



Seminar Grundlagen Machine Learning

Methoden und Algorithmen zur praktischen
Umsetzung mit Python

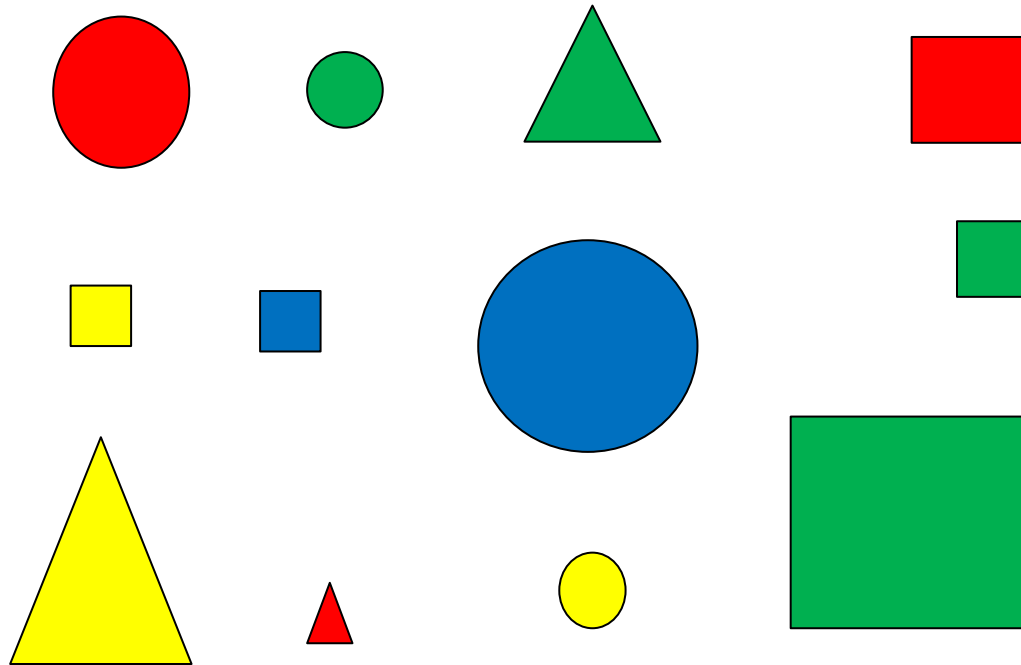


03: Classification

Fundamentals

Defining classification

Remember the introduction to “clustering”?
The task of clustering is to group instances

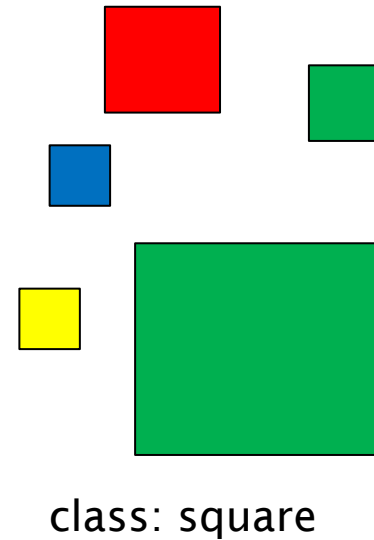
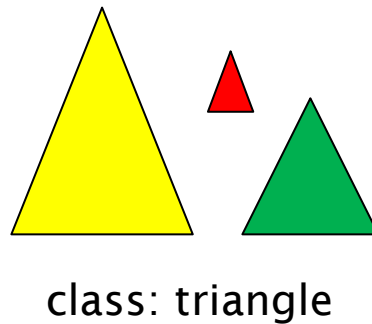
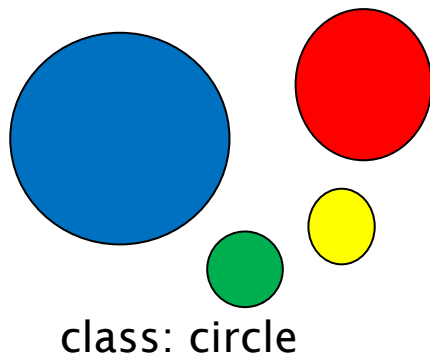


➤ **Classification:**
“Classify” an instance as any of N **pre-defined** classes.

Fundamentals

Defining classification

Example 1: Classify the instances as one of the three classes “circle”, “square”, “triangle”



Fundamentals: Machine Learning – on one slide

Classification of apples and pears

Training

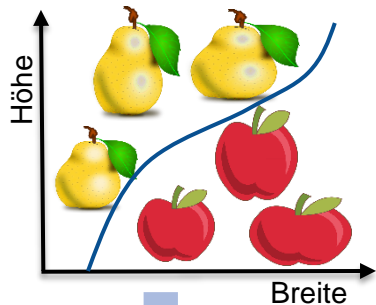
1. training set + class labels are passed to a model
2. the model learns a decision function



„feature vectors“ +
„class labels“

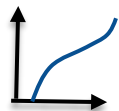
width: 7cm, height: 6cm + „apple“
width: 5cm, height: 9cm + „pear“
...

„feature space“



**Machine Learning
model**

„decision function“

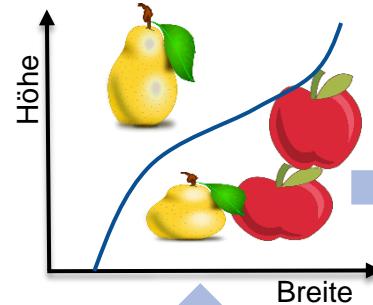


Test

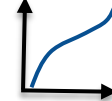
1. test data is passed to model without class labels
2. the model classifies the test data using the decision function
3. results are compared to the true class labels


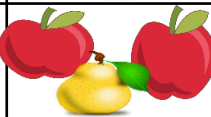


width: 6 cm, height: 5 cm
width: 4 cm, height: 8 cm
...



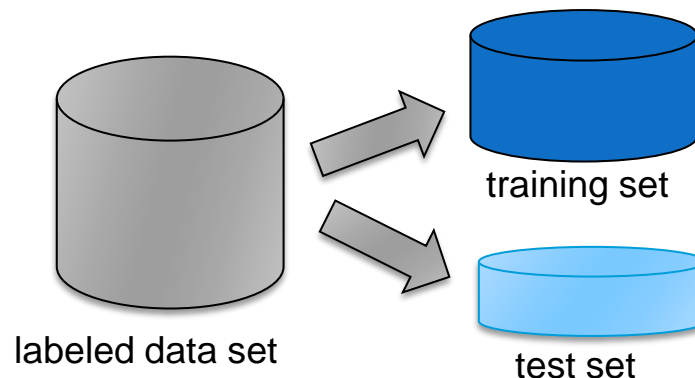
**Machine Learning
model**



pear	
apple	
error (simple error measure)	$\frac{1}{4} = 25\%$

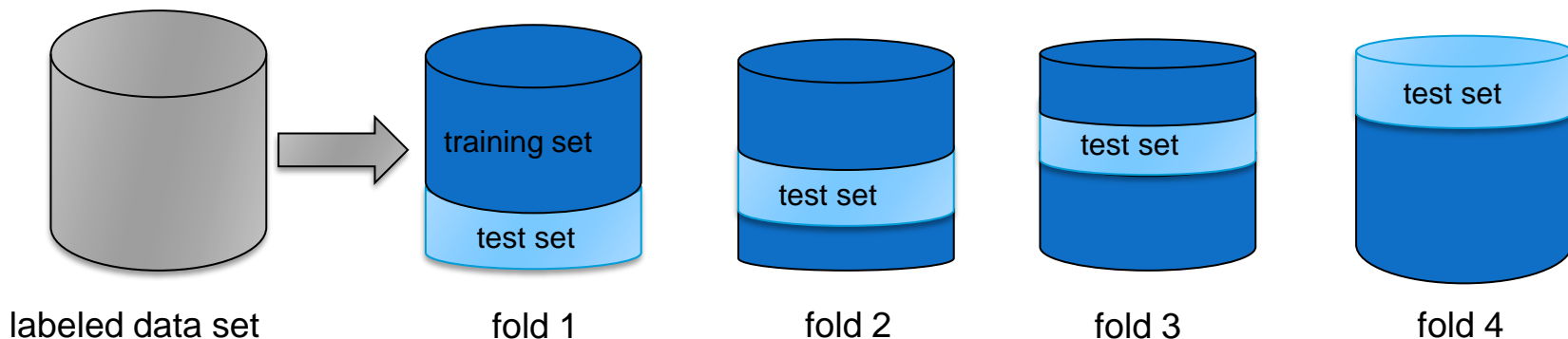
Separation of training and test set: „hold-out“

- split the labeled data set into training set and test set
- **we should not use instances of the training set during test and vice-versa!**



Separation of training and test set: „cross-validation“

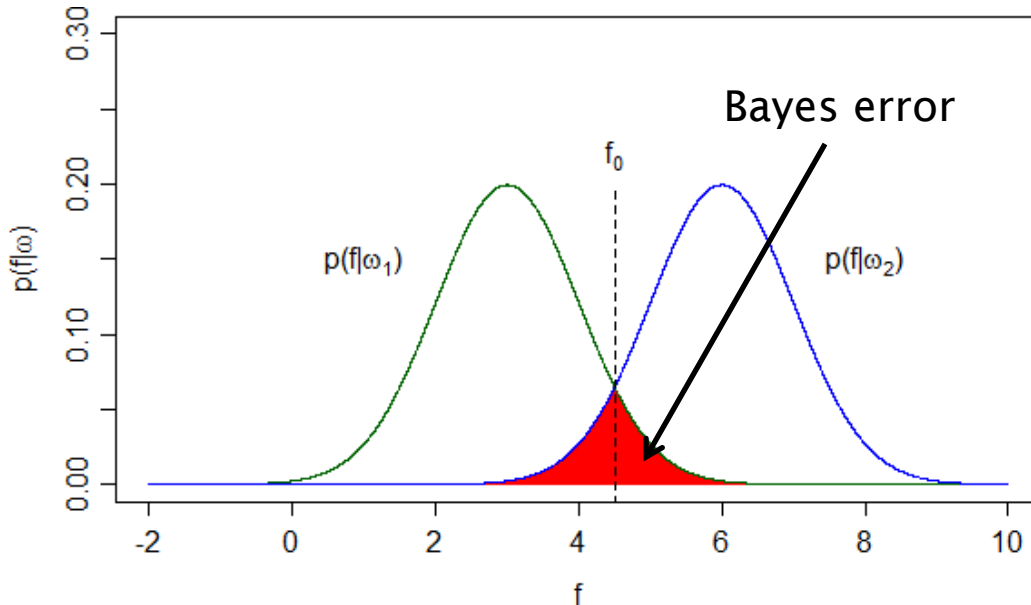
- typically there is a lack of labeled training data
- **one common method is cross-validation with „k-fold“:**
 - randomly split the training set into k chunks
 - run training and test k times using each chunk as test set once and the remaining k-1 chunks as training set
 - outputs k accuracies, that we can average
=> can also be used to estimate the variability of the expected results



Fundamentals

Statistical background

The Bayes error



- The **Bayes error** lower bounds the error rate for **any classifier**.
- note: the error is formulated independently of classifier properties
- the error exclusively depends on the properties of the data set in feature space F

In applications, the Bayes error is usually unknown!

Calculating the Bayes error requires knowledge of:

1. the type of probability density functions
2. the statistical parameters of the probability distributions
3. the prior class probabilities

Evaluation criteria for Machine Learning models (Han et al.):

1. accuracy
2. speed (computational costs)
3. robustness (robustness against noise in training sets)
4. scalability (e.g. computational costs)
5. interpretability (understandability of the classifier or its results)

Fundamentals

Machine learning: classification





Measuring the classification results

In the test phase, the number of correctly and incorrectly classified instances are counted and stored in the so-called

“confusion matrix” (=“contingency table”):

(Note: rows and columns are not consistent in tools and literature!

Might be vice versa!)

classification results (prediction)	class labels (true class)	
	„square“	„circle“
„square“	true „squares“: 	false „squares“: 
„circle“	false „circles“: 	true „circles“: 

Fundamentals

Classification results

Confusion matrix

- ▶ measuring classifier outputs (here for two classes):

result (prediction)	class label (true class)	
	POSITIVE	NEGATIVE
positive	TP	FP
negative	FN	TN

Based on the confusion matrix, a variety of measures can be calculated (see e.g. (Fawcett 04)). Some of them are:

1. *overall accuracy* $= \frac{TP+TN}{M} = \frac{TP+TN}{POSITIVE+NEGATIVE}$
2. *true positive rate TPR* $= \frac{TP}{POSITIVE} = \frac{TP}{TP+FN}$
3. *precision_{pos}* $= \frac{TP}{positive} = \frac{TP}{TP+FP}$

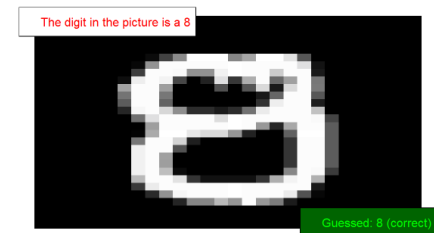
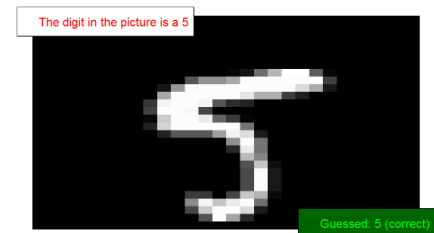
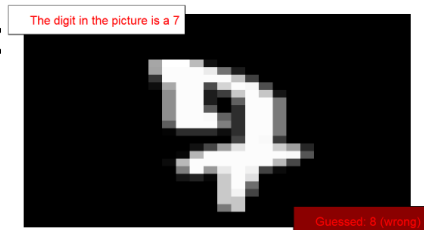
TP: number of true positives
FP: number of false positives
FN: number of false negatives
TN: number of true negatives
POSITIVE = TP + FN
NEGATIVE = TN + FP
M = number of instances
= POSITIVE + NEGATIVE

Fundamentals

Classification results

For many classes, the confusion matrix becomes hard to read
 Example: classification of handwritten digits 0...9 („MNIST“)

Prediction	Reference									
	0	1	2	3	4	5	6	7	8	9
0	973	0	3	0	2	2	6	2	5	3
1	0	1127	2	0	1	0	2	2	0	4
2	0	2	1013	1	2	0	1	11	3	1
3	1	0	2	997	0	10	0	1	3	5
4	0	0	1	0	963	1	4	0	5	7
5	1	0	1	4	0	865	4	1	3	3
6	2	2	1	0	5	6	941	0	2	0
7	1	0	5	3	1	1	0	1006	4	6
8	1	4	4	2	1	4	0	1	944	0
9	1	0	0	3	7	3	0	4	5	980



Accuracy : 0.9809

Statistics by Class:

Class: 0	Class: 1	Class: 2	Class: 3	Class: 4	Class: 5	Class: 6	Class: 7	Class: 8	Class: 9
0.9929	0.9930	0.9816	0.9871	0.9807	0.9697	0.9823	0.9786	0.9692	0.9713

Fundamentals

Classification results

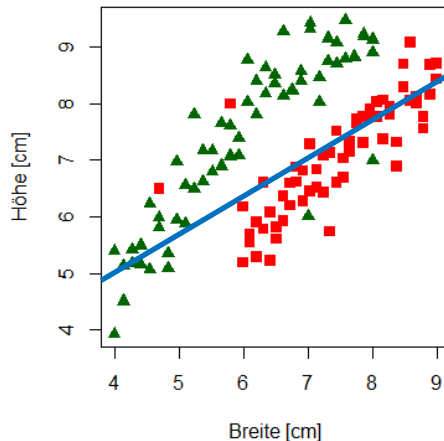
- ▶ Interpret the confusion matrix on the previous slide
- ▶ Name three findings you can see in the matrix

- ▶ *Possible solutions:*
 - *6 digits that are „9“ were misclassified as „7“*
 - *980 digits that are „9“ were correctly classified as „9“*
 - *98% of all digits were classified correctly*
 - *„0“ classified as „0“: 99,29%, ...*

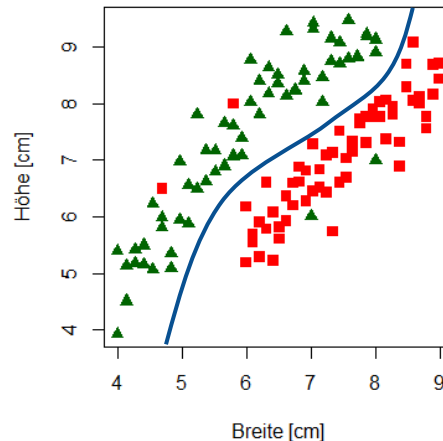
Artificial neural networks (ANNs)

Overfitting and underfitting

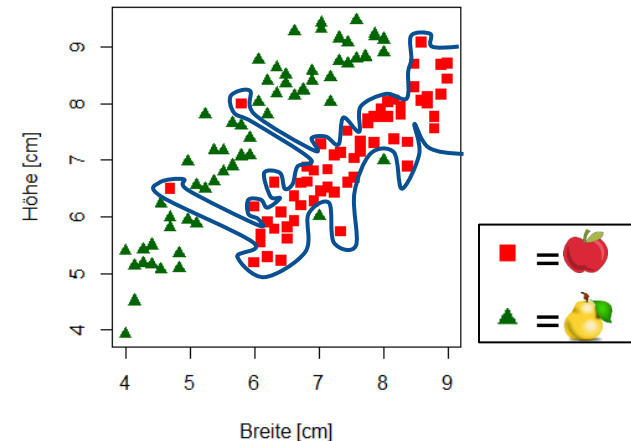
- ▶ **overfitting:** the classifier learns the specifics of the training set, but does not generalize well
- ▶ **underfitting:** the classifier does not learn the training set well enough (e.g. because the decision functions are not flexible enough)



„underfitting“:
error on training set: high
error on test set: high



good model:
error on training set: low
error on test set: low



„overfitting“:
error on training set: very low
error on test set: high

➤ Note:

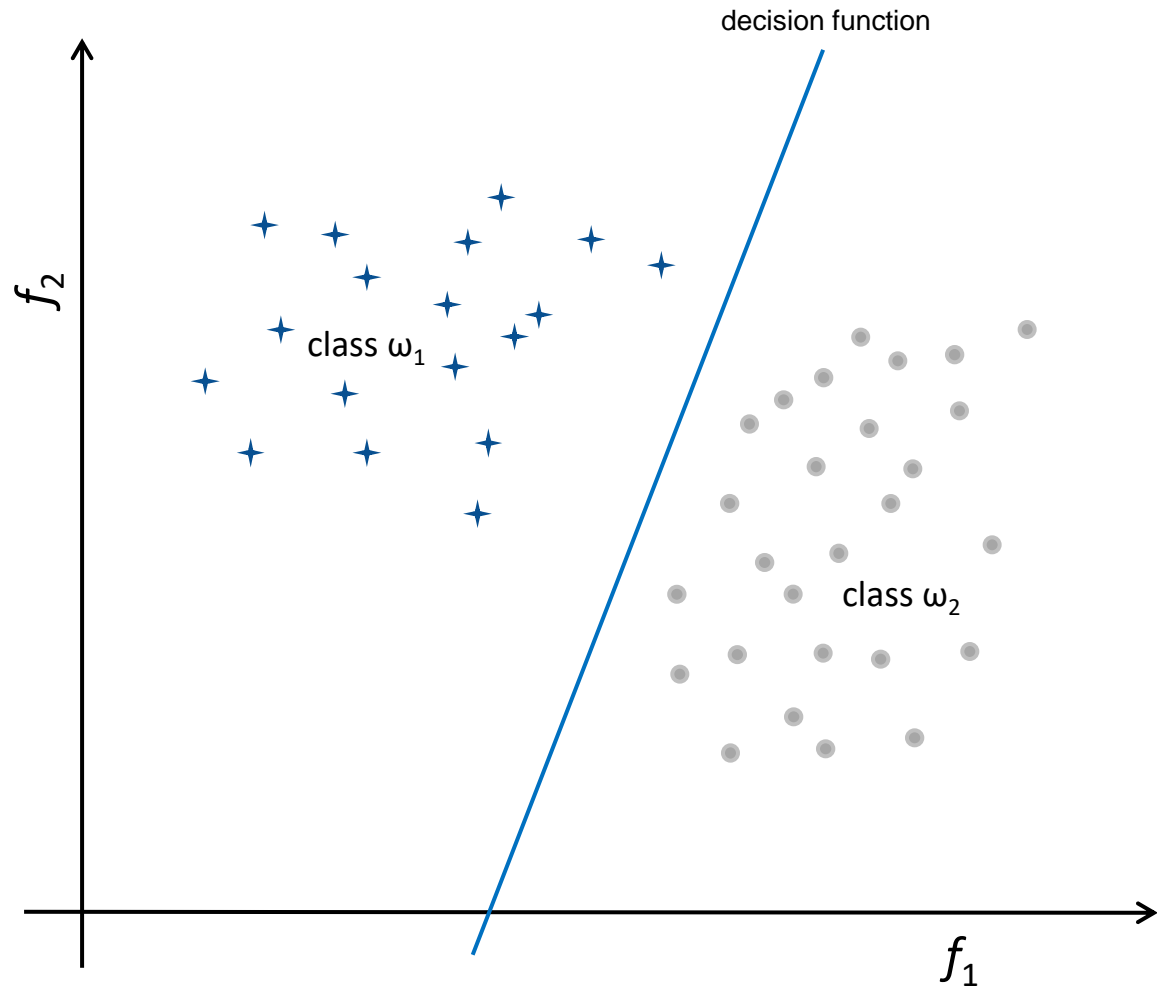
We do not want the optimal solution on the training set.
We want an optimal solution on unseen data!

