Machine Learning & Pattern Recognition

Andrea Leone

March 2023 - June 2023

Sommario

1 Numerical Libraries

1.1 Numpy

We will mainly use one-dimensional and two-dimensional arrays. Dimensions can also be called *axes*. Arrays have some attributes that descrive the array itself, like:

```
ndarray.size # Total number of elements
ndarray.shape # Tuple with the number of elements for each axis
ndarray.ndim # Number of axes
ndarray.dtype # The data type stored in
```

It's possible to create an array in many different ways:

```
# from a python list / tuple
numpy.array([1,2,3])
# it's also possible to specify the data type
numpy.array([1,2,3], dtype=numpy.float64)
# and to use nested lists for multi-dimensional arrays
numpy.array([[1,2,3],[4,5,6]])

# as a copy of an other array
numpy.array(other_array)

# using functions that create predefined arrays
numpy.zeros((2,3), dtype=numpy.float32) # (2,3) is the shape
numpy.ones(5) # 1D arrays of ones
numpy.arange(4) # [0, 1, 2, 3]
numpy.arange(0, 6, 2) # [0, 2, 4] (range with step)
numpy.eye(3) # identity matrix of size 3x3
numpy.linspace(0, 5, 4) # 4 equally spaced points in [0, 5]
```

Numerical operators between numpy arrays operate *element-wise*, so they require arrays with matching shape. It's also possible to perform non element-wise operations, like the matrix multiplication, which is accomplished by means of the following method

```
prod = numpy.dot(x, y)
# or
prod = x @ y
```

In this case the number of cols of x should be equal to the number of rows of y. An array can be transposed with

```
x = numpy.arange(3).reshape((3, 1)) # [[1], [2], [3]]
y = x.T # y = [1, 2, 3]
```

Reshaping is an operation that changes the shape of an array, preserving the order of the data:

```
x = [[1, 2], [3, 4], [5, 6]] # shape (3, 2)
y = x.reshape((2, 3)) # y = [[1, 2, 3], [4, 5, 6]]
```

Of course the total number of elements should be the same. This is usefull to create row-vectors and column-vectors.

```
# ravel is a method that reshapes to a 1D vector
x = numpy.array([[0, 1], [2, 3]])
x.ravel() # array([0, 1, 2, 3])
```

Note: pay attention that 1D arrays (shape (n,)) are not the same as row vectors (shape (1, n)). We will usually represent data as column vectors, and matrices will consist of horizonally stacked column vectors.
Usually, MLlibs use the opposite convention for efficiency reasons, but for us it is simpler to use column vectors because of a natural correspondence with mathematical notation, like $y = Ax$.

Numpy also provides functions for reduction operations:

```
x = numpy.array([[1,2,3], [4,5,6]])
x.sum() # 21
x.max() # 6
# these functions can also be applied over a specific axis,
                                 like
x.sum(axis = 0) # sum of rows = [5, 7, 9]
x.sum(axis = 1) # sum of cols = [6, 15]
```

It's also possible to apply a function to all elements of the array like

```
numpy.exp(x) # for each element of x compute e^x
numpy.log(x) # for each element of x compute log_e(x)
# log returns NaN for negative numbers
```

Note: whenever possible avoid iterating over an array, but apply numpy functions.

Numpy arrays can be sliced like conventional Python lists, but numpy also allows specifying a slice for each axis:

```
x = numpy.arange(15).reshape(3,5)
# [[ 0, 1, 2, 3, 4],
#  [5, 6, 7, 8, 9],
#  [10, 11, 12, 13, 14]]
x[1, 0:3] # [5, 6, 7]
x[1:2, 0:3] # [[5, 6, 7]] See the difference with previous line
```

Numpy also allows complex indexing, which can be done with integer or boolean arrays, for example

```
idx = numpy.array([0, 0, 2])
x[idx, :] # takes rows 0, 0, 2 of x, and all columns
x[:, idx] # takes all rows of x, and columns 0, 0, 2
# To use this approach on all dimensions use
x[idx, :][:, jdx] # otherwise you get an unexpected behavior

mask = numpy.array([1, 0, 1], dtype=numpy.bool) # [T, F, T]
x[mask] # takes row i only if mask[i] = True
mask = x > 5 # matrix with same shape of x
x[mask] # is a 1D array containing only x's elements > 5
```

Note: slicing creates an array view, which means that the original array data is not copied. Complex indexing, instead, copies the data. To check if a certain array owns his data use

```
x.flags.owndata # can be either True or False
```

An important feature of numpy arrays is **broadcasting**. Broadcasting allows applying elementwise operations, such as addition and multiplication, to arrays with different shapes.
Whenever arrays have different shapes:

1. 1's will be prepended to the shapes of smaller arrays until all arrays have the same number of dimensions

2. Axes with shape 1 are treated as if they had the same dimension as the array with largest size along the axis, andall elements were the same along the axis

Pay attention to broadcasting happening unintentionally.

Numpy arrays can be concatenated (horizontally or vertically):

```
x1 = numpy.array([1,2,3])
x2 = numpy.array([4,5,6])
numpy.hstack([x1, x2]) # [1,2,3,4,5,6]
numpy.vstack([x1, x2]) # [[1,2,3], [4,5,6]]
```

Finally, numpy also provides linear algebra functions inside *numpy.linalg*. For example, to compute the eigenvalue decomposition of a 2D array:

```
x = numpy.arange(9).reshape(3,3)
numpy.linalg.eig(x) # returns an array with eigenvalues and a
                                  matrix with eigenvectors
# eigh returns the same result sorted by increasing eigenvalues
```

1.2 Matplotlib (pyplot)

To plot values:

```
import matplotlib.pyplot as plt
x = numpy.linspace(0, 5, 1000)
plt.plot(x, numpy.sin(x)) # plots sin(x)
plt.xlabel('x-axis') # give a label to the x axis
plt.ylabel('y-axis') # give a label to the y axis
plt.show() # show the plot
```

The plot function receives many optional parameters to specify the color, the linestyle, the markersize, etc..
To visualize 2D data:

```
D = numpy.random.random((2, 100))
plt.scatter(D[0], D[1]) # !!
plt.show()
```

To visualize a histogram:

```
D = numpy.random.normal(size=1000)
plt.hist(D, bins=20, density=True, ec='black', color='#800000')
# bins is the number of bars showed by the histogram
plt.show()
```

The same figure can show multiple plot. The important thing is to call the *show* function at the end of all the plots.

2 MLPR Intro

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.
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.

Define models that are able to capture the regularities in our data and allow performing inference about the properties we are interested in. The models should not simply specify a set of human–defined rules, but should be able to learn from data. The learning stage should leverage observed data to improve the quality of the inference.
An example is the *classification problem*: given a set of objects, assign them to discrete categories. Find a mapping from the space of input vectors representing the objects to a discrete set of labels.

Depending on the task, we have:

- **Supervised Learning**: along with the input data the system is provided with output data (class labels). The goal is estimating a mapping between input and output data.
- **Unsupervised Learning**: no feedback is provided to the system, whose goal is to identify some structure of the data.

Unsupervised methods are often used as a pre-processing step for supervised tasks.
In this course we will focus on the classification problem (supervised technique) and on the density-estimation problem (unsupervised technique).

2.1 Pattern classification

Assign a pattern to a class. A class represent characteristics of the objects that we are considering. Depending on the number of output label, we can have :

- **Binary Classification**: only 2 output classes (e.g. intrusion detection, identity verification)
- **Multiclass Classification**: more than 2 output classes (e.g. object categorization, speech decoding)
  - **closed-set**
  - **open-set**: includes an extra class ”None of the others” but it is very complex to manage.

Binary classification is just a special case of multiclass classification, but it has some extra properties.
The classification problem can be divided into three different stages: feature extraction, dimensionality reduction, classification. The input is the test sample, the output is the predicted label.
In practice these components often interact. For example the dimensionality reduction can be seen as part of the feature extraction process.

2.1.1 Feature Extraction

We present an object in terms of numerical attributes, usually arranged as vectors or matrices. Sometimes, mapping an object into a numerical vector may be difficult. It's important to choose a significant encoding for objects, because the numerical rapresentation must contain all the useful information. Since this step is very task-specific, we won't cover it.

2.1.2 Dimensionality Reduction

The feature space is often very large and contains a large amount of useless and potentially harmful information. In this step, we compute a mapping from the n-dimensional feature space to a m-dimensional space, with $m < n$.
Dimensionality Reduction is usefull to compress information, remove noise, visualize data (to understand the relevant features), and to simplify the classification. In fact it reduces *curse of dimensionality* (if data is too sparse it's difficult to understand relevant patterns) and *overfitting* (a too complex model works well with training data but poorly with test data).

2.1.3 Classification

Perform a mapping from the m-dimensional space (obtained after dimensionality reduction) to the labels space. The mapping is also called *decision function*.
Our goal is to design suitable decision functions. The decision functions should provide a good generalization error. We want models providing good predictions on unseen data, avoiding both *overfitting* and *underfitting*.

- **Discriminant model**: the function $f(x)$ is a black-box model mapping a feature vector x directly to a label. It's the weakest model and doesn't allow to measure uncertainty. It can't take into account prior information.
- **Discriminative non-probabilistic model**: the function $f(x)$ maps a feature vector x to a set of "scores", one for each label. It can't take into account prior information.
- **Discriminative probabilistic model**: computes the model class *posterior probabilities* $P(C_k|x)$ and assigns labels according to posterior probabilities. It can't incorporate *application-dependent* information.

- **Generative probabilistic model** model the *joint distribution* of features and labels $P(x, C_k) = P(x|C_k)P(C_k)$ and applies Bayes theorem to compute posterior probabilities $P(C_k|x)$. This is the most powerful model. It can incorporate *application-dependent* prior class information ($P(C_k)$, not depending on the data distribution).

We will mainly focus on probabilistic models, so on the last two.

2.1.4 Inference problem

Create a model M describing relationships between features and labels.

- **Parametric models**: the model depends on a set of parameters θ whose size doesn't depend on the available data. For example, the separation line between two classes in a 2D dataset is always linear.

- **Non-parametric models**: the model complexity grows with the sample size, for example in the k-NN problem the separation line between two classes in a 2D dataset changes every time we add some new data.

Create a labeled training set $D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$.
Then learn the model parameters:

- **Non-probabilistic approach**: estimate optimal values for the model parameters using the training set: $D, M \to \theta$.

- **Probabilistic (Bayesian) approach**: update the prior belief over the model parameters using the observed data: $D, M, P(\theta, M) \to P(\theta|D, M)$

Finally predict the class label for an unseen test sample.

- **Generative model**

- **Discriminative model**

2.1.5 Test and Validation

We are interested in measuring the model performance on unseen data. Since it is not possible to do this, we simulate the system behaviour on known data that the system didn't see yet. This is done by means of a **test set**, which is labeled data treated as unlabeled. The test set should contain data which is similar to our use case and shouldn't contain data overlapping with the training set. Be aware that this approach may lead to biased conclusions.
Models often contain **hyperparameters** (e.g. the number of dimensions of reduced features). The optimal value of these parameters can't be estimated using the same criterion seen in Section 2.1.4.

In other cases we are interested in selecting the best model among *competing models*. This can be accomplished by means of a **validation set**. The validation set is used to asses the influence of hyperparameters on the models,

in order to select good ones.
To build a validation set it is sufficient to extract some data from the training set. Part of the training set will be used for estimating the models, while the remaining part will simulate the evaulation data (validation) and will be used to infer hyperparameters and to select the best model.
Unfortunally, sometimes we don't have much data to work with. That's where **cross-validation** comes in handy. With **K-fold** cross validation we divide the training set in K folds and repeatedly use $K-1$ folds as training data. The remaining one is used for validation. This process is iterated until every fold has been used for validation. Finally, we combine the results to select hyperparameters and the best model.
Since some models may increase performance when trained with much more data, it may be a waste to leave out an entire fold from the training set. A solution may be to use smaller folds. A particular case is the **leave-one-out** approach, where the size of each fold is 1.
Note: evaluation data should not be used to estimate anything.

3 Dimensionality Reduction

As already said, dimensionality reduction consists in reducing a n-dimensional feature space to a m-dimensional space, with $m < n$. There may be many reasons to do this, and different goals may require different approaches. For example when compressing information we want to retain the *maximum amount of information* for a given output size, while when improving classification we want to retain *discriminant information* and remove noise.
We will focus on two linear methods:

- **Principal Component Analysis** (PCA): unsupervised
- **Linear Discriminant Analysis** (LDA): supervised

In both cases our goal is to reduce the feature space retaining the most useful information.

3.1 Linear Algebra recalls

Let $\mathbf{A} \in \mathbb{R}^{n\times n}$ be a symmetric square matrix.
$\mathbf{A}$ admits an **eigen-decomposition**: $\mathbf{A} = \mathbf{V}\mathbf{\Sigma}\mathbf{V}^{-1} = \mathbf{V}\mathbf{\Sigma}\mathbf{V}^T$ where $\mathbf{V}$ is an orthogonal $n \times n$ matrix whose columns are the *eigenvectors* of $\mathbf{A}$, and $\mathbf{\Sigma}$ is a diagonal $n \times n$ matrix whose elements are the eigen-values of $\mathbf{A}$. Since $\mathbf{V}$ is orthogonal, $\mathbf{V}^T\mathbf{V} = \mathbf{V}\mathbf{V}^T = \mathbf{I}$ and $\mathbf{V}^{-1} = \mathbf{V}^T$.
A generical rectangular matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ always admits a **Singular Value Decomposition** (SVD) of the form $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$ where $\mathbf{U}$ is an orthogonal $n \times n$ matrix of eigenvectors of $\mathbf{A}\mathbf{A}^T$, $\mathbf{V}$ is an orthogonal $m \times m$ matrix of eigenvectors of $\mathbf{A}^T\mathbf{A}$, and Σ is a diagonal rectangular matrix containing the *singular values* of $\mathbf{A}$.

Given a unit vector $\mathbf{u}$ representing a direction, $y = \mathbf{u}^T\mathbf{x}$ is the projection of $\mathbf{x}$ over $\mathbf{u}$. $\hat{\mathbf{x}} = y\mathbf{u} = (\mathbf{u}^T\mathbf{x})\mathbf{u}$ is the representation of the projected point in the original space.
Given a m-dimensional subspace $\mathbf{U} = [\mathbf{u_1}, \dots, \mathbf{u_m}]$, where the set of $\{\mathbf{u_i}\}$ forms a basis of the m-dimensional space, the projection of $\mathbf{x}$ over $\mathbf{U}$ is $\mathbf{y} = \mathbf{U}^T\mathbf{x}$. The representation of $\mathbf{y}$ in the original space can be obtained as $\hat{\mathbf{x}} = \mathbf{U}\mathbf{y} = \mathbf{U}\mathbf{U}^T\mathbf{x}$.

3.2 Principal Component Analysis

Given a zero-mean dataset $\mathbf{X} = \{\mathbf{x_1}, \dots, \mathbf{x_K}\}$, where $\mathbf{x}_k \in \mathbb{R}^n$, we want to find a subspace of $\mathbb{R}^n$ that allows preserving most of the information. A subspace can be represented as a matrix $\mathbf{P} \in \mathbb{R}^{n\times m}$ with orthonormal columns. The columns of the matrix form a m-dimensional subspace of the n-dimensional space. The projection of $\mathbf{x}$ over the subspace is given by $\mathbf{y} = \mathbf{P}^T\mathbf{x}$. The coordinates of the projected point in the original space are $\hat{\mathbf{x}} = \mathbf{P}\mathbf{y}$. In this section, our objective is to define a criterion for *estimating* $\mathbf{P}$. A reasonable criterion is to minimize the *average reconstruction error*:

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

So, our goal is to find a $\mathbf{P}$ which minimizes the above *objective function.* By exploiting some matrix properties and removing some useless constant factors, we can demonstrate that our goal is to maximize

$$-\operatorname{Tr}(\mathbf{P}^T[\sum_{i=1}^{K}\mathbf{x}_i\mathbf{x}_i^T]\mathbf{P}) \tag{2}$$

The optimal solution to our problem is given by the matrix $\mathbf{P}$ whose columns are the m *eigenvectors* of $\frac{1}{K}\sum_{i=1}^{K}\mathbf{x}_i\mathbf{x}_i^T$ corresponding to the m *largest eigenvalues.*
If the dataset is not centered in the origin, the first direction of the found subspace will point towards the dataset mean (usually not an interesting information). In practice, we center the dataset by removing the dataset mean before computing the PCA. So, if the mean is $\bar{\mathbf{x}}$, the PCA subspace is computed from the eigenvectors of the **empirical covariance matrix**

$$\mathbf{C} = \frac{1}{K}\sum_{i}(\mathbf{x}_i - \bar{\mathbf{x}})(\mathbf{x}_i - \bar{\mathbf{x}})^T \tag{3}$$

PCA can be interpreted as the linear mapping that preserves the directions with highest variance (which mean those directions whose variation is significant for our analysis). We can say that the eigenvalue i corresponds to the variance of the data over the i-th direction.
To go back to the original space we do the following: $\hat{\mathbf{x}}_i = \mathbf{P}\mathbf{y}_i + \bar{\mathbf{x}}$.
The selection of the optimal m can be done using cross-validation with a validation set. We can also choose m as to retain a given percentage t of the variance of the data. This percentage can be found by dividing the sum of the m largest eigenvalues by the sum of all the eigenvalues.
Testing this method on the MNIST dataset, with a classifier based on Euclidean distance, we would find out that most of the dimensions can be safely removed. For each class we compute the mean vector $\mu_{\mathbf{c}} = \frac{1}{n_c}\sum_{i=1}^{n_c}\mathbf{x}_{c,i}$. For a test sample $\mathbf{x}_t$ we predict its label as the label of the class whose mean is closest to the test sample itself.
By doing this, the accuracy obtained with 50 dimensions is just 0.1% worse than the one obtained with 100 dimensions, which is still very similar to the one obtained without PCA.

The following code computes the PCA for a given dataset D and a fixed number of dimensions m. In particular, it takes the a dataset and the number of dimensions as parameters, and returns the dataset projected over the m-dimensional subspace.

```
    def PCA(D, m):
        # center the dataset in the origin
        mu = vcol(D.mean(1))
        DC = D - mu
        # compute the matrix
        C = numpy.dot(DC, DC.T) / D.shape[1]
        # get eigenvectors
        _, U = numpy.linalg.eigh(C)
        # keep only the first m eigenvectors
        P = U[:, ::-1][:, :m]
        # project the dataset over the m-dimensional subspace
        DP = numpy.dot(P.T, D)
        # return projected dataset
        return DP
```

3.3 Linear Discriminant Analysis

PCA is an unsupervised technique, so we don't have any guarantee of obtaining *discriminant* directions. In fact, the direction with the largest variance may not be an optimal choice.
As always we represent a direction as a unit vector $\mathbf{w}$, and the projected point is the scalar value $y = \mathbf{w}^T\mathbf{x}$.
To obtain a discriminant direction, one solution may be to take the direction of the line connecting the class menas. But if the data points of each class are scattered along the same direction of the mean, we still can't properly separate the classes.
To find a direction that has the best separation between classes, we measure spread between classes in terms of class **covariance**. The objective is to maximize the *between-class* variability over *within-class* variability ratio for the transformed samples.

$$\max_w \mathcal{L}(\mathbf{w}) = \max_w \frac{\mathbf{w}^T\mathbf{S}_B\mathbf{w}}{\mathbf{w}^T\mathbf{S}_W\mathbf{w}} \tag{4}$$

$\mathcal{L}(\mathbf{w})$ is the objective function. The between and within class variability matrices are

$$\mathbf{S}_B = \frac{1}{N}\sum_{c=1}^{K} n_c(\mu_c - \mu)(\mu_c - \mu)^T \tag{5}$$

$$\mathbf{S}_W = \frac{1}{N}\sum_{c=1}^{K}\sum_{i=1}^{n_c}(\mathbf{x}_{c,i} - \mu_c)(\mathbf{x}_{c,i} - \mu_c)^T \tag{6}$$

where $\mathbf{x}_{c,i}$ is the i-th sample of class c, n_c is the number of samples of class c, K is the total number of classes, N is the total number of samples, μ is the dataset mean and μ_c is the mean of class c. The between class covariance matrix can be interpreted as a (weighted) covariance matrix for the class means, while the within class covariance matrix can be seen as a (weighted) average of the covariance matrix of each class. The sum of $\mathbf{S}_B$ and $\mathbf{S}_W$ gives us the covariance matrix $\mathbf{C}$ of the entire dataset.

