Moritz' Machine Learning Summary

Moritz Gück

2023-02-12

Contents

Summary

This is a reference for machine learning approaches and methods. The topics range from basic statistics to complex machine learning models and explanation methods. For each method and model, I have provided the underlying formulas (objective functions, prediction functions, etc.) as well as code snippets from the respective python libraries. I made this reference to quickly look up things I have studied already. I published it to give data scientists a catalog to find methods for their problem, refresh their knowledge and give references for further reading. If you find errors or unclear explanations in this text, please file an issue under: `github.com/MoritzGuck/All_of_ML-under_construction`

Chapter 1

Probability Theory & Linear Algebra

1.1 Probability Theory

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

1.1.1 Probability Basics

1.1.1.1 Probability interpretations

- **Frequentist:** 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 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.

1.1.1.2 Probability Space

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

1.1.1.3 Axioms of Probability

The mapping P must fulfill the axioms of probability:

1. $P(a)\ \ 0$
2. $P() = 1$
3. $a, b\ \ S$ and $a\ \ b = \{\}\ \ P(a\ \ b) = P(a) + P(b)$

a, b are events.

1.1.1.4 Random Variable (RV)

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

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

1.1.1.5 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**:
$A = true, B = false\ \ a\ \ \check{n}b$. Propositions can be continuous, discrete or boolean.

1.1.2 Probability distributions

Probability distributions assign probabilities to to all possible points in (e.g. $P(Weather) = 0.3, 0.4, 0.2, 0.1$, 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 Œ 4 matrix.)

1.1.2.1 Cumulative Distribution Function (CDF)

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

1.1.2.2 Probability Density Function (PDF)

For continuous functions the PDF is defined by

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

The probability of x being in a finite interval is

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

A PDF is shown in the following figure.

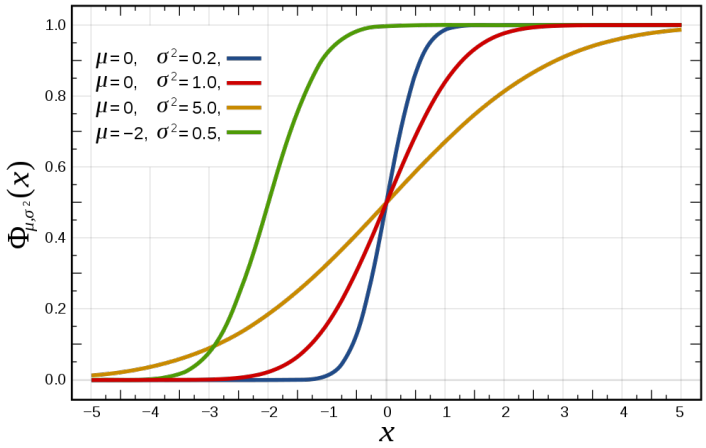

Figure 1.1: Cumulative distribution function of a normal distribution for different mean () and variance (). *Source: user Inductiveload on wikimedia.org.*

1.1.2.3 Properties of Distributions

- The **expected value** (E) or **mean** () is given by $E[X] = {}_{xX}\, x\ p(x)$ for discrete RVs and $E[X] = {}_{X}\, x\ p(x)dx$ for continuous RVs.
- The **variance** measures the spread of a distribution: $var[X] = {}^{2} = E[(X\ )^2] = E[X]^2\ {}^{2}$.
- The **standard deviation** is given by: $var[X] = $.
- The **mode** is the value with the highest probability (or the point in the PDF with the highest value):
- 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)

1.1.2.4 Dirac delta function

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

$$(x) = \{^{,} \begin{matrix} \text{if } x = 0 \\ 0, \quad \text{if } x\ \ 0 \end{matrix} \qquad \text{and} \quad (x)dx = 1$$

(Needed for distributions further on)

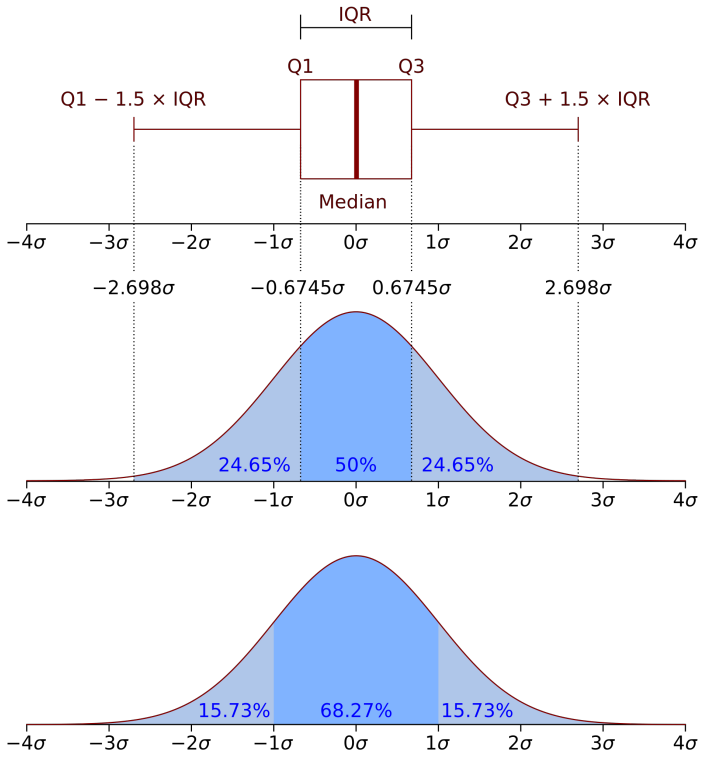

Figure 1.2: Probability density function of a normal distribution with variance (). In red a range from a Box-plot is shown with quartiles (Q1, Q3) and interquartile range (IQR). For the cutoffs (borders to darker blue regions) the IQR (on top) and are chosen. Another common cutoff is the confidence interval with light blue regions having a probability mass of 2 /2. *Source: user Jhguch on wikimedia.org.*

1.1.2.5 Uniform distribution

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

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

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

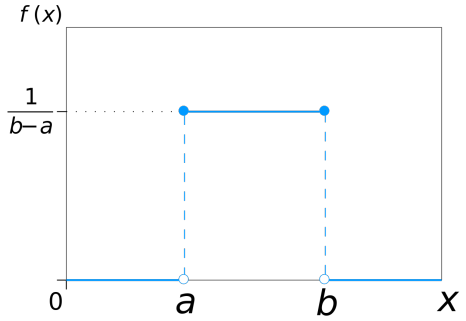

Figure 1.3: Uniform distribution. *Source: user IkamusumeFan on wikimedia.org.*

1.1.2.6 Discrete distributions

Used for random variables that have discrete states.

