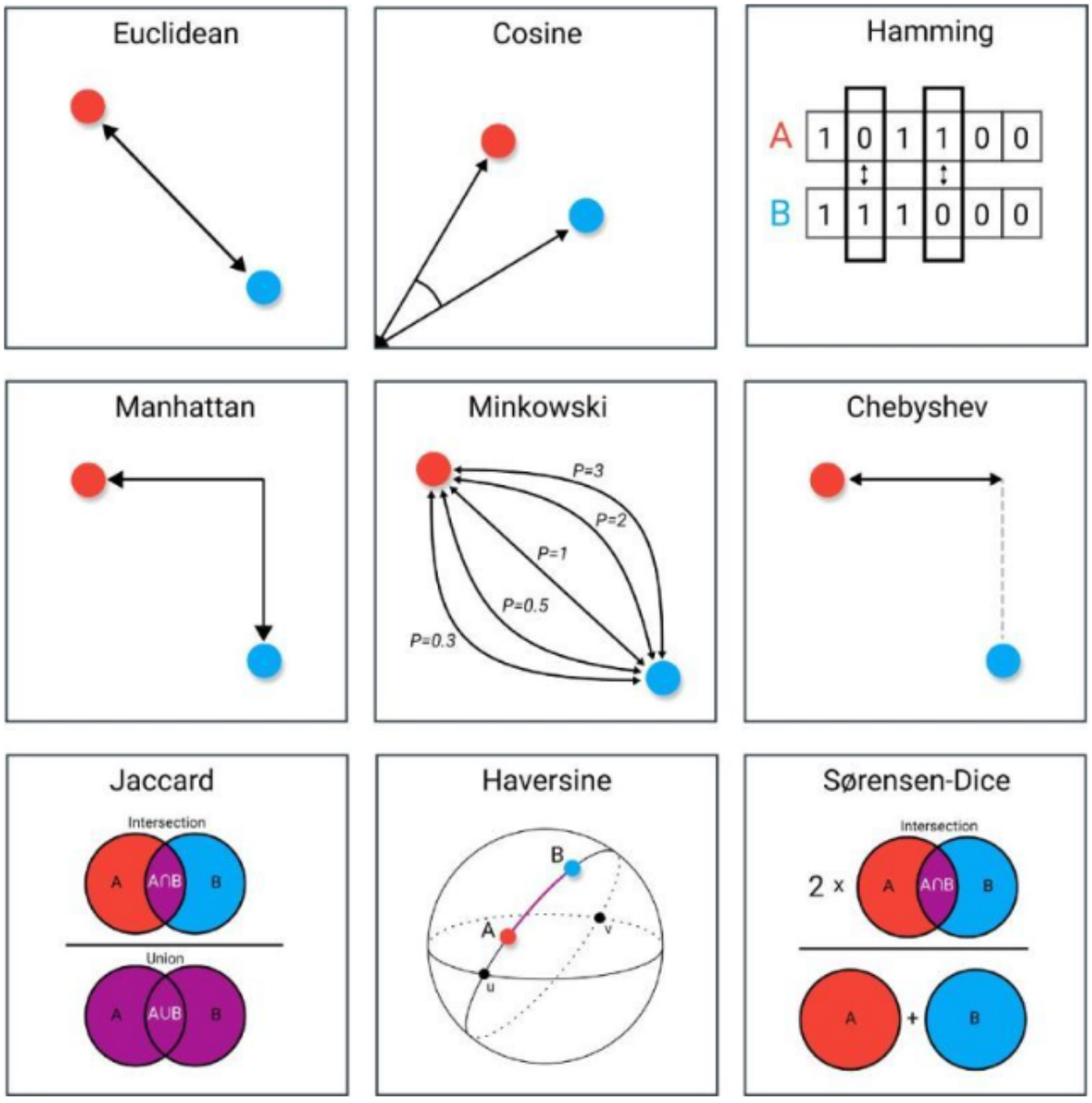
Lluís Giménez Gabarró

Theory. Statistical Learning

For the exam, there will be a questionnaire about this paper:
https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1005752

This is recommended to learn about PCA:
https://pubs.rsc.org/en/content/articlehtml/2014/ay/c3ay41907j

Distances (metrix) used in ML



1. Introduction to statistical learning

Machine (origin is computer science) or statistical (origin bio-statisticians) learning is a subfield of computer science that uses algorithms that learn from and make predictions on data without explicit programming.

As we said, ML learns from the data. So, how do we organize the data?

The idea is that we have 2 groups:
- Healthy people (control)
- Sick people

We want to predict if an individual is sick or not. To do this, we need data (numbers or factors) to train our model and make an accurate prediction, for example:
- Body temperature
- Heart rate
- Fatigue level
- …

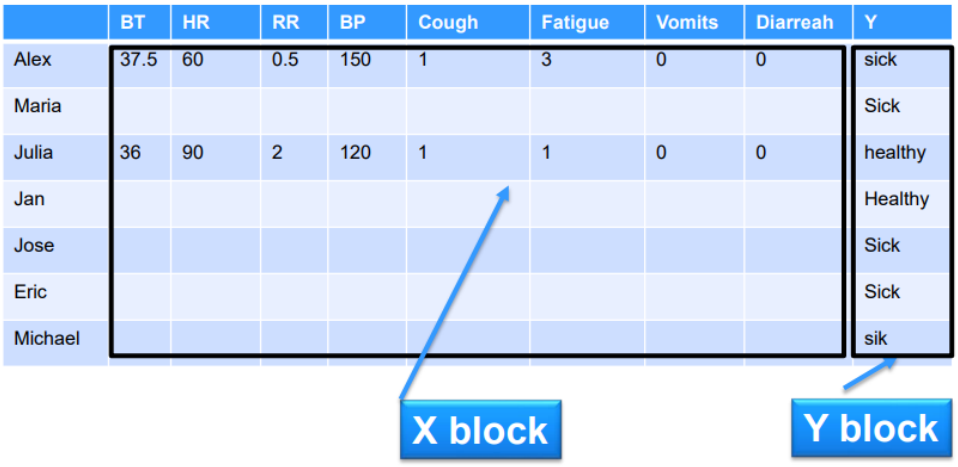
Pattern

So, we collect a large number of indicators about the health condition. We can put those indicators in a vector, which is the **pattern**.

A **pattern** is a composite of traits or features characteristic of an individual/sample, that are repeated. In classification tasks, a pattern is a pair of variables {x,ω} where:
- x is a collection of observations or features (feature vector)
- ω is the concept behind the observation (label)
    - Sick or Healthy

Feature table

	BT	HR	RR	BP	Cough	Fatigue	Vomits	Diarreah	Y
Alex	37.5	60	0.5	150	1	3	0	0	sick
Maria									Sick
Julia	36	90	2	120	1	1	0	0	healthy
Jan									Healthy
Jose									Sick
Eric									Sick
Michael									sik
Eva?	37	80	0.2	90	1	2	0	0	?



The feature table (training data) is going to be used to develop the predictive model:

- X block: Predictors that we use
- Y block: Things we want to predict (label)

At the end, your algorithm may be thought of as a mathematical function that takes as input the vector and provides an output. These functions can be called “mappings”.

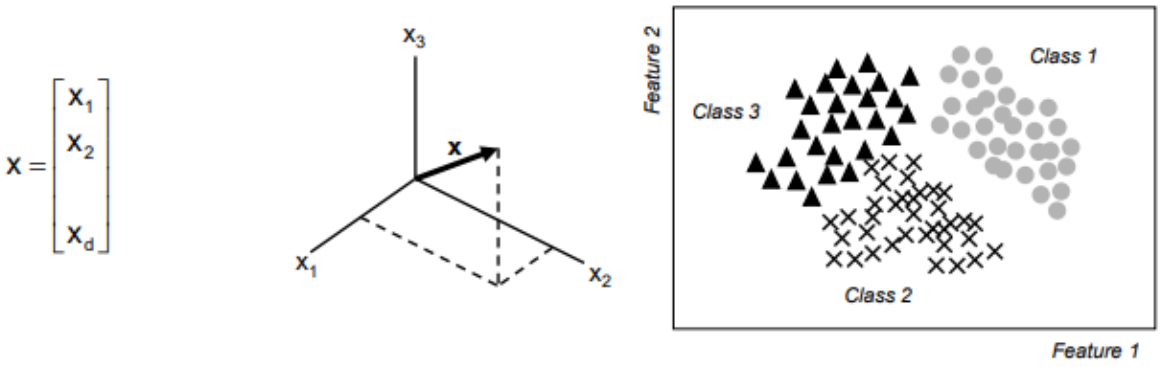
Feature

The feature vector is in a multidimensional space. This space can have a huge number of dimensions. Example:

- In microarrays we can analyze 10.000 genes. Meaning that the feature vector will have a huge dimensionality

We would like to visualize how the arrows of these vectors are distributed in space. Obviously, we can only visualize in 3D. Thus, we will need to reduce the dimensionality (remove features that are not informative and also because the algorithms work better with few dimensions, since algorithms have difficulties to learn when working with high dimensions).

The combination of **d features** is represented as a d-dimensional column vector called a feature vector .The d-dimensional space defined by the feature vector is called feature space. Objects are represented as points in feature space. This representation is called a scatter plot



Feature vector **Feature space (3D)** **Scatter plot (2D)**

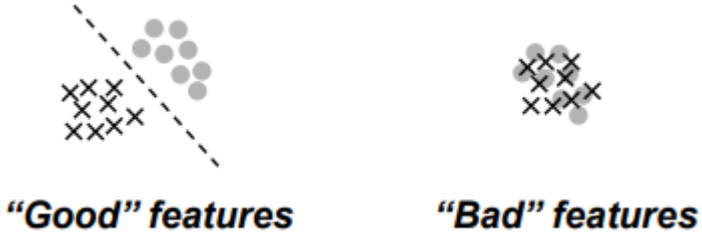
Given the scatterplot, the algorithm will do a partition of the space. Then, if a patient we want to predict falls in one of the partitions, we will assign a the label of that partition.

What makes a "good" feature vector? There are some features that are informative and others that are not. Example:

- To predict if someone is sick or not, one of the attributes could be if he is a Bioinformatic (which has nothing to do).

The quality of a feature vector is related to its ability to discriminate examples from different classes

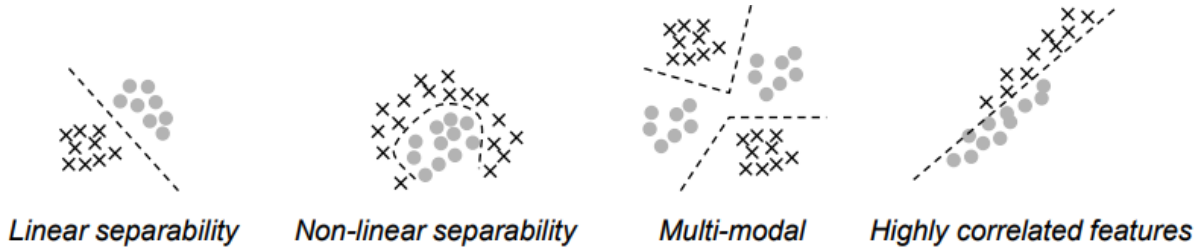
- Examples from the same class should have similar feature values
- Examples from different classes have different feature values



Most of the time, at the beginning of the study, we do not know which features are relevant and which are not. So, the algorithm process needs to identify which features are informative → We call this "Biomarker Discovery"

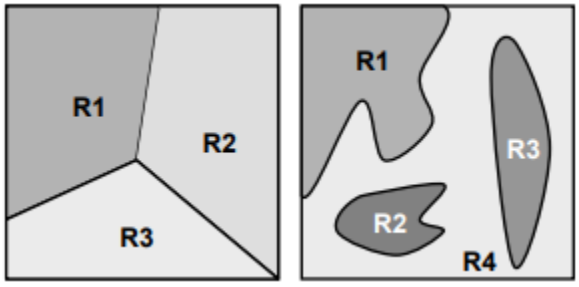
More feature properties

There are highly correlated features. Thus, if you know the concentration of one feature you do not need to know the other feature. In fact, we just need to use one feature since both features are equally informative.



Classifiers

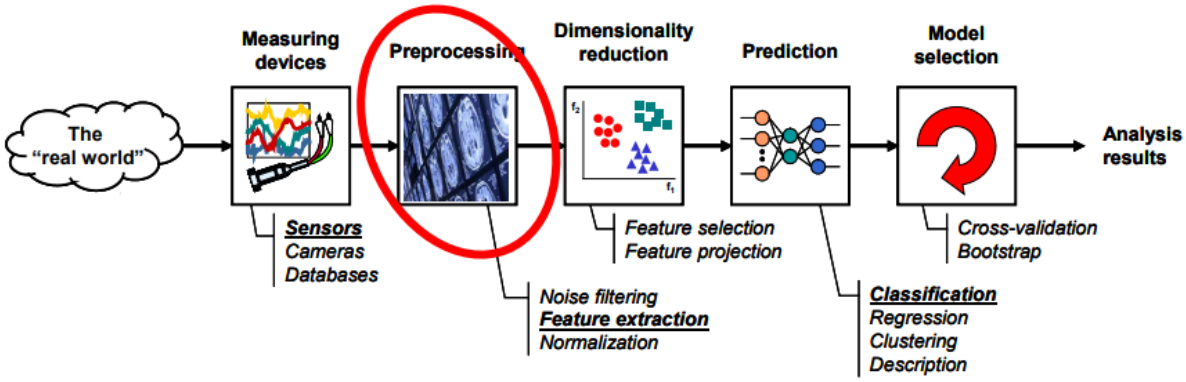
The goal of a classifier is to partition feature space into class-labeled decision regions. The borders between decision regions are called decision boundaries



Components of ML solution

The development of the ML predictive algorithm is a full pipeline

- Measuring Devices (GeneChip, LC-MS, Sequencing Machine, Imaging technique)
- A preprocessing mechanism (VIP!!!)
- A feature extraction mechanism (VIP!!)
- A classification algorithm (or quantitative predictor)
- A set of examples (training set) already classified or described



Philosophy of Predictive Models

There are 2 phases:

- **Model development:** Process of building the model
- **Deployment:** Once the model is ready, it is deployed. So, there will be a new sample to which we will apply the final algorithm and it will give an individual prediction.

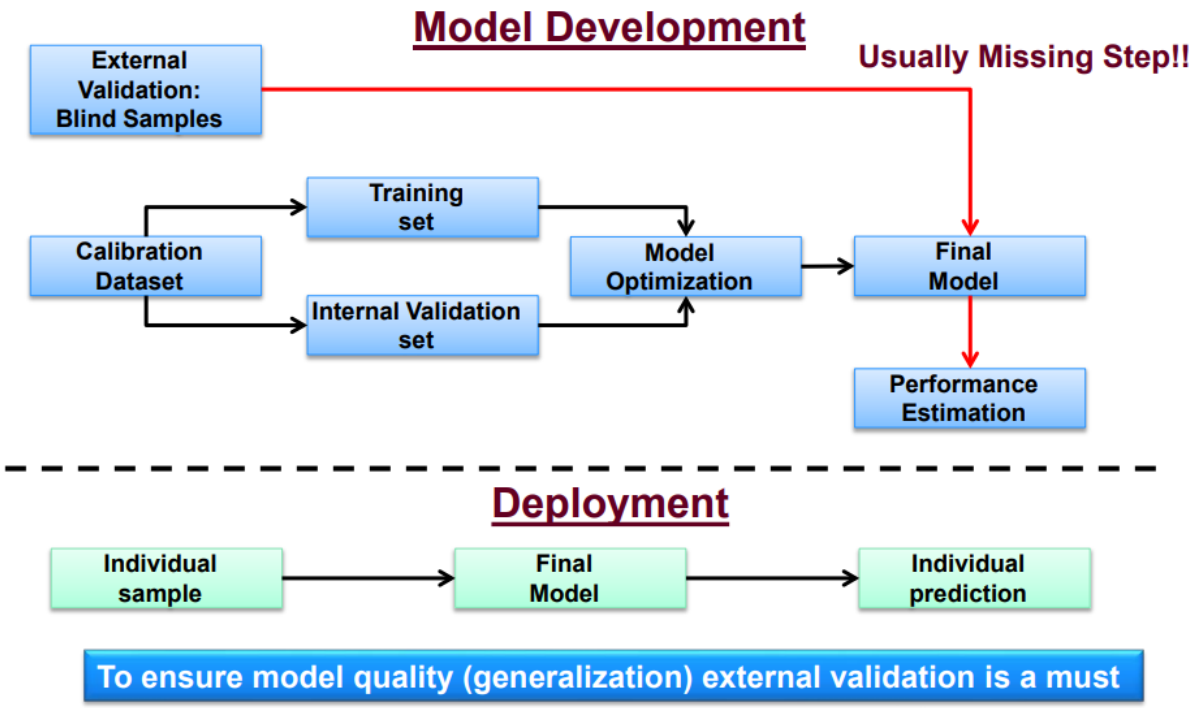
Model development

We have a dataset that will be divided into training and internal validation sets. With these sets, we will build a model.

- The internal validation set has the purpose of controlling the complexity of the algorithm. There are many algorithms and we have to decide if we need a simple or complex algorithm. Even if we are inside the same family of algorithms, we need to decide which are the optimal parameters.
  If the algorithm is too simple or complex for our data, it will not perform correctly.

  So, these 2 sets will be used to train our model and to optimize it.

Once we have the final model, we need to externally validate (cross-validation) it with more data that has not been used during the calibration dataset. Then we make a performance estimation. Note that once we have done this, we can not restart, since the external validation data will be used as a part of the training set.



Here we have an introduction to the most fundamental problems that ML addresses.

1. **Classification**

We need some examples, that typically are composed of:
- Feature vector
- Category or label

The questions we need to address:
- What is the best label “t” given a new sample?
- What is the probability of “t” given the new feature vector? The answer to this question is a little bit more informative since we are also saying which is the probability of the prediction.

  So, having an output in terms of probability is more informative.

At some point we need to evaluate if the model is working well or not (figures of merit). In the easiest case, which is a binary classification problem, we may define the **accuracy or classification rate** (how many samples have been correctly classified), the **error rate** (complementary of the CR), the **sensitivity** (accuracy in the positive class), the **specificity** (accuracy for the negative class) or the **AUC** (Area Under the Roc Curve).
- We want to maximize the CR
- We want to minimize the ER
- AUC equals 1 if we use a perfect classifier and 0.5 if we use a random classifier.

2. Regression

The difference between classification and regression is that in regression the target is a number. We want to predict a certain number and not just a category.

In this case we also have some data pairs:

- Feature vector (x)
- Scalar label (t).

The idea is that we will have a certain mathematical function f(x) that we do not know, that relates the input data and the output. We also have some noise (biological or instrumental variability).

$$t = f(x) + noise$$

So, the algorithm needs to learn which is the relationship between the input and the target.

Questions:

- What does the function f(x) look like?
- What's the best value of t given x?
- What's the probability of t given x? p(t/x)

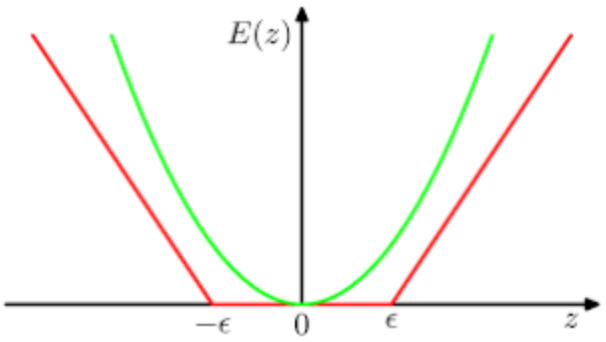
Figures of merit:

- Mean Square Error in Prediction (L2 loss)

$$L2 = 1/N \cdot \text{Sumatory } (Y_{calc} - Y_{pred})^2$$

- Absolute Error (L1 loss)

$$L1 = 1/N \cdot \text{Sumatory } |Y_{calc} - Y_{pred}|$$



Z is the difference between the prediction and the real value
E(z) is L2 loss

If the difference is 0, then we do not have any loss (prediction is perfect). When the differences increase, the loss increases

In this case we want to minimize the differences over all the points in our validation set between the real value and the predicted value. So, it is a figure of merit to evaluate how good the regression model is.

There is a variation of the L1 loss called “epsilon insensitive loss function”, which gives a 0 loss if the error is below a certain threshold “epsilon” (red line).

- There is only a penalty if the error exceeds the threshold

So, as we can see, there are multiple ways to decide how good a regression model is.

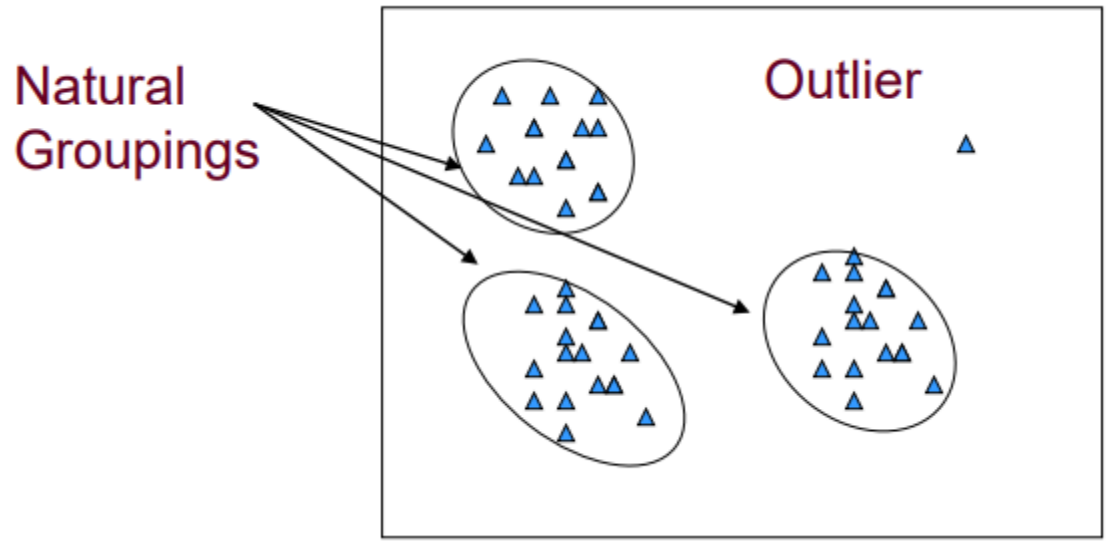
3. Clustering

In clustering we do not have an output, we just have data “x” (input vectors).

We want to learn if the points in the input space appear in groups. But we do not know how many groups we have. So we 2 problems:

- How do we find the groups?
- How many groups are there?

We can also identify outliers and discard them.



In this image we have 3 groups that are “natural”. But, we would need to have a figure of merit that is maximized or minimized when we have the right structure of the clusters.

This is an unsupervised learning problem, since we do not know which is the label (we only have the data).

2. Loss functions and figures of merit

Let's imagine a binary classification problem:

- Class Healthy = +1
- Class Sick = -1

We have our data in dimension “N”, meaning that we have N features. We want to learn a function that maps the input space to another set with 2 numbers (labels 1 and -1).
The data pairs that I have are:

- X → Input
- Y → Output (supervised learning)

We have “l” examples.

$$f : \Re^N \rightarrow \{\pm 1\} \qquad (\mathbf{x}_1, y_1),\ldots,(\mathbf{x}_l, y_l) \in \Re^N \times \{\pm 1\}$$

How do we obtain this function? The best function f is the one that minimizes the loss function or Expected Risk:

$$R[f] = \int l(f(\mathbf{x}), y) dP(\mathbf{x}, y)$$

The expected risk of function “f” is the integral that goes over the loss function but it also takes into account the probabilistic distribution of the data. The loss function is a function of 2 things:

- It depends on the data and the predictions

This loss function is a measure of how well the predicting model is doing the associated task.

The expected risk is approximated by the empirical risk (we can calculate it):

- Mean of the loss function over all the data

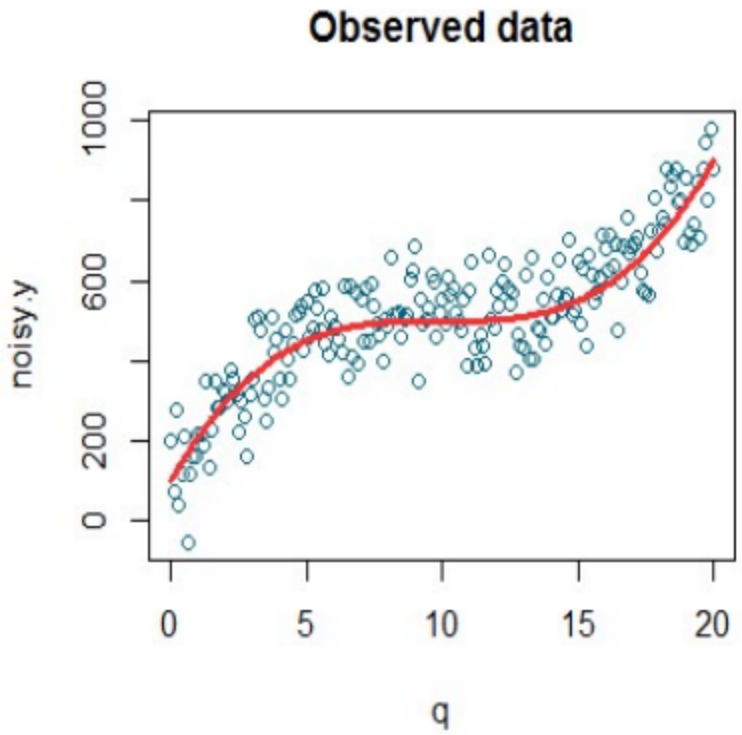
$$R_{emp}[f] = \frac{1}{n} \sum_{i=1}^{n} l(f(\mathbf{x}_i), y_i)$$

Where does this empirical loss take an important role? In 2 aspects of the workflow:

- In the optimization of the algorithm
- In the final assessment of the performance of the algorithm.

Here we have the data and we want to find the function that explains the data. There are infinite functions that we could use.

- We can use the quadratic loss function → l(f(x), y) = (f(x)-y)^2
- This is for a single data or sample.
- Then we calculate the empirical risk with all the data or samples in order to find the best function (we want to minimize).



In most of the cases we use the RMSE instead of the MSE, because it can be interpreted as the same units as the target. Imagine we want to predict the volume in cm^3 of a tumor:

- If we use the MSE, we will have units of (cm^3)^2
- If we use the RMSE, we will have units of cm^3

$$RMSE = \sqrt{MSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y})^2}$$

Then we can say that we are estimating the volume of the tumor with an error of 1 cm^3.

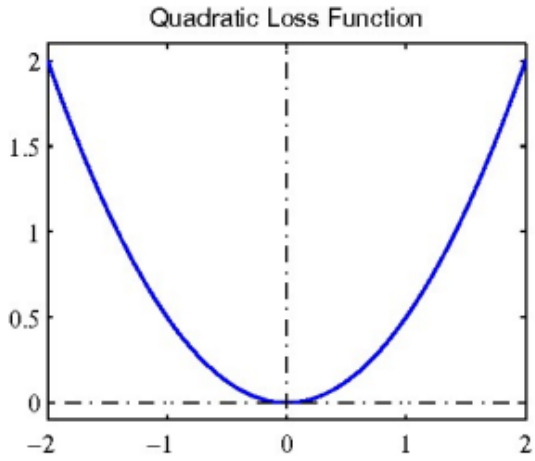
Sometimes we are even more precise and we add more information. We can compute the RMSE in calibration or in training, we can have the RMSE in cross-validation or internal validation or we can have the RMSE in prediction or external validation.

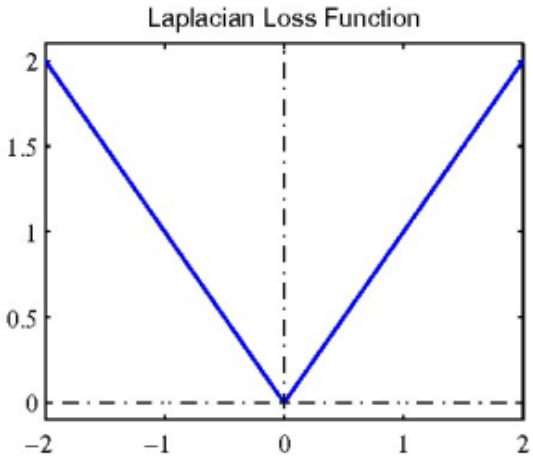
So, we can measure the same figure of merit in different sets of data. The important one is the RMSE of the external validation.

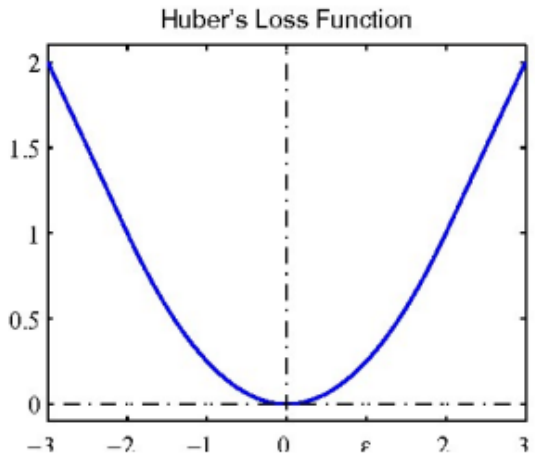
Loss Functions

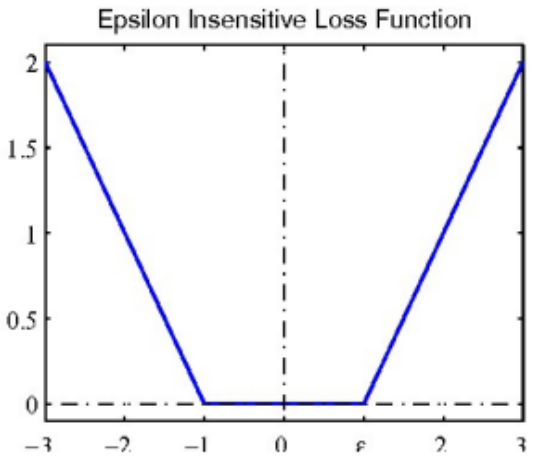
The loss functions can be different:

- Quadratic → L2
- Laplacian → L1 or Absolute error
- Huber's → Combines L1 (for large differences) and L2 (for small differences)
- Epsilon Insensitive → L1 variant









Why don't we just use L2? Because the minimization of the L2 has some problems:

- It is very sensitive to outliers

So, we change the loss functions to make the algorithms more robust against strange data.

Loss Function in Classification

We have been comparing loss functions for regression problems. Let's use loss functions for binary classification problems.
Accuracy or CR is the most common figure of merit for classification problems. But what is the Loss Function associated with accuracy?

The indicator function counts the number of disagreements between the sign of the prediction (f(x)) compared to the target (Y).

$$f : \Re^N \to \{\pm 1\} \qquad (\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_l, y_l) \in \Re^N \times \{\pm 1\}$$

- **Indicator function** $l(f(\mathbf{x}), y) = \Theta(-yf(\mathbf{x}))$ — **Heaviside function**

The **heaviside** function is the "step function" (converts a continuous value into binary).

- Imagine that Y and f(x) have the same sign. In other words, the prediction of the algorithm has the same sign as Y, meaning that the prediction was correct.
- Then the product is negative.
- Then the loss is 0
- If the sign is not the same, meaning that we are not predicting correctly, then the loss is 1.

f(x) will provide an output which is a scalar (a probability). But I want to collapse this output to +1 or -1. Thus, we just add a threshold:

- If f(x) is > 0, then +1
- If f(x) < 0, then -1

Using this threshold, I can convert a numerical output into a label and use the indicator function. If they share the sign, they share the same label.

