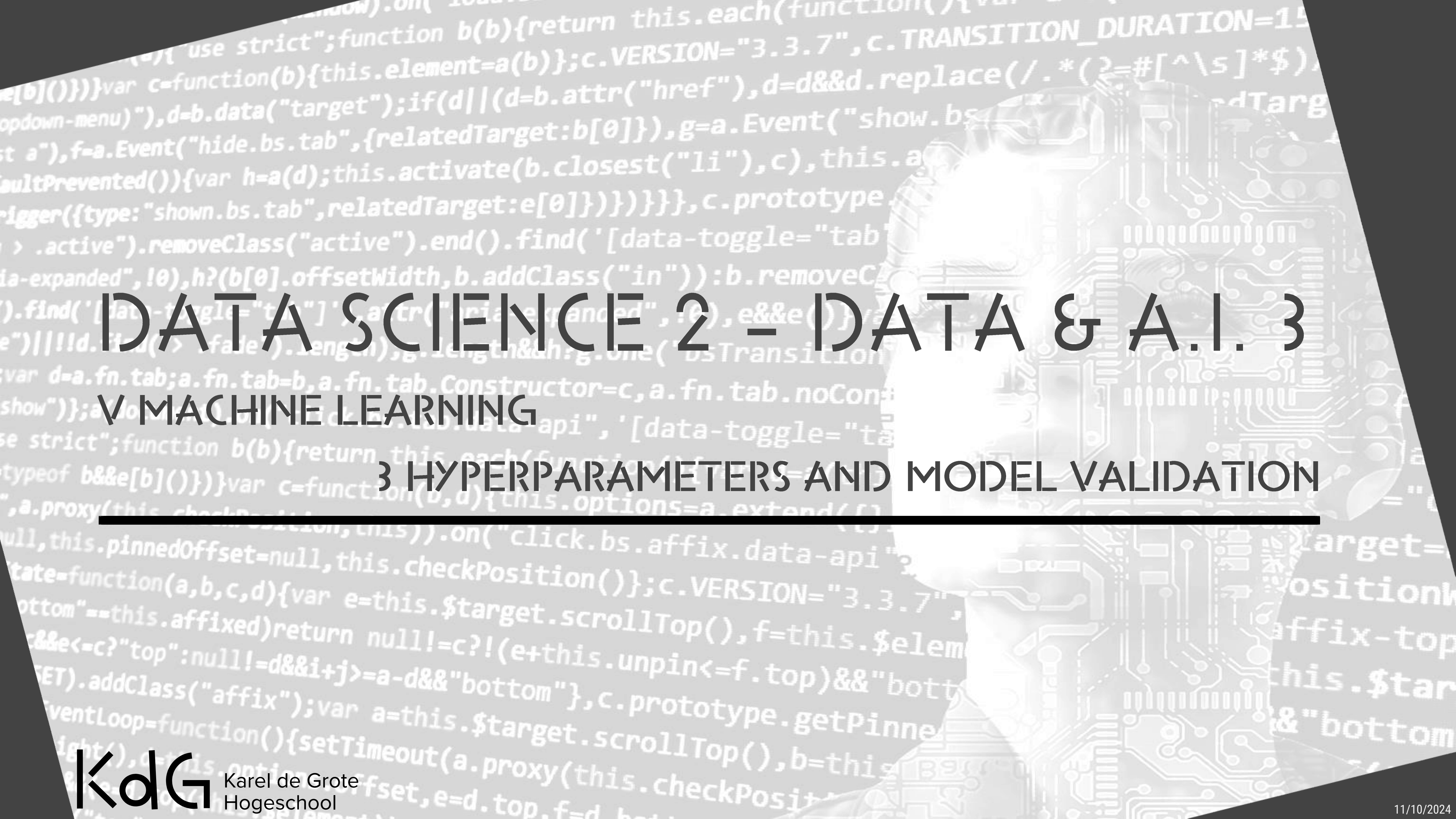


DATA SCIENCE 2 – DATA & A.I. 3

V MACHINE LEARNING

3 HYPERPARAMETERS AND MODEL VALIDATION

Karel de Grote Hogeschool

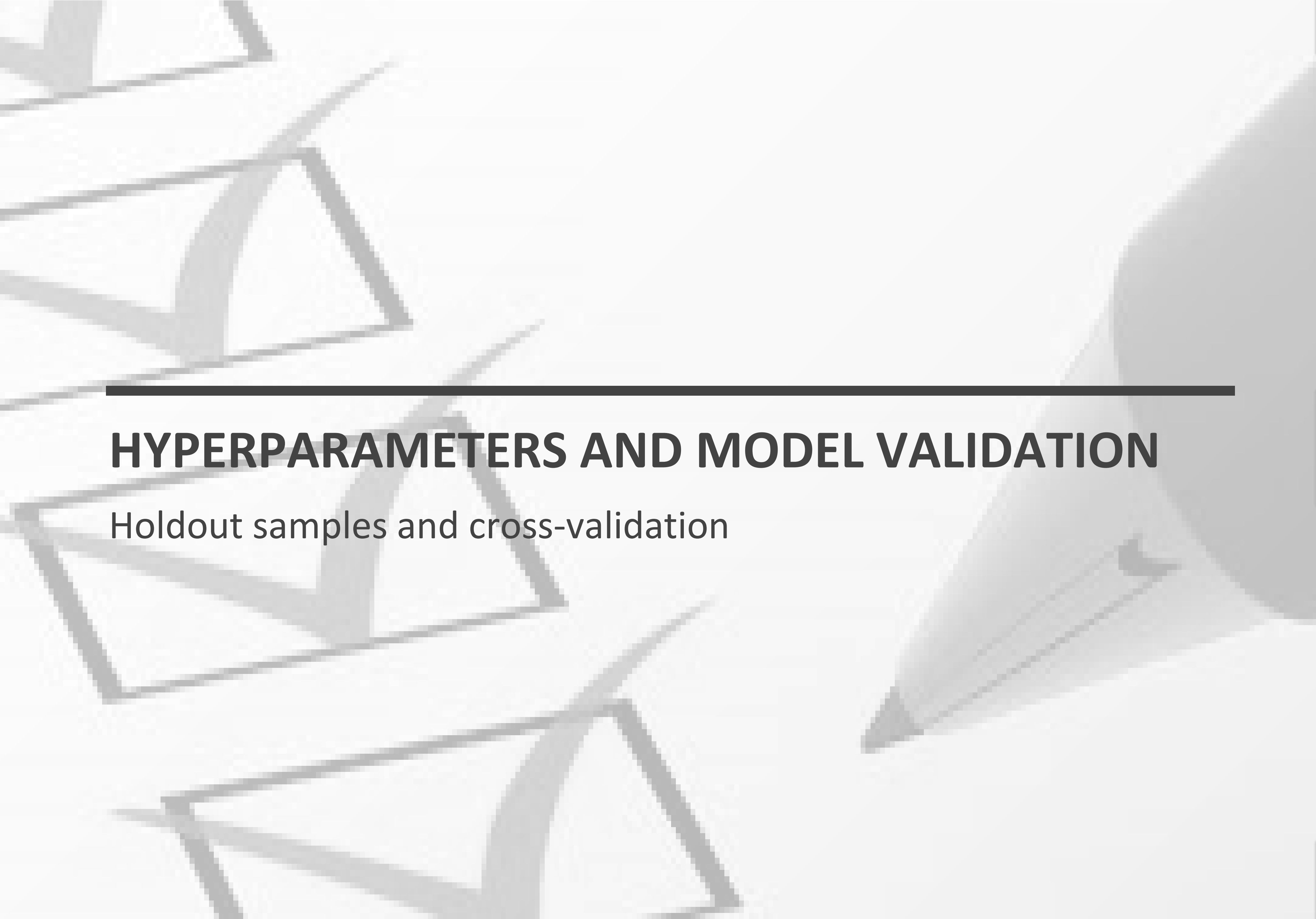
DATA SCIENCE 2
DATA & A.I. 3

V

MACHINE LEARNING

scikit-learn

MACHINE LEARNING



HYPERPARAMETERS AND MODEL VALIDATION

Holdout samples and cross-validation

WHAT KIND OF PROBLEMS CAN YOU SOLVE WITH MACHINE LEARNING

SUPERVISED

- Value estimation / regression: predict a continuous variable
  e.g. sales prediction; car use
- Classification: predict a categorical variable
  e.g. churn prediction; diagnosis