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

$$X\ \ \text{Bin}(n,\ ), \quad \text{Bin}(k|n,\ ) = \binom{n}{k}^{k}(1\ \ )^{n\ k}, \quad \binom{n}{k} = \frac{n!}{k!(n\ \ k)!},$$

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

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

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

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

$$\text{Mu}(x|n,\ ) = \binom{n}{x_1, ..., x_K}\ {}_{i=1}^{K}\ {}^{x_j}_{j} = \frac{n!}{x_1!, ..., x_k!}\ {}_{i=1}^{K}\ {}^{x_j}_{j},$$

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*.

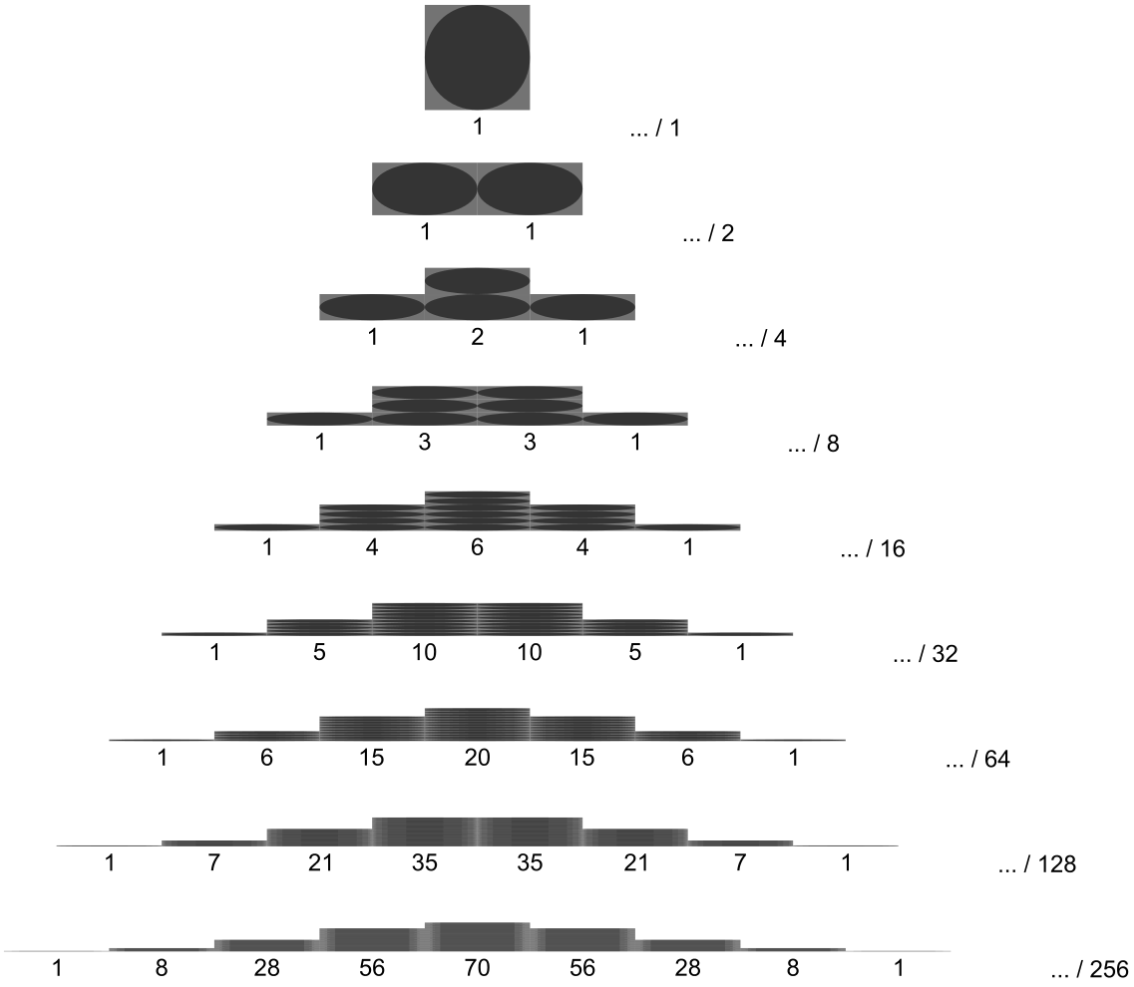

Figure 1.4: Binomial distribution of balls in Pascals triangles with different numbers of layers (The top one has 0 layers). Example: For a triangle with $n = 6$ layers, the probability that a ball lands in the middle box $k = 3$ is $20/64$. *Source: user Watchduck on wikimedia.org*

1.1.2.6.4 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).

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

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

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

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

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

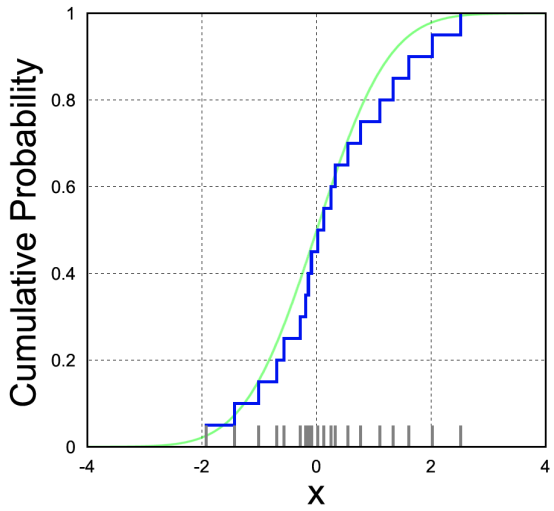

Figure 1.5: Cumulative empirical distribution function (blue line) for samples drawn from a standard normal distribution (green line). The values of the drawn samples is shown as grey lines at the bottom. Source: user nagualdesign on wikimedia.org.

