BIT07: Machine Learning

Syllabus

Abstract

Study material for the seventh course of the Advanced Bachelor in Bioinformatics

Glenn Broeckx

Advanced Bachelor Bioinformatics

Contents

Contents **1**

Chapter 1: Introduction **5**
1.1. Supervised learning 5
1.2. Unsupervised learning 5
1.3. Reinforcement learning 6
1.4. Overview of ML algorithms 6
1.5. Machine learning approach 6

Chapter 2: Regression **7**
2.1. Linear regression 7
2.2. Finding best values for weights 7
2.2.1. Gradient Descent 7
2.2.2. Learning rate 7
2.2.3. Optima 8
2.3. Multivariate linear regression 8
2.4. Coding regression 8
2.5. Evaluating a regressor 8
2.5.1. Mean Absolute Error (MAE) 9
2.5.2. Mean Squared error (MSE) 9
2.5.3. Coefficient of determination (R^2 score) 9
2.6. Scaling of values 9
2.6.1. Min-max scaling 9
2.6.2. Standard scaling 10
2.6.3. Robust scaling 10
2.7. Feature engineering 10
2.7.1. Feature expansion 10
2.7.2. One-hot encoding 11
2.8. Overfitting and underfitting 11
2.9. Regularization 11
2.9.1. Ridge regression (L2) 12
2.9.2. Lasso regression (L1) 12
2.9.3. Hyperparameter λ/α 12
2.9.4. Regularization in code 12

Chapter 3: Classification **13**
3.1. Logistic regression 13
3.2. Cost function 13
3.3. Logistic regression in code 14
3.3.1. The model 14
3.3.2. Polynomial expansion 14
3.4. Overfitting and underfitting 15
3.5. Multi-class classification 15
3.5.1. One versus all 15
3.5.2. One versus one 15
3.5.3. Multiclass classification in code 15
3.6. Evaluating a classifier 16

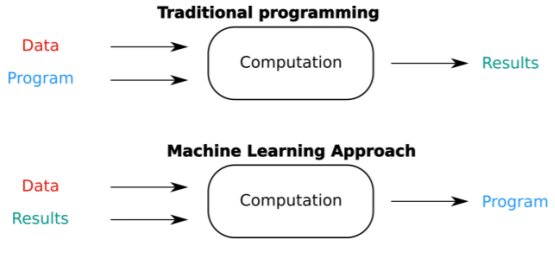
3.6.1. Confusion matrix and metrics 16
3.6.2. Receiving Operating Characteristic 16
Chapter 4: Hyperparameter tuning 17
4.1. Automated hyperparameter tuning 17
4.1.1. Grid Search 17
4.1.2. Cross validation (k-fold) 17
4.1.3. Random Search 18
4.1.4. Bayes Optimization 18
Chapter 5: Unbalanced data 19
5.1. Hyperparameter tuning 19
5.2. Collect more data 19
5.3. Undersampling majority class 19
5.4. Oversampling minority class 19
5.5. Change scoring system 19
5.6. Class-weight balancing 20
5.7. Data augmentation 20
Chapter 6: Naïve Bayes 21
6.1. Discriminative classifiers: 21
6.2. Generative classifiers: 21
6.3. Example of generative classification 21
6.4. Naïve Bayes 22
6.4.1. Example 1 22
6.4.2. Example 2 23
6.5. Laplacian smoothing 23
6.6. Log usage 23
6.7. Naïve Bayes in sklearn 24
Chapter 7: Decision trees 25
7.1. Building a decision tree 25
7.1.1. Entropy 25
7.1.2. Information gain 25
7.1.3. Gini impurity 26
7.1.4. Constructing an entire tree 26
7.2. Preventing overfitting 26
7.2.1. Tree pruning 26
7.2.2. Random forest tree 26
Chapter 8: Ensemble learning 27
8.1. Stacking 27
8.2. Bagging 27
8.3. Boosting 28
Chapter 9: Clustering 29
9.1. Partitioning methods: K-Means clustering 29
9.1.1. Euclidean distance 30
9.1.2. Step 1: Initializing centroids 30
9.1.3. Step 2: Categorizing samples 30
9.1.4. Step 3: Update centroids 30

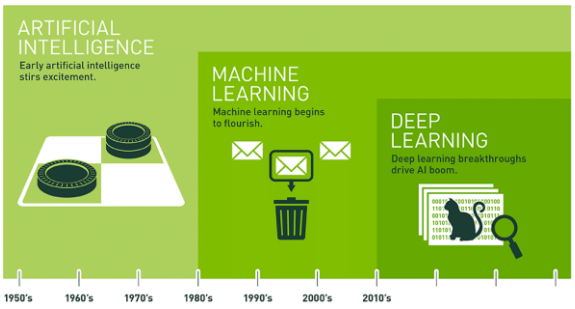
9.1.5. Repeat step 2 and 3 31
9.1.6. Determining the optimal number of clusters 31
9.2. Hierarchical based methods 32
9.2.1. Types of Hierarchical based methods 32
9.2.2. Calculate distances/linkages between clusters 32
Chapter 10: Dimensionality reduction 33
10.1. Principal component analysis (PCA) 33
10.1.1. Step 1: Determine highest covariance 33
10.1.2. Step 2: Determine the eigenvectors 34
10.1.3. Step 3: Rotate axes 34
10.1.4. Step 4: Project on PC1 34
10.1.5. Reconstruction of data using PCA 34
10.1.6. Determining a good number of principal components 34
Chapter 11: Neural networks 35
11.1. Architectures 35
11.1.1. Perceptron 36
11.1.2. Feed forward neural network 36
11.1.3. Deep feed forward neural network 36
11.1.4. XOR example 37
11.1.5. Output layer 37
11.2. Learning algorithm: backpropagation 37
11.2.1. Learning rate 37
11.2.2. Error function (=loss) 38
11.3. Activation functions 38
11.3.1. Step function 38
11.3.2. Linear function (Adaline) 38
11.3.3. Sigmoid function 38
11.3.4. Hyperbolic tangent (tanh) 39
11.3.5. Rectified linear unit (Relu) 39
11.3.6. Leaky Relu 39
11.3.7. What activation function to use 39
11.4. Underfitting and overfitting 39
11.4.1. Drop out 39
11.5. Coding neural networks 40
11.5.1. The sequential model 40
11.5.2. Built-in activation functions 40
11.5.3. Learning rate optimizers 40
11.5.4. Batch modes 40
11.5.5. Loss function 41
11.5.6. Metrics 41
11.5.7. Other parameters 41
11.6. Other architectures 42
11.6.1. Feed forward neural network (FFNN) 42
11.6.2. Autoencoders (AE) 42
11.6.3. Recurrent neural networks (RNN) 43
11.6.4. Convolutional neural networks (CNN) 43
11.6.5. Generative adversarial neural networks (GANN) 44

Chapter 1: Introduction

Machine Learning (ML): computer learning form data to carry out certain automated tasks

Why ML? - Some tasks easy to program → easy algorithms
- Others: no easy algorithm solution → let computer explore the data
- No human errors/biases





Why now? - Faster hardware (RTX 3080)
- Better algorithms (
- More data (More data → better accuracy)
- (Open source) frameworks (Sci Kit Learn, TensorFlow, …)

3 types of ML: - Supervised learning: inputs and outputs given → task driven
- Unsupervised learning: outputs not given → data driven
- Reinforcement learning: decision based on reward → reaction to surroundings

1.1. Supervised learning

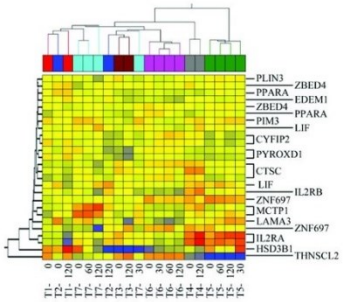
Learning based on labelled training set (algorithm gets the dataset with the correct answers)

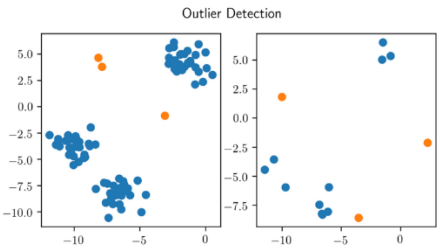
- Regression
- Classification

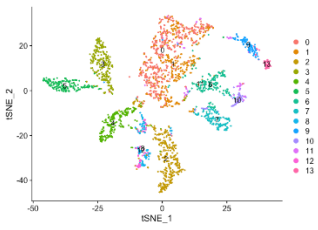
1.2. Unsupervised learning

Learning based on data without correct answers

- Clustering
- Anomaly detection
- Dimensionality reduction



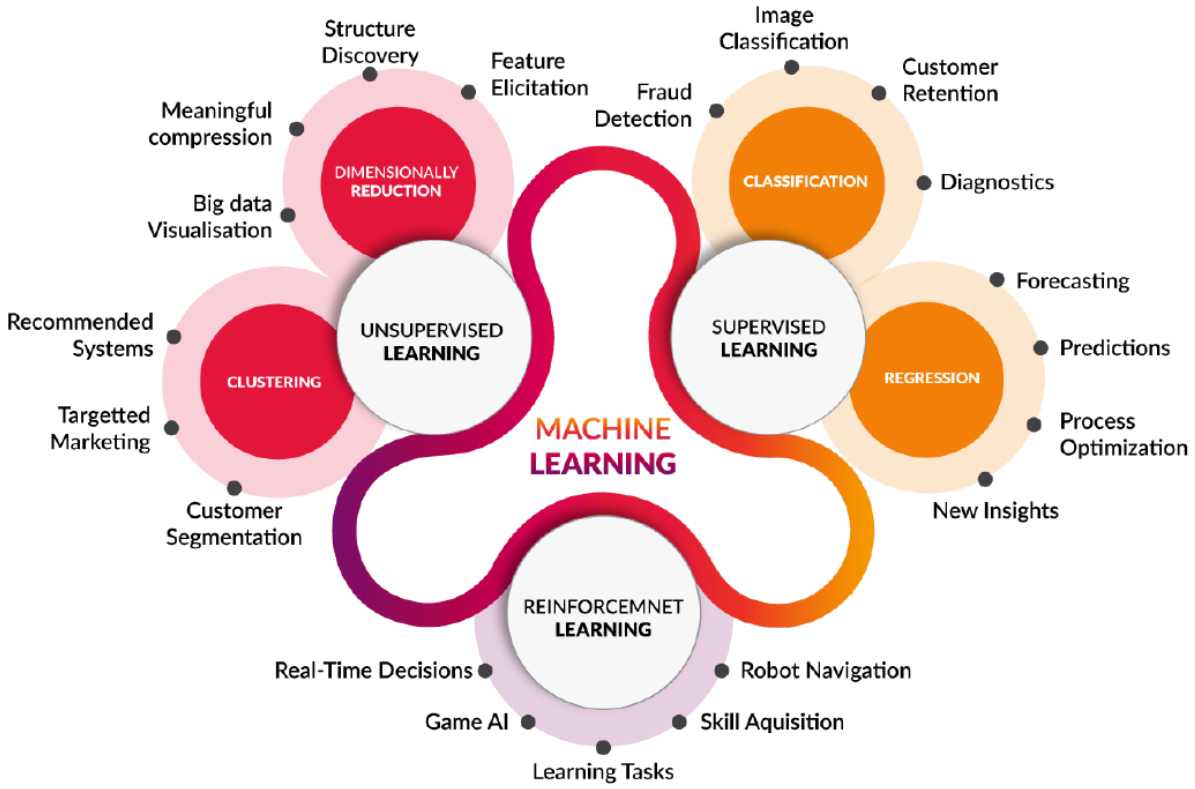




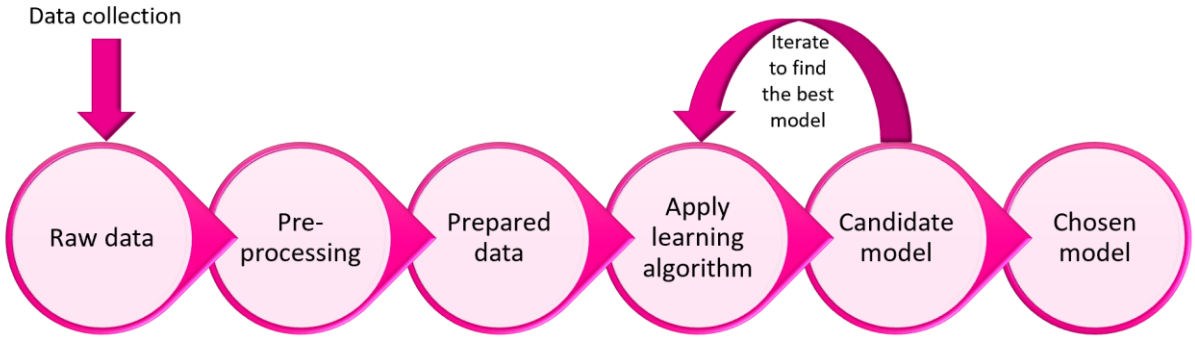
1.3. Reinforcement learning

Learns from trial and error (e.g., a game trying to play itself). Little to no value in bioinformatics

1.4. Overview of ML algorithms



1.5. Machine learning approach



Pre-processing:
- Describe dataset (df.describe())
- Drop unnecessary columns (df.drop("column", axis=1, inplace=True)
- Remove outliers
- Check correlations (pairplot, heatmap)
- Split data in features and targets
- Split data in training set and test set

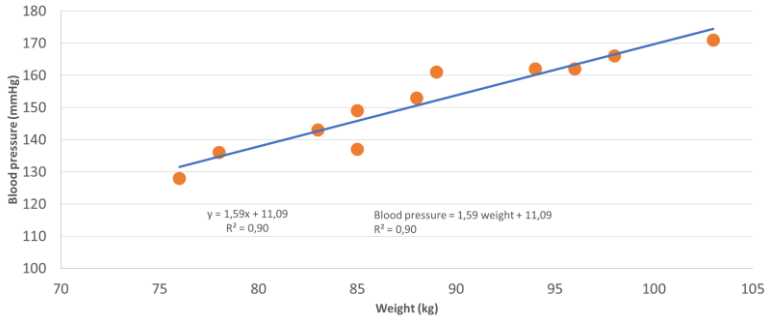
Chapter 2: Regression

Regression: method of modelling a continuous target value based on independent predictors