- We do not care if the probability is very high or low, we just look at the sign.

We can also use the square loss:

- Y and f(x) are equal (1 and 1 or -1 and -1) → then the loss is 0
- But in this case, the square loss is taking into account the "distance" between the prediction and the target. It is not the same to have f(x) = 0.9 and f(x) = 0.3. Both of them can be collapsed to +1, but the first prediction is more accurate.

- **Square loss** $l(f(x), y) = (1 - yf(x))^2$

Binary classifiers are Detectors: Signal detection theory

Statisticians: Hypothesis testing	Engineers: Detection theory
Test statistics (T(x) and v-threshold)	Detector
Null hypothesis	Noise hypothesis
Alternative hypothesis	Signal+noise hypothesis
Type I error (decide H_1 when H_0 true)	False Alarm
Type II error (decide H_0 when H_1 true)	False Negative (or Miss)
Level of Significance or Risk α	Probability of False Alarm
Probability of Type II error β	Probability of Miss
Power of test (1-β)	Probability of Detection

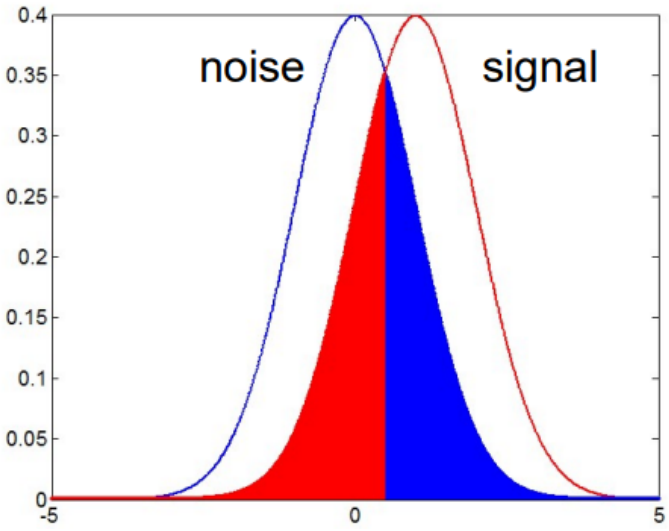
Decision	H_o true	H_o false
Accept H_o	1-α	β (unknown) (errot type II)
Reject H_o	α (error type I)	1-β

Signal = H1
Noise = H0

Imagine that I want to detect if someone is responding to a certain therapy:

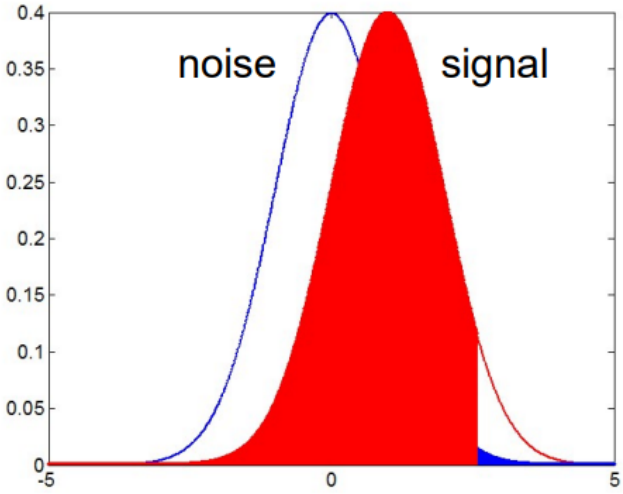
- H0 or noise = It does nothing
- H1 or signal = It responds

The point is that your algorithm provides an output and you need to put a threshold in order to differentiate the 2 classes. But the algorithm will make some errors, as we can see in the following graph.



- Blue on the right are people that do not respond to the treatment but we predict that they do respond (wrong prediction) → False Alarms
- Red on the left are people that do respond to the treatment but we predict that they do not respond (wrong prediction) → Misses

If we change the position of this threshold, the proportion of the 2 errors changes. If we move the threshold to the right, most of the signals are going to be missed.



There is a trade-off between false alarms and false negatives. The proportion depends on the position of the threshold.

Remember that this threshold is something that we put at the output of the classifier in order to condense a numerical output to a label.

At any position of the threshold I can calculate all the figures of merit.

		Real: Normal	Real: Alarm
Decision	Normal	True-negatives (TN)	False negative (FN)
	Alarm	False positive (FP)	True-positives (TP)

Accuracy (Classification Rate)= (TP+TN)/(TP+TN+FP+FN)

Sensitivity (Recall)=TP/(TP+FN) – Probability to correctly classify an Alarm

Specificity=TN/(TN+FP) – Probability to correctly classify a Normal state

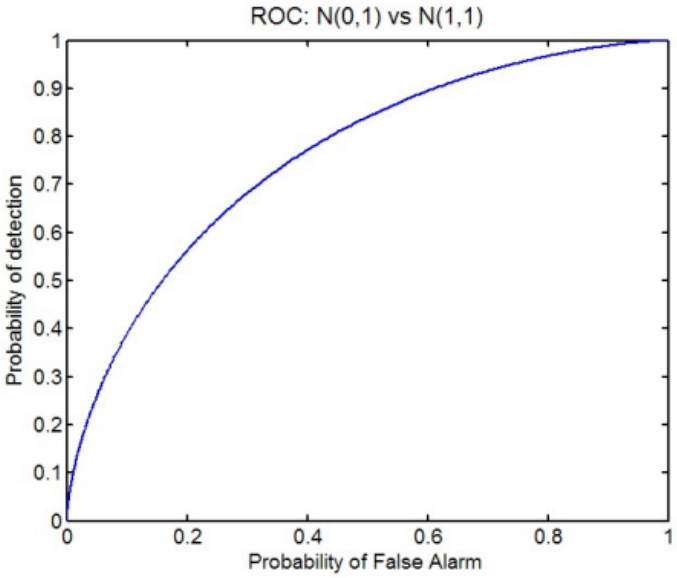
Precision (Positive Predictive Power)=TP/(TP+FP) – Reliability of Alarm

Negative Predictive Power= TN/(TN+FN) – Reliability of no-alarm

Positive predictive power → Once I have an alarm, can I believe it?

We can also calculate the ROC curve, which is used to evaluate the performance of a binary classification model:

- X axis: 1-specificity
- Y axis: Sensitivity



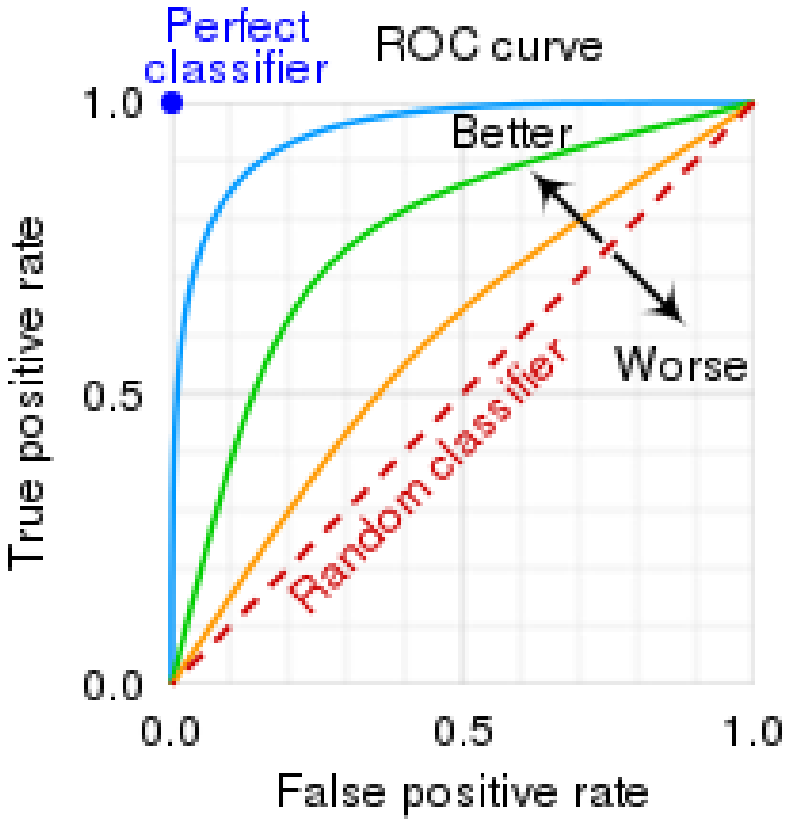
Every point of the curve is a position of the threshold. So, the ROC curve explores the trade-off between a model's sensitivity and specificity for all possible values of the threshold.

- At first you do not detect anything but you do not have any false alarm.
- At the end you detect everything but there are plenty of false alarms.
- A diagonal line represents the performance of a random classifier with no discriminatory power

The Area Under the Curve (AUC) is a commonly used figure of merit to evaluate classifiers with an analog output.

Why are we interested in these ROC curves as a figure of merit? Because otherwise you are never sure.

- Your algorithm goes better than mine. But what happens if we change the threshold?
- With the ROC curve of both algorithms, we can see which algorithm is better independently of the position of the threshold.
    - Plot both curves in the same plot and decide.



Permutation Test

Imagine I have designed an algorithm and I have computed its ROC curve. Then I see that the AUC is 0.75. But I want to know if this could be obtained by chance or not.

- To know this, I apply the permutation test

In a permutation test, we are testing the H0. We destroy the relationship between X and Y, by randomly permuting Y (then the targets are random).

By doing this 1000 times, you have a distribution of AUC values you would expect under H0. Then we calculate the probability that by chance the AUC is bigger than 0.75. If this probability is smaller than 0.05, then we can say that my AUC is statistically significant under a permutation test (reject H0).

So, we are calculating many times the AUC that you could obtain by chance.

The more data you have, the narrower the interval will be. So, typically, this type of problems appear because you have insufficient data (in that case, you must increase your data).

- My AUC is not statistically significant, so lets increase the data and maybe I will realize that it is statistically significant.

The AUC is an estimator, and estimators have uncertainty. The important thing is to realize that you need to compute the uncertainty of your estimator.

Imagine that someone says “algorithm 1 has a CR of 0.891 and algorithm 2 has a CR of 0.892, so algorithm 2 is better”

- We could ask for what threshold…
- Are we sure that CR 2 is bigger than CR1? No, we need to know what are the 95% confidence intervals associated with measure 1 and 2. Without this information, we can not say which is better.
    - Maybe we realize that the uncertainty classification rate is 0.1 and therefore both algorithms are the same.

Confusion Matrix

Many of the presented figures of merit are for binary classifications. But we can have classification problems with multi-classes.

- Note that you can always revert a multiclass problem to a binary problem by doing a 1 vs all. If we have 4 classes:
    - Class 1 against the other 3
    - Class 2 against the other 3
    - Class 3 against the other 3
    - Class 4 against the other 3

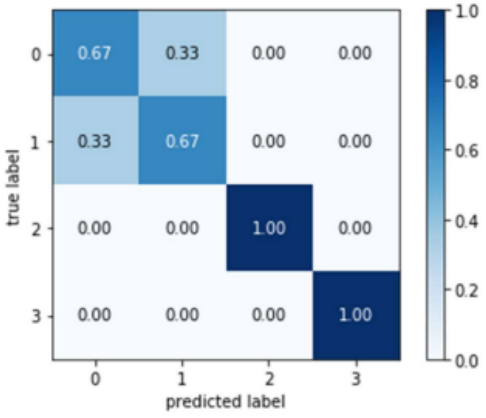
    So, we are transforming a multiclass problem into multiple binary classification problems.

If we do not want to use this strategy, we can always use the confusion matrix → Array of the true and predicted label.

If the classifier is perfect, we expect a diagonal full of 1, meaning that it has predicted the label correctly.

Otherwise, the classifier is making some errors.

With this information we can also compute the Classification Rate.



Introduction to classifiers

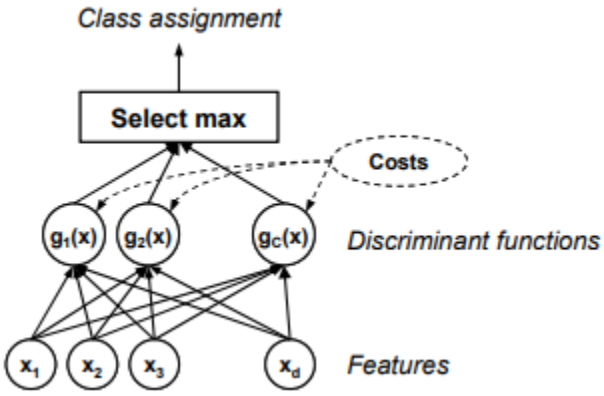
Discriminant functions

Most of the classifiers can be casted into discriminant functions. The idea is that I have the feature vector and as many discriminant functions as classes.

- If I have 2 classes I will have 2 discriminant functions

These discriminant functions have to be built in such a way that they provide a large intensity or output when the feature belongs to a particular class.
So, if the feature vector belongs to class 1, I expect that the discriminant factor G1 is bigger than the others.

At the end, I will compare the output of all the discriminant functions and I will select the one that has the highest value → Meaning that the feature corresponds to that class. So, we assign the feature vector to the class that has the biggest discriminant function.



The different algorithms have different proposals of building these discriminant functions. For that reason, they are different and give different results.

Nearest Centroid Classifier

Let's propose as a discriminant function the "minus euclidian distance between the vector and the center of the class".

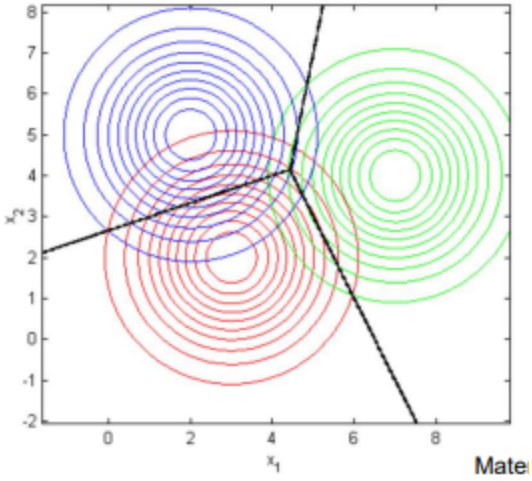
$$g_i(x) = -(x-\mu_i)^T(x-\mu_i)$$

We have a "minus" because we want the discriminant function to be large when the probability is high and here small distances are good. Thus, we need to add the "minus" sign so that small values are transformed into large values.

So, we want the discriminant function to have a large value when the probability of being in the class is high. But the problem is that the value obtained is going to be small when we are close to the center and therefore we have to add a “minus” to revert that.

- The discriminant function will give -0.1 if the distance is very small (for example)
- The discriminant function will give -6 if the distance is very large (for example)

If we are closer, the value is less negative.



The first thing I need to do is calculate the centroid (arithmetic mean of the data) of all classes. Then, for each feature vector, I compute the distance to all centroids. The feature will correspond to the class that has the nearest centroid.

Note that the decision boundaries are straight lines. So, this is the easiest linear classifier that we can find.

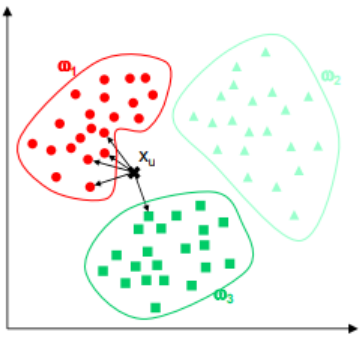
- If the classes are well separated, it is a good option.

If the point is exactly in the line (will never happen), we just select one class at random.

K Nearest Neighbor Classifier

A given unlabeled example is classified based on their similarity with the training data. So, we just find the k “closest” labeled examples in the training dataset and assign the samples to the class that appears most frequently within the k-subset. It requires:

- An integer K
- A set of labeled examples
- A metric to measure “closeness”



You are what you are close to. If K = 5, you need to compare the 5 nearest neighbors:

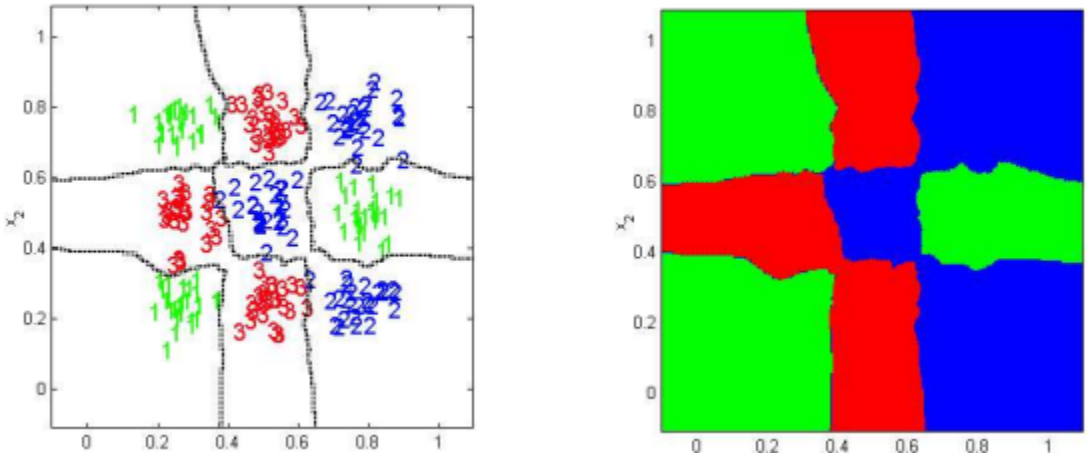
- If 4 are red and 1 is green, the assigned class will be the red one.

The problem here is that we need to compute a lot of distances (euclidean or whatever distance we choose). In fact, for each point we need to compute all the distances with all the points in the dataset (to know which are the closest ones) and then sort them and select the k neighbors that are closest.

The drawback is that it's very computationally expensive and you need to store all the data (the nearest centroid does not have this problem).

- So, it is useful if the dataset is NOT huge.
- It is called a lazy model because we do not build any model.
- The only parameter to optimize is K in the internal validation.

We generate data for a 2-dimensional 3- class problem, where the class-conditional densities are multi-modal, and non-linearly separable. This will not work with the nearest centroid classifier.



We used kNN with:

- k = five
- Metric = Euclidean distance

Partitions

3 way split (simplest data partition)→ Randomize the dataset and divide the data in:

- Training dataset
- Internal validation
- External validation

If I want to optimize K (hyperparameter of the K-NN), then I will repeat the analysis with different K and compare the results.

- Then I say, I will use K = 3 because I obtain the highest CR in the internal validation.

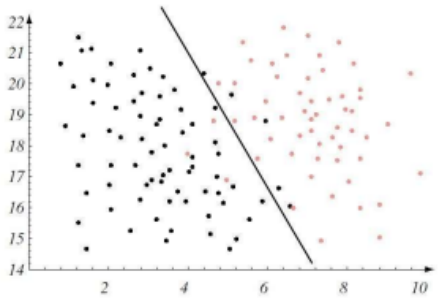
Once I have selected the optimal parameter, the internal validation and training data is going to be fused together and I have a larger training dataset. So, we are retraining the algorithm with the sum of both datasets (so, we use more data!).

Complexity Control

Using algorithms that are too complex for the data is risky. Thus, we use simpler algorithms.

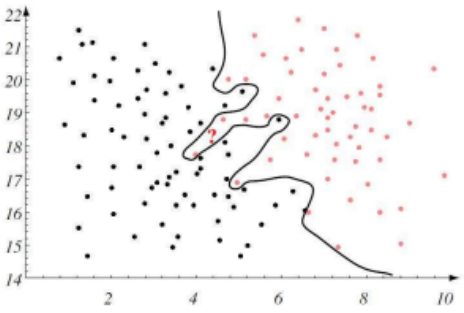
For example, in this case, one person uses a linear model and another one a more complex model.

- Too simple model:
- Low complexity



- Large training errors
- Large test errors

- Too complex model:
- High complexity



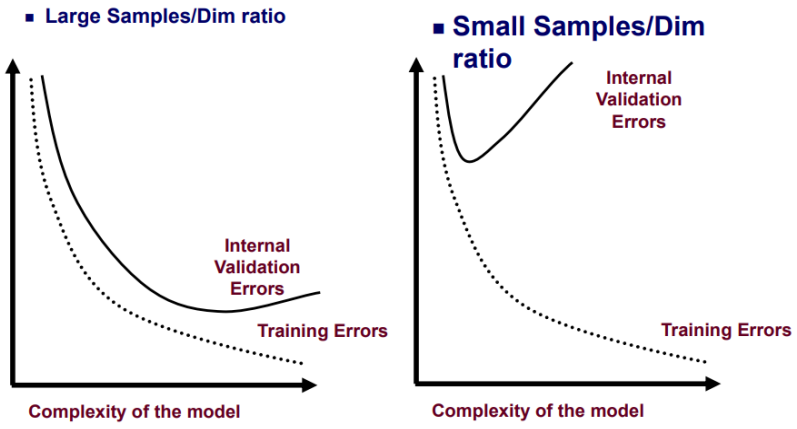
- Zero training errors
- Large test errors
- Poor generalization

The complex model may not be better than the model on the left. The more complex model will have overfitting, meaning that the model fits very specific data but it may not be as good if we use other data.

The correct classification of the dataset in the plots is not our ultimate objective. The new data is our objective.

The complexity of the model has to be controlled for good performance.

- If we have small data for training, the algorithm must not be very complex. For example, if you have 2 data points, you can not fit a 10 order polynomial.
- If we have a huge data, then we can use a complex algorithm



In neural networks, if we keep increasing the number of neurons, we will eventually have a training error equal to 0. But we need to keep an eye on the errors in the internal validation.

In the internal validation, I will see if using a more complex algorithm (varying the order of the polynomial, for example) is better or not. Otherwise, we do not know which is better.

- Evaluate the prediction errors in the internal validation of different algorithms to see which fits the data better.

This is the problem of overfitting → Focussing too much in the training data and having very bad predictions with other data.

Dimensionality Reduction

There are 2 methods to decrease the complexity of a model:

- Have the model running in a smaller input space. It is easier to fit the model in 3 dimensions than in 1000 dimensions.
- Regularization: Method to fit models that are apparently too complex even if you have few data. We do this by adding penalties.

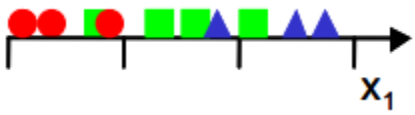
It is not optimal to have a very large number of variables and only a few samples. The possibilities of running into overfitting are very big.

- Everyone wants to work with a lot of samples and a small number of variables or features.

Curse of dimensionality: The performance of learning algorithms is clearly suboptimal when there is a small number of examples / dimensionality ratio. For a given sample size, there is a maximum number of features above which the performance of our classifier will degrade rather than improve

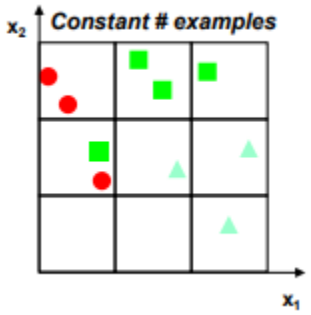
Consider the following example: Imagine that I want to classify 3 classes using a single dimension.
My classifier works the following way:

- Divide the input in bins or regions
- Compute the ratio of examples for each class at each bin
- For a new example, find its bin and choose the predominant class in that bin.
- In this case, the first bin will be red, the second will be green and the third will be blue.



Notice that there exists a lot of overlap between classes to improve discrimination, so we decide to incorporate a second feature.

Moving to 2 dimensions increases the number of bins from 3 to 3^2. As we can see, it is becoming more empty. Meaning that there are some bins that do not have any example, others have a single example…



In order to have the same number of samples per bin, I would need to increase the data. Imagine if we use 3D or even 1000D. The space will be too big or the data.

If we have few examples in a space that is huge, the space will be empty and the algorithm will not be capable of learning anything.

The point is that we can use the dimensionality as one of my complexity parameters and I will need to see at which particular dimension the performance of my algorithm is maximum.

- So, we will need to optimize the dimensionality the same way we are optimizing any other parameter.

We said that there are 2 ways to reduce the dimensionality:

- Feature selection
- Feature extraction

Feature selection refers to selecting some of the dimensions as the important ones (we remove all the others). Imagine we are trying to find if someone has a risk of a heart attack. We would remove features such as the color of his hair, for example, since it is not informative.

So, we will use a feature selection algorithm to identify which are the features that are informative.

- The idea is to propose a set of features, train the classifier and see if the predictions are good in the internal validation.
- Then I propose another set of features and compare.

So, the important thing is to decide which subsets of features we are going to evaluate. One option is to randomly generate subsets of features if we have a big computer (brute force).

- Maybe we will find a combination of features that is better than using all of them

Feature extraction refers to creating new sets of features by combining existing ones. We are not picking but building new features.

- We are building new features as a mathematical function of the original ones.

The easiest thing is to use linear methods. I just multiply my original input vector by a rectangular matrix and then I go to a space of smaller dimensions.

In either case, the goal is to find a low-dimensional representation of the data that preserves (most of) the information or structure in the data.

- Note that feature selection is probably a better option than feature extraction, if they both give the same results, because of the interpretability of the result. If we pick some original features, those original features retain the original interpretation.

  If I use feature extraction, my new features will be: $0.4 \cdot F1 + 1.3 \cdot F2 + 8.1 \cdot F3 \ldots$

Here we have 2 ideas to do dimensionality reduction with feature extraction:

1. Feature extraction can be supervised → LDA (Linear Discriminant Analysis)
2. Feature extraction can be unsupervised → PCA does not use the labels, it only tries to catch the distribution of the data. It tries to find a subspace that explains most of the variance.
   a. First find the centroid
   b. Then look for the first direction that explains most of the variance
   c. Then look for a second dimension (orthogonal or uncorrelated to the first one) that explains most of the variance

   PCA is also used to detect outliers. If I have a sample that is very far away from my plane or within the plane but very far away, it is an outlier. So, there are 2 statistics to find outliers:
   - Distance to the plane → Residuals.
   - T^2 statistic tells us how far from the centroid the sample is.

   If the values of any of those statistics is very big, then we have an outlier.

Supervised methods are not always better than unsupervised methods, because LDA overfits a lot (it is the most overfitting algorithm ever).

- If we see a plot of LDA, we must know if it is training or validation data. If it is training data, throw the paper away.

How do I control that I am not overfitting? I have to check what happens in internal and eventually in external validation.

Regularization

Method to fit models that are apparently too complex even if you have few data. We do this by adding penalties.

Here I want to fit Y as a function of a vector of “p” dimensions. The regression coefficients are “beta”.

$$y_k = f(X_k) + \varepsilon_k = \beta_0 + \sum_{j=1}^{p} x_{k,j}\beta_j + \varepsilon_k$$

The default way to find the coefficients is to minimize the sum of squared errors. But if we have few data, even this model will overfit.

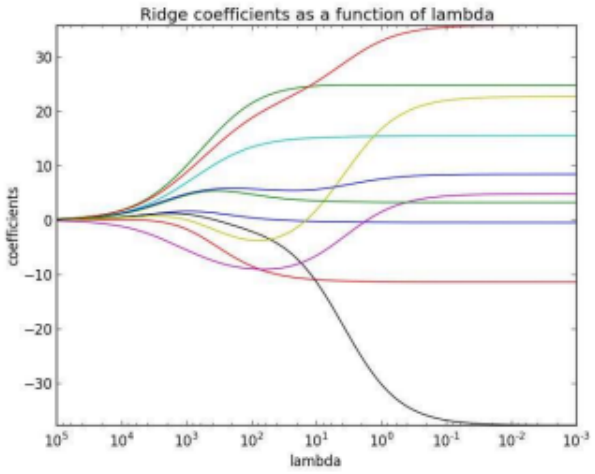
$$\hat{\boldsymbol{\beta}} = \boldsymbol{arg}\min_{\boldsymbol{\beta}} \left\{ \sum_{i=1}^{N} \left(\boldsymbol{y_i} - \boldsymbol{\beta_0} - \sum_{j=1}^{p} \boldsymbol{x_{ij}\beta_j} \right) \right\}$$

Thus, we must use another method called “Ridge Regression”. We use the same figure of merit (Least Mean Squared Error) but I add a penalty which is the sum of the squares of the coefficients.

- Lambda is the regularization parameter that controls the strength of the penalty. The bigger lambda, the smaller the coefficients will be. So, by increasing lambda we are constraining the model, since the coefficients can not grow (otherwise there will be a penalty on the thing we want to minimize).

$$\hat{\beta}^{ridge} = \arg\min_{\beta} \left\{ \sum_{i=1}^{N} \left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j \right)^2 + \lambda \sum_{j=1}^{p} \beta_j^2 \right\}$$

If lambda increases (to the left), all the coefficients decrease up to a point that they are equal to 0. Then you check for different values of lambda the results in internal validation and find the optimum.



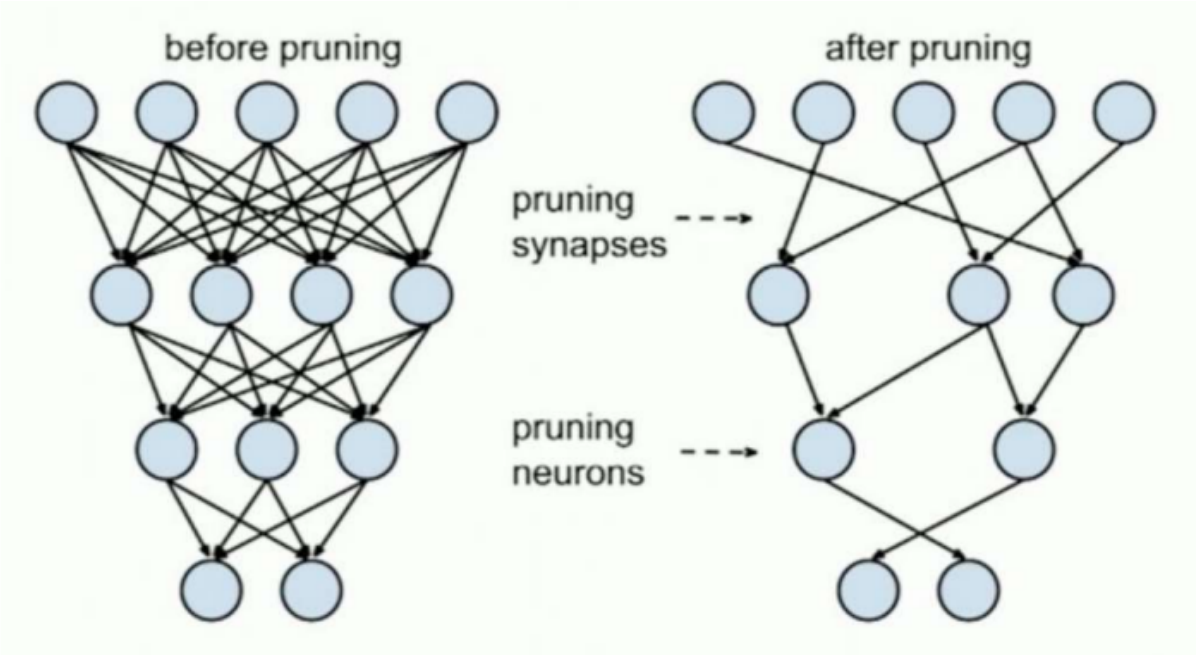
So, the solution of Least Squares is not the best to predict new data, but one with smaller coefficients predicts better than the original ones → This is the idea of regularization.

