Homework 7

CS 559 - Neural Networks - Fall 2019

Matteo Corain 650088272

November 20, 2019

1 Question 1

1.1 Training set generation

The training set for the SVM to be designed has been generated by randomly selecting $n = 100$ points in $[0, 1]^2$ through the `uniform()` function offered by the NumPy package. The class labels associated to such points has been selected as:

$$d_i = \begin{cases} +1 & \text{if } x_{i2} < 0.2\sin(10x_{i1}) + 0.3 \vee (x_{i2} - 0.8)^2 + (x_{i1} - 0.5)^2 < 0.15^2 \\ -1 & \text{otherwise} \end{cases}$$

The obtained set of points is shown in figure 1, which is produced by the `plot_data()` function. This procedure simply plots the data points and the "exact" separator, after having evaluated it on a set of equidistant values in $[0, 1]$.

1.2 SVM problem formalization

Let us recall that the SVM problem is to define the values of $\alpha_1, \ldots, \alpha_n$ that satisfy the following optimization problem:

$$\max_{\substack{\alpha_1,\ldots,\alpha_n \geq 0 \\ \sum_{i=1}^n \alpha_i d_i = 0}} \left(\sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j d_i d_j K(x_i, x_j) \right)$$

Where $K(x, y)$ denotes the chosen kernel function. This has the form of a quadratic optimization problem, which is in general be expressed as:

$$\min_{\substack{Gx \leq h \\ Ax = b}} \left(\frac{1}{2} x^T P x + q^T x \right)$$

Where P, q, G, h, A, b are opportunely defined matrices and x denotes the solution vector. In order to solve our original problem, therefore, it is necessary to express it using the presented notation, which can be used by numerical calculus packages to find an appropriate solution. First, let us rewrite our original maximization problem as a minimization problem; since maximizing a

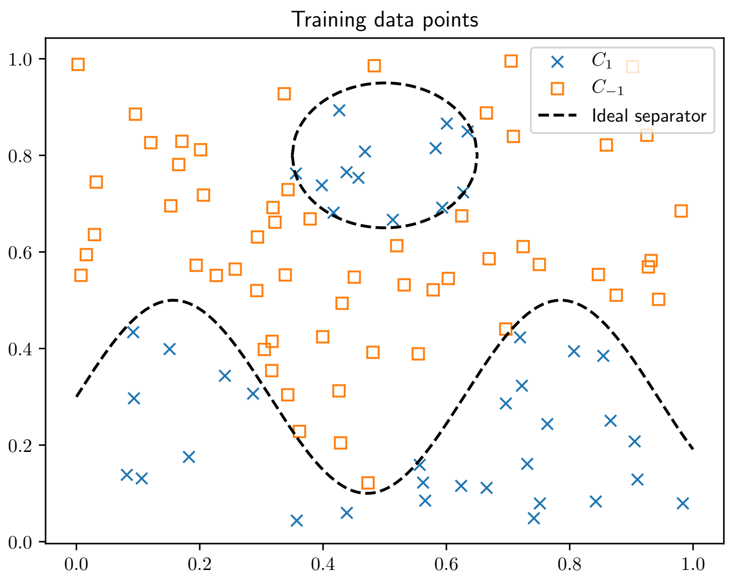

Figure 1: Data set for the exercise

function is equivalent to minimizing its opposite, this is simply obtained by changing the sign of the quantity inside the max operator:

$$\min_{\substack{\alpha_1,\ldots,\alpha_n \geq 0 \\ \sum_{i=1}^{n} \alpha_i d_i = 0}} \left(-\sum_{i=1}^{n} \alpha_i + \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j d_i d_j K(x_i, x_j) \right)$$

In this problem, the solution vector is represented by the vector of the quantities α_i we aim to find values for:

$$x = \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_n \end{bmatrix} \in \mathbb{R}^{n,1}$$

The first term of the expression to be minimized can be used to define matrix q; in fact, this has to satisfy the following equation:

$$q^T x = q^T \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_n \end{bmatrix} = \sum_{i=1}^{n} q_i \alpha_i = -\sum_{i=1}^{n} \alpha_i \Rightarrow q = \begin{bmatrix} -1 \\ \vdots \\ -1 \end{bmatrix} \in \mathbb{R}^{n,1}$$