1.1.2.7 Continuous distributions

Used for random variables that have continuous states.

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

$$p(x|^2) = \frac{1}{2^2}\exp[\frac{(x\ )^2}{2^2}],$$

where is the mean and ^{2} is the variance. The CDF is given by:

$$(x) = \frac{1}{2}\ ^x\ e^{\frac{t^2}{2}}dt$$

1.1.2.7.2 Multivariate normal/Gaussian distribution For T datapoints with k dimensions (features). The pdf is:

$$p(x|,) = \frac{1}{(2)^k||}\exp[\frac{1}{2}(x\ )^1(x\ )],$$

where x now has multiple dimension $(x_1, x_2, ..., x_k)$ and is the $k\,\mathbb{E}k$ covariance matrix: $= \mathrm{E}[(X\ )(X\ )]$. The covariance between features is: $\mathrm{Cov}[X_i, X_j] = \mathrm{E}[(X_i\ _i)(X_j\ _j)]$

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

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

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

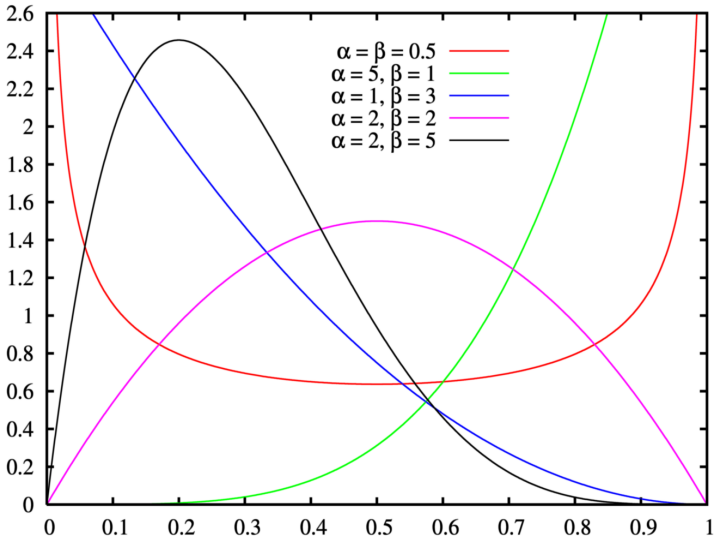

Figure 1.6: Probability density function of a beta-distribution with different parameter values. *Source: user MarkSweep on wikimedia.org.*

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

$$\mathrm{Dir}(x|)\ \ \frac{1}{B()}\ _{i=1}^{K}\ x_i^{i1}, \quad \ _{i=1}^{K}\ x_i = 1, \quad x_i\ \ 0\ i$$

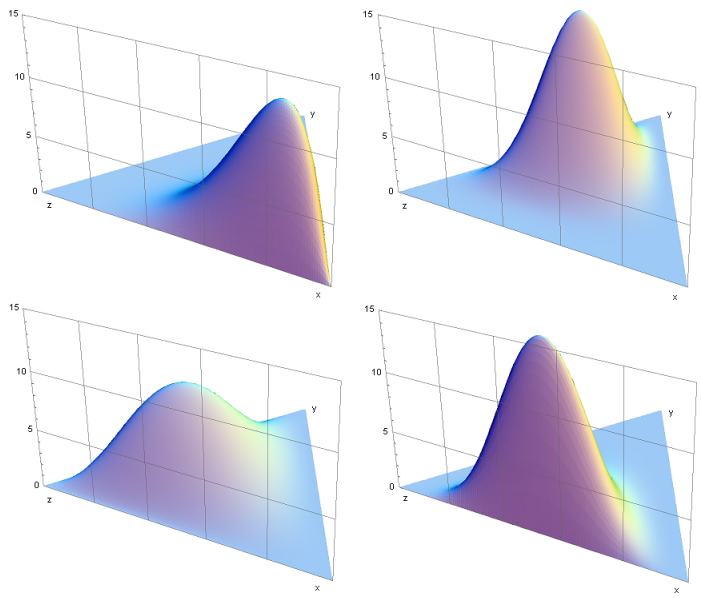

Figure 1.7: Probability density function of a Dirichlet-distribution on a 2-simplex (triangle) with different parameter values. Clockwise from top left: = (6,2,2), (3,7,5), (6,2,6), (2,3,4). *Source: user ThG on wikimedia.org.*

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

1.1.3 Central limit theorem

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

1.1.4 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)).

1.1.4.1 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" $p(C|S) = 0.2$, 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:

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

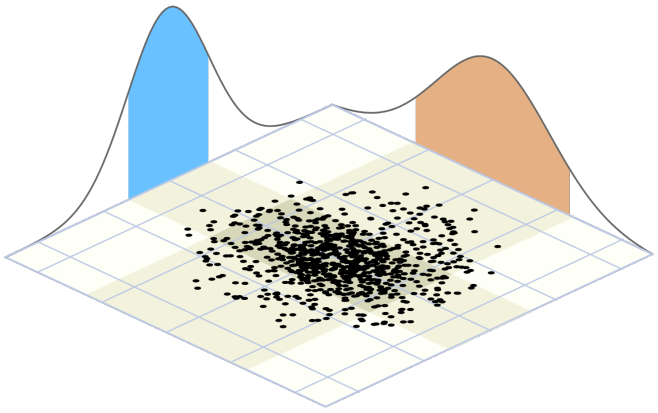

Figure 1.8: Data following a 2D-Gaussian distribution. Marginal distributions are shown on the sides in blue and orange. *Source: user Auguel on wikimedia.org.*

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

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

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

$$P(Y|E=e) = P(Y, E=e) = \sum_h P(Y, E=e, H=h)$$

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 $p(A|B) \ p(B|A)$
! 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.

