Laboratory 9

In this laboratory we will focus on Support Vector Machines for binary classification tasks.

Linear SVM

Support Vector Machines are linear classifiers that look for maximum margin separation hyperplanes. The (primal) SVM objective consists in minimizing

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

where n is the number of training samples, C is a hyper-parameter and z_i is the class label for the i-th sample, encoded as

$$z_i = \begin{cases} +1 & \text{if } x_i \text{ belongs to class } \mathcal{H}_T \\ -1 & \text{if } x_i \text{ belongs to class } \mathcal{H}_F \end{cases}$$

As we have seen, to solve the SVM problem we can also consider the dual formulation

$$J^D(\boldsymbol{\alpha}) = -\frac{1}{2}\boldsymbol{\alpha}^T\boldsymbol{H}\boldsymbol{\alpha} + \boldsymbol{\alpha}^T\mathbf{1}$$

$$\text{s. t. } 0 \le \boldsymbol{\alpha}_i \le C \ ,\forall i \in \{1 \dots n\} \ , \quad \sum_{i=1}^{n}\boldsymbol{\alpha}_i z_i = 0$$

where $\mathbf{1}$ is a n-dimensional vector of ones, and $\boldsymbol{H}$ is a matrix whose elements are

$$\boldsymbol{H}_{ij} = z_i z_j \boldsymbol{x}_i^T\boldsymbol{x}_j$$

The SVM dual solution is the maximizer of $J^D(\boldsymbol{\alpha})$. The dual and primal solutions $\boldsymbol{\alpha}^*$ and $\boldsymbol{w}^*$ are related through

$$\boldsymbol{w}^* = \sum_{i=1}^{n}\boldsymbol{\alpha}_i^* z_i \boldsymbol{x}_i$$

and the optimal bias b^* can be computed considering a sample $\boldsymbol{x}_i$ that lies on the margin:

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

and then solving for b^*. As we have discussed, the dual problem also allows for non-linear separation through the kernel trick. Indeed, we can replace dot-products $\boldsymbol{x}_i^T\boldsymbol{x}_j$ with kernel functions $k(\boldsymbol{x}_i, \boldsymbol{x}_j)$, so that $\boldsymbol{H}_{ij} = z_i z_j k(\boldsymbol{x}_i, \boldsymbol{x}_j)$, and we can compute $\boldsymbol{w}^{*T}\boldsymbol{x}$ through kernel functions:

$$\boldsymbol{w}^{*T}\boldsymbol{x} = \sum_{i=1}^{n}\boldsymbol{\alpha}_i^* z_i k(\boldsymbol{x}_i, \boldsymbol{x}) = \sum_{i|\boldsymbol{\alpha}_i \neq 0}\boldsymbol{\alpha}_i^* z_i k(\boldsymbol{x}_i, \boldsymbol{x})$$

The latter expressions can be used, for example, when solving for b^* and when we want to evaluate the score of a test sample $\boldsymbol{x}$.

For linear SVMs, we can optimize either the primal or dual formulation. Unfortunately, the unconstrained formulation of the primal objective function is non-differentiable. While the L-BFGS method may still be able to find the minimizer of J, we have *no guarantee* that the algorithm will stop close to the optimal value of $(\boldsymbol{w}, b)$. Ad-hoc numerical solvers have been developed, however they are out of the scope of our course.

The dual formulation is differentiable, however it contains both box constraints (i.e. constraints of the form $a \le \alpha_i \le b$) and the additional constraint $\sum_{i=1}^{n}\alpha_i z_i = 0$. The L-BFGS algorithm that we employed is able to handle box constraints, however it cannot incorporate the latter.

The constraint $\sum_{i=1}^{n}\alpha_i z_i = 0$ derives from the presence of the bias term in the SVM primal formulation. We therefore slightly modify the SVM problem as to make the constraint disappear. In this way we will be able to employ L-BFGS-B (the B stands for box-constraints) to solve the dual problem.

