2022-10-14 - Initial concepts & vocabulary

Intro

Table of contents

Lesson resources.

Slides here.

Introduction

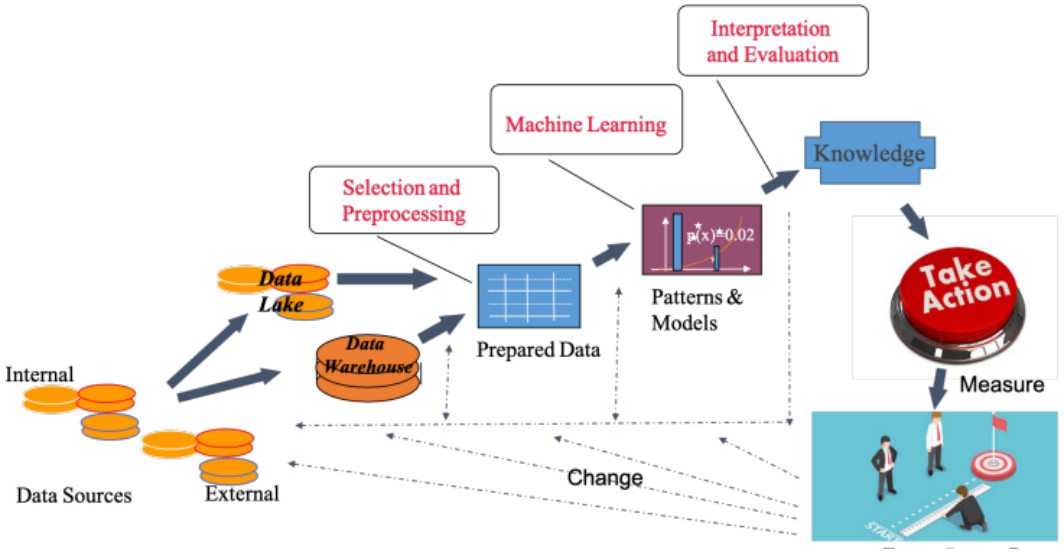
Knowledge vs Data

- **Data** are facts about reality or a certain entity
- **Knowledge** is information extrapolated from said data

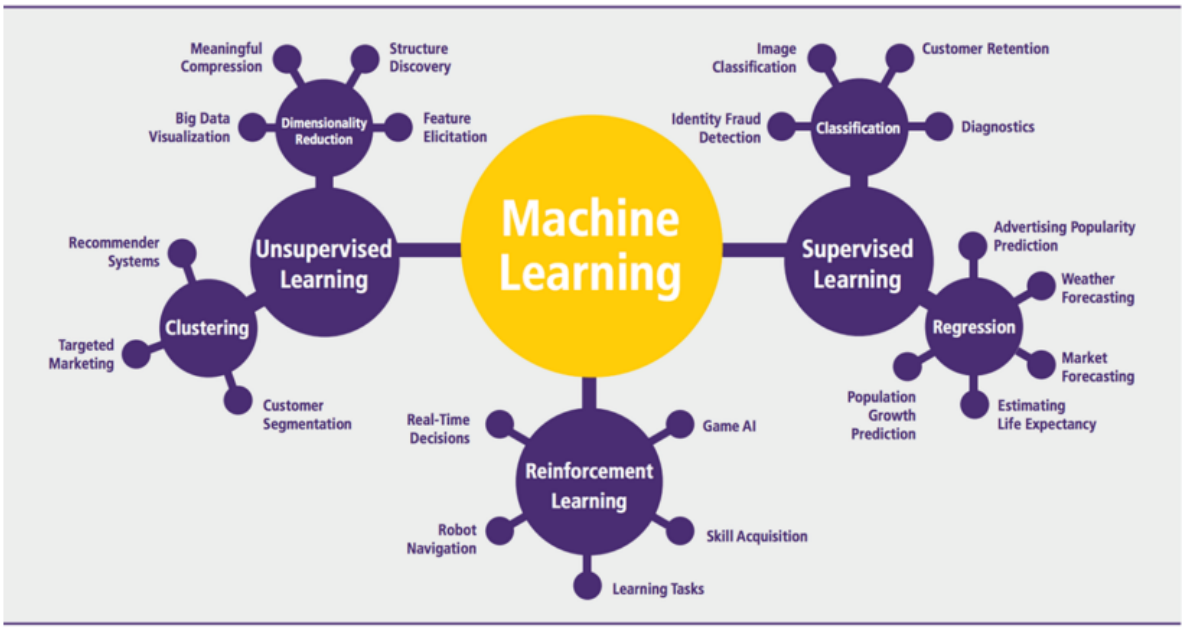
Some vocabulary

- **Data Mining** – The entire discovery process sets up a pipeline starting from the data and ending with the general patterns and their translation into useful actions.
    - From this whole process, we acquire knowledge.
- **Machine Learning** – the application of methods and algorithms to extract the patterns from the data.

The Discovery Process in Data Mining (Map)



The Machine Learning Map



Tasks to "solve" in the process

(this list can be seen as an index of what we will see in the next weeks).
Here are some of the tasks to do during the data mining process.

- *Classification* and *class probability estimation*:
  - among the customers of a phone company, which are likely to respond to a given offer? can I sort them on the basis of the probability of responding?
  -
- **Regression (value estimation)**
  - given a set of numeric attribute values for an individual, estimate the value of another numeric attribute.
    - i.e. How much will a given customer, whose characteristics are known, use a given service?
  - it is related to classification, but the methods are completely different.
  - It is a supervised activity
- **Similarity matching**: identify similar individuals based on data known about them
  - necessitates similarity measures
    - e.g. which are the companies similar to our best customers? they could be target of our next customer acquisition campaign
- **Clustering**: groups individual in a population on the basis of their similarities.
  - i.e. DNA sequences could be clustered in functional groups
- **Co–occurrence grouping**: attempts to find associations between entities according to the transactions in which the appear together
  - (also known as frequent item set mining, association rule discovery, market basket analysis)
  - i.e. what items are commonly purchased together?
- **Profiling** (also known as behavior description)
  - i.e. What is the typical cell phone usage of this customer segment?
  - the behavior can be described in a complex way over an entire population
  - usually the population is divided in groups of similar behavior
  - useful also to detect *anomalies*
- **Link analysis and prediction**: in a world where there exist items and connections (i.e. a graph), try to infer missing connections from the existing ones

    - since you and Karen share ten friends, maybe you would like to be Karen's friend?

- **Data reduction**: attempts to take a large set of data and replace it with a reduced one, preserving most of the important information.
    - a smaller set can be easier to manipulate, or even show more general insights
    - involves *loss of information*.
    - looks for the best trade–off between information loss and improved insight

- **Causal modeling**: understand what events or actions actually influence others
    - i.e. consider that we use predictive modeling to target advertisements to consumers, and we observe that indeed the targeted consumers purchase at a higher rate after having been targeted. Was this because the advertisements influenced the consumers to purchase? Or did the predictive models simply do a good job of identifying those consumers who would have purchased anyway?

Supervised vs unsupervised methods

Let's consider these questions:

- Does our population *naturally* fall into different groups?
    - When asking this question, there is no specific purpose or target for grouping: it should emerge by observing the characteristics of the individuals.
    - This is an example of **unsupervised mining**.
- Can we find groups of customers who have particularly high likelihoods of canceling their service soon after their contracts expire?
    - We have just defined a specific target : cancelling or not.
    - This is an example of **supervised mining**.
    - This problem is called *churn analysis*.

In summary:

- supervised → specific target
- *unsupervised* → natural process, without a specific target

The techniques for supervised situations are substantially different from those of the unsupervised ones.
Being supervised or unsupervised is a characteristic of the problem and/or the data, it is not a design choice.
Supervised information is usually added to the attributes of the individuals.

How to obtain supervised information?

There are two main ways to obtain supervised information:

- information provided by experts e.g. the soybean disease labels of the example of page
- history
    - e.g. we have an history of the customers who cancelled their service subscription.
    - the supervised information is not available run–time, when we must decide what to do.
    - later on the history will tell us the value of the unknown attribute which influences our actions.
    - we want to learn how to guess the unknown attribute from the known ones.

Reinforcement Learning (a few words)

- target: **a sequence of actions** which obtains the best result
    - it is not a label, or a group, or a value, or a class
- learn: a policy
- how: try a policy – get a reward – change the policy focus: the overall policy, rather than the single actions

[the following are orphan notes...]

Decision Process

The **decision process**

Model of data

A **model of data** is

In this case, the model is the analisys that the soft

2022-10-19 - The Data

Table of contents

Lesson's resources

Slides here

What is a dataset?

A dataset is a collection of data related to some specific information.
We can represent data in two ways:

- **Tabular data**: easiest way of representing data, using rows and columns.
- **Untabular data**: al the rest.

Unrelated: CSV is a dataset type, and stands for *comma-separated values*.

Issues on data

Some issues on data are:

- **Type**: can be qualitative (non-number) or quantitative (number), or structured.
- **Quality**: we have to consider the quality of data.
  - In the process of representing reality with data, we may make mistake. It is in relation to how data is *collected*.
  - The better the quality, the better result from machine learning
  - Some mining techniques are better than others (some data collecting processes are also more prone to error).
- *Pre-processing* (or preprocess) is the activity of modifying data to ease mining activities. So it is done before the "real" machine learning.

Data Types and numerical properties

This table is pretty important:

Data Type		*Description*	*Examples*	*Descriptive statistics allowed*
Categorical	Nominal	The values are a set of labels, the available information allows to distinguish a label from another Operators: $=$ and $\neq$	zip code, eye color, sex, ...	mode, entropy, contingency, correlation, χ^2 test
	Ordinal	The values provide enough information for a total ordering Operators: $<>\leqslant\geqslant$	hardness of minerals, non–numerical quality evaluations (bad, fair, good, excellent)	median, percentiles, rank correlations
Numerical	Interval	The difference is meaningful Operators: $+-$	Calendar dates, temperatures in centigrades and Fahrenheit	average, standard deviation, Pearson's correlation, F and t tests
	Ratio	Have a univocal definition of 0 Allow all the mathematic operations on numbers	Kelvin temperatures, masses, length, counts	geometric mean, harmonic mean, percentage variation

A little precisation:

- Ratio values: they are similar to interval values, but they are in relation of something with an absolute and unique reference (of 0). i.e. Kelvin temperatures, which can't go below 0.

> **Note**
>
> In this table, the "description" and "descriptive statistics" columns are incremental, i.e. the properties described in the row are added to the other properties describe above.

Here's an example:

Patient	Treatment	Treatment Day	Temperature	Pain
XXXX	a	2	37	3
XXXX	a	3	37	2
XXXX	a	4	36.5	1
YYYY	b	1	38	3
YYYY	b	2	37.5	2

Example 1

Patient	Weight	BirtyYear	Age	Sex
XXXX	78	1970	50	M
YYYY	56	1980	40	F

Example 2

- Column patient is a *nominal* value. It can also be seen as an identifier, but are not useful for machine learning, but only to connect tables in the relational model.
- Treatment is a *nominal* value. In this case, it is not important the order of values.

Allowed Transformation

This table is a companion to the previous one:

Data Type		*Transformation*	*Comment*
Categorical	Nominal	Any one–to–one correspondence	the SSN can be arbitrarily reassigned (masking)
	Ordinal	Any order preserving transformation new $\leftarrow f(\text{old})$ where f is a monotonic function	(bad, fair, good, excellent) can be substituted by (1,2,3,4)
Numerical	Interval	Linear functions new $\leftarrow a + b * \text{old}$	centigrades and Fahrenheit temperatures can be converted either way
	Ratio	Allow any mathematical function, *standardization*, variation in percentage	Kelvin temperatures, masses, length, counts

Some notes:

- in an ordinal value, using any order preserving transformation.
  - This transformation though introduces some assumption that might not necessarily be true.

Number of values

An additional issue for a column is the number of values.
For example, assume we have a discrete domain.

- We could have a finite number of values
  In some special cases, we would have:
- binary attributes
- identifiers (which are only useful in data manipulation).

Continuous domains are typical represented as floating point variables, even if:

- Nominals and ordinals are discrete
- Intervals and ratio are continuous
- Counts are discrete and ratio type.

Asymmetric attributes

In this case, only the presence of the value is considered important (a non-null value)

General characteristics of data sets

- **Dimensionality**: the difference between having a small or a large (hundreds, thousands,..) of attribute is also qualitative. In these cases, the algorithm that I can use are different.

- **Sparsity**: when there are many missing values (zeros or nulls).
- Beware the *nulls in disguise*: a widespread bad habit is to store zero or some special value when a piece information is not available.
  - i.e. missing values doesn't necessarily mean that the student has not passed the exam.
- **Resolution** of the data has also a great influence on the results.
  - Too general data can hide patterns.
  - Too detailed data can be affected by noise.

How to record data?

- Tables: i.e. relation
- Transction: a row is composed by: TID + set of Items
- Data matrix: like tables, but all the columns value are numbers.
- Sparse data matrix: asymmetric values of the same type

Data matrix example:

Projection of x load	*Projection of y load*	*Distance*	*Load*	*Thickness*
10.23	5.27	15.22	2.7	1.2
12.65	6.25	16.22	2.2	1.1

Transactional data example:

TID	*Items*
1	bread, coke, milk
2	beer, bread
3	beer, coke, diaper, milk
4	beer, bread, diaper, milk
5	coke, diaper, milk

2022-10-21 - Classification I

There are 2 types of classification: supervised and unsupervised classifications.
As of now, we're only interested in **supervised** classifications.

Supervised Classification

Consider the "soybean" example shown in the introduction (here, slides 21).
The data set X contains N individuals described by D attribute values each.
We have also a Y vector which, for each individual x contains the *class* value $y(x)$.
The class allows a finite set of different values (e.g. the diseases), let's call it C.
The class values are provided by experts: *the supervisors*.

We want to learn how to guess the value of the $y(x)$ for individuals which have not been examined by the experts.
We want to learn a **classification model**.

Classificiation Model

A classification model is an algorithm which, given an individual for which the class is not known, computes the class.
The algorithm is *parametrized* in order to optimize the results for the specific problem at hand.

Developing a classification model requires:

- choose the *learning algorithm*
- let the algorithm learn its parametrization (through the use of already supervised data)
- assess the quality of the classification model

The *classification model* is used by a run–time **classification algorithm** with the developed parametrization. This means that we'll use the model to classify new data.

Usually, there will also be an additional parameter (called θ, usually it is a threshold) that will influence the output. We'll see shortly.

Classification model or, shortly, ***classifier***

A *classifier* is a decision function which, given a data element x whose class label $y(x)$ is unknown, makes a prediction as

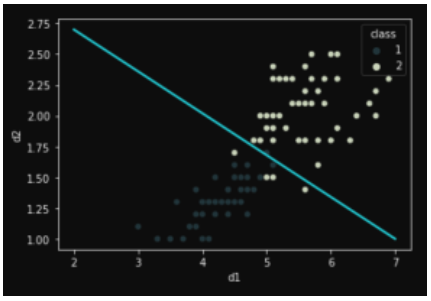
$$M(x, \theta) = y(x)_{pred}$$

where θ is a set of values of the parameters of the decision function.
The prediction can be true or false.

The learning process for a given classifier $M(.,.)$, given the dataset X and the set of *supervised class labels* Y, determines θ in order to reduce the prediction error as much as possible.

```
Why am I not able to reduce the prediction error to 0? The answer is that it is because I cannot predict
all the possible situations in advance. In the real world the information can be slightly or no
The second reason is that my decision function will never be able to learn every possible decision
function.
```

Example of a decision function



In this example, we have a dataset with two dimensions and 2 classes.
The color of the dot represents the class.
One could decide a decision function as a straight line, and we'll get a function like this:

$$\theta_1 * d_1 + \theta_2 * d_2 + \theta_3 \geqslant 0 \Rightarrow c_1$$

$$\theta_1 * d_1 + \theta_2 * d_2 + \theta_3 < 0 \Rightarrow c_2$$

This model makes some errors, and even the best choice of parameters (the θs) cannot avoid any errors.
The phrase "all models are wrong, but some are useful" captures the fact that we cannot model reality in precision.

Nevertheless, different models can have different power to *shatter the dataset into subsets* with homogeneous classes.
In the previous example, we could use a quadratic function for example.

Vapnik-Chervonenkis Dimension

This property relates to the *shattering power* of a classification model.

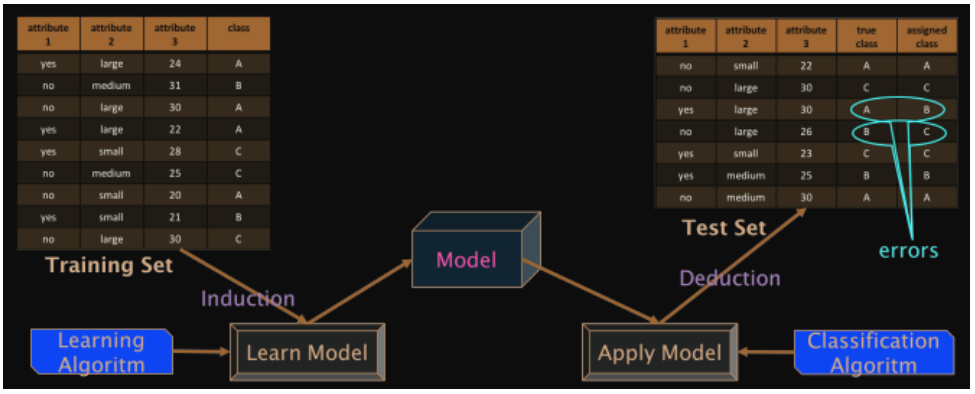
- Given a dataset with N elements there are 2^N possible different learning problems.
- If a model $M(.,.)$ is able to shatter *all the possible learning problems* with N elements, we say that it has *Vapnik-Chervonenkis Dimension* equal to N
- The straight line has VC dimension 3, since i can separate for sure 3 elements with a single straight line.
  - don't worry, frequently, in real cases, data are arranged in such a way that also a straight line is not so bad

Classification workflow

1. Learning the model for the given set of classes, meaning that:
   1. a *training set is available*, containing a number of individuals
   2. for each individual *the value of the class label is available* (also named ground truth)
   3. the *training set should be representative* as much as possible (the training set should also be obtained by a random process)
      - Meaning that the training data should be highly comparable with the data used in the test set.
   4. the model is fit learning from data the best parameter setting
2. Estimate the accuracy of the model (the number of good predictions made on the test set).

1. a *test set* is available, for which the class labels are known.
    - Essentially, I used my supervised data and I split it into 2 parts: a test set and a training set.
2. the model is run by a classification algorithm to assign the labels to the individuals on the test set.
3. the labels assigned by the model are compared with the true ones, to estimate the accuracy.

3. The model is used to label new individuals.



Two flavor classification

- **Crisp**: the classifier assigns to each individual *one* label.
- **Probabilistic**: the classifier assigns a *probability for each of the possible labels*.

Decision Trees

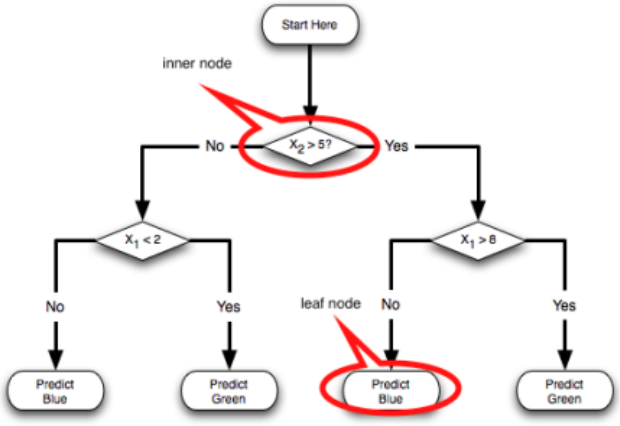
A run–time classifier structured as a decision tree is a *tree–shaped set of tests*.
The decision tree has inner nodes and leaf nodes:

- Inner nodes: `if test on attribute d of element x then execute node1 else execute node2`
- Leaf nodes: `predict class of element x as c2`

Generating this model

Given a set X of elements for which the class is known (supervised data), grow a decision tree as follows:

- if all the elements belong to class c or "X is small" generate a leaf node with label c
- otherwise
    - choose a test based on a *single attribute with two or more outcomes*.
    - make this test the root of a tree with one branch for each of the outcomes of the test
    - *partition X into subsets* corresponding to the outcomes and *apply recursively the procedures to the subsets*.



Essentially: if we can stop, fine, otherwise we must partition the dataset. In the beginning of the tree, I have an entire dataset. According to the outcome of the inner node I split the dataset into two parts.

This recursive procedure allows us to reduce the *search space*.

Note: in the example here we are testing 2 attributes at the same time. If we were to test only a single attribute, then we would be using a line which is parallel to one of the 2 main axis.

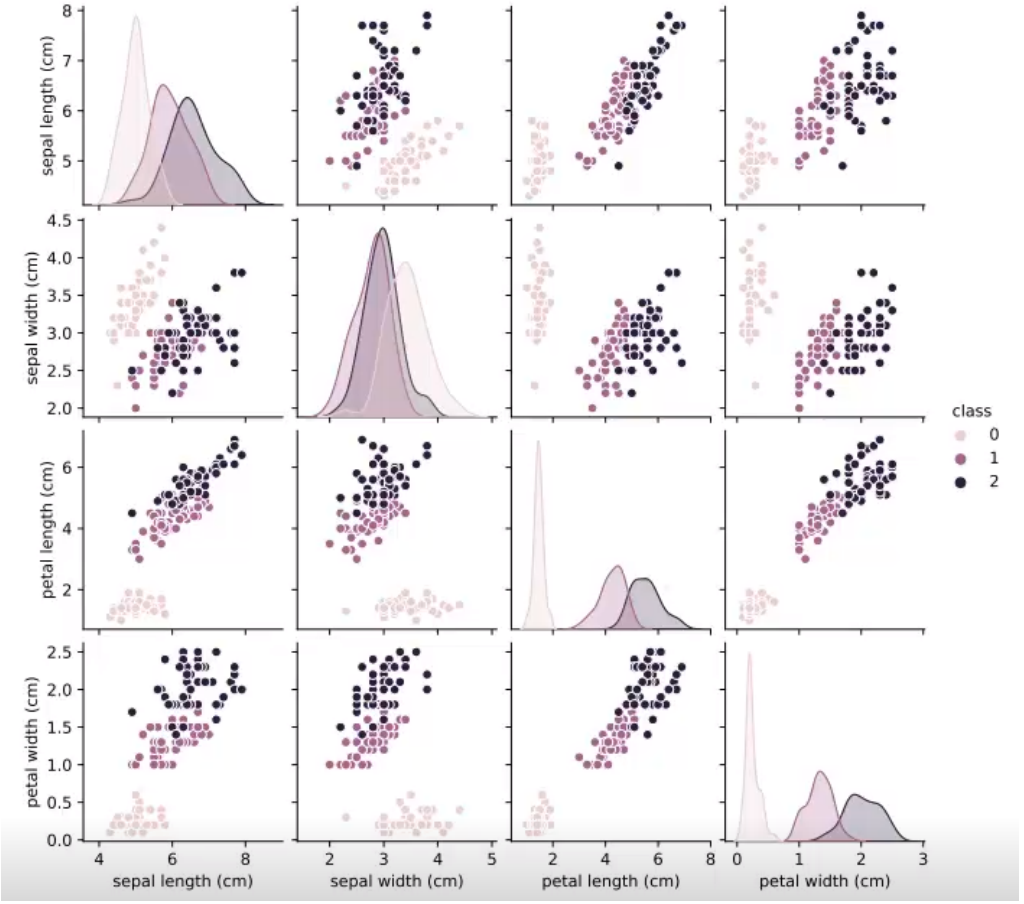
Problems to solve

1. which attribute should we test?
2. which kind of tests? (binary, multi–way, ... , depends also on the domain of the attribute)
3. what does it mean "X is small", in order to choose if a leaf node is to be generated also if the class in X is not unique?

Pairplots

A *pairplot* is a two-dimensional representation of a dataset, which can be seen as a collection of scatterplots.

The diagonal shows the distribution of values for a single attribute. while each cell shows us the so-called *scatterplots*. In each scatterplot, each point is associated with 2 attributes, while the color represents the class.



It is considered one of the most powerful visual representation of a dataset.

Supervised learning goals

Our goal is to design an algorithm to find interesting patterns (in classification, we mean to find a label), in order to forecast the values of an attribute given the values of other attributes.
Also:

- we need to distinguish real patterns from illusions (not all patterns are interesting).
  In our case, we must find patterns to **guess the class** given the other values.

How much can we evaluate if a pattern is interesting?
There are several methods, one of them is based on information theory, which is primarily used in telecommunication, and is based in the concept of *entropy*.

- Also, evaluating how much a pattern is interesting is like trying to find the best threshold to 'split' the dataset (i.e. splitting the dataset according to the value of an attribute)

Entropy

To understand this concept, let's start with an example:

- Given a variable with 4 possible values and a given probability distribution $P(A) = 0.25$, $P(B) = 0.25$, $P(C) = 0.25$, $P(D) = 0.25$
- an observation of the data stream could return `BAACBADCDADDDA...`

I could decide to use a 2-bit encoding:

$$A = 00, B = 01, C = 10, D = 11$$

Converting the transmission will be very easy (it will be `01000010010011101100111111100`).

But...

When the probability distribution are uneven, the problem becomes quite complex.
Let's consider:

$$P(A) = 0.5, P(B) = 0.25, P(C) = 0.125, P(D) = 0.125$$

We could use the same encoding as the first example, but we can do better.
Is there a coding requiring only 1.75 bit per symbol, on the average? YES!
$A = 0, B = 10, C = 110, D = 111$

We can improve it tho!

What if there are only three symbols with equal probability? $P(A) = 1/3, P(B) = 1/3, P(C) = 1/3$
Again, the two–bit coding shown above is still possible.
But is there a coding requiring less than 1.6 bit per symbol, on the average? YES!
$A = 0, B = 10, C = 11$ or any permutation of the assignment

General case

We can generalize what has been just said using the following considerations.
Given a source X with V possible values, with probability distribution:

$$P(v_1) = p_1, P(v_2) = p_2, \ldots, P(v_v) = p_v$$

the best encoding allows the transmission with an average number of bits given by:

$$H(X) = -\sum_j p_j \log_2(p_j)$$

$H(X)$ is the **entropy** of the information source X.

Spoiler: we will consider as information source the *information of classes* of each individual, so the different values will be the different labels of the classes.

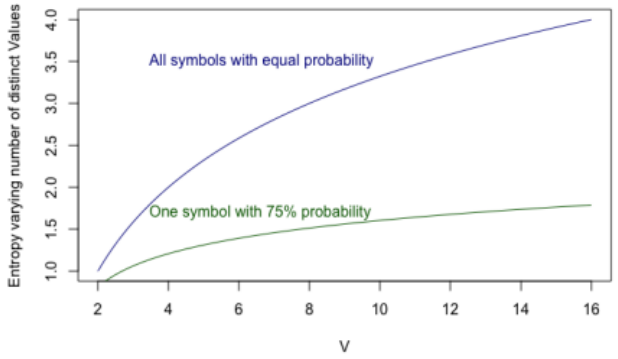
What happens if we have only 1 symbol?
Then, some probabilities will be 0 and one will be 1, and the $H(x)$ will be 0. *That means that if we have 0 variability, we have 0 entropy*.

Meaning of entropy

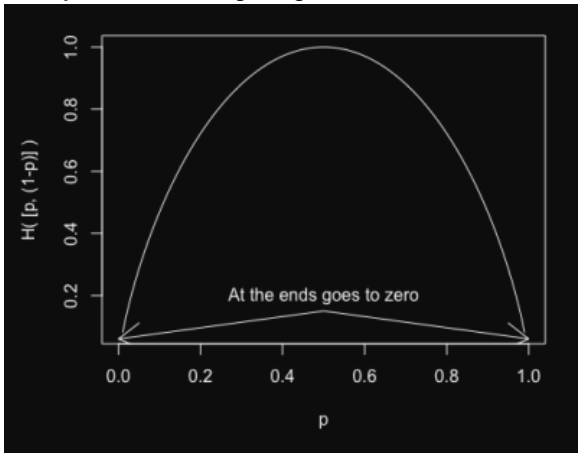
In general:

- **High entropy** means high uncertainty, and that *the probabilities are mostly similar*. So the histogram would be flat.
- **Low entropy** means low uncertainty, so *some symbols have much higher probability* (wrt others). Our histogram will have peaks.