1.1.4.2 Independence

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

1.1.4.3 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("vocabulary"|"height","age") = P("vocabulary"|"age").

1.1.4.4 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:

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

1.1.4.4.1 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 ($P(A,B)$)

1.1.4.4.2 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.

1.1.5 Further Concepts

1.1.5.1 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 from your target X should be zero.

$$\lim_n P(|X_n\ \ X| >) = 0$$

1.1.5.2 Bernoulli's Theorem / Weak Law of Large Numbers

$$\lim_n P(|\frac{\sum_{i=1}^{n} X_i}{n}\ \ | >) = 0,$$

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 $|\frac{\sum_{i=1}^{n} X_i}{n} | <$ for any > 0.

1.2 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.

1.2.1 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 $[0, 0]$.

1.2.1.1 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*.

$$I = \begin{matrix} 1 & 0 & 0 \\ [0 & 1 & 0] \\ 0 & 0 & 1 \end{matrix}$$

1.2.1.2 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.

1.2.1.3 Determinants

Determinants can be used to measure how much a linear combination 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.

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 Œ 2 matrix can be viewed as a transformation mapping from 2-D to 3-D space.

The *dot product* of two vectors is calculated like a linear transformation between a 1 Œ 2 matrix and a 2 Œ 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.

1.2.1.4 System of equations

Linear algebra can help you solve systems of equations.

$$\begin{array}{l} 1x + 2y + 3z = 4 \\ 4x + 5y + 6z = 7 \\ 8x + 9y + 0z = 1 \end{array} \qquad \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 8 & 9 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 4 \\ 7 \\ 1 \end{bmatrix} \qquad Ax = v$$

You can imagine this as as searching a vector x that will land on v after the transformation A.

To find 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:

$$A^1 Ax = A^1 v \qquad x = A^1 v$$

1.2.1.5 Eigenvalues and Eigenvectors

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

$$Av = v$$

Transforming to a scaled identity matrix I and factoring out v, we get:

$$(A \ I)v = 0$$

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

An *eigenbasis* I 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*).

1.2.1.6 Eigenvalue decomposition

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

$$AU = U \qquad A = UU^1$$

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

1.2.1.7 Singular value decomposition

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

$$A = UV^T$$

where U is an orthogonal m Œ r matrix, is a diagonal r Œ r matrix and V^T is an orthogonal r Œ n matrix. U contains the *left singular vectors*, V the *right singular vectors* and the $SingularValues$.
This decomposition technique can be used to approximate the original matrix A with only the largest singular values. Thereby you can save computation time in matrix operations without loosing a lot of information.
For applications, please see SVD for lower dimensional mapping.

Chapter 2

Data Basics

2.1 Similarity and Distance Measures

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

2.1.1 Metrics

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

- Positivity: $d(x_1, x_2)$ 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: $d(x_1, x_3)$ $d(x_1, x_2) + d(x_2, x_3)$

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

2.1.2 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

2.1.2.1 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.

2.1.2.2 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.

2.1.2.3 Euclidian distance

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

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

2.1.2.4 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.

2.1.2.5 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 . More info at wikipedia.org.

2.2 Preprocessing data

2.2.1 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(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

2.2.2 Encoding categorical features

The string values (e.g. “male”, “female”) of features have to be converted into integers. This can be done by two methods:

2.2.2.1 Ordinal Encoding

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

```
from sklearn.preprocessing import OrdinalEncoder
ord_enc = preprocessing.OrdinalEncoder()
ord_enc.fit(X)
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.

2.2.2.2 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).

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

More info: scikit-learn.org

2.2.3 Imputing missing values

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.

2.2.3.1 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 still experimen
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

2.3 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

2.4 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.

2.4.1 A priori feature selection

2.4.1.1 Low variance threshold

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

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

More info: scikit-learn.org

2.4.1.2 Mutual information

This method works by choosing the features that have the highest dependency between the features and the label.

$$I(X,Y) = D_{KL}\left(P(X=x, Y=y), P(X=x)\ P(Y=y)\right) = {}_{yY\ xX} P(X=x, Y=y) \log\left(\frac{P(X=x, Y=y)}{P(X=x)P(Y=y)}\right)$$

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_regre
X_new = SelectKBest(mutual_info_classif, k=8).fit_transform(X, y)
```

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

2.4.2 wrapper methods

2.4.2.1 Greedy feature selection

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.

```
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

2.4.3 Advice & Pitfalls

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).

2.5 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.

2.5.1 Grid search

The classic approach is exhaustive grid search: You create a grid of hyperparameters 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.

2.5.2 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 experimental
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 costl
                                    scoring = 'balanced_accuracy',
                                    random_state=0).fit(X, y)
```

More info: scikit-learn.org

2.6 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.

2.6.1 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.

