LAB1
Optimisation

Support Vector Machines

Teacher: Patrick MAILLE
Assistants: Augustin GODINOT, Anaelle JACOB, Maël PEREIRA

augustin.godinot@univ-rennes.fr

The goal of this lab is to study a typical classification problem from the original formulation to a numerically solvable optimization problem. Along the way, you will also learn how to use a very handy optimization library named `CVXPY`.

1. Linear separating hyperplane

Consider n points $X = (\boldsymbol{x}_1, ..., \boldsymbol{x}_n)^T \in \mathbb{R}^{n \times d}$, with labels $Y = (y_1, ..., y_n) \in \{-1, 1\}^n$. Based on all the examples in X and their labels in Y, the goal is to find a function $\hat{f} : \mathbb{R}^d \to \{-1, 1\}$ able to predict the label of unseen points $\boldsymbol{x} \in \mathbb{R}^d$. To begin with, we consider functions of the form $\hat{f}_{\boldsymbol{w}} = \text{sgn}(\boldsymbol{w}^T \boldsymbol{x})$, $\boldsymbol{w} \in \mathbb{R}^d$ called *linear classifiers*. Thus, finding a *good* $\hat{f}$ reduces to finding a *good* $\boldsymbol{w} \in \mathbb{R}^d$.

Before any theoretical or practical development, we want to be able to easily test our ideas and implementation on a simple case. To do so, we study a dataset with n points (n even) $(\boldsymbol{x}_1, ..., \boldsymbol{x}_n)^T \in \mathbb{R}^{n \times 2}$ and a scale parameter $\sigma \in \mathbb{R}_+^*$.

$$\begin{aligned} \forall i \in [\![1, \frac{n}{2}]\!], \quad \boldsymbol{x}_i &\sim \mathcal{N}\left(\begin{bmatrix} -1 \\ 1 \end{bmatrix}, \sigma I_2\right) \\ \forall i \in [\![\frac{n}{2} + 1, n]\!], \quad \boldsymbol{x}_i &\sim \mathcal{N}\left(\begin{bmatrix} 1 \\ -1 \end{bmatrix}, \sigma I_2\right) \end{aligned} \tag{1}$$

A good practice when importing/creating datasets is to *standardize* it by centering and normalizing each feature. The operation is described in (2), where $\oslash$ denotes element-wise division.

$$\begin{aligned} \forall j \in [\![d]\!], \, m_j &= \frac{1}{n} \sum_{i=1}^{n} (\boldsymbol{x}_i)_j, \\ s_j &= \frac{1}{n} \sum_{i=1}^{n} \left((\boldsymbol{x}_i)_j - m_j \right)^2 \\ \forall i \in [\![n]\!], \, \text{Standardize}(\boldsymbol{x}_i) &= (\boldsymbol{x}_i - \boldsymbol{m}) \oslash \sqrt{\boldsymbol{s}} \end{aligned} \tag{2}$$

PY1 Write a function to generate and standardize the dataset described in (1). You can take $n = 50$ in your plot.

```python
def generate_2d_dataset(n_samples: int, scale: float) ->
tuple[np.ndarray, np.ndarray]
```

Q1 Why would it be important to standardize the features of the dataset before trying to find linear classifier $f_{\boldsymbol{w}}$?

PY2 Visualize the dataset as in Figure 1 for a scale parameter $\sigma = 0.6$. For this, you can use the `matplotlib` function `scatter(X, Y)`.