Another example with neural networks:

Imagine that you have a neural network and you train it. You see that it overfits and it is a disaster. A solution could be regularization:

- Maybe there are some weights that have a value close to 0. Thus, I can remove them and the network will not be fully connected.
- Maybe there are some neurons that never get activated. Thus, I can remove them.

Thus, we are going from a very large NN to a less complex one.



So, we start with a complex model and we simplify it by reducing its dimensionality.

3. Data preprocessing

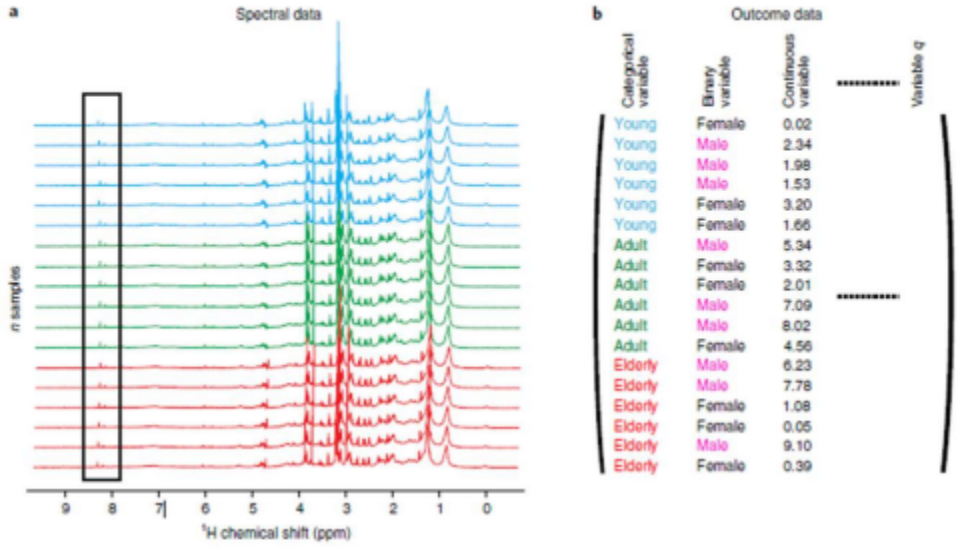
Data preprocessing refers to everything you must do before ML, which is application dependent. Because depending on the type of data you are working with, there are different procedures to do the data preprocessing.

Here we have some examples of NMR metabolomics. The idea is that you have some raw data, in this case NMR spectra:

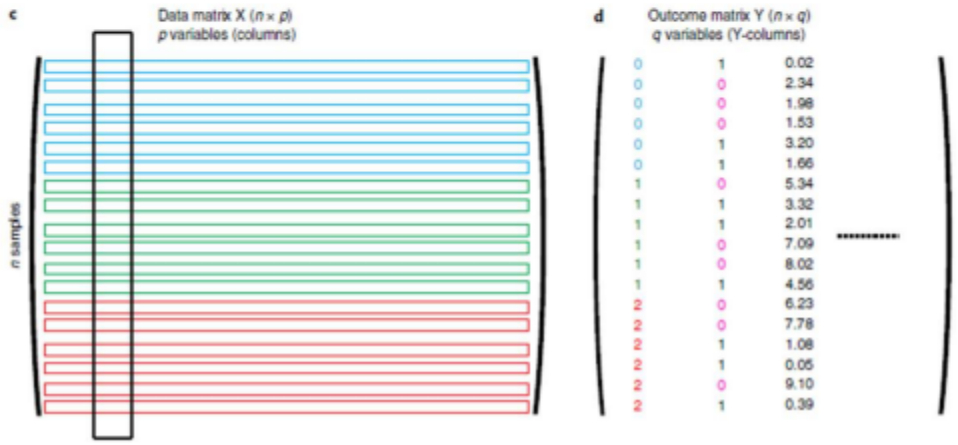
- We have a number of samples and for each one we have an spectra.

We may also have some additional information:

- Categorical variables such as the age (young, adult, elderly)
- Binary variables such as the sex (male, female)
- Continuous variables…



We have to convert this into our final X block. Then we will have a matrix with n samples with certain values. Also the categories can be transformed into numbers.



One-hot-encoding approach:

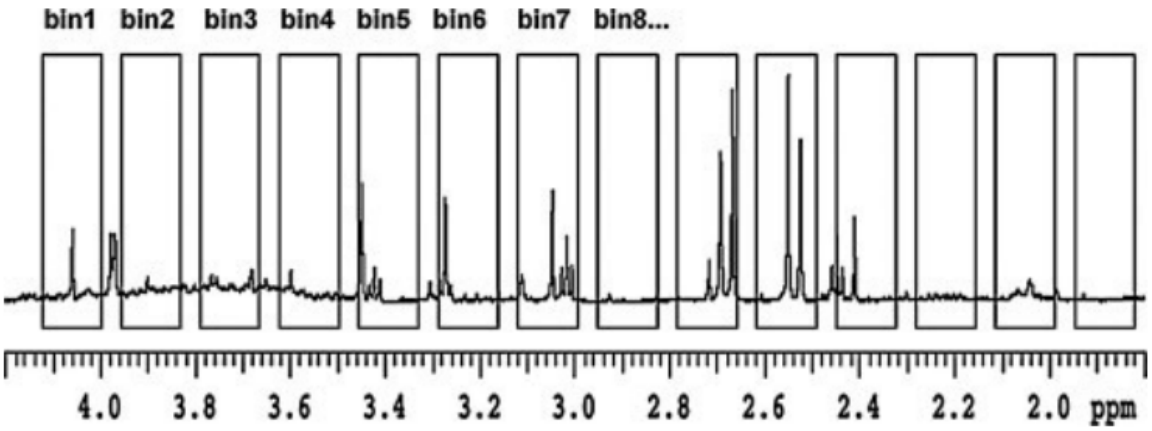
- Convert a category (age) into multiple columns, for example:
- 1 0 0
- 1 0 0
- 1 0 0
- 0 1 0
- 0 1 0
- 0 1 0
- 0 0 1
- 0 0 1
- 0 0 1

How can we convert the spectra into something that is smaller? The idea is to use binning

Binning (Reduces Dimensionality)

Divide the spectra in segments and integrate the segment (calculate the area under the curve)

- We must know how wide the bin must be
    - If you are using bins that are too large, maybe you are merging peaks that contain different information.
    - If the bin is too small, then the data reduction that you are doing is limited.



Using this, we are going from a spectra to an array.

Alternative:

- Detect the peaks (which contain the information) and compute the area under the peaks

Quality control samples (used in biofluids analysis)

It refers to the repeated analysis of samples of constant composition to compensate for the instrumental errors. The objective is to control the stability of the measurements.

It allows us to determine the instrumental error associated with the measurement of each variable/feature. This instrumental error is typically expressed in terms of Relative Standard Deviation:

RSD = variance of every feature / mean

If the instrument is perfect, the result of the measurement of the different metabolites would be always the same.

Then, we can measure the variability/error/noise associated with every feature. Imagine we want to calculate the area under the curve of bin number 5, which corresponds to feature number 5. We calculate this feature in the control samples all along the study and the result should be the same, because the quality control sample is always the same. But this will not happen and we will be able to compute the Relative Standard Deviation, which is a measure of the error that we have in every feature.

Note that if the RSD > 0.5, it would be good to remove that feature because the measurement of that feature is not reliable.

Unsupervised feature filtering

Another way to remove features from the analysis is to use an unsupervised feature filtering. Features that show no significant variability in the full dataset can be excluded.

- Some features vary among samples and others don't. The features that does not change across the dataset probably has nothing of interest.

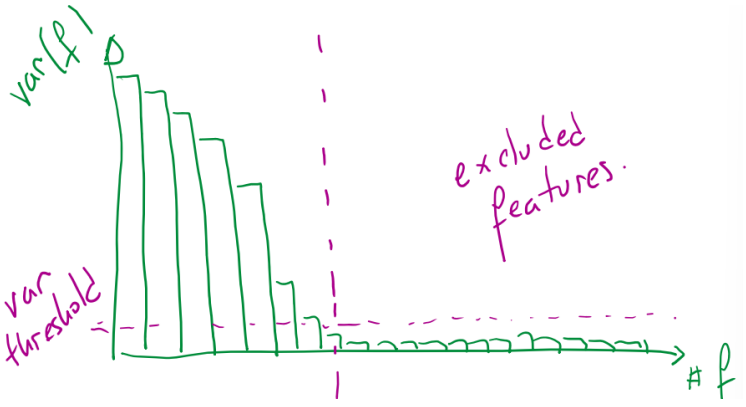
Compute the variance of every feature, we sort them and then we define a variance threshold. If the variance is too small, we can remove those features.

The important thing is that we can calculate this variance all across the dataset without any use of the labels (unsupervised).

The chosen threshold is up to us. We can be more or less stringent.

- We can choose it by eye or we can optimize it in the internal validation

Another more extreme example would be to have a dataset with only males. We should remove this feature because it is not giving any additional information.



Non-linear transformations

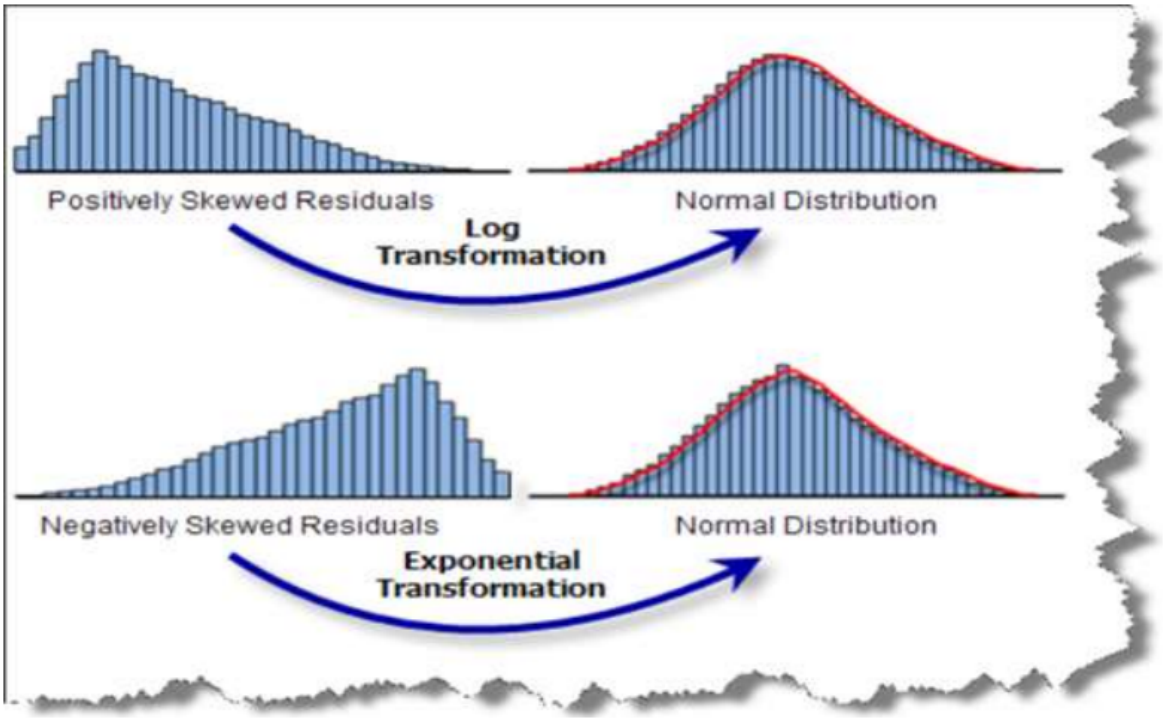
There are plenty of methods or algorithms that are dependent on the distribution of the features and they work better when the features are normally distributed.

- When the distribution has very long tails, these algorithms suffer.
- Particularly methods based on Least Squares, variance, PCA (distance)…

If we use these kind of methods, it is nice to make a histogram of the distribution of the features and check if the distribution is normal or not.

Non-linear transformations aim to improve the normality of the distribution of features across samples.

- Log transformation for positively skewed residuals. This is the case of metabolomics, because the data is always positive (because we are working with concentrations of compounds).
- Exponential transformation for negatively skewed residuals



Normalization or scaling?
Both terms are similar but they are not the same.

Scaling refers to equalizing the importance of features. It is something that you calculate across samples to modify a feature. Autoscaling is the typical transformation:

fs = (f-mean)/sd → per feature

mean and sigma are calculated across samples

The features become dimensionless, because they do not have units. Thus, when we are doing an analysis where a column is in Kg and other columns that are in cm, maybe they take very different values and once we do the scaling, the units no longer matter.

Imagine that for feature 1, the values are very high and for feature 2 the values are very small. Most algorithms will automatically think that feature 1 is more important than feature 2. So, in order to compensate for this or to make the algorithms blind to this fact, we use autoscaling.

After doing the autoscaling, the variance of all features will be equal to 1 and the mean will be equal to 0. So, all features vary the same.

Is this always good? It depends. Because in this scenario all features are equally important. But this may not be true. So, we need to check if the autoscaling improves or not the results.

Normalization equalizes the importance of samples. It is something that you calculate across features to normalize a sample.

$$\tilde{x}_i = \frac{x_i}{\sqrt{\sum_i x_i^2}}$$

“i” runs across features. So, we are doing transformations for a single sample across features.

Introduction to Validation Techniques

This is the workflow that we should follow in all analyses:
Dataset → Data partition (separate your data for external validation, otherwise the model is trained with those examples) → Feature selection → ML

So, at the beginning of the workflow, we must do the data partition.

The more data, the better:
- Better models
- Better validation

But data collection is expensive. Bioinformatics data is characterized by:
- Many features
- Few samples (individuals)

Validation techniques are motivated by two fundamental problems in data analysis: model selection and performance estimation
- Model selection or complexity control consists in finding the correct complexity of the algorithm.
    - Almost invariably, all data processing techniques have one or more free parameters
    - How do we select the “optimal” parameter(s) or model for a given classification problem?
- Performance estimation
    - Once we have chosen a model, how do we estimate its performance?
        - Performance is typically measured by the TRUE ERROR RATE, the classifier’s error rate on the ENTIRE POPULATION
        - Beyond accuracy other figures of merit as:
            - Sensitivity
            - Specificity
            - Area Under the Curve
        - Should be evaluated on data not used to build the model.

For both Model selection and performance estimation we will need to find a figure of merit. A figure of merit for regression would be:
- Mean Absolute Error
- Mean Squared Error

A figure of merit for classification would be:
- Accuracy (% of samples that are classified correctly)
- AUC

We need to decide a figure of merit to optimize the model and finally compute the final performance of the model.

If we had access to an unlimited number of examples these questions have a straightforward answer (all the things that we are going to see now do not matter):

- We just choose the model that provides the lowest error rate on the entire population and, of course, that error rate is the true error rate

In real applications we only have access to a finite set of examples, usually smaller than we wanted:

- One approach is to use the entire training data to select our classifier and estimate the error rate
    - This naïve approach has two fundamental problems
    - The final model will normally overfit the training data
        - This problem is more pronounced with models that have a large number of parameters
    - The error rate estimate will be overly optimistic (lower than the true error rate)
        - In fact, it is not uncommon to have 100% correct classification on training data
- A much better approach is to split the training data into disjoint subsets: the holdout method

Validation levels

Internal validation (part of the model development phase) and is also referred as cross-validation

- Used to optimize the model and biomarker discovery. So, it is used in model selection.

External validation is used for performance assessment. There are different ways to do it:

- **Temporal validation:** Developed model is applied to data gathered later on using the same nominal conditions. Example: You collect some data in spring, you get some results, wait 6 months and check if the conclusions are still valid.

- **Geographical validation:** Developed model is applied in other centers by other investigators, maybe different devices (of the same kind). Example: You do an study in a particular hospital and then you see if the results can be transferred to another hospital.

- **Different setting/domain:** Developed models are applied to other instruments (same technology different vendor), applied in diverse conditions (primary care vs secondary care), maybe different ethnic groups, diverse subject baseline conditions……

All these types of external validations provide more evidence that support that the result of your model is consistent.

- My model works irrespective of the instrument used…

So, the important thing of the external validation is to check the generalization power of your model, the reliability of the model… If the model is very sensitive to the particular conditions of the analysis, then the model is no longer useful.

- If the model only works in one hospital, then it is not useful.

Very important:

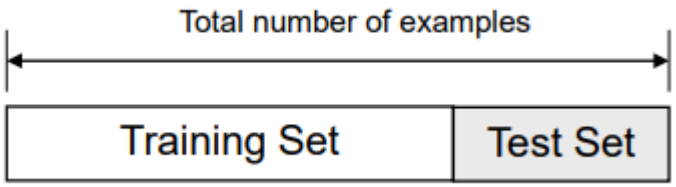
- The validation level controls the domain of validity of the model. Extrapolating the performance of the model to other conditions is not guaranteed.
    - You have to define your population (range of applicability of the model). Your model is going to predict something in a population. If the population is the patients in BCN, then you do not run the model with patients from madrid. So, we have to check if with an independent cohort within the population, the model is still valid.
- Model validation is not to repeat the statistical analysis with maybe other techniques, it is to apply the model to new data.

Data partition inside the internal validation

The holdout method

Split the dataset used to build the model into two groups

- Training set: used to train the classifier
- Test set: used to estimate the error rate of the trained classifier



Fundamental trade-off

- The larger the training set, the more accurate the classifier will be. But then the test-set will be very small and it will not be reliable to estimate the error. Imagine that we only have 1 sample, the estimation of the figure of merit will not be reliable.
    - If the test-set is small, the variance of the estimator for the figure of merit will be huge.
- The larger the test-set, the more accurate the error estimation. But the training set will be very small.

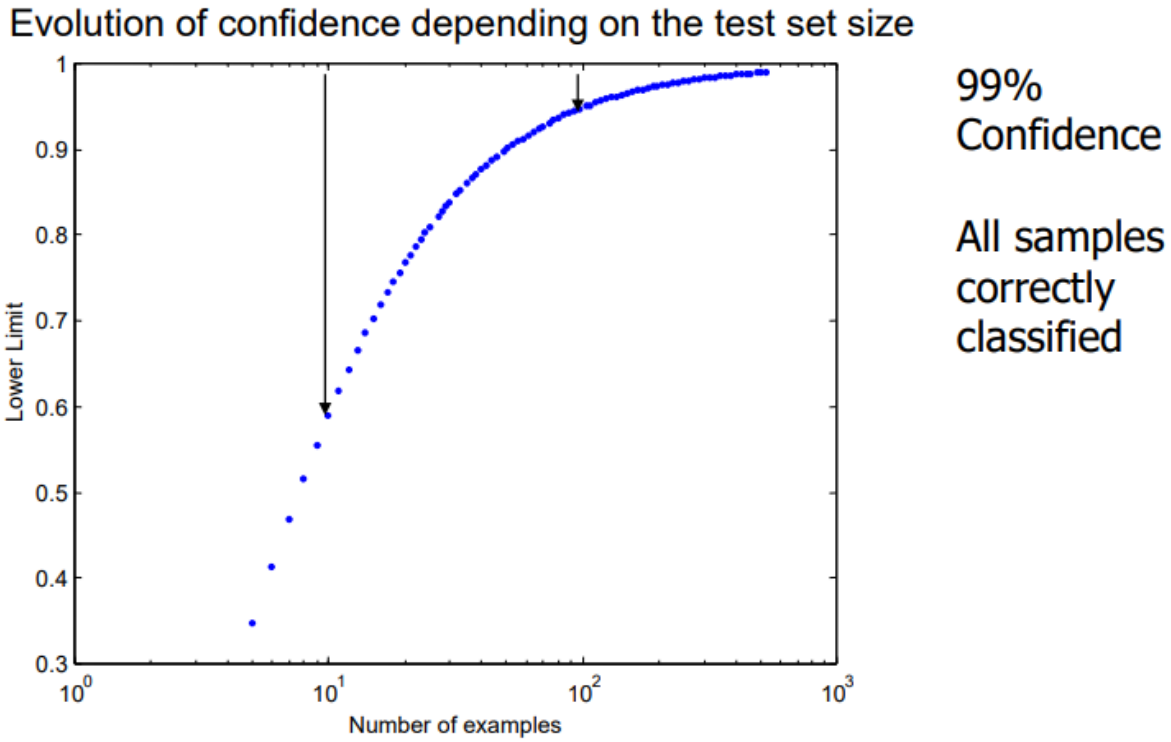
Can I use all the samples for training and then all the samples for validation? Yes, we will see how.

Distribution of proportions (confidence limits)

When we estimate something such as the CR (accuracy), there is some associated uncertainty that comes from the fact that we are using a limited number of samples.

So, the thing that we have calculated here is with a 99% confidence interval. If all the samples are correctly classified, what is the confidence interval of my CR.

- I have 10 samples and the 10 samples are classified correctly. How sure am I that the CR is equal to 100%? If we go to the plot, we can see that at the 99% confidence, the CR could be between 60% and 100% despite that all samples have been classified correctly.
- If we go to 100 samples, if all of them have been classified correctly, then the uncertainty will be smaller. The CR will be between 95% and 100%.
- So, the more samples we use, the lower the uncertainty.



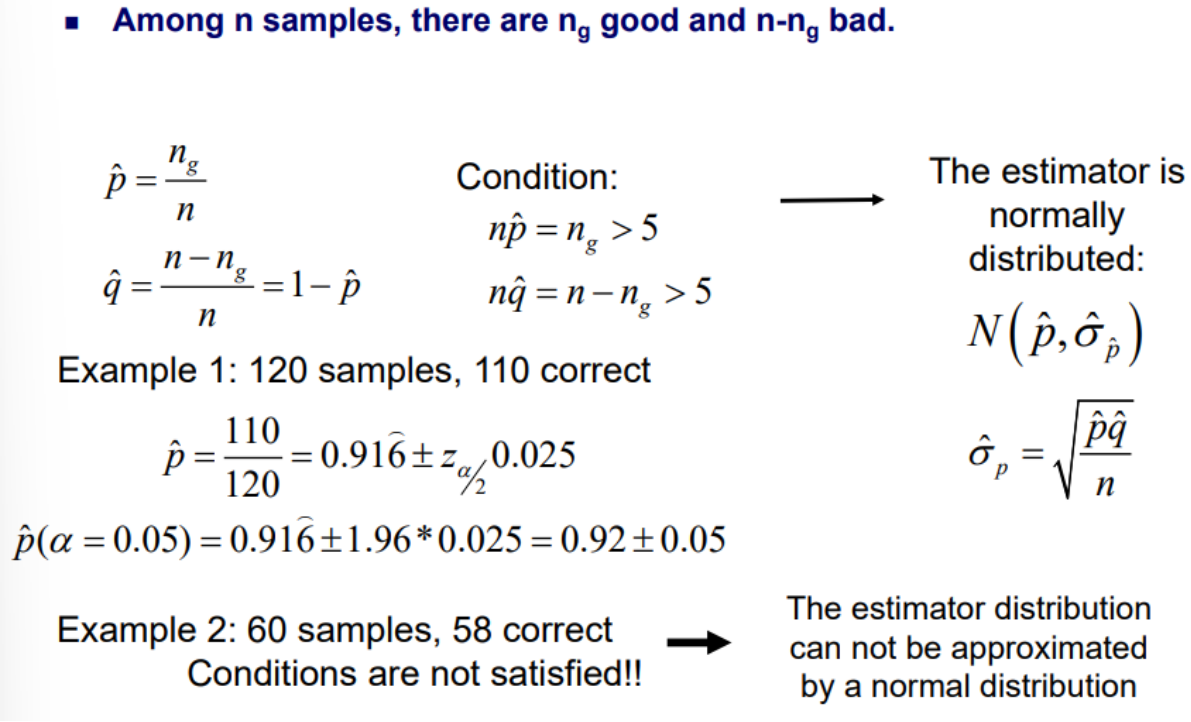
So, for whatever samples we have, we need to calculate the associated uncertainty to the CR.

Here we have an example:

Let's imagine a case where we have 120 samples and 110 are correct. Then I can calculate the CR, which is around 92%.
Since I only have 120 samples, I will have some uncertainty in regard to this classification.

In some conditions, if the number of misclassified samples is bigger than 5, then this ratio can be assumed to be distributed normally.

If I calculate the variance, then I multiply it by a factor to have a risk of 5% and then I can claim that my CR is equal to 0.92+-0.05.

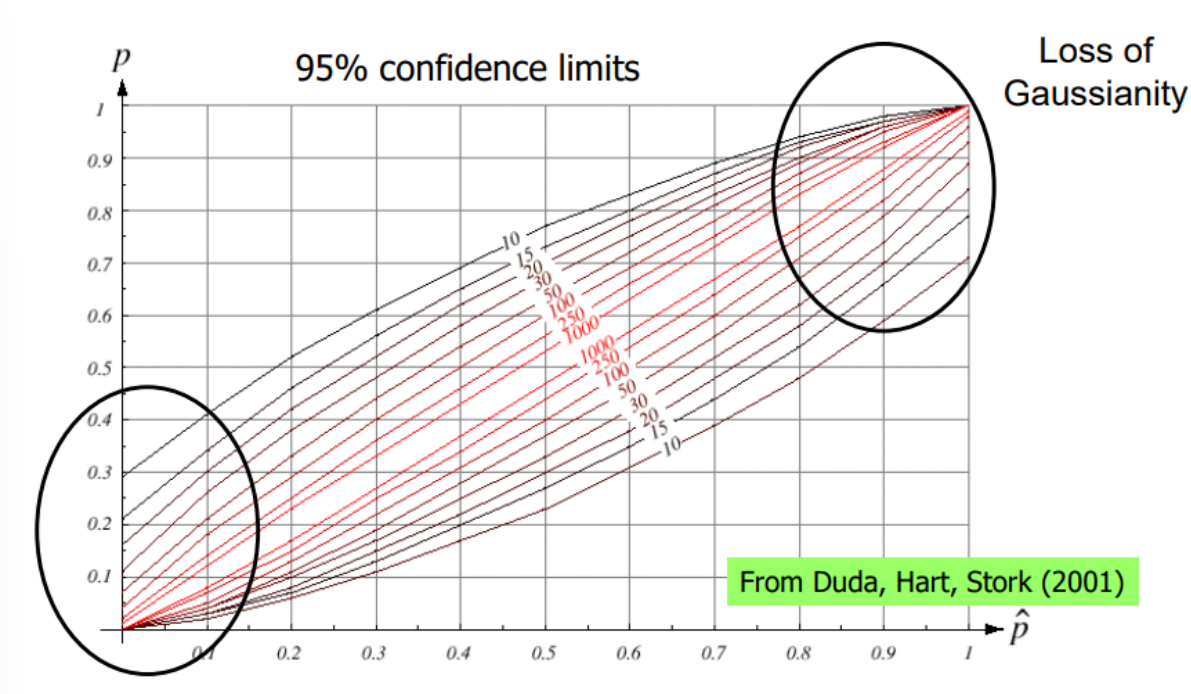


Here we have another way to see the same.

At the 95% confidence interval (5% risk), once you have a certain value for the CR, depending on the number of samples used, then you will have more or less limits of confidence for your estimator.

We have the estimated proportion, the number of samples and the real data.

- Proportion X axis (accuracy)
- middle is the number of samples
- Real value Y axis



Even for a large number of samples we will have some uncertainty (look distance between lines).

The holdout method

The holdout method has two basic drawbacks

- In problems where we have a sparse dataset we may not be able to afford the "luxury" of setting aside a portion of the dataset for testing.
- Since it is a single train-and-test experiment, the holdout estimate of error rate will be misleading if we happen to get an "unfortunate" split.

Is there a better way to use the samples that we have?

The limitations of the holdout can be overcome with a family of resampling methods at the expense of more computations

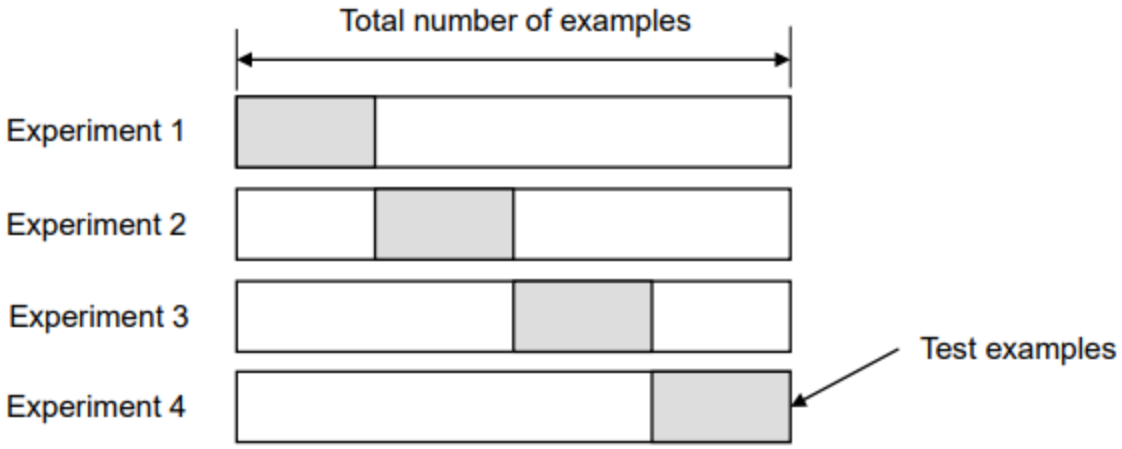
- Cross Validation
  - Random Subsampling
  - K-Fold Cross-Validation
  - Leave-one-out Cross-Validation
- Bootstrap

K-fold cross-validation

The idea is that we are going to use all the samples for training and all the samples for validation. But the model will be different each time.

Create a K-fold partition of the the dataset

- For each of K experiments, use K-1 folds for training and the remaining one for testing.



Here K = 4 → We are using 4 different algorithms. Then we join all results and compute a single estimator of the classification rate. So, we calculate an overall CR despite the fact that you have been using 4 different algorithms.

This is just to decide the optimal hyperparameters, which are the ones that control the complexity of the model.

Once we have the hyperparameters that better fit the model (it provides the best CR under 4-fold CV) then we do the training with the correct parameters with all the data (no partition) and finally we apply it to the external validation for performance assessment.

Depending on the computation power that we have, we will use a bigger K or not.

K-Fold Cross validation is similar to Random Subsampling

- The advantage of K-Fold Cross validation is that all the examples in the dataset are eventually used for both training and testing

As before, the true error is estimated as the average error rate

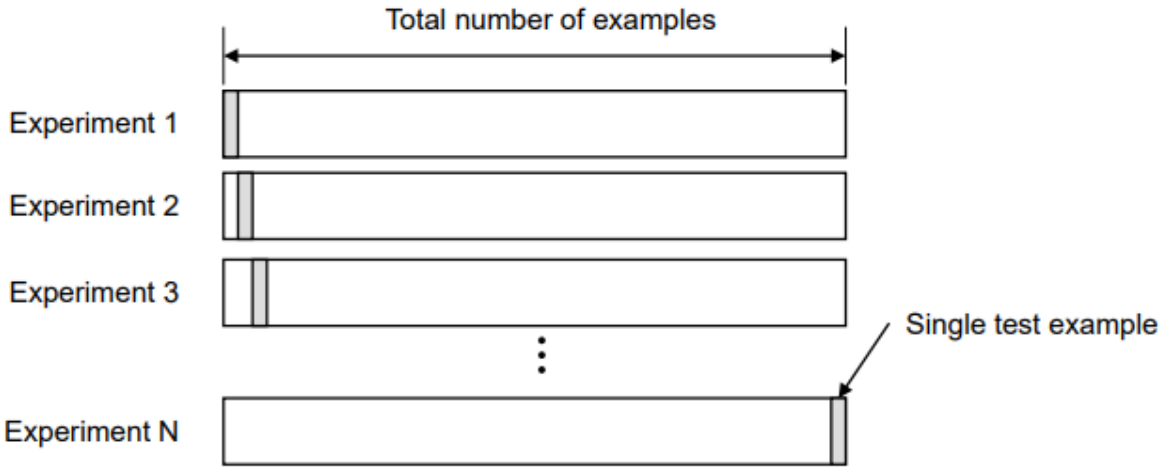
$$E = \frac{1}{K}\sum_{i=1}^{K} E_i$$

Leave-one-out Cross Validation

Leave-one-out is the degenerate case of K-Fold Cross Validation, where K is chosen as the total number of examples. Used when there are very small samples.

