Laboratory 7

In this laboratory we will focus on Logistic Regression models for classification.

Numerical optimization

The Logistic Regression model is obtained by minimizing the average cross-entropy between the model predictions and the observed labels. As we have seen, this corresponds also to a Maximum Likelihood solution for the observed labels. While for Gaussian models closed form expressions are available for the ML solutions, this is not the case for Logistic Regression. Therefore, we turn to numerical optimization to find the maximizer of the class likelihoods, or, equivalently, the minimizer of the average cross-entropy.

Numerical optimization algorithms look for the minima of a function $f(\boldsymbol{x})$ with respect to the argument $\boldsymbol{x}$. A simple, iterative method to find a local minimum of f is gradient descent (GD). Given a point $\boldsymbol{x}_t$, gradient descent looks for a descent direction of the function. The direction is given by the negative of the gradient of f. The algorithm then moves a step α_t from $\boldsymbol{x}_t$ along the descent direction:

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t - \alpha_t \nabla_{\boldsymbol{x}} f(\boldsymbol{x})$$

Under mild assumptions on α_t (e.g. $\alpha_t \to 0$, $\sum_{t=1}^{\infty} \alpha_t \to \infty$) the algorithm converges to a local minimum of f.

A drawback of gradient descent is that it can be quite slow. Faster convergence can be obtained by considering second-order information, such as the Hessian of the function. In this laboratory we will use the L-BFGS algorithm. L-BFGS builds an incremental approximation of the Hessian, that is used to identify a search direction p_t at each iteration. The algorithm then proceeds at finding an acceptable step size α_t for the search direction p_t, and uses the direction and step size to update the solution.

The algorithm is implemented in `scipy` (requires importing `scipy.optimize`). We will use the `scipy.optimize.fmin_l_bfgs_b` interface to the numerical solver.

`scipy.optimize.fmin_l_bfgs_b` requires at least 2 arguments (check the documentation for more details):

- `func`: the function we want to minimize.
- `x0`: the starting value for the algorithm.

The L-BFGS algorithm requires computing the objective function and its gradient. To pass the gradient we have different options:

- Through `func`: `func` should return a tuple $(f(\boldsymbol{x}), \nabla_{\boldsymbol{x}} f(\boldsymbol{x}))$
- Through the optional parameter `fprime`: `fprime` is a function computing the gradient. In this case, `func` should only return the objective value $f(\boldsymbol{x})$
- Let the implementation compute an approximated gradient: pass `approx_grad = True`. Also in this case, `func` should only return the objective value $f(\boldsymbol{x})$

The last option does not require writing a function that computes the gradient, as an approximation of the gradient is automatically obtained through finite differences. While this has the advantage that we do not need to derive and implement the gradient, it has two drawbacks:

- The gradient computed through finite differences may not be accurate enough
- The computations are much more expensive, since we need to evaluate the objective function a number of times at least D, where D is the size of $\boldsymbol{x}$, at each iteration, and if we want a more accurate approximation of the gradient we may need to evaluate f many more times

For example, a way to compute a numerical approximation of the gradient consists in computing

$$\frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}_i} \approx \frac{f(\boldsymbol{x} + \epsilon \boldsymbol{e}_i) - f(\boldsymbol{x} - \epsilon \boldsymbol{e}_i)}{2\epsilon}$$

where $\boldsymbol{e}_i$ is a vectors of zeros, except the element in position i, which is one: $\boldsymbol{e}_1 = [1, 0, 0 \dots 0], \boldsymbol{e}_2 = [0, 1, 0 \dots 0] \dots \boldsymbol{e}_D = [0, 0, 0 \dots 1]$, and ϵ is a small value, e.g. $\epsilon = 10^{-7}$. This requires computing f a number of times equal to $2D$.

Using the numerical solver, find the minimum of

$$f(y, z) = (y + 3)^2 + \sin(y) + (z + 1)^2$$

The function is convex, so it has a unique minimum.

STEP 1: Implement f. The sin function can be computed using `numpy.sin` . `f` should accept a 1-D numpy array `x` of shape `(2,)`. The first component corresponds to variable y, while the second corresponds to variable z. The function `f` should return the value $f(y, z)$.