Let's now consider the between and within class variance of the projected samples $\mathbf{w}^T\mathbf{x}$. The global mean and class mean in direction $\mathbf{w}$ are $m = \mathbf{w}^T\mu$ and $m_c = \mathbf{w}^T\mu_c$. The between and within class covariance over the direction $\mathbf{w}$ can be obtained as $s_B = \mathbf{w}^T\mathbf{S}_B\mathbf{w}$ and $s_C = \mathbf{w}^T\mathbf{S}_C\mathbf{w}$. Also, for the sake of simplicity, let's assume that $\mathbf{S}_W$ is full rank. In this way, the objective function becomes $\mathcal{L}(\mathbf{w}) = \frac{s_B}{s_W}$.
To maximize the function w.r.t. $\mathbf{w}$ we can set $\nabla\mathcal{L}(\mathbf{w}) = 0$, from which

$$\mathbf{S}_W^{-1}\mathbf{S}_B\mathbf{w} = \mathcal{L}(\mathbf{w})\mathbf{w} \tag{7}$$

i.e. the optimal solution is an *eigenvector* of $\mathbf{S}_W^{-1}\mathbf{S}_B$ and the *eigenvalue* corresponding to the solution is the value of the ratio we want to maximize, which is the largest eigenvalue of $\mathbf{S}_W^{-1}\mathbf{S}_B$.

This method was originally introduced for binary problems, where the solution can be simply expressed as $\mathbf{w} \propto \mathbf{S}_W^{-1}(\mu_2 - \mu_1)$. Once the data has been reduced on a single dimension it is possible to classify it by checking if it is higher or lower than a certain *treshold*.
But LDA has also found large success as a dimensionality reduction technique: in this case we are intereset in looking for the m most discriminant directions. We represent these directions as a matrix $\mathbf{W}$ whose column are the horizontally stacked directions we want to find. Note that we do not require $\mathbf{W}$ to be orthogonal, since we are not interesent in how the data is scaled but only in its distribution along the directions. This time the projected points are computed as $\hat{\mathbf{x}} = \mathbf{W}^T\mathbf{x}$. The projected between and within class covariance matrices become $\hat{\mathbf{S}_B} = \mathbf{W}^T\mathbf{S}_B\mathbf{W}$ and $\hat{\mathbf{S}_W} = \mathbf{W}^T\mathbf{S}_W\mathbf{W}$. The criteria to generalize the 1-dimensional case becomes to maximize

$$\mathcal{L} = \mathrm{Tr}(\hat{\mathbf{S}_W}^{-1}\hat{\mathbf{S}_B}) \tag{8}$$

It can be shown that the solution is given by the m eigenvectors corresponding to the m largest eigenvalues of $\mathbf{S}_W^{-1}\mathbf{S}_B$.
Note: from the definition of $\mathbf{S}_B$, the number of non-zero eigenvalues is at most $C-1$, where C is the number of classes. So, LDA allows estimating at most $C-1$ directions.
It is often helpful to pre-process the data using PCA before applying LDA.
The following code can be used to apply LDA on a dataset D to find the m most discriminant directions.

```
def LDA(D, L, m):
    # compute matrices S_w, S_b with the following function
    S_w, S_b = compute_Sw_Sb(D, L)
    # compute eigenvectors of S_b^(-1) * S_w
    _, U = scipy.linalg.eigh(S_b, S_w)
    # take eigenvecs associated with the m biggest eigenvals
    W = U[:, ::-1][:, :m]
    # return the data projected on the m-dimensional subspace
    return numpy.dot(W.T, D)
```

```
def compute_Sw_Sb(D, L):
    # retrieve the number of classes | (0, 1, 2) -> 3
    num_classes = L.max()+1
    # separate the data into classes
    D_c = [D[:, L==i] for i in range(num_classes)]
    # get the number of elements for each class
    n_c = [D_c[i].shape[1] for i in range(num_classes)]
    # compute the dataset mean
    mu = vcol(D.mean(1))
    # compute the mean for each class
    mu_c = [vcol(D_c[i].mean(1)) for i in range(len(D_c))]
    # compute S_w and S_b as previously explained
    S_w, S_b = 0, 0
    for i in range(num_classes):
        DC = D_c[i] - mu_c[i]
        C_i = numpy.dot(DC, DC.T) / DC.shape[1]
        S_w += n_c[i] * C_i
        diff = mu_c[i] - mu
        S_b += n_c[i] * numpy.dot(diff, diff.T)
    S_w /= D.shape[1]
    S_b /= D.shape[1]
    # return S_w and S_b
    return S_w, S_b
```

Unfortunally, linear transformations are not always suited for our data, for example if we have 2D data spread in a circular shape. In this case we can try to remap the data to polar coordinates (ρ, θ) instead of cartesian (x, y) and then try to apply linear transformations to the remapped data.
There exist also non-linear alternatives which will not be discussed here.

To solve the generalized eigenvalue problem [(7)], we may use an alternative method which is the *joint diagonalization* of $\mathbf{S}_W$ and $\mathbf{S}_B$ that makes $\mathbf{S}_W$ become the identity matrix and $\mathbf{S}_B$ a diagonal matrix.
We first **whiten $\mathbf{S}_W$**. Whitening is procedure that makes features uncorrelated by transforming the covariance matrix in the identity matrix. However, we are not guaranteed that the transformed features will be significant for the dataset.
First we compute the eigen-value decomposition

$$\mathbf{S}_W = \mathbf{U}_W \mathbf{\Sigma}_W \mathbf{U}_W^T \tag{9}$$

and then we apply to the dataset the whitening transformation described by $\mathbf{P}_W = \mathbf{U}_W \mathbf{\Sigma}_W^{-\frac{1}{2}} \mathbf{U}_W^T$ so that $\mathbf{S}_W$ becomes $\mathbf{I}$ and $\mathbf{S}_B$ becomes $\mathbf{P}\mathbf{S}_B\mathbf{P}^T$. Then we diagonalize the transformed $\mathbf{S}_B$ by computing the eigen-value decomposition

$$\mathbf{P}\mathbf{S}_B\mathbf{P}^T = \mathbf{U}_B \mathbf{\Sigma}_B \mathbf{U}_B^T \tag{10}$$

and we diagonalize it by projecting the data over $\mathbf{U}_B^T$. So in the end $\mathbf{S}_W \rightarrow \mathbf{I}$ and $\mathbf{S}_B \rightarrow \mathbf{\Sigma}_B$. The first m directions of the transformaed samples correspond to the LDA subspace. The transformation we applied on the data is described by the matrix

$$\mathbf{W} = \mathbf{P}_W^T \mathbf{U}_B \tag{11}$$

4 Probability and density estimation

Our goal is to make predictions that allow us to take actions. If too many factors can influence the outcome of a phenomena, we can model the phenomena in terms of random events. To descrive a random event we use **probability**. Probability can be seen as:

- The fraction of favorable outcomes of an event (*classical interpretation*).
- The frequency of an outcome under a large number of trials (*frequentist interpretation*).
- The *measure* of belief that an event will occur (*Bayesian interpretation*).

To understand the last one, think about a political election. If today a certain party wins, it's very likely that it will win again if I repeat the same election tomorrow.

From now on, we will refer to Σ as a set of possible outcomes and $\mathcal{A}$ as a σ-field over Σ (a σ-field is a collection of subsets of Σ including the empty set).

4.1 Probability

Probebility is a function $P : \mathcal{A} \to \mathbb{R}^+$ such that $P(\Omega) = 1$ and for which the additivity property holds. This means that summing the probability of disjoint events will result in 1. The triplet $(\Sigma, \mathcal{A}, P)$ is called a *probability space*, whose properties won't be summarized here.
Conditional probability is the probability that an event A occures, knowing that B happened:

$$P(A|B) = \frac{P(A,B)}{P(B)} \tag{12}$$

Where $P(A,B)$ is the joint probability of A and B. The *Bayes formula* holds for conditional probability:

$$P(B|A) = P(A|B)\frac{P(B)}{P(A)} \tag{13}$$

Two events are said to be **independent** when $P(A,B) = P(A)P(B)$.

4.2 Random variables

Random variables (RV) allow extending probabilistic reasoning to quantities that depend on events. A random variable X is defined as a function $X : \Omega \to \mathbb{R}$, such that $\forall x \in \mathbb{R}$ the event $\{\omega \in \Omega : X(\omega) \leq x\}$ belongs to $\mathcal{A}$.
Essentially X is a function of the outcomes ω for which we can compute the probability that it takes values no greater than x:

$$P(X \leq x) = P(\{\omega \in \Omega : X(\omega) \leq x\}) \tag{14}$$

Let X be a random variable: its **cumulative distribution function** is defined as

$$F_X(x) = P(X \leq x) \tag{15}$$

The cumulative distribution function is always a number in the range $[0, 1]$ and it is *non-decreasing*, it tends to 0 at $-\infty$ and to 1 at $+\infty$ and it is *right-continuous*. Because of this, we have that $P(a < X \leq b) = F_X(b) - F_X(a)$.

4.2.1 Discrete random variables

A random variable is said to be **discrete** if it takes a *finite* or *countably infinite* number of values. For DRVs we can define the **probability mass function** or **discrete density** as

$$f_X(x) = P(X = x) \tag{16}$$

This function is always non-negative, and it is equal to 0 for all x except for those values taken by X (the *support* of X). It turns out that we can exploit the density function to compute the c.d.f. as

$$F_X(x) = \sum_{t \leq x} f_X(t) \tag{17}$$

For example, consider the problem of tossing a coin until we get head. Let p be the probability to get a head (it's equal to 0.5 for fair coins), and let's assume that each toss is independent from the others. The set of possible events is $\Omega = \{(H),\ (TH)\ (TTH),\ \dots\}$, which contains a countably infinite number of values. The probability of the events to occur is respectively $\{\frac{1}{2},\ \frac{1}{4},\ \frac{1}{8},\ \dots\}$. Now, let X be the function that maps each element of Ω to the number of tails it requires before getting a head, respectively $\{0,\ 1,\ 2,\ \dots\}$. Then, we can define the density function of X as

$$f_X(x) = \begin{cases} p(1-p)^x & x \in \mathbb{N}^+ \\ 0 & \text{otherwise} \end{cases} \tag{18}$$

4.2.2 Continuous random variables

A random variable for which the c.d.f. is a continuous function will be referred to as **continuous random variable**. If F_X is continuous, then $P(X = x) = 0$. In this case, we define the **density function** as a function $f : \mathbb{R} \to \mathbb{R}$ which is always non-negative, integrable over $\mathbb{R}$ and such that

$$\int_{-\infty}^{+\infty} f(x)dx = 1 \tag{19}$$

Thanks to this we can define F_X as

$$F_X(x) = \int_{-\infty}^{x} f_X(t)dt \tag{20}$$

and if F_X is differentiable, then f_X is the derivative of F_X with respect to x.

Given two (discrete or continuous) random variables X and Y, we define the **conditional density** of X given Y as

$$f_{X|Y}(x|y) = \frac{f_{X,Y}(x,y)}{f_Y(y)} \tag{21}$$

The Bayes rule applies here too.

4.2.3 Random vectors

An m-dimensional **random vector** $\mathbf{X} = (X_1, \dots, X_m)$ is a vector whose components are random variables. The c.d.f. of $\mathbf{X}$ is defined as

$$F_{\mathbf{X}}(\mathbf{x}) = P(X1 \leq x1, \dots, X_m \leq x_m) \tag{22}$$

which is the *joint* cumulative distribution for $X_1, \dots, X_m$.

4.2.4 Transformations of random variables

Sometimes we are interested in knowing the distribution of a function of a random variable, $Y = g(X)$.
If the random variable is discrete, the solution is straightforward:

$$f_Y(y) = \sum_{x|g(x)=y} f_X(x) \tag{23}$$

For continuous random variable, the solution is more complex, but it can be shown that if g is monotonic, differentiable and with differentiable inverse, then

$$f_Y(y) = f_X(g^{-1}(y)) \frac{d}{dy} g^{-1}(y) \tag{24}$$

This can be extended to random vectors, where if $\mathbf{Y} = g(\mathbf{X})$, then

$$f_{\mathbf{Y}}(y) = f_{\mathbf{X}}(g^{-1}(\mathbf{y}) |\det \mathbf{J} g^{-1}(\mathbf{y})| \tag{25}$$

where $\mathbf{J} g^{-1}(\mathbf{y})$ is the *Jacobian* matrix of g^{-1}.

Many times we are interested in the function $Z = X + Y$. In this case

$$F_Z(z) = \int_{\{(x,y)|x+y<z\}} f_{X,Y}(x,y)dxdy \tag{26}$$

$$f_Z(z) = \int f_{X,Y}(x, z-x)dx = \int f_{X,Y}(z-y, y)dy \tag{27}$$

4.2.5 Expectations

The expectation is a *linear operator*.
We define the **mean** or **expected value** of a random variable as

$$\mathbb{E}_X(X) = \begin{cases} \displaystyle\sum_{x \in \mathcal{S}} x f_X(x) & \text{discrete RV} \\ \displaystyle\int_{\mathcal{S}} x f_X(x) dx & \text{continuous RV} \end{cases} \tag{28}$$

In general, the **k-th moment** of X as

$$\mathbb{E}_X(X^k) = \begin{cases} \displaystyle\sum_{x \in \mathcal{S}} x^k f_X(x) & \text{discrete RV} \\ \displaystyle\int_{\mathcal{S}} x^k f_X(x) dx & \text{continuous RV} \end{cases} \tag{29}$$

The **variance** of a RV is given by

$$\operatorname{var} X = \mathbb{E}_X[(X - \mathbb{E}_X(X))^2] = \mathbb{E}_X(X^2) - \mathbb{E}_X(X)^2 \tag{30}$$

The *standard deviation* is the square root of the variance.

It is also possible to define the expectation for a function $g(X)$ (substituite $g(X)$ to x in the previous formulas) and for a random vector (as a vector of expectations).
If X and Y are two RVs, the **covariance** of X and Y is

$$\operatorname{cov}(X, Y) = \mathbb{E}_{X,Y}(XY) - \mathbb{E}_X(X)\mathbb{E}_Y(Y) = \operatorname{cov}(Y, X) \tag{31}$$

If the covariance is equal to 0, the random variables are *uncorrelated.* From the covariance we can define also the *Pearson correlation coefficient* as

$$\operatorname{corr}(X, Y) = \frac{\operatorname{cov}(X, Y}{\sqrt{\operatorname{var}(X) \operatorname{var}(Y)}} \tag{32}$$

The correlation is a value in the range $[-1,\ 1]$ and if it is equal to 1, then $Y = aX + b$ for some $a \neq 0, b \in \mathbb{R}$.
Finally, for a random vector we can define a **covariance matrix** as

$$\Sigma = \operatorname{cov}(\mathbf{X}) = \mathbb{E}[(\mathbf{X} - \mathbb{E}(\mathbf{X}))(\mathbf{X} - \mathbb{E}(\mathbf{X})^T)] \tag{33}$$

that results in $\begin{bmatrix} \operatorname{var}(\mathbf{X_1}) & \operatorname{cov}(\mathbf{X_1}, \mathbf{X_2}) & \dots & \operatorname{cov}(\mathbf{X_1}, \mathbf{X_m}) \\ \operatorname{cov}(\mathbf{X_2}, \mathbf{X_1}) & \operatorname{var}(\mathbf{X_2}) & \dots & \operatorname{cov}(\mathbf{X_2}, \mathbf{X_m}) \\ \vdots & \vdots & \ddots & \vdots \\ \operatorname{cov}(\mathbf{X_m}, \mathbf{X_1}) & \operatorname{cov}(\mathbf{X_m}, \mathbf{X_2}) & \dots & \operatorname{var}(\mathbf{X_m}) \end{bmatrix}.$

This matrix is symmetric and positive semi-definite.

4.3 Known discrete random variables

4.3.1 Bernoulli distribution

The Bernoulli distribution can be used to model the outcome of a binary event, like the tossing of a coin. Let $X \in \{0, 1\}$ be a binary random variable that takes value 1 with probability p. Its distribution is defined as $X \sim \text{Ber}(p)$ where

$$P_X(x) = \text{Ber}(x|p) = \begin{cases} p & \text{if } x = 1 \\ 1-p & \text{if } x = 0 \end{cases} = p^x(1-p)^{1-x} \tag{34}$$

4.3.2 Binomial distribution

Suppose we want to count the number of successes in n repeated trials (for example n coin tosses). Let p denote the probability of success for a single trial and let X be a random variable that tells the probability of getting x successes over n trials. The distribution of X is the *Binomial* distribution $X \sim \text{Bin}(n, p)$, where

$$P_X(x) = \binom{n}{x} p^x(1-p)^{n-x} \tag{35}$$

We can observe that $\text{Ber}(p) \sim \text{Bin}(1, p)$, so the Binomial distribution is a generalization of the Bernoulli distribution.

4.3.3 Categorical distribution

The Bernoulli distribution can be extended to events that have K possible outcomes, for example a dice roll. Let X be a *Categorical* distribution $X \sim \text{Cat}(\mathbf{p})$, where $p = (p_1, p_2, \dots, p_K)$ be the vector probability that indicates the probability of each of the K outcomes.
We will usually represent outcomes as a 1-of-K encoding vector $\mathbf{X}$ which has all zeros except for the one outcome that is indicated with a one. For example $X = 2 \Rightarrow \mathbf{X} = (0, 1, \dots, 0)$. By using this representation we can express the density function as

$$f_{\mathbf{X}}(\mathbf{x}) = \prod_{i=1}^{K} p_i^{x_i} \tag{36}$$

4.3.4 Multinomial distribution

Also the categorical distribution can be generalized to a set of n trials, encoded as $\mathbf{x} = (x_1, \dots, x_K)$ where x_i is the number of occurrencies of outcome i. Obviously the sum of the elements of x is n.
In this situation, $\mathbf{X}$ follows a *Multinomial* distribution $\mathbf{X} \sim \text{Mul}(n, \mathbf{p})$

$$f_{\mathbf{X}}(\mathbf{x}) = \frac{n!}{x_1! \dots x_K!} \prod_{i=1}^{K} p_i^{x_i} \tag{37}$$

We can observe that $\text{Mul}(1, \mathbf{p}) \sim \text{Cat}(\mathbf{p})$ and that the sum of n Categorical distributions is a Multinomial distribution (this follows from the way we represent $\mathbf{x}$).

4.4 Gaussian distribution

The **Gaussian** or **normal** distribution is probably the most employed example of continuous distributions. Let X be a Gaussian distribution with *mean* μ and *variance* σ^2, $X \sim \mathcal{N}(\mu, \sigma^2)$. Then

$$f_X(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{38}$$

If $X \sim \mathcal{N}(0, 1)$ then we way that X follows a *standard* normal distribution.
This distribution is symmetric, centered around μ, and higher variance corresponds to a *flatter* density.
The **central limit** theorem says that a sequence of independent and identically distributed random variables, with same mean μ and same variance σ^2 sums up to the standard normal distribution $\mathcal{N}(0, 1)$.