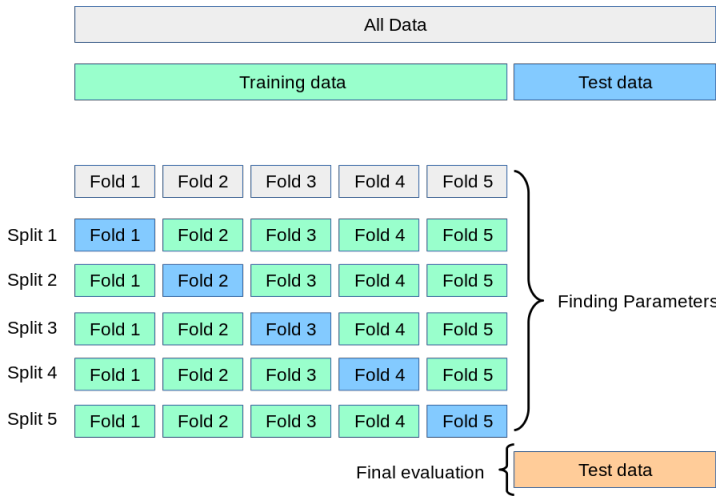

Figure 2.1: Schema of the process for 5-fold Cross Validation. The data is first split into training- and test-data. The training data is split into 5 sub-sets. The algorithm is trained on 4 sub-sets and evaluated on the remaining sub-set. Each sub-set is used for validation once. *Source: scikit-learn.org.*

```
from sklearn import svm
from sklearn.model_selection import cross_val_score
```

```
SVM_clf = svm.SVC (kernel='polynomial')
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.

2.6.2 Bootstrapping

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

2.7 Errors in machine learning

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.

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

2.7.1 Bias and Variance

2.7.1.1 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 that is an estimator for the true . You want to know whether your model over- or underestimates the true systematically.

$$\text{Bias}[] = \text{E}_{X|D}[]$$

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

2.7.1.2 Variance of an estimator

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

$$\text{Var}[] = \text{E}_{X|D}[(\text{E}_{X|D}[] \ )^2]$$

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).

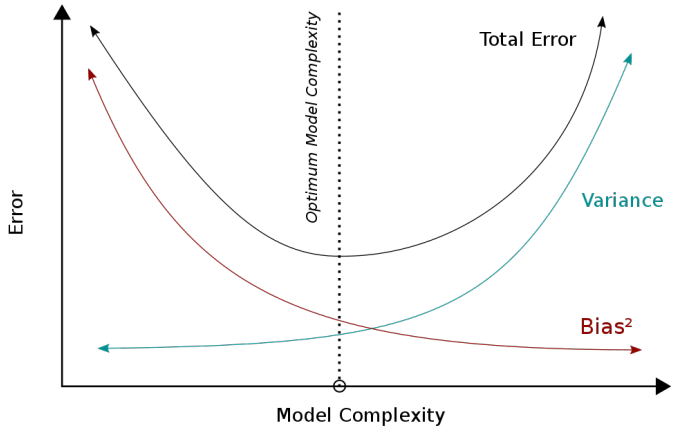

Figure 2.2: Relationship between bias, variance and the total error. The minimum of the total error lies at the best compromise between bias and variance. *Source: User Bigbossfarin on wikimedia.org..*

More info: wikipedia.org

2.7.2 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:

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

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 bias of estimator!). R is the regularization function and 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

2.7.3 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.

2.7.4 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.

2.7.5 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

2.8 Tips for machine learning projects

2.8.1 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.

2.9 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 bad 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.

- **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.

Chapter 3

Classification Methods

Classification is the assignment of objects (data points) to categories (classes). It requires a data set (i.e. training set) of points with known class labels. If the class labels are not known you can instead group the data using clustering algorithms (chapter 3).

3.1 Evaluation of Classifiers

3.1.1 Confusion matrix

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

3.1.2 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 = $2\frac{\text{precisionrecall}}{\text{precision+recall}}$
  This is useful in cases of unbalanced classes to balance the trade-off between precision and recall.

3.1.3 Area under the Curve

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

3.1.3.1 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.

! This measure is not sensitive to class imbalance!

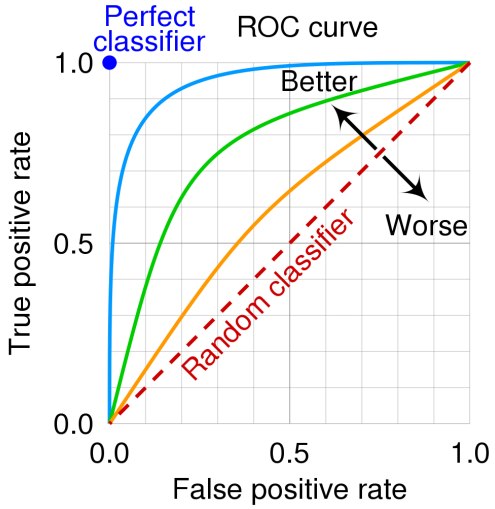

Figure 3.1: Area under the precision recall curve. Source: user cmglee on wikipedia.org

3.1.3.2 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.

3.1.4 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:)

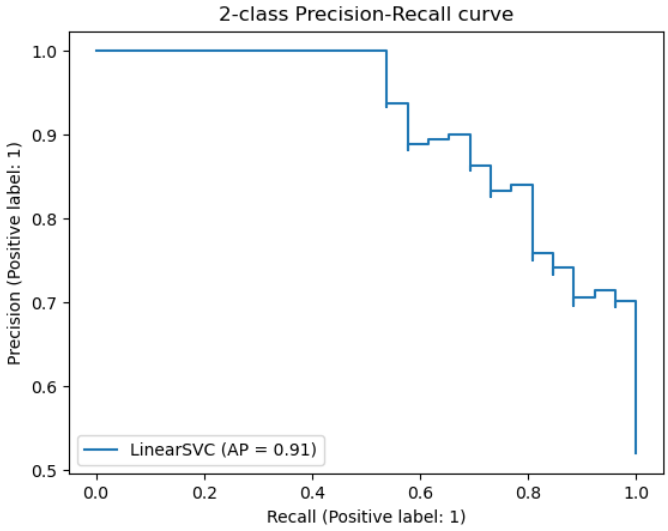

Figure 3.2: The precision-recall curve. Source: scikit-learn.org

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

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)
```

3.2 Classification Algorithms

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

scikit-learn.org

3.2.2 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:

$$P(Y = y_i|X = x) \;\; P(X = x|Y = y_i) \;\; P(Y = y_i)$$

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:

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

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

3.2.3 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:

$$log(P(y = y_i|x) = \frac{1}{2}(x \ _i)^{t1}(x \ _i) + \log P(y = y_i)$$

$_i$ is the mean of class i, $(x \ _i)^{t1}(x \ _i)$ 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

