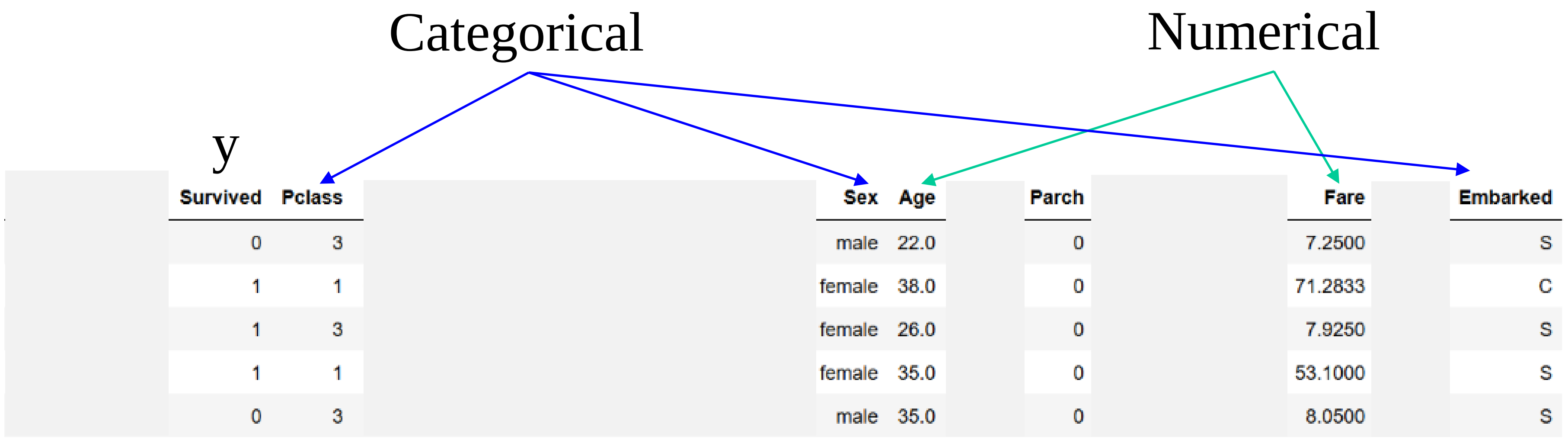
```
...
# Combine the three features:
combined_features = FeatureUnion([("f1",     first_selector),
                                  ("f2", second_selector),
                                  ("f3", third_selector)])

knn = KNeighborsRegressor()
# Construct the pipeline
f1f2f3_knn = Pipeline([("features", combined_features),
                       ("knn", knn)])
# Fit it
f1f2f3_knn = f1f2f3_knn.fit(X_train, y_train)
# We access to the 'features' step of the trained pipeline and use it to transform the test set
new_X_test = f1f2f3_knn['features'].transform(X_test)
# We see that the new data matrix has three features
print(new_X_test[:5,:])

Out[]:
array([[4.2, 1.3, 4.2],
       [4.4, 1.4, 4.4],
       [1.6, 0.2, 1.6],
       [4.6, 1.5, 4.6],
       [5.6, 1.4, 5.6]])
```

Transforming individual features

- Up to now, all pre-processing steps process all attributes in the dataset
- But in some cases, different attributes/features need to follow different pre-processing steps.
- For instance, categorical attributes should undergo some pre-processing and numerical attributes some other pre-processing.
- **ColumnTransformer** can be used for that
- Important: all pre-processing steps in a pipeline transform numpy arrays into numpy arrays, but ColumnTransformer can start from Pandas dataframes (and transform them into numpy arrays)

