Machine Learning and Pattern Recognition

We can use these definitions:

- **Machine Learning**: “A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E”.
- **Pattern recognition**: “Automatic discovery of regularities in data through the use of computer algorithms to take actions such as classifying the data into different categories”

We want to define models that are able to capture the regularities in our data and allow performing inference about the properties we are interested in.

⇒ The learning part is not human-defined and will be based on observed data.

Example are:

- Classification (the one we will be interested).
- Regression
- Density estimation

Depending on the task, we identify two main branches:

- **Supervised learning**: Along with the input data the system is provided with output data (e.g. class labels, function values, ...)
- **Unsupervised learning**: No feedback is provided to the system, whose goal is to identify some (useful) structure of the data (an example is clustering).

Unsupervised methods are often used as pre–processing steps for supervised learning tasks (a dimensionality reduction technique known as PCA is unsupervised).

But how do we classify by pattern?

- Assign a pattern to a class
- Classes represent characteristics of the objects that we are considering.

We can further divide the classification problem in different types:

- **Binary** classification.
- **Multiclass** classification which can be further divided in:
  - Closed-set classification (a class for each pattern).
  - Open-set classification (a “none of the above” class). In this situation we have a class that contains objects that are very different, so it will be hard to find a pattern.

We will divide this process of training models in three main steps:

- **Feature extraction**, representation of an object in a numerical form. We won’t see it in this course. Often it involves (complex) manipulations of the object to extract useful representations.
- **Dimensionality Reduction**: if we represent the data as a vector of dimensionality N we may want to reduce the dimensionality to a number M where M << N.
- **Classification**: mapping from the reduced vector representing the object to a label.

Dimensionality reduction helps reducing two problems:

- **Overfitting**: An over-complex model fits very accurately the observed data (very small training error), but is not able to provide accurate predictions for unseen data.
- **Curse of dimensionality**: Volumes in high-dimensional spaces grow very fast, and data becomes very sparse. This creates a situation where for the model it is very difficult to identify the patterns ⇒ very hard to classify.

We will call the mapping from vector data to the label **decision function**.
There are different types of decision function:

- **Discriminant model**: directly construct a function f(x) mapping feature vector x directly to a label.
- **Discriminative non-probabilistic model**: construct a function f(x) mapping feature vector x to a set of "scores".
- **Discriminative probabilistic model**: model class posterior probabilities $P(C_k|x)$ (where C_k is a label and x a sample) and assigns labels according to posterior probabilities.
- **Generative probabilistic model**: model the joint distribution of features and labels $P(x, C_k) = P(x|C_k)\,P(C_k)$ and apply Bayes theorem to get the posterior probability.

The decision functions should provide good generalization error:

⇒ Must be able to provide good predictions on **unseen** data (generalization)

- If it's a good function for seen data but terrible for unseen data it's **overfitting**
- If it's bad for seen and unseen data is **underfitting**

We won't study discriminant models because they tend to fail a lot and are too simple.

Generative models work by using **Bayes theorem**

$$P(\mathcal{C}_k|\boldsymbol{x}) = \frac{P(\boldsymbol{x}|\mathcal{C}_k)P(\mathcal{C}_k)}{P(\boldsymbol{x})}$$

Where $P(x \mid C_k)$ is the class conditional distribution and $P(C_k)$ is the application prior.
To compute the probability of class C_k given x we can compute the probability of x given the class C_k and multiply it by the probability of C_k. We don't need to compute P(x) because it's the same value independent of the class.

With the Discriminant probabilistic models we directly compute $P(\mathcal{C}_k|\boldsymbol{x}_t)$, we will need to incorporate application-dependent prior. We assign based on which class has the highest probability.

Discriminant non-probabilistic models will output a score and not a probability. We will still assign labels based on highest values.

Definition of a Model

Create a model M describing relationships between features and labels:

- Parametric models: The model depends on a set of parameters θ that we learn from the train data and whose size does not depend on the available data.
- Non-parametric models: The model complexity grows with the sample size (not good)

An example of non-parametric model is the k-NN, where we use nearest observed data to classify a sample.

We will focus on Parametric models because once we train the model and we get the parameters we don't need the train sample anymore, so we will carry much less information.

We need a training set which we will call D

$$\mathcal{D} = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_N, y_N)\}$$

Where x is an observed sample and y its label.

For learning our parameters θ we will use the **frequentist approach**: from the training set we estimate the best parameters θ*, usually by maximizing an objective function.

Given a model M and parameters θ* we can now start classifying a new sample x:

- Generative approach

  $$P(\boldsymbol{x}_t|\mathcal{C}_k, \boldsymbol{\theta}^*, \mathcal{M})$$

- Discriminant approach

  $$P(\mathcal{C}_k|\boldsymbol{x}_t, \boldsymbol{\theta}^*, \mathcal{M})$$

If we train different models how do we understand the best one?

⇒ We divide our train set in two sets:

- A **train set** to train the model.
- A **test set** to evaluate the model.

We use the test set because we need to know the real label and we want to understand if the model can correctly classify it.
It's important that the two sets are disjoint, so we are sure that the model is classifying unseen data.

Our models will also use *hyperparameters* like the number of dimension to reduce or the number of clusters in GMM:

- Those cannot be trained using the train sample but we need to understand to find the best values for the dataset.

  ⇒ We will employ **cross-validation**.

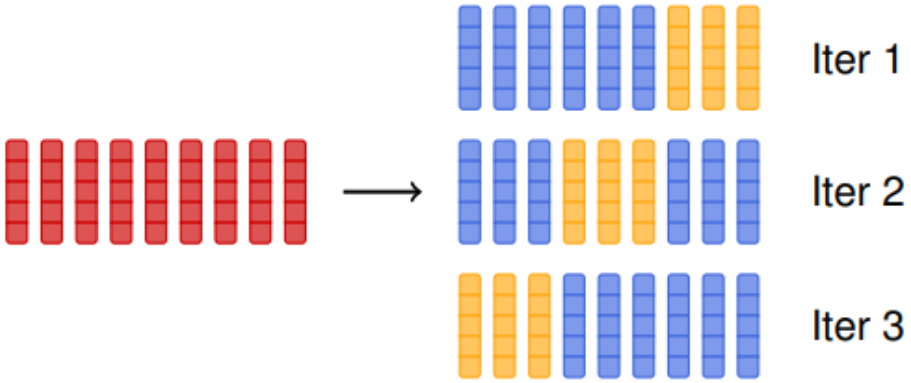
For cross-validation we will use a third set from our original dataset, called the validation **set**. The validation set is used to assess the influence of model hyperparameters on prediction accuracy in order to select good hyperparameter values.

So we have:

- Test set.
- Training set, which we divide in
    - Training set for models
    - Validation set.

This creates the **K-fold cross validation**:

- Subdivide the training data in K folds
- Repeatedly use K − 1 folds as training data, the remaining fold as validation set
- Combine the validation results on the K folds and select optimal values for hyperparameters
- Retrain the model using the selected hyperparameters but using all data



Dimensionality Reduction

Dimensionality reduction techniques compute a mapping from the n–dimensional feature space to a m–dimensional space, with m << n.

- To remove noise from the data
- To simply classification
- To avoid overfitting
- Easier to see patterns in our data

We want to retain **discriminant information**, information useful to classify unobserved data.

We will focus on two linear methods:

- Unsupervised: Principal Component Analysis (PCA)
- Supervised: Linear Discriminant Analysis (LDA)

In both cases, we want to find a subspace of the feature space that preserves most of the “useful” information.

For computing this subspace we will need some notion of linear algebra.

Let $\boldsymbol{A}$ be a square symmetric $n \times n$ matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$

$\boldsymbol{A}$ admits an eigen–decomposition

$$\boldsymbol{A} = \boldsymbol{V}\boldsymbol{\Sigma}\boldsymbol{V}^{-1} = \boldsymbol{V}\boldsymbol{\Sigma}\boldsymbol{V}^{T}$$

- $\boldsymbol{V}$ is an orthogonal $n \times n$ matrix whose columns are the (right) eigenvectors of $\boldsymbol{A}$
- $\boldsymbol{\Sigma}$ is a diagonal $n \times n$ matrix whose elements are the eigenvalues of $\boldsymbol{A}$

Let A be a matrix of real numbers. We know the dot product between a matrix A and a vector x is Ax.
For some x we get Ax = ax, meaning that Ax doesn't create a new vector but shrinks or stretches the original vector x. In this case x is an eigenvector and a is the corresponding **eigenvalue**.

Example: For this matrix

$$\begin{bmatrix} -6 & 3 \\ 4 & 5 \end{bmatrix}$$

an eigenvector is

$$\begin{bmatrix} 1 \\ 4 \end{bmatrix}$$

with a matching eigenvalue of 6

Let's do some matrix multiplies to see if that is true.

Av gives us:

$$\begin{bmatrix} -6 & 3 \\ 4 & 5 \end{bmatrix}\begin{bmatrix} 1 \\ 4 \end{bmatrix} = \begin{bmatrix} -6\times1+3\times4 \\ 4\times1+5\times4 \end{bmatrix} = \begin{bmatrix} 6 \\ 24 \end{bmatrix}$$

λv gives us :

$$6\begin{bmatrix} 1 \\ 4 \end{bmatrix} = \begin{bmatrix} 6 \\ 24 \end{bmatrix}$$

We also need the singular value decomposition (SVD):

A generic rectangular matrix $\boldsymbol{A} \in \mathbb{R}^{n\times m}$ always admits a Singular Value Decomposition (SVD) of the form:

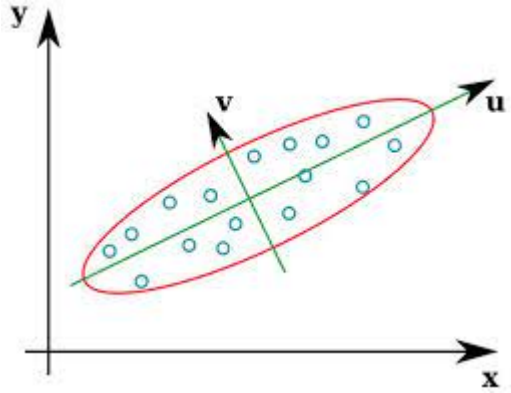
$$\boldsymbol{A} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$$

where

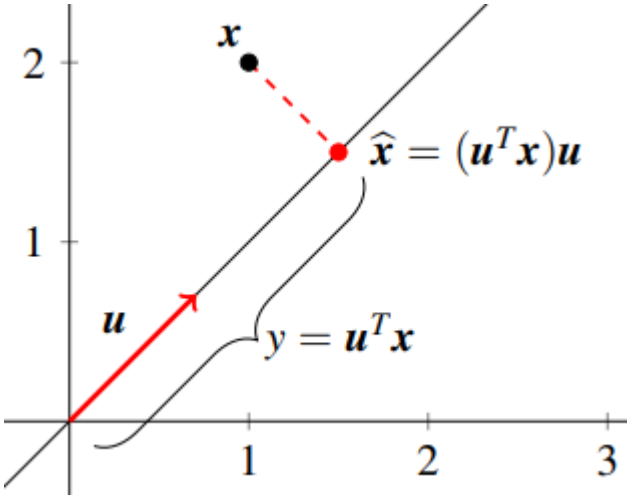
- $\boldsymbol{U}$ is an orthogonal $n \times n$ matrix of eigenvectors of $\boldsymbol{A}\boldsymbol{A}^T$
- $\boldsymbol{V}$ is an orthogonal $m \times m$ matrix of eigenvectors of $\boldsymbol{A}^T\boldsymbol{A}$
- $\boldsymbol{\Sigma}$ is a diagonal rectangular matrix containing the singular values of $\boldsymbol{A}$ (in decreasing order)

The most important part here is the following:

If we depict our matrix of data with geometrical form (typically an ellipse) we can find out that the eigenvector **always represents** the direction with most variance, and the more variance the greater the eigenvalues. An example here:



The eigenvectors are u and v and the eigenvalue of u is greater than v.



Last thing is the projection of a point x in a subspace u:

We can compute $y = (u^T x)$ the projection of x over u.
X is the sample in the original n-dimensional space, while u is the m-dimensional subspace.
In this situation n = 2 and m = 1 (but this holds in general)

We can compute the projected point in the original n-dim space as $\hat{x} = (u^T x)u$.
In the previous image we can see that $\hat{x}$ and the original x end up in different positions.

For U defined as a matrix we can compute the subspace of U and project our vectors:

The columns of $\boldsymbol{U} = [\boldsymbol{u}_1, \boldsymbol{u}_2]$ form a basis of $\mathbb{R}^2$

$\boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \boldsymbol{u}_1^T \boldsymbol{x} \\ \boldsymbol{u}_2^T \boldsymbol{x} \end{bmatrix} = \boldsymbol{U}^T \boldsymbol{x}$ is the projection of $\boldsymbol{x}$ over the column space of $\boldsymbol{U}$

The representation (reconstruction) of $\boldsymbol{y}$ in the original space can be obtained as

$$\hat{\boldsymbol{x}} = y_1 \boldsymbol{u}_1 + y_2 \boldsymbol{u}_2 = \boldsymbol{U}\boldsymbol{y} = \boldsymbol{U}\boldsymbol{U}^T \boldsymbol{x}$$

Principal Component Analysis

We are given X, the matrix representing our dataset, where $x \in R^N$. We need to find a projection in a new dimension R^M that allows preserving **most of the information**.

We will defined P the projection matrix, an orthonormal matrix with m columns so it creates a subspace in R^M. The new dataset is composed of the projection of x into the subspace of P, $y = P^T x$ and we can reconstruct x as $\hat{x} = P\ y$.

We need to define a criterion for estimating P: the minimization of the **average reconstruction error**. We want the matrix P such that the original x and the newly projected x have as much information in common (so the reconstruction error, their difference, is the minimum). Using K as the number of samples:

$$\frac{1}{K} \sum_{i=1}^{K} \|\boldsymbol{x}_i - \hat{\boldsymbol{x}}_i\|^2$$

We know that $\hat{x} = Py$ so

$$\frac{1}{K} \sum_{i=1}^{K} \|\boldsymbol{x}_i - \boldsymbol{P}\boldsymbol{y}_i\|^2$$

and also $y = P^T x$

$$\frac{1}{K} \sum_{i=1}^{K} \|\boldsymbol{x}_i - \boldsymbol{P}\boldsymbol{P}^T \boldsymbol{x}_i\|^2$$

Therefore we want P such that we minimize this average reconstruction error.

$$\arg\min_{\boldsymbol{P}} \frac{1}{K}\sum_{i=1}^{K} \left\|\boldsymbol{x}_i - \boldsymbol{P}\boldsymbol{P}^T\boldsymbol{x}_i\right\|^2$$

With some math tricks we can show that this is equal to minimizing

$$\frac{1}{K}\sum_{i=1}^{K} \left(\boldsymbol{x}_i^T\boldsymbol{x}_i - \mathrm{Tr}(\boldsymbol{P}^T\boldsymbol{x}_i\boldsymbol{x}_i^T\boldsymbol{P})\right)$$

We also need to note that changing P won't affect the values $1/K$ and $x_i^T x_i$ so the only thing we can do is maximize $\mathrm{Tr}(\boldsymbol{P}^T\boldsymbol{x}_i\boldsymbol{x}_i^T\boldsymbol{P})$ for each x_i.

> ⇒It can be shown that the optimal solution is then given by the matrix P whose columns are the m eigenvectors of corresponding to the m largest eigenvalues of the **covariance matrix**

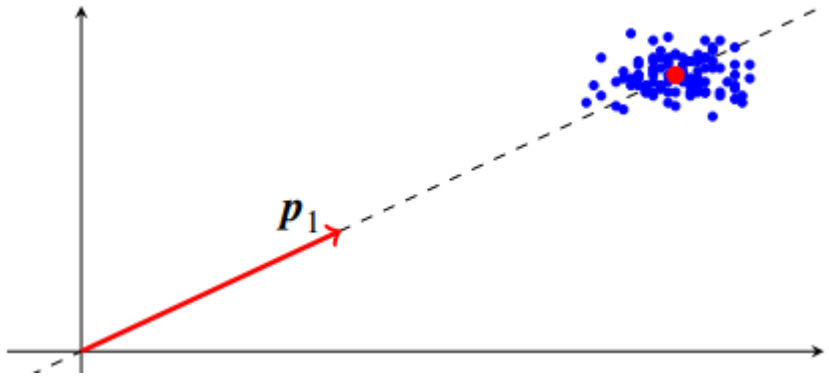
Using SVD on the covariance matrix

$$\frac{1}{K}\sum_{i=1}^{K} \boldsymbol{x}_i\boldsymbol{x}_i^T = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{U}^T$$

We can define P* using the columns of the eigenvector matrix U, taking m columns to define the m feature space that we need:

$$\boldsymbol{P}^* = [\boldsymbol{u}_1 \ldots \boldsymbol{u}_m]$$

We have a problem if the dataset is far from the origin: PCA will **always** create a useless direction that connects the origin to the dataset mean:



Being useless we will compute P using the **empirical covariance matrix**: removing the mean from all the points.

$$\frac{1}{K}\sum_{i} (\boldsymbol{x}_i - \bar{\boldsymbol{x}})(\boldsymbol{x}_i - \bar{\boldsymbol{x}})^T$$

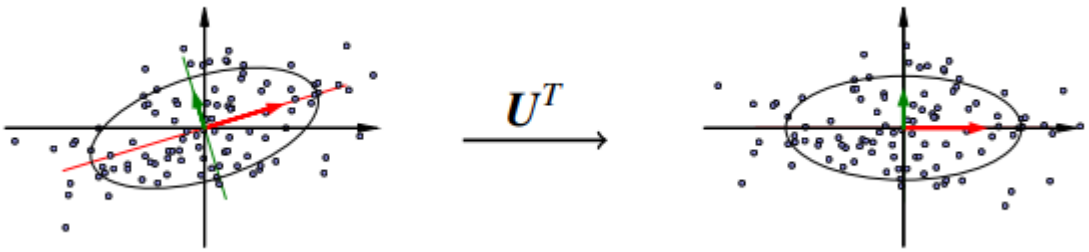
So the steps for using PCA are

- Compute sample mean $\bar{x}$
- Center data: $z_i = x_i - \bar{x}$
- Compute the sample covariance matrix

$$C = \frac{1}{K} \sum_i (x_i - \bar{x})(x_i - \bar{x})^T = \frac{1}{k} \sum_i z_i z_i^T$$

- Compute the eigen–decomposition of $C = U \Sigma U^T$
- Project the data in the subspace spanned by the m columns of U corresponding to the m highest eigenvalues (matrix P): $y_i = P^T z_i = P^T (x_i - \bar{x})$

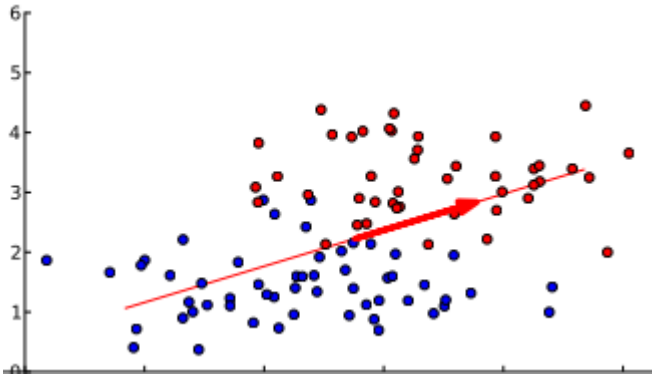
Using m = n we transpose our dataset such that it's centered and principal directions where the data spans are uncorrelated:



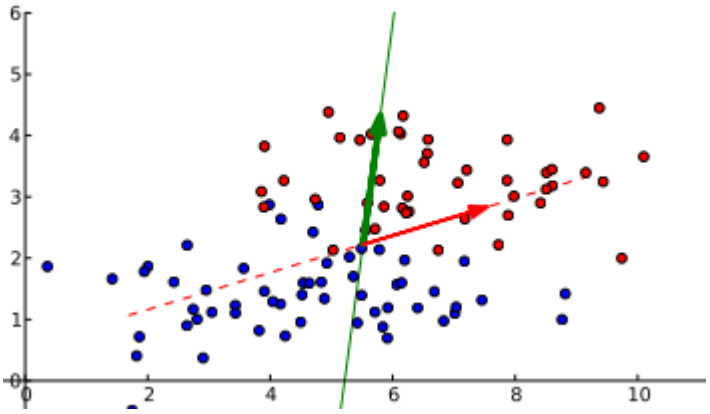
How can we find m? It's a hyperparameter so cross-validation.