The second term of the expression to be minimized can be used instead to define matrix P; in fact, this has to satisfy the following equation:

$$\frac{1}{2}x^TPx = \frac{1}{2}\begin{bmatrix}\alpha_1 & \cdots & \alpha_n\end{bmatrix} P \begin{bmatrix}\alpha_1 \\ \vdots \\ \alpha_n\end{bmatrix} = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j p_{ij} = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j d_i d_j K(x_i, x_j)$$

$$\Rightarrow P = \begin{bmatrix} d_1^2K(x_1,x_1) & \cdots & d_1d_nK(x_1,x_n) \\ \vdots & \ddots & \vdots \\ d_nd_1K(x_n,x_1) & \cdots & d_n^2K(x_n,x_n)\end{bmatrix} \in \mathbb{R}^{n,n}$$

As for the first optimization constraint ($Gx \leq h$), this can be used to express the original requirement according to which $\alpha_1, \ldots, \alpha_n \geq 0$ (or, equivalently, $-\alpha_1, \ldots, -\alpha_n \leq 0$). Specifically, a choice of matrices that can be used for expressing this restriction is:

$$G = \begin{bmatrix} -1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & -1\end{bmatrix} \in \mathbb{R}^{n,n}, h = \begin{bmatrix}0 \\ \vdots \\ 0\end{bmatrix} \in \mathbb{R}^{n,1}$$

In fact:

$$Gx \leq h \Rightarrow \begin{bmatrix} -1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & -1\end{bmatrix}\begin{bmatrix}\alpha_1 \\ \vdots \\ \alpha_n\end{bmatrix} \leq \begin{bmatrix}0 \\ \vdots \\ 0\end{bmatrix} \Rightarrow \begin{cases} -\alpha_1 \leq 0 \\ \vdots \\ -\alpha_n \leq 0\end{cases}$$

Finally, as for the second optimization constraint ($Ax = b$), this can be used to express the original requirement according to which $\sum_{i=1}^{n}\alpha_i d_i = 0$. Specifically, a choice of matrices that can be used for expressing this restriction is:

$$A = \begin{bmatrix}d_1 & \cdots & d_n\end{bmatrix} \in \mathbb{R}^{1,n}$$
$$b = [0] \in \mathbb{R}$$

In fact:

$$Ax = b \Rightarrow \begin{bmatrix}d_1 & \cdots & d_n\end{bmatrix}\begin{bmatrix}\alpha_1 \\ \vdots \\ \alpha_n\end{bmatrix} = 0 \Rightarrow \sum_{i=1}^{n}\alpha_i d_i = 0$$

1.3 Implementation of SVM training procedure

For the solution of the aforementioned optimization problem, it was chosen to use the `qpsolvers` library for the Python programming language, which internally uses the `quadprog` module as backend but offers an API that is similar to Matlab's quadratic programming functions. The computation of the solution to the SVM problem is carried out in the `solve_svm()` function, which performs the following actions:

- It defines the matrices P, q, G, h, A, b using the x_i and d_i training data and the kernel function passed as parameters; for the stability of the algorithm, that requires P to be positive definite although this condition may be violated due to computations in finite machine arithmetic, the elements on the main diagonal of matrix P are slightly incremented by adding them a small positive quantity (10^{-9}), so that, in the end, the possible small, negative eigenvalues of the original matrix P are mapped to small, positive numbers (which is a sufficient condition for matrix P to be positive definite).

- It runs the solver by invoking the `solve_qp()` method offered by the `qpsolvers` library.

- It sets all elements which are smaller than the `tol` parameter to zero: due to the approximations in finite machine arithmetic, in fact, the numerical optimization procedure does not return a perfectly zero value for the α_i coefficients that are not correspondent to a support vector, but instead returns some small (even negative in some cases) values; since those values do not carry any information as they are only byproducts of the optimization procedure, they can safely be mapped to zero.

- Based on the indices of non-zero α_i values, it constructs the lists of the support vectors x_{SV} and of their associated output d_{SV} and coefficients α_{SV}.