The Gaussian distribution can be extended to random vectors. Let $\mathbf{X}$ be a random vector of *standard normal* distributed random variables. The distribution of $\mathbf{X}$ is given by the joint distribution of its components

$$f_{\mathbf{X}}(\mathbf{x}) = \prod_{i=1}^{N} f_{X_i}(x_i) = (2\pi)^{-\frac{N}{2}} e^{-\frac{1}{2}\mathbf{x}^T\mathbf{x}} \tag{39}$$

and we say that $\mathbf{X}$ follows a **standard multivariate normal** distribution $\mathbf{X} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$.
A variable $\mathbf{X}$ that can be written as a linear transformation of a standard multivariate normal distributed random vector $\mathbf{Y}$ ($\mathbf{X} = \mathbf{A}\mathbf{Y} + \boldsymbol{\mu}$), follows a **multivariate Gaussian** distribution (MVG) with mean $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$, where $\boldsymbol{\Sigma} = \mathbf{A}\mathbf{A}^T$. In this case

$$f_{\mathbf{X}}(\mathbf{x}) = (2\pi)^{-\frac{N}{2}} |\boldsymbol{\Sigma}|^{-\frac{1}{2}} e^{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})} \tag{40}$$

To avoid numerical issues due to exponentiation of large numbers, in many practical cases it's more convenient to work with the logarithm of the density

$$\log \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = -\frac{N}{2}\log 2\pi - \frac{1}{2}\log|\boldsymbol{\Sigma}| - \frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}) \tag{41}$$

The following code can be used to compute the above function. It isn't the most optimized version since it uses a for loop which is pretty slow in Python. A more efficient way of computing the same function would be to exploit broadcasting, avoiding the loop.
Since the code computes the log of the distribution, if you want to plot the result just be sure to print its exponential.

```
# X is the data, mu its mean and C the covariance matrix
def logpdf_GAU_ND(X, mu, C):
    Y = []
    # get the number of features
    N = x.shape[0]
    # for each input data
    for x in X.T:
        x = vcol(x)
        # compute the constant term
        const = N*numpy.log(2*numpy.pi)
        # compute the second term
        logC = numpy.linalg.slogdet(C)[1]
        # compute the third term
        mult = numpy.dot(numpy.dot((x-mu).T, numpy.linalg.inv(C
                                   )), (x-mu))[0,0]
        # append the result of the function for this input data
        Y.append(-0.5*(a+b+c))
    # return the result array
    return numpy.array(Y)
```

4.5 Density estimation

In this subsection, let's consider an example: we want to predict whether a coin flip will result in head (H) or tail (T), but we don't know whether the coin is biased or not. However, we have observed a number of tosses n. In this example we will also assume that all the tosses are independent from each other, and identically distributed.
Let's denote the observed results as $(x_1, \dots, x_n)$, where $x_i = 1$ if we got a head in toss i and $x_i = 0$ otherwise. These can be considered outcomes of RVs $(X_1, \dots, X_n)$.
To predict if a new toss X_t will result in head, we want to model the distribution $X_t|(X_1 = x1, \dots, X_n = x_n)$.
If we knew that $P(H) = \pi$, then we could model $X_i \sim X \sim \text{Ber}(\pi)$, and also $X_t \sim X \sim \text{Ber}(\pi)$. The problem is that we don't know π, so we have to *estimate* it.

4.5.1 Frequentist approach

With this approach we assume that a *true* value π_T exists. To estimate a good π we look for the value that best explains the observed tosses. We thus define the **likelihood** function

$$\mathcal{L}(\pi) = f_{X_1 \dots X_n|\pi}(x_1 \dots x_n|\pi) = P(X_1 = x1 \dots X_n = x_n|\pi) \tag{42}$$

Since we are assuming that all the tosses are independent, and each toss is modeled as a Bernoulli random variable with $p = \pi$, the likelihood function becomes

$$\mathcal{L}(\pi) = \prod_{i=1}^{n} P(X_i = x_i|\pi) = \prod_{i=1}^{n} \pi^{x_i}(1-\pi)^{1-x_i} \tag{43}$$

To compute the *Maximum Likelihood* estimate (ML) for π, we can consider the logarithm of the likelihood function (for convenience) and set its derivative equal to zero, from which we obtain that

$$\pi^*_{ML} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{44}$$

At this point, we can say that

$$P(X_t = 1 | X_1 = x_1 \dots X_n = x_n) = \pi^*_{ML} \tag{45}$$

If we have *enough* observations (more is better) our estimation will often be good. But if we have a poor dataset we could get wrong estimations because this method could not consider some possible outcomes which are less likely. For example if we observe just 3 coin tosses which give H as result, we would have $\pi^*_{ML} = 1$.

An alternative to the frequentist approach is the **Bayesian approach**, that won't be covered in this course. Just to get the concept, let's say that the Bayesian approach requires us to specify our knowledge about the possible values that π may assume in terms of a *prior* distribution $f_\Pi(\pi)$. In this way it's possible to consider also outcomes that didn't happen in out observations. Combining the prior distribution with the likelihood we can compute the *posterior* distribution for the model parameters.
Although this method has some limitations, like the fact that the prior distribution is somewhat arbitrary and the computation of the posterior may be intractable, it provides better results for scenarios where data is scarce.

4.5.2 Density estimation with Gaussian distributions

When modeling continuous values, Gaussian distributions arise naturally in a wide variety of contexts (for example data histograms often present a Gaussian-like shape). If we consider that working with Gaussian density is easy, then it's reasonable to assume that (in many cases) our data has been generated by Gaussian RVs.
Let's assume we have some data $\mathcal{D} = (x_1, \dots, x_n)$ which we decide to model as samples of a Gaussian distribution with mean μ and variance ν. The *precision* is the inverse of the variance, $\lambda = \nu^{-1}$. We also assume that the points have been generated by independent, identically distributed RVs $X_i \sim X$.
The model parameters are $\theta = (\mu, \nu)$, and the distribution of X is $f_{X|\theta}(x) = \mathcal{N}(x|\mu, \nu)$.
As for the discrete case, we can express the *likelihood* function for θ as the joint density of all the observations

$$\mathcal{L}(\theta) = \prod_{i=1}^{n} f_{X_i|\theta}(x_i) = \prod_{i=1}^{n} \mathcal{N}(x_i|\mu, \nu) \tag{46}$$

Let's use a frequentist approach again, assuming the existence of *true* but unknown values for the model parameters θ. If we knew μ_T, ν_T, then the density for unseen samples X_T would be $f_{X_t}(x_t) = \mathcal{N}(x_t|\mu_T, \nu_T)$. But again, we don't know these values, so we need to estimate them.
An **estimator** is a function T of the data, that maps our dataset $\mathcal{D}$ to values for the model parameters, $\theta^* = T(\mathcal{D})$. We need a *consistent* estimator, which is an estimator that converges to the *true* distribution parameters θ_T as the sample size grows.
A simple way to produce consistent estimators is the **method of moments** (MOM), which consists in matching the moments of the assumed distribution to those of the data. For the normal distribution the moments are μ and ν. Thus, we can say that

$$\begin{cases} \mu^*_{MOM} = \dfrac{1}{n}\displaystyle\sum_{i=1}^{n} x_i \\ \nu^*_{MOM} = \dfrac{1}{n}\displaystyle\sum_{i=1}^{n}(x_i - \mu^*_{MOM})^2 \end{cases} \tag{47}$$

The MOM approach doesn't produce very accurate estimators, in general.
An other estimator is the one we used in 4.5.1, the **Maximum Likelihood** (ML) estimator, defined as the value that maximizes the likelihood

$$\theta^*_{ML} = \arg\max_{\theta} \mathcal{L}(\theta) \tag{48}$$

Again, it's convenient to work with the logarithm of the likelihood function, since the log operator is monotonically increasing. So we have that $\arg\max_\theta \mathcal{L}(\theta) = \arg\max_\theta \log \mathcal{L}(\theta)$, where we will indicate $\log \mathcal{L}(\theta)$ as $l(\theta)$.
The ML estimate can be obtained by solving

$$\begin{cases} \frac{\partial l}{\partial \mu} = 0 \\ \frac{\partial l}{\partial \lambda} = 0 \end{cases} \tag{49}$$

that gives us

$$\begin{cases} \mu^*_{ML} = \dfrac{1}{n}\displaystyle\sum_{i=1}^{n} x_i \\ \nu^*_{ML} = (\lambda^*_{ML})^{-1} = \dfrac{1}{n}\displaystyle\sum_{i=1}^{n}(x_i - \mu^*_{ML})^2 \end{cases} \tag{50}$$

In this case, the solution is the same as the one obtained by the MOM approach, but in general this doesn't hold.
From a practical point of view, it is possible to check that these parameters correspond to those which maximize the (log) likelihood by computing the likelihood value with different parameters, and do some comparisons. Calculating the log likelihood is trivial once we have computed the log MVG:

```
def loglikelihood(X, mu, C):
    return logpdf_GAU_ND(X, mu, C).sum()
```

5 Generative Models (linear & quadratic)

We consider a (closed set) classification problem. We have a pattern $\mathbf{x}_t$ that we want to classify as belonging to one of k classes. With the *probabilistic* model, we assume that $\mathbf{x}_t$ is a realization of RV $\mathbf{X}_t$ and we also assume that its class label can be described by a RV $C_t \in \{1, \dots, k\}$.
The **optimal Bayes decision** is to assign the class with highest *posterior probability* $c_t^* = \arg\max_c P(C_t = c|\mathbf{X}_t = \mathbf{x}_t)$. To do this, we must compute the probability that the class for the test sample t is c conditioned on the observed value $\mathbf{x}_t$, for all labels $c \in \{1 \dots K\}$.
We will assume that the samples are independent and distributed according to $(\mathbf{X}_t, C_t) \sim (\mathbf{X}, C)$ for any test sample t, which means that train and test data have the same distribution. Let $f_{\mathbf{X},C}$ be the joint density of $\mathbf{X}, C$: we can compute the joint likelihood for the hypothesized class c for the observed test sample $\mathbf{x}_t$ as $f_{\mathbf{X},C}(\mathbf{x}_t, c)$ and then use Bayes rule to compute the class *posterior probability*

$$P(C_t = c|\mathbf{X}_t = \mathbf{x}_t) = \frac{f_{\mathbf{X},C}(\mathbf{x}_t, c)}{\sum_{c' \in C} f_{\mathbf{X},C}(\mathbf{x}_t, c')} \tag{51}$$

The joint density for $(\mathbf{X}_t, C_t)$ can be expressed as $f_{\mathbf{X}_t,C_t}(\mathbf{x}_t, c) = f_{\mathbf{X}|C}(\mathbf{x}_t|c)P(c)$ where $P(c)$ is the *prior* probability for class c and represents the probability of the class being c before we observe the test sample. The prior probability is application-dependent and shouldn't be included in the model. Our objective is to model the class-conditional distribution $\mathbf{X}|C$, i.e. we want to estimate the density $f_{\mathbf{X}|C}(\mathbf{x}_t|c)$. To do this, we will assume that the data of each class can be modeled by a (Multivariate) Gaussian Distribution.

For example, if we are trying to do gender inference where the data is just a set of height measurements, we can fit a Gaussian density over the samples of each class (C=M, C=F). To fit the densities we may use ML estimates. At this point we can easily obtain the value $f_{\mathbf{X}|C}(\mathbf{x}_t|c)$, and multiply it by the prior probability of class c, $P(c)$, to obtain $f_{\mathbf{X},C}(\mathbf{x}_t, c)$. At this point we would need to divide the obtained values by $P(\mathbf{x}_t)$, which is the sum at the denominator of (48), but since we are dealing with a binary problem we can just compute the *class posterior ratio* for the two hypotheses:

$$\frac{P(C = M|\mathbf{x}_t)}{P(C = F|\mathbf{x}_t)} = \frac{f_{\mathbf{X},C}(\mathbf{x}_t, M)}{f_{\mathbf{X},C}(\mathbf{x}_t, F)} \tag{52}$$

Actually, we can avoid computing the denominator whenever we are just interested in doing a comparison between the class posterior probabilities.

5.1 Multivariate Gaussian Classifier

5.1.1 Standard version

We are now going to formalize the method we just employed in the example. We assume that our data, given the class, can be described by a Gaussian distribution.

$$(\mathbf{X}_t|C_t = c) \sim (\mathbf{X}|C = c) \sim \mathcal{N}(\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) \tag{53}$$

We have one mean and one covariance matrix per class. If we knew $\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c$ then we could compute $f_{\mathbf{X}_t|C_t=c}$ as

$$f_{\mathbf{X}_t|C_t}(\mathbf{x}_t|c) = \mathcal{N}(\mathbf{x}_t|\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) \tag{54}$$

However, we don't know the model parameters $\boldsymbol{\theta} = [(\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1), \ldots, (\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)]$. On the other hand, we have at our disposal a *labeled training* dataset

$$\mathcal{D} = \{(\mathbf{x}_1, c_1), \ldots, (\mathbf{x}_n, c_n)\} \tag{55}$$

where $\mathbf{x}_i$ is the i-th observed sample, which has label $c_i \in \{1, \ldots, k\}$.
From now on, we will assume that given the model parameters $\boldsymbol{\theta}$, all the observations are *independent and identically distributed* (i.i.d.), so

$$[(\mathbf{X}_i, C_i) \perp\!\!\!\perp (\mathbf{X}_j, C_j)] \text{ and } (\mathbf{X}_i, C_i)|\boldsymbol{\theta} \sim (\mathbf{X}, C)|\boldsymbol{\theta} \tag{56}$$

Also, since we assume Gaussian distribution for $\mathbf{X}|C$, we have

$$(\mathbf{X}|C = c, \boldsymbol{\theta}) \sim \mathcal{N}(\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) \tag{57}$$

i.e., the class-conditional distribution for all observations is a Gaussian with class-dependent mean $\boldsymbol{\mu}_c$ and class-dependent covariance matrix $\boldsymbol{\Sigma}_c$. To estimate these model parameters, we follow a frequentist approach. We have already seen that a possible way to estimate the model parameters is to maximize the data (log-)likelihood. The likelihood for $\boldsymbol{\theta}$ is

$$\begin{aligned}\mathcal{L}(\boldsymbol{\theta}) &= f_{\mathbf{X}_1\ldots\mathbf{X}_n, C_1\ldots C_n|\boldsymbol{\theta}}(\mathbf{x}_1 \ldots \mathbf{x}_n, c_1 \ldots c_n|\boldsymbol{\theta}) \\ &= \prod_{i=1}^{n} f_{\mathbf{X},C|\boldsymbol{\theta}}(\mathbf{x}_i, c_i|\boldsymbol{\theta}) = \prod_{i=1}^{n} \mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) P(c_i)\end{aligned}$$

Thus, the log-likelihood is

$$l(\boldsymbol{\theta}) = \log \mathcal{L}(\boldsymbol{\theta}) = \sum_{i=1}^{n} \log \mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) + \sum_{i} \log P(c_i)$$

We can ignore the last term of this equation since we are just interesting in maximizing the likelihood with respect to $\boldsymbol{\theta}$ and it is a constant term. By splitting the first term over all the classes we obtain

$$l(\boldsymbol{\theta}) = \sum_{c=1}^{k} \sum_{i|c_i=c} \log \mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) + \xi = \sum_{c=1}^{k} l_c(\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) + \xi \tag{58}$$

This means that we can just maximize the log-likelihood independently for each class, which is something we already know how to do. In fact, by doing all the math, we will obtain

$$\begin{cases} \boldsymbol{\mu}_c^* = \dfrac{1}{N_c} \displaystyle\sum_{i|c_i=c} \mathbf{x}_i \\ \boldsymbol{\Sigma}_c^* = \dfrac{1}{N_c} \displaystyle\sum_{i|c_i=c} (\mathbf{x}_i - \boldsymbol{\mu}_c^*)(\mathbf{x}_i - \boldsymbol{\mu}_c^*)^T \end{cases} \tag{59}$$

Then, we can use this result to compute for each class c its likelihood for test point $\mathbf{x}_t$ as $f_{\mathbf{X}_t|C_t}(\mathbf{x}_t, c) = \mathcal{N}(\mathbf{x}_t|\boldsymbol{\mu}_c^*, \boldsymbol{\Sigma}_c^*)$.

Let's now consider a binary task with two classes $C \in \{h_1, h_0\}$. We assign the label to a test sample $\mathbf{x}_t$ according to the highest posterior probability, comparing $P(C = h_1|\mathbf{x}_t)$ to $P(C = h_0|\mathbf{x}_t)$. We can express the comparison in terms of *class posterior (log-)ratio.*

$$\log r(\mathbf{x}_t) = \log \frac{P(C = h_1|\mathbf{x}_t)}{P(C = h_0|\mathbf{x}_t)} = \log \frac{f_{\mathbf{X}|C}(\mathbf{x}_t|h1)}{f_{\mathbf{X}|C}(\mathbf{x}_t|h0)} + \log \frac{P(C = h_1)}{P(C = h_0)} \tag{60}$$

We can see that the posterior log-ratio depends on the likelihoods $f_{\mathbf{X}|C}(\mathbf{x}_t|c)$ and on the prior class probabilities. We also call the first element of the sum *log-likelihood ratio*, $llr(\mathbf{x}_t)$, which represents the ratio between the likelihood of observing the sample given that it belongs to h_1 or to h_0. The second term of the sum represents the *prior (log-)odds*, and since we are treating a binary problem, we have that $P(C = h_1) = \pi,\ P(c = h_0) = 1 - \pi$, so

$$\log r(\mathbf{x}_t) = \log \frac{f_{\mathbf{X}|C}(\mathbf{x}_t|h1)}{f_{\mathbf{X}|C}(\mathbf{x}_t|h0)} + \log \frac{\pi}{1-\pi} \tag{61}$$

Our system should just focus on providing the first term of the log-likelihood ratio, and whoever will use the system will just plug the prior probabilities for his own task, and compute posterior log-probability ratios. The optimal class decision is based on the comparison $\log r(\mathbf{x}_t) \lesseqgtr 0$, i.e.

$$llr(\mathbf{x}_t) \lesseqgtr -\log \frac{\pi}{1-\pi} \tag{62}$$

The log-likelihood ratio acts as a *score*, with probabilistic interpretation. Greater scores values imply our system favors class h_1, lower values mean it favors class h_0. The decision requires comparing the *llr* to a treshold depending on the application.

At this point, we want to understand what kind of decision surface we have found, i.e. what is the shape of the surface that separates our data. By computing the log-likelihood ratio we find out that it is a *quadratic* function in $\mathbf{x}$

$$llr(\mathbf{x}) = \mathbf{x}^T \mathbf{A} \mathbf{X} + \mathbf{x}^T \mathbf{b} + c \tag{63}$$

where $\mathbf{A},\ \mathbf{b},\ c$ are combinations of the model parameters $(\boldsymbol{\mu}, \boldsymbol{\Lambda}) = (\boldsymbol{\mu}, \boldsymbol{\Sigma}^{-1})$. This means that the *llr* can be either a *parabola*, an *ellipsis* or an *hyperbole*.

For multiclass problems $C \in \{h_1, h_2, \dots, h_k\}$ we can compute *closed-set posterior probabilities* as

$$P(C = h|\mathbf{x}_t) = \frac{f_{\mathbf{X}|C}(\mathbf{x}_t|h)P(h)}{\sum_{h'} f_{\mathbf{X}|C}(\mathbf{x}_t|h')P(h')} \tag{64}$$

So all the posterior probabilities are proportional to $f_{\mathbf{X}|C}(\mathbf{x}_t|h)P(h)$, and the optimal decision can be computed as

$$c_t^* = \arg\max_h f_{\mathbf{X}|C}(\mathbf{x}_t|h)P(h) = \arg\max_h [\log f_{\mathbf{X}|C}(\mathbf{x}_t|h) + \log P(h)] \tag{65}$$

Again, the first term should be the output of the classifier, whereas the second term depends on the application.