Classifiers

Linear classifiers

General description of linear classifiers

- **the idea:**
separate classes with a linear decision function

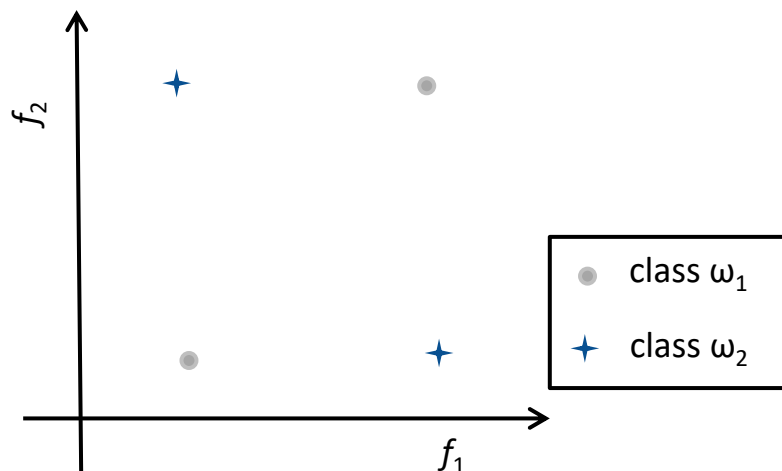


Classifiers

Non-linear classifiers

The „XOR-problem“

- linear classifiers have a strong limitation:
 - they work well when the classes are linearly separable, if this is not the case they can't find a good solution
- let's have a look at the so-called “XOR-problem”:
 - a data set with only four instances that appears to be quite simple
 - try to separate the two classes using any type of linear classifier

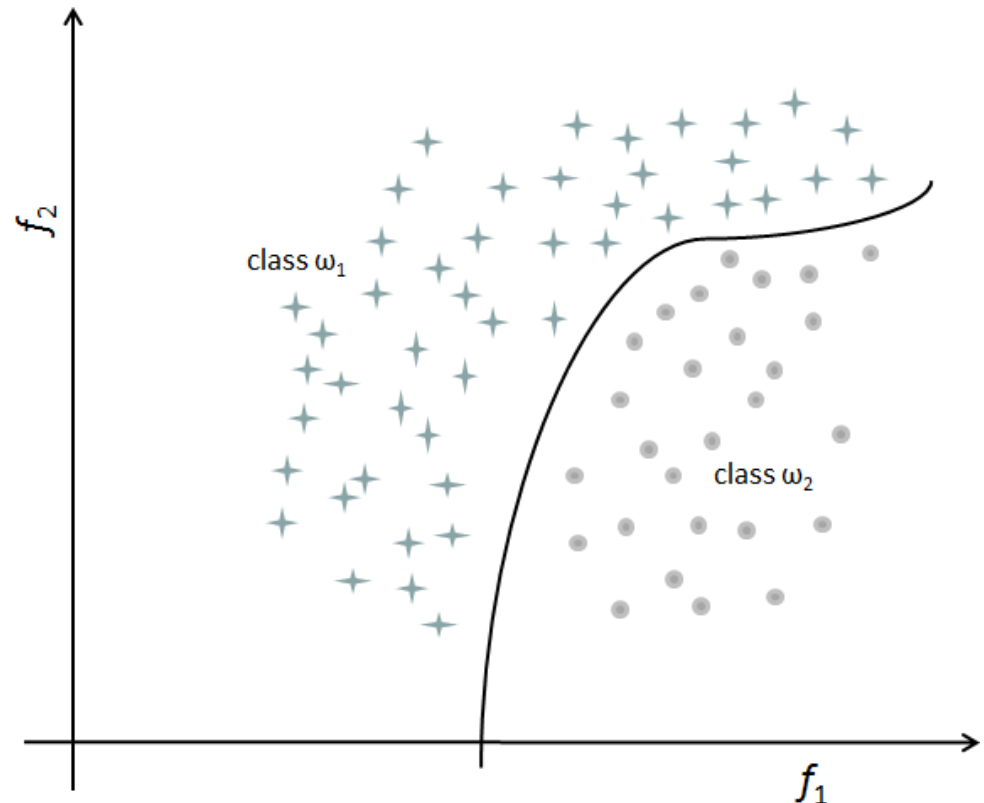


Classifiers

Non-linear classifiers

Introducing non-linear classifiers

- to overcome the problem of linearly non-separable data sets: classifiers with non-linear decision boundaries are introduced in the following slides



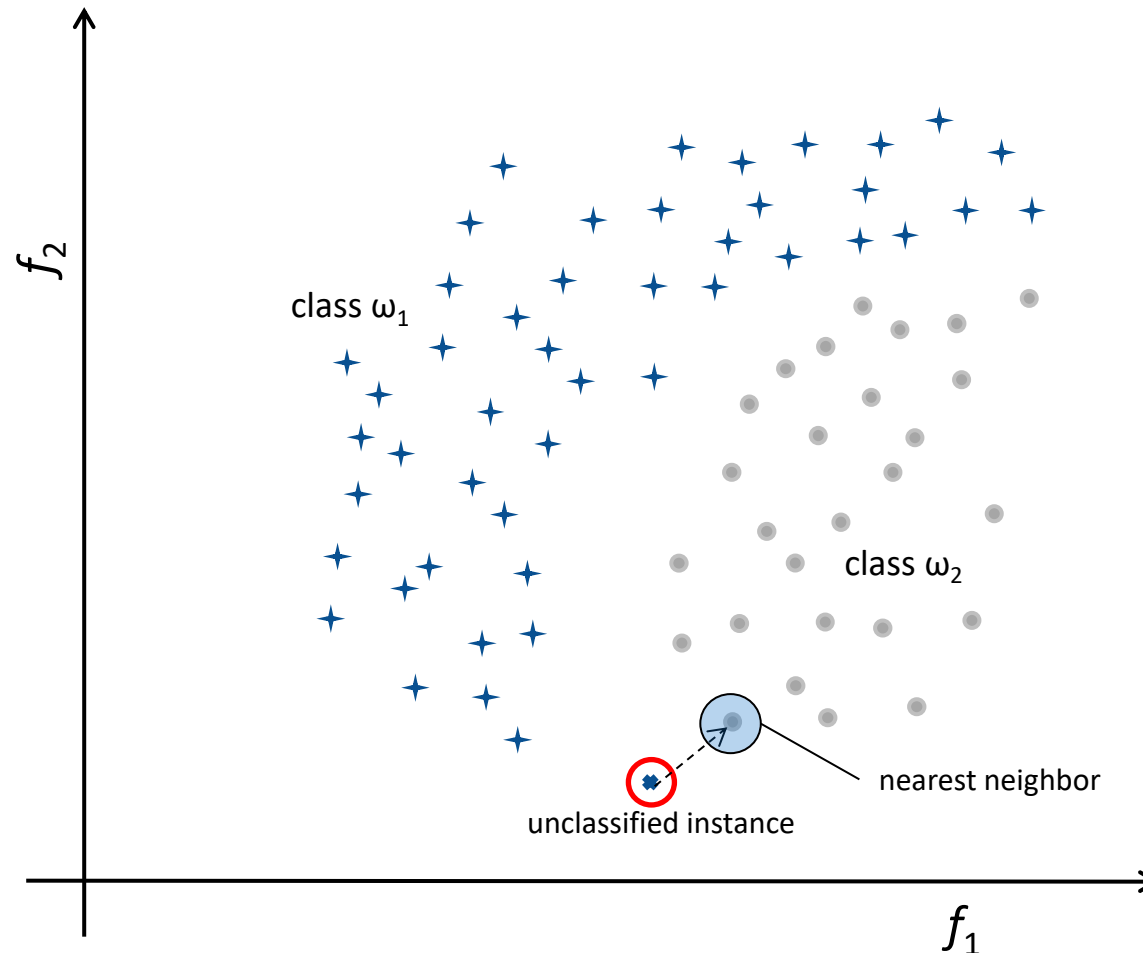
nearest neighbor (NN, 1-NN)

- **the idea:**
an instance belongs to the class of its nearest neighbor
- **how it works:**
 1. store all instances from a training set
 2. for an unclassified instance find the nearest neighbor in the training set
 3. assign the class of the nearest neighbor

Classifiers

Non-linear classifiers

nearest neighbor (NN, 1-NN)



Classifiers

Non-linear classifiers

nearest neighbor (NN, 1-NN): evaluation

- interpretability
- easy to implement
- can be applied to any type of data set, if a distance measure can be defined
- training phase is extremely fast (training corresponds to just storing the instances)

advantages +

- robustness: sensitive to individual outliers in the training set
- scalability: classification can be slow, since it requires visiting of all instances to find the nearest neighbor

disadvantages
-

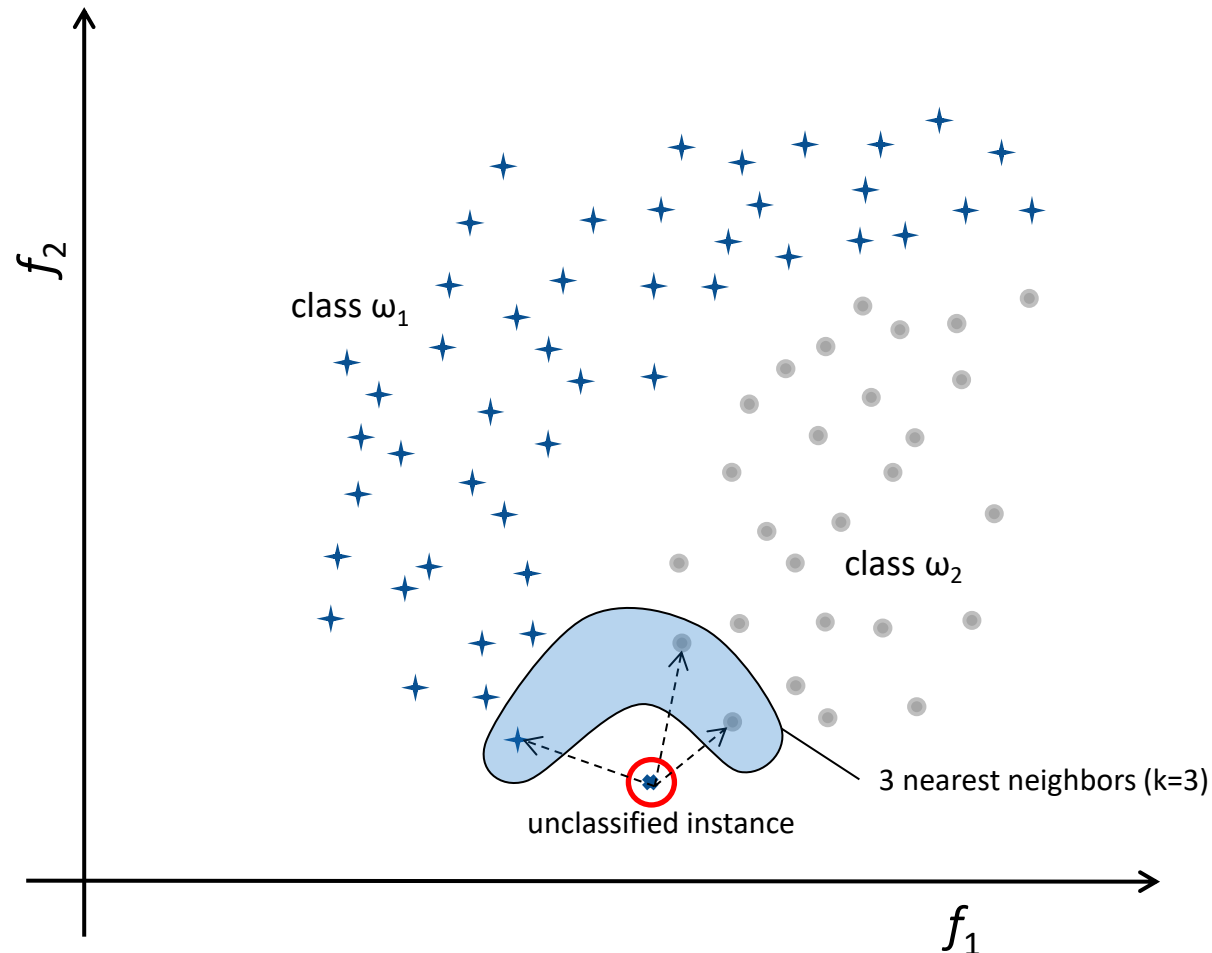
k-nearest neighbors (k-NN with $k > 1$)

- **the idea:**
an instance belongs to the class of its k nearest neighbors
- **how it works:**
 1. store all instances from a training set
 2. select the parameter k
 3. for an unclassified instance find the k nearest neighbor in the training set
 4. classify the instance based on the majority class in the k nearest neighbors (typically odd numbers are used: 3, 5, 7, ...)

Classifiers

Non-linear classifiers

► k-nearest neighbors (k-NN)



Classifiers

k-nearest neighbours (k-NN)



```
# sklearn imports
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix, accuracy_score
from sklearn.preprocessing import MinMaxScaler

# create data set
data, labels = make_moons(n_samples=500, noise=0.1)

# split into training and test set
train_data, test_data, train_labels, test_labels = train_test_split(data,
                                                                    labels, test_size = 0.5)

[...]
```

Classifiers

k-nearest neighbours (k-NN)



[...]

```
# min-max scaling: determine scaling parameters
```

```
scaler = MinMaxScaler().fit(train_data)
```

```
# scale train set and test set
```

```
train_data = scaler.transform(train_data)
```

```
test_data = scaler.transform(test_data)
```

```
# create k-nearest neighbours with k = 3
```

```
model = KNeighborsClassifier(n_neighbors=3)
```

```
# train model on training set
```

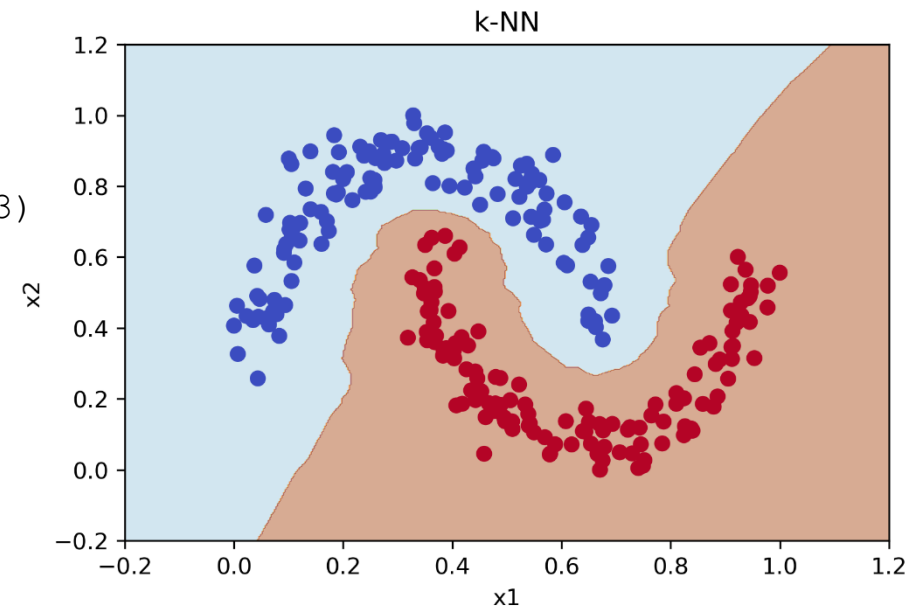
```
model.fit(train_data, train_labels)
```

```
# classification of test set
```

```
predictions = model.predict(test_data)
```

```
acc = accuracy_score(test_labels, predictions)
```

```
cm = confusion_matrix(test_labels, predictions)
```



Classifiers

Non-linear classifiers

k-nearest neighbors (k-NN): evaluation

- interpretability
- easy to implement
- can be applied to any type of data set, if a distance measure can be defined
- training phase is extremely fast (training corresponds to just storing the instances)

advantages +

- scalability: classification can be slow, since it requires visiting of all instances to find the k nearest neighbors

disadvantages -

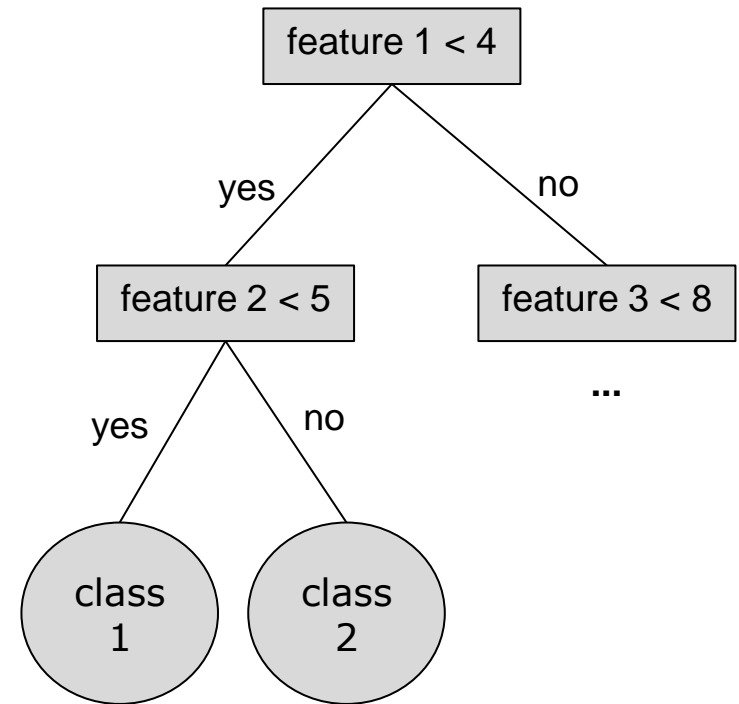
Decision trees

The idea:

separate the classes in the training set by recursively splitting the data by thresholding individual features

The algorithm:

1. create a node
2. select the best split using some **splitting criterion**
3. grow branches according to the split (binary vs. multiple branches)
4. recursively continue with 1. until some stopping criterion is met or all branches contain only feature vectors from one class („purity“)
=> bottom nodes become „leaves“



Decision trees

- ▶ learning a decision tree from a training set is referred to as **decision tree induction**
- ▶ to avoid overfitting, the resulting tree can be pruned
- ▶ there are different splitting criteria, i.e. techniques to identify the features to be used in the current split together with the threshold
- ▶ per split/node **one feature is used** (univariate split)
- ▶ however, there are advancements of trees combining features (multivariate splits)

- ▶ common decision tree algorithms:
 - ▶ ID3 (Quinlan, 1986)
 - ▶ C4.5 (Quinlan, 1993)
 - ▶ C5.0 (Quinlan, 2017)
 - ▶ CART (Classification and Regression Trees) (Breiman, 1984)

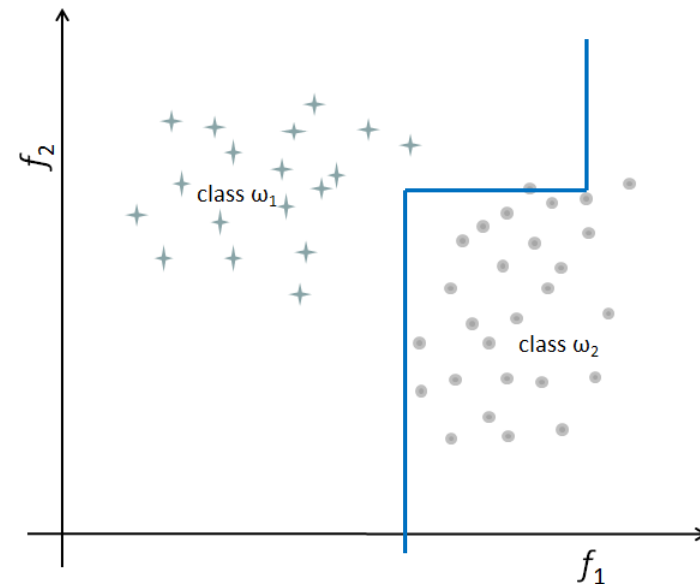
- ▶ although having been around for many years, decision trees are worth to study:
 - ▶ they are interpretable (a currently hot research topic in AI)
 - ▶ they are the components of powerful, advanced machine learning methods like **random forests** and **gradient boosting machines (e.g. xgboost)**

Classifiers

Non-linear classifiers

Decision trees

- the resulting decision boundaries are piecewise linear
- a decision tree can be transformed into a rule-base:
 if feature 1 < x ...
- (there are advanced techniques using a combination of several features per node)



Decision trees

Splitting criterion: information gain

Information gain uses **entropy** from information theory (Shannon)

- ▶ the split with the highest information gain is chosen, i.e. the split that minimizes the information needed to classify the remaining feature vectors
- ▶ in other words: the split that creates the **least "impurity"**
- ▶ a split can create multiple branches ($N > 2$)

- ▶ in the context of decision trees this entropy corresponds to a measure of impurity

The entropy is measured as information in bits and is calculated by:

$$I(D) = - \sum_{i=1}^{|C|} p_i * \log_2(p_i)$$

- ▶ with D : data set, $|C|$: number of classes, $p_i = \frac{|C_i \text{ in } D|}{|D|}$: probability that a feature vector in D belongs to class C_i
- ▶ $I(D) = 0 \dots 1$, where $I(D) = 0$, if all feature vectors belong to the same class

Decision trees

Splitting criterion: information gain

In order to find the best split, the entropy for all possible splits is calculated. For a split A:

$$I(D_{split\ A}) = - \sum_{j=1}^{|S|} \frac{|D_j|}{|D|} * I(D_j)$$

with:

- ▶ $|S|$: number of possible splits, i.e. for discrete, categorical or binary values, the number of different values
- ▶ $|D_j|$: number of feature vectors in partition j
- ▶ $\frac{|D_j|}{|D|}$: in order to weight according to the number of feature vectors per partition

From all candidate splits, the one is selected that maximizes the **information gain IG** :

$$IG_{split\ A} = I(D) - I(D_{split\ A})$$

- ▶ in other words: $IG_{split\ A}$ determines how much is gained by the splitting candidate A

Decision trees

Splitting criterion: information gain

Exercise: „Information gain“

Calculate the information gain for the following two-class problem with categorical attributes

- ▶ number of feature vectors: 14
- ▶ two classes: P and N
- ▶ 9 feature vectors of class P, 5 of class N

No.	Attributes			Class
	Outlook	Temperature	Humidity	
1	sunny	hot	high	N
2	sunny	hot	high	N
3	overcast	hot	high	P
4	rain	mild	high	P
5	rain	cool	normal	P
6	rain	cool	normal	N
7	overcast	cool	normal	P
8	sunny	mild	high	N
9	sunny	cool	normal	P
10	rain	mild	normal	P
11	sunny	mild	normal	P
12	overcast	mild	high	P
13	overcast	hot	normal	P
14	rain	mild	high	N

adapted from original ID3 paper: (Quinlan, 1986)

1. Entropy of D :

- ▶ $p_P = \frac{9}{14}$; $p_N = \frac{5}{14}$
- ▶ $I(D) = -\sum_{i=1}^{|C|} p_i * \log_2(p_i)$
 $= -\frac{9}{14} * \log_2\left(\frac{9}{14}\right) - \frac{5}{14} * \log_2\left(\frac{5}{14}\right) = 0.94 \text{ bits}$

2. Entropy of possible split by feature „outlook“:

- ▶ $|S| = 3$; $S = (\text{sunny, overcast, rain})$
- ▶ $|D_1| = 5$; $|D_2| = 4$; $|D_3| = 5$: (possible splits by S)

- ▶ $I(D_{\text{split Outlook}}) = -\sum_{j=1}^{|S|} \frac{|D_j|}{|D|} * I(D_j)$
 $= \frac{5}{14} * \left(-\frac{2}{5} * \log_2\left(\frac{2}{5}\right) - \frac{3}{5} * \log_2\left(\frac{3}{5}\right) \right)$
 $+ \frac{4}{14} * \left(-\frac{4}{4} * \log_2\left(\frac{4}{4}\right) \right)$
 $+ \frac{5}{14} * \left(-\frac{3}{5} * \log_2\left(\frac{3}{5}\right) - \frac{2}{5} * \log_2\left(\frac{2}{5}\right) \right) = 0.694 \text{ bits}$

3 out of the 5 feature vectors with feature=„sunny“ are of class N

3. Information gain for this split:

- ▶ $IG_{\text{split Outlook}} = I(D) - I(D_{\text{split Outlook}}) = 0.246 \text{ bits}$

4. Repeat this for features „Temperature“ and „Humidity“ and find maximum information gain.

Decision trees

Splitting criterion: Gini index

Gini index finds that **binary split** that minimizes the “impurity”

The Gini index for D is calculated:

$$Gini(D) = 1 - \sum_{i=1}^{|C|} p_i^2$$

with notation equivalent to information gain:

- ▶ D : data set, $|C|$: number of classes, $p_i = \frac{|C_i \text{ in } D|}{|D|}$: probability that a feature vector in D belongs to class C_i
- ▶ $Gini(D) = 0 \dots 1$, where $Gini(D) = 0$, if all feature vectors belong to the same class, which is the ideal case

Decision trees

Splitting criterion: Gini index

In order to find the **best binary split**, the weighted sum of the Gini indices $Gini(D_j)$ resulting from the **two partitions** is calculated:

$$Gini_{split\ A}(D) = \frac{|D_1|}{|D|} * Gini(D_1) + \frac{|D_2|}{|D|} * Gini(D_2)$$

The reduction of impurity is then calculated by:

$$\Delta Gini = Gini(D) - Gini_{split\ A}(D)$$

- ▶ All possible split candidates are tested, and the one is selected where $\Delta Gini$ is maximum.