- A higher number of allowed symbols increases the entropy.



In a binary source with symbol probabilities p and (1-p) when p is 0 or 1, the entropy goes to 0. This can be seen clearly in the following image:



Entropy after a threshold–based split

Our objective is to use a decision function in order to split the dataset in two parts according to a threshold on a numeric attribute.
Thus, the entropy changes, and becomes the weighted sum of the entropies of the two parts.
The weights are the relative sizes of the two parts.

Let $d \in D$ be a real-valued attribute, let t be a value of the domain of d, let c be the class attribute.

We define the entropy of c with respect to d with threshold t as:

$$H(c|d:t) = H(c|d<t) * P(d<t) + H(c|d \geq t) * P(d \geq t)$$

which means that is the sum between:

- The entropy of c for individuals where d is less than the threshold t, multiplied by the probability of those individuals.
- The entropy of c for individuals where d is more than the threshold t, multiplied by the probability of those individuals.

In general, the new entropy will be smaller than the previous one.

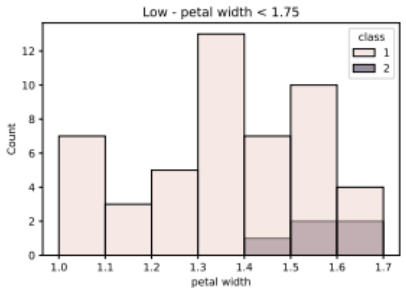
Information gain for binary split

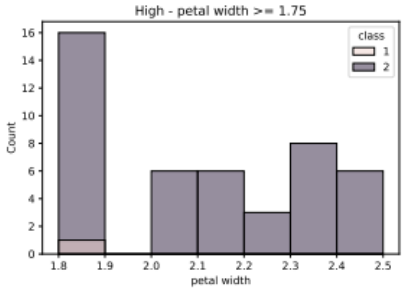
It is the *reduction of the entropy of a target class* obtained with a *split of the dataset* based on a threshold for a given attribute.
We define $IG(c|d:t) = H(c) - H(c|d:t)$, it is the information gain provided when we know if, for an individual, d exceeds the threshold t in order to forecast the class value.

We define $IG(c|d) = max_t\ IG(c|d:t)$.

- "The information gain of c making the best split on attribute d, is the maximum (in t) of the Information Gain of c making the best split on attribute d with threshold t."

Let's consider the iris dataset for example:



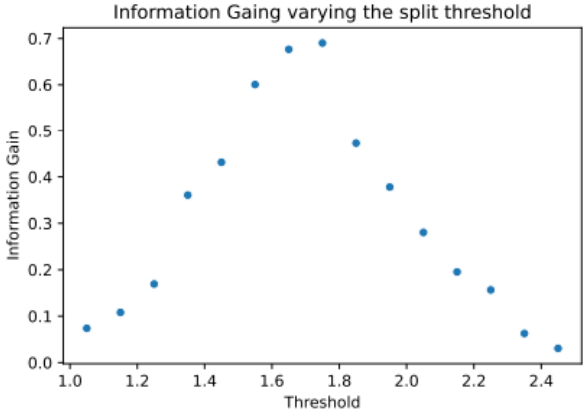


	low	high
1	49	1
2	5	45
	54	46

$$H(class|petalwidth : 1.8) = 0.31 =$$
$$- \quad (49/54 * log_2(49/54) + 5/54 * log_2(5/54)) * 0.54 +$$
$$(1/46 * log_2(1/46) + 45/46 * log_2(45/46)) * 0.46$$

In this case, choosing threshold $t = 1.8$ results in a pretty good split. Since we have mostly the same individuals in both cases (with a small margin of error: some of the low are on high and viceversa).

If we plot the information gain changing the threshold, we'll see that there is a maximum at around 1.8:



What can we do with information gain?

Predict the probability of long life given some historical data on person characteristics and life style:
Ex.

- $IG(LongLife|HairColor) = 0.01$
- $IG(LongLife|Smoker) = 0.2$
- $IG(LongLife|Gender) = 0.25$
- $IG(LongLife|LastDigitSSN) = 0.00001$

The attribute that gives us the most information is the Gender.
The attribute "LastDigitSSN" is random noise, and is negligible.

Decision Tree generation pt. 2

Which attribute should we test?

- test the attribute which **guarantees the maximum IG** for the class attribute in the current data set X
- partition X according to the test outcomes
- recursion on the partitioned data

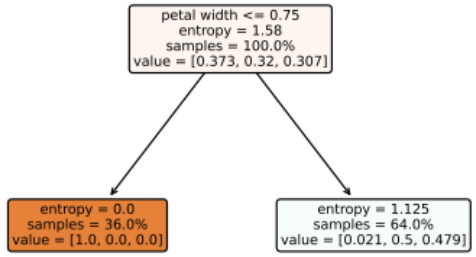
The supervised dataset will be split in two parts, randomly:

- One will be used as the **training set**, used to learn the model.
- The other will be used as the **test set**, used to evaluate the learned model on fresh data.

The proportion of the split is decided by the experimenter, but the parts *must* have similar characteristics (i.e. same proportion of attributes).
Some of the proportions are 80-20, 67-33 and 50-50.

One stump decision

It's the easiest decision tree, and has only one split.

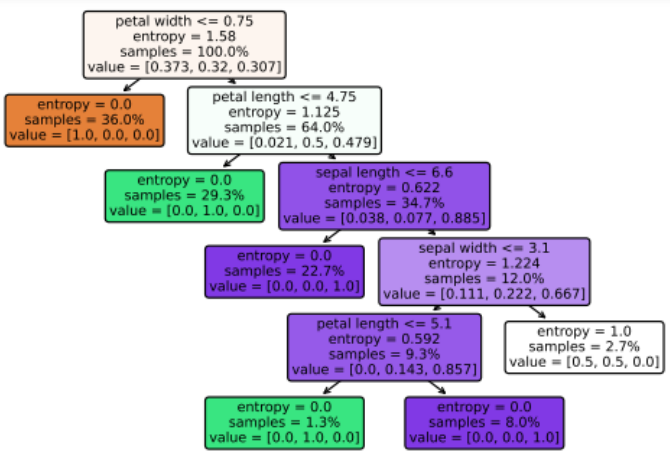


To sum up...

To generate a decision tree:

- choose the attribute giving the highest IG
- partition the dataset according to the chosen attribute
- choose as class label of each partition the majority

Decision tree example



In this case, our decision tree considers the whole iris dataset.
The more we go deep in this tree, the less confusion about classification there is: we can see this cleary in the leaves, where the entropy is 0.

In this examples, also, the entropy from a node to another sometimes increases. This is because we are considering a weighted entropy on the whole layer (depth), and not for the whole tree.
So, for example, we have entropy 1.224 weighted with 12% etc...

Let's consider the last white leaf on the right. As we can see, the entropy of this leaf is 1.0, and the value is [0.5, 0.5, 0], meaning that this leaf *has individuals of two different classes*. And there's no attribute capable of giving us an information gain: we can't go any further ein the decision process.
This is due to a **training set error**: either my model or the data are not able to discreminate completely the population. So, my model is not able to determine completely the ground truth.

We will see the training set error in the next lesson.

2022-10-26 - Classification II

Training Set Error

A **training set error** means that either my model based on the decision tree or the data are not able to discriminate completely the population.
This is because the model I generated with the available training data is not able to reproduce completely the ground truth.

To determine the training set error, we must execute the generated decision tree on the training set itself, and count the

number of discordances between the true and the predicted class: this is *the training set error*. The main causes of this are:

- the limits of decision trees in general: a decision tree based on tests on attribute values can fail
- insufficient information in the predicting attribute

So, we have a limit in the type of the model (that is, the decision tree) or in the data.

In our Iris dataset example, we had a training error of 1.35% (1 of 75 examples in the training set is not correctly classified by the learned decision tree).
It is the error we make on the data we used to generate the classification model, and *it is the lower limit of the error* that we make *with the training data*, which are the information we already know.

But, we are much more interested to an *upper limit*, or to a more significant value. Essentially, we are more interested in the error in the worst case.

Test Set Error

The **test set error** is *more indicative* of the expected behaviour with new data. Additional statistic reasoning can be used to infer *error bounds* given the test set error.

	Num Errors	Set Size	%*Wrong*
Training Set	1	75	1.35
Test Set	13	75	17.33

As we can see, the test error is *much* more than the training set error. This is due to something called overfitting.

Overfitting

Overfitting means, literally, that the model is too much fitted.
Definition: overfitting happens when *the learning is affected by noise*.
When a learning algorithm is affected by noise, the performance on the test set is (much) worse than that on the training set.

A decision tree is a *hypothesis* of the relationship between the predictor attributes and the class. Here are some definitions:

- h = hypothesis
- $error_{train}(h)$ = error of the hypothesis on the training set
- $error_{x}(h)$ = error of the hypothesis on the entire dataset

h **overfits** the training set if there is an alternative hypothesis h' such that

$$error_{train}(h) < error_{train}(h')$$
$$error_{\mathcal{X}}(h) > error_{\mathcal{X}}(h')$$

What are the possible causes of overfitting?

1. Presence of noise
   - individuals in the test set can have *bad values* in the predicting attributes and/or in the class label.
2. Lack of representative instances
   - Some situations of the real world can be *underrepresented*, or not represented at all, in the *training set*.
   - This is quite common.

Generalization the process is using the model fitted for a small portion of data with a bigger portion of the data, different from the previous.

A good hypothesis has a low generalization error, which means it works well on examples different from those in training.

Occam's Razor theory

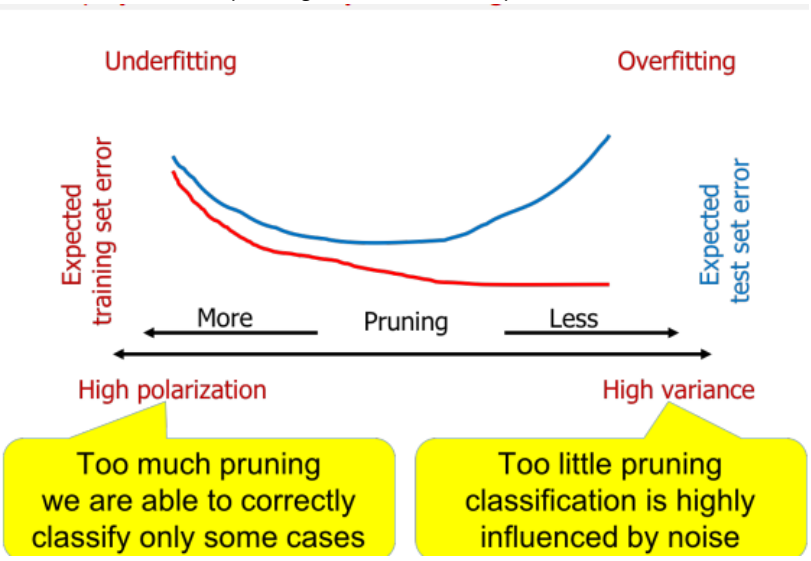
One of the thing we could do to reduce overfitting is to prune the decision tree.

"Everything should be made as simple as possible, but not simpler"

- This means that simple theories are preferable to complex ones, and a long hypothesis that fits the data is more likely to be a coincidence.
- *Pruning* a decision tree is a way to simplify it, but we need precise guidelines for effective pruning.

General effect of model simplification

This is the effects of pruning summarized in a picture:



What is in the middle of this graph is basically the *best situation* we could possibly have, that is a balance between test set error and training set error.

The situation shown in the figure happens with **ALL classification methods**, *not just decision trees*. This means that there's no best model, and we must try to find the optimal configuration.

So, our model must have like a slider that we must use to tune the behavior of the fitting.
In decision tree, we can tune the depth of the tree for example.

For example, let's consider the case in which both nodes in a decision tree have the same information gain. The fitting process needs to make a choice between the 2 variables, and Scikit learn chooses randomly.

This can be seen when using the code:

```
random.state = 3
model = DecisionTreeClassificafier(random.state = random.state)
model.fit(x, y, max_depth = 2)
```

As we've said, *depth*, for a decision tree, basically defines the amount of pruning/simplification in the model.

Let's suppose that we chose `random.state = 5`, and we instead have a training error of 3% instead of 5%.
A good model should not give us surprises, so I should be able to forecast a worst case ever, like if an error shouldn't more than X%, so if I observe that changing only the random effects, the error changes from 3% to 5%, then I should not rely on 3%.

What we could do is try some different random states randomly and consider the worst case.

N.B.: `max_depth` parameter is an example of the θ we've talked about in the previous lesson. These "sliders" are called **hyper-parameters**, and they influence how our model is generated. These parameters are chosen by the ML expert and must be optimized in order to achieve high precision.

Choice of the attribute to split the dataset

We are looking for the split generating the maximum **purity**. We need a measure for the purity of a node:

- a node with two classes in the same proportion has low purity
- a node with only one class has highest purity

A node with high purity means that it has a low diversity of labels, so a node with only one class has high purity.

Impurity functions (measuring the impurity of a node)

- Entropy (Information Gain instead guides us to chose the best split and measures the change of purity).
- **Gini Index** (used by Scikit learn by default).
- Misclassification Error (we skipped it).

Gini Index

Consider p node with C_p classes. Which is the frequency of the wrong classification in class j given by a random assignment based only on the class frequencies in the current node?

For class j:

- frequency $f_{p,j}$
- frequency of the other classes $1 - f_{p,j}$
- probability of wrong assignment $f_{p,j} * (1 - f_{p,j})$

The Gini index is **the total probability of wrong classification**:

$$\sum_j f_{p,j} * (1 - f_{p,j}) = \sum_j f_{p,j} - \sum_j f_{p,j}^2 = 1 - \sum_j f_{p,j}^2$$

The maximum value is when all the records are uniformly distributed over all the classes:
$1 - 1/C_p$.
The minimum value is when all the records belong to the same class: 0.

- When a node p is split into ds descendants, say $p_1, \ldots, p_{ds}$.
- Let $N_{p,i}$ and N_p be the number of records in the i-th descendant node and in the root respectively.
- We choose the split giving the maximum reduction of the Gini Index:

$$GINI_{split} = GINI_p - \sum_{i=1}^{ds} \frac{N_{p,i}}{N_p} GINI(p_i)$$

Algorithm for building DTs

There are several variants of DT building algorithms.

- We could also have tests based on linear combinations of numeric attributes, but these situation are very much complex.

Complexity of DT building

- N instances and D attributes in X
  - tree height is $O(logN)$
- Each level of the tree requires the consideration of all the dataset (considering all the nodes)
- Each node requires the consideration of all the attributes overall cost is $O(DNlogN)$.

The fitting (pruning) requires to consider globally all instances at each level, generating an additional $O(NlogN)$, which does not increase complexity (and so it is quite cheap).

Characteristics of DT Induction

- It is a *non–parametric approach* to build classification models. It does not require any assumption on the probability distributions of classes and attribute values.
  - "In statitics, the data is represented using a probability function, and I must find the parameters of this probability function" i.e. approximanting the distribution with a function (like Gaussian).
- In principle, finding the best DT is NP-complete, since we are following a greedy approach. Which means that the heuristic algorithms allow to find sub-optimal solution, but it is fast.
  - It is suboptimal because we do not find the best DT (we will need to consider every DT possible!)
- Runtime cost of using a DT to classify new instances is extremely low ($O(h)$ where h is the height of the tree).
- Redundant attributes do not cause any difficulty.
  - An attribute is redundant if it has the same ability as another attribute to guess the label.

- The nodes at a high depth are easily irrelevant (and therefore pruned), due to the low number of training records they cover
- In practice, the impurity measure has low impact on the final result
- In practice, the pruning strategy has high impact on the final result

N.B.: DTs are able to predict discrete values (the class) on the basis of *continuous* or discrete predictor attributes. This is because the algorithm doesn't know that, for example, only integers values are allowed.

2022-10-28 - Evaluation of performance of a classifier (Classification II)

Evaluation of a classifier

Model Selection

When we evaluate a model, we must consider a number of properties, and accuracy isn't necessarily the most important. There might be other types of errors or properties which do not influence accuracy, but are still important to consider.

i.e. if I have a model that is used to classify oil spills, which are a very rare occurrence, if our model always says that there aren't any oil spills (even if they are), our model accuracy will still be 99%.

Hyperparameter tuning

There's still the problem of *hyperparameter* tuning.
As we've seen, in supervised learning, the training set performance is **overoptimistic**.

- Evaluate how much the theory fits the data
- Evaluate the cost generated by prediction errors
- Empirically (and intuitively) the more training data we use the best performance we should expect
- Statistically, we should expect a larger covering of the situation that can occur when classifying new data
- We must consider the effect of random changes
- The evaluation is independent from the algorithm used to generate the mode

The meaning of ***test error***.

Let us suppone that the test set is a good representation, on the average, of the entire dataset X.

The relationship between the training set and X will be subject to probabilistic variability.
The evaluation of a classifier can be done either at different levels

- **general** - the whole performance of the classifier
- **local** - the *local* performance of a component of the model, i.e. a node of a decision tree

If test set error is X, we should expect a run-time error $x \pm ???$, meaning we should expect X and some variability.

Confidence interval in error estimation - Bernoulli process

Forecasting each element of the test set is like one experiment of a **Bernoulli process**.
A Bernoulli process is a repetition of binary experiments, where all the experiments are supposed to have the same probability of success (and failure).
So, in ML, a Bernoulli process looks something like this:

- good prediction ⇒ success (S)

- bad prediction ⇒ error (E)
  the events are the same as N independent binary random events of the same type.
  We define $f = E/N$ the **empirical frequency of error** (f or e).
  But we are not interested in the empirical frequency, but in the **true frequency of error**.

Empirical frequency and true frequency

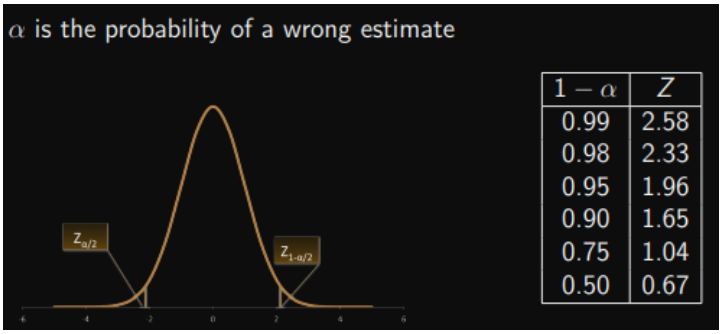
Deviations of the empirical frequency from the true frequency are due to noise. Usually, noise is assumed to have a normal distribution around the true probability (for $N \geq 30$)

We choose the *confidence level*, i.e. the probability that the true frequency of success is below the pessimistic frequency that we will compute. The confidence level can be computed as:

$$P\left(z_{\alpha/2} \leqslant \frac{f-p}{\sqrt{p(1-p)/N}} \leqslant z_{1-\alpha/2}\right) = 1-\alpha$$

- we assign the confidence level (which allows us to compute Z)
- we measure the empirical frequency (f in the formula)
- N is the number of experiments (in practice, it is the number of elements in the test set)

We can visualize what the confidence level looks like:
($\pm\, Z_\alpha$ are the values such that the tails of the Gaussian have area $\alpha/2$ each).



$1-\alpha$	Z
0.99	2.58
0.98	2.33
0.95	1.96
0.90	1.65
0.75	1.04
0.50	0.67

According to the Gaussian, if I wanto to be sure that my estimate is true, I need to go towards the tail ends of the graph. The pessimistic evaluation is close to the empirical frequency.

With a little algebra we can compute this formula:

$$\frac{1}{1+\frac{1}{N}z^2}\left[f+\frac{1}{2N}z^2 \pm z\sqrt{\frac{1}{N}f(1-f)+\frac{1}{4N^2}z^2}\right]$$

Where, if we substitute the $\pm$ with $+$ we'll get the **pessimistic error** e_{max}.
So, if I assign α, which is the probability that my pessimistic evaluation is wrong (i.e. 95% of confindence of being true) then:

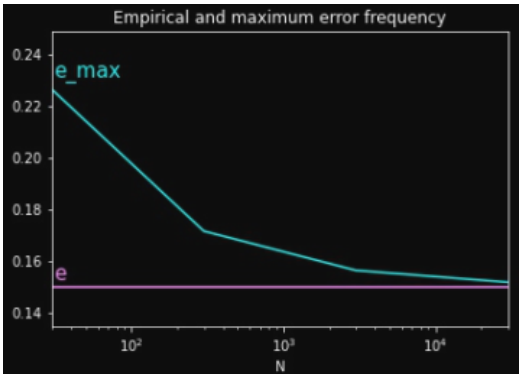
- I measure the f, the empirical frequency.
- N is known
- Z depends on the desired confidence level α (according to the table seen in this image)
  - it is the abscissa delimiting the area $1-\alpha$ for a normal distribution

Then *I can compute* the maximum error (**pessimistic error**) e_{max} according to the previous formula.

Here are some important observations on the formula:

- We can observe that as Z increases, the amount added to e to obtain e_{max} increases.
- A bigger N means a *bigger sample of data* for the **test**, so:
  - if I have a big N, it means that the training data is less (since test and training data are extracted from a single dataset).
  - So the quantities that I add to compute e_{max} are smaller, so ideally, *if I could use an infinite amount of test data*, the *test error* (the uncertainty) would be *0*.

We can see this in the following figure:



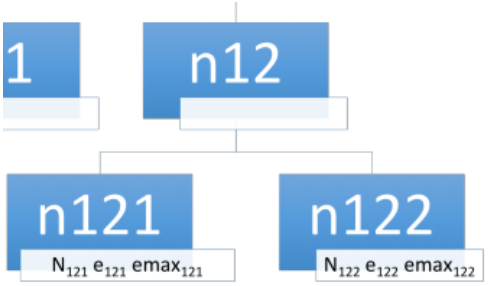
The *purple* one is empirical error frequency, while the *blue* line is the maximum error frequency.
We can see that if we have an higher number of test elements, the *uncertainty* is reduced, meaning that empirical error is very close to the maximum error.

Statistical pruning with error estimation

What can we do with the statistical concepts that we've just introduced?
We could, for example, apply the *C4.5 strategy*, and use these concepts to decide if we have to prune a tree or not.

- Consider a subtree near the leaves
- Compute its maximum error e_l as a weighted sum of the maximum error of the leaves.
- Compute the maximum error e_r of the root of the subtree transformed into a leaf

Consider this subtree:



We have to leaves n_{121} and n_{122}.
If we prune prune this nodes, $n12$ becomes a leaf, which *will contain* $n121 + n122$ *elements*.
So, if:

$$(N_{121} + N_{122}) * e_{max_{12}} < N_{121} * e_{max_{121}} + N_{122} * e_{max_{122}})$$

Then we can *prune*.
This is because if we are in the described situation, we have a bigger number of example (N) in the node, so the uncertainty (e_{max}) is smaller.
In practice, this means that when I am in the nodes at the top, the uncertainty is smaller (since I have a big number of elements), but when I come to the bottom, I have less elements so the uncertainty increases.

Essentially, we there are tools that can tell us if we should prune a tree and how to do it. So let's consider a decision tree, and let's focus on the leaves.

What does it mean in practice?
That the bigger number of the tree is small. So, it maybe the case that if I prune part of the tree, the pessimistic error frequency is smaller.

Obviously, this doesn't work very well with a small dataset.

Accuracy of a classifier

The *error frequency* is the simplest indicator of the quality of a classifier. It is the sum of errors on any class divided by the number of tested records.
From now on, for simplicity, we will use the **empirical error frequencies** (f). Remember that in real cases the *maximum error frequencies* (e_{max}) should be used instead.

The hyperparameters

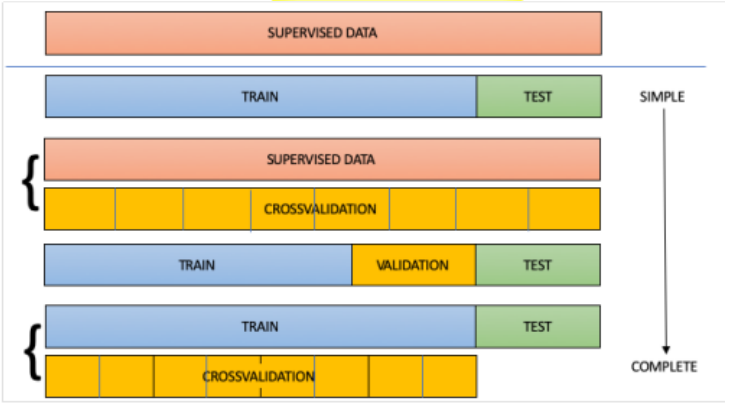
Every ML algorithm has them, and they are parameters that influence its behavior.

We need to do several tests/loops to find the best set of parameters.
This is not so trivial, since each test can take a long time.

Testing strategies

The optimization process (of finding the best hyperparameters) makes use of the test set.
Since I use the test set for both testing and optimizing, and it can take too much time to do, we have to use alternative strategies.

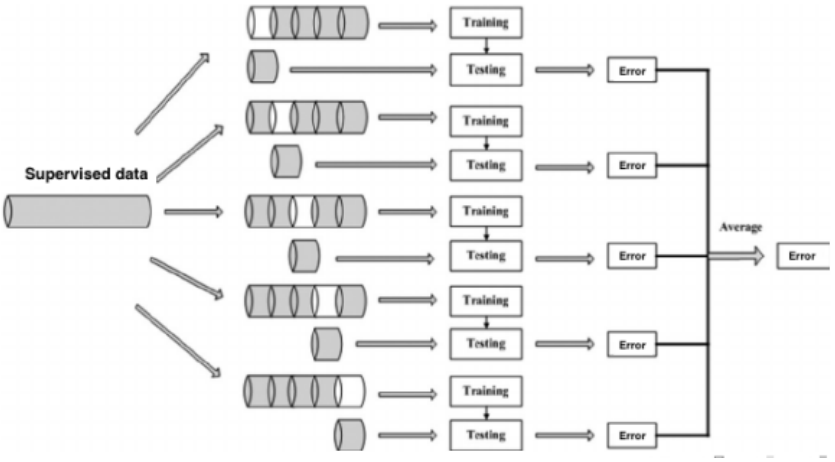
- **Holdout**: if I need to do a model validation, I split the data into *training set*, *validation set* and *test set*.
  - (validation is the process of testing if the model is overfit).
- **Cross validation**: it is the most effective solution.



Cross-validation

The *training set* is randomly *partitioned* into k subsets (if necessary, use stratified partitioning, meaning that each the proportion of classes is equal).

- I do k iterations, using one of the subsets for *test* and the others for *training*.
  - In this way, each record is used $k-1$ times for *training* and once for *testing*
- Optimal use of the supervised data.
- Typical value: $k = 10$



In the end I will make the final testing with the remaining data.

Here are the tradeoff of this method:

- The higher is the K, the less data I have for testing (and the *more iterations* I have to perform).

But it has a lot of advantages:

- The estimate of the performance is averaged on k runs ⇒ *more reliablity*
- All the examples are used once for testing
- The final model is obtained using all the examples ⇒ *best use of the examples*

So, what can I do is doing cross validation to find the best hyperparameters. And the test-set is considered fresh data,

so it will be more reliable.

Leave one out

This is an extreme case of cross validation, in which $k = N$ (so no partitioning). It can be uses

In short

Our objectives are:

- Find the *best hyperparameters* setting
- Give a *reliable estimation* of the *run-time performance*, i.e. in regards to the expected ability to classify new data (overestimation of the expected performance should by avoided).

Binary predictions

A *binary prediction* is, for example, a case in which we have to predict if a certain attribute is true or false. In this cases, a bad prediction results in either a false positive or a false negative.
Let's consider a table like this one:

		Predicted class	
		P	N
True class	P	TP	FN
	N	FP	TN

TP stands for True Positive, and FN stands for false negative, etc...
This matrix/table is then extended and called a **confusion matrix**, with a *frequency* (i.e. a number of FPs) *associated to each predicted class*. You can see this clearly in the "Beyond accuracy" paragraph below.

