COMP 3105A Introduction to Machine Learning
Assignment 1

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

Fall 2025
School of Computer Science
Carleton University

Deadline: 11:59 pm, Sunday, Sept. 28, 2025

Instructions: **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 specifies 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 will 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 them. Any group found to cheat or violate 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 (7.5%) 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%) L_2 Regression

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) (3%) L_∞ Regression

Here we are going to solve the L_∞ loss regression problem

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

Recall that this 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

$$\begin{aligned} \min_{\mathbf{w}, \delta} \quad & \delta \\ \text{s.t.} \quad & \delta \geq 0 \\ & X\mathbf{w} - \mathbf{y} \preceq \delta \cdot \mathbf{1}_n \\ & \mathbf{y} - X\mathbf{w} \preceq \delta \cdot \mathbf{1}_n \end{aligned} \tag{2}$$

Now we want to convert it into a form that is solvable by the `cxvopt` linear programming (LP) solver, which solves the following form of LP

$$\begin{aligned} \min_{\mathbf{u}} \quad & \mathbf{c}^\top \mathbf{u} \\ \text{s.t.} \quad & G\mathbf{u} \preceq \mathbf{h}. \end{aligned}$$

Let the unknown variables be $\mathbf{u} = \begin{bmatrix} \underset{d\times 1}{\mathbf{w}} \\ \underset{1\times 1}{\delta} \end{bmatrix} \in \mathbb{R}^{d+1}$.

For the constraints, since we have three sets of constraints, the matrix G and $\mathbf{h}$ can be decomposed into three parts

$$G \cdot \mathbf{u} = \begin{bmatrix} \underset{1\times(d+1)}{G^{(1)}} \\ \underset{n\times(d+1)}{G^{(2)}} \\ \underset{n\times(d+1)}{G^{(3)}} \end{bmatrix} \begin{bmatrix} \mathbf{w} \\ \delta \end{bmatrix} \preceq \begin{bmatrix} \underset{1\times 1}{\mathbf{h}^{(1)}} \\ \underset{n\times 1}{\mathbf{h}^{(2)}} \\ \underset{n\times 1}{\mathbf{h}^{(3)}} \end{bmatrix} = \mathbf{h}$$

and each part corresponds to a constraint specified in Eq. (2).

Write your answers to the following questions in the PDF report. **Please specify the shape of each part of your expression clearly.**

(b.1) (0.25%) For the objective function, we want $\mathbf{c}^\top \mathbf{u} = \delta$. What should $\mathbf{c} \in \mathbb{R}^{d+1}$ be?

(b.2) (0.25%) We want $G^{(1)}\mathbf{u} \preceq \mathbf{h}^{(1)} \iff \delta \geq 0$. What should $G^{(1)} \in \mathbb{R}^{1\times(d+1)}$ and $\mathbf{h}^{(1)} \in \mathbb{R}$ be?

(b.3) (0.25%) We want $G^{(2)}\mathbf{u} \preceq \mathbf{h}^{(2)} \iff X\mathbf{w} - \mathbf{y} \preceq \delta \cdot \mathbf{1}_n$. What should $G^{(2)} \in \mathbb{R}^{n\times(d+1)}$ and $\mathbf{h}^{(2)} \in \mathbb{R}^n$ be?

(b.4) (0.25%) We want $G^{(3)}\mathbf{u} \preceq \mathbf{h}^{(3)} \iff \mathbf{y} - X\mathbf{w} \preceq \delta \cdot \mathbf{1}_n$. What should $G^{(3)} \in \mathbb{R}^{n\times(d+1)}$ and $\mathbf{h}^{(3)} \in \mathbb{R}^n$ be?

(b.5) (2%) Based on your derivations in (b), 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

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

Note that if you do not complete part (b), you won't receive marks for part (c).

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

(c) (2%) Synthetic Regression Problem

In this part, you will evaluate your implemented algorithms on a synthetic dataset.

