UNIVERSITY OF EASTERN FINLAND

Please carefully read and follow the general instructions regarding coding assignments. Failing to meet the requirements might lead to penalties. https://elearn.uef.fi/mod/page/view.php?id=248672

If you suspect that something is wrong with some task instructions, please contact the lecturer.

If you face persistent issues while working on a task, do ask for help, e.g. during a course meeting or by contacting the lecturer via email.

Datasets

- *iris* from https://archive.ics.uci.edu/dataset/53/iris, `iris-SV-sepal.csv` and `iris-VV-length.csv` contain variants of the dataset from the UCI data repository, as used in the lecture.
- *credit* from https://archive.ics.uci.edu/dataset/144/statlog+german+credit+data, `creditDE.csv` contains a variant of the dataset from the UCI data repository.

Tools

- `classification_resources.py` some code snippets, such as functions for implementing support vector machines and evaluation measures.

! Imports of external libraries other than those that appear in the `classification_resources.py` file are not allowed.

Task 1. Implement the linear SVM algorithm with hard-margin and soft-margin variants (you may use snippets from `classification_resources.py`, i.e. fill the dots in the provided code). Apply them respectively to the *irisSV* and *irisVV* datasets.

That is, divide the *irisSV* dataset into training and test subsets in proportions $4/5$–$1/5$ at random, i.e. assign one fifth of instances, chosen at random, to the test dataset and the rest to the training dataset. Train a hard-margin SVM on the training subset, and apply the resulting model to the test subset.

Write down the confusion matrix and compute the accuracy, recall and precision.

Give the equation of the separating hyperplane. Plot the separating hyperplane and highlight the support vectors.

Do the same with soft-margin SVM on the *irisVV* dataset (setting $c = 2$, for example).

Task 2. Run an evaluation of the soft-margin SVM on the *irisVV* dataset with cross-validation.

That is, run 10 rounds of cross-validation with 5 folds on the *irisVV* dataset. Report the mean and variance of the classifier's accuracy across the successive rounds.

Task 3. Implement the AdaBoost algorithm with instance weighting done via sampling. Apply it to the *credit* dataset with linear SVM. Try using less aggressive weight updates, calculating the update factor as

$$\alpha_t \leftarrow \beta \cdot \ln((1 - \epsilon_t)/\epsilon_t)/2$$

where $0 < \beta < 1$. What happens if $\beta = 0$?

Task 4. Empirically compare the impact of boosting and bagging when combined to a linear SVM vs. to a SVM with a RBF kernel on the *credit* dataset.

Note on implementing a SVM using CVXOPT

Hard-margin SVM. Recall from the lecture that the dual problem for a hard-margin SVM, which we need to solve to obtain the vector of Lagrange multipliers $\boldsymbol{a}$, is as follows

$$\max L_D = \sum_{j=1}^{j=n} a_j - \frac{1}{2}\sum_{j=1}^{j=n}\sum_{i=1}^{i=n} a_j a_i y_j y_i \boldsymbol{x}^{(j)} \cdot \boldsymbol{x}^{(i)}$$
$$\text{s.t. } 0 \leq a_j \text{ and } \sum_{j=1}^{j=n} a_j y_j = 0 \; \forall j$$

where n is the number of training instances.

Letting $\boldsymbol{H}$ be a symmetric matrix such that $H_{i,j} = y_j y_i \boldsymbol{x}^{(j)} \cdot \boldsymbol{x}^{(i)}$ and denoting as $\mathbf{1}$ a vector size n full of ones, we can write the optimization problem as

$$\max_{\boldsymbol{a}} \mathbf{1}^T a_j - \frac{1}{2}\boldsymbol{a}^T \boldsymbol{H}\boldsymbol{a} \tag{4.1}$$
$$\text{s.t.} \quad a_j \geq 0 \quad \forall j \tag{4.2}$$
$$\boldsymbol{y}^T \boldsymbol{a} = 0 \tag{4.3}$$