STEP 2: Call the numerical optimization function `scipy.optimize.fmin_l_bfgs_b` . Pass to the function the previously implemented `f` , and `approx_grad = True` . As starting point you can use values $[0, 0]$ (pass a numpy array, not a list). If you pass the optional argument `iprint = 1` you can visualize the iterations of the algorithm.

`scipy.optimize.fmin_l_bfgs_b` returns a tuple with three values `x, f, d`:

- `x` is the estimated position of the minimum
- `f` is the objective value at the minimum
- `d` contains additional information (check the documentation)

You should find the minimum at `[-2.57747138, -0.99999927]`, with value (truncated) `-0.356143012`.

We can also try providing an explicit gradient:

$$\frac{\partial f(y, z)}{\partial y} = 2(y + 3) + \cos(y) \ , \quad \frac{\partial f(y, z)}{\partial z} = 2(z + 1)$$

Rewrite function `f` so that it returns $f(\boldsymbol{x})$ as well the gradient of f as a numpy array with shape `(2,)`. Call again the solver, but do not pass `approx_grad` . You should obtain `x = [-2.57747137, -0.99999927]` . In this case the numerical approximation was good enough. However, check the values of the third returned value `d` in the two cases. `'funcalls'` provides the number of times `f` was called. The numerical approximation of the gradient is significantly more expensive, and the cost becomes relatively worse when the dimensionality of the domain of f increases.

Binary logistic regression

We can now turn our attention to Logistic Regression. In this section we will implement the binary version of the logistic regression to discriminate between iris virginica and iris versicolor. We will ignore iris setosa. We will represent labels with 1 (iris versicolor) and 0 (iris virginica).

You can load the filtered data with

```
def load_iris_binary():
    D, L = sklearn.datasets.load_iris()['data'].T, sklearn.datasets.load_iris()
        ['target']
    D = D[:, L != 0] # We remove setosa from D
    L = L[L!=0] # We remove setosa from L
    L[L==2] = 0 # We assign label 0 to virginica (was label 2)
    return D, L

D, L = load_iris_binary()
(DTR, LTR), (DVAL, LVAL) = split_db_2to1(D, L)
```

Function `split_db_2to1` was defined in Laboratory 5.

The regularized Logistic Regression objective can be written in different ways (we adopt the average-risk expression which divides the loss by the number of samples):

$$J(\boldsymbol{w},b) = \frac{\lambda}{2}\|\boldsymbol{w}\|^2 - \frac{1}{n}\sum_{i=1}^{n}\left[c_i \log\sigma(\boldsymbol{w}^T\boldsymbol{x}_i+b) + (1-c_i)\log\left(1-\sigma(\boldsymbol{w}^T\boldsymbol{x}_i+b)\right)\right] \tag{1}$$

$$J(\boldsymbol{w},b) = \frac{\lambda}{2}\|\boldsymbol{w}\|^2 + \frac{1}{n}\sum_{i=1}^{n}\log\left(1+e^{-z_i(\boldsymbol{w}^T\boldsymbol{x}_i+b)}\right) \;, \quad z_i = \begin{cases} 1 & \text{if } c_i = 1 \\ -1 & \text{if } c_i = 0 \end{cases} \text{ (i.e. } z_i = 2c_i - 1) \tag{2}$$

$$J(\boldsymbol{w},b) = \frac{\lambda}{2}\|\boldsymbol{w}\|^2 + \frac{1}{n}\sum_{i=1}^{n}\left[c_i \log\left(1+e^{-\boldsymbol{w}^T\boldsymbol{x}_i-b}\right) + (1-c_i)\log\left(1+e^{\boldsymbol{w}^T\boldsymbol{x}_i+b}\right)\right] \tag{3}$$

Note that (3) follows either from (2), observing that $c_i = 1$ for $z_i = 1$ and $c_i = 0$ for $z_i = -1$, or from (1), observing that

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

and

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

The reason for preferring (2) or (3) to (1) is due to numerical issues that may arise when explicitly computing sigmoids followed by natural logarithms (see below).

