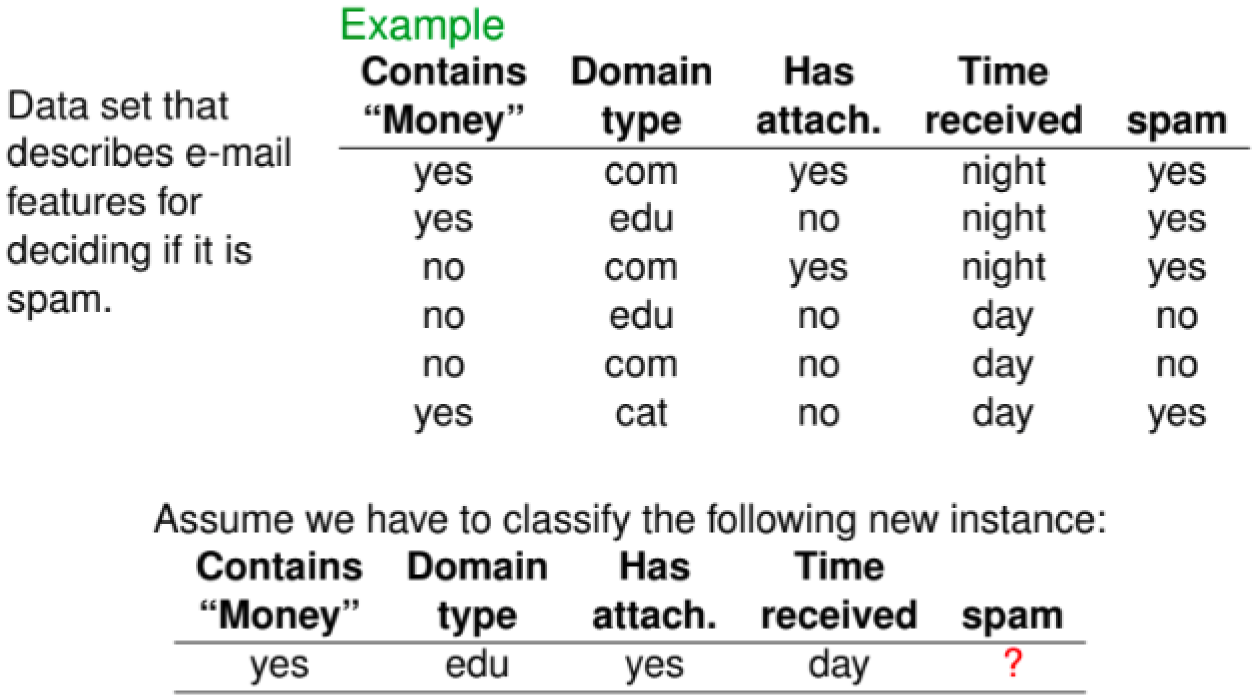
Classification

Classification is defined as such:

> Given n_c different classes, a classifier algorithm builds a model that **predicts** for every unlabelled instance *i* the class *C* to which it belongs, with a certain degree of accuracy.

In other words, we build a classification model in order to predict a *label* given an *instance*, where *label* is how we classify a certain record (yes/no are an example of binomial labels, and a record can be classified according to whether it belongs or not to a certain category) and *instance* is a table record, containing information regarding the element we are analyzing.

An instance is defined by **features**, which are what the classifier uses to decide how to classify the instance.



Example

Data set that describes e-mail features for deciding if it is spam.

Contains "Money"	Domain type	Has attach.	Time received	spam
yes	com	yes	night	yes
yes	edu	no	night	yes
no	com	yes	night	yes
no	edu	no	day	no
no	com	no	day	no
yes	cat	no	day	yes

Assume we have to classify the following new instance:

Contains "Money"	Domain type	Has attach.	Time received	spam
yes	edu	yes	day	?

Many different kinds of classifier exist. Most of them can be found in the python library **scikit-learn**, which can be imported by simply running

```
from sklearn import
```

Most scikit-learn classifiers implement two methods:

- **fit(train_data, train_label)** - method for the training phase, it accepts as argument a training dataset on which it applies an algorithm defined in the *fit* method itself.
- **predict(test_data)** - method for the prediction phase, applies the previous algorithm to the test_data in order to predict its labels.

Majority Class Classifier

The **majority class classifier** is the most basic classifier:

- in the **training phase**, it computes the majority class of the dataset - the label that appears the most in the training dataset
- in the **prediction phase**, it ouputs the majority class of the dataset - outputs an array containing as many values as the test datasets, and all of them have value equal to the label previously found in the training phase.

This is implemented in the *sickit-learn* library via the **DummyClassifier**, with **strategy='most_frequent'** option.

```
from sklearn.dummy import DummyClassifier
mjclass = DummyClassifier(strategy="most_frequent")
```

k-Nearest Neighbours [k-NN Classifier]

In k-NN classifier, the *k* elements closest to the instance analysed are used to predict, by majority class, the label of the instance.