(c.1) (1%) Implement a Python function

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

that returns a 2×2 matrix `train_loss` of *average* training losses and a 2×2 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, is_training=False):
        '''
        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
        if is_training:
            y[0] *= -0.1
        return X, y

    n_runs = 100
    n_train = 30
    n_test = 1000
    d = 5
    noise = 0.2
    train_loss = np.zeros([n_runs, 2, 2])  # n_runs * n_models * n_metrics
    test_loss = np.zeros([n_runs, 2, 2])  # n_runs * n_models * n_metrics
```

```python
# TODO: Change the following random seed to one of your student IDs
np.random.seed(42)

for r in range(n_runs):

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

    w_L2 = minimizeL2(Xtrain, ytrain)
    w_Linf = minimizeLinf(Xtrain, ytrain)

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

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

# TODO: compute the average losses over runs
# TODO: return a 2-by-2 training loss variable and a 2-by-2 test loss variable
```

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

In the PDF report, show 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_∞ loss
L_2 model		
L_∞ model		

Table 2: Different test losses for different models

Model	L_2 loss	L_∞ loss
L_2 model		
L_∞ model		

Debug Hint: For your verification, we include some toy data in the `toy_data` folder. If you train with the `regression_train.csv` (with augmentation), you should get the following models

$$\mathbf{w}_{L2} = \begin{bmatrix} -26.72481802 \\ -1.1663904 \end{bmatrix} \qquad \mathbf{w}_{L_\infty} = \begin{bmatrix} -25.3365398 \\ -1.39556938 \end{bmatrix}$$

with the following training and test losses

Table 3: Different training losses for different models for `regression_train.csv`

Model	L_2 loss	L_∞ loss
L_2 model	2.00305568	7.31711154
L_∞ model	2.36347873	6.6215357

Table 4: Different test losses for different models for the `regression_test.csv`

Model	L_2 loss	L_∞ loss
L_2 model	1.66071948	5.58874693
L_∞ model	1.9868478	5.02664782

(c.2) (1%) Looking at your tables from above, analyze the results and discuss any findings you may have and the possible reason behind them.

(d) (1.5%) Real-World Regression Problem

(d.1) (1%) Here you will apply the linear regression algorithms to the concrete compressive strength (CCS) dataset (click to see the data website).

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

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

that takes an absolute path (result of `os.path.abspath`) of the CCS `dataset_folder` (where the `Concrete_Data.xls` 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) and (b)).

(d.2) (0.5%) Implement a Python function

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

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

```python
def runCCS(dataset_folder):

    X, y = preprocessCCS(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, 2, 2])  # n_runs * n_models * n_metrics
    test_loss = np.zeros([n_runs, 2, 2])  # n_runs * n_models * n_metrics

    # TODO: Change the following random seed to one of your student IDs
    np.random.seed(42)

    for r in range(n_runs):

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

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

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

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

    # TODO: compute the average losses over runs
    # TODO: return a 2-by-2 training loss variable and a 2-by-2 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 (7.5%) Logistic Regression

In this question, you will implement logistic regression, a classification method, and solve it using `scipy.optimize.minimize`.

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) (2%) Solving Convex Problem

(a.1) (1%) Before diving into logistic regression, let's first revisit the linear regression in Q1(a). Implement two Python functions

```
obj_val = linearRegL2Obj(w, X, y)
gradient = linearRegL2Grad(w, X, y)
```

that take a $d \times 1$ vector of parameters `w`, an $n \times d$ input matrix `X` and an $n \times 1$ label vector `y` as inputs. The first function returns a scalar value `obj_val` that is the objective value in Eq. (1) (i.e., the value of $\frac{1}{2n}\|X\mathbf{w} - \mathbf{y}\|_2^2$). The second function returns a vector `gradient` that 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`.

(a.2) (1%) Write a Python function

```
w = find_opt(obj_func, grad_func, X, y)
```

that find the optimal solution of a convex optimization problem, using the `minimize` from `scipy.optimize`. The function takes an objective function `obj_func` (that admits the I/O as in part (a)), an $n \times d$ input matrix `X` and an $n \times 1$ label vector `y` as inputs. It returns the final parameter vector `w` of size $d \times 1$. Specifically, it should work as follows

```python
def find_opt(obj_func, grad_func, X, y):
    d = X.shape[1]
    w_0 = # TODO: Initialize a random 1-D array of parameters of size d
    # TODO: Define an objective function `func` that takes a single argument (w)
    # TODO: Define a gradient function `gd` that takes a single argument (w)
    return minimize(func, w_0, jac=gd)['x'][:, None]
```

Hint: Although in machine learning, we usually represent vectors as column vectors (for example, we write $\mathbf{w}$ as a $d \times 1$ vector), some libraries work with row vectors (1-D array) by default. In the case of `scipy.optimize.minimize`, the function `func` and the gradient (i.e., Jacobian with only one output) `gd` have to take **a single 1-D array** as input. Since our `obj_func` and `grad_func` require column vectors as inputs, you would need to do some conversion in your `func` and `gd`. This is also the reason why we have `[:, None]` at the end, which converts the 1-D optimal solution to a column vector.