3.2.4 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 ($x^T + _0 = 0$) and the points. If points cannot be perfectly separated by the decision boundary, a soft margin SVM is used with a slack variable that punishes points in the margin or on the wrong side of the hyperplane. The optimization problem is given by [Hastie et al., 2009] :

$$\begin{aligned} &\max_{,0,=1} M, \\ \text{subject to } y_i(x_i^T + _0) \ &1 \ _i, \quad i, \\ _i \ 0, \quad _i \ &constant, \quad i = 1, ..., N, \end{aligned}$$

where 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:

$$G(x) = \text{sign}[x^T + _0]$$

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 [Platt, 1999]. 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])
```

3.2.5 Decision Trees

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

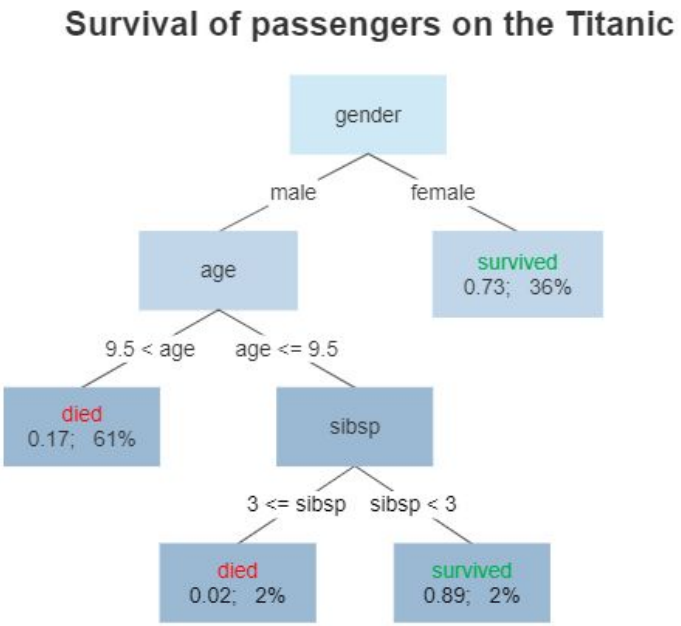

Figure 3.3: Decision trees work like flowcharts and have a root (the top node), branches (possible outcomes of a test), nodes (tests on one attribute), leafs (the class labels). This one shows the decision tree for the classifier predicting if a patient survives the sinking of the Titanic. Source: user Gilgoldm on wikipedia.org

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:

- 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

Due to the overfitting risk and the instability, ensembles of decision trees are commonly used.

3.2.5.1 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

3.2.5.2 Gradient boosted decision trees

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

Chapter 4

Unsupervised Learning

4.1 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.

4.1.1 metrics for Clustering algorithms

4.1.1.1 Silhouette coefficient

The silhouette coefficient compares the average distance of a point and the points in its own cluster $d(x, _C)$ to the average distance between the point and and the points of the second nearest Cluster $d(x, _C)$.

$$s(x) = \frac{d(x, _C)\ d(x, _C)}{\max(d(x, _C), d(x, _C))}$$

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.

```
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.

4.1.1.2 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)
```

4.1.2 K-Means Clustering

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

$$V(D) = \sum_{i=1}^{k} x_j \ C_i(x_j \ _i)^2,$$

where V is the variance, C_i is a cluster, $_i$ 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.

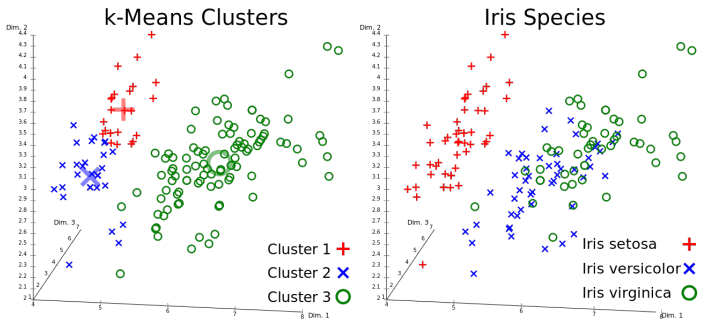

Figure 4.1: Data from the *Iris flower data set* clustered into 3 clusters using k-Means. On the right the data points have been assigned to their actual species. *Figure from user Chire on wikimedia.org.*

```
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.

4.1.3 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).

4.1.3.1 Basic Graph-Based Clustering

The basic algorithm works like this:

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

4.1.3.2 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 of the point (including the point itself).
2. A point is *directly reachable* if it is not a *core point* but within 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 unnecessary.

4.1.3.3 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 between the clusters C (equal to cutting edges between nodes that are least similar):

$$\min \frac{1}{2} \sum_{a=1}^{k} \sum_{b=1}^{k} (C_a, C_b)$$
$$\text{where } (C_a, C_b) = \sum_{v_i C_a, v_j C_b, ab} W_{ij}$$
$$\text{and } (C_a, C_a) = 0$$

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).

4.1.4 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)):

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

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:

$$LU = U \qquad L = UU^1$$

where U is the matrix of eigenvectors and 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

4.1.5 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:

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

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

4.1.6 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.

4.1.6.1 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 ($p(x_i \ \ C_k|\ă_{i,k}, _k)$).
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

4.1.6.2 Other models

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

4.1.7 Artificial Neural Networks for Clustering

See chapter *Neural Networks* (5)

4.2 Mapping to lower dimensions

4.2.1 Manifold learning

4.2.1.1 Isomap

4.2.1.2 Local linear embedding

4.2.1.3 Multi dimensional scaling

4.2.2 Decomposition techniques

4.2.2.1 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.

