COMP 3105 Introduction to Machine Learning
Assignment 1

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

Fall 2023
School of Computer Science
Carleton University

Deadline: 11:59 pm, Sunday, Oct. 1, 2023

Instructions: Download the file `A1files.zip` from Brightspace and unzip it to see the available files. **Submit the following three files** to Brightspace for marking

- A Python file `A1codes.py` that includes all your implementations of the required functions
- A pip requirements file named `requirements.txt` that speficifies the running environment including a list of Python libraries/packages and their versions required to run your codes.
- A PDF file `A1report.pdf` that includes all your answers to the written questions. It should also specify your team members (names and student IDs). Please clearly specify question/sub-question numbers in your submitted PDF report so TAs can see which question you are answering.

Do not submit a compressed file, or it may result in a mark deduction. We recommend trying your code using Colab or Anaconda/Virtualenv before submission.

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 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 when the workload is shared.
- You may consult others (classmates/TAs/LLMs) about general ideas but don't share codes/answers. Please specify in the PDF file any individuals or programs (e.g., ChatGPT) you consult for the assignment. If you use large language models (LLMs), clearly show us how you use it. Any group found to cheat or violating this policy will receive a score of 0 for this assignment.
- 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 (9%) Linear Regression

In this question, you will implement linear regression from scratch, in Python using NumPy/SciPy, and evaluate their performances on different datasets. 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 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:

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

Use the analytic solution above for your implementation.

(**Hint**: You may find `np.linalg.solve` or `np.linalg.inv` helpful.)

(b) (2%) 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 $\begin{bmatrix} \mathbf{w} \\ \boldsymbol{\delta} \end{bmatrix} \in \mathbb{R}^{d+n}$ as follows

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

(**Hints**: You may find `np.zeros`, `np.ones`, `np.concatenate` and `cvxopt.solvers.lp` helpful. Make sure all matrices/vectors are converted to `cvxopt.matrix` first before calling the solver. You should also set `solvers.options['show_progress'] = False` to silence the solver in your final submission.)

(c) (2%) 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 $\begin{bmatrix}\mathbf{w}\\ \delta\end{bmatrix} \in \mathbb{R}^{d+1}$ as follows

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

(**Hints**: You may find `np.zeros`, `np.ones`, `np.concatenate` and `cvxopt.solvers.lp` helpful. Make sure all matrices/vectors are converted to `cvxopt.matrix` first before calling the solver. You should also set `solvers.options['show_progress'] = False` to silence the solver in your final submission.)

(**Hint**: After finishing parts (a)-(c), you can use `A1testbed.py` to visualize your models and make sure they are reasonable.)

(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 100 runs as follows

```python
def synRegExperiments():

    def genData(n_points):
        '''
        This function generate synthetic data
        '''
        X = np.random.randn(n_points, d)  # input matrix
        X = np.concatenate((np.ones((n_points, 1)), X),  axis=1)  # augment input
        y = X @ w_true + np.random.randn(n_points, 1) * noise  # ground truth label
        return X, y

    n_runs = 100
    n_train = 30
    n_test = 1000
    d = 5
    noise = 0.2
    train_loss = np.zeros([n_runs, 3, 3])  # n_runs * n_models * n_metrics
    test_loss = np.zeros([n_runs, 3, 3])  # n_runs * n_models * n_metrics

    for r in range(n_runs):

        w_true = np.random.randn(d + 1, 1)
        Xtrain, ytrain = genData(n_train)
        Xtest, ytest = genData(n_test)

        # Learn different models from the training data
        w_L2 = minimizeL2(Xtrain, ytrain)
        w_L1 = minimizeL1(Xtrain, ytrain)
        w_Linf = minimizeLinf(Xtrain, ytrain)

        # TODO: Evaluate the three models' performance (for each model,
        #       calculate the L2, L1 and L infinity losses on the training
        #       data). Save them to `train_loss`
```

```
        # TODO: Evaluate the three models' performance (for each model,
        #       calculate the L2, L1 and L infinity losses on the test
        #       data). Save them to `test_loss`

    # TODO: compute the average losses over runs
    # TODO: return a 3-by-3 training loss variable and a 3-by-3 test loss variable
```

Note that the L_1 and L_2 losses should be the average loss over training/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 below.

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			

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			

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

(f) (1%) Now you will apply the learning algorithms to a real-world problem, the Auto-mpg dataset (click to see the data website). Specifically, we will use the `auto-mpg.data` file.