We reformulate the primal problem as the minimization of

$$\widehat{J}(\widehat{\boldsymbol{w}}) = \frac{1}{2}\|\widehat{\boldsymbol{w}}\|^2 + C\sum_{i=1}^{n}\max(0, 1 - z_i(\widehat{\boldsymbol{w}}^T\widehat{\boldsymbol{x}}_i))$$

where

$$\widehat{\boldsymbol{x}}_i = \begin{bmatrix}\boldsymbol{x}_i\\ 1\end{bmatrix}, \quad \widehat{\boldsymbol{w}} = \begin{bmatrix}\boldsymbol{w}\\ b\end{bmatrix}$$

We can observe that $\widehat{\boldsymbol{w}}^T\widehat{\boldsymbol{x}}_i = \boldsymbol{w}^T\boldsymbol{x} + b$, i.e. the scoring rules have the same expression as the original formulation. However, in contrast with the original SVM problem, we are also regularizing the value of the bias term, since we regularize the norm of $\widehat{\boldsymbol{w}}$:

$$\|\widehat{\boldsymbol{w}}\|^2 = \|\boldsymbol{w}\|^2 + b^2$$

Regularization of the bias can, in general, lead to sub-optimal decisions in terms of separating margin. We can mitigate this effect by using a mapping

$$\widehat{\boldsymbol{x}}_i = \begin{bmatrix}\boldsymbol{x}_i\\ K\end{bmatrix}$$

As K becomes larger, the effects of regularizing b become weaker. However, as K becomes larger, the dual problem also becomes harder to solve (i.e. the algorithm may requires many additional iterations).

The dual objective of the modified primal SVM becomes the maximization of

$$\widehat{J}^D(\boldsymbol{\alpha}) = -\frac{1}{2}\boldsymbol{\alpha}^T\widehat{\boldsymbol{H}}\boldsymbol{\alpha} + \boldsymbol{\alpha}^T\mathbf{1}$$
$$\text{s. t. } 0 \le \boldsymbol{\alpha}_i \le C \ , \forall i \in \{1 \dots n\}$$

i.e., the same formulation as before but with a different matrix $\widehat{\boldsymbol{H}}$ and without the linear constraint $\sum_{i=1}^n z_i\boldsymbol{\alpha}_i$.

NOTE: the matrix $\boldsymbol{H}$ is different with respect to the original SVM formulation. For example, for $K = 1$,

$$\widehat{\boldsymbol{H}}_{i,j} = z_i z_j \widehat{\boldsymbol{x}}_i^T\widehat{\boldsymbol{x}}_j = z_i z_j(\boldsymbol{x}_i^T\boldsymbol{x}_j + 1)$$

Write a function that computes the primal SVM solution through the dual SVM formulation $\widehat{J}^D$.

Suggestions:

- You can work directly with vectors $\widehat{\boldsymbol{x}}$. Build the extended matrix of training data that contains all training samples

$$\widehat{\boldsymbol{D}} = \begin{bmatrix}\boldsymbol{x}_1 & \boldsymbol{x}_2 & \dots & \boldsymbol{x}_n\\ 1 & 1 & \dots & 1\end{bmatrix}$$

- Compute $\widehat{\boldsymbol{H}}$. You can use `for` loops. However, you can speed up computations exploiting `numpy.dot` to compute the matrix $\widehat{\boldsymbol{G}}$ all products $\widehat{\boldsymbol{G}}_{ij} = \widehat{\boldsymbol{x}}_i^T\widehat{\boldsymbol{x}}_j$ from $\widehat{\boldsymbol{D}}$ in a single call, and broadcasting to compute $\widehat{\boldsymbol{H}}$ from $\widehat{\boldsymbol{G}}$.

