COMP9417 - Machine Learning
Homework 1: Gradient Descent & Friends

Introduction In this homework, you will be required to manually implement (Stochastic) Gradient Descent in Python to learn the parameters of a linear regression model. You will make use of the publicly available 'real_estate.csv' dataset, which you can download directly from the course Moodle page. The dataset contains 414 real estate records, each of which contains the following features:

- transactiondate: date of transaction
- age: age of property
- nearestMRT: distance of property to nearest supermarket
- nConvenience: number of convenience stores in nearby locations
- latitude
- longitude

The target variable is the property price. The goal is to learn to predict property prices as a function of a subset of the above features.

Question 1. (Pre-processing)

A number of pre-processing steps are required before we attempt to fit any models to the data.

(a) Remove any rows of the data that contain a missing ('NA') value. List the indices of the removed data points. Then, delete all features from the dataset apart from: age, nearestMRT and nConvenience.

Solution:

The indices of rows with missing values are: $[19, 41, 109, 144, 230, 301]$.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
df = pd.read_csv("real_estate.csv")
df = df.dropna()
X = df[["age", "nearestMRT", "nConvenience"]].values

```

(b) An important pre-processing step: feature normalisation. Feature normalisation involves rescaling the features such that thay all have similar scales. This is also important for algorithms like gradient descent to ensure the convergence of the algorithm. One common technique is called min-max normalisation, in which each feature is scaled to the range $[0, 1]$. To do this, for each feature we

must find the minimum and maximum values in the available sample, and then use the following formula to make the transformation:

$$x_{\text{new}} = \frac{x - \min(x)}{\max(x) - \min(x)}.$$

After applying this normalisation, the minimum value of your feature will be 0, and the maximum will be 1. For each of the features, provide the mean value over your dataset.

Solution:

The mean values for the three features are: $[0.40607933, 0.16264268, 0.4120098]$.

```
            # preprocessing
            from sklearn.preprocessing import MinMaxScaler
            scaler = MinMaxScaler()
            X = scaler.fit_transform(X)
            y = df["price"].values

```

Question 2. (Train and Test sets)

Now that the data is pre-processed, we will create train and test sets to build and then evaluate our models on. Use the first half of observations (in the same order as in the original csv file excluding those you removed in the previous question) to create the training set, and the remaining half for the test set. Print out the first and last rows of both your training and test sets.

Solution:

- first row Xtrain: $[0.73059361, 0.00951267, 1.]$
- last row Xtrain: $[0.87899543, 0.09926012, 0.3]$
- first row Xtest: $[0.26255708, 0.20677973, 0.1]$
- last row Xtest: $[0.14840183, 0.0103754, 0.9]$
- first row ytrain: 37.9
- last row ytrain: 34.2
- first row ytest: 26.2
- last row ytest: 63.9

```
        # train/test splot
        X = np.concatenate((np.ones([len(X), 1]), X), axis=1)
        split_point = X.shape[0] // 2
        X_train = X[:split_point]
        X_test = X[split_point:]
        y_train = y[:split_point]
        y_test = y[split_point:]