Features: independent variable to predict the target
Target: continuous variable to predict

2.1. Linear regression

Plot feature against target → plot trendline



The relationship has following formula: $h_\theta(x) = \theta_0 + \theta_1 x$

θ_0 = intercept with the y-axis (value if features are 0)
θ_1 = slope of the trendline

Weights

Finding the best value for the weights → **training/learning**

2.2. Finding best values for weights

Calculate **cost function** via **Least Mean Squares method (LMS)** → find values with least cost

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x_i) - y_i)^2$$

Means: point on regression line minus real value, squared → sum → divided by 2x samples

A cost function plotted is a parabola → bottom of the plot is the best theta value

How to know which theta's the test? → gradient descent

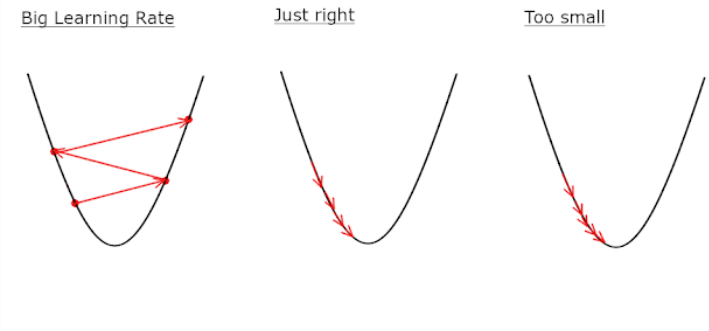
2.2.1. Gradient Descent

Calculate the derivatives from the cost function
Multiply with the learning rate (μ) → result shows if you need to go up or down

2.2.2. Learning rate

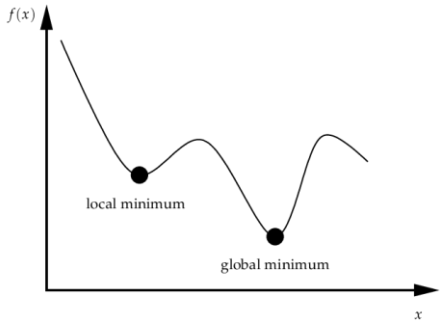
Needs to be optimal.

- If too big it will swing in the cost function
- If too small, it will take too long to learn
- In normal circumstances → computer will update learning rate automatically



2.2.3. Optima

Other problem is learning rate is too small → learning gets stuck in **local optimum**
Sometime needs to increase the learning rate to find the **global optimum**



2.3. Multivariate linear regression

Linear regression with more than 1 feature → just add new theta times x for each feature

$$h_\theta(x) = \theta_0 + \sum_{i=1}^{n} \theta_i x_i$$

2.4. Coding regression

Create an empty model

```
lregmodel=linear_model.LinearRegression()
```

Fit our training data on the model

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

Resulting coefficients and intercept

```
print(f"Coefficients: {lregmodel.coef_}")
print(f"Intercept: {lregmodel.intercept_}")
```

Predict values

```
house = np.array([0.04,80,4.95,0.41,6.63,23.40,5.12,4,245,19.20,396.90,4.70])
price = lregmodel.predict(house.reshape(1,-1))
print(price)
```

2.5. Evaluating a regressor

Evaluation methods: - Mean Absolute Error (MAE)
- Mean Squared Error (MSE)
- Coefficient of determination (R^2 score)

2.5.1. Mean Absolute Error (MAE)

Average of the absolute values between actual and predicted values (mean error to be expected on prediction)

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

```
from sklearn.metrics import mean_absolute_error
y_pred = lregmodel.predict(X_test)
print(f"MAE: {mean_absolute_error(y_test,y_pred)}")
```

2.5.2. Mean Squared error (MSE)

Problem: MAE can become zero if equally distributed over trendline
Solution: MSE → Average of the sum of the squared errors

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

```
from sklearn.metrics import mean_squared_error
y_pred = lregmodel.predict(X_test)
print(f"MSE: {mean_squared_error(y_test,y_pred)}")
```

2.5.3. Coefficient of determination (R^2 score)

R^2 score → proportion of the variability based on the features

$$R^2 = 1 - \frac{MSE}{MAE} = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

Perfect prediction: $R^2 = 1$
Negative R^2: the mean of the target is better than the model itself

```
from sklearn.metrics import r2_score
y_pred = lregmodel.predict(X_test)
print(f"R2: {r2_score(y_test,y_pred)}")
```

2.6. Scaling of values

Different ranges in values → squeezed Gradient Descent → more difficult to train
If all variable in the same range/distribution → easier to train (not a lot of distance between values)

Types of scaling: - Min-max scaling
- Standardized scaling
- Robust scaling

2.6.1. Min-max scaling

Scales all values between 0 and 1

Disadvantages: - Bad with outliers
- Don't use if lot of outliers or not normal distributed

Advantages: - Good on Gaussian distribution with small variance (use if data is normal distributed)
- Retains skew

2.6.2. Standard scaling

Scales all values to a mean of 0 and a standard deviation of 1

Disadvantages: - Not all variable on same scale
- Still possible to have large range

Advantages: - Not affected by many outliers
- Use when there are a lot of outliers or when data is not normal distributed)

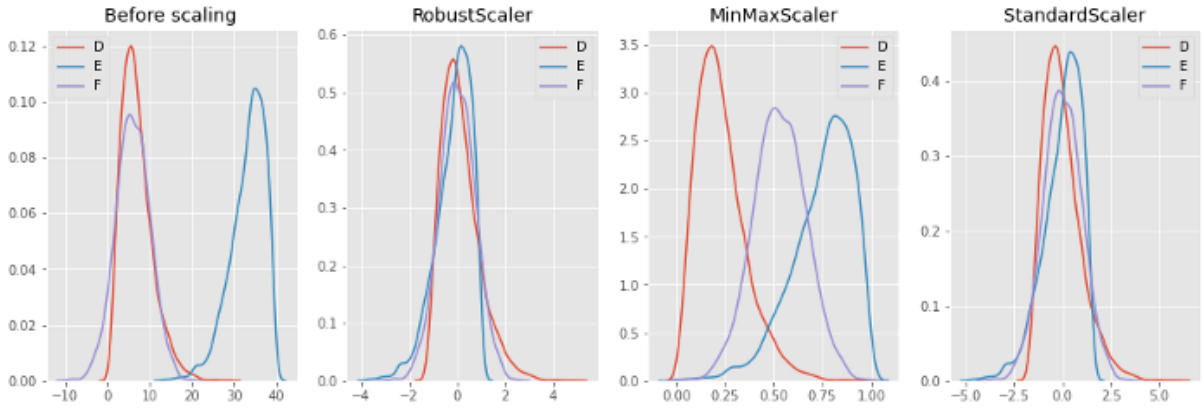
2.6.3. Robust scaling

Like Min-max scaling but uses interquartile distances instead of whole range

Disadvantages: - Not all variable on same scale
- Still possible to have large range (when many outliers)

Advantages: - Not affected by outliers (unless there are a lot)
- Use when large range of not normal distributed data

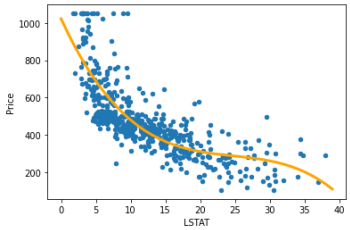
Best way to know which one to use → try them all out!




```
from sklearn.preprocessing import MinMaxScaler, StandardScaler, RobustScaler
#Depending on the scaler that you want:
scaler=MinMaxScaler()
#scaler=StandardScaler()
#scaler=RobustScaler()

scaler.fit(X_train)
#Actually transform the data, both X_train and X_test
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

2.7. Feature engineering

Several techniques: - Feature expansion
- One-hot encoding
- Ordinal encoding



2.7.1. Feature expansion

Add new feature to the dataset: - Calculate new features (area from length and width, …)
- Add new measurements (not always possible)
- Polynomial expansion

Polynomial expansion: linear correlation is not guaranteed → add higher order features

Can be very useful when training a model! Beware of overfitting!

```
From sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(degree=degree)
poly.fit(data)
data = poly.transform(data)
```

2.7.2. One-hot encoding

Machine learning can't learn from strings → convert them to binary/one-hot code

	Fruit	Price
0	apple	5
1	mango	10
2	apple	15
3	orange	20

	Fruit_apple	Fruit_mango	Fruit_orange	price
0	1	0	0	5
1	0	1	0	10
2	1	0	0	15
3	0	0	1	20

```
dataset = pd.concat([dataset,pd.get_dummies(dataset['Fruit'],prefix='Fruit')],axis=1)
dataset.drop(['Fruit'],axis=1,inplace=True)
```

2.8. Overfitting and underfitting

Underfitting: Model can't predict the training AND the test set

- Model is too simple
- Model has a high bias

→ Add new features, e.g., polynomial expansion

→ Add new data

Overfitting: Model can predict training very well, but not the test set

- Model is too complex (learns training set by heart)
- Noise of individual datapoints is picked up
- Model has a high variance

→ Add new data

→ Remove features

→ Regularization

2.9. Regularization

Regularization: Prevent overfitting of the model

→ adding an extra term to the cost function

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x_i) - y_i)^2 + \boldsymbol{R}(\boldsymbol{\theta})$$

Three types of regularization: - Ridge regression (L2 regularization)

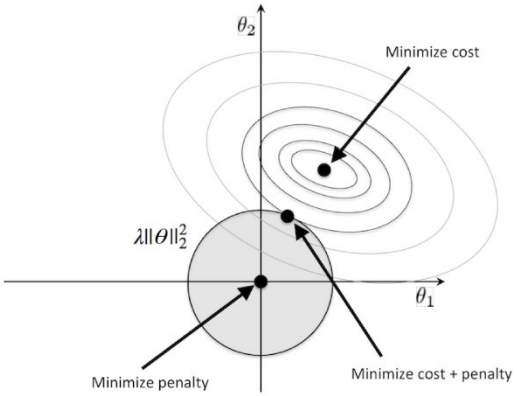
- Lasso regression (L1 regularization)
- ElasticNet regression (L1 + L2 regularization)

2.9.1. Ridge regression (L2)

Regulates the sum of the squares of theta with tuning parameter lambda/alpha

$$R(\theta) = \lambda_2 \sum_{j=1}^{m} \theta_j^2$$

Prevents the Gradient descent of going to the global optimum

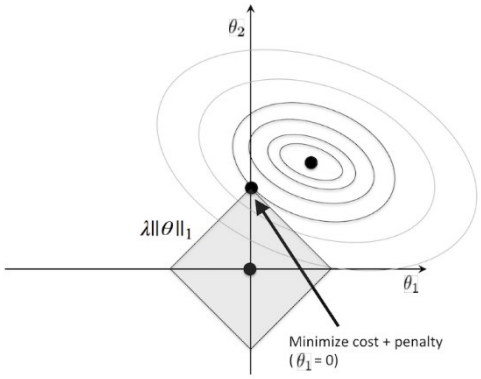


2.9.2. Lasso regression (L1)

Regulates the sum of absolute values

$$R(\theta) = \lambda_1 \sum_{j=1}^{m} |\theta_j|$$

Prevents the Gradient descent of going to the global optimum



2.9.3. Hyperparameter λ/α

λ = The tuning parameter (regulates the allowed complexity)

α = Same parameter, but used in Sci Kit Learn

- Lower value of λ: lower bias, higher variance (overfitting)
- Higher value of λ: higher bias, lower variance (underfitting

2.9.4. Regularization in code

Ridge regression and Lasso regression are different models that can be trained like linear regression

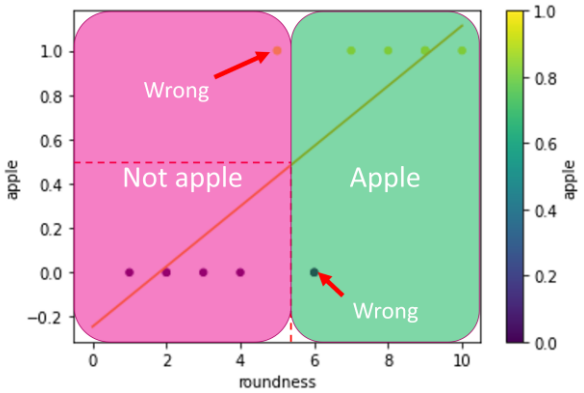
```
from sklearn.linear_model import Lasso, Ridge
alpha=0.1
# fit_intercept: use a theta0 or not (most cases: True)
Lregmodel = Lasso(alpha=alpha, fit_intercept=True)
Lregmodel.fit(X_train, y_train)
score = Lregmodel.score(X_test,y_test)
Rregmodel = Ridge(alpha=alpha, fit_intercept=True)
Rregmodel.fit(X_train, y_train)
score = Rregmodel.score(X_test,y_test)
```

Chapter 3: Classification

Classification: predictive modelling problem → class prediction based on input data

Types of classifiers: - Binary/binomial classifier: sample divided in 2 groups (yes or no)
- Multiclass classifier: samples divided in 2+ groups (colours, cloths, …)
- Multilabel classifier: multiple labels per sample (content analysis)

Why not regression? → outliers are affected by cut-offs/thresholds
→ Values can be higher than the label of the target (0/1)



Solution: Model fixating outcome between 0 and 1
Closer to 1 (>0.5) → Probable class 1
Closer to 0 (<0.5) → Probable class 2

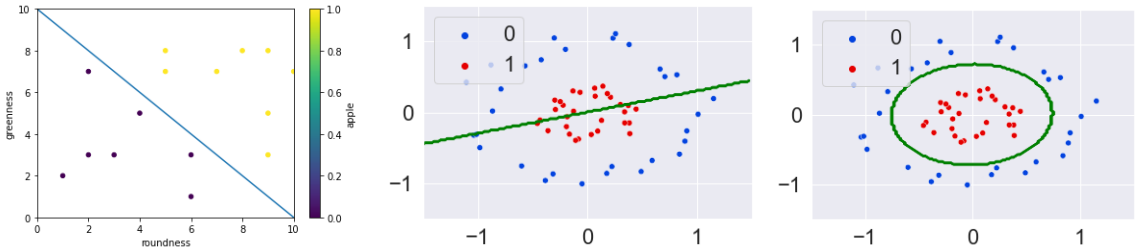
→ **Logistic regression**

3.1. Logistic regression

Fixate linear regression to values between 0 and 1:

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}$$

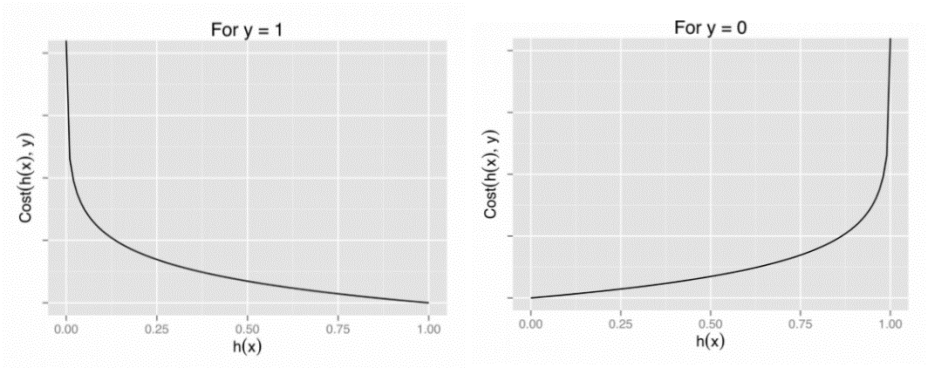
$$\theta^T x = \theta_0 + \theta_1 * x1 + \theta_2 * x2$$



In case of non-linear data → use polynomial expansion!

3.2. Cost function

$$J(\theta) = \begin{cases} -\ln\left(h_\theta(x)\right) & if\ y = 1 \\ -\ln\left(1 - h_\theta(x)\right) & if\ y = 0 \end{cases}$$



$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^i \ln\left(h_\theta(x^i)\right) + (1-y^i)\ln\left(1-h_\theta(x^i)\right)\right]$$

3.3. Logistic regression in code

3.3.1. The model