Linear Discriminant Analysis

The problem of PCA lies in its being an unsupervised model. It will work without looking at the labels of the data.
Like for the following dataset:



The red line is the most variance dimension, but this dimension doesn't separate our class.
A better choice would be the green line:



A solution could be to use as the subspace a line that is proportional to the distance between the means of the two classes (not used because it's not perfect).

The idea of **LDA** is:

- ➢ "find a direction that has a large separation between the classes and small spread inside each class".

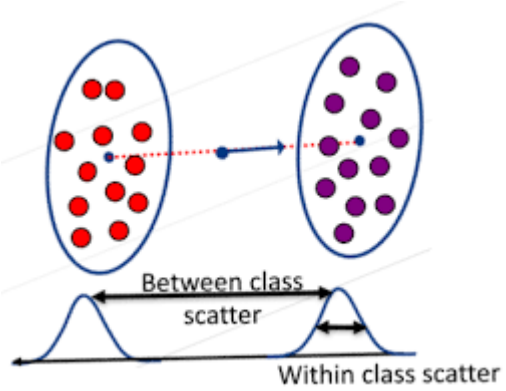
For doing this we define two new matrices:

- Between class variability (Sb): it measures the distance between each mean class and the mean of the dataset.
- Within class variability (Sw): it measures the distance between all the points and their respective classes.

$$\boldsymbol{S}_B \triangleq \frac{1}{N}\sum_{c=1}^{K} n_c \left(\boldsymbol{\mu}_c - \boldsymbol{\mu}\right)\left(\boldsymbol{\mu}_c - \boldsymbol{\mu}\right)^T$$

$$\boldsymbol{S}_W \triangleq \frac{1}{N}\sum_{c=1}^{K}\sum_{i=1}^{n_c} \left(\boldsymbol{x}_{c,i} - \boldsymbol{\mu}_c\right)\left(\boldsymbol{x}_{c,i} - \boldsymbol{\mu}_c\right)^T$$

Ideally what we are computing is:



So our LDA objective can be rewritten as maximize the between-class variability over within-class variability ratio:

$$\max_{\boldsymbol{w}} \frac{\boldsymbol{w}^T \boldsymbol{S}_B \boldsymbol{w}}{\boldsymbol{w}^T \boldsymbol{S}_W \boldsymbol{w}}$$

Instead of compute the two matrice for the projected points, we can show that this is equal to compute our matrices for the **original** samples and then we transpose those with $w^T S\, w$. This simplifies the resolution of our objective function:

$$\mathcal{L}(\boldsymbol{w}) = \frac{s_B}{s_W} = \frac{\boldsymbol{w}^T \boldsymbol{S}_B \boldsymbol{w}}{\boldsymbol{w}^T \boldsymbol{S}_W \boldsymbol{w}}$$

Solving the gradient of L(w) equal to 0 we get:

$$\boldsymbol{S}_W^{-1} \boldsymbol{S}_B \boldsymbol{w} = \lambda(\boldsymbol{w}) \boldsymbol{w}$$

$$\lambda(\boldsymbol{w}) = \mathcal{L}(\boldsymbol{w}) = \frac{\boldsymbol{w}^T \boldsymbol{S}_B \boldsymbol{w}}{\boldsymbol{w}^T \boldsymbol{S}_W \boldsymbol{w}}$$

So we see that w is an eigenvector and lambda w an eigenvalue.

The maximum of $\mathcal{L}$ is thus the eigenvector of $\boldsymbol{S}_W^{-1} \boldsymbol{S}_B$ corresponding to the largest eigenvalue

For the binary case we get:

$$\boldsymbol{w} \propto \boldsymbol{S}_W^{-1} (\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)$$

So w it's a line connecting the two means and rebalanced using the within class matrix.

This version of LDA was originally used as a classifier, we need to change it a little bit so it becomes a **dimensionality reduction algorithm**.

What we need to change is the objective function. One of the most used is:

$$\mathcal{L} = \mathrm{Tr}\left(\widehat{S_W}^{-1}\widehat{S_B}\right)$$

The solution to the problem doesn't change, we will take the m eigenvectors corresponding to the m largest eigenvalues of

$$S_W^{-1}S_B$$

From the definition of Sb (matrix of distances between class mean and dataset mean) we understand that we have at most C - 1 eigenvectors. So the value of m can be **at most C - 1 (with C number of classes)**.

There is an algorithm to resolve this without the need of finding the eigenvalues and eigenvectors:

1. Whiten Sw : using the SVD decomposition of Sw we can transform Sw in the identity matrix.
2. Diagonalize Sb: using the SVD decomposition of the new Sb (we changed the data by changing Sw) we get the m eigenvectors that we need, taking just the first m columns.

The first step is:

$$S_W \rightarrow I,\ S_B \rightarrow PS_BP^T$$

Second step is

$PS_BP^T = U_B\Sigma_B U_B^T$ using U_B^T to find the eigenvectors.

Putting Sw equal to the identity matrix ensures that all the data is equidistant to the mean, so we can take the most variance directions from the new Sb (taking the highest variance directions ensures that the class means don't get closer).

Probability review

Since our goal is to make **predictions** (so we can take a set of actions) we need a way to model complex phenomena:

- We will use **random variables for modeling random events**

We describe random events in terms of their probability:

- **Classical interpretation**: probability is equal to the ratio of number of favorable events and total number of events.
- **Frequentist**: similar to classical, but here we measure doing a large number of repeated trials.
- **Bayesian**: probability as a measure of belief of something to happen.

Let Ω be the set of all possible outcomes.
We define A as the set of all possible events with the following properties:

- A is a collection of subsets of Ω.
- $\Omega \in$ A.
- If $A \in$ A also $A^C \in$ A.
- If $A_1\ A_2\ A_3 \ldots \in$ A then their union must be equal to A: $\cup_i A_i$ = A.

A probability function is a function P : A → R with the following properties:

- P(Ω) = 1 (the probability of all events must be equal to 1).
- P($\cup_i A_i$) = $\sum P(A_i)$ if the events are mutually exclusive

We also have some other properties that we can prove, for any set of events A and B:

- $P(\emptyset) = 0$
- $A \subset B \implies P(A) \leq P(B)$
- $P(A) \leq 1$
- $P(A^C) = 1 - P(A)$
- $P(A \cup B) = P(A) + P(B) - P(A \cap B)$
- $P\left(\bigcup_n A_n\right) = 1 - P\left(\bigcap_n A_n^C\right)$

In the case of rolling a dice we have:

$$\Omega = \{1, 2, 3, 4, 5, 6\}$$

And if we want to compute the probability of getting an even number:

$$P(\{2, 4, 6\})$$

We can note that those are independent events so we can use our properties

$$= P(\{2\}) + P(\{4\}) + P(\{6\}) = \frac{1}{2}$$

We can also define the **conditional probability** as:

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

The probability of events A to happen when we know that event B has occurred.

The **bayes formula** helps us to compute conditional probability:

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

Two events are independent iff:

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

Meaning the probability of the two events happening together is equal to the product of their individual probabilities ⇒ one event does not affect the other one.

1. Independence: Two events, A and B, are considered independent if the occurrence of one event does not affect the probability of the other event happening. In other words, knowing that event A has occurred provides no information about the likelihood of event B occurring.

To denote the probability that two events happen together we will use

$P(A, B)$ instead of $P(A \cap B)$

We extend the notion of probability function to **random variables**:

⇒ A random variable X is a function $\Omega \rightarrow$ R.

A random variable can be used to compute probabilities such that

$$P(X \leq x) = P(\{\omega \in \Omega : X(\omega) \leq x\})$$

For each outcome of Ω we compute a real value. The probability of the random variable of being minor a x is equal to the probability of a subset of Ω.

For example we can throw two dice and sum the results. This defines a random variable, and we can compute the probability of the sum being lower than a threshold.

More in general computing the probability

$P(X \in A)$ where A is a subset of the real numbers

Can be computed as the probability of the set

$$\{\omega \in \Omega : X(\omega) \in A\}$$

We can also compute more complex probabilities as

$$\{\omega : a < X(\omega) \leq b\} = \{\omega : X(\omega) > a\} \cap \{\omega : X(\omega) \leq b\}$$

We can thus define the **cumulative distribution function (cdf)** as

$$F_X(x) = P(X \leq x)$$

With the properties

1) $0 \leq F_X(x) \leq 1 \quad \forall x \in \mathbb{R}$

2) F_X is non-decreasing, i.e. $x_1 < x_2 \implies F_X(x_1) \leq F_X(x_2)$

3) $\lim_{x \to -\infty} F_X(x) = 0, \lim_{x \to \infty} F_X(x) = 1$

4) F_X is right-continuous: $\lim_{x \downarrow x_0} F_X(x) = F_X(x_0)$

So now we can compute

$$P(a < X \leq b) = F_X(b) - F_X(a)$$

$$P(X = x_0) = F_X(x_0) - \lim_{x \uparrow x_0} F_X(x)$$

If the random variable is **discrete** we can compute a **probability mass function (pmf)**

$$f_X(x) = P(X = x)$$

With the properties

$$f_X(x) \geq 0$$

$$\sum_{x \in \mathcal{S}} f_X(x) = 1$$ With S **support**, all the values x such that $f_X(x) > 0$.

We can compute c.d.f from pmf:

$$F_X(x) = \sum_{t \leq x} f_X(t)$$

But what happens when the values that we map are **continuous**?
We can still define the c.d.f. but now the mass function always returns zero:

$$P(X = x) = 0$$

(in the continuous realm the probability of getting a specific value is 1/infinity).
And so:

$$P(a < X \leq b) = P(a \leq X \leq b) = P(a < X < b) = P(a \leq X < b)$$

We can still define a density also for continuous random variable, with similar properties:

1) $f(x) \geq 0 \quad \forall x \in \mathbb{R}$

2) f is integrable over $\mathbb{R}$

3) $\int_{-\infty}^{+\infty} f(x)dx = 1$

Point 3 is the same as the case for the discrete random variable, we just replace the sum with an integral.

We can also compute the c.d.f from p.m.f., again we replace the sum with an integral:

$$F_X(x) = \int_{-\infty}^{x} f_X(t)dt$$

With density we can compute different probabilities just by integrating:

$$P(a \leq X \leq b) = \int_{a}^{b} f_X(x)dx$$

Random Vectors

By putting together a set of m random variables we can create a **Random Vector X** whose components X1 . . . Xm .

If we define $x = (x_1, x_2, ..., x_m)$ as a set of threshold we can still define the c.d.f as

$$F_{\boldsymbol{X}}(\boldsymbol{x}) = P(X_1 \leq x_1, \ldots, X_m \leq x_m)$$

Joint cumulative distribution function.

Also (for the discrete case, we don't need the others):

$$\begin{aligned} f_{\boldsymbol{X}}(\boldsymbol{x}) &= P(\boldsymbol{X} = \boldsymbol{x}) \\ &= P(X_1 = x_1, \ldots, X_m = x_m) \end{aligned}$$

$$F_{\boldsymbol{X}}(\boldsymbol{x}) = \sum_{y_1 \leq x_1} \cdots \sum_{y_m \leq x_m} f_{\boldsymbol{X}}(\boldsymbol{y})$$

Here **y** is a vector of possible thresholds.

Remember that the , is just a way to represent the probability of two events happening together $P(A, B) \Rightarrow P(A \cap B)$

Lastly we define the conditional density:

$$f_{X|Y}(x|y) = \frac{f_{X,Y}(x,y)}{f_Y(y)}$$

Mean and Variance

We need to define two important measures in probability theory (note that we have the same definition for both random variable and vector)

The **mean**:

Discrete R.V.:

$$\mathbb{E}_X[X] = \sum_{x \in \mathcal{S}} x\, f_X(x)$$

Continuous R.V.:

$$\mathbb{E}_X[X] = \int_{\mathcal{S}} x\, f_X(x)\, dx$$

And the **variance**:

$$\operatorname{var}(X) = \mathbb{E}_X\left[(X - \mathbb{E}_X[X])^2\right]$$

The mean of the distance between each point and its mean squared.

For the random vectors instead of getting a value we get another vector, a vector of means:

$$\mathbb{E}_{\boldsymbol{X}}[\boldsymbol{X}] = \begin{bmatrix} \mathbb{E}_{X_1}[X_1] \\ \vdots \\ \mathbb{E}_{X_m}[X_m] \end{bmatrix}$$

We just compute the mean for each random variable.

Using the definition of covariance for the random variables:

$$\operatorname{cov}(X,Y) = \mathbb{E}_{X,Y}[XY] - \mathbb{E}_X[X]\mathbb{E}_Y[Y]$$

(that tell us if two random variable are correlated and how much)

We can define the **covariance matrix for a random vector**

$$\boldsymbol{\Sigma} = \operatorname{cov}(\boldsymbol{X}) = \mathbb{E}\left[(\boldsymbol{X} - \mathbb{E}[\boldsymbol{X}])(\boldsymbol{X} - \mathbb{E}[\boldsymbol{X}])^T\right]$$

$$= \begin{bmatrix} \operatorname{var}(\boldsymbol{X}_1) & \operatorname{cov}(\boldsymbol{X}_1, \boldsymbol{X}_2) & \cdots & \operatorname{cov}(\boldsymbol{X}_1, \boldsymbol{X}_m) \\ \operatorname{cov}(\boldsymbol{X}_1, \boldsymbol{X}_2) & \operatorname{var}(\boldsymbol{X}_2) & \cdots & \operatorname{cov}(\boldsymbol{X}_2, \boldsymbol{X}_m) \\ \vdots & \vdots & \ddots & \vdots \\ \operatorname{cov}(\boldsymbol{X}_m, \boldsymbol{X}_1) & \operatorname{cov}(\boldsymbol{X}_m, \boldsymbol{X}_2) & \cdots & \operatorname{var}(\boldsymbol{X}_m) \end{bmatrix}$$

That tells us how much each component of the random vector is correlated to the other components.If the matrix is diagonal then **each component is uncorrelated from another**. We also note that Cov(A,B) = Cov(B, A) so the covariance matrix is symmetric.

Distributions

Bernoulli distribution is used to compute the outcome of a binary event given the probability p. We will denote the random variable as {0, 1} where 1 represents the success of probability p.

$$= \operatorname{Ber}(x|p) = \begin{cases} p & \text{if } x = 1 \\ 1 - p & \text{if } x = 0 \end{cases}$$

$$= p^x (1-p)^{1-x}$$

The **binomial distribution** is used to count the number of success of probability p in a given number of trials n:

$$\binom{n}{x} p^x (1-p)^{n-x}$$ where $$\binom{n}{x}$$ compute how many ways we can arrange the x successes in the n trials. **binomial coefficient**
That's because we don't care about the order of the successes, so we need to take into account that there are more possibilities of a fixed number of successes to happen.

The binomial and Bernoulli works for events with just two outcomes. For K outcome we have two generalization:

Categorical distribution: given a vector P of probabilities (P[i] probability of event i)

$$P(X = x) = p_x = \prod_i p_i^{\mathbb{I}[x=i]}$$

Where $\mathbb{I}[x=i]$ returns 1 when x==i, 0 otherwise.
If we want the probability of i = 3 we get 1 * 1 * P[3] * 1 *...

In case we are interested in getting more than one probability (probability that events 2 and 3 happens together) we can use a vector X = (0, 1, 1, 0, … 0)
And compute:

$$f_{\boldsymbol{X}}(\boldsymbol{x}) = \prod_i p_i^{x_i}$$

For the **multinomial** we generalize the definition of the binomial. We will hence write:

$$\boldsymbol{X} \sim \mathrm{Mul}(n, \boldsymbol{p})$$

$$f_{\boldsymbol{X}}(\boldsymbol{x}) = \frac{n!}{x_1! \cdots x_K!} \prod_{i=1}^{K} p_i^{x_i}$$

x in this situation is the number of occurrences of each event: x[i] = number occurrences of event i.
A categorical multiplied to take into account the fact that we don't care of the order of the successes.

Last distribution is the **normal/gaussian distribution**:

$$\mathcal{N}(\mu, \sigma^2)$$

Where

μ is the mean, and as the name suggests $\mathbb{E}[X] = \mu$

σ is called standard deviation, and $\mathrm{var}(X) = \sigma^2$

Later we will work with the inverse of the variance, the **precision**: $\lambda = \frac{1}{\sigma^2}$

We can extend gaussian distribution to random vectors. In this context **each r.v.** is an identical and independently distributed gaussian.

So the distribution of X is given by the joint distribution:

$$f_{\boldsymbol{X}}(\boldsymbol{x}) = \prod_{i=1}^{N} f_{X_i}(x_i)$$

Also we write $\boldsymbol{X} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ to define the vector of means and the **covariance matrix** of X, the multivariate Gaussian distribution.

We also have the precision matrix, the inverse of the covariance matrix: $\boldsymbol{\Lambda} = \boldsymbol{\Sigma}^{-1}$.

Density estimation

Our objective is being able to estimate the parameters of a distribution.

Let's start with a simple example: we want to understand if a coin is biased or not. So we toss it n times and see the results.

So we want to predict our next toss given our past tosses: $X_t | X_1 = x_1 \dots X_n = x_n$

We will use the i.i.d. Hypothesis so we can model our coin as a Bernoulli distribution with parameter π.

Now, how do we estimate π? Again we need to observe our previous tosses.
We use the **frequentist** approach: we predict there is a true value π^T and we try to get as close as possible by estimating it.