To start, you need to preprocess the data. Implement a Python function

```
X, y = preprocessAutoMPG(dataset_folder)
```

that takes a string of the (absolute) path of the auto-mpg `dataset_folder` (where the `auto-mpg.data` file is located) and returns the preprocessed $n \times d$ input matrix `X` and an $n \times 1$ label vector `y` (that are suitable to use for your functions in (a) to (c)). This function needs to remove the `origin` and `car_name` features, the data points with missing values (check the `horsepower` feature) and use the `mpg` as the target/label and the rest as input features.

(g) (1%) Implement a Python function

```
train_loss, test_loss = runAutoMPG(dataset_folder)
```

that takes the (absolute) dataset path and returns the training and test losses. It runs as follows

```
def runAutoMPG(dataset_folder):

    X, y = preprocessAutoMPG(dataset_folder)
    n, d = X.shape
    X = np.concatenate((np.ones((n, 1)), X),  axis=1)  # augment

    n_runs = 100
    train_loss = np.zeros([n_runs, 3, 3])  # n_runs * n_models * n_metrics
    test_loss = np.zeros([n_runs, 3, 3])  # n_runs * n_models * n_metrics

    for r in range(n_runs):
```

```
        # TODO: Randomly partition the dataset into two parts (50%
        #       training and 50% test)

        # TODO: Learn three different models from the training data
        #       using L1, L2 and L infinity losses

        # TODO: Evaluate the three models' performance (for each model,
        #       calculate the L2, L1 and L infinity losses on the training
        #       data). Save them to `train_loss`

        # TODO: Evaluate the three models' performance (for each model,
        #       calculate the L2, L1 and L infinity losses on the test
        #       data). Save them to `test_loss`

    # TODO: compute the average losses over runs
    # TODO: return a 3-by-3 training loss variable and a 3-by-3 test loss variable
```

In the PDF file, report the *averages* (over 100 runs) for each kind of loss and each kind of model using tables similar to Table 1 and Table 2.

Question 2 (6%) 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) (i.e., the value of $\frac{1}{2n}\|X\mathbf{w} - \mathbf{y}\|_2^2$), and the second return value `grad` is the *analytic form* gradient of size $d \times 1$ (both are evaluated at the given parameters `w`).

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

(b) (1%) Write a Python function

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

that implements the gradient descent (GD) algorithm. Recall that starting from an initial point $\mathbf{w}^{(0)}$, GD iterates as follows

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla J(\mathbf{w}^{(t)}) \tag{2}$$

for a given objective function $J(\mathbf{w})$, using a step size $\eta > 0$. The function 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` (i.e., $\mathbf{w}_0$)
- An $n \times d$ input matrix `X`
- An $n \times 1$ label vector `y`
- A positive float step size `eta` (i.e., η)
- 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, eta, max_iter, tol):
    # TODO: initialize w
    for _ in range(max_iter):
        # TODO: Compute the gradient of the obj_func at current w
        # TODO: Break if the L2 norm of the gradient is smaller than tol
        # TODO: Perform gradient descent update to w
    return w
```

(**Debug hint**: You can pass the objective function from Q2(a) to your `gd` function:

```
w = gd(linearRegL2Obj, w_init, X, y, eta, max_iter, tol)
```

With a proper step-size, you should get a GD solution $\mathbf{w}$ very close to your analytic solution from Q1(a) for linear regression problems. If so, then it is very likely that your `gd` function is correct.)

(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):

$$J(\mathbf{w}) = \frac{1}{n}\left[-\mathbf{y}^\top \log(\sigma(X\mathbf{w})) - (\mathbf{1}-\mathbf{y})^\top \log(\mathbf{1}-\sigma(X\mathbf{w}))\right] \tag{3}$$

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`):

$$\nabla J(\mathbf{w}) = \frac{1}{n} X^\top (\sigma(X\mathbf{w}) - \mathbf{y}).$$

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

(**Numerical issue**: Note that the sigmoid function 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.)

(**Hint**: You can use `A1testbed.py` to visualize your models and make sure they are reasonable.)

(d) (1%) In this part, you will evaluate your implementation on a synthetic dataset. 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 100 runs as follows