Decision trees

Splitting criterion: Gini index

Exercise: „Gini index “

Calculation of Gini index for the following two-class problem with categorical attributes

- ▶ number of feature vectors: 14
- ▶ two classes: P and N
- ▶ 9 feature vectors of class P, 5 of class N

1. Gini of D :

Gini creates binary splits, so let's define a split on the feature „Outlook“ as
(1: sunny ; 2:rain or overcast)

$$\text{▶ } p_P = \frac{9}{14} \quad ; \quad p_N = \frac{5}{14}$$

$$\text{▶ } Gini(D) = 1 - \sum_{i=1}^{|C|} p_i^2 = 1 - \left(\frac{9}{14}\right)^2 - \left(\frac{5}{14}\right)^2 = 0.46$$

2. Gini of partitions

$$Gini_{split \text{ Outlook}}(D) = \frac{|D_1|}{|D|} * Gini(D_1) + \frac{|D_2|}{|D|} * Gini(D_2) = \dots$$

3. Calculate the split's improvement of Gini

$$\Delta Gini = Gini(D) - Gini_{split A}(D)$$

4. Repeat this for other split options on this feature and for the features „Temperature“ and „Humidity“ and find the best split.

No.	Attributes			Class
	Outlook	Temperature	Humidity	
1	sunny	hot	high	N
2	sunny	hot	high	N
3	overcast	hot	high	P
4	rain	mild	high	P
5	rain	cool	normal	P
6	rain	cool	normal	N
7	overcast	cool	normal	P
8	sunny	mild	high	N
9	sunny	cool	normal	P
10	rain	mild	normal	P
11	sunny	mild	normal	P
12	overcast	mild	high	P
13	overcast	hot	normal	P
14	rain	mild	high	N

adapted from original ID3 paper: (Quinlan, 1986)

Decision trees

Splitting criteria

- ▶ in addition to Gini and entropy, there are more splitting criteria.
- ▶ Gini and entropy are the most common ones.
- ▶ the splitting criterion can be viewed as a hyperparameter, so various can be tested
- ▶ results with Gini and entropy are often not very different, if results are different entropy tends to yield more balanced trees (Geron, 2019)
- ▶ computation of Gini is faster
- ▶ in the python-library scikit-learn, Gini is the default

Decision trees

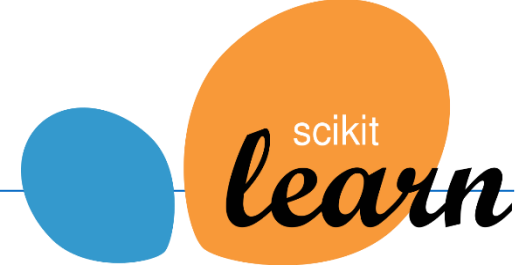
scikit-learn



- ▶ <https://scikit-learn.org>
- ▶ In the python library **scikit-learn (sklearn)** a decision tree is available with the class `DecisionTreeClassifier`
- ▶ uses CART (Classification and Regression Trees) (Breiman, 1984)
- ▶ some important parameters:
 - ▶ `max_depth`, allowing to prune the tree
 - ▶ `criterion`, splitting criterion: Gini index is the default, entropy can be used

Decision trees

scikit-learn



```
1 # imports
2 # own file containing the code with the created data sets
3 from my_datasets import *
4
5 # own file containing code to plot decision functions in a 2D plot
6 from my_plotting import my_plotDecisionFunction
7
8 import numpy as np
9 import matplotlib.pyplot as plt
10
11 # sklearn imports
12 from sklearn.model_selection import train_test_split
13 from sklearn.tree import DecisionTreeClassifier
14 from sklearn.metrics import confusion_matrix, accuracy_score
```

Decision tree

```
1 # own data set
2 data, labels = createData_XOR()
```

```
1 # split data set into train and test set ("hold-out")
2 # (random_state just used for reproducible plots!)
3 # param test_size allows to specify the percentage (0..1) of the test set
4 train_data, test_data, train_labels, test_labels = train_test_split(
5     data, labels, test_size = 0.5, random_state=123)
```

Training

```
1 # common machine learning models are included in sklearn as classes
2 # here the decision tree classifier is used
3 model = DecisionTreeClassifier()
```

```
1 # train model on training set
2 model.fit(train_data, train_labels)
```

```
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')
```

Test

```
1 # classification of test set
2 predictions = model.predict(test_data)
3
4 print("### Results on test set: ###")
5
6 acc = accuracy_score(test_labels, predictions)
7 print("Overall accuracy: ", acc)
8
9 print("Confusion matrix")
10 cm = confusion_matrix(test_labels, predictions)
11 print(cm)
```

```
### Results on test set: ###
Overall accuracy:  1.0
Confusion matrix
[[102  0]
 [ 0 98]]
```

Decision trees

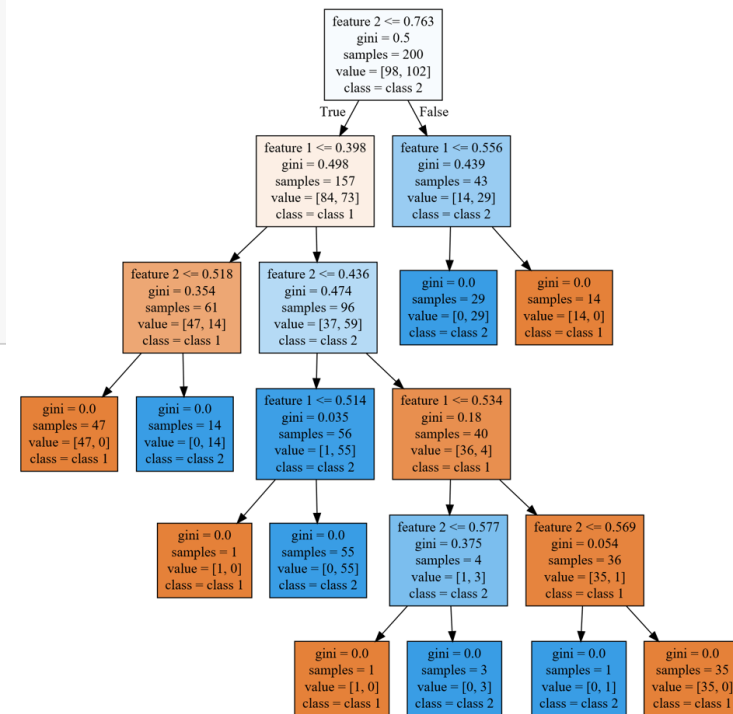
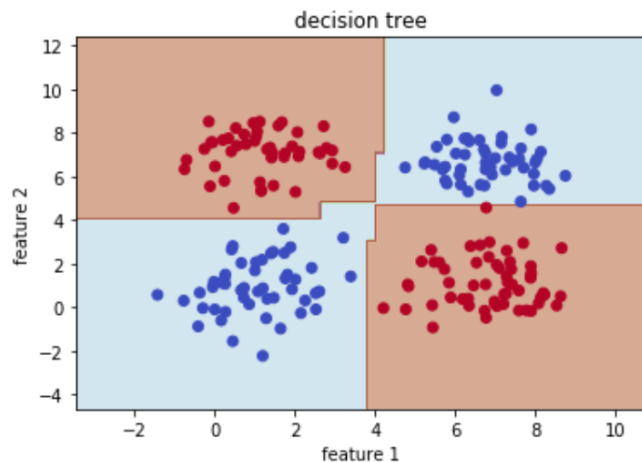
scikit-learn



Decision tree with Python and scikit-learn

Decision tree on XOR data set

```
1 data, labels = createData_XOR()
2
3 # split data set into train and test set
4 train_data, test_data, train_labels, test_labels = train_test_split(
5     data, labels, test_size = 0.5, random_state=123)
6
7 # training
8 model = DecisionTreeClassifier()
9 model.fit(train_data, train_labels)
10
11 # plot decision function that was learned from training set
12 my_plotDecisionFunction(train_data, train_labels, model, title = "decision tree")
13
14 # classification of test set
15 predictions = model.predict(test_data)
16 cm = confusion_matrix(test_labels, predictions)
17
```



Decision trees

Evaluation

advantages +

- interpretability: results are human-readable, which is a benefit for many domains (e.g. medical systems)
- works with numerical and categorical data
- no scaling of input data required
- can return feature importance
- can also be used for regression (regression trees)

disadvantages -

- decision boundaries are not very flexible (piecewise-linear), they evaluate one feature at a time
- at each step the locally optimal decision is made, does not necessarily lead to a globally optimal solution
- a tree may overfit the data (pruning should be used)
- slightly different data (e.g. by randomly splitting train and test set), may lead to completely different trees

Random forests (Breiman, 2001)

The idea:

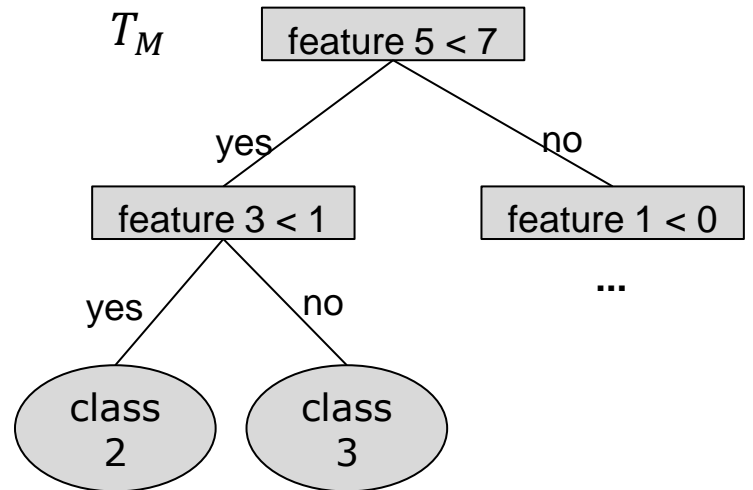
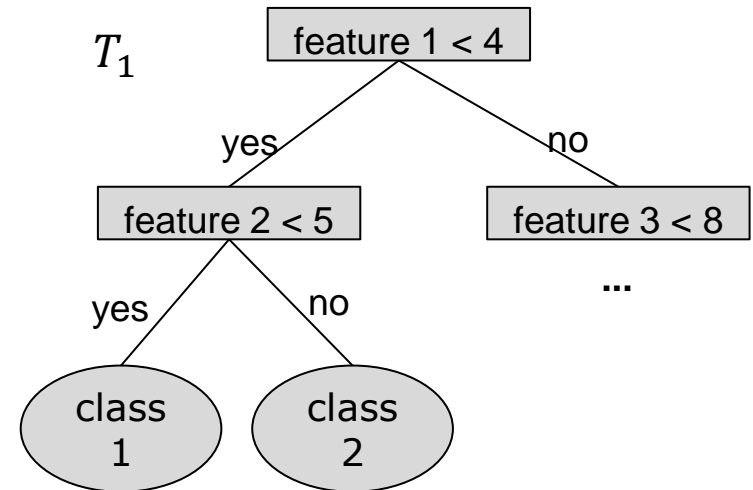
Create many decision trees and combine the result with majority voting.

Assumption: the result of a combination of different classifiers is likely to be better than of a single classifier.

A random forest is an **ensemble method**, i.e. a random forest is an ensemble of trees.

The algorithm:

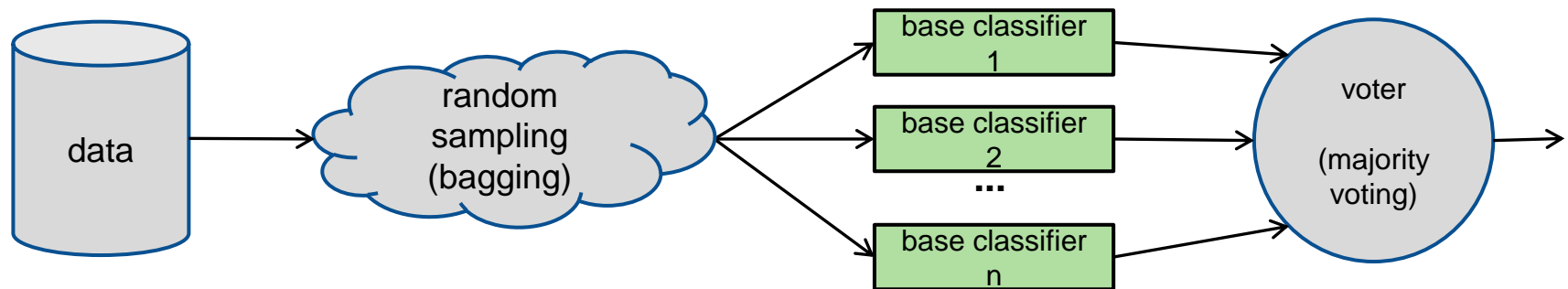
1. create many different decision trees $T_1 \dots T_M$
2. the different trees are created by **randomly subsampling** the features and the data set **at each node of each tree**
3. a feature vector in the test set is classified by each tree T_i
4. the classification result is the most frequent result from $T_1 \dots T_M$ (majority voting)



Random forests

A random forest is an **ensemble** of randomly created, different decision trees (base classifiers).

- ▶ the number of trees is a hyperparameter (e.g. 100)



Random forests

An ensemble performs best, if the base classifiers are diverse and unrelated.

In a random forest this is achieved by:

1. **bagging**: training each of the decision trees with a different subset of the data

- ▶ bootstrap aggregation (bagging) is typically used for this
- ▶ in bagging, from a data set D consisting of N feature vectors, N of these feature vectors are **randomly drawn with replacement** and become the training set A
- ▶ i.e. a feature vector can be drawn multiple times
- ▶ the feature vectors not drawn become the test set B
- ▶ it can be shown, that on average 63.2% of the feature vectors are in the training set

2. **feature subsampling**: randomly selecting subsets of features at each node of each tree

- ▶ for example for $|F|$ features, $\sqrt{|F|}$ can be selected as candidates at each node
- ▶ using a splitting criterion, the best split is determined from this subset

Random forests

Evaluation

advantages +

- avoids overfitting
- reduces the variance
- often better results compared to single trees
- works with numerical and categorical data
- no scaling of input data required
- works for large, high-dimensional data sets, since at each split only a subset is tested
- can return feature importance
- can also be used for regression

disadvantages -

- in contrast to single decision trees, a random forest is not (easily) interpretable
- computationally expensive, compared to single tree



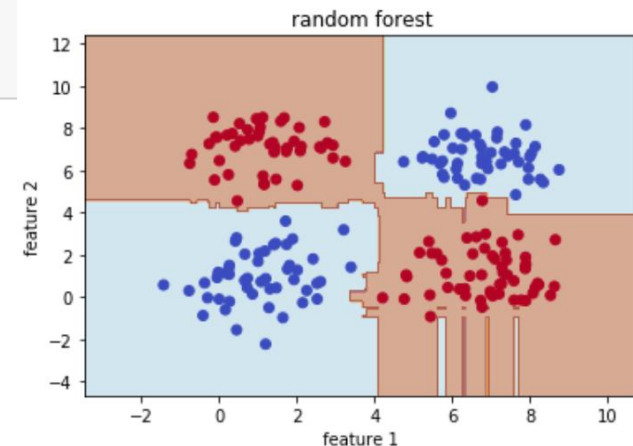
<https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier>

- ▶ In **scikit-learn (sklearn)** a random forest is available with the class `RandomForestClassifier`
- ▶ some important parameters:
 - ▶ `n_estimators`, determining the number of decision trees to be created
 - ▶ `max_depth`, allowing to prune the tree
 - ▶ `max_features`, number of features to consider at each split
- ▶ current defaults are: `n_estimators=100`, `criterion='gini'`, `max_depth=None`

Random forest

```
1 from sklearn.ensemble import RandomForestClassifier
2
3 data, labels = createData_XOR()
4
5 # split data set into train and test set
6 train_data, test_data, train_labels, test_labels = train_test_split(
7     data, labels, test_size = 0.5, random_state=123)
8
9 # training
10 model = RandomForestClassifier()
11 model.fit(train_data, train_labels)
12
13 print(model)
14
15 # plot decision function that was learned from training set
16 my_plotDecisionFunction(train_data, train_labels, model, title = "random forest")
17
18 # classification of test set
19 predictions = model.predict(test_data)
20 cm = confusion_matrix(test_labels, predictions)
21 print(cm)
```