```
# Initialise the model
logreg_model=linear_model.LogisticRegression(penalty="l2", C=1e5) # penalty here is not required since
default is l2
# Fit the data to the model
logreg_model.fit(features,targets)
# Take a look at the score
score = logreg_model.score(features,targets)
print(f"Score: {score}") # This is the accuracy of the model
# Print coefficients and intercept
print("Coefficients: ", coefs:=logreg_model.coef_[0])
print("Intercept: ", intercept:=logreg_model.intercept_)
# Classify a new sample
# Roundness = 8 and greenness = 6:
print("Roundness = 8 and greenness = 6:")
sample=np.array([8,6]).reshape(1,-1)
# Reshape will turn [8,6] into [[8,6]] which is what the model expects
print("Apple:", logreg_model.predict(sample))
probabilities = logreg_model.predict_proba(sample)
print("Probability [chance for not apple, chance for apple]", probabilities)
# Roundness = 4 and greenness = 4:
print("Roundness = 4 and greenness = 4:")
sample=np.array([4,4]).reshape(1,-1)
print("Apple:", logreg_model.predict(sample))
probabilities = logreg_model.predict_proba(sample)
print("Probability [chance for not apple, chance for apple]", probabilities)
```

3.3.2. Polynomial expansion

```
from sklearn.preprocessing import PolynomialFeatures

X = features.values
y= targets.values
# For example third degree features:
degree=3

# Initialize the polynomial features function
poly=PolynomialFeatures(degree)
# Fit and transform the data (could be split into two lines=> fit, transform)
features_poly = poly.fit_transform(X)

# This will add all possible combinations up to three degrees: Roundness=R and greenness = G:
# R, G, R², G², RG, R³, R²G, RG², G³
# First column is a column of ones (1) representing the 0th power.
# You can disable it with the parameter include_bias=False
# In general we don't change this.
print(features_poly)

logreg_poly = linear_model.LogisticRegression(C=1,max_iter=1000)
logreg_poly.fit(features_poly, y)
```

3.4. Overfitting and underfitting

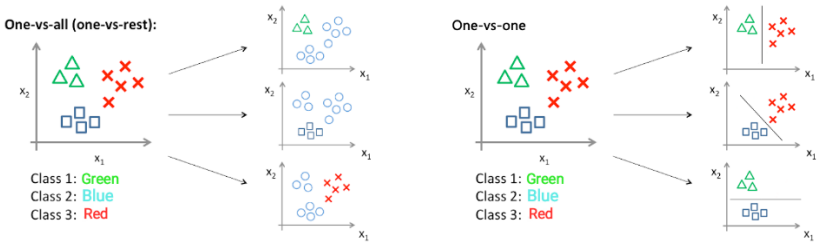
Underfitting: model too simple to explain variance (bad accuracy on training and test set)
Overfitting: model too good to be true (good accuracy on training set, but bad on test set)

Complexity of model is regulated by: - Penalty: regularization (L1 or L2)
- C hyperparameter

C hyperparameter: - Increase → more complex model (but risk of overfitting)
- Decrease → simpler model (but risk of underfitting

3.5. Multi-class classification

Until now → only 2 classes, but what if more? → 2 strategies: - One-vs-all/one-vs-rest
- One-vs-one



3.5.1. One versus all

Each class is classified and separated from the rest: - Part of class? Yes or no
- Total # classifiers for N-classes: N
- More sensitive for non-balanced data

3.5.2. One versus one

Each class is classified against each class separately: - Part of class 1 or part of class 2?
- Total # classifiers for N-classes: $\frac{N(N-1)}{2}$
- Less sensitive for non-balanced data
- More computationally intensive

3.5.3. Multiclass classification in code

One versus one

```
from sklearn.multiclass import OneVsOneClassifier
# Create a logistic regression model like normal
lrModel = LogisticRegression(C=0.1)
# Wrap it in the OneVsOneClassifier function
ovo = OneVsOneClassifier(lrModel)
# Fit the data to the OneVsOneClassifier
ovo.fit(X_train,y_train)
# Predict based on the trained model
y_pred= ovo.predict(X_test)
```

One versus all

```
lrModel = LogisticRegression(C=0.1, multi_class="ovr")
```

3.6. Evaluating a classifier

Evaluation methods: - Confusion matrix and metrics
- Receiving Operating Characteristic (ROC) curve
- Area Under Curve (AUC)

3.6.1. Confusion matrix and metrics

Metrics in classifiers always start from a **confusion matrix**: table with number of predicted and true positive and negative values.

		Predicted	
		Negative	Positive
Real	Negative	TN	FP
	Positive	FN	TP

TN: True Negatives
TP: True Positives
FN: False negatives (predicted – but reality +)
FP: False Positive (predicted + but reality –)

All other metrics are calculated from these values

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN}$$

$$FPR = 1 - specificity = \frac{FP}{TN + FP}$$

$$TPR = Recall = Sensitivity = \frac{TP}{TP + FN}$$

$$Precision = \frac{TP}{TP + FP}$$

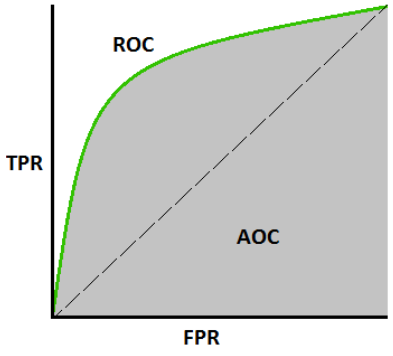
$$F1\ score = 2 * \frac{recall * precision}{recall + precision}$$

FNR and TNR can also be calculated. Precision and F1 can also be calculated for the negative cases.

```
print(cf:=confusion_matrix(targets,y_pred)) # Note the use of the walrus operator.
tn, fp, fn, tp = cf.ravel()
from sklearn.metrics import ConfusionMatrixDisplay
matrix = ConfusionMatrixDisplay(cf,display_labels=["Negative","Positive"])
matrix.plot()
plt.show()
print(f"Accuracy: {accuracy_score(targets,y_pred)}\n")
print(f"Recall (positive): {recall_score(targets,y_pred)}")
print(f"Recall (negative): {recall_score(targets,y_pred,pos_label=0)}\n")
print(f"Precision (positive): {precision_score(targets,y_pred)}")
print(f"Precision (negative): {precision_score(targets,y_pred,pos_label=0)}\n")
print(f"F1 score (positive): {f1_score(targets,y_pred)}")
print(f"F1 score (negative): {f1_score(targets,y_pred,pos_label=0)}")
```

3.6.2. Receiving Operating Characteristic

ROC is a plot illustrating the diagnostic ability of a binary classifier system as its discrimination threshold is varied between 0 and 1.



X-axis: FPR (false positive rate) for each threshold
Y-axis: TPR (Recall/Sensitivity) for each threshold

AUC: metric for comparing TPR and FPR
> AUC = high TPR, low FPR (= best model)

ROC-heaven: best point is found at [0,1] (upper left corner)
ROC-hell: everything is classified wrong (solution: switch classes)

Chapter 4: Hyperparameter tuning

Examples of hyperparameters:
- α
- C
- Penalty (L1, L2)
- Solver (= kernel)
- Learning rate
- Number of epochs
- Number of branches in decision trees
- Number of clusters in clustering

} Next chapters

4.1. Automated hyperparameter tuning

Possible strategies:
- For loop over the parameters
- **Grid Search**
- **Random Search**
- **Bayesian Optimization**
- Gradient Based optimization
- Evolutionary Optimization

4.1.1. Grid Search

Tries all possible combinations of given hyperparameters

Example:

```
parameters = [   {'kernel': ['linear'], 'C': np.linspace(0.01,20,10)},
{'kernel': ['rbf'], 'C': np.linspace(0.01,20,10), 'gamma': [0.0001, 0.001, 0.01, 0.1, 0.2]},
{'kernel': ['poly'], 'C': np.linspace(0.01,20,10)} ]
```

The above grid makes 70 possibilities.

Problem: always same training set → can lead to overfitting
Solution: cross validation

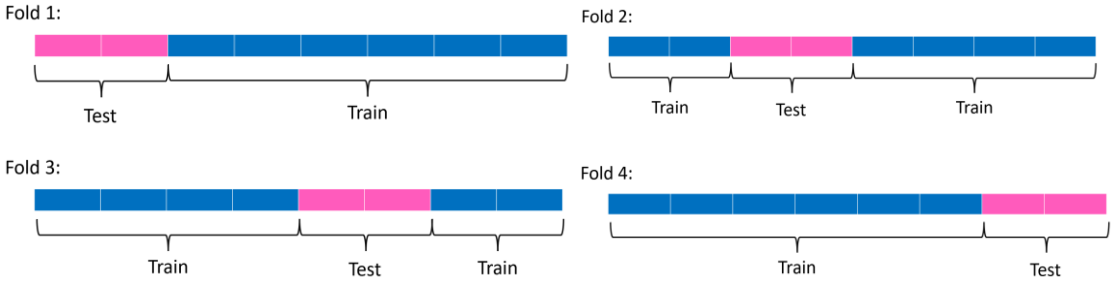
4.1.2. Cross validation (k-fold)

After splitting data in training and test set → split training set k-times in k-parts (leave test set out)

Example:
Data: 

Training: 

Test:  (not a part of k-fold validation)



For each fold: a score for the test/validation set is given. The best or mean score is retained.

What is the best k-value? - Higher folds → less bias, but large variance (risk of overfitting)
- Fewer folds → similar behaviour as test split
- Ideally: 5-10 folds (depending on data)

4.1.3. Random Search

The parameters are randomly selected out of a grid, specifying ranges of values.

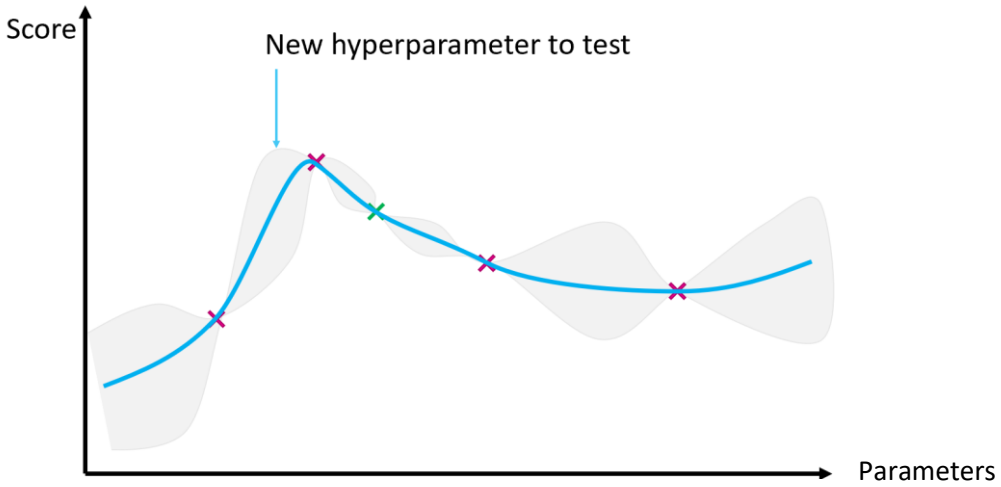
Example:

```
parameters = {'solver': ['lbfgs','liblinear'], 'C': uniform(0.01, 20), 'penalty': ['L1','L2']}
```

In the cross validation you need to specify the number of models to try with n_iter

4.1.4. Bayes Optimization

Starts as a random search and will perform extra optimization around the best parameters
Probabilistic model of the objective function (based on chances it calculates new parameters)
Very efficient and effective!



Chapter 5: Unbalanced data

Unbalanced data: data where classes are not equally distributed

Example: 39 benign diagnoses
5 malignant diagnoses

Problem with unbalanced data: When training → majority is of one class → classifying as this class already give high accuracy but recall for minority class is low.

Possible solutions:
- Hyperparameter tuning
- Collect more data
- Undersampling majority class
- Oversampling minority class
- Change scorer
- Class-weight balancing
- Data augmentation

5.1. Hyperparameter tuning

Not always effective with unbalanced data → might cause overfitting on minority class

5.2. Collect more data

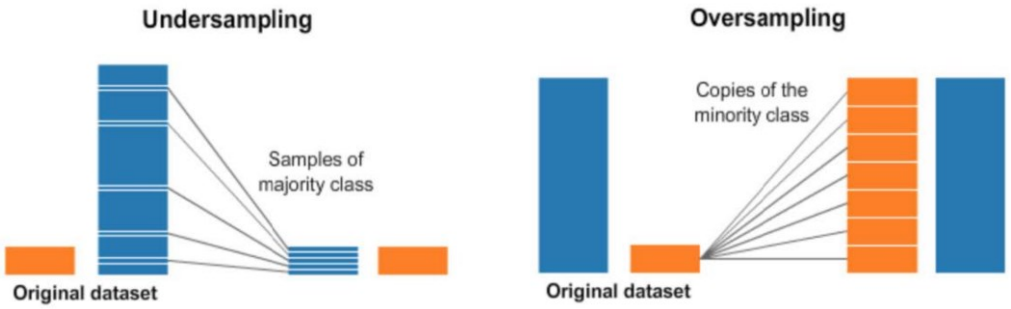
More data always gives better models!
Add more samples of minority class (not always possible)

5.3. Undersampling majority class

Level the majority class down to equal to the minority class
Disadvantage: you lose a lot of data to train and improve your model with.

5.4. Oversampling minority class

Take the minority multiple times in your dataset
Disadvantage: might cause overfitting on minority class



5.5. Change scoring system

Default a cross validation uses 'accuracy' as default scoring system.

Other possible scoring systems:
- Recall
- F1 score
- Fβ score
- Balanced_accuracy (average of recalls)
- See more on website of sklearn

5.6. Class-weight balancing

By default: error of both classes contributes equally to scores
You can give the error for the minority class a higher weight → more costly to make a mistake

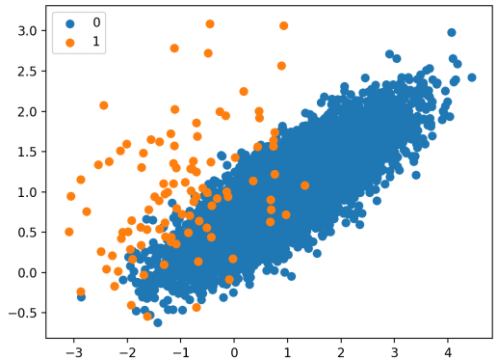
Most models have a parameter called *class_weight*

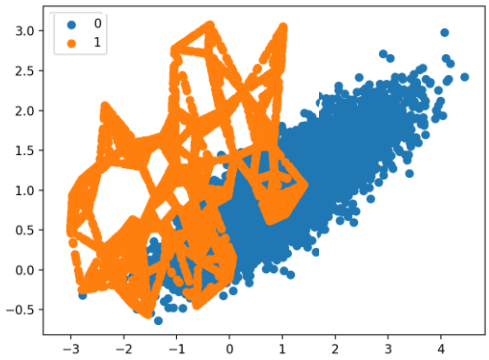
```
model=LogisticRegression(class_weight='value here')
# Value can be:
class_weight="balanced"
class_weight=[1,9] #first encountered class has weight 1, the second weight 2
class_weight={
        "Name of class 1": 1,
        "Name of class 2": 9
}
```

5.7. Data augmentation

Impute values to level out amount of minority class by trying to predict possible values

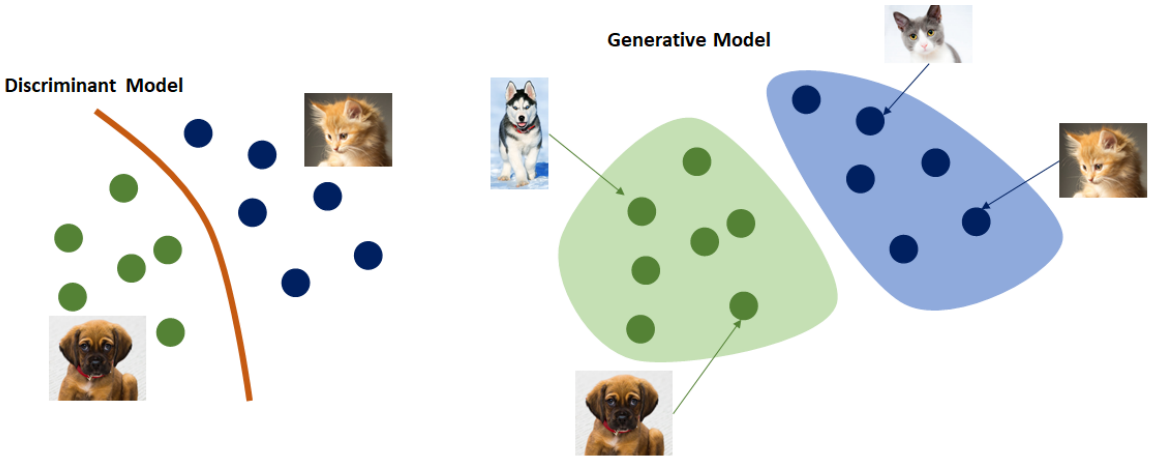
Like oversampling minority class, but the data points are not the same





Chapter 6: Naïve Bayes

Discriminative vs generative classifiers:



6.1. Discriminative classifiers:

Classify data, based on a decision boundary
No idea on distributions of the classes
No idea what it means to be of that class
Learns $P(y|X)$ = called **posterior**
(What is the chance our sample belongs to class "y", given features "X")
Examples: logistic regression, support vector machines, decision trees

6.2. Generative classifiers:

Classify data based on distributions and meaning of features
Learns the distributions of the classes
Learns what it means to be of a specific class
Learns $P(X|y)$ → what is the chance of having features "X" if this is class "y" (**likelihood**)
Learns $P(y)$ → what is the chance of being class "y" (**prior**)
Learns $P(X)$ → what is the chance of having feature "X" (**marginal**)
Calculated **posterior**: $P(y|X) = \frac{prior * likelihood}{marginal} = \frac{P(y)*P(X|y)}{P(X)}$
Examples: Naïve Bayes; Hidden Markov Models, Variational autoencoders

6.3. Example of generative classification

Chance a person has cancer: 1% $P(cancer) = prior$
Test is 90% positive when person has cancer: $P(positivetest|cancer) = likelihood$
Test is 90% negative when person has no cancer

What is chance a person has cancer when test is positive? $P(cancer|testpositive) = posterior$

$$P(cancer|testpositive) = \frac{P(cancer) * P(testpositive|cancer)}{P(testpositive)} = \frac{0{,}01 * 0{,}90}{0{,}108} = 0{,}0833$$

P(testpositive) is calculated by looking at positive tests in positive (90% of 1%) and negative people (10% of 99%): 0,9 * 0,01 + 0,99 * 0,10 = 0,108

6.4. Naïve Bayes

Above example is quite simple, but it is difficult with a lot of features

outlook	temperature	humidity	windy	play
overcast	hot	high	FALSE	yes
overcast	cool	normal	TRUE	yes
overcast	mild	high	TRUE	yes
overcast	hot	normal	FALSE	yes
rainy	mild	high	FALSE	yes
rainy	cool	normal	FALSE	yes
rainy	cool	normal	TRUE	no
rainy	mild	normal	FALSE	yes
rainy	mild	high	TRUE	no
sunny	hot	high	FALSE	no
sunny	hot	high	TRUE	no
sunny	mild	high	FALSE	no
sunny	cool	normal	FALSE	yes
sunny	mild	normal	TRUE	yes

o=outlook, t=temperature, h=humidity, w=windy

$$P(play|o,t,h,w) = P(play) * \frac{P(o,t,h,w|play)}{P(\text{o,t,h,w})}$$

$$\Rightarrow P(play) * \frac{P(o|t,h,w,play) * P(t|o,h,w,play) * P(h|o,t,w,play) * P(w|o,t,h,play)}{P(\text{o,t,h,w})}$$

$P(o|t,h,w,play)$ is the probability that we have a certain outlook (e.g., sunny) based on all possible combinations of the features + target

Difficult to calulate → Naive Bayes → only target feature, no combinations of features

$$P(play|o,t,h,w) = P(play) * \frac{P(o,t,h,w|play)}{P(\text{o,t,h,w})}$$

According to naïve bayes:

$$P(play|o,t,h,w) = P(play) * \frac{P(o|play) * P(t|play) * P(h|play) * P(w|play)}{P(\text{o, t, h, w})}$$

General formula:

$$P(y|x) = P(y) * \frac{\prod_{i=1}^{n} P(x_i|y)}{P(x)}$$

P(X) is always the same, therefore it doesn't need to be calculated. If P(X | y) raises, so does P(y | X)

Final formula: $P(y|X) \propto P(y) * \prod_{i=1}^{n} P(X_i|y)$ (means proportional to)

play	outlook	counts	Formula P(outlook|play)	P(outlook|play)
no	rainy	2	$\frac{\#outlook=rainy}{\#play=no} = \frac{2}{5}$	0,40
no	sunny	3	$\frac{\#outlook=sunny}{\#play=no} = \frac{3}{5}$	0,60
yes	overcast	4	$\frac{\#outlook=overcast}{\#play=yes} = \frac{4}{9}$	0,44
yes	rainy	3	$\frac{\#outlook=rainy}{\#play=yes} = \frac{3}{9}$	0,33
yes	sunny	2	$\frac{\#outlook=sunny}{\#play=yes} = \frac{2}{9}$	0,22

play	temperature	counts	P(temperature|play)
no	cool	1	0,20
no	hot	2	0,40
no	mild	2	0,40
yes	cool	3	0,33
yes	hot	2	0,22
yes	mild	4	0,44

play	humidity	counts	P(humidity|play)
no	high	4	0,80
no	normal	1	0,20
yes	high	3	0,33
yes	normal	6	0,67

play	windy	counts	P(windy|play)
no	FALSE	2	0,40
no	TRUE	3	0,60
yes	FALSE	6	0,67
yes	TRUE	3	0,33

6.4.1. Example 1

Will we play on a sunny, hot day with high humidity and no wind?

Test for target = yes:

$$P(play = yes) * P(sunny|yes) * P(hot|yes) * P(high|yes) * P(false|yes)$$
$$= \frac{9}{14} * 0{,}22 * 0{,}22 * 0{,}33 * 0{,}67 = 0{,}0070$$

Test for target = no

$$P(play = no) * P(sunny|no) * P(hot|no) * P(high|no) * P(false|no)$$
$$= \frac{5}{14} * 0{,}60 * 0{,}40 * 0{,}80 * 0{,}40 = 0{,}0492$$

$$P(yes|X) < P(no|X)$$

Model will say that it predicts to **not play**

6.4.2. Example 2

Play on an overcast hot day with high humidity and no wind?

Target = yes: $\frac{9}{14} * 0{,}44 * 0{,}22 * 0{,}33 * 0{,}67 = 0{,}0138$

Target = no: $\frac{5}{14} * 0 * 0{,}4 * 0{,}8 * 0{,}4 = 0$

Problem: because no situation of not playing when overcast in training → chance is 0 → not helpful

6.5. Laplacian smoothing

Prevent having 0 chance values → adding small chance to unseen values → adding +1 to every occurrence between feature and target (Syn. Add-one smoothing).

play	outlook	counts	Formula P(outlook|play)	P(outlook|play)
no	overcast	0+1	$\frac{\#outlook=overcast}{\#play=no} = \frac{0+1}{5+1*3}$	0,125
no	rainy	2+1	$\frac{\#outlook=rainy}{\#play=no} = \frac{2+1}{5+1*3}$	0,375
no	sunny	3+1	$\frac{\#outlook=sunny}{\#play=no} = \frac{3+1}{5+1*3}$	0,50
yes	overcast	4+1	$\frac{\#outlook=overcast}{\#play=yes} = \frac{4+1}{9+1*3}$	0,42
yes	rainy	3+1	$\frac{\#outlook=rainy}{\#play=yes} = \frac{3+1}{9+1*3}$	0,33
yes	sunny	2+1	$\frac{\#outlook=sunny}{\#play=yes} = \frac{2+1}{9+1*3}$	0,25

Also, on denominator add 1 times of play occurrences

Back to example 2:

Target = yes: $\frac{9+1}{14+1*2} * \frac{2+1}{9+1*3} * \frac{2+1}{9+1*3} * \frac{3+1}{9+1*2} * \frac{6+1}{9+1*2} = 0.0090$

Target = no: $\frac{5+1}{14+1*2} * \frac{0+1}{5+1*3} * \frac{2+1}{5+1*3} * \frac{4+1}{5+1*2} * \frac{2+1}{5+1*2} = 0.0054$

P(target=no | X) < P(target=yes | X) ➔ play is more likely

In sklearn Laplacian smoothing is an α value

6.6. Log usage

Problem: - How large dataset, how smaller the chances (especially in sparse datasets)
- Risk of underflow: computer just converts it to 0

Solution: Use log function to prevent this underflow

$$\log(P(y|X)) \propto \log(P(y) * \prod_{i=1}^{n} \log(P(Xi|y))$$

But

$$\log(x * y) = \log x + \log y$$

Thus

$$\log\big(P(y|X)\big) \propto \log P(y) + \sum_{i=1}^{n} \log P(Xi|y)$$

6.7. Naïve Bayes in sklearn

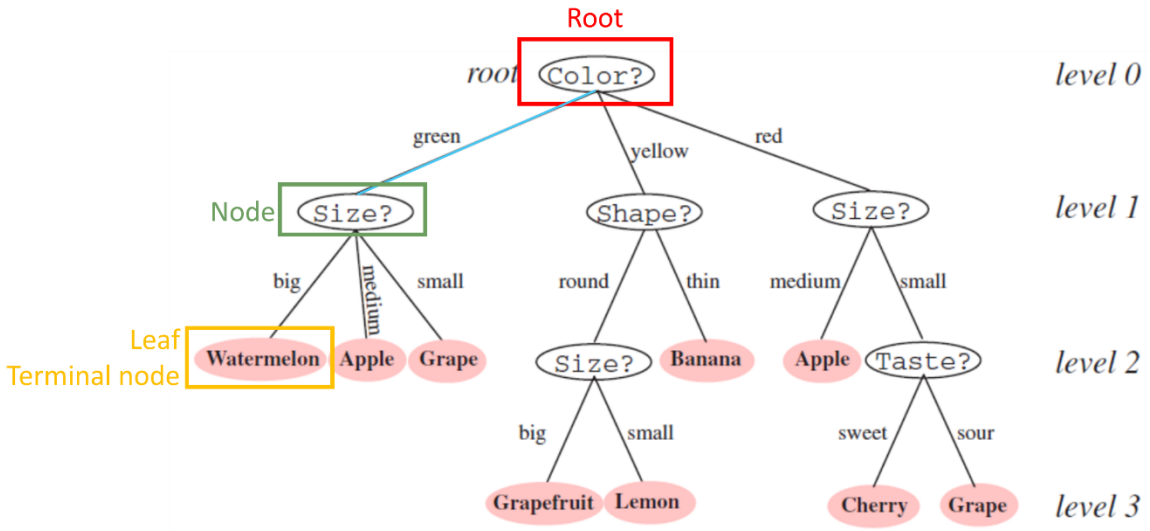
Same as with all models in sklearn

