COMP9417 - Machine Learning
Homework 1: Regularized Regression & Numerical Optimization

Introduction In this homework we will explore some algorithms for *gradient* based optimization. These algorithms have been crucial to the development of machine learning in the last few decades. The most famous example is the backpropagation algorithm used in deep learning, which is in fact just an application of a simple algorithm known as (stochastic) gradient descent. We will first implement gradient descent from scratch on a deterministic problem (no data), and then extend our implementation to solve a real world regression problem.
Points Allocation There are a total of 28 marks.

- Question 1 a): 2 marks
- Question 1 b): 1 mark
- Question 1 c): 4 marks
- Question 1 d): 1 mark
- Question 1 e): 1 mark
- Question 1 f): 2 marks
- Question 1 g): 3 marks
- Question 1 h): 3 marks
- Question 1 i): 1 mark
- Question 1 j): 4 marks
- Question 1 k): 5 marks
- Question 1 l): 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 some questions you will be requested to provide screen shots of code used to generate your answer — only include these when they are explicitly asked for.
- **.py file(s) containing all code you used for the project, which should be provided in a separate .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. Start each question on a new page if necessary.
- You **cannot** submit a Jupyter notebook; this will receive a mark of zero. This does not stop you from developing your code in a notebook and then copying it into a .py file though, or using a tool such as **nbconvert** or similar.
- We will set up a Moodle forum for questions about this homework. Please read the existing questions before posting new questions. Please do some basic research online before posting questions. Please only post clarification questions. Any questions deemed to be *fishing* for answers will be ignored and/or deleted.
- Please check Moodle announcements for updates to this spec. It is your responsibility to check for announcements about the spec.
- Please complete your homework on your own, do not discuss your solution with other people in the course. General discussion of the problems is fine, but you must write out your own solution and acknowledge if you discussed any of the problems in your submission (including their name(s) and zID).
- As usual, we monitor all online forums such as Chegg, StackExchange, etc. Posting homework questions on these site is equivalent to plagiarism and will result in a case of academic misconduct.
- You may **not** use SymPy or any other symbolic programming toolkits to answer the derivation questions. This will result in an automatic grade of zero for the relevant question. You must do the derivations manually.

When and Where to Submit

- Due date: Week 4, Monday **June 20th**, 2022 by **5pm**. Please note that the forum will not be actively monitored on weekends.
- Late submissions will incur a penalty of 5% per day **from the maximum achievable grade**. For example, if you achieve a grade of 80/100 but you submitted 3 days late, then your final grade will be $80 - 3 \times 5 = 65$. Submissions that are more than 5 days late will receive a mark of zero.
- Submission must be done through Moodle, no exceptions.

Question 1. Gradient Based Optimization

The general framework for a gradient method for finding a minimizer of a function $f : \mathbb{R}^n \to \mathbb{R}$ is defined by

$$x^{(k+1)} = x^{(k)} - \alpha_k \nabla f(x_k), \qquad k = 0, 1, 2, \ldots, \tag{1}$$

where $\alpha_k > 0$ is known as the step size, or learning rate. Consider the following simple example of minimizing the function $g(x) = 2\sqrt{x^3 + 1}$. We first note that $g'(x) = 3x^2(x^3 + 1)^{-1/2}$. We then need to choose a starting value of x, say $x^{(0)} = 1$. Let's also take the step size to be constant, $\alpha_k = \alpha = 0.1$. Then we have the following iterations:

$$\begin{aligned}
x^{(1)} &= x^{(0)} - 0.1 \times 3(x^{(0)})^2((x^{(0)})^3 + 1)^{-1/2} = 0.7878679656440357 \\
x^{(2)} &= x^{(1)} - 0.1 \times 3(x^{(1)})^2((x^{(1)})^3 + 1)^{-1/2} = 0.6352617090300827 \\
x^{(3)} &= 0.5272505146487477 \\
&\vdots
\end{aligned}$$

and this continues until we terminate the algorithm (as a quick exercise for your own benefit, code this up and compare it to the true minimum of the function which is $x_* = -1$). This idea works for functions that have vector valued inputs, which is often the case in machine learning. For example, when we minimize a loss function we do so with respect to a weight vector, β. When we take the step-size to be constant at each iteration, this algorithm is known as gradient descent. For the entirety of this question, **do not use any existing implementations of gradient methods, doing so will result in an automatic mark of zero for the entire question.**

(a) Consider the following optimisation problem:

$$\min_{x \in \mathbb{R}^n} f(x),$$

where

$$f(x) = \frac{1}{2}\|Ax - b\|_2^2 + \frac{\gamma}{2}\|x\|_2^2,$$

and where $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$ are defined as

$$A = \begin{bmatrix} 1 & 2 & 1 & -1 \\ -1 & 1 & 0 & 2 \\ 0 & -1 & -2 & 1 \end{bmatrix}, \qquad b = \begin{bmatrix} 3 \\ 2 \\ -2 \end{bmatrix},$$

and γ is a positive constant. Run gradient descent on f using a step size of $\alpha = 0.1$ and $\gamma = 0.2$ and starting point of $x^{(0)} = (1, 1, 1, 1)$. You will need to terminate the algorithm when the following condition is met: $\|\nabla f(x^{(k)})\|_2 < 0.001$. In your answer, clearly write down the version of the gradient steps (1) for this problem. Also, print out the first 5 and last 5 values of $x^{(k)}$, clearly indicating the value of k, in the form:

$$\begin{aligned}
k = 0, &\qquad x^{(k)} = [1, 1, 1, 1] \\
k = 1, &\qquad x^{(k)} = \cdots \\
k = 2, &\qquad x^{(k)} = \cdots \\
&\vdots
\end{aligned}$$

What to submit: an equation outlining the explicit gradient update, a print out of the first 5 ($k = 5$ inclusive) and last 5 rows of your iterations. Use the round function to round your numbers to 4 decimal places. Include a screen shot of any code used for this section and a copy of your python code in solutions.py.