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
    max_depth=None, max_features='auto', 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, n_estimators=10, n_jobs=1,
    oob_score=False, random_state=None, verbose=0,
    warm_start=False)
```



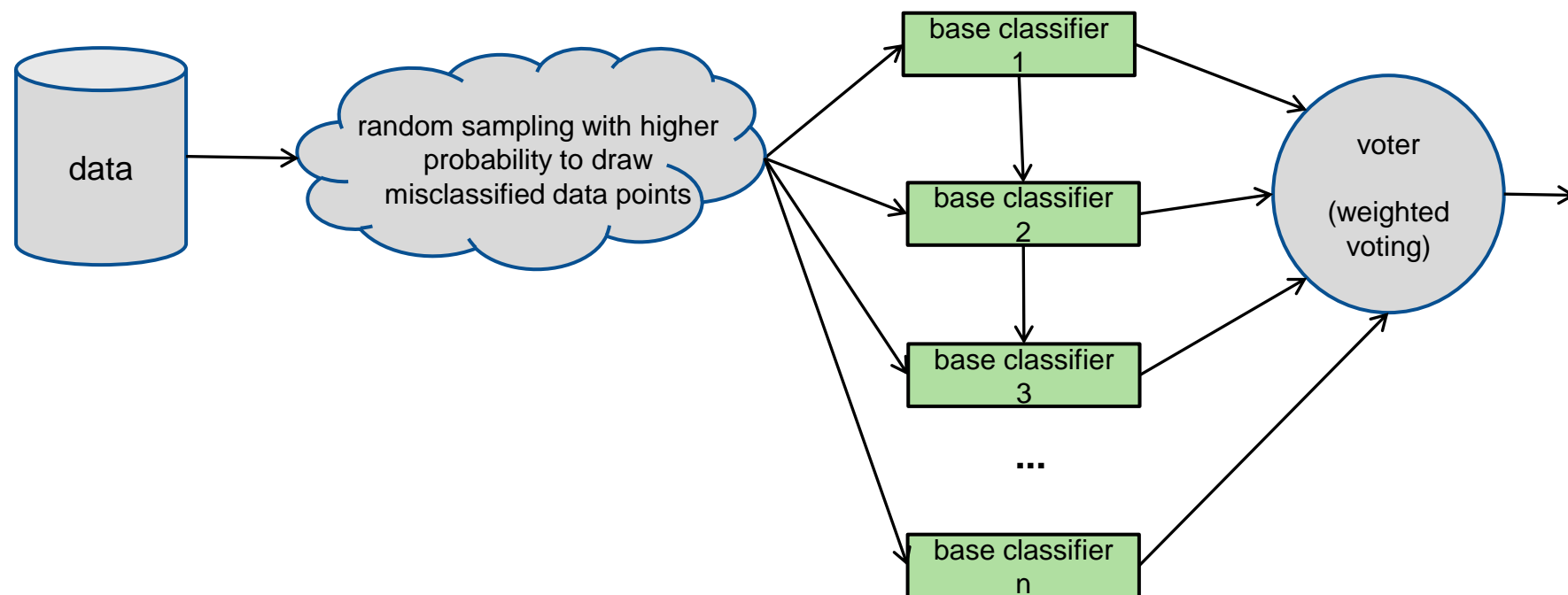
Gradient boosting machines

- ▶ **Gradient boosting machines** (GBM) refers to a family of methods of ensembles of trees using boosting
- ▶ GBMs are also referred to as **gradient tree boosting**
- ▶ an early boosting algorithm with decision trees is **AdaBoost** (Adaptive Boosting)
- ▶ later generalized and then termed **Gradient Boosting Machines** (Friedmann, 2000)
- ▶ a widely used scalable implementation is **XGBoost** (Extreme Gradient Boosting) (Chen and Guestrin, 2016)

Gradient boosting machines

Boosting

- ▶ boosting uses an ensemble of subsequent base classifiers (weak learners)
- ▶ the idea of boosting is, that each base classifier k tries to correctly classify those feature vectors misclassified by base classifier $k-1$
- ▶ this is achieved by random sampling, with increased probability of drawing a previously misclassified feature vector (weights are iteratively assigned to feature vectors)
- ▶ the final results is a combination of each classifiers' results, weighted by its accuracy
- ▶ early algorithms are AdaBoost and AdaBoost.M1



Gradient boosting machines

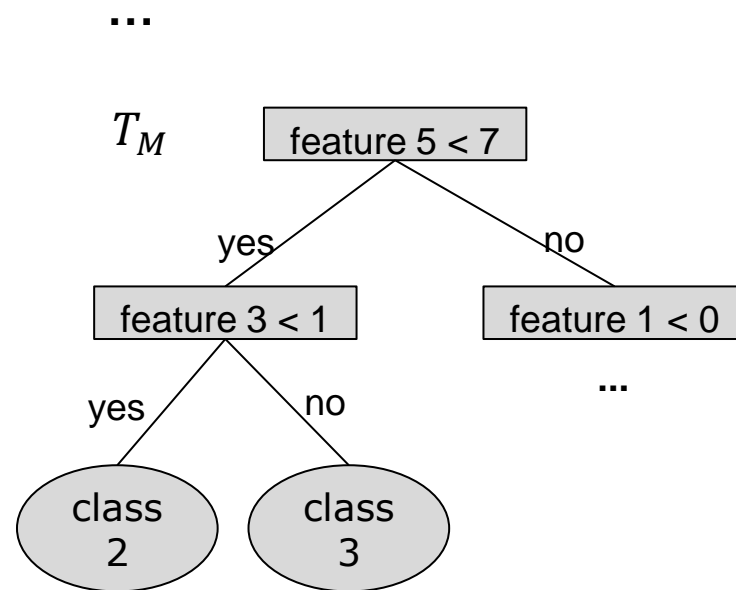
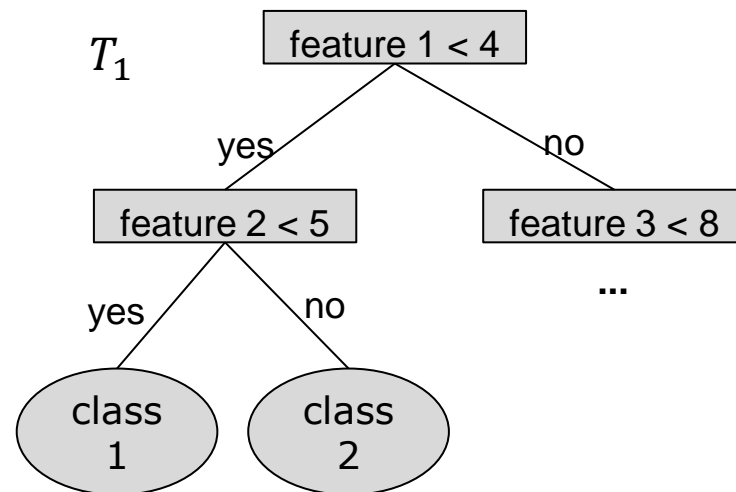
XGBoost (Extreme Gradient Boosting) (Chen and Guestrin, 2016)

The idea:

- ▶ Create many sequential decision trees $T_1 \dots T_M$, where each new tree T_i tries to minimize the errors of the previous tree T_{i-1} .
- ▶ Combine the results with weighted voting.
- ▶ XGBoost is an **ensemble method** consisting of sequential trees.

The algorithm:

1. add one tree T_i per step
2. the new tree is found such that the error of the previous tree T_{i-1} is minimized
3. goto 1. until some stopping criterion is reached (e.g. number of steps)
4. the classification result is found by using the results of $T_1 \dots T_M$, weighted by their accuracies
 - ▶ i.e. the votes of better trees have higher weights



Gradient boosting machines

XGBoost (Extreme Gradient Boosting)

GBMs and in specific XGBoost use a special form of boosting, referred to as **gradient boosting**

- ▶ one tree T_i is added in each step
- ▶ regression trees are used (i.e. continuous outputs), allowing to sum up the subsequent outputs
- ▶ the new tree T_i is selected such that an objective function obj is minimized

- ▶ general form of objective function:

$$obj = loss + regularisation$$

with:

- ▶ *loss*: some kind of error function expressing the error between predicted values and true values, i.e. $loss(predictions, labels)$
 - ▶ e.g. sum of squared errors
- ▶ *regularisation* : a term controlling the model T_i , i.e. $regularisation(T_i)$, e.g. to avoid overfitting
 - ▶ e.g. model complexity, like number of leaves
- ▶ minimizing obj is achieved by the trade-off of minimizing *loss*, while keeping *regularisation* minimal
- ▶ achieved with a gradient descent approach, referred to as **functional gradient descent**
 - ▶ (beyond scope, see e.g. (Friedman,2000) or (Chen and Guestrin, 2016))

XGBoost

Evaluation

advantages +

- avoids overfitting
- usually better results compared to single trees
- works with numerical and categorical data
- can handle missing values
- no scaling of input data required
- works for large, high-dimensional data sets, since at each split only a subset is tested
- can return feature importance

disadvantages -

- in contrast to single decision trees, a random forest is not (easily) interpretable
- computationally expensive compared to single trees, however XGBoost was implemented towards efficiency and scalability

XGBoost

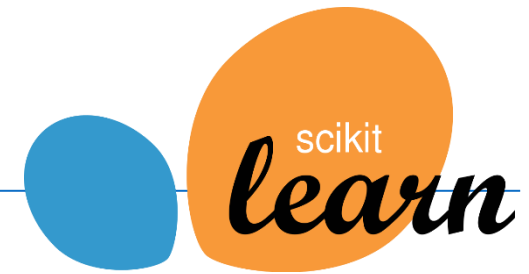
Python module xgboost

https://xgboost.readthedocs.io/en/latest/python/python_intro.html

- ▶ XGBoost is not contained scikit-learn (sklearn), module **xgboost** required
- ▶ however it can be combined with scikit-learn
- ▶ some important parameters:
 - ▶ `n_estimators`, determining the number of decision trees to be created
 - ▶ `max_depth`, allowing to prune the tree
 - ▶ `reg_lambda`, regularisation parameter controlling the trade-off between loss and reg. term
 - ▶ `learning_rate`, also referred to as η . controls the contribution of each new tree (0,1)

XGBoost

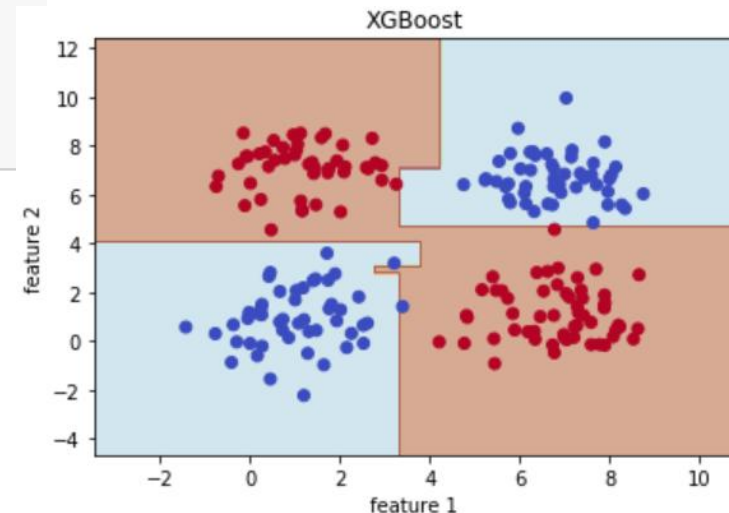
Python module xgboost + scikit-learn



XGBoost

```
1 # requires additional package xgboost,
2 #currently (02/2020) to be installed with: pip3 install xgboost
3 from xgboost import XGBClassifier
4
5 # own data set, not included in scikit-learn
6 data, labels = createData_XOR()
7
8 # split data set into train and test set
9 train_data, test_data, train_labels, test_labels = train_test_split(
10     data, labels, test_size = 0.5, random_state=123)
11
12 # training
13 model = XGBClassifier()
14 model.fit(train_data, train_labels)
15 print(model)
16
17 # plot decision function that was learned from training set (own function!)
18 my_plotDecisionFunction(train_data, train_labels, model, title = "XGBoost")
19
20 # classification of test set
21 predictions = model.predict(test_data)
22 cm = confusion_matrix(test_labels, predictions)
23 print(cm)
```