- It defines the "optimal" separator according to the formula:

$$g(x) = \sum_{i=1}^{n_{SV}} \alpha_{SV,i} d_{SV,i} K(x_i, x) + \theta$$

$$\theta = d_{SV,1} - \sum_{i=1}^{n_{SV}} \alpha_{SV,i} d_{SV,i} K(x_i, x_{SV,1})$$

  Where $x_{SV,i}$, $d_{SV,i}$ and $\alpha_{SV,i}$ denote the i-th support vector and its associated output and coefficient. The choice of the support vector used to define the bias θ is arbitrary; in this case, it was chosen to always use the first one.

- Finally, it returns the optimal separator (as a lambda function) and the list of the support vectors and their associated outputs.

1.4 Separator evaluation and plotting

The obtained separator, together with the training set and the list of support vectors, is then passed to the `plot_sep()` function. This plots the training set and the support vectors (using a round, bigger marker, to encircle the corresponding data point) in a way similar to the `plot_data()` function. Then, it proceeds to evaluate the separator on a `eval_pts*eval_pts` grid of points in $[0, 1]^2$; the results of the evaluation are stored in the `evals` matrix and plotted using the `contours()` function offered by the Matplotlib library, after setting the levels of interest to $g(x) = -1$, $g(x) = 0$ and $g(x) = 1$ (corresponding respectively to $\mathcal{H}^-$, $\mathcal{H}$ and $\mathcal{H}^+$).

1.5 Obtained results

The presented methodology was applied to train and compare different support vector machines using different kernels; in all cases, the tolerance parameter of the `solve_svm()` function was set to 10^{-6}. Three kernels have been tested:

- A polynomial kernel with $d = 2$: $K(x, y) = (1 + x^T y)^2$;
- A polynomial kernel with $d = 10$: $K(x, y) = (1 + x^T y)^{10}$;
- A Gaussian kernel with $c = 1$: $K(x, y) = e^{-||x-y||^2}$.

The polynomial kernel with degree 2 turned out however to be too simple to solve this problem, as the resulting SVM was not able to correctly classify all the training patterns. Instead, both the SVMs built using the 10-degree polynomial and the Gaussian kernels were appropriate for the task; the obtained separation in the two cases is shown in figures 2 and 3.

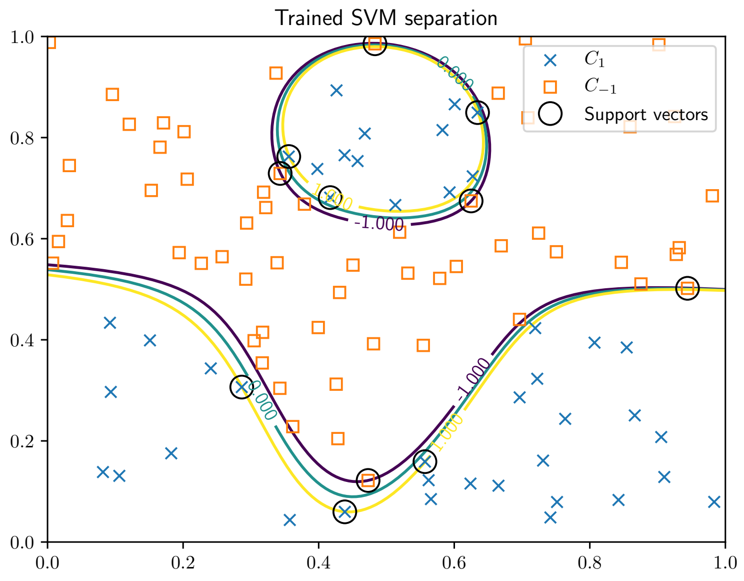

Figure 2: Separation obtained with the polynomial kernel SVM

The solution using the polynomial kernel is based on the following support vectors:

- For C_1:

$$\begin{bmatrix}0.43857224\\0.0596779\end{bmatrix}, \begin{bmatrix}0.63440096\\0.84943179\end{bmatrix}, \begin{bmatrix}0.48303426\\0.98555979\end{bmatrix}, \begin{bmatrix}0.55678519\\0.15895964\end{bmatrix}, \begin{bmatrix}0.35591487\\0.76254781\end{bmatrix}, \begin{bmatrix}0.28653662\\0.30646975\end{bmatrix}$$