- in the **training phase**, all instances are stored in memory
  - it is not appropriate for big data, due to huge requirements in memory space
- in the **prediction phase**, a majority classifier is applied on the k-nearest instances
  - the choice of *k* is very important, and is best done by first inspecting the data
    - large K value reduces overall *noise*
    - generally K is between 3-10
  - a definition of k-nearest is in need, as well as an algorithm to compute said distance (examples are *Euclidean d.*, *Manhattan d.*, *Minkowski d.*, *Hamming d.* (only for categorical variables))
  - distance computation can be computationally intensive

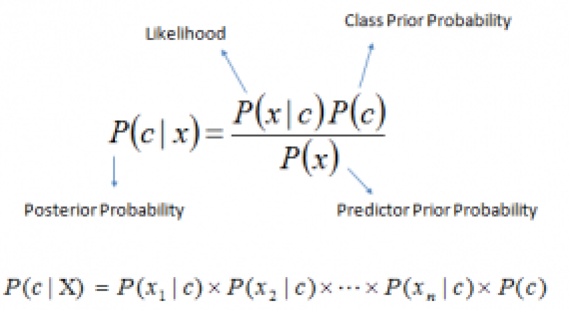
Implementation

The k-NN classifier is implemented in the *scikit-learn* library in **sklearn.neighbors**, which contain **KNeighborsClassifier**:

```
from sklearn.neighbors import KNeighborsClassifier
knnClass = KNeighborsClassifier(n_neighbors=3, metric='euclidean')
```

Naïve Bayes

Naïve Bayes classifiers are a family of simple probabilistic classifiers, based on applying Bayes' theorem, but with strong independence assumptions between the features - the features should not influence each other in any way (low correlation among features).



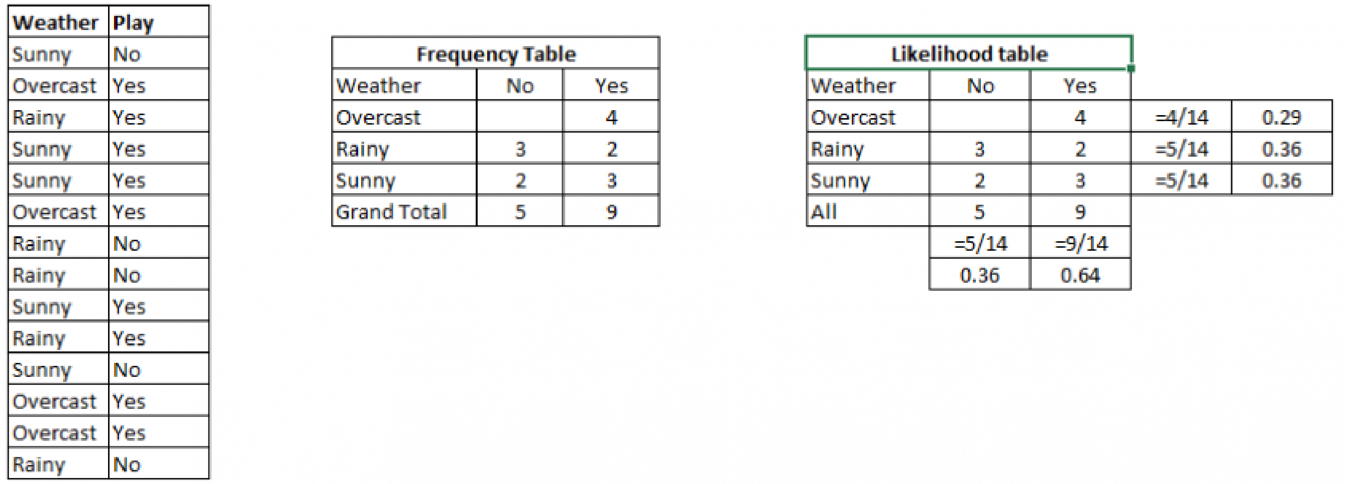
$$P(c \mid x) = \frac{P(x \mid c) P(c)}{P(x)}$$

$$P(c \mid X) = P(x_1 \mid c) \times P(x_2 \mid c) \times \cdots \times P(x_n \mid c) \times P(c)$$

- P(c|x) is the **posterior probability of class** (c, target) given predictor (x, attributes) - probability of the instance x to belong to class c given its x_n attributes.
- P(c) is the **prior probability of class**.
- P(x|c) is the **likelihood** which is the **probability of predictor given class** - probability of an instance of class x to have the attributes x_n like in instance x.
- P(x) is the **prior probability of predictor**.

What this classifier does is estimating the probability of observing attribute x_n and the prior probability *P(c)*.

Naïve Bayes algorithm: an example



Weather	Play
Sunny	No
Overcast	Yes
Rainy	Yes
Sunny	Yes
Sunny	Yes
Overcast	Yes
Rainy	No
Rainy	No
Sunny	Yes
Rainy	Yes
Sunny	No
Overcast	Yes
Overcast	Yes
Rainy	No

Frequency Table		
Weather	No	Yes
Overcast		4
Rainy	3	2
Sunny	2	3
Grand Total	5	9

Likelihood table				
Weather	No	Yes		
Overcast		4	=4/14	0.29
Rainy	3	2	=5/14	0.36
Sunny	2	3	=5/14	0.36
All	5	9		
	=5/14	=9/14		
	0.36	0.64		

With the training dataset in the first image, we need to classify (AKA predict) whether players will play or not based on weather condition. A few easy steps can be followed to do that:

1. convert the data from the default dataset (image1) to a frequency table (image2)
2. build the Likelihood table (image3) by finding the probabilities of each weather (sum rows/columns and divide by total cases) - these are the P(x|c) of the Naive Bayes equation
3. Calculate the posterior probability for each class with the Naive Bayes equation. The class with the highest posterior probability is the outcome of prediction.

So, if we want to predict whether players will paly if the weather is sunny, our equation would become:

> P(Yes|Sunny) = P(Sunny|Yes) * P(Yes)/P(Sunny) = 3/9 * 9/14 * 5/14 = 0.6

Multinomial Naïve Bayes

A particular case of Naïve Bayes classifier is the Multinomial Nave Bayes: it is used for document classification, because it computes the probability of a document *d* of being in class *c*. The document is considered as a bag-of-words: the estimation then becomes the probability of observing word *w* and the prior probability *P(c)*:

$$P(c|d) = \frac{P(c)\prod_{w\in d} P(w|c)^{n_{wd}}}{P(d)}$$

where $n_{w_d\text{</sub>}}$ is the number of times the word *w* appears in the document *d*.