```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, gamma=0, learning_rate=0.1,
              max_delta_step=0, max_depth=3, min_child_weight=1, missing=None,
              n_estimators=100, n_jobs=1, nthread=None,
              objective='binary:logistic', random_state=0, reg_alpha=0,
              reg_lambda=1, scale_pos_weight=1, seed=None, silent=None,
              subsample=1, verbosity=1)
```



Support vector machines (SVM)

Introduction

Notation

x_i : one dimension in the feature space, i.e. one feature

\vec{x}_i : one feature vector (one instance)

\vec{x} : the feature space

$\vec{x}_i \vec{x}_j$: inner product, dot product, scalar product of \vec{x}_i and \vec{x}_j

N : number of instances

n : number of features

\vec{w} : vector of weights

$||\vec{w}||$: norm of the vector \vec{w}

ω_i : class i

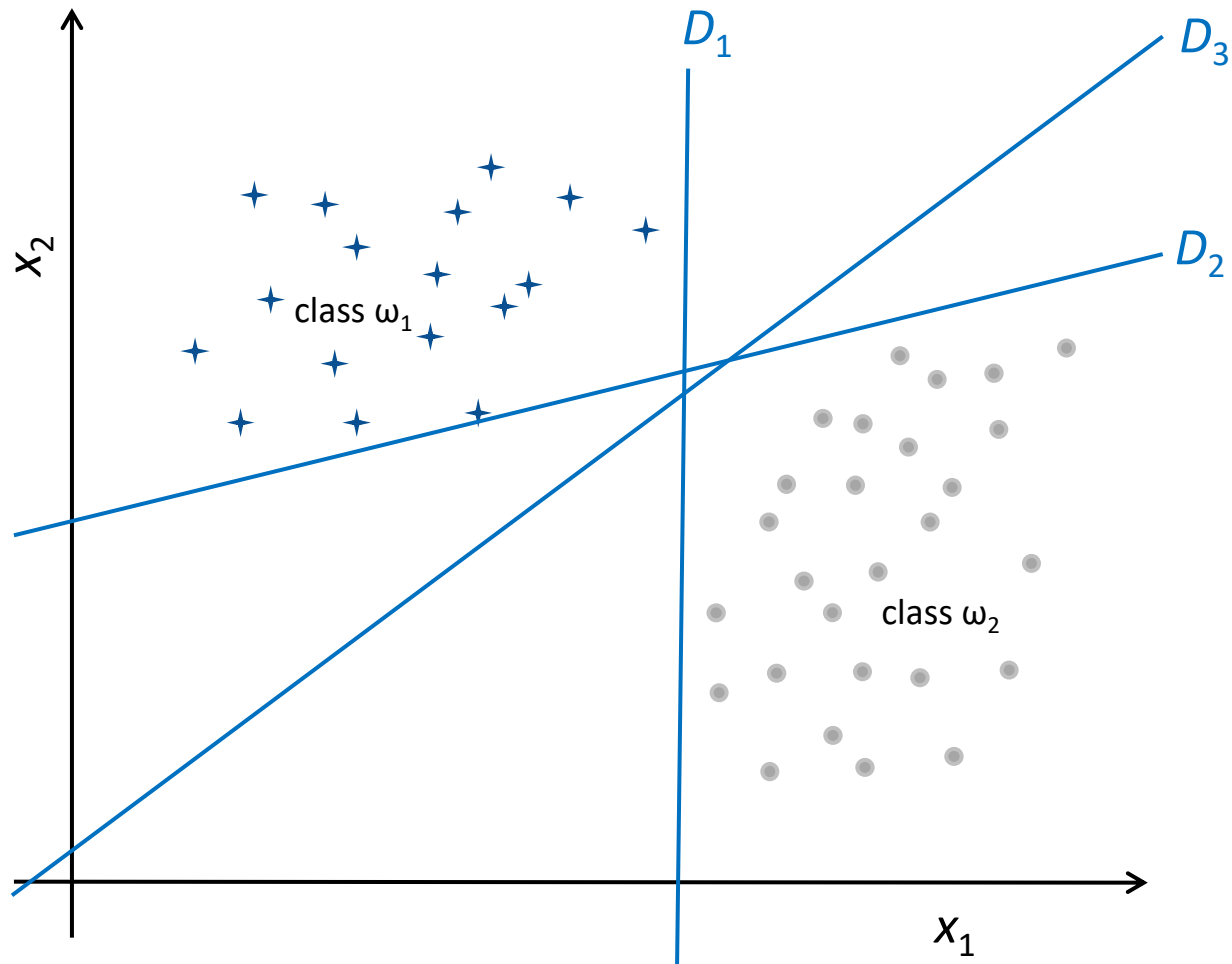
y_i : label (+1 or -1)

We use arrows to denote vectors, in contrast to most of the literature on data mining, where vectors are written without arrows.

Support vector machines (SVM)

Introduction

Separating two classes with linear decision functions



The number of possible decision functions is infinite.

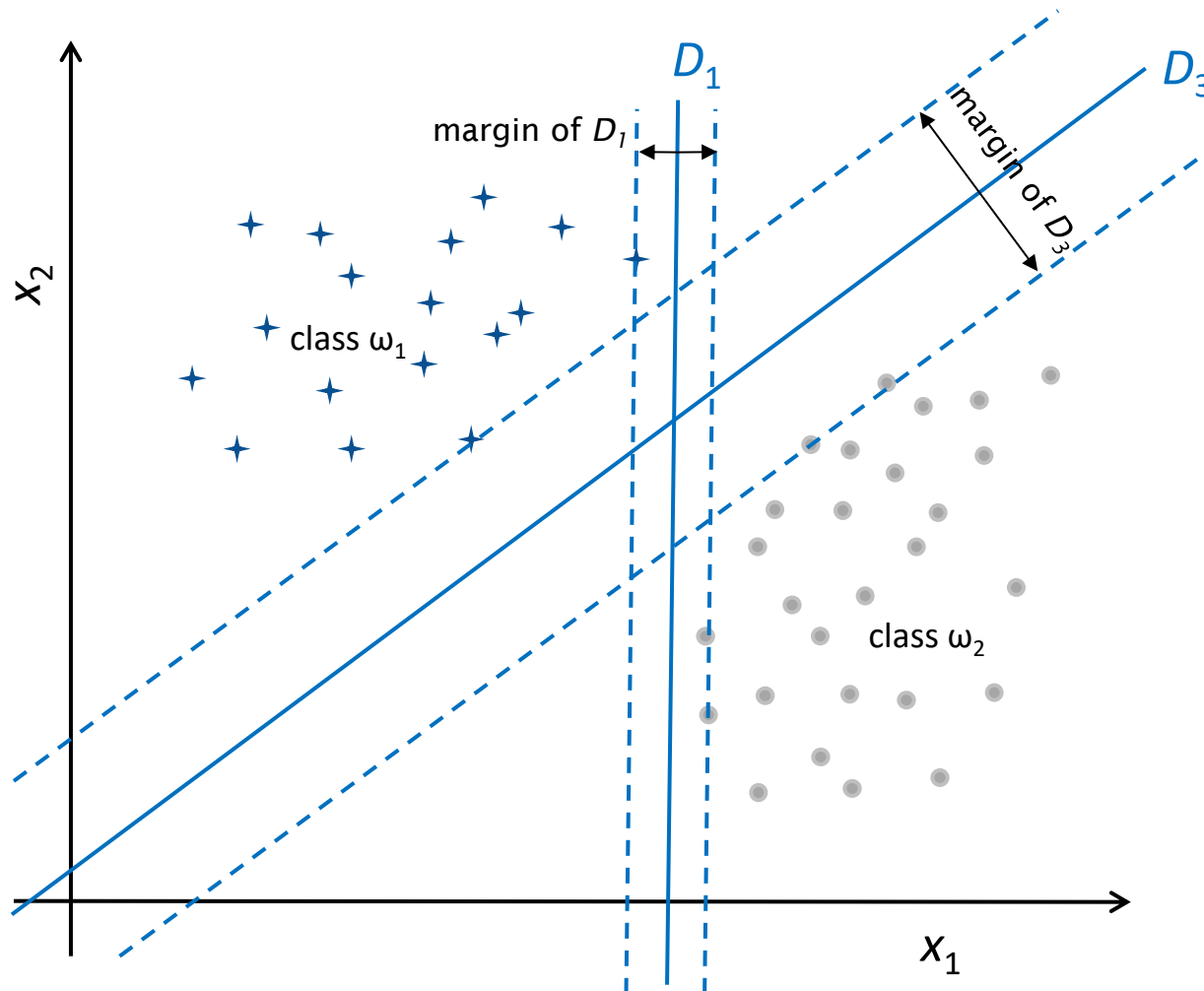
So which one should be chosen?

Intuitively D_3 seems to be the best, but why?

Support vector machines (SVM)

Introduction

Finding the optimal linear decision function



Based on the informal explanation, let us compare the two candidates D_1 and D_3

Our intuition was correct:

- D_3 has the maximal margin, it is much larger than the margin of D_1
- D_3 is the optimal decision function

Support vector machines (SVM)

Introduction

Support vector machines for linear separable data

the idea:

separate the instances of two classes using a linear decision function referred to as “hyperplane”

how it works: (informal explanation for a two-dimensional space)

1. find two parallel lines, one intersecting one or more instances at the boundary of class ω_1 and the other line intersecting one or more instances of class ω_2
2. find the line, with equal distance to each of the two parallel lines (the line „in the middle“)
3. measure the distance between the two outer lines
 - this distance is referred to as the „margin“
 - the optimal decision function is the one with the maximum margin
4. the decision function is expressed using instances from the training set, **the so-called support vectors**

Support vector machines (SVM)

Expressing the decision function

Expressing the decision function in two-dimensional space

A linear function in a two-dimensional space can be expressed by the well-known „slope-intercept-form“:

$$y = mx + b$$

The $y(x)$ -form is constrained to two dimensions. To make the formula more generic we use the dimensions x_1 and x_2 and w_1 instead of m :

$$x_2 = w_1 x_1 + b$$

which can be reformulated as

$$w_1 x_1 + w_2 x_2 + b = 0$$

Substituting the scalars with the vectors $\vec{w} = \begin{pmatrix} w_1 \\ w_2 \end{pmatrix}$ and $\vec{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$

leads to the **general form of the linear decision function**:

$$\vec{w} \vec{x} + b = 0$$

Support vector machines (SVM)

Expressing the decision function

Expressing the decision function using vectors

Up to now we have looked at a two-dimensional space (2 features), where the linear decision function is simply a line.

In a three-dimensional space (3 features) the line becomes a plane and in higher-dimensional spaces (>3) it is referred to as a „**hyperplane**“.

We will use the term hyperplane, independent of the number of dimensions.

The hyperplane is expressed as

$$\vec{w} \vec{x} + b = 0$$

or
$$w_1 x_1 + w_2 x_2 + \dots + w_n x_n + b = 0$$

with
$$\vec{w} = \begin{pmatrix} w_1 \\ \dots \\ w_n \end{pmatrix} \quad \text{and} \quad \vec{x} = \begin{pmatrix} x_1 \\ \dots \\ x_n \end{pmatrix}$$

where n is the number of features. (For all data points \vec{x} becomes a matrix)

Support vector machines (SVM)

Expressing the decision function

Using the decision function to classify instances

Instead of class labels ω_1 and ω_2 we will use +1/-1 to label the classes:

$$y_i = +1 \quad \text{if } \vec{x}_i \text{ is } \omega_1 \quad \text{and} \quad y_i = -1 \quad \text{if } \vec{x}_i \text{ is } \omega_2$$

Classification of a feature vector \vec{x}_i is done using the sign-function:

$$D(\vec{x}_i) = \text{sign}(\vec{w} \vec{x}_i + b)$$

i.e.:

$$\begin{aligned} D(\vec{x}_i) &= +1 & \text{if } \vec{w} \vec{x}_i + b > 0 \\ D(\vec{x}_i) &= -1 & \text{if } \vec{w} \vec{x}_i + b < 0 \end{aligned}$$

Please note: $w \neq \omega$ (ω refers to the classes and w to the so-called weights, in accordance with the common notation used in literature)

Support vector machines (SVM)

Finding the optimal decision function

Having done some math... the optimization problem is given by

$$\text{minimize} \quad \frac{1}{2} \|\vec{w}\|^2 \quad (\text{a})$$

$$\text{subject to} \quad y_i (\vec{w} \cdot \vec{x}_i + b) \geq 1 \quad \text{for all } i \quad (\text{b})$$

Using the „method of Lagrange“, we can incorporate (b) into (a).

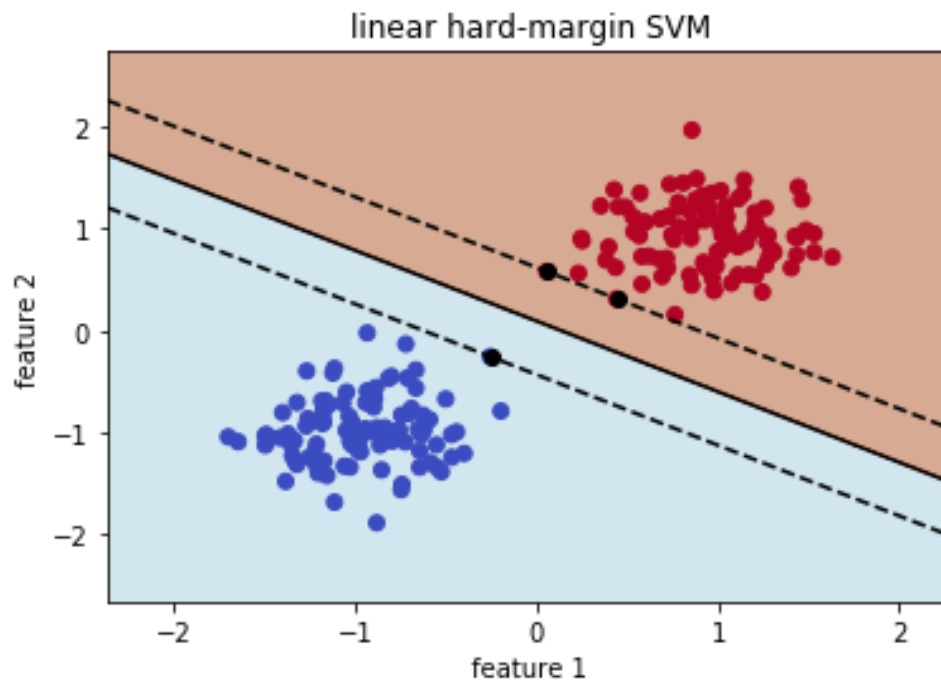
Deriving the new equation and setting it to 0 yields the following optimization problem:

$$\begin{aligned} \text{maximize } L(\vec{\alpha}) &= \sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i,j=1}^N \alpha_i \alpha_j y_i y_j \vec{x}_i \cdot \vec{x}_j \\ \text{subject to } \sum_{i=1}^N \alpha_i y_i &= 0 \\ \alpha_i &\geq 0 \quad \text{for all } i \end{aligned}$$

where α_i and α_j are the so-called Lagrange multipliers.

Support vector machines (SVM)

Finding the optimal decision function



Support vector machines (SVM)

Finding the optimal decision function

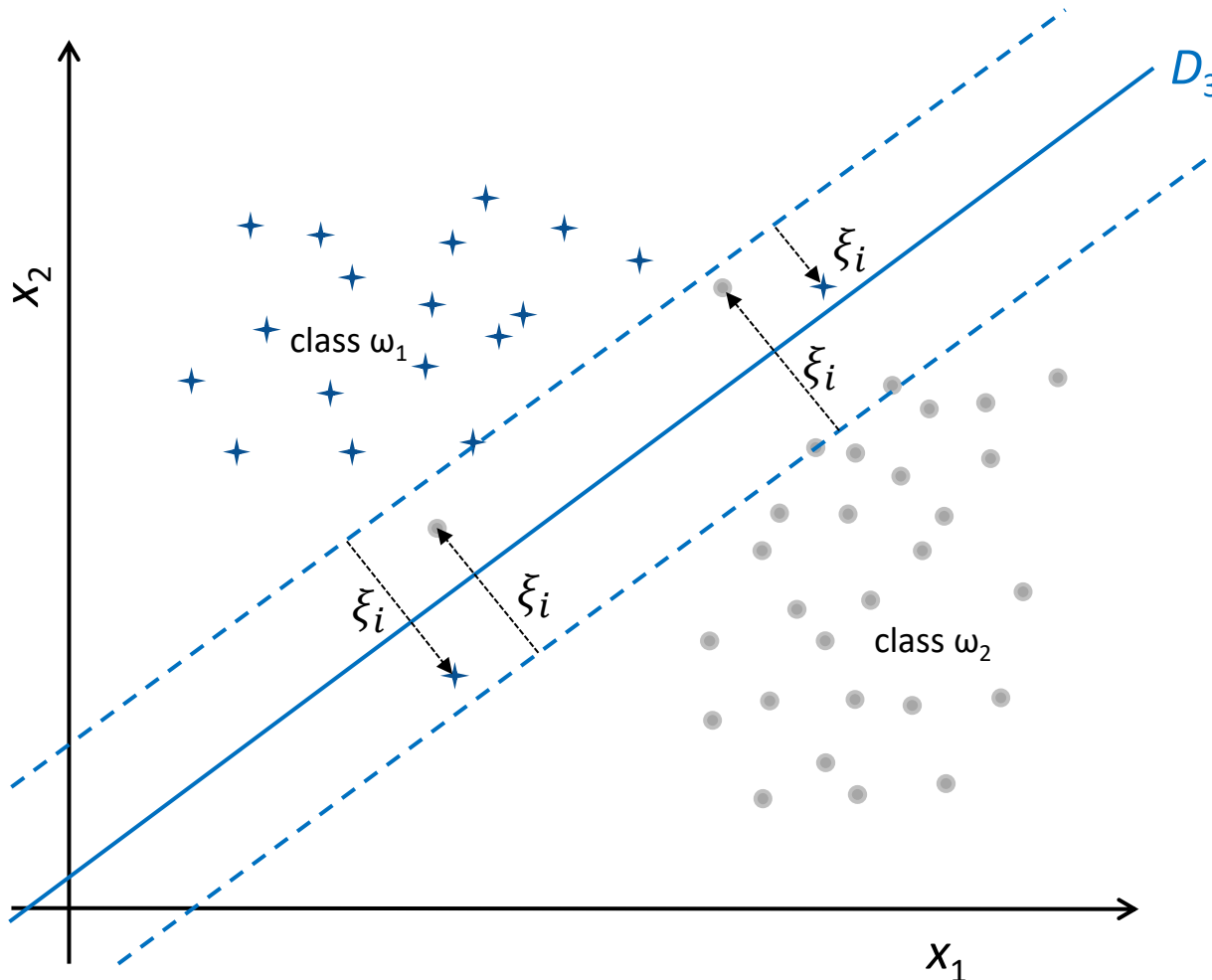
Summary: SVM for linearly separable classes

- SVM finds the optimal linear decision function:
the „maximum margin hyperplane“
- the found hyperplane is guaranteed to be the optimal solution, there are no local minima
- the decision function is expressed using instances \vec{x}_i from the training set
 - the so-called „**support vectors**“
- this type of SVM is called „**hard-margin SVM**“ and can be used if the classes are linearly separable

Support vector machines (SVM)

Soft-margin SVM

What if the data is not fully linearly separable?



A hard-margin SVM cannot find a decision function if the classes are not linearly separable.

Solution:

- allow some instances to be on the opposite side of the supporting hyperplanes
- penalize those instances by introducing so-called slack variables ξ_i :
 - $\xi_i > 0$ if the instance \vec{x}_i is not within the boundary
 - $\xi_i = 0$ otherwise

This type of SVM is called „**soft-margin SVM**“

Support vector machines (SVM)

Soft-margin SVM

Optimization problem for the soft-margin SVM

$$\text{minimize} \quad \frac{1}{2} \|\vec{w}\|^2 + C \sum_{i=1}^N \xi_i$$

$$\text{subject to} \quad y_i(\vec{w} \cdot \vec{x}_i + b) \geq 1 - \xi_i \quad \text{for all } i$$

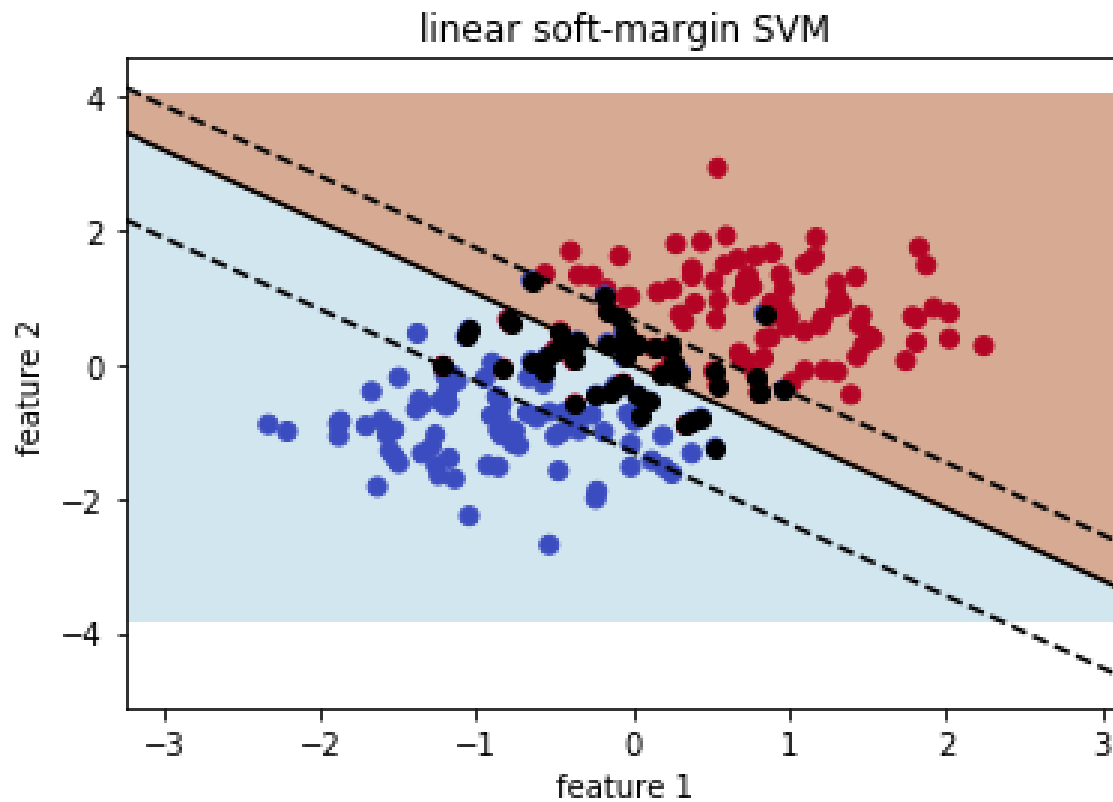
where C is a regularization parameter controlling how many and how far instances may lie outside of the supporting hyperplanes.

C is a hyperparameter, regularizing the influence of the slack variables

Again, using the method of Lagrange, the optimization problem can be reformulated.

Support vector machines (SVM)

Soft-margin SVM



Support vector machines (SVM)

Non-linear decision functions

The soft-margin SVM works fine, if individual instances prevent the SVM from finding a linear decision function.

If the entire data set is not linearly separable, there is a better solution:

A data set that is not linearly separable in the given feature space

$$\mathbb{R}^N$$

can be linearly separated in a higher-dimensional space

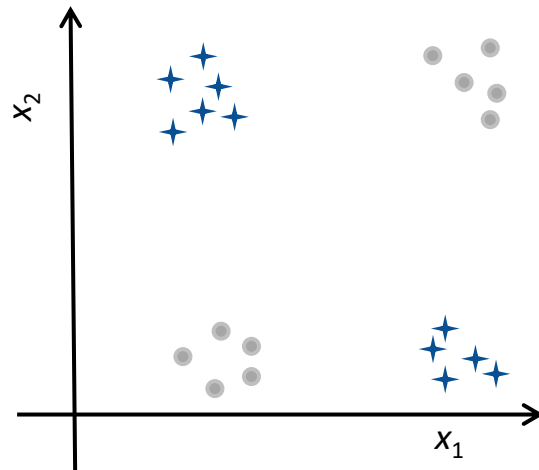
$$\mathbb{R}^M$$

$$\mathbb{R}^N \rightarrow \mathbb{R}^M \quad \text{where } M > N$$

Support vector machines (SVM)

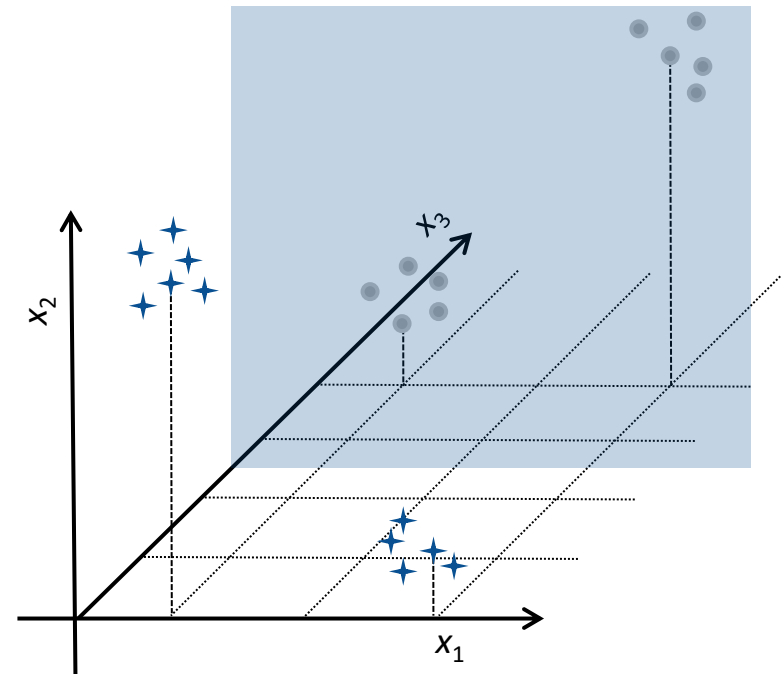
Non-linear decision functions

Example: the XOR-problem



$$R^2 \rightarrow R^3$$

● class ω_1
★ class ω_2



If we

1. add a third dimension x_3 and
 2. „somehow“ shift the instances of class ω_1 on that dimension
- we can linearly separate the two classes in the new feature space R^3

Support vector machines (SVM)

Non-linear decision functions

Mapping to higher-dimensional space using a mapping function

Let us consider a mapping function

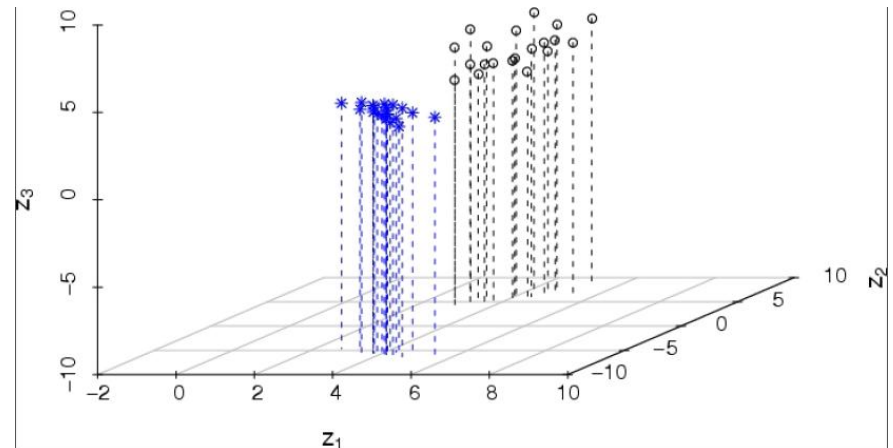
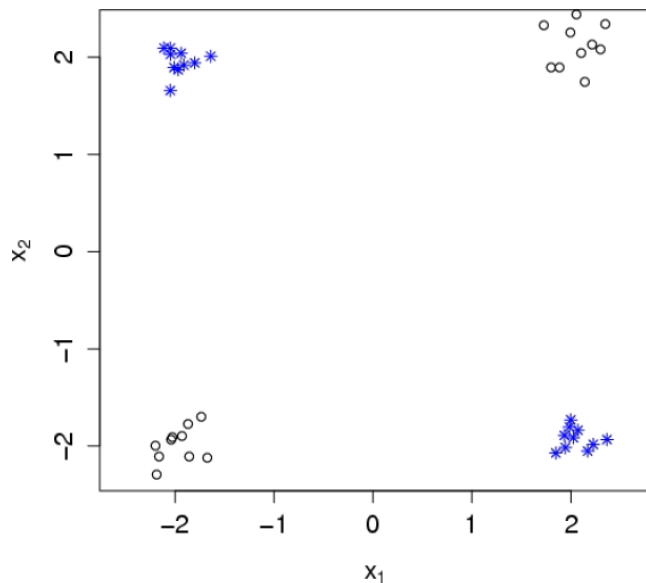
$$Z := \phi(\vec{x})$$

which maps $R^2 \rightarrow R^3$

Using the mapping

$$\begin{aligned} z_1 &:= x_1^2 \\ z_2 &:= \sqrt{2}x_1x_2 \\ z_3 &:= x_2^2 \end{aligned}$$

the classes can be linearly separated.



Support vector machines (SVM)

Non-linear decision functions

For the mapping, we need to map each instance \vec{x}_i to an instance \vec{z}_i in the new feature space using $\vec{z}_i = \phi(\vec{x}_i)$.

Incorporating $\phi()$ into the optimization problem yields

$$L(\vec{\alpha}) = \sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i,j=1}^N \alpha_i \alpha_j y_i y_j \phi(\vec{x}_i) \phi(\vec{x}_j)$$

Problem:

Obviously we cannot just use any mapping function. In the two previous examples, the mapping functions were ideal for the data set. They nicely separated the two classes.

If we need to find specific mapping functions for each data set, the approach would be infeasible.

Support vector machines (SVM)

Non-linear decision functions

The kernel trick

Looking at the optimization problem, we notice that \vec{x}_i and \vec{x}_j are represented by the inner product as $\phi(\vec{x}_i)\phi(\vec{x}_j)$

Now we use the so-called „**kernel trick**“:

Instead of actually doing the mapping using $\phi()$, we replace the inner product by

$$K(\vec{x}_i, \vec{x}_j) = \phi(\vec{x}_i)\phi(\vec{x}_j)$$

which leads to the following in the Lagrange-transformed optimization problem:

$$L(\vec{\alpha}) = \sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i,j=1}^N \alpha_i \alpha_j y_i y_j K(\vec{x}_i, \vec{x}_j)$$

Now we use a **kernel function** $K(\vec{x}_i, \vec{x}_j)$ that returns a value for each pair of \vec{x}_i and \vec{x}_j has a parameter that can be tuned during training.

Support vector machines (SVM)

Non-linear decision functions

Kernel functions

Two widely used kernel functions are

- polynomial kernel:

$$K(\vec{x}_i, \vec{x}_j) = (\vec{x}_i \vec{x}_j + r)^d$$

sometimes with scaling parameter γ : $K(\vec{x}_i, \vec{x}_j) = (\gamma \vec{x}_i \vec{x}_j + r)^d$

- radial basis function kernel (Gaussian):

$$K(\vec{x}_i, \vec{x}_j) = e^{-\frac{\|\vec{x}_i - \vec{x}_j\|^2}{2\sigma^2}} = e^{-\gamma \|\vec{x}_i - \vec{x}_j\|^2} \quad \text{where } \gamma = \frac{1}{2\sigma^2}$$

- For non-linear problems one typically starts with the radial basis function (RBF) kernel.

the kernel parameter is a hyperparameter

Support vector machines (SVM)

Non-linear decision functions



```
# split into training and test set
train_data, test_data, train_labels, test_labels = train_test_split(
    data, labels, test_size = 0.5, random_state=123)

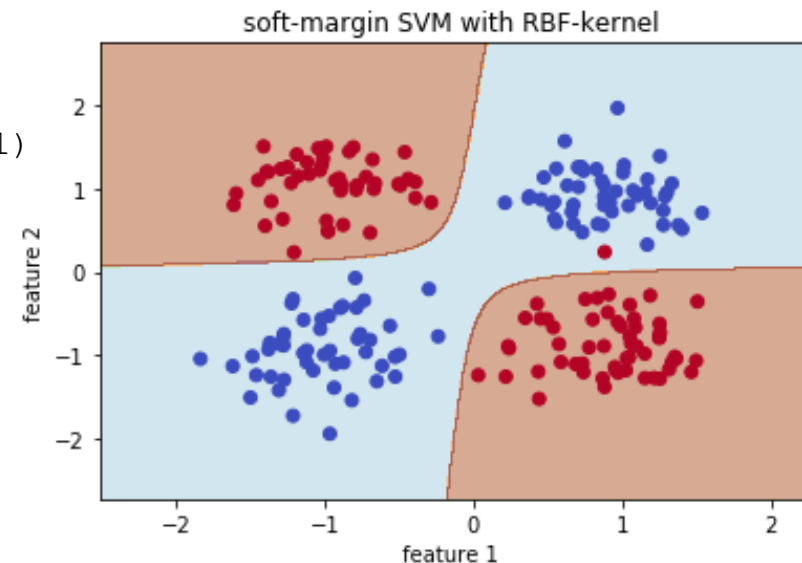
# z-score scaling: determine scaling parameters
scaler = StandardScaler().fit(train_data)

# scale train set and test set
train_data = scaler.transform(train_data)
test_data = scaler.transform(test_data)

# soft-margin SVM with RBF kernel
# gamma="auto": SVC tries to set a good value
model = SVC(kernel = "rbf", gamma = "auto", C = 1)