Solution:
The gradient is $\nabla f(x) = A^T Ax - A^T b + \gamma x$, so the steps are

$$\begin{aligned} x^{(k+1)} &= x^{(k)} - \alpha_k (A^T A x^{(k)} - A^T b + \gamma x^{(k)}) \\ &= x^{(k)} - \alpha_k ((A^T A + \gamma I) x^{(k)} - A^T b), \qquad k = 0, 1, 2, \ldots. \end{aligned}$$

The first 5 steps are:

$$\begin{aligned} k = 0, &\quad x_k = [1, 1, 1, 1] \\ k = 1, &\quad x_k = [0.98, 0.98, 0.98, 0.98] \\ k = 2, &\quad x_k = [0.9624, 0.9804, 0.9744, 0.9584] \\ k = 3, &\quad x_k = [0.9427, 0.9824, 0.9668, 0.9433] \\ k = 4, &\quad x_k = [0.9234, 0.9866, 0.9598, 0.9295] \\ k = 5, &\quad x_k = [0.9044, 0.9916, 0.9526, 0.9169] \end{aligned}$$

The last 5 steps are:

$$\begin{aligned} k = 271, &\quad x_k = [0.0667, 1.3365, 0.4929, 0.3251] \\ k = 272, &\quad x_k = [0.0666, 1.3366, 0.4928, 0.3251] \\ k = 273, &\quad x_k = [0.0666, 1.3366, 0.4928, 0.325] \\ k = 274, &\quad x_k = [0.0665, 1.3366, 0.4927, 0.325] \\ k = 275, &\quad x_k = [0.0664, 1.3367, 0.4927, 0.3249]. \end{aligned}$$

So the final value is $[0.0664, 1.3367, 0.4927, 0.3249]$. The code used is:

```
A = np.array([[1,2,1,-1], [-1,1,0,2], [0,-1,-2,1]])
b = np.array([3,2,-2])
gamma = 0.2
alpha = 0.1
tol = 10**-3

def grad_f(x, gamma):
    return A.T @ (A@x - b) + gamma * x

xk = np.array([1,1,1,1])
k = 0
while np.linalg.norm(grad_f(xk, gamma)) > tol:
    print(f'k &= {k}, \quad x_k= [{round(xk[0],4)}, {round(xk[1],4)}, {round(xk[2],4)
    }, {round(xk[3],4)} ]\\\\')
    xk = xk - alpha * grad_f(xk, gamma)
    k += 1
```

(b) In the previous part, we used the termination condition $\|\nabla f(x^{(k)})\|_2 < 0.001$. What do you think this condition means in terms of convergence of the algorithm to a minimizer of f ? How would

making the right hand side smaller (say 0.0001) instead, change the output of the algorithm? Explain.
What to submit: some commentary.

Solution:
Since we are aiming to find a minimizer f, we know that a minimizer $\hat{x}$ should satisfy $\nabla f(\hat{x}) \approx 0 \in \mathbb{R}^n$ (here zero denotes the zero vector in $\mathbb{R}^n$). So the condition $\|\nabla f(x^{(k)})\|_2$ checks if the gradient of f at the current iteration is sufficiently small. Since this is a numerical routine, checking if it the gradient is exactly zero will result in our algorithm never converging. The value 0.001 is called the tolerance, and we can make the tolerance smaller if we want a more accurate solution, at the cost of a higher number of iterations of course.

(c) In lab 2, we introduced PyTorch and discussed how to use it to perform gradient descent. In this question you will replicate your previous analysis in part (a) but using PyTorch instead. As in part (a), clearly write down the version of the gradient steps (1) for this problem. Also, print out the first 5 and last 5 values of $x^{(k)}$, clearly indicating the value of k. You may use the following code as a template if you find it helpful. Note that you may not make any calls to NumPy here.

```
import torch
import torch.nn as nn
from torch import optim

A = ###
b = ###
tol = ###
gamma = 0.2
alpha = 0.1

class MyModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.x = ####

    def forward(self, ###):
        return ###

model = MyModel()
optimizer = ###
terminationCond = False
k = 0
while not terminationCond:
    ### compute loss, find gradients, update, check termination cond. etc

```

What to submit: a print out of the first 5 ($k = 5$ inclusive) and last 5 rows of your iterations. Use the round function to round your numbers to 4 decimal places. Include a screen shot of any code used for this section and a copy of your python code in solutions.py.

Solution:
results should be identical to (a) here. The code used is:

```
    import torch
    import torch.nn as nn
    from torch import optim
```

```
A = torch.Tensor([[1,2,1,-1], [-1,1,0,2], [0,-1,-2,1]])
b = torch.Tensor([3,2,-2])
tol = 10**-3
gamma = 0.2
alpha = 0.1

class MyModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.x = nn.Parameter(torch.ones(4, requires_grad=True))

    def forward(self, A):
        return A @ self.x

model = MyModel()
optimizer = optim.SGD(model.parameters(), lr=alpha)
terminationCond = False
k = 0
while not terminationCond:
    print(f'k={k}, x_k = {model.x.data}')
    yhat = model.forward(A)
    loss = 0.5 * torch.linalg.norm(yhat - b, ord=2)**2
    loss += gamma * 0.5 * torch.linalg.norm(model.x, ord=2)**2
    loss.backward()
    optimizer.step()
    if torch.linalg.norm(model.x.grad) <= tol:
        terminationCond = True
    optimizer.zero_grad()
    k+=1
```