        # printing
```

```
        print("first row Xtrain: ", Xtrain[0])
        print("last row Xtrain: ", Xtrain[-1])
        print("first row Xtest: ", Xtest[0])
        print("last row Xtest: ", Xtest[-1])
        print("first row ytrain: ", ytrain[0])
        print("last row ytrain: ", ytrain[-1])
        print("first row ytest: ", ytest[0])
        print("last row ytest: ", ytest[-1])

```

Question 3. (Loss Function)

Consider the loss function

$$\mathcal{L}_c(x, y) = \sqrt{\frac{1}{c^2}(x-y)^2 + 1} - 1,$$

where $c \in \mathbb{R}$ is a hyper-parameter. Consider the (simple) linear model

$$\hat{y}^{(i)} = w_0 + w_1 x_1^{(i)} + w_2 x_2^{(i)} + w_3 x_3^{(i)}, \qquad i = 1, \ldots, n.$$

We can write this more succintly by letting $w = (w_0, w_1, w_2, w_3)^T$ and $X^{(i)} = (1, x_1^{(i)}, x_2^{(i)}, x_3^{(i)})^T$, so that $\hat{y}^{(i)} = w^T X^{(i)}$. The mean-loss achieved by our model (w) on a given dataset of n observations is then

$$\mathcal{L}_c(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}_c(y^{(i)}, \hat{y}^{(i)}) = \frac{1}{n}\sum_{i=1}^{n} \left[\sqrt{\frac{1}{c^2}(y^{(i)} - \langle w^{(t)}, X^{(i)}\rangle)^2 + 1} - 1\right],$$

Compute the following derivatives:

$$\frac{\partial \mathcal{L}_c(y^{(i)}, \hat{y}^{(i)})}{\partial w_k}, \qquad k = 0, 1, 2, 3.$$

You must show your working for full marks.

Solution:

After applying the chain rule twice, we get results similar to the L2 loss cases:

$$\frac{\partial \mathcal{L}_c(y^{(i)}, \hat{y}^{(i)})}{\partial w_k} = \frac{X_k^{(i)}(\langle w^{(t)}, X^{(i)}\rangle - y^{(i)})}{c^2\sqrt{\frac{1}{c^2}(\langle w^{(t)}, X^{(i)}\rangle - y^{(i)})^2 + 1}}, \qquad k = 0, 1, 2, 3.$$

Question 4. (Gradient Descent Psuedocode)

For some value of $c > 0$, we have

$$\frac{\partial \mathcal{L}_c(y^{(i)}, \hat{y}^{(i)})}{\partial w_k} = \frac{x_k^{(i)}(\langle w^{(t)}, X^{(i)}\rangle - y^{(i)})}{2\sqrt{(\langle w^{(t)}, X^{(i)}\rangle - y^{(i)})^2 + 4}}, \quad k = 0, 1, 2, 3.$$

Using this result, write down the gradient descent updates for w_0, w_1, w_2, w_3 (using pseudocode), assuming a step size of η. Note that in gradient descent we consider the loss over the entire dataset, not just at a single observation. Further, provide pseudocode for stochastic gradient descent updates. Here, $w^{(t)}$ denotes the weight vector at the t-th iteration of gradient descent.