```
model = MultinomialNB(alpha=1)
model.fit(X_train,y_train)
print("Test score:",model.score(X_test,y_test))
```

Naïve bayes can also be used on numerical targets.

Alpha can never be 0 → it will automatically set to a very small value

Chapter 7: Decision trees



Benefits: - Interpretability: a tree can be expressed as a logical expression
- Rapid classification: sequence of simple queries
- High accuracy and speed

Disadvantages: Prone to overfit!!!

7.1. Building a decision tree

Three metrics are used:- **Entropy**: disorder of data (more disorder → higher entropy)
- **Information Gain**: if split by specific rule, what is the information we get?
- **Gini impurity**: chance of classifying wrongly

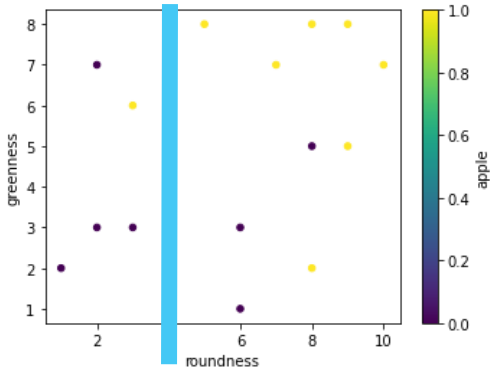
7.1.1. Entropy

Disorder of data expressed as the negative sum of the chance to have a value time log 2 that value

$$H = \sum_{i=1}^{n} P_i \log_2 \frac{1}{P_i} = -\sum_{i=1}^{n} P_i \log_2 P_i$$

7.1.2. Information gain

Expressed as the entropy before split (see above) minus the entropy after split



Left:

$P_{apple} = \frac{1}{5} = 0{,}20 \quad and \quad P_{notapple} = \frac{4}{5} = 0{,}80$

$H_{left} = -(0{,}20 * \log_2(0{,}20) + 0{,}80 * \log_2(0{,}80)) = 0{,}72$

Right:

$P_{apple} = \frac{7}{10} = 0{,}70 \quad and \quad P_{notapple} = \frac{3}{10} = 0{,}30$

$H_{right} = -(0{,}70 * \log_2(0{,}70) + 0{,}30 * \log_2(0{,}30)) = 0{,}88$

$$H_{total\ after\ split} = \frac{\#\ left}{\#\ total} * H_{left} + \frac{\#\ right}{\#\ total} * H_{right} = 0{,}8267$$

$$Information\ gain = H_{before\ split} - H_{after\ split} = 0{,}9974 - 0{,}8267 = 0{,}1707$$

Do the same for greenness is 4 → information gain = 0,2274 → higher information gain → best split

7.1.3. Gini impurity

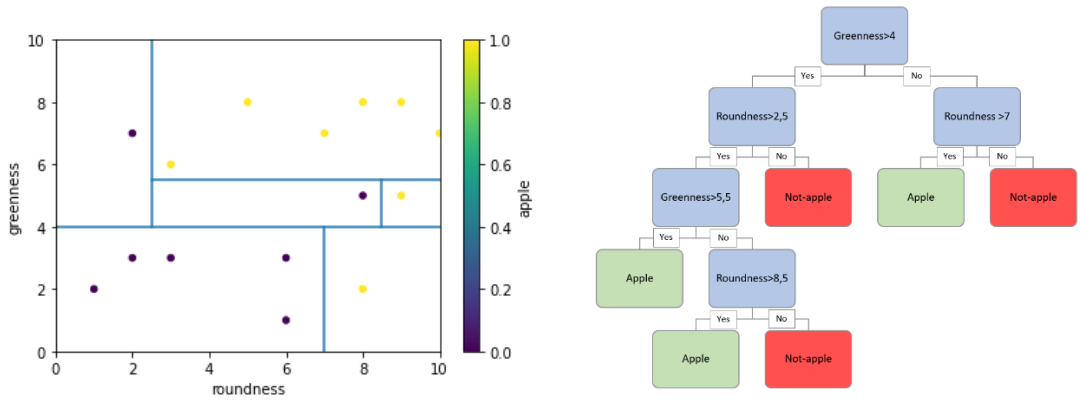
Chance of classifying something wrong based on the splits made through the training set

$$G = 1 - \sum_{i=1}^{n} P_i^2$$

After split calculated a weighted G (repeat for other features) → smallest G → best split

7.1.4. Constructing an entire tree

Keep on splitting after a previous split, until all classes are split perfect or near perfect
Splitting until perfect split → problem of **overfitting**!!!



7.2. Preventing overfitting

Methods for preventing overfitting: - Tree pruning
- Random Forest tree

7.2.1. Tree pruning

Not allowing your tree to fully separate all cases in the tree

This can be done with hyperparameter tuning on 2 levels: pre and post pruning

Pre-pruning - max_depth: number of layers your tree is allowed to have
- min_samples_leaf: minimal number of samples a leaf should have
- min_samples_split: minimal number of samples your splits should have

Post-pruning - ccp_alpha: complexity parameter. If tree is to complex → cut some branches

7.2.2. Random forest tree

Working with multiple trees (a forest) on subsets of the training data (random)
The class is chosen by the class that is predicted by most of the trees (majority voting)

The size of the dataset is set with the hyperparameter "max_samples"

Can also be used for regression (average value of all trees)

Chapter 8: Ensemble learning

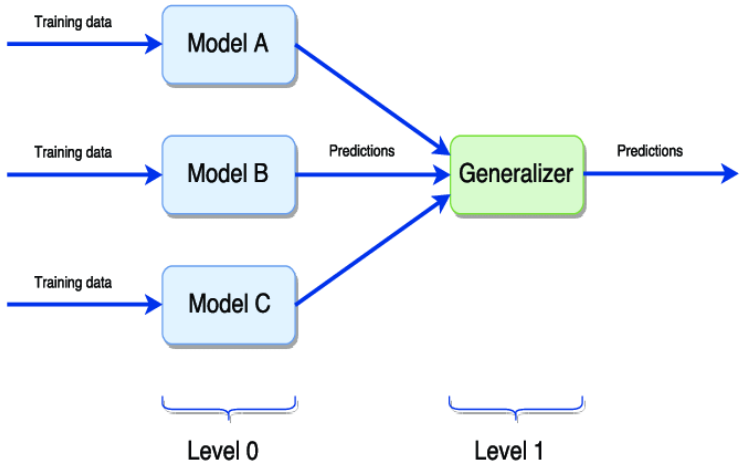
Use multiple models → often better performance than single model
Random forest tree is an example of ensemble learning

3 types of ensemble learning: - Stacking
- Bagging
- Boosting

8.1. Stacking

Train multiple (different models) on the same training data → to final model → prediction final class

Example: random forest tree without bagging (bagging=False)

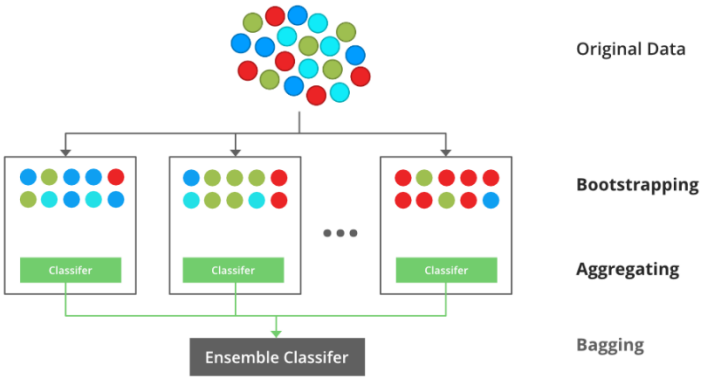



```
from sklearn.ensemble import StackingClassifier
from sklearn.naive_bayes import MultinomialNB
from sklearn.tree import DecisionTreeClassifier


estimators = [('logisticModel',LogisticRegression(C=10, solver='liblinear')),
              ('nbModel',MultinomialNB(alpha=1)),
              ('dtModel',DecisionTreeClassifier(max_depth=1))]

stackClassifier = StackingClassifier(estimators=estimators)
# Fit data
stackClassifier.fit(X_train,y_train)
# Predict testing data
y_pred = stackClassifier.predict(X_test)
```

8.2. Bagging

Short for: **Bootstrapping Aggregating**
Bootstrapping: getting bags of different subsets of the data
Aggregating: combining output of different models into a final classifier

Example: random forest tree with bagging (bagging=True)



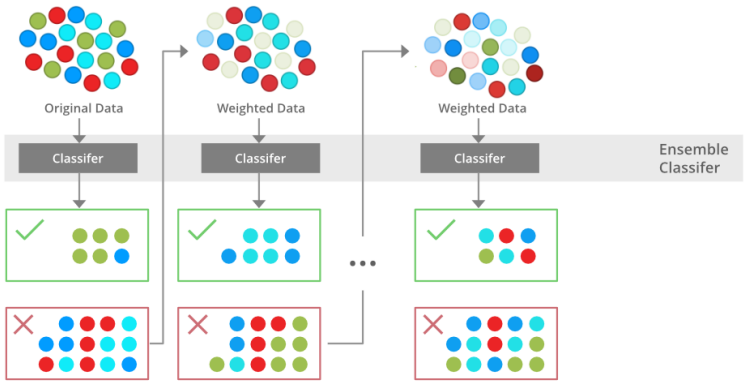

```
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import BaggingClassifier
# Bagging with logistic regression
number_of_estimators = 100
baseModel = LogisticRegression(C=5,solver='liblinear')
lregbagging = BaggingClassifier(base_estimator=baseModel,
                                n_estimators=number_of_estimators,
                                max_features=2)
# Fit our training data
lregbagging.fit(X_train,y_train)
# Predict testing data
y_pred = lregbagging.predict(X_test)
```

8.3. Boosting

Like bagging → also different bags of subsets, but wrongly classified samples will have a higher chance of being selected in the bag of the next classifier.

Disadvantages: - Cannot be done parallel, because is a serial ensemble classifier.
- Prone to overfitting

Example: adaboost (adaptive boosting)



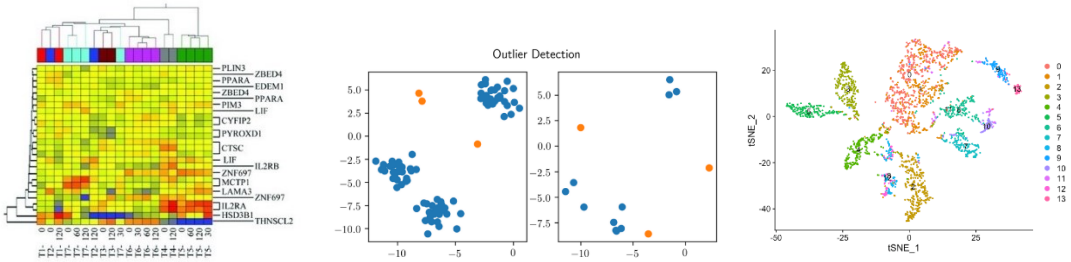

