Laboratory 11

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

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

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 $a \leq \alpha_i \leq b$) 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 the algorithm may require more iterations to converge to a solution.

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

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

i.e., the same formulation as before but without the equality constraint and with matrix $\widehat{\boldsymbol{H}}$ computed from the extended features $\widehat{\boldsymbol{x}}_i$ rather than from the original features $\boldsymbol{x}_i$:

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

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. For K = 1, you would compute

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

- 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

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

- 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

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

  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 $\boldsymbol{w}^{*T}\boldsymbol{x}_t + b^*$ (notice that, if you choose the latter option and set $K \neq 1$, then you need to appropriately re-scale the value of b^* to ensure that $\widehat{\boldsymbol{w}}^{*T}\widehat{\boldsymbol{x}}_t = \boldsymbol{w}^{*T}\boldsymbol{x}_t + b^*$)

- The SVM decision rule assigns 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.

- A possible stopping criterion for the optimizer consists in checking whether two successive iterations failed to provide a relevant decrease of the objective function. You can control the precision of this criterion 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. Depending on the architecture and the underlying numerical libraries, even setting a very low value for `factr` may not be sufficient to obtain low duality gap. In this case, we can try to improve the quality of our solution by disabling the termination condition based on the decrease of the objective function and rely only on the norm of the projected gradient becoming close to zero. For this, you can set `factr=numpy.nan`, and optionally set a value for parameter `pgtol` (default is `pgtol=1e-5`)

- 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=numpy.nan` and `pgtol=1e-5`. The accuracy on the evaluation set is also reported for cross-checking (the predictions are obtained comparing the SVM score with a threshold $t = 0$). 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.

NOTE: Due to numerical precision issues you may obtain slightly different values for the dual and primal loss, and, depending on these values, significantly different values for the duality gap. In any case, if the duality gap is small (few orders of magnitude lower than the objective itself), the found solution is close to optimal. In some cases, you may also observe very small, negative duality gaps, which is not possible according to theory. This is typically due to numerical errors in computing the primal and dual solutions, and the duality gap should be close to the machine precision.

The table also reports minDCF and actual DCF. Since the SVM scores have no probabilistic interpretation, they may typically show significant miscalibration, especially when the application prior is far from the empirical training set prior. We can still compute minimum DCFs, and compute actual costs assuming that scores were LLRs, however we should keep in mind that using Bayes decisions for SVM scores may lead to very poor decisions and poor actual costs - on the other hand, we may also be lucky and obtain good decisions without any form of post-processing of the scores - a validation set can help us in judging if score calibration is indeed required or not.

K	C	Primal loss	Dual loss	Error rate	minDCF ($\pi_T = 0.5$)	actDCF ($\pi_T = 0.5$)
1	0.1	3.774974e+00	3.774974e+00	2.9%	0.0556	0.0625
1	1.0	1.577993e+01	1.577993e+01	5.9%	0.0556	0.1181
1	10.0	7.896896e+01	7.896893e+01	5.9%	0.0556	0.1181
10	0.1	2.983577e+00	2.983576e+00	11.8%	0.0000	0.2222[†]
10	1.0	1.131707e+01	1.131707e+01	5.9%	0.0556	0.1181
10	10.0	5.464486e+01	5.464483e+01	5.9%	0.0625	0.1181

[†]This is potentially the best model (with an optimal threshold we would get a minDCF of 0, thus we could also achieve an error rate of 0%), but, due to poor threshold selection, we actually achieve a DCF of 0.2222, and an error rate of 11.8%, i.e. the worst results in the table

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 directly. However, we can exploit the primal-dual solution relationship to express the primal objective in terms of $\boldsymbol{\alpha}$:

$$\begin{aligned} \widehat{J}(\widehat{\boldsymbol{w}}) &= \frac{1}{2} \|\widehat{\boldsymbol{w}}\|^2 + C \sum_{i=1}^{n} \max(0, 1 - z_i(\widehat{\boldsymbol{w}}^T \widehat{\boldsymbol{x}}_i)) \\ &= \frac{1}{2} \boldsymbol{\alpha}^T \boldsymbol{H} \boldsymbol{\alpha}^T + C \sum_{i=1}^{n} \max(0, 1 - \boldsymbol{H}_{i\cdot} \boldsymbol{\alpha}) \end{aligned}$$

where $\boldsymbol{H}_{i\cdot}$ is the i-th row of $\boldsymbol{H}$. Note that the terms $\boldsymbol{H}_{i\cdot}\boldsymbol{\alpha}$ correspond to the components of the vector $\boldsymbol{H}\boldsymbol{\alpha}$, and can thus be computed from the latter.

The score of a test sample can be computed 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 $\xi = K^2$. 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 primal and dual objective value and the accuracy for different configurations.