UNSUPERVISED

- Segmentation / clustering: split or group cases/observations
  e.g. customer segmentation; document topic search
- Co-occurence / association rule discovery: events happening together
  e.g. market basket analysis ; recommendation system

MODEL VALIDATION BASICS (SUPERVISED LEARNING)

COMPARE PREDICTED VALUES WITH REAL VALUES (LABELS/GROUND TRUTH/GOLDEN STANDARD)

For supervised learning, model validation is easy. As you start from a labeled dataset, you have a set of examples for which you know the real outcome (labels). This is also called the ground truth or gold standard. So you can compare the predicted values with the real values (labels):

- Derive (train/fit) a model based on labeled data
- Use the derived model to predict the target variable in the labeled data based on the independent features/predictors in the labeled data
- Compare the predicted target feature values from the previous step with the real target feature values (labels) in the labeled dataset

HOW TO COMPARE REAL AND PREDICTED VALUES FOR REGRESSION METHODS (PREDICTION OF NUMERICAL FEATURES)

Start calculating the difference between the real and predicted values. Based on those differences, validation metrics can be calculated:

- MAE : mean absolute error
- MAPE : mean absolute percentage error
- RMSE : root mean squared error
- …

MODEL VALIDATION BASICS (SUPERVISED LEARNING)

HOW TO COMPARE REAL AND PREDICTED VALUES FOR CLASSIFICATION METHODS (PREDICTION OF CATEGORICAL FEATURES)

Start compiling the confusion matrix based on the number of correctly and wrongly predicted cases. Based on this confusion matrix, validation metrics can be calculated:

- Accuracy
- Precison
- Recall
- F-value
- AUC
- ROC
- …

SEE DATA SCIENCE 1 / DATA & A.I. 2 AND SCIKIT-LEAN DOCUMENTATION

- sklearn.metrics : https://scikit-learn.org/stable/api/sklearn.metrics.html

DATA ANALYTICS PIPELINE (WITH VALIDATION)

DATA PREPARATION

- Load (labeled) source data
- Compile feature matrix
- Compile target array (for supervised methods)

MODEL SELECTION AND HYPERPARAMETER SELECTION (MODEL SPECIFIC)

- Decide on the method to use (linear regression, decision tree, K-means clustering, …)
- Decide on the hyperparameter to use (degree of polynomial, tree depth, number of clusters, …)

  Hyperparameters are parameters that the algorithm uses to derive a model

  Hyperparameters depend on the method (every methods has it's on kind of hyperparameters)

DERIVE MODEL FROM LABELED DATA (TRAIN MODEL/FIT MODEL)

- Apply the method/algorithm on the labeled data to derive the model

DISPLAY MODEL (MODEL SPECIFIC)

- Display the resulting model

  The resulting model depends on the method used (equation of regression line with intercept and slope, decision tree with nodes and split conditions, groups of observations with centroid, …)

DATA ANALYTICS PIPELINE (WITH VALIDATION)

VALIDATE MODEL USING LABELED DATA (SUPERVISED LEARNING)

- Predict target feature for the labeled data
- Calculate the difference between predicted and real values for the labeled data / calculate confusion matrix (classification)
- Calculate validation metrics for the labeled data
  - Regression : MAE, MAPE, RMSE, …
  - Classification : accuracy, precision, recall, f-score, AUC, ROC

APPLY MODEL ON NEW DATA

- Apply the model on new data
  - In case of supervised machine learning methods, this will predict the target feature for new data
  - In case of unsupervised machine learning methods, this will restructure the feature data (clusters, association rules, new feature matrix with reduced dimensions)

VALIDATION METRICS WITH SCIKIT-LEARN (REGRESSION)

```
# DATA PREPARATION
import pandas as pd
pd.options.display.max_rows = None
import seaborn as sns
iris = sns.load_dataset('iris')
X = iris[['sepal_width', 'sepal_length', 'petal_width']] # Predictors
y = iris['petal_length'] # Target feature to predict

# MODEL SELECTION AND HYPERPARAMETER SELECTION (MODEL SPECIFIC)
from sklearn.linear_model import LinearRegression
model = LinearRegression(fit_intercept=True)
print(model)
# List all selected hyperparameters
print(model.get_params(deep=True))

# DERIVE MODEL FROM LABELED DATA (TRAIN MODEL/FIT MODEL)
model.fit(X,y)
```

VALIDATION METRICS WITH SCIKIT-LEARN (REGRESSION)

```
# DISPLAY MODEL (MODEL SPECIFIC)
print(model.intercept_, model.coef_)

# VALIDATE MODEL USING LABELED DATA
from sklearn.metrics import mean_absolute_error, mean_absolute_percentage_error,
root_mean_squared_error

# Predict target feature for the labeled data
y_pred = pd.Series(model.predict(X), name='y_pred')

# Calculate the difference between predicted and real values for the labeled data
err = pd.Series(y_pred-y, name='err')
display(pd.concat([y, y_pred, err], axis=1))

# Metrics
mae = mean_absolute_error(y_true=y, y_pred=y_pred)
mape = mean_absolute_percentage_error(y_true=y, y_pred=y_pred)
rmse = root_mean_squared_error(y_true=y, y_pred=y_pred)
print(f'MAE : {mae:.3f} - MAPE : {mape:.3f} - RMSE : {rmse:.3f}')
```

VALIDATION METRICS WITH SCIKIT-LEARN (REGRESSION)

```
# APPLY MODEL ON NEW DATA
X_pred = …. (new feature data to predict the target feature for)
y_pred = model.predict(X_pred)
```

VALIDATION METRICS WITH SCIKIT-LEARN (CLASSIFICATION)

```
# DATA PREPARATION
import pandas as pd
pd.options.display.max_rows = None
import seaborn as sns
iris = sns.load_dataset('iris')
y = iris['species'] # Target feature to predict
X = iris.copy().drop('species', axis=1) # Predictors

# MODEL SELECTION AND HYPERPARAMETER SELECTION (MODEL SPECIFIC)
from sklearn.tree import DecisionTreeClassifier
model = DecisionTreeClassifier(max_depth=1)
print(model)
# List all selected hyperparameters
print(model.get_params(deep=True))

# DERIVE MODEL FROM LABELED DATA (TRAIN MODEL/FIT MODEL)
model.fit(X,y)
```

VALIDATION METRICS WITH SCIKIT-LEARN (CLASSIFICATION)

```
# DISPLAY MODEL (MODEL SPECIFIC)
from sklearn.tree import plot_tree
plot_tree(model)

# VALIDATE MODEL USING LABELED DATA
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_score,
f1_score, precision_recall_fscore_support, classification_report
import matplotlib.pyplot as plt

# Predict target feature for the labeled data
y_pred = pd.Seeries(model.predict(X), name='y_pred')

# Calculate the difference between predicted and real values for the labeled data
err = pd.Series(y_tst_pred.reset_index(drop=True)!=y_tst.reset_index(drop=True),
name='err').astype(int)
display(pd.concat([y_tst.reset_index(drop=True), y_tst_pred.reset_index(drop=True), err],
axis=1))
```

VALIDATION METRICS WITH SCIKIT-LEARN (CLASSIFICATION)

```
# Confusion matrix
# Display as text (console output)
class_labels = sorted(list(pd.concat([y_tst,y_tst_pred], axis=0).unique()))
# Alternative : model.classes_
cm = confusion_matrix(y_true = y_tst, y_pred = y_tst_pred)
print('Predicted label')
print(class_labels)
print(cm)
# Display as heatmap (nicer output in Jupyter)
disp = sns.heatmap(cm, square=True, annot=True, cbar=True, cmap='Greys',
xticklabels=class_labels, yticklabels=class_labels)
plt.xlabel('Predicted label')
plt.ylabel('True label')
disp.xaxis.tick_top()                  # Put x-axis tickers on top
disp.xaxis.set_label_position('top') # Put x-axis label on top
```