Transforming individual features

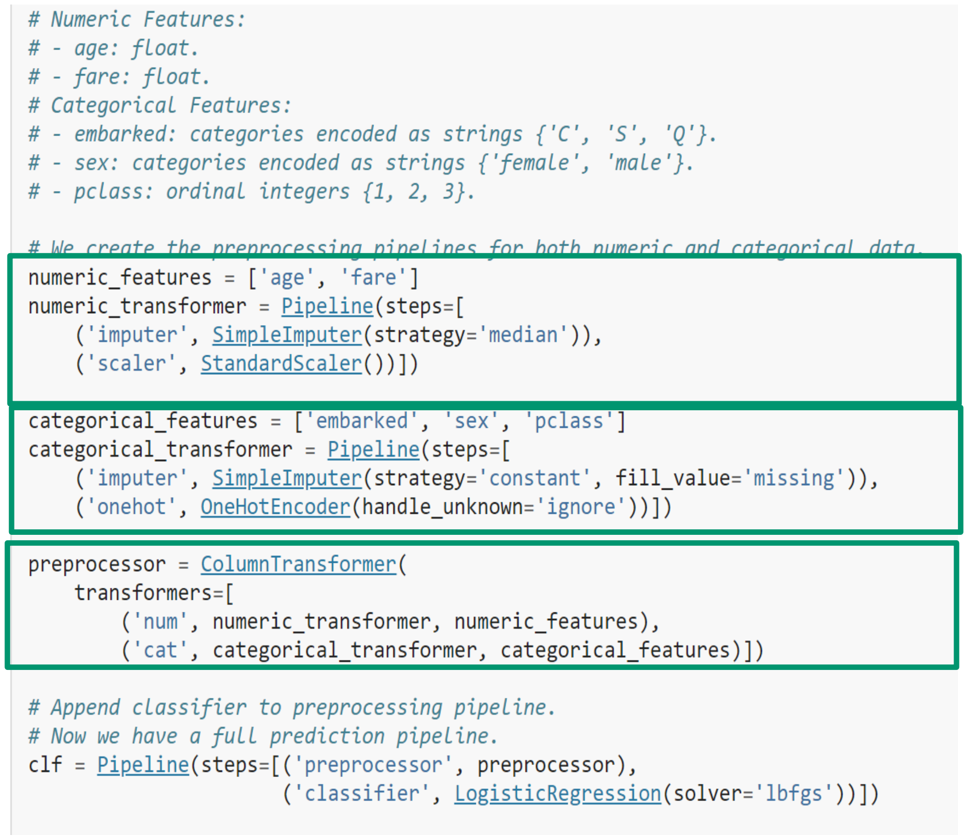
- Let's suppose that we start with the titanic dataset which is a **Pandas dataframe**


Categorical
Numerical
y

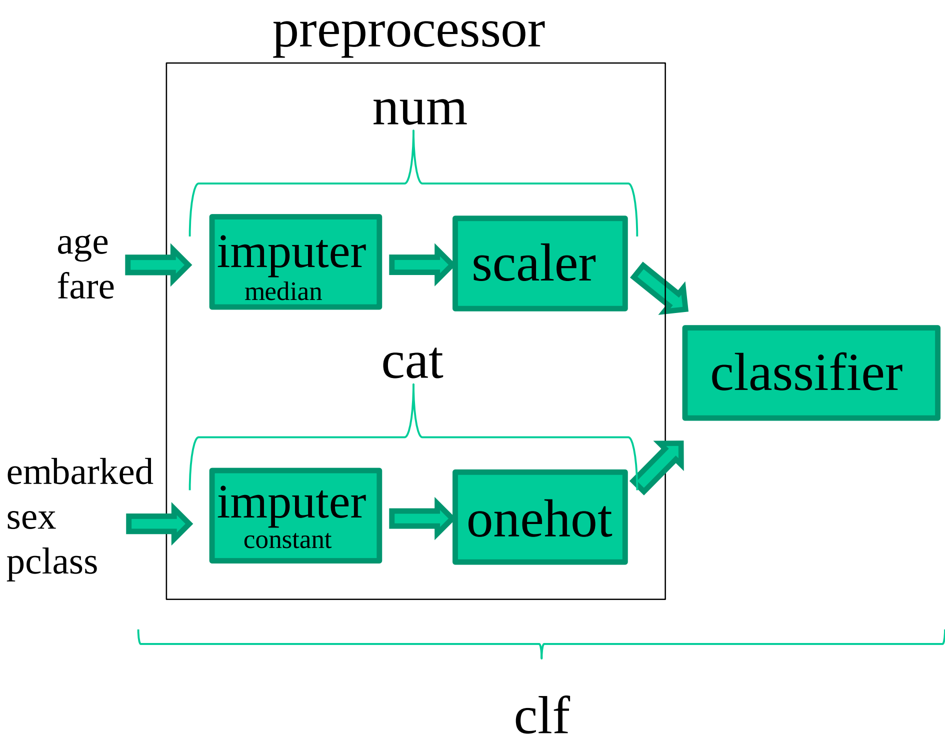
Survived	Pclass	Sex	Age	Parch	Fare	Embarked
0	3	male	22.0	0	7.2500	S
1	1	female	38.0	0	71.2833	C
1	3	female	26.0	0	7.9250	S
1	1	female	35.0	0	53.1000	S
0	3	male	35.0	0	8.0500	S

