SIAD M2 DS – 2024/2025

# Machine Learning – ML Engineering

Nadarajen Veerapen

Faculté des sciences économiques, sociales et des territoires – Université de Lille

# Feature Engineering

# Feature engineering

Feature engineering consists in transforming, adding or combining features using **domain knowledge**.

**Examples**

Interaction features (products of features) can help linear models.

While tree-based models are only concerned by the order of features, linear models and neural networks are strongly tied to the scale and distribution of each feature. If there is a non-linear relationship between the variable and the target, it can be difficult to model, especially in regression. Functions like log and exp can help by adjusting the relative scale of the data.

Most models work better if each feature, and the target in a regression context, are vaguely Gaussian. Using log and exp can be a simple way of achieving this.

# Feature Selection

# Feature Selection

There are many ways to create new features (like one-hot-encoding, creating polynomial features, non-linear transformations, ...) and some data sets also have many features from the start.

However, more features make for more complex models, increasing the risk of overfitting. It can be a good idea to reduce the number of features to only the most useful ones, discarding the rest.

**3 basic strategies for feature selection**

- Univariate statistics
- Model-based selection
- Iterative selection

Outside of these, one of the simplest approaches is to remove a feature based on a variance threshold.

```
from sklearn.feature_selection import VarianceThreshold
selection = VarianceThreshold(threshold) # the default is to remove zero-
variance features, i.e. same value everywhere
X_sel = selection.fit_transform(X)
```

# Univariate Statistics

The principe of **univariate statistics** feature selection is to compute whether there is a statistically significant relationship between each feature and the target. Features with the highest confidence will be selected.

In scikit-learn, we need to choose **a test** and a **method to discard features** based on the p-values found by the test.

**Tests**

- For regression: `f_regression`, `mutual_info_regression`
- For classification: `chi2`, `f_classif`, `mutual_info_classif`

**Some selection methods**

- `SelectKBest` – keeps the features with the k best values
- `SelectPercentile` – selects a fixed percentage of features

```
from sklearn.feature_selection import SelectPercentile, chi2
selection = SelectPercentile(chi2, percentile)
X_train_sel = selection.fit_transform(X_train)
```

# Model-Based Feature Selection

In model-based feature selection, a **supervised ML model** is used to judge the importance of each feature. The model used for feature selection does not need to be the same one as the one used for the final learning and prediction.

The feature selection model needs to be able to provide feature importance information:

- L1-based feature selection, e.g. LogisticRegression
- Tree-based feature selection, e.g. RandomForestClassifier

```
from sklearn.feature_selection import SelectFromModel
from sklearn.ensemble import RandomForestClassifier
selection = SelectFromModel(
        RandomForestClassifier(n_estimators=100, random_state=42),
        threshold="median")
selection.fit(X_train, y_train)
X_train_sel = selection.transform(X_train)
```

# Iterative Feature Selection

In univariate testing, no model is used. In model-based selection, a single model is used. In iterative feature selection, a series of models are built, with varying numbers of features. One specific method is **Recursive Feature Elimination (RFE)**.

RFE:

- Start with all features;
- Build a model and discard least important feature according to the model;
- Build a new model with the remaining features and so on until a specified number of features is reached.

As with model-based selection the model used needs to be able to provide information on feature importance.

```
from sklearn.feature_selection import RFE
from sklearn.ensemble import RandomForestClassifier
selection = RFE(
        RandomForestClassifier(n_estimators=100, random_state=42)
selection.fit(X_train, y_train)
X_train_sel = selection.transform(X_train)
```