VALIDATION METRICS WITH SCIKIT-LEARN (CLASSIFICATION)

```
# Metrics
acc = accuracy_score(y_true=y, y_pred=y_pred)
prec = precision_score(y_true=y, y_pred=y_pred, average='weighted')
rec = recall_score(y_true=y, y_pred=y_pred, average='weighted')
f1 = f1_score(y_true=y, y_pred=y_pred, average='weighted')
# Mind this is a multiclass classification problem, so precision, recall and F1
# are calculated by class and averaged.
print(f'ACC : {acc:.3f} - PREC : {prec:.3f} - REC : {rec:.3f} - F1 : {f1:.3f}')

# The easiest way to get results by class is to use precision_recall_fscore_support
class_labels = sorted(list(pd.concat([y_tst,y_tst_pred], axis=0).unique()))
# Display precision/recall/fscore/support table as text (consule output)
print(class_labels)
display(precision_recall_fscore_support(y_true=y_tst, y_pred=y_tst_pred))
# Display precision/recall/fscore/support as pandas dataframe (nicer outputin Jupyter)
display(pd.DataFrame(precision_recall_fscore_support(y_true=y_tst, y_pred=y_tst_pred),
index=['prec','rec','fscore','sup'], columns=class_labels))

# Or use classification_report
print(classification_report(y_true=y_tst, y_pred=y_tst_pred, target_names=class_labels))
```

VALIDATION METRICS WITH SCIKIT-LEARN (CLASSIFICATION)

```
# APPLY MODEL ON NEW DATA
X_pred = …. (new feature data to predict the target feature for)
y_pred = model.predict(X_pred)
```

MODEL VALIDATION THE WRONG WAY

DO NOT TRAIN AND VALIDATE THE MODEL ON THE SAME DATA

Training and validating a model on the same data is not a good idea. It is like evaluating students based on an examination with questions that the students got beforehand, including the answers. If a student performs well on that kind of examination, you will never know whether that student really understands the questions and answers, or just learned the answers by heart.

The same problem arises with machine learning; machine learning builds a model by deriving patterns from example data. If you use the same dataset for training and validation, you will never know whether the model really captured the underlying patterns, or just learned the target value by heart for every example (e.g. a decision tree with rules that are too detailed, classifying every single example with a specific rule, instead of a limited set of general rules classifying a large set of similar cases).

OVERFITTING VERSUS GENERALISATION

Mind that the purpose of supervised machine learning is not to predict the labeled training data (you know already the outcome for that data), but to predict new data, data never seen before (and hence not included during training). That means the model should be able to generalise, i.e. be able to capture the overall patterns to be able to predict new data.

So, for validation, we are not that much interested in how well the model can predict the data used for training. We are interested in how well the model can predict new data. A model that scores well on training data but scores bad on new data is an overfitted model. That means it tries too hard to predict the training data, but is not able to grasp the general patterns in the data. It is not able he generalise, so it predicts the training data very well, but is not able to predict new data. Such a model is useless in practice.

What we need is a model that score well on new data.

MODEL VALIDATION THE RIGHT WAY

DO NOT USE THE LABELED TRAINING DATA FOR VALIDATION

Again, we are not interested in how well the model scores on the labeled training data, but on new data. That means that the labeled training data **MUST NOT** be used for validation. Hence, after training, a new labeled dataset needs to be constructed for the validation.

Constructing a new labeled dataset after training is not a practical approach; constructing a labeled dataset might require expert knowledge in the domain at hand. As a data scientist, you cannot have expert knowledge for every potential domain. So labeling a dataset might require external experts. It is not always possible to call those in after every training job.

The solutions is to find a way to use the labeled dataset for both training and validation.

USE A HOLDOUT SAMPLE FOR VALIDATION

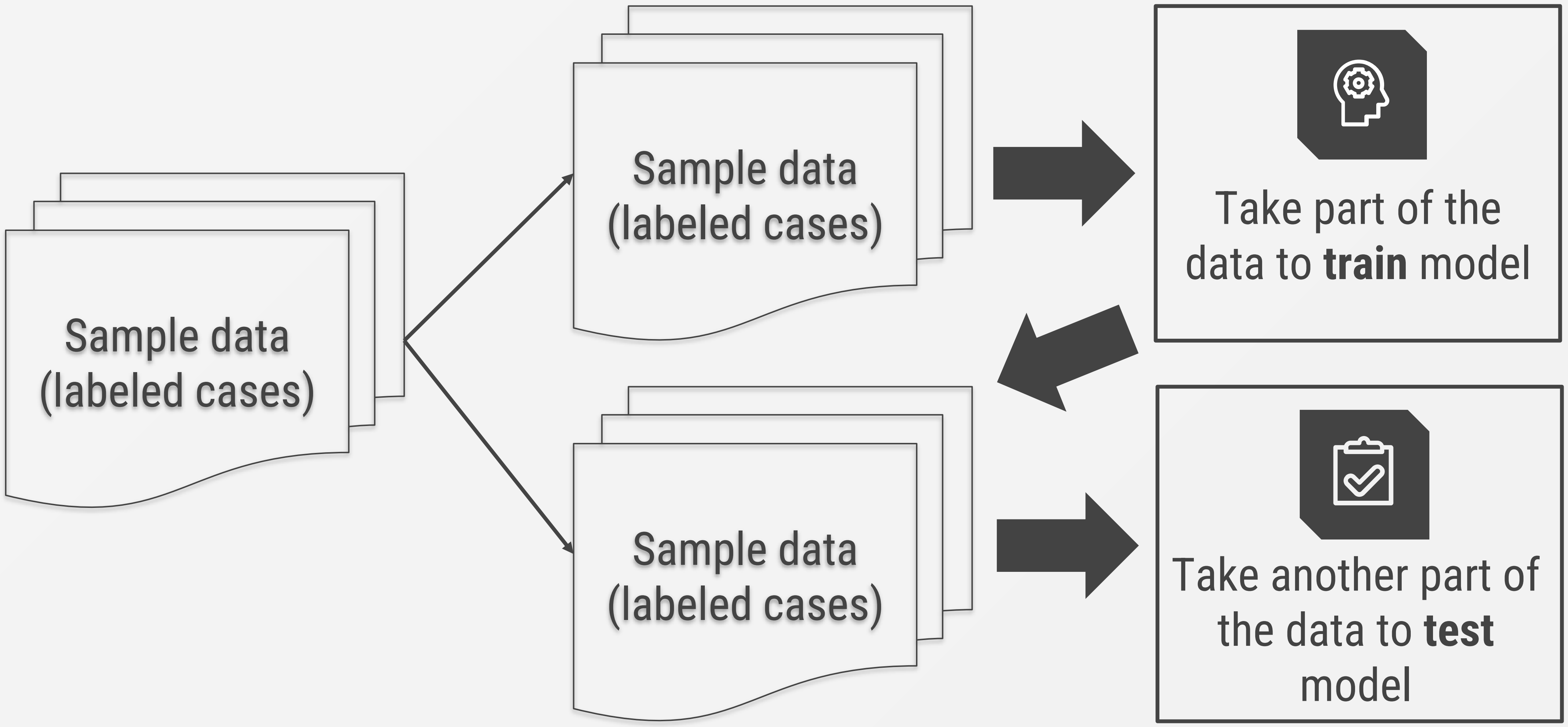
The answer to that problem is to use a holdout sample. That means that some labeled examples are set aside before training and are not used during training. After training, the holdout sample, i.e. the labeled examples that were set aside and not used in training, can now be used for an independent validation as they are new to the model.

SPLIT LABELED DATA INTO TRAINING SET AND TEST SET

In practice, it is as simple as to split the labeled data set in two subsets, one subset to be used for training, and one subset to be used for testing (validation).

The split does not need to be 50/50. The more training examples the better, but of course there need to be enough test examples to get a representative sample for testing. Common ratio's are 70%/80% for training and 30%/20% for testing.

TRAIN - TEST SPLIT



Sample data (labeled cases)
Sample data (labeled cases)
Take part of the data to **train** model
Sample data (labeled cases)
Take another part of the data to **test** model

DATA ANALYTICS PIPELINE (WITH TRAIN - TEST SPLIT)

DATA PREPARATION

- Load (labeled) source data
- Compile feature matrix
- Compile target array (for supervised methods)

SPLIT LABELED DATA INTO TRAIN - TEST SAMPLE (RANDOMLY!)

