Machine Learning Reference

Quickly find what you need for your ML-project

Moritz Gück

2023-08-29

Table of contents

Part I

Introduction

Quickly find what you need for your ML-project

Quick links

Statistics

Data cleaning

Classification

Regression

Clustering

Neural Networks

Check lists

Project Management

Quick info

> **What is this?**
>
> This is a reference on applied machine learning for daily use. The topics range from math fundamentals to complex machine learning models and explanation methods. The focus lies on concise explanations and practical code snippets.

> **How do I use it?**
>
> As an ML engineer or data scientist, you can …
>
> - Use the search-field and quickly find solutions for your tasks and copy the code snippet into your project.
> - Refresh you knowledge about a topic and find references for further reading.
>
> The currently incomplete sections are marked in grey.

I want to get involved

That's awesome! `r emo::ji("smiley")` Do you want to:

- Write/adapt a chapter, section, paragraph?
- Create figures?
- Or proofread new chapters?

Please write an email to ml_reference[ät]icloud.com
Your benefits: Good karma, "While we teach, we learn", Being listed as an author on this page.

I want to report something

You found errors or unclear explanations? You have a good idea? Please file an issue under: github.com/MoritzGuck/Machine-Learning-Reference or write an email to ml_reference[ät]icloud.com.

Related work

- Machine Learning Glossary: An online book with more details on underlying mechanisms and a large chapter on fundamentals in maths and neural networks.

- R Cookbook: A comprehensive online book on R and specific problems.

- Google Machine Learning Glossary: Glossary with brief explanations on fundamental machine learning topics.

- Distill: Beautiful and easy to understand visualizations and articles on machine learning algorithms.

Icons created by Freepik - Flaticon

1 Setup Environment

How to setup your packages and virtual environment.

1.0.1 Conda

Conda is a package and virtual environment manager. It resolves dependencies of your python and non-python packages.

Install:

conda docs - Installation

Installation via UNIX CLI

```
mkdir -p ~/miniconda3
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh -O ~/minic
bash ~/miniconda3/miniconda.sh -b -u -p ~/miniconda3
rm -rf ~/miniconda3/miniconda.sh
~/miniconda3/bin/conda init bash
~/miniconda3/bin/conda init zsh
```

List your existing environments:

```
conda env list
```

Create new environment:

Create a yaml file with required packages like this one: github - ml-reference-env.yml

Create a conda env from that file:

```
conda env create -f environment.yml
```

Change to an existing environment:

```
conda activate test
conda env list
```

Check and update package versions:

After activating an env:

```
python --version
```

```
conda update python
```

Install packages:

Latest version:

```
conda install pandas
```

Specific version:

```
conda install pandas==2.0.0
```

From different channel:

```
conda install pandas -c conda-forge -y
```

switch back to base environment:

```
conda deactivate
```

Removing environment:

```
conda env remove -n env-name
```

More info: Intro to Conda virtual environments

Export environment to conda:

```
conda env export > environment.yml
```

Part II

Math fundamentals

2 Statistics

A probability is a measure of how frequent or likely an event will take place.

2.0.1 Probability Basics

2.0.1.1 Probability interpretations

- **Frequentist:** Probability is the fraction of positive samples, if we measured infinitely many samples.
- **Objectivist:** Probabilities are due to inherent uncertainty properties. Probabilities are calculated by putting outcomes of interest into relation with all possible outcomes.
- **Subjectivist:** An agent's *rational* degree of belief (not external). The belief needs to be coherent (i.e. if you make bets using your probabilities you should not be guaranteed to lose money) and therefore need to follow the rules of probability.
- **Bayesian:** (Building on subjectivism) A reasonable expectation / degree of belief based on the information available to the statistician / system. It allows to give certainties to events, where we don't have samples on (e.g. disappearance of the south pole until 2030).

Also the frequentist view is not free of subjectivity since you need to compare events on otherwise similar objects. Usually there are no completely similar objects, so you need to define them.

2.0.1.2 Probability Space

The probability space is a triplet space containing a sample/outcome space Ω (containing all possible atomic events), a collection of events S (containing a subset of Ω to which we want to assign probabilities) and the mapping P between Ω and S.

2.0.1.3 Axioms of Probability

The mapping P must fulfill the axioms of probability:

1. $P(a) \geq 0$
2. $P(\Omega) = 1$
3. $a, b \in S$ and $a \cap b = \{\} \Rightarrow P(a \cup b) = P(a) + P(b)$

a, b are events.

2.0.1.4 Random Variable (RV)

A RV is a **function** that maps points from the sample space Ω to some range (e.g. Real numbers or booleans). They are characterized by their distribution function. E.g. for a coin toss:

$$X(\omega) = \begin{cases} 0, & \text{if } \omega = heads \\ 1, & \text{if } \omega = tails. \end{cases}$$

2.0.1.5 Proposition

A Proposition is a conclusion of a statistical inference/prediction that can be true or false (e.g. a classification of a datapoint). More formally: A disjunction of events where the logic model holds. An event can be written as a **propositional logic model**:
$A = true, B = false \Rightarrow a \wedge \neg b$. Propositions can be continuous, discrete or boolean.

2.0.2 Probability distributions

Probability distributions assign probabilities to to all possible points in Ω (e.g. $P(Weather) = \langle 0.3, 0.4, 0.2, 0.1 \rangle$, representing Rain, sunshine, clouds and snow). Joint probability distributions give you a probability for each atomic event of the RVs (e.g. $P(weather, accident)$ gives you a 2×4 matrix.)

2.0.2.1 Cumulative Distribution Function (CDF)

The CDF is defined as $F_X(x) = P(X \leq x)$ (See figure CDF).

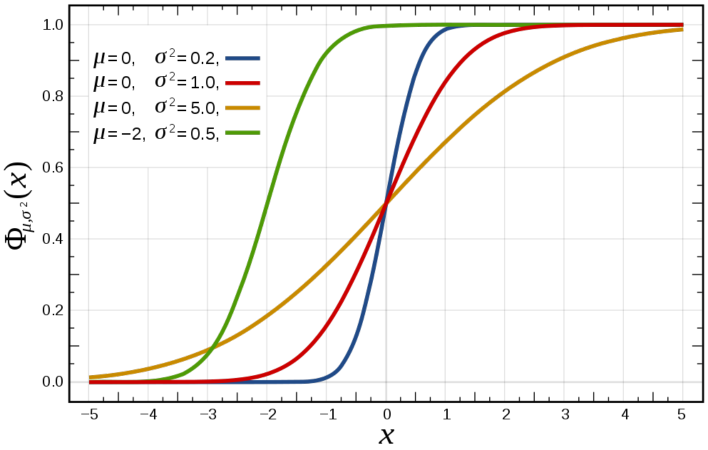


Figure 2.1: Cumulative distribution function of a normal distribution for different mean (μ) and variance (σ). *Source: user Inductiveload on wikimedia.org.*

2.0.2.2 Probability Density Function (PDF)

For continuous functions the PDF is defined by

$$p(x) = \frac{d}{dx} p(X \leq x).$$

The probability of x being in a finite interval is

$$P(a < X \leq b) = \int_a^b p(x)dx$$

A PDF is shown in the following figure.

2.0.2.3 Properties of Distributions

- The **expected value** (E) or **mean** (μ) is given by $E[X] = \sum_{x \in X} x * p(x)$ for discrete RVs and $E[X] = \int_X x * p(x)dx$ for continuous RVs.
- The **variance** measures the spread of a distribution: $var[X] = \sigma^2 = E[(X - \mu)^2] = E[X]^2 - \mu^2$.
- The **standard deviation** is given by: $\sqrt{var[X]} = \sigma$. It is interpreted as the deviation from the mean that needs to be expected.
- The **mode** is the value with the highest probability (or the point in the PDF with the highest value):

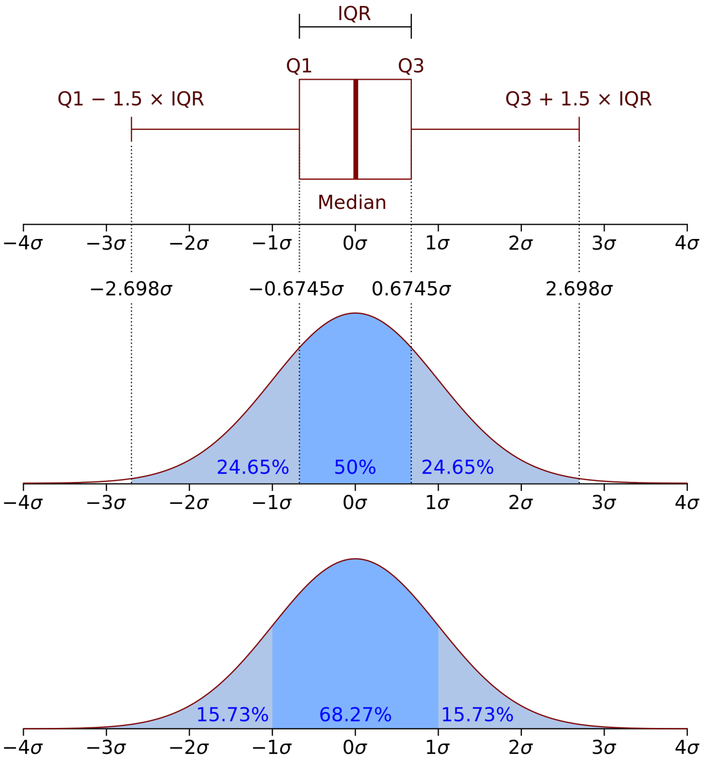


Figure 2.2: Probability density function of a normal distribution with variance (σ). In red a range from a Box-plot is shown with quartiles (Q1, Q3) and interquartile range (IQR). For the cutoffs (borders to darker blue regions) the IQR (on top) and σ are chosen. Another common cutoff is the confidence interval with light blue regions having a probability mass of $2 * \alpha/2$. *Source: user Jhguch on wikimedia.org.*

- The **median** is the point at which all point less than the median and all points greater than the median have the same probability (0.5).
- The **quantiles** (Q) divide the datapoints into sets of equal number. The Q_1 quartile has 25% of the values below it. The **interquartile range** (IQR) is a measure to show the variability in the data (how distant the points from the first and last quartile are)

2.0.2.4 Dirac delta function

The **dirac delta** is simply a function that is infinite at one point and 0 everywhere else:

$$\delta(x) = \begin{cases} \infty, & \text{if } x = 0 \\ 0, & \text{if } x \neq 0 \end{cases} \qquad \text{and} \int_{-\infty}^{\infty} \delta(x) dx = 1$$

(Needed for distributions further on)

2.0.2.5 Uniform distribution

The uniform distribution has the same probability throughout a specific interval:

$$\text{Unif}(a, b) = \frac{1}{b-a} \mathbb{1}(a < x \leq b) = \begin{cases} \frac{1}{b-a}, & \text{if } x \in [a, b] \\ 0, & \text{else} \end{cases}$$

$\mathbb{1}$ is a vector of ones.

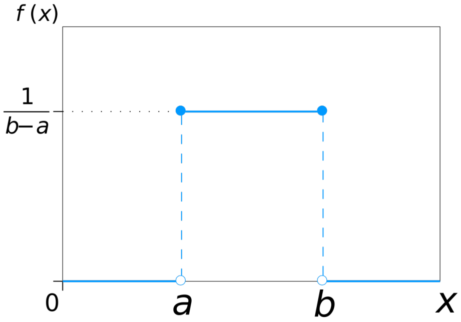


Figure 2.3: Uniform distribution. *Source: user IkamusumeFan on wikimedia.org.*

2.0.2.6 Discrete distributions

Used for random variables that have discrete states.

2.0.2.6.1 Binomial distribution

Used for series of experiments with two outcomes (success or miss. e.g. a series of coin flips).

$$X \sim \mathrm{Bin}(n, \theta), \quad \mathrm{Bin}(k|n, \theta) = \binom{n}{k} \theta^k (1-\theta)^{n-k}, \quad \binom{n}{k} = \frac{n!}{k!(n-k)!},$$

where n is the number of total experiments, k is the number of successful experiments and θ is the probability of success of an experiment.

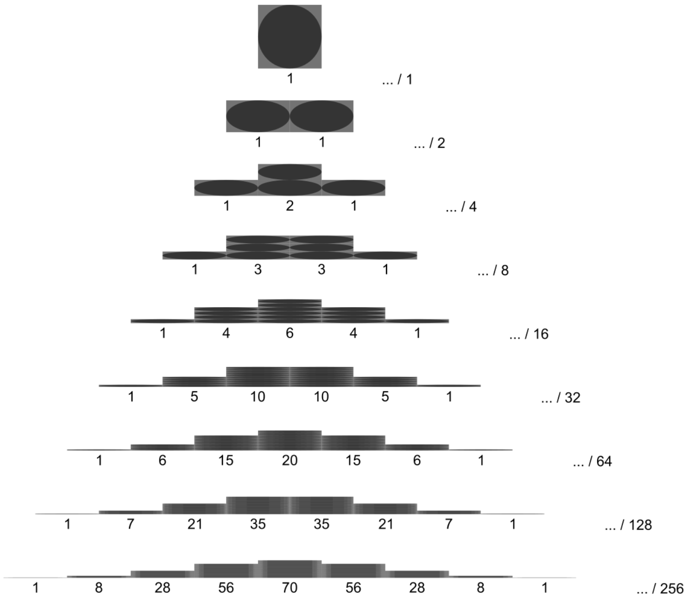


Figure 2.4: Binomial distribution of balls in Pascals triangles with different numbers of layers (The top one has 0 layers). Example: For a triangle with $n = 6$ layers, the probability that a ball lands in the middle box $k = 3$ is $20/64$. *Source: user Watchduck on wikimedia.org*

2.0.2.6.2 Bernoulli distribution

Is a special case of the binomial distribution with $n = 1$ (e.g. one coin toss).

$$X \sim \mathrm{Ber}(\theta), \quad \mathrm{Ber}(x|\theta) = \theta^{\mathbb{1}(x=1)}(1-\theta)^{\mathbb{1}(x=0)} = \begin{cases} \theta, & \text{if } x = 1 \\ 1-\theta, & \text{if } x = 0 \end{cases}$$

2.0.2.6.3 Multinomial distribution

Used for experiments with k different outcomes (e.g. dice rolls: Probability of different counts of the different sides).

$$\mathrm{Mu}(x|n,\theta) = \binom{n}{x_1, ..., x_K} \prod_{i=1}^{K} \theta_j^{x_j} = \frac{n!}{x_1!, ..., x_k!} \prod_{i=1}^{K} \theta_j^{x_j},$$

where k is the number of outcomes, x_j is the number times that outcome j happens. $X = (X_1, ..., X_K)$ is the *random vector.*

2.0.2.6.4 Multinoulli distribution

Is a special case of the multinomial distribution with $n = 1$. The random vector is then represented in *dummy-* or *one-hot-encoding* (e.g. $(0, 0, 1, 0, 0, 0)$ if outcome 3 takes place).

$$\mathrm{Mu}(x|1,\theta) = \prod_{j=0}^{K} \theta_j^{\mathbb{1}(x_j=1)}$$

2.0.2.6.5 Empirical distribution

The empirical distribution follows the empirical measurements strictly. The CDF jumps by 1/n every time a sample is “encountered” (see figure).

$$\mathrm{p}_{\mathrm{emp}}(A) = \frac{1}{N} \sum_{i=1}^{N} \delta_{x_i}(A), \quad \delta_{x_i} = \begin{cases} 1, & \text{if } x \in A \\ 0, & \text{if } x \notin A \end{cases},$$

w where $x_1, ..., x_N$ is a data set with N points. The points can also be weighted:

$$p(x) = \sum_{i=1}^{N} w_i \delta_{x_i}(x)$$

2.0.2.7 Continuous distributions

Used for random variables that have continuous states.

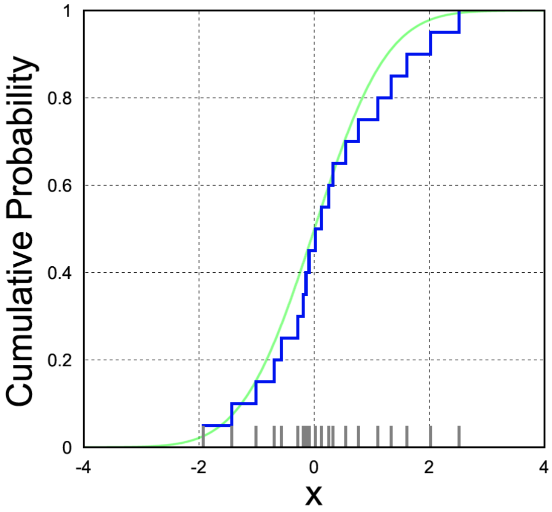


Figure 2.5: Cumulative empirical distribution function (blue line) for samples drawn from a standard normal distribution (green line). The values of the drawn samples is shown as grey lines at the bottom. Source: user nagualdesign on wikimedia.org.

2.0.2.7.1 Normal/Gaussian distribution

Often chosen for random noise because it is simple and needs few assumptions (see sect. CLT). The PDF is given by:

$$p(x|\mu\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right],$$

where μ is the mean and σ^2 is the variance. The CDF is given by:

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{\infty}^{x} e^{\frac{-t^2}{2}} dt$$

2.0.2.7.2 Multivariate normal/Gaussian distribution

For T datapoints with k dimensions (features). The pdf is:

$$p(x|\mu,\Sigma) = \frac{1}{\sqrt{(2\pi)^k |\Sigma|}} \exp\left[-\frac{1}{2}(x-\mu)^\top \Sigma^{-1} (x-\mu)\right],$$

where x now has multiple dimension $(x_1, x_2, ..., x_k)$ and Σ is the $k \times k$ covariance matrix: $\Sigma = \mathrm{E}[(X-\mu)(X-\mu)]$. The covariance between features is: $\mathrm{Cov}[X_i, X_j] = \mathrm{E}[(X_i - \mu_i)(X_j - \mu_j)]$

2.0.2.7.3 Beta distribution

defined for $0 \leq x \leq 1$ (see figure Beta distribution). The pdf is:

$$f(x|\alpha,\beta) = \frac{1}{B(\alpha,\beta)} x^{\alpha-1}(1-x)^{\beta-1}$$

The beta function B is there to normalize and ensure that the total probability is 1 .