We can compute the accuracy as $$
\dfrac{TP + TN}{N_{tests}}

Other performance indicators Is the accuracy the only performance indicator for a classifier? No, we have also: - Velocity - Robustness w

F1_score = 2\dfrac{prec \cdot rec}{prec + rec}

It is a value between 0 and 1. It increases when precision and recall are __balanced__. It is usually interesting when we want to find the best c

k = \dfrac{Pr(c) - Pr(r)}{1- Pr(r)}

with $-1 \le k \le 1$. Some rules of thumb: ![[K_rule_thumbs.png]] ## Cost of errors Our decisions are driven by predictions. Bad predictions

2022-11-02 - Lift chart, Naive Bayes, Laplace smoothing (Classification II & III)

Classifier II - From Probabilistic to CRISP

Lift chart

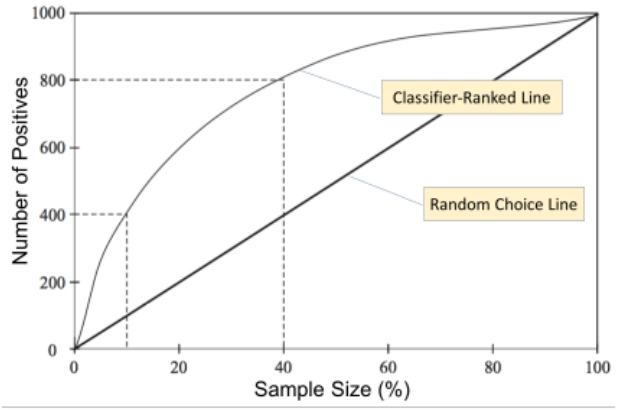
The lift chart can be used to extract CRISP values from a probabilistic result. Essentially, it can help us evaluate and *extract* the examples that are *the most promising* from a dataset.

It is obtained by *sorting* all the examples in order of *decreasing probability of being true*, if cons me can forecast how many of the true values we can find.
It measures the capabilities how able is our classifier to "put data in the beginning of the chart" the example that are positive. So, the more curve we have in the beginning, the more effective is our classifier.

In short, the **lift chart** essentially tell us how able is our model to correctly classify objects as positive.
We have two lines:

- The *straight line* plots the *number of positives* obtained with a *random choice* of a sample of test data.

- The *curve* plots the *number of positives* obtained drawing a fraction of test data with decreasing probability.



The larger the area between the two lines, the better the classification model (The more the curvature is high on the left, the better is our classifier).

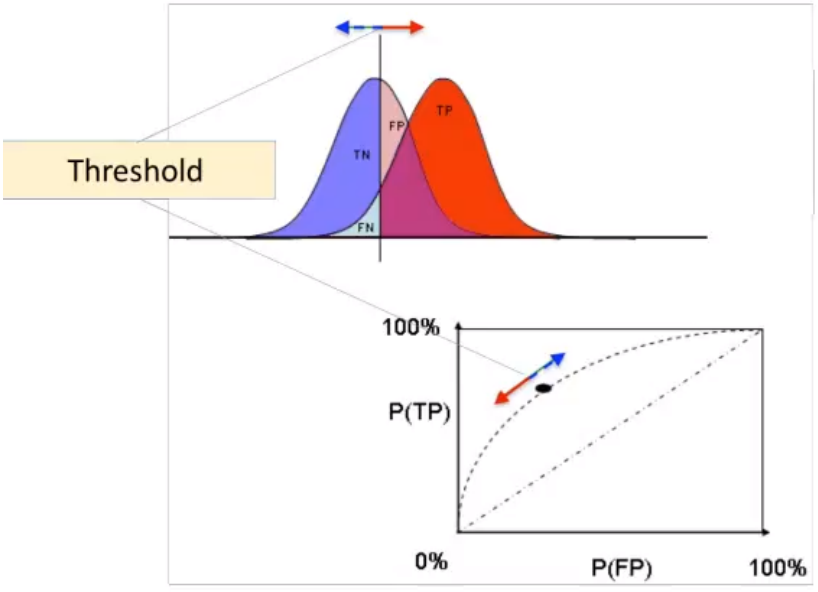
The lift chart is especially useful when we have to work in batches.

ROC Curve

Keep in mind: Soft classifier = probabilistic classifier.

Let's consider that we have a lot of data, and it has some noise. Some of this noise alters our data, and we have to distinguish the data to extract knowledge from it. In particular, we have to distinguish between hit rate and false alarm in a noisy channel, and so the noise alters the two levels according to a *Gaussian distribution*.
We could impose a threshold to the values, but we would have some FP and FN.

With less noise, the two Gaussian curves are better separated.



Moving the threshold towards right increases both the rate of true positives and false positives caught.

(N.B.: The curve on the bottom right of the image is not a lift chart. The values on the 2 axis are different (Probability of a TP on the y, Probability of a FP on the x)).

What we could do is imposing multiple thresholds, using **threshold steps**. Varying the threshold the behavior of the classifier changes, by changing the ratio of TP and FP.

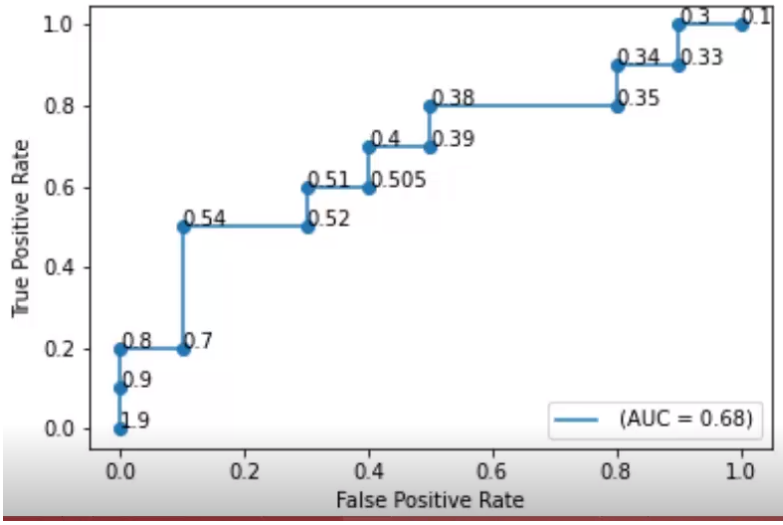
So, the soft classifier (= probabilistic classifier) can be converted into a crisp one by choosing thresholds, in a ROC curve *the quality of the classifier* is summarized by the **Area Under Curve** (AUC) (the bigger, the better).
Here's how to choose the thresholds:

```
sort the test elements by decreasing positive probability
set the threshold to the highest probability, set TP and FP to zero

repeat
```

```
        - update the number of TP and FP with probability from the threshold to 1
        - draw a point in the curve
        - move to next top probability of positive en
end repeat
```

Here's an example of a ROC curve:

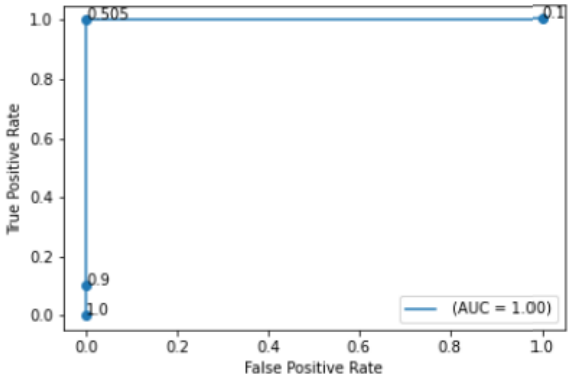


inst	*class*	*score*	*tp*	*fp*
1	p	0.9	1	0
2	p	0.8	2	0
3	n	0.7	2	1
4	p	0.6	3	1
5	p	0.55	4	1
6	p	0.54	5	1
7	n	0.53	5	2
8	n	0.52	5	3
9	p	0.51	6	3
10	n	0.505	6	4
11	p	0.4	7	4
12	n	0.39	7	5
13	p	0.38	8	5
14	n	0.37	8	6
15	n	0.36	8	7
16	n	0.35	8	8
17	p	0.34	9	8
18	n	0.33	9	9
19	p	0.3	10	9
20	n	0.1	10	10

The best soft classifier should give the higher score to all the positives, so that the shape of the curve catches all the positives at the beginning (and looks like a rectangular triangle almost).

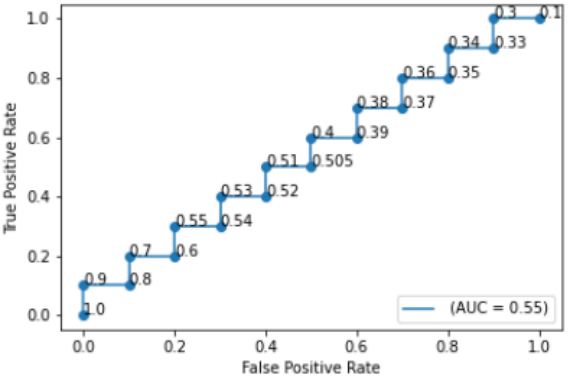
Here's an example of a perfect selection:

item#	*class*	*score*
1	p	0.90
2	p	0.80
3	p	0.70
4	p	0.60
5	p	0.55
6	p	0.54
7	p	0.53
8	p	0.52
9	p	0.51
10	p	0.51
11	n	0.40
12	n	0.39
13	n	0.38
14	n	0.37
15	n	0.36
16	n	0.35
17	n	0.34
18	n	0.33
19	n	0.30
20	n	0.10



Here's an example of a random selection:

item#	class	score
1	p	0.90
2	n	0.80
3	p	0.70
4	n	0.60
5	p	0.55
6	n	0.54
7	p	0.53
8	n	0.52
9	p	0.51
10	n	0.51
11	p	0.40
12	n	0.39
13	p	0.38
14	n	0.37
15	p	0.36
16	n	0.35
17	p	0.34
18	n	0.33
19	p	0.30
20	n	0.10



What is the best method for classification? (Classification III)

Spoiler: it depends on the data.

Naive Bayes Classifier

It is Naive, and based on the Bayes theorem:

$$Pr(d_1 = v_1, d_2 = v_2 | c = c_x) = Pr(d_1 = v_1 | c = c_x) \cdot Pr(d_2 = v_2 | c = c_x)$$

It considers the contribution of all the attributes together (unlike DT which considers one attribute at a time).
We *assume* that each attribute is **independent** from each other (i.e. height and weight are not independent, but color of hair and weight is).

Here's an example.
Given this data:

Outlook			Temperature			Humidity			Windy			Play	
	yes	no		yes	no		yes	no		yes	no	yes	no
sunny	2	3	hot	2	2	high	3	4	false	6	2	9	5
overcast	4	0	mild	4	2	normal	6	1	true	3	3		
rainy	3	2	cool	3	1								
sunny	2/9	3/5	hot	2/9	2/5	high	3/9	4/5	false	6/9	2/5	9/14	5/14
overcast	4/9	0/5	mild	4/9	2/5	normal	6/9	1/5	true	3/9	3/5		
rainy	3/9	2/5	cool	3/9	1/5								

We will have:

Outlook: sunny, Temperature: cool, Humidity: high, Windy: true, Play: ?

- Treat the five features and the overall likelihood that *play* is *yes* or *no* as equally important
  - they are independent pieces of evidence, the overall likelihood is obtained by multiplying the probabilities (i.e. the frequencies)

$$\text{likelyhood of } yes = 2/9 \times 3/9 \times 3/9 \times 3/9 \times 9/14 = 0.0053$$
$$\text{likelyhood of } no = 3/5 \times 1/5 \times 4/5 \times 3/5 \times 5/14 = 0.0206$$

- Normalize to 1

$$\mathbf{Pr}(yes) = \frac{0.0053}{0.0053 + 0.0206} = 20.5\% \qquad \mathbf{Pr}(no) = \frac{0.0206}{0.0053 + 0.0206} = 79.5\%$$

- *no* is more likely than *yes*, about four times

(N.B.: we refer to likelihood of yes instead of probability of yes since the values are not normalized, meaning that sum of

the likelihood values is not 1).

Each likelihood is calculated by using the probability of each possible event condition.

Bayes theorem

This theorem states that, given a hypothesis H and an evidence E that bears on that hypothesis:

$$Pr(H|E) = \frac{Pr(E|H)\ Pr(H)}{Pr(E)}$$

(In the previous example, the H would be $play?$, while the E would be all the values of the attributes).
Usually, the hypothesis is the class, say c, the evidence is the tuple of values of the element to be classified.

We can split the evidence into pieces, one per attribute, and, if the attributes are independent inside each class, we can state that:

$$Pr(c|E) = \frac{Pr(E_1|c)\ Pr(E_2|c)\ Pr(E_3|c)\ Pr(c)}{Pr(E)}$$

We can see E as a *joint event*, where $E_1 \dots E_3$ are (some) of the attributes that compose the event.

So:

- we compute the conditional probabilities from examples
- Apply the theorem

Notice that the denominator is the same for all the classes, and is eliminated by the normalization step.

The Naive Bayes method is called *naive* since the assumption of independence between attributes is quite simplistic. Nevertheless, it works quite well in many cases.

Drawbacks

What if value v of attribute d never appears in the elements of class c?
In this case we have that $Pr(d = v|c) = 0$. This makes the probability of the class for that evidence drop to zero, and we would need to overcast it, which we don't want.

We need an alternative solution.

Laplace smoothing

The basic idea of this other classifier is to *smooth* the probability.
Let's start ignoring the details of the dataset, we consider only the value domains, and we know that for a given attribute d there are V_d distinct values.

Then, *a simple guess* for the frequency of each distinct value of d in each class is $1/V_d$.

We can smooth the computation of the posterior probabilities of values inside a class balancing it with the prior probability.

The parameters are:

- α , the *smoothing* parameter, typical value is 1.
- $af_{d=vi,c}$, the *absolute frequency* of value v_i in attribute d over class c.
- V_d , the *number of distinct values in attribute* d over the dataset.
- af_c : *absolute frequency* of class c in the dataset.

We could also consider:

- the a-priori frequency, in which we don't consider the class.
- the a-posteriori frequency in which we consider the class.

The final **smoothed frequency** is:

$$sf_{d=v_i,c} = \frac{af_{d=v_i,c} + \alpha}{af_c + \alpha V_d}$$

- If $\alpha = 0$, we obtain the standard, *unsmoothed* formula (of the Bayes classifier).

- If $\alpha \to \infty$, then we would have that the frequency is $1/V$, in which we don't consider the class at all. In this case we consider the frequency of each value as equal, without any reference to a class.
- if $af_{d=v_i,c} = 0$, then the $sf_{d=v_i,c}$ considers only the frequency of the class (and thus, takes in account, in someway, the different values of d).

The bigger is α, the more importance we give to the prior probabilities (without considering the class).

In this case, the missing values do not affect the model, and so it is not necessary to discard an instance with missing values (i.e., for decision trees, we cannot deal with missing values. In Naive-Bayes, if the value is missing (so, not 0!) we simply disregard the attribute).

Normally, we simply discard that attribute with the missing values (in both train and test instances), as if it was not present. The descriptive statistics are based on Known, non-null values.

- In test instances, it results in the likelihood will be higher for all the classes, but this is compensated by the normalization.

Numeric values

With discrete values, frequencies are easy to count, and it is not so easy with real values.
Since the method is probabilistic, we assume a *Gaussian distribution of the values*.
From our data, we can easily compute the mean (μ) and the variance (σ) inside each class.

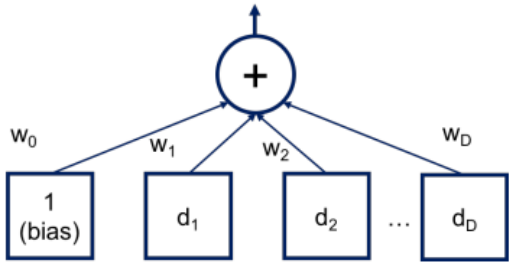
$$f(x) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

- Probability and probability density are closely related, but are not the same thing.
  - On a continuous domain, the probability of a variable assuming exactly a single real value is zero
  - A single value in the density function is the probability that the variable lies in a small interval around that value.
  - The value we use are, of course, rounded at some precision factor, and since that precision factor is the same for all the classes, then we can disregard it
- If numeric values are missing, mean and standard deviation are based only on the values that are present

While these solutions provide excellent results in many cases, we would have a dramatic degradation if the simplistic conditions are not met.

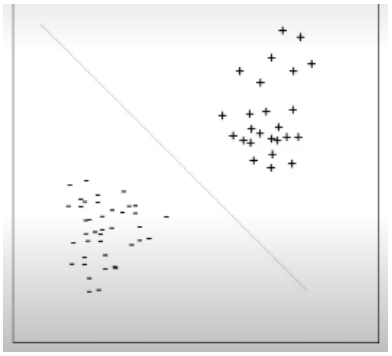
- *Violation of independence* – for instance, if an attribute is simply a copy of another (or a linear transformation), the weight of that particular feature is enforced (something like squaring the probability)
- *Violation of Gaussian distribution* – use the standard probability estimation for the appropriate distribution, if known, or use estimation procedures, such as Kernel Density Estimation.

Linear classification with Perceptron



A **perceptron** is also called an *artificial neuron*. In practice, it's a linear combination of weighted inputs.
The values in the boxes $d_1 \dots d_D$ represents attributes, while the added value of 1 with w_0 associated to it is the bias.
The output is the weighted sum of these inputs.

For a dataset with numeric attributes, we need to find (or learn) an *hyperplane* (a straight line) such that all positives lay on one side and all the negatives on the other, and are separated.



As we know from classification, the task is: how can we learn this hyperplane from data?

The hyperplane formula

The hyperplane is described by a set of weights $w_0, \ldots, w_D$ in a linear equation on the data attributes $x_0, \ldots, x_D$.
The fictitious attribute $x_0 = 1$ is added to allow a hyperplane that does not pass through the origin (and it is like the bias of a perceptron).
There are either none or infinite such hyperplanes. Let's assume there's a solution in this case though:

$$w_0 * x_0 + w_1 * x_1 + \ldots + w_D * x_D \quad \begin{cases} > 0 \Rightarrow \text{positive} \\ < 0 \Rightarrow \text{negative} \end{cases}$$

Learning the hyperplane

```
set all weights to zero
        while there are examples incorrectly classified do
                for each training instance x do
                        if x is incorrectly classified then // we need to change weights
                                if class of x is positive then//this if is the core of the method
                                        add the x data vector to the vector of weights // weight ch.
                                else
                                        subtract the x data vector from the vector of weights
```

How can we prove that this "sum" in the weights is an actual improvement over the prior set of weights?

Linear perceptron convergence

Each change of weights *moves the hyperplane towards the misclassified instance*, consider the equation after the weight change for a positive instance x which was classified as negative:

$$(x_0 + w_0) * x_0, \ldots, (x_D + w_D) * x_D \$\$The result is increase by a positive amount:$$

x_0^2 + \dots + x_D^2

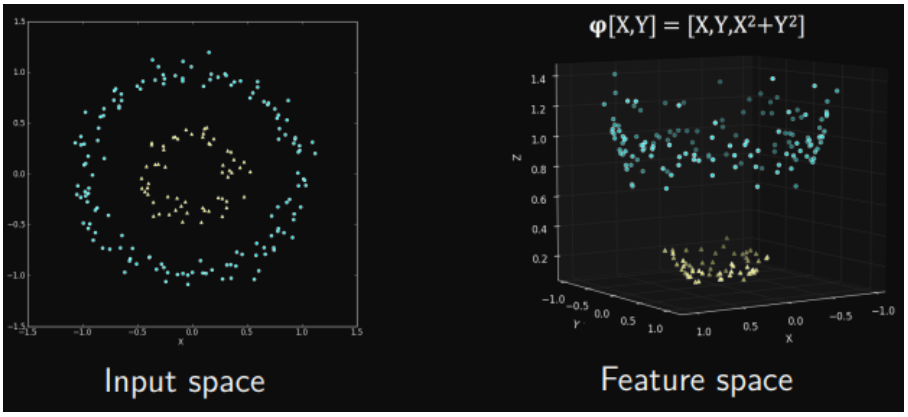
Therefore, ==the result will be less negative or, possibly, even positive==. It is analogous for a negative instance which was classified as posit

2022-11-04 - SVMs, Neural Networks (Classification III)

Non-linear classifiers

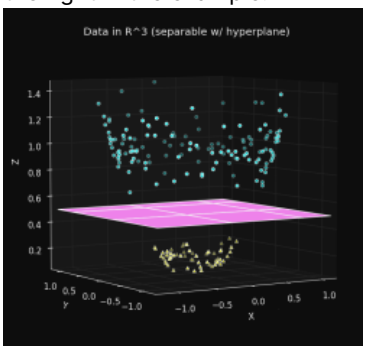
Non-linear class boundaries in SVM

What if we have a situation with a non-linear distribution of data?



If we have a situation like the one in the figure in the left, we can see that the there is no parameter for which an hyperplane can separate these two classes.

In this case, the *nonlinearity of boundaries* can be overcome with a **non–linear mapping**.
So, what we can do imagine another different space, called the **feature space**, in which a dimension is added. This space is calculated using a mathematical transformation, but in general it is done in a clever way, like in the image on the right in the example.



Now, our data is linearly separable by adding a plane.

This method is compatible with only a family of function (**kernel functions**).

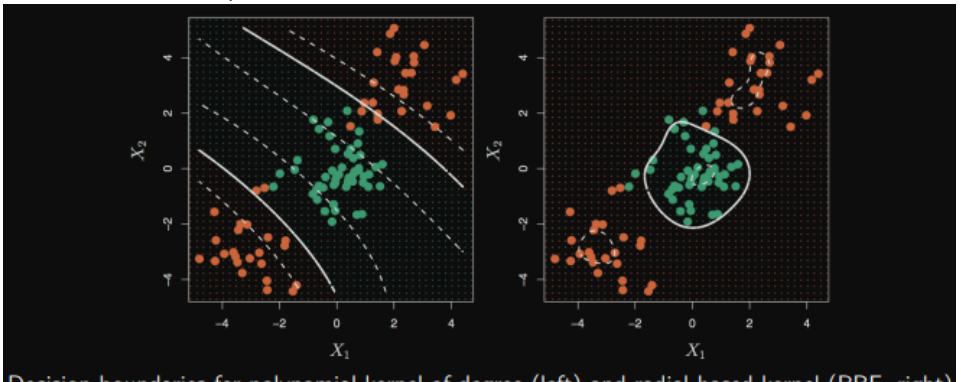
The kernel trick

- The separating hyperplane computation requires a series of *dot product* computations among the training data vectors.
- Defining the mapping on the basis of a particular family of functions, called *kernel functions*, or simply *kernels*, the mapping does not need to be explicitly computed, and the computation is done in the input space.
- This avoids an increase in the complexity.

Some kernel functions:

linear	$\langle x, x' \rangle$
polynomial	$(\gamma \langle x, x' \rangle + r)^{dg}$
rbf	$\exp(-\gamma \|x - x'\|^2)$
sigmoid	$\tanh(\langle x, x' \rangle + r)$

Here are some examples:



Decision boundaries for polynomial kernel of degree (left) and radial based kernel (RBF, right)
In this case they seem to be both appropriate, but the generalization capabilities change, depending on where new data could be expected

SVM complexity

- The time complexity is mainly influenced by the efficiency of the optimization library.
- `libSVM` library scales from $O(D \cdot N^2)$ to $O(D \cdot N^3)$, depending on the effectiveness of *data caching* in the library, which is data dependent.

SVM final remarks

- *Learning* is in general *slower* than simpler methods, such as decision trees
- *Tuning is necessary* to set the parameters (not discussed here)
- The results can be very accurate, because subtle and complex decision boundaries can be obtained
- Are not affected by local minima
- Do not suffer from the curse of dimensionality: do not use any notion of distance (we will see later).
- SVMs do not directly provide probability estimates, these can be calculated using rather expensive estimation techniques.
  - nevertheless, SVM can produce a confidence score related to the distance of an example from the separation hyperplane.

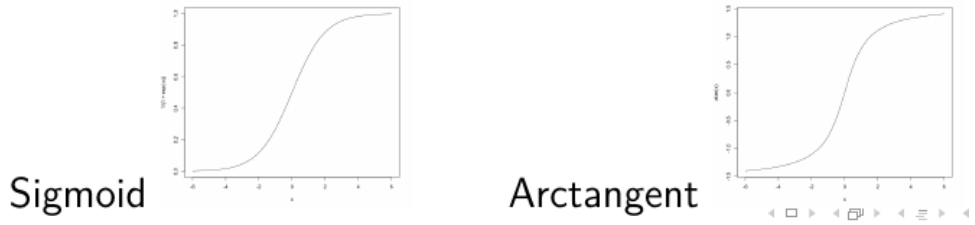
Neural networks

Arrange many perceptron–like elements in a hierarchical structure, to overcome the limit of *linear decision boundary*.

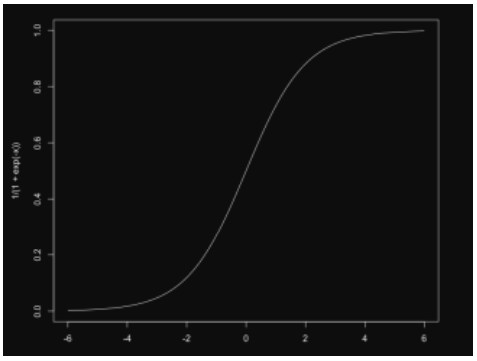
A neuron is a *signal processor* with *threshold*. Meaning that we have an output only if our input reaches a threshold. Signal transmission from one neuron to another is weighted, and *weights change over time*, also due to learning.

The signals transmitted are modeled as real numbers. The threshold of the biological system is modeled as *a mathematic function*. If the function is continuous and differentiable, the mathematics is much easier (since the derivative can be expressed in terms of the function itself).

There are several functions available which capture this behavior, such as a sigmoid function (or an $arctan$).



Sigmoid function



It is called *squashing function*, since it "squashes" the values into [0, 1].
It Is continuous, differentiable, non–linear. And has this formula:

$$\frac{1}{1+e^{-x}}$$

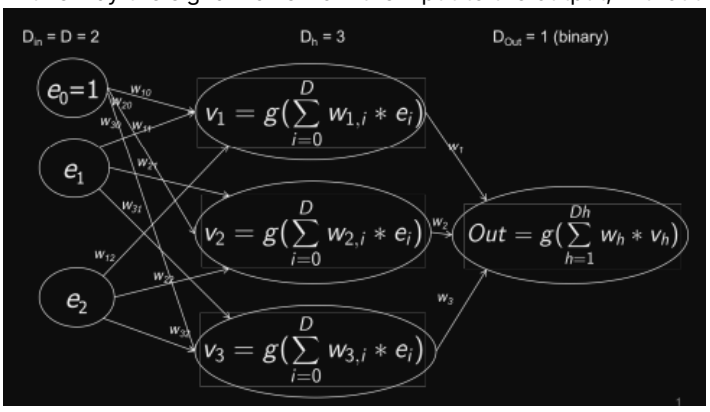
The fact that it is non-linear is pretty important.
A perceptron is linear, so the results is non satisfactory. This is because in a linear system, we have that $f(x_1 + x_2) = f(x_1) + f(x_2)$, so if x_2 presents some noise, it is completely transferred to the output. In general, this is not true in a non-linear system.

The shape of the function *can influence* the learning speed.

Feed–forward multi–layered network

- Inputs feed an *input layer*, one input node for each dimension in the training set.
- Input layer feeds (with weights) a *hidden layer*.
- Hidden layer feeds (with weights) an *output layer*.
  - the number of nodes in the hidden layer are a parameter of the network
  - the number of nodes in the output layer is related to number of different classes in the domain
    - one node if there are two classes
    - one node per class in the other cases

In this way the signal flows from the input to the output, without loops.



Just like in perceptron, we add a *bias* ($e_0 = 1$ in this case) so that if the input is 0, the output is not 0.
e_1 and e_2 are the 2 variables.
In each of the node of the hidden layer, we have a *transfer function* (i.e. the sigmoid), which is represented as g.

Since we only have one output, we have a binary classifier in this case (so it can recognize between 2 classes).
If we want to distinguish 4 classes, we need to add an output neuron.
We could decide instead to have 4 output nodes for 4 classes (difference between encoding or nodes for each possible output).