So we create our **likelihood function**:

$$\mathcal{L}(\pi) = f_{X_1 \dots X_n | \pi}(x_1 \dots x_n | \pi) = P(X_1 = x_1 \dots X_n = x_n | \pi)$$

It's the probability of our observed tosses given the parameter π. This function gives us a value of how much our observed results are possible given the parameter. So if we saw more head than tail we will estimate a greater value for π.

By the i.i.d. Hypothesis:

$$P(X_1 = x_1 \dots X_n = x_n | \pi) = \prod_{i=1}^{n} P(X_i = x_i | \pi)$$

And since we said we could model P as a bernoulli distribution:

$$P(X_i = x_i | \pi) = \pi^{x_i} (1 - \pi)^{1 - x_i}$$

So:

$$\mathcal{L}(\pi) = \prod_{i=1}^{n} \pi^{x_i} (1 - \pi)^{1 - x_i}$$

Here we can introduce the **maximum likelihood estimate** where to estimate the parameters we find the values that maximize our likelihood function. So in this context our ML is:

$$\arg\max_{\pi} \prod_{i=1}^{n} \pi^{x_i}(1-\pi)^{1-x_i}$$

Also we will usually not work with the likelihood function but with the logarithm likelihood function, because it will simplify our expressions.

Doing the math (we take the derivative and we set it to zero) for this situation we get:

$$\pi^*_{ML} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

So to estimate our π we take the mean.
Our frequentist approach gives us an important property: as long as the number of tosses grows the parameter π will get closer and closer to the real parameter π^T.

For modeling **continuous** values we will use the **gaussian hypothesis**, meaning we will look at our data as distributed by a gaussian distribution and we will estimate the parameters μ and v.

Given some data $\mathcal{D} = (x_1, \ldots, x_n)$ we can express our **likelihood** as:

$$\mathcal{L}(\theta) = \prod_{i=1}^{n} \mathcal{N}(x_i|\mu, v)$$ where $\theta = (\mu, v)$

Using our i.i.d. Hypothesis and frequentist approach.

In general the function that mapping a dataset D to a set of model parameters is called an **estimator**: $\theta^* = T(\mathcal{D})$.

We want a **consistent** estimator: those tend to converge to the true distribution parameters as the size of n grows. The ML estimator is a consistent one.

Another approach to produce a consistent estimator is using the **method of moments**: matching the moments of the assumed distribution to those of the data. In this case we set the mean and the variance of our model to the mean and the variance of our dataset. As long as our dataset is well behaved (like the number of outliers is low) this will work.

For the ML approach we will maximize our likelihood function:

$$\theta^*_{ML} = \arg\max_{\theta} \mathcal{L}(\theta)$$

And also in this context we will use the log likelihood:

$$\ell(\theta) = \sum_{i=1}^{n} \log \mathcal{N}(x|\mu, \lambda^{-1})$$

By the property of logarithm we know that:

$$\arg\max_{\theta} \ell(\theta) = \arg\max_{\theta} \mathcal{L}(\theta)$$

So maximizing the log likelihood is equivalent to maximizing the original likelihood.

As we did before we can set derivative of the log likelihood to zero and do it for the two parameters to get:

$$\mu^*_{ML} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

$$v^*_{ML} = \frac{1}{n} \sum_{i=1}^{n} (x_i - \mu^*_{ML})^2$$

The solution thus corresponds to the **empirical mean** and **empirical covariance matrix** of the data.
In this case, we can observe that the solution is the same as the one obtained by the MOM approach, because the model parameters are matching the moments of the dataset.

And so after observing n data we can estimate the probability of getting a value as:

$$f_{X_t|X_1 \ldots X_n}(x_t|x_1 \ldots x_n) \approx \mathcal{N}(x_t \,|\mu^*_{ML}, v^*_{ML})$$

Generative classifier

We consider a closed set problem, so we want to assign a sample to one of 1…k classes.

We will employ a **probabilistic model** meaning we sample x comes from a random variable Xt and its class is a realization of the r.v. Ct.

Optimal bayes decision: assign the class that maximize the posterior probability:

$$c^*_t = \arg\max_c P(C_t = c|\boldsymbol{X}_t = \boldsymbol{x}_t)$$

Given a sample xt we compute $P(C_t = c|\boldsymbol{X}_t = \boldsymbol{x}_t)$ and we choose the class that maximizes the probability.

We just consider the samples as independent and distributed according to a single random variable X.

Also we can introduce the bayes rule to compute our **posterior** probability as:

$$P(C_t = c|\boldsymbol{X}_t = \boldsymbol{x}_t) = \frac{f_{\boldsymbol{X},C}(x_t, c)}{\sum_{c' \in \mathcal{C}} f_{\boldsymbol{X},C}(\boldsymbol{x}_t, c')}$$

And since the bottom part doesn't depend on the class we will just focus on the **joint** probability.

The joint can be expressed as:

$$f_{\boldsymbol{X},C}(\boldsymbol{x}_t, c) = f_{\boldsymbol{X}|C}(\boldsymbol{x}_t|c)P_C(c)$$

The **class conditional probability** multiplied by the **application prior probability**.

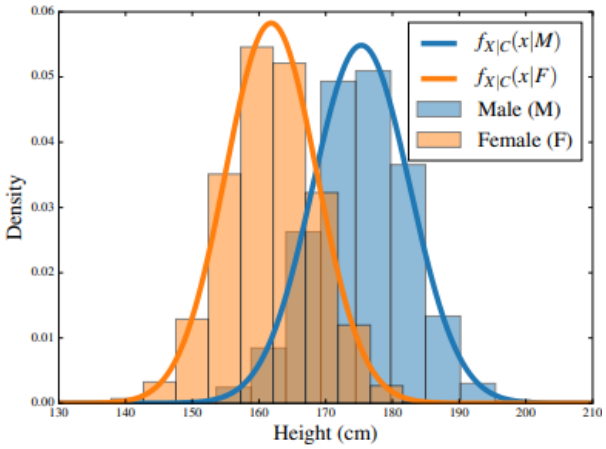
The application prior is something that is given when we apply the model to represent how much we believe the sample belongs to the class even before seeing the sample. (example if we want to predict the weather based on some data and we are in the desert the prior for the class "raining" will be very low).

Our classifier will model the class conditional distribution.

Gaussian classifiers

A subset of Generative models where we suppose that the class conditional probability can be modeled by a gaussian distribution.

For each class we can estimate a gaussian distribution by using the ML estimate. An example using heights of male and female:



Now we can compute the posterior as:

$$P(C = M|X = 174) = \frac{f_{X|C}(174|M)P(C = M)}{f_X(174)}$$

$$P(C = F|X = 174) = \frac{f_{X|C}(174|F)P(C = F)}{f_X(174)}$$

Note that comparison wise the computation of $f_X(174)$ is useless since it doesn't depend on the class.

We can also compute the **class posterior ratio** as:

$$\frac{P(C=M|X=174)}{P(C=F|X=174)} = \frac{f_{X|C}(174|M)}{f_{X|C}(174|F)} \frac{P(C=M)}{P(C=F)}$$

Note that if the two classes have the same prior of 0.5 then this would compute the class condition ratio.

In a more general case, we are seeing our data as distributed as a multivariate gaussian distribution, so for each class:

$(\boldsymbol{X}|C=c) \sim \mathcal{N}(\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)$ One mean and one covariance matrix per class.

Given the parameters we can compute: $f_{\boldsymbol{X}|C}(\boldsymbol{x}_t|c) = \mathcal{N}(\boldsymbol{x}_t|\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)$.

We need to estimate our parameters $\boldsymbol{\theta} = [(\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1) \ldots (\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)]$ given our labeled training dataset.

We keep using our likelihood function:

$$\begin{aligned}
\mathcal{L}(\boldsymbol{\theta}) &= \prod_{i=1}^{n} f_{\boldsymbol{X},\boldsymbol{C}|\boldsymbol{\theta}}(\boldsymbol{x}_i, c_i|\boldsymbol{\theta}) \\
&= \prod_{i=1}^{n} f_{\boldsymbol{X}|C,\boldsymbol{\theta}}(\boldsymbol{x}_i|c_i, \boldsymbol{\theta}) P(c_i) \\
&= \prod_{i=1}^{n} \mathcal{N}\left(\boldsymbol{x}_i|\boldsymbol{\mu}_{c_i}, \boldsymbol{\Sigma}_{c_i}\right) P(c_i)
\end{aligned}$$

And we will choose the parameters that maximize the log likelihood.

$$\ell(\boldsymbol{\theta}) = \log \mathcal{L}(\boldsymbol{\theta})$$
$$= \sum_{i=1}^{n} \log \mathcal{N}\left(\boldsymbol{x}_i | \boldsymbol{\mu}_{c_i}, \boldsymbol{\Sigma}_{c_i}\right) + \sum_{i} \log P(c_i)$$
$$= \sum_{c=1}^{k} \sum_{i|c_i=c} \log \mathcal{N}\left(\boldsymbol{x}_i | \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c\right) + \xi$$

We can see that log likelihood is a sum of sums (instead of summing for all samples we sum for each class c the i samples that belong to that class). We collected in ξ the value of the sum of the prior because that doesn't depend on the parameters.

We can thus rewrite the log likelihood as:

$$\ell(\boldsymbol{\theta}) = \sum_{c=1}^{k} \ell_c(\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) + \xi \ , \quad \ell_c(\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) = \sum_{i|c_i=c} \log \mathcal{N}\left(\boldsymbol{x}_i | \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c\right)$$

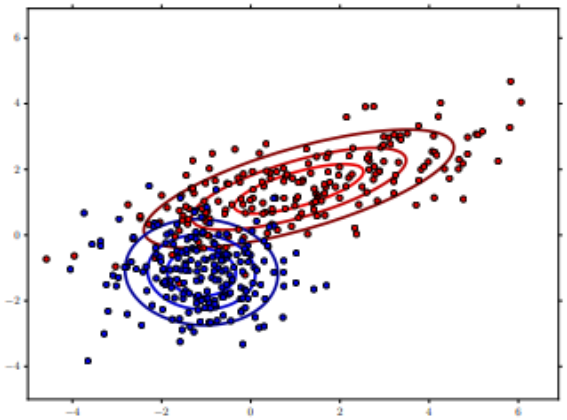
So maximizing the log likelihood for the dataset is equivalent to maximizing independently all the terms $\ell_c(\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)$.

But those terms are simply the log-likelihood of a Gaussian model for the data of class c.
For multivariate Gaussian models we can show that we can estimate our parameters by:

$$\boldsymbol{\mu}_c = \frac{1}{N_c} \sum_{i|c_i=c} \boldsymbol{x}_i$$

$$\boldsymbol{\Sigma}_c = \frac{1}{N_c} \sum_{i|c_i=c} (\boldsymbol{x}_i - \boldsymbol{\mu}_c)(\boldsymbol{x}_i - \boldsymbol{\mu}_c)^T$$

Here an example of two MVG estimated over a dataset of two classes:



Binary classification

We will denote the two classes with h1 and h0. Now that we know how to compute our gaussian parameters we can analyze again our class posterior ratio:

$$r(\boldsymbol{x}_t) = \frac{P(C = h_1|\boldsymbol{x}_t)}{P(C = h_0|\boldsymbol{x}_t)}$$

If r(x) >= 1 the class is assigned to h1, otherwise to h0.
We will move to a logarithmic ratio for two reasons:

- The domain are now from (-∞, 0) and (0, +∞)
- We can separate likelihood and prior probability.

$$\log r(\boldsymbol{x}_t) = \log \frac{f_{X|C}(\boldsymbol{x}_t|h_1)P(C = h_1)}{f_{X|C}(\boldsymbol{x}_t|h_0)P(C = h_0)}$$
$$\log \frac{f_{X|C}(\boldsymbol{x}_t|h_1)}{f_{X|C}(\boldsymbol{x}_t|h_0)} + \log \frac{P(C = h_1)}{P(C = h_0)}$$

The sum of the **log likelihood ratio** and the **prior probability ratio**.
Also since we are in a binary problem we can define the probability of class h1 as pi

$$P(C = h_1) = \pi \ , \quad P(C = h_0) = 1 - P(C = h_1) = 1 - \pi$$

And so our ratio becomes:

$$\log r(\boldsymbol{x}_t) = \log \frac{f_{X|C}(\boldsymbol{x}_t|h_1)}{f_{X|C}(\boldsymbol{x}_t|h_0)} + \log \frac{\pi}{1 - \pi}$$

Note that if pi is 0.5 then our prior prob ratio becomes zero.

Since our comparison is:

$$\log r(\boldsymbol{x}_t) \gtrless 0$$

It becomes:

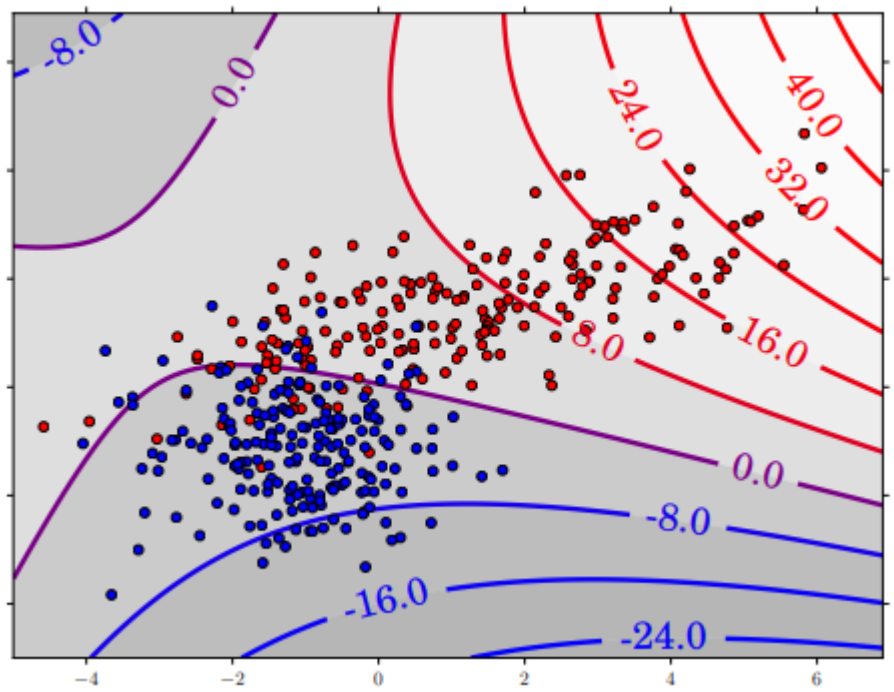
$$\log \frac{f_{X|C}(\boldsymbol{x}_t|h_1)}{f_{X|C}(\boldsymbol{x}_t|h_0)} \gtrless -\log \frac{\pi}{1 - \pi}$$

Our log likelihood ratio will provide a **score**, where greater positive values will favor h1 while greater negative values will favor h0. Our prior provides a **threshold** to compare our score and decide the class. Later we will see how to construct an optimal threshold given different application dependent parameters.

Lastly we can compute the loglike hood and notice that it has the corresponding form:

$$llr(\boldsymbol{x}) = \boldsymbol{x}^T \boldsymbol{A}\boldsymbol{x} + \boldsymbol{x}^T \boldsymbol{b} + c$$

(we don't care about A and b, they just depend on the means and the covariance matrices)
The first term is a quadratic term, because we have x multiplied by itself.
So this can create decision boundaries with the form of ellipsis or parables.
Here is an example just considering llr, so pi = 0.5.



In case we want to generalize to k classes problems we can't use the ratio anymore, but we can still choose the class with highest posterior probability:

$$c_t^* = \arg\max_h f_{\boldsymbol{X}|C}(\boldsymbol{x}_t|h)P(h) = \arg\max_h \log f_{\boldsymbol{X}|C}(\boldsymbol{x}_t|h) + \log P(h)$$

The Gaussian classifiers have a major problem:

- We split our dataset in classes, so few samples lead to inaccurate parameters. This is more evident with covariance matrices.

So we see two Gaussian classifiers with more hypothesis:
Naive bayes: we suppose that all the components/features of the datasets are all **independent** for all the classes. Doing so in general the density of a sample becomes:

$$f_{\boldsymbol{X}|C}(\boldsymbol{x}|c) \approx \prod_{j=1}^{D} f_{X_{[j]}|C}(x_{[j]}|c)$$

Where D is the number of features.
So if x is a vector of 3 components we can compute the density as f(x[1]|c)*f(x[2]|c)*f(x[3]|c).

In the context of the Gaussian this means we are computing D gaussian single variable using the mean and the variance of component j:

$$f_{X_{[j]}|C}(x_{[j]}|c) = \mathcal{N}(x_{[j]}|\mu_{c,[j]}, \sigma^2_{c,[j]})$$

So we don't need anymore a complete covariance matrix, just the diagonal of that matrix (according to the naive assumption the other values are zero).

$$\boldsymbol{\Sigma}_c = \begin{bmatrix} \sigma^2_{c,[1]} & 0 & \dots & 0 \\ 0 & \sigma^2_{c,[2]} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sigma^2_{c,[D]} \end{bmatrix}$$

Using the log likelihood our ML solution is:

$$\mu^*_{c,[j]} = \frac{1}{N_c} \sum_{i|c_i=c} x_{i,[j]} \ , \quad \sigma^2_{c,[j]} = \frac{1}{N_c} \sum_{i|c_i=c} (x_{i,[j]} - \mu_{c,[j]})^2$$

And to compute the density of a sample:

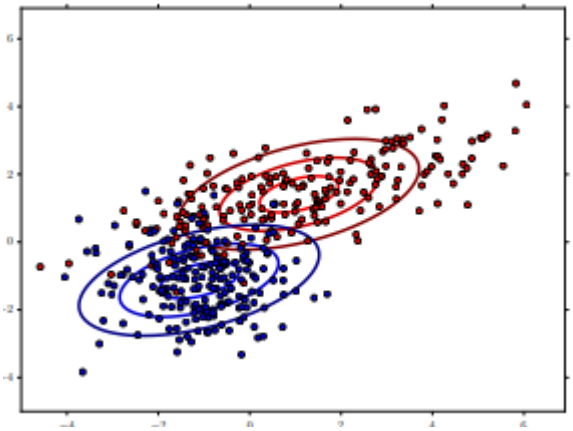
$$f_{\boldsymbol{X}|C}(\boldsymbol{x}|c) = \prod_{j=1}^{D} \mathcal{N}(x_{[j]}|\mu^*_{c,[j]}\sigma^2_{c,[j]})$$

Note that now we don't have just a mean and a covariance matrix per class but a mean and a variance for each class and component.

The only problem with this model is when the naive assumption is weak for the dataset (data components are highly correlated): in that case the performance of our classifier may be terrible.

The third classifier is the **tied classifier**: here we assume that the covariance matrices are similar for all classes.
We suppose class independent noise: for each class there is a different mean but the distribution of data across the mean is the same for each class. Hence we can compute a single covariance matrix and use it for all classes.