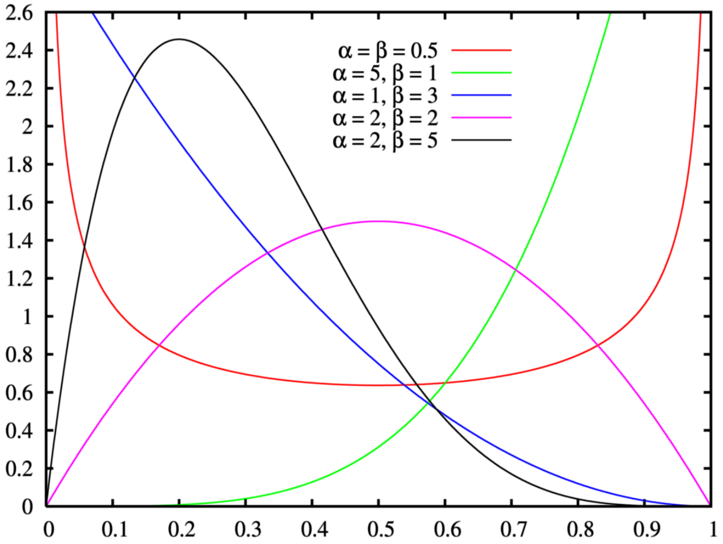


Figure 2.6: Probability density function of a beta-distribution with different parameter values. *Source: user MarkSweep on wikimedia.org.*

2.0.2.7.4 Dirichlet distribution

The multivariate version of the Beta distribution. The PDF is:

$$\mathrm{Dir}(x|\alpha) \triangleq \frac{1}{B(\alpha)} \prod_{i=1}^{K} x_i^{\alpha_i - 1}, \quad \sum_{i=1}^{K} x_i = 1, \quad x_i \geq 0 \ \forall i$$

2.0.2.7.5 Marginal distributions

Are the probability distributions of subsets of the original distribution. Marginal distributions of normal distributions are also normal distributions.

2.0.3 Central limit theorem

In many cases the sum of random variables will follow a normal distribution as n goes to infinity.

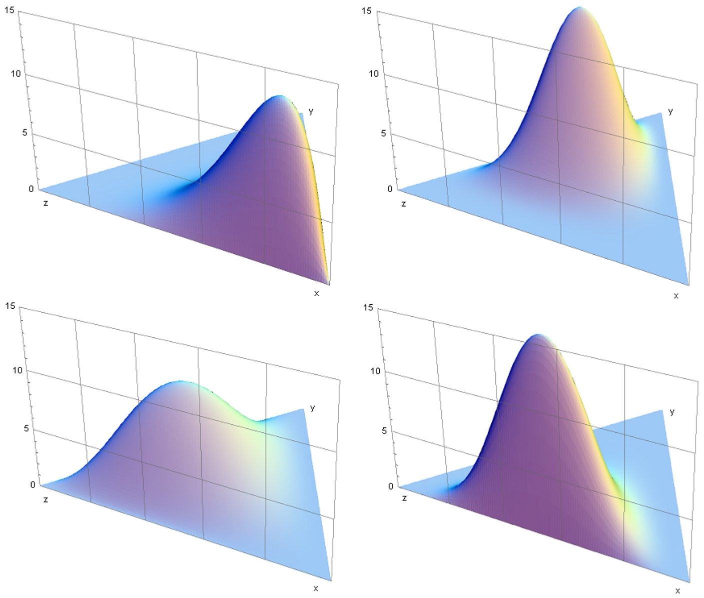


Figure 2.7: Probability density function of a Dirichlet-distribution on a 2-simplex (triangle) with different parameter values. Clockwise from top left: α = (6,2,2), (3,7,5), (6,2,6), (2,3,4). *Source: user ThG on wikimedia.org.*

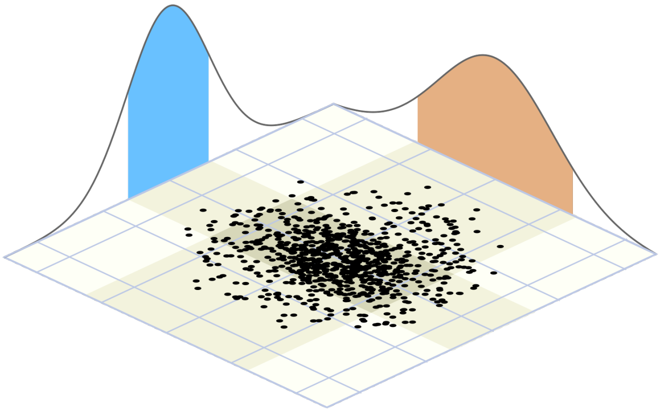


Figure 2.8: Data following a 2D-Gaussian distribution. Marginal distributions are shown on the sides in blue and orange. *Source: user Auguel on wikimedia.org.*

2.0.4 Bayesian probability

Baeysian probability represents the plausibility of a proposition based on the available information (i.e. the degree at which the information supports the proposition). The use of this form of statistics is especially useful if random variables cannot be assumed to be i.i.d. (i.e. When an event is not independent of the event before it (e.g. drawing balls without laying them back into the urn)).

2.0.4.1 Conditional/Posterior Probability

Expresses the probability of one event (Y) under the condition that another event (E) has occurred. (e.g. $C =$ "gets cancer", $S =$ "is a smoker" $\rightarrow p(C|S) = 0.2$, meaning: "given the *sole information* that someone is a smoker, their probability of getting cancer is 20%.")

The conditional probability can be calculated like follows. By defining the joined probability like so:

$$P(A \cap B) = P(A \mid B)P(B)$$

you solve for $P(A \mid B)$:

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)} = \frac{P(A, B)}{P(B)} = \alpha P(A, B),$$

where α is used as a normalization constant. If you have hidden variables (confounding factors) you need to sum them out like so:

$$P(Y|E = e) = \alpha P(Y, E = e) = \alpha \sum_{h} P(Y, E = e, H = h)$$

where X contains all variables, Y is called *query variable*, E is called *evidence variable*, $H = X - Y - E$ is called *hidden variable* or *confounding factor*. You get the joint probabilities by summing out the hidden variable.

! Usually $p(A|B) \neq p(B|A)$
! Priors are often forgotten: E.g. $P($"COVID-19"$)$ is confused with $P($"COVID-19"$|$"Person is getting tested"$)$ (because only people with symptoms go to the testing station).
! Base rate neglect: Under-representing the prior probability. E.g. You have a test with a 5% false positive rate and a incidence of disease of 2% in the population. If you are tested positive in a population screening your probability of having the disease is only 29%.
Conditional distributions of Gaussian distributions are Gaussian distributions themselves.

2.0.4.2 Independence

For independent variables it holds: $P(A|B) = P(A)$ or $P(B|A) = P(B)$

2.0.4.3 Conditional independence

Two events A and B are independent, given C: $P(A|B,C) = P(A|C)$. A and B must not have any information on each other, given the information on C. E.g. for school children: $P(\text{"vocabulary"}|\text{"height", "age"}) = P(\text{"vocabulary"}|\text{"age"})$.

2.0.4.4 Bayes Rule

Bayes rule is a structured approach to update prior beliefs / probabilities with new information (data). With the conditional probability from before ($P(A,B) = P(A|B)P(B) = P(B|A)P(A)$) we get **Bayes rule** by transforming the right-side equation to:

$$P(\text{hypothesis}|\text{evidence}) = \frac{P(\text{evidence}|\text{hypothesis})P(\text{hypothesis})}{P(\text{evidence})}$$

often used as:

$$P(\text{model}|\text{data}) = \frac{P(\text{data}|\text{model})P(\text{model})}{P(\text{data})}$$

2.0.4.4.1 Terminology:

- $P(\text{hypothesis}|\text{evidence})$ = Posterior (How probable hypothesis is after incorporating new evidence)
- $P(\text{evidence}|\text{hypothesis})$ = Likelihood (How probable the evidence is, if the hypothesis is true)
- $P(\text{hypothesis})$ = Prior (How probable hypothesis was before seeing evidence)
- $P(\text{evidence})$ = Marginal (How probable evidence is under all possible hypotheses)
- $\frac{P(\text{evidence}|\text{hypothesis})}{P(\text{evidence})}$ = Support B provides for A
- $P(\text{data}|\text{model})P(\text{model})$ = joint probability ($P(A,B)$)

2.0.4.4.2 Example for Bayes Rule using COVID-19 Diagnostics

$$P(\text{COVID-19}|\text{cough}) = \frac{P(\text{cough}|\text{COVID-19})P(\text{COVID-19})}{P(\text{cough})} = \frac{0.7 * 0.01}{0.1} = 0.07$$

Estimating $P(\text{COVID-19}|\text{cough})$ is difficult, because there can be an outbreak and the number changes. However, $P(\text{cough}|\text{COVID-19})$ stays stable, $P(\text{COVID-19})$ and $P(\text{cough})$ can be easily determined.

2.0.5 Further Concepts

2.0.5.1 Convergence in Probability of Random Variables

You expect your random variables (X_i) to converge to an expected random variable X. I.e. after looking at infinite samples, the probability that your random variable X_n differs more than a threshold ϵ from your target X should be zero.

$$\lim_{n \to \infty} P(|X_n - X| > \epsilon) = 0$$

2.0.5.2 Bernoulli's Theorem / Weak Law of Large Numbers

$$\lim_{n \to \infty} P(|\frac{\sum_{i=1}^{n} X_i}{n} - \mu| > \epsilon) = 0,$$

where $X_1, ..., X_n$ are independent & identically distributed (i.i.d.) RVs. ⇒ With enough samples, the sample mean will approach the true mean. The **strong law of large numbers** states that $|\frac{\sum_{i=1}^{n} X_i}{n} - \mu| < \epsilon$ for any $\epsilon > 0$.

2.0.6 Statistical hypothesis tests

2.0.6.1 Terminology

- Null-hypothesis (over-simplified): “The differences between these two sets of samples are due to chance.” Or “There is no (positive/negative) effect of the independent variables on the dependent variable.”
- Alternative hypothesis (over-simplified): “The differences between the two sets of samples cannot be due to chance.” Or “There is a (positive/negative) effect ...” (More examples)
- P-value: The probability to observe an at least as extreme value (e.g. sample mean of a distribution) as the measured one, if the null-hypothesis was true.
- α-Value / Significance level: Probability of falsely rejecting the null-hypothesis, given the null-hypothesis is true.
- Statistically significant: The result is significant if p $< \alpha$.

2.0.6.2 t-test

Tests for significant difference between two independent sets of samples.

$$\mathrm{t} = \frac{\bar{X}_1 - \bar{X}_2}{\sqrt{\frac{\hat{\sigma}_1^2}{n_1} + \frac{\hat{\sigma}_2^2}{n_2}}}$$

where $\bar{X}$ is the sample mean, $\hat{\sigma}^2$ is the sample variance, n is the sample size.

```
from scipy import stats
stats.ttest_ind(rvs1, rvs2, equal_var=True, alternative = "two-sided")
```

More info: scipy.org

Using the **one-ample t-test**, you can test if the sample mean ($\bar{X}$) is significantly different to an expected mean ($\hat{\mu}_0$).

```
from scipy import stats
stats.ttest_1samp(rvs, popmean=0.0)
```

More info: scipy.org

The **t-test for paired samples** is used, when the samples have been matched (e.g. subjects are measured before and after treatment).

```
from scipy import stats
stats.ttest_rel(rvs1, rvs2, alternative = "two-sided")
```

2.0.6.3 z-test

Same as t-test, but requires that the standard distribution of the population is known. Therefore it is not used often (except for large sample sizes).

$$z = \frac{\bar{X} - \mu_0}{\sigma}$$

2.0.6.4 Pearson Chi-squared (χ^2) test

Tests for significant relationships between categorical features. It is used with large sample sizes.

$$\chi^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i}$$

where χ^2 is the test-statistic, O_i is the number of observations of feature i, E_i $(= Np_i)$ is the expected count of feature i.

```
from scipy.stats import chi2_contingency
contingency_table = np.array([[10, 10, 20], [20, 20, 20]])
test_stat, p_val, degOfFreedome, expected_freq = chi2_contingency(contingency_table)
```

More info: scipy.org

When the sample sizes are smaller, **Fisher's exact test** is used. It uses the assumption of fixed marginal distributions.

```
from scipy.stats import fisher_exact
res = fisher_exact(table, alternative='two-sided')
res.pvalue
```

More info: scipy.org

3 Linear Algebra

This section is meant to give an intuitive understanding of the underlying mechanisms of many algorithms. It is mainly a summary of the course from 3Blue1Brown and deepai.org.

For details on the calculations see wikipedia.org.

3.0.1 Vectors

There are two relevant perspectives for us:

- **Mathematical:** Generally quantities that cannot be expressed by single number. They are objects in a *vector space*. Such objects can also be e.g. functions.
- **Programmatical / Data:** Vectors are ordered lists of numbers. You model each sample as such an ordered list of numbers and the numbers represent the feature-value of that feature.

Your vectors are organized in a *coordinate system* and commonly rooted in the *origin* (point $[0, 0]$.

3.0.1.1 Linear combinations

You create *linear combinations* of vectors by adding their components (entries in a coordinate). Th All points that you can reach by linear combinations are called the *span* of these vectors. If a vector lies in the span of another vector, they are *linearly dependent*.
You can *scale* (stretch or squish) vectors multiply vectors by *scalars* (i.e. numbers). A vector with length 1 is called *unit vector*. The unit vectors in each direction of the coordinate system are its *basis vectors*. The basis vectors stacked together form an *identity matrix*: a matrix with 1s on its diagonal. Since there are only values on its diagonal it is also a *diagonal matrix*.

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

3.0.1.2 Linear transformations

Linear transformations are functions that move points around in a vector space, while preserving the linear relationships between the points (straight lines stay straight, the origin stays the origin). They include rotations and reflections. You can understand the calculation of the linear transformation of a point as follows: You give the basis vectors a new location. You scale the new location basis vectors with the components of the respective dimension of the vector you want to transform. You take the linear combination of the scaled, transformed basis vectors:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = x \begin{bmatrix} a \\ c \end{bmatrix} + y \begin{bmatrix} b \\ d \end{bmatrix} = \begin{bmatrix} xa + yb \\ xc + yd \end{bmatrix}$$

likewise, you can view matrix vector multiplication as a transformation of your space. Full explanation: youtube.com - 3Blue1Brown Multiplying two matrices represents the sequential combination of two linear transformations in your vector space.

A *transpose* A^T of a matrix A is achieved by mirroring the matrix on its diagonal and therefore swapping its rows and columns. This commonly makes sense when evaluating if elements of two matrices line up in regard to their scale. You can also check if matrices are orthogonal.

An *orthogonal/orthonormal matrix* is a matrix for which holds $A^T A = AA^T = I$, where I is the identity matrix. The columns of orthogonal matrices are linearly independent of each other.

An *inverse matrix* A^{-1} of a matrix A is the matrix that would yield no transformation at all, if multiplied with A.

The *dot product* of two vectors is calculated like a linear transformation between a 1×2 matrix and a 2×1 matrix. It therefor maps onto the 1-D Space and can be used as a measure of collinearity.

The *cross product* of two vectors is a perpendicular vector that describes the parallelogram that the two vectors span. Its magnitude can be seen as the area of the parallelogram. Beware: The order of the vectors in the operation matters. The cross product can be expressed by a determinant. If two vectors are collinear or perpendicular, the cross product is zero.

3.0.1.3 Determinants, rank and column space

Determinants can be used to measure how much a linear transformation compresses or stretches the space. If a transformation inverts the space, the determinant will be negative. If a determinant is 0 it means that the transformation maps the space onto a lower dimension.

$$\det\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}\right) = (a * d) - (c * b)$$

The dimensions that come out of a transformation/matrix are its *rank*. All possible outputs of your matrix (the span constructed by its columns) is the *column space*. All vectors that are mapped to 0 (onto the origin) are the *null space* or *kernel* of the matrix.

Determinants can only be calculated for square matrices. An e.g. 3×2 matrix can be viewed as a transformation mapping from 2-D to 3-D space.

3.0.1.4 System of equations

Linear algebra can help you solve systems of equations.

$$\begin{array}{l} 1x + 2y + 3z = 4 \\ 4x + 5y + 6z = -7 \\ 8x + 9y + 0z = 1 \end{array} \quad \rightarrow \quad \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 8 & 9 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 4 \\ -7 \\ 1 \end{bmatrix} \quad \rightarrow \quad A\vec{x} = \vec{v}$$

You can imagine this as as searching a vector $\vec{x}$ that will land on $\vec{v}$ after the transformation A.

To find $\vec{x}$ you need the *inverse* of A:

$$A^{-1}A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

You now multiply the matrix equation with A^{-1} and get:

$$A^{-1}A\vec{x} = A^{-1}\vec{v} \quad \rightarrow \quad \vec{x} = A^{-1}\vec{v}$$

3.0.1.5 Eigenvalues and Eigenvectors

For a linear transformation A, the eigenvectors $\vec{v}$ represent the vectors that stay on their span (keep orientation) and the eigenvalues λ are the scalars by which the eigenvectors get scaled.

$$A\vec{v} = \lambda\vec{v}$$

Transforming λ to a scaled identity matrix I and factoring out $\vec{v}$, we get:

$$(A - \lambda I)\vec{v} = \vec{0}$$

This tells us, that the transformation $(A - \lambda I)$ needs to map the vector $\vec{v}$ onto a lower dimension.

An *eigenbasis* λI is a basis where the basis vectors are eigenvectors. They will sit on the diagonal of your basis matrix (→ it will be a *diagonal matrix*).

3.0.1.6 Eigenvalue decomposition

An *eigen(value)decomposition* is the decomposition of a matrix into the matrix of eigenvalues and eigenvectors.

$$AU = U\Lambda \quad \rightarrow \quad A = U\Lambda U^{-1}$$

where U is the matrix of the eigenvectors of A and Λ is the eigenbasis. Thus matrix operations can be computed more easily, since Λ is a diagonal matrix.

3.0.1.7 Singular value decomposition

Singular Value decomposition is also applicable to a non-square $m \times n$-matrix (with m rows and n columns). If you have a matrix with rank r, you can decompose it into

$$A = U\Sigma V^T$$

where U is an orthogonal $m \times r$ matrix, Σ is a diagonal $r \times r$ matrix and V^T is an orthogonal $r \times n$ matrix. U contains the *left singular vectors*, V the *right singular vectors* and Σ the $SingularValues$.
This decomposition technique can be used to approximate the original matrix A with only the k largest singular values. This lets you work in a space with only k dimensions given by $U_k\Sigma_k$. Thereby you can save computation time and memory space without loosing a lot of information.

For more detailed explanation, see this Stack Exchange Thread.
For applications, please see SVD for lower dimensional mapping.