5.1.2 Naive Bayes

The standard Multivariate Gaussian Classifier requires computing mean and covariance matrix for each class. If the samples are few compared to thei dimensionality, then the estimates can be inaccurate. The issue is evident for covariance matrices, since they have $\frac{D\times(D+1)}{2}$ independent elements. There is also an evident performance issue: what if $D = 10^6$? We would have problems not only in computing the matrix, but also in storing it.
But if we can assume that the components of the feature vectors are (almost) uncorrelated, we can say that our covariance matrix would have all 0 elements except on the diagonal.
Also, if we know that for each class the different components are approximately independent, we can simplify the estimate assuming that the distribution of $\mathbf{X}|C$ can be factorized over its components

$$f_{\mathbf{X}|C}(\mathbf{x}|c) \sim \prod_{j=1}^{D} f_{X_{[j]}|C}(x_{[j]}|c) \tag{66}$$

where $x_{[j]}$ is the j-th component of $\mathbf{x}$ (not to be confused with $\mathbf{x}_j$, the j-th dataset sample).
This model is called **Naive Bayes**. The Naive assumption, convined with Gaussian assumption, models the distribution of each component as a univariate Gaussian. We can again compute the ML estimates. In this case the log-likelihood factorizes over sample components

$$l(\boldsymbol{\theta}) = \xi + \sum_{j=1}^{D}\sum_{c=1}^{k}\sum_{i|c_i=c} \log \mathcal{N}(\mathbf{x}_{i,[j]}|\mu_{c,[j]}, \sigma^2_{c,[j]}) \tag{67}$$

So we can optimize the log-likelihood independently for each component, obatining

$$\begin{cases} \mu^*_{c,[j]} = \dfrac{1}{N_c}\displaystyle\sum_{i|c_i=c} \mathbf{x}_{i,[j]} \\ \sigma^2_{c,[j]} = \dfrac{1}{N_c}\displaystyle\sum_{i|c_i=c} (\mathbf{x}_{i,[j]} - \mu_{c,[j]})^2 \end{cases} \tag{68}$$

This solution is perfectly equivalent to the one we found for the Multivariate Gaussian Classifier, assuming that the covariance matrix is a *diagonal* matrix.

5.1.3 Tied Classifier

Another common Gaussian model assumes that the covariance matrices of the different classes are *tied.* This means that the noise is class-independent, so the distribution of class samples around the class mean is the same. Because of this we can assume that

$$f_{\mathbf{X}|C}(\mathbf{x}|c) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_c, \boldsymbol{\Sigma}) \tag{69}$$

where the covariance matrix is the same for all classes. By estimating the parameters using the ML framework, we obtain that

$$\begin{cases} \boldsymbol{\mu}_c^* = \dfrac{1}{N_c} \displaystyle\sum_{i|c_i=c} \mathbf{x}_i \\ \boldsymbol{\Sigma}^* = \dfrac{1}{N} \displaystyle\sum_{c} \sum_{i|c_i=c} (\mathbf{x}_i - \boldsymbol{\mu}_c)(\mathbf{x}_i - \boldsymbol{\mu}_c)^T \end{cases} \tag{70}$$

If we compute the *binary* log-likelihood ratio for the tied model, we obtain that

$$llr(\mathbf{x}) = \mathbf{x}^T\mathbf{b} + c \tag{71}$$

where $\mathbf{b}$ and c are combinations of $\boldsymbol{\Lambda} = \boldsymbol{\Sigma}^{-1}$ and $\boldsymbol{\mu}_c$. This means that with the tied covariance assumptions, the separation line between classes is linear instead of quadratic.
This model is also closely related to LDA. Remember that two-class LDA looks for the direction which maximizes the generalized Rayleigh quotient

$$\frac{\mathbf{w}^T\mathbf{S}_B\mathbf{w}}{\mathbf{w}^T\mathbf{S}_W\mathbf{w}} \tag{72}$$

with $\mathbf{S}_B = \boldsymbol{\Lambda}^-1 = \boldsymbol{\Sigma}$ and $\mathbf{S}_B = (\boldsymbol{\mu_1} - \boldsymbol{\mu}_0)(\boldsymbol{\mu_1} - \boldsymbol{\mu}_0)^T$. We already saw how to solve this problem, so we won't do it again. In the end we obtain that the projection over the LDA subspace is, up to a scaling factor, given by

$$\mathbf{w}^T\mathbf{x} = k \cdot \mathbf{x}^T\boldsymbol{\Lambda}(\boldsymbol{\mu_1} - \boldsymbol{\mu_0}) \tag{73}$$

Note: the difference with the standard LDA is that in that case we don't have a rule to find a discriminant *treshold* which is necessary to classify the data. In this case, instead, we know from theory that the optimal treshold depends on the class *prior* probabilities

$$t = \log\frac{\pi}{1-\pi} \tag{74}$$

Note: From all of this we can understand why LDA assumes that all classes have the same within-class covariance.

5.1.4 Considerations about Gaussian Classifiers

In general, we can say that

- If data is high-dimensional, PCA can simplify the estimation
- PCA also allows removing dimensions with very small variance (e.g. in the MNIST dataset, pixels that are white for all images regardless of the digit)
- Multivariate models perform better if we have enough data to reliably estimate the covariance matrices
- Naive Bayes can simplify the estimation, but may perform poorly if data are highly correlated
- Tied covariance models can capture correlations, but may perform poorly when classes have very different distributions, otherwise it will provide a more reliable estimate
- If a Gaussian model is not adequate for our data we can use different distributions that are more appropriate
- Alternatively, the Gaussian model may still be effective for transformed data

By testing the different models on the MNIST dataset, we can notice that

- The best model is MVG, because classes have significantly different between class covariances (covariance for 0 is different than covariance for 1)
- The Naive Bayes assumption is not good, since there is a strong within-class correlation among near pixels (if one is black, we expect the near ones to be black too)
- PCA helps reducing the complexity up to a certain point, then it starts giving out bad results
- Tied model + LDA achieve the same performance as Tied model without LDA, since the LDA subspace contains all the information used by the Tied model
- The Naive Tied model, without LDA, performs slightly worse since it ignores within-class correlations
- Our implementation of LDA withens the within-class covariance matrix, so Tied Naive and Tied model are equivalent, since $\mathbf{\Sigma} = \mathbf{I}$

5.2 Modeling discrete values

5.2.1 Categorical features

We now consider a problem characterized by discrete features. For the moment, we assume we have a single categorical feature $x \in \{1 \dots m\}$.
Let's assume that our task is to predict a cat gender from the fur color: we have a set of labeled training samples $\mathcal{D} = \{(x_1, c_1), \dots, (x_n, c_n)\}$ where x_i is a fur color and c_i is the cat gender (male / female). As always, we also assume that samples are i.i.d.
We want to compute the probabilities $P(X_t = x_t | C_t = c_t) = \pi_{c,x_t}$ for different hypotheses c for the observed test sample x_t. Let $\boldsymbol{\pi}_c = (\pi_{c,1}, \dots, \pi_{c,m})$ be the model parameters for class c, with the constraint

$$\sum_{i=1}^{m} \pi_{c,i} = 1 \tag{75}$$

Again, we can adopt a frequentis approach and estimate the ML solution for $\mathbf{\Pi} = (\boldsymbol{\pi}_1, \dots, \boldsymbol{\pi}_k)$, where k is the number of classes. We can express the likelihood for a training set with n samples as

$$\mathcal{L}(\mathbf{\Pi}) = \prod_{i=1}^{n} P(X_i = x_i | C_i = c_i) P(C_i = c_i) \tag{76}$$

When computing the log-likelihood all the priors will sum up to a constant which can be discarded, since we are interested in finding a $\mathbf{\Pi}$ that *maximizes* it. It turns out that the log-likelihood can be computed from the log-likelihoods of each class

$$l(\mathbf{\Pi}) = \sum_{c=1}^{k} l_c(\boldsymbol{\pi}_c) + \xi \tag{77}$$

The log-likelihood for a single class c is

$$l_c(\boldsymbol{\pi}_c) = \sum_{i|c_i=c} \log \pi_{c,x_i} = \sum_{j=1}^{m} N_{c,j} \log \pi_{c,j} \tag{78}$$

where $N_{c,j}$ is the number of times we observed $x_i = j$ in the dataset for class c.
In this case, solving for π_{c,x_i} is a bit more complex that in the Gaussian case, since we have the constraint that $\sum_{j=1}^{m} \pi_{c,j} = 1$.
A solution can be obtained by means of **Lagrange multipliers**, from which we derive

$$\pi^*_{c,j} = \frac{N_{c,j}}{N_c} \tag{79}$$

i.e. the frequenty with which we observe value j in class c.
At this point we can say that $P(X_t = x_t | C_t = c) = \pi^*_{c,x_t}$.

If we have more than one categorical attribute, we may model their joint probability as a categorical R.V. with values given by all possible combinations of the attributes. For example if we want to predict the gender of a cat from its fur color and eye color, we treat every $(color_{fur}, color_{eye})$ as a possible outcome of the R.V.
The number of elements would quickly become intractable, but we can adopt again a Naive Bayes approximation and assume that features are independent. We can obtain ML estimates for each feature, and then combine them:

$$P(\mathbf{X}_t = \mathbf{x}_t | C_t = c) = \prod_j \pi^j_{c, x_{t,[j]}} \tag{80}$$

where j denotes the feature index.
CAPIRE MEGLIO FOTO A SLIDE 68

5.2.2 Event occurrences

We now consider an extended version of the problem, where features represent occurrences of events: typical examples are topic or language modeling.
We may, for example, represent a text in terms of the words that appear inside. Different topics will result in different sequences of words. Modeling all possible combinations of words is impractical, since the number of categories would grow exponentially: as an approximation, we may represent documents in terms of occurrences of single words. Thus, we have feature vectors $\mathbf{x} = (x_{[1]}, \dots, x_{[m]})$ where each $x_{[i]}$ represents the number of times we observed word i in the document, and m is the size of our dictionary. Be careful that $\mathbf{x}$ is the feature vector of one document, but we have n documents!
We have seen that occurrences can be modeled by multinomial distributions. Thus, for each class c, we have a set of parameters $\boldsymbol{\pi}_c = (\pi_{c,1}, \dots, \pi_{c,m})$ that represents the probability of observing a single instance of word i. The probability for feature vectore $\mathbf{x}$ is given by the multinomial density. Since we are not interested in constant factors because when computing the log-likelihood they just sum up to a constant which can be ignored, we have that

$$P(\mathbf{X} = \mathbf{x} | C = c) \propto \prod_{j=1}^{m} \pi_{c,j}^{x_{[j]}} \tag{81}$$

And the log-likelihood is

$$l(\mathbf{\Pi}) = \sum_c l_c(\boldsymbol{\pi}_c) + \xi \tag{82}$$

where

$$l_c(\boldsymbol{\pi}_c) = \sum_{i|c_i=c} \sum_{j=1}^{m} x_{i,[j]} \log \pi_{c,j} = \sum_{j=1}^{m} N_{c,j} \log \pi_{c,j} \tag{83}$$

In this case $N_{c,j}$ is the total number of occurrences of event (word) j for the samples of class c, i.e. $N_{c,j} = \sum_{i|c_i=c} x_{i,[j]}$. We can notice that the solution has exactly the same form as the one for the categorical case, thus the ML solution is again

$$\pi_{c,j} = \frac{N_{c,j}}{N_c} \tag{84}$$

where, in this case, N_c is the total number of words for class c, and $\pi_{c,j}$ is again the relative frequency of word j in class c.
Finally, we can calculate the log-likelihood ratio for a two class problem

$$llr(\mathbf{x}) = \log \frac{P(\mathbf{X} = \mathbf{x}|C = h_1)}{P(\mathbf{X} = \mathbf{x}|C = h_0)} = \sum_{j=1}^{m} x_{[j]} \log \pi_{h_1,j} - \sum_{j=1}^{m} x_{[j]} \log \pi_{h_0,j} \tag{85}$$

which is again a linear decision function that can be written as $llr(\mathbf{x}) = \mathbf{x}^T\mathbf{b}$ where $\mathbf{b} = (\log \pi_{h_1,1} - \log \pi_{h_0,1}, \ldots, \log \pi_{h_1,m} - \log \pi_{h_0,m})$. Pay attention that in this case $\mathbf{x}$ represents a single test sample. Notice that the vector $\mathbf{b}$ already tells us what are the most discriminant feature values (words).

5.2.3 Considerations about discrete value classifiers

Practical consideration:

- We can model a dataset of categorical samples as n independent categorical random variables $X_i \in \{1, \ldots, m\}$ where each variable represents a token and the distribution is described by a vector of probabilities $\boldsymbol{\pi}$ that allow computing $P(X = j) = \pi_j$
- Alternatively, we can model the dataset in terms of occurrenes of events. We have a random vector $\mathbf{Y} = (Y_1, \ldots, Y_m)$ whose components are random variables Y_i corresponding to the number of occurrences of event i in the dataset. The distribution is again described by a vector of probabilities $\boldsymbol{\pi}$ that represent probabilities of single events
- The two models are related by $Y_j = \sum_{i=1}^{n} \mathbb{I}[X_i = j]$. Actually the two models are equivalent and they have the same ML estimates, Bayesian posterior probabilities for the model parameters, and inference.
- Rare words can cause problems: if a word does not appear in a class we will estimate a probability $\pi_{c,j} = 0$. Any test sample that contains this word will have 0 probability of being from class c
- We can mitigate this issue introducing *pseudo-counts*, i.e. assuming that each topic contains a sample were all words appear a fixed number of times
- In practice, we can add a fixed value to the class occurrences N_c before computing the ML solution
- The model doesn't consider the order of the words: to do it we can consider pairs, triplets, etc. of words to partially account for correlations.
- It is possible to combine different models (categorical, multinomial, Gaussian, ...) through a naive Bayes assumption.

6 Logistic Regression

Logistic regression is a discriminative approach for classification. Rather than modeling the distribution of observed samples $\mathbf{X}|C$ we directly model the class posterior distribution $C|\mathbf{X}$.

6.1 Binary logistic regression

For a 2-class problem we have already seen that the Gaussian model with tied covariances provides log-likelihood ratios that are linear functions of our data

$$llr(\mathbf{x}) = \log \frac{f_{\mathbf{X}|C}(\mathbf{x}|h_1)}{f_{\mathbf{X}|C}(\mathbf{x}|h_0)} = \mathbf{w}^T\mathbf{x} + c \tag{86}$$

and the class log-posterior probability ratio is

$$\log \frac{P(C = h_1|\mathbf{x})}{P(C = h_0|\mathbf{x})} = \log \frac{f_{\mathbf{X}|C}(\mathbf{x}|h_1)}{f_{\mathbf{X}|C}(\mathbf{x}|h_0)} + \log \frac{\pi}{1-\pi} = \mathbf{w}^T\mathbf{x} + b \tag{87}$$

where the prior information has been absorbed in the *bias* term b.
The points where $s(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b = 0$ represent the separation line between our data. In general, $s(\mathbf{x})$ is the *score* function which is positive for samples of class h_1 and negative for samples of class h_0.

Given $\mathbf{w}$ and b, we can prove that the expression for the posterior class probability is

$$P(C = h_1|\mathbf{x}, \mathbf{w}, b) = \sigma(\mathbf{w}^T\mathbf{x} + b) = \sigma(s(\mathbf{x})) \tag{88}$$

where

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{89}$$

is the **sigmoid function**.
Although this may appear a useless step, since we could just do the inference by checking if $s(\mathbf{x})$ is positive or negative, it is actually necessary to find a way to compute the optimal $(\mathbf{w}, b)$, which we don't know a priori.

Assuming we have a labeled training dataset $\mathcal{D} = [(\mathbf{x}_1, c_1), \dots, (\mathbf{x}_n, c_n)]$ where classes are independently distributed, the class-posterior model allows expressing the likelihood for the observed labels as

$$\mathcal{L}(\mathbf{w}, b) = P(C_1 = c_1, \dots, C_n = c_n|\mathbf{x}_1, \dots, \mathbf{x}_n, \mathbf{w}, b) = \prod_{i=1}^{n} P(C_i = c_i|\mathbf{x}_i, \mathbf{w}, b) \tag{90}$$

We can apply a ML approach to estimate the model parameters that best describe the observed labels $(c_1, \dots, c_n)$. We want to find the value of $\mathbf{w}$ that maximizes the likelihood of our training labels.

We assume that the classes h_1, h_0 have labels 1 and 0 respectively. Also, let $y_i = P(C_i = 1|\mathbf{x}_i, \mathbf{w}, b) = \sigma(\mathbf{w}^T\mathbf{x}_i + b)$. Thanks to a property of the sigmoid which says that $1 - \sigma(x) = \sigma(-x)$ it follows that $P(C_i = 0|\mathbf{x}_i, \mathbf{w}, b) = \sigma(-\mathbf{w}^T\mathbf{x} - b)$. The distribution for $C_i|\mathbf{x}_i, \mathbf{w}, b$ is a Bernoulli distribution

$$C_i|\mathbf{x}_i, \mathbf{w}, b \sim \text{Ber}(\sigma(\mathbf{w}^T\mathbf{x}_i + b)) = \text{Ber}(y_i) \tag{91}$$

Since classes are independently distributed, the likelihood for the labeled set is

$$\mathcal{L}(\mathbf{w}, b) = \prod_{i=1}^{n} y_i^{c_i}(1 - y_i)^{(1-c_i)} \tag{92}$$

As always working with the log-likelihood is more practical, so we have

$$l(\mathbf{w}, b) = \sum_{i=1}^{n}[c_i \log y_i + (1 - c_i)\log(1 - y_i)] \tag{93}$$

Our goal is the maximization of l with respect to $\mathbf{w}$ and b.

Now, we have three ways of thinking about this:

1. Maximizing the ML means looking for $\mathbf{w}^*, b^*$ that represent the training set in the best way possible, i.e. the obtained separation line is the one that better separets the two classes in the training set.

2. The ML solution is also the solution that minimizes the average **cross-entropy** between the distribution of observed and predicted labels. The cross-entropy is $\mathbf{J}(\mathbf{w}, b) = -l(\mathbf{w}, b)$.

3. $\mathbf{J}(\mathbf{w}, b)$ can also be interpreted as a *loss function*, thus our goal is the minimization of this loss.

The cross-entropy between the distribution of observed and predicted labels for the i-th sample is represented by the expression

$$H(c_i, y_i) = -[c_i \log y_i + (1 - c_i)\log(1 - y_i)] \tag{94}$$

So $\mathbf{J}(\mathbf{w}, b)$ is the sum over all samples of the above expression. More in general, if P and Q are two distributions over the same domain, the cross-entropy between the two distributions is defined as

$$H(P, Q) = -\mathbb{E}_{P(x)}[\log Q(x)] \tag{95}$$

In our case, P is the empirical distribution of class labels, from the point of view of an observer $\mathcal{E}$ who knows the actual label. We have that

$$\begin{cases} P(C_i = 1|\mathbf{X}_i = \mathbf{x}_i, \mathcal{E}) = c_i \\ P(C_i = 0|\mathbf{X}_i = \mathbf{x}_i, \mathcal{E}) = 1 - c_i \end{cases} \tag{96}$$

i.e. $P \sim \text{Ber}(c_i)$.
Distribution Q, instead, is the distribution for the *predicted* labels according to our recognizer $\mathcal{R}$

$$\begin{cases} Q(1) = P(C_i = 1 | \mathbf{X}_i = \mathbf{x}_i, \mathcal{R}(\mathbf{w}, b)) = y_i = \sigma(\mathbf{w}^T \mathbf{x} + b) \\ Q(0) = P(C_i = 0 | \mathbf{X}_i = \mathbf{x}_i, \mathcal{R}(\mathbf{w}, b)) = 1 - y_i = 1 - \sigma(\mathbf{w}^T \mathbf{x} + b) \end{cases} \tag{97}$$

i.e. $Q \sim \text{Ber}(y_i)$.

Logistic regression looks for the minimizer of the average cross-entropy between the distributions for the training set labels of an evaluator $\mathcal{E}$ who knows the real label and the distributions for the training set labels as predicted by the model $\mathcal{R}$ itself.
The cross-entropy, as a function of Q, is minimized when $Q = P$. In our case it measures how different is the predicted distribution $\text{Ber}(y_i)$ from the empirical label distribution $\text{Ber}(c_i)$. Minimization of the average cross-entropy means we are looking for a label distribution that is as similar as possible to the empirical one.