- For a dataset with N examples, perform N experiments
- For each experiment use N-1 examples for training and the remaining example for testing

The benefit is that we maximize the training set



As usual, the true error is estimated as the average error rate on test examples

$$E = \frac{1}{N}\sum_{i=1}^{N} E_i$$

How many folds are needed?
With a large number of folds

- Proportion of training data to test data for each is experiment is large.
- The bias of the estimator will be small (more training samples)
- The variance of the true error rate estimator will be large (few validation samples)
- The computational time will be very large as well (many experiments)

With a small number of folds

- The number of experiments and, therefore, computation time are reduced
- The variance of the estimator will be small (more validation samples)
- The bias of the estimator will be large (few training samples)

In practice, the choice of the number of folds depends on the size of the dataset

- For large datasets, even 3-Fold Cross Validation will be quite accurate
- For very sparse datasets, we may have to use leave-one-out in order to train on as many examples as possible

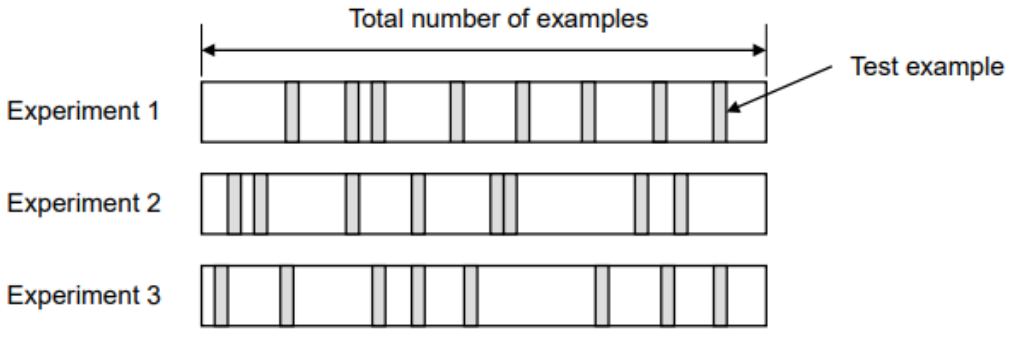
A common choice for K-Fold Cross Validation is K=10

Random subsampling
It is an evolution of K-fold. You decide a fraction of samples for validation (20%) and then you resample the dataset as many times as you want.

- In k-fold, if K=4 you do 4 experiments
- In this case, you can do as many experiments as you want

Random Subsampling performs K data splits of the dataset

- Each split randomly selects a (fixed) no. examples without replacement
- For each data split we retrain the classifier from scratch with the training examples and estimate Ei with the test examples



The true error estimate is obtained as the average of the separate estimates Ei

- This estimate is significantly better than the holdout estimate

$$E = \frac{1}{K}\sum_{i=1}^{K} E_i$$

Using this, we can have a very unbalanced dataset. For example, we have all males in the training set and none in the test set (by chance).

- There is something called “stratification”, which refers to having the same proportion of the outcome on the training and validation. So, we are balancing the partition.
    - If in the dataset we have 70% males, in the training and test set we will have the same proportion.

Repeated K-fold

Repeated K-fold is an improvement of K-fold where after each k-fold, samples suffer a random order permutation, and the process is repeated.

- So, I perform a k-fold, I souffle the data and make another k-fold, I souffle the data and make another k-fold…

In practice the performance of repeated k-fold is similar to random subsampling.

For repeated 4-k fold:

1. I have the samples ordered in some manner
2. I take the first 25%
3. I souffle the data
4. I take the 25%...

It is more computationally expensive, but it gives better results. But in k-fold and repeated k-fold, you are ensuring that every sample appears once in every k-fold experiment. While in random subsampling, since it is random, you do not have control of how many times every sample appears.

All these techniques are designed to control the complexity of the model so that it does not overfit.

Bootstrap (best one)

The bootstrap is a resampling technique with replacement. We make a number of random experiments to estimate the figure of merit of your model.

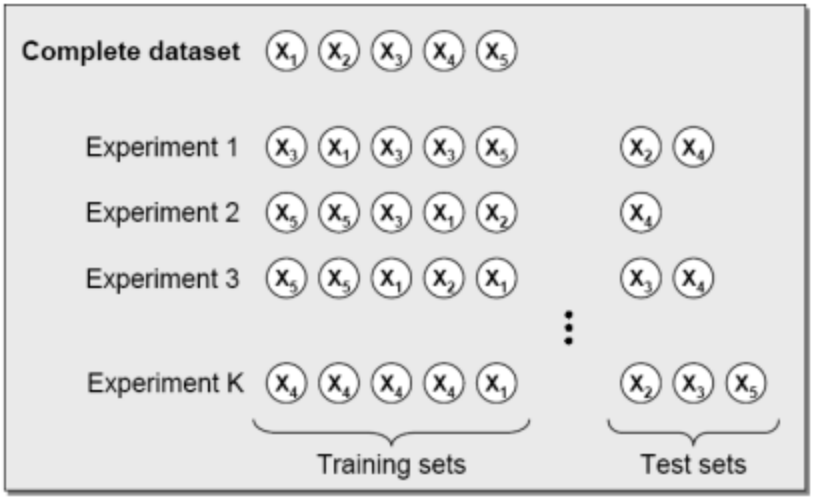
Bootstrap is similar to random subsampling. But the difference is that:

- In bootstrap you have sampling with replacement and you sample as many times as samples you have. If you have 100 samples, you make 100 experiments.
- In random subsampling, it is without replacement.

Here we have an experiment with 5 samples.

- We select one sample for the training set, we put the sample inside the dataset.
- We select another sample for the training set, we put the sample inside the dataset.
- We do this 5 times.
- The remaining samples that have never been chosen go to the test set.

Note that we can select the same sample multiple times since we have replacement.
Note that it could happen that no samples go to the test set and then we must discard that experiment. Or that we have the same sample for the training set all the time (so, you can not train anything) and you will also have to remove this iteration.
Note that we could repeat this experiment as many times as I want.



- From a dataset with N examples
  - Randomly select (with replacement) N examples and use this set for training
  - The remaining examples that were not selected for training are used for testing
    - This value is likely to change from fold to fold
  - Repeat this process for a specified number of folds (K)
- As before, the true error is estimated as the average error rate on test examples

There are 2 problems here:

- You can have repeated samples and many algorithms do not like this.
- In every experiment, the number of data that you leave out is different.

Results on Cross-Validation

We can have overfitting if I use all my data for training.

Your estimation in internal validation is always more overoptimistic than the final performance in external validation. So, when you estimate the figure of merit in internal validation it is going to be better than the one in external validation.

Here we have the result of a numerical experiment where I divide a single class in 2 at random and I try to classify.

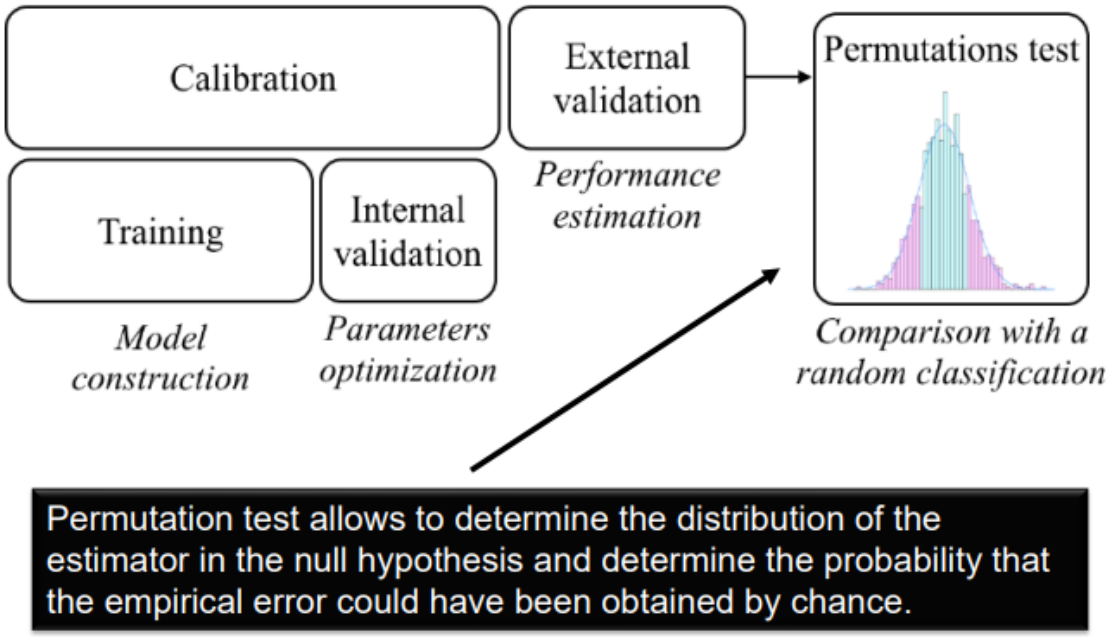
- In internal validation I will have a CR of nearly 67%, despite I know that the accuracy must be 50% because there is no real separation between the 2 classes.

This degree of overoptimism depends on the cross validation method that we use.

- The one that provides a lower error is bootstrap, then Repeated K-fold (BLP) or Random Subsampling

We have been doing everything to decide the complexity of our algorithm (k-fold, bootstrap…). Then we fuse everything, we retrain the algorithm, we apply the algorithm in external validation and then, as a final test it is important to do a permutation test to check the distribution of the CR in the null hypothesis.

- Null hypothesis in a classification is: There is no difference between the 2 classes.



Permutation test consists in destroying the relationship between the X and Y block (thing we want to predict), so we souffle the Y (target) and we train the algorithm and we obtain a CR. We souffle again and we train the algorithm again…

Then we get a distribution of the CR under the Null Hypothesis that typically is centered at 0.5.

You can calculate the percentile of your actual CR and see how far you are from the null hypothesis. Then you can say: My classification rate is 0.95+-0.3 and with a p = 0.00125 (for example)

- So, we provide the CR estimated in external validation with the associated uncertainty (calculated with the binomial distribution) and on top of that you provide a p value by a permutation test to estimate the distribution of the null hypothesis.

Selection of non-linear features

Algorithm 1: Apply a set of rules in order to define the clusters
Algorithm 2: I know the labels and I want to propose the set of rules that are used to create the clusters.

The difference is that in algorithm 2 we are using a supervised algorithm (we have labels).

Unsupervised: I apply a set of rules to define the clusters. So, I have an algorithm that explains to me how to cluster the individuals. For example:

- PCA: The set of rules are that independent variables are highly correlated to each other…

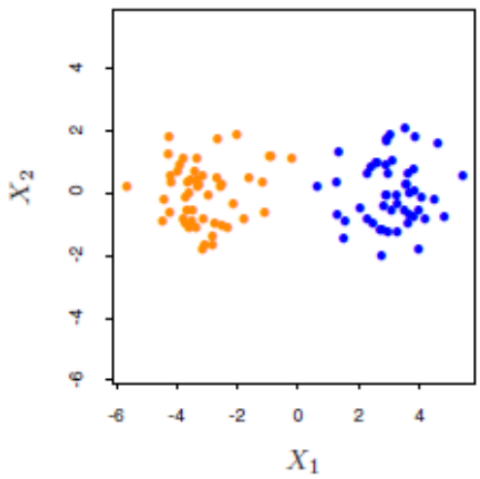
Supervised: I have the data with their labels and therefore I want to use an algorithm to generate a statistical model that will contain the rules I should apply in order to get those labels. So, in this case I want to know which are the rules used to classify the data.

When do we use unsupervised analysis? To explore the data, for example.
The first thing you must do is to clean the data (exploratory analysis):

- If there is an outlier, why? It reflects something of the data or is it a problem of the codification (mistake)
- Is there missing data, why? Someone forgot to put it or is there a semantic pattern for the missing data.

Remember that unsupervised analysis is based on a set of assumptions. Thus, I need to understand those assumptions.

Here we can see 2 clusters, based on color or distance. But depending on the assumptions that you are making you will obtain a different result. For example, which type of distance are you using?



Which criteria should we use to cluster points?

We can use 2 approaches:

- **Partitional**: I have a set of individuals/points and based on a set of rules I split in k clusters. Usually, each cluster will be defined by the properties of the individuals that belong to that cluster. The point that represents all the points of the cluster is called a "prototype".
  Define groups by iteratively optimizing an objective function. Need of number of prototype points to define what a cluster is. They are also called prototype-based clustering algorithms.

  The clustering can be done by just applying a set of rules based on distance or we can imagine that the clusters are representative of statistical models that generate the data- (generative models).

- **Hierarchical**: We are not doing clusters but analyzing the relationships between the items that I have in my set. I am clustering the items iteratively, so I end up with a tree. Once I have the tree, I will decide a posteriori a cutoff that says "After this cutoff, each tree is different from the other ones (cluster)".

  Iteratively clusters points creating a (binary) dendrogram. Clusters are automatically defined by setting a threshold in the deepness of the tree.

K-means

K-means is the classical unsupervised clustering analysis.

Algorithm 13 *K*-Means Clustering

1: Select K points as initial centroids.
2: **repeat**
3: Form K clusters by assigning each point to its closest centroid.
4: Recompute the centroid of each cluster.
5: **until** convergence criterion is met.

We just want to reduce the Sum of the Squared Errors (SSE) within each cluster.

$$SSE(C) = \sum_{k=1}^{K} \sum_{x_i \in C_k} \| x_i - c_k \|^2$$

First sumatory → From cluster 1 to the last cluster K
Second sumatory → For all the points that belong to that specific cluster, compute the euclidean distance (L2 norm) to the centroid of that specific cluster

The centroid of the cluster is defined as the mean of the values for each feature of the elements that belong to that cluster.

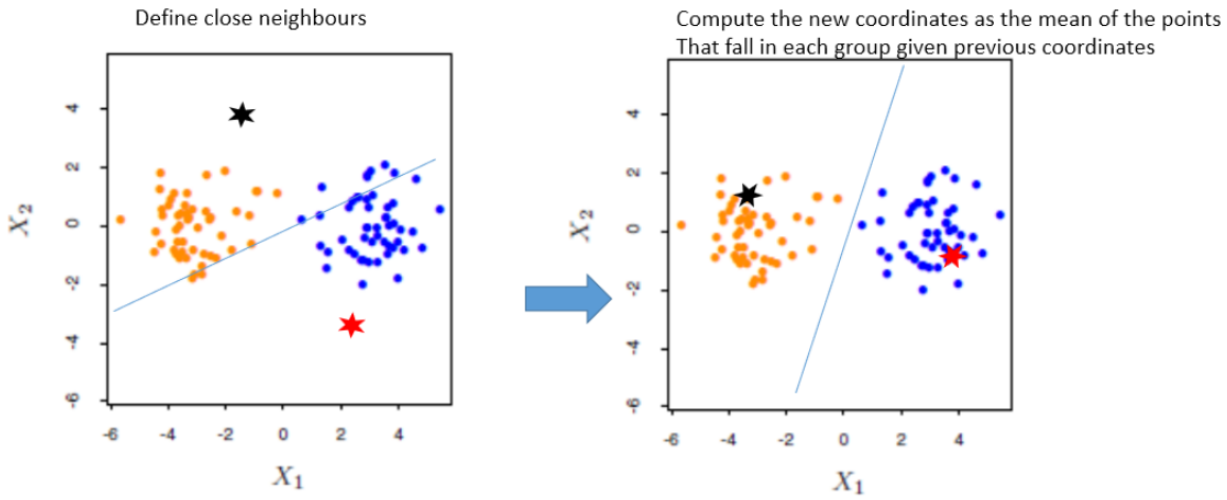
$$c_k = \frac{\sum_{x_i \in C_k} x_i}{|C_k|}$$

Let's imagine that we have the following 2 clusters and we define at random the 2 centroids. As mentioned before, all the points that are closer to one cluster, will belong to that cluster. As we can see, the classification is not perfect because the SSE is not minimized.

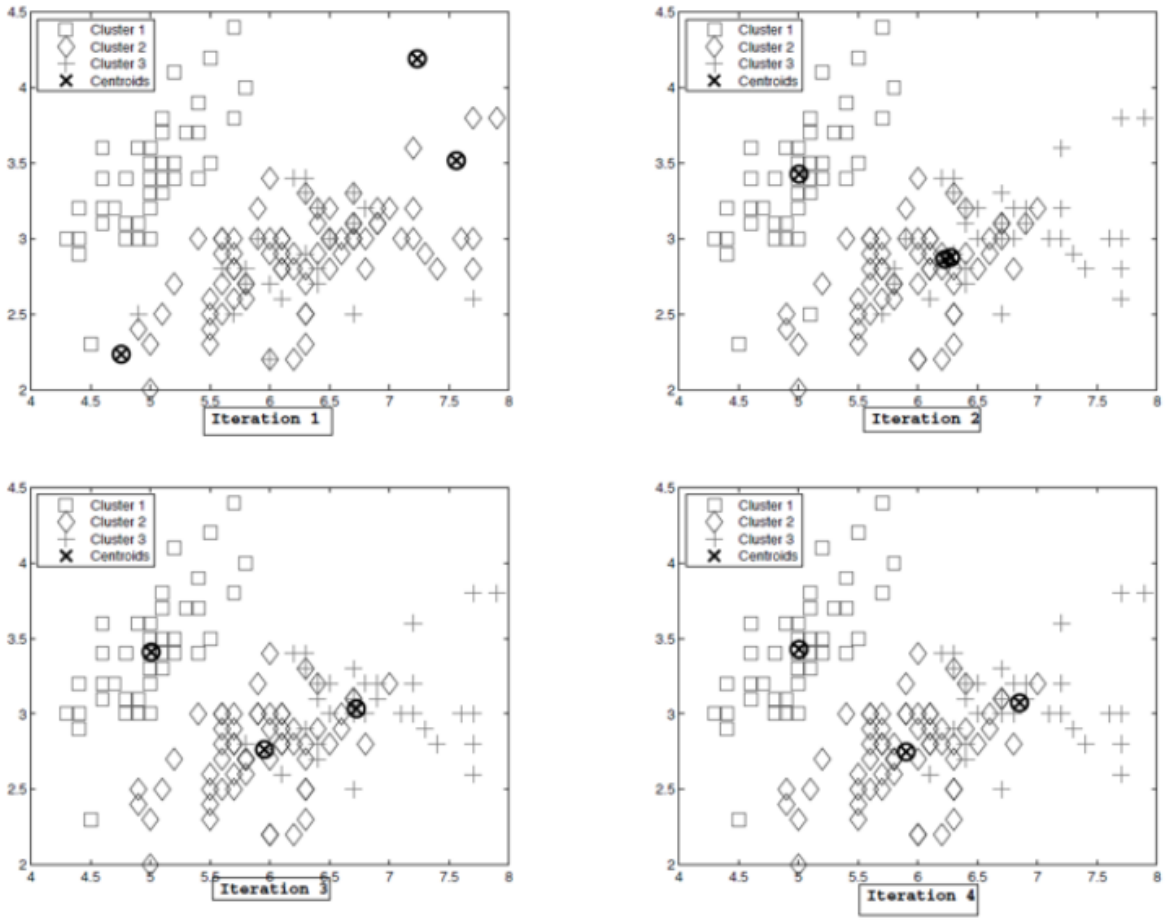
Now we recompute the centroids using the new clusters and we will obtain that some samples that belonged before to cluster orange now belong to the other one.

- **The new centroid will be the mean of the values of each feature of the elements that belong to that cluster**
- Keep iterating until we minimize SSE

If we minimize the SSE, we will obtain a centroid that represents the mean of each of the features for each cluster. Thus, we obtain a different classification.



If we do this iteratively, we will obtain the separation of the 2 clusters. Note that in each iteration, the clusters may be different.



It is a greedy algorithm because in each iteration it depends on the previous outcome. Clusters depend not only on the nature of the data, but also on which hyperparameters you use.

- A hyperparameter is a parameter that I need to define a priori before running the algorithm (the number of clusters, the number of iterations, the definition of distance, for example).
- The robustness of the algorithm depends on the hyperparameters we use.
    - But we do not know which is the correct hyperparameter. We need to create a grid of hyperparameters and compare the outputs. If you get the same result for all hyperparameters, then initialization of the algorithm is robust for any given hyperparameter. But, you can also get very different results. In that case, you need to choose the hyperparameter that gives best results.

How do we define the initial centroids?

Choose a random number and it will converge. This will only work if the space of solutions is smooth, since it will always converge. Otherwise (there are a lot of valleys and hills), we can not do this, because depending on the position you start the algorithm will get stuck and we will obtain the wrong solution.

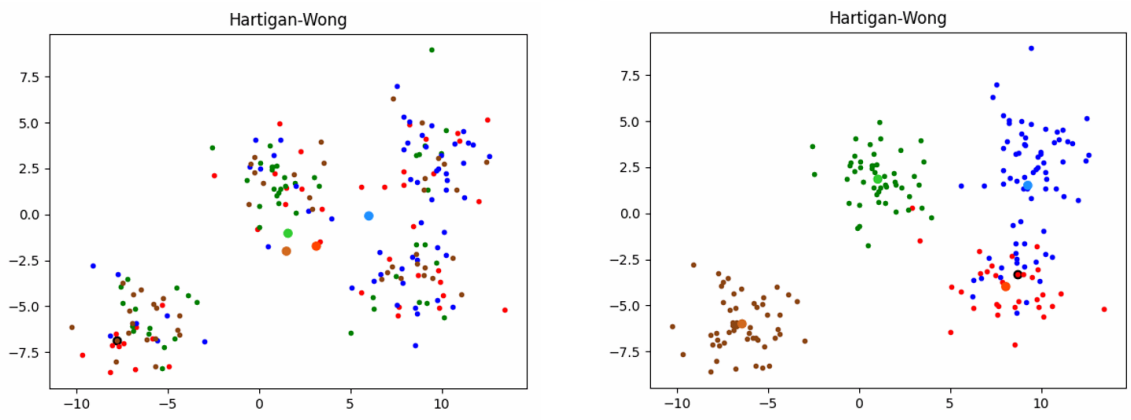
- We can see this because if we repeat the same analysis starting at a different position, we will obtain a different result.
- If we obtain different results, it is a consequence of initializing in different subspaces and it does not converge.

To initialize the points, we can use:

- **Hartigan and Wong:** Divide the dataset at **random** and assign the subsets to one of the clusters. Initially, we will have a complete mess, since each point can belong to any cluster. You run the initialization and hope that as we iterate the algorithm, the centroids are going to be more and more accurate towards the real centroids of each cluster.

  Once the centroids are assigned, we classify the data.

  This algorithm is not good, it is very sensitive to outliers and it takes a lot of time to converge. But if you do not have many points, it works.

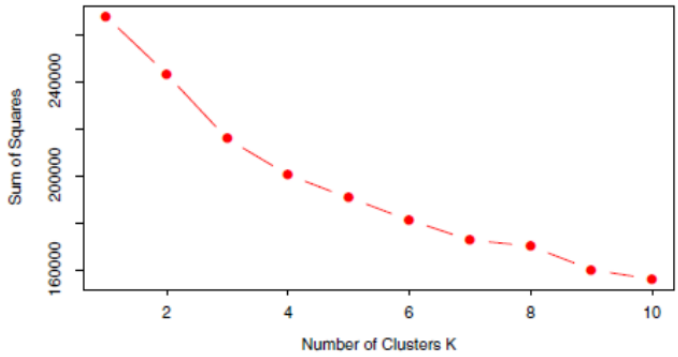


Other strategies:

- **Milligan**: Combines several algorithms. It applies an agglomerative hierarchical clustering and suggests the initial centroids. But you can also get trapped into local minima.
- **K-Means++**: It is based on the fact that the clusters, by definition, should be far away. So, it first selects a random centroid and the next centroid selected is the one which is farthest from the currently selected centroid. This selection is decided based on a weighted probability score. The selection is continued until we have k centroids and then K-means clustering is done using these centroids.

There are many other approaches, which illustrate that there is no optimal solution. So, in addition to testing the different ML algorithms I will also need to test the different algorithms used to define what a centroid is.

Which is the optimal number of clusters? I will need more assumptions. Now I have assumptions about why an individual belongs to a cluster but I also need to add assumptions about how many clusters are created.



As the number of clusters K is changed, the cluster memberships can change in arbitrary ways (things that before were not in the same cluster at k-1, can be in the same cluster at k!).

Clustering means: Finding a common pattern between all the points that belong to that cluster (and that is why they belong to that cluster).

- The pattern that we are finding in K-means is based on distance, since things that are close together belong to the same cluster. They share properties and for this reason the distance between them is small.
    - A pattern is a set of features that are repeated, therefore they are not random.

      When I cluster individuals, we say that a set of individuals belong to the same cluster if they have the same or similar value of the features and therefore their distance is very small and they correspond to the same cluster.

There are many ways to define which is the optimum number of clusters:

- **Calinski-Harabasz Index:** The function computes the scale average distance between the points that belong to different clusters and divides it by the average distance that you observe within each cluster.

$$CH(K) = \frac{\frac{B(K)}{(K-1)}}{\frac{W(K)}{N-K}}$$

Distance of points Between clusters

Distance of points within clusters

Do we want to maximize or minimize this function?

Number of points

Number of clusters

We want to maximize this function because we want the clusters to be as separated as possible and we want the points inside each cluster to be as close as possible.

We just compute this index for multiple K values and we select the optimum K that maximizes the function.

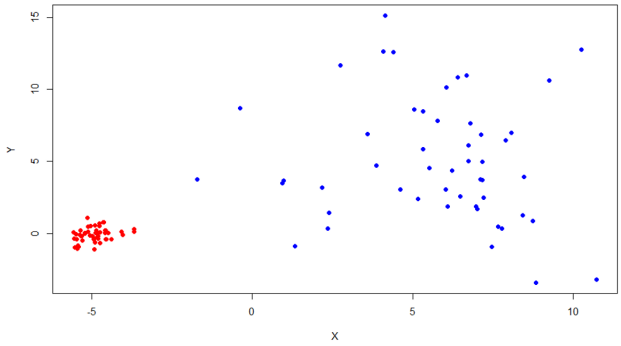
Now we have been talking about unsupervised clustering, but in supervised clustering there is overfitting.

- **Overfitting** means that the model has learned the noise of the data and not the features. The model learns the specific peculiarities of the data and not on the specific peculiarities that any dataset could have. It is saying that each individual is a pattern.

  Example of hair color: We train a model to distinguish hair color. But if the model learns each specific type of color of each sample (brown 1, brown 2…), the model will not be able to know the hair color of a new person. Because it will not find a common pattern since each individual is a different pattern.

When is k-means not going to work?

- When there data is not spherical
- When the variables are not quantitative
- When there are outliers or the standard deviation of the samples of each cluster is very different.



Note that the distances need to be standardize or normalized:

- Imagine I have a set of fruits and I want to classify them based on their shape. Imagine I have 2 different types of variables:
  - Weight in kg
  - If they have a branch on top in cm

  I have a watermelon and an apple. With the following values of weight and branch:
  - 5, 0.2, 5.4, 0.18 Kg
  - 0.3, 0.2, 0.32, 0.18 cm

  I am going to use these features to create a distance. According to the euclidean distance, 0.5-0.2 + 0.3-0.2. The first term will have a bigger distance and therefore I am biasing our analysis towards the variables with the highest values.
  To solve this, I should do a standardization. To do this, you compute:

  standardize value = observation - mean / sd

So, you always need to scale the data.

Variations of K-means: K-medoids

There are some flavors for each algorithm. Meaning that the new variations solve some specific problem of the original algorithm but they have other problems.

In this case, instead of using a centroid that is defined as the mean of each cluster, it selects an individual that will be the centroid or prototype and all the other individuals will go around. So, the individual that is the prototype will be the average pattern of all the individuals that belong to that cluster. **Unlike the centroid, the medoid is an actual data point.**

Algorithm:

- Pick individuals at random and assign them as a prototype
- Assign all the other individuals based on proximity to those prototypes
- Within each cluster, propose a new individual to be the centroid
- Keep iterating

Variations of K-means: K-medians

We have said that k-means is sensible to outliers because the mean, as a statistic, is sensible to outliers.

So, in this case, we use the median to define the prototype of each cluster instead of the mean.

Another approach is the Tukey's trimean (Q1 = First quantile, Q2 = Second quantile, Q3 = Third quantile)

$$TM = \frac{Q_1 + 2Q_2 + Q_3}{4}$$

This is a kind of a mean but it is more robust to outliers.

Variations of K-means: K-mode

It is used to work with CATEGORICAL data more efficiently than K-means (which basically must transform the categories into numbers).

Variations of K-means: Fuzzy K-Means

Instead of saying that a sample belongs to one cluster and another sample belongs to another cluster, it says "you belong to one cluster in probability". This probability is computed based on the distance.

The weight of a point to a centroid is estimated inversely proportional to its distance. Closer points to the prototype of the cluster "j" have more weight. If you are far away, you are not going to contribute much.

So, the new centroid of the cluster is a weighted mean based on the distance of each of the points to that cluster.

Gaussian mixture

We have introduced the concept of probability in the cluster. When computing the centroids, we will give more weight to the points that are closer…

This introduces the concept of “generative models”. So, my data has been produced by a set of probabilistic models and therefore each point belongs to one of those probabilistic models.

- I want to define: To which probabilistic model each point belongs and which are the parameters that define this probability.

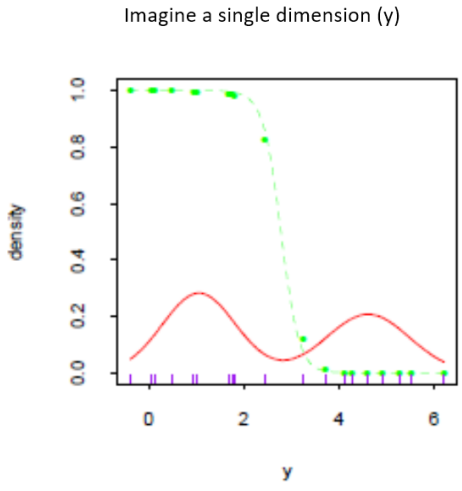
Let's assume that we have 1 feature and have 2 groups. I want to divide my data into 2 clusters. This feature is a mixture of gaussian distributions.

- I can compute the probability that a point belongs to this distribution and the only thing I need to know is, which is the mean and sd of that distribution. If I know these 2 parameters, I will be able to say which is the likelihood that a particular point belongs to that cluster.

Each point of my data is a mixture of probabilities. It can belong to group 1 or 2 with a certain probability. If it can belong to any group, we can not classify them. The thing is that once we surpass a certain probability threshold, we say that the sample belongs to one group.

If we only have 2 clusters, I only need to compute one parameter that tells me which is the probability that I belong to cluster 1. There is no need for a second parameter because by knowing one probability, I can deduce the other.

$$g_\theta(z) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}(z-\mu)^2/\sigma^2}$$



If I do this for one point, I can estimate the likelihood that I belong to one of the clusters given that I know which is the probability that belongs to that cluster.

If I do this for all the points, I can just multiply the likelihood by the different points.