Solution:

We have the following updates for GD, for $t = 1, \ldots, T$:

$$w_k^{(t+1)} = w_k^{(t)} - \eta \frac{1}{n} \sum_{i=1}^{n} \frac{X_k^{(i)}(\langle w^{(t)}, X^{(i)} \rangle - y_i)}{2\sqrt{(\langle w^{(t)}, X^{(i)} \rangle - y_i)^2 + 4}}$$

Similarly for SGD, we have updates

$$w_k^{(t+1)} = w_k^{(t)} - \eta \frac{X_k^{(i)}(\langle w^{(t)}, X^{(i)} \rangle - y_i)}{2\sqrt{(\langle w^{(t)}, X^{(i)} \rangle - y_i)^2 + 4}}$$

Question 5. (Gradient Descent Implementation)

In this section, you will implement gradient descent from scratch on the generated dataset using the gradients computed in Question 3, and the pseudocode in Question 4.

(a) Initialise your weight vector to $w^{(0)} = [1, 1, 1, 1]^T$. Consider step sizes

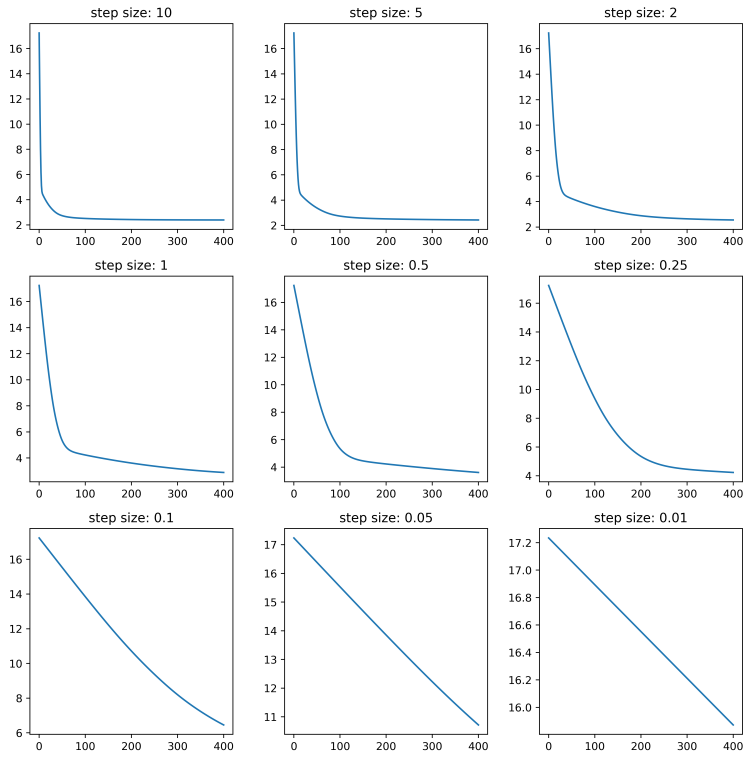
$$\eta \in \{10, 5, 2, 1, 0.5, 0.25, 0.1, 0.05, 0.01\}$$

(a total of 9 step sizes). For each step-size, generate a plot of the loss achieved at each iteration of gradient descent. You should use 400 iterations in total (which will require you to loop over your training data in order). Generate a 3×3 grid of plots showing performance for each step-size. Add a screen shot of the python code written for this question in your report (as well as pasting the code in to your solutions.py file). The following code may help with plotting:

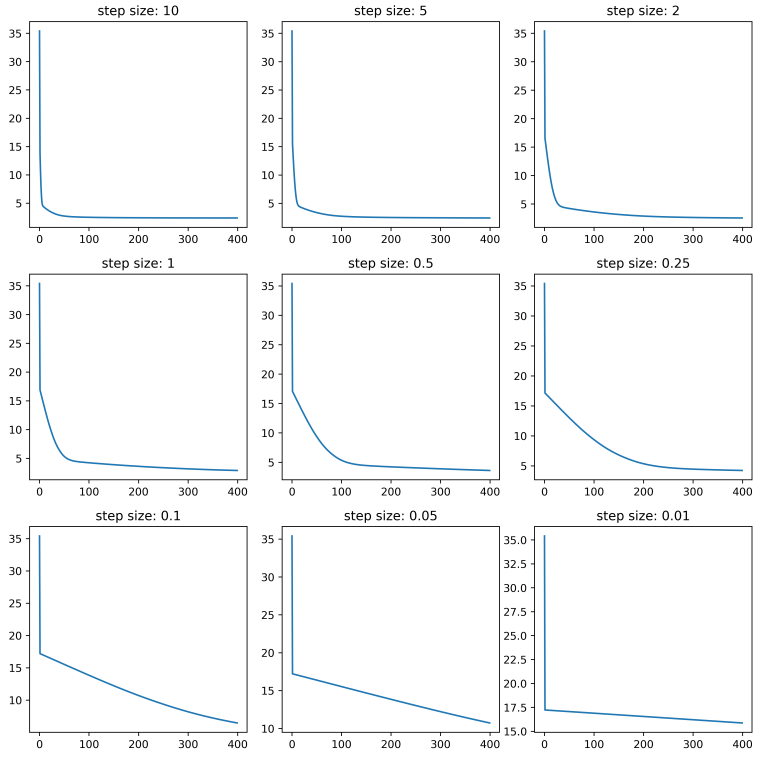
```
        fig, ax = plt.subplots(3,3, figsize=(10,10))
        nIter = 400
        alphas = [10,5,2, 1,0.5, 0.25,0.1, 0.05, 0.01]
        for i, ax in enumerate(ax.flat):
            # losses is a list of 9 elements. Each element is an array of length nIter
    storing the loss at each iteration for
            # that particular step size
            ax.plot(losses[i])
            ax.set_title(f"step size: {alphas[i]}")  # plot titles
        plt.tight_layout()      # plot formatting
        plt.show()

```

Solution:

The plot should look like:



If the student did not figure out that the provided gradient corresponds to the case $c = 2$, then they were told they could use either $c = 1$ or $c = 2$. This affects their loss values, even though they are using the correct gradients, and in that case their plots would look like:



Example code is as follows:

```
        def calc_loss(y, y_pred, c=2):
            return np.mean(np.sqrt((1/c**2) * (y - y_pred)**2 + 1) - 1)

        def calc_grad(X_train, y_train, w, c=2):
            Xw = X_train @ w
            const = (Xw - y_train) / (c**2 * np.sqrt((1/c**2) * (Xw - y_train)**2 + 1)
    )
            grad = np.mean(X_train * np.repeat(const, 4).reshape(-1, 4), axis=0)
            return grad

        nmb_iter = 400
        w = np.zeros(shape=(nmb_iter+1, 4))
        w[0] = np.array([1,1,1,1])

        alphas = [10,5,2,1,0.5,0.25,0.1,0.05, 0.01]
        losses = []
        cur_c = 2
        for alpha in alphas:
            train_loss = np.zeros(shape=(nmb_iter+1))
            train_loss[0] = calc_loss(ytrain, Xtrain @ w[0], c=cur_c)
            for i in range(1, nmb_iter+1):
                w[i] = w[i-1] - alpha * calc_grad(Xtrain, ytrain, w[i-1])
                train_loss[i] = calc_loss(ytrain, Xtrain @ w[i], c=cur_c)
            losses.append(train_loss)

```

```
        fig, ax = plt.subplots(3,3, figsize=(10,10))
        for i, ax in enumerate(ax.flat):
            ax.plot(losses[i])
            ax.set_title(f"step size: {alphas[i]}")