```
def synClsExperiments():

    def genData(n_points, d):
        '''
        This function generate synthetic data
        '''
        c0 = np.ones([1, d])  # class 0 center
        c1 = -np.ones([1, d])  # class 1 center
        X0 = np.random.randn(n_points, d) + c0  # class 0 input
        X1 = np.random.randn(n_points, d) + c1  # class 1 input
        X = np.concatenate((X0, X1), axis=0)
        X = np.concatenate((np.ones((2 * n_points, 1)), X), axis=1)  # augmentation
        y = np.concatenate([np.zeros([n_points, 1]), np.ones([n_points, 1])], axis=0)
        return X, y

    def runClsExp(m=100, d=2, eta=0.1, max_iter=1000, tol=1e-10):
        '''
        Run classification experiment with the specified arguments
        '''

        Xtrain, ytrain = genData(m, d)
        n_test = 1000
        Xtest, ytest = genData(n_test, d)

        w_init = np.zeros([d + 1, 1])
```

```
        w_logit = gd(logisticRegObj, w_init, Xtrain, ytrain, eta, max_iter, tol)
        ytrain_hat = # TODO: Compute predicted labels of the training points
        train_acc = # TODO: Compute the accuarcy of the training set

        ytest_hat = # TODO: Compute predicted labels of the test points
        test_acc = # TODO: Compute the accuarcy of the training set

        return train_acc, test_acc

    n_runs = 100
    train_acc = np.zeros([n_runs, 4, 3])
    test_acc = np.zeros([n_runs, 4, 3])
    for r in range(n_runs):
        for i, m in enumerate((10, 50, 100, 200)):
            train_acc[r, i, 0], test_acc[r, i, 0] = runClsExp(m=m)
        for i, d in enumerate((1, 2, 4, 8)):
            train_acc[r, i, 1], test_acc[r, i, 1] = runClsExp(d=d)
        for i, eta in enumerate((0.1, 1.0, 10., 100.)):
            train_acc[r, i, 2], test_acc[r, i, 2] = runClsExp(eta=eta)

    # TODO: compute the average accuracies over runs
    # TODO: return a 4-by-3 training accuracy variable and a 4-by-3 test accuracy variable
```

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

Table 3: Training accuracies with different hyper-parameters

m	Train Accuracy	d	Train Accuracy	eta	Train Accuracy
10		1		0.1	
50		2		1.0	
100		4		10.0	
200		8		100.0	

Table 4: Test accuracies with different hyper-parameters

m	Test Accuracy	d	Test Accuracy	eta	Test Accuracy
10		1		0.1	
50		2		1.0	
100		4		10.0	
200		8		100.0	

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

(f) (1%) Now you will apply logistic regression to real-world problems, the Sonar, Mines vs. Rocks dataset (click to see the data website). Specifically, We will use the `sonar.all-data` file.

To start, you need to preprocess the data. Implement a Python function

```
X, y = preprocessSonar(dataset_folder)
```

that takes a string of the (absolute) path of the `dataset_folder` (where the `sonar.all-data` file is located) and returns the preprocessed $n \times d$ input matrix `X` and an $n \times 1$ label vector `y` (that are suitable to use for your GD learning). In this function, you need convert the last column into labels (`R` to 0 and `M` to 1) and the rest as input features.

(g) (1%) Implement a Python function

```
train_acc, val_acc, test_acc = runSonar(dataset_folder)
```

that takes the (absolute) dataset path and returns the training, validation and test accuracies. It runs as follows

```
def runSonar(dataset_folder, dataset_name):

    X, y = preprocessSonar(dataset_folder)
    n, d = X.shape
    X = np.concatenate((np.ones((n, 1)), X),  axis=1)  # augment

    eta_list = [0.1, 1, 10, 100]
    train_acc = np.zeros([len(eta_list)])
    val_acc = np.zeros([len(eta_list)])
    test_acc = np.zeros([len(eta_list)])

    # TODO: Randomly partition the dataset into three parts (40%
    #       training (use the round function), 40% validation and
    #       the remaining ~20% as test)

    for i, eta in enumerate(eta_list):

        w_init = np.zeros([d + 1, 1])
        w = gd(logisticRegObj, w_init, Xtrain, ytrain, eta, max_iter=1000, tol=1e-8)

        # TODO: Evaluate the model's accuracy on the training
        #       data. Save it to `train_acc`

        # TODO: Evaluate the model's accuracy on the validation
        #       data. Save it to `val_acc`

        # TODO: Evaluate the model's accuracy on the test
        #       data. Save it to `test_acc`

    return train_acc, val_acc, test_acc
```

In the PDF file, report the accuracies of each dataset in the following table:

Table 5: Accuracies with different learning rates

eta	Train Accuracy	Validation Accuracy	Test Accuracy
0.1			
1			
10			
100			

Looking at your results here, which learning rate do you recommend to use for this dataset and why?