In the next few parts, we will use gradient methods explored above to solve a real machine learning problem. Consider the CarSeats data provided in `CarSeats.csv`. It contains 400 observations with each observation describing child car seats for sale at one of 400 stores. The features in the data set are outlined below:

- Sales: Unit sales (in thousands) at each location
- CompPrice: Price charged by competitor at each location
- Income: Local income level (in thousands of dollars)
- Advertising: advertising budget (in thousands of dollars)
- Population: local population size (in thousands)
- Price: price charged by store at each site
- ShelveLoc: A categorical variable with Bad, Good and Medium describing the quality of the shelf location of the car seat
- Age: Average age of the local population
- Education: Education level at each location
- Urban A categorical variable with levels No and Yes to describe whether the store is in an urban location or in a rural one
- US: A categorical variable with levels No and Yes to describe whether the store is in the US or not.

The target variable is Sales. The goal is to learn to predict the amount of Sales as a function of a subset of the above features. We will do so by running Ridge Regression (Ridge) which is defined as follows

$$\hat{\beta}_{\text{Ridge}} = \arg\min_{\beta} \frac{1}{n}\|y - X\beta\|_2^2 + \phi\|\beta\|_2^2,$$

where $\beta \in \mathbb{R}^p$, $X \in \mathbb{R}^{n\times p}$, $y \in \mathbb{R}^n$ and $\phi > 0$.

(d) We first need to preprocess the data. Remove all categorical features. Then use `sklearn.preprocessing.StandardScaler` to standardize the remaining features. Print out the mean and variance of each of the standardized features. Next, center the target variable (subtract its mean). Finally, create a training set from the first half of the resulting dataset, and a test set from the remaining half and call these objects X_train, X_test, Y_train and Y_test. Print out the first and last rows of each of these.

What to submit: a print out of the means and variances of features, a print out of the first and last rows of the 4 requested objects, and some commentary. Include a screen shot of any code used for this section and a copy of your python code in solutions.py.

Solution:

Feature Means should all be zero (or very very small numbers), and variances should all be 1.

- first row X_train: [0.85045499, 0.15536099, 0.65717702, 0.07581929, 0.17782345,-0.69978222, 1.18444912]
- last row X_train: [-0.1942498 , 0.6920138 ,-0.24615909, 0.47665602, 0.43155489, 0.65991828, 0.03820804]
- first row X_test: [1.24221929, 0.83512122,-0.99893918, 0.57176982, 1.27732635, 0.53630914, -0.72595268]
- last row X_test: [0.58927879,-1.13260576,-0.99893918,-1.61584759, 0.17782345,-0.26715025, 0.80236876]
- first row Y_train: 2.003675
- last row Y_train: -1.076325
- first row Y_test: -1.936325
- last row Y_test: 2.213675

```
import pandas as pd
from sklearn.preprocessing import StandardScaler

df = pd.read_csv("CarSeats.csv")
df = df.drop(columns=['ShelveLoc', 'Urban', 'US'])
X = df.iloc[:,1:]
Y = df.iloc[:,0]
Y = Y - Y.mean()
scaler = StandardScaler().fit(X)
scaled_X = scaler.transform(X)
```

```
    print(f"Feature Means: {scaled_X.mean(axis=0)}")                # all effectively
    equal to 0
    print(f"Feature Variances: {scaled_X.var(axis=0)}")             # all effectively
    equal to 1

    split_point = scaled_X.shape[0] // 2
    X_train = scaled_X[:split_point]
    X_test = scaled_X[split_point:]
    Y_train = Y[:split_point].to_numpy()
    Y_test = Y[split_point:].to_numpy()

    print('\item first row X\_train: ', np.array2string(X_train[0], separator=','))
    print('\item last row X\_train: ', np.array2string(X_train[-1], separator=','))
    print('\item first row X\_test: ', np.array2string(X_test[0], separator=','))
    print('\item last row X\_test: ', np.array2string(X_test[-1], separator=','))
    print('\item first row Y\_train: ', np.array2string(Y_train[0], separator=','))
    print('\item last row Y\_train: ', np.array2string(Y_train[-1], separator=','))
    print('\item first row Y\_test: ', np.array2string(Y_test[0], separator=','))
    print('\item last row Y\_test: ', np.array2string(Y_test[-1], separator=','))

```

(e) It should be obvious that a closed form expression for $\hat{\beta}_{\text{Ridge}}$ exists. Write down the closed form expression, and compute the exact numerical value on the training dataset with $\phi = 0.5$.
What to submit: Your working, and a print out of the value of the ridge solution based on (X_train, Y_train). Include a screen shot of any code used for this section and a copy of your python code in solutions.py.

