WASP course
“Artificial Intelligence and Machine Learning”
Module-2

Choose one project. Send it before May 30 (including). Contact me, if you need more time.

General recommendations. Plots have to be nice, legends (if present) have to be readable. Think about this assignment as a section with experiments in your research paper. Then imagine me as an adversarial reviewer who is going to reject your paper if presentation doesn't look good.

Additionally to the code, the pdf report has to contain all plots, conclusions, observations, and instructions how to run your code if it is not straightforward. If you use Jupyter or Pluto notebooks, you may include all this information directly there (without a pdf report).

If you know a better way of presenting this assignment but not sure whether it is allowed, ask me in Canvas.

In 3 projects below the dataset is generated by a function `generate_halfmoon` provided in the supplementary. I give Python and Julia variants, let me know if you use a different language and are not able to translate that code.

☞ **Bonus questions are not mandatory!**

In general, don't hesitate to ask me if you have some doubts.

Project 1: Classification with logistic regression

Given dataset $(x_1, y_1), \dots, (x_n, y_n)$ with $x_i \in \mathbb{R}^d$ and binary labels $y_i \in \{-1, 1\}$, logistic regression classifies new data x by $y = \operatorname{sign}(\langle \theta, \varphi(x) \rangle)$, where $\varphi\colon \mathbb{R}^d \to \mathbb{R}^m$ is the feature map and θ is a solution of the logistic regression problem

$$\min_\theta \frac{1}{n} \sum_{i=1}^n \log(1 + e^{-y_i \langle \theta, \varphi(x_i) \rangle}), \tag{1}$$

which is our empirical risk minimization problem. Your task is to implement an abstract logistic regression. Meaning that you code has to be general enough to allow me to consider different datasets and feature maps φ. Concretely,

1. Implement functions that compute the objective and gradient in (1).
2. Check whether your implementation of the gradient is correct using automatic differentiation.
3. To solve logistic regression (1), you can use any solver that uses first-order information (function and its gradient). You can call an optimization solver directly, or alternatively implement your own.

4. Now apply all this machinery to the concrete problem. The dataset (x_i, y_i) of $n = 200$ points is generated by `generate_halfmoon(n1=100, n2=100, max_angle=3.14)`. Solve it for three different feature maps, which are polynomials of degree 1, 2, and 3:

$$\varphi(x) = [1,\, x_1,\, x_2]$$
$$\varphi(x) = [1,\, x_1,\, x_2,\, x_1^2,\, x_2^2,\, x_1x_2]$$
$$\varphi(x) = [1,\, x_1,\, x_2,\, x_1^2,\, x_2^2,\, x_1x_2,\, x_1^3,\, x_2^3,\, x_1^2x_2,\, x_1x_2^2]$$

   (I hope it is clear that in the equations above x_1 is the coordinate of x). If you prefer, you can work with kernels instead of using φ directly.

5. Recall that our predictor is $f_\theta(x) = \operatorname{sign}(\langle\theta, \varphi(x)\rangle)$ and we have already learned θ. Now for a better visualization we have to plot the decision boundaries (also known as decision surface, or region). For the test data we choose a uniform grid of points in our region. For each of these point we compute its label. With these labels we can make a contour plot of f_θ. An example is given in Figure 1.

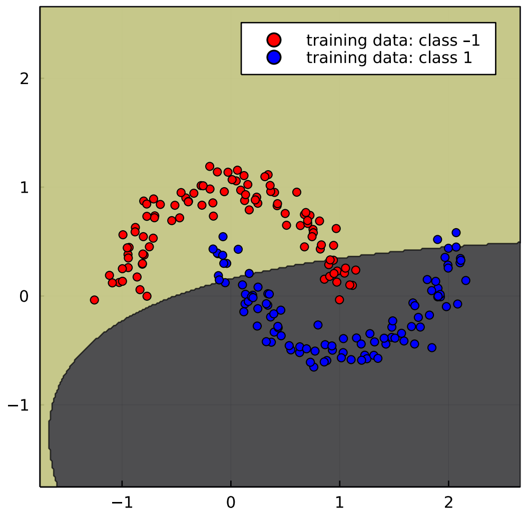

Figure 1: An example of a decision boundary for classification problem (with 200×200 points in the grid). The function takes only two values ± 1, hence only two regions.

6. Write some conclusions and all interesting observations you have collected. If you have some questions that you find interesting, write them down too.

Bonus: Make an animation/gif to show how the decision boundary changes with each iteration of your optimization algorithm.

PROJECT 2: STEPSIZE (LEARNING RATE) IN OPTIMIZATION

In this task you have to study the impact of the stepsize (learning rate) on the convergence of optimization algorithms. Our optimization problem is $\min_\theta F(\theta)$, where F is the logistic regression

$$F(\theta) = \frac{1}{n}\sum_{i=1}^{n} \log(1 + e^{-y_i\langle x_i, \theta\rangle}),$$

as in (1) with $\varphi(x) = x$. You have to implement several algorithms. All three algorithms require some initial θ_0.

- Gradient Descent: $\theta_{k+1} = \theta_k - \alpha\nabla F(\theta_k)$, where $\alpha \in (0, \frac{2}{L})$.

- Accelerated Gradient Descent

$$\rho_k = \theta_k + \frac{k-1}{k+2}(\theta_k - \theta_{k-1})$$
$$\theta_{k+1} = \rho_k - \alpha \nabla F(\rho_k),$$

  where $\alpha \in (0, \frac{2}{L})$.

- Adaptive Gradient Descent

$$\alpha_k = \min\left\{\sqrt{1+\frac{\alpha_{k-1}}{\alpha_{k-2}}}\alpha_{k-1}, \frac{\|\theta_k - \theta_{k-1}\|}{2\|\nabla F(\theta_k) - \nabla F(\theta_{k-1})\|}\right\}$$
$$\theta_{k+1} = \theta_k - \alpha_k \nabla F(\theta_k),$$

  where α_0, α_1 are arbitrary.