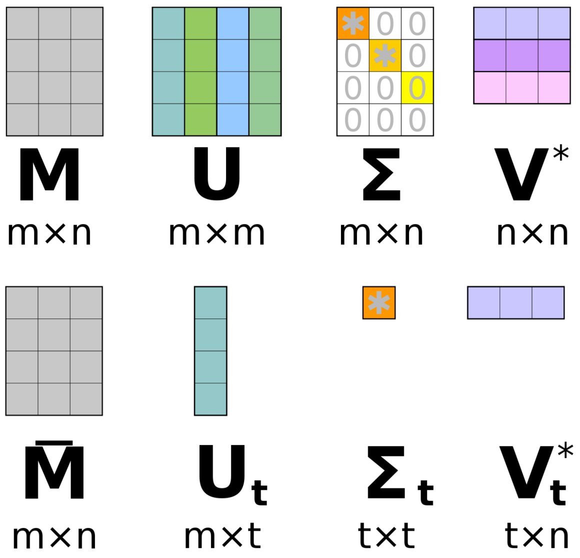


Figure 3.1: SVD and truncated SVD. The upper schema shows the decomposition of the matrix M with m rows and n columns into U, Σ and V. The lower schema shows the truncated SVD for lower dimensional mapping $U_t\Sigma_t$: The first t eigenvalues from Σ have been chosen to reconstruct a compressed version $\bar{M}$. This figure has been adapted from user Cmglee on wikimedia.org.

Part III

Data

4 Similarity and Distance Measures

Choosing the right distance measures is important for achieving good results in statistics, predictions and clusterings.

4.0.1 Metrics

For a distance measure to be called a metric d, the following criteria need to be fulfilled:

- Positivity: $d(x_1, x_2) \geq 0$
- $d(x_1, x_2) = 0$ if and only if $x_1 = x_2$
- Symmetry: $d(x_1, x_2) = d(x_2, x_1)$
- Triangle inequality: $d(x_1, x_3) \leq d(x_1, x_2) + d(x_2, x_3)$

There may be distance measures that do not fulfill these criteria, but those are not metrics.

4.0.2 Similarity measures on vectors

These measures are used in many objective functions to compare data points.

```
from sklearn.metrics import pairwise_distances
X1 = np.array([[2,3]])
X2 = np.array([[2,4]])
pairwise_distances(X1,X2, metric="manhattan")
```

The available metrics in sklearn are: ‘cityblock’, ‘cosine’, ‘euclidean’, ‘l1’, ‘l2’, ‘manhattan’, and from scipy: ‘braycurtis’, ‘canberra’, ‘chebyshev’, ‘correlation’, ‘dice’, ‘hamming’, ‘jaccard’, ‘kulsinski’, ‘mahalanobis’, ‘minkowski’, ‘rogerstanimoto’, ‘russellrao’, ‘seuclidean’, ‘sokalmichener’, ‘sokalsneath’, ‘sqeuclidean’, ‘yule’
More info: scikit-learn.org

4.0.2.1 Manhattan distance

The distance is the sum of the absolute differences of the components (single coordinates) of the two points:

$$d(A,B) = \sum_{i=1}^{d} |A_i - B_i|$$

More info at wikipedia.org.

4.0.2.2 Hamming distance

This metric is used for pairs of strings and works equivalently to the Manhattan distance. It is the number of positions that are different between the strings.
More info at wikipedia.org.

4.0.2.3 Euclidian distance

$$d(A,B) = |A - B| = \sqrt{\sum_{i=1}^{d} (A_i - B_i)^2}$$

More info on the euclidian distance on wikipedia.org.
The usefulness of this metric can deteriorate in high dimensional spaces. See curse of dimensionality

4.0.2.4 Chebyshev distance

The Chebyshev distance is the largest difference along any of the components of the two vectors.

$$d(A,B) = \max_i(|A_i - B_i|)$$

More info at wikipedia.org.

4.0.2.5 Minkowski Distance

$$d(A,B) = (\sum_{i=1}^{d} |A_i - B_i|^p)^{\frac{1}{p}}$$

For $p = 2$ the Minkowski distance is equal to the Euclidian distance, for $p = 1$ it corresponds to the Manhattan distance and it converges to the Chebyshev distance for $p \to \infty$. More info at wikipedia.org.

4.0.3 Kernels

Kernels are functions that output the relationship between points in your data. They correspond to mapping the data into high-dimensional space and allow to implicitly draw nonlinear decision boundaries with linear models. The *kernel trick* denotes, that you don't have to map the points into high dimensional space explicitly.

4.0.3.1 Closure properties of kernels

- If k_1 and k_2 are kernels, then $k_1 + k_2$ is a kernel as well.
- If k_1 and k_2 are kernels, then their product is a kernel as well.
- If k is a kernel and α is a kernel, then αk is a kernel as well.
- If you define k only on a set D, then points that are not in D will have a value of $k_0 = 0$ which is still a valid kernel.

4.0.3.2 Polynomial kernel

$$K(x_1, x_2) = \langle x_1, x_2 + c \rangle^d$$

$\langle\rangle$ is the dot-product. The *linear kernel* is a polynomial kernel with $d = 1$.

4.0.3.3 Gaussian Radial Basis Function (RBF) kernel

This is the most widely used non-linear kernel.

$$K(x_1, x_2) = exp\left(-\frac{||x_1 - x_2||^2}{2\sigma^2}\right)$$

4.0.3.4 constant kernel

4.0.3.5 delta dirac kernel

4.0.3.6 R convolution kernels

4.0.3.7 String kernels

5 Data Analysis

Data analysis is conducted iteratively once you get hold of your data, when you cleaned it, when you processed it and when you analyse the outputs of your model.

5.1 Exploratory data analysis (EDA)

5.1.1 Initial analysis

After getting hold of the data, these are important properties to extract:

```
import pandas as pd
pd.options.display.float_format = '{:,.2f}'.format
print("First 5 samples:"
print(df.head())
print(":.. and last 5 samples:")
print(df.tail())
print("First sample per month:")
print(df_transport.groupby("Month").first())
# The number of non-null values and the respective data type per column:
df.info()
# The count, uniques, mean, standard deviation, min, max, quartiles per column:
df.describe(include='all')
print("rows: "+ df.shape[0])
print("columns: "+ df.shape[1])
print("empty rows: "+ df_transport.isnull().sum())

# Rarely used:
train_df["Day"].unique() # returns unique values in a column
```

Go through this check-list after data import.

5.1.2 After preprocessing

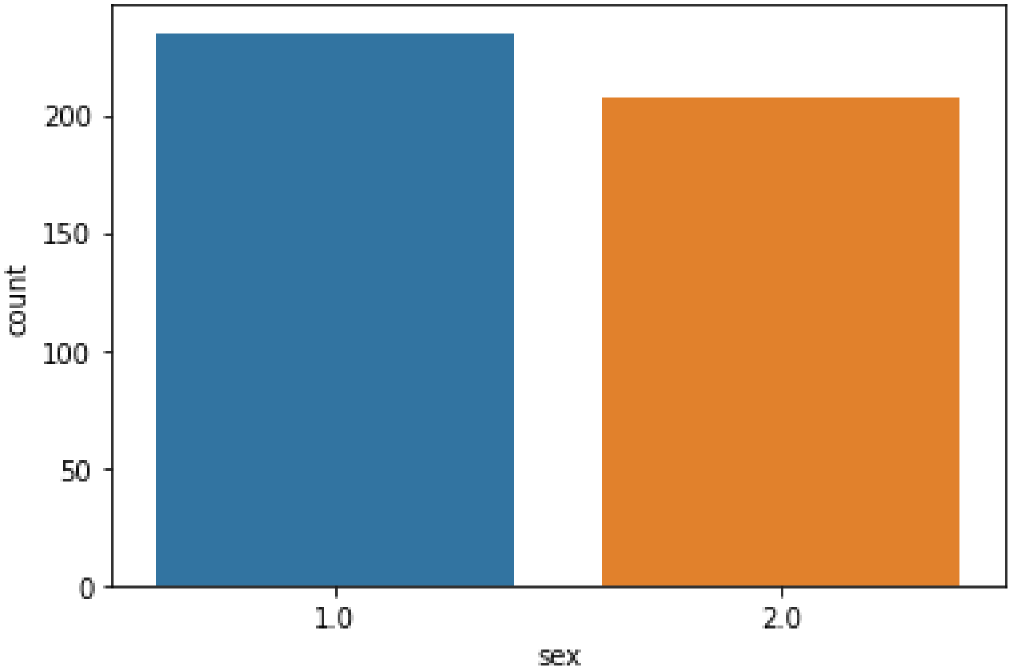
5.1.2.1 Univariate Analysis

Analyse only one attribute.

5.1.2.1.1 Categorical / discrete data: Bar chart

Plot the number of occurrences of each category / number. This helps you find the distribution of your data.

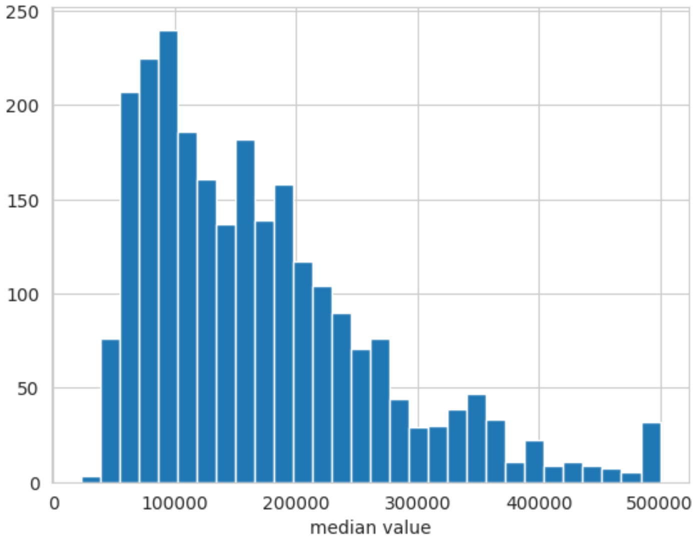
```
import seaborn as sns
import matplotlib.pyplot as plt
sns.countplot(df["sex"])
plt.ylabel("number of participants")
```



5.1.2.1.2 Continuous data

A **histogram** groups data into ranges and plot number of occurrences in each range. This helps you find the distribution of your data.

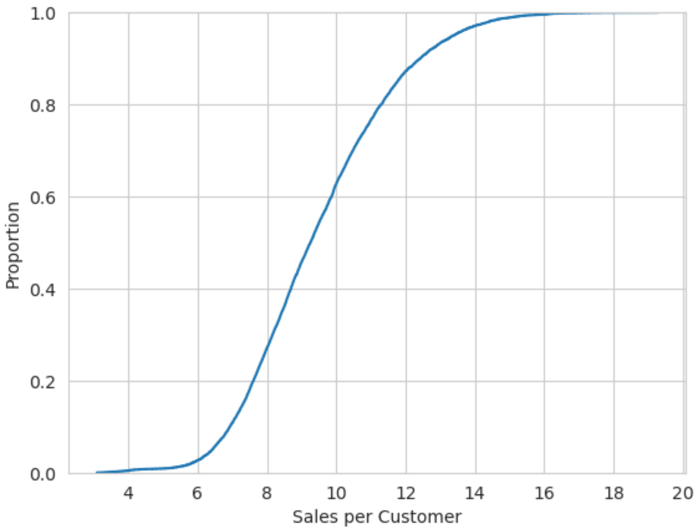
```
import seaborn as sns
import matplotlib.pyplot as plt
sns.set_style('whitegrid')
sns.histplot(data=df_USAhousing, x='median_house_value', bins=30)
plt.xlabel('median value')
```



More info: seaborn.pydata.org

A **empirical cumulative distribution function** shows the proportion of samples with values below a certain value.

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style('whitegrid')
sns.ecdfplot(data=train_df["feature"].sample(10000))
plt.xlabel('Sales per Customer')
```



More info: seaborn.pydata.org

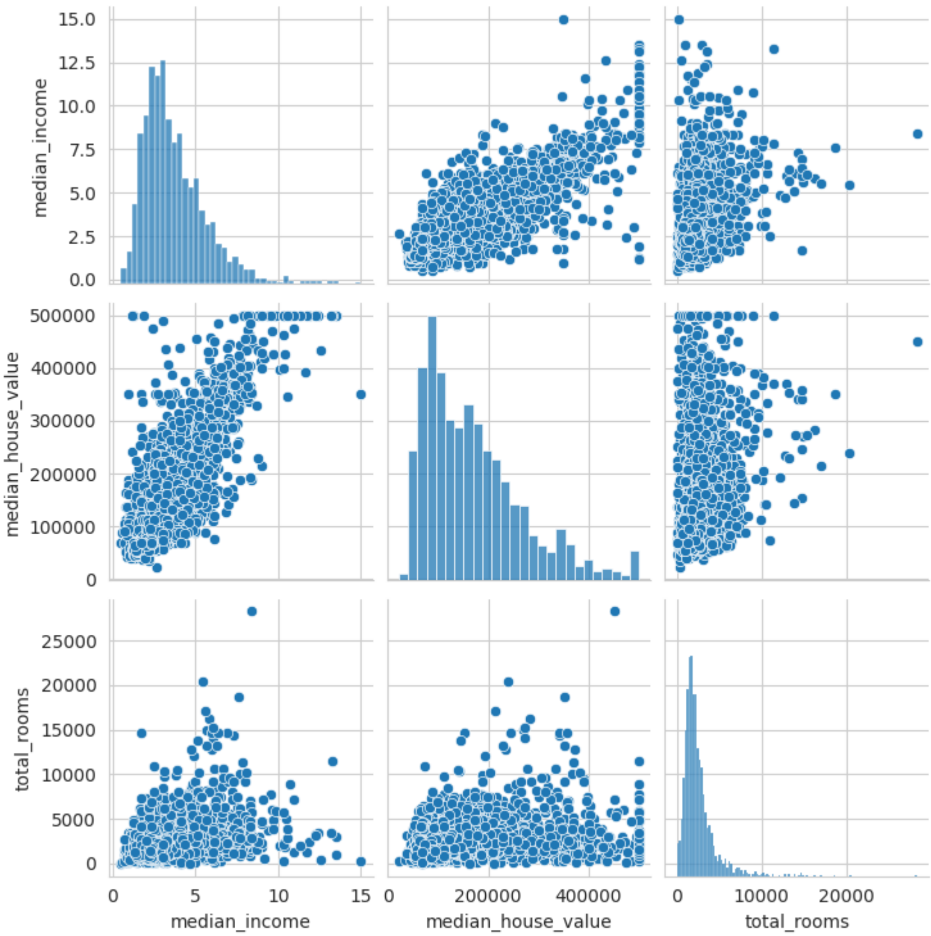
5.1.2.2 Multivariate Analysis

5.1.2.2.1 Continuous vs Continuous

Scatter-plots plot the values of the datapoints of one attribute on the x-axis and the other attribute on the y-axis. This helps you find the correlations, order of the relationship, outliers etc.

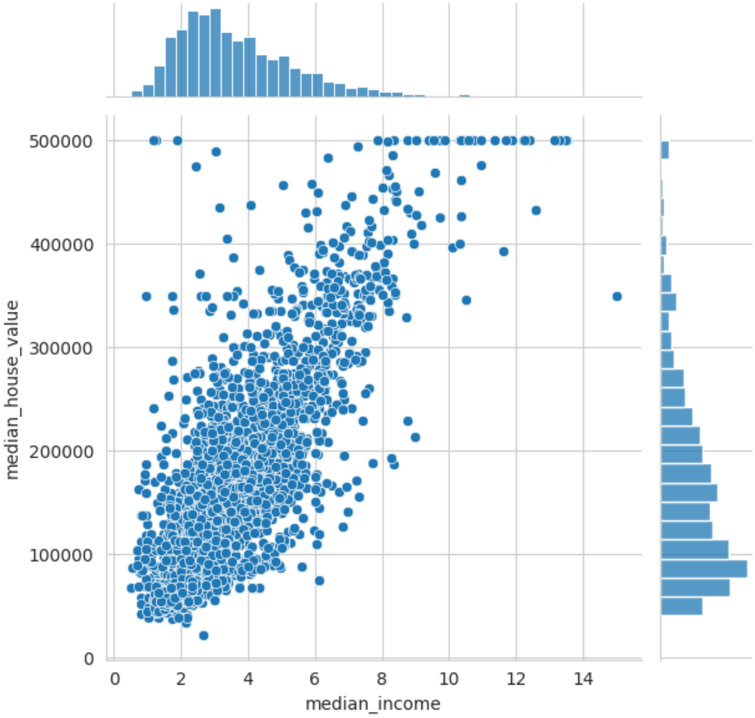
Use a **pairplot** to make a scatter plot of multiple features against each other.

```
import seaborn as sns
sns.pairplot(df_USAhousing[["median_income", "median_house_value", "total_rooms"]], diag_k
```



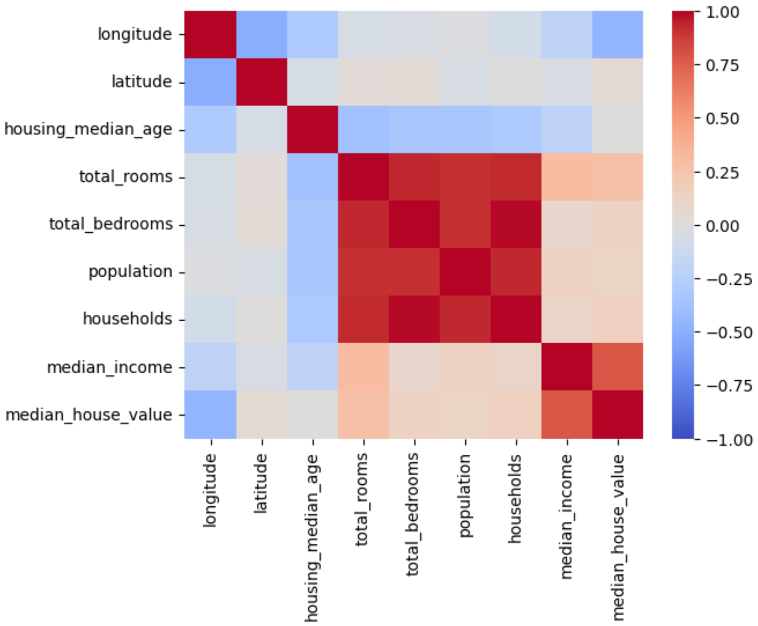
Alternatively use **joint plots**, to visualize the marginal (univariate) distributions on the sides:

```
sns.jointplot(data=df_USAhousing, x="median_income", y="median_house_value")
```



Heatmaps plot the magnitude of values in different categories. It is commonly used in exploratory data analysis to show the correlation of the different attributes.

