COMP 3105 Introduction to Machine Learning
Assignment 1

Instructor: Junfeng Wen (junfeng.wen [AT] carleton.ca)

Fall 2022
School of Computer Science
Carleton University

Deadline: 11:59 pm, Friday, Sept. 30, 2022

Instructions: Download the file `A1files.zip` from Brightspace and unzip it to see the available files. You need to write a single Python file `A1codes.py` that includes all your implementations of the required functions. You also need to write a single PDF file `A1report.pdf` that includes all your answers to the written questions. The PDF should also specify your running environment including a list of Python libraries/packages and their versions required to run your codes. When finished, zip the two files into a single zip file `A1answers.zip` and upload it to Brightspace for marking.

Rubrics: This assignment is worth 15% of the final grade. Your codes and report will be evaluated based on their scientific qualities including but not limited to: Are the implementations correct? Is the analysis rigorous and thorough? Are the codes easily understandable (with proper comments)? Is the report well-organized and clear?

Policies:

- You can finish this assignment in groups of two. All members of a group will receive the same mark.
- You may consult others (classmates/TAs) about general ideas but don't share answers. Please specify in the PDF file any individuals you consult for the assignment. Any group found to cheat will receive a score of 0 for this assignment and may be subject to further punishment.
- Remember that you have **three** excused days *throughout the term* (rounded up to the nearest day), after which no late submission will be accepted.
- Specifically for this assignment, you can use libraries with general utilities, such as matplotlib, numpy/scipy, cvxopt, and pandas for Python. **However, you must implement everything by yourselves without using any pre-existing implementations of the algorithms or any functions from an ML library (such as scikit-learn)**. The goal is for you to really understand, step by step, how the algorithms work.

Question 1: Linear Regression

In this question, you will implement linear regression from scratch, in Python using Numpy/Scipy, and evaluate their performances on a synthetic dataset. You will learn the basics of array manipulations and matrix/vector operations (e.g., use `@` for matrix multiplication, `X.T` to transpose a matrix `X` etc). You will also learn some essential built-in functions like `numpy.linalg.solve` to solve linear systems and `cvxopt.solvers.lp` to solve linear programmings.

All of the following functions must be able to handle arbitrary $n > 0$ and $d > 0$. The vectors and matrices are represented as numpy arrays. Your functions shouldn't print additional information to the standard output.

(a) (1%) Implement a Python function

```
w = minimizeL2(X, y)
```

that takes an $n \times d$ input matrix `X` and an $n \times 1$ target/label vector `y`, and returns a $d \times 1$ vector of weights/parameters `w` corresponding to the solution of the L_2 losses:

$$\mathbf{w} = \operatorname*{argmin}_{\mathbf{w} \in \mathbb{R}^d} \frac{1}{2n} \|X\mathbf{w} - \mathbf{y}\|_2^2. \tag{1}$$

(b) (1%) Similar to above, implement a Python function

```
w = minimizeL1(X, y)
```

that returns a $d \times 1$ vector of weights/parameters `w` corresponding to the solution of minimum L_1 loss

$$\mathbf{w} = \operatorname*{argmin}_{\mathbf{w} \in \mathbb{R}^d} \frac{1}{n} \|X\mathbf{w} - \mathbf{y}\|_1.$$

Recall that the optimization can be expressed as a linear programming with the joint parameters $\begin{bmatrix} \mathbf{w} \\ \boldsymbol{\delta} \end{bmatrix} \in \mathbb{R}^{d+n}$

$$\begin{aligned} \min_{\mathbf{w}, \boldsymbol{\delta}} \quad & \boldsymbol{\delta}^\top \mathbf{1} \\ \text{s.t.} \quad & \boldsymbol{\delta} \succeq \mathbf{0} \\ & X\mathbf{w} - \mathbf{y} \preceq \boldsymbol{\delta} \\ & \mathbf{y} - X\mathbf{w} \preceq \boldsymbol{\delta} \end{aligned}$$

(c) (1%) Similar to above, implement a Python function

```
w = minimizeLinf(X, y)
```

that returns a $d \times 1$ vector of weights/parameters `w` corresponding to the solution of minimum L_∞ loss

$$\mathbf{w} = \operatorname*{argmin}_{\mathbf{w} \in \mathbb{R}^d} \|X\mathbf{w} - \mathbf{y}\|_\infty.$$

Recall that the optimization can be expressed as a linear programming with the joint parameters $\begin{bmatrix} \mathbf{w} \\ \delta \end{bmatrix} \in \mathbb{R}^{d+1}$

$$\begin{aligned} \min_{\mathbf{w}, \delta} \quad & \delta \\ \text{s.t.} \quad & \delta \geq 0 \\ & X\mathbf{w} - \mathbf{y} \leq \delta \cdot \mathbf{1} \\ & \mathbf{y} - X\mathbf{w} \leq \delta \cdot \mathbf{1} \end{aligned}$$