Side Note: Laplacian Smoothing

Sometimes, in queries there could be some words which are not in the vocabulary. This would generate a probability of 0 for the whole query, but that's clearly not true.

In order to solve this problem, we introduce a correction in the probability computation, which is the **Laplacian correction**. Given a *k* factor, the correction is applied by:

- summing *k* to the numerator and **k x #classes** to the denominator for the prior probability
- summing *k* to the numerator and **#words in vocabulary** to the denominator for the likelihood.

Example 2 : Sentiment analysis

Let's suppose we have the following dataset:

ID	Text	Sentiment
T1	glad happy glad	+
T2	glad glad joyful	+
T3	glad pleasant	+
T4	miserable glad sad	-
T5	glad sad miserable pleasant glad	?

We want to compute the probabilities of T5 of belonging to one of the two classes (+ for positive sentiment, - for negative sentiment).

P(+|T5) = P(+)P(T5|+)/P(T5) = 3/4 P(T5|+)

P(-|T5) = P(-)P(T5|-)/P(T5) = 1/4 P(T5|-)

In the computation the value of P(T5) is ignored because it does not influence in any way the meaning of the probabilities. In order to compute P(T5|+) and P(T5|-), let's build the frequency table:

ID	glad	happy	joyful	pleasant	sad	miserable	class
T1	2	1	0	0	0	0	+
T2	2	0	1	0	0	0	+
T3	1	0	0	1	0	0	+
T4	1	0	0	0	1	1	-

ID	glad	happy	joyful	pleasant	sad	miserable	class
T5	2	0	0	1	1	1	?

P(+|T5) = 3/4 P(T5|+) = 3/4 P(glad|+)^2 P(sad|+) P(miserable|+) P(pleasant|+) = 3/4 (5/8)^2 1/8 0/8 1/8 = 0

P(-|T5) = 1/4 P(T5|-) = 1/4 P(glad|-)^2 P(sad|-) P(miserable|-) P(pleasant|-) = 1/4 (1/3)^2 1/3 1/3 0/3 = 0

Due to the presence of a factor equal to 0, it is possible to apply a Lagrangean smoothing, with k = 1 and #classes = 6:

P(+|T5) = 3/4 P(T5|+) = 3/4 (6/14)^2 2/14 1/14 2/14 = 2x10^-4

P(-|T5) = 1/4 P(T5|-) = 1/4 (2/9)^2 2/9 2/9 1/9 = 6x10^-5

Pros/cons and applications

Naive Bayes model is easy to build and particularly useful for very large data sets. Along with simplicity, Naive Bayes is known to outperform even highly sophisticated classification methods, if the assumption of independence holds. It perform well in case of categorical input variables compared to numerical variable(s).

Naive Bayes algorithms are very useful for:

- Real time prediction - it's very fast
- Multi class prediction
- Text classification/Spam filtering/Sentiment Analysis
- Recommendation System

Implementation

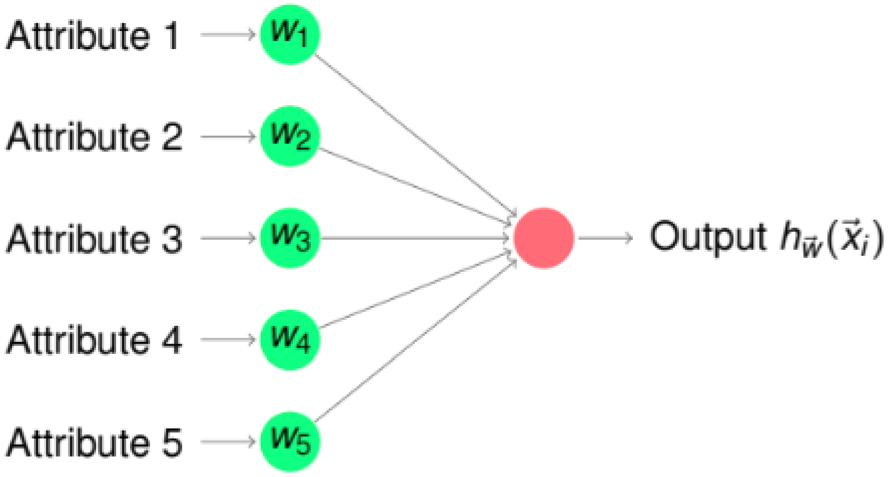
The **sklearn.naive_bayes** module implements Naïve Bayes algorithms in the *scikit-learn* library. There are a few different classifiers based on the Naïve Bayes algorithm, among which:

- GaussianNB() - usually use this
- BernoulliNB() - for multivariate Bernoulli models
- MultinomialNB() - for multinomial models

```
from sklearn.naive_bayes import GaussianNB
nbClass = GaussianNB()
```

Perceptron and Neural Networks

Perceptron / Neuron



The perceptron is the very basic component of any artificial neural network. It is modeled after the behavior of the neurons in the brain, having a number of input channels, a processing core and an output channel. Three phases can be identified for how the processing work in a neuron:

1. **weights** - a certain value in input, namely an attribute, is multiplied by a *weight* value that is assigned to the particular input; during the **learning phase**, the perceptron can adjust the weights based on the error of the last test result.
2. **sum** - the weighted signals are summed up to a single value, applying an offset called *bias* which is also updated during the learning phase.
3. **activation** - the result of the neuron's calculation is turned into an output signal by feeding it to a *activation function* (also called *transfer function*). Different functions can be used here, such as the *sign* function, the *step* function, or the *sigmoid* function.