Solution:
An identical calculation to the one in part (a) shows that

$$\hat{\beta}_{\text{Ridge}} = ((\frac{1}{n})X^TX + \phi I)^{-1}(\frac{1}{n})X^Ty = (X^TX + n\phi I)^{-1}X^Ty.$$

The exact value of on the training set for this problem is given by:

$$[0.680673, 0.28229334, 0.65157017, 0.00834835, -1.17129533, -0.400892, -0.10063355].$$

Students might use the sklearn implementation in which case they get:

$$[0.68003143, 0.28365092, 0.64997492, 0.00772555, -1.17165944, -0.39999182, -0.09961697].$$

```
n = X_train.shape[0]
p = X_train.shape[1]
phi = 0.5
betaRidge = (1/n)*np.linalg.inv( (1/n) *X_train.T @ X_train+ phi * np.eye(p)) @
    X_train.T @ Y_train
print(betaRidge)

# using sklearn
ridgeModwInt = Ridge(alpha=0.5*n, fit_intercept=True).fit(X_train,Y_train)
ridgeModwInt.coef_

```

We will now solve the ridge problem but using numerical techniques. As noted in the lectures, there are a few variants of gradient descent that we will briefly outline here. Recall that in gradient descent our update rule is

$$\beta^{(k+1)} = \beta^{(k)} - \alpha_k \nabla L(\beta^{(k)}), \qquad k = 0, 1, 2, \ldots,$$

where $L(\beta)$ is the loss function that we are trying to minimize. In machine learning, it is often the case that the loss function takes the form

$$L(\beta) = \frac{1}{n} \sum_{i=1}^{n} L_i(\beta),$$

i.e. the loss is an average of n functions that we have lablled L_i. It then follows that the gradient is also an average of the form

$$\nabla L(\beta) = \frac{1}{n} \sum_{i=1}^{n} \nabla L_i(\beta).$$

We can now define some popular variants of gradient descent .

(i) Gradient Descent (GD) (also referred to as batch gradient descent): here we use the full gradient, as in we take the average over all n terms, so our update rule is:

$$\beta^{(k+1)} = \beta^{(k)} - \frac{\alpha_k}{n} \sum_{i=1}^{n} \nabla L_i(\beta^{(k)}), \qquad k = 0, 1, 2, \ldots.$$

(ii) Stochastic Gradient Descent (SGD): instead of considering all n terms, at the k-th step we choose an index i_k randomly from $\{1, \ldots, n\}$, and update

$$\beta^{(k+1)} = \beta^{(k)} - \alpha_k \nabla L_{i_k}(\beta^{(k)}), \qquad k = 0, 1, 2, \ldots.$$

Here, we are approximating the full gradient $\nabla L(\beta)$ using $\nabla L_{i_k}(\beta)$.

(iii) Mini-Batch Gradient Descent: GD (using all terms) and SGD (using a single term) represents the two possible extremes. In mini-batch GD we choose batches of size $1 < B < n$ randomly at each step, call their indices $\{i_{k_1}, i_{k_2}, \ldots, i_{k_B}\}$, and then we update

$$\beta^{(k+1)} = \beta^{(k)} - \frac{\alpha_k}{B} \sum_{j=1}^{B} \nabla L_{i_j}(\beta^{(k)}), \qquad k = 0, 1, 2, \ldots,$$

so we are still approximating the full gradient but using more than a single element as is done in SGD.

(f) The ridge regression loss is

$$L(\beta) = \frac{1}{n} \|y - X\beta\|_2^2 + \phi \|\beta\|_2^2.$$

Show that we can write

$$L(\beta) = \frac{1}{n} \sum_{i=1}^{n} L_i(\beta),$$

and identify the functions $L_1(\beta), \ldots, L_n(\beta)$. Further, compute the gradients $\nabla L_1(\beta), \ldots, \nabla L_n(\beta)$
What to submit: your working.

Solution:

$$\begin{aligned} L(\beta) &= \frac{1}{n}\|y - X\beta\|_2^2 + \phi\|\beta\|_2^2 \\ &= \frac{1}{n}\sum_{i=1}^{n}(y_i - x_i^T\beta)^2 + \phi\|\beta\|_2^2 \\ &= \frac{1}{n}\sum_{i=1}^{n}[(y_i - x_i^T\beta)^2 + \phi\|\beta\|_2^2], \end{aligned}$$

so that

$$L_i(\beta) := (y_i - x_i^T\beta)^2 + \phi\|\beta\|_2^2.$$

We also get that

$$\nabla L_i(\beta) := -2x_i(y_i - x_i^T\beta) + 2\phi\beta.$$

(g) In this question, you will implement (batch) GD from scratch to solve the ridge regression problem. Use an initial estimate $\beta^{(0)} = 1_p$ (the vector of ones), and $\phi = 0.5$ and run the algorithm for 1000 epochs (an epoch is one pass over the entire data, so a single GD step). Repeat this for the following step sizes:

$$\alpha \in \{0.000001, 0.000005, 0.00001, 0.00005, 0.0001, 0.0005, 0.001, 0.005, 0.01\}$$