Imagine I have 2 distributions. For a specific point, the probability that it belongs to cluster 1 is 0.7 and the probability that it belongs to the other cluster is 0.3.

- If I know the mean and standard deviation, then I can estimate the likelihood that it belongs to each cluster.
- Given a set of points, which is the probability that each of them belongs to the cluster they say in the probability estimated. You just multiply each probability.

$$L(\theta; \mathbf{Z}) = \prod_{i=1}^{N} g_\theta(z_i),$$

- But if we multiply numbers that are close to 0, we will tend to 0. Thus, we apply a log transformation.

$$\begin{aligned} \ell(\theta; \mathbf{Z}) &= \sum_{i=1}^{N} \ell(\theta; z_i) \\ &= \sum_{i=1}^{N} \log g_\theta(z_i), \end{aligned}$$

EM

We want to estimate to which cluster, in probability, each point belongs and the mean and sd of each of the clusters. Basically, we want to maximize the likelihood using a Expectation Maximization algorithm (EM).

This algorithm has 2 steps:

- Expectation: For each data point, calculate the probability that it belongs to each Gaussian component.
- Maximization: Update the parameters of the Gaussian components to maximize the log-likelihood of the data.

Imagine that I pick the first point.

- I know the probability of belonging to cluster 2.
- I multiply this by the likelihood that I belong to that cluster, given the parameters of that cluster (mean and sd).
- I divide this by the same + the probability that I belong to the other cluster multiplied by the likelihood.
- This formula is similar to Bayes

$$\hat{\gamma}_i = \frac{\hat{\pi}\phi_{\hat{\theta}_2}(y_i)}{(1-\hat{\pi})\phi_{\hat{\theta}_1}(y_i) + \hat{\pi}\phi_{\hat{\theta}_2}(y_i)}, \quad i = 1, 2, \ldots, N.$$

Probability that point *i* belongs to group 2 given the data

Prior probability of belonging to group 2

Likelihood of the data *i* in group 2

- This is the posterior probability that it is going to be used to weight the mean and sd on the gaussian distribution.
    - Normally, when computing the mean each element contributes the same. In weighted means some elements contribute more than others. It is very useful for statistical learning

Given two school classes — one with 20 students, one with 30 students — and test grades in each class as follows:

Morning class = {62, 67, 71, 74, 76, 77, 78, 79, 79, 80, 80, 81, 81, 82, 83, 84, 86, 89, 93, 98}
Afternoon class = {81, 82, 83, 84, 85, 86, 87, 87, 88, 88, 89, 89, 89, 90, 90, 90, 90, 91, 91, 91, 92, 92, 93, 93, 94, 95, 96, 97, 98, 99}

The mean for the morning class is 80 and the mean of the afternoon class is 90. The unweighted mean of the two means is 85. However, this does not account for the difference in number of students in each class (20 versus 30); hence the value of 85 does not reflect the average student grade (independent of class). The average student grade can be obtained by averaging all the grades, without regard to classes (add all the grades up and divide by the total number of students):

mean = 4300/50 = 86

Or, this can be accomplished by weighting the class means by the number of students in each class. The larger class is given more "weight":

*mean = (20*80) + (30*90) / 20+30 = 86*

$$\hat{\mu}_1 = \frac{\sum_{i=1}^{N}(1-\hat{\gamma}_i)y_i}{\sum_{i=1}^{N}(1-\hat{\gamma}_i)},$$ → Weighted mean for group 1 (1-probability of assigning to group 2)

$$\hat{\mu}_2 = \frac{\sum_{i=1}^{N}\hat{\gamma}_i y_i}{\sum_{i=1}^{N}\hat{\gamma}_i},$$ → Weighted mean for group 2

$$\hat{\sigma}_1^2 = \frac{\sum_{i=1}^{N}(1-\hat{\gamma}_i)(y_i-\hat{\mu}_1)^2}{\sum_{i=1}^{N}(1-\hat{\gamma}_i)},$$ → Weighted variance for group 1 (1-probability of assigning to group 2)

$$\hat{\sigma}_2^2 = \frac{\sum_{i=1}^{N}\hat{\gamma}_i(y_i-\hat{\mu}_1)^2}{\sum_{i=1}^{N}\hat{\gamma}_i},$$ → Weighted variance for group 2

In the exam there will be questions about this:

More than two clusters, more than one dimension

- Download the package mclust
  - install.packages("mclust");
- Check the command Mclust
  - ?Mclust
  - Which is the purpose of this command?
  - Which is the input data?
- Create a dummy dataset "result" with 50 entries and 2 dimensions. For each 10 entries, fill the two rows of the matrix with values from a normal distribution with mean and standard deviation different from the others.
  - res <- Mclust(result, G=c(2:30));
  - plot(res);
  - Which is the best number of clusters? How do you estimate them?

Hierarchical clustering

We have seen a type of clustering that partitions the data. What can happen is that you do a clustering k=2 and you obtain 2 clusters. When you apply k=3, you expect that within one of the previous clusters you get another division. But this may not happen because the partitioning may not take into account a hierarchical structure.

- Things that were in different clusters may now be in the same cluster

Hierarchical clustering assumes that data is iteratively clustered into a bigger cluster. So, this does not happen.

- There is a hierarchy of clusters within the topology. So, every time you increase or decrease the number of clusters, they are going to be congruent with the previous clusters that you had.

There are 2 different strategies that we can apply:

- Agglomerative (bottom-up): Sequences that are more similar share a common ancestor, then we build a tree (UPGMA)... We start from the bottom and we end up at the top with a common ancestor.
  Here they do not share a common ancestor but common properties.
- Divisive (top-down)

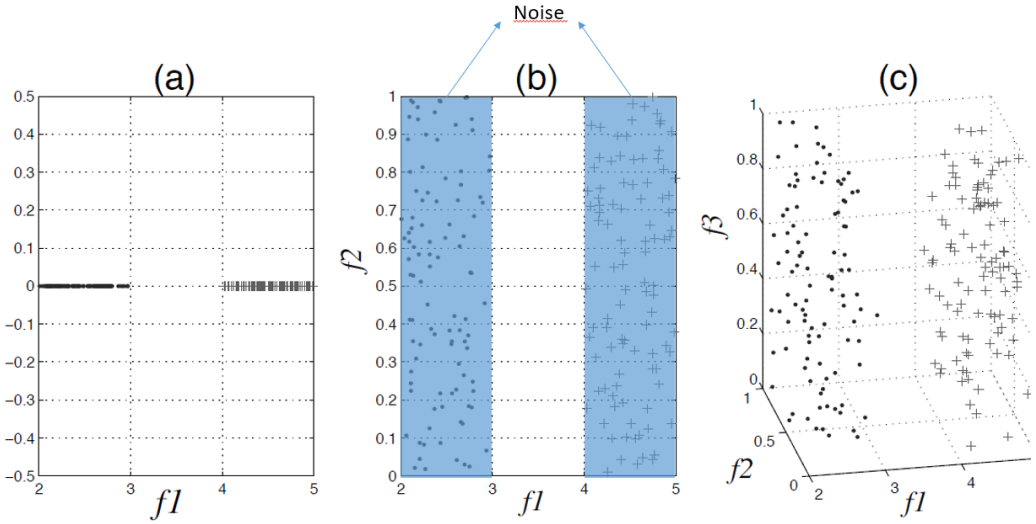
Selection of non-linear features

It can happen that there are features that are non-informative and others that provide noise. Thus, the model will focus on the specificities of those features that are noise, since each individual for those features will be unique. Thus, the model will find spurious patterns (overfitting).

Some features are important for the problem we want to address. So, we want to remove the features that provide noise and keep the ones that are informative for the problem we want to address.

Imagine we have 3 features and we want to classify the items into 2 categories.

- As we can see, feature 1 is the most important, since we can distinguish the 2 clusters.
- If we used only features 2 or 3, we would not see the correct clusters. There is no rank of values where I can separate the different categories.



What happens if you take the 3 features in your model to make the inference? In this case, even the worst algorithm would see that there are 2 clusters based on feature 1.
But if we had a lot of features that are just a little bit informative and other features that provide noise, the model will focus on all the features of each individual point and will not learn a pattern (set of features that occur together) → Overfitting

Curse of dimensionality: As the quantity of features grows, patterns become less distinct, leading to a scenario where each data point possesses its own distinct characteristics (overfitting). Thus, I will not have a subset of distances that allows me to cluster dots that are similar. Because each dot will be equally different.

In order to alleviate this problem (adding features and leading to overfitting), there are different techniques that you can apply. We are going to do a dimensionality reduction:

- **Feature extraction:** Refers to creating new sets of features by combining existing ones. We are not picking but building new features.
    - We are building new features as a mathematical function of the original ones.
    - We have a huge amount of features and I will compress them into very few ones that are informative. For example, PCA (linear combinations, orthogonal), LDA (Linear Discriminant Analysis), SVD.
    - As mentioned, the new features that we get are combinations of the original features. Thus, it is difficult to understand how the model is working and interpret the results. We do not know which feature is more important.
- **Feature selection:** Instead of compressing the data, we pick the features from the original dataset that are more informative for the problem we want to address. So, we are identifying a small subset of features that minimize redundancy and maximize relevance to the target. In this case, we have a better interpretability but pattern identification is constrained to the actual values of the selected features.

Feature selection

There are different strategies that we can apply:

- **Filter model:** You go feature by feature, you apply a test. Imagine you want to classify your individuals in 3 categories, so you go to the first feature and say will I obtain a good separation of those 3 categories using this feature (apply anova)? Then you sort for the ones that better classify your data and select the best ones.

  Problem: Once you have the features that you think are the best ones, you will not use the selected features in the dataset that you have used to select them. You need another dataset, otherwise you will be overfitting, since you know that in that particular dataset, those features are really good. But maybe the selected features are good because of the noise and therefore I need to apply them to another dataset (internal validation) and see if they also work.
    - Maybe, if I have a really large number of features, it could happen that just by chance a feature is good to separate the 3 categories. But this will not work with another dataset. Example bulls eye (each dot is a feature)

  So, in feature selection we will need to divide the training data:
    - Dataset 1: Training dataset used in Feature Selection
    - Dataset 2: Training dataset for the model building
    - Dataset 3: Replication dataset to check out if the model is working
    - Dataset 4: Final test

Instead of just selecting one feature at a time, we can select multiple features.

The problem that we have when we want to select a set of features is which set of features will you pick? If you have 1000 features and you want to select the best 10 ones to make a classification, you have a lot of combinations to do.

Imagine that you have 1.000.000 features. Thus, exploring this space is time consuming and very difficult for the greedy algorithm. Thus, feature selection is a NP-hard problem.

Reverse engineering

By looking at the algorithms that nature applied, not consciously, you can find ways to solve problems that are their own domain.

Look for a natural process that is optimized and decompose it in its steps and apply it for other problems.

Why do natural algorithms work? For example, why does the algorithm bats use to explore space work in other problems? Because each bat is doing a local space search, exchanging information and being able to modify my state to another state based on the information obtained.

We have a problem that we do not know how to solve and we know an algorithm that nature uses. We apply this algorithm to my problem and see if it works. Normally it does work because these metaheuristic approaches that we are discussing are extremely robust.

Evolutionary algorithms

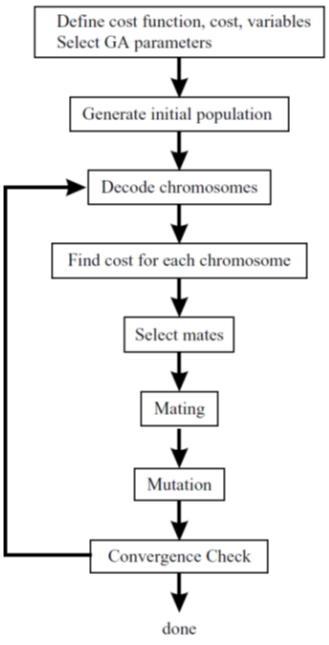
Now, each bat is a chromosome. In the case of feature selection, the problem I want to address is that I want to find the best genes (features) that allow me to solve a classification problem. For example:

- Chromosome 1, features 1, 5 and 9
- Chromosome 2, features 3, 7 and 10

For each of those combinations of chromosomes, I should be able to estimate how good it is for modeling. So, I will be able to select the best chromosome with its selected features.

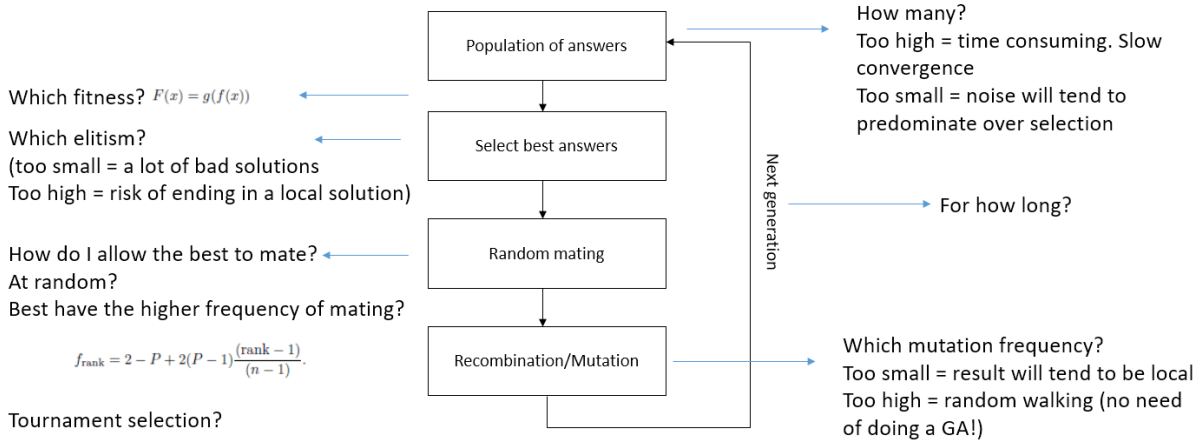
But if I pick another subset of features, maybe I get a better result. So, what I will do is:

- For all the chromosomes selected and with their random selection of features, I will select the best solutions
- Then I will allow them to make new combinations (recombination)
    - The same feature can not be repeated in the same chromosome
- Then we add a mutation to add variability (explore the space of possibilities), otherwise the population will be the **same** in a few iterations.



What is a hyperparameter? Parameter that you need to define a priori in order to be able to run a model.

- Mutation rate
- Size of the chromosome (number of features)
- Number of iterations
- How do we define the cross-over
- Etc



The robustness of an algorithm is associated with the hyperparameters chosen. So, they are highly important and you need to carefully choose them. You may create a grid and compare the results.

- Run the algorithm with different seeds

Tournament selection: Pick a subset of individuals at random from the population and ascertain the best one to reproduce in the next mating.

- So, each individual has the opportunity to reproduce
- This is good if a very bad solution is very close to the best one. So, if the bad solution does not reproduce, we will never reach the best solution.

In the other cases, only the best are able to reproduce.

Linear/Quadratic Bayesian Classifiers

If we want to describe the distribution of these clouds of points in our input space, the easiest thing we can do is to model each class as a gaussian distribution.

In order to introduce a multivariate gaussian distribution, we need the covariance matrix.

- Variances in one diagonal
- Covariances in the other diagonal

$$COV[X] = \Sigma = E[(X-\mu)^T(X-\mu)] \approx \frac{1}{N}(X-\mu)^T(X-\mu)$$

$$= \begin{bmatrix} E[(x_1-\mu_1)(x_1-\mu_1)] & \dots & E[(x_1-\mu_1)(x_N-\mu_N)] \\ \dots & \ddots & \\ E[(x_N-\mu_N)(x_1-\mu_1)] & \dots & E[(x_N-\mu_N)(x_N-\mu_N)] \end{bmatrix} = \begin{bmatrix} \sigma_1^2 & \dots & c_{1N} \\ \dots & \dots & \\ c_{1N} & \dots & \sigma_N^2 \end{bmatrix}$$

Depending on the correlation coefficient, we can have different distributions (as we saw in the first practical).

Multivariate Normal or Gaussian density

We are familiar with the univariate gaussian distribution, which has a bell curve and mean and sd as parameters.

This is the formula for the multivariate gaussian:

- X is a vector
- mu is the mean vector
- ∑-1 is the inverse of the covariance matrix

$$f_X(x) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left[-\frac{1}{2}(X-\mu)^T\Sigma^{-1}(X-\mu)\right]$$

Bayes Theorem

It allows to reverse the conditional probability

$$P[B_j \mid A] = \frac{P[A \cap B_j]}{P[A]} = \frac{P[A \mid B_j] \cdot P[B_j]}{\sum_{k=1}^{N} P[A \mid B_k] \cdot P[B_k]}$$

For pattern recognition, Bayes Theorem can be expressed as:

$$P(\omega_j \mid x) = \frac{P(x \mid \omega_j) \cdot P(\omega_j)}{\sum_{k=1}^{N} P(x \mid \omega_k) \cdot P(\omega_k)} = \frac{P(x \mid \omega_j) \cdot P(\omega_j)}{P(x)}$$

- **$P(\omega_i)$** *Prior probability* (of class ω_i)
- **$P(\omega_i|x)$** *Posterior Probability* (of class ω_i given the observation ***x***)
- **$P(x|\omega_i)$** *Likelihood* (conditional prob. of ***x*** given class ω_i)
- **P(x)** A normalization constant that does not affect the decision

We want to estimate which is the probability of the class given the feature vector.

- We want to estimate the probability of each class after the measurement

We can express this posteriori probability to the class given the vector as a function of the probability of the vector given the class.

We do a measurement and with this information we want to know the probability of belonging to each class.

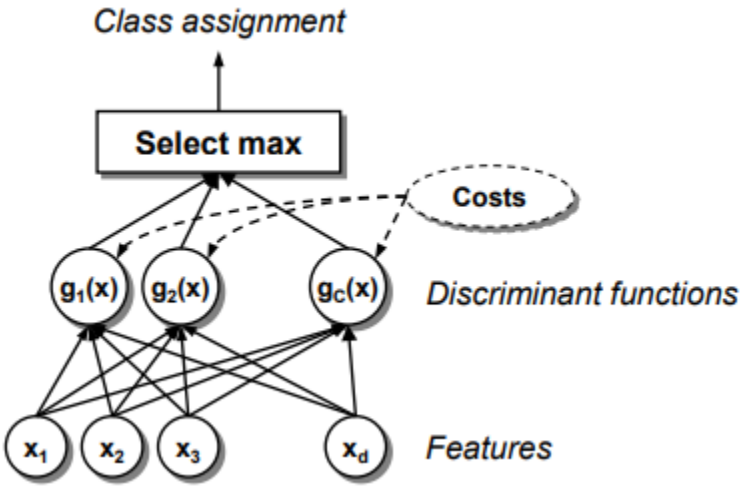
- We take the class that is maximized → The one that has the maximum posteriori probability. By doing this, we are minimizing the error!

On the other hand, the probability of the vector given the class is something that we can model. We are going to model these distributions per class as gaussians.

- To do this we need the mean vector and covariance matrix per class.

Discriminant functions

We have a feature vector, then a number of discriminant functions (as many as classes we have) and we will take the maximum.



These discriminant functions will be related to the posteriori probability of the class given the vector.

Quadratic classifiers

Let's assume that the likelihood densities are gaussian. If I have 3 classes, each one will have a formula like this one:

$$P(x \mid \omega_i) = \frac{1}{(2\pi)^{n/2}\left|\sum_i\right|^{1/2}} \exp\left[-\frac{1}{2}(x-\mu_i)^T \sum_i^{-1}(x-\mu_i)\right]$$

Using Bayes rule, the Map discriminant functions (take the class that has the highest probability) become:

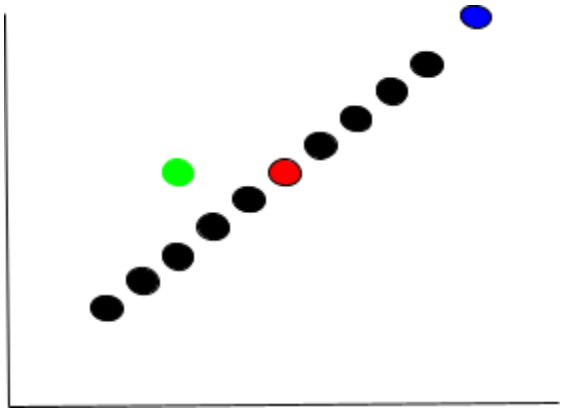
$$g_i(x) = -\frac{1}{2}(x-\mu_i)^T \sum_i^{-1}(x-\mu_i) - \frac{1}{2}\log\left(\left|\sum_i\right|\right) + \log(P(\omega_i))$$

This is known as a **Quadratic Discriminant Function**
The quadratic term is known as the Mahalanobis distance (first term: from the first to the last minus).

Mahalanobis distance

The centroid of the class is in red.

- The black points correspond to that class

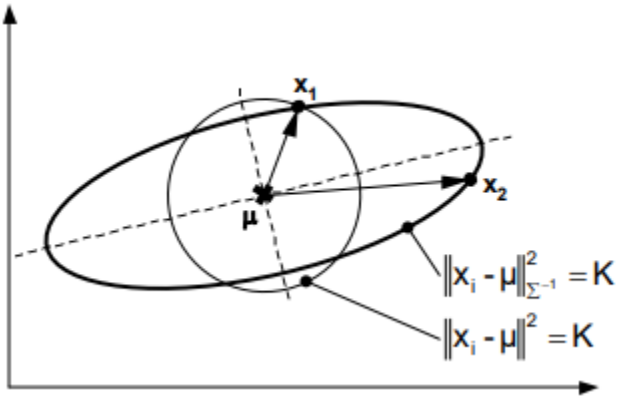


Now I add 2 more points. Which is closer to our data? In euclidean terms, the green point is more similar. But maybe in this case the euclidean distance is not a good measure of proximity.

- Blue
- Green

The mahalanobis distance is like a contour plot of the gaussian distribution.

- So, it takes into account the gaussian distribution of the data



Now we will see a number of particular cases of the previous formula (simplifications):

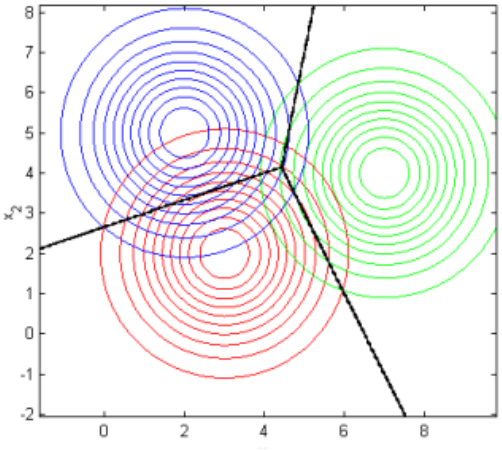
$$g_i(x) = -\frac{1}{2}(x-\mu_i)^T \Sigma_i^{-1}(x-\mu_i) - \frac{1}{2}\log(|\Sigma_i|) + \log(P(\omega_i))$$

Special case 1. Nearest Centroid Classifier

The first simplification: All the prior probabilities and covariance matrices of all the classes are the same and also the covariance matrix is the identity.

Then the discriminant function is:

$$g_i(x) = -(x-\mu_i)^T(x-\mu_i)$$

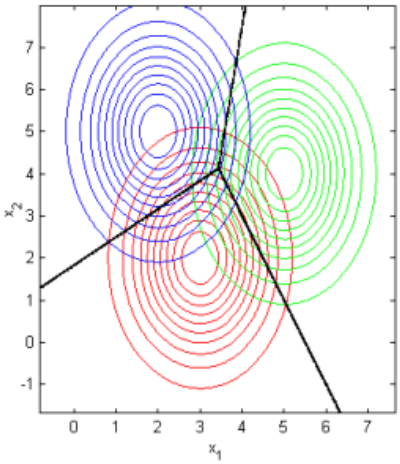


So, the classes are equally spread and the shape of the distribution is such that there is no correlation between the axis and the sd in both dimensions is the same

Special case 2. Mahalanobis Distance Classifier

All the classes have the same prior distribution and covariance matrix, which is diagonal but not the identity.

$$g_i(x) = -\frac{1}{2}(x-\mu_i)^T \Sigma^{-1}(x-\mu_i)$$

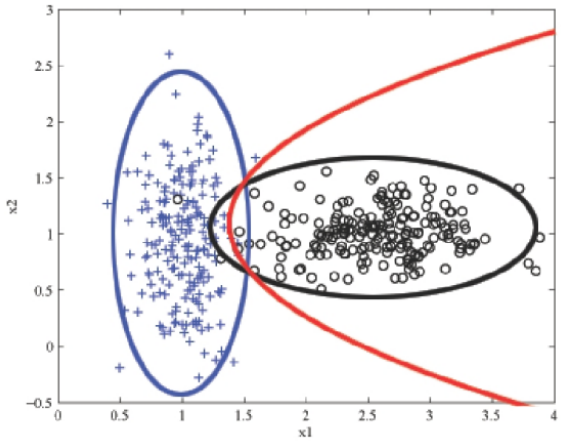


The sd in one dimension (from centroid to the top. Y axis) is different to the other dimension (from centroid to the right. X axis).

Special case 3. Naive Bayes Classifier

They have different diagonal covariance matrices for each class.

- This is why they are oriented in different directions



If I have a general formula, why do I need the other simplifications? This is impacted by the curse of dimensionality. Depending on the case we use, the number of parameters is going to be different.

Introduction to Partial Least Squares - Discriminant Analysis (PLS-DA)

It is a linear discriminant analysis method based on the combination of:

- Supervised Dimensionality Reduction
- Multilinear Regression (default versión) in the reduced space
- Less common ideas (use other algorithms in the reduced space)
    - K-NN, Nearest Centroid, Mahalanobis distance, etc.

Complexity control based on the number of latent variables (Dimensionality). Optimize the number of dimensions.

Main advantage:

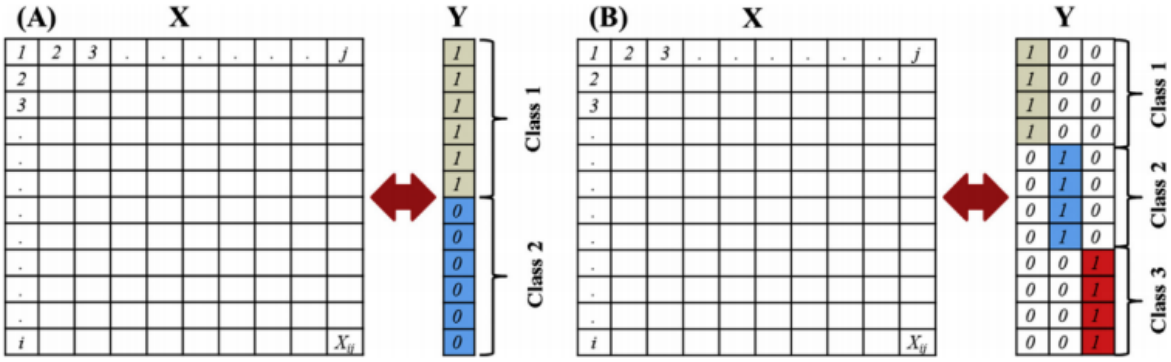
- Adapted to cases where input dimensionality >> number examples. It has a lot of data visualization capabilities.
- Adapted to multi-class problems
- Overcomes dimensionality reduction limits of LDA

Major disadvantages:

- **Overfits very easily**
- Linear decisión boundaries

Class coding

We have the X block (features and samples) and the Y vector that contains the label. If we have a binary decision problem, we can encode it in 1 or 0 (Y needs to be numeric). If we have multiclasses, we use the one-hot-encoding.



The output of the algorithm is not a single number but a vector. If we have 3 classes, the algorithm will provide a vector of length 3 with the probability of each class (the sum of the vector is not normalized to 1).

To assign a class, we just take the maximum.

But this can be changed. Imagine that you obtain the probabilities 0.2, 0.1 and 0.3. Then, under the previous rule, you would select the class that has the highest probability. But the values are really small, so maybe it is better to say that we can not make the prediction.

PLS-DA scores plot vs PCA

PLS-DA finds directions in the input space that take into account the correlation with the Y vector.

- In PCA you just look at the directions of maximum variance in an unsupervised manner.

PLS-DA does not maximize the same figure of merit as LDA.

It is a supervised algorithm that builds a linear model by sequentially identifying "latent variables" or "components," which maximize the variance of the projected vector and the correlation between the projected vector and the target vector.

Algorithm

1. Autoscale input matrix
2. Calculate the inner product between y and every x column:
3. Use those coefficients to find the first latent variable
    a. The first latent variable is a linear combination of the columns weighted by the inner product. So, we are weighting the different columns according to their correlation to the Y.
4. Orthogonalize the input towards the latent variable.
5. Repeat the process. I can repeat it until I have as many latent variables as the number of original dimensions.

The set of zm are the latent variables. The user determines by Cross-validation the correct number of them. Latent variables are orthogonal.

Note that the construction of the latent variables uses the capacity of every direction to predict the output.

It can be proved that PLS latent variables maximize:

$$Corr^2(y, Px)Var(Px)$$

where P is the projection of X into the LV.

Var(Px) → PCA only maximizes the variance of the projected vector
Corr^2 (y, Px) · Var(Px) → PLS-DA also correlates the projected vector and the target vector.

The first latent variable is more correlated to the Y, then the second variable…

PLS-DA Example

Water toxicity for amphipods depending on the content of metals (SEDTOX: Toxicity Database).

Input variables: Concentrations of:
As, Cd, Cr, Cu, Pb, Hg, Ni, Ag

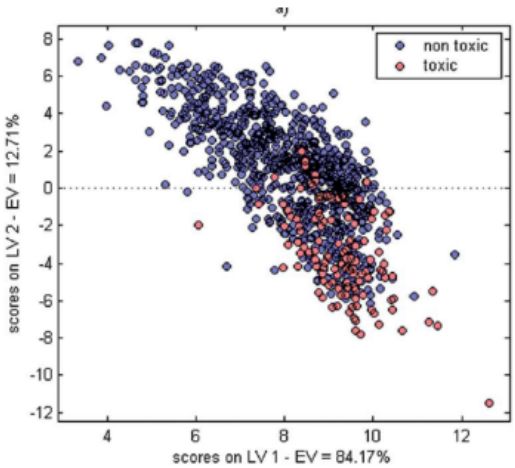
Output variable:
- Toxic: less than 80% individuals survive 10 days
- Non-toxic: More than 80% individuals survive 10 days.

Table 1 Sediment dataset: sample partition in toxic and non-toxic classes, as well as in training and test sets

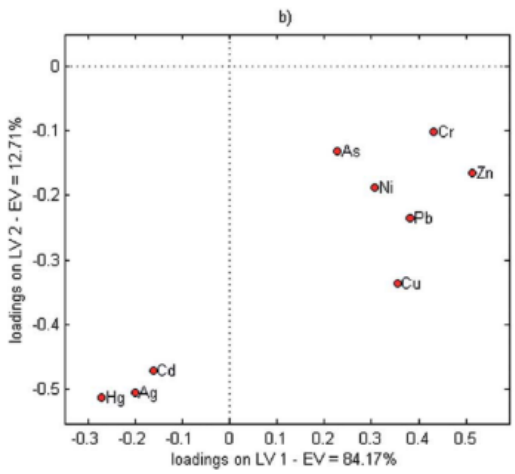
	Non-toxic (class 1)	Toxic (class 2)	Total
Training set	1218	195	1413
Test set	406	65	471
Total	1624	260	1884

Scoreplot → Prediction of the data into the latent variables (as the PCA)
Loading plot → Projection of the individual variables into the latent variables (new subspace). I can see how they are distributed and how they are correlated between them.
- We can see that some metals are correlated and others are orthogonal.
- If the division of the data between toxic and non-toxic follows one of the directions, we can assess which metals are toxic and which not.