```
import seaborn as sns
sns.heatmap(df.corr(), cmap="coolwarm", vmin=-1, vmax=1, annot=True)
```

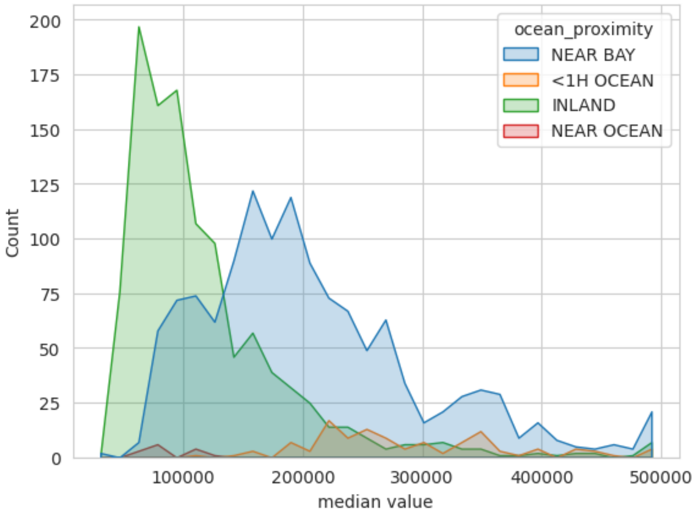


More info: seaborn.pydata.org

5.1.2.2.2 Continuous vs. Categorical data

Overlapping histograms plot the marginal distribution of the continuous distributions, using different colors for each category:

```
import seaborn as sns
sns.set_style('whitegrid')
sns.histplot(data=df_USAhousing, x='median_house_value', hue="ocean_proximity", element="p
plt.xlabel('median value')
```



Use separate **violin plots** for each of the different categories:

```
import seaborn as sns
sns.catplot(data=df, x="cont_col", y="cat_col", hue="binary_col", kind="violin")
```

Use **heatmaps** with two categorical feature as x- and y-axis respectively and a continuous attribute as magnitude ("heat").

```
import seaborn as sns
sns.heatmap(df.pivot(index="cat_col1", columns="cat_col2", values="cont_col"), annot=True,
```

5.1.2.2.3 Categorical vs Categorical

Categorical plots plot the count / percentage of different categorical attributes in side-by-side bar charts

```
import seaborn as sns
sns.catplot(data=df, y="cat_col1", hue="cat_col2", kind="bar")
```

More info: seaborn.pydata.org

5.2 Output Analysis

5.2.1 Performance

See chapters:

- Evaluation of classification models
- Evaluation of regression models
- Evaluation of clustering algorithms

6 Preprocessing data

6.0.1 Joining / merging separate tables

```
import pandas as pd
merged_df = pd.merge(df1, df2, how = "inner", on = "reference_column")
```

More info: pandas.pydata.org

6.0.2 Missing & wrong data

Some algorithms assume that all features of all samples have numerical values. In these cases missing values have to be imputed (i.e. inferred) or (if affordable) the samples with missing feature values can be deleted from the data set.

6.0.2.1 Iterative imputor by sklearn

For features with missing values, this imputor imputes the missing values by modelling each feature using the existing values from the other features. It uses several iterations until the results converge.
! This method scales with $O(nd^3)$, where n is the number of samples and d is the number of features.

```
from sklearn.experimental import enable_iterative_imputer # necessary since the imputor is
from sklearn.impute import IterativeImputer
from sklearn.ensemble import RandomForestRegressor
rf_estimator = RamdomForestRegressor(n_estimators = 8, max_depth = 6, bootstrap = True)
imputor = IterativeImputer(random_state=0, estimator = rf_estimator, max_iter = 25)
imputor.fit_transform(X)
```

More info: scikit-learn.org

6.0.2.2 Median / average imputation

Simply replace missing values with the median or average of the feature:

```
import pandas as pd
df["feature"] = df["feature"].fillna(df["feature"].median())
```

6.0.2.3 Deleting missing values

```
import pandas as pd
df.dropna(how="any") # how="all" would delete a sample if all values were missing
```

More info: pandas.pydata.org

6.0.2.4 Deleting duplicate entries

Duplicate entries need to be removed (exception: time series), to avoid over representation and leakage into test set.

```
import pandas as pd
df.drop_duplicates(keep=False)
```

6.0.2.5 Replacing data

```
import pandas as pd
df.Col.apply(lambda x: 0 if x=='zero' else 1)
```

6.0.2.6 Filter out data

```
import pandas as pd
df = df[(df["Feature1"] == 0) & (df["Feature2"] != 0)]
```

6.0.3 Continuous data

6.0.3.1 Polynomial transform

You spread out small and large values of a feature to help the algorithm to distinguish cases. It can also be used to combine two features to represent mutually supporting effects.

```
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(degree=2, interaction_only=False, include_bias=False)
poly.fit_transform(df[["feature1", "feature2"]])
```

6.0.3.2 Reduce skew

Heavy skew in a distribution can be a problem for many models (outlier effects). To reduce it you can use a **power transform** to map the data to a Gaussian distribution…

```
from sklearn.preprocessing import PowerTransformer
pt = PowerTransformer()
pt.fit_transform(df["skew_feature"])
```

More info: scikit-learn.org

… or a **quantile transform** to map the data to a uniform (or Gaussian) distribution

```
from sklearn.preprocessing import QuantileTransformer
qt = QuantileTransformer(n_quantiles=100, output_distribution="uniform") # alternvative di
qt.fit_transform(df["skew_feature"])
```

More info: scikit-learn.org

6.0.4 Categorical data

There are multiple ways to encode categorical data, especially non-vectorized data, to make it suitable for machine learning algorithms. The string values (e.g. “male”, “female”) of categorical features have to be converted into integers. This can be done by two methods:

6.0.4.1 Ordinal encoding

An integer is assigned to each category (e.g. “male”=0, “female”=1)

```
from sklearn.preprocessing import OrdinalEncoder
ord_enc = OrdinalEncoder(min_frequency=0.05)
ord_enc.fit(X) # multiple columns can be transformed at once
X_transf = ord_enc.transform(X)
```

More info: scikit-learn.org
This method is useful when the categories have an ordered relationship (e.g. “bad”, “medium”, “good”). If this is not the case (e.g. “dog”, “cat”, “bunny”) this is to be avoided since the algorithm might deduct an ordered relationship where there is none. For these cases one-hot-encoding is to be used.

For **encoding the label** for classification tasks, you can also use the scikit-learn’s `LabelEncoder`. More info here: scikit-learn.org

6.0.4.2 One-hot encoding

One-hot encoding assigns a separate feature-column for each category and encodes it binarily (e.g. if the sample is a dog, it has 1 in the dog-column and 0 in the cat and bunny column).

6.0.4.3 sklearn

```
from sklearn.preprocessing import OneHotEncoder
onehot_enc = OneHotEncoder(handle_unknown='ignore')
onehot_enc.fit(X)
onehot_enc.transform(X)
```

More info: scikit-learn.org

6.0.4.4 Pandas

```
import pandas as pd
pd.get_dummies(X, columns = ["Sex", "Type"], drop_first=True)
```

More info: pandas.pydata.org

6.0.4.5 Discretizing / binning data

You can discretize features and targets from continuous to discrete/categorical (e.g. age in years to child, teenager, adult, elderly).

```
pd.cut(df["Age"], bins=[0,12, 20, 65, 150], labels =["child", "teenager", "adult", "elderl
```

More info: pandas.pydata.org

Pros:

- It makes sense for the specific problem (e.g. targeting groups for marketing).
- Improved signal-to-noise ratio (bins work like regularization).
- possibly highly non-linear relationship of continuous feature to target is hard to learn for model.
- Better interpretability of features, results and model.
- Can be used to incorporate domain knowledge and make learning easier.

Cons:

- Your model and results lose information
- Senseless cut-offs between bins can create "artificial noise" and make learning harder.

More info: stackexchange.com
See also: wikipedia: Sampling (signal processing).

6.0.4.6 Combining rare categories

Rare categories can lead to noise in the data and blow up the amount of features when using one-hot encoding. These categories should be combined, when there are only few occurrences (e.g. When analysing page visits, combine the categories "blackberry", "jolla", "windows phone" into the category "other").

6.0.4.7 Pandas

```
import pandas as pd
import numpy as np
counts_ser = pd.value_counts(df["feature"])
categories_to_mask = counts_ser[(counts_ser/counts_ser.sum()).lt(0.05)].index # using 5% c
```

```
df["feature"] = np.where(df["feature"].isin(categories_to_mask),'other',df["feature"])
```

More info: stackoverflow

6.0.4.8 sklearn

In sklearn, rare categories can be filtered out when one-hot encoding the feature using the parameter `min_frequency`.

```
from sklearn.preprocessing import OneHotEncoder
enc = OneHotEncoder(handle_unknown='ignore', min_frequency=0.05)
enc.fit_transform(df["feature"])
```

More info:scikit-learn.org

6.0.4.9 PyCaret

Use the parameter `rare_to_value` of the `setup` function.

```
from pycaret.time_series import ClassificationExperiment # or use other Experiment type
exp = ClassificationExperiment()
exp.setup(train_df, target="Sales", rare_to_value = 0.05)
```

More info: PyCaret Docs

6.0.5 Date- and time-data

You can convert to the **datetime format** as follows:

```
import pandas as pd
pd.to_datetime(df.date_col, infer_datetime_format=True)
```

You create columns for **year**, **month**, **day** like this:

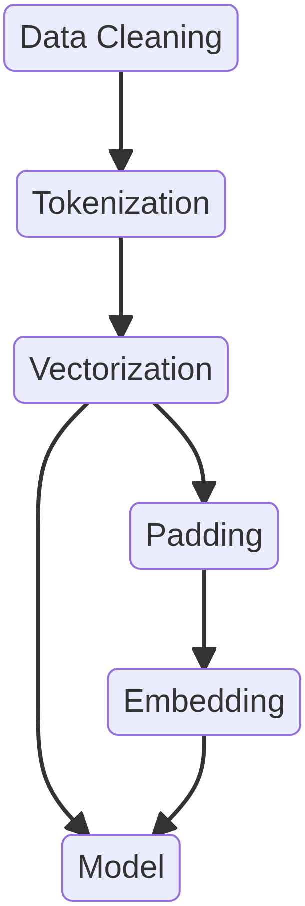
```
import pandas as pd
df['year'] = df.Date.dt.year
df['month'] = df.Date.dt.month
df['day'] = df.Date.dt.day
```

6.0.6 Graph representation of data

The similarity/distance between points can be represented in graphs. The data points are represented as nodes, the distances/similarities as edges.

6.0.7 Text data

These are the common steps of pre-processing text data:



6.0.7.1 Cleaning text data

The aim is to remove errors, parts that are irrelevant for the task and to standardize.

6.0.7.2 clean-text

The Clean-text only requires one command for several cleaning tasks:

Install the packge:

```
pip install clean-text
```

Usage (see steps in parameters):

```
from cleantext import clean

clean("some input",
    fix_unicode=True,               # fix various unicode errors
    to_ascii=True,                  # transliterate to closest ASCII representation
    lower=True,                     # lowercase text
    no_line_breaks=False,           # fully strip line breaks as opposed to only normalizi
    no_urls=False,                  # replace all URLs with a special token
    no_emails=False,                # replace all email addresses with a special token
    no_phone_numbers=False,         # replace all phone numbers with a special token
    no_numbers=False,               # replace all numbers with a special token
    no_digits=False,                # replace all digits with a special token
    no_currency_symbols=False,      # replace all currency symbols with a special token
    no_punct=False,                 # remove punctuations
    replace_with_punct="",          # instead of removing punctuations you may replace the
    replace_with_url="<URL>",
    replace_with_email="<EMAIL>",
    replace_with_phone_number="<PHONE>",
    replace_with_number="<NUMBER>",
    replace_with_digit="0",
    replace_with_currency_symbol="<CUR>",
    lang="en"                       # set to 'de' for German special handling
)

# or simply:
clean("some input", all= True)

# use within pandas:
df["text"] = df["text"].apply(lambda txt : cleantext.clean_words(txt))
```

The command `clean_words` additionally returns the words as a list.

More info:
aim - Guide to CleanText

Gitub - clean-text repo

6.0.7.3 Pandas

```
import pandas as pd
import re

df["text"] = df["text"].str.lower()               # make all words lowercase
df["text"] = df["text"].str.replace('ü', 'u')     # replace characters
df["text"] = df["text"].str.replace(r"https?:\/\/.\S+","", regex = True) # remove URLs
df["text"] = df["text"].str.replace(r"<.*?>","", regex = True) # remove html-tags

# Reference : https://gist.github.com/slowkow/7a7f61f495e3dbb7e3d767f97bd7304b
def remove_emoji(text):
    emoji_pattern = re.compile("["
                           u"\U0001F600-\U0001F64F"  # emoticons
                           u"\U0001F300-\U0001F5FF"  # symbols & pictographs
                           u"\U0001F680-\U0001F6FF"  # transport & map symbols
                           u"\U0001F1E0-\U0001F1FF"  # flags (iOS)
                           u"\U00002702-\U000027B0"
                           u"\U000024C2-\U0001F251"
                           "]+", flags=re.UNICODE)
    return emoji_pattern.sub(r'', text)
df["text"] = df["text"].apply(lambda text : remove_emoji(text))

df["text"] = df["text"].str.strip()               # strip away leading and trailing spaces
df["text"] = df["text"].str.replace(r"[^\w\s]", "", regex = True) # remove punctuation

# Rarely used
df["text"] = df["text"].str.lstrip("123456789")   # strip away leading numbers rstrip for
df["text"] = df["text"].str.replace(r"\(.*?\)","", regex = True) # remove everything betwe
df["year"] = df["year"].str.extract(r'^(\d{4})', expand=False) # extracts year numbers
```

6.0.7.4 Tokenization

Tokenization is the act of splitting a text into sentences or words (i.e. tokens).

6.0.7.4.1 Word-Tokenization

6.0.7.5 NLTK

Split the text into words:

```
import nltk
nltk.download('punkt')
from nltk.tokenize import word_tokenize
words = word_tokenize(cleaned_text)
```

6.0.7.6 SpaCy

SpaCy uses a sophisticated text annotation method.

1. Download trained English linguistic annotation model:

```
!python -m spacy download en_core_web_sm
```

2. Tokenize text:

```
import spacy
nlp = spacy.load("en_core_web_sm")
doc = nlp(text_doc)
tokens = [(token.text, token.pos_, token.dep_) for token in doc]
```

Attributes:
`pos_`: Part-of-speech (e.g. noun, adjective, punctuation),
`dep_`: Syntactic dependency relation (e.g. “Does … have” → Does (auxiliary verb), have (root verb))

More info:
SpaCy - Features

6.0.7.6.1 Sentence Tokenization

6.0.7.7 NLTK

```
from nltk.tokenize import sent_tokenize
sentences = sent_tokenize(sentences_text)
```

6.0.7.8 SpaCy

```
import spacy
nlp = spacy.load("en_core_web_sm")
doc = nlp(text_doc)
sentences = [sent for sent in doc.sents]
```

More info: Tutorial on SpaCy Sentencer

6.0.7.9 Vectorization

Transform sequence of tokens into numerical vector that can be processed by models.

6.0.7.9.1 Word count encoding

This is part of the bag-of-words method. It works as follows:

1. Create a vocabulary / corpus of all words in the training data.
2. Each word in the vocabulary becomes its own feature
3. For each document, count how many times the word occurs.

```
from sklearn.feature_extraction.text import CountVectorizer
count_vect = CountVectorizer()
word_counts = count_vect.fit_transform(doc_array)
```

More info: sklearn - extract features from text

Pros:

- Simple and easily interpretable.

Cons:

- Order and relation between words is lost
- Sparse representation is not easily usable for many models. (Large vocabularies make it worse → Use stemming)

6.0.7.9.2 Term frequency-inverse document frequency (tf-idf)

This measure reflects the importance of a word to a document:
Term frequency: What is the frequency of this word in this document.
Inverse document frequency: How rare is this word among all documents.
Thus, terms that occur a lot in one document but rarely in others get a higher value.

```
from sklearn.feature_extraction.text import TfidfTransformer
tf_transformer = TfidfTransformer()
word_tf_idfs = tf_transformer.fit_transform(word_counts) # uses wordcounts from count-vect
```

More info: sklearn - extract features from text

6.0.7.10 Padding

Since some sequences are shorter than others, we need to fill up the remaining parts of them ones with zeros. Thus we achieve sequences of the same length. First we need to make an ordinal encoding and create word-sequences.