Implement Logistic regression using expression (2). This requires minimizing expression (2). For this, you need to write a function `logreg_obj` that, given $\boldsymbol{w}$ and b, allows computing $J(\boldsymbol{w},b)$. You can then provide this function to the numerical solver to obtain the minimizer of J.

NOTES:

- Function `logreg_obj` should receive a single numpy array `v` with shape `(D+1,)`, where `D` is the dimensionality of the feature space (e.g. $D = 4$ for IRIS). `v` should pack all model parameters, i.e. $\boldsymbol{v} = [\boldsymbol{w}, b]$. Inside the function you can then unpack the array e.g. `w, b = v[0:-1], v[-1]`
- The function `logreg_obj` needs to access also `DTR`, LTR and λ, which are required to compute the objective. You can address this in different, alternative ways:
  - Embed the objective function and its optimization *inside* a function `trainLogReg` that receives both `DTR`, `LTR` and a value for `l`. The objective function is defined *inside* `trainLogReg`, thus it can access the variables `DTR` and `LTR` in the outer function scope (i.e., those passed to `trainLogReg`. The code looks like:

```
def trainLogReg(DTR, LTR, l):

    def logreg_obj(v):
        # ...
        # Compute and return the objective function value using DTR,
            LTR, l
        # ...

    # Here we find the minimizer of logreg_obj
    xf = scipy.optimize.fmin_l_bfgs_b(func = logreg_obj, x0 = numpy.
        zeros(DTR.shape[0]+1), approx_grad=True)[0]
    return xf
```

  - Write a function `logreg_obj` that accepts additional arguments `logreg_obj(v, DTR, LTR, l)`. Optimize the function by calling `scipy.optimize.fmin_l_bfgs_b`. You can pass the additional arguments to the objective function through the `args` parameter of `fmin_l_bfgs_b` (check the documentation of `scipy.optimize.fmin_l_bfgs_b`):

```
def logreg_obj(v, DTR, LTR, l):
     # ...
     # Compute and return the objective function value using DTR, LTR,
         l
     # ...

# in the main portion, after loading the data:
scipy.optimize.fmin_l_bfgs_b(func = logreg_obj, args=(DTR, LTR, l),
    ...)
```

– Write a function `logreg_obj_wrap` that accepts as input `DTR`, LTR and λ. Inside the function, define `logreg_obj` as before. `logreg_obj` has now access to the scope of the enclosing function `logreg_obj_wrap`. Make `logreg_obj_wrap` return the created function.

```
def logreg_obj_wrap(DTR, LTR, l):
    def logreg_obj(v):
        # ...
        # Compute and return the objective function value using DTR,
            LTR, l
        # ...
    return logreg_obj

# in the main portion, after loading the data:
logreg_obj = logreg_obj_wrap(DTR, LTR, l)
scipy.optimize.fmin_l_bfgs_b(func = logreg_obj, ...)[0]
```

– `logreg_obj_wrap` can also be any callable object that accepts a single parameter. You can use an instance (object) of a class that has a single method `logreg_obj(self, v)`, and store in the object the values you need to access. These can be passed to the method that initializes the object

```
class logRegClass:
    def __init__(self, DTR, LTR, l):
        self.DTR = DTR
        self.LTR = LTR
        self.l = l
    def logreg_obj(self,  v):
        # Compute and return the objective function value. You can
            retrieve all required information from self.DTR, self.LTR,
            self.l

# in the main portion, after loading the data, instantiate a new object
logRegObj = logRegClass(DTR, LTR, l)
# You can now use logRegObj.logreg_obj as objective function:
scipy.optimize.fmin_l_bfgs_b(func = logRegObj.logreg_obj, ...)
```

- The computation of $\log\left(1+e^{-z_i(\boldsymbol{w}^T\boldsymbol{x}_i+b)}\right)$ can lead to numerical issues when $z_i(\boldsymbol{w}^T\boldsymbol{x}_i+b)$ is large, since the sum will make the contribution of the exponential term disappear. We can avoid the issue by using the `numpy.logaddexp` function, which computes

$$\texttt{numpy.logaddexp}(a,b) = \log\left(e^a+e^b\right) \ .$$

  In our example, we need to compute `numpy.logaddexp` $\left(0, -z_i(\boldsymbol{w}^T\boldsymbol{x}_i+b)\right)$.