- The `scipy` implementation of L-BFGS-B computes the minimizer of a function, but we want to maximize $\widehat{J}^D(\boldsymbol{\alpha})$. We can cast the problem as **minimization** of

$$\widehat{L}^D(\boldsymbol{\alpha}) = -\widehat{J}^D(\boldsymbol{\alpha}) = \frac{1}{2}\boldsymbol{\alpha}^T\widehat{\boldsymbol{H}}\boldsymbol{\alpha} - \boldsymbol{\alpha}^T\mathbf{1}$$
$$\text{s. t. } 0 \le \boldsymbol{\alpha}_i \le C \ , \forall i \in \{1 \dots n\}$$

- Write the function that, given a numpy array `alpha` containing values for vector $\boldsymbol{\alpha}$, computes $\widehat{L}^D(\boldsymbol{\alpha})$, and its gradient (although the gradient may be automatically computed, consider that $\boldsymbol{\alpha}$ has a number of elements equal to the number of samples, and the number of additional function evaluations would therefore be significantly higher). The gradient of $\widehat{L}^D(\boldsymbol{\alpha})$ is

$$\nabla_{\boldsymbol{\alpha}} \widehat{L}^D = \widehat{\boldsymbol{H}}\boldsymbol{\alpha} - \mathbf{1}$$

  Remember to re-shape the gradient so that it's a 1-D `numpy` array of shape `(n,)`

- Use `scipy.optimize.fmin_l_bfgs_b` to minimize $\widehat{L}^D$. You have to specify the box constraints $0 \leq \boldsymbol{\alpha}_i \leq C$. These can be provided through argument `bounds` of `scipy` function `scipy.optimize.fmin_l_bfgs_b` . `bounds` should be a list of pairs `(min, max)`. Each element of the list corresponds to a different optimization variable. In our case, the list should have N elements, and should be

```
[(0, C), (0, C), ...  (0, C)]
```

- Once you have computed the dual solution, you can recover the primal solution through

$$\widehat{\boldsymbol{w}}^* = \sum_{i=1}^{n} \boldsymbol{\alpha}_i z_i \widehat{\boldsymbol{x}}_i$$

- To classify a pattern, you have to compute the score $\widehat{\boldsymbol{w}}^{*T}\widehat{\boldsymbol{x}}_t$. You can either compute the extended data matrix for the evaluation set, or extract the terms $\boldsymbol{w}^*, b^*$ from $\widehat{\boldsymbol{w}}^*$ and then compute the solution as $\boldsymbol{w}^{*T}\boldsymbol{x}_t + b^*$

- The SVM decision rule is to assign a pattern to class $\mathcal{H}_T$ if the score is greater than 0, and to $\mathcal{H}_F$ otherwise. However, SVM decisions are not probabilistic, and are not able to account for different class priors and mis-classification costs. Bayes decisions thus require either a score post-processing step, i.e. score calibration, or cross-validation to select the optimal threshold for a specific application. Below we simply use threshold 0 and compute the corresponding accuracy.

- Hyper-parameter C should be selected through cross-validation. Below you can find results for different values of C.

- You can check the dual solution is correct by computing the duality gap. At the optimal solution, we have $\widehat{J}(\widehat{\boldsymbol{w}}^*) = \widehat{J}^D(\boldsymbol{\alpha}^*)$. Implement a function that computes the primal objective

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

  and compute the duality gap

$$\widehat{J}(\widehat{\boldsymbol{w}}^*) - \widehat{J}^D(\boldsymbol{\alpha}^*) = \widehat{J}(\widehat{\boldsymbol{w}}^*) + \widehat{L}^D(\boldsymbol{\alpha}^*)$$

  The smaller the duality gap, the more precise is the dual (and thus the primal) solution.

- 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 `factr=1.0` was used.**

- The `scipy` implementation of L-BFGS-B 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`

The following table reports primal and dual objectives for the solution returned by L-BFGS with different values of C and K on IRIS, with `factr=1.0`. The accuracy on the evaluation set is also reported for cross-checking. The training and evaluation data are obtained using the same splits of previous laboratories . Higher values of K correspond to weaker regularization of the SVM bias term.