- For C_{-1}:

$$\begin{bmatrix}0.34317802\\0.72904971\end{bmatrix}, \begin{bmatrix}0.94416002\\0.50183668\end{bmatrix}, \begin{bmatrix}0.48303426\\0.98555979\end{bmatrix}, \begin{bmatrix}0.6249035\\0.67468905\end{bmatrix}, \begin{bmatrix}0.472913\\0.12175436\end{bmatrix}$$

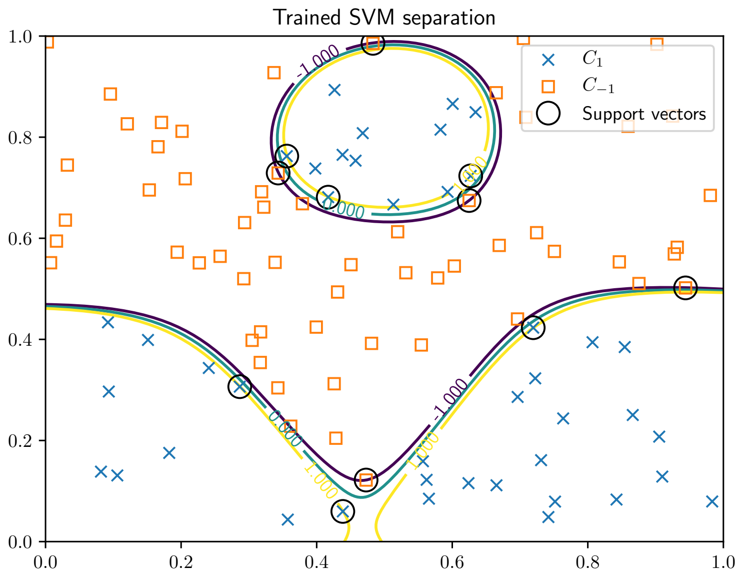

Figure 3: Separation obtained with the Gaussian kernel SVM

As for the solution based on the Gaussian kernel, the following support vectors were used:

- For C_1:

$$\begin{bmatrix}0.71946897\\0.42310646\end{bmatrix}, \begin{bmatrix}0.43857224\\0.0596779\end{bmatrix}, \begin{bmatrix}0.41702221\\0.68130077\end{bmatrix}, \begin{bmatrix}0.62724897\\0.72341636\end{bmatrix}, \begin{bmatrix}0.35591487\\0.76254781\end{bmatrix}, \begin{bmatrix}0.28653662\\0.30646975\end{bmatrix}$$

- For C_{-1}:

$$\begin{bmatrix}0.34317802\\0.72904971\end{bmatrix}, \begin{bmatrix}0.94416002\\0.50183668\end{bmatrix}, \begin{bmatrix}0.48303426\\0.98555979\end{bmatrix}, \begin{bmatrix}0.6249035\\0.67468905\end{bmatrix}, \begin{bmatrix}0.472913\\0.12175436\end{bmatrix}$$