# Data Leakage

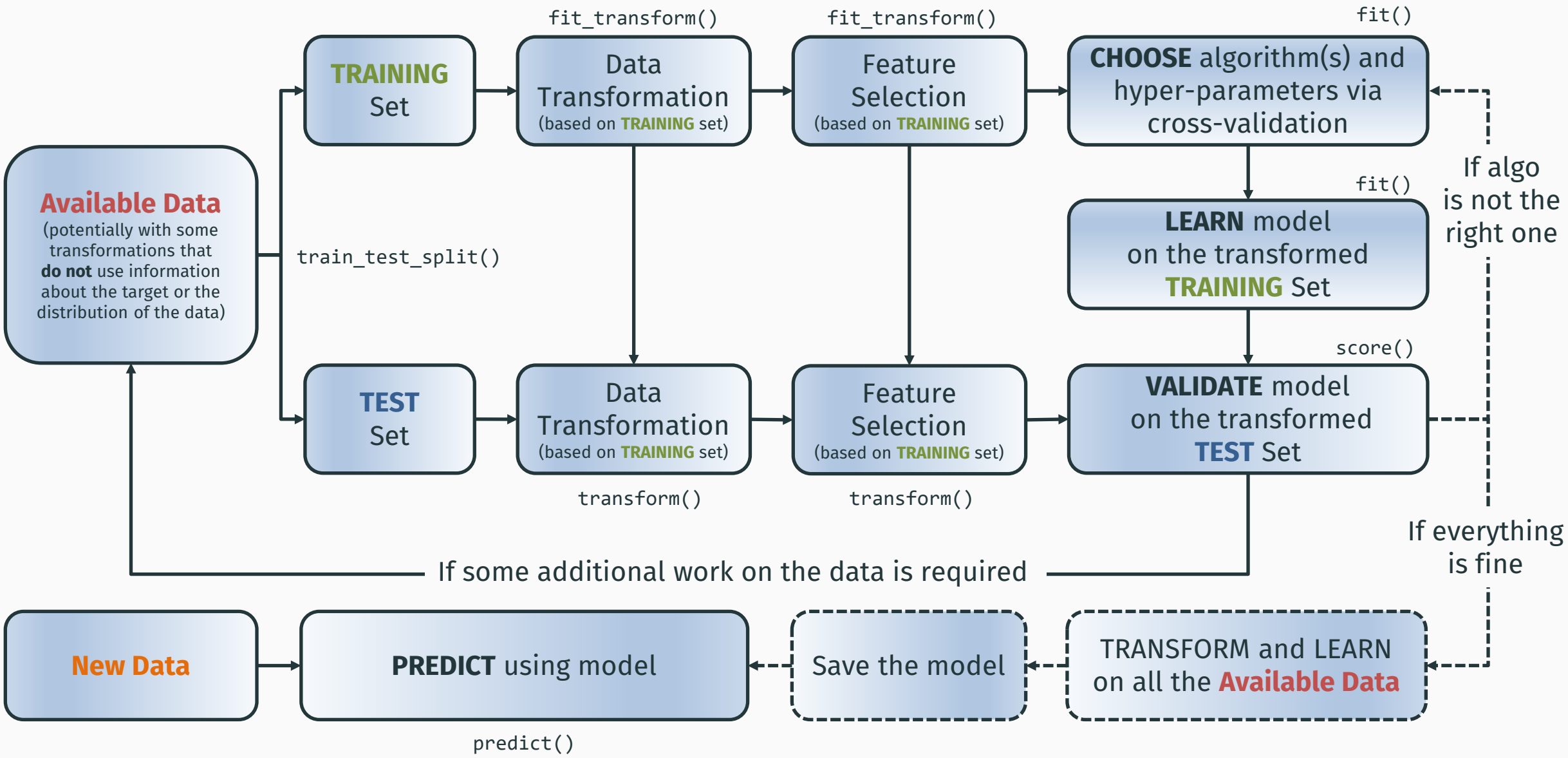
# Data Leakage

The objective of the split into training and test sets is to preserve some **unseen data** in order to **objectively evaluate** a model.

Transforming (or preprocessing ) the data or features before the split can **introduce implicit information** about the **test set**. This is called a **data leak** or data leakage. This invalidates the results obtained on the test set.

For example, if normalization is performed on all the available data before the split, the mean and standard deviation across the entire data set will have been used, thereby leaking to the training phase, and therefore the model, some information about the test set. Normalization should be performed on the training set. Then a separate normalization step is applied to the test set using the mean and standard deviation computed on the training set.

# ML workflow


Available Data
(potentially with some transformations that do not use information about the target or the distribution of the data)
train_test_split()
TRAINING Set
TEST Set
fit_transform()
Data Transformation
(based on TRAINING set)
fit_transform()
Feature Selection
(based on TRAINING set)
transform()
Data Transformation
(based on TRAINING set)
transform()
Feature Selection
(based on TRAINING set)
fit()
CHOOSE algorithm(s) and hyper-parameters via cross-validation
fit()
LEARN model on the transformed TRAINING Set
score()
VALIDATE model on the transformed TEST Set
If algo is not the right one
If everything is fine
If some additional work on the data is required
New Data
predict()
PREDICT using model
Save the model
TRANSFORM and LEARN on all the Available Data

# Validation

Especially Cross-Validation

# Validation

The objective of **validation** is to test the capacity of a model to make a predict on unseen data that have not been used to train the model in order to:

— Indentify overfitting problems or selection bias;

— Get an idea of whether the model will generalize on new data (in a production environment).