```
from keras.preprocessing.text import Tokenizer
from keras.utils import pad_sequences
# Convert text to sequence
tokenizer = Tokenizer(num_words = vocab_size)
tokenizer.fit_on_texts(train_texts)
X_train_sequences = tokenizer.texts_to_sequences(train_texts)
# Padd the sequences
train_texts_padded = pad_sequences(train_texts, padding='post', maxlen=max_sequence_length
```

6.0.7.11 Embedding

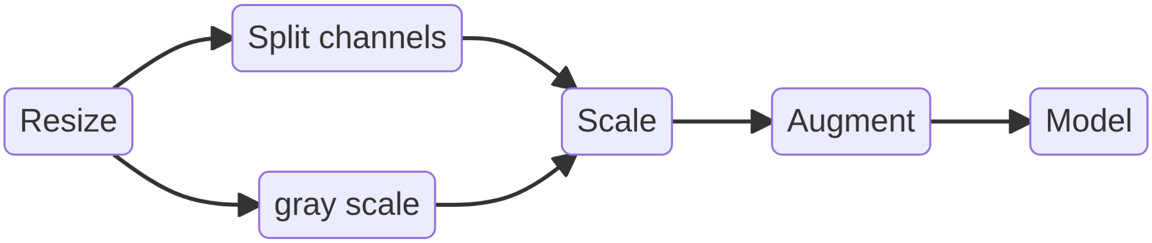
Embedding is the mapping of words from the sparse one-hot-encoded space into a dense space, that should reflect the meaning of the words (i.e. similar words are close together).

This is done in neural networks via an embedding layer:

```
model = Sequential()
model.add(layers.Embedding(input_dim=vocab_size,
                           output_dim=embedding_dim,
                           input_length=sequence_length))
# ... add further layers ...
model.compile()
```

You can reuse trained embeddings for other tasks. See transfer-learning More info: Google Machine Learning - Prepare Your Data

6.0.8 Image data



6.0.8.1 Keras ImageDataGenerator

```
from tf.keras.preprocessing.image import ImageDataGenerator
datagen = ImageDataGenerator(samplewise_std_normalization=True,
                            rotation_range=180,
                            shear_range=20,
                            zoom_range=0.1,
                            horizontal_flip=True,
                            vertical_flip=True,
                            validation_split=0.7)
imgs_train = datagen.flow_from_directory(directory = "data/dir",
                                         target_size=(256, 256),
                                         batch_size=32,
                                         class_mode="categorical", # classes will be deter
                                         subset="training")
imgs_test = datagen.flow_from_directory(directory = "data/dir",
                                        target_size=(256, 256),
                                        batch_size=32,
                                        class_mode="categorical", # classes will be deter
                                        subset="validation")
```

More info: keras.io

6.0.8.2 Keras image-loader

Load image dataset

```
from tf.keras.utils import image_dataset_from_directory
imgs_train, imgs_test = image_dataset_from_directory(directory="path/tofolder", labels="in
    label_mode="int")
```

load single image

```
from tf.keras.utils import load_img
from tf.keras.utils.image import img_to_array
img = load_img(path="path/toimg.png", grayscale=False, color_mode="rgb", target_size=(256,
img = img_to_array(img)
```

More info: keras.io

Augmentation

```
from tf.keras import layers, Sequential
import numpy as np

resize_and_rescale = Sequential([
  layers.Rescaling(1./255),
  layers.RandomFlip("horizontal_and_vertical"),
  layers.RandomRotation(0.2),
])

images = []
for idx in range(10):
  augmented_image = data_augmentation(img)
  images.append(augmented_image)

img_ar = np.array(images)
```

More info: keras.io

6.0.8.3 OpenCV

```
# Adapted from https://github.com/bnsreenu
import os
import numpy as np
import glob # To go through folders
import cv2
```

```
train_split = 0.7
img_size = 256

images_train = []
images_test = []
labels_train = []
labels_test = []

for dir_path in img_dir:
    label = dir_path.split("/")[-1]
    print(label)
    img_paths = glob.glob(os.path.join(dir_path, "*.jpeg"))
    for img_idx, img_path in enumerate(img_paths):
        img = cv2.imread(img_path, cv2.IMREAD_COLOR)
        img = cv2.resize(img, (img_size, img_size))
        img = cv2.cvtColor(img, cv2.COLOR_RGB2BGR)
        img = img/(255/2)-1 # scales 0 to 255 range to -1 to 1 (more or less zero-centered
        if img_idx < train_split * len(img_paths):
            images_train.append(img)
            labels_train.append(label)
            # flip image horizontally:
            images_train.append(cv2.flip(img, 1))
            labels_train.append(label)
            # flip image vertically:
            images_train.append(cv2.flip(img, 0))
            labels_train.append(label)
        else:
            images_test.append(img)
            labels_test.append(label)

images_train = np.array(images_train)
labels_train = np.array(labels_train)
images_test = np.array(images_test)
labels_test = np.array(labels_test)
```

scikit-image - userguide

Neptune.ai - Image processing methods you should know

6.1 Standardization

Many machine learning models assume that the features are centered around 0 and that all have a similar variance. Therefore the data has to be centered and scaled to unit variance before training and prediction.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit_transform(input_df)
```

More info: scikit-learn.org

Another option for scaling is normalization. This is used, when the values have to fall strictly between a max and min value.
More info: scikit-learn.org

6.2 Splitting in training- and test-data

You need to split your training set into test- and training-samples. The algorithm uses the training samples with the known label/target value for fitting the parameters. The test-set is used to determine if the trained algorithm performs well on new samples as well. You need to give special considerations to the following points:

- Avoiding data or other information to leak from the training set to the test-set
- Validating if the predictive performance deteriorates over time (i.e. the algorithm will perform worse on new samples). This is especially important for models that make predictions for future events.
- Conversely, sampling the test- and training-sets randomly to avoid introducing bias in the two sets.

```
# assuming you already imported the data and separated the label column:
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)
```

More info: scikit-learn.org

6.3 Feature selection

Usually the label does not depend on all available features. To detect causal features, remove noisy ones and reduce the running and training costs of the algorithm, we reduce the amount of features to the relevant ones. This can be done a priori (before training) or using wrapper methods (integrated with the prediction algorithm to be used).
! There are methods that have feature selection already built-in, such as decision trees.

6.3.1 A priori feature selection

A cheap method is to remove all features with **variance** below a certain threshold.

```
from sklearn.feature_selection import VarianceThreshold
selector = VarianceThreshold(threshold=0.1)
selector.fit_transform(X)
```

More info: scikit-learn.org

The **Mutual information score** works by choosing the features that have the highest dependency between the features and the label. {#mutual_info}

$$I(X,Y) = D_{KL}\left(P(X=x,Y=y), P(X=x)\otimes P(Y=y)\right) = \sum_{y\in Y}\sum_{x\in X} P(X=x,Y=y)\log\Bigg(\frac{P(X=x,Y=y)}{P(X=x)P(Y=y}$$

where, D_{KL} is the Kullback–Leibler divergence (A measure of similarity between distributions). The log-Term is for quantifying how different the joint distribution is from the product of the marginal distributions.

```
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import mutual_info_classif # for regression use mutual_info
X_new = SelectKBest(mutual_info_classif, k=8).fit_transform(X, y)
```

More info: scikit-learn.org
wikipedia.org/wiki/Mutual_information

6.3.2 wrapper methods

Using **greedy feature selection** as a wrapper method, one commonly starts with 0 features and adds the feature that returns the highest score with the used classifier.

```python
from sklearn.feature_selection import SequentialFeatureSelector
from sklearn.tree import DecisionTreeClassifier
classifier = DecisionTreeClassifier()
selector = SequentialFeatureSelector(classifier, n_features_to_select=8)
selector.fit_transform(X, y)
```

More info: scikit-learn.org

Part IV

Supervised learning

Classification is the assignment of objects (data points) to categories (classes). Regression allows you to assign a continuous output to your data by estimating the relationship between your features and the output.

Both require a training data-set of points with known class labels and a test data-set for evaluation.

7 General methods and concepts

7.1 Hyper-parameter tuning

The hyper-parameters (e.g. kernel, gamma, number of nodes in tree) are not trained by algorithm itself. An outer loop of hyper-parameter tuning is needed to find the optimal hyper parameters.
! It is strongly recommended to separate another validation set from the training set for hyper-parameter tuning (you'll end up with training-, validation- and test-set). See Cross Validation for best practice.

7.1.1 Grid search

The classic approach is exhaustive grid search: You create a grid of hyper-parameters and iterate over all combinations. The combination with the best score is used in the end. This approach causes big computational costs due to the combinatorial explosion.

```
from sklearn.model_selection import GridSearchCV # combines grid search with cross-validat
from sklearn.neighbors import KNeighborsClassifier

kn_model = KNeighborsClassifier(n_neighbors=3)
parameters = {"n_neighbors": range(2,10), "p": [1,2], "weights": ["uniform", "distance"]}
clf = GridSearchCV(kn_model, parameters, cv=5)
clf.fit(X_train, y_train)
```

More info: scikit-learn.org

7.1.2 Randomized search

This approach is used, if there are too many combinations of hyper-parameters for tuning. You allocate a budget of iterations and the combinations of parameters are sampled randomly according to the distributions you provide.

If you want to evaluate on a large set of hyperparameters, you can use a halving strategy: You tune a large combination of parameters on few resources (e.g. samples, trees). The best performing half of candidates is re-evaluated on twice as many resources. This continues until the best-performing candidate is evaluated on the full amount of resources.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.experimental import enable_halving_search_cv  # since this method is still ex
from sklearn.model_selection import HalvingRandomSearchCV
from sklearn.utils.fixes import loguniform

rf_clf = RandomForestClassifier()

param_distributions = {"max_depth": [3, None],
                       "min_samples_split": loguniform(1, 10)}
hypa_search = HalvingRandomSearchCV(rf_clf, param_distributions,
                               resource='n_estimators',
                               max_resources=10,
                               n_jobs=-1, # important since hyper-parameter tuning is very
                               scoring = 'balanced_accuracy',
                               random_state=0).fit(X, y)
```

More info: scikit-learn.org

7.2 Model selection

The candidates for hyper-parameters must not be evaluated on the same data that you trained it on (over-fitting risk). Thus, we separate another data-set from the training data: The validation set. This is reduces the amount of training data drastically. Therefore we use the approaches of Cross Validation and Bootstrapping.

7.2.1 Cross Validation

In k-fold Cross Validation, we split the training set into k sub-sets. We train on the samples in k-1 sub-sets and validate using the data in the remaining sub-set. We iterate until we have validated on each sub-set once. We then average out the k scores we obtain.

```
from sklearn import svm
from sklearn.model_selection import cross_val_score
SVM_clf = svm.SVC (kernel='polynomial')
```

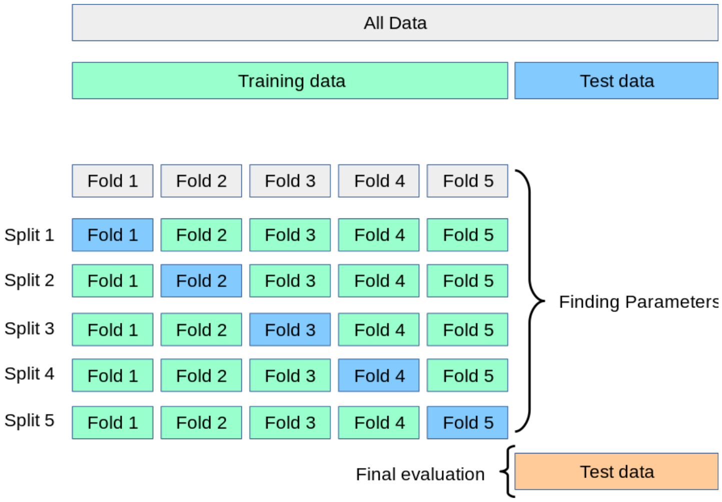


Figure 7.1: Schema of the process for 5-fold Cross Validation. The data is first split into training- and test-data. The training data is split into 5 sub-sets. The algorithm is trained on 4 sub-sets and evaluated on the remaining sub-set. Each sub-set is used for validation once. *Source: scikit-learn.org*.

```
cv_scores = cross_val_score(SVM_clf, X, y, cv = 7)
cv_score = cv_scores.mean()
```

More info: scikit-learn.org

! If you have time-series data (and other clearly not i.i.d.) data, you have to use special cross-validation strategies. There are further strategies worth considering.

7.2.1.1 Bootstrapping

Instead of splitting the data into k subsets, you can also just sample data into training and validation sets.
More info: wikipedia.org.

7.2.2 Errors & regularization

There are irreducible errors and reducible errors. Irreducible errors stem from unknown variables or variables we have no data on. Reducible errors are deviations from our model to its desired behavior and can be reduced. Bias and variance are reducible errors.

$$\text{Error} = \text{Bias} + \text{Var} + \text{irr. Error}$$

7.2.3 Bias and Variance

7.2.3.1 Bias of an estimator

Bias tells you if your model oversimplifies the true relationship in your data (underfitting). You have a model with a parameter $\hat{\theta}$ that is an estimator for the true θ. You want to know whether your model over- or underestimates the true θ systematically.

$$\mathrm{Bias}[\hat{\theta}] = \mathrm{E}_{X|\mathcal{D}}[\hat{\theta}] - \theta$$

E.g. if the parameter captures how polynomial the model / relationship of your data is, a too high value means that your model is underfitting.

More info: wikipedia.org

7.2.3.2 Variance of an estimator

Variance tells you if your model learns from noise instead of the true relationship in your data (overfitting).

$$\mathrm{Var}[\hat{\theta}] = \mathrm{E}_{X|\mathcal{D}}[(\mathrm{E}_{X|\mathcal{D}}[\hat{\theta}] - \hat{\theta})^2]$$

i.e. If you would bootstrap your data, it would show you how much your parameter would jump around its mean, when it learns from the different sampled sets.

Your goal is now to find the sweet spot between a too biased (too simple model) and a model with too high variance (too complex model).

More info: wikipedia.org

7.2.4 Regularization

To combat overfitting, we can introduce a term into our loss-function that penalizes complex models. For linear regression, our regularized loss function is will be:

$$\min L(\hat{y}, y) = \min_{W,b} f(WX + b, y) + \lambda R(W)$$

where f is the unregularized loss function, W is the weight matrix, X is the sample matrix and b is the bias or offset term of the model (bias term $\neq$ bias of estimator!). R is the

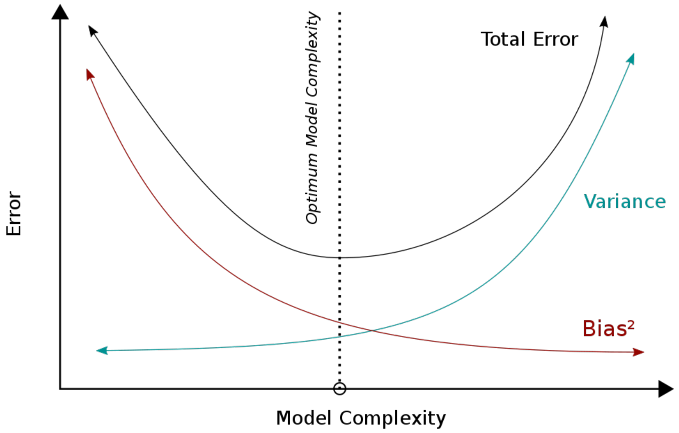


Figure 7.2: Relationship between bias, variance and the total error. The minimum of the total error lies at the best compromise between bias and variance. *Source: User Bigbossfarin on wikimedia.org.*.

regularization function and λ is a parameter controlling its strength.
i.e. The regularized loss function punishes large weights W and leads to flatter/smoother functions.

More info: wikipedia.org

7.2.5 Bagging

Train several instances of a complex estimator (aka. strong learner, like large decision trees or KNN with small radius) on a subset of the data. Then use a majority vote or average the scores for classifying to get the final prediction. By training on different subsets and averaging the results, the chances of overfitting are greatly reduced.

```
from sklearn.ensemble import BaggingClassifier
from sklearn.neighbors import KNeighborsClassifier
bagging = BaggingClassifier(KNeighborsClassifier(), max_features=0.5, n_estimators=20)
```

More info: scikit-learn.org

A classic example for a bagging classifier is Random Forest Classifier or its variant Extremely Randomized Trees which further reduces variance and increases bias.

7.2.6 Boosting

Compared to bagging, we use weak learners that are not trained independently of each other. We start with a single weak learner (e.g. a small decision tree) and repeat the following steps:

1. Add an additional model and train it.
2. Increase weights of training samples that are falsely classified, decrease weights of correctly classified samples. (to be used by next added model.)
3. Reweight results from the models in the combined model to reduce the training error.

The final model is an weighted ensemble of weak classifiers.
The most popular ones are gradient boosted decision tree algorithms.

7.2.7 Stacking

Stacking closely resembles bagging: An ensemble of separately trained base models is used to create an ensemble model. However, the continuous (instead of discrete) outputs of commonly fewer heterogeneous models (instead of same type of models) are used. The continuous outputs are then fed into a final estimator (commonly logistic regression classifier).

```
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline
from sklearn.ensemble import StackingClassifier

classifiers = [
    ('svc', SVC()),
    ('knn', KNeighborsClassifier()),
    ('dtc', DecisionTreeClassifier())
    ]

clf = StackingClassifier(
    classifiers=estimators, final_estimator=LogisticRegression()
    )

clf.fit(X, y)
```

More info: scikit-learn.org

8 Classification

8.1 Evaluation of Classifiers

8.1.1 Confusion matrix

This gives a quick overview on the distribution of true positives (TP), false positives (FP) , TN true negatives, FN false negatives.

	predicted positive	*predicted negative*
actual positive	TP	FN
actual negative	FP	TN

```
from sklearn.metrics import confusion_matrix
import seaborn as sns
confusion = confusion_matrix(y_true=y_test, y_pred=y_pred, labels=categories_list)
sns.heatmap(confusion, annot=True, xticklabels=categories_list, yticklabels=categories_lis
```

8.1.2 Basic Quality Measures

- Accuracy / Success Rate $= \frac{\text{correct predictions}}{\text{total predictions}} = \frac{\text{TP+TN}}{\text{TP+TN+FP+FN}}$
  This metric should only be used in this pure form, when the number of positive and negative samples are balanced.
- Precision $= \frac{\text{TP}}{\text{TP+FP}}$
  i.e. How many of your positive predictions are actually positive?
- True positive rate / Recall / Sensitivity $= \frac{\text{TP}}{\text{TP+FN}}$
  i.e. How many of the positive samples did you catch?
- True negative rate / Specificity / Selectivity $= \frac{\text{TN}}{\text{TN+FP}}$
  i.e. How many of the negative samples did you catch as negative (i.e. are truly negative)?
- F-score $= 2\frac{\text{precision}*\text{recall}}{\text{precision+recall}}$
  This is useful in cases of unbalanced classes to balance the trade-off between precision and recall.

```
from sklearn.metrics import classification_report
classification_report(y_true, y_pred)
```

more info: scikit-learn.org

8.1.3 Area under the Curve

This class of measures represents the quality of the classifier for different threshold values θ by calculating the area under the curve spanned by different quality measures.

8.1.3.1 Area under the Receiver Operating Characteristics Curve (AUROC or AUC)

The AUC can be interpreted as follows: When the classifier gets a positive and a negative point, the AUC shows the probability that the classifier will give a higher score to the positive point. A perfect classifier has an AUC of 1, and AUC of 0.5 represents random guessing.

```
from sklearn.metrics import roc_auc_score
roc_auc_score(y, clf.decision_function(X)) # instead of dec. func. you can use clf.predict
```

More info: scikit-learn.org

! This measure is not sensitive to class imbalance!

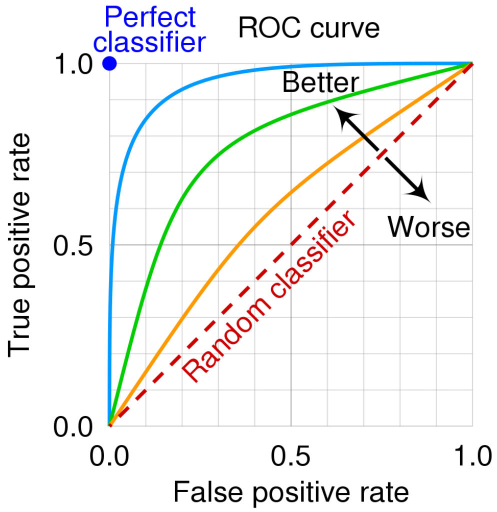


Figure 8.1: Area under the precision recall curve. Source: user cmglee on wikipedia.org

8.1.3.2 Area under the Precision-Recall Curve (AUPRC) / Average Precision (AveP)

This measure can be used for unbalanced data sets. It represents the average precision as a function of the recall. The value of 1 represents a perfect classifier.