- Training data sample with 50%/70%/75%/80% of labeled example cases to be used for training the model
- Test data sample with 50%/30%/25%/20% of labeled example cases to be used for validating the model

MODEL SELECTION AND HYPERPARAMETER SELECTION (MODEL SPECIFIC)

- Decide on the method to use (linear regression, decision tree, K-means clustering, …)
- Decide on the hyperparameter to use (degree of polynomial, tree depth, number of clusters, …)
  Hyperparameters are parameters that the algorithm uses to derive a model
  Hyperparameters depend on the method (every methods has it’s on kind of hyperparameters)

DERIVE MODEL FROM <u>TRAINING</u> DATA (TRAIN MODEL/FIT MODEL)

- Apply the method/algorithm on the training data to derive the model

DATA ANALYTICS PIPELINE
(WITH TRAIN - TEST SPLIT)

DISPLAY MODEL (MODEL SPECIFIC)

- Display the resulting model

  The resulting model depends on the method used (equation of regression line with intercept and slope, decision tree with nodes and split conditions, groups of observations with centroid, …)

VALIDATE MODEL USING TEST DATA (SUPERVISED LEARNING)

- Predict the target feature for the test data
- Calculate the difference between predicted and real values for the test data / calculate confusion matrix (classification)
- Calculate validation metrics for the test data

  Regression : MAE, MAPE, RMSE, …

  Classification : accuracy, precision, recall, f-score, AUC, ROC

APPLY MODEL ON NEW DATA

- Apply the model on new data

  In case of supervised machine learning methods, this will predict the target feature for new data

  In case of unsupervised machine learning methods, this will restructure the feature data (clusters, association rules, new feature matrix with reduced dimensions)

SPLIT TRAIN - TEST PIPELINE WITH SCIKIT-LEARN (CLASSIFICATION)

```
# DATA PREPARATION
import pandas as pd
pd.options.display.max_rows = None
import seaborn as sns
iris = sns.load_dataset('iris')
y = iris['species'] # Target feature to predict
X = iris.copy().drop('species', axis=1) # Predictors

# SPLIT LABELED DATA INTO TRAIN - TEST SAMPLE
from sklearn.model_selection import train_test_split
# Split the data randomly into 80% training set and 20% test set
X_tr, X_tst, y_tr, y_tst = train_test_split(X, y, random_state=0, train_size=0.8)
# (use random_state to be sure that every time the same random sample is drawn)

# MODEL SELECTION AND HYPERPARAMETER SELECTION (MODEL SPECIFIC)
from sklearn.tree import DecisionTreeClassifier
model = DecisionTreeClassifier(max_depth=1)
```

SPLIT TRAIN - TEST PIPELINE WITH SCIKIT-LEARN (CLASSIFICATION)

```
# DERIVE MODEL FROM TRAINING DATA (TRAIN MODEL/FIT MODEL)
model.fit(X_tr,y_tr)

# DISPLAY MODEL (MODEL SPECIFIC)
from sklearn.tree import plot_tree
plot_tree(model)

# VALIDATE MODEL USING TEST DATA
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_score,
f1_score, precision_recall_fscore_support, classification_report
import matplotlib.pyplot as plt

# Predict target feature for the test data
y_tst_pred = pd.Series(model.predict(X_tst), name= 'y_tst_pred')
```

SPLIT TRAIN - TEST PIPELINE WITH SCIKIT-LEARN (CLASSIFICATION)

```
# Calculate the difference between predicted and real values for the test data
err = pd.Series(y_tst_pred.reset_index(drop=True)!=y_tst.reset_index(drop=True),
name='err').astype(int)
display(pd.concat([y_tst.reset_index(drop=True), y_tst_pred.reset_index(drop=True), err], axis=1))

# Confusion matrix
# Display as text (console output)
class_labels = sorted(list(pd.concat([y_tst,y_tst_pred], axis=0).unique()))
# Alternative : model.classes_
cm = confusion_matrix(y_true = y_tst, y_pred = y_tst_pred)
print('Predicted label')
print(class_labels)
print(cm)
# Display as heatmap (nicer output in Jupyter)
disp = sns.heatmap(cm, square=True, annot=True, cbar=True, cmap='Greys', xticklabels=class_labels,
yticklabels=class_labels)
plt.xlabel('Predicted label')
plt.ylabel('True label')
disp.xaxis.tick_top()                   # Put x-axis tickers on top
disp.xaxis.set_label_position('top') # Put x-axis label on top
```

SPLIT TRAIN - TEST PIPELINE WITH SCIKIT-LEARN (CLASSIFICATION)

```
# Metrics
acc = accuracy_score(y_true=y_tst, y_pred=y_tst_pred)
prec = precision_score(y_true=y_tst, y_pred=y_tst_pred, average='weighted')
rec = recall_score(y_true=y_tst, y_pred=y_tst_pred, average='weighted')
f1 = f1_score(y_true=y_tst, y_pred=y_tst_pred, average='weighted')
# Mind this is a multiclass classification problem, so precision, recall and F1
# are calculated by class and averaged.
print(f'ACC : {acc:.3f} - PREC : {prec:.3f} - REC : {rec:.3f} - F1 : {f1:.3f}')

# The easiest way to get results by class is to use precision_recall_fscore_support
class_labels = sorted(list(pd.concat([y_tst,y_tst_pred], axis=0).unique()))
# Display precision/recall/fscore/support table as text (consule output)
print(class_labels)
display(precision_recall_fscore_support(y_true=y_tst, y_pred=y_tst_pred))
# Display precision/recall/fscore/support as pandas dataframe (nicer outputin Jupyter)
display(pd.DataFrame(precision_recall_fscore_support(y_true=y_tst, y_pred=y_tst_pred),
index=['prec','rec','fscore','sup'], columns=class_labels))

# Or use classification_report
print(classification_report(y_true=y_tst, y_pred=y_tst_pred, target_names=class_labels))
```

SPLIT TRAIN - TEST PIPELINE WITH SCIKIT-LEARN (CLASSIFICATION)

```
# APPLY MODEL ON NEW DATA
X_pred = …. (new feature data to predict the target feature for)
y_pred = model.predict(X_pred)
```

MODEL VALIDATION BASICS (UNSUPERVISED LEARNING)

METHOD SPECIFIC

For unsupervised learning (clustering, dimensionality reduction, association rules), model validation is not easy. There are no labels to compare with, there is no ground truth or golden standard.

The way to validate this is very method specific.

Some methods allow to derive some metrics that can be used in validation:

- Clustering : Inter cluster distance (should by high), intra cluster distance (should be low), silhouette (should be high)
- Association rules : lift, confidence

MIND VALIDATION METRICS AS THESE ARE USED TO DERIVE THE MODEL

Do not blindly rely on validation metrics of unsupervised methods, as these metrics are used under the hood to derive the model:

- Clustering : inter cluster distance and intra cluster distance is used to form the clusters
- Association rules : lift and confidence are used to select relevant rules

Using these metrics for validation of unsupervised models is a bit like using training data to validate supervised models. If unsupervised models score low on those metrics, you can be sure the model is bad, but the opposite is not true, if unsupervised models score high on those metrics, you still can not know for sure the model is good.

MODEL VALIDATION BASICS (UNSUPERVISED LEARNING)

THE PROOF OF THE PUDDING IS IN THE EATING

The only way to check whether an unsupervised model is good is to implement it and check if it does the job that has to be done, or solve the problem it has to solve. That kind of validation can be very hard and time consuming.

CROSS-VALIDATION

ALTERNATIVE FOR SPLITTING LABELED DATA INTO TRAINING SET AND TEST SET

The problem with splitting labeled example data into a training set and test set is that less labeled examples are available for training (as they are set aside for validation). In general, the more labeled examples for training, the better the model. But labeling a dataset can be very costly, hence to ‘loose’ valuable labelled examples to testing might not be preferable, especially for smaller labeled dataset.

The solution might be cross validation.