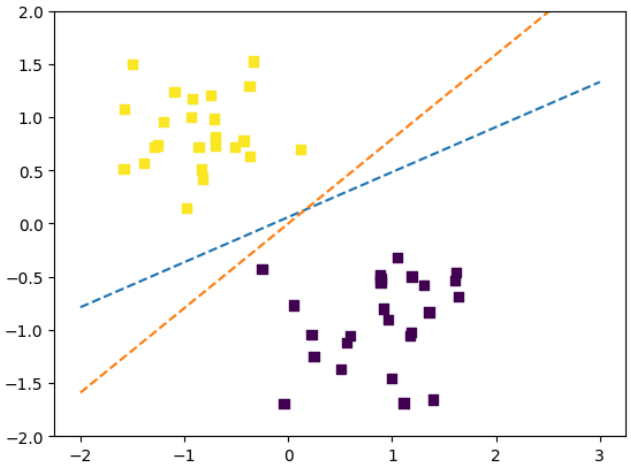

Figure 1: An example of 2D classification with two classes

Q2 In the litterature, you will often see linear classifiers defined as $\hat{f}^{\text{aff}}_{\boldsymbol{w}',\beta}(\boldsymbol{x}') = (\boldsymbol{w}')^T\boldsymbol{x}' + \beta$ (actually, this is an *affine* classifier). Show that any affine classifier $\hat{f}^{\text{aff}}_{\boldsymbol{w}',\beta} : \mathbb{R}^d \to \mathbb{R}$ can be expressed as a linear classifier $\hat{f}_{\boldsymbol{w}} : \mathbb{R}^{d+1} \to \mathbb{R}$.

From now on, when we study classifiers in 2D, we will mean that $d = 2$ and $\boldsymbol{x}_i \in \mathbb{R}^2$ but we will use linear classifiers $\hat{f}_{\boldsymbol{w},\beta} : \mathbb{R}^3 \to \mathbb{R}$ using the equivalence showed in **Q2**. To visualize the classifier, it is useful to plot the *decision boundary* of $\hat{f}_{\boldsymbol{w}}$ defined as $\mathcal{H}_{\boldsymbol{w}} = \{\boldsymbol{x} \in \mathbb{R}^d \mid \boldsymbol{w}^T\boldsymbol{x} = 0\}$.

Q3 Explain how to obtain the decision boundary in 2D (with $\boldsymbol{w} \in \mathbb{R}^3$).

PY3 Create a function to compute the 2D decision boundary from a linear classifier's weights $\boldsymbol{w} \in \mathbb{R}^3$.

```
def decision_boundary_2d(w: np.ndarray) -> np.ndarray
```

PY4 Try a few set of weights and plot them along the dataset as in Figure 1. The weights used in Figure 1 are

```
w = np.array([-24, 20, 0])
w_1 = np.array([-1, 2, 0])
```

Q4 There are two classification functions in Figure 1 that perfectly separate the two classes. In fact, there are infinitely many, how can this be an issue?

2. Maximum margin linear classifier

To avoid the problem raised in **Q4**, we introduce the notion of *margin* of $\hat{f}_{\boldsymbol{w}}$ as the distance between the decision boundary and the closest point to it. The goal is to find the set of weights $\hat{\boldsymbol{w}}$ that separate the data while maximizing the distance from the closest example to the decision boundary.

$$\begin{aligned} \hat{\boldsymbol{w}} = \underset{\boldsymbol{w}\in\mathbb{R}^d}{\text{argmax}} \quad & \overbrace{\min_{i\in[\![n]\!]} d(\boldsymbol{x}_i, \mathcal{H}_{\boldsymbol{w}})}^{\text{margin}} \\ \text{s.t.} \quad & \underbrace{y_i = \hat{f}_{\boldsymbol{w}}(\boldsymbol{x}_i) \quad \forall i \in [\![n]\!]}_{\text{separation of data}} \end{aligned} \tag{3}$$

Q5 Assuming that $\|\boldsymbol{w}\| = 1$, express the distance $d(\boldsymbol{x}_i, \mathcal{H}_{\boldsymbol{w}})$ between any point and the decision boundary. Why can we assume that $\|\boldsymbol{w}\| = 1$ without loss of generality?

Q6 Prove that $y_i = \hat{f}_{\boldsymbol{w}}(\boldsymbol{x}_i) \Leftrightarrow y_i(\boldsymbol{w}^T \boldsymbol{x}_i) > 0$.

Q7 Show that our initial problem (3) is equivalent to

1. Solve

$$\begin{aligned} \hat{\boldsymbol{w}}_0 = \underset{\boldsymbol{w}}{\operatorname{argmin}} \quad & \frac{1}{2}\|\boldsymbol{w}\|^2 \\ \text{s.t.} \quad & y_i \boldsymbol{w}^T \boldsymbol{x}_i \geq 1 \quad \forall i \in [\![n]\!] \end{aligned} \tag{4}$$

2. Return $\hat{\boldsymbol{w}} = \frac{\boldsymbol{w}_0}{\|\boldsymbol{w}_0\|}$

Since we have not studied the optimization algorithms yet, we will use the `CVXPY` library. This library allows to very easily specify an optimization problem using a set of predefined primitives. In addition, the user is provided with a set of already implemented solvers wrapped in a convenient unified interface.