If we add a node to an hidden layer into the network, we would have more parameters, and thus more *separation*. The more complex the hidden layer → the more power to distinguish the classes.

Training a NN

The algorithm for training neural networks is similar to something like this:

```
set all weights to random values
while termination condition is not satisfied do
        for each training instance x do
                feed the network with x and compute the output nn(x)
                compute the weight corrections for nn(x) - xOut // output - desired out
                propagate back the weight corrections
```

The distance between the desired output and the actual output that we obtain is **propagated back**, considering the transfer function.

Possible termination conditions are:

- checking the quality measures.
  - checking the f-measure.
  - checking the accuracy.
- using a maximum number of iterations.
- We should also evaluate the change of the weights: if it is small, then we stop the iteration.

Each training loop is called an *epoch*. Sometimes, an epoch can require minutes or hours, so we should terminate the loop in this case.

Remarks on NNs Training

- In NNs, *the weights encode the knowledge* given by the supervised examples. The encoding is *not easily understandable:* it looks like a structured set of real numbers.
- Convergence is not guaranteed.

Computing the error

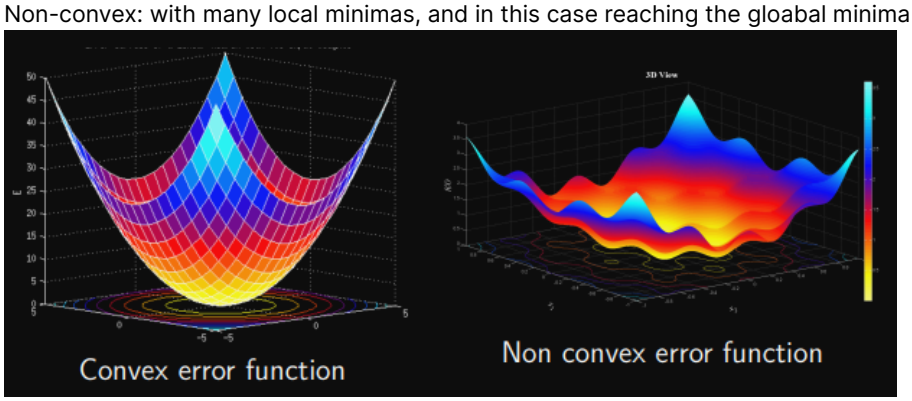
- Let x and y be the *input vector* and the *desired output* of a node, respectively.
- Let w be the *input weight vector* of a node.
- The error is:

$$E(w) = \frac{1}{2}(y - Transfer(w, x))^2$$

($Transfer()$ is *transfer function*).

Depending on the data, my error function can be:

- Convex: with a single global minima
- Non-convex: with many local minimas, and in this case reaching the gloabal minima is not so easy.



To move towards the local minimum of the error function, we need to *compute the gradient*, and follow it.
To compute the gradient, we need to compute the derivates of the error function w.r.t. the weights.

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

$$\frac{d}{dx}\text{sgm}(x) = \frac{e^{-x}}{(1 + e^{-x})^2} = (1 - \text{sgm}(x))\text{sgm}(x)$$

We can see that the derivative of the sigmoid is computed using the function itself, and as we've said, this eases the computation.
By computing the derivatives in respect to the weights, we know in which direction to move.

The weight is changed subtracting the partial derivative multiplied by a **learning rate** constant.

- The learning rate influences the convergence speed and can be adjusted as a *tradeoff between speed and precision*.
- The subtraction moves towards smaller errors.
- The derivatives of the input weights of the nodes of a layer can be computed if the derivatives for the following layer are known.

The algorithm to train neural networks could be rewritten in this way:

```
set all weights to random values
        while termination condition is not satisfied do
                for each training instance x do
                        1 - feed the network with x and compute the output nn(x)
                        2 - compute error prediction at output layer nn(x) - xOut
                        3 - compute derivatives and weight corrections for output layer
                        4 - compute derivatives and weight corrections for hidden layer
```

Steps 1 and 2 are *forward* (computation), steps 3 and 4 are *backward* (adjustment).

- computation → adjustment, computation → adjustment, ...

Learning modes

There are several variants for learning modes in machine learning.

Stochastic

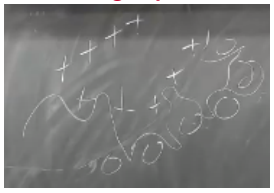
- each forward propagation *is immediately followed* by a weight update (as in the algorithm of previous slide).
- *introduces some noise* in the gradient descent process, since the gradient is computed from a single data point. It can potentially slow the learning process.
- reduces the chance of getting stuck in a local minimum.
- good for online learning (meaning that I want to adjust my network while the data is arriving).

Batch

- *many propagations* occur before updating the weights, *accumulating errors* over the samples within a batch
- generally yields faster and stable descent towards the local minimum, since the update is performed in the direction of the average error

Some remarks

- A learning round over all the samples of the network is called epoch
- In general, after each epoch *the network classification capability will be improved*
- Several epochs will be necessary
- After each epoch the starting weights will be different
- **Overfitting is possible** w.r.t. the complexity of the decision problem.

  A situation like the one seen in this picture will not generalize well.

Design choices

Here some design choices we can make during the design process of a NN.

- The structure of input and output layers *is determined by the domain* (the training set), i.e. the number of columns.
- The number of nodes in the hidden layer can be changed, or I can have multiple hidden layers.
- The learning rate can be changed in different epochs
  - in the beginning a higher learning rate can push faster towards the desired direction
  - in later epochs a lower learning rate can push more precisely towards a minimum

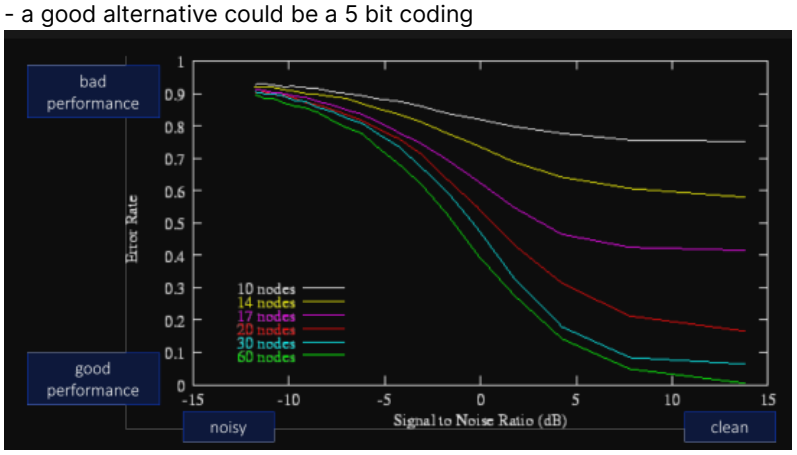
Regularization

As we've said, overfitting is possible, but we can use **regularization** to improve the generalization capabilities of the model.
What we do is modify the performance function, which is normally chosen to be the sum of squares of the network errors on the training set.
Improvement of performance is obtained by *reducing a loss function* (i.e. the sum of squared errors). Regularisation corrects the loss function in order to smooth the fitting to the amount of data. Obviously, regularisation must be tuned.

NNs application example

- Recognition of characters with noise
- The images are 7×5 arrays of floating points (before noise -1 for black, 1 for white)
- 35 input nodes
- up to 60 hidden nodes

- 26 output nodes
  - only one at a time should give a value near 1, all the others should be near 0
  - a good alternative could be a 5 bit coding



  With a noisy signal, the performance of the model is bad in every case.
  (A good rule of thumb: you should a number of nodes that is almost double the nodes of the output).

What we learn from this is that NNs behave very well in presence of noise.

K-nearest neighbors classifier (or KN).

While the other methods encode the knowledge in a model (i.e. DT or NNs), this method keeps all the training data (i.e. the model is the entire training set).
It is the basis of clustering.

- Future predictions by computing the similarity between the new sample and each training instance (by using for example any similarity function)
- picks the K entries in the database which are closest to the new data point.
- majority vote
- main parameters:
  - the number of neighbors to check
  - the metric used to compute the distances (the Mahalanobis distance has good performance)

Essentially, I go on in checking each neighbor, and by checking the nearest neighbor to each instance, I decide the class of the new individual.

From binary classifier to multiclass classification.

As we've seen, several classifiers generate a binary classification model.
There are two ways to deal with multi–class classification:

- transform the training algorithm and the model, sometimes at the expenses of an increased *size* of the problem (not important)
- *use a set of binary classifiers* and *combine the results*, at the expenses of an increased *number* of problems to solve.
  - one-vs-one and one-vs-rest strategies

One-vs-one strategy (OVO)

Consider that we've 3 classes, for example A1, A2, A3. We would need to build a classifier that distinguishes between (A1,A2), (A2,A3) and (A3,A1).

So, consider all the possible pairs of classes and generate a *binary classifier for each pair*.
(we'll have $C * (C-1)/2$ pairs).
Each binary problem considers only the examples of the two selected classes.
At prediction time *apply a voting scheme*:

- an unseen example is submitted to the $C * (C-1)/2$ binary classifiers, each winning class receives a $+1$
- the class with the highest number of $+1$ wins

One-vs-rest strategy (OVR)

Consider C binary problems where class c is a positive example, and all the others are negatives.

- build C binary classifiers, and at prediction time *apply a voting scheme*
  - an unseen example is submitted to the C binary classifiers, obtaining a confidence score
  - the confidences are combined and the class with the highest global score is chosen

In other words, it's like if we have 3 classifiers,

1. A1 vs !A1
2. A2 vs !A2
3. A3 vs !A3

OVR vs OVO

- OVO requires *solving a higher number of problems*, even if they are of smaller size.
- OVR tends to be *intrinsically unbalanced*. If the classes are evenly distributed in the examples, each classifier has a proportion positive to negative 1 to C - 1.

Here's a small, not really important, deviation to the normal lesson.

Deep learning - a small introduction

Deep learning deals with deep networks, which are neural networks with a large number of hidden layers. Normal NNs are called Shallow Networks.

The idea of neural networks is very old (80s), but at the time computational power was not enough to compute deep networks, but also there were numerical problems (like stability etc..).

With the advent of GPUs, which are very complex parallel computing units, we could finally deal with millions of parameters.

Deep networks need more parameters, so also they need large dataset. This means that while deep learning is a very nice concept, it isn't always the right method to apply.

We could say that deep learning is an evolution of machine learning.

2022-11-09 - Ensemble methods, Regression

Ensemble methods

Ensemble methods (aka Classifier combination) is based on the idea that instead of using a single classifier, we can use several classifiers that work together.
So, it is based on training a set of **base classifiers.**
The final prediction is obtained *taking the votes of the base classifiers*.
In general, ensemble methods tend to perform better than a single classifier.

Why should this be better?

Let us consider 25 binary classifiers, each of which has error rate $e = 0.35$.

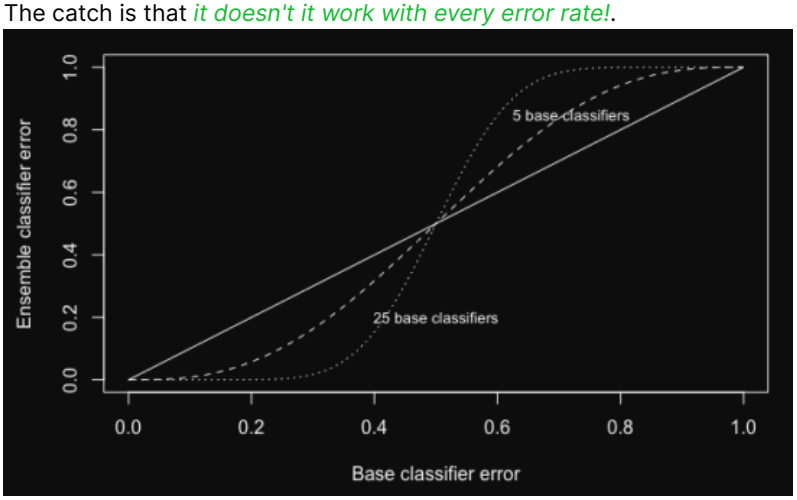
The ensemble classifier output is the majority of the predictions (13 or more).

- If the base classifiers are equal, the ensemble error rate is still e (so this operation is useless)
- If the base classifiers *are completely independent* and have all error rate e, i.e. their errors are uncorrelated, then *the **ensemble** will be wrong only when the **majority** of the base classifier is wrong*.

$$e_{ensemble} = \sum_{i=13}^{25} \binom{25}{i} e^i (1-e)^{25-i} = 0.06$$

So, we need to combine all the errors of the classifiers. And what we obtain is that the errors of the ensemble is much smaller (0.06) than the error of each classifier (0.35).
[Avengers, ensemble!!!]

The catch is that *it doesn't it work with every error rate!*.



If the base error rate is 0.5, the resulting error rate is still 0.5. So, it works only if the base error rate is $e \leq 0.5$.

In general, ensemble methods are useful if:

1. the base classifiers are independent
2. the performance of the base classifier is better than random choice (0.5)
    - If we're near random choice (i.e. $e = 0.45$) then it won't help, but if we're at $e = 0.35$ we'll get a good performance.

Creating independence

But how can we make classifiers so that they're independent? How can we make 2 completely different decision tree?

By manipulating the training set

With a *manipulation of the training* set we can obtain *some independence*.
There are 2 major methods for dataset manipulation:

- **Bagging**: we *resample the training set repeatedly*, with replacement (in the training set) according to a uniform probability distribution.
    - i.e. 3 different subsets of the training set to 3 classifiers.
    - There's some independence, but not total independence.
    - *Sampling with replacement* means that a value is reinserted after it has been extracted.
- **Boosting**: iteratively changes the distribution of training examples so that the base classifier focus on examples which are hard to classify.
    - I train my classifier and I predict the examples, some of them will be correctly predicted, the others won't.
    - I will increase the probability of being chosen for the bad predictions. In this way, I will force my classifier to focus on the examples that are badly predicted.
- Adaboost: the importance of each base classifier depends on its error rate.

Boosting and Bagging manipulate the training set, Adaboost manipulates the combination of the final results.

By manipulating the input features

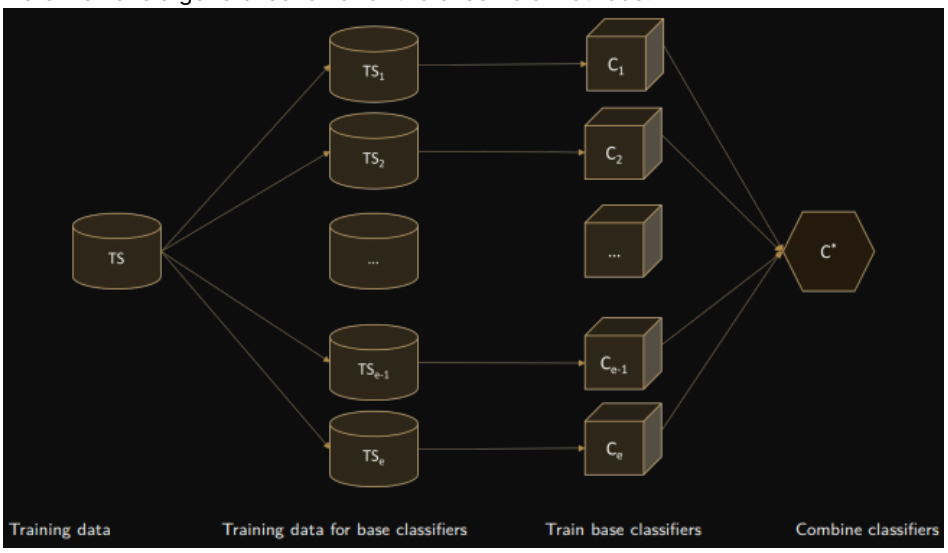
Subset of input features can be chosen either random or according to suggestions from domain experts. Meaning, we use different *subset of attributes* for each classifier.
Random forest: uses *decision trees* as *base classifiers*. It frequently produces very good results.

This is effective only if we have a large number of features/attributes.

By manipulating the class labels

When we have a large number of class labels, for each base classifier we randomly partition the class labels into 2 subsets, say A1, A2 and re-label the dataset.
We then train binary classifiers with respect to the two classes.
At testing time, when a subset is selected, all the classes that it includes will receive a vote.
As always, the class with the top score will win.

Here we have a general schema for the ensemble methods.



Forest of randomized trees

A diversified set of classifiers is created by introducing *randomness* in the classifier construction.
Each tree in the ensemble is *built from a sample drawn with replacement* (i.e., a bootstrap sample) from the training set.
Furthermore, when splitting each node during the construction of a tree, the best split is found either from all input features or a *random subset* of size `max_features`.

Bias–vs–Variance tradeoff

- **Bias** is the *simplifying assumptions* made by the model to make the *target function easier to approximate*
  - If a model has a bias, usually it means that it is a very simplified model
- **Variance** is the *amount that the estimate of the target function will change*, given different training data
- Bias-variance trade-off is the sweet spot where our machine model performs between the errors introduced by the bias and the variance

"If Bias vs Variance was the act of reading, it could be like Skimming a Text vs Memorizing a Text"

The purpose of the two sources of randomness is to *decrease the variance* of the forest estimator.

- individual decision trees typically exhibit high variance and tend to overfit
- the *injected randomness* in forests generates decision trees with somewhat decoupled prediction errors.
- by taking an average of those predictions, some errors can cancel out
- random forests achieve a reduced variance by combining diverse trees, sometimes at the cost of a slight increase in bias.
- In practice the variance reduction is often significant hence yielding an overall better model.

Boosting

In some cases, we can give weights to classes, or *to each training instance* (so that, for example, an error in a certain instance should be more penalizing).
Different classifiers are trained with those weights, and *the weights are modified iteratively according to classifier performance*.

- For example, I increase the weights of the individuals which are badly classified, so that it the next round the model will focus on them.

The idea is "change the weights in order to compensate for the errors".

AdaBoost

- fit a sequence of *weak learners* on *repeatedly modified versions of the data.*

- the *predictions* from all of them are then *combined* through a *weighted majority vote* (or sum) to produce the final prediction.

    - Majority vote: the prediction given by the hightest number of classifiers wins.

- the data modifications at each so-called **boosting iteration** consist of *applying and modifying weights* to each of the training sample.

- initially, the weights $w_1, w_2, \ldots, w_N$ are all set to $1/N$, so that the first step simply trains a weak learner on the original data.

- for each successive iteration, the *sample weights* are *individually modified* and the learning algorithm is reapplied to the *reweighted data*.
    - at a given iteration, the training examples that were incorrectly predicted by the boosted model induced at the previous step have *their weights increased*, whereas the *weights are decreased* for those that were *predicted correctly*.

- as iterations proceed, *examples that are difficult to predict receive ever-increasing influence*; each subsequent weak learner is thereby forced to concentrate on the examples that are missed by the previous ones in the sequence.

Regression

Regression is a *supervised* activity, and the workflow for regression is similar to the workflow of classification.

- The target variable is numeric
- Our objective is to *minimize the error* of the prediction with *respect to the target*.
    - We can't just count the wrong predictions like in classification.

While classification considers as a target a discrete value (i.e. an int), most preferably with a small number of different possible values, in regression the *target value is a ccontinuous value*, a real number.

The process of training a regressor can be completely different from training a classifier, but some of the algorithm for creating classification models can be modified to work with regression.

Linear regression

The starting point of regression is **linear regression**.

- Consider a dataset X with N rows and D columns.

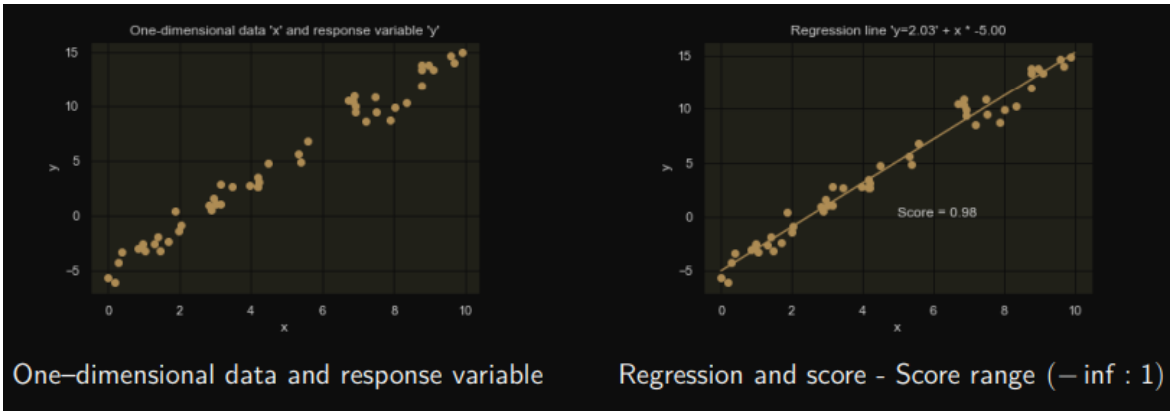
- x_i is a D dimensional data element

- Consider then a response vector $\hat{y}$ with N values y_i.

- w is a D-dimensional vector of **coefficients** that *needs to be learned* (the coefficients of the linear regression).

- we model the dependence of each response value y_i from the corresponding independent variables x_i as $y_i \approx w^T \cdot x_i \quad \forall i \in [1 \ldots N]$

- such that the *error of modelling* is *minimised*.



We see that the easiest approximation of the data in the image is a line.
The *score* is obtained by computing the difference between each real value with its corresponding prediction, so it can formalized through an objective function that we want to minimize.

Objective function in regression

Our objective function that we want to minimize is:

$$\mathcal{O} = \sum_{i=1}^{N}(w^T \cdot x_i - y_i)^2 = ||Xw^T - y||^2$$
$$= (Xw^T - y)^T.(Xw^T - y)$$

- Gradient of O with respect to w: $2X^T(Xw^T - y)$
- By constraining the gradient to 0, we obtain the optimization condition:

$$X^T X w^T = X^T y$$

If the symmetric matrix $X^T X$ is *invertible* the solution can be derived as:

$$w = (X^T X)^{-1} X^T y$$

and the **forecast** (y^f) is given by:

$$y^f = X \cdot w^T$$

If $X^T X$ is not invertible, there could be issues.

Quality of the fitting

How we can measure the quality of the fitting?
The most important quality measure is the **coefficient of determination** R^2:

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

Where:

Mean of the observed data	$y^{avg} = \frac{1}{N}\sum_i y_i$
Sum of squared residuals	$SS_{res} = \sum_i (y_i - y_i^f)^2$
Total sum of squares	$SS_{tot} = \sum_i (y_i - y^{avg})^2$

> ⚠ **Warning**
>
> The total sum of squares can be w.r.t. to the model (SSM, also called sum of squares for regression):
>
> $$SSM = \sum_i (y_i^{predicted} - y_i^{avg})^2$$

or the to the total (SST):

$$SST = \sum_i (y_i - y_i^{avg})^2$$

R^2 is an absolute number, and it compares the fit of the chosen model with that of a horizontal straight line. Despite the name, it isn't the square of anything.
If the model does not follow the trend of the data the numerator of the second term can reach or exceed the denominator. R^2 *can also be negative*.

We could also use the **Sum of squared residuals** as a QM, but it is strictly related to the size of the data.

Another surprising fact about R^2 is that we could see as the ratio between the actual error (SS_{res}) and the **total sum of squares** (SS_{tot}), subtracted from 1.

The *perfect value* of R^2 would be *1*, since if $SS_{res} = 0$ then the prediction would not have any error whatsoever.

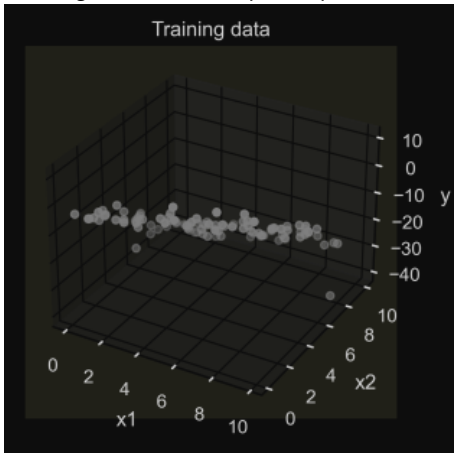
Mean Squared Error and R^2

- Both refer to the error of the predictions, but...
- R^2 is a standardized index,
- MSE measures the mean error, this it is influenced by the order of magnitude of the data.

Multiple regression

Regression can be univariate, but also *multiple regression* (meaning, with a vector of independent variables).

- The response variable depends by more than one features
- The regression technique is quite similar to that of simple regression



In this image, we have 2 independent variables x1 and x2, and a single response y.

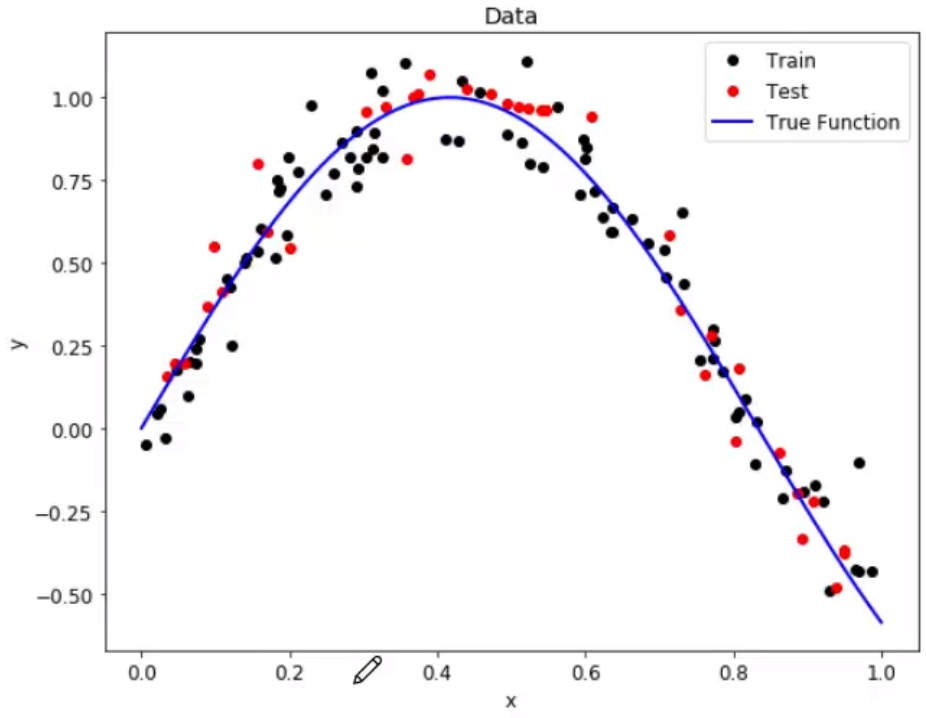
In this situation, the estimation strategy is the same as the single regression.

Overfitting and Regularisation in Regression

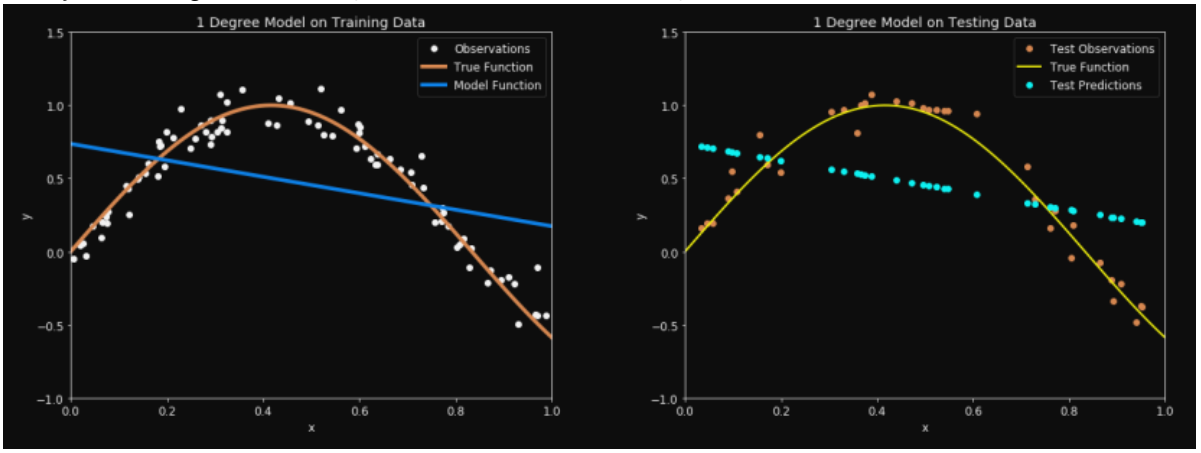
- In presence of high number of features, *overfitting* is possible, and performance on test data becomes much worse
- *Regularisation* reduces the influence of less interesting attributes and therefore reduces overfitting.
  - [not important] There are regression techniques such as the Lasso regression that try to select the most interesting attribute and show us how things change if we would have used a small number of attributes, so that we can tune to our liking.