```
from sklearn.metrics import average_precision_score
average_precision_score(y_true, y_pred)
```

More info: scikit-learn.org

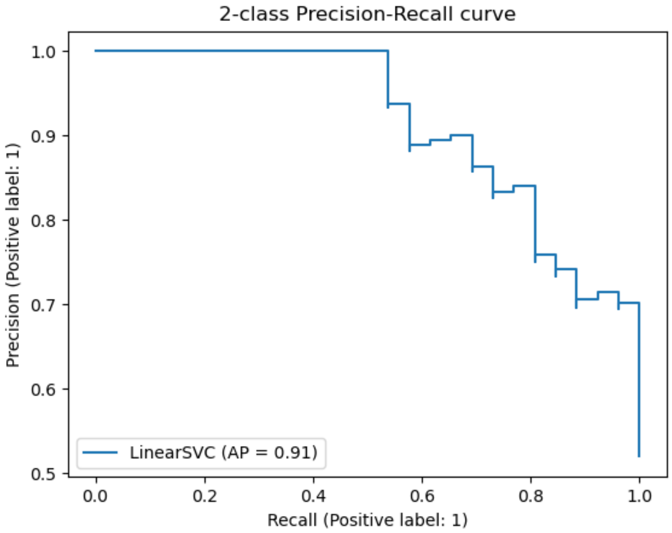


Figure 8.2: The precision-recall curve. Source: scikit-learn.org

8.1.4 Handling Unbalanced Data

Having many more samples in one class than the others during training can lead to high accuracy values event though the classifier performs poorly on the smaller classes. You can handle the unbalance by:

- up-sampling the smaller data set (creating more artificial samples for that class)
- giving more weight to the samples in the smaller data set
- using a quality measure that is sensitive to class imbalance

Oversampling using imbalanced-learn (see:)

```
from imblearn.over_sampling import RandomOverSampler
ros = RandomOverSampler(random_state=0)
features_resampled, labels_resampled = ros.fit_resample(df[feature_cols], df[label_col])
```

Unbalance-sensitive quality measures: Sensitivity, specificity, precision, recall, support and F-score

```
y_true = df[label_col]
y_pred = classifier.predict(df[feature_cols])

from imblearn.metrics import sensitivity_specificity_support
sensitivity, specificity, support = sensitivity_specificity_support(y_true, y_pred)

from sklearn.metrics import precision_recall_fscore_support
precision, recall, fscore, support = precision_recall_fscore_support(y_true, y_pred)
```

8.1.5 Nearest Neighbors Classifier

This classifier predicts the class label using the most common class label of its k nearest neighbors in the training data set.

Pros:

- Classifier does not take time for training.
- Can learn complex decision boundaries.

Cons:

- The prediction is time consuming and scales with n.

```
from sklearn.neighbors import KNeighborsClassifier
kn_model = KNeighborsClassifier(n_neighbors=5)
kn_model.fit(X, y)
kn_model.predict([[5,1]])
```

scikit-learn.org

8.2 Naive Bayes Classifier

Naive Bayes classifies works on the assumption that the features are conditionally independent given the class label. For every point a simplified version of Bayes rule is used:

$$P(Y = y_i | X = x) \propto P(X = x | Y = y_i) * P(Y = y_i)$$

where Y is the RV for the class label and X is the RV that contains the feature values. This holds since $P(X = x)$ is the same for all classes. Since the different features X_j are assumed to be independent they can be multiplied out. The label y_i with the highest probability is the predicted class label:

$$\arg\max_{y_i} P(Y = y_i | X = x) \propto P(Y = y_i) \prod_{j=1}^{d} P(X_j = x_j | Y = y_i)$$

One usually estimates the value of $P(Y)$ as the ferquency of the different classes in the training data or assumes that all classes are equally likely.
To estimate the $P(X_j = x_j | Y = y_i)$ the following distributions are commonly used:

- For binary features: Bernoulli distribution
- For discrete features: Multinomial distribution
- For continuous features: Normal / Gaussian distribution

For discrete features, you need to use a smoothing prior (add 1 to every feature count) to avoid 0 probabilities for samples with features being 0 in the training data.

Pros:

- Naive Bayes training is fast.
- Combine descrete and continuous features since a different distribution can be used for each feature.
- You can have straight decision boundaries (when classes have same variance), circular decision boundaries (different variance, same mean) and parabolic decision boundaries (different mean, different variance).

Cons:

- The probability estimates from Naive Bayes are usually bad.
- The independence assumption between the features is usually not given in real life.

```
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
clf = GaussianNB()
clf.fit(X, y)
```

More info: scikit-learn.org

8.3 Linear discriminant analysis (LDA)

Contrary to Naive Bayes, the features in LDA are not assumed to be independently distributed. As with Bayes rule a distribution for each class is calculated according to Bayes rule. $P(X = x|Y = y_i)$ is modeled as a multivariate Gaussian distribution. The Gaussians for each class are assumed to be the same. The log-posterior can be simplified to:

$$log(P(y = y_i|x) = -\frac{1}{2}(x - \mu_i)^t \Sigma^{-1}(x - \mu_i) + \log P(y = y_i)$$

μ_i is the mean of class i, $(x - \mu_i)^t \Sigma^{-1}(x - \mu_i)$ corresponds to the Mahalanobis distance. Thus, we assign the point to the class whose distribution it is closest to.
LDA can also be thought of projecting the data into a space with $k - 1$ dimensions (k being number of classes). More info: wikipedia.org. It can also be used as a dimensionality reduction method.

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
clf = LinearDiscriminantAnalysis()
clf.fit(X, y)
```

More info: scikit-learn.org

8.4 Support Vector Classifier (SVC)

SVCs use hyperplanes to separate data points according to their class label with a maximum margin (M) between the separating hyperplane ($x^T\beta + \beta_0 = 0$) and the points. If points cannot be perfectly separated by the decision boundary, a soft margin SVM is used with a slack variable ξ that punishes points in the margin or on the wrong side of the hyperplane. The optimization problem is given by:

$$\max_{\beta,\beta_0,\beta=1} M,$$
$$\text{subject to } y_i(x_i^T\beta+\beta_0)\geq 1-\xi_i, \quad \forall i,$$
$$\xi_i \geq 0, \quad \sum \xi_i \leq constant, \quad i=1,...,N,$$

where β are the coefficients and x are the N data points. The support vectors are the points that determine the orientation of the hyperplane (i.e. the closest points). The classification function is given by:

$$G(x) = \text{sign}[x^T\beta+\beta_0]$$

If you only calculate the inner part of the function you can get the distance of a point to your hyperplane (in SKlearn you need to divide by the norm vector w of your hyperplane to get the true distance). To get the probability of a point being in a class, you can use Platt's algorithm. SVMs are sensitive to the scaling of the features. Therefore, the data should be normalized before classification.

```
from sklearn import svm
# train the model
svc_model = svm.SVC()
svc_model.fit(train_df[feautre_cols], train_df[label_col])
# test the model
y_predict = svc_model.predict(test_df[feature_cols])
```

8.5 Decision Trees

A decision tree uses binary rules to recursively split the data into regions that contain only a single class.

Pros:

- Interpretable results, if trees are not too big.
- Cheap to train and predict.
- Can handle categorical and continuous data at the same time.
- Can be used for multi-output problems (e.g. color and shape of object).

Cons:

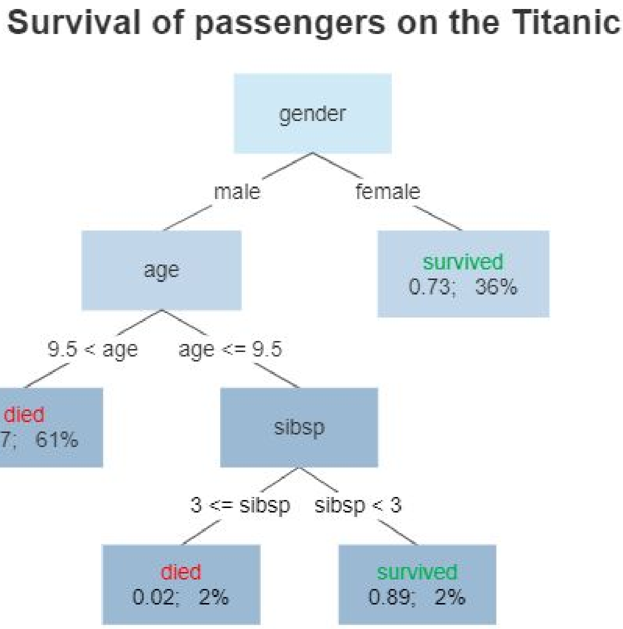


Figure 8.3: Decision trees work like flowcharts and have a root (the top node), branches (possible outcomes of a test), nodes (tests on one attribute), leafs (the class labels). This one shows the decision tree for the classifier predicting if a patient survives the sinking of the Titanic. Source: user Gilgoldm on wikipedia.org

- Overfitting risks
- Some concepts are hard to learn (X-OR relationships, Decision boundaries are not smooth)
- Unstable predictions: Small changes in data can lead to vastly different decision trees.

Tips:

- Doing PCA helps the tree find separating features.
- Visualize the produced tree.
- Setting a lower boundary on the split-sizes of the data, reduces the chance of overfitting.
- Balance the dataset or weight the samples according to class sizes to avoid constructing biased trees.

```
from sklearn.tree import DecisionTreeClassifier
clf = DecisionTreeClassifier(max_depth=10, min_samples_split=0.01, class_weight="balanced"
clf = clf.fit(X, Y)
```

More info: scikit-learn.org

8.5.1 Random forests

Random forests are a version of a bagging classifier employing decision trees. To reduce the variance, the separate trees can be assigned a limited number of features as well.

```
from sklearn.ensemble import RandomForestClassifier
clf = RandomForestClassifier(max_depth=10, max_features="sqrt", class_weight="balanced")
clf.fit(X, y)
```

More info: scikit-learn.org

8.5.2 Gradient boosted decision trees (GBDTs)

Gradient boosted decision tree models are a form of boosting employing decision trees.

```
import lightgbm as lgbm
clf = lgbm.LGBMClassifier(class_weight= "balanced")
clf.fit(X, y)
```

More info: lightgbm documentation, Parameter tuning,
Further Parameter tuning
Similar model: scikit-learn.org

9 Regression

9.0.1 Evaluation of regression models

9.0.1.1 Mean squared error

This measure shows the deviation of the predicted value $\hat{y}$ to the target value y. The squaring penalized large deviations and avoids respective cancellation of positive and negative errors.

$$MSE = 1/n \sum_i (y_i - \hat{y}_i)^2$$

```
from sklearn.metrics import mean_squared_error
mean_squared_error(y_true, y_pred)
```

More info: scikit-learn.org

9.0.1.2 R^2 score / coefficient of determination

This measure shows how much of the variance of the target/dependent variable y can be explained by the model/independent variable $\hat{y}$.

$$R^2 = 1 - \frac{\text{Unexplained Variance}}{\text{Total Variance}} = \frac{SS_{res}}{SS_{tot}} SS_{res} = \sum_i (y_i - \hat{y}_i)^2 SS_{tot} = \sum_i (y_i - \bar{y}_i)^2$$

where $\bar{y}$ is the mean of the target y.

The value commonly reaches from 0 (model always predicts the mean of y) to 1 (perfect fit of model to data). It can however be negative (e.g. wrong model, heavy overfitting, …). The *adjusted* R^2 compensates for the size of the model (more variables), favoring simpler models. More info: wikipedia.org

```
from sklearn.metrics import r2_score
r2 = r2_score(y_true, y_pred)
```

More info: scikit-learn.org

9.0.2 Linear Models

9.0.2.1 Ordinary Least Squares

Linear models are of the form:

$$y_i = x_i\beta + \epsilon_i = \beta_0 + \beta_1 x_{i(1)} + \beta_2 x_{i(2)} + \epsilon_i$$

- $x_{i(d)}$ is the dimension d of point i. x_i is called *regressor, independent, exogenous or explanatory variable*. The regressors can be non-linear functions of the data.
- y_i is the observed value for the *depependent variable* for point i.
- β_0 is called bias or intercept (not the same as bias of a machine learning model).
- $\beta_{0,1...n}$ are the *regression coefficients*, $\beta_{1,...n}$ are called *slope*. In linear models the regression coefficients need to be linear.
- ϵ is the *error term* or *noise*.

Predicted values are denoted as $\hat{\beta}, \hat{y}$. We try to minimize the ϵ-term using the least squared error method.

```
from sklearn.linear_model import LinearRegression
model = LinearRegression(fit_intercept=True)
model.fit(X, y)
```

More info: scikit-learn.org

Pros:

- Easy to interpret
- Fast to train and predict

Cons:

- Assumption of linear relation between dependent and independet variables. (-> possibly underfitting)
- Sensitive to outliers (-> possibly overfitting)

> Tips for interpreting linear models
>
> - When comparing the strength of different coefficients: Take the scale of the feature into consideration (e.g. don't compare "m/s" and "km/h").
> - Only when the features have been standardized / normalized, you can safely compare them.
> - Check for robustness of coefficients: Make cross validation and obsever their vari-

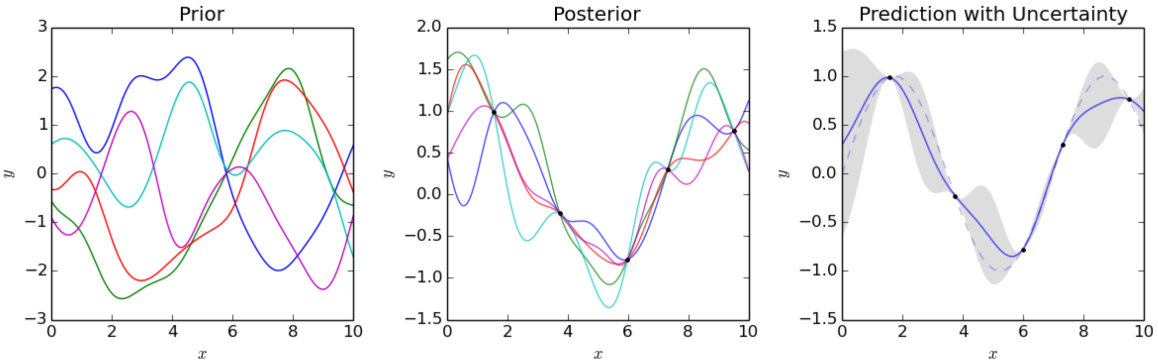
ability. High variability can be a sign of correlation with other features.

- Correlation does not mean causation. r emo::ji("point_up_2") r emo::ji("nerd")

9.0.3 Gaussian process regression

Gaussian process regression is based on Bayesian Probability: You generate many models and calculate the probability of your models given the samples. You make predictions based on the probabilities of your models.

You get non-linear functions to your data by using non-linear kernels: You assume that input data points that are similar, will have similar target values. The concept of similarity (e.g. same hour of the day) is encoded in the kernels that you use.


Prior

Posterior

Prediction with Uncertainty

Figure 9.1: Schema of the training process of Gaussian process regression. The left graph shows the prior samples of functions before. These functions are then conditioned on the data (graph in middle). The right graph shows the predictions with the credible intervals in gray. *Source: user Cdipaolo96 on wikimedia.org* .

Pros:

- The model reports the predictions with a certain probability.
- Hyperparameter tuning is built into the model.

Cons:

- Training scales with $O(n^3)$. (approximations are FITC and VFE)
- You need to design or choose a kernel.

```
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import DotProduct, WhiteKernel, RBF, ExpSineSquared
kernel = DotProduct() + WhiteKernel() + RBF() + ExpSineSquared() # The kernel hyperparamet
gpr = GaussianProcessRegressor(kernel=kernel)
gpr.fit(X, y)
gpr.predict(X, return_std=True)
```

More info: scikit-learn.org

9.0.4 Gradient boosted tree regression

Apart from classification, gradient boosted trees also allow for regression. It works like gradient boosted trees for classification: You iteratively add decision tree regressors that minimize the regression loss of the already fitted ensemble. A decision tree regressor is a decision tree that is trained on continuous data instead of discrete classification data, but its output is still discrete.

```
from sklearn.ensemble import GradientBoostingRegressor
gbr = GradientBoostingRegressor(n_estimators = 500, min_samples_split =5, max_depth = 4, m
gbr.fit(X, y)
```

More info: scikit-learn.org

9.1 Time Series Forecasting

For “normal” settings the order of the samples does not play a role (e.g. blood sugar level of one sample is independent of the others). In time series however, the samples need to be represented in an ordered vector or matrix (e.g. The temperature of Jan 2nd is not independent of the temperature on Jan 1st).

```
import pandas as pd
df = pd.read_csv("data.csv", header=0, index_col=0, names=["date", "sales"])
sales_series = df["sales"] # pandas series make working with time series easier
```

9.1.0.1 ARIMA(X) Model

univariate time series model with exogenous regressor.

9.1.0.2 VARIMA(X) Model

Multivariate time series model, where the variables can influence each other and the target can influence the variables and vice versa.

9.1.0.3 Prophet-Model

Part V

Unsupervised learning

10 Custering methods

Clustering methods are used to group data when no class labels are present. You thereby want to learn an intrinsic structure of the data.

10.1 Evaluation of clustering algorithms

10.1.1 Silhouette coefficient

The silhouette coefficient compares the average distance of a point and the points in its own cluster $d(x, \mu_C)$ to the average distance between the point and and the points of the second nearest Cluster $d(x, \mu_{C'})$.

$$s(x) = \frac{d(x, \mu_{C'}) - d(x, \mu_C)}{\max(d(x, \mu_C), d(x, \mu_{C'}))}$$

where C is the own cluster and C' is the second nearest cluster. If a point is clearly in its own cluster, $s(x)$ is close to 1. If a point is between two clusters, $s(x)$ is close to 0. If a point is closer to another cluster, $s(x)$ is negative.
By varying the number of clusters, one can find the number with the highest silhouette coefficients.

Pros:

- The score is high for dense and highly separated clusters.

Cons:

- The silhouette coefficient is mainly suitable for convex clusters, since it gives high values to this kind of clusters.

```python
from sklearn.metrics import silhouette_score
from sklearn.cluster import KMeans
clu = KMeans(n_clusters=4)
clu.fit(X)
labels = clu.labels_
silhouette_score(X, labels, metric='manhattan')
```

More info: scikit-learn.org

A faster alternative is the Davies-Bouldin score, where values closer to 0 indicate a better clustering.