For the first two algorithms the theoretical bound for the stepsize is $\alpha \in (0, \frac{2}{L})$, where L is the Lipschitz constant of ∇F. You can either compute L theoretically or tune α in practice by trial and error. To be well-defined, the third algorithm requires some α_{-2}, α_{-1} and θ_{-1}. You can choose $\alpha_{-2} = \alpha_{-1} = 1$ and $\theta_{-1} = \theta_0(1+\varepsilon)$ with some small ε, say $\varepsilon = 1e-5$ (or use your own rule, it shouldn't be that important).

1. Compute the gradient of F analytically. Check whether your implementation is correct using automatic differentiation.
2. Implement all three algorithms.
3. Test them on the dataset (x_i, y_i) with 200 points generated by `generate_halfmoon(n1=100, n2=100, max_angle=3.14)`.
4. For comparison, for each algorithm plot $\|\nabla F(\theta_k)\|$ vs the number of iterations, k. Here your goal is to reach a high-accuracy solution, say 10^{-8}. To display plots, use a semilog axis (log for y-axis).
5. Write some conclusions and all interesting observations you have collected. If you have some questions that you find interesting, write them down too.

Bonus: Try to do the same with a real dataset. For instance, you can choose one in UCI Machine Learning Repository (classification category). Also some libraries provide such datasets more conveniently.

PROJECT 3: SGD, TRAINING AND TEST ERRORS

We have a training dataset (x_i, y_i) of $n = 200$ points generated by `generate_halfmoon(n1=100, n2=100, max_angle=2)`. The test dataset is generated in the same way (randomness will make sure that they are close but different). We want to linearly separate the data. To this end, we want to learn $\theta = [w, b] \in \mathbb{R}^3$ by solving a logistic regression $\min_\theta F(\theta)$, where

$$F(\theta) = \frac{1}{n}\sum_{i=1}^{n} \log(1 + e^{-y_i(\langle x_i, w\rangle + b)}),$$

Let $F_i(\theta) = \log(1 + e^{-y_i(\langle x_i, w\rangle + b)})$. Your goal is to solve this problem by stochastic gradient descent (SGD):

$$\text{Choose } i_k \in \{1, \dots, n\} \text{ uniformly at random}$$
$$\theta_{k+1} = \theta_k - \alpha \nabla F_{i_k}(\theta_k)$$

and show how the train and test error change with iterations. Recall that the train error is $F(\theta)$ and the test error is also $F(\theta)$, but with a test dataset in the definition of F. For simplicity we use a fixed stepsize α (usually it has to be small).

Do the following:

1. Compute the gradient of F analytically. Check whether your implementation is correct using automatic differentiation.

2. Implement SGD on a training dataset. Every n-th iteration (i.e., each epoch), compute the train and test errors, make correspondent plots.

3. Do the same for different stepsize α in some reasonable range. Which one seems to be the best (recall what is our ultimate goal)?

4. Write some conclusions and all interesting observations you have collected. If you have some questions that you find interesting, write them down too.

Bonus: Make an animation/gif to illustrate how a linear separator constructed in every iteration (or every n-th iteration) changes with time.

PROJECT 4: SVM, KERNELS AND NON-LINEAR DATA

We have a dataset of 200 points $x_i \in \mathbb{R}^2$ each drawn from the standard 2-dimensional normal distribution. We want to learn a classifier that separates those points, whose coordinate have the same sign, from the rest. For this we will use SVM with kernels. We will use two kernels: $k(x, x') = (1 + \langle x, x' \rangle)^2$ and $k(x, x') = e^{-\gamma\|x-x'\|^2}$, where $\gamma > 0$.

1. For each kernel, formulate SVM classification problem (with soft margins) as an optimization problem (any formulation).

2. Solve it by any optimization solver you like and obtain final w, and b.

3. Alternatively use some library that solves SVM with a predefined kernel directly. Recover optimal w, b.

4. How does the predictor for SVM with kernels look? For a linear kernel it would be $f_{w,b}(x) = \text{sign}(\langle w, x \rangle + b)$. Write it down as an equation. Now for a better visualization, plot the decision boundaries (also known as decision surface, or region). For the test data we choose a uniform grid of points in our region. For each of these point we compute its label based on our predictor $f_{w,b}$. With these labels we can make a contour plot of $f_{w,b}$. An example (for a different dataset) is given in Figure 1.

5. For the second kernel do the same for different values of $\gamma \in \{0.5, 1, 2\}$. Is there any difference in the results?

6. Write some conclusions and all interesting observations you have collected. If you have some questions that you find interesting, write them down too.

Bonus: Choose a function $k(x, x')$ that may be not a positive definite kernel, for instance $k(x, x') = \frac{1}{1+\|x-x'\|^2}$ or use your imagination. Can you perform the same task as above with this kernel?

Supplementary

The following Python code you can also find in the separate file `dataset.py`. Similarly, Julia code can be found in `dataset.jl`.

```python
import numpy as np

def generate_halfmoon(n1, n2, max_angle=np.pi):
  alpha = np.linspace(0, max_angle, n1)
  beta = np.linspace(0, max_angle, n2)
  X1 = np.vstack([np.cos(alpha), np.sin(alpha)])\
      + 0.1 * np.random.randn(2,n1)
  X2 = np.vstack([1 - np.cos(beta), 1 - np.sin(beta) - 0.5])\
      + 0.1 * np.random.randn(2,n2)
  y1, y2 = -np.ones(n1), np.ones(n2)
  return X1, y1, X2, y2
```