Polynomial regression

What if we have data that is not linear at all? (this can be possible even in a univariate situation).



If a try a linear regression model, we could at most draw a line, like this:



This obviously results in **underfitting**.

So, the most obvious idea is to use, instead of a linear model, a *polynomial model* (duh...).

Is the model still linear? Intuitively, we could say that a model is linear if the things that I want it to guess are linear, since we just want to guess the coefficients (which are linear values, i.e. with a exponent of 1).
So, this is not so different from the general multivariate regression, we're just exploring a different perspective.

In the case of polynomial regression, we can choose the order of our polynomial, and consequently the number of the coefficient that we want to use.

Obviously, the higher the degree of the polynomial, the more accurate will be the approximation, but *it can also lead to overfitting*.

2022-11-11 - Preprocessing and dissimilarities

Preprocessing

As we mentioned before, we need to preprocess data so that the whole learning process becomes more efficient and improves in general.

There are various kinds of preprocessing methods:

- Aggregation

- Sampling
- Dimensionality
- Reduction
- Feature subset selection
- Feature creation
- Discretization and Binarization Attribute Transformation

Aggregation

It consists in combining two or more attributes (or objects) into a *single attribute* (or object). We could do this by, for example, reducing the data, or in a more general point of view, to change scale.

This process has many purposes, such as:

- **Data reduction**: reduce the number of attributes or objects. i.e. reducing the number of records by collapsing into one.
- **Change of scale**: cities aggregated into regions, states, countries, etc...
- **More stable data**: aggregated data tends to have less variability (and *less overfitting*)

i.e. we could consider the yearly data instead of the monthly data.

Sampling

The process of sampling is important for both preliminary investigation and final data analysis:

- Statistician perspective: obtaining the entire data set could be impossible or too expensive.
- Data processing perspective: processing the entire data set could be too expensive or time consuming.

We saw sampling in the train-test split & cross validation.
Using a sample will work almost as well as using the entire data sets, if the sample is representative (obv. this could never be 100% true).
In general, a sample is **representative** if it has approximately the same property (of interest) as the original set of data.

- Same property like same distribution of values in each feature.

Types of sampling

1. **Simple random**: a single random choice of an object with given probability distribution.
2. **With replacement**: repetition of independent extractions of type 1
   - We already saw that when we've mentioned Bagging.
3. **Without replacement**: repetition of extractions, *extracted element is removed* from the population.
4. **Stratified**: split data into several partitions *according to some criteria*, then draw the *random samples from each partition*.
   - used when the data set is split into subsets with homogeneous characteristics
   - the *representativity is guaranteed inside each subset.*

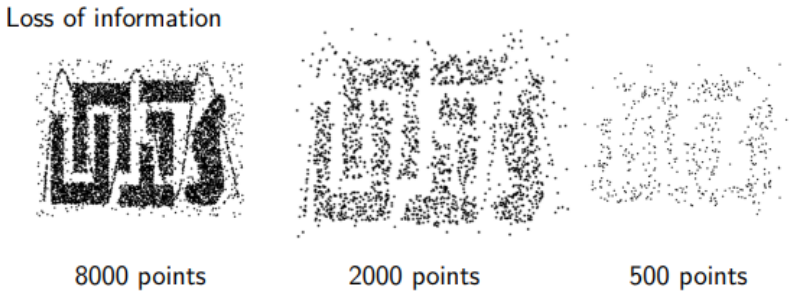
Stratification is also important in cross-validation, since the split should be done guaranteeing stratification as much as possible.

Usually, there's a tradeoff between sampling and precision.

Sampling with/without replacement

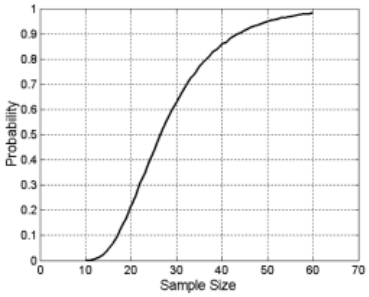
Sampling with and without replacement are *nearly equivalent* if sample size is a small fraction of data set size.
With replacement, in a small population (a small subset) could be underestimated.
Sampling with replacement is much easier to implement, and it is much easier to be interpreted from a statistical point of view extractions are statistically independent.

Here's an example of the consequences of sample size:



Sample size - Missing class

The following graph shows the probability of sampling at least one element for each class with replacement (It is independent from the size of the data set!).



For example, let's say we have 10 classes: A B C D E F G H I J.
The graph above represents the probability of capturing at least one element from each class.

We can see that if we draw a sample of 60 elements, the probability gets very close to 1.
The strange thing is that this probability *does not depend on the size of the dataset*.

This aspect becomes relevant, for example, in a supervised dataset with a high number of different values of the target.
If the number data elements is not big enough, it can be difficult to guarantee a stratified partitioning in train/test split or in cross–validation split.
i.e. if we have a dataset of N = 1000, C = 10, test–set–size = 300, cross–validation–folds = 10 (meaning that in each fold we have 30 elements) the probability of folds without adequate representation of some classes becomes quite high.

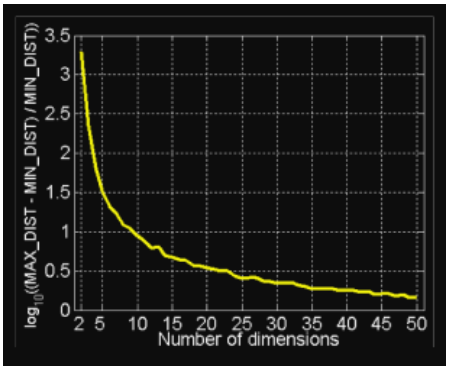
When designing the training processes it is necessary to consider those aspects, and it would be better to do a 3 folds cross validation instead of 10.

Dimensionality

The curse of dimensionality

- When dimensionality is very high the occupation of the space becomes very sparse.
- *Discrimination on the basis of the distance becomes ineffective*.

To better understand, let's consider a random generation of 500 points and plot the relative difference between the maximum and the minimum distance between pairs of point.



In the x axis we have the number of the dimensions, while in the y axis we measure the ratio between the maximum distances of points in a randomly generated dataset (meaning $\frac{max_d - min_d}{min_d}$).
We obtain a value that it's near 1 when minimum is very small and near 0 when the minimum is very high.

- A value near 1 means that it's to distinguish *the points that are near* from *the points that are far away* (or, in other words, the range of distances becomes very very small).

The value $\frac{max_d - min_d}{min_d}$ *decreases* when *the number of dimensions increases*, so it means that it becomes more difficult to distinguish the points that are near from the points that are far away.
In this situation, the k-nearest neighbors method becomes non-effective, and thus it is impossible to reason with distances.

Dimensionality reduction

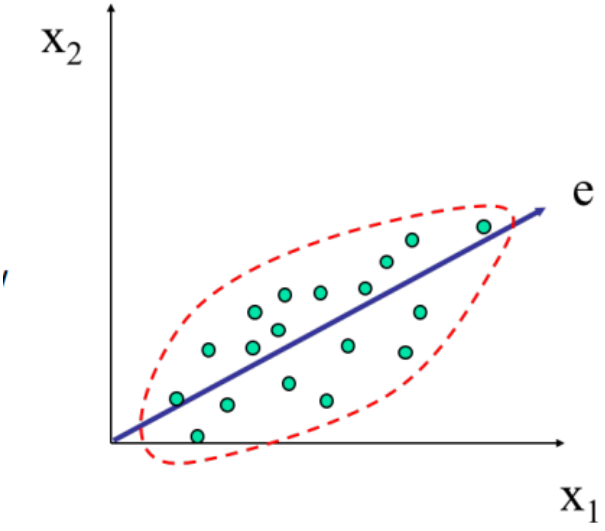
To solve this problem, we can reduce the dimensionality of values through **dimensionality reduction**.

- Purposes:
  - avoid the curse of dimensionality
  - *noise reduction*
  - *reduce time and memory complexity* of the mining algorithms
  - visualization
- Techniques:
  - principal component analysis (PCA)
  - singular values decomposition (SVD)
  - others...

PCA (Principal Component Analysis)

This is perhaps the most important method in dimensionality reduction. This is an *unsupervised technique*, meaning that it is independent from the classes.

Let's consider the following figure:



Suppose that we have a 2-dimensional dataset where the data is distributed as in the figure.
If we drop the x1 dimension, data will be *projected* on the x2 axis, and viceversa.
By dropping a dimension we lose some information, but if I project the data on line e, I lose less information (in particular, I lose less variability in comparison).

Principal Component Analysis is based in this exact notion, and its objective is to determine the *projection which allows to capture the most variability*.
"Find projections that capture most of the data variation". In mathematical terms, we'll have to:

- Find the eigenvector of the covariance matrix
- the eigenvectors define the new space

The new dataset will have only the attributes which capture most of the data variation.

Feature subset selection

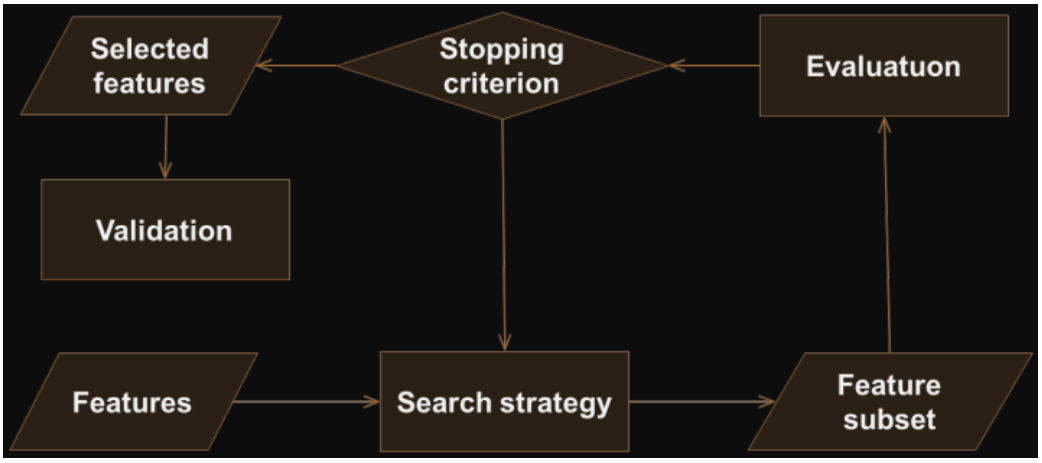
There are also *local ways* to reduce dimensionality, which do not need to compute eigenvectors or eigenvalues:

- Essentially, we just drop values that are deemed *redundant* or *irrelevant* (i.e. identification numbers), and which do not contain any information useful for the analysis.

Some of the methods of this type of selection are:

- **Bruteforce**: try all possible feature subsets as input to data mining algorithm and measure the effectiveness of the algorithm with the reduced dataset.
  - I try all the possible configurations of attributes, and seek for the best performance.
  - It does require a lot time and computing power.
- **Embedded approach**: feature selection occurs naturally as part of the data mining algorithm.
  - i.e. decision trees. We could compute a decision on which dimension to drop simply by looking at a decision tree.
  - We only use the attributes that have been used in the actual decision tree.
- **Filter approach**: features are selected before data mining algorithm is run
- Wrapper approaches: a data mining algorithm can choose the best set of attributes.
  - [the prof skipped them]

Here's an architecture for feature (attribute) subset selection:



Feature creation

Why would I want to create more features? The answer becomes clear when we consider, for example, the face recognition in smartphone images. Since pixel-by-pixel recognition is very much slow, some of the facial features are extracted from the face and used as attributes.

New features can capture more efficiently data characteristics:

- Feature extraction
  - pixel picture with a face ⇒ eye distance, . . .
- Mapping to a new space
  - e.g. signal to frequencies with Fourier transform
- New features
  - e.g. volume and weight to density

Data-type conversion

Many algorithms require numeric features, so *categorical features* must be transformed into *numeric*, ordinal features must also be transformed into numeric, with the addition that the order must be preserved.

Classification requires a target with nominal values ⇒ a numerical target can be discretized. Discovery of association rules require boolean features, a numerical feature can be discretized and transformed into a series of boolean features.

Attribute d allowing V values $\Rightarrow$ V binary attributes.

Quality	*Quality–Awful*	*Quality–Poor*	*Quality–OK*	*Quality–Good*	*Quality–Great*
Awful	1	0	0	0	0
Poor	0	1	0	0	0
Ok	0	0	1	0	0
Good	0	0	0	1	0
Great	0	0	0	0	1

One-Hot-Encoding (Nominal2Numeric)

A feature with V unique values is substituted by V binary features each one corresponding to one of the unique values. If object x has value v in feature d then the binary feature corresponding to v has `True` for x, all the other binary features have value `False`

- True and False are represented as 1 and 0, therefore can be processed by also by procedures working only on numeric data.

Ordinal2numeric

- The *ordered sequence* is transformed into *consecutive integers*.
  By default the lexicographic order is assumed.
  The user can specify the proper order of the sequence.

Numeric2Binary with threshold

- Not greater than the threshold becomes zero
- *Greater* than the threshold becomes *one*

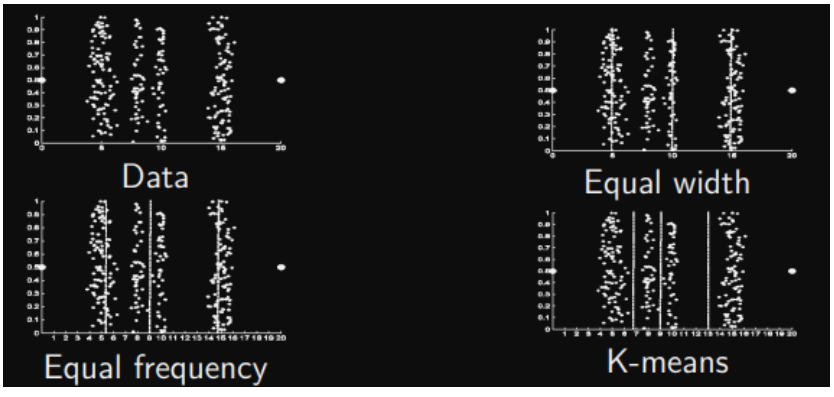
Discretization

This is less strong than binarization.

- Some algorithms work better with categorical data.
- A small number of distinct values:
  - let the algorithms to be less influenced by noise and random effects.
  - let patterns emerge more clearly.

Discretization can happen in two ways:

- Continuous ⇒ discrete: we use many thresholds.
- Discrete with many values ⇒ Discrete with less values: we use some domain knowledge.



In this graphs, we can see how the data in the y axis is uniform, while the data on the x axis is not (we have groupings). We discretize it in the following ways:

- In the first case, we have many different thresholds which delimit the area of a label. But, this may ignore some data or patterns.
- Equal frequency means that the thresholds such that in each interval there will be the same amount of points.

Similarity and dissimilarity (proximity functions)

- **Similarity**
  - *Numerical measure* of *how alike* two data objects are
  - Is higher when objects are more alike
  - Often falls in the range [0,1]
- **Dissimilarity**
  - Numerical measure of *how different* are two data objects
  - Lower when objects are more alike
  - Minimum dissimilarity is often 0
  - *Upper limit varies*
- Proximity refers to a similarity or dissimilarity

Here's a table which shows how dissimilarity and similarity are computed given the attribute type. p and q are the values of an attribute for two data objects.

Attribute type	Dissimilarity	Similarity
Nominal	$d = \begin{cases} 0 \text{ if } p = q \\ 1 \text{ if } p \neq q \end{cases}$	$s = \begin{cases} 1 \text{ if } p = q \\ 0 \text{ if } p \neq q \end{cases}$
Ordinal Values mapped to integers 0 to V-1	$d = \frac{\lvert p-q \rvert}{V-1}$	$s = 1 - \frac{\lvert p-q \rvert}{V-1}$
Interval or Ratio	$d = \lvert p - q \rvert$	$s = \frac{1}{1+d}$ or $s = 1 - \frac{d - \min(d)}{\max(d) - \min(d)}$

Continuous values

If our dataset is represented by continuous values, then we are in the domain usually covered by the Euclidean distance.

Euclidean distance - L_2

$$dist = \sqrt{\sum_{d=1}^{D} (p_d - q_d)^2}$$

Where D is the number of dimensions (attributes) and p_d and q_d are, respectively, the d-th attributes (components) of data objects p and q.
Standardization/Rescaling is necessary if scales differ.

- If an attribute is in the order of magnitude that is way bigger than the others, then *it will influence the distance more* (and in turn, if the values of this attributes are affected by noise *noise will influence the final value even more*).
- For this reason, it is extremely common to rescale the values, using for example a MinMax scaler.
- Some machine learning algorithms *are sensitive to feature scaling* while others are virtually invariant to it.

(N.B.: in DTs, we dont need to scale since we have entropy. Liner perceptrons are influenced by different scales, and so are SVMs, and k-nearest neighbors).

Minkowski distance – L_r

$$dist = (\sum_{d=1}^{D} |p_d - q_d|^r) \frac{1}{r}$$

- Where D is the number of dimensions (attributes) and p_d and q_d are, respectively, the d-th attributes (components) of data objects p and q.
- Standardization/Rescaling is necessary if scales differ.
- r is a *parameter* which is chosen depending on the data set and the application.
  - if $r = 1$, we have the Manhattan distance.
    - It is used when data is binary (i.e. when data is computed in bits)
    - it works better than euclidean distance with high-dimensional data.
  - if $r = \infty$, we have the Chebyshev distance, $\max_d |p_d - q_d|$.

Mahalanobis Distance

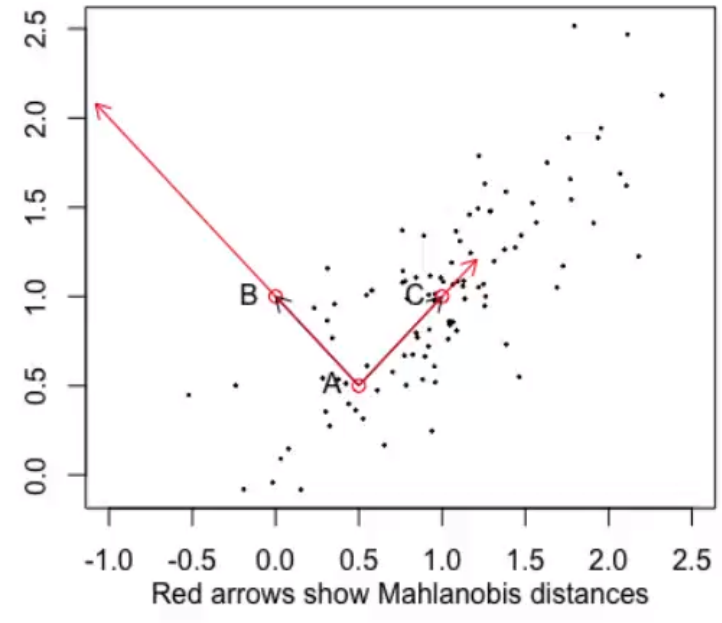
It is strictly related to PCA.
The **Mahalanobis** distance between two points p and q *decreases* if, keeping the same Euclidean distance, the *segment connecting the points* is *stretched* along *a direction* of *greater variation* of data.
The distribution is described by the *covariance matrix* of the data set:

$$\Sigma_{ij} = \frac{1}{N-1}\sum_{k=1}^{N}(x_{ki} - \bar{x}_i)(x_{kj} - \bar{x}_j)$$

$$\text{dist}_m = \sqrt{(p-q)\Sigma^{-1}(p-q)^T}$$

This notion is explained more effectively using a picture:

Consider the 3 points A,B,C, which have equal euclidean distance as shown in the picture.



$$\text{dist}_m(A, B) = 2.236068$$
$$\text{dist}_m(A, C) = 1$$

The Mahalanobis distance between A and B is 2.23.. because in that direction there is less variability.

Covariance matrix

- Variation of pairs of random variables
- The summation is over all the observations
- The main diagonal contains the variances
- The values are positive if the two variables grow together
- If the matrix is diagonal the variables are non–correlated
- If the variables are standardized the diagonal contains “one”
  - Standardised, meaning they are mapped onto a standard
- If the variables are standardised and non correlated, the matrix is the identity and the Mahalanobis distance is the same as the euclidean

Common properties of a distance

1. Positive definiteness: $\text{Dist}(p, q) \geqslant 0\ \forall p, q$ and $\text{Dist}(p, q) = 0$ if and only if $p = q$
2. Symmetry: $\text{Dist}(p, q) = \text{Dist}(q, p)$
3. Triangle inequality: $\text{Dist}(p, q) \leqslant \text{Dist}(p, r) + \text{Dist}(r, q) \forall p, q, r$

A distance function satisfying all the properties above is called a metric

Common properties of similarities

1. $Sim(p, q) = 1$ only if $p = q$
2. $Sim(p, q) = Sim(q, p)$

Similarity between binary vectors

- Consider the counts below
  - M_{00} the number of attributes where p is 0 and q is 0
  - M_{01} the number of attributes where p is 0 and q is 1
  - M_{10} the number of attributes where p is 1 and q is 0
  - M_{11} the number of attributes where p is 1 and q is 1
- Simple Matching Coefficient

$$\text{SMC} = \frac{\text{number of matches}}{\text{number of attributes}} = \frac{M_{00} + M_{11}}{M_{00} + M_{01} + M_{10} + M_{11}}$$

- Jaccard Coefficient

$$\text{JC} = \frac{\text{number of 11 matches}}{\text{number of non–both–zero attributes}} = \frac{M_{11}}{M_{01} + M_{10} + M_{11}}$$

Cosine similarity

- It is the cosine of the angle between two vectors

$$\cos(p, q) = \frac{p \cdot q}{\|p\| \|q\|}$$

- Example

$p = 3\ 2\ 0\ 5\ 0\ 0\ 0\ 2\ 0\ 0$

$q = 1\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 2$

$p \cdot q = 3*1 + 2*0 + 0*0 + 5*0 + 0*0 + 0*0 + 0*0 + 2*1 + 0*0 + 0*2 = 5$

$\|p\| = \sqrt{3*3 + 2*2 + 0*0 + 5*5 + 0*0 + 0*0 + 0*0 + 2*2 + 0*0 + 0*0} = 6.481$

$\|q\| = \sqrt{1*1 + 0*0 + 0*0 + 0*0 + 0*0 + 0*0 + 0*0 + 1*1 + 0*0 + 2*2} = 2.245$

$\cos(p, q) = .3150$

It is used in word processing.
The cosine becomes 1 when they're equal.

Which proximity measure to choose?

It depends on data.

- Dense, continuous data: a metric measure, such as the euclidean distance
- Sparse, asymmetric data: cosine

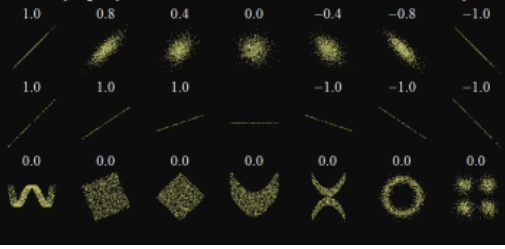
Correlation

Measure of the linear relationship between a pair of attributes

- Standardize the values
- For two given attributes p and q, consider as vectors the ordered lists of the values over all the data records
- Compute the dot product of the vectors

$$\mathbf{p} = [p_1, \ldots, p_N] \xrightarrow{standardize} \mathbf{p}'$$
$$\mathbf{q} = [q_1, \ldots, q_N] \xrightarrow{standardize} \mathbf{q}'$$
$$\text{corr}(p, q) = \mathbf{p}' \bullet \mathbf{q}'$$

- Independent variables $\Rightarrow$ correlation is zero
  - the inverse is not valid *in general*
- Correlation zero $\Rightarrow$ absence of *linear relationship* between the variables
- Positive values imply positive linear relationship



In the picture below on this slide, we see what the correlation values are in the various situations (aka different points distribution).
As we can see, a line has *maximum correlation*. So, correlation is useful, but only when the relationship is linear.

Correlation between nominal attributes

Consider the case in which we have the city of birth and the city of residence of a person. We can find a correlation between these two values, called the **Symmetric Uncertainty**.

$$U(p, q) = 2\frac{H(p) + H(q) - H(p, q)}{H(p) + H(q)}$$

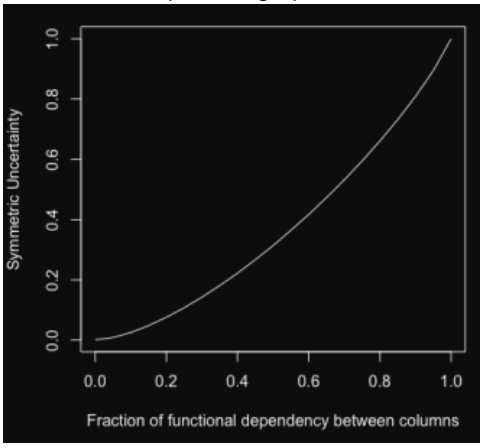
This correlation is computed using the notion of entropy $H()$.

- $H(,)$ is the joint entropy, computed from the joint probabilities.
- The entropy is 0 when $H(p) + H(q) = H(p, q)$.

Let's see an example to understand this better:

- Let's consider a SU for two independent uniformly distributed discrete attributes, say p and q. In a variable fraction of records, the value of p is copied to q.

The SU will output this graph:



- The graph goes from complete independence (left) to complete biunivocal correspondence (right).

- when *there is independence*, the joint entropy is the sum of the individual entropies, and *SU is zero*.
- when there is *complete correspondence*, the individual entropies and the joint entropy are equal and *SU is one*.

Gradient descent algorithms and transformations

Machine learning algorithms that use *gradient descent* as an *optimization technique* require data to be scaled.

- e.g. linear regression, logistic regression, neural network, etc.
- The presence of feature value X in the formula *will affect the step size* of the gradient descent. So the bigger is the scale, the bigger is the step in that direction.
- The difference in ranges of features will cause different step sizes for each feature.
- Similar ranges of the various features ensure that the gradient descent moves smoothly towards the minima and that the steps for gradient descent are updated at the same rate for all the features

Some transformation strategies:

- Map the entire set of values to a new set according to a function
  - $x^k, \log(x), e^x, |x|$
  - in general they change the *distribution of values*
- Standardization: $x \rightarrow \frac{x-\mu}{\sigma}$
  - if the original values have a *gaussian* distribution, the transformed values will have a *standard* gaussian distribution ($\mu = 0, \sigma = 1$)
  - translation and shrinking/stretching, no change in distribution
- MinMax scaling (a.k.a. Rescaling): the domains are mapped to standard ranges

$$x \rightarrow \frac{x - x_{min}}{x_{max} - x_{min}} \quad \text{(0 to 1)} \qquad x \rightarrow \frac{x - \frac{x_{max}+x_{min}}{2}}{\frac{x_{max}-x_{min}}{2}} \quad \text{(-1 to 1)}$$

  - translation and shrinking/stretching, no change in distribution

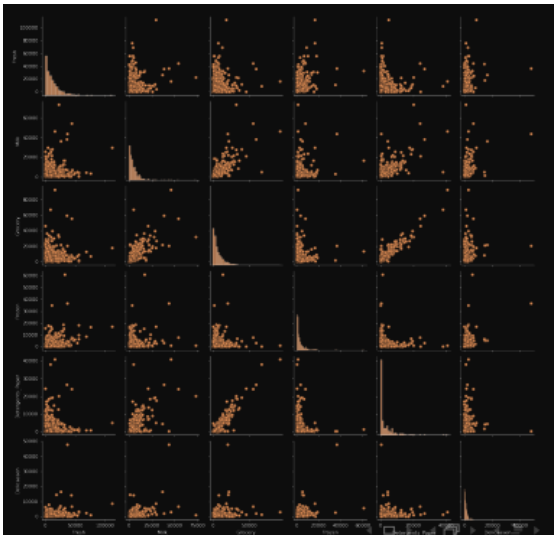
changing the distribution values means also changing the histograms for example.