CROSS-VALIDATION

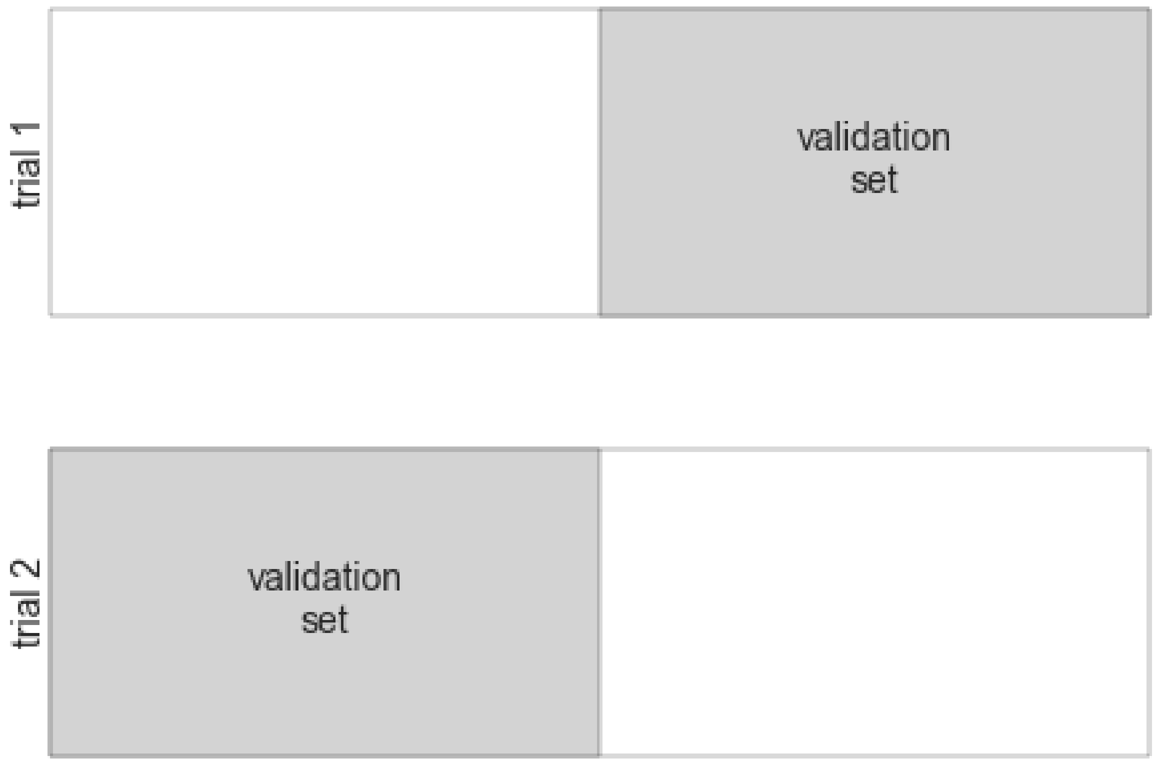
Cross-validation validation means that the labeled examples are split into multiple partitions and that multiple models are trained on different partitions. Every time, one partition of the labelled data is hold out for testing, and all other partitions are used for training (and the hold out partition changes every time). So, for N partitions, N different models are trained, with one of the N partitions as hold out sample for testing, and the N-1 other partitions for training. As a result, we have N models, all trained on slightly different data, covering all the examples in the labeled dataset (all examples are used N-1 times for training, and once for testing), resulting in N validation metrics. Those validation metrics can be combined, e.g. by calculating the mean over the N validation metric values.

The advantage of this method is that all data is used both for training and testing, and that the combined validation metrics over all parts of the data give a better measure for global performance.

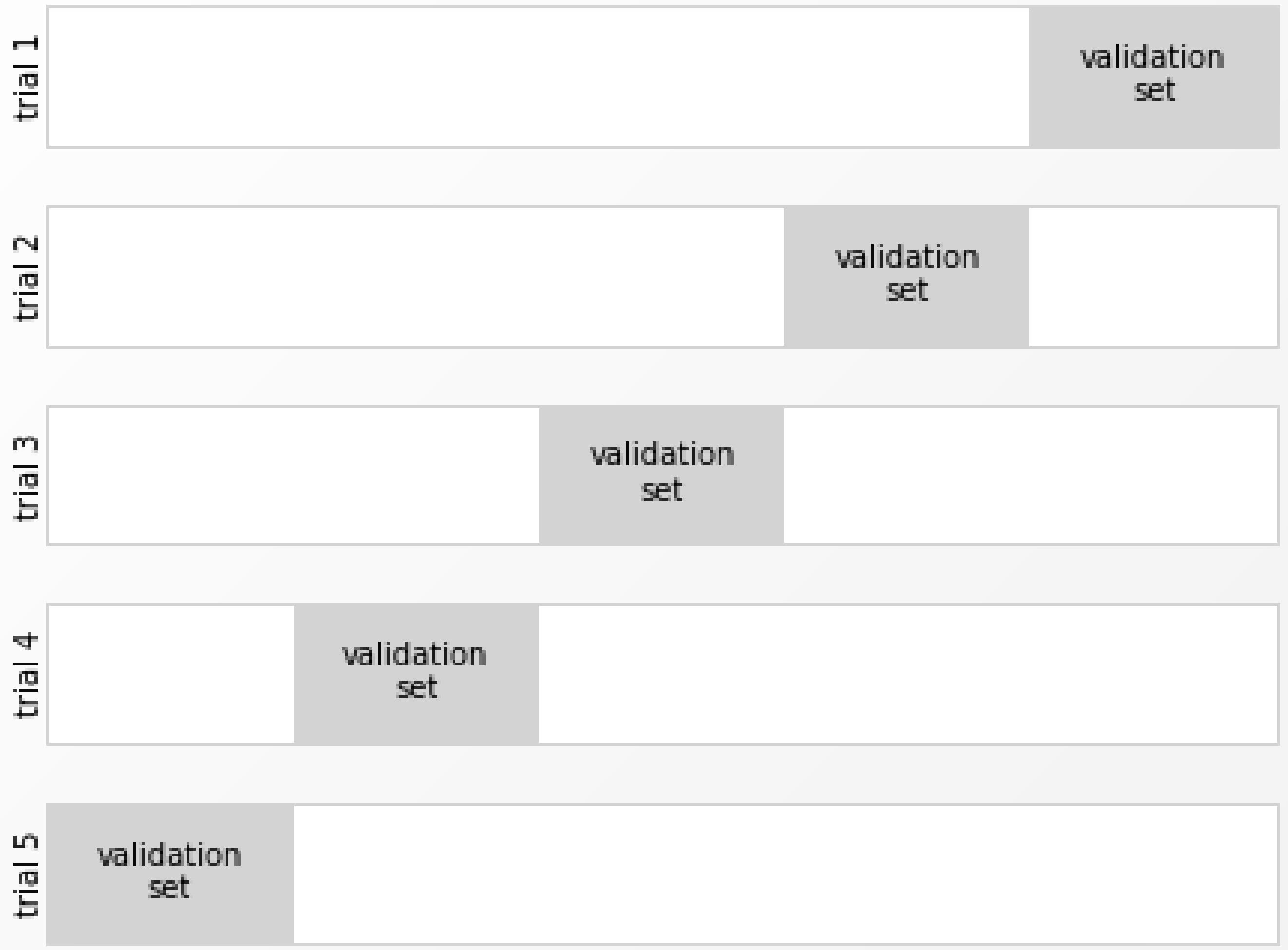
The disadvantage is that we end up with multiple models. That means there is no single model to use for predicting new data. One can select the best model, but that choice might be biassed, because the difference in performance is due to difference in training and validation data, so is based on coincidence. There is no guarantee that that model will also perform best on new data (we come back to that later).

Cross-validation is commonly used in model selection and hyperparameter tuning (see later).

CROSS-VALIDATION
(2-FOLD CROSS VALIDATION)



CROSS-VALIDATION (K-FOLD CROSS VALIDATION)



SIMPLE MODELS VERSUS COMPLEX MODELS UNDERFITTING VERSUS OVERFITTING

MULTIPLE MODELING TECHNIQUES

When using supervised machine learning, multiple techniques can be used to derive a predictive model

- regression: linear regression, polynomial regression, artificial neural network, …;
- classification: decision tree, support vector machine, artificial neural network, ….