Here blue and red have different means but the way the data is spread is the same:



By the ML solution (class mean is the same):

$$\Sigma^* = \frac{1}{N} \sum_c \sum_{i|c_i=c} (x_i - \mu_c)(x_i - \mu_c)^T$$

Two problems here:

- If the assumption is wrong the classifier will make a lot of mistakes.
- The log likelihood ratio provides **linear** boundaries, not quadratic anymore. For some distributions of data this will be problematic.

We can also create a tied naive gaussian classifier, where we have a **single diagonal** covariance matrix for all classes.

There's a strong connection between LDA and tied gaussian: with LDA we computed the within class covariance Sw for **all samples** (we supposed one for all classes). Then we whitened it. For the tied gaussian classifier this means that for the covariance matrix that we will compute will be the identity (because we whitened our data in the LDA algorithm). Also if we classify using LDA for the binary case we get the same formula of the log likelihood ratio of the tied classifier.

Discrete values modeling

Up to this point we have used continuous values for our components, but we need to address the problem of modeling discrete values.

For now let's consider the problem of modeling a single discrete set of values to a label (later we will generalize to random vectors).

We still need a training set, here it's a couple composed of a categorical variable and a class:

$$\mathcal{D} = \{(x_1, c_1) \dots (x_n, c_n)\}$$

We can't model anymore using gaussian classifiers, because we would be using a continuous model for discrete values.

What we need to compute is $P(X_t = x_t | C_t = c) = \pi_{c,x_t}$

We will define π_{c,x_t} as the probability of value xt given the class c.

We will also define the set $\boldsymbol{\pi}_c = (\pi_{c,1} \dots \pi_{c,m})$ as the model parameters of class c. Since the probability of observing any value given a class is 1 we have the constraint for each class:

$$\sum_{i=1}^{m} \pi_{c,i} = 1$$

We can still apply our ML approach by defining the likelihood function:

$$\mathcal{L}(\boldsymbol{\Pi}) = \prod_{i=1}^{n} P(X_i = x_i | C_i = c_i) P(C_i = c_i)$$

(Note that we are using the bayes theorem, so we are still using a Generative model)

As usual we work with log likelihood:

$$\ell(\boldsymbol{\Pi}) = \sum_{i=1}^{n} \log P(X_i = x_i | C_i = c_i) + \xi$$
$$= \sum_{i=1}^{n} \log \pi_{c_i,x_i} + \xi$$

And summing for each class and for the data in that class:

$$= \sum_{c=1}^{k} \sum_{i|c_i=c} \log \pi_{c,x_i} + \xi$$
$$= \sum_{c=1}^{k} \ell_c(\boldsymbol{\pi}_c) + \xi$$

Note that also here we used ξ for the useless value of the priors.

Also here we can maximize our log likelihood as a maximization of different terms:

$$\ell_c(\boldsymbol{\pi}_c) = \sum_{i|c_i=c} \log \pi_{c,x_i}$$

We can observe that if some samples have the same value for the variable x and same class we will sum the value of $log\ \pi_{c,\,x_i}$ just more time. We can thus collect the term and multiply by their number of appearances.

So if we define $N_{c,j}$ the number of times we observe x = j in class c we can define the log likelihood as:

$$= \sum_{j=1}^{m} N_{c,j} \log \pi_{c,j}$$

Subject to:

$$\sum_{i=1}^{m} \pi_{c,i} = 1$$

For solving this problem we also use the Lagrange multipliers (not really useful what's that) for taking into account the constraint. Solving we get:

$$\pi^*_{c,i} = \frac{N_{c,i}}{N_c}$$

Meaning that for the probability of getting i given c we just divide the number of elements in class c by the number of samples of class c where x = i.

We now need to generalize our model: instead of just one categorical attribute, we may have a number of categorical attributes greater than one.

We have two ways to model this:

- We compute for each class the probability of getting every possible combination.
- We use a **Naive assumption** and we consider the attributes as independent. We can thus compute the probability of a sample given a class by computing the probability of a single attribute to have a value given the class.

Let's imagine we have M features with N possible values. With the first method we have to compute the probability for N^M possible combinations. This is too heavy to compute and requires very large training data.

With the second approach we can compute $\pi_{c,i,[j]}$ which is the same computation we did before, just done for each attribute. So we can compute:

$$P(\boldsymbol{X}_t = \boldsymbol{x}_t | C_t = c) = \prod_j \pi^j_{c,x_{t,[j]}}$$

We compute the probability of the sample for each attribute and then we multiply.

Another extension is where features represent occurrences of events. An example is a random vector representing a book and each component counting the number of occurrences of a word. Note that in this context two documents with the same exact number of occurrences for each word but different order will produce the same random vector.

We already have a distribution to model the occurrences of categorical events, **the multinomial distribution**.

$$P(\boldsymbol{X} = \boldsymbol{x} | C = c) = \frac{\left(\sum_{j=1}^m x_{[j]}\right)!}{\prod_{j=1}^m x_{[j]}!} \prod_{j=1}^m \pi_{c,j}^{x_{[j]}} \propto \prod_{j=1}^m \pi_{c,j}^{x_{[j]}}$$

We will use just the second equation since the first term doesn't depend on c (so for our computation we can remove it without losing information).

Bu computing the log likelihood we get:

$$\ell_c(\boldsymbol{\pi}_c) = \sum_{j=1}^m N_{c,j} \log \pi_{c,j}$$

That we have already solved! The solution is again:

$$\pi_{c,j} = \frac{N_{c,j}}{N_c}$$

Where Nc is the total number of words for class c and Nc,j is the number of occurrences of event j in the class c.

Last thing: if we use this probability to compute logarithmic likelihood ratio we can an expression like:

$$llr(\boldsymbol{x}) = \log \frac{P(\boldsymbol{X} = \boldsymbol{x} | C = h_1)}{P(\boldsymbol{X} = \boldsymbol{x} | C = h_0)} = \boldsymbol{x}^T \boldsymbol{b}$$

So we have a linear decision function.

There is a problem that we need to take is the situation where a word never occurred in a document: in this case $\pi_{c,i,[j]} = 0$. Since we compute the probability using a multiplication this will return a probability of zero (in case the sample contains that word). In the very rare case where never appears in all classes all the probabilities will be zero! We will add the **pseudo-count**, so for each word we have a fixed number (since it is the same for all words for all classes this won't affect our results).

Logistic Regression

The logistic regression is a **discriminant** classifier: we will directly compute the posterior probability: $P(C = c|\boldsymbol{X} = \boldsymbol{x})$.

The logistic regression is developed **starting from the tied gaussian classifier**. This aspect will be discussed later. Also this works for **binary problems**.

So we compute the posterior ratio as:

$$\log \frac{P(C = h_1|\boldsymbol{x})}{P(C = h_0|\boldsymbol{x})} = \log \frac{f_{\boldsymbol{X}|C}(\boldsymbol{x}|h_1)}{f_{\boldsymbol{X}|C}(\boldsymbol{x}|h_0)} + \log \frac{\pi}{1-\pi} = \boldsymbol{w}^T\boldsymbol{x} + b$$

Where w is the model (composed of two means and the covariance matrix) and b is the posterior probability.

Since

$$\log \frac{P(C = h_1|\boldsymbol{x})}{P(C = h_0|\boldsymbol{x})} = \boldsymbol{w}^T\boldsymbol{x} + b$$

We can raise to e and get

$$\frac{P(C = h_1|\boldsymbol{x})}{P(C = h_0|\boldsymbol{x})} = e^{\boldsymbol{w}^T\boldsymbol{x} + b}$$

We can thus solve it for P(h1|x)

$$\begin{aligned} P(C = h_1|\boldsymbol{x}, \boldsymbol{w}, b) &= e^{(\boldsymbol{w}^T\boldsymbol{x}+b)} P(C = h_0|\boldsymbol{x}, \boldsymbol{w}, b) \\ &= e^{(\boldsymbol{w}^T\boldsymbol{x}+b)}(1 - P(C = h_1|\boldsymbol{x}, \boldsymbol{w}, b)) \end{aligned}$$

And we get

$$P(C = h_1 | \boldsymbol{x}, \boldsymbol{w}, b) = \frac{1}{1 + e^{-(\boldsymbol{w}^T \boldsymbol{x} + b)}}$$

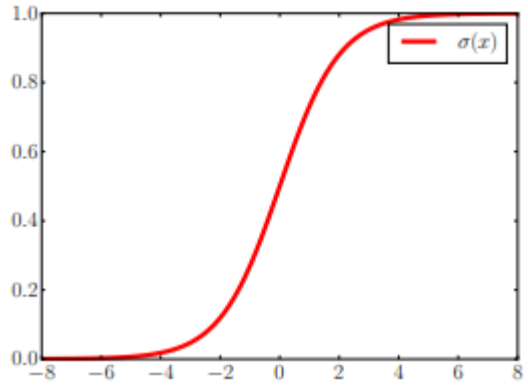
This function is an application of the **sigmoid function**.

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

So

$$P(C = h_1 | \boldsymbol{x}, \boldsymbol{w}, b) = \sigma(\boldsymbol{w}^T \boldsymbol{x} + b)$$

The sigmoid function has this shape:



The more the value returned from the model plus the prior is far from zero the higher the probability of classifying it to h1.

Also: $1 - \sigma(x) = \sigma(-x)$ So to compute the probability of h0 we compute the probability of h1.

But how do we compute w and b? We **don't** use the model of the tied gaussian (that was just an assumption to derive the model), but we will compute our w and b by using a frequentist approach, using a discriminant approach.

Given a train set:

$$\mathcal{D} = [(\boldsymbol{x}_1, c_1), \ldots (\boldsymbol{x}_n, c_n)]$$

We can use the i.i.d. assumption and create a likelihood function:

$$\prod_{i=1}^{n} P(C_i = c_i | \boldsymbol{x}_i, \boldsymbol{w}, b)$$

We can assume that h1 is mapped to the number 1 and h0 to the number 0.

By defining $$y_i = P(C_i = 1|\boldsymbol{x}_i, \boldsymbol{w}, b) = \sigma(\boldsymbol{w}^T \boldsymbol{x}_i + b)$$
It's easy to check that our posterior probability is a Bernoulli distribution (**if we know the class of the sample**):

$$P(C_i = c_i|\boldsymbol{x}_i, \boldsymbol{w}, b) = y_i^{c_i}(1 - y_i)^{(1-c_i)}$$

We also use the fact that the probability of h0 can be computed as 1 - probability of h1.

So our likelihood becomes:

$$= \prod_i y_i^{c_i}(1 - y_i)^{(1-c_i)}$$

And by using the log likelihood:

$$\ell(\boldsymbol{w}, b) = \log \mathcal{L}(\boldsymbol{w}, b) = \sum_{i=1}^{n} [c_i \log y_i + (1 - c_i) \log(1 - y_i)]$$

So our goal becomes the maximization of the log likelihood:

$$\arg\max_{\boldsymbol{w},b} \sum_{i=1}^{n} [c_i \log y_i + (1 - c_i) \log(1 - y_i)]$$

This expression is interesting because it give us some interesting properties that we need to discuss:

The first one is by noting that we can **minimize** the negation of the function:

$$J(\boldsymbol{w}, b) = -\ell(\boldsymbol{w}, b) = \sum_{i=1}^{n} - [c_i \log y_i + (1 - c_i) \log(1 - y_i)]$$

This is called the **cross-entropy** between the predicted and the observed labels.
What does it mean? Let's suppose we have defined two recognizer (equivalent term for classifier):

- R is the recognizer based on our prediction, so it classifies label according to:

$$P\left(C_i = 1|\boldsymbol{X}_i = \boldsymbol{x}_i, \mathcal{R}(\boldsymbol{w}, b)\right) = y_i = \sigma(\boldsymbol{w}^T \boldsymbol{x}_i + b)$$

- $\mathcal{E}$ is the recognizer that know the real label of the sample, it's a theoretical recognizer that we can't have but that represent the best result we can have:

$$P(C_i = 1|\boldsymbol{X}_i = \boldsymbol{x}_i, \mathcal{E}) = \begin{cases} 1 & \text{if } c_i = 1 \\ 0 & \text{if } c_i = 0 \end{cases}$$

The cross-entropy is the difference between these two recognizers. Our model would be perfect when R = Ɛ. Since this is impossible we can just try to minimize the difference, but our log likelihood function is already doing that! So we are maximizing our likelihood and also minimizing the difference with a theoretical perfect model.

Minimization of the average cross-entropy means we are looking for a label distribution that is (on average) as similar as possible to the empirical/theoretical one.

Another property can be found with some rewriting of our formula:
Let our objective function (the version that we minimize) be

$$J(\boldsymbol{w}, b) = \sum_i H(c_i, y_i)$$

where

$$H(c_i, y_i) = -\left[c_i \log y_i + (1 - c_i) \log(1 - y_i)\right]$$

And we rewrite the class as:

$$z_i = \begin{cases} 1 & \text{if } c_i = 1 \\ -1 & \text{if } c_i = 0 \end{cases}$$

And let $s_i = \boldsymbol{w}^T \boldsymbol{x}_i + b.$ so that $y_i = \boldsymbol{\sigma}(s_i)$

So for each sample the objective function becomes

$$\begin{aligned} H(c_i, y_i) &= -\left[c_i \log y_i + (1 - c_i) \log(1 - y_i)\right] \\ &= \begin{cases} -\log \sigma(s_i) & \text{if } c_i = 1 \ (z_i = 1) \\ -\log\left(1 - \sigma(s_i)\right) = -\log \sigma(-s_i) & \text{if } c_i = 0 \ (z_i = -1) \end{cases} \end{aligned}$$

For the two blue terms the only difference is the negation sign, which is equal to zi. So these can be rewritten as:

$$= -\log \sigma(z_i s_i)$$

Replacing si with the original term we get:

$$= -\log \sigma\left(z_i(\boldsymbol{w}^T \boldsymbol{x}_i + b)\right)$$

$$= \log\left(1 + e^{-z_i(\boldsymbol{w}^T \boldsymbol{x}_i + b)}\right)$$

(this last expression comes from applying the sigma function and some rewriting using some logarithm properties).

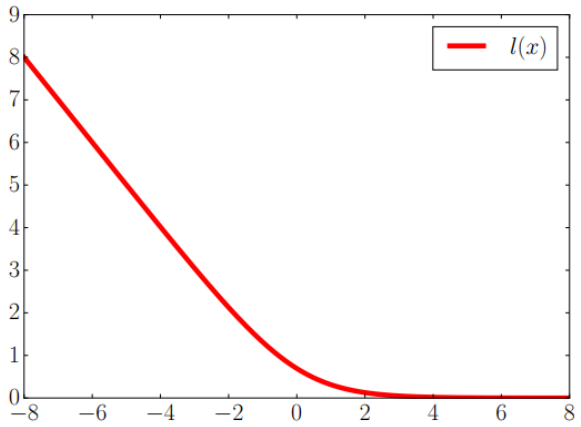
This is an application of the **logistic loss function**:

$$l(x) = \log\left(1 + e^{-x}\right)$$

And our objective function becomes:

$$\begin{aligned} J(\boldsymbol{w}, b) &= \sum_{i=1}^{n} H(c_i, y_i) \\ &= \sum_{i=1}^{n} \log\left(1 + e^{-z_i(\boldsymbol{w}^T \boldsymbol{x}_i + b)}\right) \\ &= \sum_{i=1}^{n} l\left(z_i(\boldsymbol{w}^T \boldsymbol{x}_i + b)\right) \end{aligned}$$

A minimization of logistic loss.
How does it work? Let's see the logistic loss function:



For positive numbers it immediately approaches zero, while for negative numbers we get a positive number.

Using again $s_i = \boldsymbol{w}^T \boldsymbol{x}_i + b$, we see that we are computing: $l(z_i s_i)$ for each train sample.
Let's assume the class for xi is 1 (so zi = 1), there are two situations:

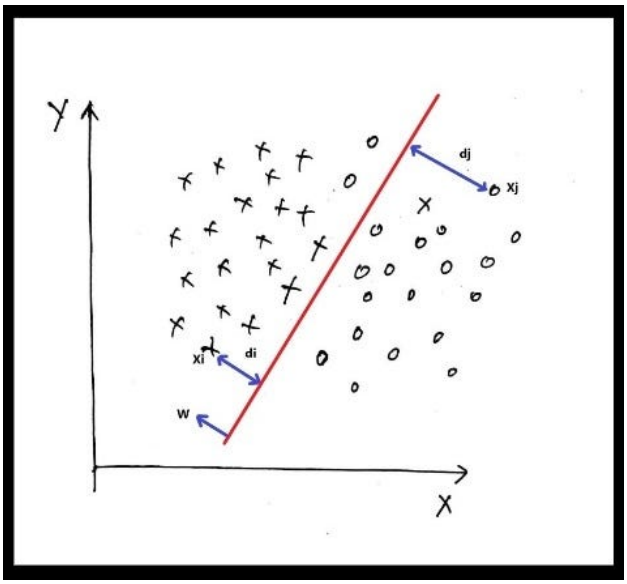
- If the model is correct si is giving a positive score, so zi*si = a positive number. You can check that for positive numbers l(N) is returning zero.
- If the model is wrong it will classify x as belonging to class zero so it will return a negative score. So zi*si is a negative score -N and l(-N) is returning a positive number, which represents our error. We can easily simplify our expression and see that the error grows linearly with the score si, so the greater the wrong score the higher the error (the model is sure about something wrong, that's bad).
- Also if the prediction is correct but the score is low the loss function returns a positive value (because the model is not really sure about the prediction).

So we can define the **empirical risk** as the sum of the logistic loss for all samples. We can see that the minimized objective function is equivalent to the empirical risk.

So our objective function is doing three things in one:

- Maximizing the log likelihood.
- Minimizing the cross-entropy with the theoretical model.
- Minimize the empirical risk.

We can see geometrically si as the distance between x and the vector w, readjusted according to our prior b:



There's a problem with our current objective function: if we are in a situation where our training set is **linearly separable** (it means that we can find w such that it perfectly separates our dataset), then our optimization of the objective function will never stop! Why is that?
Since our logistic error function returns a value closer to zero as the score gets bigger, we can increase the norm of w. This increases our score and lowers the empirical risk. But since the logistic function will never return a positive value (the classes are linearly separable) this process can go on forever.

To make the problem solvable again we introduce a **regularization** term: we introduce a norm penalty such that a bigger norm brings a lower result for the objective function.
We also have to note that maybe our training data set is linearly separable so that this won't hold for the test set, so we are avoiding overfitting.

The new objective function thus becomes:

$$\tilde{R}(\boldsymbol{w}, b) = \frac{\tilde{\lambda}}{2}\|\boldsymbol{w}\|^2 + \sum_{i=1}^{n} \log\left(1 + e^{-z_i(\boldsymbol{w}^T \boldsymbol{x}_i + b)}\right)$$

Where with λ we specify the weight of the regularization term (higher lambda means that the algorithm will immediately reject increases of the norm of w, but we may suffer from underfitting). It's a hyperparameter.