CVXOPT[1] is a Python library for convex optimization. In particular, the qp function[2] allows to solve quadratic programs. Specifically

```
cvxopt.solvers.qp(P, q, G, h, A, b)
```

solves the quadratic program

$$\min_{\boldsymbol{x}} \frac{1}{2}\boldsymbol{x}^T \boldsymbol{P}\boldsymbol{x} + \boldsymbol{q}^T \boldsymbol{x}$$
$$\text{s.t.} \quad \boldsymbol{G}\boldsymbol{x} \leq h$$
$$\boldsymbol{A}\boldsymbol{x} = b$$

So, we need to bring our optimization problem into this form, which can be done by multiplying both the objective (4.2) and the inequality (4.2) by minus one, turning the maximization into minimization in the former, and reversing the direction of the inequality in the latter.
Thus we get

$$\min_{\boldsymbol{a}} \frac{1}{2}\boldsymbol{a}^T \boldsymbol{H}\boldsymbol{a} - \mathbf{1}^T a_j$$
$$\text{s.t.} \quad -a_j \leq 0 \quad \forall j$$
$$\boldsymbol{y}^T \boldsymbol{a} = 0$$

which is in the form expected by the solver, identifying the variables as follows (qp parameters on the left, SVM optimization problem variables on the right)

$\boldsymbol{P} = \boldsymbol{H}$	a matrix of size $n \times n$
$\boldsymbol{q} = -\mathbf{1}$	a vector of size n
$\boldsymbol{G} = -\text{eye}(n)$	a diagonal matrix of size $n \times n$
$\boldsymbol{h} = \mathbf{0}$	a vector of size n full of zeros
$\boldsymbol{A} = \boldsymbol{y}$	a vector of size n
$b = 0$	a scalar
$\boldsymbol{x} = \boldsymbol{a}$	a vector of size n, is the solution of the problem that we are looking for.

So, we can prepare the parameters for the solver and obtain the solution as follows

[1] https://cvxopt.org
[2] https://cvxopt.org/userguide/coneprog.html#quadratic-programming

```
    P = cvxopt.matrix(numpy.outer(y, y) * numpy.dot(X, X.T))
    q = cvxopt.matrix(-1 * numpy.ones(n_samples))
    G = cvxopt.matrix(-1 * numpy.eye(n_samples))
    h = cvxopt.matrix(numpy.zeros(n_samples))
    A = cvxopt.matrix(numpy.array([y]), (1, n_samples))
    b = cvxopt.matrix(0.0)

    solution = cvxopt.solvers.qp(P, q, G, h, A, b)
    return numpy.ravel(solution['x'])
```

Soft-margin SVM. Recall from the lecture that the dual problem for a soft-margin SVM, which we need to solve to obtain the vector of Lagrange multipliers $\boldsymbol{a}$, is as follows

$$\max L_D = \sum_{j=1}^{j=n} a_j - \frac{1}{2} \sum_{j=1}^{j=n} \sum_{i=1}^{i=n} a_j a_i y_j y_i \boldsymbol{x}^{(j)} \cdot \boldsymbol{x}^{(i)}$$
$$\text{s.t. } 0 \leq a_j \leq C \text{ and } \sum_{j=1}^{j=n} a_j y_j = 0 \; \forall j$$

where n is the number of training instances.

So the optimization problem is very similar to the hard-margin case, adding the inequality constraint upper-bounding $\boldsymbol{a}$ by parameter C.
It can rewritten as follows

$$\begin{aligned} \min_{\boldsymbol{a}} \; & \frac{1}{2} \boldsymbol{a}^T \boldsymbol{H} \boldsymbol{a} - \mathbf{1}^T a_j \\ \text{s.t.} \quad & -a_j \leq 0 \quad \forall j \\ & a_j \leq C \quad \forall j \\ & \boldsymbol{y}^T \boldsymbol{a} = 0 \end{aligned}$$