Given an input $[x_i, y_i]$, where x_i data vector and y_i the label of the data, our goal is to minimize the error between output of the neuron (prediction) and the training label set. Generally, we want to minimize the **mean-square error** in order to increase our accuracy. This can be done by updating the set of weights, using **backward propagation of the error** from which we can compute the **weight update rule**:

Minimize Mean-square error: $J(\vec{w}) = \frac{1}{2}\Sigma(y_i - h_{\vec{w}}(\vec{x}_i))^2$

Stochastic Gradient Descent: $\vec{w} = \vec{w} - \eta \nabla J \vec{x}_i$

Gradient of the error function:

$$\nabla J = -\sum_i (y_i - h_{\vec{w}}(\vec{x}_i)) \nabla h_{\vec{w}}(\vec{x}_i)$$

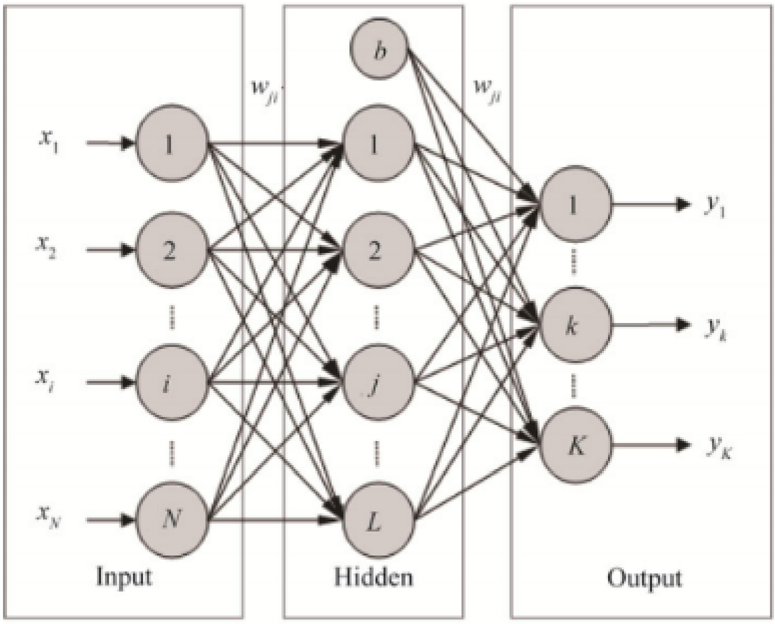
$$\nabla h_{\vec{w}}(\vec{x}_i) = h_{\vec{w}}(\vec{x}_i)(1 - h_{\vec{w}}(\vec{x}_i))$$

Weight update rule

$$\vec{w} = \vec{w} + \eta \sum_i (y_i - h_{\vec{w}}(\vec{x}_i)) h_{\vec{w}}(\vec{x}_i)(1 - h_{\vec{w}}(\vec{x}_i))\vec{x}_i$$

The stopping condition for the weight update can be either a threshold of iterations specified at the beginning, or just a non-improving criteria.

Artificial Neural Network



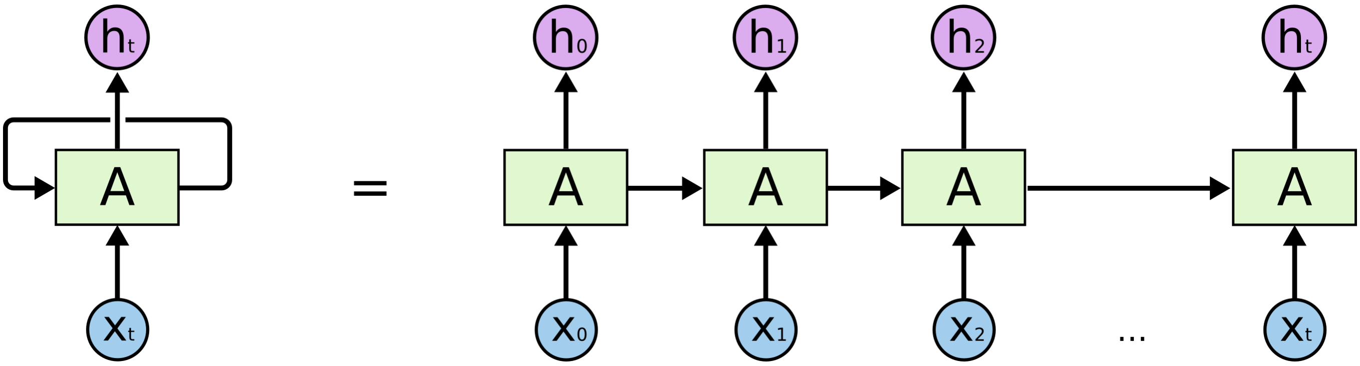
An artificial neural network is basically a network of perceptron, where a layer of neurons is fed by the previous layer, starting from the input data until the final result is obtained. Three main layers can be identified:

- **input layer**, the first layer of neurons, which are fed the input features vector and propagate their result to the next layer
- **hidden layer(s)**, one or more level of neurons, which keep propagating their results from layer to layer, until the final layer
- **output layer**, which is the last layer of neurons, which must generate the prediction.

Recurrent Neural Network

Traditional perceptron, and therefore neural networks, do not have no state persistence. The goal of RNN is to address this issue, by inserting loops in perceptron to allow persistence of information.

A RNN can be thought of as multiple copies of the same network, each passing a message to a successor. The unrolling of a recurrent neural network could look as follows:



Pros/cons and Applications

Neural networks are really powerful. They are easy to use and don't require any parameters definition. But, it is a *black-box* learning model, meaning that no easy interpretation between input and output can be understood from the model itself. Moreover, reaching convergence to a good result, especially with feature-rich data and many layers, can be quite both time and resource intensive.