- General mapping can change the distribution, while Standardization and MinMax don't.

2022-11-16 - Last remarks on Transformations, Introduction to Clustering (K-Means)

Continuation on transformations

Attribute transformations which change data distribution can be useful when data is skewed (meaning that there are large agglomerations of data, with some very evident outliers).



We could use a `PowerTransformer` to make data less skewed.
Here's an example of scaled data (standardization):

Original data		
Student	**CGPA**	**Salary**
A	3	60
B	3	40
C	4	40
D	4.5	50
E	4.2	52

Scaled data		
Student	**CGPA**	**Salary**
A	-1.18431	1.520013
B	-1.18431	-1.100699
C	0.41612	-1.100699
D	1.21635	0.209657
E	0.736212	0.471728

Here's what happens to distances before and after scaling:

Before
$$distance(A, B) = \sqrt{(40 - 60)^2 + (3 - 3)^2} = 20$$
$$distance(B, C) = \sqrt{(40 - 40)^2 + (4 - 3)^2} = 1$$

After
$$distance(A_s, B_s) = \sqrt{(1.1 + 1.5)^2 + (1.18 - 1.18)^2} = 2.6$$
$$distance(B_s, C_s) = \sqrt{(1.1 - 1.1)^2 + (0.41 + 1.18)^2} = 1.59$$

Before the scaling the two distances seemed to be very different, due to a big numeric difference in the `Salary` attribute, now *they are comparable*.

Transformations operating on a ***single*** attribute

- **Range–based scaling** stretches/shrinks and translates the range, according *to the range of the feature* (there are some variants)
  - good when we know that the data are not Gaussian, or we do not make any assumption on the distribution
  - the base variant, the **MinMax scaler**, remaps to [0, 1]
- **Standardization** *subtracts the mean* and *divides by the standard deviation* ($\frac{x - \mu}{\sigma}$)
  - the resulting distribution has mean zero and unitary standard deviation
  - *good when the distribution is Gaussian*
  - `StandardScaler`

Transformation in Scikit-Learn

These are affine transformations: linear transformation plus translation

- `MinMaxScaler` – remaps the feature to [0, 1]
- `RobustScaler` – centering and scaling statistics is based on *percentiles*
  - less influenced by outliers (in similar way to medians).

[the prof said this were important, so I added them]

Normalization

Normalization is mentioned sometimes with different meanings, frequently it refers to `MinMaxScaler`.

- in Scikit–learn the Normalizer normalizes *each data row* to *unit norm* (it considers all the attributes of an individual)
  - Data is mapped onto a circle (circumference to be exact) with the center on the origin and $radius = 1$.

Workflow in transformation

1. transform the features as required both for the train and test data
2. fit and optimize the model(s)
3. test
4. possibly, use the original data to plot relevant views (e.g. to plot cluster assignments)

Unbalanced data in classification

- The *performance minority class* (classes) has little impact on standard performance measures
- The optimized model could be *less effective* on *minority class* (classes)
- As we saw, some estimators allow to weight classes
- Some performance measures allow to take into account the contribution of minority class (classes)

Cost Sensitive learning

- several classifiers have the parameter `class_weight`, which changes the cost function to take into account the unbalancing of classes
- in practice *it is equivalent to* ***oversampling*** *the minority class*, (repeating random examples) in order to produce a balanced training set.

Undersampling

- Obtains *a balanced training set* by randomly *reducing* the number of examples of the *majority class*
- Obviously part of the knowledge embedded in the training set is dropped out

Oversampling with SMOTE (Synthetic Minority Oversampling Technique)

It synthesizes *new examples* from *the minority class*, so it is a type of *data augmentation*.
Here's how it works:

- a random example from the minority class is first chosen
- k of the nearest neighbors are found
- a *randomly selected neighbor* is chosen, and a *synthetic example* is created at a randomly *selected point between the two examples* in feature space.

This method is more robust to generalization, since it is essentially duplicating with some noise (by adding data that is somewhat plausible).

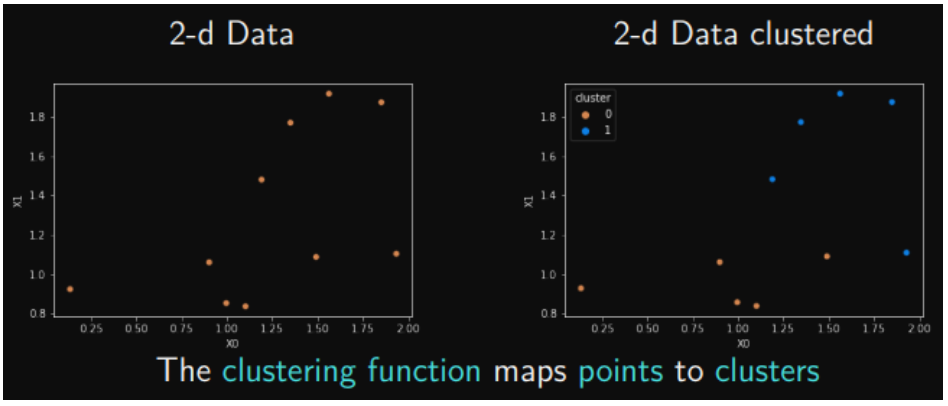
Clustering

- Given a set of N objects x_i, each described by D values x_{id}.
- Our task is to *find a natural partitioning* in K **clusters** and, possibly, a number of *noise objects*
- The result to solving a **clustering scheme**, i.e. a *function* mapping each *data* object to *the sequence* $[1 \dots K]$ (or to noise)
- Desired property of clusters: objects in the same cluster are similar
    - look for a clustering scheme which maximizes intra–cluster similarity

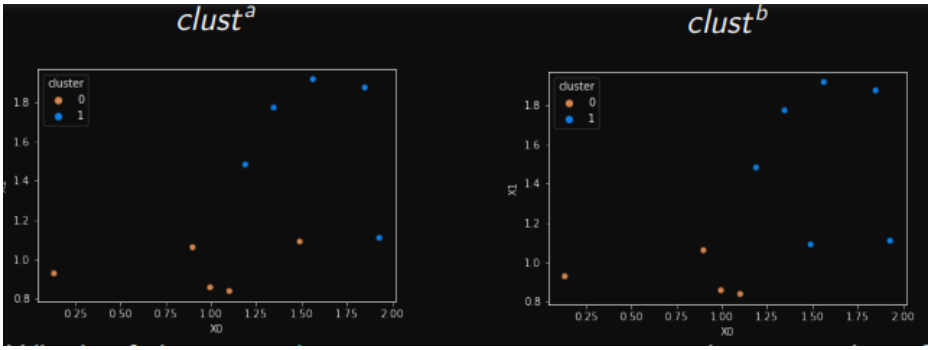
In formal wording:

- find a function $clust()$ from $\mathcal{X}$ to $1..K$ such that:
    - $\forall x_1, x_2 \in \mathcal{X}$, $clust(x_1) = clust(x_2)$ when x_1 and x_2 are *similar*
    - $\forall x_1, x_2 \in \mathcal{X}$, $clust(x_1) \neq clust(x_2)$ when x_1 and x_2 are *not similar*

So, clustering in a sense is similar to classification, but *we don't have a ground truth*, there's nothing to compare the data.



There are many ways that we can cluster data:



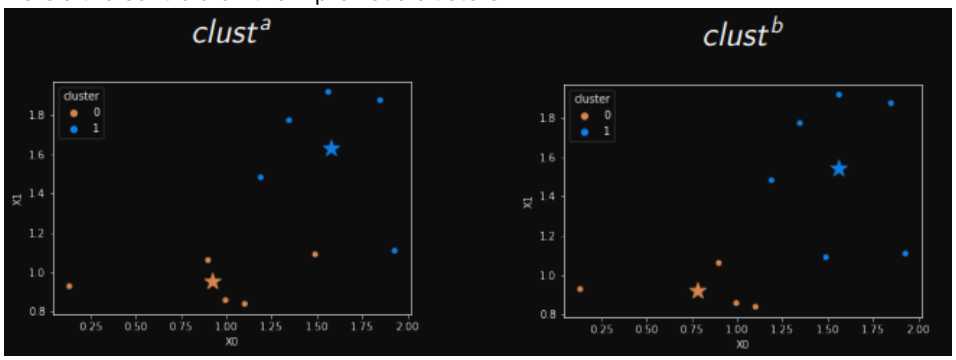
Which of the two is better, i.e. *maximizes intra–cluster similarity*? A measure is needed.

Centroid

- A **centroid** is a *point* with coordinates computed as *the average of the coordinates* of all the points in the cluster.
- in physics, it is the center of gravity of a set of points of equal mass.
- for each cluster k and dimension d, the d coordinate of the centroid is defined as:

$$centroid_d^k = \frac{1}{|x_i : clust(x_i) = k|} \sum_{x_i : clust(x_i) = k} x_{id}$$

Here's the centroid on the 2 previous clusters:

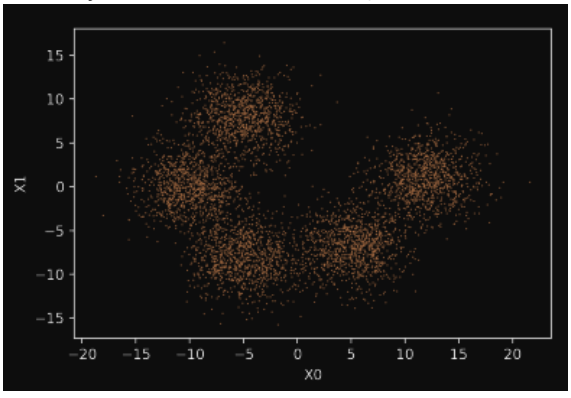


K-means

Understanding K-Means

[This first part is only related to understanding the method. You can skip most of this if you already undestand]

Let's say that we have some data, which is distributed like this.



We can clearly see that there are 5 clouds of data. But can we compute that there are (effectively) 5 clouds in a D-dimensional space?

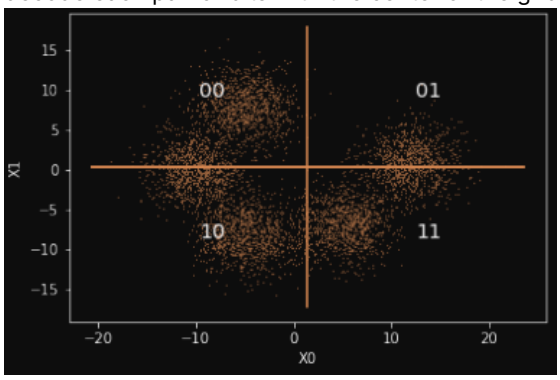
To understand, let's use the metaphor of a transmission of data.

- We transmit the coordinates of points, in particular we want to transmit where the points are distributed in the space.
- Allow only *two bits* per point in *the transmission*, so it will be lossy
  - We need a coding/decoding mechanism.
- The loss is equal to the *sum of the squared errors* (SSE) between the *real points* and their *encoding/decoding*.

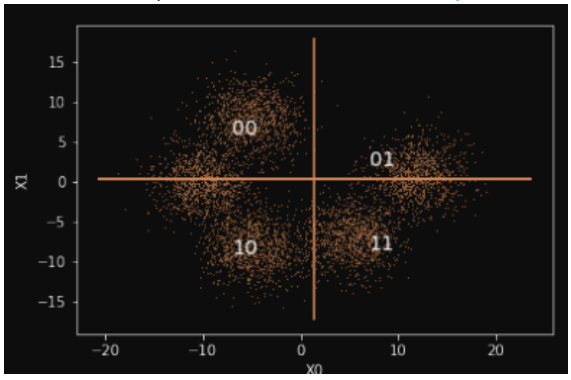
- The coding/decoding mechanism must minimize this loss.

First idea for space representation:

- partition the space into a grid of cells
- decode each pair of bits with the center of the grid cell.

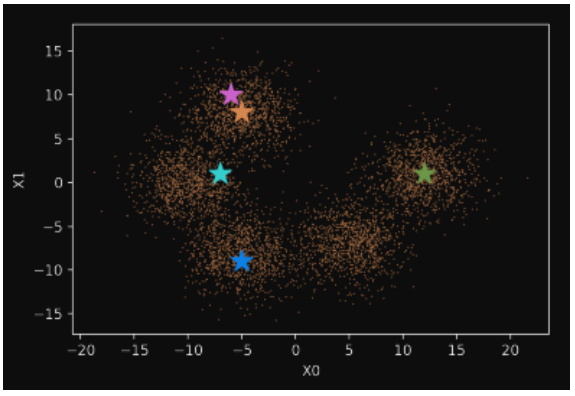


- to reduce the loss (and improve the strategy), we can take into account how the original data is distributed between the quadrants: we *decode each pair of bits* with *the centroid* of the points in the grid cell.



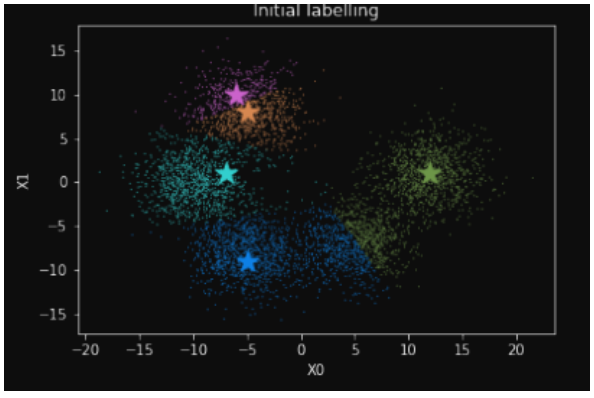
We can even do better though!

- Let's cheat a bit, and decide that we can decide the number of partitions (clusters).
- We ask the user the number of cluster K
- From the data, we select K randomly, as *temporary centers*. These are also our decoding points.



At this points, we can partition the points by using the k-nearest neighbors algorithm.

- *Each point* finds *his nearest center* and is labelled (i.e. colored) accordingly.



x_iNow that I have the coloring scheme, it is reasonable that each star is placed in a place that minimizes the loss (the centroid).
- For each center, *finds the centroid* of its points, and *move there the center*.
- Repeat the K-nearest neighbors classification, and find the centroid again etc...

Remaining questions to solve

1. What are we trying to optimize?
2. Is termination guaranteed?
3. Are we sure that the best clustering scheme is found?
   - Which is the definition of best clustering scheme?
4. How should we start?
5. How can we find the number of clusters?
   - Normally, we can't just ask the user for K...

Question 1: Distortion

We are trying to optimize the *distortion*, frequently called in the literature/physics Inertia.

- The distortion is the sum of the *differences* between *the point* and *the centroid of its cluster*.

Given:

a dataset	$\{x_i, i = 1 \dots N\}$
a coding function	$\text{Encode} : \{R\}\hat{}D \rightarrow [1..K]$
a decoding function	$\text{Decode} : [1..K] \rightarrow \mathbb{R}^D$
define	$\text{Distortion} = \sum_{i=1}^{N} (x_i - \text{Decode}(\text{Encode}(x_i)))^2$
shortcut	$\text{Decode}(k) = \mathbf{c}_k$
then	$\text{Distortion} = \sum_{i=1}^{N} (x_i - \mathbf{c}_{\text{Encode}(x_i)})^2$

N.B.: in our case, the encoding is the coloring (cluster), the decoding is the centroid.

- MEANS: _THE VALUE X_i IS THE ACTUAL VALUE, WHILE C_i is the approximation.
- or: x_i is the value, c_i is the cluster.
  The *distortion* is the value that we want to minimize.
  Which properties are requested to $c_1, \dots, c_K$ for the minimal distortion?

1. x_i must be encoded with the nearest center.

$$\mathbf{c}_{\text{Encode}(x_i)} = \underset{\mathbf{c}_j \in \{\mathbf{c}_1, \dots, \mathbf{c}_K\}}{\text{argmin}} (x_i - \mathbf{c}_j)^2$$

   (I want to find the value j minimizing that value).

2. The *partial derivative of distortion* w.r.t. the position of each center *must be zero*, because in that case the function has either a maximum or a minimum.

$$\begin{aligned}\text{Distortion} &= \sum_{i=1}^{N} \left(x_i - \mathbf{c}_{\text{Encode}(x_i)}\right)^2 \\ &= \sum_{j=1}^{K} \sum_{i \in \text{OwnedBy}(\mathbf{c}_j)} (x_i - \mathbf{c}_j)^2 \\ \frac{\partial\, \text{Distortion}}{\partial \mathbf{c}_j} &= \frac{\partial}{\partial \mathbf{c}_j} \sum_{i \in \text{OwnedBy}(\mathbf{c}_j)} (x_i - \mathbf{c}_j)^2 \\ &= -2 \sum_{i \in \text{OwnedBy}(\mathbf{c}_j)} (x_i - \mathbf{c}_j)\end{aligned}$$

When distortion is minimal

$$\mathbf{c}_j = \frac{1}{|\,\text{OwnedBy}(\mathbf{c}_j)|} \sum_{i \in \text{OwnedBy}(\mathbf{c}_j)} x_i$$

Naturally, we obtain the formula of the centroid. All this because you wanted a stupid answer to a question you may not even ask.

So, to achieve minimal distortion, we must perform these operations alternately:

1. x_i must be encoded with the nearest center
2. each center must be the centroid of the points it owns

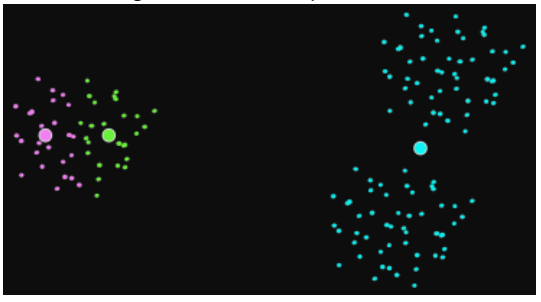
Question 2: Algorithm termination

It can be proven that after a finite number of steps the system reaches a state where neither of the two operations changes the state.

- There is only a *finite number of ways to partition* N objects into K groups
- The state of the algorithm is given by the two encode/decode functions
- The number of configurations where all the *centers are the centroids of the points* they own is *finite*
- If after one iteration the state changes, *the distortion* is *reduced*.
- Therefore, each change of state bring *to a state which was never visited before*.
  - [Asperti would say "we're in a simpler state than before, so convergence is ensured"]
- In summary, sooner or later the algorithm will stop because there are no new states reachable.

Question 3: Local or global minimum?

Is the ending state the best possible? Not necessarily. An example:



A random assignment of the centers places the center in two very bad spots.
We would want one point in the cloud on the left and two points in the two clouds on the right, but our algorithm can't move from here.
This is because the algorithm is of the *greedy* type, and doesn't necessarily explore all the possible states.
So, the algorithm guarantees only to find a *local minimum*, and is sub-optimal.

- We can try to relax this problem by inserting initial random centers that are not too close to each other.

Question 4: Looking for a good ending state

The starting point is important!

- choose randomly the first starting point
- choose in sequence the $2..K$ starting points *as far as possible* from the preceding ones.
- Re-run the algorithm with different starting points

Question 5: Choose the number of clusters

not so easy...

- *try various values*
- use a quantitative evaluation of the quality of the clustering scheme to decide among the different values
- the best value finds the optimal compromise between the minimization of intra-cluster distances and the maximization of the inter cluster distances.

In general, when we increase the number of clusters, the *distortion decreases*.

The proximity function

We've discussed about proximity functions in the past. What proximity function should we use?

- The most obvious solution, used in the previous formulas is the *Euclidean distance*.
  - good choice, in general, for vector spaces
- Several alternative solutions for specific data types and data sets see the "Data" module for additional discussions.

Sum of squared errors

The official name of the distortion is the **Sum squared errors** (SSE).

$$\begin{aligned} \text{SSE} &= \sum_{j=1}^{K} \sum_{i \in \text{OwnedBy}(\mathbf{c}_j)} (x_i - \mathbf{c}_j)^2 \\ &= \sum_{j=1}^{K} \text{SSE}_j \end{aligned}$$

We've also already seen SSE here.

- *A cluster j with high SSE_j has low quality*.
- $SSE_j = 0$ if and only if all the points are coincident with the centroid.
- SSE decreases for increasing K, is zero when $K = N$
- ⇒ *minimizing SSE is not a viable solution to choose the best K*.

Empty clusters

It may happen, at some step, that a centroid does not own any point.
Thus, we may need to *choose a new centroid*.

Outliers

Outliers are points *with high distance from their centroid*.
They have high contribution to SSE, and have a bad influence on the clustering results.
Sometimes *it is a good idea to remove them*, the choice is related to the application domain.

Uses of K-means

- It can be easily used in the beginning, for the *exploration of data*.
- In a one-dimension space, it is *a good way to discretize the values* of a domain in non-uniform buckets.
- Also, it is *very fast*.
- Used for choosing the color palettes, GIF compressed images: color quantization, vector quantization.

Complexity of K-mean

Complexity is

$$O(TKND)$$

where:

- T number of iterations
- K number of clusters
- N number of data points
- D number of dimensions

Pros and cons of K–means

Pros

- Fairly efficient, nearly linear in the number of data points in general $T, K, D << N$

Cons

- in essence it is defined for spaces where the centroid can be computed
  - e.g. when the Euclidean distance is available, also other distance functions work well
  - *cannot work with nominal data*
- *requires the K parameter*
  - nevertheless the best K can be found with iterations
- it is very *sensitive to outliers*
- does *not deal with noise*
- does *not deal properly with non-convex clusters*.
  - A cluster that has a concavity in this space.

2022-11-18 - Evaluation of a clustering algorithm, Hierarchical clustering, Density based clustering

Evaluation of a clustering scheme

The evaluation of a *clustering scheme* is related only to the *result*, not to the *clustering technique*.

- Clustering is a non supervised method
  - the evaluation is critical, because there is very little a-priori information, such as class labels.
  - we need one or more *score (or index) function* to measure various properties of the clusters and of the *clustering scheme* as a whole
  - if some *supervised data are available*, they can be used to *evaluate the clustering scheme*.
- In 2D the clusters can be examined visually. In higher order spaces the 2D projections can help, but in general it is better to use more formal methods

Issues on clustering evaluation

- Distinguish patterns from random apparent regularities
- Find the best number of clusters
- *Non supervised* evaluation
- *Supervised* evaluation
- Relative comparison of clustering schemes

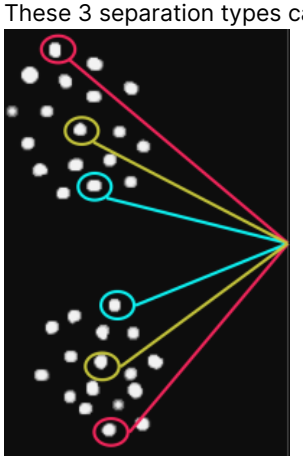
Unsupervised evaluation

Measurement criteria

- **Cohesion**
  - *proximity of objects in the same cluster should be high* (they should be near)
- **Separation** between two clusters. We can measure it in several ways:
  - *distance* between the *nearest objects* in the two clusters
  - *distance* between the *most distant objects* in the two clusters

- *distance* between the *centroids* of the two clusters

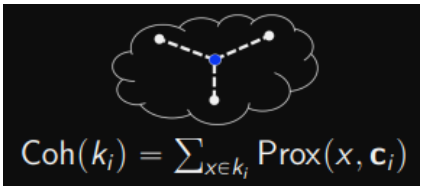
These 3 separation types can be seen here:



- **Similarity – Proximity**
  - a two variable function measuring how much *two objects are similar*, *according to the values of their properties*.
  - Seen a ton of times already.
- **Dissimilarity**
  - a two variable function measuring *how much two objects are different*, according to the values of their properties (e.g. the Euclidean distance).
  - Seen also here.

Cohesion

The **cohesion** is the *sum* of the *proximities* between *the elements of the cluster* and the geometric center (the *centroid* or *medoid*).



- As we know, a centroid is a point in the space whose coordinates are the means of the dataset
- A **medoid** is an element of the dataset *whose average dissimilarity with all the elements of the cluster is minimal*. It is not necessarily unique, used in contexts where the mean is not defined, e.g. 3D trajectories or gene expressions.
- While a centroid is not necessarily a part of a cluster, a medoid is.
  - Association between average and median.

Separation

Separation between two clusters is measured through the proximity between the prototypes.

Global separation of a clustering scheme

Can be computed by calculating the **Sum of Squares Between clusters** (SSB).

$$SSB = \sum_{i=1}^{K} N_i Dist(c_i, c)^2$$

Where c is the global centroid of the dataset.

Total Sum of Squares

The TSS is a global property of the dataset, independent from the clustering scheme.

$$TSS = SSE + SSB$$

- SSE is the Sum of Squared Errors.
- SSB is the Sum of Squares Between clusters.

- for a given dataset, minimize SSE ⇔ maximize SSB

Clustering

In principle, we should consider all the possible clusterings of each number, so it is a huge number.
What we have to do is find a shortcut.

Evaluation of specific clusters and objects

- Each *cluster can have its own evaluation*, and the worst clusters can be considered for additional split.
- A weakly separated pair of clusters could be considered for merging
- *Single objects* can give *negative contribution* to the *cohesion* of a cluster or to the separation between two clusters (*border objects*).

Silhouette score of a cluster

Requirements for a clustering quality score

- values are in a *standard range*, e.g. -1, 1
- *increases with the separation* between clusters.
  - The higher the silhouette score, the more the clusters are well separated: it can be used as a QM for clustering technique.
- *decreases for clusters with low cohesion*, or, in other words, with *high sparsity*.

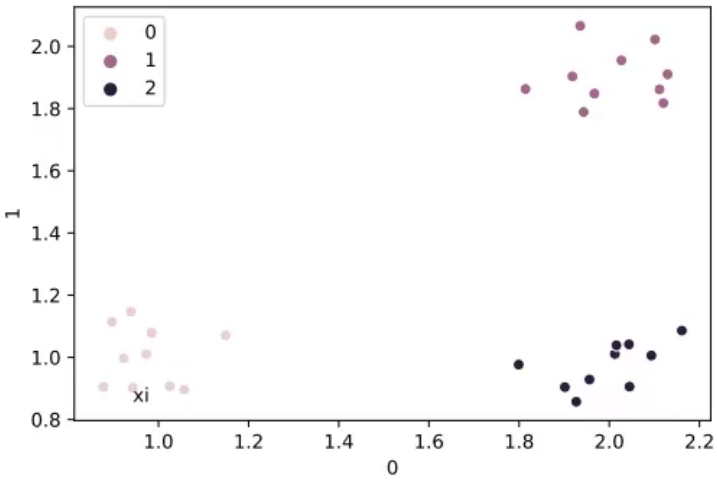
The silhouette score is an *index*, meaning that *it does not have a measure*.