These inequality constraints can be added by concatenating the correspond rows to qp parametersthe solver parameters `G` and `h`

$$\begin{aligned} \boldsymbol{G} &= [-\text{eye}(n), \text{eye}(n)]^T && \text{a matrix of size } 2n \times n \\ \boldsymbol{h} &= [\mathbf{0}, \boldsymbol{C}]^T && \text{a vector of size } 2n \end{aligned}$$

and the corresponding code, where the upper-bound parameter is denoted as `c`

```
    P = cvxopt.matrix(numpy.outer(y, y) * numpy.dot(X, X.T))
    q = cvxopt.matrix(-1 * numpy.ones(n_samples))
    G = cvxopt.matrix(numpy.vstack((-1 * numpy.eye(n_samples),
                                    numpy.eye(n_samples))))
    h = cvxopt.matrix(numpy.hstack((numpy.zeros(n_samples),
                                    numpy.ones(n_samples) * c)))
    A = cvxopt.matrix(numpy.array([y]), (1, n_samples))
    b = cvxopt.matrix(0.0)

    solution = cvxopt.solvers.qp(P, q, G, h, A, b)
    return numpy.ravel(solution['x'])
```

Note on computing the bias of a soft-margin SVM

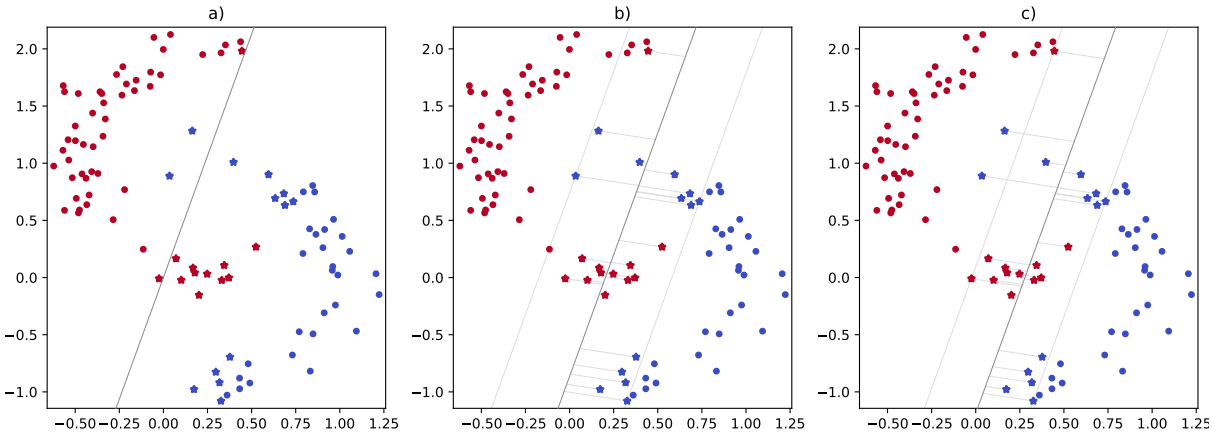

Figure 1: Training instances and hyperplanes for a soft-margin SVM example

The graphs in Figure 1 show training instances from a two-dimensional dataset with two classes, with support vectors marked as stars. Let S denote the set of support vectors (training points associated to a non-zero Lagrange multiplier), with $S_{\star}$ and $S_{\star}$ the support vectors from the blue and red classes, respectively.

Considering a hyperplane defined by weights w and bias b. Assume the red class is the positive class, and the prediction is made such that points are predicted as positive if $w \cdot x + b > 0$, negative otherwise. Let $\theta_x = w \cdot x + b$ denote the (signed) distance from data point x to the hyperplane defined by weights w and bias b.

Figure 1.a) shows the situation before adjusting the bias, when it is set to $b = 0$, by default. Values θ_x are computed with this default value of the bias and used to adjust it to get the correct equation for the separating hyperplane.

Figure 1.b) shows the situation where the bias is set as

$$b = -\big(\max_{x \in S} \theta_x + \min_{x \in S} \theta_x\big)/2$$

Figure 1.c) shows the situation where the bias is set as

$$b = -\big(\max_{x \in S_{\star}} \theta_x + \min_{x \in S_{\star}} \theta_x\big)/2$$

Note how in Figure 1.b) the hyperplane is placed in the middle between the support vectors furthest away, i.e. having highest and lowest θ values respectively. However, the support vector furthest away on the left side is actually a blue instance. It is not on the margin, it actually falls outside the margin, on the wrong side of the decision boundary. It should not be used to determine the bias. Here, it results in the hyperplane being incorrectly shifted to the left.

In Figure 1.c) instead the red support vector furthest on the left side and the blue support vector furthest on the right side, i.e. furthest on their respective correct sides, are considered, allowing to correctly set the bias.