We can use some **pre-processing** since our regularized version of the model is not invariant to linear transformation (so the same data but centered or with the whitened covariance matrix will give different results):

- Centering our data.
- Whiten the covariance matrix (it becomes the identity).
- L2 (or length) normalization: divides every data with its norm, so all vectors/data have length 1.

Which is better for our dataset? We need cross-validation!

And what about the prior probabilities?
Since we are using an algorithm to compute both w and b we are not including anymore the prior probabilities.
A way to reintroduce the prior probabilities is to modify our objective function such that the empirical risk is weighted by the given prior:

$$R(w) = \frac{\lambda}{2}\|\boldsymbol{w}\|^2 + \frac{\pi_T}{n_T}\sum_{i|z_i=1} l(z_i s_i) + \frac{1-\pi_T}{n_F}\sum_{i|z_i=-1} l(z_i s_i)$$

Problem: we are using the prior probability for the training and not for the classification phase. So in case we have a new application that requires a new prior we should retrain our model to use the new prior.

Multiclass logistic regression

We now need to generalize our model to classify in a multiclass context. We will work with K classes, each one mapped to a number from 1 to K.

We will start again from the tied Gaussian classifier by defining:

$$\log \frac{P(C=j|\boldsymbol{x})}{P(C=r|\boldsymbol{x})} = (\boldsymbol{w}_j - \boldsymbol{w}_r)^T\boldsymbol{x} + (b_j - b_r)$$

We don't care about the values of the two w or b, we just care that the ratio can be rewritten with this form. For each class we will derive a w and b.

Note that this also is consistent with:

$$\log \frac{P(C=r|\boldsymbol{x})}{P(C=r|\boldsymbol{x})} = 0$$

We will use class r as a reference class, we don't care which class we just choose one to do our mathematical derivation.

By doing the trick we did for the logistic regression (raising to the power of e and then multiply by the divisor) we get that for any class j in K:

$$P(C=j|\boldsymbol{x}) = P(C=r|\boldsymbol{x})e^{(\boldsymbol{w}_j-\boldsymbol{w}_r)^T\boldsymbol{x}+(b_j-b_r)}$$

One property we have that given an x the sum of the classes is 1:

$$\sum_{j=1}^{K} P(C=j|\boldsymbol{x}) = 1$$

So for any class (here we will use the reference class r):

$$P(C=r|\boldsymbol{x}) = 1-\sum_{j\neq r} P(C=j|\boldsymbol{x})$$

But we already computed the probability of any class with respect to r:

$$P(C=r|\boldsymbol{x}) = 1-\sum_{j\neq r} P(C=r|\boldsymbol{x})e^{(\boldsymbol{w}_j-\boldsymbol{w}_r)^T\boldsymbol{x}+(b_j-b_r)}$$

Solving by the posterior prob of r we get:

$$P(C=r|\boldsymbol{x}) = \frac{1}{1+\sum_{j\neq r} e^{(\boldsymbol{w}_j-\boldsymbol{w}_r)^T\boldsymbol{x}+(b_j-b_r)}}$$

And it can be shown that this is:

$$= \frac{e^{\boldsymbol{w}_r^T\boldsymbol{x}+b_r}}{\sum_j e^{\boldsymbol{w}_j^T\boldsymbol{x}+b_j}}$$

This computation can be done for <u>any</u> class and so we get that for any class k:

$$P(C=k|\boldsymbol{x}) = \frac{e^{\boldsymbol{w}_k^T\boldsymbol{x}+b_k}}{\sum_j e^{\boldsymbol{w}_j^T\boldsymbol{x}+b_j}}$$

Which is an application of the **softmax function**:

$$f_i(\boldsymbol{s}) = \frac{e^{s_i}}{\sum_j e^{s_j}}$$

So for our model we need to compute **W** and **B** and W is the vector of vectors w and B is the vector of scalars b.

$$\boldsymbol{W} = \begin{bmatrix} \boldsymbol{w}_1 & \dots & \boldsymbol{w}_K \end{bmatrix}, \quad \boldsymbol{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_K \end{bmatrix}$$

Also if we consider a sample x the probability $P(C = k|\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{x})$ works like a categorical distribution $\text{Cat}(\boldsymbol{y}_i)$ (**Categorical** distribution: given a vector P of probabilities (P[i] probability of event i) returns the required probability by multiplying each probability), where

$$\boldsymbol{y}_{ik} = \frac{e^{\boldsymbol{w}_k^T \boldsymbol{x}_i + b_k}}{\sum_j e^{\boldsymbol{w}_j^T \boldsymbol{x}_i + b_j}}$$

We compute the probability given x for <u>all</u> classes and then we can compute the result yik.

This part is used for the computation of the **likelihood** for this model:
We can start by constructing the function as always

$$\ell(\boldsymbol{W}, \boldsymbol{b}) = \sum_{i=1}^{n} \log P(C_i = c_i | \boldsymbol{X}_i = \boldsymbol{x}_i, \boldsymbol{W}, \boldsymbol{b})$$

And using the categorical distribution with the log we get:

$$\log P(C_i = c_i | \boldsymbol{X}_i = \boldsymbol{x}_i, \boldsymbol{W}, \boldsymbol{b}) = \sum_{k=1}^{K} \boldsymbol{z}_{ik} \log \boldsymbol{y}_{ik}$$

Where

$$\boldsymbol{z}_{ik} = \begin{cases} 1 & \text{if } c_i = k \\ 0 & \text{otherwise} \end{cases}$$

And so our log likelihood objective function becomes:

$$\ell(\boldsymbol{W}, \boldsymbol{b}) = \sum_{i=1}^{n} \sum_{k=1}^{K} \boldsymbol{z}_{ik} \log \boldsymbol{y}_{ik}$$

Last thing we need to discuss: comparing our models we find that the logistic regression works better then the tied gaussian model (we removed the tied assumption) but is far worse than the multivariate gaussian classifier. Why is that?

➔ Since we derived the logistic regression from the tied gauss classifier, we kept the linear separation surface.

Obviously a quadratic separation is better than a linear, so the Gaussian classifier is better at separating the classes.

We have two ways to resolve this issue:

- Compute a new logistic regression starting from the multivariate and not the tied gaussian classifier (but it's time consuming and we lose some properties).
- We change the feature space such that now the linear separation becomes a quadratic separation.

We will use the second approach: we will define

$$\phi(\boldsymbol{x}) = \begin{bmatrix} \text{vec}(\boldsymbol{x}\boldsymbol{x}^T) \\ \boldsymbol{x} \end{bmatrix}$$

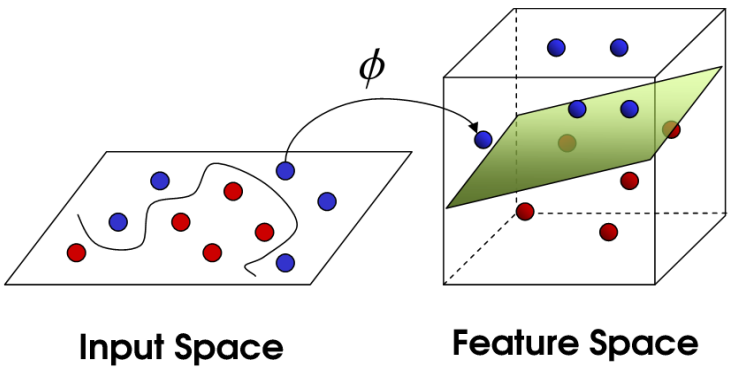
$$\boldsymbol{w} = \begin{bmatrix} \text{vec}(\boldsymbol{A}) \\ \boldsymbol{b} \end{bmatrix}$$

Where A and b come from the gaussian classifier and vec is an operator that stacks the column of a matrix. For our training purpose these are now the parameters we need to estimate.

So **each vector get mapped according to $\Phi(x)$**.
This is called the **expanded feature space**.

Our posterior then becomes: $s(\boldsymbol{x}, \boldsymbol{w}, c) = \boldsymbol{w}^T \phi(\boldsymbol{x}) + c$

Since we are linearly separating in $\Phi(x)$, which is a quadratic form of the original feature space, we are creating a separation which is quadratic in the original feature space.
We can visualize it:



For some situations the quadratic logistic regression can give better results compared to the Gaussian classifier! We just have to pay attention to the dimensionality of the new feature space: it grows exponentially with the dimensionality of the original feature space.

Model Evaluation

For this part we have two objectives:

1) Develop a way to measure the goodness of our model.
2) Compare different models to find the best one.

Let's begin with **binary** classification:
A natural solution to the first objective is to compute the **accuracy** of our model:

$$accuracy = \frac{\#\ of\ correctly\ classified\ samples}{\#\ of\ samples}$$

$$error\ rate = \frac{\#\ of\ incorrectly\ classified\ samples}{\#\ of\ samples} = 1 - accuracy$$

The problem with accuracy is that it can be misleading in case of unbalanced classes. Let's look at the problem of predicting if it's raining or not.
Our model classify as:

	Rain	Clear
Prediction: Rain	15 days	30 days
Prediction: Clear	20 days	300 days

This is a **confusion matrix**: in position [x][y] there is the number of samples classified as class x where the actual class is y. It's easy to note that to count the number of correct samples we look at the diagonal of the matrix.

In this case: $accuracy = \frac{300 + 15}{365} \approx 86\%$
Problem: if a dumb model classifies everything as Clear we get accuracy = 330/365 which is bigger compared to our model!

For this reason we will work with confusion matrix looking at each cell and computing **rates**:

	Class $\mathcal{H}_F$	Class $\mathcal{H}_T$
Prediction $\mathcal{H}_F$	True Negative	False Negative
Prediction $\mathcal{H}_T$	False Positive	True Positive

False negative rate FNR (false rejection / miss rate): $\frac{FN}{FN+TP}$

False positive rate FPR (false acceptance): $\frac{FP}{FP+TN}$

True positive rate TPR (recall, sensitivity): $\frac{TP}{FN+TP} = 1 - \text{FNR}$

True negative rate TNR (specificity): $\frac{TN}{FP+TN} = 1 - \text{FPR}$

We can see that the FNR and FPR work column-wise: they compute the error rate for each class. With the example of before our dumb classifier would have a false “clear” rate of 100%!

We can thus compute a weighted accuracy:

$$acc = \alpha FPR + (1 - \alpha)FNR$$

We have seen how different models produce some score s:

- For the generative is the log-likelihood ratio.
- For discriminants is the posterior log-likelihood ratio.
- Non-probabilistic produces a score directly (we will study SVM).

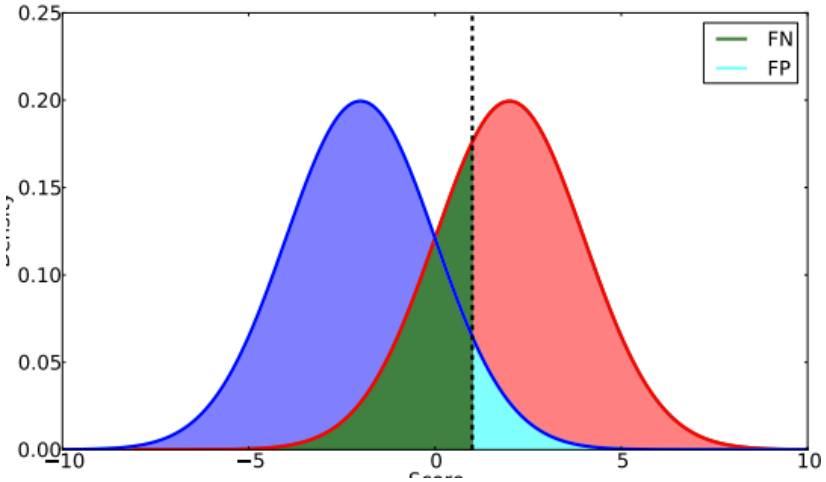
Then this score s would be compared to a threshold t to assign to a class.

$$s \geq t \longrightarrow \mathcal{H}_T$$

$$s < t \longrightarrow \mathcal{H}_F$$

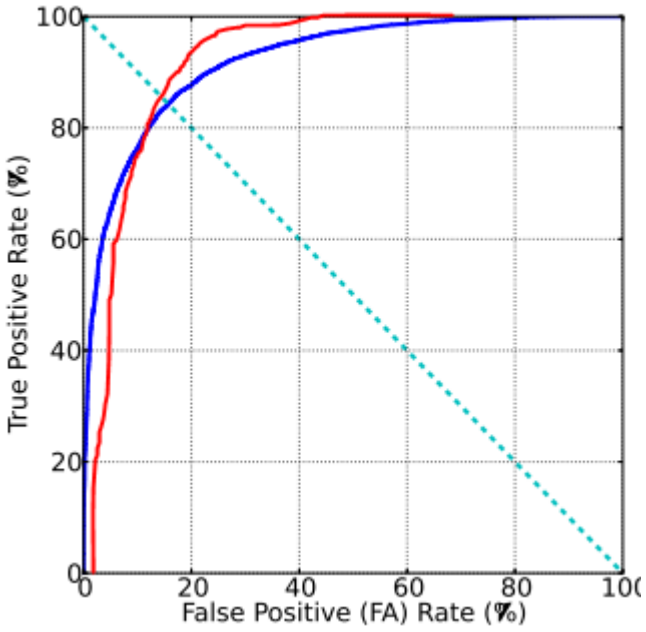
In the case of Gaussian classifiers we computed the threshold t from the prior probability.

We can easily visualize how different threshold affect the FPR and FNR:



We may want a threshold that minimizes both or we may lower one even if it means increasing the other (with the concept of error this will be more clear).

For comparing classifiers we can plot the FNR and FPR for different threshold, like the ROC curve:



The left part represents a low TPR with a low FPR, while on the right we represent a high TPR but with a high FPR (of course we would love to have high TPR and low FPR but it's unrealistic for real application).
Which of the two models is better? If we need low FPR the blue is better but if we just need high TPR we should choose the red model.

Bayes decision

The goal of our classifier is to help us to make decisions.
In the context of a hospital we can use a model to decide if to give medication or not.

In the following we consider the set of actions corresponding to labeling a sample with label a.

We can associate to each action a **cost** depending on the class of the sample:

$\mathcal{C}(a|k)$ is the cost to pay when we choose action a and the sample belongs to class k.

For us this represents the cost of labeling the sample as belonging to class a when it actually belongs to class k.

We do not know k, however we have a classifier R that allows us computing class posterior probabilities for x.

We can thus compute the **expected cost** of **action a** for sample x as:

$$\mathcal{C}_{x,\mathcal{R}}(a) = \mathbb{E}[\mathcal{C}(a|k)|x,\mathcal{R}] = \sum_{k=1}^{K} \mathcal{C}(a|k)P(C=k|x,\mathcal{R})$$

And so the **bayes decision** consists in choosing the class that minimize the expected cost:

$$a^*(x,\mathcal{R}) = \arg\min_a \mathcal{C}_{x,\mathcal{R}}(a)$$

Note that the bayes term comes from the bayes interpretation of probability: probability is a personal belief of how likely something is, but it can change the classifier! Everyone has an idea of how likely something is. So our expected cost is computed according to the probability that is returned by our classifier.

Let's see an example: suppose we are given a cost matrix and priors:

$$\boldsymbol{C} = \begin{bmatrix} 0 & 1 & 2 \\ 1 & 0 & 1 \\ 2 & 1 & 0 \end{bmatrix}, \quad \boldsymbol{\pi} = \begin{bmatrix} 0.3 \\ 0.4 \\ 0.3 \end{bmatrix}$$

And our classifier gave us this vector of posterior probabilities:

$$\boldsymbol{q}_t = \begin{bmatrix} P(C=1|\boldsymbol{x}_t,\mathcal{R}) \\ P(C=2|\boldsymbol{x}_t,\mathcal{R}) \\ P(C=3|\boldsymbol{x}_t,\mathcal{R}) \end{bmatrix} = \begin{bmatrix} 0.40 \\ 0.25 \\ 0.35 \end{bmatrix}$$

Before this chapter we would have chosen class 1 for xt but what happens if we include the cost?

$$\mathcal{C}_{\boldsymbol{x}_t,\mathcal{R}}(1) = 0 \times 0.40 + 1 \times 0.25 + 2 \times 0.35 = 0.95$$
$$\mathcal{C}_{\boldsymbol{x}_t,\mathcal{R}}(2) = 1 \times 0.40 + 0 \times 0.25 + 1 \times 0.35 = 0.75$$
$$\mathcal{C}_{\boldsymbol{x}_t,\mathcal{R}}(3) = 2 \times 0.40 + 1 \times 0.25 + 0 \times 0.35 = 1.05$$

The expected cost is lower if we assign class 2, so according to bayes decision the best class for xt is the class 2.

As already said: these results depend on the application (it gives us cost and prior) and on the classifier belief (bayes decision).

Let's go back to binary problems and let's see how we can use Bayes decision.
First of all we can construct a confusion matrix with the cost:

	Class $\mathcal{H}_F$	Class $\mathcal{H}_T$
Prediction $\mathcal{H}_F$	0	$\mathcal{C}(\mathcal{H}_F\|\mathcal{H}_T) = C_{fn}$
Prediction $\mathcal{H}_T$	$\mathcal{C}(\mathcal{H}_T\|\mathcal{H}_F) = C_{fp}$	0

We assume that the two given costs have a positive value.

The computation of the expected cost now becomes:

$$\mathcal{C}_{x,\mathcal{R}}(\mathcal{H}_T) = C_{fp}P(\mathcal{H}_F|x,\mathcal{R}) + 0 \cdot P(\mathcal{H}_T|x,\mathcal{R}) = C_{fp}P(\mathcal{H}_F|x,\mathcal{R})$$
$$\mathcal{C}_{x,\mathcal{R}}(\mathcal{H}_F) = C_{fn}P(\mathcal{H}_T|x,\mathcal{R}) + 0 \cdot P(\mathcal{H}_F|x,\mathcal{R}) = C_{fn}P(\mathcal{H}_T|x,\mathcal{R})$$

And according to bayes decision (assign the one with lowest cost):

$$a^*(x,\mathcal{R}) = \begin{cases} \mathcal{H}_T & \text{if } C_{fp}P(\mathcal{H}_F|x,\mathcal{R}) < C_{fn}P(\mathcal{H}_T|x,\mathcal{R}) \\ \mathcal{H}_F & \text{if } C_{fp}P(\mathcal{H}_F|x,\mathcal{R}) > C_{fn}P(\mathcal{H}_T|x,\mathcal{R}) \end{cases}$$

This is usually done by computing the **ratio** between the two expected costs:

$$a^*(x,\mathcal{R}) = \begin{cases} \mathcal{H}_T & \text{if } r(x) > 0 \\ \mathcal{H}_F & \text{if } r(x) < 0 \end{cases}$$

$$r(x) = \log \frac{C_{fn}P(\mathcal{H}_T|x,\mathcal{R})}{C_{fp}P(\mathcal{H}_F|x,\mathcal{R})}$$

Now let's assume that R is a generative model: so the posterior probability can be expressed using the prior and the likelihood:

$$r(x) = \log \frac{\pi_T C_{fn}}{(1-\pi_T)C_{fp}} \cdot \frac{f_{X|\mathcal{H},\mathcal{R}}(x|\mathcal{H}_T)}{f_{X|\mathcal{H},\mathcal{R}}(x|\mathcal{H}_F)}$$

This changed our decision function to:

$$\log \frac{f_{X|\mathcal{H},\mathcal{R}}(x|\mathcal{H}_T)}{f_{X|\mathcal{H},\mathcal{R}}(x|\mathcal{H}_F)} \lessgtr -\log \frac{\pi_T C_{fn}}{(1-\pi_T)C_{fp}}$$

It's very similar to the formula we got for the gaussian classifier with two classes:

$$\log \frac{f_{X|C}(\boldsymbol{x}_t|h_1)}{f_{X|C}(\boldsymbol{x}_t|h_0)} \lessgtr -\log \frac{\pi}{1-\pi}$$

That's why when we introduced the gaussian classifier we said it was based on **optimal bayes decision**. We didn't said it, but the optimal bayes decision is based on the assumption that the class has the same cost.