```
from sklearn.ensemble import AdaBoostClassifier
# When not specifying a specific base estimator, a DecisionTreeClassifier is used (One tree).
clf_adaboost = AdaBoostClassifier(n_estimators=150,learning_rate=0.9)
# Fit data
clf_adaboost.fit(X_train,y_train)
# Predict testing data
y_pred = clf_adaboost.predict(X_test)
```

Chapter 9: Clustering

Clustering (and dimensionality reduction) are a part of unsupervised learning

Unsupervised learning: unlabeled training set (no correct answers) → find groups in data

- Find similar data groups (clusters)
- Find outliers in data (anomaly detection)
- Reduce the amount of data, but keep information (dimensionality reduction)



Applications: - Related search results on Google
- Related news articles
- Market segmentations
- Social media clustering
- Image segmentation (image analysis)
- Sequence analysis (phylogenetic trees)
- High throughput genotyping (BIT09)

Clustering: divide data into groups that fit together with different methods

- Partitioning methods
- Hierarchical based methods
- Density based methods
- Grid based methods

9.1. Partitioning methods: K-Means clustering

Partition the samples into k-clusters and each partition forms one cluster, often based on distance between samples as partition criterion (e.g., K-means clustering)

Three major steps: 1. Initialization centroids
2. Categorizing samples
3. Update centroids

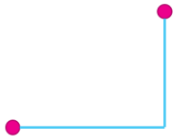
Characteristics: - Easy to use and easy to interpret
- Can get stuck at local optima (result depends on initialize centroid, try few models)
- Very sensitive to outliers
- Problem with nonlinear data → use other algorithms (spectral clustering)
- Check sklearn website for other clustering algorithms

9.1.1. Euclidean distance

K-means clustering categorizing items into k groups of similarity (based on **Euclidean distance**)

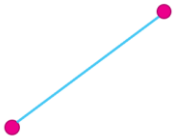
Manhattan distance:

$$D = |x1 - x2| + |y1 - y2|$$



Euclidean distance:

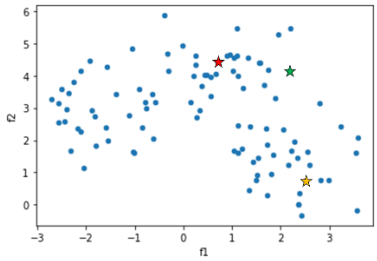
$$D = \sqrt{(x1 - x2)^2 + (y1 - y2)^2}$$



9.1.2. Step 1: Initializing centroids

Centroid: middle value of the cluster

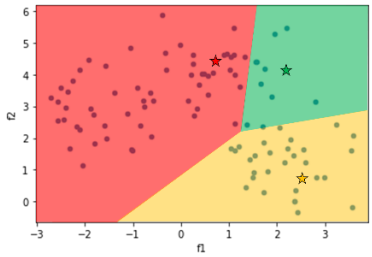
Just create k random points on the dataset (usually k random points from the dataset



9.1.3. Step 2: Categorizing samples

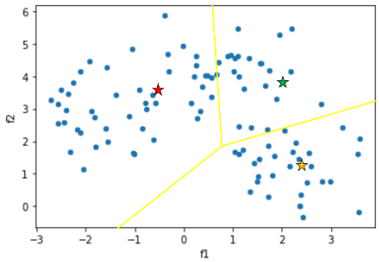
Determine the closest centroid for each sample (with Euclidean distance)

Draw a border between the categories



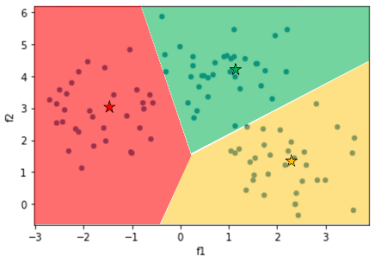
9.1.4. Step 3: Update centroids

Move the centroids towards the mean of the categories



9.1.5. Repeat step 2 and 3

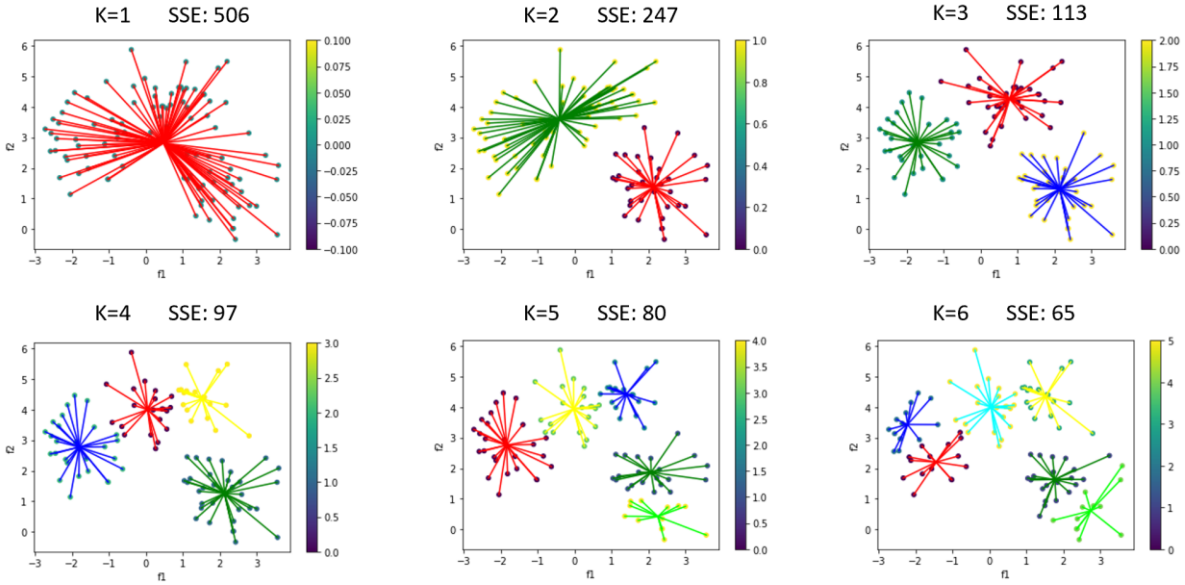
Repeat step 2 and 3 until convergence



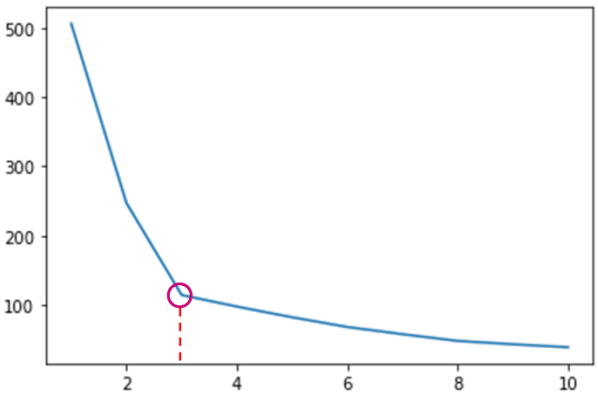
9.1.6. Determining the optimal number of clusters

Different strategies possible: - Visual inspection of data
- Silhouette clustering: score: how good a clustering technique is (~1)
- Elbow method

Elbow method: Vary the number of clusters → calculate the sum of squared errors (distance)



Plot SSE against K and look for the elbow in the graph



9.2. Hierarchical based methods

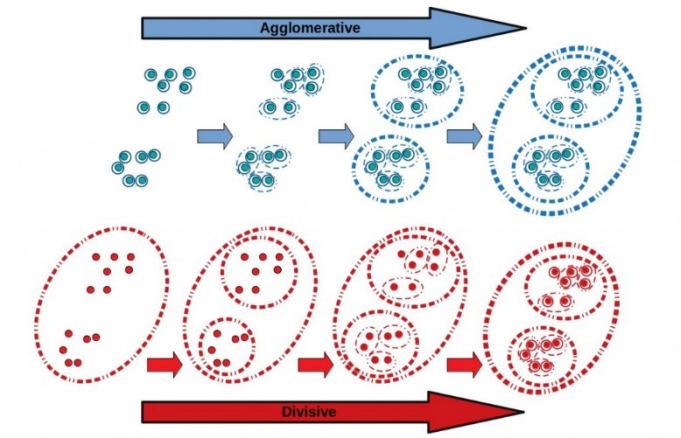
Family of algorithms building nested clusters by merging/splitting successively
Can be represented as a tree/dendrogram (root is a cluster containing all samples)
Leaves: cluster with only one sample

Advantages: - No need to know how many clusters
- Structure of tree could be useful (e.g., phylogenetic trees)

Disadvantages: - Computational exhaustive (calculations are squared dependent on data points)
- Not useable for very big data sets

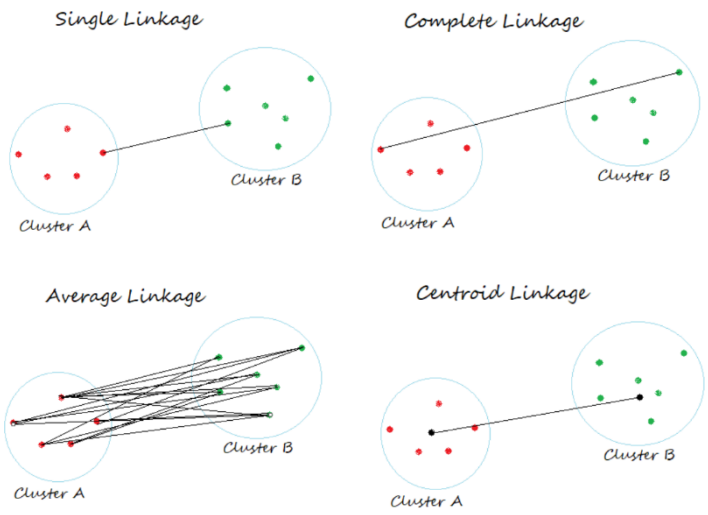
9.2.1. Types of Hierarchical based methods

Two major types: - Agglomerative clustering: combining smaller clusters into larger ones
- Divisive clustering: dividing larger clusters into smaller ones



9.2.2. Calculate distances/linkages between clusters

4 possible methods: - Single linkage: distance between the closest points of each cluster
- Complete linkage: distance between furthest points of each cluster
- Average linkage: average distance between each point of 2 clusters
- Ward linkage: centroid linkage in a hierarchical way



Chapter 10: Dimensionality reduction

Dimensionality reduction: process of reducing number of random variables under consideration

Application: - Visualisation (not possible with more than 3 features)
- Removing noise
- Data compression
- Curse of dimensionality (some models do not perform well with lot of features)

Algorithms: - Principal component analysis (PCA)
- Linear Discriminant Analysis (LDA)
- Multidimensional scaling (MDS)
- Uniform Manifold Approximation and Projection (UMAP)
- t-distributed stochastic neighbour embedding (t-SNE)
- Autoencoders (special neural network)
- Missing values Ratio
- Low variance filter

} Pre-processing techniques on throwing data away

Practical applications: - Classification of genes
- Face recognition
- Voice recognition
- Handwriting recognition
- Compression
- Outlier detection

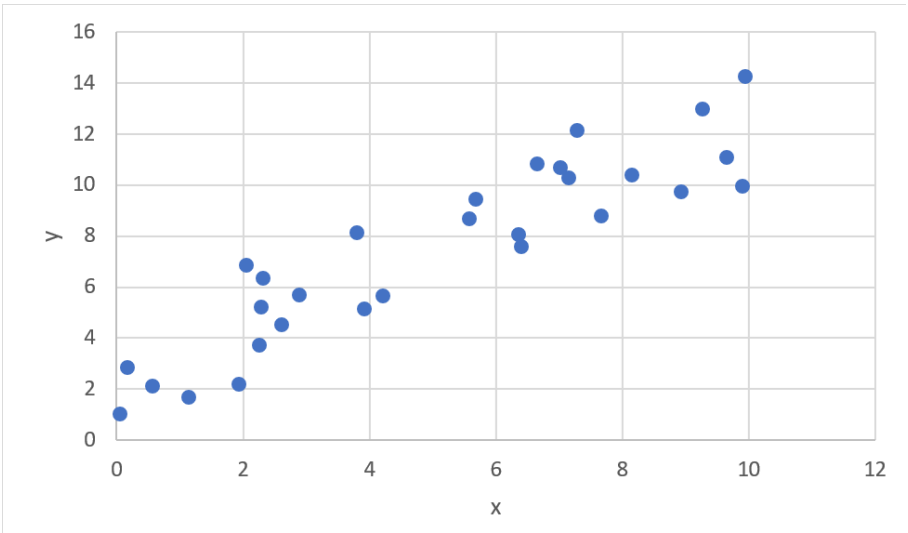
10.1. Principal component analysis (PCA)

PCA transforms several features to smaller amount of non-correlated features whilst preserving as much information as possible

Determination of eigenvectors: based on data's covariance matrix (scaled and shifted)

10.1.1. Step 1: Determine highest covariance

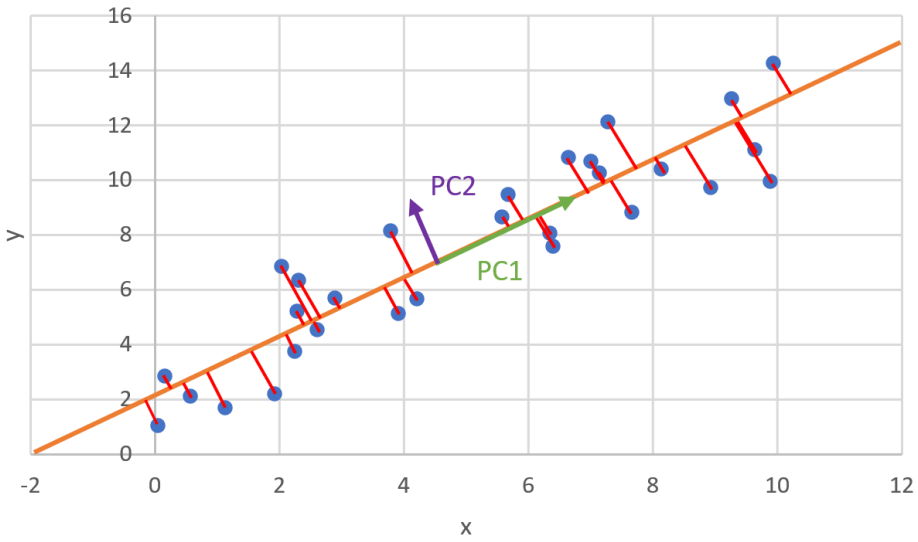
Determine the axis with the most variance (=PC1)



10.1.2. Step 2: Determine the eigenvectors

PC1: axis along highest variance

PC2: perpendicular axis, based on smallest SSE (~MSE in linear regression)



10.1.3. Step 3: Rotate axes

Rotate until PC1 becomes X-axis and all data is centred around O

10.1.4. Step 4: Project on PC1

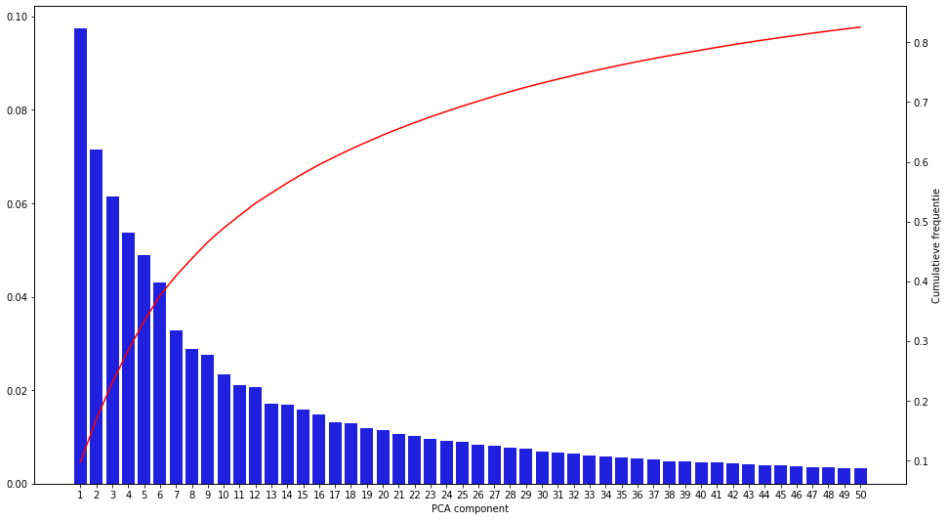
You now still have the same number of features, but just project data on PC1 → PC1 is only feature left now.

10.1.5. Reconstruction of data using PCA

Each PCA values needs to be multiplied with that PC. Use inverse_transform to go back to all features.

10.1.6. Determining a good number of principal components

- Plot the explained variance ratio of each PC
- Plot cumulative sum of explained variance ration



Chapter 11: Neural networks

(Artificial) Neural Networks: - Subset of machine learning → deep learning (when hidden layers)

Inspired by human brain → mimic interneural signalling