To monitor the performance of the algorithm, we will plot the value

$$\Delta^{(k)} = L(\beta^{(k)}) - L(\hat{\beta}),$$

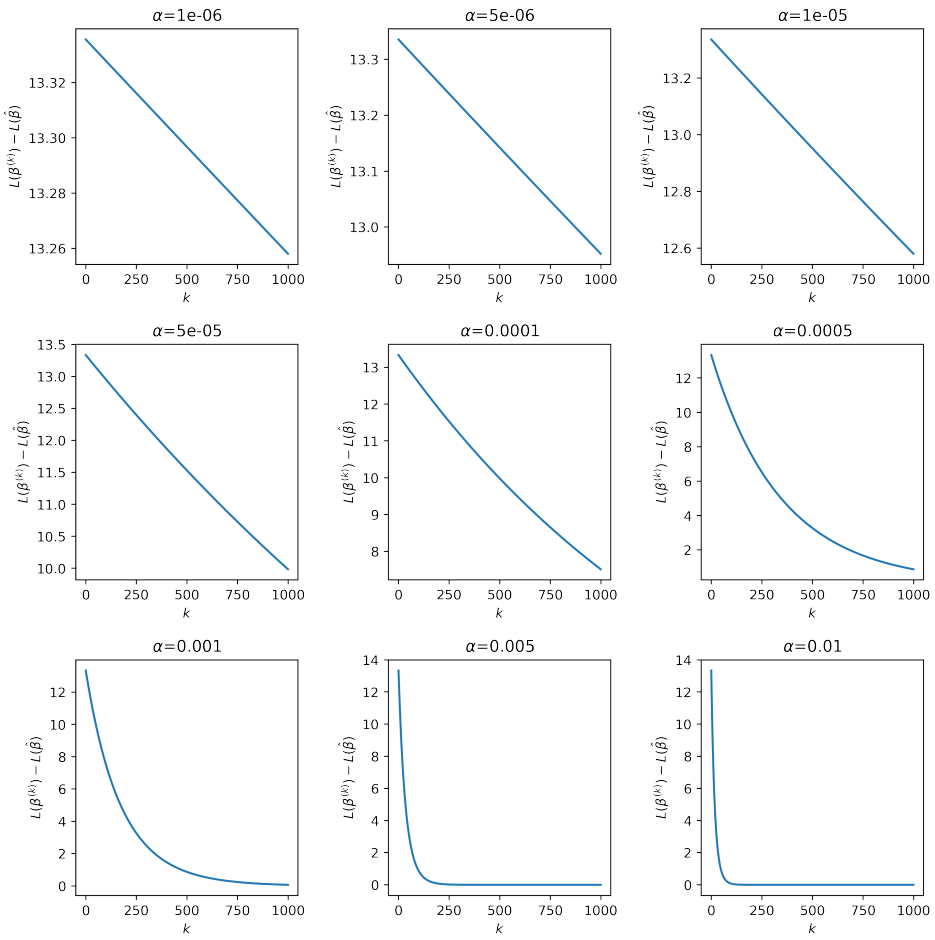
where $\hat{\beta}$ is the true (closed form) ridge solution derived earlier. Present your results in a 3×3 grid plot, with each subplot showing the progression of $\Delta^{(k)}$ when running GD with a specific step-size. State which step-size you think is best and let $\beta^{(K)}$ denote the estimator achieved when running GD with that choice of step size. Report the following:

(i) The train MSE: $\frac{1}{n}\|y_{\text{train}} - X_{\text{train}}\beta^{(K)}\|_2^2$

(ii) The test MSE: $\frac{1}{n}\|y_{\text{test}} - X_{\text{test}}\beta^{(K)}\|_2^2$

What to submit: a single plot, the train and test MSE requested. Include a screen shot of any code used for this section and a copy of your python code in solutions.py.

Solution:
The plot generated is:



The best step size seems to be $\alpha = 0.01$ since it results in the quickest convergence. The train and test MSE are 4.5589 and 4.3804 respectively. The code used in this section is:

```
phi = 0.5
n = X_train.shape[0]
p = X_train.shape[1]
n_epochs = 1000
n_iter = n_epochs
alphas = [0.000001, 0.000005, 0.00001,  0.00005, 0.0001,   0.0005, 0.001, 0.005, 0.01]
beta_star = np.linalg.inv(X_train.T @ X_train + n * phi * np.eye(p)) @ X_train.T @
    Y_train

def loss(beta, X, y):
    return (1/n) * np.linalg.norm(y-X@beta)**2 + phi * np.linalg.norm(beta)**2

def calc_grad(beta, X, y):
    return -(2/n) * X.T @ (y - X @ beta) + 2 * phi * beta

fig, axes = plt.subplots(3,3,figsize=(10,10))
```

```
for i, ax in enumerate(axes.flat):
    betas = np.zeros((n_iter, p))
    betas[0] = np.ones(p)
    loss_diffs = np.ones(n_iter) * np.inf
    loss_diffs[0] = loss(betas[0], X_train, Y_train) - loss(beta_star, X_train,
    Y_train)
    alpha = alphas[i]
    for j in range(1, n_iter):
        betas[j] = betas[j-1] - alpha * calc_grad(betas[j-1], X_train, Y_train)
        loss_diffs[j] = loss(betas[j], X_train, Y_train) - loss(beta_star, X_train,
    Y_train)
    ax.plot(np.arange(n_iter), loss_diffs)
    ax.set_title(rf'$\alpha$={alpha}')
    ax.set_ylabel(r'$L(\beta^{(k)}) - L(\hat{\beta})$')
    ax.set_xlabel(rf'$k$')
plt.tight_layout()
plt.savefig("figures/batchGD.png", dpi=400)
plt.show()

def MSE(beta, X, y):
    n = X.shape[0]
    return (1/n) * np.linalg.norm(y-X@beta)**2

print("train MSE: ", MSE(betas[-1], X_train, Y_train))
print("test  MSE: ", MSE(betas[-1], X_test, Y_test))

