# Laboratory 8

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 for 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 $\alpha_t$ 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 $\alpha_t$ (e.g. $\alpha_t \rightarrow 0$ , $\sum_{t=1}^{\infty} \alpha_t \rightarrow \infty$) 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 $\alpha_t$ 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: $\boldsymbol{e}_1 = [1, 0, 0 \ldots 0], \boldsymbol{e}_2 = [0, 1, 0 \ldots 0] \ldots \boldsymbol{e}_D = [0, 0, 0 \ldots 1]$, and $\epsilon$ 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 $f(x)$ 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), (DTE, LTE) = 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):

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

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

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

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 (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) or (3). 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` has to access also `DTR`, LTR and $\lambda$, which are required to compute the objective. You can address this in different ways (choose one, the first is the easiest to implement, the other two options offer more insights on the language):
  - Write function `logreg_obj` so that it accepts additional arguments `logreg_obj(v, DTR, LTR, l)`. Pass the additional arguments when calling `scipy.optimize.fmin_l_bfgs_b`. This can be achieved by passing `args=(DTR, LTR, l)` to `fmin_l_bfgs_b` (check the documentation of `scipy.optimize.fmin_l_bfgs_b`)
  - Write a function `logreg_obj_wrap` that accepts as input `DTR`, LTR and $\lambda$. 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)
```

  - `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(logRegObj.logreg_obj, ...)
```

- The computation of $\log(1+x)$ can lead to numerical issues when x is small, since the sum will make the contribution of $x$ disappear. We can avoid the issue using function `numpy.log1p`, which computes $\log(1+x)$ in a numerically more stable way.
- $\lambda$ is a hyper-parameter. As usual, we should employ a validation set to estimate good values of $\lambda$. 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-likelihood 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`. 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 $\lambda$.

| | $J(\boldsymbol{w}^*, b^*)$ | Error rate |
|---|---:|---:|
| $\lambda = 0$ | 9.8582E-5 | 11.8% |
| $\lambda = 10^{-6}$ | 7.5415E-3 | 11.8% |
| $\lambda = 10^{-3}$ | 0.11000 | 8.8% |
| $\lambda = 1.0$ | 0.63164 | 14.7% |

## 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 $\log \boldsymbol{y}_{ik}$:

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

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 $\boldsymbol{Y}_{ki}^{log} = \log \boldsymbol{y}_{ik}$ (note that the indices are swapped: in $\boldsymbol{Y}^{log}$ the first index represents the class, the second the sample). $\boldsymbol{Y}_{ki}^{log}$ 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), (DTE, LTE) = 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 $\lambda$

| | $J(\boldsymbol{w}^*, b^*)$ | Error rate |
|---|---|---|
| $\lambda = 0$ | 3.94373E-2 | 4.0% |
| $\lambda = 10^{-6}$ | 3.96791E-2 | 4.0% |
| $\lambda = 10^{-3}$ | 9.69097E-2 | 4.0% |
| $\lambda = 1.0$ | 0.821155 | 18.0% |