Neural networks have been used on a variety of tasks, including **computer vision**, **speech recognition**, **machine translation**, social network filtering, playing board and video games, **medical diagnosis** and in many other domains.

Implementation

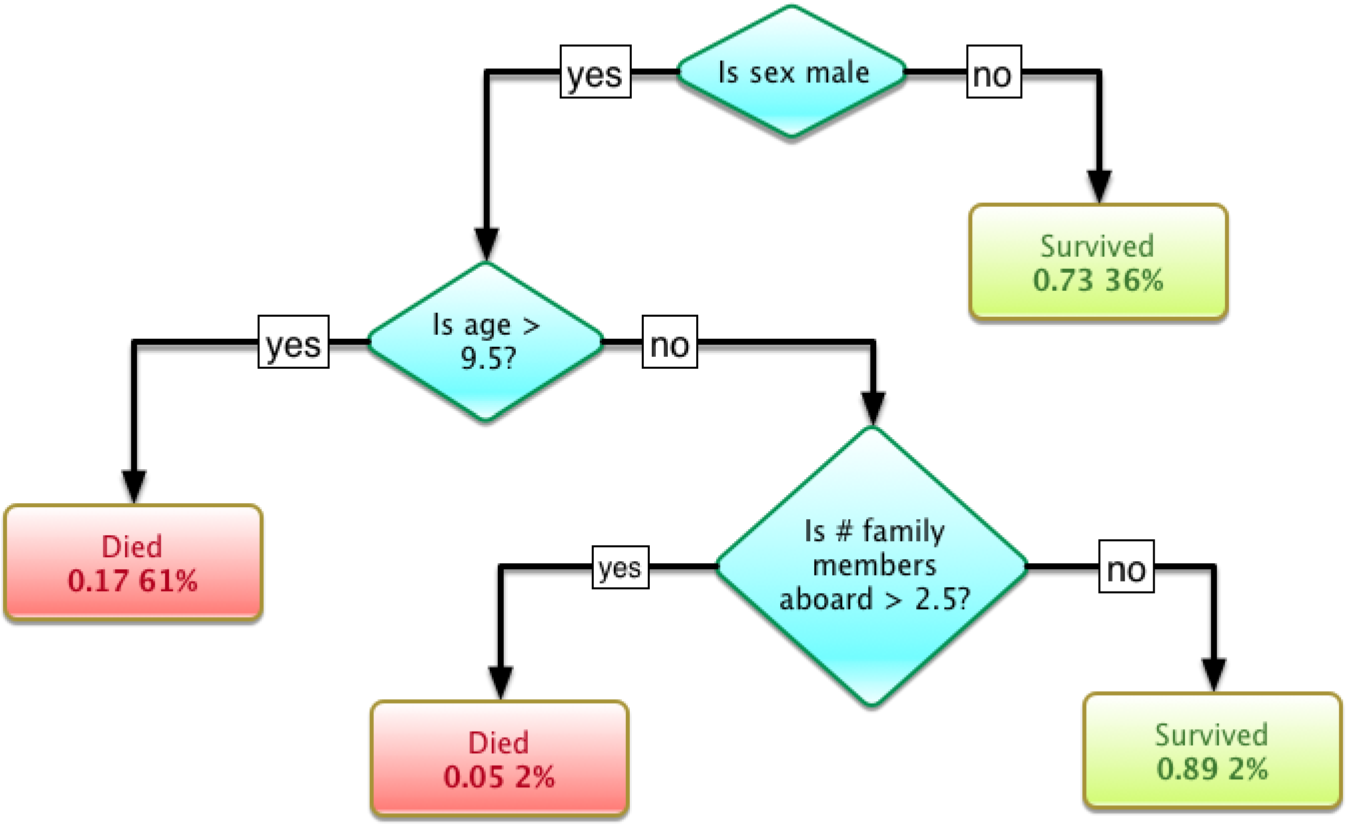
Artificial neural network are included in the *scikit* library under the name of **Multi-layer Perceptron**. It can be used for classification purposes by importing **MLPClassifier** from the **sklearn.neural_network**.

```
from sklearn.neural_network import MLPClassifier
mlpClass = MLPClassifier(hidden_layer_sizes=(5, 2)
#5 neurons on first hidden layer, 2 neurons on the second.
```

Decision Trees

Decision tree learning is based on building a structure (the **decision tree**) to classify an instance by splitting features according to their value, according to an **induction strategy**. Feature value allow to decide the label for the test instance, but in order to do so a splitting criteria has to be decided, such as *Gini impurity*, *information gain*, *variance*.

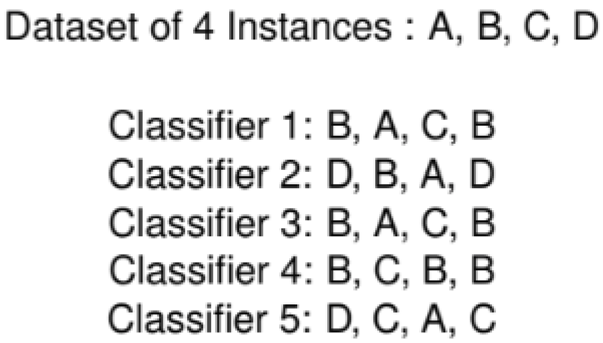
An example is the following, about the survival of passengers on the Titanic:



Ensemble techniques

Often, a decision tree is not enough to have a good accuracy. Ensemble methods are used to build more than one tree and join the results.