The challenge is to select the technique that is most appropriate for your problem or data. That does not necessarily mean the most advanced or complex method. The best choice has to do with underfitting and overfitting.

SIMPLE METHODS

A simple method is a method that uses a limited set of parameters to create a predictive model. The limited number of parameters reduces the flexibility to capture patterns in data.

The problem with simple models is **UNDERFITTING**, i.e. the model is not flexible enough to capture the patterns in the data.

A good example of a simple method is linear regression, which simply tries to fit a straight line through the data, and hence only uses an intercept parameter and one slope parameter for every dimension (every independent feature or predictor). That works very well if the data has more or less the shape of a cloud around a linear line. But that will not work if the data does not have a linear shape (e.g. U-shaped data). The only thing linear regression can do is to fit a straight line, nothing more, nothing less.

SIMPLE MODELS VERSUS COMPLEX MODELS UNDERFITTING VERSUS OVERFITTING

COMLEX METHODS

A complex method is a method that uses an extended set of parameters to create a predictive model. The extended number of parameters increases the flexibility to capture patterns in data.

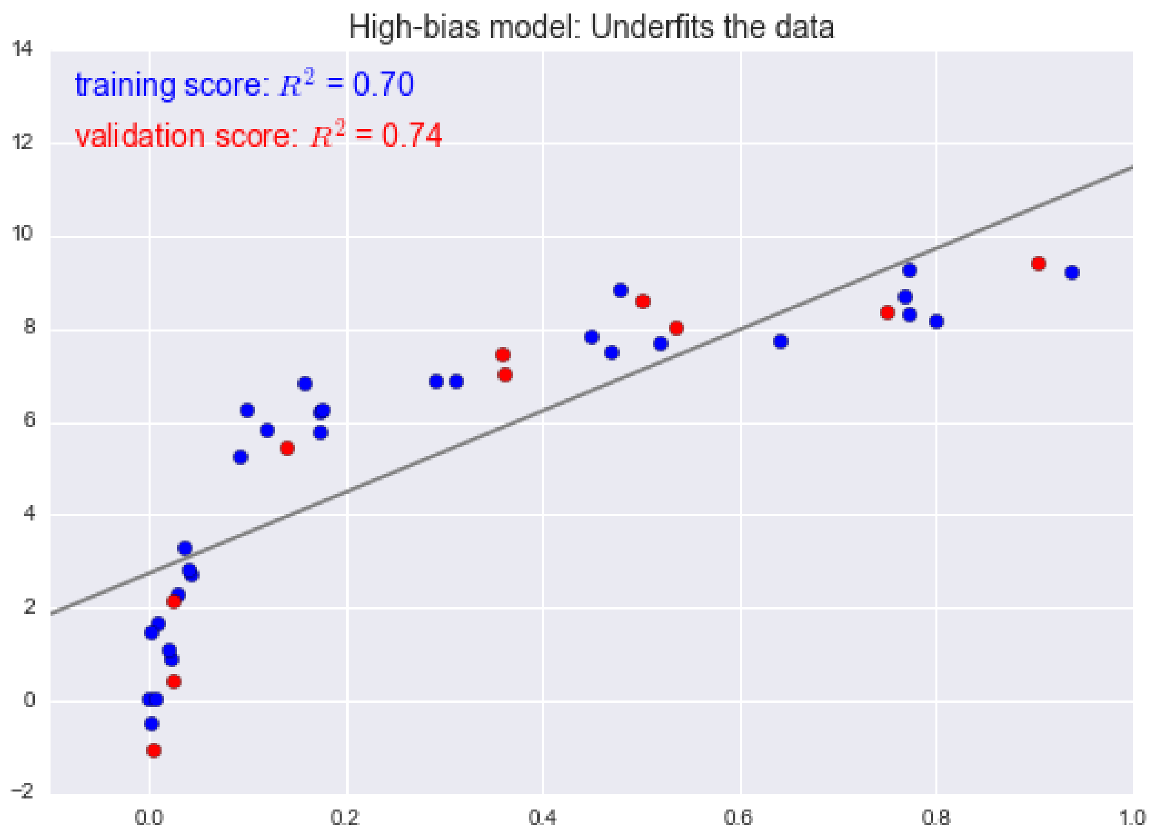
The problem with complex models is **OVERFITTING**, i.e. the model is too flexible and does not grasp the general pattern but just grasps the characteristics of individual cases.

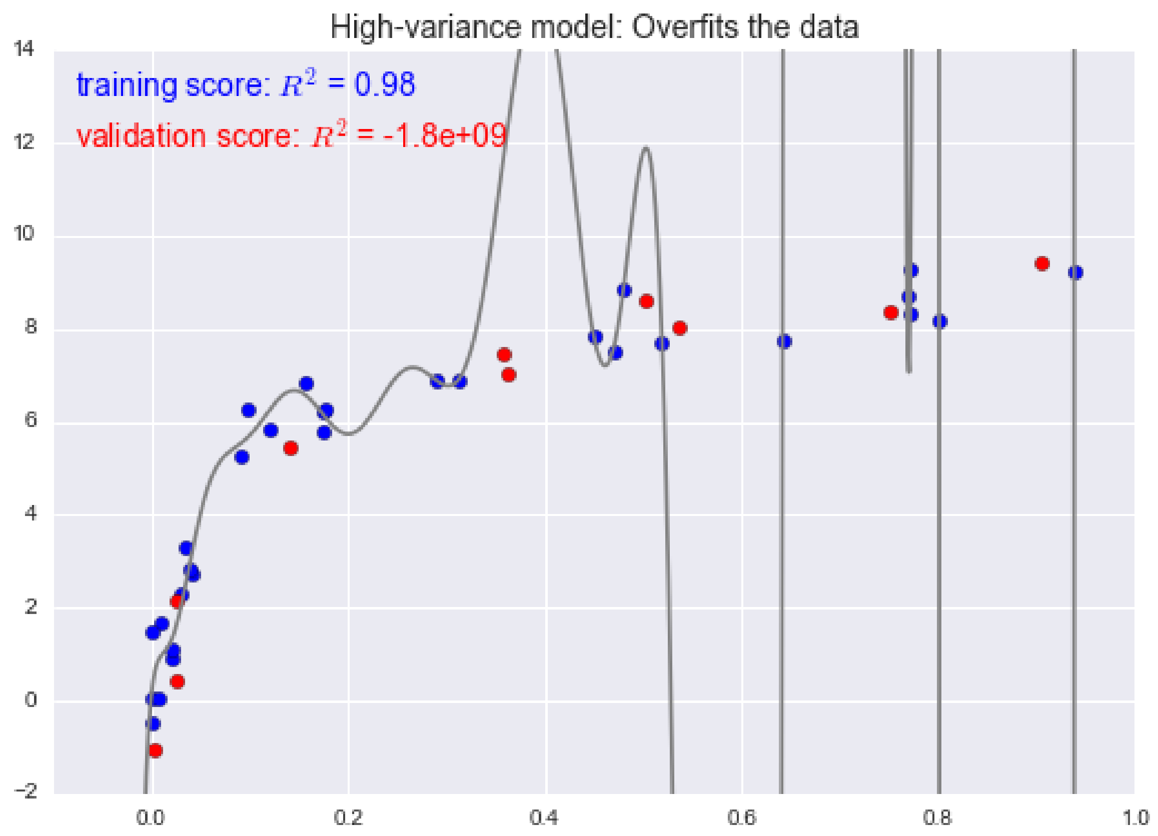
A good example of a complex method is high order polynomial regression, which tries to fit a high order polynomial through the data, and hence requires much more parameters besides intercept and slope ($c_1 x^1$, $c_2 x^2$, $c_3 x^3$, … $c_n x^n$). That works very well if the data has more or less the shape of the polynomial, but can go wrong if the data has a different shape, because the polynomial will start predicting individual cases instead of the general pattern.