Transforming individual features

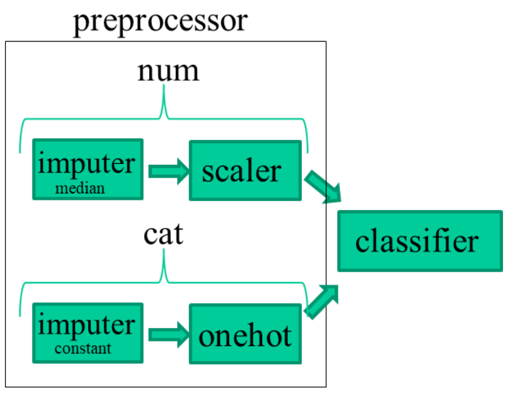
- Each attribute or each type of attribute (numeric, categorical, ...) can be transformed in a different way
  - https://scikit-learn.org/stable/modules/compose.html#pipeline



```
# Numeric Features:
# - age: float.
# - fare: float.
# Categorical Features:
# - embarked: categories encoded as strings {'C', 'S', 'Q'}.
# - sex: categories encoded as strings {'female', 'male'}.
# - pclass: ordinal integers {1, 2, 3}.

# We create the preprocessing pipelines for both numeric and categorical data.
numeric_features = ['age', 'fare']
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())])

categorical_features = ['embarked', 'sex', 'pclass']
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='constant', fill_value='missing')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))])

preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)])

# Append classifier to preprocessing pipeline.
# Now we have a full prediction pipeline.
clf = Pipeline(steps=[('preprocessor', preprocessor),
                      ('classifier', LogisticRegression(solver='lbfgs'))])
```



Transforming individual features



```
# Numeric Features:
# - age: float.
# - fare: float.
# Categorical Features:
# - embarked: categories encoded as strings {'C', 'S', 'Q'}.
# - sex: categories encoded as strings {'female', 'male'}.
# - pclass: ordinal integers {1, 2, 3}.

# We create the preprocessing pipelines for both numeric and categorical data.
numeric_features = ['age', 'fare']
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())])

categorical_features = ['embarked', 'sex', 'pclass']
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='constant', fill_value='missing')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))])

preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)])

# Append classifier to preprocessing pipeline.
# Now we have a full prediction pipeline.
clf = Pipeline(steps=[('preprocessor', preprocessor),
                      ('classifier', LogisticRegression(solver='lbfgs'))])
```

Hyper-parameters can be accessed with the usual __ notation:

preprocessor__num__imputer__strategy

and they can be set with:
clf.set_params(**{'preprocessor__num__imputer__strategy': '**mean**'})

Pipeline persistence

- Trained pipelines can be saved into a file in pickle format, to be used later
- Caution! if the version of sklearn changes, or a different architecture is used (e.g. saving in Windows10 and loading in Linux), this would lead to unexpected results

```
from joblib import dump, load
dump(pca_sel_knn, 'pca_sel_knn.joblib')
pca_sel_knn = load('pca_sel_knn.joblib')
```