4.2.2.2 Principle Component analysis (PCA)

MAke subchapter: Kernel PCA

4.3 Outlier detection

4.3.1 Local outlier factor

4.3.2 Isolation forest

Chapter 5

Generative models

5.1 Generative Models for Discrete Data

5.1.1 Bayesian Concept Learning

can learn a concept $c \quad C$ from positive examples alone. For that define the posterior: $p(c|D)$. To get to learn a concept you need a hypothesis space H and a version space (a subset of H) that is consistent with D. You choose a hypothesis h by assuming that samples are randomly chosen from the true concept and calculate $p(D|h) = [\frac{1}{|h|}]^N$ (sampling the N data points from h). You than choose the hypothesis that has the highest probability (thereby you choose suspicious coincidences of too broad models). The priors can be chosen e.g. by giving lower priority to concepts with complex rules (e.g. "all powers of 2 below 100 but not 64."). This is subjective, however often beneficial for rapid learning. Using Bayes rule, we can calculate the posterior:

$$p(h|D) = \frac{p(D|h)p(h)}{p(D)} = \frac{p(D|h)p(h)}{_{h\,H}\, p(D|h)p(h)} = \frac{\mathbb{I}(D \quad h)p(h)}{_{h\,H}\, \mathbb{I}(D \quad h)p(h)},$$

where $\mathbb{I}(D \quad h)p(h) = 1$ if the data adhere to the h. The maximum of $p(h|D)$ is the **MAP estimate**.

With more data the MAP-estimate converges to the MLE. If the true hypothesis is in H then MLE and MAP will converge to it (consistent estimators). If you take the entire distribution of the hypotheses you get a distribution for the estimate (and not a point prediction) **posterior predictive distribution**.

$$p(x|D) = _{h}\, p(x|h)p(h|D)$$

This weighting of hypotheses is called **Bayes model averaging**. For small data sets you get a vague posterior and broad predictive distribution. You can replace the posteriors with their delta-function:

$$p(x|D) = _{h}\, p(x|h)_{h_{\mathrm{MAP}}}(h)$$

plug-in approximation (under-represents uncertainty).

5.1.2 Beta-binomial model

This is a distribution that uses a binomial distribution as its likelihood and a beta-distribution over it's parameter as its prior.

5.1.2.1 Likelihood

$$p(D|) = \text{Bin}(k|n,) \ ^k(1\ )^{nk},$$

where k are the successful trials, n are the total trials and are the success-probabilities of the single experiments. k and $n\ k$ are *sufficient statistics of the data*: $p(|D = p(|k, n\ k))$.

5.1.2.2 Prior

$$\text{Beta}(|,) \ ^1(1\ )^1$$

the parameters and are used as hyper parameters. The prior has the same form as the likelihood *conjugate prior*.

5.1.2.3 Posterior

$$p(|D) \ \text{Bin}(k|n,)\text{Beta}(|,) = \text{Beta}(|k+, n\ k+)$$

We add pseudo-counts $(,)$ to empirical counts (N, k). The posterior predictive distribution is:

$$\begin{aligned} p(x=1|D) &= \tfrac{1}{0}\, p(x=1|)p(|D)\text{d} = \tfrac{1}{0}\,\text{Beta}(|N\ k+, k+)\text{d} \\ &= \text{E}[|D] = \frac{N\ k+}{N\cancel{k+k}+\ +} \end{aligned}$$

If instead we used the MLE for instead ($p(|D) = p(|_{\text{MLE}})$) and we only had little data and no failures (e.g. 3 coin flips and all are tails). Then the MLE estimate would be $_{\text{MLE}}) = 0/3 = 0$. This is called *zero count estimate* problem or *black swan paradox* (i.e. you don't attribute possibilities to something you have never seen before). Solution: You use a uniform prior ($= \ = 1$): $p(x=1|D) = \dfrac{N\ k+1}{N+2}$.

5.1.3 Dirichlet-multinomial model

Before: model of k , now: k times a (e.g. four pips on a die roll) in n experiments.

5.1.3.1 Prior:

The Dirichlet distribution:

$$\text{Dir}(|) = \frac{1}{B()} \ _{k=1}^{K} \ _k^{k1}$$

5.1.3.2 Posterior:

$$\begin{aligned} p(|D) \ & p(D|)p(|) \\ & \ _{k=1}^{K} \ _k^{N_k\ k1} = \ _{k=1}^{K} \ _k^{N_k+k1} \\ & = \text{Dir}(|_1 + N_1, \ , _K + N_K) \end{aligned}$$

The posterior predictive distribution is:

$$\begin{aligned} p(X = j|D) &= p(X = j|)p(|D)\mathrm{d} \\ &= p(X = j|_j)\left[p(_j, _j|D)\mathrm{d}_j\right]\mathrm{d}_j \\ &= {}_j p(_j|D)\mathrm{d}_j = \mathrm{E}[_j|D] = \frac{N_j + {}_j}{{}_k N_k + {}_k} \end{aligned}$$

(Again, we avoid the the zero-count problem via the pseudo counts.)

Chapter 6

Regression

6.1 Evaluation of regression models

6.1.1 R^2 score

6.1.2 Mean squared error

6.1.3 Visual tools

6.2 Linear Models

6.2.1 Ordinary Least Squares

6.2.2 Lasso regression

6.2.3 Ridge regression

6.2.3.1 Kernel ridge regression

6.2.4 Bayesian regression

6.2.5 Generalized linear models

6.3 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.

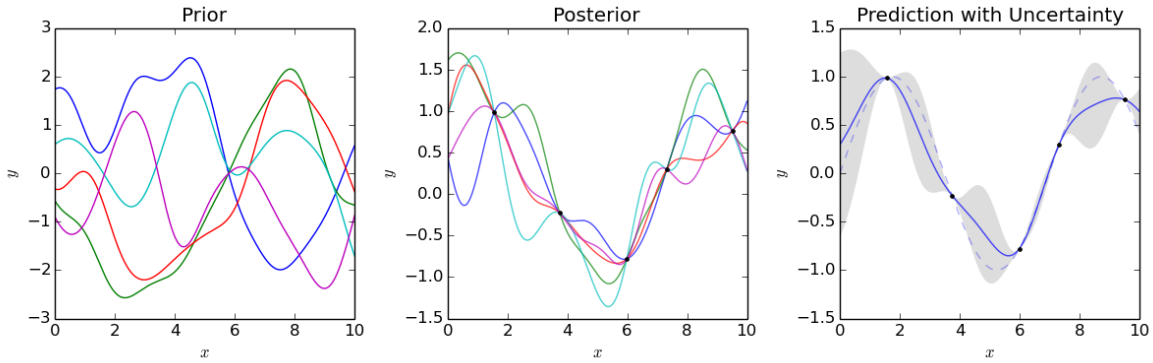

Figure 6.1: Schema of the training process of Gaussian process regression. The left graph shows the prior samples of functions before. These functions are then conditioned on the data (graph in middle). The right graph shows the predictions with the credible intervals in gray. *Source: user Cdipaolo96 on wikimedia.org* .