- Broadcasting can significantly speed-up the computations. You can compute a vector of "scores" `S`

$$\boldsymbol{S} = [(\boldsymbol{w}^T\boldsymbol{x}_1+b)\dots(\boldsymbol{w}^T\boldsymbol{x}_n+b)]$$

  using simple matrix-vector multiplication: `S = (vcol(w).T @ DTR + b).ravel()`. Remember to reshape the result to a 1-D array (`.ravel()`). You can then multiply each element of the resulting vector by the corresponding label: `-ZTR * S`, where `ZTR = 2 * LTR - 1`. `numpy.logaddexp`

supports broadcasting so you can compute in a single shot all terms

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

with `numpy.logaddexp(0, -ZTR * S)`

- To speed-up computations, it's useful to provide also the gradient of the function. We can express in vector form the derivatives with respect to the components of $\boldsymbol{w}$ and b as (notice that there is no minus in the exponential term $z_i(\boldsymbol{w}^T\boldsymbol{x}_i+b)$):

$$\nabla_{\boldsymbol{w}}J = \left[\frac{\partial J}{\partial \boldsymbol{w}_1}\cdots\frac{\partial J}{\partial \boldsymbol{w}_d}\right] = \lambda\boldsymbol{w} + \frac{1}{n}\sum_{i=1}^{n}\frac{-z_i}{1+e^{z_i(\boldsymbol{w}^T\boldsymbol{x}_i+b)}}\boldsymbol{x}_i$$

$$\frac{\partial J}{\partial b} = \sum_{i=1}^{n}\frac{-z_i}{1+e^{z_i(\boldsymbol{w}^T\boldsymbol{x}_i+b)}}$$

where d is the dimensionality of the feature vectors (`DTR.shape[0]`). These terms can be efficiently computed by computing the vector `G`:

$$\boldsymbol{G} = \left[\frac{-z_1}{1+e^{z_1(\boldsymbol{w}^T\boldsymbol{x}_1+b)}}\cdots\frac{-z_n}{1+e^{z_n(\boldsymbol{w}^T\boldsymbol{x}_n+b)}}\right]$$

so that

$$\nabla_{\boldsymbol{w}}J = \lambda\boldsymbol{w} + \frac{1}{n}\sum_{i=1}^{n}\boldsymbol{G}_i\boldsymbol{x}_i$$

$$\frac{\partial J}{\partial b} = \sum_{i=1}^{n}\boldsymbol{G}_i$$

which can be computed, using broadcasting, from the vector of scores `S` as `G = -ZTR / (1.0 + numpy.exp(ZTR * S))` (note: you may obtain overflow from the exponentiation due to well-classified samples, however you can safely ignore the issue since it will simply cause the corresponding $\boldsymbol{G}_i$ becoming exactly 0 rather than almost 0, and this will not cause numerical issues). You can then compute all terms $\boldsymbol{G}_i\boldsymbol{x}_i$ with broadcasting: `(vrow(G) * DTR)`.

Remember that `scipy.optimize.fmin_l_bfgs_b` requires that you pack $\nabla_{\boldsymbol{w}}J$ and $\frac{\partial J}{\partial b}$ in a single vector $\boldsymbol{v}_{grad} = \left[\nabla_{\boldsymbol{w}}J, \frac{\partial J}{\partial b}\right]$. You have to modify the function `logreg_obj` so that it returns both the objective and the gradient $\boldsymbol{v}_{grad}$ (and set `approx_grad = False` when calling the optimizer function).

- λ is a hyper-parameter. As usual, we should employ a validation set to estimate good values of λ. For this laboratory, we can simply try different values and see how this affects the performance
- The starting point does not significantly influence the result, since the objective function is convex (there may be slight differences, but should be very small). You can use as initial value an array of all zeros `x0 = numpy.zeros(DTR.shape[0] + 1)`
- The `scipy` implementation of L-BFGS calls the objective function a maximum of 15000 times, and the algorithm stops when this threshold is reached. You can specify a larger amount for the maximum number of calls through the `maxfun` argument
- You can also control the maximum number of allowed iterations through the argument `maxiter`
- You can control the precision of the L-BFGS solution through the parameter `factr` . The default value is `factr=10000000.0` . Lower values result in more precise solutions (i.e. closer to the optimal solution), but require more iterations. Below the default value was used.