(d) (1%) In this part, you will evaluate your implemented algorithms on a synthetic dataset. Implement a Python function

```
train_loss, test_loss = synRegExperiments()
```

that returns a 3×3 matrix `train_loss` of *average* training losses and a 3×3 matrix `test_loss` of *average* test losses (See Table 1 and Table 2 below). It repeats the following steps 100 times

1. Generate the training data as follows

```
n = 30  # number of data points
d = 5  # dimension
noise = 0.2
X = np.random.randn(n, d)  # input matrix
X = np.concatenate((np.ones((n, 1)), X),  axis=1)  # augment input
w_true = np.random.randn(d + 1, 1)  # true model parameters
y = X @ w_true + np.random.randn(n, 1) * noise  # ground truth label
```

2. Use your implementations to learn three linear models with L_2, L_1 and L_∞ losses. Evaluate their performance in terms of three different criteria L_2, L_1 and L_∞ losses on the *training data*, i.e., in the following form

Table 1: Different training losses for different models

Model	L_2 loss	L_1 loss	L_∞ loss
L_2 model			
L_1 model		(training loss)	
L_∞ model			

   The L_1 and L_2 losses here should be the average loss over training points.

3. Generate 1000 new test data points using the same generation code (without changing the true model parameters `w_true`). Evaluate the three models on the *test data*

Table 2: Different test losses for different models

Model	L_2 loss	L_1 loss	L_∞ loss
L_2 model			
L_1 model		(test loss)	
L_∞ model			

   The L_1 and L_2 losses here should be the average loss over test points.

In the PDF file, report the *averages* (over 100 runs) for each kind of loss and each kind of model in two tables (one for training and the other for test).

(e) (1%) Looking at your tables from above, analyze the results and discuss any findings you may have and the possible reason behind them.

Question 2: Gradient Descent & Logistic Regression

In this question, you will implement gradient descent, a commonly used optimization procedure, and apply it to solve logistic regression, a classification method.

All of the following functions must be able to handle arbitrary $n > 0$ and $d > 0$. The vectors and matrices are represented as numpy arrays. Your functions shouldn't print additional information to the standard output.

(a) (1%) Before diving into gradient decent and logistic regression, let's first revisit the linear regression in Q1(a). Implement a Python function

```
obj_val, grad = linearRegL2Obj(w, X, y)
```

that takes a $d \times 1$ vector of parameters `w`, an $n \times d$ input matrix `X` and an $n \times 1$ label vector `y`. Its first return value `obj_val` is the (scalar) value of the objective function in Eq. (1) and the second return value `grad` is the *analytic form* gradient of size $d \times 1$ (both at the given parameters `w`).

(**Debug hint**: You may want to compare your analytic gradient with the numeric gradient computed by the `grad` function from `autograd` or `jax`.)

(b) (2%) Write a Python function

```
w = gd(obj_func, w_init, X, y, eta, max_iter, tol)
```

that implements the gradient descent algorithm. It takes the following arguments

- An objective function `obj_func` (that admits the I/O as in part **(a)**)
- A $d \times 1$ initial parameter vector `w_init`
- An $n \times d$ input matrix `X`
- An $n \times 1$ label vector `y`
- A positive float step size `eta`
- A positive integer `max_iter` indicating the maximum number of iterations
- A positive float tolerance `tol`

It returns the final parameter vector `w` of size $d \times 1$ when either the algorithm has taken `max_iter` number of gradient steps, or the L_2 norm of the gradient is smaller than `tol`. Specifically, it should work as follows

```
def gd(obj_func, w_init, X, y, step_size, max_iter, tol):
    # TODO: initialize w
    for _ in range(max_iter):
        # TODO: compute gradient at current w
        # TODO: break if the L2 norm of the gradient is smaller than tol
        # TODO: perform gradient update to w
    return w
```

(**Debug hint**: You may want to compare your analytic solution from Q1(a) with the GD solution here with a proper step-size.)

(c) (1%) Implement a Python function

```
obj_val, grad = logisticRegObj(w, X, y)
```

that takes a $d \times 1$ vector of parameters `w`, an $n \times d$ input matrix `X` and an $n \times 1$ label vector `y`. Its first return value `obj_val` is the (scalar) value of the objective function (cross-entropy loss)

$$\mathbf{y}^\top \log(\sigma(X\mathbf{w})) + (\mathbf{1} - \mathbf{y})^\top \log(\mathbf{1} - \sigma(X\mathbf{w})) \tag{2}$$

where the sigmoid function σ is applied element-wisely, and the second return value `grad` is the *analytic form* gradient of size $d \times 1$ (both at the given parameters `w`).

(**Debug hint**: You may want to compare your analytic gradient with the numeric gradient computed by the `grad` function from `autograd` or `jax`.)