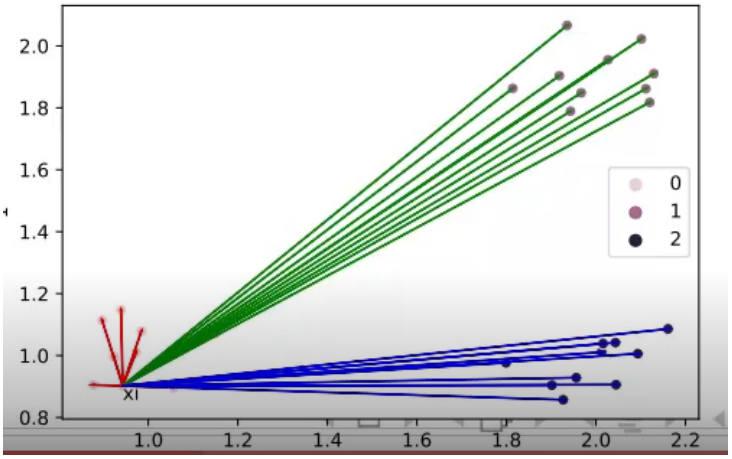
Consider the individual contribution of each object, say x_i.
We can compute the contribution of the individual to the cluster *sparsity*.

$$a_i = \operatorname*{average}_{j, y(x_j)=y(x_i)} dist(x_i, x_j)$$

and the contribution to *separation* from *other clusters*:

$$b_i = \min_{k \in \mathcal{Y}, k \neq y(x_i)} \left(\text{average } dist(x_i, x_j) \right)_{j, y(x_j)=k}$$





In this graph:

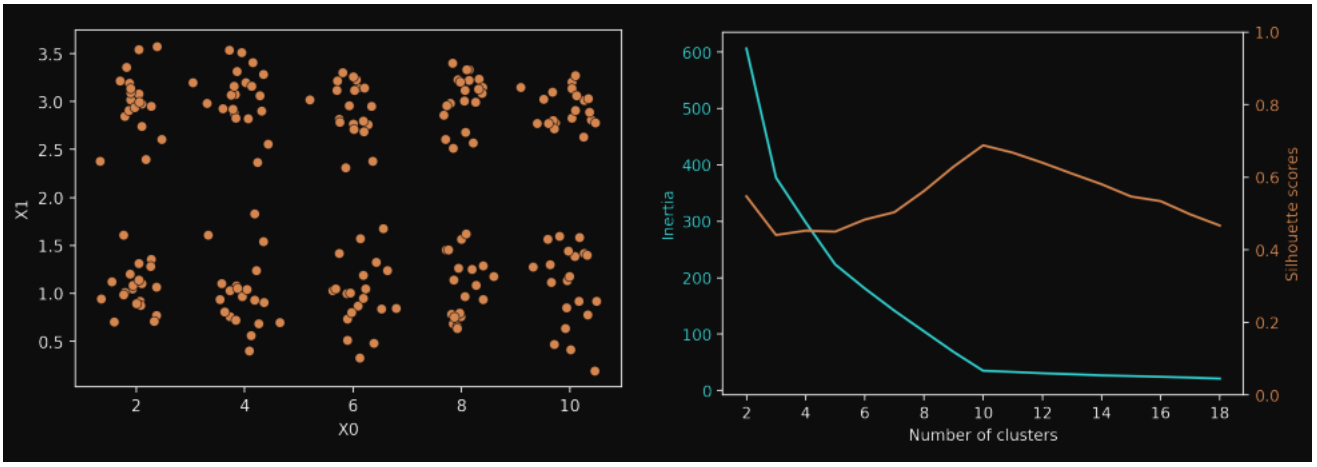
- The average of the red distances is a_i
- The minimum of the two averages of green and blue distances is b_i

Given these definitions, the **Silhouette score** of x_i is:

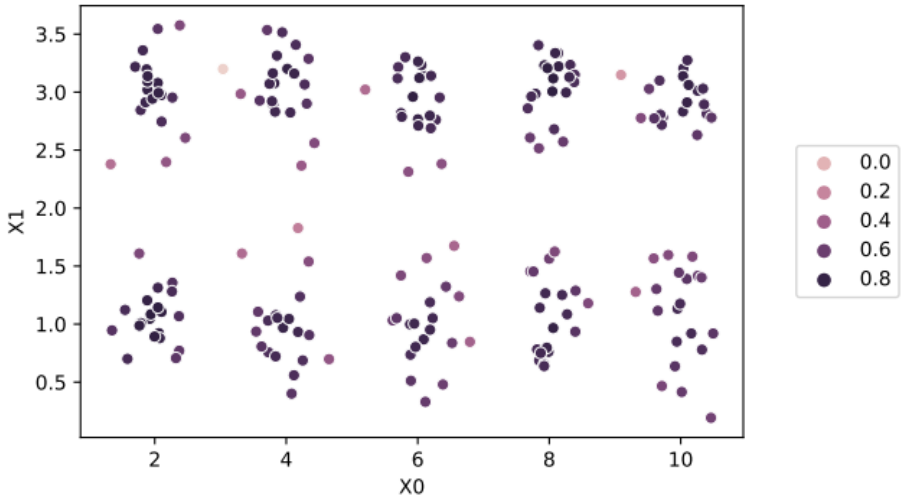
$$s_i = \frac{b_i - a_i}{\max(a_i, b_i)} \in [-1, 1]$$

- For the global score of a cluster/clustering scheme *compute the average score* over the cluster/dataset
- If the score is negative, it means that it's a bad clustering (since $b_i < a_i$). In particular:
  - it means that there is a dominance of *objects in other clusters* at a *smaller distance* than *objects of the same cluster*.

Inertia and silhouette scores



- Blue line is the inertia, while the orange line is the silhouette score.
- The inertia improvement is a lot until the number of cluster reaches 10, and then increasing the number of classes doesn't result in a lot of improvement.
  - Similarly, the silhouette score starts improving at 3 cluster, until it reaches 10 cluster, in which it declines.
  We can see also in the same dataset how Silhouette scores change.



Looking for the best number of clusters

- Some algorithms, such as K-means, require the number of clusters as a parameters.
- Measures, such as *SSE* and *Silhouette*, are obviously influenced by the number of clusters, thus they can be *used to optimize K*. But...
  - Computation of Silhouette score *is expensive*!
  - SSE decreases monotonically for increasing K.

Elbow method

For this reason, we must consider *inertia* as a possible parameter for deciding K.
The *inertia* varying K has frequently *one or more points where the slope decreases* (i.e. 10 clusters in the previous image): one of this points is frequently *a plausible value for* K.
This is called **elbow method**.

The *silhouette score* varying K has frequently *a maximum*, in this case it indicates the best value for K [so, even if it is very expensive, it can be used for small datasets].

Supervised evaluation

Gold standard

Consider a partition of a dataset *similar* to the data *to be clustered*, which we call **gold standard**, and defined by a *labelling scheme* $y_g(.)$.

- it is the same as the *labels attached to supervised data* for training a classifier.
- In clustering, using the gold standard is the same as doing training and test on two different datasets.
  - Meaning that I can cluster ignoring the labels first, and then recluster the same data with the labels and compare the results.
  - If there's a good agreement between the 2 methods, then we can use the model that we've used for clustering with other, unsupervised data.

Now, consider a *clustering scheme* $y_k(.)$.

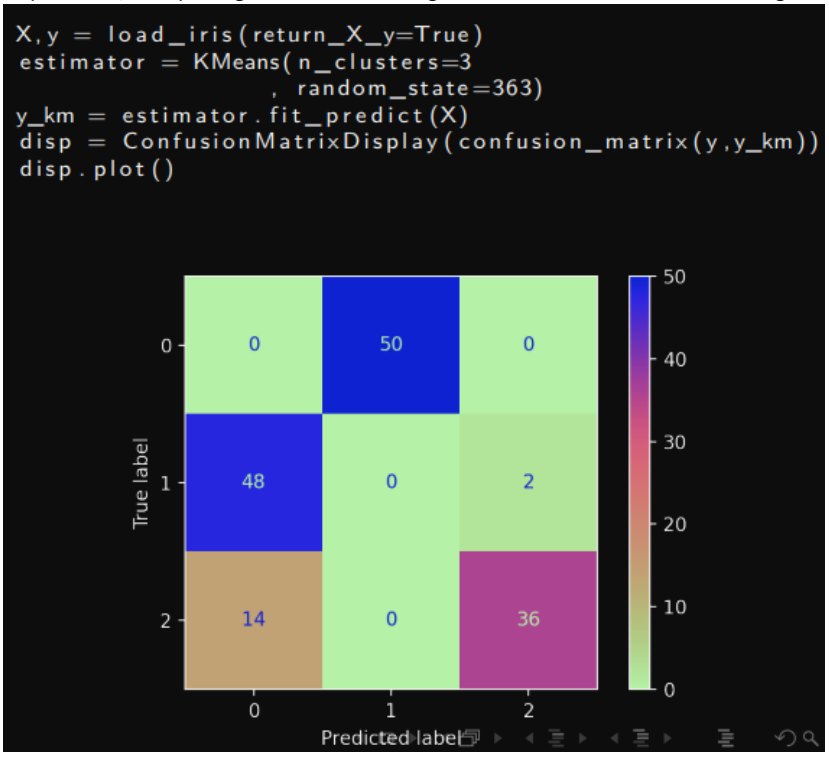
- the *cardinalities of the sets of* ***distinct*** *labels* generated by the two schemes V_g and V_k *can be different*, and also in case of identity of the two grouping schemes, a permutation of labels could be necessary to make them equal.

The gold standard technique can be used to *validate a clustering technique*, which can be applied later to new, unlabeled data.

- the purpose is quite similar to testing a classifier. The difference is that in this case *we are more interested in grouping new data* than in labelling them following the Gold Standard scheme.

In practice, comparing a cluster with a gold standard is similar to testing a classifier.

```
X,y = load_iris(return_X_y=True)
estimator = KMeans(n_clusters=3
                   , random_state=363)
y_km = estimator.fit_predict(X)
disp = ConfusionMatrixDisplay(confusion_matrix(y,y_km))
disp.plot()
```



What is shown by this image is the confusion matrix obtained through the gold standard procedure. Strangely, the large values are not on the diagonal (like in normal confusion matrices). This is because the labels are assigned randomly by the gold standard, and are essentially different from the original values of the labels.

- but, if we remap the values obtained, we will obtain a standard confusion matrix.

Similarity oriented measures

Consider again a *gold standard* $y_g(.)$ and *clustering scheme* $y_k(.)$.
Any pair of objects can be labelled in 4 different ways:

- *SGSK* if they belong to the same set in $y_g(.)$ and $y_k(.)$
- *SGDK* if they belong to the same set in $y_g(.)$ and not in $y_k(.)$
- *DGSK* if they belong to the same set in $y_k(.)$ but not in $y_g(.)$
- *DGDK* if they belong to different sets both in $y_g(.)$ and $y_k(.)$

In a optimal situation (the perfect assignment), the number of elements labelled SGDK or DGSK should be 0.

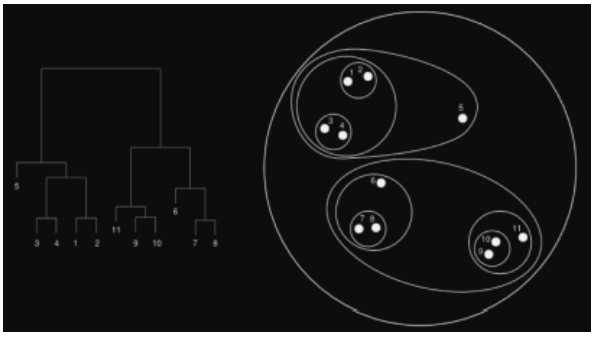
Hierarchical clustering

Another clustering methods. It generates a nested structure of clusters. It can be of two types:

- **Agglomerative (bottom up)**: as a starting state, *each data point is a cluster*.
  - in each step *the two less separated clusters are merged into one*
  - a measure of separation between clusters is needed
- **Divisive (top down)**: as a starting state, the *entire dataset is the only cluster*.
  - in each step, *the cluster with the lowest cohesion is split*
  - a measure of cluster cohesion and a split procedure are needed

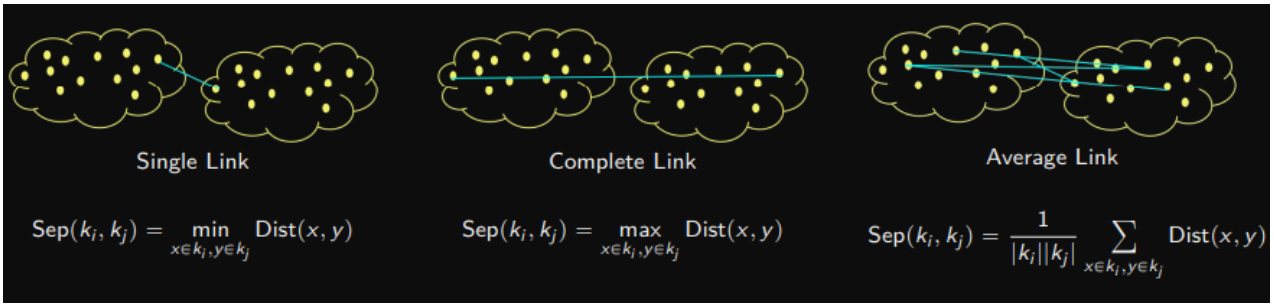
We won't see the divisive approach.

Output of hierarchical clustering



- Dendrogram (left)
- Nested cluster diagram (right)
- They represent the same structure
- The representation is the same for agglomerative and divisive.

Separation techniques in hierarchical clustering



- In *single link*, we measure that the separation between the two clusters as the *minimum distance* between *pairs of objects* in the two clusters.
- In *complete link*, we measure that the separation between the two clusters as the *maximum distance* between *pairs of objects* in the two clusters.
- In *average link*, we measure that the separation between the two clusters as the *average distance* between *pairs of objects* in the two clusters.

Other separation techniques

We could use the distance between the centroids, or:

- **Ward's method**: given two sets with the respective SSE, the separation between the two is measured as the *difference* between the *total SSE* resulting *in case of a merge* and *the sum or the original SSEs*;
  - *smaller separation* implies a *lower increase* in the SSE after *merging*.
  - i.e. Considering two clusters 1 and 2: $SSE_{1,2} - (SSE_1 + SSE_2)$.

There's a nice example of agglomerative hierarchical clustering in slides 78 here.

Single linkage algorithm

Algorithm for clustering based on agglomerative hierarchical clustering, using *single link*.

What we realize for this algorithm is that we need to compute *a lot of distances*.
So, we can compute a *distance matrix*, so that we can lookup the distances without computing them at each step.

Here's the algorithm:

- While the number of clusters is greater than 1
  - find the *two clusters* with the *lowest separation*, say k_r and k_s
  - *merge them* in a cluster
  - *delete* from the distance matrix *the rows* and columns r and s and *insert one new row* and *column* with the *distances of the new cluster from the others*.
  - The new distance is computed as:

$$\text{Dist}(k_k, k_{(r+s)}) = \min(\text{Dist}(k_k, k_r), \text{Dist}(k_k, k_s)) \forall k \in [1, K]$$

Complexity of SLA

- Space and time: $O(N^2)$ for the computation and the storage of the *distance matrix*.
- Worst case: $N - 1$ iterations to reach the final single cluster.
- For the i-th step of the main iteration:
  - search of the pair to merge $O((N - i)^2)$
  - recomputation of the distance matrix $O(N - i)$
- Time, *in summary*: $O(N^3)$. Can be reduced to $O(N^2 log(N))$ with indexing structures.

There's a nice example of clustering Italian cities in slides 86 here, as well as another example using animals (and complete linkage!).

Density based clustering

It overcomes another problem of K-Means, namely the fact that it is not very good with convex datasets (hyperspherical clusters).

In density based clusterings, clusters are *high–density regions* separated by *low–density regions*.

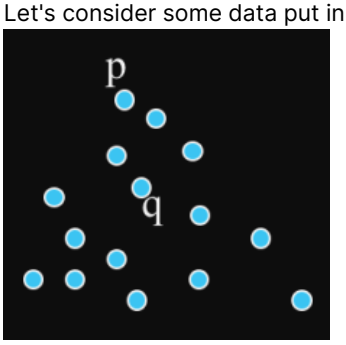
How can we define *density*? The two most obvious solutions are:

- **Grid–based**
  - split the (hyper)space into a *regularly spaced grid* (in hypercubes essentially)
  - *count* the number of objects *inside each grid element*
- **Object–centered**
  - *define the radius* of a (hyper)sphere (a radius of interest).
  - *attach to each object* the *number of objects* which are *inside that sphere*.

Ok nice, but how do we actually define *the value* for which a set of values is considered dense? Well, it depends on the dataset...

DBSCAN – Density Based Spatial Clustering of Applications with Noise

Let's consider some data put in this way:



In this set, intuitively, p is a border point, while q is a core point.
We define a ==**radius** ϵ ==and define as *neighborhood of a point* the ϵ–*hypersphere* centered at that point.
Points p and q are one in the neighborhood of the other: neighborhood is symmetric.

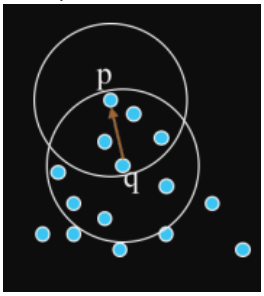
We then define a threshold `minPoints` and define as *core* a point with *at least* `minPoints` *points in its neighborhood*, as *border* otherwise.

Direct density reachability

We define that a point p is **directly density reachable** from point q if:

- q is core
- q is in the neighborhood of p

So, a core point can attract a border point that is in its neighborhood. And so, we want to build clusters starting from core points and collecting companions that are in the neighborhood.



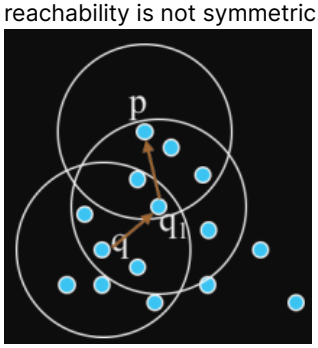
- *direct density reachability is not symmetric*: in the example q is not directly density reachable from p, since p *is border*.
  - a border point cannot reach anyone.

Density reachability

A point p is **density reachable** from point q iff:

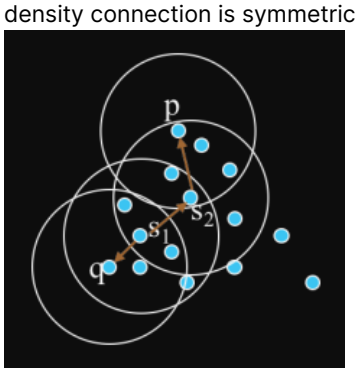
- q is core
- there is a sequence of points point q_i such that q_{i+1} is directly density reachable from q_i , $i \in [1, nq]$, q_1 is directly reachable from q and p is directly density reachable from q_{nq}.
- in short: there's a sequence of directly reachable core points from q to p.

- reachability is not symmetric.



Direct connection

- a point p is *density connected* to point q iff there is a point s such that p and q are density reachable from s.
- density connection is symmetric



Generation of clusters

In this context, a *cluster* is a *maximal set of points* connected by *density*.
Border points which are *not connected* by density *to any core* point *are labelled as noise*.

Of course, if there are outliers, there might be a problem (connection between 2 clusters).

How to set ϵ and `minPoints`?

- As in many other machine learning algorithms, a **grid search** over *several combinations* of *hyperparameters* can be useful.
- As a rule of thumb, you can try $minPoints = 2 * D$, the number of dimensions.
- Noise suggest an increase in minPoints
- A guess for ϵ requires more effort, considering the distance of the k–nearest neighbor, with $k = minPoints$
- if I increase ϵ, it means that I will look further, and some border points will become core points.
  - we have this same situation if instead of increase ϵ I decrease `minPoints`.
- Decreasing ϵ and increasing `minPoints` *reduces the cluster size* and increases the number of *noise points*.

Good guess for ϵ

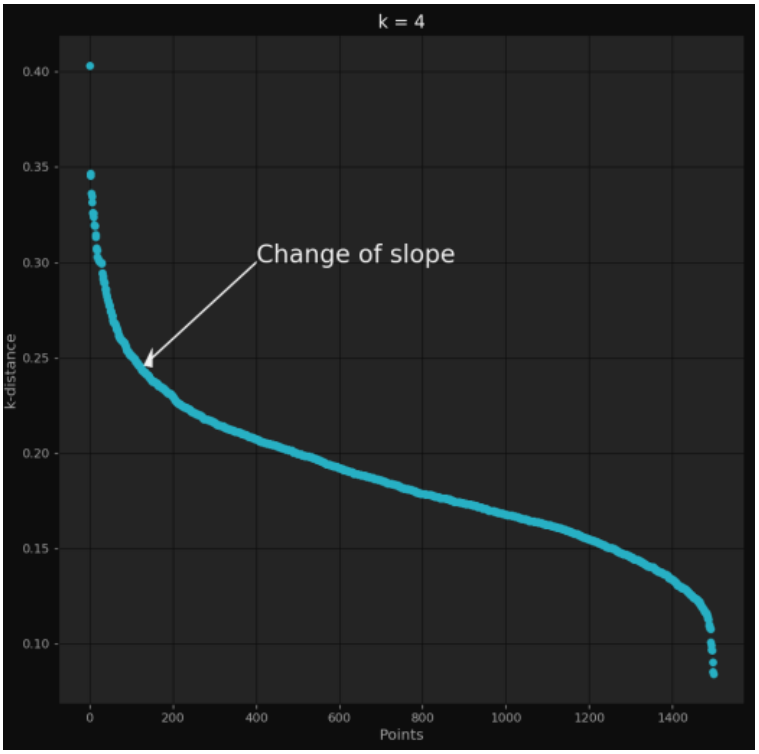
Consider the vector of the **k-distances**:

- choose k

- for each point we compute *the distance of its k–nearest neighbor*, and we sort the points for *decreasing k–distance*.

- Choosing a given k–distance as ϵ, it turns out that all the points with a *k–distance bigger than* ϵ will be considered as *border*.

Usually, datasets which exhibit some tendency to clustering exhibit also a change of slope. The best ϵ can be found with a *grid search* in the *area of the change of slope*.



DBSCAN, PROS AND CONS

Pros

- Finds clusters of any shape
- Is robust w.r.t. noise

Cons

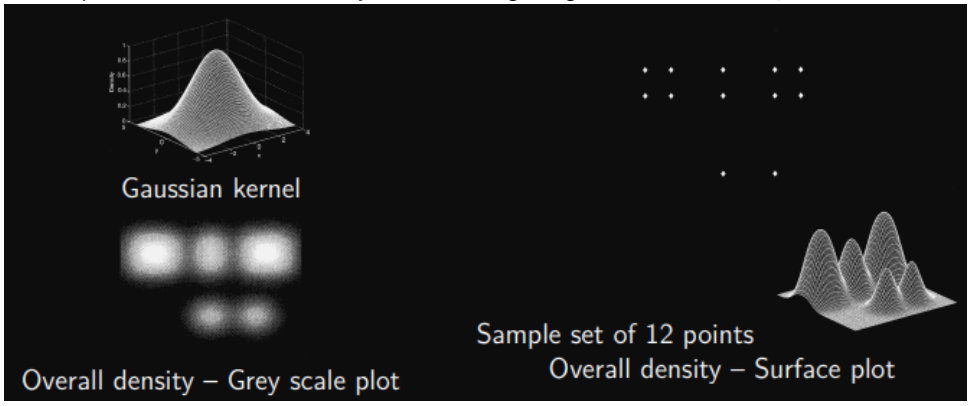
- Problems if clusters have widely varying densities

Complexity is $O(N^2)$, reduced to $O(Nlog(N))$ if we use spatial indexes.

Kernel Density Estimation (KDE)

- The overall density function is the sum of the *influence functions* (or **kernel functions**) *associated with each point*.
  - The kernel function must be symmetric and monotonically decreasing
  - usually has a parameter to set the decreasing rate
- It's very good for vector spaces.

- KDE in particular is used as a way of smoothing a signal (or in this case, a noised dataset)



DENCLUE algorithm

1. Derive a density function for the space occupied by the data points
2. Identify the points that are local maxima
3. Associate each point with a density attractor by moving in the directions of maximum increase in density
4. Define clusters consisting of points associated with a particular density attractor
5. Discard clusters whose density attractor has a density less than a user–specified threshold ξ
6. Combine clusters that are connected by a path of points that all have a density of ξ or higher

[not so important, the professor skimmed through it.]

DENCLUE comments

Pros

- It has a strong theoretical foundation on statistics precise computation of density.
  - DBSCAN is a special case of DENCLUE where the influence is a step function
- Good at dealing with noise and clusters of different shapes and sizes

Cons

- expensive computation $O(N^2)$ (can be optimized with approximated grid based computation)
- Troubles with high dimensional data and clusters with different densities

---

This Lesson was a fucking pain to listen. It was very difficult to understand some of the meanings and the concepts were pretty complex.

2022-11-23 - Statistic based clustering, Association rules

Model based (or statistic based) clustering

The clustering models that we've seen so far are do not make assumptions on the distribution of data, so they work either consider distances or look around etc...

Model based clustering focuses on *estimating the parameters* of a statistical model to *maximize* the ability of the model to *explain the data*.

The main technique is to use the *mixture models* (i.e. Gaussian models). So, we view the data as a set of observation from a mixture of different probability distributions.

- i.e., we could have a series of points in a space, that could be approximated by two different Gaussians for axis x and y respectively

- Or, we could have a series of points in a single line expressed as two Gaussian in sequence, we'll see that more clearly in this example.

Usually, the base model is a *multivariate normal distribution*.
The estimation is usually done using the *maximum likelihood*: given a set of data X, the *probability of the data*, in function of the parameters of the model, is called a *likelihood function*.

- Means: find the parameters which maximize the likelihood of the model w.r.t. the data(??).
  Attributes are assumed to be *random independent variables*.

[These methods do not require the number of clusters as an input :D]

Gaussian Mixture a.k.a. Expectation Maximization (EM)

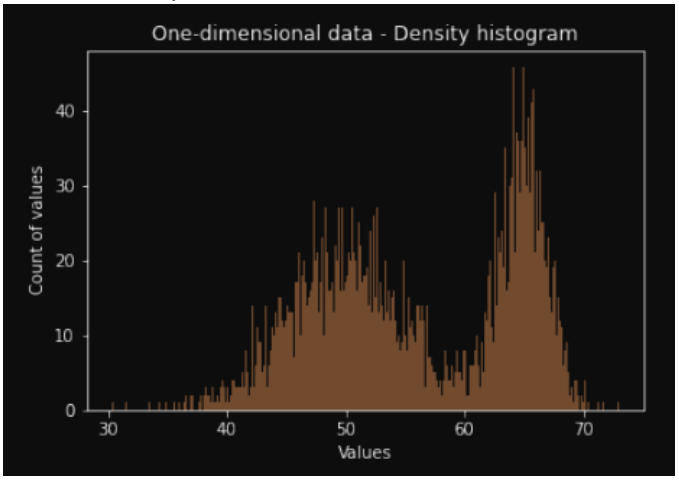
If the data can be approximated by a single distribution, the derivation of the parameters is straightforward.
In the general case, with many mixed distributions, the EM algorithm is used.

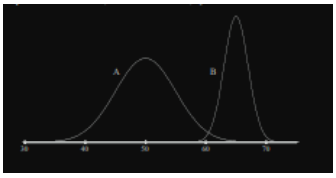
Here's the actual algorithm:

1. Select an initial set of model parameters
2. repeat:
   1. **Expectation Step** – For each object, calculate the *probability* that *each object belongs to each distribution*.
      - i.e., if we have a graph with 3 Gaussians, we compute the probability of belonging in each model.
   2. **Maximization Step** – Given the probabilities from the expectation step, *find the new estimates of the parameters* that *maximize the expected likelihood*.
3. until – the parameters do not change (or the change is below a specified threshold)

[Notice: this algorithm is very much similar to k-means, but in practice are completely different.]

Here's an example. We have one dimension of data with a mixture of two models.





If we have two clusters of values, like in the example, we need to estimate *5 parameters*:

- *mean* and *standard deviation* for cluster A
- mean and standard deviation for cluster B
- *sampling probability* p for cluster A (meaning, the probability of belonging to cluster A).