The simple approach of splitting the data into a training and test set is called **testset validation** or **holdout method**.

**Cross-validation** takes the above method and performs multiple rounds of training and testing on different subsets of the data in order to obtain an average error and achieve a more accurate estimate of prediction performance.

# Holdout - Simple Split into Training and Test Sets


To evaluate a single model:

- Build the model on the training set;
- Measure the error/score of the model on the test set;
- The error/score of the model directly corresponds to the error/score on the test set;
- Advantage: simple to code, fast evaluation ;
- Disadvantage: does not exploit all the data and the results depend on the split.

# K-fold Cross-Validation

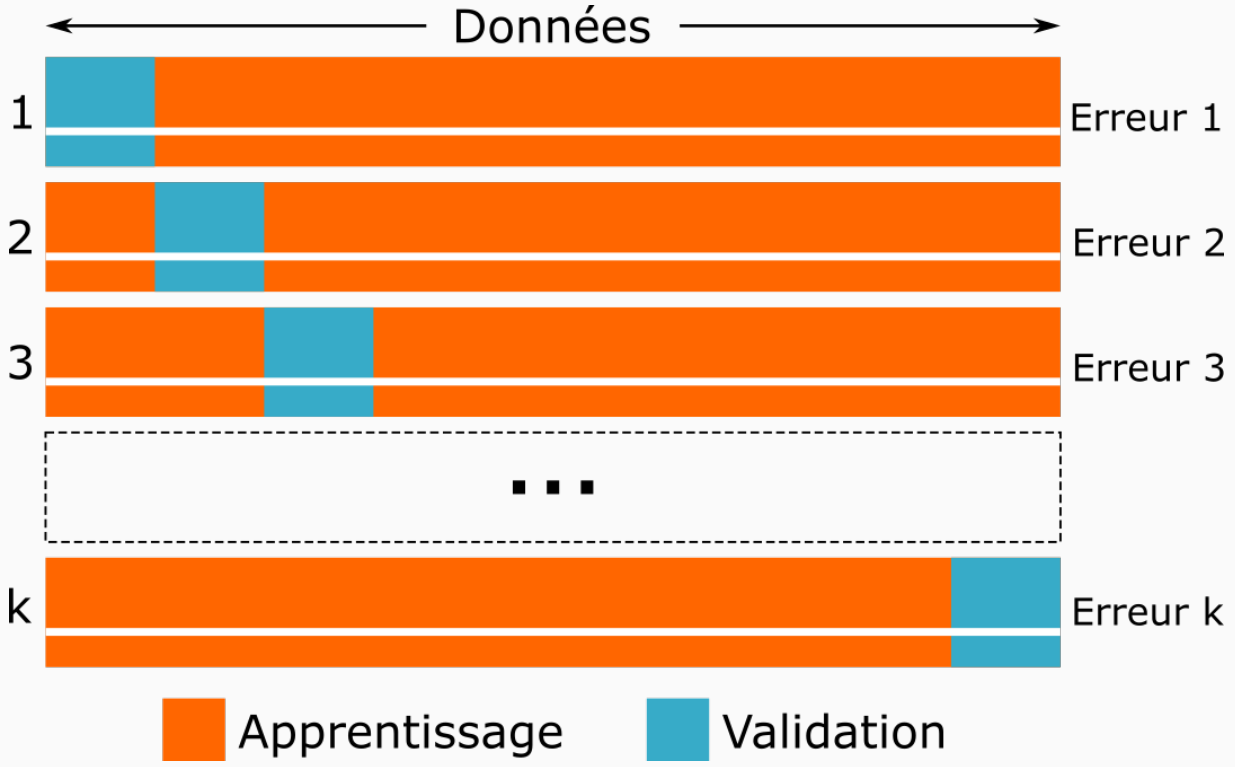
To evaluate a model building procedure:

- Split the data into $k$ subsets and repeat $k$ times
  - Build a model using ($k$–1) subsets as training set,
  - Measure the error/score on the remaining subset;
- The error/score of the procedure is the mean of the $k$ different computed errors/scores;

**avantage** : good estimation of the generalisation performance (error/score) ;

**inconvénient** : potentially costly (computation time).

A split that is equal to the number of observations (rows) is called **leave-one-out** cross-validation.