# train model on training set
model.fit(train_data, train_labels)

# classification of test set
predictions = model.predict(test_data)
```



Support vector machines (SVM)

Non-linear decision functions

Training and test of soft-margin SVMs with kernels

Usually soft-margin SVMs with non-linear kernels like the RBF kernel are used.

During training the parameter C and the kernel parameter (σ for the RBF kernel) are tuned so that the training data is separated with a low error rate.

Looking at classification results in the original feature space, it can be seen that this type of SVM finds non-linear decision functions.

Support vector machines (SVM)

Support vector machines

advantages +

- robustness: robust against noise in the training set if soft-margin SVMs are used
- highly-flexible decision boundary is kernel trick is used
- global optimum is found (for a given set of hyperparameters)
- no random parts involved, in contrast to ANNs

disadvantages -

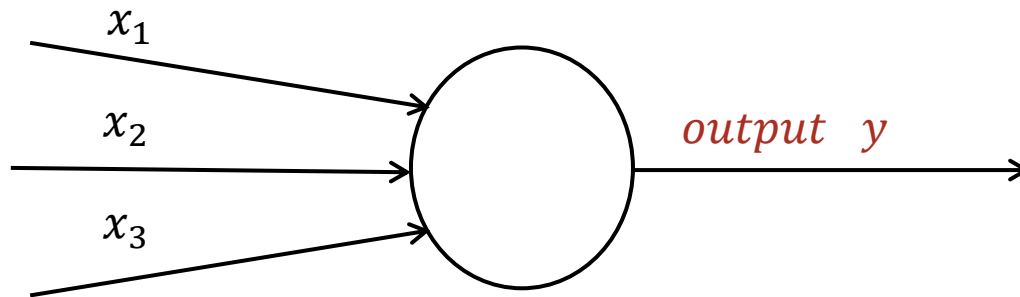
- in the standard case works only for numeric data (data transformation or specific kernels required)
- interpretability: poor interpretability, if the data is mapped to higher-dimensional space

Artificial neural networks

Fundamentals: Perceptron

Perceptron – a simple artificial neural network: overview

- ▶ one trivial artificial neural network consists of one node („neuron“) with multiple inputs and one output
- ▶ proposed by Rosenblatt in 1957 (see e.g. (Geron, 18) chapter 10)



- ▶ can be used for **linear classification of two classes** („binary classification“)
- ▶ data is passed to the input, one dimension/attribute per input
 - ▶ e.g. input could be height, width, colour value of apples and pears
- ▶ the output is the classification result
 - ▶ e.g. output could be: „it's an apple“ or „it's a pear“

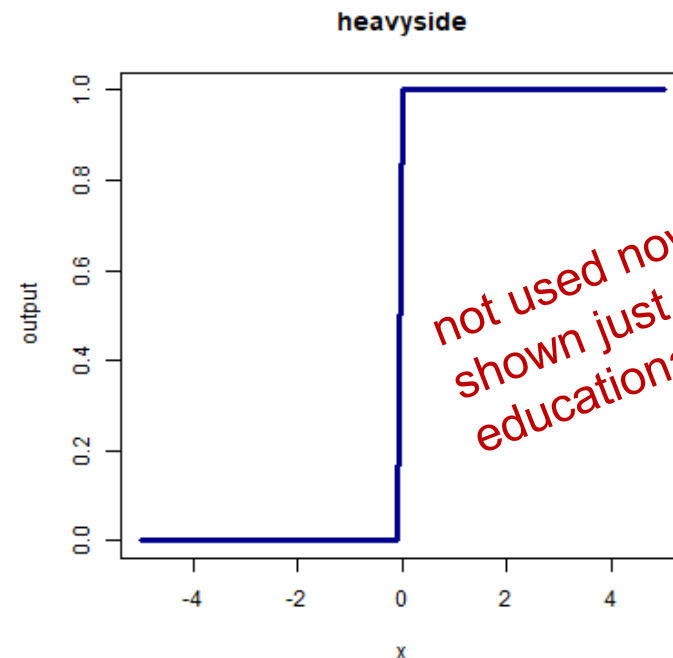
Artificial neural networks

Fundamentals

Simple activation function: heavyside function

▶ $f(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$

▶ $\mathbb{R} \rightarrow [0,1]$



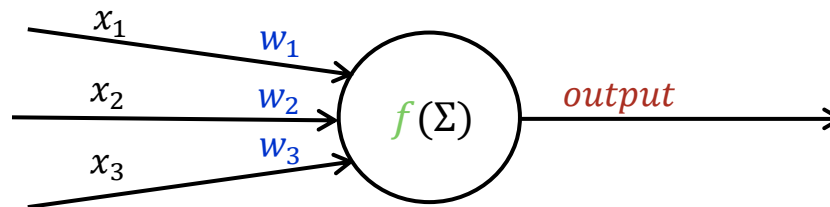
- ▶ if the sum of of the network's weighted inputs is >0, the output is 1
- ▶ otherwise the output is 0
- ▶ an output of 0 or 1 allows to classify data into two classes based on inputs

Classification

Fundamentals

In a nutshell: Perceptron – a simple artificial neural networks:

- ▶ can be used for linear classification of two classes („binary classification“)
- ▶ data is passed to the input, one dimension/attribute per input
- ▶ output is the classification result

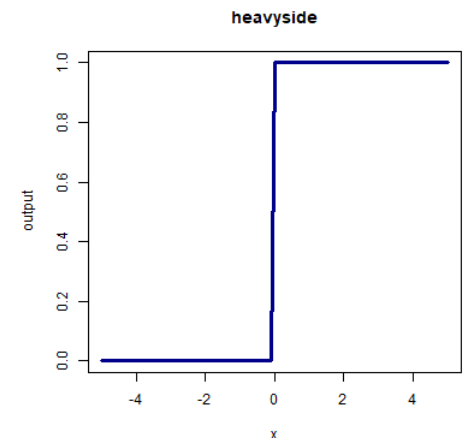


- the inputs of each node x_i are weighted with w_i and summed up:

$$\sum_{i=1}^n w_i x_i$$

- the sum is passed as to a so-called activation function $f()$ (e.g. heavyside) and $f(x)$ is the node's output

$$\text{output}_i = f\left(\sum_{i=1}^n w_i x_i\right)$$



Artificial neural networks

Fundamentals

Exercise: Calculations with perceptron

Calculate the neural networks output for the following inputs:

- ▶ input:
 - ▶ data point 1: (2, 1, 4)
 - ▶ data point 2: (1, 2, 1)

- ▶ output:
 - ▶ 0 for class 0 (e.g. apple)
 - ▶ 1 for class 1 (e.g. pear)

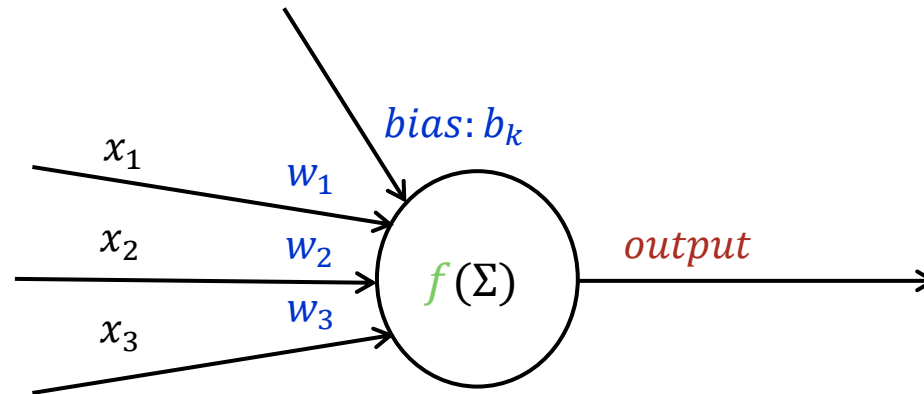
- ▶ weights: (0.5, 1.5, -1.0)

- ▶ activation function: heavyside

Artificial neural networks

Fundamentals

The bias term



- in most networks a so-called bias is added to each node
- the bias functions like an offset, it allows to change the output independently of the node's inputs
- geometrically it is the intercept of a plane described by $w_1 * x_1 + w_2 * x_2 \dots$

The formula of the weighted sum changes to

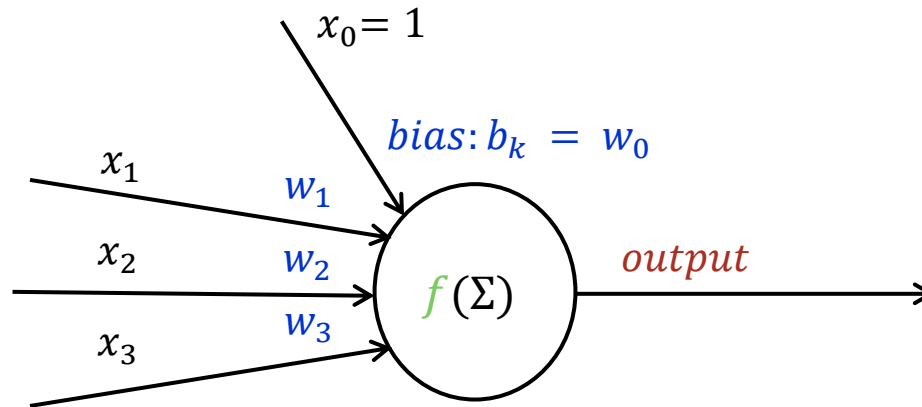
$$w_1 * x_1 + w_2 * x_2 + w_3 * x_3 + b_k = b_k + \sum_{i=1}^n w_i x_i$$

The output is now given by

$$\text{output} = f\left(b_k + \sum_{i=1}^n w_i x_i\right)$$

Classification Fundamentals

Vector form with bias term incorporated into vectors



Sometime in literature the bias term b_k is incorporated into the vector of weights as w_0 and the vector \mathbf{x} is enhanced by $x_0 = 1$.

This allows for a more compact formulation:

$$\text{output} = f(\mathbf{w}\mathbf{x})$$

- where \mathbf{w} is the vector $\begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \end{pmatrix}$ and \mathbf{x} is the vector $\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix}$

Artificial neural networks

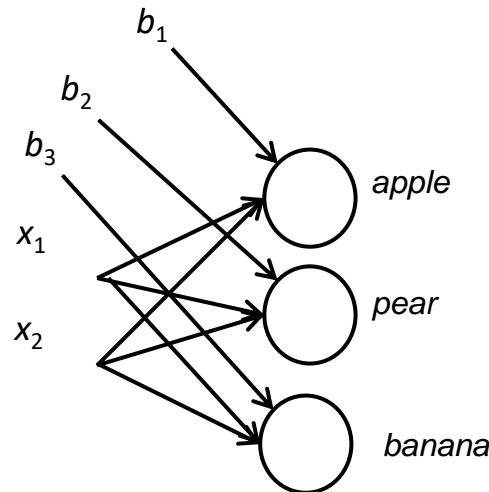
Perceptron for multiple classes

Perceptron for more than two classes:

- one node can be used to separate two classes
- for more than two classes: one output node per class is used

example:

$f_1 = 7 \text{ cm}$; $f_2 = 6 \text{ cm}$



example:

output for *apple* = 1,0,0
output for *pear* = 0,1,0

- one output node corresponds to one class, decision is taken by selecting the maximum output value
- the ANN is now a **linear classifier** for multiple classes

Artificial neural networks

Activation functions

Activation functions

The heavyside activation function $f(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$

was used in early networks.

However, only linear classifiers can be built based on that function.

A variety of activation functions have been proposed and used since. Some of them are:

- ▶ linear
 - ▶ logistic (a sigmoid function)
 - ▶ tangens hyperbolicus tanh (a sigmoid function)
 - ▶ rectifier (the unit is then called called ReLU = rectified linear unit)
-
- ▶ Different activation functions can be used in layers of the network

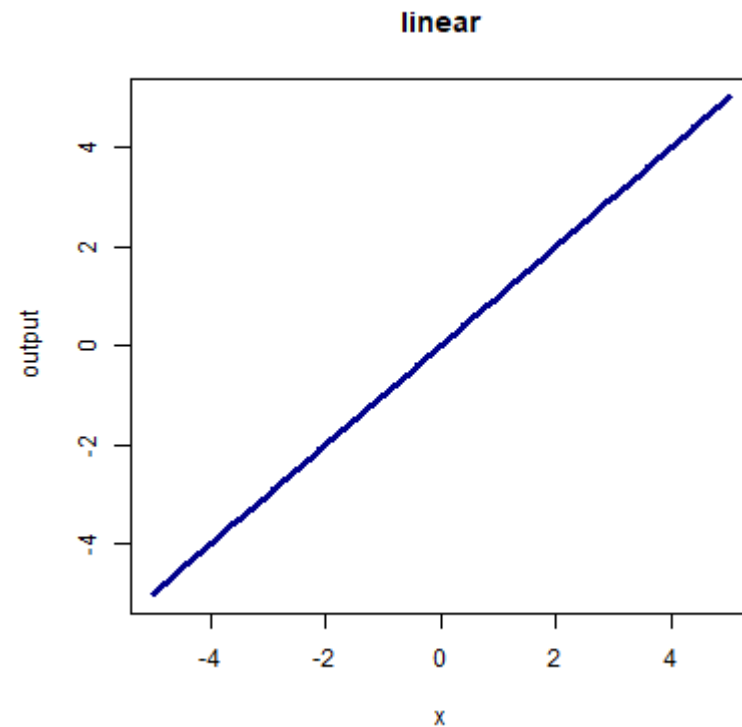
Artificial neural networks

Activation functions

Activation functions: linear

linear

► $\mathbb{R} \rightarrow \mathbb{R}; f(x) = x$



Artificial neural networks

Activation functions

Activation functions: sigmoid functions

sigmoid functions are S-shaped
they are differentiable

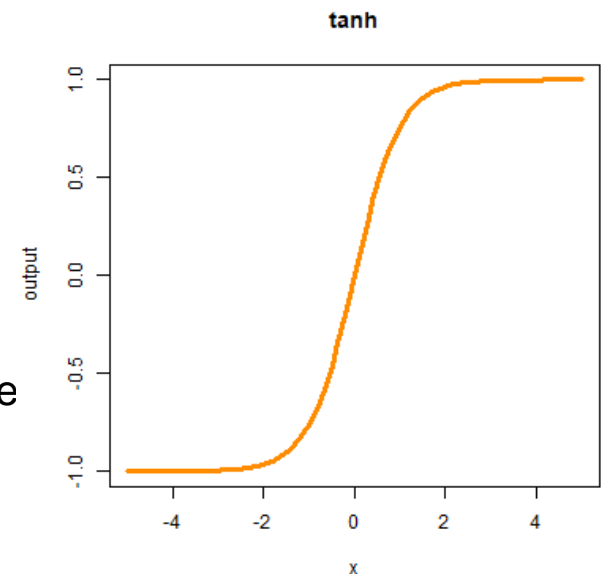
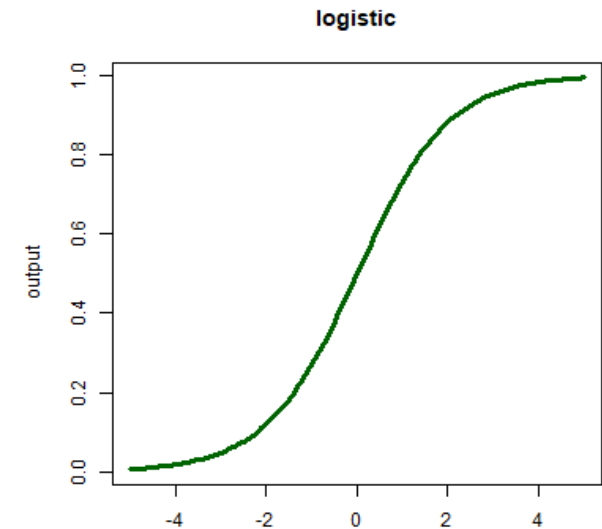
- ▶ logistic:

- ▶ $\mathbb{R} \rightarrow (0,1); f(x) = \frac{1}{1+e^{-x}}$

- ▶ tanh:

- ▶ $\mathbb{R} \rightarrow (-1, +1); f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

- ▶ sigmoid functions were used many years, and still are used
- ▶ they have problems when using many layers (not covered in this course, yet)

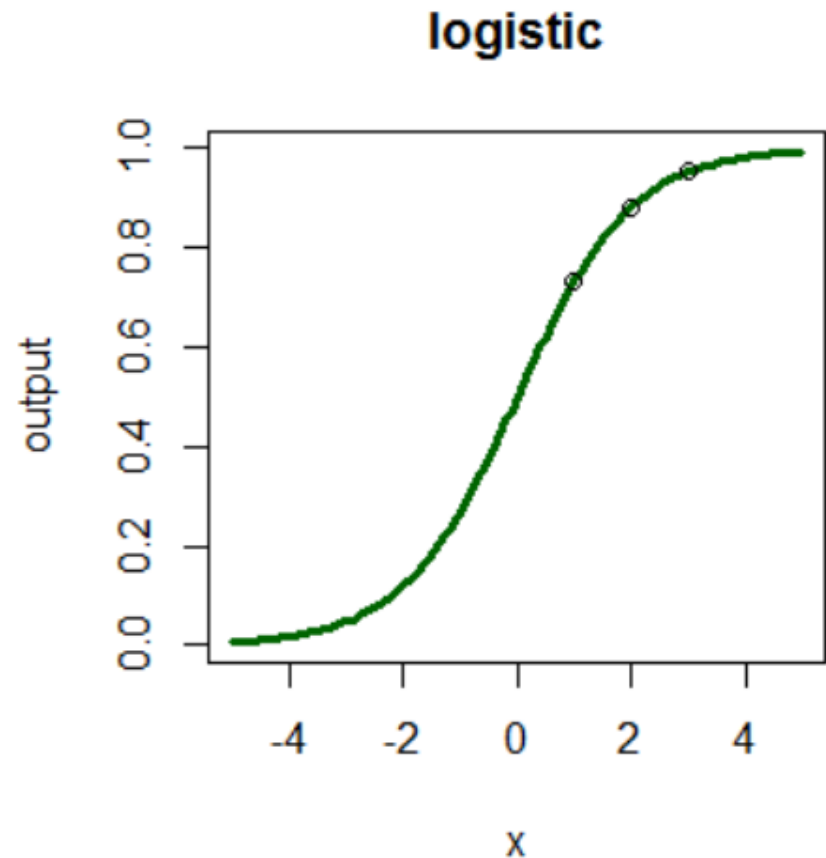


Artificial neural networks

Activation functions

Activation functions: sigmoid functions

- ▶ as can be seen, sigmoid functions fulfill a non-linear mapping of inputs to outputs
- ▶ the decision function with n parallel nodes is, however, still linear!



Artificial neural networks

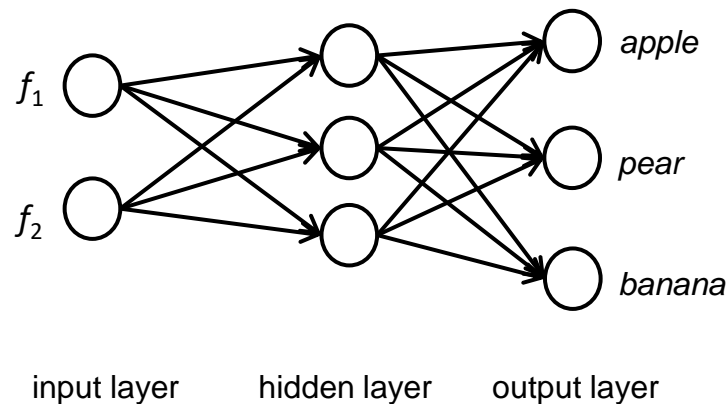
Multilayer perceptron (MLP)

Building an ANN:

- several neurons (nodes) in parallel are one layer
- several layers are connected sequentially
- prediction according to outputs at output layer
(note: a two-class problem can be solved with one or two output nodes)
- by connecting the neurons
 - a „network“ is set up: the „artificial neural network“

Input data is passed to the **input layer**

Example:
 $f_1 = 7 \text{ cm}$; $f_2 = 6 \text{ cm}$



The **output layer** outputs the results

example:
apple = 0.7
pear = 0.2
banana = 0.1

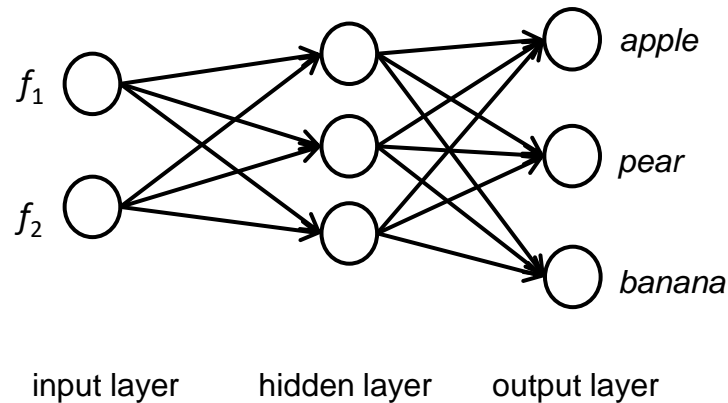
=> prediction = max output: apple

Artificial neural networks

Functioning of ANNs: overview

Input data is passed to the **input layer**

Example:
 $f_1 = 7 \text{ cm}$; $f_2 = 6 \text{ cm}$



The **output layer** outputs the results

example:
 $\text{apple} = 0.7$
 $\text{pear} = 0.2$
 $\text{banana} = 0.1$
 \Rightarrow prediction: *apple*

„Machine learning“ by an ANN:

- adaptation of the weights w_i , such that for input data the ANN takes the correct decision, i.e. output corresponds to class labels (technique: „backpropagation“)
 - iterative improvement by using many repetitions on the training set
 - one run on the entire training set is called an epoch, very many epochs are required
- example: input: (8 cm ; 7 cm) + „apple“ \Rightarrow output: $\text{apple} = 0,2$; $\text{pear} = 0,8$, $\text{banana} = 0,0$
- adaptation of weights w_i , such that classification is correct (goal: $\text{apple} = 1$; $\text{pear} = 0$; $\text{banana} = 0$)

Artificial neural networks

Functioning of ANNs: overview

Typical traditional ANN – the multi-layer perceptron (MLP)

There is an enormous variety of neural networks, the „**traditional**“ type that is typically used is as follows:

- ▶ feed-forward multilayer perceptron:
 - ▶ „feed-forward“: the information flow is strictly from left to right
 - ▶ „multilayer perceptron“: consists of multiple layers of perceptrons
- ▶ activation function: logistic or tanh
- ▶ training using backpropagation
- artificial neural networks have recently become highly relevant again in the field of **deep learning**, where so-called deep neural networks are used
- in principle these deep neural networks work similar as shown here, but use some advancements

Artificial neural networks

Functioning of ANNs: overview

Artificial neural networks (ANNs)

advantages +

- robustness: robust against noise in the training set
- accuracy: highly flexible decision boundaries
- can be used for different tasks besides classification, e.g. forecasting, regression

disadvantages -

- scalability: long training period
- interpretability: poor interpretability, the result of the training is a vector/matrix of weights
- network-structure and learning rate has to be pre-redefined
- random parts involved (e.g. initialization of weights)
- global minimum not guaranteed with the standard backpropagation gradient-descent

Artificial neural networks (ANNs)

Backpropagation

- ▶ backpropagation is used for parameter tuning, where the parameters are the weights including the biases
- ▶ backpropagation uses gradient descent (dt. Gradientenabstiegsverfahren)
- ▶ the error at the output layer is iteratively reduced

- ▶ the weights are randomly initialized and tuned during the training process
- ▶ the weights are changed in a way to reduce the error at the output layer

Artificial neural networks (ANNs)

Backpropagation

- ▶ in order to be able to reduce the error, the error needs to be calculated
- ▶ this is done using an error function also called loss function
- ▶ one common loss function is the „mean sum of squared errors“ MSE:

$$E = \frac{1}{m} \sum_{i=1}^m (\hat{y}_i - y_i)^2$$

- ▶ where y_i is a vector with the label of the data point i , \hat{y}_i is the networks output (prediction) for data point i , and m is the number of data points
- ▶ \hat{y}_i is a function of the weights
- ▶ note: there are other loss functions

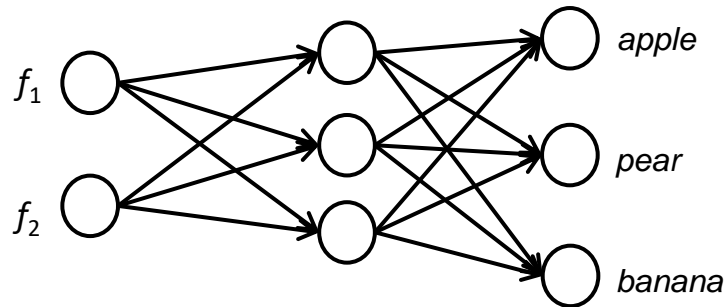
Artificial neural networks (ANNs)

Backpropagation

One-hot encoding

- ▶ in the loss function, y_i is required to be a vector in order to be comparable to the networks output
- ▶ this is achieved by transforming the labels using **one-hot encoding**
- ▶ the class label then corresponds to the desired network output

For this example ANN:



- ▶ the one-hot encoded labels are $\text{apple} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$, $\text{pear} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$, $\text{banana} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$
- ▶ predictions (outputs) are floating point numbers,
 - ▶ e.g. for apple i.e. $\text{apple} = \begin{pmatrix} 0.8 \\ 0.1 \\ 0.1 \end{pmatrix}$ or $\begin{pmatrix} 0.6 \\ 0.2 \\ 0.2 \end{pmatrix}$
- ▶ the goal is an output that is close to the one-hot encoded label, i.e. for an apple close to $\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$

Artificial neural networks (ANNs)

Backpropagation

- ▶ when the loss function $E = 0$, the output for all data points corresponds to the class labels
- ▶ $E = 0$ is unlikely to be achieved
- ▶ note that correct predictions are possible without $E = 0$:

Example:

- ▶ an ANN shall have an output node for apple, pear and banana
- ▶ the output for one data point x_i , which is an apple, shall be $\begin{pmatrix} 0.8 \\ 0.1 \\ 0.1 \end{pmatrix}$
- ▶ decision will be apple

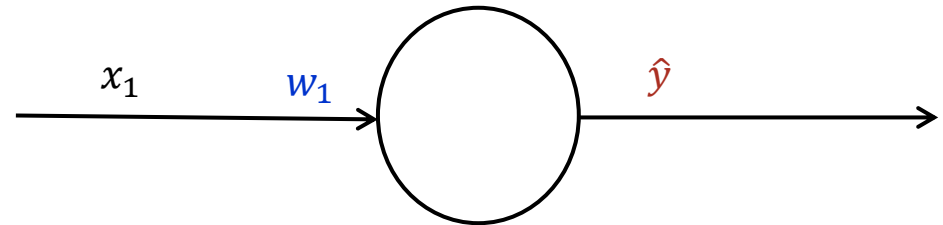
Exercise: What is the error for this data point ?

$$\begin{aligned} E_i &= (\hat{\mathbf{y}}_i - \mathbf{y}_i)^2 \\ &= \text{sum}(\begin{pmatrix} 0.8 \\ 0.1 \\ 0.1 \end{pmatrix} - \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix})^2 = (0.8 - 1)^2 + (0.1 - 0)^2 + (0.1 - 0)^2 = 0.06 \end{aligned}$$

Artificial neural networks (ANNs)

Backpropagation: Gradient descent with simplified example

- ▶ backpropagation uses gradient descent
- ▶ for a very simplified network:
 - ▶ one input
 - ▶ one output
 - ▶ „trained“ with one data point
 - ▶ ignoring the activation function (or assuming a linear activation function $y=x$)
- ▶ for this simplified example with one data point the loss function



$$E = \frac{1}{m} \sum_{i=1}^m (\hat{y}_i - y_i)^2$$

can be rewritten with $\hat{y}_i = w_1 x_1$:

$$E = (w_1 x_1 - y_1)^2$$

- ▶ so the error is a function of the weight, x_1 and y_1 are constant
- ▶ we want this error to be minimal

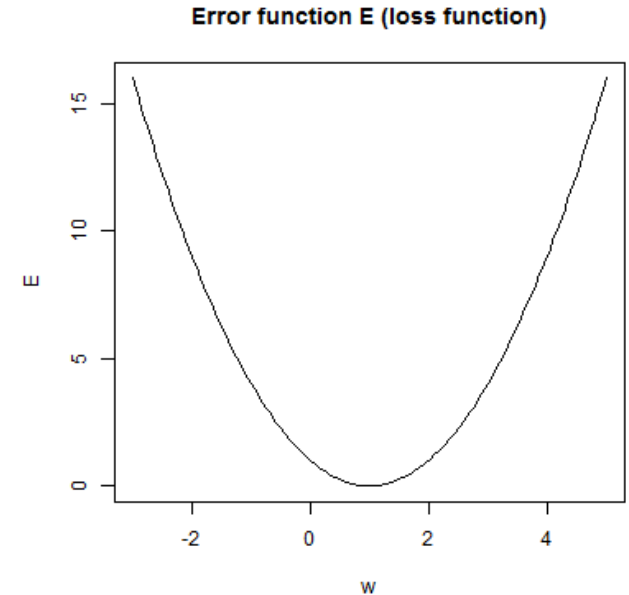
Artificial neural networks (ANNs)

Backpropagation: Gradient descent with simplified example

The loss function

$$E = (w_1 x_1 - y_1)^2$$

is obviously a quadratic function



- ▶ We want to find the error function's minimum
- ▶ We can use the function's derivative and find the value of w_1 , where the function is 0

Artificial neural networks (ANNs)

Backpropagation: Gradient descent with simplified example

Using the chain rule (dt. Kettenregel)

$$f(x) = g(h(x)) \rightarrow f'(x) = g'(h(x)) * h'(x)$$

we find the derivative of

$$E = (w_1 x_1 - y_1)^2$$

to be

$$E' = 2 * (w_1 x_1 - y_1) * x_1$$

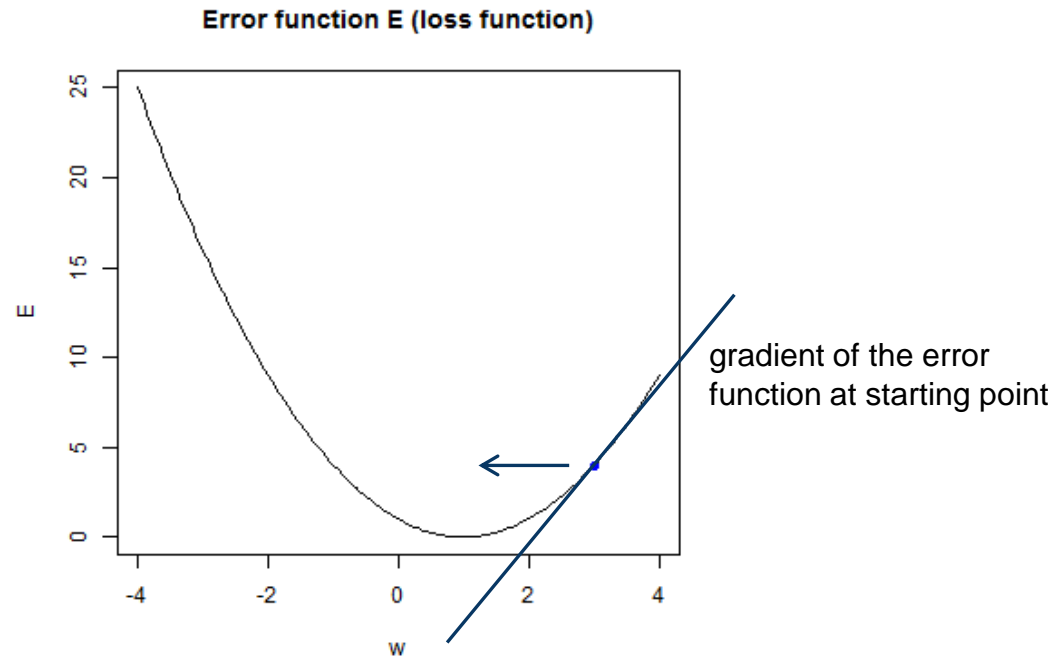
- ▶ during training we do not know all function values, we only know E for the current input x_1 and the current value of weight w_1
- ▶ starting with a random w_1 we want to change w_1 in a way that minimizes E

Artificial neural networks (ANNs)

Backpropagation: Gradient descent with simplified example

We know the type of error function (quadratic in our example)

So we can calculate $E(w_1)$ and use the gradient $E'(w_1)$ to find the direction and the extent of the desired change of w



- ▶ the gradient yields the direction and the extent of the weight update that reduces E

Artificial neural networks (ANNs)

Backpropagation: Gradient descent with simplified example

Learning rate

To control the extent of the weight update we introduce a learning rate $\eta \in (0,1)$

The weights are iteratively (step-wise) changed as follows:

$$w_{(\text{step } t+1)} = w_{(\text{step } t)} - \eta * E'(w_{(\text{step } t)})$$

- ▶ the learning rate regulates the trade-off between convergence speed and the stability of the process to converge
- ▶ the learning rate is a hyperparameter
- ▶ for small networks a constant learning rate of 0.1 is commonly used

Artificial neural networks (ANNs)

Backpropagation: Gradient descent with simplified example

Example: gradient descent

- ▶ in our example let $x_1 = 1$ and $y_1 = 1$, $\eta = 0.1$
- ▶ in other words for an input of 1 we want the network's output to be 1
- ▶ (note that $w_1 = 1$ is the obvious solution)

Starting with a random $w_1 = 3$, the error in the first step is:

$$\begin{aligned} E(w_1) &= (w_1 x_1 - y_1)^2 \\ &= (3 * 1 - 1)^2 = 4 \end{aligned}$$

the gradient for $w_1 = 3$ is

$$\begin{aligned} E'(w_1) &= 2 * (w_1 x_1 - y_1) * x_1 \\ &= 2 * (3 * 1 - 1) * 1 = 4 \end{aligned}$$

We can calculate the weight of the next step:

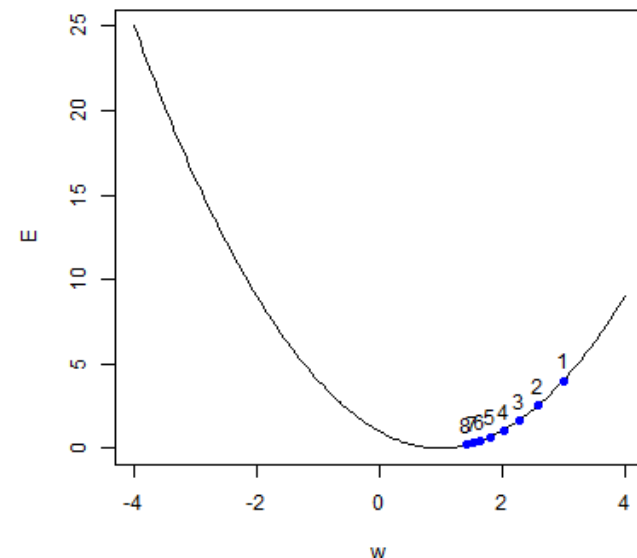
$$w_{(\text{step } t+1)} = w_{(\text{step } t)} - \eta * E'(w_{(\text{step } t)})$$

$$w_{(\text{step } t+1)} = 3 - 0.1 * 4 = 2.6$$

steps of gradient descent:

step	w_1	E
1	3	4
2	2.6	2.56
3	2.28	1.63
4	2.02	1.04
5	1.82	0.67
6	1.65	0.43

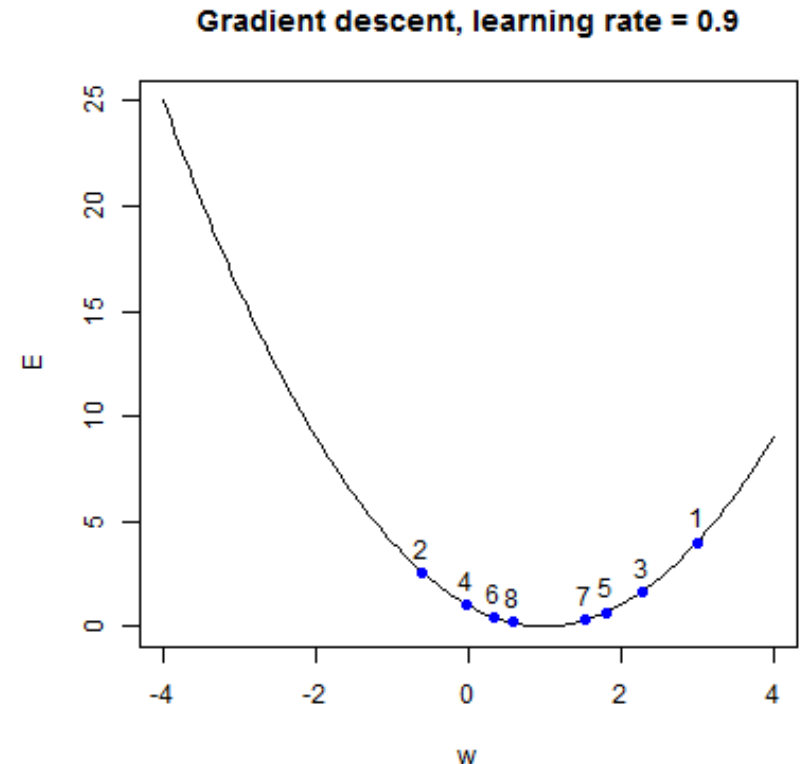
Gradient descent, learning rate = 0.1



Artificial neural networks (ANNs)

Backpropagation: Gradient descent with simplified example

- ▶ higher learning rates lead to faster changes
- ▶ step might be so large that minimum is over-stepped
- ▶ then the next step will change the weight in the opposite direction
- ▶ oscillation is possible
- ▶ might diverge instead of converge



Artificial neural networks (ANNs)

Training with backpropagation

Backpropagation on multiple layers

In the loss function

$$E = \frac{1}{m} \sum_{i=1}^m (\hat{\mathbf{y}}_i - \mathbf{y}_i)^2$$

$\hat{\mathbf{y}}_i$ is the vector of the output layer, e.g. $\begin{pmatrix} 0.8 \\ 0.1 \\ 0.1 \end{pmatrix}$

- ▶ each component of this vector is given by the weighted sum of that node's inputs with the activation function applied, i.e. $f(\mathbf{w}\mathbf{x}) = f(\sum_{i=0}^n w_i x_i)$
- ▶ this is recursive, i.e. a node's output in layer N depends on the node's inputs, which depend on the nodes' outputs of layer N-1, ...
- ▶ in addition to the previous simplified example, the derivative of the activation function is incorporated into the gradient descent steps

Artificial neural networks (ANNs)

Training with backpropagation

Backpropagation – a „quick generalization“ to multiple layers

In the loss function

$$E = \frac{1}{m} \sum_{i=1}^m (\hat{\mathbf{y}}_i - \mathbf{y}_i)^2$$

$\hat{\mathbf{y}}_i$ is the vector of the output layer, e.g. $\begin{pmatrix} 0.8 \\ 0.1 \\ 0.1 \end{pmatrix}$

- ▶ each component of this vector is given by the weighted sum of that node's inputs with the activation function applied, i.e. $f(\mathbf{w}\mathbf{x}) = f(\sum_{i=0}^n w_i x_i)$
- ▶ this is recursive, i.e. a node's output in layer N depends on the node's inputs, which depend on the nodes' outputs of layer N-1, ...
- ▶ in addition to the previous simplified example, the derivative of the activation function is incorporated into the gradient descent steps

Artificial neural networks (ANNs)

Some common loss functions for classification

Mean Squared Error(MSE):

$$MSE = \frac{1}{n} \sum_{i=1}^n \left(\vec{y}_i - \hat{\vec{y}}_i \right)^2 = \frac{1}{n} \sum_{i=1}^n \sum_{j=1}^{|C|} (\hat{y}_{i,j} - y_{i,j})^2$$

where i is the index of the data points $1 \dots n$ and j is the index of the classes $1 \dots |C|$.

Binary Cross-Entropy (BCE) – for two classes:

$$BCE(\vec{w}) = -\frac{1}{n} \sum_{i=1}^n [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

Categorical Cross-Entropy (CCE):

$$CCE(\vec{w}) = -\frac{1}{n} \sum_{i=1}^n \sum_{j=1}^{|C|} y_{i,j} \log(\hat{y}_{i,j})$$

where $y_{i,j}$ denotes the vector component j of class label y_i and $\hat{y}_{i,j}$ the prediction indexed by i at node j .

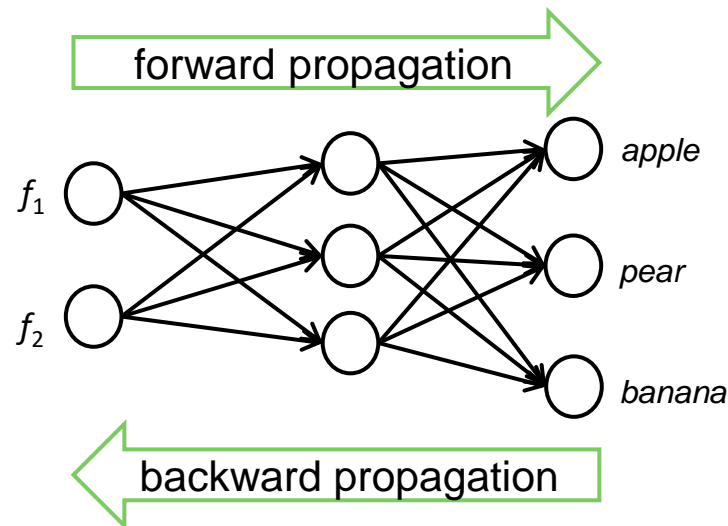
Artificial neural networks (ANNs)

Training with backpropagation

The weights of all nodes need to be tuned in order to reduce the error

Each iteration of backpropagation has the following two steps:

1. forward propagation: the data is passed to the input layer and the error is calculated at the output layer for all passed data points
2. using gradient descent, the weights of all nodes in all layers are changed using gradient descent



- ▶ one iteration with all data points is called an epoch
- ▶ this process is repeated many times, typically thousands of iterations

Artificial neural networks (ANNs)

Training with backpropagation

An ANN has a high number of weights

- ▶ so the function to be optimized is not as simple as shown in the example for one weight by the quadratic function
- ▶ it is a function of very many variables

The process, however, is the same:

- ▶ the function's gradient is calculated in order to determine the direction and extent the of the weight update
- ▶ instead of the derivative, the partial derivatives are used

One can imagine that process for two variables as a skate bowl, where we roll a ball and we want it to stop at the minimum:



image:
CONCRETESKATEPARKS
Design and construction
of a skateboarding recreational facility.
Bachelor Thesis, Teodor Daskalov

Artificial neural networks (ANNs)