1.6 Complete Python code

```
import numpy as np
import matplotlib.pyplot as plt
import qpsolvers as qp

# Use TeX for text rendering
plt.rc('text', usetex=True)

# Set the random seed for reproducibility
np.random.seed(123)

def kern_poly(d):
    # Generate a polynomial kernel of degree d
    return lambda x, y : (1 + x.transpose() @ y) ** d
```

```
def kern_gaus(c):
    # Generate a Gaussian kernel with constant c
    return lambda x, y : np.exp(-c * np.linalg.norm(x - y) ** 2)

def solve_svm(n, x, d, kernel, tol):
    # Define matrices for quadprog
    P = np.array([d[i] * d[j] * kernel(x[i], x[j]) for i in range(n) for j in range(n)]).reshape(n, n) + 1e-9 *
         np.eye(n)
    q = -np.ones(n)
    G = -np.eye(n)
    h = np.zeros(n)
    A = np.array(d)
    b = np.zeros(1)

    # Run the solver
    a = qp.solve_qp(P, q, G, h, A, b)

    # Remove all small alphas
    a[a < tol] = 0

    # Find the support vectors
    isv = np.nonzero(a)[0]
    asv = [a[i] for i in isv]
    xsv = [x[i] for i in isv]
    dsv = [d[i] for i in isv]

    # Compute the separator
    theta = dsv[0] - sum([asv[i] * dsv[i] * kernel(xsv[i], xsv[0]) for i in range(len(isv))])
    sep = lambda z : sum([asv[i] * dsv[i] * kernel(xsv[i], z) for i in range(len(isv))]) + theta

    # Return separator and support vectors
    return sep, xsv, dsv

def plot_data(n, x, d):
    # Create a new figure
    plt.figure()

    # Plot the training set
    plt.plot([x[i][0] for i in range(n) if d[i] == 1], [x[i][1] for i in range(n) if d[i] == 1], linestyle="
        none", marker="x", fillstyle="none")
    plt.plot([x[i][0] for i in range(n) if d[i] == -1], [x[i][1] for i in range(n) if d[i] == -1], linestyle="
        none", marker="s", fillstyle="none")

    # Plot ideal separator
    plt.plot(np.linspace(0, 1, 1000), [0.2 * np.sin(10 * x) + 0.3 for x in np.linspace(0, 1, 1000)], linestyle=
        "--", color="k")
    plt.plot(np.linspace(0.35, 0.65, 100), [np.sqrt(abs(0.15 ** 2 - (x - 0.5) ** 2)) + 0.8 for x in np.linspace
        (0.35, 0.65, 100)], linestyle="--", color="k")
    plt.plot(np.linspace(0.35, 0.65, 100), [-np.sqrt(abs(0.15 ** 2 - (x - 0.5) ** 2)) + 0.8 for x in np.
        linspace(0.35, 0.65, 100)], linestyle="--", color="k")

    # Decorate plot
    plt.legend(["$C_{1}$", "$C_{-1}$", "Ideal separator"])
    plt.title("Training data points")
    plt.savefig(fname="data.pdf")

def plot_sep(n, x, d, xsv, sep, eval_pts):
    # Create a new figure
    plt.figure()

    # Plot the training set
    plt.plot([x[i][0] for i in range(n) if d[i] == 1], [x[i][1] for i in range(n) if d[i] == 1], linestyle="
        none", marker="x", fillstyle="none")
```

```
   plt.plot([x[i][0] for i in range(n) if d[i] == -1], [x[i][1] for i in range(n) if d[i] == -1], linestyle="
        none", marker="s", fillstyle="none")

   # Plot support vectors
   plt.plot([x[0] for x in xsv], [x[1] for x in xsv], color="k", marker="o", markersize=12, linestyle="none",
        fillstyle="none")

   # Evaluate the separator function on a eval_pts*eval_pts grid
   evals = np.zeros((eval_pts, eval_pts))
   for i in range(eval_pts):
      for j in range(eval_pts):
         evals[j][i] = sep(np.array([i / eval_pts, j / eval_pts]))
      print(i)

   # Plot H, H+ and H-
   xx, yy = np.meshgrid(np.linspace(0, 1, eval_pts), np.linspace(0, 1, eval_pts))
   contours = plt.contour(xx, yy, evals, levels=[-1, 0, 1])
   plt.clabel(contours)

   # Decorate plot
   plt.legend(["$C_{1}$", "$C_{-1}$", "Support vectors"])
   plt.title("Trained SVM separation")
   plt.savefig(fname="sep.pdf")

# Generate the training set
n = 100
x = [np.random.uniform(0, 1, size=2) for i in range(n)]
d = [1 if x[i][1] < 0.2 * np.sin(10 * x[i][0]) + 0.3 or (x[i][1] - 0.8) ** 2 + (x[i][0] - 0.5) ** 2 < 0.15 **
     2 else -1 for i in range(n)]

# Plot the training set
plot_data(n, x, d)

# Compute the separator
k1 = kern_poly(10)
k2 = kern_gaus(1)
sep, xsv, dsv = solve_svm(n, x, d, k2, 1e-6)

# Print support vectors
print("Support vectors:")
for sv in xsv:
   print(sv.transpose(), sep(sv))

# Evaluate the separator for drawing
eval_pts = 1000

# Plot the results
plot_sep(n, x, d, xsv, sep, eval_pts)
```