(**Numerical issue**: Note that the σ can produce a float number that is very close to zero, or exactly zero due to underflow. In such cases, $\log(0)$ will produce an `NAN`. To avoid this, a numerically stable implementation is required. You may want to check the `numpy.logaddexp` function. In general, you need to be careful whenever `exp` or `log` is called.)

(d) (1%) In this part, you will evaluate your implementation on a synthetic dataset using the data generation code below

```
# data generation
n = 100  # number of data points
d = 2  # dimension
c0 = np.ones([1, d])  # class 0 center
c1 = -np.ones([1, d])  # class 1 center
X0 = np.random.randn(n, d) + c0  # class 0 input
X1 = np.random.randn(n, d) + c1  # class 1 input
X = np.concatenate((X0, X1), axis=0)
X = np.concatenate((np.ones((2 * n, 1)), X), axis=1)  # augmentation
y = np.concatenate([np.zeros([n, 1]), np.ones([n, 1])], axis=0)

# learning
eta = 0.001  # learning rate
max_iter = 1000  # maximum number of iterations
tol = 1e-10  # tolerance
w_init = np.random.randn(d + 1, 1)
w_logit = gd(logisticRegObj, w_init, X, y, eta, max_iter, tol)
```

Implement a Python function

```
train_acc, test_acc = synClsExperiments()
```

that returns a 4×3 matrix `train_acc` of average training accuracies and a 4×3 matrix `test_acc` of average test accuracies (See Table 3 and Table 4 below). It repeats the following steps 100 times

1. Generate the training data as above
2. Learn a binary classifier with your logistic regression + GD. Unless specified otherwise, use the default hyper-parameters above. Compute and store the training classification accuracy with different hyper-parameters (when fixing others as the default)

Table 3: Training accuracies with different hyper-parameters

`n`	Train Accuracy	`d`	Train Accuracy	`eta`	Train Accuracy
10		1		0.001	
50		2		0.01	
100		4		0.1	
200		8		1.0	

3. Generate 1000 new test data points using the same generation code. Evaluate the model performance on the test data

Table 4: Test accuracies with different hyper-parameters

`n`	Test Accuracy	`d`	Test Accuracy	`eta`	Test Accuracy
10		1		0.001	
50		2		0.01	
100		4		0.1	
200		8		1.0	

In the PDF file, report the *averages* (over 100 runs) for each accuracy in two tables (one for training and the other for test).

(e) (1%) Looking at your tables from above, analyze the results and discuss any findings you may have and the possible reason behind them.

Question 3: Real-world Datasets

In this question, you will apply the algorithms in Q1 and Q2 to real-world datasets and investigate their performance. The datasets we will use are

- (Regression) Auto-mpg (click to see the data website). We will use the `auto-mpg.data` file.
- (Classification) Parkinsons. We will use the `parkinsons.data` file.
- (Classification) Sonar, Mines vs. Rocks. We will use the `sonar.all-data` file.

(a) (1%) Implement a Python function

```
X, y = loadData(dataset_folder, dataset_name)
```

that takes a string of the (absolute) path of the dataset folder and a string of name of the data file (`"auto-mpg.data"`, `"parkinsons.data"` or `"sonar.all-data"`) and returns the pre-processed $n \times d$ input matrix `X` and an $n \times 1$ label vector `y` (that are suitable to use for your functions in Q1 & Q2). Depending on the dataset, you need to perform the following preprocessing

- Auto-mpg: Remove the `origin` and `car name` features. Remove the data points with missing values (check the `horsepower` feature). Use the `mpg` as the target and the rest as input features.
- Parkinsons: Use the `status` as the label and the rest as input features.
- Sonar: Convert the last column into labels (`R` to 0 and `M` to 1) and the rest as input features.

(b) (1%) Implement a Python function

```
train_metric, test_metric = realExperiments(dataset_folder, dataset_name)
```

that takes the dataset path and dataset name as inputs and returns the training and test evaluation metrics (either loss for regression or accuracy for classification) similar to Q1(d) and Q2(d). It repeats the following steps 100 times

1. Randomly take 50% of the dataset as test points (round up). Use the remaining 50% to train a model (three linear regression models with different losses as in Q1 and one logistic regression model).
2. Evaluate the model performance on the training data (a 3×3 matrix as in Table 1 for regression and a 4×1 vector as the last column of Table 3 with different step-sizes).
3. Evaluate the model performance on the test data (as in Table 2 for regression and the last column of Table 4 with different step-sizes).

In the PDF file, for each dataset, report the *averages* (over 100 runs) for each metric in two tables (one for training and the other for test).

(c) (1%) Looking at your tables from above, analyze the results and discuss any findings you may have and the possible reason behind them. For regression, which method do you recommend and why? For classification, which learning rate do you recommend and why?

(d) (1%) Improve the performance of the models on the test sets by performing *additional* feature preprocessing steps. Clearly state your method and why you think it can improve performance.