Once you have trained the model, you can compute posterior log-probability ratios by simply computing, for each test sample $\boldsymbol{x}_t$, the score

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

Compute the array of scores `S = vcol(w).T @ DVAL + b`. You can then compute class assignments by thresholding the scores with 0 (i.e. `S[i] > 0` $\implies$ `LP[i] = 1`, where `LP` is the array of predicted

labels for the test samples).

To check that you implemented the algorithm correctly, you can find below values of the objective function and error rate (1 - accuracy) for different values of λ.
The logistic regression model provides scores that can be interpreted as log-posterior-ratio for the two classes, with a prior corresponding to the empirical training set prior. To adapt the scores to different applications, however, we need to transform the scores so that they can be interpreted as log-likelihood ratios, otherwise the application-dependent thresholds that we employ for LLRs cannot be used anymore. To obtain a score that behaves like a LLR we can simply remove the empirical prior log-odds from the score:

$$s_{llr}(\boldsymbol{x}_t) = \boldsymbol{w}^T\boldsymbol{x}_t + b - \log\frac{\pi_{emp}}{1-\pi_{emp}} = s(\boldsymbol{x}_t) - \log\frac{\pi_{emp}}{1-\pi_{emp}}$$

where π_{emp} is the empirical training set prior (fraction of samples of class 1).

Compute minimum DCF, optimal Bayes predictions and actual DFCs for different applications. Optimal Bayes predictions require comparing s_{llr} with the standard application-dependent threshold. Below you can find the results for $\pi_T = 0.5$.

	$J(\boldsymbol{w}^*, b^*)$	Error rate ($\boldsymbol{w}^T\boldsymbol{x}_t + b \lessgtr 0$)	minDCF ($\pi_T = 0.5$)	actDCF ($\pi_T = 0.5$)
$\lambda = 10^{-3}$	`1.100009e-01`	8.8%	0.0625	0.1181
$\lambda = 10^{-1}$	`4.539407e-01`	11.8%	0.0556	0.1111
$\lambda = 1.0$	`6.316436e-01`	14.7%	0.1111	0.1667

Prior-weighted logistic regression and calibration

The prior-weighted logistc regression model allows us to simulate different priors for class 1. The objective function becomes

$$J(\boldsymbol{w}, b) = \frac{\lambda}{2}\|\boldsymbol{w}\|^2 + \sum_{i=1}^{n}\xi_i \log\left(1 + e^{-z_i(\boldsymbol{w}^T\boldsymbol{x}_i + b)}\right) \quad , \quad \xi_i = \begin{cases} \frac{\pi_T}{n_T} & \text{if } z_i = +1 \quad (c_i = 1) \\ \frac{1-\pi_T}{n_F} & \text{if } z_i = -1 \quad (c_i = 0) \end{cases}$$

where π_T is the target prior, and n_T and n_F are the number of samples of class 1 and 0, respectively.

Implement the prior-weighted version of the model, and test it for an application with $\pi_T = 0.8$ (you can check the results below). In this case, the gradients can be computed as

$$\nabla_{\boldsymbol{w}} J = \lambda\boldsymbol{w} + \sum_{i=1}^{n}\xi_i \frac{-z_i}{1 + e^{-z_i(\boldsymbol{w}^T\boldsymbol{x}_i + b)}}\boldsymbol{x}_i = \lambda\boldsymbol{w} + \sum_{i=1}^{n}\xi_i \boldsymbol{G}_i\boldsymbol{x}_i$$

$$\frac{\partial J}{\partial b} = \sum_{i=1}^{n}\xi_i \frac{-z_i}{1 + e^{-z_i(\boldsymbol{w}^T\boldsymbol{x}_i + b)}} = \sum_{i=1}^{n}\xi_i \boldsymbol{G}_i$$

where $\boldsymbol{G}$ is defined as in the standard model
To verify your implementation, you can also compute the solution when $\pi_T = \pi_{emp}$. You should obtain the same loss and same output $(\boldsymbol{w}, b)$ as for the original, non-weighted model.