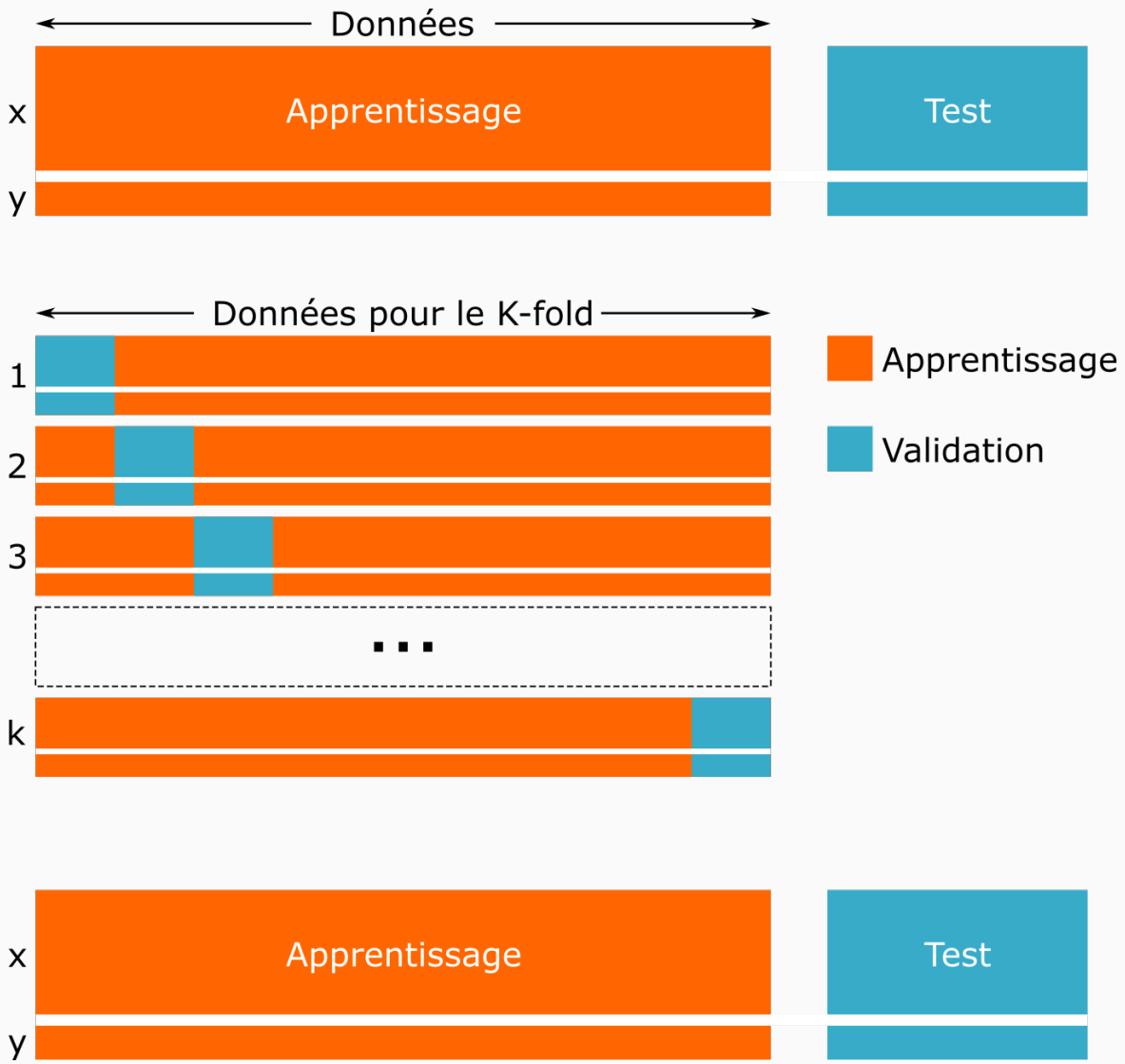
# Cross-Validation with 3 sets

1. Set aside a test set, separate from the training set;

2. Use cross-validation on the training set in order to select a model (its parameters);

3. *Retrain the selected model on all of the training set*;

4. Validate the final model on the test set.



# Hyper-parameter Tuning

# Hyper-Parameter Tuning

**Hyper-parameters** are the parameters of ML algorithms (e.g. the degree of a polynomial regression, the k in k-NN, the depth of a decision tree, the number of neurons per layer in a neural network, ...).