PY5 Using the `CVXPY` library, implement the minimization of (4) as a function

```
def robust_linear(x: np.ndarray, Y: np.ndarray) -> np.ndarray
```

PY6 Using your `decision_boundary_2d` function, plot the decision boundary in 2D. Visualize the margin by plotting the boundaries $\mathcal{H}_{\boldsymbol{w}}^+ = \{\boldsymbol{x} \in \mathbb{R}^d \mid \boldsymbol{w}^T \boldsymbol{x} = 1\}$ $\mathcal{H}_{\boldsymbol{w}}^- = \{\boldsymbol{x} \in \mathbb{R}^d \mid \boldsymbol{w}^T \boldsymbol{x} = -1\}$.

Q8 Progressively increase the scale σ of the gaussians, what happens to the solution of the optimization problem?

3. What if the data is not perfectly separable?

As you observed in **Q8**, our classification algorithm breaks down when the data is not separable. Even worse, in practice this breakdown occurs even if the linear separation hypothesis is only slightly broken. To overcome this difficulty, we relax the optimization problem by introducing *slack variables* $\boldsymbol{\xi} = (\xi_1, ..., \xi_n) \in \mathbb{R}_+^n$ with ℓ_1 regularization (C is a positive constant).

$$\begin{aligned} \hat{\boldsymbol{w}}_0 = \underset{\boldsymbol{w}, \boldsymbol{\xi}}{\operatorname{argmin}} \quad & \frac{1}{2}\|\boldsymbol{w}\|^2 + \overbrace{C\, \mathbf{1}^T \boldsymbol{\xi}}^{\ell_1 \text{ regularization}} \\ \text{s.t.} \quad & y_i \boldsymbol{w}^T \boldsymbol{x}_i \geq 1 - \xi_i \qquad \forall i \in [\![n]\!] \\ & \underbrace{\xi_i \geq 0}_{\substack{\text{slack} \\ \text{variables}}} \qquad \forall i \in [\![n]\!] \end{aligned} \tag{5}$$

Q9 Show that if the data is linearly separable, the solution of (5) is the same as the solution of (4)

PY7 Using the `CVXPY` library, implement the minimization of (5) as a function

```
def svm_linear(x: np.ndarray, Y: np.ndarray, C: float) -> np.ndarray
```

Q10 Vary the scale parameter σ and the regularization parameter C, what is their influence on the type of solutions. To help your analysis, you can plot the decision boundaries as in **PY6**.

LAB2 Optimisation1

Optimization algorithms

Teacher: Patrick MAILLE
Assistants: Augustin GODINOT, Anaelle JACOB, Maël PEREIRA

augustin.godinot@univ-rennes.fr

The goal of this practical lab is to implement and understand how classical algorithms for convex optimization work. The final objective is to apply these algorithms to the problem we studied in LAB1.

Instructions. In the first part, the tasks are split into multiple questions in order to guide you. Throughout the lab, you will have more and more autonomy. Do not hesitate to use the structure of the questions of the first part to help you organize the rest of the lab.

1. Unconstrained optimization

In this section, we (actually you) are going to implement two optimization algorithms for *unconstrained optimization*. We consider the following problem with $f : \mathbb{R}^d \to \mathbb{R}$ a convex, twice continuously differentiable function.

$$\min_{\boldsymbol{x} \in \mathbb{R}^d} f(\boldsymbol{x}) \tag{6}$$

We will study two methods:

- The gradient descent as an example of a classical first order method.
- The Newton algorithm as a classical second order method.

1.1. Preliminaries

To test your code while you write it, it is always a good idea to first create simple examples to visualize the results. The two functions we are going to consider are

$$\begin{aligned} f_1(x_1, x_2) &= (x_1 - 2)^2 (\sin(x_2 - 1))^2 + x_1^2 + x_2^2 \\ f_2(x_1, x_2) &= x_1^2 + 3x_2^2 \end{aligned} \tag{7}$$

Q1 Express the gradient and Hessian of f_1 and f_2 in closed form.