Scoreplot



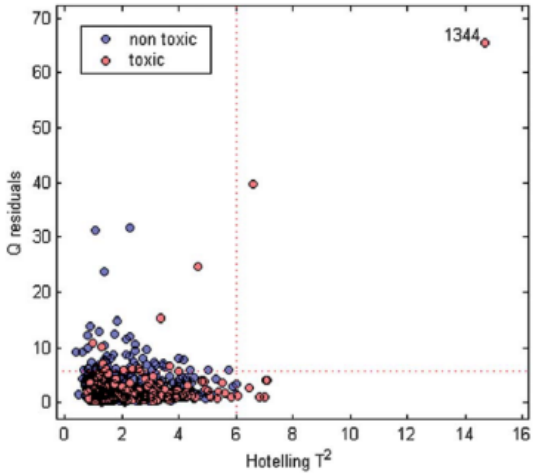
Loading Plot



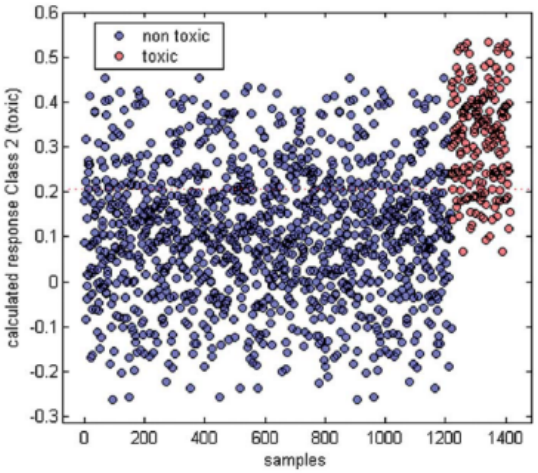
This technique can be used for outlier detection.

- Hotelling T^2 (mahalanobis distance in the reduced space or subspace) vs Q residuals (distance orthogonal to the subspace). So, I can have an outlier for 2 reasons:
  - The data is very far away from the rest of the data in the subspace
  - The data is very far away from the rest of the data in the orthogonal subspace

Outlier Detection

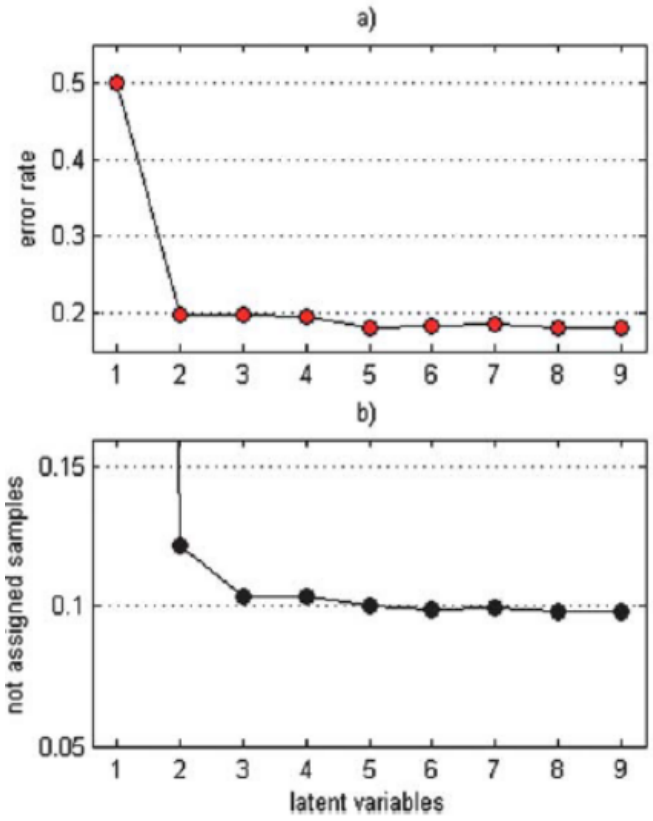


Model Prediction



Model complexity optimization:

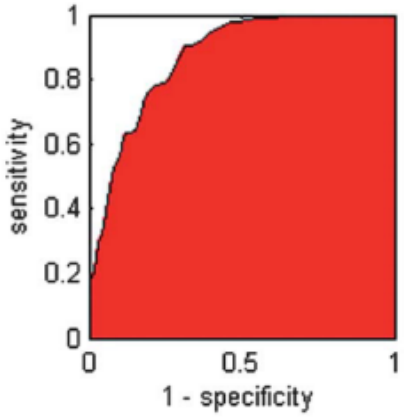
- Total error rate
- Number of not assigned samples



They also look at the number of not assigned samples depending on the number of latent variables they use.
The number of NA samples is reduced when they have 3 latent variables and for this reason they choose 3 latent variables.
Even though the ER is still the same.

Performance assessment:

- Confusion Matrix
- ROC Curve



Summary

PLS-DA is among the most popular classifiers in omics data analysis
PLS-DA combines the visualization options of PCA, with a supervised dimensionality reduction and diverse options of classifiers
Care has to be taken concerning the interpretation of scoreplots due to easy overfitting

Linear and Multilinear regression

Until now we have seen unsupervised learning or supervised learning devoted to predicting a label. Now we are going to predict a number.

There are 2 families of methods:

- Linear methods → Now
- Non-linear methods (covered by oscar)

Linear methods are good for big dimensionalities and a low number of samples.

The idea is that we have a numerical feature vector and the target is a number.

The underlying idea is that there is some function that relates the input to the output + some noise.

$$t=f(x)+noise$$

We want to know what is the function f(x)
Can we predict the target for a given sample? Or even better, what is the probability of a target for a given sample?

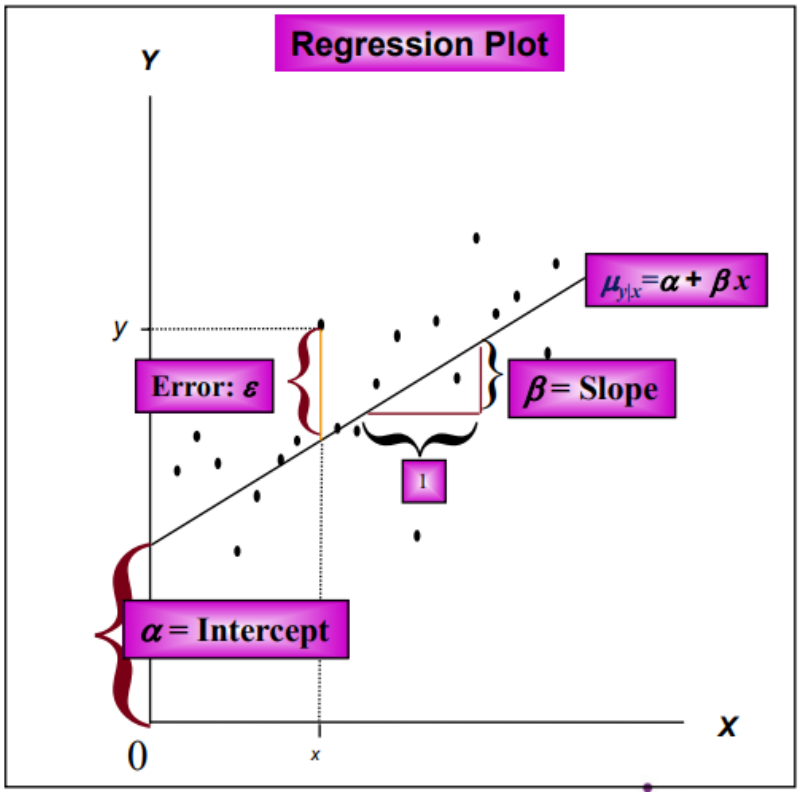
First we need to understand **univariate linear regression.**

We have the following formula that assesses the values of Y (target) for a given X (sample)

$$Y_i = \alpha + \beta X_i + \varepsilon_i$$

Alpha (intercept) and **Beta** (slope) are the parameters of the model.
We want to estimate them!

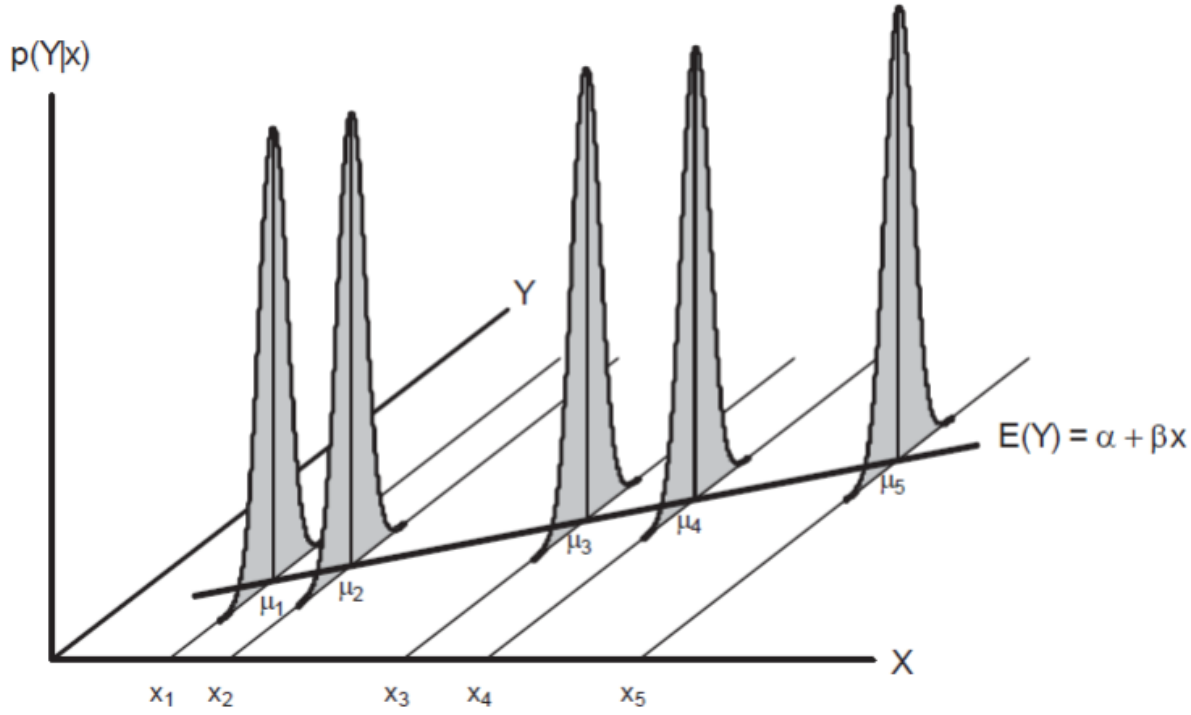
The errors are unknown perturbations that are assumed to be in Y, which is not true.



Assumptions

- **Linearity**: We assume there is a linear relationship and the errors are a bias. The expected value of the error is 0. Thus, the expected value of Y for a given X is α+βX
- **Constant variance (homoscedasticity)**
- **Normality**: The distribution of the errors is normal or gaussian.
- **Independence**: The observations are sampled independently.
- **Error is independent of the X, they are not correlated**

Under all these assumptions, we have something like this:



For a given X, we can have many possible values of Y. As mentioned, the errors follow a gaussian distribution and the mean corresponds to the expected value of Y.

With this model we are trying to minimize the Least Squares Solution or Sum of Squares of the Error (SEE):

- Sum of the squares of the difference between the real value and the prediction

$$SSE(\alpha,\beta) = \sum_{i=1}^{n} (Y_i - \alpha - \beta X_i)^2$$

If all the assumptions are met, then the estimators (alpha and beta) are unbiased and they follow a normal distribution (mean and sd).

- The variance of the estimators is proportional to the variance of the error
- The variance decreases when we have a larger span in X. This makes sense because the variance of the fitted line (model) will be reduced if we have more 15 points separated than if they are all on top of the other.

$$A \sim N\left[\alpha, \frac{\sigma_\varepsilon^2 \sum x_i^2}{n\sum(x_i-\overline{x})^2}\right]$$
$$B \sim N\left[\beta, \frac{\sigma_\varepsilon^2}{\sum(x_i-\overline{x})^2}\right]$$

Theoretically, we do not know the variance of the errors. Thus, we use an estimator of the variance of the error through the variance of the residuals.

Note that the variance of the slope and the variance of the intercept are correlated. Because when we look at the covariance of both, we use their formulas of the variance.

If we know that alpha and beta have some errors associated, we can propagate that to the prediction of Y. So, the standard error for the prediction is:

$$SE(Y_{\text{new}}) = \widehat{\sigma}\sqrt{1+\frac{1}{n}+\frac{(\overline{X}-X_{\text{new}})^2}{\sum_{i=1}^{n}\left(X_i-\overline{X}\right)^2}}.$$

If the new X is very far from the mean X, the SE of the prediction will increase.

Thus, we will be able to create a prediction interval.

In order to have some figures of merit, people use:

- Correlation coefficient
- RMSEP. M is the number of validation samples. Here the error will have the same units. So it is more informative

$$\boldsymbol{R^2 = 1 - \frac{Var(residuals)}{Var(Y)}}$$

$$\boldsymbol{RMSEP = \sqrt{\frac{\sum_{i=1}^{M}(Y_i - A - BX_i)^2}{M}}}$$

Multivariate Regression

The objective is to have the best prediction with the smallest errors for new data.

In this case, instead of dealing with 1 independent variable, we deal with multiple.
For example, we can have 2 independent variables (bilinear model):

$$Y_i = \beta_1 X_{1,i} + \beta_2 X_{2,i} + \beta_3$$

$$\begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} x_{1,1} & x_{1,2} & 1 \\ \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & 1 \end{pmatrix} \begin{pmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{pmatrix}$$

Typically the parameters of the model are estimated by minimizing the square errors of the residuals to the plane.

$$RSS(\beta) = \sum_{i=1}^{N} (y_i - f(X_i))^2$$

In a regression problem with multiple predictors (regressors) it may be convenient to get a sparse model where the prediction power is concentrated in few variables, even at the expense of losing a bit of prediction accuracy.

- So, maybe we can remove some of the predictors because they are not informative.

Ordinary Least Squares (just MSE)

We have our model and we add a vector of inputs X = (X1, X2, … Xn) to predict the output Y.

$$y_k = f(X_k) + \varepsilon_k = \beta_0 + \sum_{j=1}^{p} x_{k,j} \beta_j + \varepsilon_k$$

The method of Ordinary Least Squares aims to minimize the sum of the squared residuals (differences between the observed and predicted values of the dependent variable).

- OLS calculates the coefficients (B1, B2, …, Bn) that minimize the sum of the squared residuals.

Take into account that regressors may be extended by non-linear functions of the original vector, and the formalism remains.

- The prediction is not linear in X
- But the model is linear in the parameters

$$y_k = \beta_0 + \beta_1 x_{k,1} + \beta_2 x_{k,2} + \beta_3 x_{k,1}^2 + \beta_4 x_{k,2}^2 + \beta_5 x_{k,1} x_{k,2} + \varepsilon_k$$

One of the main problems of Multilinear Regression is **Collinearity**.

Two or more independent variables in a multiple regression model are highly correlated with each other (common in metabolomics). This high correlation can cause issues in the estimation of the regression coefficients.

- **Perfect collinearity** occurs when one independent variable is a perfect linear function of another. In other words, the relationship between the variables is exact, and there is no variability in one of the variables that is not already captured by the other. Perfect collinearity occurs when one independent variable is a perfect linear function of another. In other words, the relationship between the variables is exact, and there is no variability in one of the variables that is not already captured by the other.
- **High multicollinearity** refers to a situation where independent variables are highly correlated but not perfectly correlated. In this case, there is redundancy or near-redundancy among the independent variables, making it difficult for the model to distinguish the individual effects of each variable.

There are multiple methods to address collinearity:

- Variable selection
- Regularization techniques like Ridge regression or Lasso, which introduce a penalty term to prevent overfitting and help stabilize the coefficient estimates.

Example: We have the following model.

- We have the X data (samples) in the first matrix, then the 2 parameters and the last matrix is the Y (predicted values)

In the original model, the estimated parameters are B1 = 2 and B2 = 1.

- Note that the matrix X is not singular (but close), we can not calculate the inverse (because we have 3.999 instead of 4).
- In these types of scenarios, the parameters are extremely sensitive to perturbations.

If we add a small error in Y or X, we will have a wrong estimation of the parameters.

Original

$$\begin{pmatrix} 1 & 2 \\ 2 & 3.999 \end{pmatrix}\begin{pmatrix} \beta_1 \\ \beta_2 \end{pmatrix}=\begin{pmatrix} 4 \\ 7.999 \end{pmatrix}$$

$$\begin{pmatrix} \beta_1 \\ \beta_2 \end{pmatrix}=\begin{pmatrix} 2 \\ 1 \end{pmatrix}$$

Errors in Y

$$\begin{pmatrix} 1 & 2 \\ 2 & 3.999 \end{pmatrix}\begin{pmatrix} \beta_1 \\ \beta_2 \end{pmatrix}=\begin{pmatrix} 4.001 \\ 7.998 \end{pmatrix}$$

$$\begin{pmatrix} \beta_1 \\ \beta_2 \end{pmatrix}=\begin{pmatrix} -3.999 \\ 4.000 \end{pmatrix}$$

Errors in X

$$\begin{pmatrix} 1.001 & 2.001 \\ 2.001 & 3.998 \end{pmatrix}\begin{pmatrix} \beta_1 \\ \beta_2 \end{pmatrix}=\begin{pmatrix} 4.000 \\ 7.999 \end{pmatrix}$$

$$\begin{pmatrix} \beta_1 \\ \beta_2 \end{pmatrix}=\begin{pmatrix} 3.994 \\ 0.001 \end{pmatrix}$$

So, when the X matrix is close to singular, the parameters are totally unreliable.

A system of equations is considered to be **well/ill-conditioned** if a small change in the matrix or a small change in the right-hand side results in a **small/large** change in the solution vector

If we compute the **"condition number"** as the ratio of the maximum singular value and the minimal singular value, then this number will tell us how far we are from singularity.

$$cond(\mathbf{X}) = \frac{\lambda_{\max}(\mathbf{X})}{\lambda_{\min}(\mathbf{X})}$$

In this example, cond(x) = 25000

The bigger, the closer to singularity and therefore the worse for the stability of the parameters. The condition number must be < 100

Now we will look at different families of methods that try to solve the problem of collinearity:

- Methods based on **regularization** → Ridge and Lasso regression
- Methods based on **subset selection**
- Methods based on **projections** → PCR and PLS

So, we will do multilinear regression with 3 different strategies

Ridge Regression (L2)

We are willing to make the model more rigid not by decreasing the dimensionality or reducing the number of regressors, but by adding a penalty that makes the space of potential solutions smaller.

So, it shrinks the regression coefficients by imposing a penalty on their size.
The optimal parameters over Ridge are the parameters that minimize the following expression:

- The first part of the expression is OLS
- The second part is the L2 penalty. We penalize the big parameters.

$$\hat{\beta}^{ridge} = \underset{\beta}{\operatorname{argmin}} \left\{ \sum_{i=1}^{N} \left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j \right)^2 + \lambda \sum_{j=1}^{p} \beta_j^2 \right\}$$

λ ≥ 0 is a complexity parameter that controls the amount of shrinkage.
In most of the cases, if we reduce the parameters, we obtain models that are more accurate. So, we need to control lambda.

- The optimal lambda may be found by cross-validation.

LASSO (L1)

It is similar to Ridge Regression but with a different penalty term.

$$\hat{\beta}^{ridge} = \arg\min_{\beta} \left\{ \sum_{i=1}^{N} \left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j \right)^2 + \lambda \sum_{j=1}^{p} \left| \beta_j \right| \right\}$$

The minimization problem is no longer quadratic → **Better interpretability (same units)**
It is known that linear penalty (L1) terms produce sparse solutions. In this sense, the Lasso makes a fast rejection of un-informative features.

- The algorithm will make that some predictions are multiplied by 0 and thus disappear from the model.

In the following graphs, lambda increases to the left.

1. First plot is Ridge Regression
2. Second plot is Lasso Regression

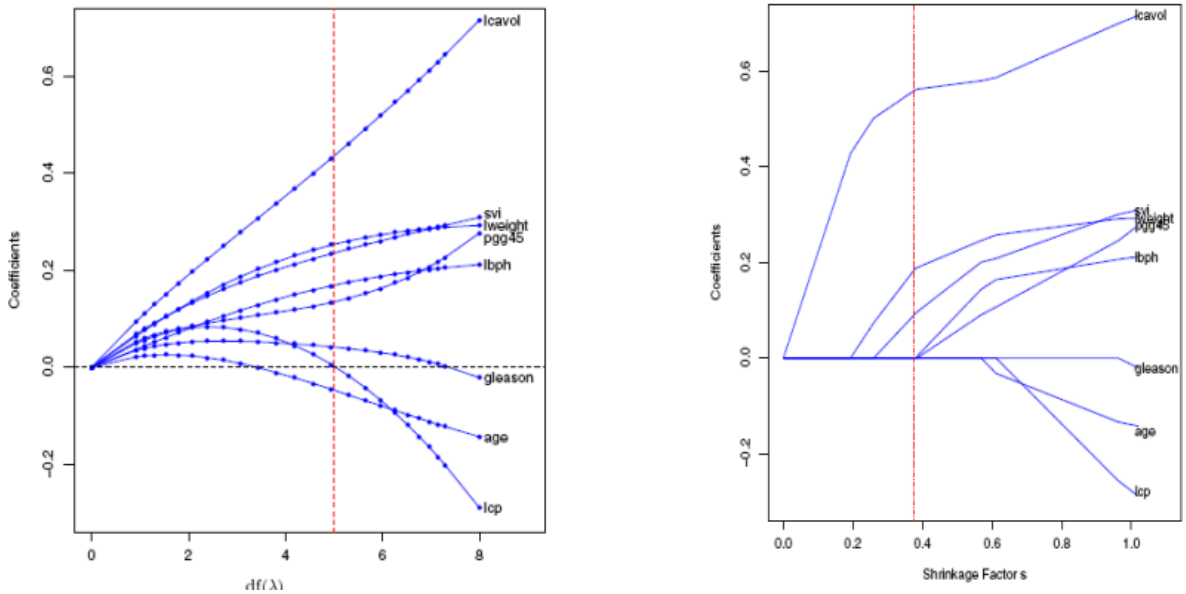
When we increase lambda all the parameters become smaller. At the end all parameters are equal to 0 and they do not predict anything.

The red line is the value of lambda that minimizes the error in cross-validation. So, there is an optimum lambda that produces better results than the OLS.

- So, we have created a simpler model with smaller parameters that has better predictions in internal validation.

The nice thing about Lasso is that most of the variables are equal to 0 for the optimum value of lambda. So, we have reduced the complexity of the algorithm.

- So, it provides a feature selection that is embedded in the algorithm. By optimizing the lambda, we are finding the most relevant parameters of the model.



Note the change of sign in the Ridge regression. It does not make sense.

- But we are not interested in the interpretation
- We are interested in the good results

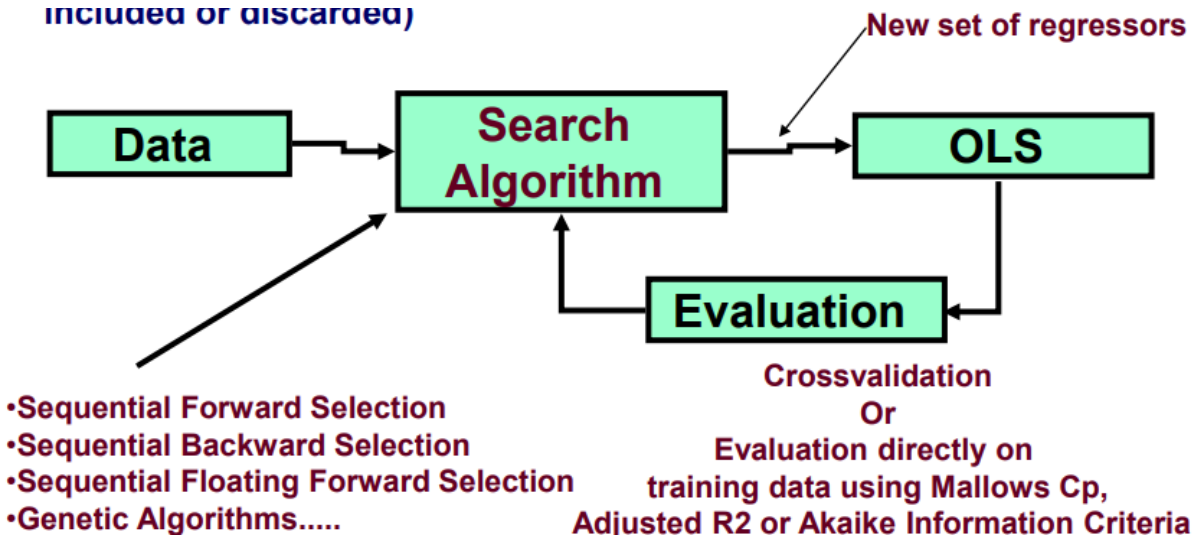
There is another method called “Elastic Net” that combines both L1 and L2 penalties.

Subset Selection (good interpretability)

They are a subset of families that do feature selection.

We have the data, we apply an algorithm that proposes combinations of regressors, then we apply OLS to the combination of regressors and then you evaluate.

- You keep iterating



In the evaluation step there are 2 strategies:

- Cross-validation. You need to do data partition
- Other criteria that you can compute directly on training OLS + Penalty in the complexity of the model

The first algorithm that we can use is the **Leaps and Bounds**:

- Computational efficient algorithm to make an exhaustive exploration

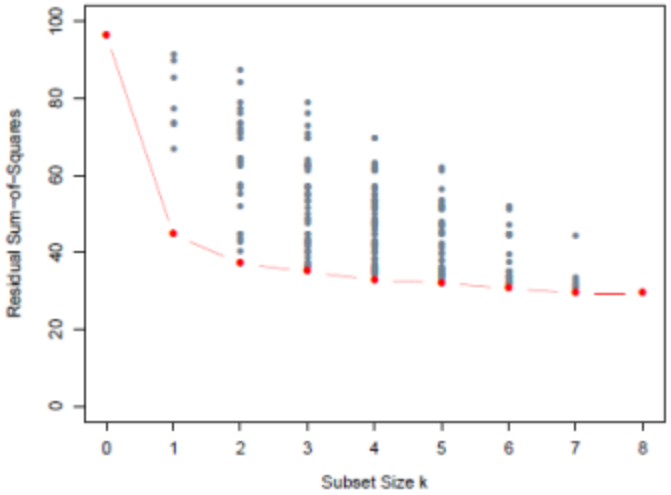
So it is used to find which is the best subset of regressors. If you have 4 regressors, you have 16 possibilities, so you can explore all of them. 2^n

- It can be used in problems with up to 30-40 parameters. Beyond that it becomes too computationally intensive.

At the end, you obtain a plot where for every size of subsets, which is the optimal.

In this case we would use the model with 2 parameters

You need to apply this in TRAINING! Not CV



We also have other algorithms that perform sequential searches.

- **Sequential Forward Selection (SFS):**

FS starts with zero parameters and at each iteration adds the predictor that produces a larger decrease of the error. So, we are adding features according to how good they are individually. We are not taking into consideration that maybe 2 parameters do not cooperate well together.

- We have 1 parameter, then we add another parameter that gives the best result. Then we add another parameter that gives the best result out of all the other parameters…

We are not exploring the whole space of possibilities:

- In the first iteration we explore n parameters
- In the second n-1
- etc

Important: Once a variable has been added it cannot be removed. So, maybe there is a better combination.
Some implementations do FS on the training set, but the recommended practice is to evaluate FS on the internal validation set

- **Sequential Backward Selection (SBS):**

BS starts with the full model and at each iteration it removes the predictor that produces the best improvement in model performance.

- We are removing the worst

In this case, we are also not exploring the whole space of possibilities.

Important: Once a variable has been removed it cannot be added again. So, maybe there is a better combination.
The recommended implementation is to do this in the internal validation set. Not all BS routines do so. Some do it in the training set, and then they just look for the removal of parameters that produce the smaller decrease in the prediction accuracy-

- **Sequential Forward Floating Selection (SFFS):**

We start adding features (SFS), but once adding more features is no longer giving better results, we start removing them (SBS). Once removing features give no better results, we start adding features again.

- We are looking at the optimal point by adding, then by removing, then by adding…

Thus, we are more flexible.

We could also add 3 features, then remove one, then add 3 more… So, there are different strategies.

Note that we are removing the features that are correlated and therefore we do not need them.

Methods based on Projections

The most important ones are:

- Principal Component Regression (PCR)
- Partial Least Squares (PLS) here the target is a number. This is the only difference with PLS-DA (here the target was 0 or 1).

In **PCR** you first perform a PCA and then an OLS on the scores of the PCA.

- The selection of the number of PCA components (hyperparameter) is carried out by checking the prediction accuracy in the internal validation set. We are discarding coefficients.

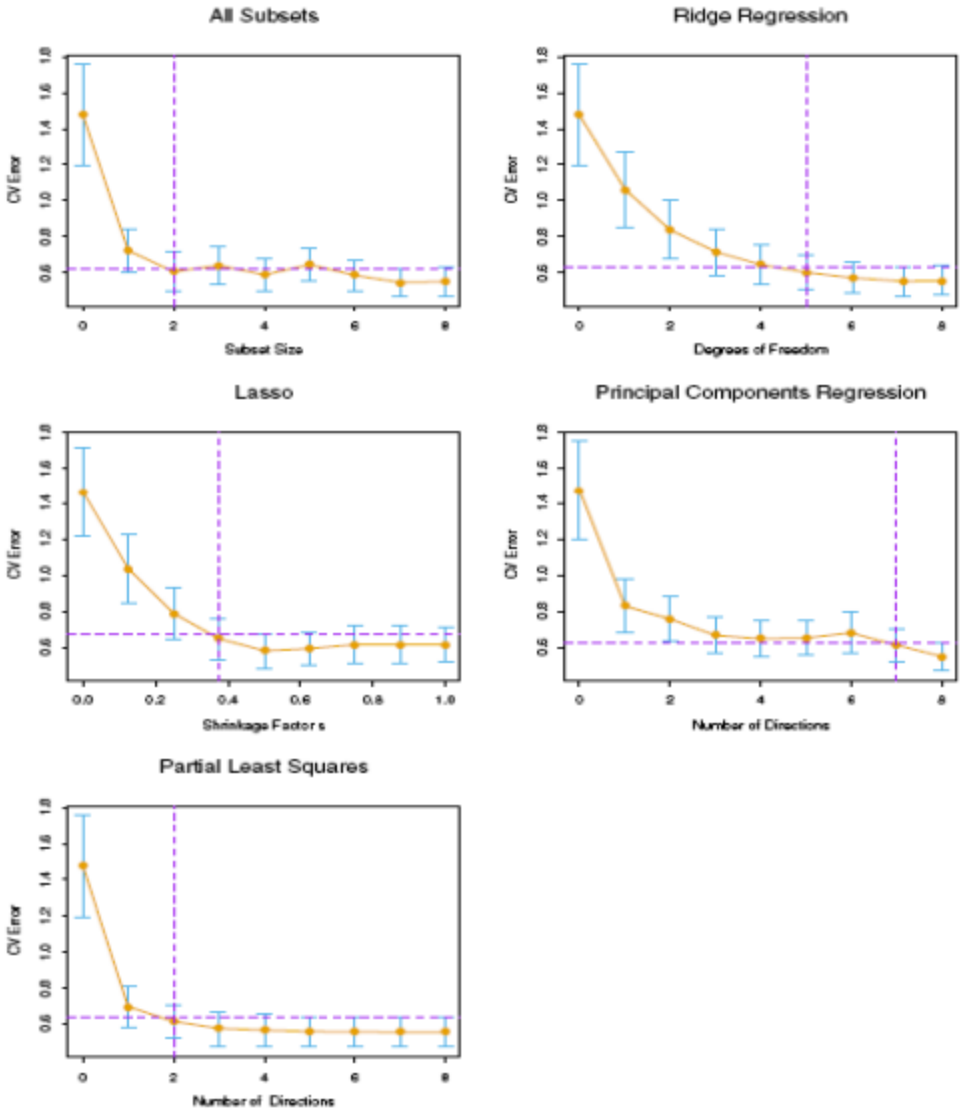
The problem is that the dimensionality reduction is unsupervised. Thus, we are decreasing the complexity but it is blind to the prediction.