10.1.2 Adjusted mutual information score

If you have labelled samples, you can use the mutual information score to test if the classes correspond to your clusters. The *adjusted* mutual information score adjusts for chance.

```
from sklearn.metrics import adjusted_mutual_info_score
adjusted_mutual_info_score(Y, clusters)
```

10.2 K-Means Clustering

Goal: Divide data into K clusters so that the variance within the clusters is minimized. The objective function:

$$V(D) = \sum_{i=1}^{k} \sum x_j \in C_i (x_j - \mu_i)^2,$$

where V is the variance, C_i is a cluster, μ_i is a cluster mean, x_j is a datapoint. The algorithm works as follows:

1. Assign the data to k initial clusters.
2. Calculate the mean of each cluster.
3. Assign the data points to the closest cluster mean.
4. If a point changed its cluster, repeat from step 2.

```
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=4)
kmeans.fit(X)
kmeans.predict([[5, 1]])
```

More info: scikit-learn.org

A faster alternative is mini batch K-Means.

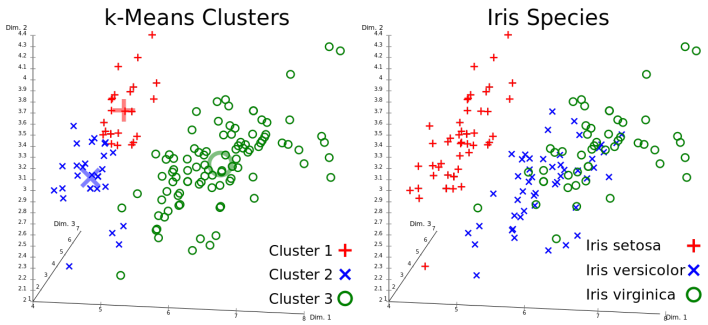


Figure 10.1: Data from the *Iris flower data set* clustered into 3 clusters using k-Means. On the right the data points have been assigned to their actual species. *Figure from user Chire on wikimedia.org.*

10.3 Graph-Based Clustering

You represent data set D as a graph $G = (V, E)$ and divide it up in connected sub-graphs that represent your clusters. Each edge e_{ij} (between nodes v_i and v_j) has a weight w_{ij} (which is commonly a similarity or distance measure).

10.3.1 Basic Graph-Based Clustering

The basic algorithm works like this:

1. Define a weight-threshold θ.
2. For all edges: if $w_{ij} > \theta$: remove e_{ij}.
3. If nodes are connected by a path (found via *depth first search*): Assign them to the same cluster.

10.3.2 DBScan

Density-Based Spatial Clustering of Applications with Noise is a more noise robust version of basic graph-based clustering. You create clusters based on dense and connected regions. It works like this:

1. A point is a *core point* if at least minPts are within a radius of ϵ of the point (including the point itself).
2. A point is *directly reachable* if it is not a *core point* but within ϵ from a *corepoint*.
3. All other points are not part of the cluster (and may not be part of any cluster).

! For points between clusters, the assignment to a cluster depends on the order of point assignments.

```
from sklearn.cluster import DBSCAN
dbscan = DBSCAN(eps=3, min_samples=4)
dbscan.fit(X)
```

More info: scikit-learn.org

There is a newer version of this algorithm (Hierarchical DBSCAN), that allows for clusters with varying density, more robustness in cluster assignment and makes tuning ϵ unnecessary.

10.3.3 Cut-Based Clustering

You introduce a **adjacency/similarity** matrix W (measures similarity between data points) and define the number of clusters k. You now try to minimize the weight of edges κ between the clusters C (equal to cutting edges between nodes that are least similar):

$$\begin{aligned} \min \frac{1}{2} \sum_{a=1}^{k} \sum_{b=1}^{k} \kappa(C_a, C_b) \\ \text{where } \kappa(C_a, C_b) = \sum_{v_i \in C_a, v_j \in C_b, a \neq b} W_{ij} \\ \text{and } \kappa(C_a, C_a) = 0 \end{aligned}$$

→ You only add up the similarities/edge-weights between your clusters (but not within your clusters).
For constructing the similarity matrix, different kernels can be used (commonly the linear kernel or the Gaussian kernel).

10.3.4 Spectral Clustering

Spectral clustering works by non-linearly mapping the matrix-representation of the graph onto a lower-dimensional space based on its spectrum (set of eigenvectors) and group the points there. The mapping preserves local distances, i.e. close points stay close to each other after the mapping. It employs three steps: Preprocessing, decomposition and grouping.

Preprocessing

We create a Laplacian matrix L (Laplacian operator in matrix form, measuring how strongly a vertex differs from nearby vertices (because the edges are similarity measures)):

$$L = D - W$$

$$D_{ij} = \begin{cases} \sum_{j=1}^{N} W_{ij} \\ 0 \text{ if } i \neq j \end{cases}$$

where D is the degree matrix (the (weighted) degree of each node is on the diagonal) and W is the adjacency/similarity matrix (measures similarity between data points).

Decomposition

You first normalize the Laplacian to avoid big impacts of highly connected vertices/nodes. More info on the calculation on wikipedia.org.

We make eigenvalue decomposition:

$$LU = \Lambda U \quad \rightarrow \quad L = U \Lambda U^{-1}$$

where U is the matrix of eigenvectors and Λ is the diagonal matrix of eigenvalues. You can now find a lower-dimensional embedding by choosing the k smallest non-zero eigenvalues. The final data is now represented as a matrix of k eigenvectors.

Grouping

You have multiple options:

- You can cut the graph by using the chosen eigenvectors and splitting at 0 or median value.
- You get the final cluster assignments by normalizing the now k-dimensional data and applying k-means clustering to it.

```
from sklearn.cluster import SpectralClustering
scl = SpectralClustering(n_clusters=4,
        affinity='rbf',
        assign_labels='cluster_qr', # assigns labels directly from Eig vecs,
        n-jobs = -1)
scl.fit(X)
```

More info: scikit-learn.org

10.4 Sparse Subspace Clustering (SSP)

The underlying assumption of SSP is that the different clusters reside in different subspaces of the data. Clusters are therefore perpendicular to each other and points in a cluster can only be reconstructed by combinations of points in the same cluster (→ self-expressiveness, the reconstruction vectors ought to be sparse). For each point you try to find other points that can be used to recreate that point - these then form the same cluster. Doing that for all points gives you a data matrix X and a matrix of reconstruction vectors V:

$$X = X * V \text{ s.t. } \operatorname{diag}(V) = 0.$$

You now try to minimize the V-matrix according to the L1-norm (giving you a sparse matrix). This matrix can then be used for e.g. spectral clustering.

More details in the original paper on SSC-OMP.

```
from cluster.selfrepresentation import SparseSubspaceClusteringOMP
ssc = SparseSubspaceClusteringOMP(n_clusters=3,affinity="symmetrize")
ssc.fit(X)
```

More info: github.com

10.5 Soft-assignment Clustering

Soft clustering assigns to each point the probabilities of belonging to each of the clusters instead of assigning it to only one cluster. This gives you a measure on how certain the algorithm is about the clustering of a point.

10.5.1 Gaussian Mixture Models

Gaussian mixture models try to find an ensemble of gaussian distributions that best describe your data. These distributions/components are used as your clusters. Your points belong to each cluster with a certain probability. To find these distributions, we use an *expectation maximization* algorithm:

1. Assume the centers of your Gaussians (e.g. by k-means) and calculate for each point the probability of being generated by each distribution ($p(x_i \in C_k | \ \phi_i, \mu_k, \sigma_k)$).
2. Change the parameters to maximize the likelihood of the data, given the cluster probabilities for all points.

The probability of a data point belonging to a cluster can be calculated via Bayes theorem.

More info on the theory: brilliant.org.

```
from sklearn.mixture import GaussianMixture
gm = GaussianMixture(n_components=4, covariance_type='full')
gm.fit(X)
gm.predict_proba(X)
```

More info: scikit-learn.org

10.5.2 Other models

Other models also have means to calculate cluster probabilities for points. For the HDBSCAN-algorithm see here.

10.5.3 Hierarchical Clustering

Instead of clustering a point to only one cluster, you assign it to a hierarchy.

Pros:

- Hierarchies of clusters reflects the data set and therefore the relationship between points better.

Cons:

- It is more difficult to make clear statements of cluster membership → define a limit for hierarchical depth

10.6 Artificial Neural Networks for Clustering

See chapter *Neural Networks* (5)

11 Mapping to lower dimensions

11.0.1 Manifold learning

11.0.1.1 Isomap

11.0.1.2 Local linear embedding

11.0.1.3 Multi dimensional scaling

11.0.2 Decomposition techniques

11.0.2.1 Singular value decomposition

Singular value decomposition is used to compress large matrices of your data into smaller ones, with much less data, but without loosing a lot of information. Please visit the mathematical explanation for the underlying mechanisms.

```
from sklearn.decomposition import TruncatedSVD
svd = TruncatedSVD(n_components=10)
svd.fit_transform(X_train)
svd.transform(X_test)
```

11.0.2.2 Principle Component analysis (PCA)

11.1 Outlier detection

11.1.1 Local outlier factor

11.1.2 Isolation forest

Part VI

Neural Networks

12 Neural Networks

13 Neural Networks

13.1 Fundamentals

On the most basic level, neural networks consist of many simple models (e.g. linear and logistic models) that are chained together in a directed network. The models sit on the neurons (nodes) of the network. The most important components of neurons are:

1. **Activation**: $a = Wx + b$ (W = weights and b = bias)
2. **Non-linearity**: $f(x, \theta) = \sigma(a)$ (e.g. a sigmoid function for logistic regression, giving you a probability output. θ is a threshold)

The neurons (nodes) in the first layer uses as its input the sample values and feeds its output into the activation function of the next nodes in the next layer, a.s.o. The later layers should thereby learn more and more complicated concepts or structures.

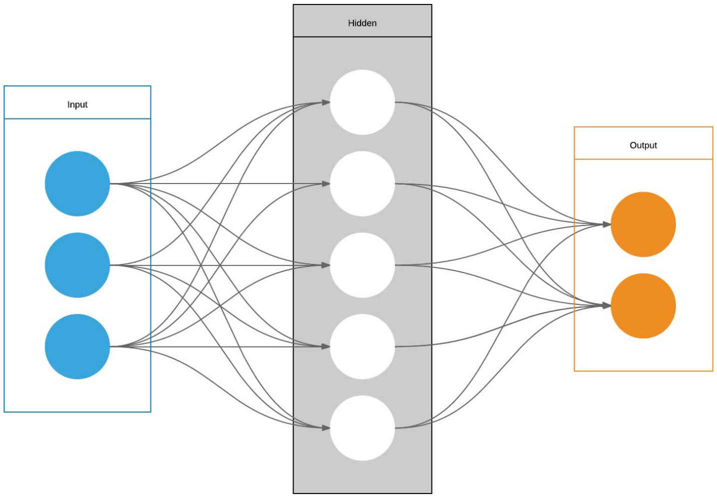


Figure 13.1: Model of an artificial neural network with one hidden layer. *Figure from user LearnDataSci on wikimedia.org.*

Explanation on the idea and mechanisms of neural networks: Stanford Computer Vision Class

13.1.1 Non-Linearities

Different non-linear functions can be used to generate the output of the neurons.

13.1.1.1 Sigmoid/Logistic Functions

This activation function is often used in the last layer of NNs for classification (since it scales the output between 0 and 1).

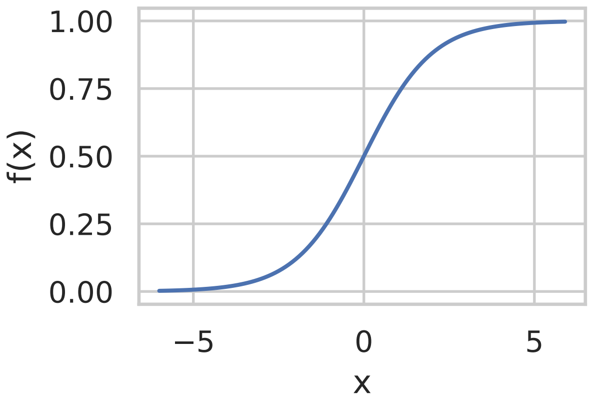
$$f(x) = \frac{1}{1+e^{-x}}$$

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import math

x = np.arange(-6, 6, 0.1)
f = 1 / (1+math.e**(-x))

sns.set(rc={'figure.figsize':(3,2)}, style="whitegrid")
sns.lineplot(x=x, y=f, )
plt.xlabel('x')
plt.ylabel('f(x)')
```

```
Text(0, 0.5, 'f(x)')
```



Pros:

- Scales output between 0 and 1 (good for output layer in classification tasks)
- Outputs are bound between 0 and 1 → No explosion of activations

Cons:

- No saturation / dying neuron / vanishing gradient: When f(x) = 0 or 1, the gradient of f(x) is 0. This blocks back-propagation (see here)
- Output not centered around 0: All weight-updates during backpropagation are either positive or negative, leading to zig-zag SGD instead of direct descent to optimum (see here or here)
- computationally more expensive than ReLu

13.1.1.2 Tanh Functions

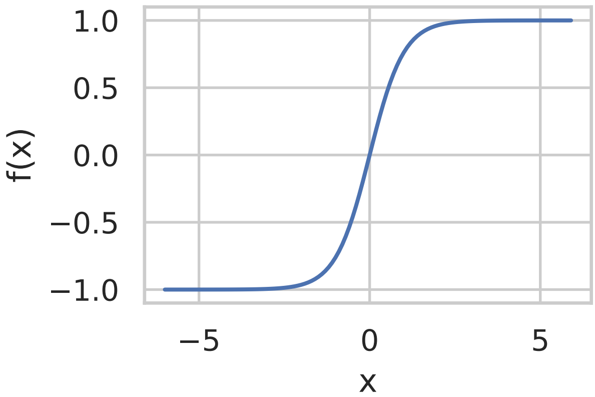
$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import math

x = np.arange(-6, 6, 0.1)
f = (math.e**x - math.e**(-x) ) / (math.e**(x)+math.e**(-x))

sns.set(rc={'figure.figsize':(3,2)}, style="whitegrid")
sns.lineplot(x=x, y=f, )
plt.xlabel('x')
plt.ylabel('f(x)')
```

```
Text(0, 0.5, 'f(x)')
```



Pros:

- Centered around zero

Cons:

- saturation / dying neuron / vanishing gradient problem
- computationally more expensive than ReLu

13.1.1.3 Rectifiers/ReLU

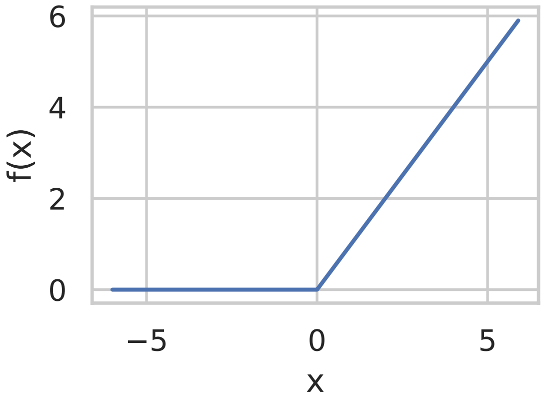
$$f(x) = \max(0, x)$$

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import math

x = np.arange(-6, 6, 0.1)
f = [max(0,x_i) for x_i in x]
```

```
sns.set(rc={'figure.figsize':(3,2)}, style="whitegrid")
sns.lineplot(x=x, y=f, )
plt.xlabel('x')
plt.ylabel('f(x)')
```

```
Text(0, 0.5, 'f(x)')
```



Pros:

- Computationally cheap
- No saturation for positive values

Cons:

- Not zero centered
- Saturation for negative values

13.1.1.4 Leaky ReLU

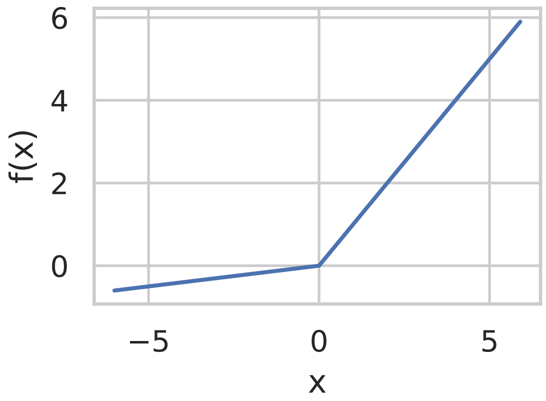
$$f(x) = \begin{cases} \alpha x \text{ if } x < 0 \\ x \text{ if } x \geq 0 \end{cases}$$

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import math
```

```
a = 0.1
x = np.arange(-6, 6, 0.1)
f = [(a*x_i if x_i < 0 else x_i) for x_i in x]

sns.set(rc={'figure.figsize':(3,2)}, style="whitegrid")
sns.lineplot(x=x, y=f, )
plt.xlabel('x')
plt.ylabel('f(x)')
```

```
Text(0, 0.5, 'f(x)')
```



Pros:

- No saturation problem
- fast to compute
- more zero-centered than e.g. sigmoid-activation

13.1.2 Terminology

- **Input layer/visible layer:** Input variables
- **Hidden layer:** Layers of nodes between input and output layer
- **Output layer:** Layer of nodes that produce output variables
- **Size:** Number of nodes in the network

- **Width:** Number of nodes in a layer
- **Depth:** Number of layers
- **Capacity:** The type of functions that can be learned by the network
- **Architecture:** The arrangement of layers and nodes in the network

13.1.3 Feedforward Neural Network / Multi-Layer Perceptron

This is the simplest type of proper neural networks. Each neuron of a layer is connected to each neuron of the next layer and there are no cycles. The outputs of the previous layer corresponds to the x in the activation function. Each output (x_i) of the previous layer gets it's own weight (w_i) in each node and a bias (b) is added to each node. Neurons with a very high output are "active" neurons, those with negative outputs are "inactive". The result is mapped to the probability range by (commonly) a sigmoid function. The output is then again given to the next layer.
If your input layer has 6400 features (80*80 image), a network with 2 hidden layers of 16 nodes will have $6400 * 16 + 16 * 16 + 16 * 10 + 16 + 16 + 10 = 102'858$ parameters. This is a very high number of degrees of freedom and requires a lot of training samples.