The cross-entropy can be rewritten in terms of $z_i = 2c_i - 1$, i.e. z_i still represent class labels, but we would have $z_i = 1$ for $c_i = 1$ and $z_i = -1$ for $c_i = 0$. Let $s_i = \mathbf{w}^T \mathbf{x} + b$ be the score given by the model. We can rewrite H as

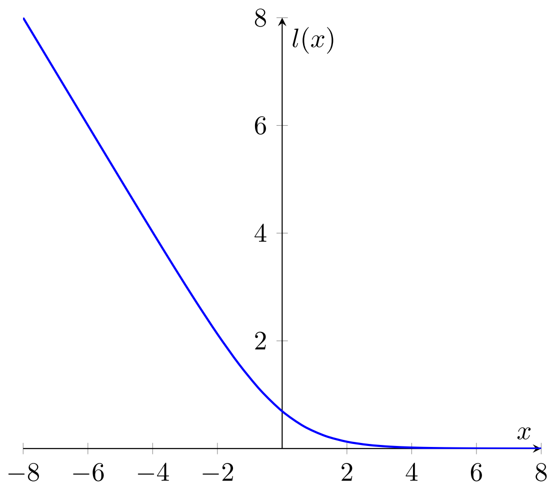
$$H(c_i, y_i) = -\log \sigma(z_i s_i) = \log(1 + e^{-z_i(\mathbf{w}^T \mathbf{x}_i + b)}) \tag{98}$$

The objective function can thus be rewritten as

$$\mathbf{J}(\mathbf{w}, b) = \sum_{i=1}^{n} l(z_i(\mathbf{w}^T \mathbf{x}_i + b)) \tag{99}$$

where $l(x) = \log(1 + e^{-x})$ is the *logistic loss* function.
Watching the following plot of $l(x)$ it is possible to understand the meaning of all of this, and why l is a loss/cost function:

- All the training samples which are on the correct side w.r.t. the separation line will have $z_i s_i > 0$ which means that $l(z_i s_i) \simeq 0$, so they will have a very low cost
- All the training samples which are on the wrong side will have $z_i s_i < 0$ and so $l(z_i s_i) > 0$. Also, l will grow linearly with s_i and we would have a certain cost (with a significant value) for the wrong prediction
- If a point is correctly classified but it is very near to the separation line, although $l(z_i s_i)$ will be small it will still be greater than 0, but not neglectable, which means that we also pay a small cost for correct prediction that are "almost on the other side". This is because we want the separation line to separe the data in the best possible way.



In this way we have reinterpreted the logistic regression objective as a measure of an *empirical risk* (empirical because computed on the observed samples). Our goal is minimizing this risk, and we do it by minimizing the cost (loss) function.
Note: logistic regression solutions can't be computed in closed form, which is why we will resort to numerical solvers, i.e. an algorithm that iteratively looks for the minimizer of a function. The algorithm requires a function that computes the loss and its gradient with respect to $\mathbf{w}$ and b.

If classes are linearly separable, the logistic regression solution is not defined. *Linearly separable classes* means that there exist $(\mathbf{w}, b)$ such that all training samples lie on the correct side of the corresponding separation surface. In this case we would have an infinite number of solutions which would give arbitrarily high s_i values, because it is just sufficient to increase the norm of $\mathbf{w}$.
As we increase $||\mathbf{w}||$, the loss becomes lower, thus we are decreasing the objecrive function. The function doesn't have a minimum but it tends to 0 as $||\mathbf{w}|| \to \infty$. To make the problem solvable again, we can look for solutions with small norm by introducing a norm penalty to the objective function

$$R(\mathbf{w}, b) = \frac{\lambda}{2}||\mathbf{w}||^2 + \frac{1}{n}\sum_{i=1}^{n} \log(1 + e^{-z_i(\mathbf{w}^T\mathbf{x}+b)}) \tag{100}$$

where λ is the *regularization coefficient*, and we are also averaging the risk over all samples. Note that λ is a hyperparameter of the model whose optimal value may be found with cross-validation. In fact, λ can't be obtained by trying to minimize R with respect to λ, as we would obtain $\lambda = 0$ which would remove the regularization term from the equation.
The regularization term can be intrpreted as a term that favors simpler solutions (those with smaller $||\mathbf{w}||$), and it also reduces the risk of over-fitting the training data.

If λ is too large, we would obtain a solution that has small norm, but is not able to well separate the classes, while if it is too small we would get a solution that has good separation on the training set but may have poor classification accuracy for unseen data (i.e. poor generalization).

Let's now make some considerations about the Logistic Regression classifier:

- The non-regularized model is invariant to linear transformations of the feature vectors while the regularized version is not invariant: this may be a problem.
- It is therefore useful, in some cases, to pre-process data so that dynamic ranges of different features are similar. For example, if we have data about people weight and height, with unit measures grams and meters, we may have some problems because height would be in a range $[0m, 2m]$ and weight in a range $[0g, 120000g]$. A good solution in this case may be to normalize quantities with respect to their standard deviation *(or using z-normalization).*
- The logistic regression score can be interpreted as the logarithm of the ratio between calss posterior probabilities, which reflects the empirical class prior of the training data. This may not be good in general, because we would like to have different priors.
- We can simulate different empirical priors π_T using a prior-weighted version of the model

  $$R(\mathbf{w}) = \frac{\lambda}{2}||\mathbf{w}||^2 + \frac{\pi_T}{n_T}\sum_{i|z_i=1} l(z_i s_i) + \frac{1-\pi_T}{n_F}\sum_{i|z_i=-1} l(z_i s_i) \tag{101}$$

  However, this would require us to know the priors while training.
- An other solution may be to "move" the separation line towards one of the two classes by adding a costant term to the score

  $$s_i \rightarrow s_i - \log\frac{\pi_{TR}}{1-\pi_{TR}} + \log\frac{\pi_{APP}}{1-\pi_{APP}} \tag{102}$$

  This solution doesn't require us to know the priors before training.

Finally, let's see the code to implement what we have explained in this subsection. We define a function which returns the function to pass to the numerical optimizer with a certain λ parameter. This step is necessary because the numerical optimizer needs a function which just takes one input parameter, while we are providing three parameters. Also, note how we use the *numpy.logaddexpr* function to compute the term $\log(1 + e^{-z_i(\mathbf{w}^T\mathbf{x}_i+b)})$ avoid numerical which may occur due to high exponent values.

```
# params are the training data, classes and lambda
def logreg_obj_wrap(D, c, l):
    def logreg_obj(v): # Function for the numerical optimizer
        w, b = v[0:-1], v[-1]
        z = 2*c-1 # From c_i={0,1} to z_i={-1,1}
        reg_term = l/2*numpy.linalg.norm(w)**2
        # compute exponent of e in the above formula
        exponent = -z*(numpy.dot(w.T, D)+b)
        # compute the sum of (97)
        sum = numpy.logaddexp(0, exponent).sum()
        return reg_term + sum/D.shape[1]
    return logreg_obj
```

To find the optimal $(\mathbf{w}, b)$ we pass a zeroed array as starting point to the optimizer. In the following code we assume we have the training dataset and labels (DTR, LTR) and the test dataset and labels (DTE, LTE). The optimizer returns a vector with the optimal input values of the objective function and the value of the function in the optimal point.

```
func = logreg_obj_wrap(DTR, LTR, l)
x0 = numpy.zeros(DTR.shape[0]+1)
v_opt, res_f_opt, _ = scipy.optimize.fmin_l_bfgs_b(func, x0,
                                            approx_grad=True)
w, b = v_opt[0:-1], v_opt[-1]
PL = (numpy.dot(w.T, DTE) + b) > 0
acc = (PL == LTE).sum()/LTE.shape[0]
```

6.2 Multiclass logistic regression

We now consider a problem with K classes, labeled from 1 to K. As before, we want to model the posterior probabilities $P(C = k|\mathbf{X} = \mathbf{x})$.
We observe that from the Tied MVG model, we have

$$\log f_{\mathbf{X}|C}(\mathbf{x}|c) = k - \frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_c)^T \boldsymbol{\Lambda}(\mathbf{x} - \boldsymbol{\mu}_c) + \frac{1}{2}\log|\boldsymbol{\Lambda}| \tag{103}$$

The above expression can be rewritten as

$$\log f_{\mathbf{X}|C}(\mathbf{x}|c) = k(\mathbf{x}, \boldsymbol{\Lambda}) + \mathbf{x}^T \mathbf{w}_c + b_c \tag{104}$$

where $k(\mathbf{x}, \boldsymbol{\Lambda}) = k + \frac{1}{2}\log|\boldsymbol{\Lambda}| - \frac{1}{2}\mathbf{x}^t \boldsymbol{\Lambda}\mathbf{x}$ doesn't depend on the class c, while $\mathbf{w}_c = \boldsymbol{\Lambda}\boldsymbol{\mu}_c$ and $b_c = -\frac{1}{2}\boldsymbol{\mu}_c \boldsymbol{\Lambda}\boldsymbol{\mu}_c$ depend on the class c. Note that since $\boldsymbol{\Lambda}$ is a function of $\mathbf{x}$, k is just a function of $\mathbf{x}$.
In the multiclass version of the LR model, we assume that we don't know the values of $\mathbf{w}_c$ and b_c. At this point, by applying Bayes rule, we have that

$$\log P_{C|\mathbf{X}}(c|\mathbf{x}) = \log \frac{f_{\mathbf{X}|C}(\mathbf{x}|c)P_C(c)}{f_{\mathbf{X}}(\mathbf{x})} = k(\mathbf{x}, \boldsymbol{\Lambda}) + \mathbf{x}^T \mathbf{w}_c + b_c + \pi_c - \log f_{\mathbf{X}}(\mathbf{x}) \tag{105}$$

By putting together the terms that do / don't depend on c, we get

$$\log P_{C|\mathbf{X}}(c|\mathbf{x}) = k'(\mathbf{x}) + \mathbf{w}_c^T \mathbf{x} + b'_c \tag{106}$$

where the term k' is irrelevant since it will just become a constant factor which is in common for all classes. We have

$$P(C = c|\mathbf{X} = \mathbf{x}) = \alpha(\mathbf{x}) e^{\mathbf{w}_c^t \mathbf{x} + b_c} \propto e^{\mathbf{w}_c^t \mathbf{x} + b_c} \tag{107}$$

From this it turns out that the Tied MVG model provides pairwise linear separation rules between classes, where the decision rules are in the form

$$(\mathbf{w}_i - \mathbf{w}_j)^T \mathbf{x} + (b_i - b_j) \lesseqgtr 0 \tag{108}$$

The column vectors $\mathbf{w}_i$ form the matrix $\mathbf{W} = [\mathbf{w}_1, \dots, \mathbf{w}_k]$, while the terms b_i form the vector $\mathbf{b} = [b_1, \dots, b_k]$.
Now, to find the value of $\alpha(\mathbf{x})$, we must force

$$\sum_{k=1}^{K} P(C = k|\mathbf{X} = \mathbf{x}) = 1 \tag{109}$$

from which

$$\sum_{k=1}^{K} \alpha(\mathbf{x}) e^{\mathbf{w}_k^T \mathbf{x} + b_k} = 1 \implies \alpha(\mathbf{x}) = \frac{1}{\sum_{k=1}^{K} e^{\mathbf{w}_k^T \mathbf{x} + b_k}} \tag{110}$$

If we define the **softmax** function to be

$$f_i(s) = \frac{e^{s_i}}{\sum_j e^{s_j}} \tag{111}$$

and we define a score vector $\mathbf{S}(\mathbf{x})$ such that $s_i(\mathbf{x}) = \mathbf{w}_i^T \mathbf{x} + b_i$, we just have that

$$P(C = c|\mathbf{X} = \mathbf{x}) = f_c(\mathbf{S}(\mathbf{x})) \tag{112}$$

At this point, if we define

$$y_{ik} = \frac{e^{\mathbf{w}_k^T \mathbf{x}_i + b_k}}{\sum_j e^{\mathbf{w}_j^T \mathbf{x}_i + b_j}} \tag{113}$$

as the softmax function for class k and sample $\mathbf{x}_i$, we can say that the posterior distribution of $C_i|\mathbf{X}_i = \mathbf{x}_i, \mathbf{W}, \mathbf{b} \sim \text{Cat}(\mathbf{y}_i)$, i.e. it is a categorical distribution. Since our goal is to find optimal values for $\mathbf{W}$ and $\mathbf{b}$, we define the log-likelihood for the training class labels as for the binary case:

$$l(\mathbf{W}, \mathbf{b}) = \sum_{i=1}^{n} \log P(C_i = c_i|\mathbf{X}_i = \mathbf{x}_i, \mathbf{W}, \mathbf{b}) \tag{114}$$

If we exploit the 1-of-K encoding for categorical distributions, we get

$$\log P(C_i = c_i|\mathbf{X}_i = \mathbf{x}_i, \mathbf{W}, \mathbf{b}) = \log P(C_i = c_i|\mathbf{y}_i) = \sum_{k=1}^{K} z_{ik} \log y_{ik} \tag{115}$$

where $\mathbf{z}_i$ is a vector with all elements equal to 0, except for the index c_i which is equal to 1.
The log-likelihood can thus be expressed as

$$l(\mathbf{W}, \mathbf{b}) = \sum_{i=1}^{n} \sum_{k=1}^{K} z_{ik} \log y_{ik} \tag{116}$$

As for the binary case, the expression

$$H(\mathbf{z}_i, \mathbf{y}_i) = -\sum_{k=1}^{K} z_{ik} \log y_{ik} \tag{117}$$

represents the multiclass cross-entropy between the observed and predicted label distributions for sample $\mathbf{x}_i$, and the ML solution is again the one that minimizes the average cross-entropy of the training dataset

$$\arg\max_{\mathbf{W},\mathbf{b}} l(\mathbf{W}, \mathbf{b}) = \arg\min_{\mathbf{W},\mathbf{b}} \sum_{i=1}^{n} H(\mathbf{z}_i, \mathbf{y}_i) \tag{118}$$

Compared to the binary case, the model is over-parametrized, which means that if we add a constant vector to all terms $\mathbf{w}_i$ the model wouldn't change. In particular, if for a 2-class problem, if we subtract $\mathbf{w}_2$ we end up in the binary logistic regression objective. [SCRIVERE LA SOTTRAZIONE]

Finally, the problem can again be casted as a minimization of a loss function $\mathbf{J}(\mathbf{W}, \mathbf{b})$ by rewriting the objective function in terms of c instead of $\mathbf{z}$. Again, we can add a regularization term to reduce over-fitting by looking for the minimizer of

$$R(\mathbf{W}, \mathbf{b}) = \Omega(\mathbf{W}) + \frac{1}{n}\mathbf{J}(\mathbf{W}, \mathbf{b}) \tag{119}$$

Different regularizers can be used, for example $\Omega(\mathbf{W}) = \frac{\lambda}{2}||\mathbf{W}||$.

The multiclass logistic regression performs better than the Tied Gaussian model on the MNIST dataset. Indeed, the Gaussian assumption is not very accurate for the features we are considering. LR only assumes linear separation but doesn't impose a distribution over the features. However, if we check the performance of the standard Gaussian model + PCA(50), we can see that it performs even better: how is it possible?
The Gaussian model allows quadratic separation rules, so even if the gaussian assumption is not very accurate, the data is better separated by quadratic surfaces rather than linear.

6.3 Quadratic surfaces LR

From the Gaussian classifier with non-tied covariances we have

$$\log \frac{P(C = h_1|\mathbf{x})}{P(C = h_0|\mathbf{x})} = \mathbf{x}^T\mathbf{A}\mathbf{x} + \mathbf{b}^T\mathbf{x} + c = s(\mathbf{x}, \mathbf{A}, \mathbf{b}, c) \tag{120}$$

i.e. the classes h_0 and h_1 are separated by a quadratic surface. If we assume that we don't know $\mathbf{A}, \mathbf{b}, c$ and we redo all the computations done up to now, we can find a way to estimate optimal values for the paramters.
However, we can try to find a *transformed space* in which linear surfaces correspond to quadratic surfaces in the $\mathbf{x}$-space.
In fact, $s(\mathbf{x}, \mathbf{A}, \mathbf{b}, c) = \mathbf{x}^T\mathbf{A}\mathbf{x} + \mathbf{b}^T\mathbf{x} + c$ is quadratic in $\mathbf{x}$ but *linear* in $\mathbf{A}, \mathbf{b}$. Basically, we can rewrite $\mathbf{x}^T\mathbf{A}\mathbf{x}$ as $\langle \mathbf{x}\mathbf{x}^T, \mathbf{A}\rangle = \text{vec}(\mathbf{x}\mathbf{x}^T)^T \text{vec}(\mathbf{A})$, where vec($\mathbf{M}$) is an operator that stacks the columns of matrix $\mathbf{M}$.
If we define

$$\boldsymbol{\phi}(\mathbf{x}) = \begin{bmatrix} \text{vec}(\mathbf{x}\mathbf{x}^T) \\ \mathbf{x} \end{bmatrix}, \ \mathbf{w} = \begin{bmatrix} \text{vec}(\mathbf{A}) \\ \mathbf{b} \end{bmatrix} \tag{121}$$

then the class log-posterior ratio can be expressed as

$$s(\mathbf{x}, \mathbf{w}, c) = \mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}) + c \tag{122}$$

We can thus train a LR model using feature vectors $\boldsymbol{\phi}(\mathbf{x})$ rather than $\mathbf{x}$. Of course the inference should also be done with $\boldsymbol{\phi}$.
This space is also called *expanded feature space*. We have to pay attention that the dimensionality of the expanded feature space can grow very quiclky, for example for polynomial expansions of degree d the expanded feature space dimensionality is $O(M^d)$ where M is the dimensionality of the original space.

Testing this model on the MNIST dataset, we obtain a better result with respect to that of the standard Gaussian classifier, since as we already said the gaussian distribution hypotesis is not very accurate in this case.

7 Measuring predictions

7.1 Accuracy and confusion matrix

In this section we will see how to assess how good a model is on a held-out evaluation set. As always, we start considering binary classification problems.
A possible way to classify models is to compute the **accuracy** or, equivalenty, the **error rate**. The accuracy is simply defined as the number of correct classified samples over the total number of samples, while the error rate is the number of incorrectly calssified samples over the total number of samples. It turns out that error rate and accuracy sum up to 1.
However, accuracy can be misleading if the classes are not balanced. For example, if our task is to predict wether on a certain day we will have rain or sun, what would happen if our classifier has an accuracy of 86%?
If we are in Italy, it would probabily be a good predictor, but if we are in the desert, where we get one rainy day every 3 years of sunny days, the model would perform poorly. In fact, a model that always predicts "sunny" would have an accuracy over 99% in this case, but it doesn't mean it is a good model, because it is unable to predict rainy days.
To overcome this issue, we introduce the **confusion matrix**, which is a table with all possible outcomes of the model:

	class $\mathcal{H}_F$	class $\mathcal{H}_T$
prediction $\mathcal{H}_F$	True Negative (TN)	False Negative (FN)
prediction $\mathcal{H}_T$	False Positive (FP)	True Positive (TP)

We can now compute different accuracy measures:

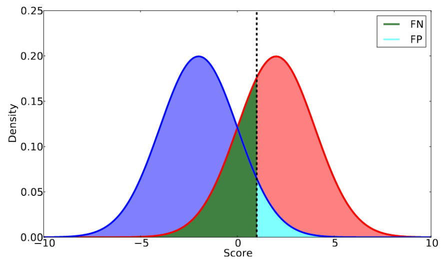
- False negative rate $FNR = \frac{FN}{FN+TP}$
- False positive rate $FPR = \frac{FP}{FP+TN}$
- True positive rate $TPR = \frac{TP}{FN+TP} = 1 - FNR$
- True negative rate $TNR = \frac{TN}{FP+TN} = 1 - FPR$

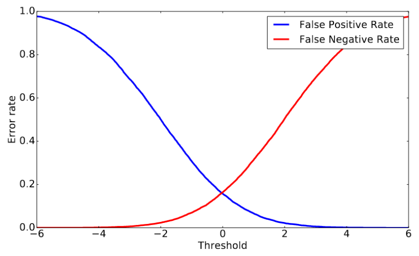
We can also compute a weighted accuracy $acc = \alpha FPR + (1-\alpha)FNR$, where the parameter α measures how important are different kind of errors.