```

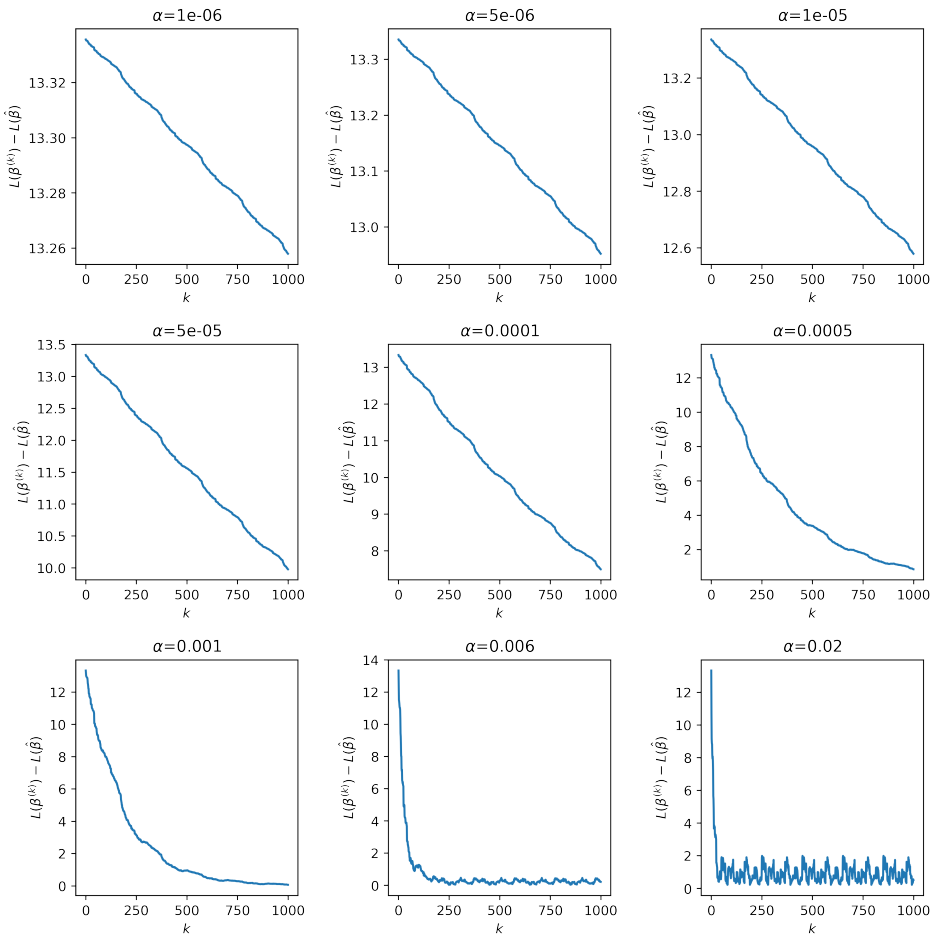
(h) We will now implement SGD from scratch to solve the ridge regression problem. Use an initial estimate $\beta^{(0)} = 1_p$ (the vector of ones) and $\phi = 0.5$ and run the algorithm for 5 epochs (this means a total of $5n$ updates of β, where n is the size of the training set). Repeat this for the following step sizes:

$$\alpha \in \{0.000001, 0.000005, 0.00001, 0.00005, 0.0001, 0.0005, 0.001, 0.006, 0.02\}$$

Present an analogous 3×3 grid plot as in the previous question. Instead of choosing an index randomly at each step of SGD, we will cycle through the observations in the order they are stored in X_train to ensure consistent results. Report the best step-size choice and the corresponding train and test MSEs. In some cases you might observe that the value of $\Delta^{(k)}$ jumps up and down, and this is not something you would have seen using batch GD. Why do you think this might be happening?

What to submit: a single plot, the train and test MSE requested and some commentary. Include a screen shot of any code used for this section and a copy of your python code in solutions.py.

Solution:
The plot generated is:



The best step size can be taken to be $\alpha = 0.006$ since it results in the quickest convergence. The train and test MSE are 4.6617 and 4.4472 respectively. Alternatively, we might also take $\alpha = 0.001$ since it achieves the lowest value after 1000 iterations, in which case the train and test MSE are 4.8622 and 4.6426 respectively.

We see the jumps because at each step we are approximating the full gradient with a single term, and so we are not actually guaranteed to be taking a step in the direction of greatest descent anymore, and so at times we see the loss value actaully increase after a step of SGD.

The code used in this section is:

```
    phi = 0.5
    n = X_train.shape[0]
    p = X_train.shape[1]
    n_epochs = 5
    n_iter = n_epochs * n
    alphas = [0.000001, 0.000005, 0.00001,  0.00005, 0.0001,   0.0005, 0.001, 0.006,
    0.009]
```

```
    beta_star = np.linalg.inv(X_train.T @ X_train+ n * phi * np.eye(p)) @ X_train.T @
    Y_train

    def loss(beta, X, y):
        return (1/n) * np.linalg.norm(y-X@beta)**2 + phi * np.linalg.norm(beta)**2

    def calc_grad_i(beta, xi, yi):
        return -2 * xi * (yi-np.dot(xi,beta)) + 2 * n * phi * beta

    fig, axes = plt.subplots(3,3,figsize=(10,10))
    for i, ax in enumerate(axes.flat):
        betas = np.zeros((n_iter, p))
        betas[0] = np.ones(p)
        loss_diffs = np.ones(n_iter) * np.inf
        loss_diffs[0] = loss(betas[0], X_train, Y_train) - loss(beta_star, X_train,
    Y_train)
        alpha = alphas[i]
        for j in range(1, n_iter):
            idx = j%n
            betas[j] = betas[j-1] - alpha * calc_grad_i(betas[j-1], X_train[idx],
    Y_train[idx])
            loss_diffs[j] = loss(betas[j], X_train, Y_train) - loss(beta_star, X_train
    , Y_train)
        ax.plot(np.arange(n_iter), loss_diffs)
        ax.set_title(rf'$\alpha$={alpha}')
        ax.set_ylabel(r'$L(\beta^{(k)}) - L(\hat{\beta})$')
        ax.set_xlabel(rf'$k$')
    plt.tight_layout()
    plt.savefig('figures/SGD.png', dpi=400)
    plt.show()

    alpha = 0.0001
    betas = np.zeros((n_iter, p))
    betas[0] = np.ones(p)
    for j in range(1, n_iter):
        idx = j%n
        betas[j] = betas[j-1] - alpha * calc_grad_i(betas[j-1], X_train[idx], Y_train[
    idx])

    print("train MSE: ", MSE(betas[-1], X_train, Y_train))
    print("test  MSE: ", MSE(betas[-1], X_test, Y_test))
```

(i) Based on your GD and SGD results, which algorithm do you prefer? When is it a better idea to use GD? When is it a better idea to use SGD?