- Maybe we are removing dimensions that are important

Comparison:

Most of the algorithms reach a similar level of error.

The only thing that changes is the complexity of the remaining algorithm.



Decision Trees

Easy to use and to interpret.

- We are able to understand how the model is working, which are the best predictors… Thus, we are able to generate new hypotheses to tackle the problem. For example, if we make a decision tree about cancer, there will be a predictor “smoking” that will have a high correlation. Therefore, we will be able to make a hypothesis relating cancer and smoking.

- In other models, the predictors are nonlinear combinations of multiple features and as a consequence it is very difficult to understand.

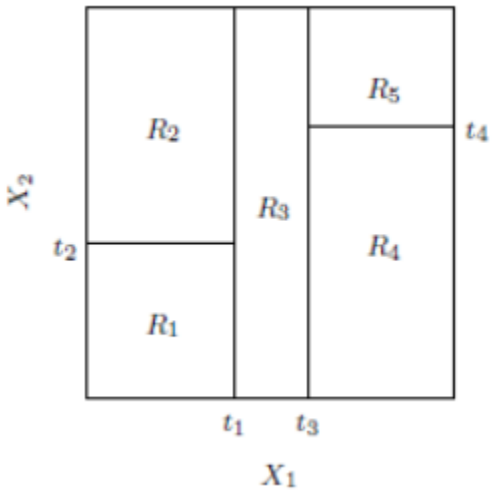
In decision trees is based on rows and we are subdividing each time the data into 2 branches and this decision will be based on a feature. For example:

- We want to predict if covid patients will be hospitalized or not based on a set of features (age, smoker, hypertension…).
  Are you a smoker? Yes or no? Based on your answer, you will go to one branch or another. You do this for all features.

  At the end, you would like to build a tree so that one of the leaves contains 100% of the individuals that are going to be hospitalized. This leaf could contain old people, smokers, lung problems…

  For us, it is very easy to follow and understand this path → Interpretability

If you are doing a regression, each subdivision should have a similar value. So, we can work with categories or with continuous variables (you can also mix them).

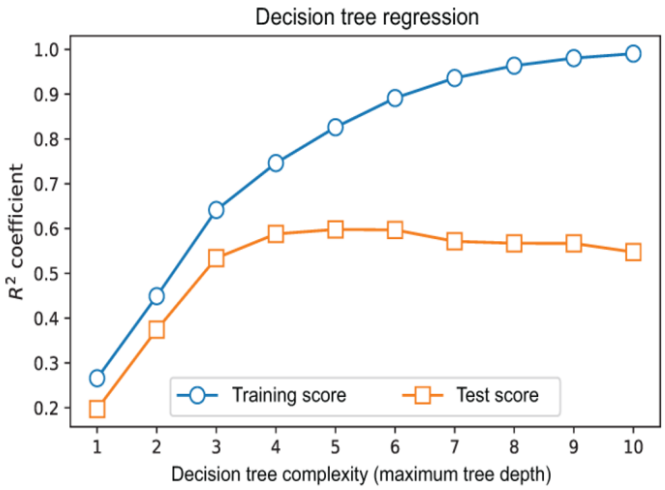


They handle irrelevant data automatically, they are really fast but they are sensitive to initial conditions or variations in the data.

Decision trees also suffer from overfitting. If we make a lot of subdivisions, each leaf will contain a single sample. For example:

- If your name is Oscar, you are bald, you have glasses… you will have covid. This prediction could happen because this patient is the only patient present in the leaf.

As many other models, as we increase the complexity of the model (number of decisions), the more likely I will have overfitting.



One of the solutions to avoid overfitting is to use pruning. You have the whole tree (maybe there is overfitting) and you start cutting some of the branches. As a consequence, in training data the prediction will become less and less good because we are cutting some branches. But this model will be more exportable to other datasets, it will be more robust.

How to build a decision tree:

- Start with the whole dataset and based on a feature you divide the dataset into 2 nodes.
- Then you go to each of the two nodes and based on another features you divide the dataset into 2 nodes…

If you have a continuous feature (number of cigarettes), you will need to define a threshold to divide the dataset.

But which features do you use in each step? There are multiple algorithms to address this problem, such as the greedy algorithm.

Example. You want to predict the performance of a horse in a race in different conditions.

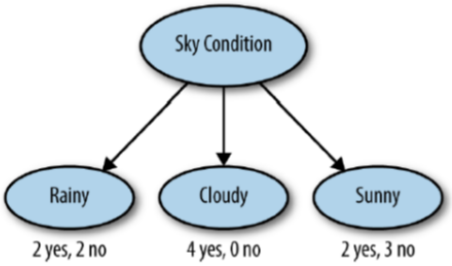
- Sky condition (cloudy, rainy, sunny)
- Wind speed (low, high)
- Humidity (high, normal)

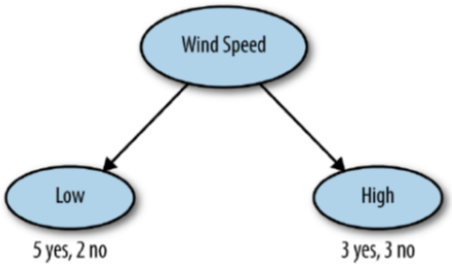
Week	Sky condition	Wind speed	Humidity	Good result
1	cloudy	low	high	yes
2	rainy	low	normal	yes
3	sunny	high	normal	yes
4	cloudy	high	high	yes
5	cloudy	low	normal	yes
6	rainy	high	high	no
7	rainy	high	normal	no
8	cloudy	high	normal	yes
9	sunny	low	high	no
10	sunny	low	normal	yes
11	rainy	low	normal	yes
12	sunny	low	high	no
13	sunny	high	high	no

You want to do a prediction for the next race where:

- Sky condition = rainy
- Wind speed = low
- Humidity = high

Imagine we have the following 2 categorical features:





To do the first split, I would use the feature “sky condition” because this feature splits the data better.

We could compute some statistics to evaluate how good a feature splits the data, one of them is “entropy”. Its a measure of uncertainty:

$$H(S) = -p_+ \log_2 p_+ - p_- \log_2 p_-$$

We want the value to be as small as possible to be certain that it predicts well.

$$H(\text{rainy}) = -\frac{2}{4}log_2\frac{2}{4} - \frac{2}{4}log_2\frac{2}{4} = 1$$

$$H(\text{cloudy}) = -\frac{4}{4}log_2\frac{4}{4} - \frac{0}{4}log_2\frac{0}{4} = 0$$

$$H(\text{sunny}) = -\frac{2}{5}log_2\frac{2}{5} - \frac{3}{5}log_2\frac{3}{5} = 0.97$$

With this value you can then compute the **gain of information/entropy**. It is measurement in bits of how relevant the feature is on scale from 0 (least useful) to 1 (most useful):

$$\text{Gain} = H(S) - \Sigma\frac{S_V}{S}H(S_V)$$

$$H(wind) = -\frac{7}{13}log_2\frac{7}{13} - \frac{6}{13}log_2\frac{6}{13}$$

$$H(low) = -\frac{5}{7}log_2\frac{5}{7} - \frac{2}{7}log_2\frac{2}{7}$$

$$H(high) = -\frac{3}{6}log_2\frac{3}{6} - \frac{3}{6}log_2\frac{3}{6}$$

$$Gain(wind) = H(wind) - H(low) - H(high)$$

We will pick the feature that maximizes the gain of information. It is a greedy algorithm because at each step I decide what I will do based on what I have available.

So, I have the whole dataset and I compute the gain of entropy for each feature. Then I divide the dataset based on the feature that has the highest gain of entropy.

Package "rpart":

- **Conduct pruning:**
  fit.pruned <- prune(fit, cp = fit$cptable[which.min(fit$cptable[,"xerror"]), "CP"])

Random forests

It's a type of ensemble: Algorithm designed to enhance predictive performance for a given task by combining the predictions from multiple estimators or models. Through this approach, an ensemble method builds a meta-estimator.

Example:

- Multiple doctors make a classification about a new patient (cancer or no cancer)
- Then a final doctor makes a final classification based on the classification of the other doctors. It collapses the assessments made from the different doctors that study different fields.

So, we are going to work with different models and an ensemble that will collapse the information produced by the models.

- By collapsing, we will get a better result than each of the predictions made by the independent models. Because we are averaging over the possible overfitting of each of the independent models.

Imagine we have multiple datasets and we compute a decision tree for each of them. These decision trees can suffer from overfitting even if we prune them. But if we have many datasets this problem is solved because each decision tree will make a decision and we will take as a final decision the average decision from all the trees.

- A large number of models are unlikely to all make the same mistake. RF relies on the power of model aggregating or model averaging

The problem comes when we have a single dataset. **How can we generate multiple decision trees using a single dataset?** Using bootstrap.

You have your dataset with your features.

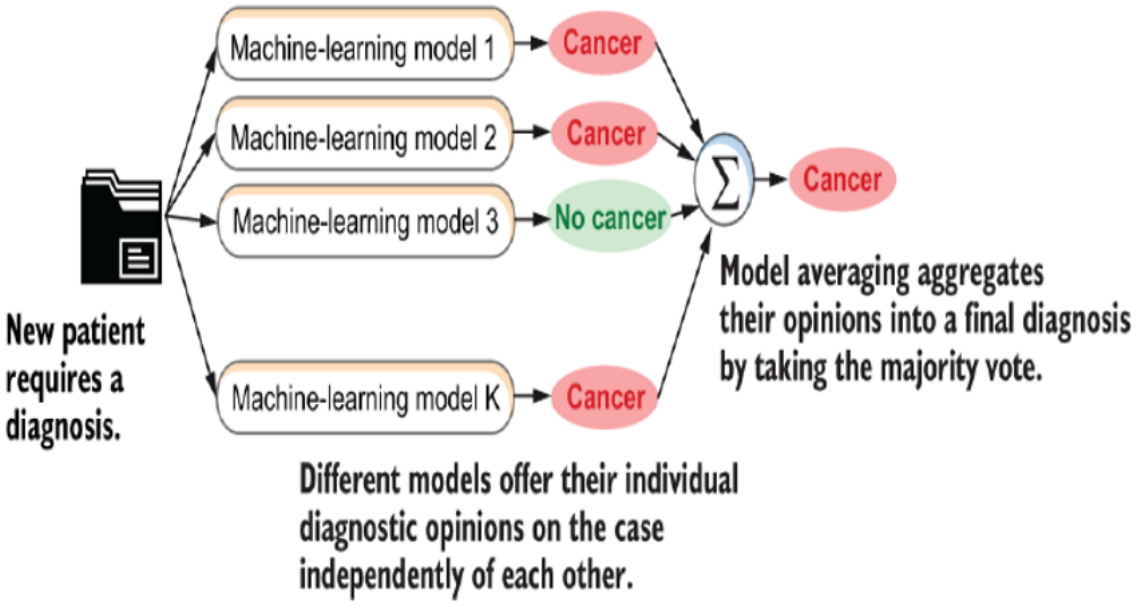
- You pick a subset of features at random with replacement (the same feature can appear multiple times)
- I make a decision tree based on this subset
- Repeat…

Instead of only doing a feature selection (by columns), we can also do a sample selection (by rows). So, we are obtaining both subsets of features and individuals.

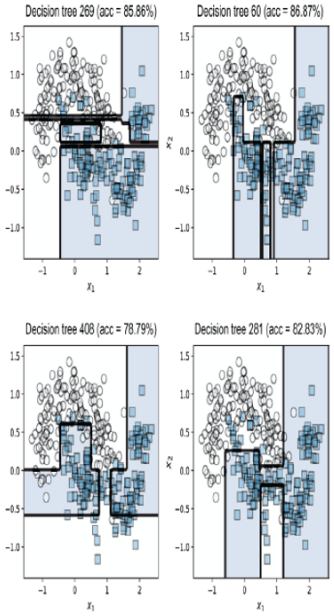
So, from the original dataset you will generate several datasets that will partially overlap (from both features and samples) but they will be unique. All the overfitting I will do from a decision tree will only affect that particular dataset.

- If I do a pullin from all the different datasets I will get a prediction that is going to be much better than any of the predictions made from any of the created datasets.

Note that when doing the extraction of individuals, you can use the individuals that have not been used to perform a validation test. **This is something related to something we will see later.**



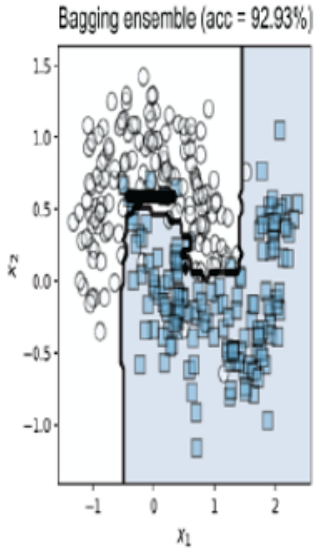
Bagging: Remove individuals/samples.
Patches: Remove both features and samples.



How does it work?
You generate multiple decision trees that will divide the space in a different way, because the features that they are using can be potentially different.

These trees will generate linear patterns as we mentioned before.

When you pull all of them it allows us to create very complex non linear patterns that make a better classification.



You need to define some hyperparameters:

- Number of decision trees. It really does not matter if we create a very large number of decision trees, because the error rate will reach a plateau for a given number of decision trees.
- How many features and samples you are going to extract.

Algorithm:

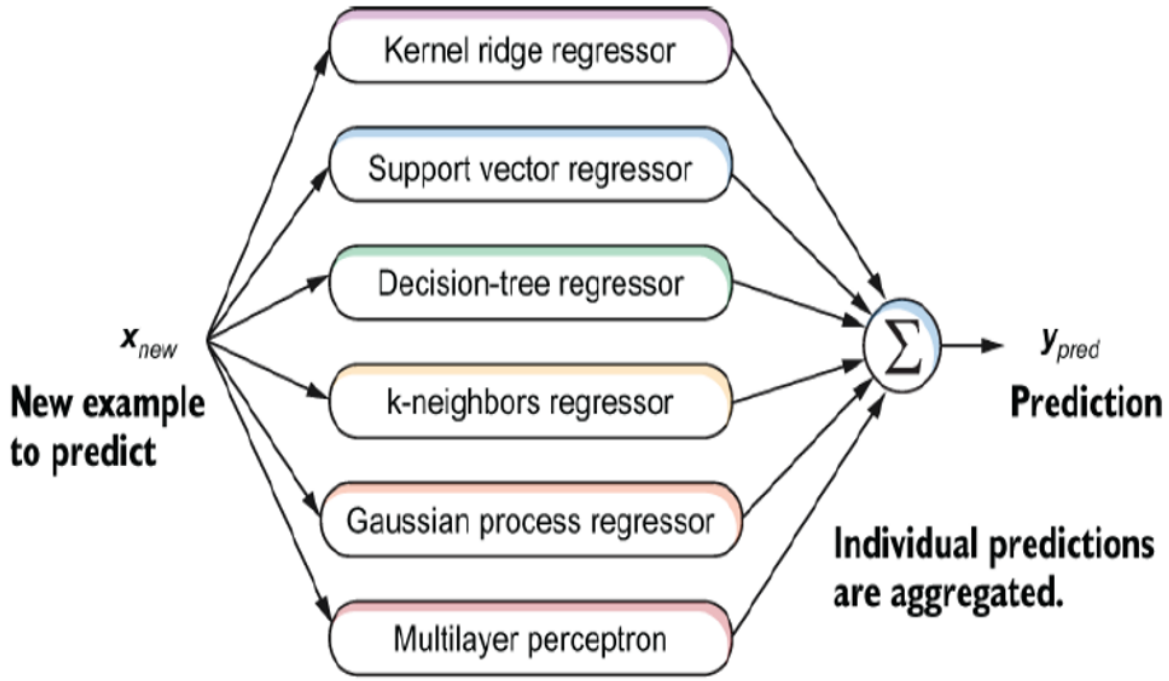
- Bootstrap
- Generate the decision trees with the selected features and samples.
- Make the ensemble of trees
- Make the new prediction

Since we are doing subselections of features, we can see which decision trees make the best predictions and therefore which are the most informative features.

- So, we could do a post feature selection

The power of ensambles

You can have a family of very different models, which make a different prediction and then you collapse the predictions. Make a new prediction based on the predictions of the different models.



Parallel homogeneous ensembles

We are generating at the same time different models that belong to the same family and we collapse the predictions to generate a final prediction.

- Random forests for example.

Parallel heterogeneous ensembles

In this case we have models that belong to different families.

How do we collapse the outputs of each model? Maybe we should trust one output more than another…

We first make a final prediction considering that all outputs have the same weight. So, we are making a prediction that is an average of all the models.

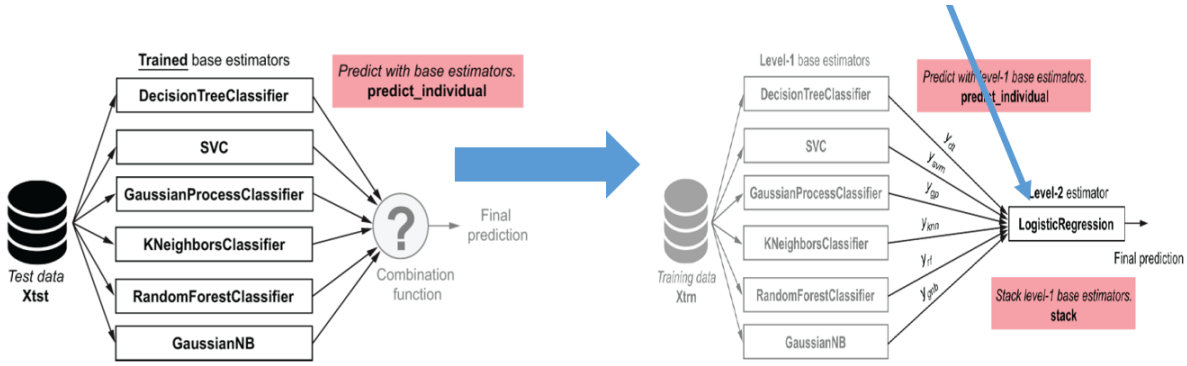
If I have done patches, I can use the samples that are not used to train the model to make a validation and assess how good that model is. So, I can give a weight to each of the models based on how well they perform.

- We could divide the accuracy of each individual model by the total accuracy across all the models. This will give a weight that goes between 0 and 1.

Instead of using the accuray, I could use the entropy of the model. So, there are other alternatives.

Stacking Ensembles

Instead of doing an average, I can use the output of the different models as input to create a new model.



Again you would use the training dataset to train the models, the validation datasets to assess the performance of each model and with this information create the different weights.

These parallel approaches are really good from the computational view because you can parallelize the work (run the models at the same time).

Sequential adaptive boosting ensembles

They can not be parallelized but they are more powerful.

- AdaBoost

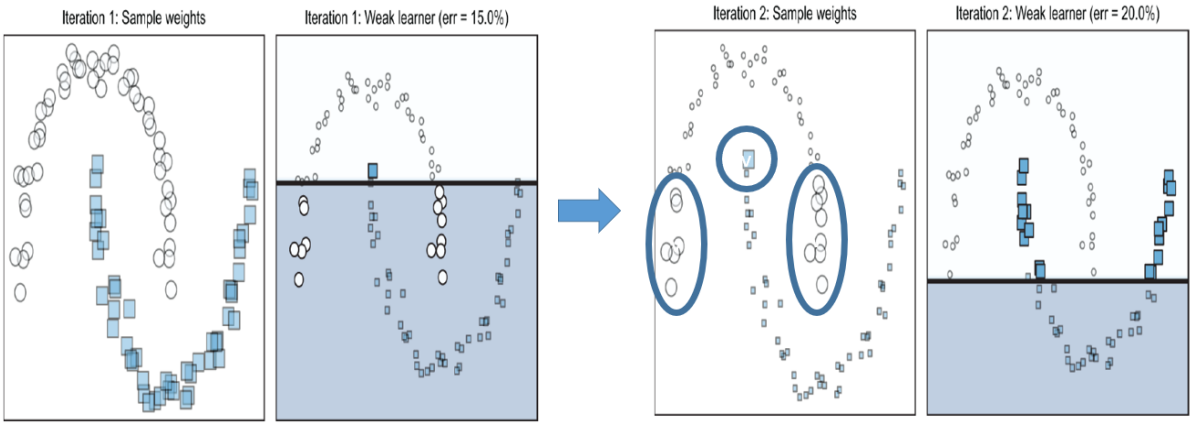
You build up a model that is really bad but still better than a random model (**Weak learner**). Based on the output of this model, you weight somehow the points:

- The ones that are worst performing (we did not predict them correctly), we give them a bigger weight.

With this new training dataset you create a new model (same family or not). With the worst performing points of the previous model, we create a new model…

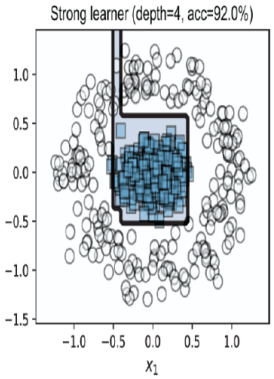
At the end, you obtain a series of models that are really bad for the whole dataset but each of them is good at predicting a specific part of the dataset, which are the points that were not correctly classified by the previous models.

Example:



The points that are missclassified, will have a higher weight than the ones that are correctly classified. So, we are forcing the next weak learner to learn the points that were misclassified in the previous step.

If you do this multiple times (each step will have a higher ER), we will obtain an overall ensemble (Strong learner) with a lower error rate. This ensemble is a combination of the weak learners



AdaBoost: There are many hyperparameters that will modify the results.

- Number of estimators or weak learners
- Algorithm used for the weak learners (decision trees, maybe)
- Learning rate → Related to the weight of each weak learner. A lower learning rate may lead to a more robust model, but it requires a higher number of weak learners to achieve the same level of accuracy.
- Algorithm used for the boosting process (**combine** the predictions of multiple weak learners to create a strong, robust model).
    - **Each model is better at a particular subset!**

Algorithm of AdaBoost:

1. Train the weak learner using the weighted training examples. At the beginning all training instances have the same weight.
2. Compute the training error of the weak learner
3. Compute the weight of the weak learner
4. Update the weights of the training examples
    a. Increase weight of misclassified examples
5. Repeat

Other ensembles. Sequential Gradient Boosting Ensembles

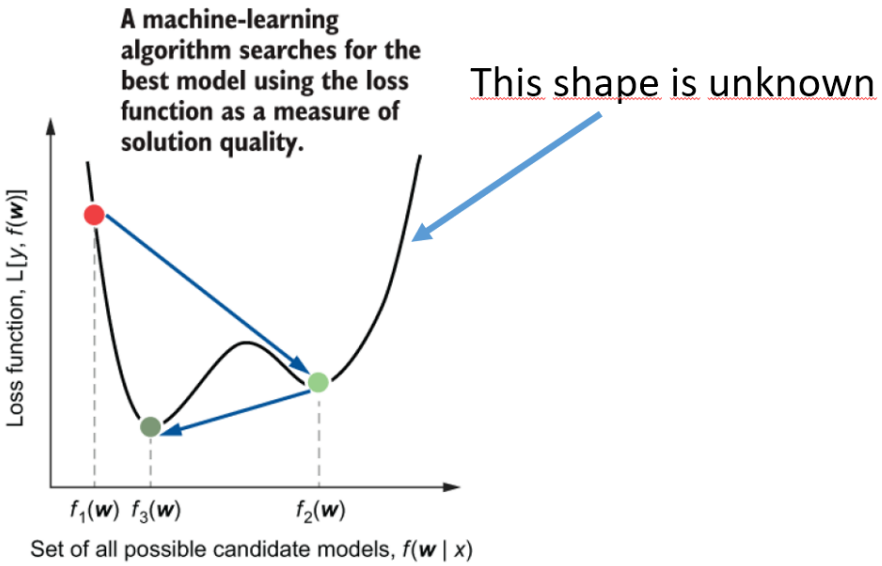
So far, we said that we give weights to the worst performing points in order to make a new model. The gradient boosting ensembles focus on something a little bit different, it focuses on the **Error**.

Instead of giving weights, we are going to make a new model that tries to minimize the error.

- We are looking for a gradient: The direction where the model should move in order to minimize the error between the prediction and the real label.

Minimizing the error means to minimize an unknown function.

- If we knew the shape of the function, it would be straight forward.



In order to find this minimum error, there are different approaches such as gradient descent.

- The only problem is that if there are a lot of valleys and hills, it is easy to get stuck depending on the starting point.

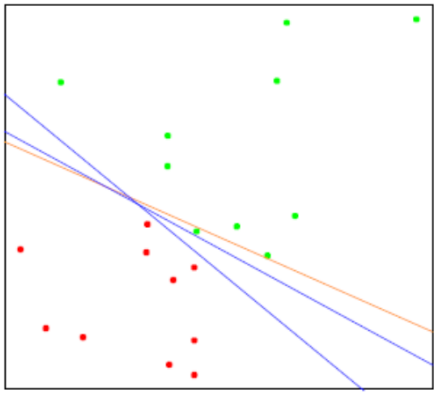
When you make a gradient, you compute which is the direction that should follow that point in order to minimize that gradient.

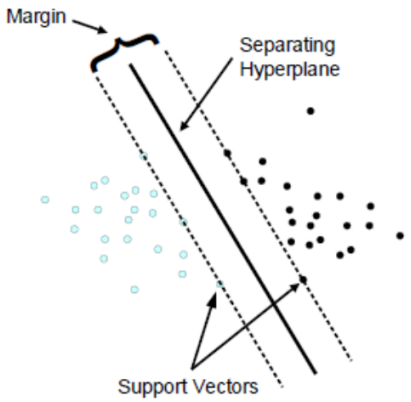
- So, we will build a new model based on the direction of the wrong points. So, instead of minimizing the error, we are minimizing the gradient of the error.

<u>Gradient descent has a random initialization and for this reason we obtain different results.</u>

Supporting Vector Machine (SVM)

If we have the following plot that contains 2 features, which line do we choose to classify the samples?





We would like the points that are close to the boundaries are as far as possible.

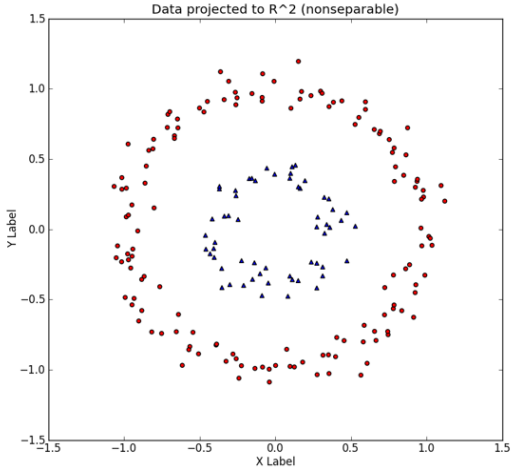
- The distance between any point to the hyperplane is maximized.

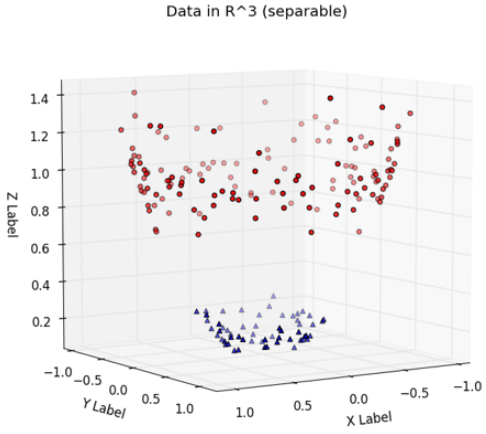
This will increase the robustness of the model.

In the SVM field, the closest points to the boundaries are called “support vectors”.

There will be situations where no matter what you do, they will be misclassified.

- So, if the data is not linearly separable, there will be no convergence

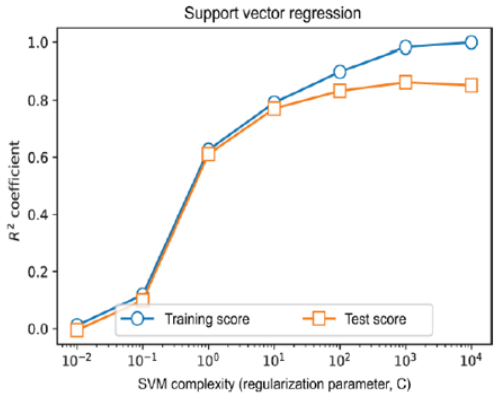




In these situations, the SVM creates a new variable with a kernel so that we have 3D.

- Now the data is predicted in this new dimension so that now we can draw a hyperplane and we can separate the 2 categories.
    - This third dimension did not exist, we have just created it with a kernel function.
    - The kernel is a hyperparameter. Depending on the kernel function you use, you will obtain one result or another.

In SVM, the complexity of the algorithm is related to the C hyperparameter. If you have a really large value, the model will be very complex and there will be a point where the performance of the model in training data will increase as we increase C, but the performance of the model in validation data will decrease as usual.



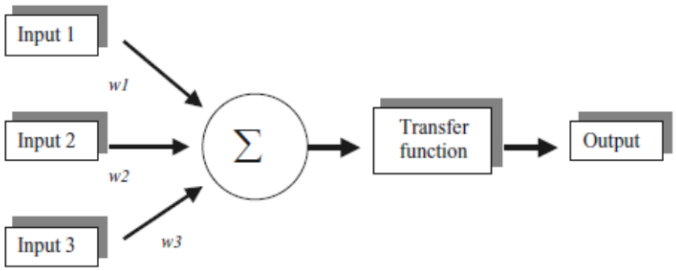
Package “e1071”:

- svm function

Artificial Neural Networks

Artificial intelligence uses the capabilities of computers and machines to mimic the problem-solving and decision-making capabilities of the human mind.

A neuron receives inputs with weights, collapses those inputs and decides to be activated or not.



The sensors are the eyes, ears, nose… the brain is the neural network and the actuators are the other parts of the body that react to the output of the brain.

One of the applications of ANN is reinforcement learning, where the computers learn to learn.

- The computer plays with other computers.
- Without knowing the rules, by playing it will understand the rules

Activation functions

The inputs are multiplied by a weight and then they are collapsed together. This will then go through an activation function that will decide whether to activate or not the neuron.

The most important ones are:

- Identity, logistic and ReLU (Rectified Linear Unit)

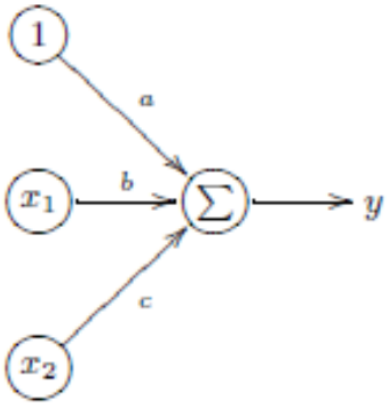
All activation functions should be derivable.