Different kind of errors ma have a different impact on different applications. Systems providing only *hard* decisions don't allow for trade-offs between different error types. But our classifiers usually output *scores*. The score s is compared to a trehsold t, in such a way that if $s \geq t$ we favor class $\mathcal{H}_T$, otherwise we favor class $\mathcal{H}_F$.
Different tresholds correspond to different error rates, so the treshold is related not only to the class priors but also to the error costs:

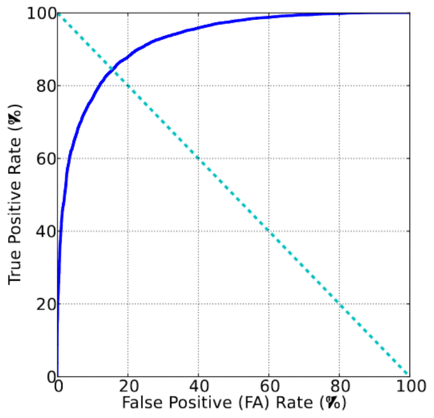
- if we increase t, less data will be classified as $\mathcal{H}_T$, so some samples of that class will be classified as $\mathcal{H}_F$, which means that we are decreasing FP and increasing FN classifications. As a consequence of this, FPR decreases and FNR increases.

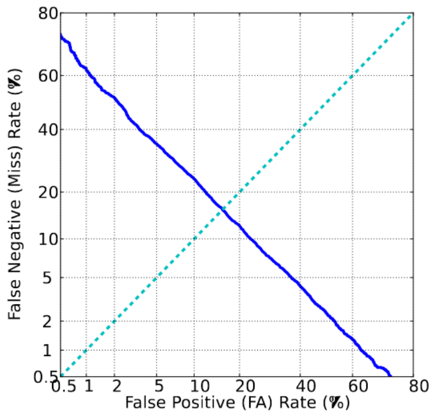
- if we decrease t we get the opposite situation, so FP increases and FN decreases (same for FPR and FNR).
- this implies that there exists a certain value for t where FPR=FNR, i.e. we have an **Equal Error Rate** (EER).





There are two graphs which allow us to visualize what happens at these accuracies when we change the value of t.
The **Receiver Operating Characteristic** (ROC) curve shows TPR vs FPR as t varies.
The **Detection Error Trade-off** (DET) curve shows FNR vs FPR as t varies.





The point where the dashed line meets the continuous line is the EER.

7.2 Bayes Decision

The goal of a classifier is to allow us to choose an action a to perform among a set of actions $\mathcal{A}$. We can associate to each action a *cost* $\mathcal{C}(a|k)$ that we have to pay when we choose action a and the sample belongs to class k. This can be seen as a missclassification cost, which depends both on the actual and predicted class.

For example, if we think of a classifier that decides if a patient is ill or healthy, and applies a treatment (action) only to ill patients, depending on what is the impact of the treatment we will have different consequences to pay for missclassifications. For example, if the treatment has a very small impact on healthy patients, it would be more acceptable to classify healthy patients as ill rather than ill patients as healthy. This mean that $\mathcal{C}(treatment|healthy)$ should be smaller than $\mathcal{C}(no\ treatment|ill)$.
In our case, actions will correspond to classes, so $\mathcal{C}(a|k)$ is the cost of labeling a sample as belonging to class a when it belongs to k.
We don't know k, but we have a classifier $\mathcal{R}$ that allows us computing posterior probabilities $P(C = k|x, \mathcal{R})$ for each sample x. We can thus compute the expected cost of action a as

$$\mathcal{C}_{x,\mathcal{R}}(a) = \mathbb{E}[\mathcal{C}(a|k)|x, \mathcal{R}] = \sum_{k=1}^{K} \mathcal{C}(a|k)P(C = k|x, \mathcal{R}) \tag{123}$$

The **Bayes decision** consists in choosing the action $a^*(x, \mathcal{R})$ that minimizes the expected cost

$$a^*(x, \mathcal{R}) = \arg\min_a \mathcal{C}_{x,\mathcal{R}}(a) \tag{124}$$

which is the action that will result in the lower expected cost, according to the recognizer $\mathcal{R}$ beliefs.
Let's now see an example of a 3-class problem where we are given a cost matrix $\mathbf{C}$, and a set of priors $\boldsymbol{\pi}$ that gives us some posterior probabilities $\mathbf{q}_t$:

$$\mathbf{C} = \begin{bmatrix} 0 & 1 & 2 \\ 1 & 0 & 1 \\ 2 & 1 & 0 \end{bmatrix}, \ \boldsymbol{\pi} = \begin{bmatrix} 0.3 \\ 0.4 \\ 0.3 \end{bmatrix} \Rightarrow \mathbf{q}_t = \begin{bmatrix} P(C = 1|\mathbf{x}_t, \mathcal{R}) \\ P(C = 2|\mathbf{x}_t, \mathcal{R}) \\ P(C = 3|\mathbf{x}_t, \mathcal{R}) \end{bmatrix} = \begin{bmatrix} 0.40 \\ 0.25 \\ 0.35 \end{bmatrix} \tag{125}$$

Using $\mathbf{q}_t$ and $\mathbf{C}$ we can compute

$$\mathbf{C}\mathbf{q}_t = \begin{bmatrix} \mathcal{C}_{\mathbf{x}_t,\mathcal{R}}(1) = 0 \times 0.40 + 1 \times 0.25 + 2 \times 0.35 = 0.95 \\ \mathcal{C}_{\mathbf{x}_t,\mathcal{R}}(2) = 1 \times 0.40 + 0 \times 0.25 + 1 \times 0.35 = 0.75 \\ \mathcal{C}_{\mathbf{x}_t,\mathcal{R}}(3) = 2 \times 0.40 + 1 \times 0.25 + 0 \times 0.35 = 1.05 \end{bmatrix} \tag{126}$$

From which it turns out that the best option for $\mathbf{x}_t$ is to classify it as $C = 2$. In fact, even if we would have a higher error rate, the cost of each mistake is way smaller.

Now let's go back to a binary problem. We have four costs:

	class $\mathcal{H}_F$	class $\mathcal{H}_T$
prediction $\mathcal{H}_F$	$\mathcal{C}(\mathcal{H}_F\|\mathcal{H}_F)$	$\mathcal{C}(\mathcal{H}_F\|\mathcal{H}_T)$
prediction $\mathcal{H}_T$	$\mathcal{C}(\mathcal{H}_T\|\mathcal{H}_F)$	$\mathcal{C}(\mathcal{H}_T\|\mathcal{H}_T)$

Without loss of generality we assume $\mathcal{C}(\mathcal{H}_T|\mathcal{H}_T) = 0$ and $\mathcal{C}(\mathcal{H}_F|\mathcal{H}_F) = 0$, i.e. correct prediction don't cost anything.
Also, $\mathcal{C}(\mathcal{H}_F|\mathcal{H}_T) \geq 0$, $\mathcal{C}(\mathcal{H}_T|\mathcal{H}_F) \geq 0$, i.e. we pay a positive cost for wrong

decisions.
From now on we will indicate $\mathcal{C}(\mathcal{H}_F|\mathcal{H}_T)$ as C_{fn} and $\mathcal{C}(\mathcal{H}_T|\mathcal{H}_F)$ as C_{fp}, i.e. they are the costs for false negative errors and false positive errors, respectively.
If we calculate the expected Bayes cost for actions $\mathcal{H}_T$ and $\mathcal{H}_F$ we get

$$\begin{aligned}\mathcal{C}_{x,\mathcal{R}}(\mathcal{H}_T) = C_{fp}P(\mathcal{H}_F|x,\mathcal{R}) + 0 \cdot P(\mathcal{H}_T|x,\mathcal{R}) = C_{fp}P(\mathcal{H}_F|x,\mathcal{R}) \\ \mathcal{C}_{x,\mathcal{R}}(\mathcal{H}_F) = 0 \cdot P(\mathcal{H}_F|x,\mathcal{R}) + C_{fn}P(\mathcal{H}_T|x,\mathcal{R}) = C_{fn}P(\mathcal{H}_T|x,\mathcal{R})\end{aligned} \tag{127}$$

and the optimal decision is the labeling that has lowest cost. Alternatively, we can compute the log-ratio between the above quantities and classify basing on the log-ratio value

$$r(x) = \log \frac{C_{fn}P(\mathcal{H}_T|x,\mathcal{R})}{C_{fp}P(\mathcal{H}_F|x,\mathcal{R})} \lesseqgtr 0 \tag{128}$$

We assign class $\mathcal{H}_T$ if $r(x) > 0$ and $\mathcal{H}_F$ otherwise.
If $\mathcal{R}$ is a generative model for x, then we can express r in terms of costs, prior probabilities and likelihoods as

$$r(x) = \log \frac{C_{fn}}{C_{fp}} \cdot \frac{\pi_T}{1-\pi_T} \cdot \frac{f_{X|\mathcal{H},R}(x,\mathcal{H}_T)}{f_{X|\mathcal{H},R}(x|\mathcal{H}_F)} \tag{129}$$

In this way, the decision rule becomes

$$\log \frac{f_{X|\mathcal{H},R}(x,\mathcal{H}_T)}{f_{X|\mathcal{H},R}(x|\mathcal{H}_F)} \lesseqgtr -\log \frac{C_{fn}\pi_T}{C_{fp}(1-\pi_T)} \tag{130}$$

The triplet (π_T, C_{fn}, C_{fp}) represents the **working point** of an application for a binary classification task. However, the working point is overparametrized, in fact we can show that it is possible to build infinite equivalent applications which have the same decision rule as the original one, but different costs and priors.
We focus our attention on one particular application $(\tilde{\pi}, 1, 1)$, with

$$\tilde{\pi} = \frac{\pi_T C_{fn}}{\pi_T C_{fn} + (1-\pi_T)C_{fp}} \tag{131}$$

We can interpret $\tilde{\pi}$ as an **effective prior**: if the class prior for $\mathcal{H}_T$ was $\tilde{\pi}$ and we assumed uniform costs, we would obtain the same decisions as for our original application.
Similarly, we can obtain an equivalent application where the effective prior is uniform $\tilde{\pi} = \frac{1}{2}$ and the application prior π_T is absorbed in **effective classification costs**.

We have, up to now, considered how to perform decisions for a sample x. We now return to the problem of evaluating the goodness of our decisions.
We assume that recognizer $\mathcal{R}$ takes decisions $a(x,\mathcal{R})$ for sample x with correct class label c. The cost of each decision is thus $\mathcal{C}(a(x,\mathcal{R})|c)$, where we think of $a(x,\mathcal{R})$ as the predicted class, while the correct class is c.

We can compute the expected cost (**Bayes risk**) of decisions made by our classifier for the evaluation population

$$\mathcal{B} = \mathbb{E}_{X,C|\mathcal{E}}[\mathcal{C}(a(x,\mathcal{R})|c)] \tag{132}$$

where $\mathcal{E}$ denotes the evaluation population, assumed to be distributed according to $X, C|\mathcal{E}$. $\mathcal{E}$ is basically an evaluator who knows the complete distribution of data.
The Bayes risk $\mathcal{B}$ for decisions made by $\mathcal{R}$ over evaluation data sampled from $X, C|\mathcal{E}$ is defined as

$$\mathcal{B} = \sum_{c=1}^{K} \pi_c \int \mathcal{C}(a(x,\mathcal{R})|c) f_{X|C,\mathcal{E}}(x|c)dx \tag{133}$$

Note that the distribution $X|C,\mathcal{E}$ reflects the knowledge of the evaluator $\mathcal{E}$, not the knowledge of the recognizer $\mathcal{R}$. The evaluator is measuring how good are the decisions made by the recognizer for his own task (data sampled from $X|C,\mathcal{E}$).
Unfortunally, we don't have access to $f_{X|C,\mathcal{E}}(x|c)$, but if we have at our disposal a set of labeled evaluation samples, we can approximate the expectations by averaging the cost over the samples. In fact, if the number of samples per class becomes large, it's possible to show that

$$\int \mathcal{C}(a(x,\mathcal{R})|c) f_{X|C,\mathcal{E}}(x|c)dx \simeq \frac{1}{N_c} \sum_{i|c_i=c} \mathcal{C}(a(x_i,\mathcal{R})|c) \tag{134}$$

So, we can finally define the **empirical Bayes risk** as

$$\mathcal{B}_{emp} = \sum_{c=1}^{K} \frac{\pi_c}{N_c} \sum_{i|c_i=c} \mathcal{C}(a(x_i,\mathcal{R})|c) \tag{135}$$

The risk measures the costs of our decisions over the evaluation samples.
We can use $\mathcal{B}_{emp}$ to compare recognizers. A recognizer that has lower cost will provide more accurate answers.
For a binary problem, the above formula can also be expressed as

$$\mathcal{B}_{emp} = \pi_T C_{fn} P_{fn} + (1-\pi_T) C_{fp} P_{fp} \tag{136}$$

In fact, for the binary case, we can derive this formula with the following procedure

- Assume we have $\tilde{N}$ samples, divided in N_T and N_F, i.e. the number of samples of class T and F respectively
- $N_T = FN + TP$
- The number of samples from class T that i missclassify is FN
- If the prior probability of class T is π_T, I expect $\pi_T \tilde{N}$ samples from class T

- The number of missclassifications will thus be $\frac{FN}{N_F}\pi_T\tilde{N}$
- The cost of misslcassifications for class T will be $C_{fn}FNR\pi_T\tilde{N}$
- With similar considerations we find the cost of missclassifications for class F and we get the final formula.

$\mathcal{B}_{emp}$ is also called (un-normalized) **Detection Cost Function** (DCF). We say "un-normalized" because this value doesn't tell a lot by itself. This is why we introduce the following concepts.
$C_{fn},\ C_{fp},\ \pi_T$ are parameters that depend only on the application.
If we consider a *dummy* system that always *accepts* a test segment (i.e. classifies it as T), we would have $P_{fp} = 1,\ P_{fn} = 0$ that imply $DCF_u = (1 - \pi_T)C_{fp}$.
Similarly, a dummy system that always *rejects* a test segment would have $P_{fp} = 0,\ P_{fn} = 1$ that imply $DCF_u = \pi_T C_{fn}$.
We compare the system DCF w.r.t. the best dummy system to get the normalized CDF

$$DCF(\pi_T, C_{fn}, C_{fp}) = \frac{DCF_u(\pi_T, C_{fn}, C_{fp})}{\min(\pi_T C_{fn}, (1 - \pi_T)C_{fp})} \tag{137}$$

Note that the best dummy system corresponds to optmial Bayes decisions based on prior information alone.
The normalized DCF is invariant to scaling, so we can rescale the un-normalized DCF by $\frac{1}{\pi_T C_{fn} + (1-\pi_T)C_{fp}}$ and obtain

$$DCF_u(\tilde{\pi}) = \tilde{\pi}P_{fn} + (1 - \tilde{\pi})P_{fp} \tag{138}$$

In terms of normalized DCF, the applications (π_T, C_{fp}, C_{fn}) and $(\tilde{\pi}, 1, 1)$ are again equivalent.

At this point, we can observe that the error rate defined at the beginning of the section corresponds to

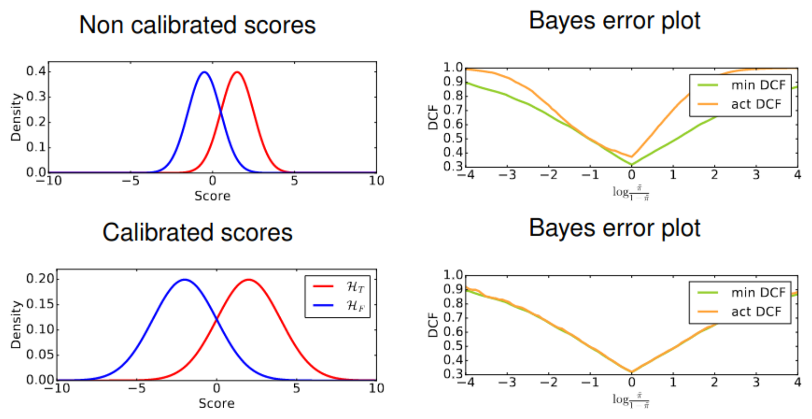
$$e = \frac{N_T P_{fn} + N_F P_{fp}}{N} = \frac{N_T}{N}P_{fn} + \frac{N_F}{N}P_{fp} \tag{139}$$

i.e., up to a scaling factor, to the DCF of an application $(\frac{N_T}{N}, 1, 1)$ that has the *empirical* prior of the test set as prior probability.

7.3 Score calibration

LLRs allow disentangling the classsifier from the application. In general, system don't produce well-calibrated LLRs, for example when we use non-probabilistic scores (SVM), or when there is a mis-match between train and test populations, or if the model assumptions are inaccurate.
In these cases we say that scores are **mis-calibrated**, i.e. the theoretical treshold $-\log\frac{\tilde{\pi}}{1-\tilde{\pi}}$ is not optimal anymore.
For a given application we can measure the additional cost due to the use of

mis-calibrated scores. We consider varying the treshold t to obtain all possible combinations of P_{fn} and P_{fp} for the evaluation set, and we select the treshold corresponding to the lowest DCF, called DCF_{min}. This corresponds to the optimal treshold for the evaluation set.
The value DCF_{min} is the cost we would pay if we knew the optimal treshold for the evaluation, and the different between actual DCF and DCF_{min} represents the loss due to score mis-calibration.
We can also compare different systems over different appplications through Bayes error plots. These plots can be used to report actual and/or minimum DCF for different applications. Since a binary application is parametrized only by $\tilde{\pi}$, we can plot the DCF as a function of prior log-odds $\log \frac{\tilde{\pi}}{1-\tilde{\pi}}$, i.e. the negative of the Bayes optimal treshold.



To reduce mis-calibration we can adopt different calibration strategies. A widely used approach is to look for a function that transforms the classifier scores s into approximately well-calibrated LLRs, in a way that is as much as possible independent from the target application, i.e. we want to compute a transformation function f such that $s_{cal} = f(s)$.
Isotonic Regression is a non-linear, monotonic transformation that provides optimal calibration for the data it's trained on, but it is very complex so it won't be covered here.
Score models are an approximation to the isotonic regression transformation. They require an assumption on the calibration transformation, for example a linear mapping between scores and calibrated scores. Next, we are going to cover the **prior-weighted logistic regression** score model.
The prior-weighted LR consists in training an LR model using scores of an other model as features. This has to be done on a dataset which is distinct from the one used to train the model: such dataset is called **calibration set**. So, we consider the non-calibrated scores as features, and assume a linear mapping from non-calibrated scores to calibrated scores

$$f(s) = \alpha s + \gamma \tag{140}$$

Since $f(s)$ should produce well-calibrated scores, $f(s)$ can be interpreted as the LLR for the two class hypotheses

$$f(s) = \log \frac{f_{S|C}(s|\mathcal{H}_T)}{f_{S|C}(s|\mathcal{H}_F)} = \alpha s + \gamma \tag{141}$$

The class posterior probability for prior $\tilde{\pi}$ corresponds to

$$\log \frac{P(C = \mathcal{H}_T|s)}{P(C = \mathcal{H}_F|s)} = \alpha s + \gamma + \log \frac{\tilde{\pi}}{1-\tilde{\pi}} = \alpha s + \beta \tag{142}$$

We can employ the prior-weighted LR model with $\pi_T = \tilde{\pi}$ to learn the model parameters α, β over our training scores. To recover the calibrated score we will need to compute

$$f(s) = \alpha s + \gamma = \alpha s + \beta - \log \frac{\tilde{\pi}}{1-\tilde{\pi}} \tag{143}$$

Note that we have to specify a prior $\tilde{\pi}$, and we are still optimizing the calibration for a specific application $\tilde{\pi}$. However, the model will often provide good calibration for a wider range of different applications.
Where to get the calibration set?

- If the mis-calibration is due to non-probabilistic scores, or to overfitting or underfitting, it can be taken from the training set material.
- If the mis-calibration is due to mismatches between training and evaluation data, it should mimic the evaluation population, so it can't be collected from the training data.