Solution:
We see that GD achieves lower train and test MSE than SGD but note that in GD in the fastest converging step size case it took roughly 50 epochs to achieve the lowest error, whereas SGD converged after less than 1 epoch. We can therefore argue that GD is better when the n is small and it is not too computationally costly to compute the full gradient. SGD is far less computationally costly but at the end of the day it requires us to use quite a crude approximation to the full gradient, and so we are not guaranteed to actually 'descend' on any given iteration of the algorithm nor descent to the best possible solution.

(j) Note that in GD, SGD and mini-batch GD, we always update the entire p-dimensional vector β at each iteration. An alternative popular approach is to update each of the p parameters individually. To make this idea more clear, we write the ridge loss $L(\beta)$ as $L(\beta_1, \beta_2 \ldots, \beta_p)$. We initialize $\beta^{(0)}$, and then solve for $k = 1, 2, 3, \ldots,$

$$\begin{aligned}
\beta_1^{(k)} &= \arg\min_{\beta_1} L(\beta_1, \beta_2^{(k-1)}, \beta_3^{(k-1)}, \ldots, \beta_p^{(k-1)}) \\
\beta_2^{(k)} &= \arg\min_{\beta_2} L(\beta_1^{(k)}, \beta_2, \beta_3^{(k-1)}, \ldots, \beta_p^{(k-1)}) \\
&\vdots \\
\beta_p^{(k)} &= \arg\min_{\beta_p} L(\beta_1^{(k)}, \beta_2^{(k)}, \beta_3^{(k)}, \ldots, \beta_p).
\end{aligned}$$

Note that each of the minimizations is over a single (1-dimensional) coordinate of β, and also that as as soon as we update $\beta_j^{(k)}$, we use the new value when solving the update for $\beta_{j+1}^{(k)}$ and so on. The idea is then to cycle through these coordinate level updates until convergence. In the next two parts we will implement this algorithm from scratch for the Ridge regression problem:

$$L(\beta) = \frac{1}{n}\|y - X\beta\|_2^2 + \phi\|\beta|_2^2$$

Note that we can write the $n \times p$ matrix $X = [X_1, \ldots, X_p]$, where X_j is the j-th column of X. Find the solution of the optimization

$$\hat{\beta}_1 = \arg\min_{\beta_1} L(\beta_1, \beta_2, \ldots, \beta_p).$$

Based on this, derive similar expressions for $\hat{\beta}_j$ for $j = 2, 3, \ldots, p$.
Hint: Note the expansion: $X\beta = X_j\beta_j + X_{-j}\beta_{-j}$, where X_{-j} denotes the matrix X but with the j-th column removed, and similarly β_{-j} is the vector β with the j-th coordinate removed. *What to submit:* your working out.

Solution:
Using the hint and applying the chain rule, we have for any $j = 1, \ldots, p$,

$$\begin{aligned}
\frac{\partial}{\partial\beta_j}\left\{\frac{1}{n}\|y - X\beta\|_2^2 + \phi\|\beta|_2^2\right\} &= \frac{\partial}{\partial\beta_j}\left\{\frac{1}{n}\|y - X_{-j}\beta_{-j} - X_j\beta_j\|_2^2 + \phi\|\beta|_2^2\right\} \\
&= -\frac{2}{n}X_j^T(y - X_{-j}\beta_{-j} - X_j\beta_j) + 2\phi\beta_j.
\end{aligned}$$

Setting this to zero and solving yields

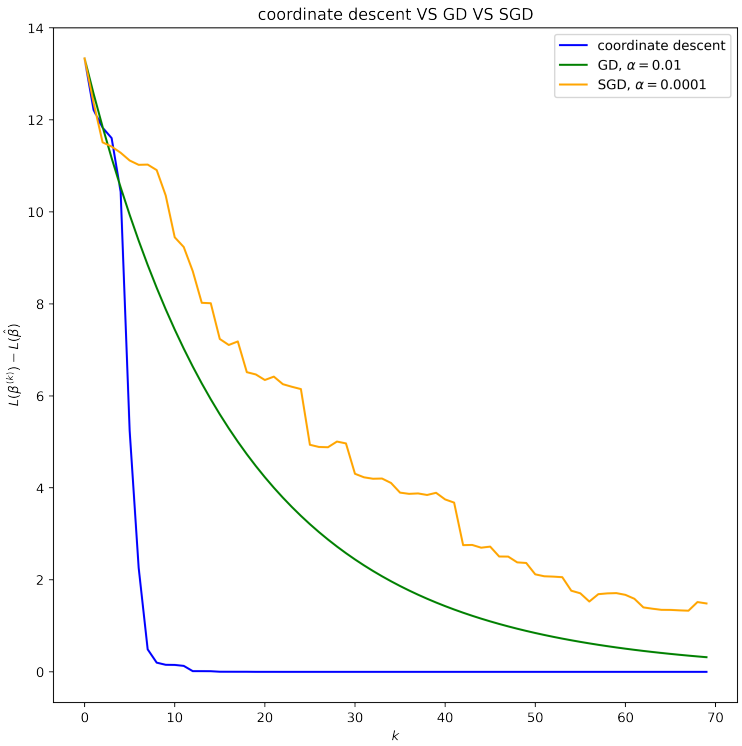
$$\hat{\beta}_1 = \frac{X_1^T(y - X_{-1}\beta_{-1})}{n\phi + \|X_1\|_2^2},$$

.