```
# split into training and test set
train_data, test_data, train_labels, test_labels = train_test_split(
    data, labels, test_size = 0.5, random state=123)
```

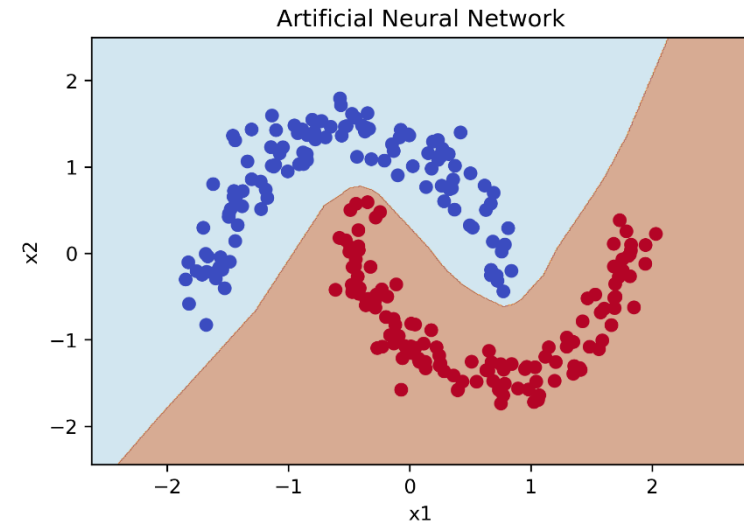
```
# z-score scaling: determine scaling parameters
scaler = StandardScaler().fit(train_data)
```

```
# scale train set and test set
train_data = scaler.transform(train_data)
test_data = scaler.transform(test_data)
```

```
# create neural network
model = MLPClassifier(hidden_layer_sizes=(20,20,),
    activation='relu', solver='adam', max_iter=1000,
    learning_rate_init=0.001, momentum=0.9)
```