- Knowledge is acquired through a learning process → repetition → better model
- Store knowledge as synaptic weights "interneuron connection strengths"
- Good for pattern recognition, speech recognition, perception, computer vision, …
- More information → learns faster
- Very good in parallel processing (classical computer is serial, one at a time) → GPU!

Comparison between human brain and computer neural network:

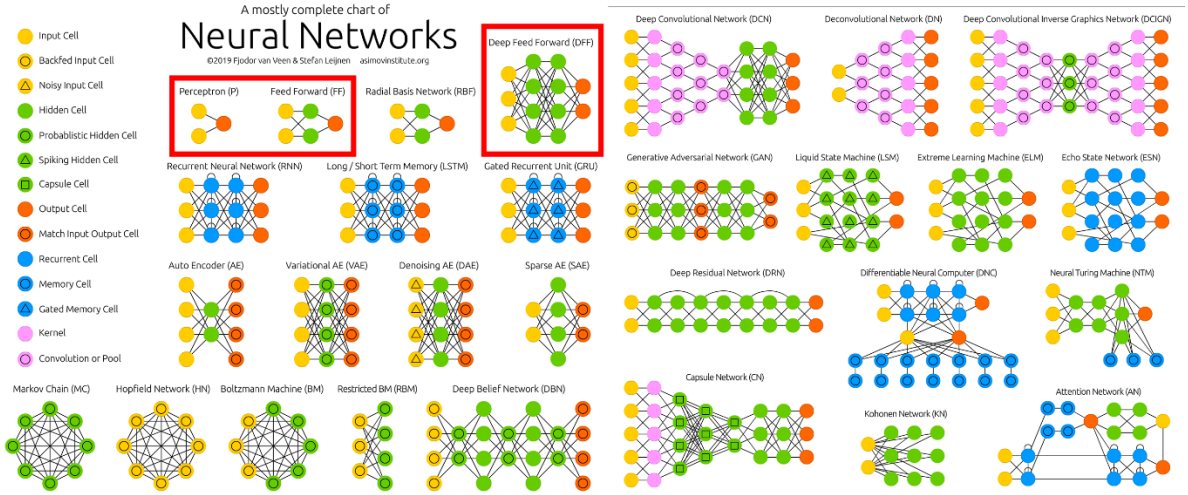
	Brain	**Computer**
Processing Elements	10^{10} neurons	10^{8} transistors
Element Size	10^{-6} m	10^{-6} m
Energy Use	30 W	30 W (CPU)
Processing Speed	10^{2} Hz	10^{12} Hz
Style Of Computation	Parallel, Distributed	Serial, Centralized
Energetic Efficiency	10^{-16} joules/opn/sec	10^{-6} joules/opn/sec
Fault Tolerant	Yes	No
Learns	Yes	A little

Characteristics: - Architecture
- Learning algorithm
- Activation functions

11.1. Architectures

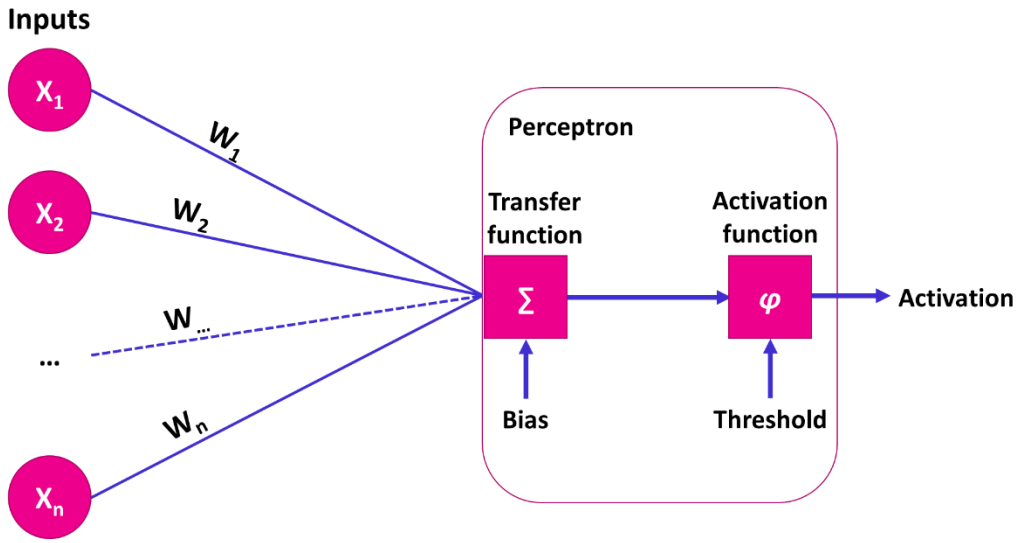
Many available!

Most important to know for the exam: - Perceptron (input to output)
- Feed forward (1 layer between input and output)
- Deep feed forward (multiple layers between I and O)



11.1.1. Perceptron

Perceptron: 1 Artificial neuron, containing transfer function and an activation function



Inputs (X_i): features or pixel in a picture

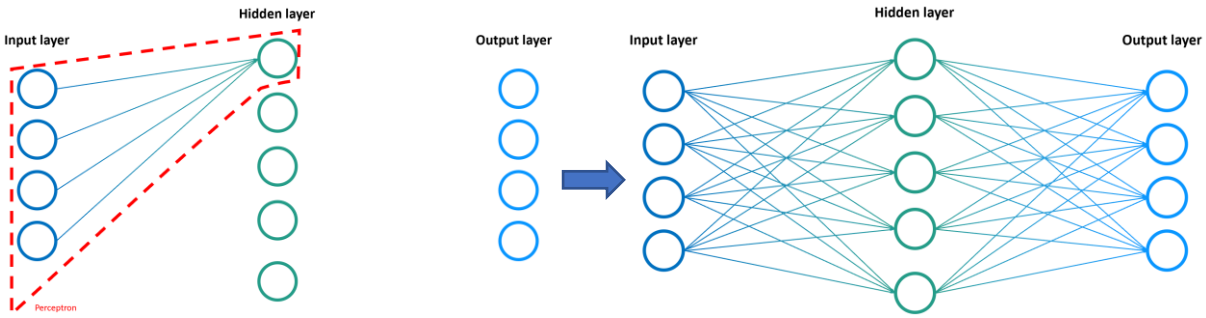
Weights (W_i): weights of each feature

Transfer function: $\mu = \sum_{i=1}^{n} X_i W_i + b$ (sum of products of the weight and input for each input + bias)

Activation function: e.g., Sigmoid function: $\varphi = \frac{1}{1+e^{-\mu}}$ (~ threshold (1 if > 0,5, else 0))

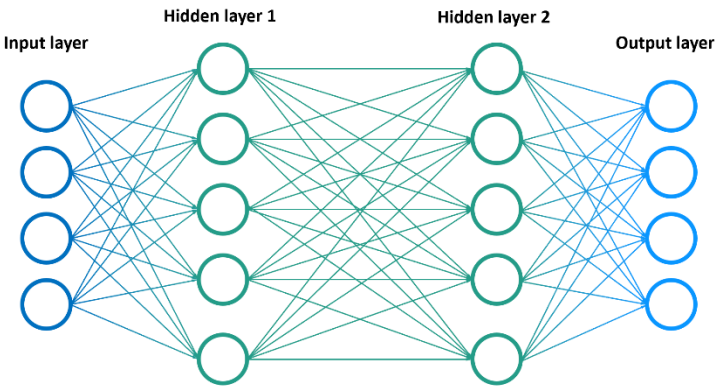
11.1.2. Feed forward neural network

Feed forward has hidden layers between input and output layer. Each connection has a weight



11.1.3. Deep feed forward neural network

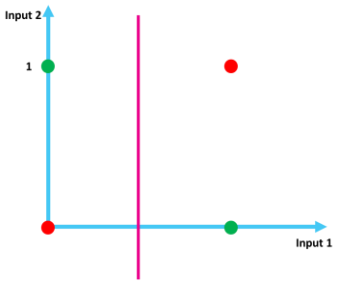
More hidden layers. Hidden layers ➔ black box of AI



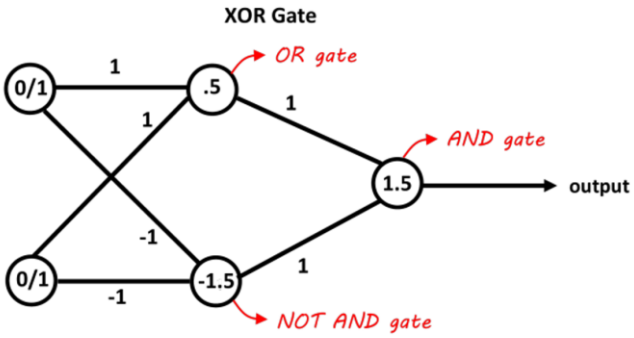
11.1.4. XOR example

XOR problem: - True if one of both is True, but not both!
- Plot on graph → output is not divisible with one line!

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0



Solution based on neural networks (XOR gate)



11.1.5. Output layer

As classifier: - # Output neurons = # of classes
- Train model: compare with expected output
- Expected output: one-hot encoded target vectors

11.2. Learning algorithm: backpropagation

Backpropagation: look at output → see what goes wrong → adjust weights, biases and thresholds

Increase output by: - Increasing bias
- Increasing positive products: increase weights, increase O previous layer
- Decreasing negative products: decrease weights, decrease O previous layer

Decrease output by: - Decreasing bias
- Decreasing positive products: decrease weights, decrease O previous layer
- Increasing negative products: increase weights, increase O previous layer

Output of previous output cannot be done directly → change weights and bias of previous layer

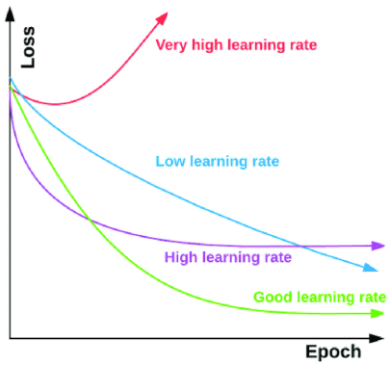
11.2.1. Learning rate

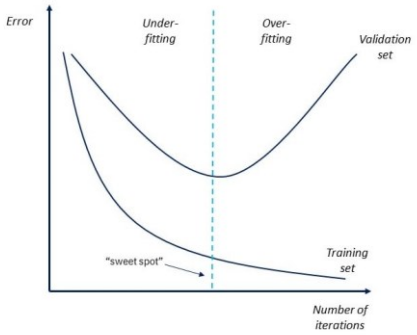
Changing values is done by the **learning rate** (**η**) (don't need to know the math behind this)

Often used: Adam (adaptive moment estimation)

11.2.2. Error function (=loss)

Error (MSE) can be plotted in function of the number of epochs → learning rate is hyperparameter
Loss can also be used to prevent overfitting → stop learning at the right moment





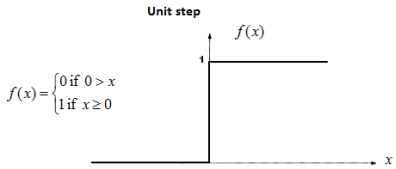
11.3. Activation functions

Several activation functions exist. Most important: - Step function (not useful)
-

11.3.1. Step function

1 if X >0; else 0

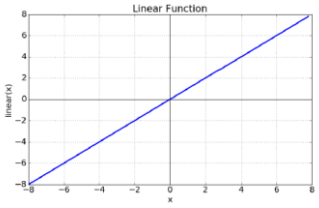
Disadvantages: - Can only return 1 or 0
- Multiple classes → what if multiple 1-values?
- Backpropagation won't work (derivative = 0)



Usage: - Don't use!

11.3.2. Linear function (Adaline)

Output is output from transfer function. → linear distribution

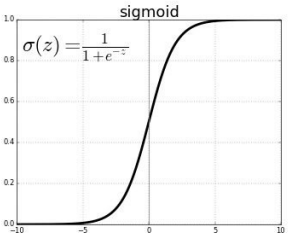


Disadvantages: - Output is always linear (independent from # layers)
- Derivative is a constant (no link to input)

Usage: - As input layer
- As output layer for regression

11.3.3. Sigmoid function

Output is $\sigma = \frac{1}{1+e^{-\mu}}$ → nonlinear + output always between 0 and 1

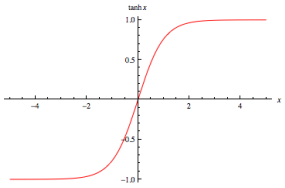


Disadvantages: - Vanishing gradient (problem with lot of layers)

Usage: - Not used often anymore
- Sometimes used as output layer for classification

Vanishing gradient: at bottom or top: gradient steps become very small and have little impact

11.3.4. Hyperbolic tangent (tanh)

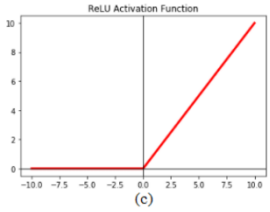


Projection of a tangent function on axis → hyperbolic → nonlinear + output between -1 and 0 (Very similar to sigmoid function with same problem)

Disadvantages: - Vanishing gradient (problem with lot of layers)

Usage: - Not used often anymore
- Sometimes used in recurrent neural networks

11.3.5. Rectified linear unit (Relu)

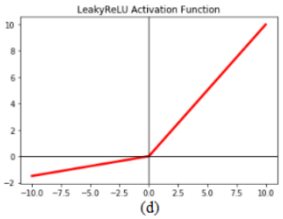


X if x > 0, else 0 (when above 0 it is linear)
→ a lot of them become 0 (sparse activation)
→ Each function approximated by combination of Relu functions

Disadvantages: - When 0 → Dead neuron (no gradient anymore)

Usage: - Used for hidden layers

11.3.6. Leaky Relu



Solution to dead neuron problem → if x<0 diff angle/slope
Variation on Relu
Neurons don't die anymore

Disadvantages: - a lot more parameters to train

Usage: - Used for hidden layers

11.3.7. What activation function to use

Hidden layers: - First use Relu
- Then try Leaky Relu
- Never use sigmoid or tanh!

Output layer: - Linear function for regression
- SoftMax/sigmoid function

SoftMax: generalized sigmoid function → good for multiclass classification

11.4. Underfitting and overfitting

Big neural network → too many capabilities → neural network tries to remember by heart (overfit)
Small neural network → too less capabilities → it does not learn much (underfit)

11.4.1. Drop out

Drop out: Randomly disable neurons in a hidden layer during training

- Other neurons need to be able to replace disabled neurons (therefore they may not be too specific)
- Used to prevent overfitting

11.5. Coding neural networks

3 players: - Keras (now fully integrated in TensorFlow)
- **TensorFlow**
- PyTorch

11.5.1. The sequential model

Sequential model: stack multiple layers on top of each other

Initializing a model