13.1.4 PyTorch

```
from torch import nn

    class CustomNet(nn.Module):
        def __init__(self):
            super(CustomNet, self).__init__()
            self.lin_layer_1 = nn.Linear(in_features=10, out_features=10)
            self.relu = nn.ReLU()
            self.lin_layer_2 = nn.Linear(in_features=10, out_features=10)

        def forward(self, x):
            x = self.lin_layer_1(x)
            x = self.relu
            x = self.lin_layer_2(x)
            return x

        def num_flat_features(self, x):
            size = x.size()[1:] # Use all but the batch dimension
            num = 1
            for i in size:
                num *= i
            return num

    new_net = CustomNet()
```

13.1.5 Keras

Example of a small Keras model for text-classification.

```
from keras.models import Sequential
from keras import layers

embedding_dim = 20
sequence_length = 50
vocab_size = 5000 # length of word index / corpus

# Specify model:
model = Sequential()
model.add(layers.Embedding(input_dim=vocab_size,
                           output_dim=embedding_dim,
                           input_length=sequence_length))
model.add(layers.SpatialDropout1D(0.1)) # Against overfitting
model.add(layers.GlobalMaxPool1D())
model.add(layers.Dense(10, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])
model.summary()

# Train model:
history = model.fit(train_texts_padded,
                    y_train,
                    epochs=5,
                    verbose=True,
                    validation_data=(test_texts_padded, y_test),
                    batch_size=100)
loss, accuracy = model.evaluate(train_texts_padded, y_train)
print("Accuracy training: {:.3f}".format(accuracy))
loss, accuracy = model.evaluate(test_texts_padded, y_test)
```

```
print("Accuracy test:  {:.3f}".format(accuracy))
```

13.1.6 Backpropagation

This is the method by which neural networks learn the optimal weights and biases of the nodes. The components are a cost function and a gradient descent method.
The cost function analyses the difference between the designated activation in the output layer (according to the label of the data) and the actual activation of that layer. Commonly a residual sum of squares is used.
You get the direction of the next best parameter-combination by using a *stochastic gradient descent* algorithm using the gradient for your cost function:

1. We use a "mini-batch" of samples for each round/step of the gradient descent.
2. We calculate squared residual of each feature of the output layer for each sample.
3. From that we calculate what the bias or weights from the output layer and the activation from the last hidden layer must have been to get this result. We average that out for all images in our mini-batch.
4. From that we calculate the weights, biases and activations of the upstream layers → we *backpropagate*.

13.1.7 Initialization

The weights of the nodes are commonly initialized randomly with a certain distribution. The biases are commonly initialized as zero, thus 0-centering of the input data is recommended.

13.2 Types of NNs

13.2.1 Convolutional Neural Networks

13.2.2 Encoder-Decoder Models

13.2.2.1 Autoencoder models

Contrary to the other architectures, autoencoders are used for unsupervised learning. Their goal is to compress and decompress data to learn the most important structures of the data.

The layers therefore become smaller for the encoding step and the later layers get bigger again, up to the original representation of the data. The optimization problem is now:

$$\min_{W,b} \frac{1}{N} * \sum_{i=1}^{N} ||x_i - \hat{x}_i||^2$$

with x_i being the original datapoint and $\hat{x}_i$ the reconstructed datapoint.

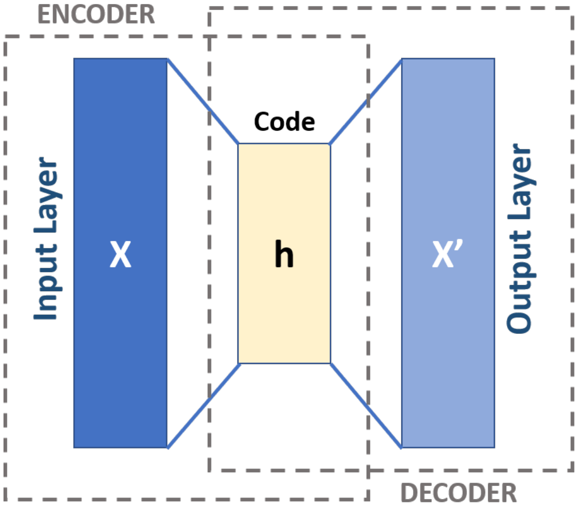


Figure 13.2: Model of an autoencoder. The encoder layers compress the data towards the code layer, the decoder layers decompress the data again. *Figure from Michela Massi on wikimedia.org.*

13.2.2.2 Autoencoders for clustering

You can look at layers of a NN as ways to represent data in different form of complexity and compactness. The code layers of autoencoders are a very compact way to represent the data. You can then use the compressed representation of the code layer and do clustering on that data. Because the code layer is however not optimized for that task. Song et al. combined the cost function of the **autoencoder and k-means clustering**:

$$\min_{W,b} \frac{1}{N} * \sum_{i=1}^{N} ||x_i - \hat{x}_i||^2 - \lambda \sum_{i=1}^{N} ||f(x_i) - c_i||^2$$

with $f(x_i)$ being the non-linearity of the code layer and λ is a weight constant.
XXXX adapted spectral clustering (section 3.3) using autoencoders by replacing the (linear) eigen-decomposition with the (non-linear) decomposition by the encoder. As in spectral clustering the Laplacian matrix is used as the the input to the decomposition step (encoder) and

the compressed representation (code-layer) is fed into k-means clustering.
Deep subspace clustering by Pan et al. employs autoencoders combined with sparse subspace clustering. They used autoencoders and optimized for a compact representation of the code layer:

$$\min_{W,b} \frac{1}{N} * \sum_{i=1}^{N} ||x_i - \hat{x}_i||^2 - \lambda ||V||_1$$
$$\text{s.t.} F(X) = F(X) * V \text{ and } \operatorname{diag}(V) = 0$$

with V being the sparse representation of the code layer $(F(X))$.

13.2.3 Generative adversarial networks

13.2.4 Recurrent neural networks (RNN)

Compared to the fully connected or convolutional neural networks, RNNs can work on variable length inputs without padding the sequences.

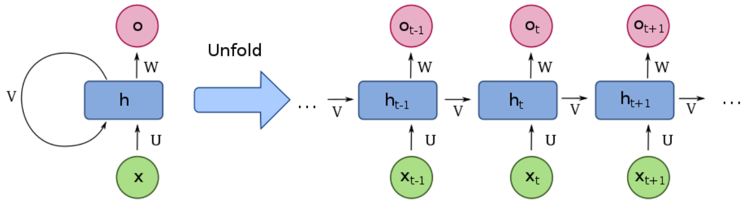


Figure 13.3: A unit of an RNN. The left side shows a static view of a unit h. The right side shows how the activations of past inputs (x_{t-1}) influence the output (o_t) of the current input (x_t). U, V and W are weights (Beware: They stay the same for different inputs). *Figure from fdeloche on wikimedia.org*.

Problem: The influence of previous inputs vanishes (or potentially explodes) with the sequence length (unfolding). This makes the network hard to train. This is mitigated by LSTMs (below).

more explanation: youtube: StatQuest

13.2.4.1 Long short-term memory (LSTM) networks

Instead of just using the influence of previous inputs (“short-term memory”), LSTMs incorporate influence of more distance inputs (“long-term memory”). → There are two paths to influence the current state.

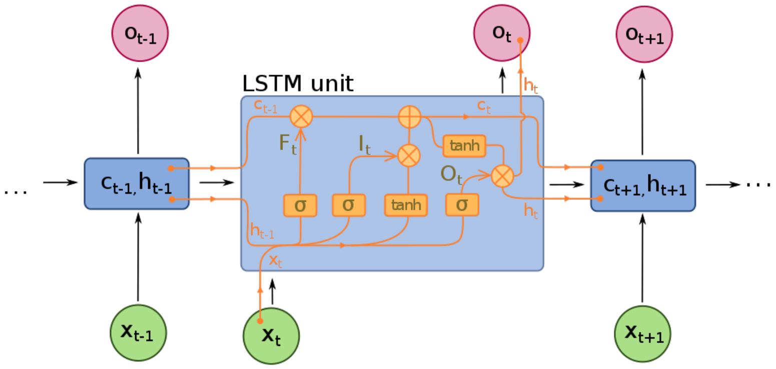


Figure 13.4: A unfolded view of a unit of an LSTM. c_t (top horizontal path) is the *cell state* and represents the long-term memory. h_t (bottom horizontal line) is the *hidden state* and represents the short-term memory. The weights are not shown in this diagram. F_t is the *forget gate* and determines the percentage of the long-term memory that is remembered based on the short-term memory and input. I_t is the *input gate* and determines how/whether to update/create the long-term memory using the input and short-term memory. O_t is the *output gate* and determines the short-term memory to be passed on to the next time-step. Thus the output of the cell is the modified long- and short-term memory. *Figure from fdeloche on wikimedia.org.*

more explanation: youtube: StatQuest

13.2.5 Transformer models

Transformers are encoder-decoder models with the following setup:

- **Encoder:** Containts one attention unit, the *multi-head attention*. It learns the relationships between elements in the input sequence.
- **Decoder:** Contains two attention units:
  - The *masked multi-head attention*: It learns relationship of the current element and previous words in the output sequence.
  - The *multi-head attention*: It learns the relationship of the (current and previous) elements in the output sequence and the learned representation of the input from the encoder.

Like LSTMs, transformers use attention mechanisms to learn relationships between input elements. There are key differences, however:

- The sequences are passed in simultaneously (not sequentially as in RNNs). Many computations can be done in parallel.
- *Positional encoding*: Since the elements are fed-in in parallel, the position of the elements within the sequence is encoded.
- Masking: Since elements are fed-in in parallel, the elements that the model shall predict, are masked (overwritten with zeros) from the output-sequence during training.
- Self-attention: The significance of an element (e.g. word) is learned by specific elements around it. Contrary to RNNs, the dependencies don't grow/shrink linearly with t, but are independent of the distance.

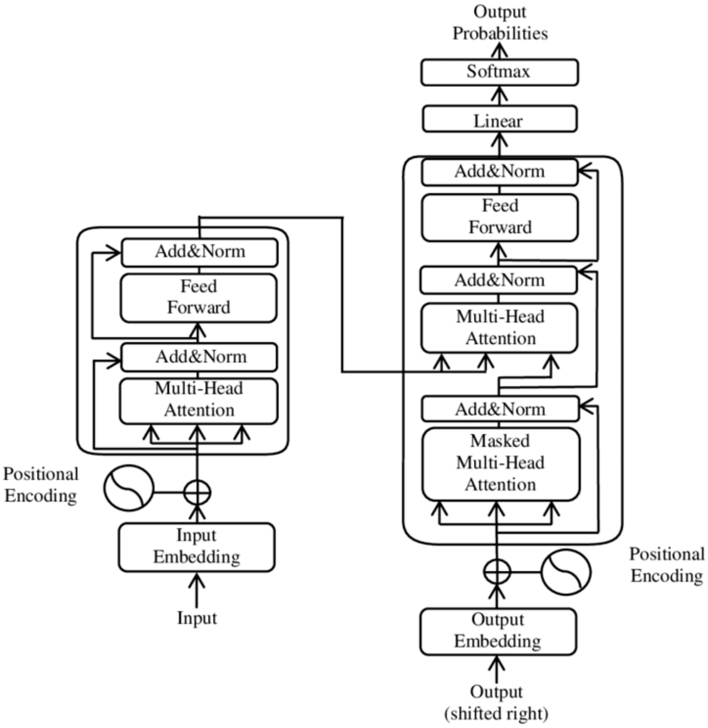


Figure 13.5: Architecture of transformer models. The encoder is on the left. The decoder is on the right. The encoder-outputs are passed to it in sequence. Output-sequence elements that come after the current input element are masked. The decoder generates the next element of the output. There are commonly multiple encoder and decoder units in sequence and multiple attention units in parallel. *Figure from Yuening Jia on wikimedia.org.*

Detailed explanation: Youtube - CodeEmporium

13.2.5.1 Large Language Models (LLMs)

LLMs are transformer models, that have been trained on a huge amount of unlabeled data.

Pros:

- They are very performant
- They are easily adaptable for other use cases.

Cons:

- The training data of the models is often unknown or insufficiently sanitized (containing false information, hate speech, outdated info, etc.)
- The models are so huge, that it takes a lot of compute power just to use it for predictions.

To deal with these problems, You can tune late layers in pre-trained LLMs to increase their accuracy for your field of application without having to train a huge model.

13.3 Learnig methods

There are specific methods for learning in neural networks.

13.3.1 Transfer learning

You train a model on one (usually large) dataset and adapt it for a related learning task. The feature space and distribution of the input and target data of the new task can be different from the old task.

13.3.1.1 Domain adaptation

In domain adaptation the feature space is similar to the original dataset.

Part VII

Other

14 ML Project Management

14.1 Basis for Machine Learning in Companies

14.2 How to automate business processes with machine learning

14.2.1 The stages from ManuaL to ML

A good business process entails a feedback loop from the client (receiver of the output) to further optimize the process:

Input → Process → *Output* → *Feedback* → Optimization → new Process → ...

Business processes evolve along these phases, whose steps should not be skipped:

1. **Individual employee** works on tasks, commonly informal rules and heuristics are used.
2. **Team** works on the tasks. The process is formalized and standardized to ensure quality and effective collaboration. Don't stay here for too long, since it is not scaleable.
3. **Digitization** is used to automate (parts of) the process. This step should be done before ML, since you need the data and architecture for your ML part anyway. Here you are more flexible and can fail and adapt quicker. Don't stay here to long, since you cannot assess quaility of your process well.
4. **Analytics** are used to measure the performance of the process and if its optimizations are successful. Don't skip this since you need these indicators for your ML optimizationa and monitoring anyway. Don't stay here too long to miss out on automation and scalability.
5. **Machine Learning** is used to automate and optimize analysis, insights and decision making on the data. You still need some people from step 2 to analyse outcomes, failures and react to it (monitoring).

More info: Google Cloud: How Google Does Machine Learning

14.2.2 Phases of the ML project

1. Framing the problem
2. Data collection & management
3. Building infrastructure (data pipeline, databases, training & deployment pipelines should at least work the same way as the designated product unless its a quick'n'dirty PoC)
4. Data ingestion, transformation & feature engineering
5. Model selection, training, testing & evaluation
6. Deployment & integration
7. Monitoring

More info: Google Cloud: How Google Does Machine Learning

14.2.3 How to frame ML problems

- **The ML-view:**
  - What is being predicted?
  - What data do we need as target and input?
- **Software development view:**
  - What info do we need from users to make a decision? (This defines the API)
  - Who will use the service? How many people will that be?
  - How is the process conducted today?
- **Data view:**
  - What data needs to be collected? From where?
  - How do we need to transform the data to analyze it & make decisions on it? (Feature engineering)
  - How do we react to the outputs of the algorithm? (e.g. kick off automatic process, inform stakeholders...)

More info: Google Cloud: How Google Does Machine Learning

14.3 Common pitfalls in machine learning

- Underestimate the effort for data collection, engineering, transformation & ingestion
- Focus too much effort in optimizing the machine learning algorithm instead of getting more data
- Having too few samples and diversity (independent attributes) or outdated/unrepresentative data
- Data is not properly maintained or not available as needed for the project
- Assume that no oversight will be necessary in data-, target-selection, feature engineering and monitoring from subject matter experts
- Optimize for a skewed indicator that causes unwanted side-effects in model decisions
- Building models from scratch, where pre-trained / off-the-shelf models do the job (especially text, image, audio and video tasks employing neural networks)
- Having no process for monitoring and retraining

More info: Google Cloud: How Google Does Machine Learning

15 Checklists

15.1 Tips for machine learning projects

15.1.1 General advice

General advice for machine learning from Pedro Domingos:

- Let your knowledge about the problem help you choose the candidate algorithms. E.g. You know the rules on which comparing samples makes most sense → Choose instance based learners. If you know that statistical dependencies are relevant → choose Graph based models.
- Don't underestimate the impact of feature engineering: Many domain specific features can boost the accuracy.
- Get more samples and candidate features (instead of focussing on the algorithm)
- Don't confuse correlation with causation. Just because your model can predict something, it does not mean that the features cause the target and you thus cannot easily deduct a clear action from it.

15.1.2 Common mistakes

Be aware: This list will never capture everything that can go wrong. ;-)

- **Data Leakage:** Information from Samples in your test data have leaked into your training data.
  - You have not deleted duplicates beforehand
  - You falsely assumed that your samples where drawn independently and have sampled the training set randomly. (E.g. multiple samples from the same patient, time series data)
  - You have the class label encoded in the training features in a way that you will not find in "Nature".
  - You just used the wrong training / test set while programming.
  - You did feature engineering like finding n-grams or Max, Min of data using your test-set data.

  - **Remedy:** Careful preliminary data analysis, deduplication,

- **Using the wrong quality measures on unbalanced data:** E.g. Accuracy on unbalanced data is not a reasonable quality measure.

- **Inconsistent preprocessing:** If you preprocess your training data in a certain way, you have to do the same with the test- and prediction-data.

  - **Remedy**: Use one preprocessing pipeline that you can use for training, testing and prediction.

- **Curse of dimensionality:**

  - You use too many features for the amount of samples that you have
  - Your distance measure is not suitable for high-dimensional space (e.g. Hamming distance, Euclidean distance)
  - **Remedy:** Use lower-dimensional mapping, feature selection.

- **Overfitting:**

  - You use a too complex algorithm (too many degrees of freedom) for the amount of data you have
  - You have too many features
  - **Remedy:** Get more samples, reduce the dimensionanlity, feature selection, regularization, bagging, boosting, stacking.

- **Bad Data:**

  - Your data is not representative of what you would find in the "real world". (skewed population, too old data, only of specific sensors, locations...)
  - Your have many missing values among your features.
  - The data that you have is only remotely linked to the target that you want to predict.
  - There are erroneous entries in your data.
  - **Remedy:** Clean data at source, impute data, clean data during preprocessing, get more representative data, limit scope of application.

15.2 Data Import Checklist

- Was the data-set correctly imported?

  - No column index as first row values.

  - No trailing comment as last row values.

- Are the sample values what you expect?
- Are columns in correct and efficient data type?
  - Has there been a shift of data between columns / rows?
  - Are there strings in a column for numerical values?
- Is the range what you expect?
  - Are there heavy outliers?
  - Is the data biased towards certain values?
- How many empty values are there?

15.3 Feature selection & engineering checklist

Selected advice from paper from Guyon and Elisseeff:

- If you have domain knowledge: Use it.
- Are your features commensurate (same proportion): Normalize them.
- Do you suspect interdependent features: Construct conjunctive features or products of features.

Other advice:

- Features that are useless on their own, can be useful in combination with other features.
- Using multiple redundant variables can be useful to reduce noise.
- There are also models (e.g. lasso regression, decision trees) that have feature selection built into the model (i.e. by only allowing for a certain number of features to be used or penalizing the use of additional features).