HOW TO DETECT UNDERFITTING AND OVERFITTING

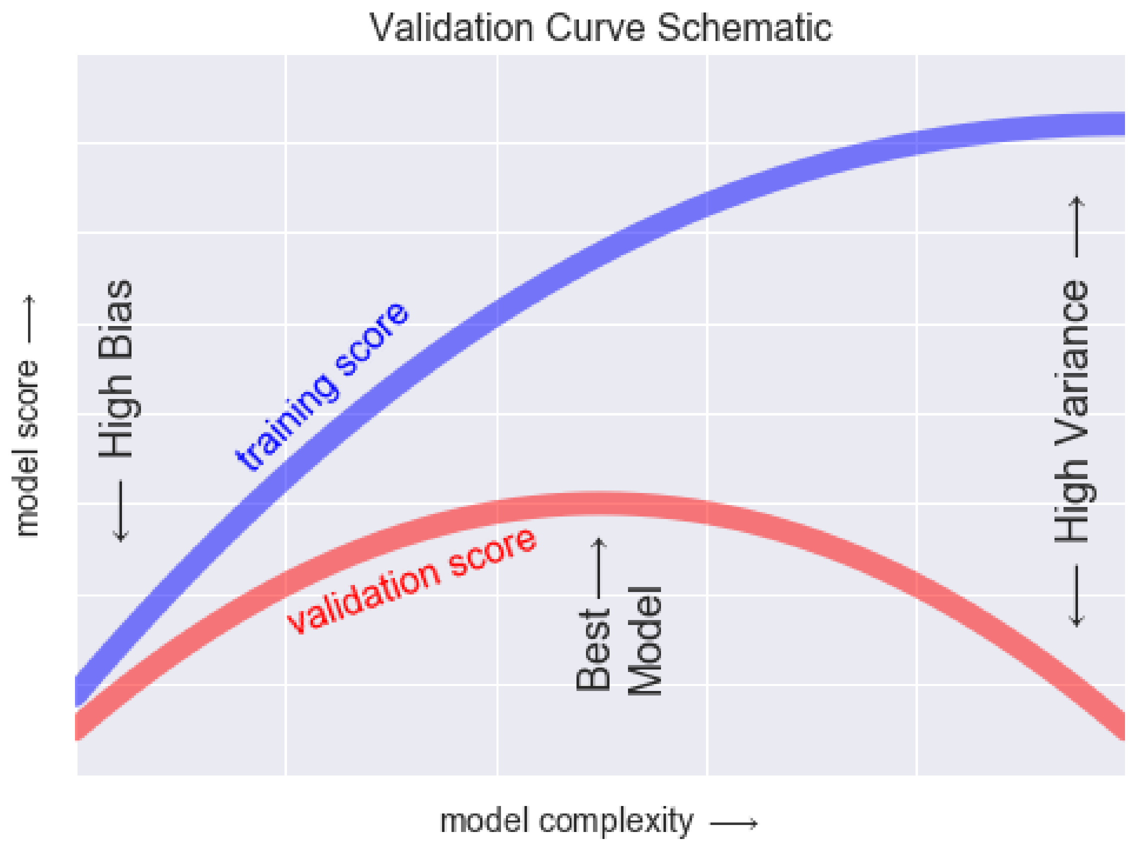
- Underfitting : the model will score bad on both training set and test set (the model is not able to capture any relevant pattern)
- Overfitting : the model will score good on the training set but bad on the test set (the model got tailored to the examples in the training set, but is not able to generalize, not able to predict new data well)

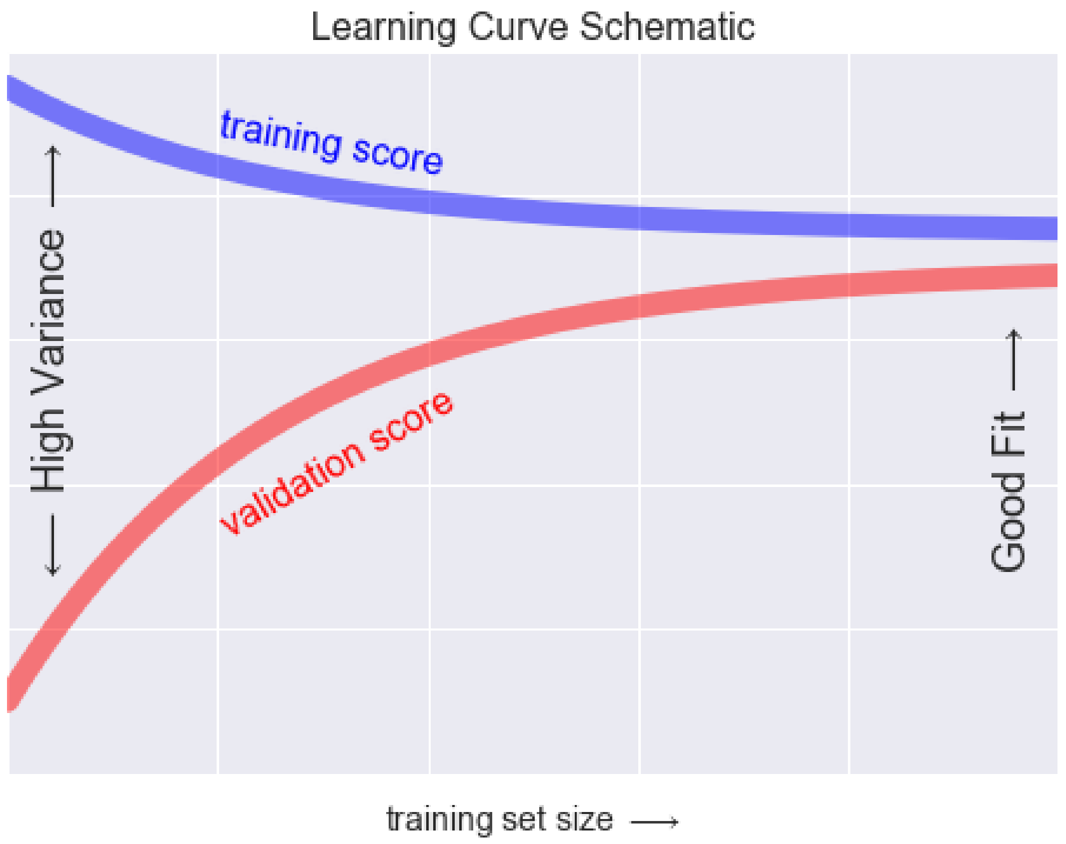
SIMPLE MODELS VERSUS COMPLEX MODELS
UNDERFITTING VERSUS OVERFITTING





SIMPLE MODELS VERSUS COMPLEX MODELS
UNDERFITTING VERSUS OVERFITTING





MODEL SELECTION AND HYPERPARAMETER TUNING
TRAIN – VALIDATE - TEST SPLIT

MODEL SELECTION AND HYPERPARAMETER TUNING

The challenge is to select the modeling technique that is most appropriate for your problem or data. This is called model selection (e.g. decision tree, support vector machine, artificial neural network).

But every method also has hyperparameters that also influence the model and hence can have a big impact on the model quality:

- Decision trees : tree depth, minimum samples split, maximum leaf nodes, . .
- Support vector machine : kernel function, C, gamma, …
- Artificial neural network : number of layers, number of neurons per layer, activation function, …

So it is not only about choosing the right modeling technique, but also the right hyperparameters. This is called hyperparameter tuning.

Mostly, it is only possible to find the best modeling technique and set of hyperparameters by trial and error. But doing this right requires a train-validate-test setup (see below), meaning that more labeled examples are lost for training.

TRAIN – VALIDATE - TEST SPLIT

Instead of a train-test split, model selection and hyperparameter tuning requires 3 subsets of labeled examples :

- Train dataset : dataset used to train multiple models with multiple hyperparameter sets. The result is a set of models with a set of hyperparameters, all trained on the same training set.
- Validation dataset : dataset to validate all models and hyperparameter sets generated in the previous step. All models and hyperparameter sets are validated using the same validation set. Select the best scoring model and hyperparameter set.
- Test dataset : dataset for the ultimate, independent test of the one selected best model and hyperparameter set from the previous step to check whether the selected model is general enough to predict new data.