        plt.tight_layout()
        plt.savefig("GDgrid.png", dpi=500)
        plt.show()
```

(b) From your results in the previous part, choose an appropriate step size (and state your choice), and explain why you made this choice.

Solution:
$\eta = 10$ seems like the best choice.

(c) For this part, take $\eta = 0.3$, re-run GD under the same parameter initilisations and number of iterations. On a single plot, plot the progression of each of the four weights over the iterations. Print out the final weight vector. Finally, run your model on the train and test set, and print the achieved losses.

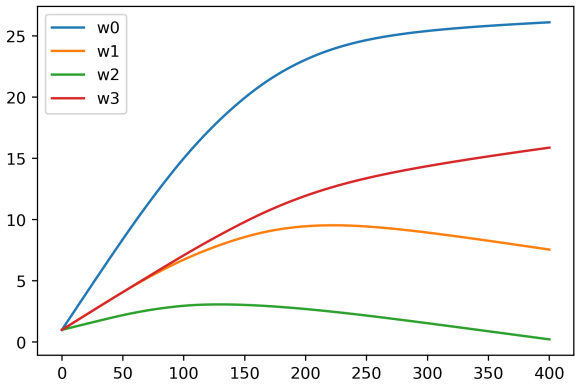
Solution:
We have

$$w = [26.11838076, 7.55495429, 0.22091267, 15.88216107]$$

Rerunning this model gives the following results:

- train loss: 4.089850739201336 / 8.887715476132039 (if $c = 1$)
- test loss: 3.8275009292188362 / 8.351707785988028 (if $c = 1$)

The plot of the weights look like:



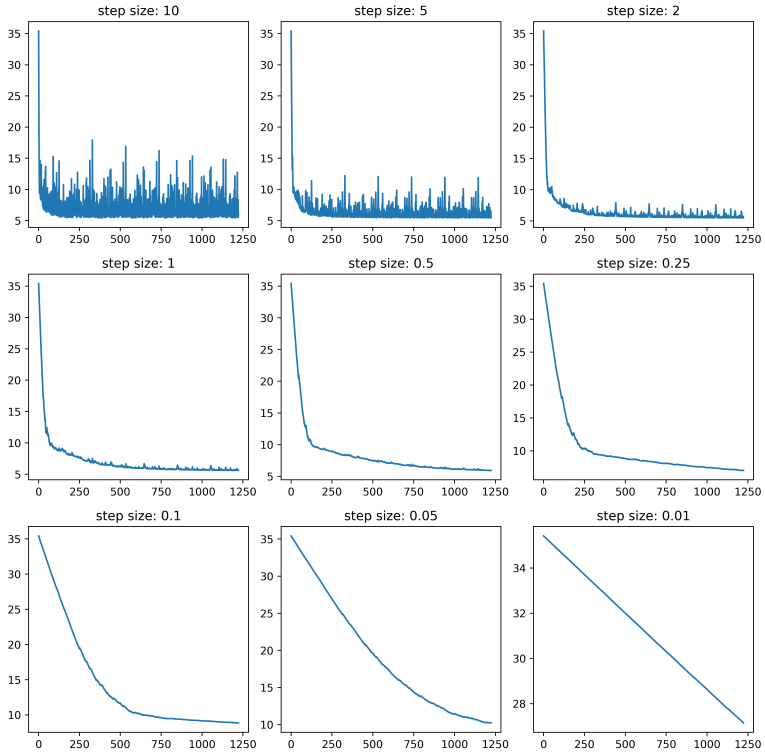
```
w = np.zeros(shape=(nmb_iter,4))
w[0] = np.array([1,1,1,1])

alpha = 0.3
for i in range(1, nmb_iter):
    w[i] = w[i-1] - alpha * calc_grad(Xtrain, ytrain, w[i-1])

plt.plot(w[:,0], label="w0")
plt.plot(w[:,1], label="w1")
plt.plot(w[:,2], label="w2")
plt.plot(w[:,3], label="w3")
plt.legend()
plt.savefig("GDweights.png", dpi=400)
plt.show()

wstar = w[nmb_iter-1]
print("train loss: ", calc_loss(ytrain, Xtrain @ wstar, c=cur_c))
print("test loss: ", calc_loss(ytest, Xtest @ wstar, c=cur_c))
```

Question 6. (Stochastic Gradient Descent Implementation)

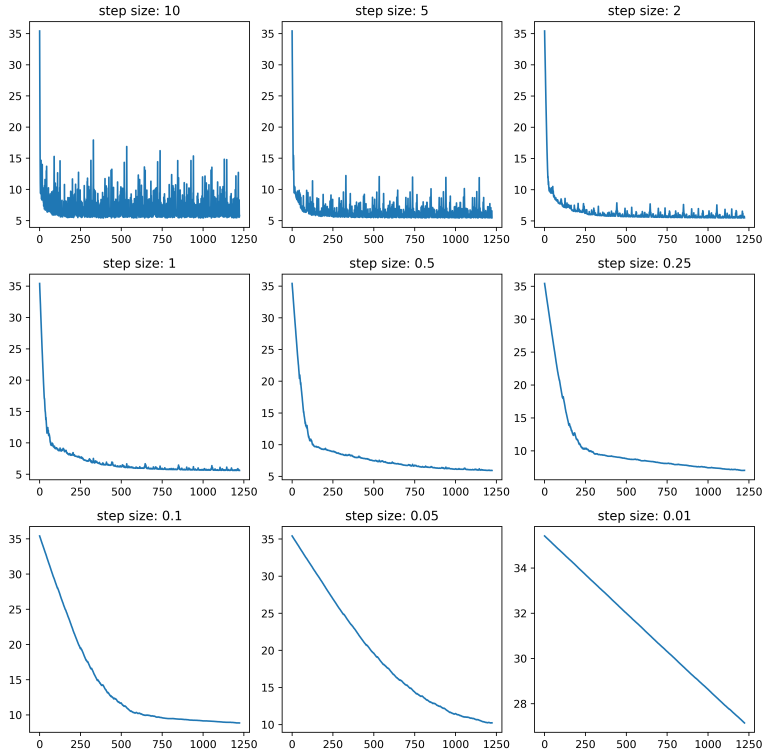
We will now re-run the analysis in question 5, but using stochastic gradient descent (SGD) instead. In SGD, we update the weights after seeing a single observation.

(a) Use the same settings as in 5 (a). For each step-size, generate a plot of the loss achieved at each iteration of stochastic gradient descent, which is to be run for 6 Epochs. An epoch is one pass over the entire training dataset, so in total there should be $6 \times n_{\text{train}}$ updates, where n_{train} is the size of your training data. **Be sure to run these updates in the same ordering that the data is stored.** Generate a 3×3 grid of plots showing performance for each step-size. Add a screen shot of the python code written for this question in your report (as well as pasting the code in to your solutions.py file).

Solution:
The plot should look like:



If students use $c = 1$, then



Example code is as follows:

```
        def calc_grad_i(w, Xi, yi, c=2):
            # compute gradient for a single observation
            xw = Xi@w
            num = xw-yi
            den = (c**2) * np.sqrt(1 + (1/c**2) * (xw-yi)**2)

            return (num/den)*Xi

        epochs = 6
        nmb_iter = Xtrain.shape[0] * epochs
        w = np.zeros(shape=(nmb_iter,4))
        w[0] = np.array([1,1,1,1])

        alphas = [10, 5, 2, 1, 0.5, 0.25, 0.1, 0.05, 0.01]
        losses = []
        cur_c = 2
        for alpha in alphas:
            train_loss = np.zeros(shape=(nmb_iter))
            train_loss[0] = calc_loss(ytrain, Xtrain @ w[0], c=cur_c)
            for i in range(1, nmb_iter):
                idx = i % Xtrain.shape[0]
                w[i] = w[i-1] - alpha * calc_grad_i(w[i-1], Xtrain[idx], ytrain[idx],
    c=cur_c)
                train_loss[i] = calc_loss(ytrain, Xtrain @ w[i], c=cur_c)
            losses.append(train_loss)
```

```
        fig, ax = plt.subplots(3,3, figsize=(10,10))
        for i, ax in enumerate(ax.flat):
            ax.plot(losses[i])
            ax.set_title(f"step size: {alphas[i]}")

        plt.tight_layout()
        plt.savefig("SGDgrid.png", dpi=500)
        plt.show()
```

(b) From your results in the previous part, choose an appropriate step size (and state your choice), and explain why you made this choice.

Solution:
0.5 seems like a reasonable choice, but 0.25 is also acceptable.

(c) Take $\eta = 0.4$ and re-run SGD under the same parameter initilisations and number of epochs. On a single plot, plot the progression of each of the four weights over the iterations. Print yoru final model. Finally, run your model on the train and test set, and record the achieved losses.

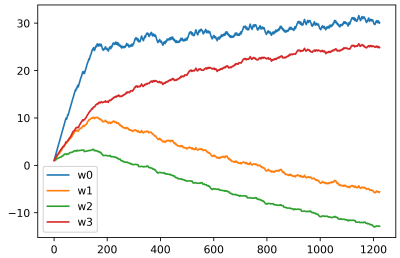
Solution:
We have

$$w = [30.13933686, -5.58250367, -12.85536393, 24.85002152]$$

Rerunning this model gives the following results:

- train loss: 2.7805130067939547 / 6.181134362027168 (if $c = 1$)
- test loss: 2.8446116656442526 / 6.329857966449784 (if $c = 1$)

The plot of the weights look like:




```
            w = np.zeros(shape=(nmb_iter,4))
            w[0] = np.array([1,1,1,1])
```

```
            alpha = .4
            loss_vec = np.zeros(shape=(nmb_iter))
            loss_vec[0] = calc_loss(ytrain, Xtrain @ w[0], c=cur_c)
            for i in range(1, nmb_iter):
                idx = i % Xtrain.shape[0]
                w[i] = w[i-1] - alpha * calc_grad_i(w[i-1], Xtrain[idx], ytrain[idx],
    c=cur_c)
                loss_vec[i] = calc_loss(ytrain, Xtrain @ w[i], c=cur_c)

            plt.plot(w[:,0], label="w0")
            plt.plot(w[:,1], label="w1")
            plt.plot(w[:,2], label="w2")
            plt.plot(w[:,3], label="w3")
            plt.legend()
            plt.savefig("SGDweights.png", dpi=500)
            plt.show()

            wstar = w[-1]

            print("w: ", wstar)
            print("train loss: ", calc_loss(Xtrain@wstar, ytrain, c=cur_c))
            print("test loss: ", calc_loss(Xtest@wstar, ytest, c=cur_c))

```

Update: for both Questions 5 and 6, if you have been unable to identify the value of c used in question 4, you may take $c = 1$ or $c = 2$ when computing the loss values. Please be sure to state clearly which value you have chosen.

Question 7. Results Analysis

In a few lines, comment on your results in Questions 5 and 6. Explain the importance of the step-size in both GD and SGD. Explain why one might have a preference for GD or SGD. Explain why the GD paths look much smoother than the SGD paths.

Solution:

This is an open question so any reasonable short answer should be given full marks. Students should mention things along the lines of:

- In general, Gradient descent takes steps towards the minimum by moving in the direction of greatest descent. For large step sizes, we may overshoot the minimum and this may result in oscillating or divergence of the algorithm, whereas too small a stepsize will lead to very slow convergence. We will in general find that GD uses a larger step size relative to SGD, since the direction (gradient) is more reliable in GD than in SGD.
- SGD is an approximation of GD, since in GD we use the entire training data before making a single update, whereas in SGD we make updates after seeing a single training point. The reason we might prefer SGD is when it is too computationally expensive to compute the gradient over the entire dataset, and SGD tends to converge much faster than GD for this reason. We see that in this case, SGD is able to achieve a lower mean loss, though we cannot directly compare the two algorithms here as the number of iterations differs, it would be interesting if a student provides a timing comparison for the two cases but not necessary.

- The SGD paths are much more turbulent because we are estimating the gradient by looking at a single training point, and so the gradient updates are much noisier than when computing the gradient over the entire training set as in standard GD.
- Any other discussion points, or mention of mini-batch GD, or other variants.

Points Allocation There are a total of 10 marks, the available marks are:

- Question 1: 1 mark (split 0.5/0.5)
- Question 2: 0.5 marks
- Question 3: 1.5 marks
- Question 4: 1 mark
- Question 5: 2.5 marks (split 1/0.5/1)
- Question 6: 2.5 marks (split 1/0.5/1)
- Question 7: 1 mark

What to Submit

- A single PDF file which contains solutions to each question. For each question, provide your solution in the form of text and requested plots. For any question in which you use code, provide a copy of your code at the bottom of the relevant section.
- **.py file(s) containing all code you used for the project, which should be provided in a seperate .zip file.** This code must match the code provided in the report.
- You may be deducted points for not following these instructions.
- You may be deducted points for poorly presented/formatted work. Please be neat and make your solutions clear.

When and Where to Submit

- Due date: Monday **8 March**, 2021 by **5:00pm**.
- Late submission incur a penalty of 10% per day for the first 5 days and 100% for more than 5 days.
- Submission has to be done through Moodle.

Final Reminder: You are required to submit one PDF file, AND also to submit the Python file(s) with all your code as a separate .zip file.