$K = \sqrt{\xi}$	C	Kernel	Primal loss	Dual loss	Error rate	minDCF ($\pi_T = 0.5$)	actDCF ($\pi_T = 0.5$)
0.0	1.0	Poly ($d = 2, c = 0$)	6.663666e+00	6.663628e+00	8.8%	0.0625	0.1736
1.0	1.0	Poly ($d = 2, c = 0$)	6.296913e+00	6.296912e+00	8.8%	0.0625	0.1736
0.0	1.0	Poly ($d = 2, c = 1$)	3.592924e+00	3.592918e+00	2.9%	0.0556	0.0556
1.0	1.0	Poly ($d = 2, c = 1$)	3.569991e+00	3.569987e+00	2.9%	0.0556	0.0556
0.0	1.0	RBF ($\gamma = 1.0$)	1.198932e+01	1.198930e+01	8.8%	0.0625	0.1736
1.0	1.0	RBF ($\gamma = 1.0$)	1.197814e+01	1.197813e+01	8.8%	0.0625	0.1736
0.0	1.0	RBF ($\gamma = 10.0$)	1.842762e+01	1.842757e+01	11.8%	0.1736	0.2292
1.0	1.0	RBF ($\gamma = 10.0$)	1.839186e+01	1.839182e+01	8.8%	0.1736	0.1736

Project

NOTE: training SVM models may require some time. To speed-up the process, we suggest that you first setup the experiments for this project using only a fraction of the model training data. Once the code is ready, you can then re-run all the experiments with the full dataset.

Apply the SVM to the project data. Start with the linear model (to avoid excessive training time we consider only the models trained with K = 1.0). Train the model with different values of C. As for logistic regression, you should employ a logarithmic scale for the values of C. Reasonable values are given by `numpy.logspace(-5, 0, 11)`. Plot the minDCF and actDCF ($\pi_T = 0.1$) as a function of C (again, use a logarithmic scale for the x-axis). What do you observe? Does the regularization coefficient significantly affect the results for one or both metrics (remember that, for SVM, low values of C imply strong regularization, while large values of C imply weak regularization)? Are the scores well calibrated for the target application? What can we conclude on linear SVM? How does it perform compared to other linear models? Repeat the analysis with centered data. Are the result significantly different?

We now consider the polynomial kernel. For simplicity, we consider only the kernel with $d = 2, c = 1$ (but better results may be possible with different configurations), and we set $\xi = 0$, since the kernel already implicitly accounts for the bias term (due to $c = 1$). We also consider only the original, non-centered features (again, different pre-processing strategies may lead to better results). Train the model with different values of C, and compare the results in terms of minDCF and actDCF. What do you observe with quadratic models? In light of the characteristics of the dataset and of the classifier, are the results consistent with previous models (logistic regression and MVG models) in terms of minDCF? What about actDCF?

For RBF kernel we need to optimize both γ and C (since the RBF kernel does not implicitly account for the bias term we set $\xi = 1$). We adopt a grid search approach, i.e., we consider different values of γ and different values of C, and try all possible combinations. For γ we suggest you analyze values $\gamma \in \left[e^{-4}, e^{-3}, e^{-2}, e^{-1}\right]$, while for C, to avoid excessive time but obtain a good coverage of possible good values we suggest log-spaced values `numpy.logspace(-3, 2, 11)` (of course, you are free to experiment with other values if you so wish). Train all models obtained by combining the values of γ and of C. Plot minDCF and actDCF as a function of C, with a different line for each value of γ (i.e., four lines for minDCF and four lines for actDCF). Analyze the results. Are there values of γ and C that provide better results? Are the scores well calibrated? How the result compare to previous models? Are there characteristics of the dataset that can be better captured by RBF kernels?

Optional

Consider again the polynomial kernel, but with $d = 4, c = 1, \xi = 0$. Train the model with different values of C (use again `numpy.logspace(-5, 0, 11)`, and compare the results in terms of minDCF and actDCF. What do you observe with quadratic models? Can you explain the better results in terms of the characteristics of the dataset? To answer, consider only the last two features of each sample (look at the scatter plots) $\boldsymbol{y}_i = \boldsymbol{x}_{i,[4:6]}$. Consider how these features would be transformed by a simple degree 2 kernel that maps each sample to $\boldsymbol{y}_i \rightarrow z_i = \boldsymbol{y}_{i,[0:1]}\boldsymbol{y}_{i,[1:2]}$ (suggestion: draw few samples on paper, one per cluster). Then consider which kind of separation surfaces would be required to separate the z_i 1-D feature vectors — remember that in 1-D linear rules correspond to a sample being on either side of a threshold, quadratic rules correspond to inequalities that involve the sample being inside a (possibly empty) interval, and so on.