Function transformers

- There are cases where you want to do some pre-processing, but sklearn does not provide that operation to be included in your pipeline.
- If the pre-processing is done with a Python function, that function can be used as a transformer

Function transformers

- Let's suppose a very simple case, where we want a transformer that removes the first column (because, for instance, we know it is an identifier, useless for prediction).
- This is a function that removes the first column (0) of a numpy dataframe

```
def drop_first_column(X):
  return X[:, 1:]
```

from sklearn.preprocessing import FunctionTransformer

Function transformers

```
def drop_first_column(X):
  return X[:, 1:]
```

- And this is the way to use it as a step in a pipeline:

```
from sklearn.preprocessing import FunctionTransformer

knn = KNeighborsRegressor()
remove_column_1 = FunctionTransformer(drop_first_column)
pipe = Pipeline([
        ('drop_col_1', remove_column_1),
        ('knn', knn)
        ])
```

Creating new transformers for pipelines

- There are cases where you want to do some pre-processing, but sklearn does not provide that operation to be included in your pipeline.
- And functionTransformer cannot be used.
- But you can extend sklearn by creating your own new pre-processing steps.
- We are going to program a transformer for "getting just the first colum" (although this is so simple that it could also be achieved via FunctionTransformer).

A simple (not very useful) transformer

- Get the first attribute/column of the input attributes
- We asume that the data matrix is a numpy matrix
- Two methods have to be defined: fit and transform

```
class get_one_col(TransformerMixin):
    def __init__(self):
        pass

    def fit(self, X, y=None):
        return(self)

    def transform(self, X):
        return(X[:,[0]])
```

A simple transformer (selecting first column)

- Before, going deeper into the definition of our new transformer, let's see how it would be used in practice.

A simple transformer (selecting first column)

- Let's try it
- We first import some modules and define my transformer

```
from sklearn.datasets import load_iris
from sklearn.pipeline import Pipeline
from sklearn.neighbors import KNeighborsRegressor
from sklearn.base import TransformerMixin

iris = load_iris()
X, y = iris.data, iris.target

class get_one_col (TransformerMixin):
    def __init__(self):
        pass

    def fit(self, X, y=None):
        return(self)

    def transform(self, X):
        return(X[:,[0]])
```

A simple transformer (selecting first column)

- Now, my transformer is initialized
- and then fitted

```
one_col_trans = get_one_col ()
one_col_trans.fit(X,y)
```

A simple transformer (selecting first column)

- and now, we apply the transformer
- We see that the first column was selected, as expected

```
# X before transformation
print(X[:3,:])
[[5.1 3.5 1.4 0.2]
 [4.9 3.   1.4 0.2]
 [4.7 3.2 1.3 0.2]]

# X after transformation

XX = one_col_trans.transform(X)

print(XX[:10,:])
[[5.1]
 [4.9]
 [4.7]]
```

A simple transformer (selecting first column)

This is the name of your transformer

```
class get_one_col(TransformerMixin):
    def __init__(self):
        pass

    def fit(self, X, y=None):
        return(self)

    def transform(self, X):
        return(X[:,[0]])
```

This is to specify that you want to define a new transformer

This is to specify what to do when creating the transformer (in this case, we do nothing: pass)

self is the transformer itself

This is how we create our new transformer:
one_col_trans = **get_one_col** ()

A simple transformer (selecting first column)

```
class get_one_col(TransformerMixin):
    def __init__(self):
        pass

    def fit(self, X, y=None):
        return(self)

    def transform(self, X):
        return(X[:,[0]])
```

- **fit** is the operation that trains the transformer.
- This particular transformer always selects column 0, independently of the training data.
- Therefore, **.fit** just returns the transformer (**self**) without changing it. That is, **.fit** does nothing.