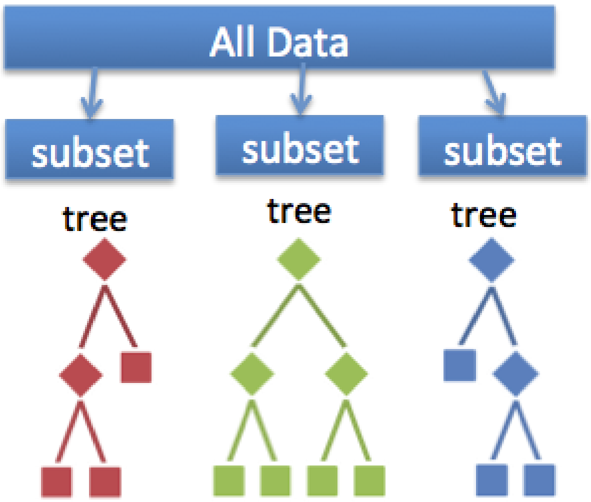
The **bagging/bootstrap** technique build a set of *M* base model, by picking random samples with replacement from a dataset of instances. In a nutshell, it generates *M* datasets in order to obtain *M* different classifier models.



Dataset of 4 Instances : A, B, C, D

Classifier 1: B, A, C, B
Classifier 2: D, B, A, D
Classifier 3: B, A, C, B
Classifier 4: B, C, B, B
Classifier 5: D, C, A, C

A specific type of bagging is **random forest**, one of the most popular classification algorithms. After a phase of bagging, it builds random trees which only use a random subset of the attributes. This combination of methods allows to generate a very accurate model, which generally beats any other model.



Finally, **boosting** is based on the concept of transorming a weak learner into a strong one: it takes the previously built model and tries to improve it incrementally. An example of this method is *AdaBoost*.

Pros/cons and Applications

A decision tree model is really easy to understand and interpret. It handles pretty well both numerical and categorical data, unlike other models (e.g. neural networks). Decision trees also requires very little data preparation: trees can handle qualitative predictors, therefore not needing data normalization. It performs extremely well with large datasets.

Trees are generally very non-robust, meaning that a slight change in data can bring a big change in the tree and consequently in the final prediction. Decision tree learners can incur into **overfitting**: the model created may be too specific for the problem at hand and not generalizable.

Stacking

Stacking (also called meta ensembling) is a model ensembling technique used to combine information from multiple predictive models to generate a new model. Often times the stacked model will outperform each of the individual models. For this reason, stacking is most effective when the base models are significantly different.

Generally, random forest is preferred over boosting, which is preferred over stacking.

Let's consider for example the spam dataset (../images/classification_example.png), posted above in the introduction. Let's suppose that three classifier have produced, for each instance, the following prediction.

Instance	True Label	Pred1	Pred2	Pred3
I1	Y	Y	N	Y
I2	Y	N	N	Y
I3	Y	Y	Y	Y
I4	N	N	N	Y
I5	N	N	Y	N

This is what is called **meta-dataset**. Finally, a **meta-classifier** has to be used to obtain a better prediction on incoming new instances.

Classification Evaluation

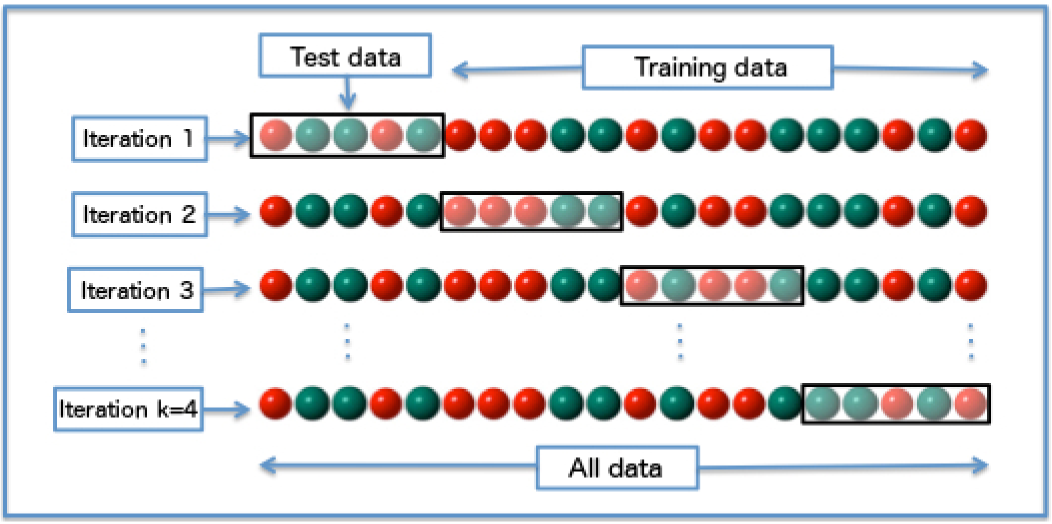
Building accurate classification models is necessary for accurate predictions. However, it is not clear how to evaluate the accuracy of a classification model.

There are many different methods for classification evaluation:

Error estimation

Hold-out is based on taking out a random and independent set of the available dataset and use it as a **test set** for the just built classification model.

k-Fold cross-validation is based on the concept that, in machine learning, knowing data label is expensive and therefore all lebeled data should be used to build the model. In cross-validation, the training dataset is split in *k* parts called *folds*, and the model is tested on the *i*-th fold, with *i* from 0 to *k-1*, testing the model *k* times. Usually, a good value is *k=10*.



Performance Measures

Confusion Matrix

A **confusion matrix**, or **error matrix**, is a table layout that allows visualization of the performance of an algorithm. Each row of the matrix represents the instances in a predicted class while each column represents the instances in an actual class (or vice versa). It makes it easy to see if the model is milabelling instances, and the percentage of it.