The triplet (prior, cost of fp, cost of fn) is called the **working point** of an application.
We won't usually work with the original working point but with an equivalent but simply version:

$$(\tilde{\pi}, 1, 1) \text{ with}$$

$$\tilde{\pi} = \frac{\pi_T C_{fn}}{\pi_T C_{fn} + (1-\pi_T)C_{fp}}$$

This allows us to work only with the prior without worrying about the costs.
By substitution we show that:

$$\frac{\tilde{\pi}}{1-\tilde{\pi}} = \frac{\pi_T C_{fn}}{(1-\pi_T)C_{fp}}$$

This prior is called the **effective prior**.

Bayes Risk

Up to this point we have used bayes decision to take into account the cost when classifying a sample. Now we will expand this concept to use it to verify how good our model is.

Instead of computing the cost of an action, we will compute the cost of the decision of our model(the class he chose) given the actual class c:

$\mathcal{C}(a(x,\mathcal{R})|c)$ where a(x, R) is the label predicted by our model.

The **Bayes risk** is the mean of the cost of the decisions of our model in an evaluation population:

$$\mathcal{B} = \mathbb{E}_{X,C|\mathcal{E}}\left[\mathcal{C}(a(x,\mathcal{R})|c)\right]$$

And should be the cost we expect to pay when we adopt our classifier.
ε represent an evaluator who has knowledge of the class of the samples.

The bayes risk is

$$= \int f_{X|\mathcal{E}}(x) \sum_{c=1}^{K} \mathcal{C}(a(x,\mathcal{R})|c) P(C|X=x,\mathcal{E}) dx$$

For each sample we compute the cost as already seen and we multiply it by its density in the evaluation population.

So as long as the distribution of data for the evaluation is similar to the distribution of data for the model the risk is minimized by **minimum-Bayes-cost decisions**.

The problem is that this assumption is not always true:

- Mismatches may exist between training and evaluation data.
- R is only a model of the data.
- The model may not be producing good approximations of the data distributions.

We could still define the cost of decision of our classifier on the evaluation population as:

$$\mathcal{B} = \mathbb{E}_{X,C|\mathcal{E}}\left[\mathcal{C}(a(x,\mathcal{R})|c)\right] = \sum_{c=1}^{K} \pi_c \int \mathcal{C}(a(x,\mathcal{R})|c) f_{X|C,\mathcal{E}}(x|c) dx$$

Unfortunately we don't have access to $f_{X|C,\mathcal{E}}(x|c)$ because it means that we would have to compute a density estimation of a class conditional probability of our evaluation population.

But what we can do is approximate our evaluation population and simply our integral as:

$$\int \mathcal{C}(a(x,\mathcal{R})|c) f_{X|C,\mathcal{E}}(x|c) dx \approx \frac{1}{N_c} \sum_{i|c_i=c} \mathcal{C}(a(x_i,\mathcal{R})|c)$$

So we compute the cost for all samples of a class and we divide by the number of samples of that class.

This give us our final result: the **empirical bayes risk**

$$\mathcal{B}_{emp} = \sum_{c=1}^{K} \frac{\pi_c}{N_c} \sum_{i|c_i=c} \mathcal{C}(a(x_i, \mathcal{R})|c)$$

We sum the cost of all samples in a class and then we weight it by the prior and the number of elements in that class.

This will be our main technique to compare models: if a model has a lower empirical bayes risk it will have a lower cost when it's applied and will make better decisions.

Let's use the example from before. We have the same cost matrix and priors, and now we compute the confusion matrix on an evaluation population:

$$\boldsymbol{M} = \begin{bmatrix} 205 & 111 & 56 \\ 145 & 199 & 121 \\ 50 & 92 & 225 \end{bmatrix}$$

We can to compute the mean cost for each class by using the number of elements for each class, the prior, the cost and looking at each column of M:

$$\frac{\pi_1}{N_1} \sum_{i|c_i=1} \mathcal{C}(a^*(x_i, \mathcal{R})|c) = \frac{0.3}{400} (0 \times 205 + 1 \times 145 + 2 \times 50) = 0.18375$$

Doing this for each class we get:

$$\mathcal{B}_{emp} \approx 0.18375 + 0.20199 + 0.17388 = 0.55962$$

So when we use this model we have to remember that we expect to pay 0.56 on average.

Bayes Risk for Binary problems

What happens when we apply the bayes risk to binary problems?

$$\begin{aligned}
\mathcal{B}_{emp} &= \frac{\pi_T}{N_T} \sum_{i|c_i=\mathcal{H}_T} \mathcal{C}(c_i^*|\mathcal{H}_T) + \frac{1-\pi_T}{N_F} \sum_{i|c_i=\mathcal{H}_F} \mathcal{C}(c_i^*|\mathcal{H}_F) \\
&= \pi_T \frac{\sum_{i|c_i=\mathcal{H}_T} C_{fn} \mathbb{I}[c_i^* = \mathcal{H}_F]}{N_T} + (1-\pi_T) \frac{\sum_{i|c_i=\mathcal{H}_F} C_{fp} \mathbb{I}[c_i^* = \mathcal{H}_T]}{N_F} \\
&= \pi_T \frac{\sum_{i|c_i=\mathcal{H}_T, c_i^*=\mathcal{H}_F} C_{fn}}{N_T} + (1-\pi_T) \frac{\sum_{i|c_i=\mathcal{H}_F, c_i^*=\mathcal{H}_T} C_{fp}}{N_F} \\
&= \pi_T C_{fn} \cdot FNR + (1-\pi_T) C_{fp} \cdot FPR \\
&= \pi_T C_{fn} P_{fn} + (1-\pi_T) C_{fp} P_{fp}
\end{aligned}$$

Where ci is the actual label and ci* is the label assigned to the classifier. So the empirical bayes risk for binary problems is just multiplying the two costs by the FPR or FNR and the prior. We use the rate because we are summing the cost for each wrong sample and dividing for the number of elements of that class.

We will abbreviate with:

- $P_{fn} = FNR$
- $P_{fp} = FPR$

This version of bayes risk is called: **detection cost function (dcf)**.
This cost has the same meaning of the bayes risk, but for comparing purpose we prefer to use the normalized version of the DCF:

- We compute the DCF of two dummy systems. Both of those assign the same label, so theoretically one rate is 100% and the other is 0:

$$P_{fp} = 1, P_{fn} = 0 \implies DCF_u = (1 - \pi_T)C_{fp}$$

$$P_{fp} = 0, P_{fn} = 1 \implies DCF_u = \pi_T C_{fn}$$

- We take the best of the two dummies and we compare it to our model. That gives us the normalized DCF:

$$DCF(\pi_T, C_{fn}, C_{fp}) = \frac{DCF_u(\pi_T, C_{fn}, C_{fp})}{\min(\pi_T C_{fn}, (1 - \pi_T)C_{fp})}$$

This version of DCF is more balanced and more used to compare different models.

We can use the trick we saw before to compute a new prior with both costs equal to one so that:

$$DCF_u(\tilde{\pi}) = \tilde{\pi} P_{fn} + (1 - \tilde{\pi})P_{fp}$$

But since the normalized version is invariant to scaling we get the same results.

Also if we look at the error rate (1 - accuracy):

$$e = \frac{N_T P_{fn} + N_F P_{fp}}{N} = \frac{N_T}{N} P_{fn} + \frac{N_F}{N} P_{fp}$$

It's just the DCF for the application $(\frac{N_T}{N}, 1, 1)$.

This also gives an insight why the accuracy and error rate are bad for unbalanced classes: since the used prior is an empirical one (it's the number of true sample divided by the number of sample) it may differ a lot from the application prior, and it may gives low cost where actually you have an higher cost.

Another important application of DCF is the **mis-calibration**:
Up to this point we have divided our results in log likelihood ratio and log prior ratio:

$$\log \frac{P(\mathcal{H}_T|x)}{P(\mathcal{H}_F|x)} = \log \frac{f_{X|C}(x|\mathcal{H}_T)}{f_{X|C}(x|\mathcal{H}_F)} + \log \frac{\tilde{\pi}}{1-\tilde{\pi}}$$

And we have used as an optimal threshold the ratio between the priors. We have done this since according to bayes decision this gives us the minimum cost (in case of different costs we can re-scale or include those in the threshold).

The problem arises from the log likelihood ratio:

- We may not have a probabilistic model (like SVM).
- Between the train and test population there may be some mis-matches.
- The assumptions that the model uses may be not accurate.

For these reasons the produced scores may be **mis-calibrated** and the theoretical threshold won't work anymore.

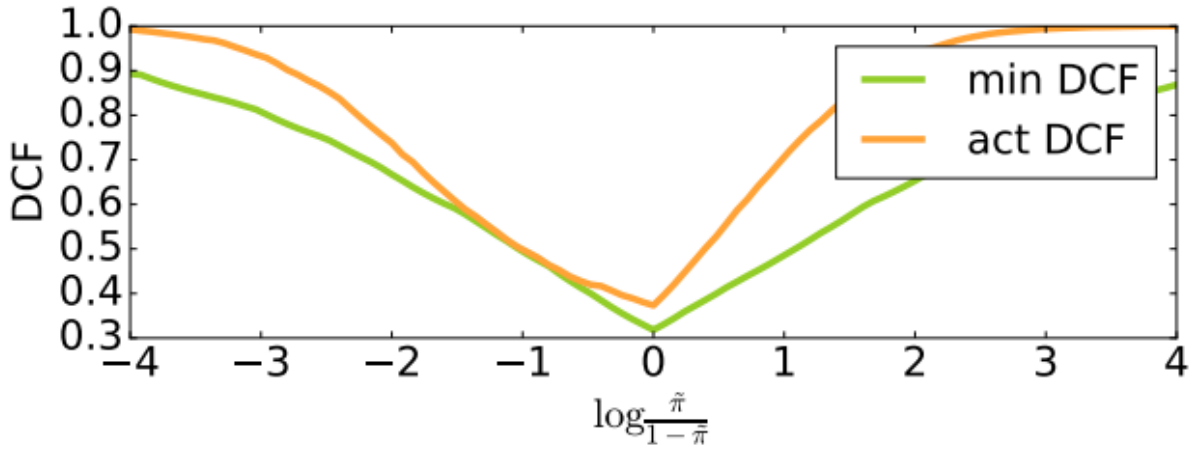
We want to understand how much the mis-calibration is affecting our model, how can we do that?

➔ We measure the difference between our normalized cost and a theoretical minimum cost. The higher the difference the higher is the effect of mis-calibration.

For computing the minimum cost we define the **minimum DCF**: it is the cost of our model in the situation where we have used the optimal threshold for the application. But how can we compute it? We try varying the threshold t and every time we compute the cost, the minimum DCF corresponds to the minimum cost we got.

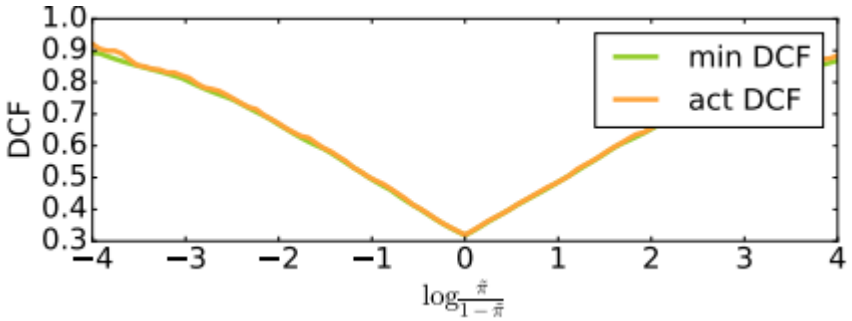
So the difference between the DCF using the actual prior and the minimum DCF is the cost we are paying due to mis-calibration between our model and our dataset.

Using **Bayes error plots** we can compare different models using their bayes costs. These plots show the cost for different application, so on the x axis we will use the optimal threshold:



In the case of this model for some applications (around -1) the log likelihood ratio is calibrated, but for other applications (greater than 0) the cost has increased a lot.

A well calibrated model should have a bayes error plots like this one:



We can also compare models by using these plots.
Problem: if a model has a low minDCF and a high actDCF compared to another model, what do we do? Theoretically this model is better compared to the other one but due to the mis-calibration it has a higher cost.

➔ We can try to use **score calibration techniques**.

We will use a technique based on score models: a general approach that looks for a function that transforms the classifier scores into approximately well-calibrated LLRs, in a way that is as much as possible independent from the target application.
Hopefully this will bring our actDCF closer to the minDCF.

Prior-weighted logistic regression:
We consider the sets of scores produced by our classifier as a single feature dataset. We want to find a mapping from these scores to the label.

We will use a linear mapping (that's the reason we will use linear regression):

$$f(s) = \alpha s + \gamma$$

Since we produce calibrated scores this function f can be transformed into a log likelihood:

$$f(s) = \log \frac{f_{S|C}(s|\mathcal{H}_T)}{f_{S|C}(s|\mathcal{H}_F)}$$

And using the class posterior ratio:

$$\log \frac{P(C=\mathcal{H}_T|s)}{P(C=\mathcal{H}_F|s)} = \alpha s + \gamma + \log \frac{\tilde{\pi}}{1-\tilde{\pi}} = \alpha s + \beta$$

We can adopt the same technique we have used for the logistic regression algorithm to find alpha and beta. For the actual prior we can use the prior given by our application and we can extract from beta the value of gamma:

$$\alpha s + \gamma = \alpha s + \beta - \log \frac{\tilde{\pi}}{1 - \tilde{\pi}}$$

Even if we still need to specify a prior (so it seems like we are optimizing for a specific application) this technique works for different applications.

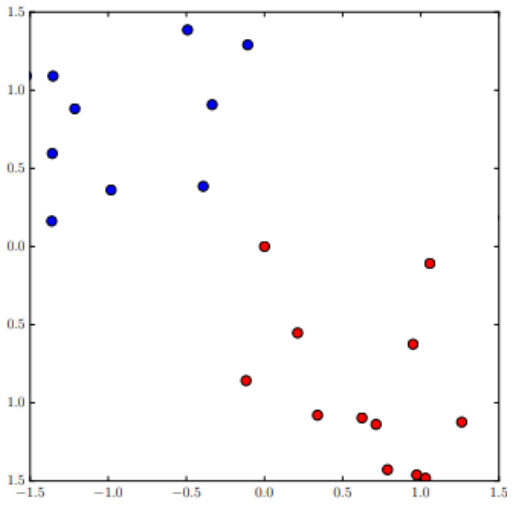
For training this new model we have to reserve a subset of our training data: **the calibration set**.

Our final model composed of the original model plus the logistic regression will produce calibrated scores that hopefully will reduce the cost for mis-calibration.

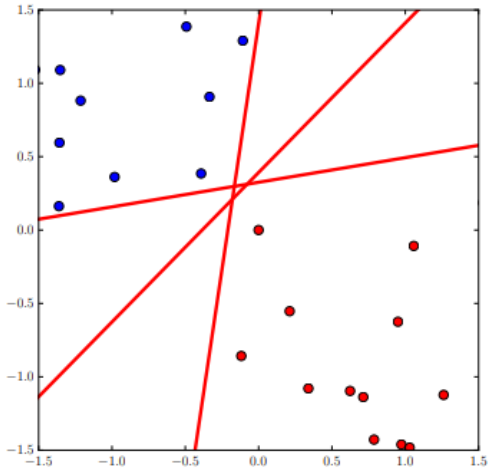
Support Vector Machine

SVM is a discriminant non-probabilistic classifier. We will start with the linear version (similar to the original logistic regression) and then we will see how to implement a non-linear separation.
The main difference from this classifier and the previous one is that this classifier doesn't have a probabilistic interpretation, but the output are just scores.

For understanding how the model works we start by using a binary linearly separable dataset (i.e. we can define a linear separation that correctly classify the two classes):

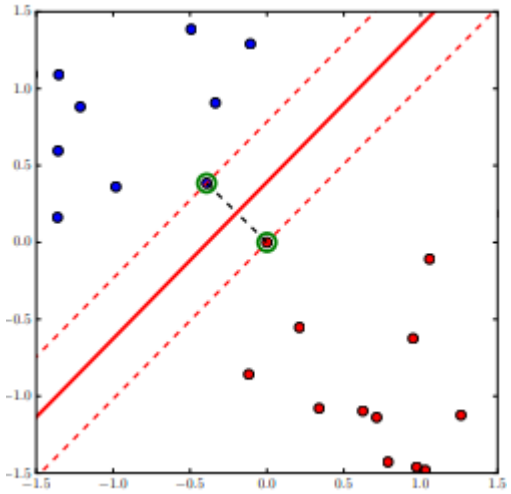


We can define an infinite set of linear separation:



But which is the better one?
When we studied logistic regression we decided that the best separation was the one that minimizes the loss (and maximizes the posterior probability).
We have also seen that in a linearly separable context we had to introduce a penalization for the norm of w, since in linearly separable.

For the SVM we will use the concept of **margin**: the best linear separation is the one with the largest margin. We can define the margin as the distance from the separation surface and the closest point of the two classes.
In the previous example we choose:



The idea is: the larger the margin the lowest the probability of making a mistake.