.transform is the operation that transforms the data. In this case, we just select column 0

A simple transformer (selecting first column)

- Let's try it
- We first import some modules and define my transformer

```
from sklearn.datasets import load_iris
from sklearn.pipeline import Pipeline
from sklearn.neighbors import KNeighborsRegressor
from sklearn.base import TransformerMixin

iris = load_iris()
X, y = iris.data, iris.target

class get_one_col (TransformerMixin):
    def __init__(self):
        pass

    def fit(self, X, y=None):
        return(self)

    def transform(self, X):
        return(X[:,[0]])
```

A simple transformer (selecting first column)

- Now, my transformer is initialized
- and then fitted
- In this simple case, fitting does nothing

```
one_col_trans = get_one_col ()
one_col_trans.fit(X,y)
```

```
class get_one_col (TransformerMixin):
    def __init__(self):
        pass

    def fit(self, X, y=None):
        return(self)

    def transform(self, X):
        return(X[:,[0]])
```

A simple transformer (selecting first column)

- And now, we apply the transformer
- We see that the first column was selected, as expected

```
# X before transformation
print(X[:3,:])
[[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]]

# X after transformation

XX = one_col_trans.transform(X)

print(XX[:10,:])
[[5.1]
 [4.9]
 [4.7]]
```

Using our transformer in a pipeline

- Our simple transformer can be used a step in a pipeline

```
one_col_trans = get_one_col()
knn = KNeighborsRegressor()

pipe = Pipeline([
        ('one_col', one_col_trans),
        ('knn', knn)
        ])

# Our pipeline is trained and knn is trained
# with just the first column (because that is
# what our transformer does)
pipe = pipe.fit(X,y)
```

A new transformer: exercise

- Program a transformer that returns a single column, which is the summation of all the input columns.
- You can sum all columns by using:

```
np.sum(X, axis=1, keepdims=True)
```

- That means that we add all the elements column-wise
- Check that it works as a standalone transformer

A new transformer: exercise

- Program a transformer that returns a single column, which is the summation of all the input columns.
- You can do that by using:

  np.sum(X, axis=1, keepdims=True)
- That means that we add all the elements column-wise
- keepdims=True is needed so that the final result is a matrix with one column, and not a vector (a vector is not a matrix).

```
X[:5,:]
Out[]:
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2],
       [4.6, 3.1, 1.5, 0.2],
       [5. , 3.6, 1.4, 0.2]])

XX=np.sum(X, axis=1, keepdims=True)

XX[:5,:]
Out[60]:
array([[10.2],
       [ 9.5],
       [ 9.4],
       [ 9.4],
       [10.2]])
```

A new transformer: exercise

```
from sklearn.datasets import load_iris
from sklearn.pipeline import Pipeline
from sklearn.neighbors import KNeighborsRegressor
from sklearn.base import TransformerMixin
import numpy as np

iris = load_iris()
X, y = iris.data, iris.target

class get_one_col (TransformerMixin):
    def __init__(self):
        pass

    def fit(self, X, y=None):
        return(self)

    def transform(self, X):
        return(<PUT CODE HERE>)
```

A new transformer: exercise

```
from sklearn.datasets import load_iris
from sklearn.pipeline import Pipeline
from sklearn.neighbors import KNeighborsRegressor
from sklearn.base import TransformerMixin
from sklearn.model_selection import train_test_split
import numpy as np

iris = load_iris()
X, y = iris.data, iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=33)

class get_one_col(TransformerMixin):
    def __init__(self):
        pass

    def fit(self, X, y=None):
        return(self)

    def transform(self, X):
        return(np.sum(X, axis=1, keepdims=True))
```

A new transformer: exercise

```
# Checking that it Works as a standalone transformer
# First, we initialize the transformer
one_col = get_one_col()
# Then, we fit it with the training data
one_col = one_col.fit(X_train,y_train)
# Finally, we use it to transform X
new_X_test = one_col.transform(X_test)

new_X_test[:5,:]
Out[]:
array([[14.1],
       [15.6],
       [ 9.7],
       [15.4],
       [15.7]])
```