```
# train model on training set
model.fit(train_data, train_labels)
```

```
# classification of test set
predictions = model.predict(test_data)
```



Artificial neural networks (ANNs)

Deep Learning

Deep Learning – an overview

Deep Learning uses ANNs as we have seen them so far, with some advancements

- ▶ more layers
- ▶ other activation functions
- ▶ advanced training methods
- ▶ additional methods to avoid overfitting
- ▶ Deep Learning needs a lot of training data and requires long training
- ▶ **the basic idea is not to do feature engineering, but let the Deep ANN determine the features it requires to separate the data points**

Artificial neural networks (ANNs)

Deep Learning

Deep Learning – an overview

in addition: Deep Learning uses network architectures for specific data types, e.g.

- ▶ CNNs for images
- ▶ RNNs for time series or text

In general: The distinction between „Deep Learning“ and ANNs without Deep Learning is not too strict. Many ideas have been around before the term Deep Learning was introduced.

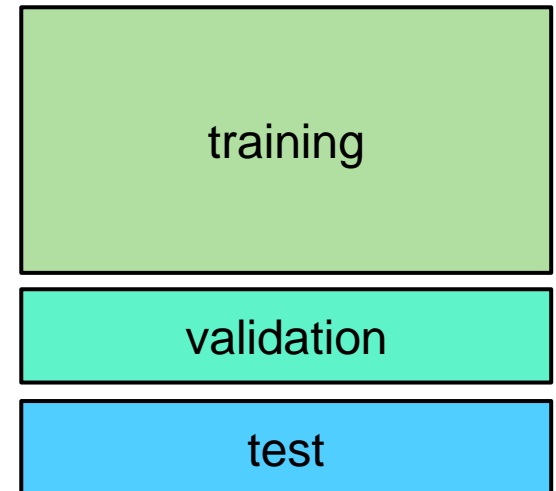
Artificial neural networks (ANNs)

Avoiding overfitting

In order to monitor the process during training, the ANNs error in the current step is calculated not only on the training set, but on a so-called validation set

this is an approach to try to detect overfitting

- this validation set is split from the initial training set
- during a step multiple validation sets can be extracted using “cross-validation”
- Recommendation: a „blind test set“ should still be kept, i.e. validation set is not the same as the test set
- the blind test set can be used as a final quality gate after the training process



Artificial neural networks (ANNs)

Avoiding overfitting: early stopping

Early stopping

- ▶ **observation:**
 - error on training set (train error, train loss) decreases
 - error on validation set (test error, test loss) decreases up to some point after that point, the error on the test set increases => overfitting



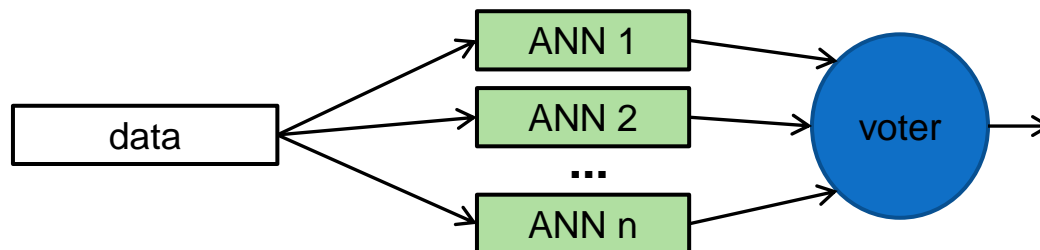
- ▶ **idea:** stop training when overfitting starts („early stopping“)
- ▶ stop training when error on validation set (validation error, validation loss) increases
- ▶ since the errors are not as stable as shown here, the errors are monitored during training and training is stopped after a „stable“ increase of the validation error

Avoiding overfitting

Ensemble

„Ensemble“ – combining several models

- ▶ **observation:** each model has different strenghts and weaknesses, i.e. each model may overfit differently
- ▶ **idea:** combining many different models („ensemble“)
- ▶ **Some options:**
 - ANNs with different architectures (number of layers, number of nodes, etc.)
 - same ANN architecture trained on different subsets of the data, hence the weights w_i will be different



Example:

ANN 1: „apple“

ANN 2: „pear“

ANN 3: „apple“

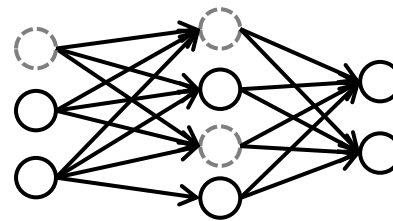
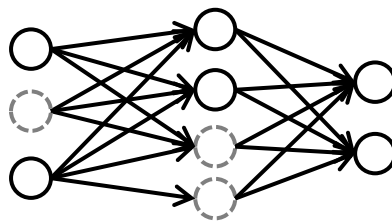
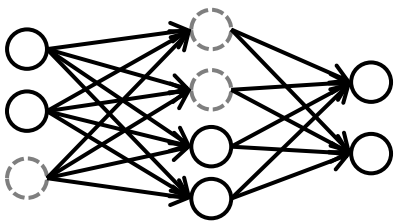
} = „apple“

Avoiding overfitting

Dropout

Dropout

- ▶ **observation:** the idea of an „ensemble“ ist good, but is computationally expensive and requires a lot of training data
- ▶ **idea:** use different representations of the same ANN
- ▶ **Functioning:**
 - for each data point („feature vector“), randomly deactivate nodes and their connections
 - deactivate nodes with some probability, e.g. $p=20\%$ in input layer, $p=50\%$ in hidden layer
 - when using the network to predict data, all nodes are active



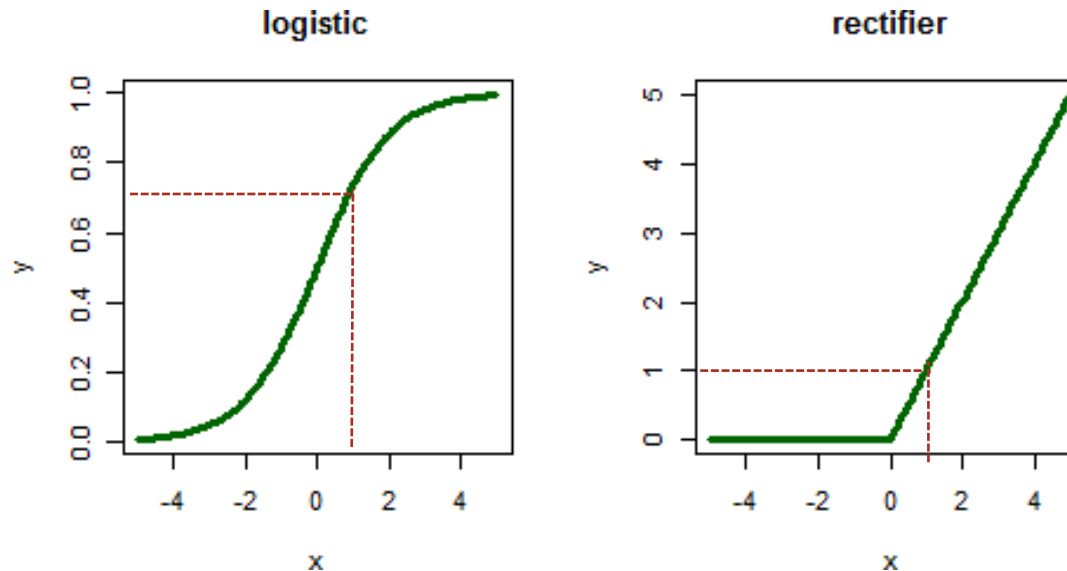
○ : active node
○ : inactive node

Avoiding overfitting

- There are more techniques to avoid overfitting, e.g. L1-regularization, L2-regularization
- Recommendation:
 - ▶ early stopping: is typically used
 - ▶ ensemble: depends on the availability of data and computational power
 - ▶ dropout: is typically used, particularly when for ANNs with many layers
 - ▶ A combination of multiple techniques is often used, e.g. early stopping and dropout

Activation functions for Deep ANNs

- commonly used activation functions: instead of logistic function Deep ANNs use rectifier functions (and variants of it)
 - ▶ rectifier: $\mathbb{R} \rightarrow [0, \infty]$; $f(x) = \max(0, x)$
 - ▶ the activation function is called rectifier, the unit is then called called ReLU (rectified linear unit)
 - ▶ (these terms are often used as equivalents also they are not)



Activation functions for Deep ANNs

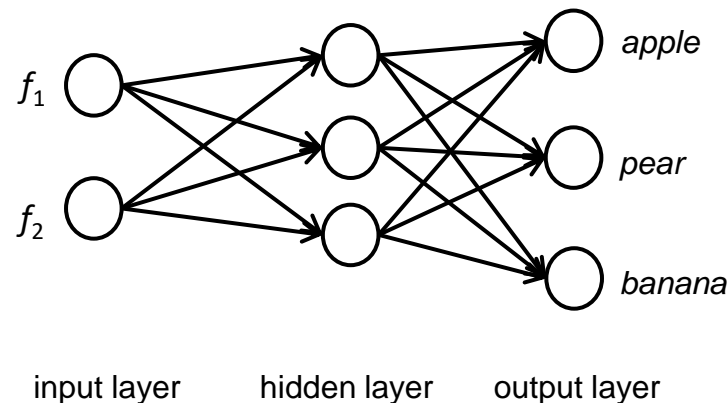
Softmax function

For classification problems, the output layer typically uses the softmax activation function

- This function scales the values such that the sum of all output nodes is 1
- The output of one node can then be interpreted as a probability

Example: output might be at the nodes:

- ▶ apple: 0.7 , pear: 0.2 , banana: 0.1
- ▶ so the ANN predicts that the data point is an apple with a probability of 0.7



Training Artificial neural networks (ANNs)

Using mini-batches

- ▶ Deep Learning requires a large training set
- ▶ Epoch learning, i.e. adapting the weights after having seen all data points, would not converge, since it would take very long
- ▶ Mini-batch updating is used

mini-batch updating (=stochastic gradient descent (SGD) with batch size > 1):

- ▶ a small number of random samples (e.g. 50) are taken from the training set
- ▶ the mean error is calculated (summing up the errors and dividing by the number of data points)
- ▶ the weights are updated
- ▶ faster update of weights, but updates depend only on a small subset of the data
- ▶ common approach for Deep Learning

Training Artificial neural networks (ANNs)

Enhancement: momentum optimization

Momentum optimization

For smaller batch sizes (mini-batch updating or case updating), the change of weights might be instable

A momentum term can be introduced, that incorporates the previous weight update into the current weight update

i.e. if the weights were rapidly updated in positive direction and the current batch indicates an update in negative direction, this change of direction is smoothed

$$w_{(step\ t+1)} = w_{(step\ t)} - (\beta m + \eta E'(w_{(step\ t)}))$$

- ▶ where m is a vector with the previous gradients
- ▶ and β is a hyperparameter between 0 and 1 (0 = no momentum, 1 = maximal momentum). A common value for β is 0.9.

Training Artificial neural networks (ANNs)

Enhancement: **Learning rate decay / adaptive learning**

Learning rate decay / adaptive learning rate

- ▶ The learning rate η can be gradually decreased during the training process
- ▶ This allows to take large steps at the beginning and finer steps towards the end of the training process
- ▶ enhanced versions of backpropagation that use an adaptive learning rate are AdaGrad and RMSProp

Learning rate decay / adaptive learning rate + momentum optimization

- ▶ momentum optimization and adaptive learning rate is combined in ADAM (Adaptive momentum estimation)

Artificial neural networks (ANNs)

Programming with tensorflow / keras

```
[...]
# train test split
train_data, test_data, train_labels_raw, test_labels_raw = train_test_split(
    data, labels_raw, test_size = 0.5)

scaler = StandardScaler().fit(train_data) # z-score
train_data = scaler.transform(train_data)
test_data = scaler.transform(test_data)

# one-hot-encoding of the labels is required for neural networks
# i.e. instead of labels 1,2,3 => vectors (1 0 0), (0 1 0), (0 0 1)
def encodeLabelsOneHot(in_labels):
    labels_onehot = to_categorical(in_labels)
    return labels_onehot

train_labels = encodeLabelsOneHot(train_labels_raw) # call own function
test_labels = encodeLabelsOneHot(test_labels_raw)   # call own function

#extract number of features and classes
n_attributes = data.shape[1]
n_classes = train_labels.shape[1]
```

Artificial neural networks (ANNs)

Programming with tensorflow / keras

```
#create neural network, see API reference e.g.: https://keras.io/
model = Sequential()

#input layer is created implicitly, based on input_dim
model.add(Dense(units=20, activation='tanh', input_dim=n_attributes)) #hidden layer
model.add(Dense(units=10, activation='tanh')) #hidden layer
model.add(Dense(units=n_classes, activation='softmax')) #output layer

optim = Adam()
model.compile(loss='mean_squared_error', optimizer=optim, metrics=['accuracy'])

# train the model
model.fit(train_data, train_labels, batch_size=20, epochs=100)

# classification of test set
pred = model.predict(test_data)

# get from network outputs to classes
predictions = np.argmax(model.predict(test_data), axis=-1)
cm = confusion_matrix(test_labels_raw, predictions)
```

Experimenting with ANNs

Experimental environments

- <https://playground.tensorflow.org>
- <https://js.tensorflow.org/>

Cloud-based services

- Pre-installed environments available
- easy to use GPUs or TPUs

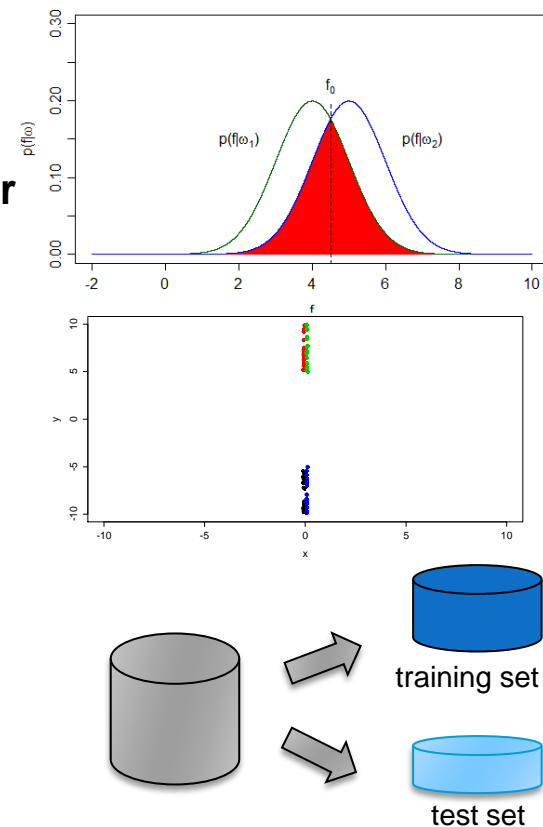
Classification: test

Question:

What could be reasons for a classifier to output very weak results?

Possible answers:

- Does the selected feature space allow us to get good results?
 - From the plot above it becomes obvious that **any classifier** will output weak results using this one feature.
- Did we scale the features?
 - If not, the features are weighted differently.
- Have we used instances from the training set to determine the classifier's accuracy?
 - In that case we get very good results after training, and possibly very weak results on unseen data.

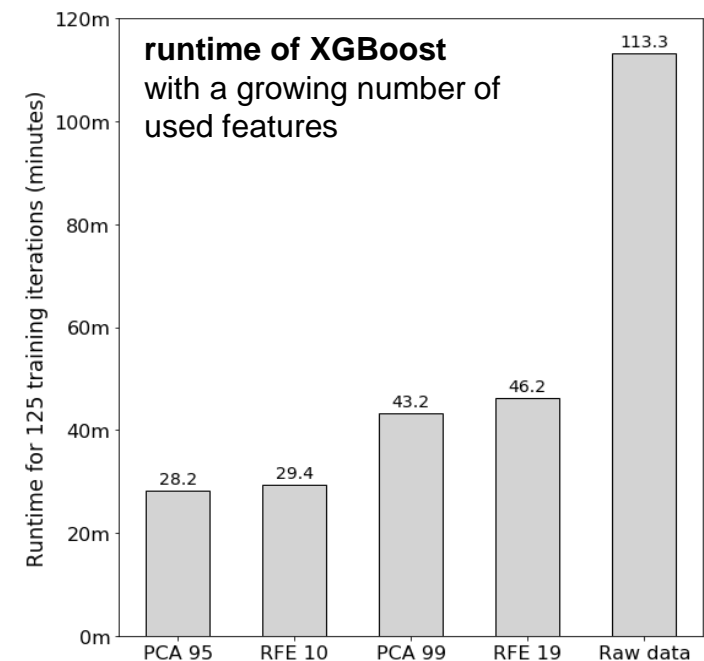
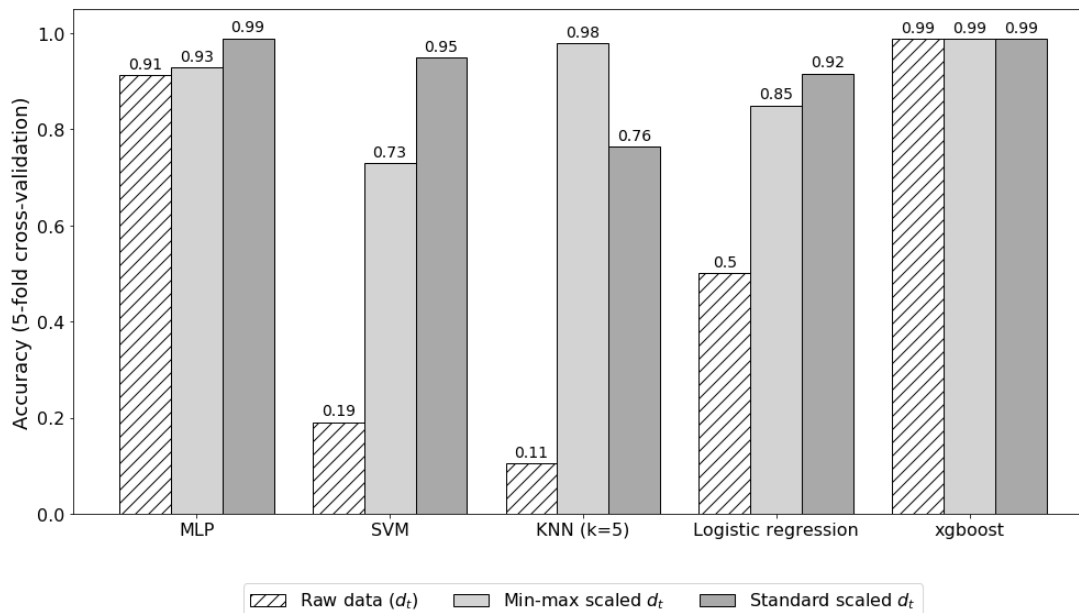


Case Study

Classification of fault types in electromagnetic drive systems

- ▶ multi-class problem: 11 fault types, i.e. 11 classes
- ▶ 48 features
- ▶ approx. 50.000 feature vectors

accuracy of different classifiers
on unscaled and scaled input data



Grüner, T., Böllhoff, F., Meisetschläger, R., Vydrenko, A., Bator, M., Dicks, A., & Theissler, A. (2020). Evaluation of Machine Learning for Sensorless Detection and Classification of Faults in Electromechanical Drive Systems. Proceedings 24th International Conference on Knowledge-Based and Intelligent Information & Engineering Systems, Procedia Computer Science, Volume 176, pages 1586-1595, 2020, Elsevier. ISSN 1877-0509
<https://doi.org/10.1016/j.procs.2020.09.170>

Case Study

Classification of fault types in electromagnetic drive systems

- ▶ decision trees can get messy...

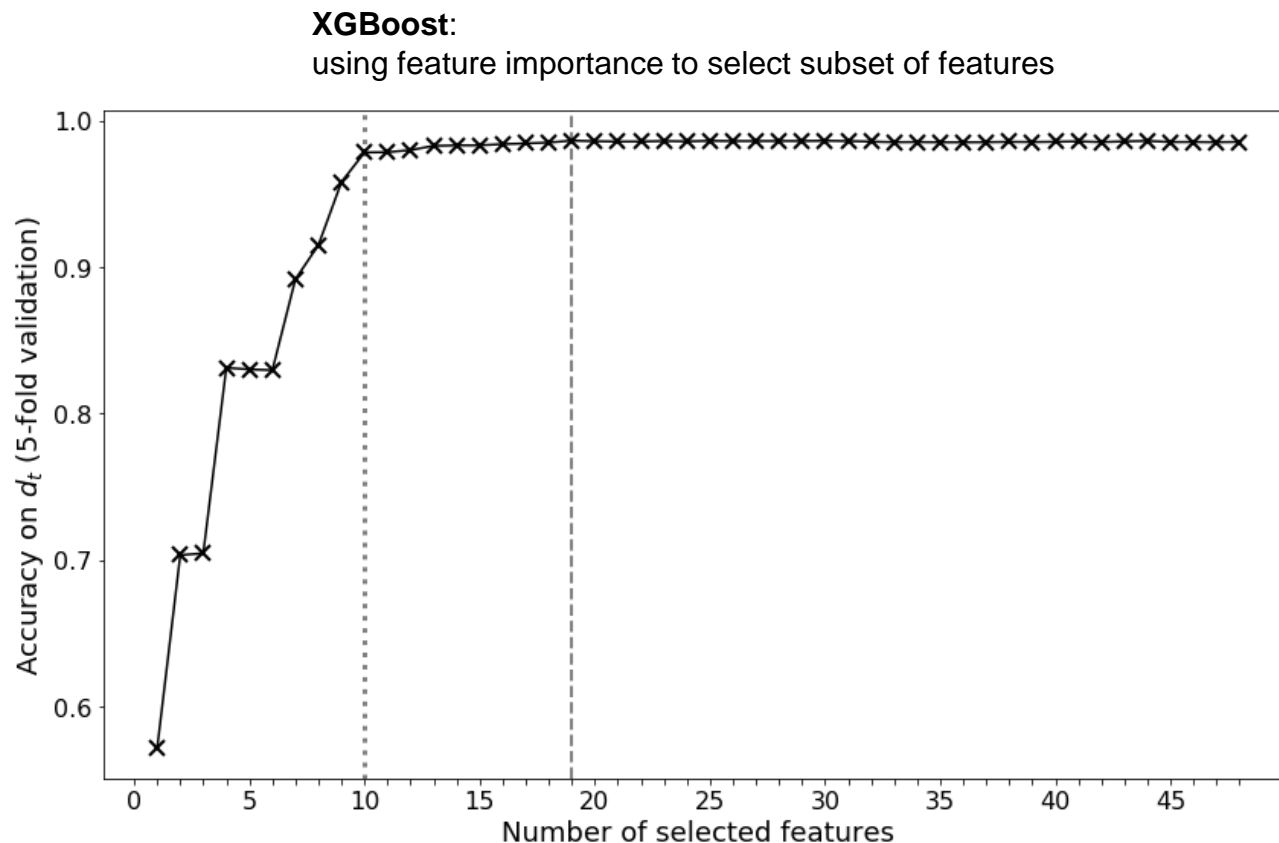


Case Study

Classification of fault types in electromagnetic drive systems

XGBoost

- ▶ classification with a subset of features
- ▶ features are selected based on feature importance by XGBoost



Further readings

Machine Learning + Deep Learning:

- ▶ J. Han, M. Kamber, J. Pei (2011). **Data Mining: Concepts and Techniques**. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers, 3rd Edition. ISBN 978-0123814791
- ▶ Mitchell, T. M. (1997). **Machine Learning**. McGraw-Hill Education (ISE Editions).
- ▶ A. Theissler (2013). **Detecting anomalies in multivariate time series from automotive systems**. PhD Thesis. Brunel University London
- ▶ Theodoridis, S. and Koutroumbas, K. (2009). **Pattern Recognition**, Fourth Edition. Academic Press, 4th edition.
- ▶ “Deep Learning” (2016).
Ian Goodfellow, Yoshua Bengio, Aaron Courville. kostenfrei online: www.deeplearningbook.org
- ▶ J. Han, M. Kamber, J. Pei. **Data Mining: Concepts and Techniques**. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers
- ▶ Mitchell, T. M. (1997). **Machine Learning**. McGraw-Hill Education
- ▶ Theodoridis, S. and Koutroumbas, K. (2009). **Pattern Recognition**, Fourth Edition. Academic Press
- ▶ Breiman (2001). Random Forests
<https://link.springer.com/article/10.1023/A:1010933404324>
- ▶ Friedman, J.H. (2000), Greedy Function Approximation: A Gradient Boosting Machine. Annals of Statistics, year = 2000, volume = {29}, pages = {1189--1232}
- ▶ Chen, Tianqi and Guestrin, Carlos (2016), XGBoost: A Scalable Tree Boosting System. Proceedings of the 22nd ACM SIGKDD 2016
<https://doi.org/10.1145/2939672.2939785>

Papers on Overfitting:

- ▶ „Dropout: A Simple Way to Prevent Neural Networks from Overfitting“ (2014)
- ▶ Srivastava, Nitish and Hinton, Geoffrey and Krizhevsky, Alex and Sutskever, Ilya and Salakhutdinov, Ruslan
- ▶ Journal Machine Learning Research. Jan. 2014, Vol. 15, Number 1, pages 1929-1958
- ▶ „Ensemble based systems in decision making“ (2006)
Robert Polikar, IEEE Circuits and Systems Magazine. 2006, Vol. 6, Issue 3, pages 21-45