Finally, remember that, to obtain scores that behave like LLRs, when using the prior-weighted model we need to remove the prior log-odds of the application (*not* the empirical prior log-odds), i.e.

$$s_{llr}(\boldsymbol{x}_t) = \boldsymbol{w}^T\boldsymbol{x}_t + b - \log\frac{\pi_T}{1-\pi_T} = s(\boldsymbol{x}_t) - \log\frac{\pi_T}{1-\pi_T}$$

For an application $\pi_T = 0.8$ you should obtain:

	$J(\boldsymbol{w}^*, b^*)$	minDCF ($\pi_T = 0.8$)	actDCF ($\pi_T = 0.8$)
$\lambda = 10^{-3}$	`9.401035e-02`	0.1667	0.2222
$\lambda = 10^{-1}$	`3.606261e-01`	0.0556	0.7222
$\lambda = 1.0$	`4.724715e-01`	0.1111	1.0000

Multiclass logistic regression (optional)

Implement the multiclass version of logistic regression. The objective function to minimize is

$$J(\boldsymbol{W}, \boldsymbol{b}) = \frac{\lambda}{2}\|\boldsymbol{W}\|^2 - \frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K} \boldsymbol{z}_{ik} \log \boldsymbol{y}_{ik}$$

where

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

and

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

NOTE: $\boldsymbol{W}$ is a $D \times K$ matrix, where D is the dimensionality of the features space (i.e. dimensionality of $\boldsymbol{x}_i$). The python function should accept a 1-D numpy array. Represent $\boldsymbol{W}$ as a 1-D numpy array of shape $(D \cdot K,)$, and reshape it only when performing computations (i.e. inside the function that computes J and when computing predictions).

Suggestion: to avoid numerical issues work directly with $\log \boldsymbol{y}_{ik}$:

$$\log \boldsymbol{y}_{ik} = \boldsymbol{w}_k^T \boldsymbol{x}_i + b_k - \log \sum_{j=1}^{K} e^{\boldsymbol{w}_j^T \boldsymbol{x}_i + b_j}$$

1) Compute the matrix of scores $\boldsymbol{S}$: $\boldsymbol{S}_{ki}$ should be equal to $\boldsymbol{S}_{ki} = \boldsymbol{w}_k^T \boldsymbol{x}_i + b_k$, i.e. each column of $\boldsymbol{S}$ contains the scores for sample $\boldsymbol{x}_i$ for all classes. You can compute the score matrix with a single product, exploiting broadcasting: `S = numpy.dot(W.T, DTR) + b`
2) Compute matrix $\boldsymbol{Y}^{log}$ containing $\boldsymbol{Y}^{log}_{ki} = \log \boldsymbol{y}_{ik}$ (note that the indices are swapped: in $\boldsymbol{Y}^{log}$ the first index represents the class, the second the sample). $\boldsymbol{Y}^{log}_{ki}$ can be computed from $\boldsymbol{S}$. Each row of $\boldsymbol{Y}_{log}$ corresponds to the same row of $\boldsymbol{S}$ minus the expression $\log \sum_{j=1}^{n} e^{\boldsymbol{w}_j^T \boldsymbol{x} + b_j}$. The last expression is the log-sum-exp of the rows of $\boldsymbol{S}$.
3) Use the 1-of-K encoding of the labels: matrix $\boldsymbol{T}$ should contain the labels, encoded as $\boldsymbol{T}_{ki} = 1 \iff c_i = k$. The other elements should be 0 (again, the indices are swapped with respect to $\boldsymbol{z}_{ik}$, the first index of $\boldsymbol{T}$ represents the class).
4) The summation in $J(\boldsymbol{w}, b)$ can be computed by element-wise multiplication of $\boldsymbol{T}$ and $\boldsymbol{Y}_{log}$, followed by the summation of all elements of the resulting matrix
5) The squared norm of $\boldsymbol{W}$ corresponds to $\sum_i \sum_j (\boldsymbol{W}_{ij})^2$, i.e. `(W*W).sum()`

Train the model using the data that we used for the Gaussian classifier:

```
D, L = load_iris()
(DTR, LTR), (DVAL, LVAL) = split_db_2to1(D, L)
```

To test the model, compute class posterior probabilities as $P(C = c|\boldsymbol{x}_t) = \boldsymbol{w}_c^T \boldsymbol{x}_t + b_c$. Predict the class with highest posterior probability (we assume uniform mis-classification costs).