A more complicated transformer

- This one is going to do imputation of numerical attributes, but using the first quartile instead of the median or the mean:

In this case, fitting the transformer puts some information inside the transformer (self)

```
from sklearn.base import TransformerMixin
import numpy as np

class SimpleImputerQuartile(TransformerMixin):
    def __init__(self):
        pass

    def fit(self, X, y=None):
        # nanquantile computes quantiles, while ignoring nan
        self.statistics_ = np.nanquantile(X, 0.25, axis = 0)
        return(self)

    def transform(self, X):
        for j in range(X.shape[1]):
            for i in range(X.shape[0]):
                if(np.isnan(X[i,j])):
                    X[i,j]=self.statistics_[j]
        return(X)
```

A more complicated transformer

- Let's analyze the **.fit** method

```
def fit(self, X, y=None):
    self.statistics_ = np.nanquantile(X, 0.25, axis = 0)
    return(self)
```

<u>input X</u>

```
print(X[:5,:])
[[nan 3.5 1.4 0.2]
 [4.9 nan 1.4 0.2]
 [4.7 3.2 nan 0.2]
 [4.6 3.1 1.5 nan]
 [5.  3.6 1.4 0.2]]
```

<u>np.nanquantile returns the 1/4 quantile (first quartile</u>

```
np.nanquantile(X, 0.25, axis = 0)

Out[]: array([5.1, 2.8, 1.6, 0.3])
```

A more complicated transformer

- This show what it is meant by .fit putting some information inside the transformer (self)

```
def fit(self, X, y=None):
    self.statistics_ = np.nanquantile(X, 0.25, axis = 0)
    return(self)
```

```
# Here, we create the transformer
my_quartile_imputer = SimpleImputerQuartile()
# And then, we train it
my_quartile_imputer = my_quartile_imputer.fit(X,y)
# And once trained, there is information inside the transformer
print(my_quartile_imputer.statistics_)
Out[]: array([5.1, 2.8, 1.6, 0.3])
```

A more complicated transformer

- Let's analyze the **.transform** method

```
def transform(self, X):
        for j in range(X.shape[1]):
            for i in range(X.shape[0]):
                if(np.isnan(X[i,j])):
                    X[i,j]=self.statistics_[j]
        return(X)
```

- It goes through all the columns (j) of X and then through all the rows (i) of column j
- If X[i,j] is np.nan, then it is replaced by the first quartile of column j, which is contained in self.statistics_[j]

A more complicated transformer

- Let's analyze the **.transform** method

```
def transform(self, X):
    for j in range(X.shape[1]):
        for i in range(X.shape[0]):
            if(np.isnan(X[i,j])):
                X[i,j]=self.statistics_[j]
    return(X)
```

input X

```
print(X[:5,:])
[[nan 3.5 1.4 0.2]
 [4.9 nan 1.4 0.2]
 [4.7 3.2 nan 0.2]
 [4.6 3.1 1.5 nan]
 [5.  3.6 1.4 0.2]]
```

```
my_quartile_imputer = SimpleImputerQuartile()

my_quartile_imputer = my_quartile_imputer.fit(X,y)

print(my_quartile_imputer.statistics_)
[5.1 2.8 1.6 0.3]

XX = my_quartile_imputer.transform(X)

print(XX[:5,:])
[[5.1 3.5 1.4 0.2]
 [4.9 2.8 1.4 0.2]
 [4.7 3.2 1.6 0.2]
 [4.6 3.1 1.5 0.3]
 [5.  3.6 1.4 0.2]]
```

A more complicated transformer

- We can also use our SimpleImputerQuartile in a pipeline:

```
quartile_imputer = SimpleImputerQuartile()
knn = KNeighborsRegressor()

qi_knn = Pipeline([
        ('quartile_imputer', quartile_imputer),
        ('knn', knn)
        ])

pipe = qi_knn.fit(X,y)
```