It is possible to automate the exploration of combinations of parameter values in order to find a good configuration. This is called **tuning**. It can be done in a couple of ways in scikit-learn:

- Exhaustive exploration of all the combinations (GridSearchCV);
- Random exploration of combinations (RandomizedSearchCV).
  https://scikit-learn.org/stable/modules/grid_search.html

There also exist other, more advanced methods, for example based on stochastic optimization techniques (but currently not in scikit-learn).

The concept of automating the design and tuning of ML algorithms is often called **AutoML**.

# Hyper-Parameter Tuning

```
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV
svc = SVC()

parameters = {'kernel':('linear', 'rbf', 'poly'),
              'C':[0.1, 1, 10]}

clf = GridSearchCV(svc, parameters,
                   refit=True) # refit is True by default

clf.fit(X_train, y_train)
clf.best_params_

clf.score(X_train, y_train)
clf.score(X_test, y_test)
```

Specify all the parameter values we want to test.

When fit() is called cross-validation is performed on the data passed as parameters.
If the `refit` parameter is set (it is set by default), the best parameters will be used to fit a final model on all of the data.
This best model can then be evaluated on the unseen test data.

# Pipelines

# Pipelines

A **pipeline** is a sequence of operations on data. In ML and in scikit-learn in particular, it is a **sequence of transformations that ends with an estimator** (classifier or regressor).

The purpose of a pipeline is to group a **series of ML steps** that can be **cross-validated together**, usually while also **tuning a number of parameters**. While it is also possible to do all the steps manually, a pipeline makes the process much easier and efficient.

A pipeline can include as many transformations on data and feature selection steps as required (objects that have a `fit_transform()` method), but should always end with an estimator (having a `fit()` method).

# Pipelines

```
from sklearn.pipeline import Pipeline
pipe = Pipeline([("scaler", MinMaxScaler()),
                 ("svm", SVC(C=100))])
```

A string of your choice

An estimator object

Last object needs to be a classifier or regressor

```
pipe.fit(X_train, y_train)
```

Calls fit_transform() of MinMaxScaler then calls fit() of SVC on the scaled training data

```
pipe.score(X_test, y_test)
```

Calls transform() of MinMaxScaler to apply the same transformation computed on the training data to the test data then calls score() of SVC on the scaled test data

Alternative:

```
from sklearn.pipeline import make_pipeline
pipe = make_pipeline(MinMaxScaler(), SVC(C=100))
```

# Pipelines and Hyper-Parameter Tuning

```
from sklearn.pipeline import make_pipeline
pipe = make_pipeline(StandardScaler(), LogisticRegression())

param_grid = {'logisticregression__C': [0.01, 0.1, 1, 10],
              'logisticregression__solver': ['lbfgs', 'saga']}
```

Name of estimator (lowercase)

Parameter of corresponding estimator

```
grid = GridSearchCV(pipe, param_grid)

grid.fit(X_train, y_train)
```

Also works for finding parameters of pre-processing steps!

```
grid.best_estimator_
grid.best_estimator_.named_steps["logisticregression"].coef_
```

# Pipelines and Searching for the Best Model

```
from sklearn.pipeline import Pipeline
pipe = Pipeline([("preprocessing", StandardScaler()),
                 ("classifier", SVC())])

param_grid = [
        {'classifier': [SVC()],
         'preprocessing': [StandardScaler(), None],
         'classifier__gamma': [0.001, 0.01, 0.1, 1, 10, 100],
         'classifier__C': [0.001, 0.01, 1, 10, 100]},
        {'classifier': [RandomForestClassifier(n_estimators=100)],
         'preprocessing': [None],
         'classifier__max_features': [1, 2, 3]}]

grid = GridSearchCV(pipe, param_grid)

grid.fit(X_train, y_train)
grid.best_params_
grid.best_score_ # best cross-validation score
grid.score(X_test, y_test)
```