The following table contains the training loss and the test error rate for different values of λ

	$J(\boldsymbol{w}^*, b^*)$	Error rate
$\lambda = 10^{-3}$	`9.69097E-2`	4.0%
$\lambda = 10^{-1}$	`0.500591`	6.0%
$\lambda = 1.0$	`0.821155`	18.0%

Project

We analyze the binary logistic regression model on the project data. We start considering the standard, non-weighted version of the model, without any pre-processing.

Train the model using different values for λ. You can build logarthimic-spaced values for λ using `numpy.logspace`. To obtain good coverage, you can use `numpy.logspace(-4, 2, 13)` (check the documentation). Train the model with each value of λ, score the validation samples and compute the corresponding actual DCF and minimum DCF for the primary application $\pi_T = 0.1$. *To compute actual DCF remember to remove the log-odds of the training set empirical prior.* Plot the two metrics as a function of λ (suggestion: use a logartihmic scale for the x-axis of the plot - to change the scale of the x-axis you can use `matplotlib.pyplot.xscale('log', base=10)`). What do you observe? Can you see significant differences for the different values of λ? How does the regularization coefficient affects the two metrics?

Since we have a large number of samples, regularization seems ineffective, and actually degrades actual DCF since the regularized models tend to lose the probabilistic interpretation of the scores. To better understand the role of regularization, we analyze the results that we would obtain if we had fewer training samples. Repeat the prebvious analysis, but keep only 1 out of 50 model training samples, e.g. using data matrices `DTR[:, ::50], LTR[::50]` (apply the filter only on the model training samples, not on the validation samples, i.e., *after* splitting the dataset in model training and validation sets). What do you observe? Can you explain the results in this case? Remember that lower values of the regularizer imply larger risk of overfitting, while higher values of the regularizer reduce overfitting, but may lead to underfitting and to scores that lose their probabilistic interpretation.

In the following we will again consider only the full dataset. Repeat the analysis with the prior-weighted version of the model *(remember that, in this case, to transform the scores to LLRs you need to remove the log-odds of the prior that you chose when training the model).* Are there significant differences for this task? Are there advantages using the prior-weighted model for our application (remember that the prior-weighted model requires that we know the target prior when we build the model)?

Repeat the analysis with the quadratic logistic regression model (again, full dataset only). Expand the features, train and evaluate the models (you can focus on the standard, non prior-weighted model only, as the results you would obtain are similar for the two models), again considering different values for λ. What do you observe? In this case is regularization effective? How does it affect the two metrics?

The non-regularized model is invariant to affine transformations of the data. However, once we introduce a regularization term affine trasnformations of the data can lead to different results. Analyze the effects of centering (optionally, you can also try different strategies, including Z-normalization and whitening, as well as PCA) on the model results. *You can restrict the analysis to the linear model.* Remember that you have to center both datasets *with respect to the model training dataset* mean, i.e., you must not use the validation data to estimate the pre-processing transformation. For this task, you should observe only minor variations, as the original features were already almost standardized.

As you should have observed, the best models in terms of minimum DCF are not necessarily those that provide the best actual DCFs, i.e., they may present significant mis-calibration. We will deal with score calibration at the end of the course. For the moment, we focus on selecting the models that optimize the minimum DCF on our validation set. Compare all models that you have trained up to now, including Gaussian models, in terms of minDCF for the target application $\pi_T = 0.1$. Which model(s) achieve(s) the best results? What kind of separation rules or distribution assumptions characterize this / these model(s)? How are the results related to the characteristics of the dataset features?

Suggestion for upcoming laboratories: the last laboratories will cover score calibration, and will require to evaluate the results of the models that you tested on an held-out evaluation set. We suggest that you save the models, and the corresponding validation scores as well, since these will be required for score calibration (you can skip the models trained with the reduced dataset, as they won't be needed).