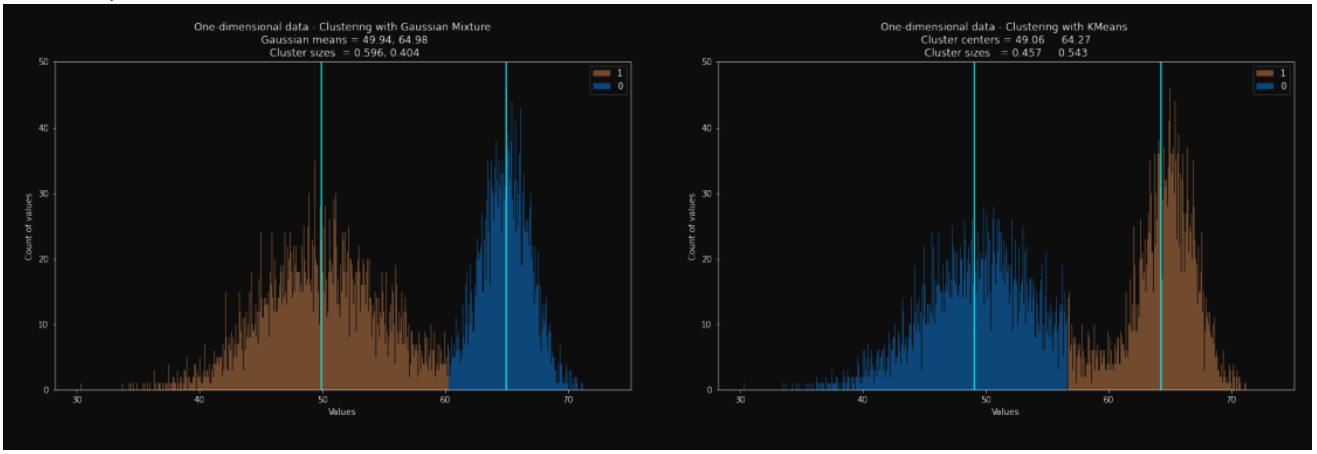
$$\Pr(A \mid x) = \frac{\Pr(x \mid A)\Pr(A)}{\Pr(x)} = \frac{f(x;\mu_A,\sigma_A)p_A}{\Pr(x)}$$

$$f(x;\mu_A,\sigma_A) = \frac{1}{\sqrt{2\pi\sigma}} e^{\frac{(x-\mu)^2}{2\sigma^2}}$$

In this case, the algorithm does something like this:

- Repeat until convergence
  - **Expectation**: Compute p_A and p_B using the current distribution parameters.
    - Compute the numerators for $Pr(A|x)$ and $Pr(B|x)$ and normalize dividing by their sum
  - **Maximization** of the distribution likelihood given the data
    - Compute the *new distributions parameters* (i.e. μ, σ...), *weighting the probabilities* according to the current distribution parameters.
- After convergence, *label each object* with A or B according to the *maximum probability*, given the last distribution parameter.

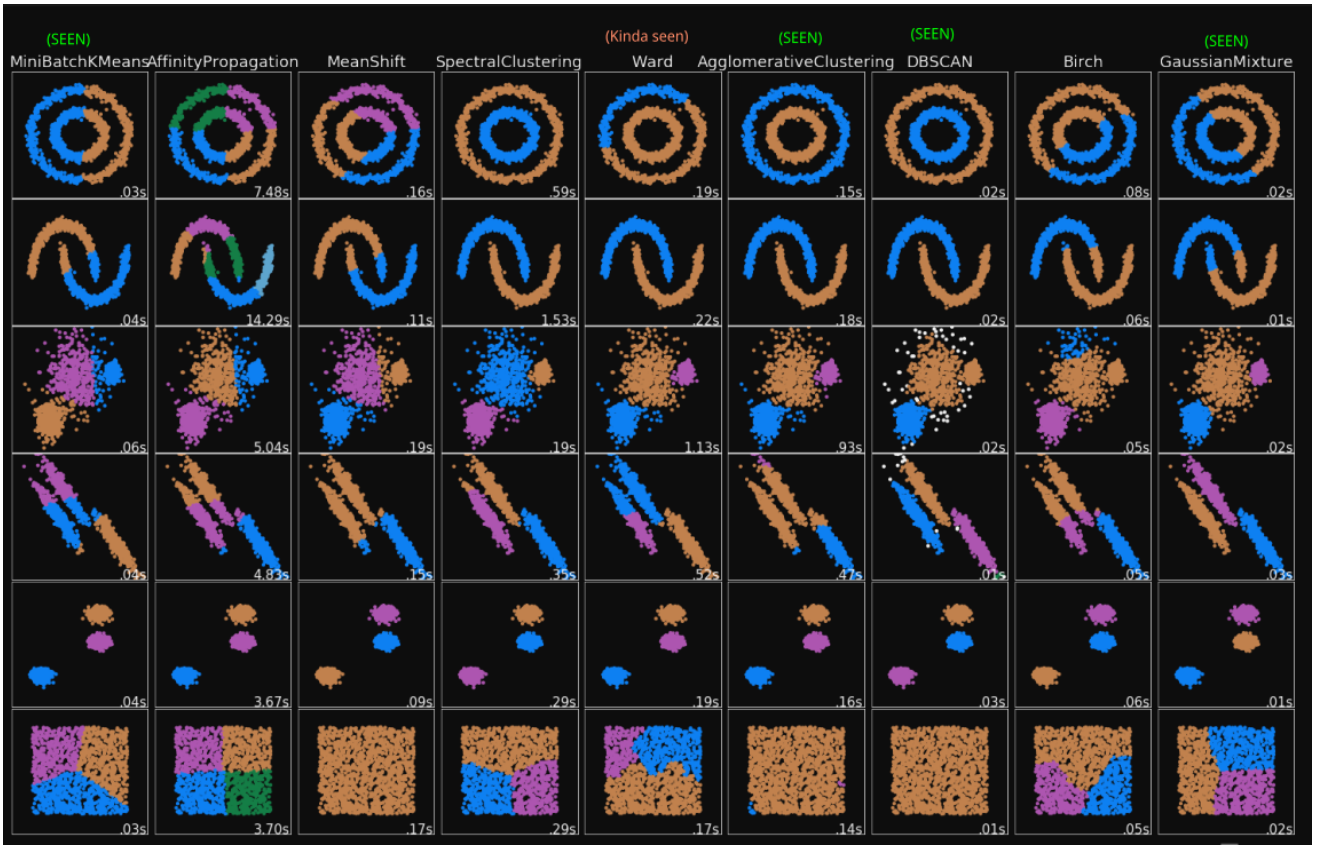
Let's compare this method with K-means:



On the left, we have clustering with EM, while on the right we have clustering with K-means.

Since these data have a Gaussian–like distribution, and the EM algorithm is founded on the hypothesis of modelling data with Gaussians, it will of course be better.
Thus, since K-Means is non–parametric, in this case its performance is worse.

Final remarks



Clustering types

- *Partitioning*: iteratively find partitions in the dataset, optimizing some quality criterion.
- *Hierarchic*: recursively compute a structured hierarchy of subsets.
- *Density based*: compute densities and aggregates clusters in high density areas.
- *Model based*: assume a model for the distribution of the data and find the model parameters which guarantee the best fitting to the data.

Clustering scalability

Effectiveness decreases with

- dimensionality D
- noise level
  Computational cost increases with
- dataset size N, at least linearly
- dimensionality D

Association Rules

It is an unsupervised kind of activity, and it used for making prediction on a transactional database.

Market basket example

Given a set of commercial transactions, find *rules* that *will predict* the *occurrence of an item* based on the *occurrences of other items* in the transaction.

Our table representing commercial transaction looks something like this:

TID	*Items*
1	Bread, Milk
2	Beer, Bread, Diaper, Eggs
3	Beer, Coke, Diaper, Milk
4	Beer, Bread, Diaper, Milk
5	Bread, Coke, Diaper, Milk

Market Basket Transactions

It looks like a relational database table.

- TID is the transaction id. Not useful for learning, but only for preprocessing.

Some examples of association rules are:

- {Diaper} → {Beer}
  - Meaning, if there is a diaper, there will probably also beer.
- {Bread, Milk} → {Coke, Eggs}
- {Beer, Bread} → {Milk}

> ⚠ Warning
>
> These are **not** logical rules, meaning that even if they are not true, they still hold some kind of weight or importance. Meaning, association rules are *quantified*, and not just true or false .
> For this reason, we consider the *strength* of each rule.

Some definitions

- **Itemset**: a collection of one or more items
  - Example: {Bread, Diaper, Milk}
- **k-itemset**: an itemset that contains k items
- **Support count** (σ): *frequency of occurrence* of an itemset
  - E.g. σ({Bread, Diaper, Milk}) = 2
- **Support**: fraction of transactions that contain an itemset
  - E.g. σ({Bread, Diaper, Milk}) = 2/5
- **Frequent Itemset**: an itemset whose support is greater than or equal to a *minsup* threshold
  - (simply, an itemset bigger than a threshold)
- **Association Rule**: an expression of the form A ⇒ C, where A and C are itemsets
  - A = Antecedent and C = Consequent
  - Example: {Diaper, Milk} → {Beer}
- **Rule Evaluation Metrics**:
  - *Support* (sup): fraction of the N transactions that *contain both* A and C.
  - *Confidence* (conf): measures *how often* all the items *in C appear in transactions* that *contain A*.

$$sup = \frac{\sigma(Beer, Diaper, Milk)}{N} = \frac{2}{5} = 0.4$$
$$conf = \frac{\sigma(Beer, Diaper, Milk)}{\sigma(Milk, Diaper)}$$

Confidence is always equal to the support or bigger [he said bigger but I think he meant lower].

Some considerations

- Rules with *low support* can be generated by *random associations*
- Rules with *low confidence* are *not really reliable*
- Nevertheless a rule with relatively low support, but high confidence can represent an uncommon but *interesting* phenomenon.
- i.e. caviar and champagne. It has small support, but big confidence, since they are not bought often, but when they are, they are always bought together.

Association Rule Mining Task

Given a set of transactions N, the *goal* of association rule mining is to find all rules having:

- support $\geq$ *minsup* threshold
- confidence $\geq$ *minconf* threshold
  These are the rules we deem interesting.

So, the next step will be to find the thresholds.
A possible approach would be the brute-force approach.

- **Brute-force approach**:
  - List all possible association rules
  - Compute the support and confidence for each rule
  - Prune rules that fail the minsup and minconf thresholds

Example of Rules:

$$\begin{aligned} \{Diaper, Milk\} &\Rightarrow \{Beer\} && (s = 0.4, c = 0.67) \\ \{Beer, Milk\} &\Rightarrow \{Diaper\} && (s = 0.4, c = 1.0) \\ \{Beer, Diaper\} &\Rightarrow \{Milk\} && (s = 0.4, c = 0.67) \\ \{Beer\} &\Rightarrow \{Diaper, Milk\} && (s = 0.4, c = 0.67) \\ \{Diaper\} &\Rightarrow \{Beer, Milk\} && (s = 0.4, c = 0.5) \\ \{Milk\} &\Rightarrow \{Beer, Diaper\} && (s = 0.4, c = 0.5) \end{aligned}$$

All the rules represents all the possible rules that we can obtain with the itemset
{Beer, Diaper, Milk}, given our table.
Notice that rules originating from the same itemset have identical support but can have different confidence

- we may decouple the support and confidence requirements.

This idea allows us to create a sort of algorithm for association rules:

- Two-step approach:
  1. *Frequent Itemset Generation*:
     - Generate all itemsets whose support is greater than minsup.
  2. *Rule Generation*:
     - Generate high confidence rules from each frequent itemset, where each rule is a binary partitioning of a frequent itemset.

But, the *frequent itemset generation* is still *computationally expensive*, we need to reduce the complexity of this procedure.

Frequent Itemset Generation

Given D items, there are $M = 2^D$ possible candidate itemsets.
The idea is to *reduce* the *number of candidates* M, by using pruning techniques.
Or, we can reduce the number of comparisons NM.

Reducing Number of Candidates

Apriori principle: if an *itemset is frequent*, then *all of its subsets* must also be *frequent*.

$$\forall X, Y : (X \subseteq Y) \Rightarrow sup(X) \geqslant sup(Y)$$

This means that the *support of an itemset* never exceeds *the support of its subsets*.

2022-11-25 - Algorithms in Association rules

Frequent Itemset Generation

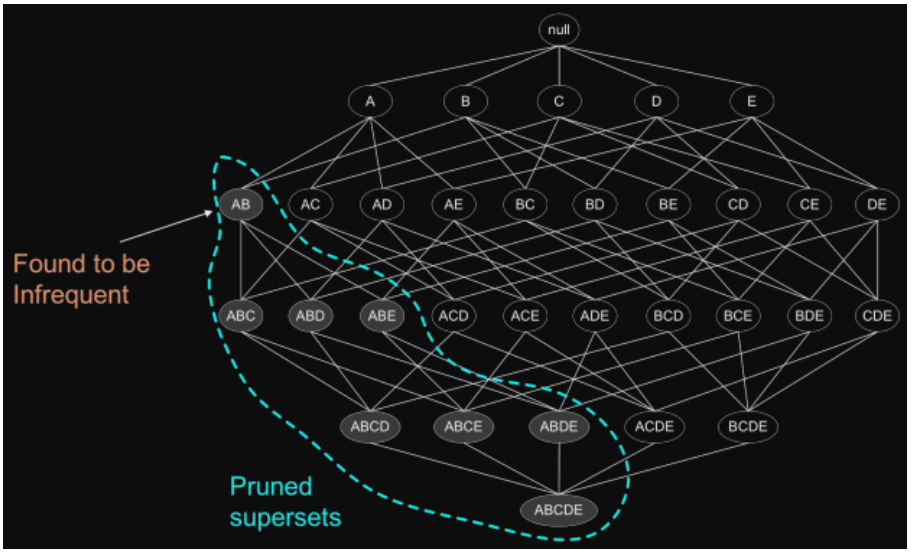
Given D items, there are $M = 2^D$ possible candidate itemsets.
The idea is to *reduce* the *number of candidates* M, by using pruning techniques.
Or, we can reduce the number of comparisons NM.

Reducing Number of Candidates

Apriori principle: if an *itemset is frequent*, then *all of its subsets* must also be *frequent*.

$$\forall X, Y : (X \subseteq Y) \Rightarrow sup(X) \geqslant sup(Y)$$

This means that the *support of an itemset* never exceeds *the support of its subsets*.



In this tree, the elements on top are contained in the elements at the bottom.
Based on the a-priori principle, if a subset of an item is not frequent, then its containing sets must also be infrequent.

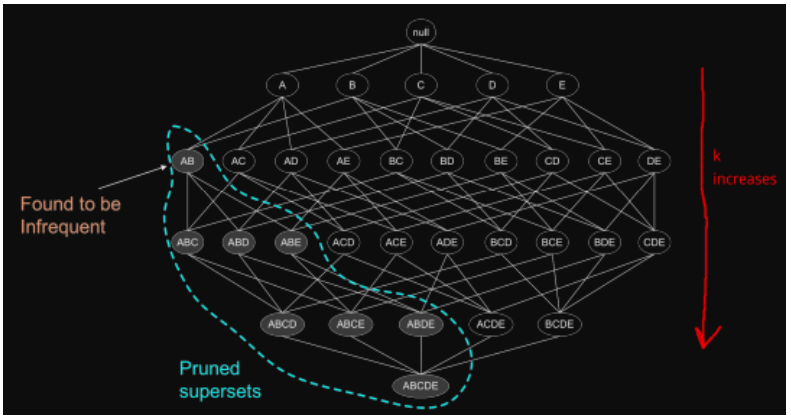
So, it is pretty much intuitive to see what we have to prune.

Apriori algorithm

Pruning Candidate generation

Some definitions:

- C_k : candidate itemsets of size k.
- L_k : frequent itemsets of size k.
  - we have C_k, and we are looking for L_k.
- $subset_k(c)$: set of the subsets of c with k elements.



Join step

- Let L_k be represented as a table with k *columns* where each *row* is a *frequent itemset*.
- Let the items in each row of L_k be in lexicographic order
  - This is because we want to generate a node. Generating a node, in this case, means making a query to the dataset.
- C_{k+1} is generated by a self join of L_k. In particular, this is the query that will generate the next candidates C_{k+1}.

[This is a complicated explanation for a stupid concept, you have been warned.]

```
insert into C_{k+1}
select p.item_1, p.item_2, ... , p.item_k, q.item_k
from L_k p, L_k q
where p.item_1=q.item_1 and ... and p.item_{k-1}=q.item_{k-1}
  and p.item_k < q.item_k;
```

Suppose that in the graph image that we've seen before, AB is infrequent (like is shown by default in the image). L_2 then will contain {AC, AD, AE, BC ...}.
Now, from AC, we will need to generate ABC, ACD and ADE. How can we do this?

- Note: meaning, how can we generate ABC from AC and AD in a literal sense.

We will need to insert in C_3 the elements through a *self join*, with L_2 with itself, so that the first $k-1$ elements are all equal in the join, and then I add the elements at the end.

- i.e. AB and AC → A = A, B < C → ABC (I don't want to generate ACB because it will be useless, the sets will be equal).
  - [I would have used a full join generates a table with *no repeating rows*, so it always generates a single set, without giving importance to the lexicographic order. It would have been less complicated in my humble opinion]

Prune step

Each $(k+1)$-itemset *which includes* a k-itemset which *is not in* L_k is *deleted* from C_{k+1}:

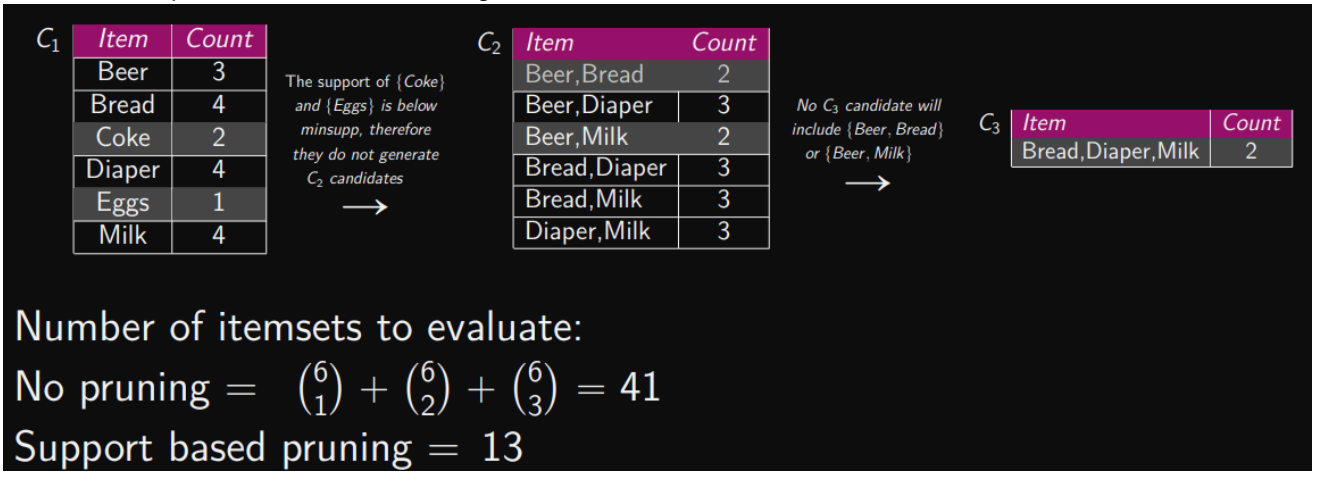
- for all $c \in C_k$ do
  - for all $s \in subset_{k-1}(c)$ do
    - if $s \in L_{k-1}$ then
      - delete c from C_k
- return C_k

For example: Even though ABC has been deemed as infrequent, I could generate it again by another join (es. AC + BC) and thus it must be deleted from the new list of candidates.

Frequent itemset generation algorithm

- L_1 ← frequent 1–itemsets
- k ← 1
- while $L_k = \emptyset$ do
  - C_{k+1} = candidates generated from L_k
  - for all t transaction in database do
    - increment candidate count in C_{k+1} for candidates found in t
  - L_{k+1} ← $c \in C_{k+1} : sup(c) \geq minsup$ _(we will keep all the surviving candidates with sup
  - k ← $k+1$ *bigger than a threshold)*
- return k, L_k

Here's an example of the outcome of this algorithm:



Factors Affecting Complexity of Apriori

- Choice of minimum support threshold
- Dimensionality (number of items) of the data set
- Size of database
- Average transaction width

Rule generation

Confidence

The confidence of a rule can be computed from the supports:

$$conf(A \implies C) = \frac{sup(A \implies C)}{sup(A)}$$

Rule generation

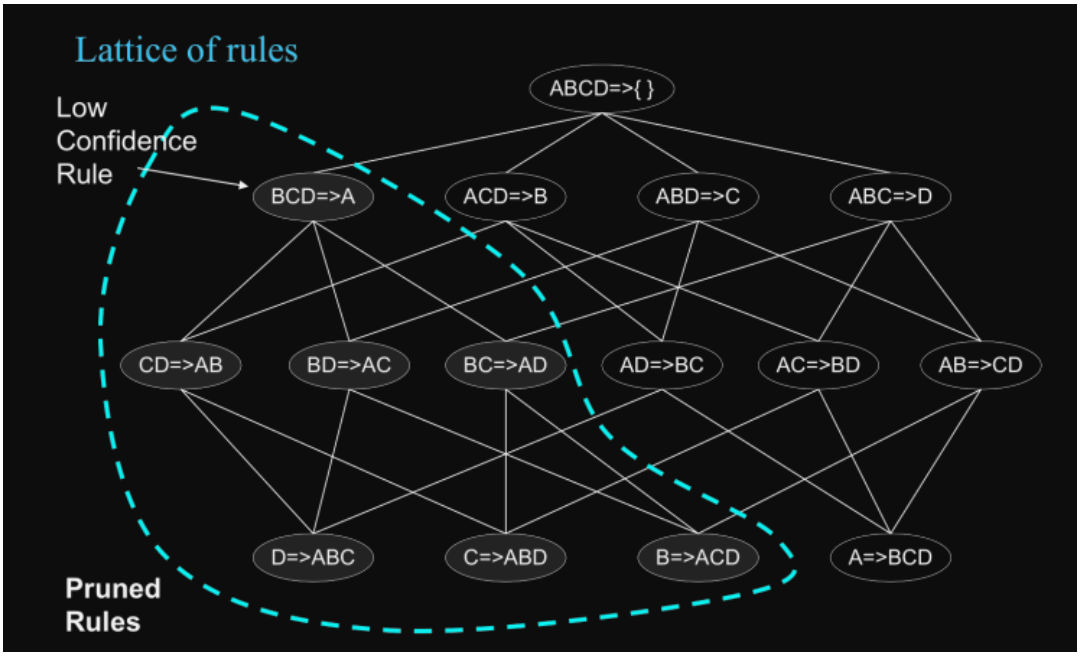
Find all the non-empty subsets $f \in L$ such that the confidence of rule $f \implies (L - f)$ is not less than the minimum confidence (set by the experiment designer)

- from $\{Beer, Diaper, Milk\}$ the possible rules are
  $Beer, Diaper \Rightarrow Milk$, $Beer \Rightarrow Diaper, Milk$,
  $Beer, Milk \Rightarrow Diaper$, $Milk \Rightarrow Beer, Diaper$,
  $Diaper, Milk \Rightarrow Beer$, $Diaper \Rightarrow Beer, Milk$
- if $|L| = k$ then *there are* $2^k - 2$ *candidate rules*
  - $L \implies \emptyset$ and $\emptyset \implies L$ can be ignored

These are a lot of rules. How to efficiently generate rules from frequent itemsets?

- In general, confidence does not have an anti-monotone property
  - $conf(ABC \rightarrow D)$ can be larger or smaller than $conf(AB \rightarrow D)$
  - But, let us consider rules generated from the same itemset:
    - e.g., $i = \{A, B, C, D\} \in L$:

      $$conf(ABC \rightarrow D) \geqslant conf(AB \rightarrow CD) \geqslant conf(A \rightarrow BCD)$$



Notice: since we know that all the children rules of that itemset will have a lower confidence (and thus we are not interested in them), *we can prune*!

- In general, we prune a rule D⇒ABC if *its subset* AD⇒BC *does not have high confidence*.
- We can also use the *support* $sup(.)$ for pruning.

Interestingness

Beside support and confidence, **Interestingness** measures can be used to prune/rank the derived patterns.

Computing Interestingness Measures

Given a rule A ⇒ C, the information needed to *compute rule interestingness* can be obtained from a *contingency table*.

	C	$\overline{C}$	
A	f_{11}	f_{10}	f_{1+}
$\overline{A}$	f_{01}	f_{00}	f_{0+}
	f_{+1}	f_{+0}	

Here's an example:

- $conf(Tea \Rightarrow Coffee) = \frac{sup(Tea, Coffee)}{sup(Tea)} = \frac{15}{20} = 0.75$
  - fairly high
- $\mathbf{Pr}(Coffee) = 0.9$ and $\mathbf{Pr}(Coffee \mid \overline{Tea}) = \frac{75}{80} = 0.9375$
  - despite the high confidence of $Tea \Rightarrow Coffee$, the absence of Tea increases the probability of $Coffee$
  - for this rule the confidence is misleading

	$Coffee$	$\overline{Coffee}$	
Tea	15	5	20
$\overline{Tea}$	75	5	80
	90	10	100

$Tea \Rightarrow Coffee$

We can't trust confidence!

- Note: this is a very specific case (coffee in itself has a *huge support* in this database), in general confidence is useful.

Other interestingness measures (from Statistics)

Lift

$$lift(A \Rightarrow C) = \frac{conf(A \Rightarrow C)}{sup(C)} = \frac{\mathbf{Pr}(A, C)}{\mathbf{Pr}(A)\,\mathbf{Pr}(C)}$$

- lift evaluates to 1 for independence
- insensitive to rule direction
- it is the ratio of true cases w.r.t. independence

Leverage

$$\begin{aligned} leve(A \Rightarrow C) &= \mathbf{Pr}(A, C) - \mathbf{Pr}(A) * \mathbf{Pr}(C) \\ &= sup(A \cup C) - sup(A)sup(C) \end{aligned}$$

- leverage evaluates to 0 for independence
- insensitive to rule direction
- it is the number of additional cases w.r.t. independence

Conviction

$$conv(A \Rightarrow C) = \frac{1 - sup(C)}{1 - conf(A \Rightarrow C)} = \frac{\mathbf{Pr}(A)(1 - \mathbf{Pr}(C))}{\mathbf{Pr}(A) - \mathbf{Pr}(A, C)}$$

- conviction is infinite if the rule is always true
- sensitive to rule direction
- it is the ratio of the expected frequency that A occurs without C (that is to say, the frequency that the rule makes an incorrect prediction) if A and C were independent divided by the observed frequency of incorrect predictions
- also called novelty

Intuition about measures

- higher support ⇒ rule applies to more records.
- higher confidence ⇒ chance that *the rule is true* for *some record* is higher.
- higher lift ⇒ chance that the rule is just a coincidence is lower.
- higher conviction ⇒ the rule is violated less often than it would be if the antecedent and the consequent were independent.
- Confidence is usually the base tool.
- Other measures can be used to test the results given by confidence and for *additional filtering*.

Transforming a relational DB into a transaction DB

A.k.a. transforming from multi-dimensional to mono-dimensional.

Let's consider a dataset deriving from sensors measuring the concentration of air pollutants:

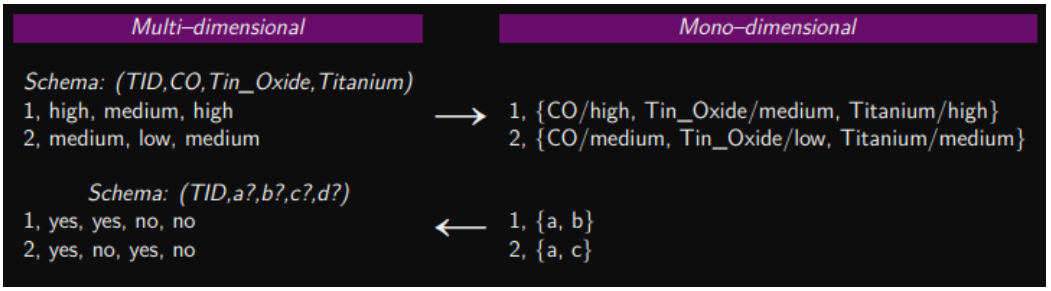
TID	CO	Tin_Oxide	Titanium
1	high	medium	high
2	medium	low	medium
3	medium	high	low
4	low	medium	medium

We could represent a transactional table from this DB by using, for each row, a tuple of booleans stating *if the value is present* or not (or we can use values in the in domain of the attribute).
Then:

- Look for rules such as *CO = high* and *Tin Oxide = high*, then *Titanium = high* (support 0.25 and confidence 1 in this dataset).
  - This way I can forecast data if, for example, values for a given sensors are not present.

Multi to mono

- Mono–dimensional (intra-attribute)
  - event: transaction
  - event description: items A, B, and C are together in a transaction
- Multi–dimensional (inter–attribute)
  - event: tuple
  - event description: attribute A has value a, attribute B has value b and attribute C has value c in a tuple.



[only the first part of this image is useful for us]

Quantitative attributes

What happens if there are quantitative attributes?

TID	CO	Tin Oxide	Titanium
1	2.6	1360	1046
2	2.0	1292	955
3	2.2	1402	939
4	1.6	1376	948

Simple, we use discretization, with the techniques that we've used so far.
In this way we have the same dataset as in the previous example.

THE END!