(k) Implement the algorithm outlined in the previous question on the training dataset. In your implementation, be sure to update the β_j's in order and use an initial estimate of $\beta^{(0)} = 1_p$ (th vector of ones), and $\phi = 0.5$. Terminate the algorithm after 10 cycles (one cycle here is p updates, one for each

β_j), so you will have a total of $10p$ updates. Report the train and test MSE of your resulting model. Here we would like to compare the three algorithms: new algorithm to batch GD and SGD from your previous answers with optimally chosen step sizes. Create a plot of k vs. $\Delta^{(k)}$ as before, but this time plot the progression of the three algorithms. Be sure to use the same colors as indicated here in your plot, and add a legend that labels each series clearly. For your batch GD and SGD include the step-size in the legend. Your x-axis only needs to range from $k = 1, \ldots 10p$. Further, report both train and test MSE for your new algorithm. *Note: Some of you may be concerned that we are comparing one step of GD to one step of SGD and the new aglorithm, we will ignore this technicality for the time being. What to submit: a single plot, the train and test MSE requested.*

Solution:
The train MSE is 4.5589, the test MSE is 4.38043. The plot of results:




```
        phi = 0.5
        n = X_train.shape[0]
        p = X_train.shape[1]
        n_cycles = 10
        n_iter = n_cycles * p
        beta_star = np.linalg.inv(X_train.T @ X_train+ n * phi * np.eye(p)) @ X_train.
    T @ Y_train

        def loss(beta, X, y):
            return (1/n) * np.linalg.norm(y-X@beta)**2 + phi * np.linalg.norm(beta)**2

```

```
        def beta_j_update(j, beta, X, y):
            n = X.shape[0]
            Xj = X[:, j]
            Xmj = np.delete(X, j, axis=1)
            betaj = beta[j]
            betamj = np.delete(beta, j)

            num = Xj.T @ (y- Xmj @ betamj)
            den = n * phi + np.linalg.norm(Xj)**2
            return num / den


        betas = np.zeros((n_iter, p))
        betas[0] = np.ones(p)
        loss_diffs = np.ones(n_iter) * np.inf
        loss_diffs[0] = loss(betas[0], X_train, Y_train) - loss(beta_star, X_train,
    Y_train)

        fig = plt.figure(figsize=(8,8))
        for i in range(n_iter-1):
            idx = i%p
            betas[i+1] = betas[i]    # new beta vec is set to old beta vec

            # update a single coordinate of new beta vec
            betas[i+1][idx] = beta_j_update(idx, betas[i+1], X_train, Y_train)
            loss_diffs[i+1] = loss(betas[i+1], X_train, Y_train) - loss(beta_star,
    X_train, Y_train)

        plt.plot(np.arange(n_iter), loss_diffs, color='blue', label='coordinate
    descent')

        print("train MSE: ", MSE(betas[-1], X_train, Y_train))
        print("test  MSE: ", MSE(betas[-1], X_test, Y_test))

        # GD
        n_epochs = 1000
        alpha = 0.01

        def calc_grad(beta, X, y):
            return -(2/n) * X.T @ (y - X @ beta) + 2 * phi * beta


        betas = np.zeros((n_iter, p))
        betas[0] = np.ones(p)
        loss_diffs_gd = np.ones(n_iter) * np.inf
        loss_diffs_gd[0] = loss(betas[0], X_train, Y_train) - loss(beta_star, X_train,
     Y_train)

        for j in range(1, n_iter):
            betas[j] = betas[j-1] - alpha * calc_grad(betas[j-1], X_train, Y_train)
            loss_diffs_gd[j] = loss(betas[j], X_train, Y_train) - loss(beta_star,
    X_train, Y_train)

        plt.plot(np.arange(n_iter), loss_diffs_gd, color='green', label='GD, $\\alpha
    =0.01$')

        # SGD
        alpha = 0.006
```

```
        betas = np.zeros((n_iter, p))
        betas[0] = np.ones(p)
        loss_diffs_sgd = np.ones(n_iter) * np.inf
        loss_diffs_sgd[0] = loss(betas[0], X_train, Y_train) - loss(beta_star, X_train
    , Y_train)
        for j in range(1, n_iter):
            idx = j%n
            betas[j] = betas[j-1] - alpha * calc_grad_i(betas[j-1], X_train[idx],
    Y_train[idx])
            loss_diffs_sgd[j] = loss(betas[j], X_train, Y_train) - loss(beta_star,
    X_train, Y_train)

        plt.plot(np.arange(n_iter), loss_diffs_sgd, color='orange', label='SGD, $\\
    alpha=0.0001$')

        plt.legend()
        plt.title(rf'coordinate descent VS GD VS SGD')
        plt.ylabel(r'$L(\beta^{(k)}) - L(\hat{\beta})$')
        plt.xlabel(rf'$k$')
        plt.tight_layout()
        plt.savefig('figures/coordGD.png', dpi=400)
        plt.show()

```

(l) In part (d), we standardized the entire data set and then split into train and test sets. In light of this, do you believe that your results in parts (e)-(k) are more reliable, less reliable, or unaffected? Explain. *What to submit: your commentary*

Solution:
Of course it makes them less reliable. The scaler was fit on the entire data (including the test set), and so our results might suffer from some overfitting. The test set should not be used for **any** form of preprocessing or validation, we must treat it as completely unseen data.