PY1 Implement the functions f_i, their gradients ∇f_i and Hessian $\nabla^2 f_i$ as python functions of the form:

```
def f(x: np.ndarray) -> float
def grad_f(x: np.ndarray) -> np.ndarray
def hess_f(x: np.ndarray) -> np.ndarray
```

The `matplotlib.pyplot` library has a function `imshow(Z)` that allows to plot 2D functions as heatmap. Additionally, to visualize isolines $\{(x_1, x_2) : f(x_1, x_2) = c\}$ (lines on which f is constant), you can use the function `contour(Z)`. Finally, the optimization algorithms we are going to study output a list of *iterates* $(\boldsymbol{z}_1, ..., \boldsymbol{z}_m)^T \in \mathbb{R}^{m \times d}$. We want to plot the path they describe in $\mathbb{R}_d$. For the 2D case, we can use the function `plot(X, Y)`.

PY2 Use `imshow(Z)` and `contour(Z)` to create a visualization function.

```
def visualize(
    f: Callable[[np.ndarray], float],
    iterates: list[np.ndarray],
) -> tuple[plt.Figure, plt.Axes]
```

Use it to plot f_1 and f_2.

1.2. Gradient descent

The gradient algorithm has a key hyperparameter: how the step size is chosen. Let's explore two options: backtracking and constant step size with normalized gradient.

PY3 Implement the gradient descent with constant step size algorithm as a function. Do not forget to normalize the step size by the gradient norm ($\alpha_t = \frac{\alpha}{\|\nabla f(x_t)\|}$).

```
def gradient_descent(
    grad_f: Callable[[np.ndarray], np.ndarray],
    init: np.ndarray,
    step_size: float,
    tolerance: float,
) -> list[np.ndarray]
```

Q2 Try your algorithm on the two example functions f_1 and f_2

Q3 Why is it important to normalize the step size by the gradient norm? What could happen if we did not normalize it?

PY4 Implement the backtracking algorithm as a function that takes in the function f, its derivative ∇f, the start point x_t and direction d and returns the step size α_t.

```
def backtracking(
    f: Callable[[np.ndarray], float],
    grad_f: Callable[[np.ndarray], np.ndarray],
    start: np.ndarray,
    direction: np.ndarray,
    alpha: float,
    beta: float,
) -> float
```

PY5 Implement the gradient descent with backtracking line search.

```
def gradient_descent_backtracking(
    grad_f: Callable[[np.ndarray], np.ndarray],
    init: np.ndarray,
    alpha: float,
    beta: float,
    tolerance: float,
) -> list[np.ndarray]
```

Q4 Try your algorithm on the two example functions f_1 and f_2.

Q5 If you knew that your optimization algorithm only accepted functions of the form $f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T A\boldsymbol{x} - \boldsymbol{b}^T\boldsymbol{x}$, how would you improve the line search step?

1.3. Newton algorithm

The gradient descent we have studied consists in following the direction of steepest descent in ℓ_2 norm. However, as you might have observed in Section 1.2, it might produce unwanted oscillations. The Newton algorithm is very similar to gradient descent but follows the direction of steepest descent in Hessian norm $\|\boldsymbol{u}\|_{\nabla^2 f} = (\boldsymbol{u}^T \nabla^2 f \boldsymbol{u})^{\frac{1}{2}}$.

PY6 Implement the two versions of Newton method (with constant step size and backtracking) using the same function signatures as in **PY3** and **PY4**. You will have to add a Hessian parameter.

Q6 Try your algorithms on the two example functions f_1 and f_2.

Q7 Compare the results you obtained with the results of gradient descent.

Q8 BONUS: how to accelerate the direction computation?