```
From tensorflow.keras.models import Sequential
model = Sequential()
```

Add new layers

```
From tensorflow.keras.layers import Dense
model.add(Dense(units=64, activation='relu', input_dim=784) # 64 hidden layer Relu units, 784 inputs
model.add(Dense(units=10, activation='softmax') # 10 softmax outputs
```

Compilation

```
model.compile(loss='categorical_crossentropy',  optimizer='sgd', metrics=['accuracy'])
model.compile(loss=keras.losses.categorical_crossentropy,
optimizer=keras.optimizers.SGD(learning_rate=0.01, momentum=0.9, nesterov=True)
)
# Loss function: which error to calculate
# Optimizer:
```

Training

```
model.fit(X_train, y_train, epochs=5, batch_size=32)
# Epochs: number of times a neural network gets to see the full training data
# Batch_size: number of samples the neural network gets to see before it updates its weight
# Update based on average error on the batch
# Iterations: number of times the weights are updated (epochs * batches)
```

Prediction

```
classes = model.predict(X_test, batch_size=128)
```

11.5.2. Built-in activation functions

- SoftMax
- Relu
- Sigmoid
- Tanh
- Linear (= no activation function)
- Leaky Relu

11.5.3. Learning rate optimizers

- **SGD (+ nesterov)**: stochastic gradient descent with (nesterov) momentum
- RMSProp: Better for recurrent neural networks
- Adagrad
- **Adam**
- Adamax

11.5.4. Batch modes

- Batch mode: Batch size = all training samples. All data is used before updating the weights
- Mini-batch mode: Batch size > 1 and < #samples
- Stochastic mode: Batch size = 1. Update the training data after each training sample

Advantages of a small batch size: - Less memory usage
- Learns faster than big batch sizes
- Faster feedback for the model

Disadvantages: - Less accurate prediction of gradient (based on last few training samples)
- Full training data is most accurate

Why not always use full training data as a batch? → Memory issues

11.5.5. Loss function

Loss functions for regression: - mean_squared_error(y_true,y_pred)
- mean_absolute_error(y_true,y_pred)
- mean_squared_logarithmic_error(y_true,y_pred)
- mean_absolute_percentage_error(y_true,y_pred)

Error losses for classification: - binary_crossentropy(y_true,y_pred) (multilabel classification)
- categorical_crossentropy(y_true,y_pred) (multiclass classification)

11.5.6. Metrics

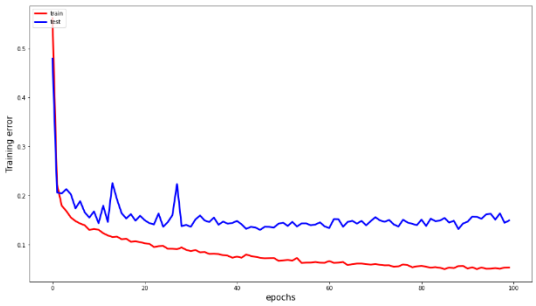
Metrics for regression: - Mean squared error: mean_squared_error (or mse)
- Mean absolute error: mean_absolute_error (or mae)
- Mean percentage error: mean_absolute_percentage_error (or mape)
- Cosine proximity: cosine

Metrics for classification: - Binary accuracy: binary_accuracy
- Categorical accuracy: categorical_accuracy
- Top k categorical accuracy: top_k_categorical_accuracy
- Precision
- Recall
- AUC

Specification of multiple metrics are possible:

```
model.compile(   loss="binary_crossentropy",
optimizer='adam',
metrics=['accuracy','Precision','AUC'])
```

Can also be plotted:



11.5.7. Other parameters

- Validation_split: fraction of training data used for validation (comparable to K-fold validation)
- Sample_weight_mode: for unbalanced data

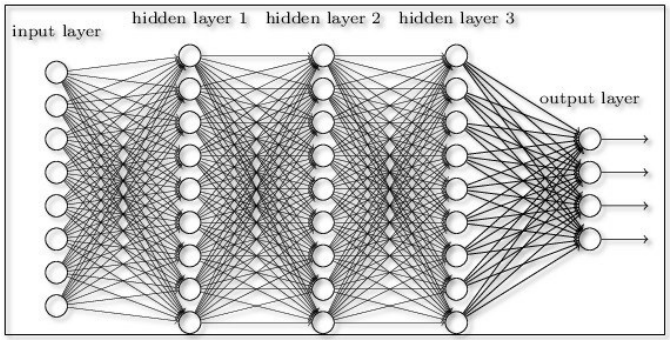
11.6. Other architectures

Don't need to learn this for the exam.

- Feed forward neural network
- Autoencoders
- Convolutional neural network

11.6.1. Feed forward neural network (FFNN)

Supervised learning where data is not sequential or time-dependent (data does not need to be in a particular order)

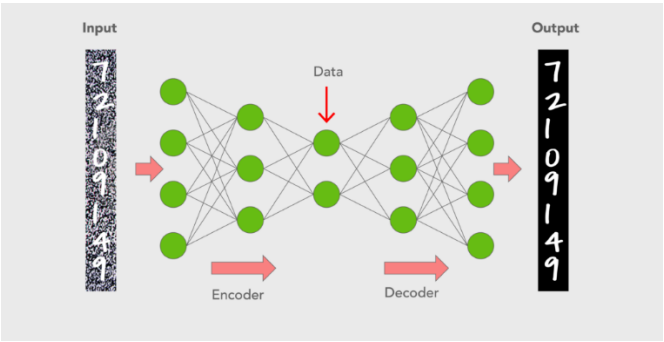


11.6.2. Autoencoders (AE)

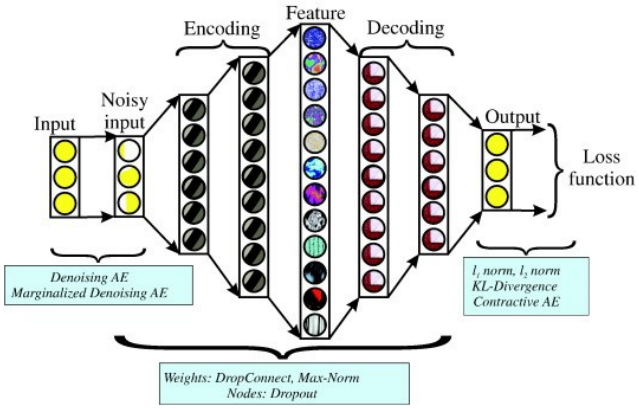
Unsupervised learning for compressing data, denoising data, detecting outliers or filling in NaN

Exists out of a: - Encoder
- Decoder

} Expected output = input



Special variants: - Variational autoencoders
- Sparse autoencoders (example)



11.6.3. Recurrent neural networks (RNN)

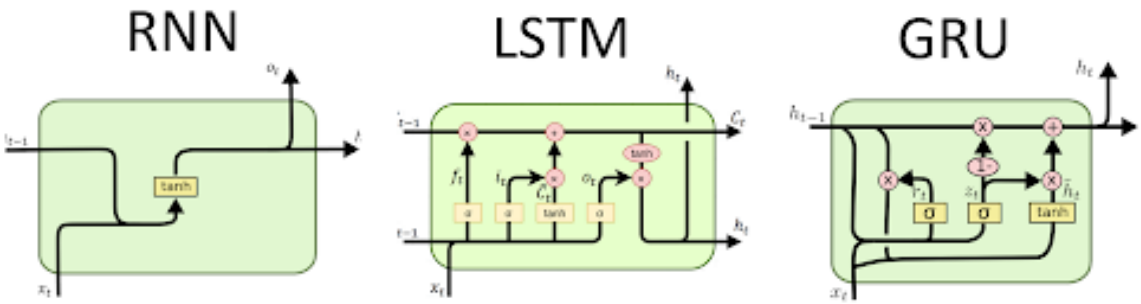
Good for time series data (sequential data). Output becomes input from same perceptron 3 types:

Basic RNN: Has a vanishing gradient problem

Long Short-Term Memory network (LSTM): Computationally expensive (remembers from longer ago)

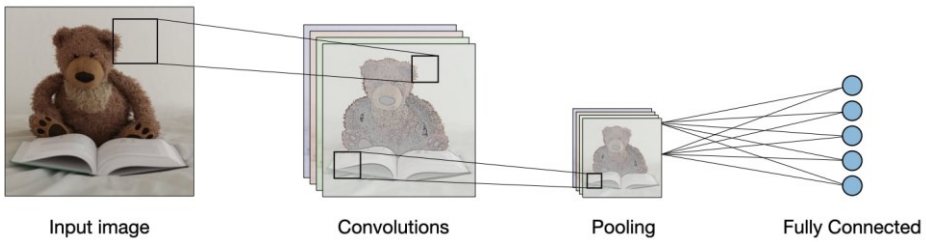
Gated Recurrent Unit (GRU): Less computationally expensive

Used for e.g., text processing, sound analysis



11.6.4. Convolutional neural networks (CNN)

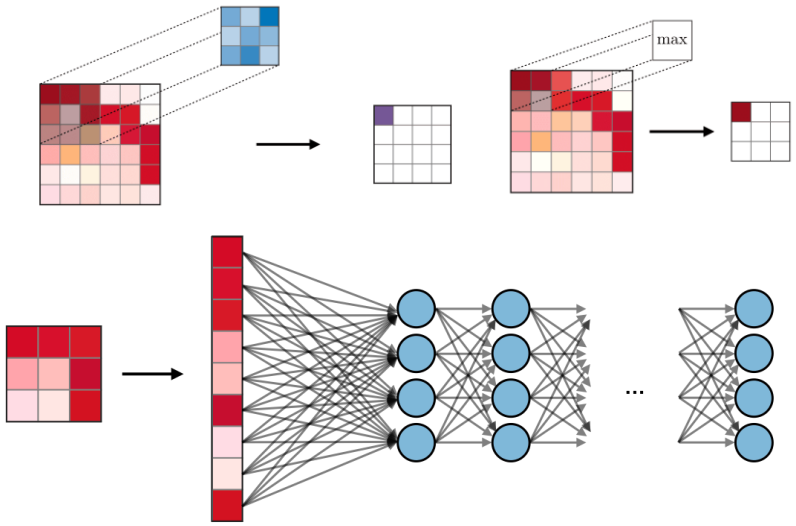
Heavily used for image analysis, based on convolutions, pooling and flattening:



Convolutions: create filters on subsets of data (not fully connected network)

Pooling: make smaller subsets of features from previous convolutional network (max, average, …)

Flatten: all convolutions transformed to fully connected network (needed for final prediction)



11.6.5. Generative adversarial neural networks (GANN)

Use case: generate new data indistinguishable from real data (deepfakes)

2 neural networks:
- **Generator**: learns to create new data and to trick the discriminator
- **Discriminator**: learns to distinguish real from fake

Repeated → both in competition