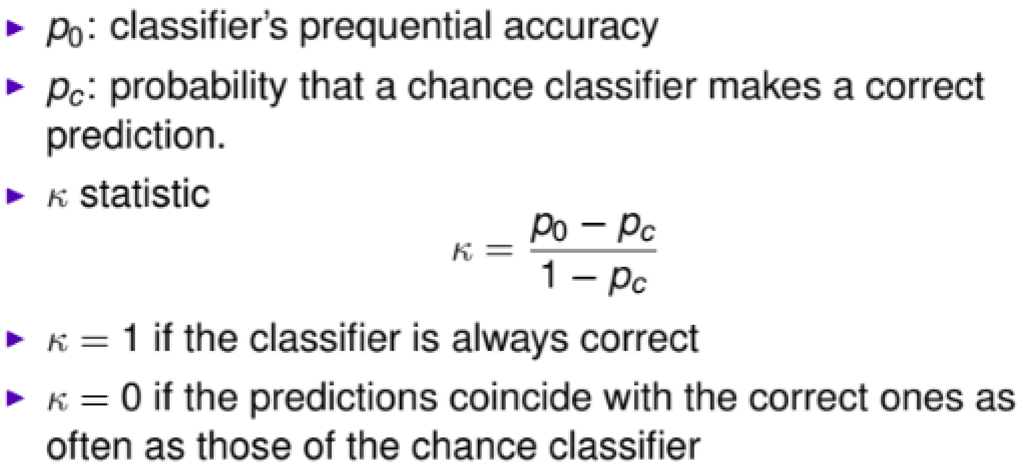
	Prediction +	**Prediction -**	**Total**
Correct +	tp	fn	tp+fn
Correct -	fp	tn	fp+tn
Total	tp+fp	fn+tn	N

Some metrics can be derived from this table:

- *Precision* = tp/(tp+fp)
- *Recall* = tp/(tp+fn) [AKA **True Prediction Rate**]
- *F1-Measure* = 2 * (precision * recall)/(precision+recall)

- *Accuracy* = (tp+tn)/N
- *Arithmetic mean* = (tp/(tp+fn) + tn/(fp+tn))/2
- *Geometric mean* = sqrt(tp/(tp+fn) * tn/(fp+tn))

Kappa Statistic



- p_0: classifier's prequential accuracy
- p_c: probability that a chance classifier makes a correct prediction.
- κ statistic

$$\kappa = \frac{p_0 - p_c}{1 - p_c}$$

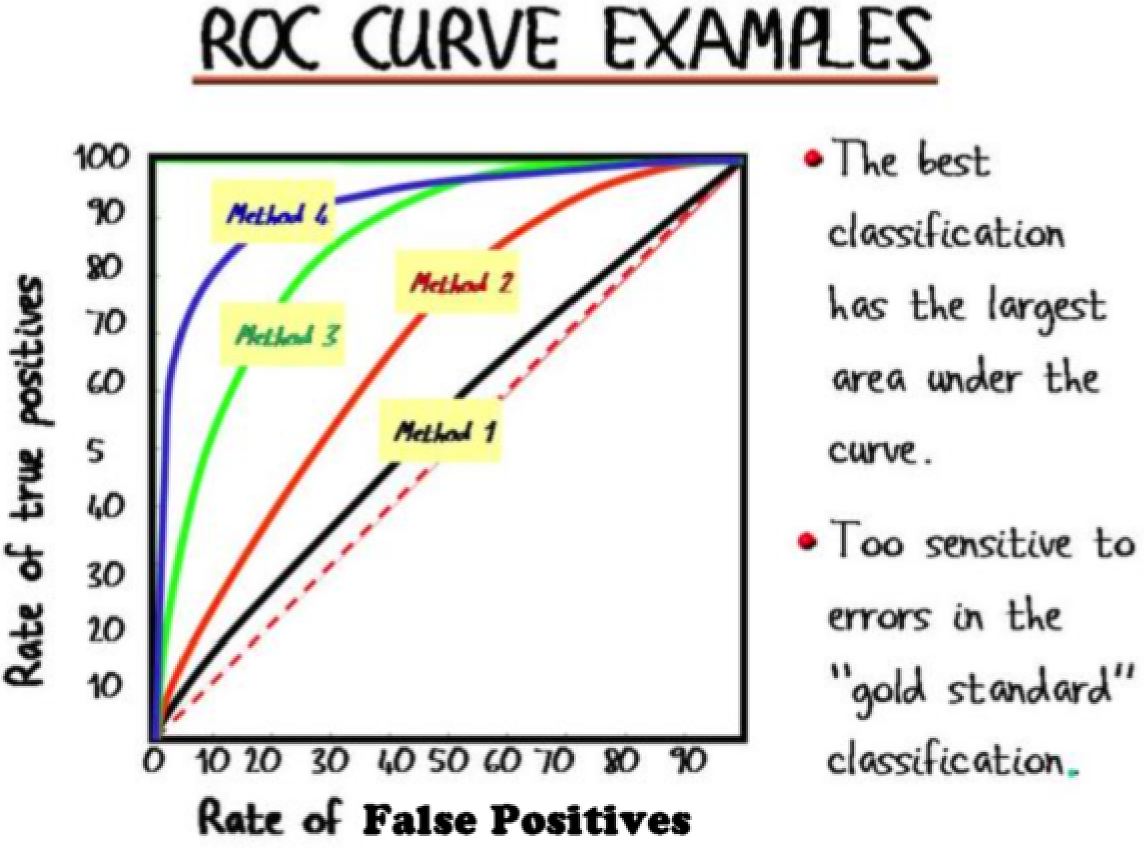
- $\kappa = 1$ if the classifier is always correct
- $\kappa = 0$ if the predictions coincide with the correct ones as often as those of the chance classifier

Matthews correlation coefficient (MCC)

$$\frac{tp \times tn - fp \times fn}{\sqrt{(tp + fp)(tp + fn)(tn + fp)(tn + fn)}}$$