2. Constrained optimization

Handling constraints is very important for applications where the environment enforces rules on the optimization variables. Recall the standard form of a constrained optimization program:

$$\begin{aligned} \min_{x\in\mathbb{R}^d} \quad & f_0(\boldsymbol{x}) \\ \text{s.t.} \quad & f_i(\boldsymbol{x}) \leq 0 \\ & A\boldsymbol{x} = \boldsymbol{b} \end{aligned} \tag{8}$$

There are a lot of techniques to deal with those constraints. In this lab, we will explore two of them:
- Constrained Newton algorithm
- Barrier method

2.1. Preliminaries

As before, to test the code while we write it, we define two constrained optimization algorithms

$$\begin{aligned} \min_{\boldsymbol{x}\in\mathbb{R}^2} \quad & (1-x_1)^2 + 100(x_2 - x_1^2)^2 \\ \text{s.t.} \quad & 3x_1 + x_2 = 1 \end{aligned} \tag{9}$$

$$\begin{aligned} \min_{\boldsymbol{x}\in\mathbb{R}^2} \quad & (1-x_1)^2 + 100(x_2 - x_1^2)^2 \\ \text{s.t.} \quad & (x_1+1)^2 + (x_2-3)^2 \leq 1 \end{aligned} \tag{10}$$

Q9 Explain which method can be used for which optimization problem. What is the geometric object described by the constraint of (9) and (10)?

Q10 Write the problems in standard form (i.e. express f_0, f_i, A and b for each of the problems). Express the gradient and Hessian of the objectives and the constraint functions.

PY7 Use the function of **PY2** to visualize the objective functions of problems (9) and (10). Add in the visualization of the constraints. For the constraints of the problem (10), you can use the following matplotlib instructions:

```
# ax is for example created with the following command
fig, ax = plt.subplots()

# Fill in x_1, x_2, and r with the appropriate values
ax.add_artist(
    Circle(
        [x_1, x_2], r, facecolor="b", hatch="/", color="m", alpha=0.4
    )
)
```

Q11 What is a feasible point and why is it important? Find a feasible point for (9) and (10). To verify that the points you find are indeed feasible, you can plot them in the visualization of **PY7**.

2.2. Handling equaliy constraints: constrained Newton method

The constrained Newton method allows to solve equality constrained optimization problems of the form

$$\begin{aligned} \min_{x\in\mathbb{R}^d} \quad & f_0(\boldsymbol{x}) \\ \text{s.t.} \quad & A\boldsymbol{x} = \boldsymbol{b} \end{aligned} \tag{11}$$

ConstrainedNewton(f_0, A, b, ε):

repeat
  compute $\Delta x_{\text{nt}}, \lambda(x)$
  if $\frac{\lambda(x)^2}{2} < \varepsilon$
    break
  Choose step size t // line search
  Update $x = x + t\Delta x_{\text{nt}}$
return x

Consider a feasible point x. The Newton step Δx_{nt} is the solution of

$$\begin{bmatrix} \nabla^2 f(x) & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} \Delta x_{\text{nt}} \\ w \end{bmatrix} = \begin{bmatrix} -\nabla f(x) \\ 0 \end{bmatrix} \tag{12}$$

The Newton increment is defined as

$$\lambda(x) = \left(-\nabla f(x)^T \Delta x_{\text{nt}}\right)^{1/2} \tag{13}$$

Q12 How is equation (12) giving the value of Δx_{nt} obtained? What does $\lambda(x)$ represent?

PY8 Implement the constrained Newton algorithm with the two line search methods and test your code on the two example problems.

PY9 What happens if you initialize your method at an infeasible point?

2.3. Handling inequality constraints: barrier methods

Barrier methods form a family of transformations from equality+inequality constrained problems to a sequence of equality constrained problems. The transformed problem introduces a *barrier function* $\phi(\boldsymbol{x})$ derived from the inequality constraint functions $(f_i)_i$ and solves the following problem for different values of t.

$$\begin{aligned} \min_{x\in\mathbb{R}^d} \quad & tf_0(\boldsymbol{x}) + \phi(\boldsymbol{x}) \\ \text{s.t.} \quad & f_i(\boldsymbol{x}) \leq 0 \end{aligned} \tag{14}$$

In this lab, we consider the log barrier function: $\phi(\boldsymbol{x}) = -\sum_i \log(-f_i(x))$.

Q13 Give the expression of $\nabla\phi(\boldsymbol{x})$ an $\nabla^2\phi(\boldsymbol{x})$ as a function of $f_i(x)$, $\nabla f_i(x)$ and $\nabla^2 f_i(x)$

PY10 Implement the barrier method with the two line search methods and test your code on the two example problems. Note that you can use the function you created in **PY8**!