MODEL SELECTION AND HYPERPARAMETER TUNING WITH CROSS-VALIDATION

USING CROSS-VALIDATION FOR MODEL SELECTION AND HYPERPARAMETER TUNING

Cross-validation comes in handy for model selection and hyperparameter tuning because it allows multiple training and testing runs on multiple labeled subsets making use of all information in the labeled dataset and excluding the coincidence factor of data selection.

Using cross-validation for model selection and hyperparameter tuning only requires the labelled example dataset to be split into two labeled subsets instead of three labeled subsets (train-validate-test subsets)

PRACTICAL SETUP : TRAIN/VALIDATE – TEST SPLIT

- Split the labeled data into two subsets: a train/validate subset and a holdout test subset
- Use the train/validate subset to train multiple modeling techniques with multiple hyperparameter sets using cross-validation

  Every modeling technique/hyperparameter set is trained and validated multiple times, each time validated by one partition and trained using all other partitions. For every modeling technique/hyperparameter set, the average of all validation scores over all runs is calculated
- Select the modeling technique and hyperparameter set with the highest average score (average of all scores over all runs of the cross-validation) in the previous step
- Train a new model using the best modeling technique and hyperparameter set as selected in the previous step. Use all data in the train/validate labeled subset for the training

  So the train/validate labeled subset is used twice, once to find the best modeling technique/hyperparameter set using cross-validation, once to train a new model based on the best modeling technique/hyperparameter set
- Test the model trained in the previous step using the holdout test set for an independent test (test generalization on new data)

HYPERPARAMETER TUNING WITH GRID SEARCH

GRID SEARCH

Grid search is way to test a method for multiple hyperparameter combinations. First a set of hyperparameters to be checked is compiled. Next, for every selected hyperparameter, a set of parameter values to be checked is compiled. The combination of all parameter values of interest forms a grid (e.g. a set of 2 hyperparameters, one with 4 parameter values of interest, one with 3 parameter values of interest, form a grid of 4 by 3, resulting in 12 hyperparameter value combinations). Next every cell of the grid is searched, i.e. a model with the combination of hyperparameter values present in the grid cell is trained and validated. In the end, the best scoring model can be used to select the best combination of hyperparameter values.

FULL GRID SEARCH AND RANDOM GRID SEARCH

The total number of hyperparameter value combination can become very high (e.g. 5 hyperparameters with an average of 4 hyperparameter values results in 1024 combinations; for neural networks, far more combinations are possible as the number of layers and the amount of neurons per layer can be high).

A full grid search with train and test a model for every cell in the grid, i.e. every combination of hyperparameter values.

For large datasets, and complex methods, training can become very long, making it practically impossible to do a full grid search. In that case, a random grid search can be used, where grid cells are randomly chosen and only those combination are trained and tested.

More advanced optimisation methods exist apart from pure random selection.

CROSS-VALIDATION AND GRID SEARCH PIPELINE WITH SCIKIT-LEARN (CLASSIFICATION)

```
# DATA PREPARATION
import pandas as pd
pd.options.display.max_rows = None
import seaborn as sns
iris = sns.load_dataset('iris')
y = iris['species'] # Target feature to predict
X = iris.copy().drop('species', axis=1) # Predictors

# SPLIT LABELED DATA INTO TRAIN/VALIDATE - TEST SAMPLE
from sklearn.model_selection import train_test_split
# Split the data randomly into 80% training set and 20% test set
X_tr, X_tst, y_tr, y_tst = train_test_split(X, y, random_state=0, train_size=0.8)
# (use random_state to be sure that every time the same random sample is drawn)
```

CROSS-VALIDATION AND GRID SEARCH PIPELINE WITH SCIKIT-LEARN (CLASSIFICATION)

```
# MODEL SELECTION AND HYPERPARAMETER TUNING (REPEAT THIS STEP FOR MULTIPLE MODELING TECHNIQUES)
from sklearn.model_selection import GridSearchCV
from sklearn.tree import DecisionTreeClassifier
model = DecisionTreeClassifier()

# Define parameter grid (model specific)
grid_param = {'criterion' : ['gini', 'entropy', 'log_loss'],
              'max_depth' : list(range(2,10)),
              'min_samples_split' : list(range(2,5))}

# Setup grid search with N-fold cross validation (e.g. 5-fold)
grid_search = GridSearchCV(model, grid_param, cv=5)

# Execute full grid search
grid_search.fit(X_tr, y_tr)

# Display best hyperparameter values and matching validation score
print(f'Best parameters : {grid_search.best_params_}')
print(f'Best score      : {grid_search.best_score_:.3f}')
```

CROSS-VALIDATION AND GRID SEARCH PIPELINE WITH SCIKIT-LEARN (CLASSIFICATION)

```
# DERIVE MODEL FROM TRAINING DATA USING BEST HYPERPARAMETER VALUES (TRAIN MODEL/FIT MODEL)
model.set_params(**grid_search.best_params_)
# List all selected hyperparameters
print(model.get_params(deep=True))

model.fit(X_tr, classification_report,y_tr)

# DISPLAY MODEL (MODEL SPECIFIC)
from sklearn.tree import plot_tree
plot_tree(model)


# VALIDATE MODEL USING TEST DATA
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_score,
f1_score, precision_recall_fscore_support
import matplotlib.pyplot as plt

# Predict target feature for the test data
y_tst_pred = pd.Series(model.predict(X_tst), name= 'y_tst_pred')
```

CROSS-VALIDATION AND GRID SEARCH PIPELINE WITH SCIKIT-LEARN (CLASSIFICATION)

```
# Calculate the difference between predicted and real values for the test data
err = pd.Series(y_tst_pred.reset_index(drop=True)!=y_tst.reset_index(drop=True),
name='err').astype(int)
display(pd.concat([y_tst.reset_index(drop=True), y_tst_pred.reset_index(drop=True), err], axis=1))

# Confusion matrix
# Display as text (console output)
class_labels = sorted(list(pd.concat([y_tst,y_tst_pred], axis=0).unique()))
# Alternative : model.classes_
cm = confusion_matrix(y_true = y_tst, y_pred = y_tst_pred)
print('Predicted label')
print(class_labels)
print(cm)
# Display as heatmap (nicer output in Jupyter)
disp = sns.heatmap(cm, square=True, annot=True, cbar=True, cmap='Greys', xticklabels=class_labels,
yticklabels=class_labels)
plt.xlabel('Predicted label')
plt.ylabel('True label')
disp.xaxis.tick_top()                # Put x-axis tickers on top
disp.xaxis.set_label_position('top') # Put x-axis label on top
```

CROSS-VALIDATION AND GRID SEARCH PIPELINE WITH SCIKIT-LEARN (CLASSIFICATION)

```
# Metrics
acc = accuracy_score(y_true=y_tst, y_pred=y_tst_pred)
prec = precision_score(y_true=y_tst, y_pred=y_tst_pred, average='weighted')
rec = recall_score(y_true=y_tst, y_pred=y_tst_pred, average='weighted')
f1 = f1_score(y_true=y_tst, y_pred=y_tst_pred, average='weighted')
# Mind this is a multiclass classification problem, so precision, recall and F1
# are calculated by class and averaged.
print(f'ACC : {acc:.3f} - PREC : {prec:.3f} - REC : {rec:.3f} - F1 : {f1:.3f}’)

# The easiest way to get results by class is to use precision_recall_fscore_support
class_labels = sorted(list(pd.concat([y_tst,y_tst_pred], axis=0).unique()))
# Display precision/recall/fscore/support table as text (consule output)
print(class_labels)
display(precision_recall_fscore_support(y_true=y_tst, y_pred=y_tst_pred))
# Display precision/recall/fscore/support as pandas dataframe (nicer outputin Jupyter)
display(pd.DataFrame(precision_recall_fscore_support(y_true=y_tst, y_pred=y_tst_pred),
index=['prec','rec','fscore','sup'], columns=class_labels))

# Or use classification_report
print(classification_report(y_true=y_tst, y_pred=y_tst_pred, target_names=class_labels))
```

CROSS-VALIDATION AND GRID SEARCH PIPELINE WITH SCIKIT-LEARN (CLASSIFICATION)

```
# APPLY MODEL ON NEW DATA
X_pred = …. (new feature data to predict the target feature for)
y_pred = model.predict(X_pred)
```