We will usually be in the first case, where we will split the training data in 3 parts

1. Effective training data.
2. Calibration data, that will be converted to scores of the trained model and then used to train the calibration parameters.
3. Validation data, that will be used to evaluate the performance of the whole model using both the trained model and the calibration parameters.

With K-fold a posible approach is the followin:

1. Apply K-fold to train the classifier
    - Train K classifier, each without fold k (model $\mathcal{R}_k$)
    - Score each fold k with model $\mathcal{R}_k$
    - Train a classifier $\mathcal{R}_{\mathcal{F}}$ over the whole training set
    - Pool the scores of each fold to obtain a score set, used as calibration data

2. Shuffle the calibration scores and apply K-fold over them
    - Train K calibration models $\mathcal{C}_k$
    - Train a calibration model over all scores $\mathcal{C}_{\mathcal{F}}$
    - Calibrate the scores of each fold k with model $\mathcal{C}_k$
    - Pool the calibrated scores and evaluate the model performance over the pooled scores to choose the best configuration

Finally, for classification, map x_T to $\mathcal{C}_{\mathcal{F}}(\mathcal{R}_{\mathcal{F}}(x_T))$.

Evaluation of *multiclass tasks* is more complex and is based on confusion matrices and empirical Bayes risk for multiclass problems. Also in this case it is possible to have mis-calibration, which is addressable with multiclass logistic regression.

7.4 Combined models

Assume we have two (or more) models where the first performs poorly when the other performs good, and vice versa. We would like to find a way make them work together in such a way that we obtain a unique model with good overall performance.

A basic solution would be to use priority voting, for example if we are using three models and they give scores $\{-5, -4, 3\}$, since we got two negative scores and one positive, we can assign the sample to the negative class. But what if we get scores $\{0.1, 0.1, -99\}$? In this case, we have that the first two models classify the data as positive, but their score is close to 0, which means they are "unsure" about their decision. The third model, instead, classifies the sample as negative with a value much lower than 0, so it is sure about his decision. In such scenario, priority voting might not be the best decision.

An other solution would be to compute a global score as a linear combination of the models' scores. We have that

$$s = \sum_{i=1}^{m} \alpha_i s_i = \boldsymbol{\alpha}^T \mathbf{s} \tag{144}$$

where m is the number of used models. In this case, we can again use prior-weighted LR to compute a mapping function

$$f(s) = \boldsymbol{\alpha}^T \mathbf{s} + \gamma \tag{145}$$

which at the same time calibrates the final score. In this case $\boldsymbol{\alpha}$ corresponds to the LR parameter $\mathbf{w}$.

8 Support Vector Machines

8.1 Motivation

In this section we will focus our attention on binary classification only.
We have seen that LR provides a linear classification rule that maximizes the log-probability of class assignments for the training set. The LR solution can be found by minimizing the *logistic loss*.
We have also seen that it is usefull to add a regularization term of the form $\frac{\lambda}{2}||\mathbf{w}||^2$ when trying to find the model parameters:

$$\mathbf{w}^*, b = \arg\min_{\mathbf{w},b}(\frac{\lambda}{2}||\mathbf{w}||^2 + \frac{1}{n}\sum_{i=1}^{n}\log(1 + e^{-z_i(\mathbf{w}^T\mathbf{x}_i+b)}) \tag{146}$$

We will now consider a different classifier that allows us to give a geometrical interpretation of the regularization term: the **Support Vector Machine**. We will show that SVM can be cast as a generalized risk minimization problem.
SVMs provide a natural way to achieve non-linear separation without the need for an explicit expansion of features.
However, the output of SVMs cannot be directly interpreted as class posteriors.

8.2 Hard Margin SVM

Let's assume we have two linearly separable classes. Usually, there is an infinite number of separating hyperplanes. LR chooses the one that maximizes class probabilities, minimizing the logistic loss. On the other hand, SVMs select the hyperplane that separates the classes with the largest **margin**. The margin is defined as the distance of the closest point w.r.t the separation hyperplane.
Let $\mathbf{f}(x) = \mathbf{w}^T\mathbf{x} + b$ be the function representing the separation surface. The distance of $\mathbf{x}_i$ from the hyperplane is

$$d(\mathbf{x}_i) = \frac{|f(\mathbf{x}_i)|}{||\mathbf{w}||} \tag{147}$$

Since the classes are linearly separable, we will consider solutions which correctly classify all points, so $f(\mathbf{x}_i) > 0$ if $c_i = \mathcal{H}_T$ and $\mathbf{x}_i < 0$ if $c_i = \mathcal{H}_F$.
If we represent classes with $z_i = \pm 1$ (as for LR) the distance of $\mathbf{x}$ from the hyperplane can be rewritten as

$$d(\mathbf{x}) = \frac{|z_i(\mathbf{w}^T\mathbf{x} + b)|}{||\mathbf{w}||} \tag{148}$$

and the maximum margin hyperplane is the one which maximizes the minimum distance of all points from the hyperplane

$$\mathbf{w}^*, b^* = \arg\max_{\mathbf{w},b}\min_{i\in\{1\ldots n\}} d(\mathbf{x}_i) = \arg\max_{\mathbf{w},b}\min_{i\in\{1\ldots n\}} \frac{|z_i(\mathbf{w}^T\mathbf{x} + b)|}{||\mathbf{w}||} \tag{149}$$

subject to the constraint $z_i(\mathbf{w}^T\mathbf{x} + b) > 0$. The formula is saying "for each possible $\mathbf{w}, b$ compute the minimum distance of all $\mathbf{x}_i$ from the hyperplane and then select the parameters that maximize this distance.
Since we assumed linearly separable classes, and the (min) distance is positive for correctly classified samples, we can drop the constraint by considering an equivalent problem

$$\mathbf{w}^*, b^* = \arg\max_{\mathbf{w},b} \frac{1}{||\mathbf{w}||} \min_{i\in\{1\dots n\}} [z_i(\mathbf{w}^T\mathbf{x} + b)] \tag{150}$$

This problem is equivalent to the previous because hyperplanes that don't satisfy the constraint (i.e. those with min distance resulting lower than 0) cannot be an optimal solution and will be discarded.
Unfortunally, direct optimization of this function is non trivial, so we need to further transform the problem. First of all, we can observe that the objective function is invariant under rescaling of the parameters:

$$\frac{1}{||\mathbf{w}||} \min_i [z_i(\mathbf{w}^T\mathbf{x} + b)] = \frac{1}{||\alpha\mathbf{w}||} \min_i [z_i(\alpha\mathbf{w}^T\mathbf{x} + \alpha b)] \tag{151}$$

so we can always find an optimal solution such that

$$\min_i z_i(\mathbf{w}^t\mathbf{x}_k + b) = 1 \tag{152}$$

This implies that for all the training points we will have

$$z_i(\mathbf{w}^t\mathbf{x}_k + b) \geq 1, \qquad i = 1, \dots, n \tag{153}$$

In this way, the problem becomes equivalent to finding the parameters that maximize $\frac{1}{||\mathbf{w}||}$, or equivalently, those that minimize the squared norm of $\mathbf{w}$ (under some constraints):

$$\begin{aligned} &\mathbf{w}^*, b^* = \arg\min_{\mathbf{w},b} \frac{1}{2}||\mathbf{w}||^2 \\ &\text{s.t. } z_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1, \qquad i = 1, \dots, n \end{aligned} \tag{154}$$

This is the **primal formulation** of the problem. It is a convex quadratic programming problem, with a convex objective function and the constraints forming a convex set.
To solve the SVM probkem we consider a *Lagrangian* formulation of the problem: for each constraint we introduce the Lagrange multiplier $\alpha_i \geq 0$

$$L(\mathbf{w}, b, \boldsymbol{\alpha}) = \frac{1}{2}||\mathbf{w}||^2 - \sum_{i=1}^{n} \alpha_i[z_i(\mathbf{w}^T\mathbf{x}_i + b) - 1] \tag{155}$$

Since the problem is convex, the optimal solution is obtained by maximizing L w.r.t. α_i while requiring that either the derivatives w.r.t. $\mathbf{w}, b$ vanish, subject to $\alpha_i \geq 0$. Setting the derivatives of L w.r.t. $\mathbf{w}, b$ equal to zero gives

$$\begin{aligned} \mathbf{w} &= \sum_{i=1}^{n} \alpha_i z_i \mathbf{x}_i \\ 0 &= \sum_{i=1}^{n} \alpha_i z_i \end{aligned} \tag{156}$$

Replacing the constraints in L gives the **dual formulation** of the SVM problem

$$\max_{\boldsymbol{\alpha}} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j z_i z_j \mathbf{x}_i^T \mathbf{x}_j \tag{157}$$

$$\text{s.t.} \begin{cases} \alpha_i \geq 0, & i = 1, \ldots, n \\ \sum_{i=1}^{n} \alpha_i z_i = 0 \end{cases}$$

The *dual objective function* can be expressed in matrix form as

$$L_D(\boldsymbol{\alpha}) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j z_i z_j \mathbf{x}_i^T \mathbf{x}_j = \boldsymbol{\alpha}^T \mathbf{1} - \frac{1}{2} \boldsymbol{\alpha}^T \boldsymbol{H} \boldsymbol{\alpha} \tag{158}$$

where the matrix $\mathbf{H}$ is defined as $H_{ij} = z_i z_j \mathbf{x}_i^T \mathbf{x}_j$. Notice how it depends only on dot products of $\mathbf{x}_i$.
A solution $\mathbf{w}, b, \boldsymbol{\alpha}$ will be optimal if and only if it satisfies the **Karush-Kuhn-Tucker** (KKT) conditions

$$\begin{cases} \nabla_{\mathbf{w}} L(\mathbf{w}, b, \boldsymbol{\alpha}) = \mathbf{w} - \sum_{i=1}^{n} \alpha_i z_i \mathbf{x}_i = \mathbf{0} \\ \dfrac{\partial L(\mathbf{w}, b, \boldsymbol{\alpha})}{\partial b} = -\sum_{i=1}^{n} \alpha_i z_i = 0 \\ z_i(\mathbf{w}^T \mathbf{x}_i + b) - 1 \geq 0, \quad \forall i \\ \alpha_i \geq 0, \quad \forall i \\ \alpha_i \dfrac{\partial L}{\partial \alpha_i} = \alpha_i [z_i(\mathbf{w}^T \mathbf{x}_i + b) - 1] = 0, \quad \forall i \end{cases} \tag{159}$$

The first two equations encode that, at the optimal solution $(\mathbf{w}, b)$, the gradient of the Lagrangian with respect to $\mathbf{w}, b$ becomes 0.
The third and fourth equation encode that both the primal solution $(\mathbf{w}, b)$ and the corresponding dual solution $\boldsymbol{\alpha}$ are feasible.
The last equation encodes that the optimal dual solution maximizes the Lagrangian. This requires that either the maximum is on the constraint, i.e. $\alpha_i = 0$, or the derivative of the Lagrangian is $\frac{\partial L}{\partial \alpha_i} = 0$.

In the optimal solution, for all points that don't lie on the margin we have that $z_i(\mathbf{w}^T \mathbf{x}_i + b) > 1$, and the corresponding Lagrange multiplier is $\alpha_i = 0$.

If $\alpha_i \neq 0$, the corresponding point is on the margin, and we call it a **Support Vector**. After we have obtained $\boldsymbol{\alpha}$, the KKT conditions allow us to estimate b. Also, we can obtain $\mathbf{w}$ from the first equation.
At this point, for a given test point $\mathbf{x}_t$, we can compute its score as

$$s(\mathbf{x}_t) = \mathbf{w}^T\mathbf{x}_t + b = \sum_{i=1}^{n} \alpha_i z_i \mathbf{x}_i^T \mathbf{x}_t + b \tag{160}$$

Notice that, again, this depends only on dot products $\mathbf{x}_i^T\mathbf{x}_t$. The training points that are not Support Vectors do not affect the separation surface.

8.3 Soft Margin SVM

Remember that up to now we were under the assumption that classes were linearly separable. But this is not true in the general case. Let's now focus our attention on non linear separable classes. In this case, it's impossible to find a value for $\mathbf{w}$ such that the constraint $z_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1$ is not violated.
We can try to minimize the number of points that violate the constraint by introducing the **slack variables** $\xi_i \geq 0$, which represent how much a point is violating the constraint. The constraint is replaced with

$$z_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0 \tag{161}$$

i.e. we allow training points to be inside the margin by a factor ξ_i. If we could remove the points which surpass the margins, the classes would be linearly separable and we could act as before. This corresponds to the minimization of a new objective function

$$\min_{\mathbf{w},b,\boldsymbol{\xi}} \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{n}\xi_i \tag{162}$$

$$\text{s.t.} \begin{cases} z_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 - \xi_i, & \forall i \\ \xi_i \geq 0, & \forall i \end{cases}$$

This is the primal formulation in terms of $\mathbf{w}, b, \boldsymbol{\xi}$, where we are paying a cost for each positive slack variable (i.e. for each point inside the margin), and the cost depends on how big is the slack variable (i.e. how far is the point from the margin). The sum of the slack variables is an upper bound on the number of miss-classified points, while C is a constant which specifies how much we should care about the errors.
This is called **soft margin** SVM, where we allow margins to vary, while before we have treated the **hard margin** SVM.
As we did for the hard margin SVM, we can introduce the Lagrangian problem where we need to take into account also a new set of multipliers $\boldsymbol{\mu}$ relative to the constraints $\xi_i \geq 0$.

$$L(\mathbf{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\mu}) = \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{n}\xi_i - \sum_{i=1}^{n}\alpha_i[z_i(\mathbf{w}^T\mathbf{x}_i + b) - 1 + \xi_i] - \sum_{i=1}^{n}\mu_i\xi_i \tag{163}$$

From which we get three additional KKT conditions

$$\begin{cases} \mu_i \geq 0 \\ \alpha_i = C - \mu_i \\ \mu_i \xi_i = 0 \end{cases} \tag{164}$$

From the first two we derive $0 \leq \alpha_i \leq C$ (previous conditions still hold).
At this point, if $\alpha_i = 0$, we have that $z_i(\mathbf{w}^T\mathbf{x}_i + b) - 1 + \xi_i > 0$ except for some corner cases. Also, we would have $\mu_i = C$, so $\xi_i = 0$. This means that $\mathbf{x}_i$ is correctly classified, as we already should know, since $\alpha_i = 0$.
If instead we have $0 < \alpha_i < C$, we must have $z_i(\mathbf{w}^T\mathbf{x}_i + b) - 1 + \xi_i = 0$. Also, if $\alpha < C$, $\mu_i > 0$, thus $\xi_i = 0$. So, in the end, $z_i(\mathbf{w}^T\mathbf{x}_i + b) = 1$, which means that $\mathbf{x}_i$ is exactly on the margin.
Finally, if $\alpha_i = C$ we have that $\mu_i = 0$, so $\xi_i > 0$ and the point $\mathbf{x}_i$ is inside the margin.
From the values of the Lagrangian multipliers we can understand what kind of point we have, and so we know if it contributes or not to defining the separation surface.

As we did before, we can substitute the constraints in the primal formulation to obtain the dual one

$$\max_{\boldsymbol{\alpha}} L_D(\boldsymbol{\alpha}) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i \alpha_j z_i z_j \mathbf{x}_i^T \mathbf{x}_j \tag{165}$$

$$\text{s.t.} \begin{cases} 0 \leq \alpha_i \leq C, \quad i = 1, \dots, n \\ \sum_{i=1}^{n} \alpha_i z_i = 0 \end{cases}$$

Predictions are again computed as in the hard-margin case as

$$s(\mathbf{x}_t) = \mathbf{w}^T\mathbf{x}_t + b = \sum_{i=1}^{n} \alpha_i z_i \mathbf{x}_i^T \mathbf{x}_t + b \tag{166}$$

Again, this depends only on dot products $\mathbf{x}_i^T\mathbf{x}_t$.
But if we are interested in linear separation we can directly solve the primal problem: we know that $\xi_i = 0$ for points on the correct side of the margin, otherwise we get

$$\xi_i = 1 - z_i(\mathbf{w}^T\mathbf{x}_i + b) \tag{167}$$

and by substituting this in the primal formulation we obtain

$$\min_{\mathbf{w},b} \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{n} \max[0, 1 - z_i(\mathbf{w}^T\mathbf{x}_i + b)] \tag{168}$$

where we don't have any additional constraint, since they have been absorbed by the *loss* term

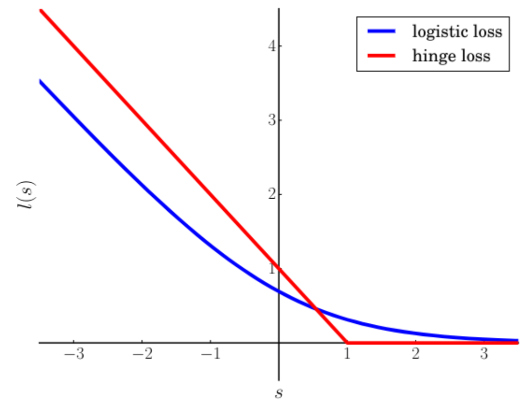
$$max[0, 1 - z_i(\mathbf{w}^T\mathbf{x}_i + b)] \tag{169}$$

In general, the function $f(s) = \max(0, 1-s)$ is called **hinge loss**. The objective can then be rewritten as

$$\min_{\mathbf{w},b} \frac{\lambda}{2}||\mathbf{w}||^2 + \frac{1}{n}\sum_{i=1}^{n} \max[0, 1 - z_i(\mathbf{w}^T\mathbf{x}_i + b)] \tag{170}$$

which is very similar to the Logistic Regression one:

$$\min_{\mathbf{w},b} \frac{\lambda}{2}||\mathbf{w}||^2 + \frac{1}{n}\sum_{i=1}^{n} \log[1 + e^{-z_i(\mathbf{w}^t\mathbf{x}_i+b)}] \tag{171}$$



We can see that for correctly classified points, the loss will be 0, while for points that are incorrectly classified or that are on the wrong side of the margin, the loss increases linearly with the distance from the separation rule.
SVM can thus be reinterpreted as the minimization of a loss function.

8.4 Kernel

At this point, let's see why we introduced SVMs for obtaining linear separation rules when LR already provides them. Actually, SVM doesn't even produce score with a probabilistic interpretation.
What makes SVM a powerfull tool, is the possibility to train the model to obtain multidimensional hyperplanes without the need to transform the feature space. In fact, we know from the dual formulation, that the score for a test sample $\mathbf{x}_t$ depends only on dot products of the input samples. Training the model also depends only on dot products.
So if we define a transformation $\mathbf{\Phi}(\mathbf{x}_i)$ to go in the expanded feature space, we don't need to actually transform our features if we can build a function that directly computes dot products

$$k(\mathbf{x}_1, \mathbf{x}_2) = \mathbf{\Phi}(\mathbf{x}_1)^T\mathbf{\Phi}(\mathbf{x}_2) \tag{172}$$

In fact, in that case, both training and scoring can be performed using just k. So we would have that

$$\mathbf{H}_{ij} = z_i z_j \mathbf{\Phi}(\mathbf{x}_i)^T \mathbf{\Phi}(\mathbf{x}_j) = z_i z_j k(\mathbf{x}_i, \mathbf{x}_j) \tag{173}$$

and

$$s(\mathbf{x}_t) = \sum_{i=1|\alpha_i>0}^{n} \alpha_i z_i \mathbf{\Phi}(\mathbf{x}_i)^T \mathbf{\Phi}(\mathbf{x}_j) + b = \sum_{i=1|\alpha_i>0}^{n} \alpha_i z_i k(\mathbf{x}_i, \mathbf{x}_j) + b \tag{174}$$

Function k is called **kernel function**. A kernel function allows training a SVM in a large dimensional space, without requiring to explicitly compute the mapping.
Also, when scoring with the primal formulation

$$s(\mathbf{x}_t) = \mathbf{w}^t \mathbf{x}_t + b \tag{175}$$

the complexity would depend by the dimension D of the feature space, that may be too large even for finite expanded spaces. When scoring with the dual formulation, instead, the complexity depends only on the number of training points.