We need a way to compute it:
If we define our linear separation function as:

$$f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + b$$

We can define the distance from any sample and the hyperplane as:

$$d(\boldsymbol{x}_i) = \frac{|f(\boldsymbol{x}_i)|}{\|\boldsymbol{w}\|}$$

Since the classes are linearly separable we know that for any xi:

$$f(\boldsymbol{x}_i) > 0 \text{ if } c_i = \mathcal{H}_T$$
$$f(\boldsymbol{x}_i) < 0 \text{ if } c_i = \mathcal{H}_F$$

In the logistic regression chapter we have defined how to compute the projection of a point in an hyperplane so we can remove f(x):

$$d(\boldsymbol{x}) = \frac{|f(\boldsymbol{x})|}{\|\boldsymbol{w}\|} = \frac{|z_i(\boldsymbol{w}^T\boldsymbol{x} + b)|}{\|\boldsymbol{w}\|}$$

Where

$$z_i = \begin{cases} +1 & \text{if } c_i = \mathcal{H}_T \\ -1 & \text{if } c_i = \mathcal{H}_F \end{cases}$$

Now we can define the best hyperplane as the one that maximizes the margin: since for the margin we use the minimum distance from the samples to the hyperplane we are maximizing the minimum of the distances from the hyperplane:

$$\boldsymbol{w}^*, b^* = \arg\max_{\boldsymbol{w},b} \min_{i \in \{1 \ldots n\}} d(\boldsymbol{x}_i) = \arg\max_{\boldsymbol{w},b} \min_{i \in \{1 \ldots n\}} \frac{|z_i(\boldsymbol{w}^T\boldsymbol{x}_i + b)|}{\|\boldsymbol{w}\|}$$

The ideas for which we define it in this way is:

- The margin is defined as the minimum of all distances from the hyperplane.
- The best hyperplane is the one with the highest margin, so it's the one that maximizes it.
- Maximizing the margin is like maximizing the minimum distance from the hyperplane.

Now we can make two considerations:

- To compute the minimum we don't really need the w (it doesn't change if we change the sample).
- Since the classes are linearly separable zi times the distance will always produce positive values (you can check it by remembering the formulas from logistic regression and the linear separable problem), so we don't need to specify the absolute value.

So our final objective function becomes:

$$= \arg\max_{\boldsymbol{w},b} \frac{1}{\|\boldsymbol{w}\|} \min_i \left[z_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) \right]$$

Equivalent to the previous one by our hypothesis.

Solve this objective function is non trivial, but we can use another property: this objective function is invariant to scaling:

$$\frac{1}{\|\boldsymbol{w}\|} \min_i \left[z_i(\boldsymbol{w}^T\boldsymbol{x}_i + b)\right] = \frac{1}{\|\alpha\boldsymbol{w}\|} \min_i \left[z_i(\alpha\boldsymbol{w}^T\boldsymbol{x}_i + \alpha b)\right]$$

So an optimal solution for w and b **doesn't** depend on the value of alpha (it's easy to see that if we remove the minimum all the values alpha can be simplified due to the division).

Since we can use wetherver solution we restrict our problem to solutions such that the minimum distance is equal to one.

So our objective function becomes:

$$\arg\max_{\boldsymbol{w},b} \frac{1}{\|\boldsymbol{w}\|}$$

$$\text{s.t.} \begin{cases} z_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) \geq 1, \quad i = 1 \ldots n \\ \min_i z_i(\boldsymbol{w}^T\boldsymbol{x}_k + b) = 1 \end{cases}$$

Since maximizing the inverse of a value is equivalent to minimizing such value we change our objective function to a minimization of w:

$$\arg\min_{\boldsymbol{w},b} \frac{1}{2}\|\boldsymbol{w}\|^2$$

$$\text{s.t.} \begin{cases} z_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) \geq 1, \quad i = 1 \ldots n \\ \min_i z_i(\boldsymbol{w}^T\boldsymbol{x}_k + b) = 1 \end{cases}$$

Last thing: we can observe that the last constraint is useless. We could prove that if a solution fails the second constraint, we could create a new solution which is better than the previous one (the intuitive idea is that you can minimize the norm of w such that for some data now the minimum is equal to one, and since we are in a minimization problem this new solution is better).

So we finally have our objective function:

$$\arg\min_{\boldsymbol{w},b} \frac{1}{2}\|\boldsymbol{w}\|^2$$

$$\text{s.t.} \quad z_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) \geq 1, \quad i = 1 \ldots n$$

Why do we like this last formulation? It has become a **convex quadratic programming** problem, which we can solve.

This formulation has the name of **primal formulation**.

For solving it we also need to include the constraint inside the function, so we will use the Lagrangian multipliers:

$$L(\boldsymbol{w}, b, \boldsymbol{\alpha}) = \frac{1}{2} \|\boldsymbol{w}\|^2 - \sum_{i=1}^{n} \alpha_i \left[z_i \left(\boldsymbol{w}^T \boldsymbol{x}_i + b \right) - 1 \right]$$

By computing the derivatives of L with respect to w, b and alpha we get the **dual SVM problem**:

$$\max_{\boldsymbol{\alpha}} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j z_i z_j \boldsymbol{x}_i^T \boldsymbol{x}_j$$

s.t.

$$\alpha_i \geq 0, \quad i = 1, \ldots, n$$

$$\sum_{i=1}^{n} \alpha_i z_i = 0$$

Is there any relation between the primal formulation and the dual one? We can show that for any feasible solution we have:

$$L_D(\boldsymbol{\alpha}) \leq L_P(\boldsymbol{w}, b)$$

And for any optimal solutions:

$$L_D(\boldsymbol{\alpha}^*) = L_P(\boldsymbol{w}^*, b^*)$$

So we can try to find an optimal solution using the Dual problem.

To finally found an optimal solution we can use the KKT conditions on the objective L:

$$\boldsymbol{w} - \sum_{i=1} \alpha_i z_i \boldsymbol{x}_i = \boldsymbol{0}$$

$$-\sum_{i=1}^{n} \alpha_i z_i = 0$$

$$z_i \left(\boldsymbol{w}^T \boldsymbol{x}_i + b \right) - 1 \geq 0 \qquad \forall i$$

$$\alpha_i \geq 0 \qquad \forall i$$

$$\alpha_i \left[z_i \left(\boldsymbol{w}^T \boldsymbol{x}_i + b \right) - 1 \right] = 0 \qquad \forall i$$

Simply put:

- The first two equations are from the primal solution: the optimal solution is the one such that the gradient becomes zero.
- The third and forth makes sure that both the primal and the dual solutions are feasible (the solution respects the constraints of both formulations).
- The last one is the most important because it gives us the idea of **support vectors**.

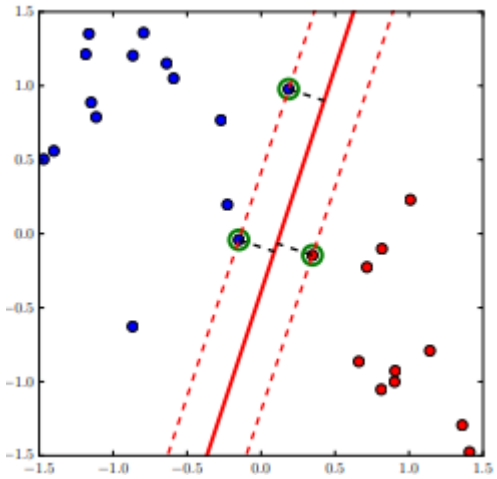
Let's analyze just the last equation:

$$\alpha_i \left[z_i \left(\boldsymbol{w}^T \boldsymbol{x}_i + b \right) - 1 \right] = 0 \qquad \forall i$$

It says that for any sample we have one of this situations:

- The alpha is equal to zero and so the distance of the point from the hyperplane can be greater than 1.
- The distance of the point from the hyperplane is equal to 1, and so alpha has a value greater than 1. All the points that lie on the margin are called the support vectors.

From the example that we used at the beginning we can define 3 support vectors:



Why do we care about support vectors? Those are the ones that we will use to classify.

We can retrieve the value of w as: $\boldsymbol{w} = \sum_{i=1}^{n} \alpha_i z_i \boldsymbol{x}_i$.
We can also estimate the value of b by using the support vectors:

- For those vector we know that the distance is equal to 1, so for any support vector $z_i \left(\boldsymbol{w}^T \boldsymbol{x}_i + b \right)$ must be equal to 1. Since we know xi, zi and w we can compute the value of b: b = 1 - zi*(w^T xi)

So to compute the score of a sample what we do is:

$$s(\boldsymbol{x}_t) = \boldsymbol{w}^T \boldsymbol{x}_t + b = \sum_{i=1}^{n} \alpha_i z_i \boldsymbol{x}_i^T \boldsymbol{x}_t + b$$

Since for vectors that do not support vector alpha is equal to zero we can see that the classification depends only on support vectors.

Non separable classes

Now we need to consider the situation where the classes are not linearly separable.
We will change a bit our strategy: since surely some point will violate our constraint

$$z_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) \geq 1$$

Because some point will be inside the margin.

So we will try to maximize the margin while trying to minimize the cost that we pay for each point inside the margin.

We will represent this cost with a **slack variable** $\xi_i \geq 0$ which represent how much a sample is violating the constraint.

So now our constraint becomes:

$$z_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) \geq 1 - \xi_i$$

If a point is inside the margin its slack variable will have a value greater than zero.

Since we want to minimize the number of point inside the margin we will try to minimize the sum of the slack variables:

$$\Phi(\boldsymbol{\xi}) = \sum_{i=1}^{n} \xi_i^{\sigma}$$

And we can introduce this inside our objective function:

$$\frac{1}{2}\|\boldsymbol{w}\|^2 + CF\left(\sum_{i=1}^{n} \xi_i^{\sigma}\right)$$

s.t.

$$z_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) \geq 1 - \xi_i \qquad \forall i$$
$$\xi_i \geq 0 \qquad \forall i$$

With C and sigma constant and F a function.
With appropriate values for C and sigma this function minimizes the number of points inside the margin.

There is a problem: an optimal solution for this objective function is **hard to find**. So we will simplify the problem in two ways:

- Sigma will be equal to one.
- The function F will be the identity function.

So our **final** objective function will be (in the primal formulation):

$$\min_{\boldsymbol{w},b,\boldsymbol{\xi}} \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{n}\xi_i$$

$$\text{s.t. } z_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) \geq 1 - \xi_i \qquad \forall i = 1\ldots n$$

$$\xi_i \geq 0 \qquad \forall i = 1\ldots n$$

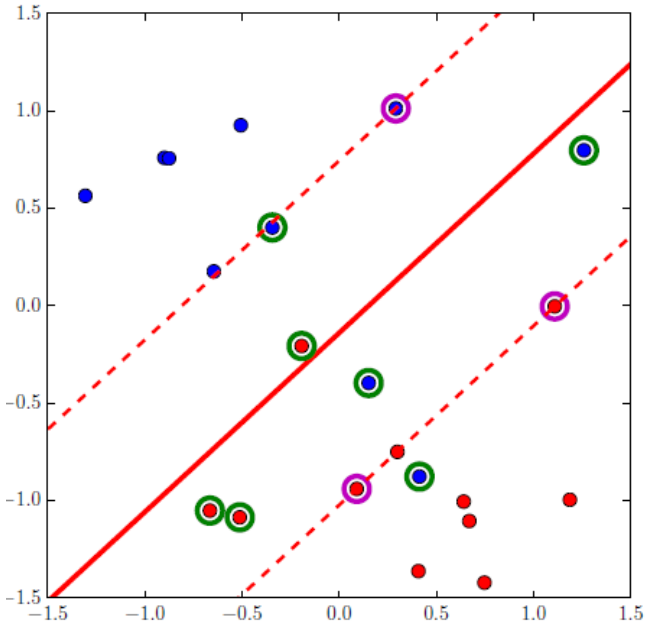
We can expand the margin as long as the cost for including samples inside the margin doesn't grow too much.

Also note that **mis-classified** points will have a slack variable greater than one.

C in an hyperparameter such that:

- Greater value of C will allow a low number of points inside the margin (with large values we get the same result of the previous version of SVM)
- A value close to zero allows a big number of points inside the margin.
- This is the same for mis-classifed points.

We can now see how the points are allowed inside the margin:



The green points have a slack variable greater than zero.

The hyperplane with no point inside the margin is called **hard margin**, while this solution is called **soft margin**.

We can solve it by using KTT and lagrangian: for our concern the solution remain the same but with an additional constraint:

$$\alpha_i \leq C$$

So alpha can't now be greater than C (before it could grow indefinitely).
And we can compute the dual problem:

$$\max_{\boldsymbol{\alpha}} L_D(\boldsymbol{\alpha}) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j z_i z_j \boldsymbol{x}_i^T \boldsymbol{x}_j$$

s.t.

$$0 \leq \alpha_i \leq C, \quad i = 1, \ldots, n$$

$$\sum_{i=1}^{n} \alpha_i z_i = 0$$

The computation of the score is the same of the hard margin case:

$$s(\boldsymbol{x}_t) = \boldsymbol{w}^T \boldsymbol{x}_t + b = \sum_{i=1}^{n} \alpha_i z_i \boldsymbol{x}_i^T \boldsymbol{x}_t + b$$

Note that we also changed the formula for support vector:

$$\alpha_i \left[z_i \left(\boldsymbol{w}^T \boldsymbol{x}_i + b \right) - 1 + \xi_i \right] = 0$$

If a point is mis-classified and lies on the margin its slack variable will be equal to 2. But this makes the part inside [...] equal to zero and so it's still considered a support vector!

How do we solve this function? We can try to solve the primal formulation but we need to remove the constraints to make it easier to solve.

Let's define the error of a sample as $1 - z_i(\boldsymbol{w}^T \boldsymbol{x}_i + b)$

How can we do that? We can see that for any xi we have two situation:

- If it is outside the margin and correctly classified: the slack variable is equal to zero and the error is negative.

- If it's inside the margin or mis-classified the slack is equal to the error, which is greater than zero.

So we don't need to the constraints on the primal formulation and it can be rewritten as:

$$\min_{\boldsymbol{w},b} \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{n} \max\left[0, 1 - z_i(\boldsymbol{w}^T\boldsymbol{x}_i + b)\right]$$

Where $f(s) = \max(0, 1-s)$ is the **hinge loss function**.

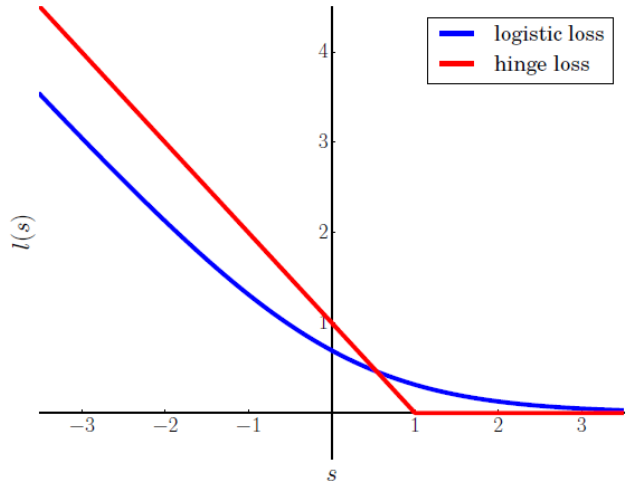
Also if we rewrite our objective by setting a parameter lambda to an appropriate value that depends on C we can see that:

$$\min_{\boldsymbol{w},b} \frac{\lambda}{2}\|\boldsymbol{w}\|^2 + \frac{1}{n}\sum_{i=1}^{n} \max\left[0, 1 - z_i(\boldsymbol{w}^T\boldsymbol{x}_i + b)\right]$$

with Logistic Regression:

$$\min_{\boldsymbol{w},b} \frac{\lambda}{2}\|\boldsymbol{w}\|^2 + \frac{1}{n}\sum_{i} \log\left[1 + e^{-z_i\left(\boldsymbol{w}^T\boldsymbol{x}_i + b\right)}\right]$$

So the SVM and the Logistic regression both try to minimize the norm of w while minimizing an empirical risk, computed using two different loss functions:



Non-linear classification

Up to this point we have worked using two equivalent problems: the dual and the primal one, where an optimal solution works for both formulations.

Let's look at how we score for both formulations:

Primal:

$$s(\boldsymbol{x}_t) = \boldsymbol{w}^T \boldsymbol{x}_t + b$$

Dual:

$$\begin{aligned} s(\boldsymbol{x}_t) &= \sum_{i=1}^{N} \alpha_i z_i \boldsymbol{x}_i^T \boldsymbol{x}_t + b \\ &= \sum_{i|\alpha_i>0} \alpha_i z_i \boldsymbol{x}_i^T \boldsymbol{x}_t + b \end{aligned}$$

The primal formulation is a vector multiplication, which increases in complexity with the dimensionality while the second formulation is a sum over the support vector, so the more support vectors we have the more time we have to multiply.

So why do we care about the dual formulation if the first one is better?

For understanding this we need to ask another question: why do we care about another linear classifier? We have seen that those are worse classifiers in general.

The trick is the same as quadratic linear regression: we will create a linear separation in an expanded feature space. With the primal formulation we would need to compute an explicit non linear mapping while with the dual formulation we can use the multiplication between xi and xt.

We will transform it into a dot product of an **expanded feature space** $\Phi(x)$.

If we can imagine to have a **kernel function** k that computed the dot product in an expanded feature space our scoring becomes:

$$s(\boldsymbol{x}_t) = \sum_{i=1|\alpha_i>0} \alpha_i z_i \boldsymbol{\Phi}(\boldsymbol{x}_i)^T \boldsymbol{\Phi}(\boldsymbol{x}_t) + b = \sum_{i=1|\alpha_i>0} \alpha_i z_i k(\boldsymbol{x}_i, \boldsymbol{x}_t) + b$$

If we go back to the complexity of the two scorings, the dual formulation is better in an expanded feature space: since our dimensionality is increasing, the complexity of the primal formulation is getting worse while the dual formulation is not getting worse (it depends only on support vector).

Let's use the same expansion we use with the logistic regression:

$$\boldsymbol{\Phi}(\boldsymbol{x}) = \begin{bmatrix} vec(\boldsymbol{x}\boldsymbol{x}^T) \\ \sqrt{2}\boldsymbol{x} \\ 1 \end{bmatrix}$$

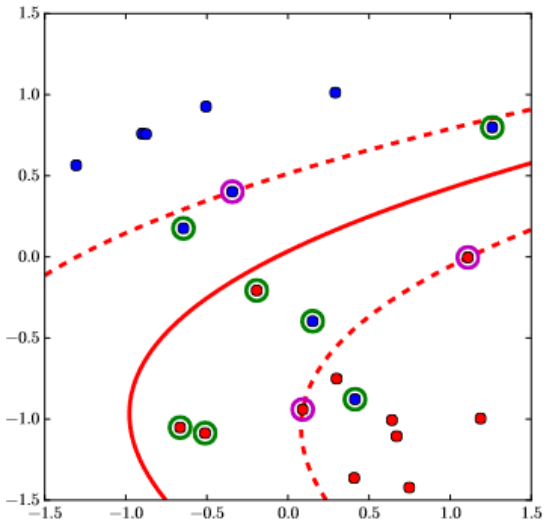
And let's compute k(x1, x2):

$$\boldsymbol{\Phi}(\boldsymbol{x}_1)^T \boldsymbol{\Phi}(\boldsymbol{x}_2) = \left(\boldsymbol{x}_1^T \boldsymbol{x}_2\right)^2 + 2\boldsymbol{x}_1^T \boldsymbol{x}_2 + 1$$

It's easy to check that this is equivalent to:

$$k(\boldsymbol{x}_1, \boldsymbol{x}_2) = \left(\boldsymbol{x}_1^T \boldsymbol{x}_2 + 1\right)^2$$

This expansion is called **polynomial kernel of degree 2**. In general we can define any polynomial kernel of degree d.
Let see it in action:



Another popular one is the **Gaussian Radial Basis Function kernel**:

$$k(\boldsymbol{x}_1, \boldsymbol{x}_2) = e^{-\gamma \|\boldsymbol{x}_1 - \boldsymbol{x}_2\|^2}$$

The trick is that we compute the distance between the points for the kernel:

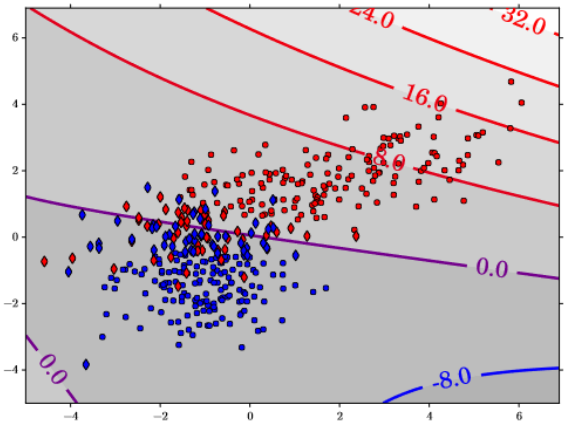
- It will tend to 1 as the points are closer.
- It will tend to 0 as the points are far.