Linear activation function

You multiply the inputs by a given weight, you add them and you return the output.

So, the output will be:

$$y = a + bx_1 + cx_2$$

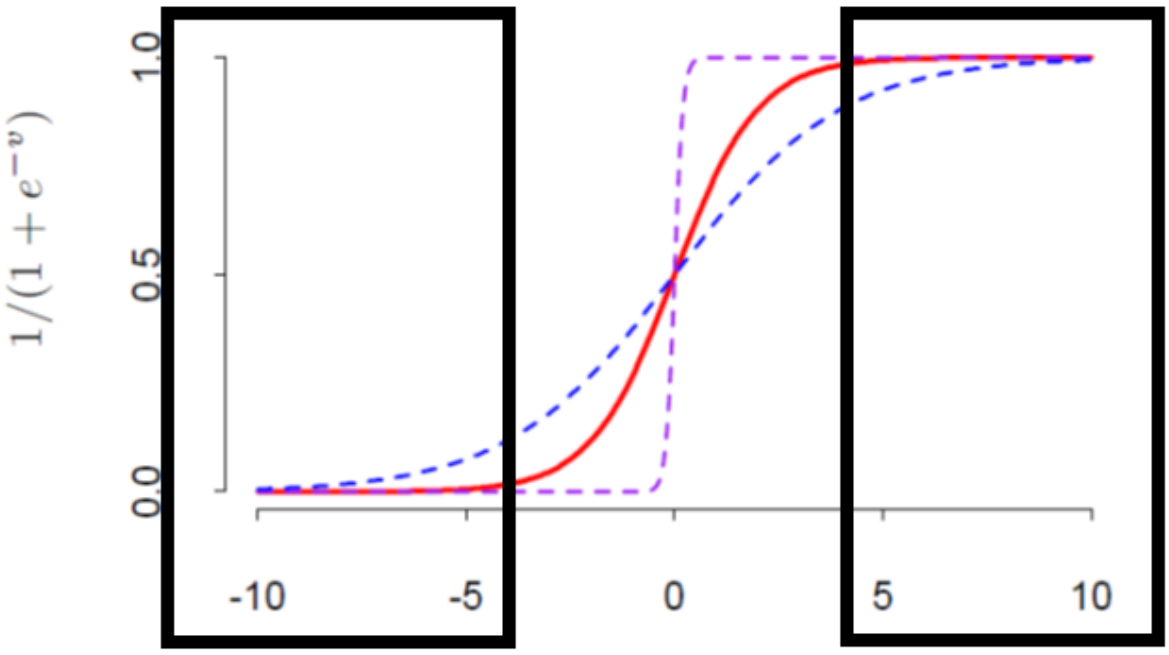


So, a linear regression is a type of ANN.

Logistic activation function
A logistic regression is another type of neural network.

- Here we use the sigmoid activation function

We can change the slope of the sigmoid function and we will obtain different results.



If the input value (after the collapse) of the activation function is below 0, the output will be 0 and if it is above 0, the output will be 1.

- So, the magnitude of the output that I am computing from the inputs will be transformed into 0 or 1.
    - Very sharp sigmoids produce a very low learning rate. Because it does not matter the magnitude of the input, since if it is bigger than 0 it will be transformed into 1. It does not matter if the input is 1, 2 or 1000.
    - So, tuning the parameters is very important

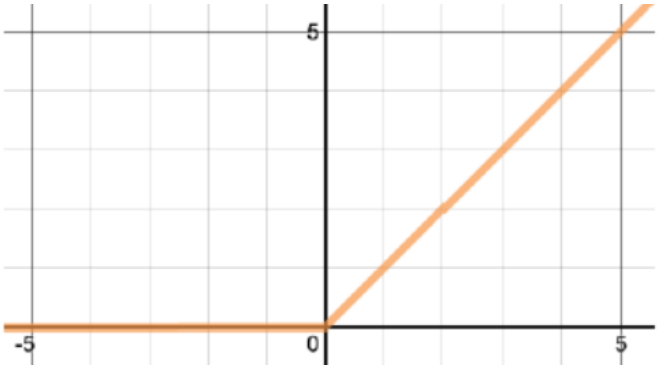
The purpose of the activation function is to create a non-linear transformation of the input.

So, ANNs are a generalization of linear and logistic regression.

ReLU
Negative values are transformed into 0 and positive values are returned in a linear way.

I want to discriminate:

- If it is below 0, I will not care at all and I will return 0. Thus, when I am sent to another neuron I will be multiplied by another weight and the result will be 0. So, this information provided is irrelevant.
- If it is above 0, I will care.



XOR problem

A single perceptron is not able to solve the XOR problem and for this reason people thought that ANN were doomed.

XOR is a logical operation that takes two binary inputs (0 or 1) and produces a binary output according to the following rule:

- 0 XOR 0 = 0
- 0 XOR 1 = 1
- 1 XOR 0 = 1
- 1 XOR 1 = 0

The perceptron receives inputs and they will be multiplied by a weight that will increase or reduce their importance. We will always have an extra input that will have a value of 1 called “bias unit”.

The perceptron will make the sum of the inputs multiplied by their weight. Finally we will transform this result using an activator function.

- In this case, we will use a Heaviside activator function that returns 0 if the input value is smaller than 0 and 1 if the input value is higher.

A perceptron can classify categories.

To do this, it just creates a hyperplane that linearly separates the space by categories.

The **NOT** operation:

- If the input is 0, it returns 1
- If the input is 1, it returns 0

To divide this space, the Heaviside function is perfect as an activation function.

Our objective is to find a combination of weights that creates a straight line that separates and classifies the categories.

- In this case, the weights could be -1 and the bias 0.5, for example.

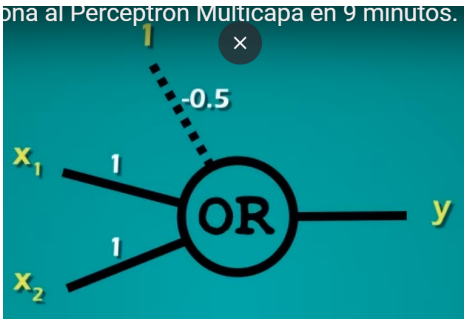
Thus, the perceptron will do the following sum: ´-x + 0.5

If the input is 0 → 0 + 0.5 > 0 → It returns 1
If the input is 1 → -1 + 0.5 < 0 → It returns 0

The **OR** operation: In this case it receives 2 inputs at the same time.

- The weights are going to be 1 and the bias will be -0.5.
- We use the Heaviside function

x1 + x2 - 0.5

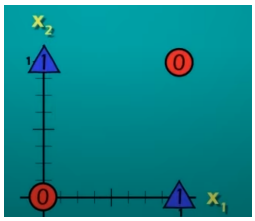


x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

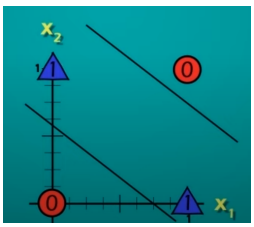
The perceptron will need to learn which are the correct weights. To do this, it will need to change the weights when the predictions are incorrect using a learning rate coefficient that is multiplied by the weights.

XOR operation:

- 0 XOR 0 = 0
- 0 XOR 1 = 1
- 1 XOR 0 = 1
- 1 XOR 1 = 0



In this case we can not use a single perceptron because we can not draw a hyperplane that separates the 2 categories.



So, if we use 2 perceptrons that separate the plane like this, we can then use a third perceptron to correctly classify the categories.

The first perceptron classifies correctly when both inputs are 1

- Returns 0 if both inputs are 1
- Returns 1 in any other case. Leading to a wrong classification when both inputs are 0

The second perceptron classifies correctly when both inputs are 0

- Returns 0 if both inputs are 0
- Returns 1 in any other case. Leading to a wrong classification when both inputs are 1

The third perceptron:

- Returns a 0 if the inputs are 0 and 1
- Returns a 1 if both inputs are 1

```
x1 <- c(0, 0, 1, 1)
x2 <- c(0, 1, 0, 1)
logic <- data.frame(x1, x2)
logic$XOR <- as.numeric(xor(x1, x2))
net.xor <- neuralnet(XOR ~ x1 + x2, logic, hidden = 2, rep = 5);
prediction(net.xor);
```

Bias unit

It's a neuron always has the value of 1.

- Its role is to provide an additional parameter that allows the neural network to make adjustments and better fit the data.

In the case of a linear model, it corresponds to the intercept. So, it allows us to scale where the activation function will start working.

Learning the weights

We want to know the topology (how many neurons and how they are connected) and the weights of the ANN.

In reality, you have a topology that works for a similar problem (transfer learning) and you apply this topology to your own problem modifying the weights using gradient descent.

- We want to minimize the error

Since the activator functions can be derivable, you can apply gradient descent. Otherwise you can not do it.

Updating the weights can be computationally expensive. One of the solutions is to use **batches** → Divide the train dataset into sets of samples.

An **epoch** refers to one complete pass through the entire training dataset during the training phase. During each epoch, the neural network processes every training example once and updates its weights based on the errors or differences between its predictions and the actual target values.

1. Initialize Weights: At the beginning of training, the weights of the neural network are usually initialized randomly.
2. Forward Pass: For each training example in the dataset, the input is fed forward through the network, layer by layer, to produce an output.
3. Compute Loss: The output of the neural network is compared to the actual target values, and a loss (or error) is calculated. The loss represents how far off the predictions are from the true values.
4. Backward Pass (Backpropagation): The calculated loss is then used to **update the weights** of the network in the opposite direction (backward pass) to minimize the loss.
5. Repeat for All Examples: Steps 2-4 are repeated for each training example in the dataset → **Avoids overfitting**. I am updating the weights in each batch, so I can not learn the specificities of a single training set.
6. One Epoch Completed: After all examples in the training dataset have been used to update the weights once, one epoch is considered complete.
7. Repeat for Multiple Epochs

Hyperparameters of ANN:

- The **topology**, how many layers, neurons, the learning algorithm (backpropagation, for example), activation function in each layer, the learning rate (gradient descent), the number of patches, the number of epochs…
- Also the preprocessing is very important!

Vanishing gradient: It is important that all the layers extract the same amount of information. If one layer extracts all the information, then the other layers will extract noise and the result will be bad.

- For this reason, we need to optimize the number of layers

In practice, if we start decomposing the layers, we will see that the activation of a neuron represents a pattern. A neuron will be activated if it recognizes a pattern and this is what will be reported to the next layer.

We can think of a neuron as an ensemble. Each previous neuron defines a model and it is adding all the information from the different models.

Loss function: We want to minimize the error

- MSE, Classification Error
- Cross entropy (as close to 0 as possible)

$$-\sum_i y_i log(\hat{y}_i)$$

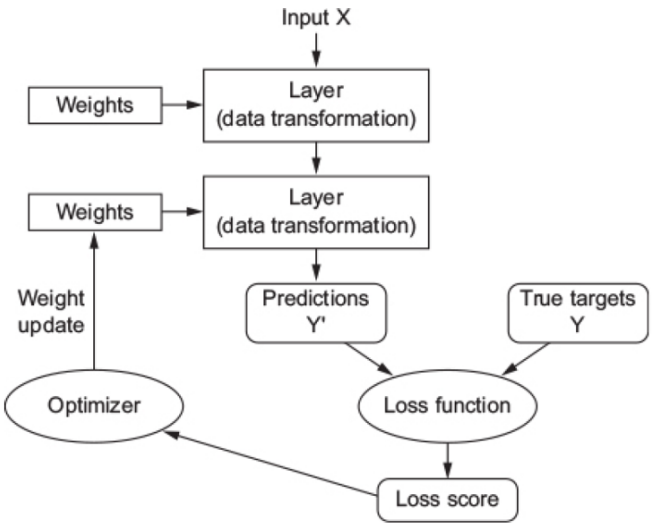
Table 1.1 Target values and computed probabilities of a neural network

	Class A	Class B	Class C
Target	0.000	1.000	0.000
Computed probability	0.128	0.719	0.153

```
H = -(0.0 * log2(0.128) + 1.0 * log2(0.719) + 0.0 * log2(0.153))
```

We will minimize this error by modifying the weights using gradient descent.

- ReLU can not be derived at 0, but this is unlikely.



In most models, having more and more data does not increase their performance. But in ANN, the more data the better.

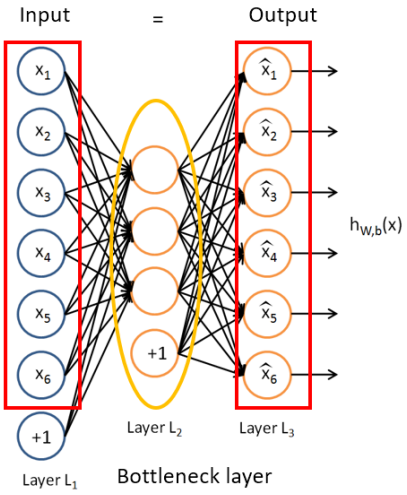
Dropouts: Removing some neurons (weight 0) during training, which helps prevent the network from relying too much on specific neurons and learning noise in the training data. It's a regularization technique used to prevent overfitting and improve the generalization ability of the model.

Transfer Learning: Pick a model that is already trained (we know the weights) and use it for my problem.

- Use a model used to predict types of trousers to predict types of proteins, for example.

You just need to remove the last layers that allow you to predict the type of trouser and place your layers. So, you only have to modify the weights of the last layers.

Autoencoders



You have an input and output layer that have the same number of neurons and they will produce the same value.

- Input = Output

It's kind of stupid that we are predicting the input, since we already know it.

The important thing are the bottleneck layers, where we are reducing the dimensionality (lowering the number of neurons) and then forcing that this reduced number of neurons reconstruct the initial input.

- We are doing a feature extraction since we are compressing the information

We can combine autoencoders with other types of architectures such as Convolutional NN.

- You have your images, you compress them into something more abstract and then you decompress them into the initial images.

You can also train the CNN to reduce the noise.

- The input is the image with noise and the output is the image without noise.

CNN

You use images to make a prediction.

If I want to identify a garlic in a photo, I will get a picture of a garlic and compare each pixel of both photos. If the pixels are the same, then I have found the garlic.

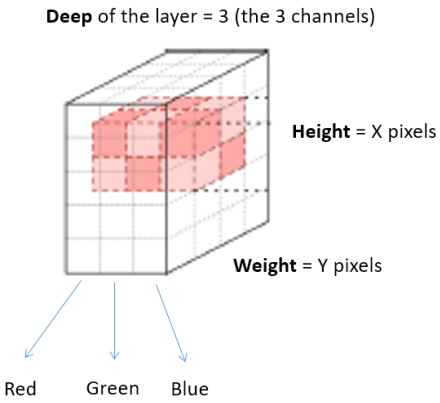
- In reality, CNN does something similar. They assign a positive value to white pixels and a negative value to dark pixels. If both pixels have the same color, then I will obtain a large positive value (I multiply them). Thus, I need to maximize a function.

But this is not that simple, because garlic can have different shapes.

Now we do not want to find a picture of a garlic, we want to find the abstraction of what a garlic is.

In CNN, the neurons are the pixels of an image.

- From one layer to the next one, we are going to create new images using filters or kernels.
  The filters capture the features that a garlic should have.



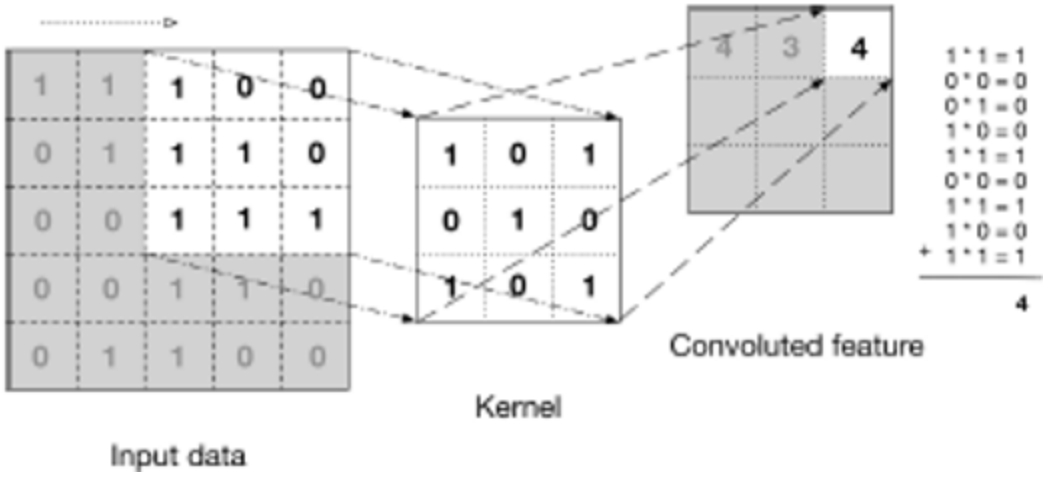
At the beginning we have 1 image that can be decomposed in 3 images:

- Red, green and blue

If I merge all 3 images, I don't have a matrix, I have a cube.

- 3D is a tensor
- More D is a tesseract

I can transform each of the images into the pixels with their values. Then multiplying the inputs by the kernels, I will create a new image that can be feed to another filter.

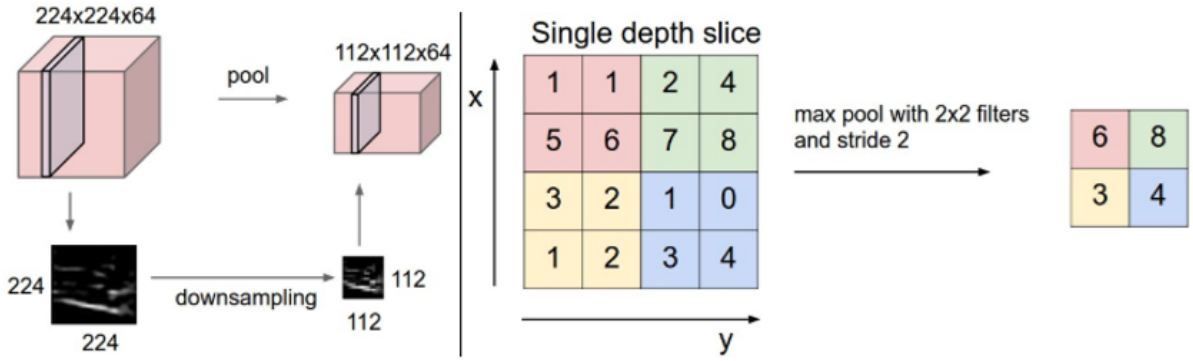


In CNN, the ReLU activation function is very useful.

- The ReLU sharpens the contours.
- It removes the features (pixels) that are not interesting and sharpening the pixels that are important.

Pooling layer: It collapses the values of the neighbor pixels so that we are compressing the images. We can collapse them using different strategies:

- Average, sum, min, max…



Hyperparameters of CNN:

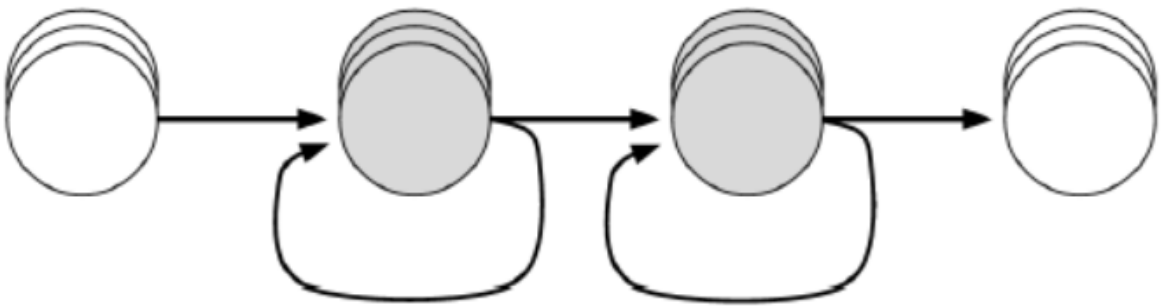
- Number of channels (number of dimensions), filter size, stride (size of sliding window), zero-padding (adding 0 in the borders of the image).

Recurrent Neural Networks (RNN)

Used when you have a time series → Data that depends on the previous data

When we talk, we can make a prediction based on what we have said before.

- We have loops that can affect the same neuron or neurons that are in the same layer
- So, in each iteration, the input comes from 2 directions. From the original input + loop. **So, I keep remembering the previous states.**



This type of ANN will introduce the concept of transformers. Very useful in text mining to predict the next word!

Learning to learn

Unsupervised: I already have a model defined about how the data must be generated. So, I have an algorithm that explains to me how to cluster the individuals, for example.
We apply this model to a new dataset.

- PCA

Supervised: I have the data with their labels and therefore I want to use an algorithm to generate a statistical model that will contain the rules I should apply in order to predict those labels. So, in this case I want to know which are the rules used to classify the data.

- KNN, linear regression, random forest, decision trees…

Autoencoders are both supervised and unsupervised, because you have a training dataset where your input is your output (supervised) but you are doing a feature extraction in the inner layers (unsupervised).

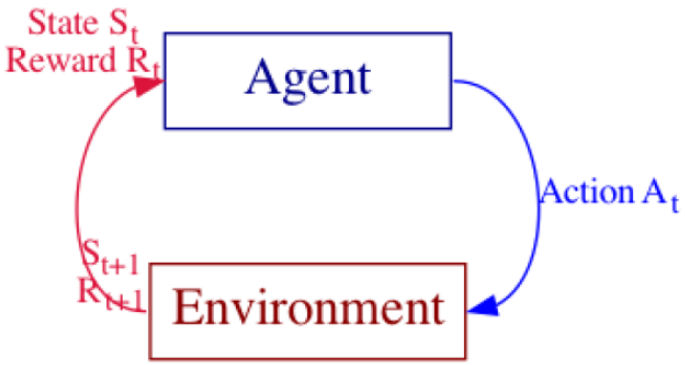
There is another type of learning called **reinforcement learning** that tries to learn the rules from the data.

- Example of a dog with food and a bell.
- Learning to play a video game

The thing is that the algorithm will perform an action based on a reward.
So, in a particular state, which action should I perform in order to get the maximum reward over time.

- In each state, I will have a number of possible actions I could take and the set of actions I take over time will be defined by my **policy** (set of rules to achieve my goal). I want to have a policy that maximizes my benefit over time, for example.
    - This set of rules (policy) can change over time. The computer needs to learn which is the policy

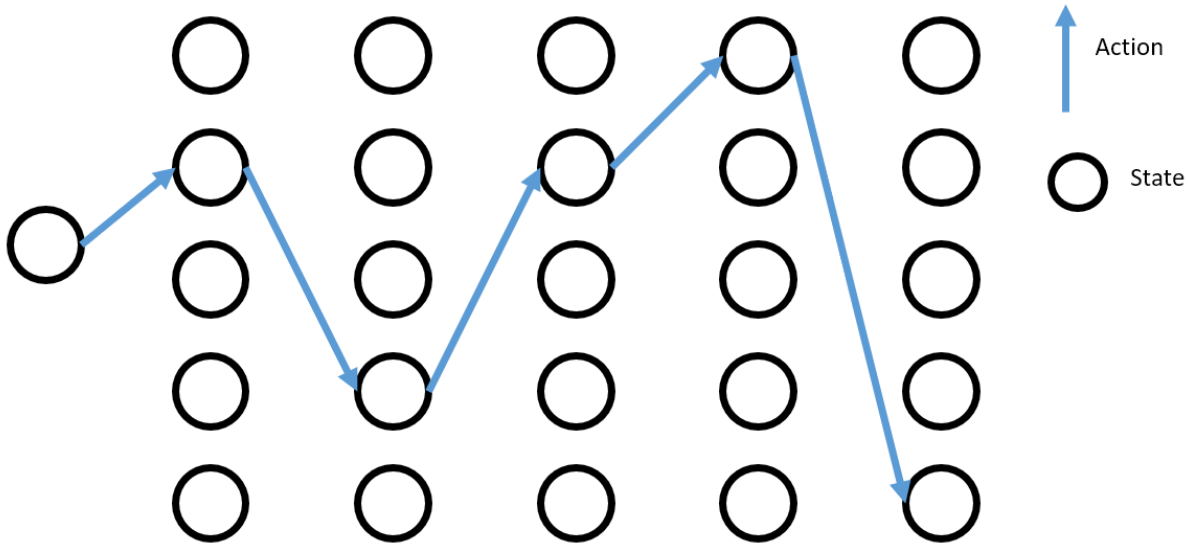
The **agent** is the "thing" that needs to decide which action to take in the next step.
Based on this action, it interacts with the environment, which is changing constantly.
Based on the interaction with the environment, I will obtain a reward (can be positive or negative). If the reward is bad, I will need to update the weights.



So, the agent's goal at any point in time is to maximize the Expected Sum of all future Rewards by controlling (at each time step) the Action as a function of the observed State (at that time step).

Imagine I start at a given state and then I can choose a set of actions that will lead me to a different state.

- Markov Decision Process → Based on my state, what should I do?



Text Mining

Here, the input data is a text.
The ANN is able to extract words from the sentences.

Data preprocessing is very important to remove non-informative data.

- Deal with missing data → Remove the samples. But this is an error because maybe that data is missing for a reason. It's a complex region to sequence, maybe.
  If the missing data follows a pattern, we should not remove those samples, since they are informative.
  - Example of someone making a misassignment to color blue (because he hates that color). This missing data is biased in a consistent way, there is a pattern.
- Deal with outliers. Maybe it's an error or maybe it's very valuable information.

Maybe there are a lot of words that are repetitive or words that are just connectors. We will need to focus on a subset of all the words that are available.

- The words that are informative, that have a semantic meaning, will be called "tokens"

Building a **dictionary or scopus of tokens** is one of the most computer challenging aspects.

- Each token is separated by a space? Las Vegas
- What do commas, colons mean?
- Acronyms?
- Capital letters?

Sparse matrix (because it contains a lot of 0)

If you have multiple texts preprocessed, you can use different tokens to know what those texts are talking about. Then, you can compute the distance between the different texts:

- You expect that the texts speak about the same thing if they have a lot of common words.

I can collapse tokens if they are synonyms (reduction of dimensionality).

So, the first thing I could do is to create a distance matrix between those pairs of texts.

- We could use the euclidean distance, but it has a problem. If the texts have different length, then the result is biased.
- Thus, we should use different distances such as the cosinus

Example:

$$D = \begin{pmatrix} & \text{lion} & \text{tiger} & \text{cheetah} & \text{jaguar} & \text{porsche} & \text{ferrari} \\ \text{Document-1} & 2 & 2 & 1 & 2 & 0 & 0 \\ \text{Document-2} & 2 & 3 & 3 & 3 & 0 & 0 \\ \text{Document-3} & 1 & 1 & 1 & 1 & 0 & 0 \\ \text{Document-4} & 2 & 2 & 2 & 3 & 1 & 1 \\ \text{Document-5} & 0 & 0 & 0 & 1 & 1 & 1 \\ \text{Document-6} & 0 & 0 & 0 & 2 & 1 & 2 \end{pmatrix}$$

The documents are talking about cats and cars.

- Jaguar has both meanings

We can decompose this matrix into different matrices.

Transformers and DL

Tokens are not only words but entities that have a semantic meaning.

- Each letter can be a token

DL works with numbers, not with letters. Thus, we need to transform each token into a number (**numericalization**).

You can also create a hot vector. It does not matter the order, because the distance between the tokens is always 2.

	Bumblebee	Megatron	Optimus Prime
0	1	0	0
1	0	0	1
2	0	1	0

But if we have a lot of tokens, this sparse matrix will become extremely large.

- So, we can limit the number of tokens in our scopus or dictionary.
- We will select the most informative tokens and the ones that are more frequent. We must remove all the words that can appear in any text because they are non informative (connectors, for example).
- The words that are not used will belong to a category that will be encoded by all 0, for example.

Texts can be longer or shorter and as mentioned, this is bad. So, in text mining they define a length of the sentences:

- If the sentence is shorter, you full the remaining tokens as 0
- If it is too long, then you split the sentence in different parts.

Remember RNN, where a neuron is self connected. Thus, it has in memory the previous outputs.

- It can happen that as the sentence is longer and longer, we lose memory about the previous steps. Meaning that we lose valuable information

We can avoid this using “attention”.

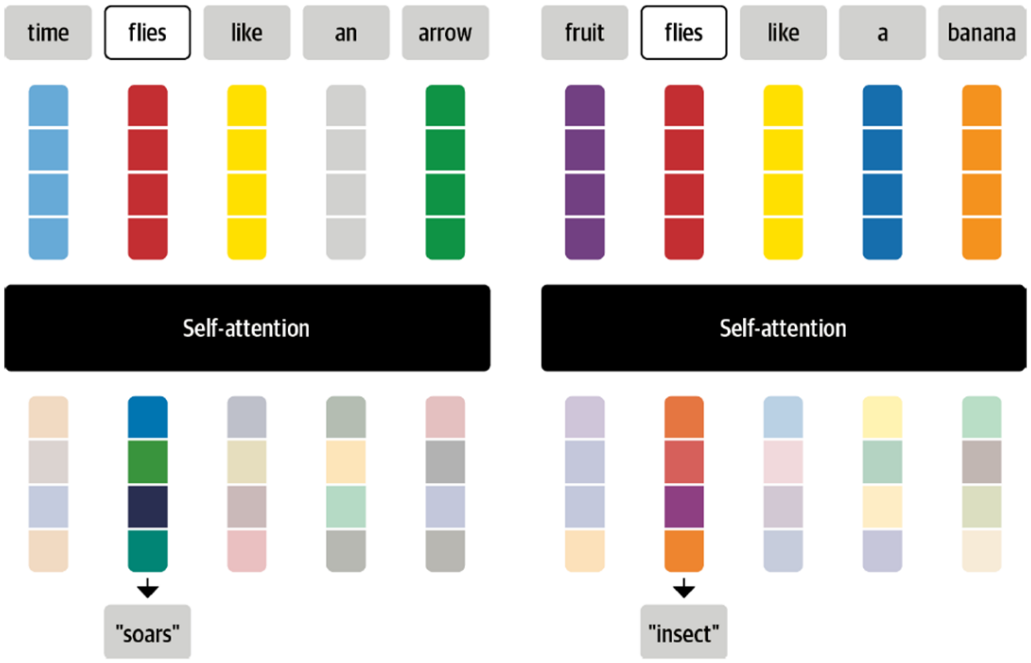
- Provide additional information indicating that attention must NOT focus on a certain section. For example in short sentences that contain a section full of 0.

So, there is an additional layer that identifies which section of each stage is more important. Thus, instead of sending the previous state as input in the RNN, now we send the most important sections as input.

The important thing is that in the transformers architecture we will have an encoder, a decoder and some layers in between of self-attention.

An attention mechanism can be considered as a memory with keys and values and a layer which, when someone queries it, generates an output from values whose keys map the input.

- **We weigh each word in the context of other words!!!**
- **The attention layer computes attention scores for each element in the input sequence relative to all other elements.**
- **This allows the model to weigh the importance of different elements dynamically, considering their relationships.**



Encoder part: convert an input sequence of text into a rich numerical representation.
Decoder part: To be used for iteratively predicting the most probable next word from a text.

Your task is to predict the next word based on the previous words (decoder inputs). Your neighbor (the encoder) has the full text. Unfortunately, they're a foreign exchange student and the text is in their mother tongue. You draw a little cartoon illustrating the text you already have (the query) and give it to your neighbor. They try to figure out which passage matches that description (the key), draw a cartoon describing the word following that passage (the value), and pass that back to you.

Application:

- Machine translation
- Summarization tasks