Pros:

- The model reports the predictions with a certain probability.
-

Cons:

- Training scales with $O(n^3)$.
- 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 hyperparameters are tuned by
gpr = GaussianProcessRegressor(kernel=kernel)
gpr.fit(X, y)
gpr.predict(X, return_std=True)
```

More info: scikit-learn.org

6.4 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, max_fea
gbr.fit(X, y)
```

More info: scikit-learn.org

6.5 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
```

6.5.1 ARIMA(X) Model

univariate time series model with exogenous regressor.

6.5.2 VARMMA(X) Model

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

6.5.3 Prophet-Model

Explain Prophet model from Facebook. Source: https://otexts.com/fpp3/prophet.html

Chapter 7

Neural Networks {#Neural Networks}

7.1 Introduction

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**: $f(x,) = (a)$ (e.g. a sigmoid function for logistic regression, giving you a probability output. 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.

7.1.1 Non-Linearities

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

7.1.1.1 Sigmoid/Logistic Functions

7.1.1.2 Tanh Functions

7.1.1.3 Rectifiers/ReLU

7.1.1.4 Terminology

- **Input layer/visible layer:** Input variables
- **Hidden layer:** Layers of nodes between input and output layer

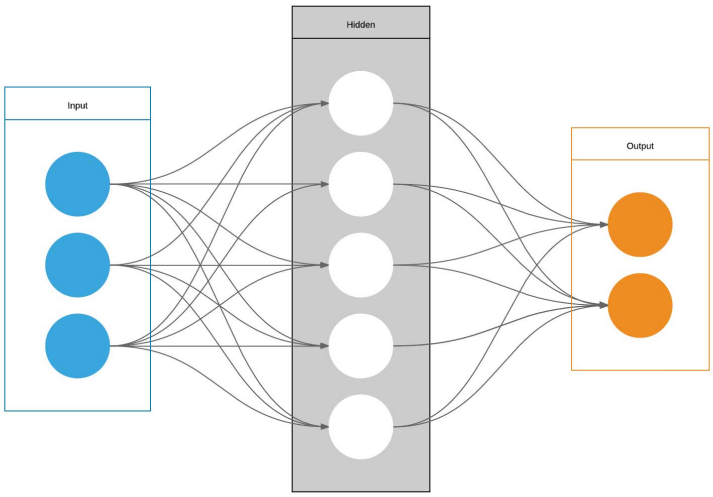

Figure 7.1: Model of an artificial neural network with one hidden layer. *Figure from user LearnDataSci on wikimedia.org.*

- **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

7.2 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 $6400\ 16 + 16\ 16 + 16\ 10 + 16 + 16 + 10 = 102858$ parameters. This is a very high number of degrees of freedom and requires a lot of training samples.

```
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()
```

7.2.1 Bekpropageshn

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 images 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*.

7.3 Convolutional Neural Networks

7.4 Autoencoders

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:

$$\min_{W,b} \frac{1}{N} \sum_{i=1}^{N} ||x_i \; x_i||^2$$

with x_i being the original datapoint and x_i the reconstructed datapoint.

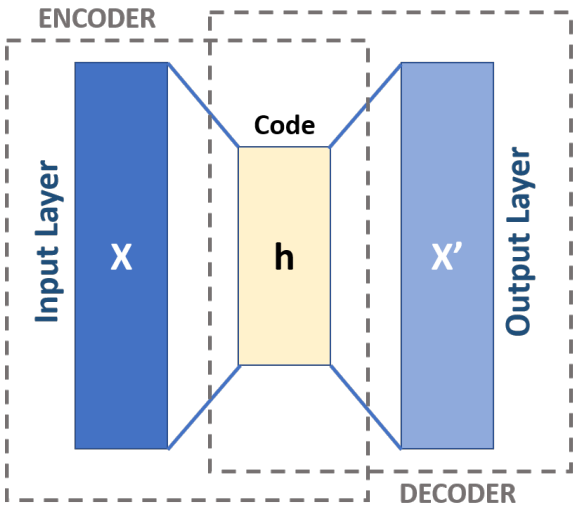

Figure 7.2: Model of an autoencoder. The encoder layers compress the data towards the code layer, the decoder layers decompress the data again. *Figure from Michela Massi on wikimedia.org.*

7.4.1 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 XXXX combined the cost function of the **autoencoder and k-means clustering**:

$$\min_{W,b} \frac{1}{N} \sum_{i=1}^{N} ||x_i \; x_i||^2 \sum_{i=1}^{N} ||f(x_i) \; c_i||^2$$

with $f(x_i)$ being the non-linearity of the code layer and 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 XXXX employs autoencoders combined with sparse subspace clustering 3.3.1. They used autoencoders and optimized for a compact representation of the code layer:

$$\min_{W,b} \frac{1}{N} \sum_{i=1}^{N} ||x_i \; x_i||^2 \; ||V||_1$$
$$\text{s.t.} F(X) = F(X) \; V \text{ and } \text{diag}(V) = 0$$

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

7.5 Generative Adversarial Networks

7.6 Recurrent Neural Networks

Chapter 8

Explanation and inspection methods

Bibliography

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. 2009. ISBN 9780387848570. doi: 10.1007/b94608_4.

John C. Platt. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. *Advances in large margin classifiers*, 34(6), 1999.