So the score using the support vector will depend on points closer to xt. With γ we control this effect:

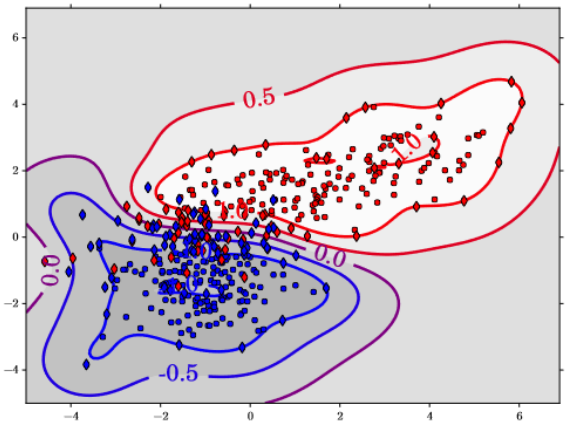
- With low values this effect is mitigated.
- With high values only the closest support vector will be used in the scoring.

Given the same dataset we can see how these kernel functions change the way the SVM classifies:

Poly — $d = 2$, $C = 1.0$

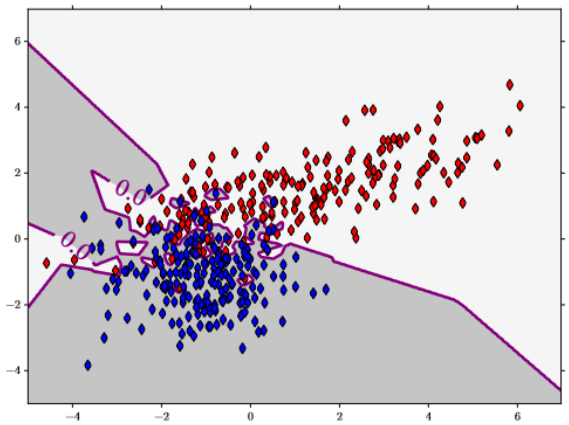


RBF — $\gamma = 0.5$, $C = 1.0$

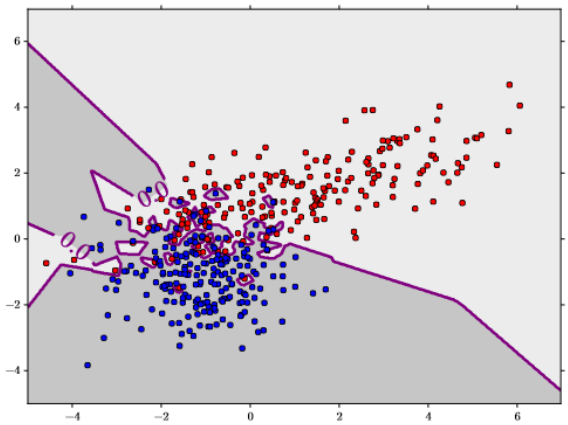


We can actually see how RBF with an high parameter classify very similarly to 1-NN:

RBF (no bias) — $\gamma = 200$, $C = 1.0$



1–NN



Lastly, in case the dataset is unbalanced we can introduce a weight inside the objective function, such that the cost of a points depend on same parameter (like the prior probability):

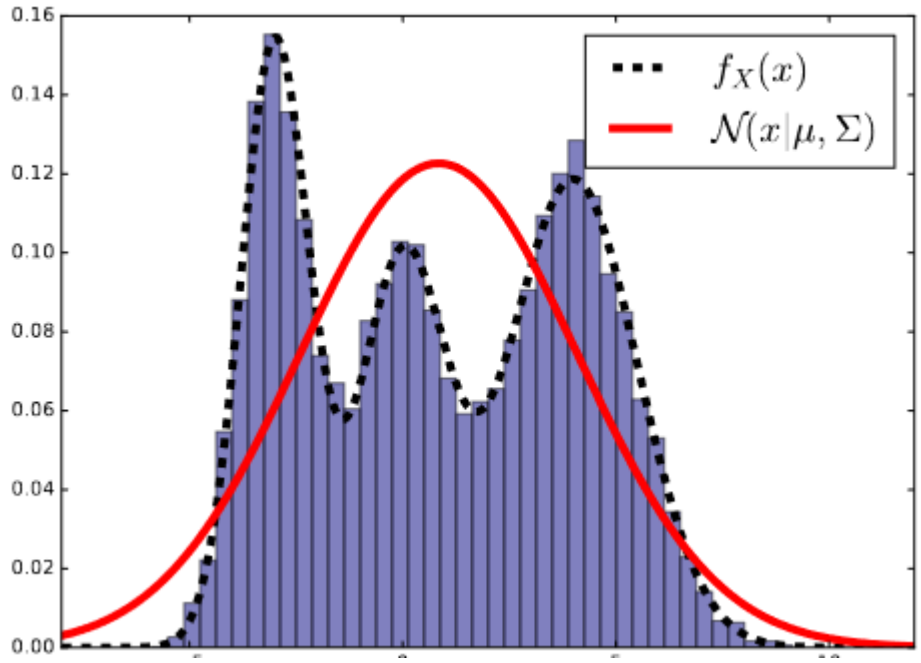
$$\arg\min_{\boldsymbol{w},b} \frac{1}{2}\|\boldsymbol{w}\|^2 + \sum_{i=1}^{n} C_i \left[1 - z_i(\boldsymbol{w}^T\boldsymbol{x}_i + b)\right]_+$$

In general for binary classification problems SVM provides the best results.

Gaussian Mixture Model

With this model we go back to generative models and the Gaussian hypothesis.

We have seen how assuming that class-conditional distributions are Gaussian bring interesting results but may be pretty inaccurate if the sample are spread in another way:



The only thing we can do is to find a model able to estimate any sufficiently regular distribution to a desired degree.

Since we are estimating the density from data, we require a good amount of data to obtain good estimates.

GMMs are a popular algorithm used for both classification and clustering. The main objective is to find subsets of samples which are distributed like a Gaussian distribution.

We have actually already encountered a GMM: when we were computing the prior probability we modeled it by using joint distribution divided by the density of x. We have never computed it because it was useless for comparing different classes but it can computed as:

$$f_X(\boldsymbol{x}) = \sum_{c=1}^{K} f_{X|C}(\boldsymbol{x}|c)P(C=c) = \sum_{c=1}^{K} \pi_c \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)$$

The second formulation is a **GMM**! In this case the prior probability of a class is the **weight** of the class.

More in general if X is a GMM we can compute the density of a sample x as:

$$\boldsymbol{X} \sim GMM(\boldsymbol{M}, \boldsymbol{\mathcal{S}}, \boldsymbol{w})$$

$$f_X(\boldsymbol{x}) = \sum_{c=1}^{K} w_c \mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)$$

So a weighted sum of Gaussian distributions.

In parenthesis we have the **distribution parameters** of the GMM: the set of means, the set of variances and the set of weights.

$$\boldsymbol{M} = [\boldsymbol{\mu}_1 \dots \boldsymbol{\mu}_K]$$

$$\boldsymbol{\mathcal{S}} = [\boldsymbol{\Sigma}_1 \dots \boldsymbol{\Sigma}_K]$$

$$\boldsymbol{w} = [w_1 \dots w_K]$$

One for each component (each gaussian) we have associated an element of each parameter.

Since we are computing a probability, when we estimate our parameters we have to remember that:

$$\int f_X(\boldsymbol{x}) = \int \sum_{c=1}^{K} w_c \mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) d\boldsymbol{x} = \sum_{c=1}^{K} w_c = 1$$

So the weight must sum up to 1.
To compute the third term from the second you simply extract the sum from the integral, and since the integral of a Gaussian is always equal to 1 you get that the sum of the weights must be equal to 1.

So given our train set D we can now try to estimate our GMM. For each class we try to estimate our parameters, since we don't need the class we will look at our estimation as an unsupervised model.

We will use a Maximum Likelihood approach with some heuristics to avoid an unbounded situation where the estimation algorithm never stops.

Let's now define our likelihood:

$$\mathcal{L}(\boldsymbol{\theta}) = \prod_{i=1}^{n} f_{X_i}(\boldsymbol{x}_i) = \prod_{i=1}^{n} GMM(\boldsymbol{x}_i|\boldsymbol{M}, \boldsymbol{\mathcal{S}}, \boldsymbol{w})$$
$$= \prod_{i=1}^{n} \left(\sum_{c=1}^{K} w_c \mathcal{N}(\boldsymbol{x}_i|\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) \right)$$

And using the log likelihood:

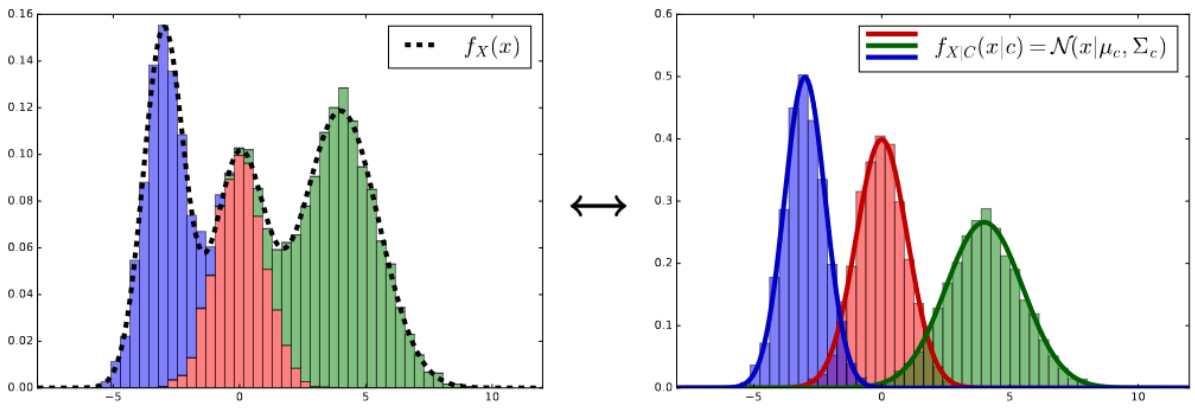
$$\ell(\boldsymbol{\theta}) = \log \mathcal{L}(\boldsymbol{\theta}) = \sum_{i=1}^{n} \log \left(\sum_{c=1}^{K} w_c \mathcal{N}(\boldsymbol{x}_i|\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) \right)$$

This part can't be directly optimized. To make it solvable we need to go back to the definition of our GMM:

$$f_X(\boldsymbol{x}) = \sum_{c=1}^{K} w_c \mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) = \sum_{c=1}^{K} f_{X_i|C_i}(\boldsymbol{x}_i|c) P(C_i = c)$$

So the density is computed like a marginal of a joint distribution of data points and corresponding **clusters**. We are partitioning our dataset into subsets (components or clusters), in such a way that the distribution of the points of each cluster can be modeled by a Gaussian distribution.

If we knew beforehand the clusters and the samples belonging to each cluster we could estimate the parameters of each Gaussian.



Unfortunately the clusters are unknown and we need to estimate those. Even if this sounds more complicated than our previous log-likelihood this will actually be easier.

We will model our cluster membership as a **random variable**. So our log likelihood will take into account both clusters and gaussian distributions.

Before writing the log likelihood, let's suppose that we have estimated our parameters:

$$\boldsymbol{\theta} = (\boldsymbol{M}, \boldsymbol{\mathcal{S}}, \boldsymbol{w})$$

We can compute the density of a sample belonging to a cluster c as:

$$f_{X_i,C_i}(\boldsymbol{x}_i, c) = w_c \mathcal{N}(\boldsymbol{x}_i; \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)$$

We can then compute the **posterior probability** of a cluster c for a sample xi as:

$$\gamma_{c,i} = P(C_i = c | \boldsymbol{X}_i = \boldsymbol{x}_i) = \frac{f_{X_i,C_i}(\boldsymbol{x}_i, c)}{f_{X_i}(\boldsymbol{x}_i)}$$

Which can be computed as:

$$= \frac{w_c \mathcal{N}(\boldsymbol{x}_i; \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)}{\sum_{c'} w_{c'} \mathcal{N}(\boldsymbol{x}_i; \boldsymbol{\mu}_{c'}, \boldsymbol{\Sigma}_{c'})}$$

We will call this value **responsibility**, representing how much a sample xi influences the cluster c.

We will assign for each sample the cluster ci* such that:

$$c_i^* = \arg\max_c P(C_i = c | \boldsymbol{X}_i = \boldsymbol{x}_i)$$

Now given that we have computed the clusters of our dataset we can compute our Gaussian distribution parameters.

This will be our algorithm:

- Assign the samples to a cluster using the corresponding Gaussian.
- For each cluster compute the Gaussian distribution.
- Stop the algorithm or restart from the first step.

A similar idea of the K-means algorithm. We will need to define a criteria such that the algorithm will stop at some point.

For our ML estimate we computing the sum of log likelihood of a MVGs with the sum of the log of the weights:

$$\ell(\boldsymbol{\theta}) = \sum_{i=1}^{n} \left[\log f_{\boldsymbol{X}_i|C_i}(\boldsymbol{x}_i|c_i^*) + \log P(C_i = c_i^*)\right]$$
$$= \sum_{i=1}^{n} \left[\log \mathcal{N}(\boldsymbol{x}_i|\boldsymbol{\mu}_{c_i^*}, \boldsymbol{\Sigma}_{c_i^*})\right] + \sum_{i=1}^{n} \left[\log w_{c_i^*}\right]$$

We thus get for each cluster c:

$$\boldsymbol{\mu}_c^* = \frac{1}{N_c} \sum_{i|c_i^*=c} \boldsymbol{x}_i \ , \quad \boldsymbol{\Sigma}_c^* = \frac{1}{N_c} \sum_{i|c_i^*=c} (\boldsymbol{x}_i - \boldsymbol{\mu}_c^*)(\boldsymbol{x}_i - \boldsymbol{\mu}_c^*)^T$$

$$w_c^* = \frac{N_c}{\sum_{c=1}^{K} N_c}$$

This algorithm is called **hard assignment** because one sample is assigned to one and only one cluster.

What happens if $P(C_i = c_1|\boldsymbol{X}_i = \boldsymbol{x}_i) \approx P(C_i = c_2|\boldsymbol{X}_i = \boldsymbol{x}_i)$? Both classes may have generated the sample xi but we will still assign x to one cluster.

Also if the we assume that the cluster have the identity matrix as covariance and same probability then our cluster assignment becomes:

$$c_i^* = \arg\max_c P(C_i = c|\boldsymbol{X}_i = \boldsymbol{x}_i) = \arg\min_c \|\boldsymbol{x}_i - \boldsymbol{\mu}_c\|^2$$

For each sample we assign it to the cluster with the closest mean. We will also need to update only the parameter mean for each cluster.

➔ The GMM becomes the **K-Means clustering algorithm**.

We will now move to **soft assignment**: each point will contribute to all the parameters but weighted by the responsibility:

$$\boldsymbol{\mu}_c = \frac{\sum_i \gamma_{c,i} \boldsymbol{x}_i}{\sum_i \gamma_{c,i}} \ w_c = \frac{N_c}{N}$$

$$\boldsymbol{\Sigma}_c = \frac{1}{N_c} \sum_i \gamma_{c,i} \left(\boldsymbol{x}_i - \boldsymbol{\mu}_c\right) \left(\boldsymbol{x}_i - \boldsymbol{\mu}_c\right)^T = \frac{1}{N_c} \sum_i \gamma_{c,i} \boldsymbol{x}_i \boldsymbol{x}_i^T - \boldsymbol{\mu}_c \boldsymbol{\mu}_c^T$$

Weighted empirical mean.

The algorithm is the same as before, we just don't assign sample to cluster but we directly compute the parameters:

- We compute the responsibilities for each sample.
- We estimate the GMM parameters.

This particular algorithm is called **Expectation-Maximization**.

Usually the soft assignment GMM gives better results compare to hard assignment GMM.

GMM for Classification

When we apply GMM in classification **for each class** we try to compute a GMM, with different clusters and parameters.

So now instead of fitting one Gaussian in a class (MVG idea) we fit K Gaussians in the samples of a class:

$$P(C_t = c | \boldsymbol{X}_t = \boldsymbol{x}_t) = \frac{P(C_t = c) \sum_{k=1}^{K_c} w_{c,k} \mathcal{N}(\boldsymbol{x}_t | \boldsymbol{\mu}_{c,k}, \boldsymbol{\Sigma}_{c,k})}{\sum_{c'} P(C_t = c') \sum_{k=1}^{K'_c} w_{c',k} \mathcal{N}(\boldsymbol{x}_t | \boldsymbol{\mu}_{c',k}, \boldsymbol{\Sigma}_{c',k})}$$

Why do we use GMM in classification?

- It's good for **open-set** problems (where in a class there are samples with more patterns instead of just one). This class is usually very heterogeneous, with GMM we can partially alleviate the issue.

For the set of clusters in each class we can use some assumptions about the covariance matrices:

- Tied: all clusters share the same cov matrix.
- Diagonal: all clusters have diagonal matrices (this is **not** the naive assumption).

The last thing we need is how to initialize the algorithm, so we need to decide the first parameters.

LBG algorithm:

- Compute a single gaussian distribution.
- From that get two gaussians by shifting the means using a constant.
- Start the EM algorithm.

After you have computed G gaussian you can split and get 2*G gaussians.

We can use **cross-validation** to find the best number of classes for our dataset and if the hypothesis works (maybe for one class the hypothesis of tied covariance matrices is good).