K	C	Primal loss	Dual loss	Duality gap	Error rate
1	0.1	3.774974e+00	3.774974e+00	6.641808e-07	2.9%
1	1.0	1.577994e+01	1.577993e+01	6.454718e-06	5.9%
1	10.0	7.896893e+01	7.896893e+01	1.888172e-06	5.9%
10	0.1	2.983576e+00	2.983576e+00	1.812922e-08	11.8%
10	1.0	1.131707e+01	1.131707e+01	1.939987e-06	5.9%
10	10.0	5.464492e+01	5.464483e+01	8.859310e-05	5.9%

Kernel SVM

SVMs allow for non-linear classification through an implicit expansion of the features in a higher-dimensional space. The SVM dual objective depends on the training samples only through dot-products, and we can compute a classification score through scalar products between training and evaluation samples. Therefore, SVM does not require that we explicitly compute the feature expansion: it's sufficient that we are able to compute the scalar product between the expanded features $k(\boldsymbol{x}_1, \boldsymbol{x}_2) = \boldsymbol{\phi}(\boldsymbol{x}_1)^T \boldsymbol{\phi}(\boldsymbol{x}_2)$. Function k is called *kernel function*.

Implement the dual SVM (without bias) classifier for generic kernel functions. You can re-use the previously developed code, however you should replace the computation of $\widehat{\boldsymbol{H}}$ with

$$\widehat{\boldsymbol{H}}_{i,j} = z_i z_j k(\boldsymbol{x}_i, \boldsymbol{x}_j)$$

In contrast with linear SVM, we are not able to compute the primal solution and its cost. However, we can compute the score of a test sample as

$$s(\boldsymbol{x}_t) = \sum_{i=1}^{n} \boldsymbol{\alpha}_i^* z_i k(\boldsymbol{x}_i, \boldsymbol{x}_t)$$

where the summation is taken over the *training* samples (in practice we can consider just the samples for which $\boldsymbol{\alpha}_i > 0$, i.e. the support vectors).

You can try different kernels:

- Polynomial kernel of degree d: $k(\boldsymbol{x}_1, \boldsymbol{x}_2) = (\boldsymbol{x}_1^T \boldsymbol{x}_2 + c)^d$
- Radial Basis Function kernel: $k(\boldsymbol{x}_1, \boldsymbol{x}_2) = e^{-\gamma \|\boldsymbol{x}_1 - \boldsymbol{x}_2\|^2}$

The choice of the kernel and of its hyper-parameters (e.g. c and γ) can also be made through cross-validation.

NOTE: To add a (regularized) bias in the non-linear SVM version it's not sufficient to simply extend the feature vectors as we previously did, but we should rather add a constant value to our kernel function:

$$\widehat{k}(\boldsymbol{x}_1, \boldsymbol{x}_2) = k(\boldsymbol{x}_1, \boldsymbol{x}_2) + \xi$$

For the linear SVM, we were doing this implicitly by expanding the feature vectors $\boldsymbol{x}_i$, which corresponded to $\xi = K^2$ ($\xi = 1$ in our examples). For the dual problem, we work with the original features, and we add the constant term directly to the kernel evaluations.

Below you can find the dual objective value and the accuracy for different configurations.

$K = \sqrt{\xi}$	C	Kernel	Dual loss	Error rate
0.0	1.0	Poly ($d = 2, c = 0$)	6.663628e+00	8.8%
1.0	1.0	Poly ($d = 2, c = 0$)	6.296912e+00	8.8%
0.0	1.0	Poly ($d = 2, c = 1$)	3.592918e+00	2.9%
1.0	1.0	Poly ($d = 2, c = 1$)	3.569987e+00	2.9%
0.0	1.0	RBF ($\gamma = 1.0$)	1.198930e+01	8.8%
0.0	1.0	RBF ($\gamma = 10.0$)	1.842757e+01	11.8%
1.0	1.0	RBF ($\gamma = 1.0$)	1.197813e+01	8.8%
1.0	1.0	RBF ($\gamma = 10.0$)	1.839182e+01	8.8%