Area Under the Curve (AUC)

A ROC space is defined by FPR = fp/(fp+tp) and TPR = tp/(tp+fn)

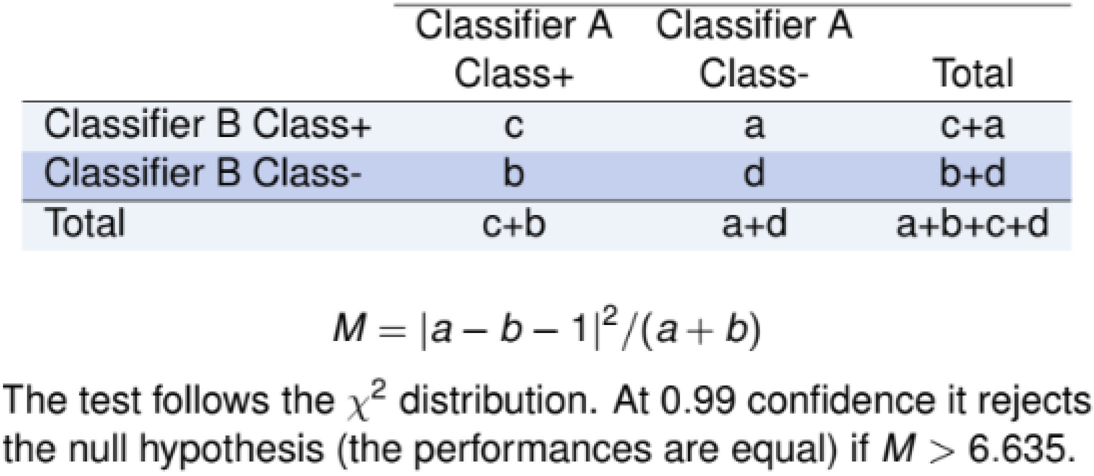


- The best classification has the largest area under the curve.
- Too sensitive to errors in the "gold standard" classification.

Statistical significance validation

In case 2 classifiers are built, their performance should be evaluated with respect to one another.

McNemar Test (2 classifiers)

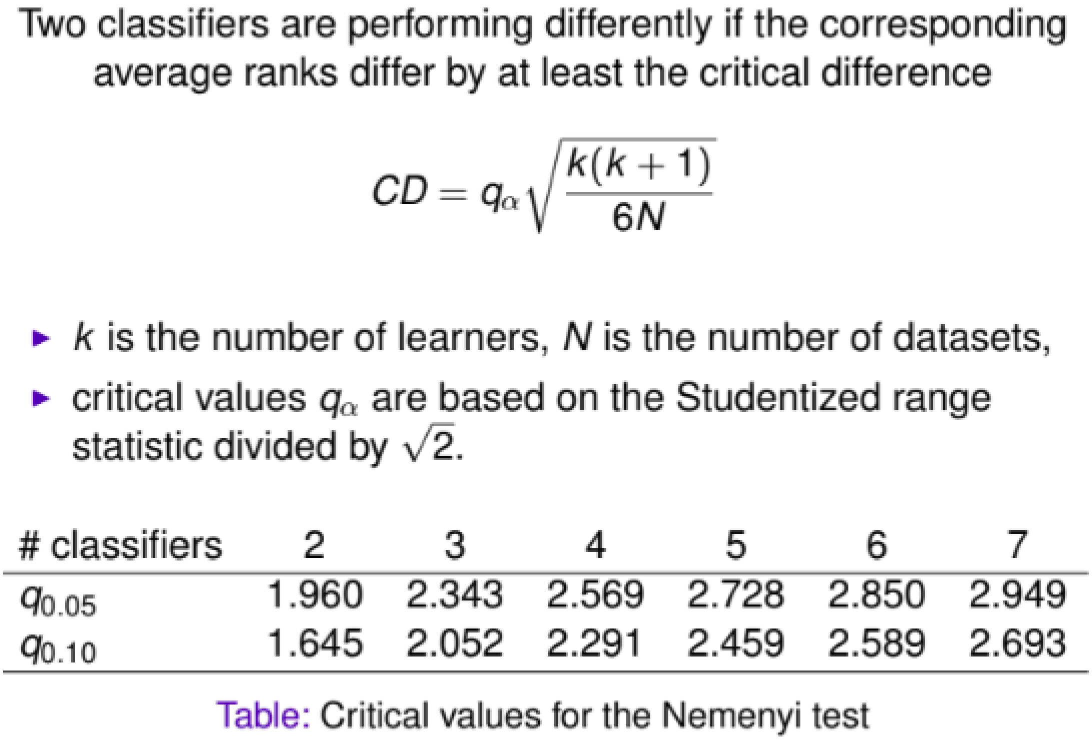


	Classifier A	Classifier A	
	Class+	Class-	Total
Classifier B Class+	c	a	c+a
Classifier B Class-	b	d	b+d
Total	c+b	a+d	a+b+c+d

$$M = |a - b - 1|^2/(a + b)$$

The test follows the χ^2 distribution. At 0.99 confidence it rejects the null hypothesis (the performances are equal) if $M > 6.635$.

Nemenyi Test (>2 classifiers)



Two classifiers are performing differently if the corresponding average ranks differ by at least the critical difference

$$CD = q_\alpha \sqrt{\frac{k(k+1)}{6N}}$$

- k is the number of learners, N is the number of datasets,
- critical values q_α are based on the Studentized range statistic divided by $\sqrt{2}$.

# classifiers	2	3	4	5	6	7
$q_{0.05}$	1.960	2.343	2.569	2.728	2.850	2.949
$q_{0.10}$	1.645	2.052	2.291	2.459	2.589	2.693

Table: Critical values for the Nemenyi test