SVC usually requires normalization
as preprocessing
but Random Forest does not

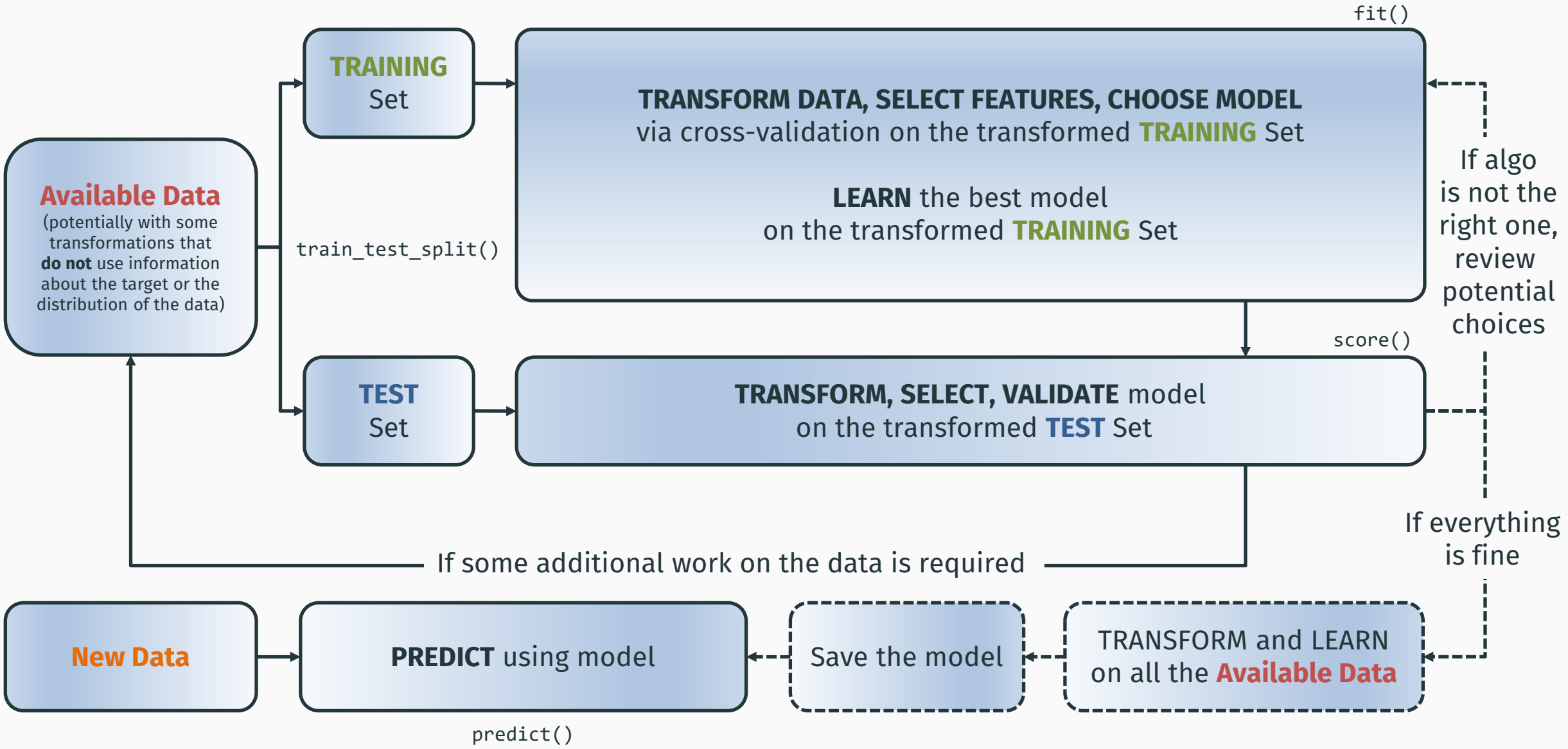
# Saving a Model

Preprocessing and training can take a lot of time. So we don't want to do it each time we use a model.
Therefore we only train a model once, save it and then reuse it as often as we want.

```
import pickle

# write model to file
with open('model.pickle', 'wb') as f:
        pickle.dump(clf, f)

# read model from file
with open('model.pickle', 'rb') as f:
        clf2 = pickle.load(f)
```

```
import joblib

# write model to file
joblib.dump(clf, 'model.joblib')


# read model from file
clf2 = joblib.load('model.joblib')
```

# ML workflow


fit()
TRAINING Set
TRANSFORM DATA, SELECT FEATURES, CHOOSE MODEL via cross-validation on the transformed TRAINING Set
LEARN the best model on the transformed TRAINING Set
If algo is not the right one, review potential choices
Available Data (potentially with some transformations that do not use information about the target or the distribution of the data)
train_test_split()
score()
TEST Set
TRANSFORM, SELECT, VALIDATE model on the transformed TEST Set
If everything is fine
If some additional work on the data is required
New Data
PREDICT using model
Save the model
TRANSFORM and LEARN on all the Available Data
predict()