We have already seen an example of feature expansion that provides quadratic separation surfaces, given by the mapping

$$\mathbf{\Phi}(\mathbf{x}) = \begin{bmatrix} \mathrm{vec}(\mathbf{x}\mathbf{x}^T) \\ \sqrt{2}\mathbf{x} \\ 1 \end{bmatrix} \tag{176}$$

Computing a dot product explicitly in this feature space would take $O(D^2)$, where D is the size of the original feature space. However, we can see that

$$\mathbf{\Phi}(\mathbf{x}_1)^T \mathbf{\Phi}(\mathbf{x}_2) = (\mathbf{x}_1^T \mathbf{x}_2)^2 + 2\mathbf{x}_1^T \mathbf{x}_2 + 1 \tag{177}$$

and define the kernel function

$$k(\mathbf{x}_1, \mathbf{x}_2) = (\mathbf{x}_1^T \mathbf{x}_2 + 1)^2 \tag{178}$$

that works in $O(D)$.
In general, we can define the **polynomial kernels** of degree d as

$$k(\mathbf{x}_1, \mathbf{x}_2) = (\mathbf{x}_1^T \mathbf{x}_2 + 1)^d \tag{179}$$

The fact that we can compute non-linear separation surfaces throught the kernel function, without actually expanding the feature space, is called **kernel trick**. In general, it is difficult to know for which kernel function we can actually find a transformation $\mathbf{\Phi}$. We will generally use well known kernels, or combinations of them. For example, it is proven that the sum of two kernels is still a kernel.

We have already seen the polynomial kernels, but there exist also an other important kernel which is associated to an *infinite-dimensional* mapping: the **Gaussian Radial Basis** Function kernel (RBF):

$$k(\mathbf{x}_1, \mathbf{x}_2) = e^{-\gamma||\mathbf{x}_1 - \mathbf{x}_2||^2} \tag{180}$$

We can see that this kernel function is close to 1 when the two points are very close, otherwise it goes to 0 very fast. Because of how we compute the scores, this means that only the closest points are relevant for the score of a test sample $\mathbf{x}_t$.
The parameter γ defines the *width* of the kernel:

- small γ: the kernel is *wide*, a S.V. influences most other points
- large γ: the kernel is *narrow*, a S.V: has very small influence on points that are not close. If we also assume $b = 0$, we obtain the same classification rule of 1-NN.

γ is an hyperparameter, like C, but since there is no dependence between the two we should always try all suitable pairs (γ, C) to determine the best values for hyperparameters. Cross-validation can also be applied for the purpose.

8.5 Considerations

In some cases linear classifiers are sufficient: in such case it isn't effieicnt to train multidimensional separation surfaces, since we know fast algorithms for training the primal problem. Furthermore, linear SVM scoring can be implemented as a simple dot product.
SVM training is not invariant under affine transformations: in particular, the bias term influences the result if we include it in the normalization term. Because of this it is often useful to center and whiten the data.

SVM scores have no probabilistic interpretation: score post-processing can be applied to estimate class posterior probabilities.
We also have to take care when training with highly unbalanced datasets or when the empirical training set prior is significantly different from the target application prior. In this case, weighting the costs of different errors may be beneficial. To re-balance the classes we can use a different value of C for different classes

$$\arg\min_{\mathbf{w},b} \frac{1}{2}||\mathbf{w}||^2 + \sum_{i=1}^{n} C_i[1 - z_i(\mathbf{w}^T\mathbf{x}_i + b)] \tag{181}$$

and the corresponding dual problem becomes

$$\arg\max_{\boldsymbol{\alpha}} \boldsymbol{\alpha}^T\mathbf{1} - \frac{1}{2}\boldsymbol{\alpha}^T\boldsymbol{H}\boldsymbol{\alpha} \tag{182}$$

$$\text{s.t.} \begin{cases} \sum_{i=1}^{n} \alpha_i z_i = 0 \\ 0 \leq \alpha_i \leq C_i, \quad i = 1, \dots, n \end{cases}$$

For class balancing, we can set $C_i = C_T$ for samples of class $\mathcal{H}_T$ and $C_i = C_F$ otherwise.
We can select $C_T = C\frac{\pi_T}{\pi_T^{emp}}$ and $C_F = C\frac{\pi_F}{\pi_F^{emp}}$.

Applying this classifier on the MNIST dataset gives great performances, in every case: linear, poly ($d = 2$) and RBF ($\gamma = 2.0$). In particular, RBF is the best performer.

SVMs provide good results for binary tasks, but it's difficult to extend to multiclass problems. There are several possibilities, for example:

- Train over all possible pairs of classes
- Train "one-versus-all" models (e.g. class 1 vs classes 2...K)

In both cases, a voting scheme is needed.
Alternatively, we can introduce an objective function that maximizes the margin between each class and all the others.

Finally, let's observe that the kernel trick is not restricted to SVMs. In fact, it can be applied whenever we have a problem which depends only on dot products, like Kernel PCA, Gaussian Processes, ...
Also Logistic Regression can incorporate kernels.

9 Gaussian Mixture Models

9.1 Motivation

One big issue of the generative models seen up to now, is that they assume a certain distribution that in many cases is not accurate, for example the MVG assumes Gausian distribution of samples.
Gaussian Mixture Models are an alternative to model a generic distribution. They allow approximating any sufficiently regular distribution to a desired degree. Of course, since we are estimating the density from data, we require a sufficient amount of data to obtain good estimates.
GMMs can be used to solve also many other problems, for example they provide an alternative to K-means.

We have already encountered an example of GMM. For the Gaussian classifier, we know that the samples of each class are modeled by

$$f_{\mathbf{X}|C}(\mathbf{x}|c) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) \tag{183}$$

To compute class posterior probabilities we had to compute the marginal $f_{\mathbf{X}}(\mathbf{x})$ using class priors as

$$f_{\mathbf{X}}(\mathbf{x}) = \sum_{c=1}^{K} f_{\mathbf{X}|C}(\mathbf{x}|c)P(C=c) = \sum_{c=1}^{K} \pi_c \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) \tag{184}$$

This is an example of a K-compontents Gaussian Mixture Model. More in general, a Guassian Mixture Model is a density model obtained as a weighted combination of Gaussians $\mathbf{X} \sim GMM(\mathbf{M}, \boldsymbol{\mathcal{S}}, \mathbf{w})$

$$f_{\mathbf{X}}(\mathbf{x}) = \sum_{c=1}^{K} w_c \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) \tag{185}$$

The distributions parameters are the component means $\mathbf{M} = [\boldsymbol{\mu}_1, \dots, \boldsymbol{\mu}_K]$, the component covariances $\boldsymbol{\mathcal{S}} = [\boldsymbol{\Sigma}_1, \dots, \boldsymbol{\Sigma}_K]$ and the weights $\mathbf{w} = [w_1, \dots, w_K]$. Since the integral of $f_{\mathbf{X}}$ must be 1, we have that the weights must sum to 1.

9.2 Hard cluster assignments

Given a dataset $\mathcal{D} = [\mathbf{x}_1, \dots, \mathbf{x}_n]$ we assume that the samples have been independently generated by a GMM. We assume that RVs describing the samples are i.i.d., as always. In particular

$$\mathbf{X}_i \sim \mathbf{X} \sim GMM(\mathbf{M}, \boldsymbol{\mathcal{S}}, \mathbf{w}) \tag{186}$$

If we are using GMMs for classification, $\mathcal{D}$ may correspond to the samples of a given class. However, in the following, we do not assume any specific task, so that $\mathcal{D}$ is just a set of samples that we want to model by means of a GMM. In particular, we consider the dataset $\mathcal{D}$ as unlabeled.

We can resort to ML to estimate the model parameters that best describe $\mathcal{D}$. However, ML estimation for GMMs is an ill-posed problem. As long as we have more than 1 component, we can devise degenerate solutions for which the likelihood is not bounded above.
In practice, if we combine some heuristics with the ML approach, we can get a good density estimate.
Given the model parameters $\boldsymbol{\theta} = [\mathbf{M}, \boldsymbol{\mathcal{S}}, \mathbf{w}]$, we can write the likelihood as

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{i=1}^{n} f_{\mathbf{X}_i}(\mathbf{x}_i) = \prod_{i=1}^{n} \sum_{c=1}^{K} w_c \mathcal{N}(\mathbf{x}_i | \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) \tag{187}$$

and the corresponding log-likelihood

$$l(\boldsymbol{\theta}) = \sum_{i=1}^{n} \log(\sum_{c=1}^{K} w_c \mathcal{N}(\mathbf{x}_i | \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)) \tag{188}$$

How do we solve this for $\boldsymbol{\theta}$?

A GMM can be interpreted as the marginal of a joint distribution of data points and corresponding clusters

$$f_{\mathbf{X}_i}(\mathbf{x}_i) = \sum_{c=1}^{K} f_{\mathbf{X}_i|C_i}(\mathbf{x}_i|c_i) P(C_i = c) = \sum_{c=1}^{K} w_c \mathcal{N}(\mathbf{x}_i; \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) \tag{189}$$

In the Gaussian model, w_c is the prior of class $C = c$, but in this case it is the weight assigned to a cluster of points. In particular, we imagine that our data is partitioned in such a way that the distribution of each cluster can be modeled by a Gaussian. If we knew the component responsible for each sample, we could estimate the parameters of each Gaussian by ML from the points of each cluster.
In general, the clusters are unknown. We treat the cluster memberships as unobserved (**latent**) random variables. Intuitively, we want to estimate both cluster assignments and model parameters as to maximize the marginal distribution of the data.
Basically, what we do is assuming parameters and assignign clusters based on **cluster posterior probabilities**, that can be computed using the assumed parameters. Then, we use the clustering to recompute the parameters and repeat the process. This is quite similar to the K-means idea.
So, let's consider a set of GMM parameters $\boldsymbol{\theta}$. The GMM defines a *joint density* of clusters. The density for sample $\mathbf{x}_i$ and cluster c is

$$f_{\mathbf{X}_i|C_i}(\mathbf{x}_i|c) = w_c \mathcal{N}(\mathbf{x}_i; \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) \tag{190}$$

We can compute *cluster posterior probabilities* as

$$\gamma_{c,i} = P(C_i = c|\mathbf{X}_i = \mathbf{x}_i) = \frac{f_{\mathbf{X}_i,C_i}(\mathbf{x}_i, c)}{f_{\mathbf{X}_i}(\mathbf{x}_i)} = \frac{w_c \mathcal{N}(\mathbf{x}_i; \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)}{\sum_{c'} w_{c'} \mathcal{N}(\mathbf{x}_i; \boldsymbol{\mu}_{c'}, \boldsymbol{\Sigma}_{c'})} \tag{191}$$

These are also called **responsibilities**. As a first approximation, we decide to assign a point to the cluster c with highest responsibility. We can thus associate a cluster label to each sample

$$c_i^* = \arg\max_c P(C_i = c|\mathbf{X}_i = \mathbf{x}_i) \tag{192}$$

Given the cluster assignments, we can then estimate by ML the new GMM parameters $\boldsymbol{\theta}^{new}$.
We treat the cluster assignments as if they were known class labels. The log-likelihood is similar to that of a multivariate Gaussian classifier

$$l(\boldsymbol{\theta}) = \sum_{i=1}^{n} [\log f_{\mathbf{X}_i|C_i}(\mathbf{x}_i|c_i^*) + \log P(C_i = c_i*)] \tag{193}$$

that expands to

$$l(\boldsymbol{\theta}) = \sum_{i=1}^{n} \log \mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_{c_i^*}, \boldsymbol{\Sigma}_{c_i^*}) + \sum_{i=1}^{n} \log w_{c_i^*} \tag{194}$$

i.e. it corresponds to a sum of two terms that depend on different subset of the parameters

$$l(\boldsymbol{\theta}) = l_{\mathcal{N}}(\mathbf{M}, \boldsymbol{\mathcal{S}}) + l_{\mathcal{C}}(\mathbf{w}) \tag{195}$$

The first term corresponds to the log-likelihood of a MVG classification model, where the class labels are assumed to be estimated by c_i^*, i.e. they are the assigned cluster labels. We already know that in this case the mean and covariance matrix of each class (cluster) correspond to the empirical mean and empirical covariance matrix of that class (cluster).
The second term corresponds to the log-likelihood of a categorical model with parameters w_c, that we already solved too: the solution is given by the empirical "weight" of each class (cluster), i.e.

$$w_c^* = \frac{N_c}{\sum_{c=1}^{K} N_c} \tag{196}$$

At this point we have obtained $\boldsymbol{\theta}^{new}$, and we can repeat the process by computing new cluster assignments using $\boldsymbol{\theta}^{new}$, and use the updated assignments to update once again the model parameters, stopping when some criterion is met.

Now, let's also assume that we fix the covariance matrices of the GMM model to $\boldsymbol{\Sigma}_c = \mathbf{I}$, and the weights to $w_c = \frac{1}{K}$. In this case, cluster assignments correspond to the rule

$$c_i^* = \arg\max_c P(C_i = c|\mathbf{X}_i = \mathbf{x}_i) = \arg\min_c ||\mathbf{x}_i - \boldsymbol{\mu}_c||^2 \tag{197}$$

i.e. we assign each sample to the cluster with the nearest mean. If means are centroid, this is the K-means rule. From this we see that GMM can be applied to clustering tasks as a generalization of K-means.

This idea has a problem though: we form **hard clusters**, i.e. a point is assigned to one and only one component of the GMM. This is fine if $P(C_i = c_i^*|\mathbf{X}_i = \mathbf{x}_i) \simeq 1$, but when $P(C_1 = c_1|\mathbf{X}_i) \simeq P(C_2 = c_2|\mathbf{X}_i)$ we are making a crude approximation: both c_1 and c_2 have been responsible for the generation of $\mathbf{x}_i$, but we are taking into account only one.
In general, the algorithm we discussed is not maximizing the likelihood of observed samples.

9.3 Soft cluster assignments

Let's now extend the algorithm to handle **soft assignments**. We will see that a point is not copmletely associated to a signle Gaussian component, but it contributes to the estimation of different components according to its cluster posterior probability.
Consider the log-likelihood for our data

$$\sum_{i=1}^{n} \log f_{\mathbf{X}_i}(\mathbf{x}_i|\boldsymbol{\theta}) = \sum_{i=1}^{n} \log(\sum_{c=1}^{K} w_c \mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)) \tag{198}$$

By setting to $\mathbf{0}$ the gradient with respect to $\boldsymbol{\mu_c}$ we obtain

$$\boldsymbol{\mu}_c = \frac{\sum_i \gamma_{c,i}\mathbf{x}_i}{\sum_i \gamma_{c,i}} = \frac{\mathbf{F}_c}{N_c} \tag{199}$$

The equation can be interpreted as a weighted empirical mean, where the weight of each sample corresponds to its responsibility. If we knew responsibilities $\gamma_{c,i}$, we could compute $\boldsymbol{\mu}_c$. However, $\gamma_{c,i}$ depends on $\boldsymbol{\mu}_c$.
The terms N_c and $\mathbf{F}_c$ are called **zero order statistic** and **first order statistic**.
We can adopt a similar strategy for the covariance matrix, obtaining

$$\boldsymbol{\Sigma}_c = \frac{1}{N_c}\sum_i \gamma_{c,i}\mathbf{x}_i\mathbf{x}_i^T - \boldsymbol{\mu}_c\boldsymbol{\mu}_c^T = \frac{\mathbf{S}_c}{N_c} - \boldsymbol{\mu}_c\boldsymbol{\mu}_c^T \tag{200}$$

where $\mathbf{S}_c$ is called **second order statistic**.
The weights can be re-estimated as

$$w_c = \frac{N_c}{N} = \frac{\sum_i \gamma_{c,i}}{\sum_{c=1}^{K}\sum_i \gamma_{c,i}} \tag{201}$$

Since we are not given $\gamma_{c,i}$ we can follow the same procedure we used for hard assignments:

- Given $\boldsymbol{\theta}$ we estimate the responsibilities for each sample of the dataset

- Given the responsibilities, we re-estimate the GMM parameters $\boldsymbol{\theta}$ using the previous expressions

This procedure is a particular instance of an algorithms known as **Expectation-Maximization**, that we won't see in details.

9.4 Considerations

As we did for MVG, we can also train a GMM with diagonal covariance matrices, to reduce the number of parameters to estimate (reducing overfitting and computational costs). The solution is given by the diagonals of $\boldsymbol{\Sigma}_c$s that we defined before. Be careful that in this case the covariance assumption **doesn't** correspond to the Naive Bayes assumption. The Naive Bayes assumption would correspond to train a different GMM model for each set of features that are assumed independent from the others.
Finally, we may also assume that all the components of the GMM have the same covariance matrix (**tied** GMM).

A critical step when implementing the GMM model is the initialization of the parameters. For example, we can use K-means (hard assignments with covariances equal to the identity matrix) to get a first approximation of cluster assignments.
An alternative approach is known as **LBG**, which allows us to estimate a 2G-components GMM from a G-components GMM (so we may start with a 1-component GMM given, for example, by a simple MVG estimate). This procedure works by splitting the G-components GMM, in the sense that we compute a $\boldsymbol{\mu}_c^+ = \boldsymbol{\mu}_c + \epsilon$ and a $\boldsymbol{\mu}_c^- = \boldsymbol{\mu}_c - \epsilon$ for each mean μ_c. Covariance matrices can be kept the same. Then we run the EM algorithm until convergence for the 2G-components GMM and iterate the process until the desired number of components is reached.
A good value for ϵ can be computed from the leading eigenvector of the covariance matrix $\boldsymbol{\Sigma}_c$.
Sometimes it's hard to tell what the correct number of gaussians is: in these cases we can resort to cross-validation, but we should be pay attention to degenerate models. In fact, even if the EM algorithm usually finds a local max that has a good behaviour, it may sometimes give us a degenerate model, especially if we have a large number of components. Some heuristics can be used to force models to be well-behaved, for example by imposing minimum values for the eigenvalues of the covariance matrices. Alternatively, we may change the initialization so that the algorithm ends up in a different local max.

9.5 Classification with GMM

To perform classification using the GMM distributions, it is necessary to fit one GMM for each class. In particular, we should fit a GMM with K_c copmonents (clusters) for each class c.

Then we may use the computed densities to compute class posterior probabilities as we did for the MVG model.

$$P(C_t = c|\mathbf{X}_t = \mathbf{x}_t) = \frac{P(C_t = c) \sum_{k=1}^{K_c} w_{c,k} \mathcal{N}(\mathbf{x}_t|\boldsymbol{\mu}_{c,k}, \boldsymbol{\Sigma}_{c,k})}{\sum_{c'} P(C_t = c') \sum_{k=1}^{K'_c} w_{c',k} \mathcal{N}(\mathbf{x}_t|\boldsymbol{\mu}_{c',k}, \boldsymbol{\Sigma}_{c',k})} \tag{202}$$

Alternatively, for binary tasks, we may compute the log-likelihood ratio (assuming class labels 1 and 0)

$$llr(\mathbf{x}_t) = \frac{\sum_{k=1}^{K_1} w_{1,k} \mathcal{N}(\mathbf{x}_t|\boldsymbol{\mu}_{1,k}, \boldsymbol{\Sigma}_{1,k})}{\sum_{k=1}^{K_0} w_{0,k} \mathcal{N}(\mathbf{x}_t|\boldsymbol{\mu}_{0,k}, \boldsymbol{\Sigma}_{0,k})} \tag{203}$$

and compare this to the desired treshold.

GMM also allows us to do an **open-set** multiclass classification. The difficulty with this type of classification tasks, is to build a robust model for the *none-of-others* class. This class is usually very heterogeneous, and explicitly modeling its sub-components requires labeled examples of its possible objects. If we assume that samples of known classes can be modeled by MVG distributions, we can collect a large set of unlabeled samples for the *none-of-others* class and model them as a GMM distribution.
However, given the complexity of the task, these kind of models often provide class-conditional likelihoods that are not calibrated.