Debug hint: You can pass the objective and gradient functions from Q2(a) to your `find_opt` function:

```
w = find_opt(linearRegL2Obj, linearRegL2Grad, X, y)
```

You should get a 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 `find_opt` function is correct.

(b) (2%) Solving Logistic Regression

Implement two Python functions

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

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

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

where the sigmoid function σ is applied element-wisely. The second one returns a vector `gradient` that is the *analytic form* gradient of size $d \times 1$:

$$\nabla J(\mathbf{w}) = \frac{1}{n} X^\top (\sigma(X\mathbf{w}) - \mathbf{y}). \tag{4}$$

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.

Debug Hint: Once done, you can use `A1testbed.py` to visualize your models and make sure that they are reasonable.

(c) (2%) Synthetic Binary Classification Problem

(c.1) (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 5 and Table 6 below). It repeats 100 runs as follows

```
def synClsExperiments():

    def genData(n_points, dim1, dim2):
        '''
        This function generate synthetic data
        '''
        c0 = np.ones([1, dim1])  # class 0 center
        c1 = -np.ones([1, dim1])  # class 1 center
        X0 = np.random.randn(n_points, dim1 + dim2)  # class 0 input
        X0[:, :dim1] += c0
        X1 = np.random.randn(n_points, dim1 + dim2)  # class 1 input
        X1[:, :dim1] += c1
        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, dim1=2, dim2=2):
        '''
        Run classification experiment with the specified arguments
        '''
```

```python
        n_test = 1000
        Xtrain, ytrain = genData(m, dim1, dim2)
        Xtest, ytest = genData(n_test, dim1, dim2)

        w_logit = find_opt(logisticRegObj, logisticRegGrad, Xtrain, ytrain)
        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 test 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])

    # TODO: Change the following random seed to one of your student IDs
    np.random.seed(42)

    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, dim1 in enumerate((1, 2, 4, 8)):
            train_acc[r, i, 1], test_acc[r, i, 1] = runClsExp(dim1=dim1)
        for i, dim2 in enumerate((1, 2, 4, 8)):
            train_acc[r, i, 2], test_acc[r, i, 2] = runClsExp(dim2=dim2)

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

Table 5: Training accuracies with different hyper-parameters

m	Train Accuracy	dim1	Train Accuracy	dim2	Train Accuracy
10		1		1	
50		2		2	
100		4		4	
200		8		8	

Table 6: Test accuracies with different hyper-parameters

m	Test Accuracy	dim1	Test Accuracy	dim2	Test Accuracy
10		1		1	
50		2		2	
100		4		4	
200		8		8	

Debug Hint: For your verification, we include some toy data in the `toy_data` folder. If you train with the `classification_train.csv` (with augmentation), you should get a model close to $\mathbf{w} = [0.0318, -2.47, -2.45]^\top$ and the training accuracy on `classification_train.csv` and test accuracy on `classification_test.csv` should be 0.93 and 0.925, respectively.

(c.2) (1%) Looking at your tables from above, analyze the results and discuss any findings you may have and the possible reason behind them.

(d) (1.5%) Real-World Binary Classification Problem

(d.1) (1%) Now you will apply logistic regression a real-world problem, the Breast Cancer Wisconsin (BCW) dataset (click to see the data website).

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

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

that takes an absolute path (result of `os.path.abspath`) of the BCW `dataset_folder` (where the `wdbc.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 `find_opt` learning). In this function, you need to remove the ID column and use the correct target variable, converting `B` to 0 and `M` to 1.

(d.2) (0.5%) Implement a Python function

```
train_acc, test_acc = runBCW(dataset_folder)
```

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

```python
def runBCW(dataset_folder):

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

    n_runs = 100
    train_acc = np.zeros([n_runs])
    test_acc = np.zeros([n_runs])

    # TODO: Change the following random seed to one of your student IDs
    np.random.seed(42)

    for r in range(n_runs):

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

        w = find_opt(logisticRegObj, logisticRegGrad, Xtrain, ytrain)

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

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

    # TODO: compute the average accuracies over runs
    # TODO: return two variables: the average training accuracy and average test accuracy
```