COMP9417 - Machine Learning
Homework 1: Regularized Optimization & Gradient Methods

Introduction In this homework we will explore *gradient* based optimization. Gradient based 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 a particular 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 30 marks.

- Question 1 a): 2 marks
- Question 1 b): 4 marks
- Question 1 c): 2 marks
- Question 1 d): 2 marks
- Question 1 e): 6 marks
- Question 1 f): 6 marks
- Question 1 g): 4 marks
- Question 1 h): 2 marks
- Question 1 i): 2 marks

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: **October 12th**, 2025 by **6pm**. 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 made on 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 $x_* = -1$[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

$$A = \begin{bmatrix} 3 & 2 & 0 & -1 \\ -1 & 3 & 0 & 2 \\ 0 & -4 & -2 & 7 \end{bmatrix}, \qquad b = \begin{bmatrix} 3 \\ 1 \\ -4 \end{bmatrix},$$

and γ is a positive constant. Run gradient descent on f using a step size of $\alpha = 0.01$ and $\gamma = 2$ 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:

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

[1] Does the algorithm converge to the true minimizer? Why/Why not?

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:

$$\begin{aligned} = 0, \quad & x_k = [1, 1, 1, 1] \\ k = 1, \quad & x_k = [0.98, 1.07, 1.08, 0.58] \\ k = 2, \quad & x_k = [0.9393, 1.0117, 1.0908, 0.4222] \\ k = 3, \quad & x_k = [0.8973, 0.934, 1.0835, 0.3383] \\ k = 4, \quad & x_k = [0.8586, 0.862, 1.0712, 0.2795] \\ k = 5, \quad & x_k = [0.8236, 0.8004, 1.0571, 0.2328]. \end{aligned}$$

The last 5 steps are:

$$\begin{aligned} k = 382, \quad & x_k = [0.4302, 0.5845, 0.0479, -0.217] \\ k = 383, \quad & x_k = [0.4302, 0.5845, 0.0479, -0.217] \\ k = 384, \quad & x_k = [0.4302, 0.5845, 0.0479, -0.217] \\ k = 385, \quad & x_k = [0.4302, 0.5845, 0.0479, -0.217] \\ k = 386, \quad & x_k = [0.4302, 0.5845, 0.0479, -0.217]. \end{aligned}$$

So the final value is $[0.4302, 0.5845, 0.0479, -0.217]$. The code used is:

```
A = np.array([[3,2,0,-1], [-1,3,0,2], [0,-4,-2,7]])
b = np.array([3,1,-4])
gamma = 2
alpha = 0.01
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

```

Consider now a slightly different problem: let $y, \beta \in \mathbb{R}^p$ and $\lambda > 0$. Further, we define the matrix $W \in \mathbb{R}^{(p-2)\times p}$ as

$$W = \begin{bmatrix} 1 & -2 & 1 & & & & \\ & 1 & -2 & 1 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & 1 & -2 & 1 & \\ & & & & 1 & -2 & 1 \end{bmatrix},$$

where blanks denote zero elements.[2] Define the loss function:

$$L(\beta) = \frac{1}{2p}\|y - \beta\|_2^2 + \lambda\|W\beta\|_2^2. \tag{2}$$

The following code allows you to load in the data needed for this problem[3]:

```
import numpy as np
import matplotlib.pyplot as plt
t_var = np.load("t_var.npy")
y_var = np.load("y_var.npy")
plt.plot(t_var, y_var)
plt.show()

```

Note, the t variable is purely for plotting purposes, it should not appear in any of your calculations.

(b) Show that

$$\hat{\beta} = \arg\min_{\beta} L(\beta) = (I + 2\lambda p W^T W)^{-1} y.$$

Update the following code[4] so that it returns a plot of $\hat{\beta}$ and calculates $L(\hat{\beta})$. Only in your code implementation, set $\lambda = 0.9$.

```
def create_W(p):
   ## generate W which is a p-2 x p matrix as defined in the question
    W = np.zeros((p-2, p))
    b = np.array([1,-2,1])
    for i in range(p-2):
        W[i,i:i+3] = b
    return W

def loss(beta, y, W, L):
    ## compute loss for a given vector beta for data y, matrix W, regularization
    parameter L (lambda)
    # your code here
    return loss_val

## your code here, e.g. compute betahat and loss, and set other params..

plt.plot(t_var, y_var, zorder=1, color='red', label='truth')
plt.plot(t_var, beta_hat, zorder=3, color='blue',
            linewidth=2, linestyle='--', label='fit')
```

[2] If it is not already clear: for the first row of W: $W_{11} = 1, W_{12} = -2, W_{13} = 1$ and $W_{1j} = 0$ for any $j \geq 4$. For the second row of W: $W_{21} = 0, W_{22} = 1, W_{23} = -2, W_{24} = 1$ and $W_{2j} = 0$ for any $j \geq 5$ and so on.

[3] a copy of this code is provided in `code_student.py`

[4] a copy of this code is provided in `code_student.py`

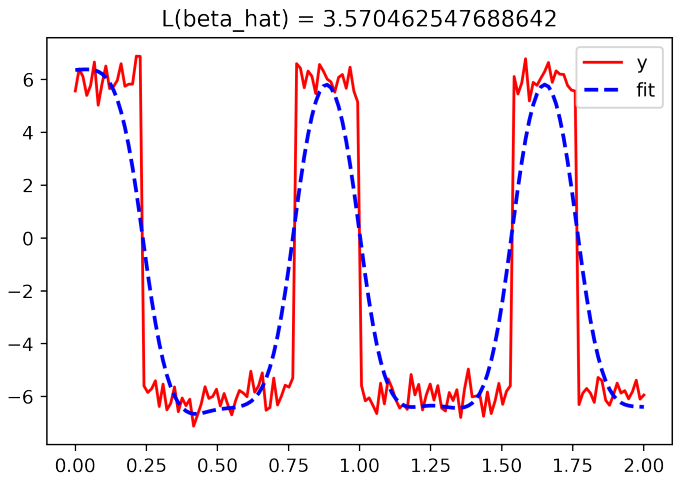
```
plt.legend(loc='best')
plt.title(f"L(beta_hat) = {loss(beta_hat, y, W, L)}")
plt.show()

```

What to submit: a closed form expression along with your working, a single plot and a screen shot of your code along with a copy of your code in your .py file.

Solution:
Differentiating and setting to zero gives

$$\nabla L(\beta) = -\frac{1}{p}(y-\beta) + 2\lambda W^T W\beta = 0 \iff \hat{\beta} = (I + 2\lambda p W^T W)^{-1} y$$

The plot and loss are:



The code used is:

```
def create_W(p):
    W = np.zeros((p-2, p))
    b = np.array([1,-2,1])
    for i in range(p-2):
        W[i,i:i+3] = b
    return W

W = create_W(p)

def loss(beta, y, W, L):
    return  (1/(2 * p)) * np.linalg.norm(y-beta)**2 + L * np.linalg.norm(W @ beta)**2

p = y.shape[0]
L = 0.9
beta_hat = np.linalg.inv(2 * L * p * W.T @ W + np.eye(p)) @ y
print(loss(beta_hat, y, W, L))
```

```
17
18 plt.plot(t, y, zorder=1, color='red', label='truth')
19 plt.plot(t, beta_hat, zorder=3, color='blue',
20             linewidth=2, linestyle='--', label='fit')
21 plt.legend(loc='best')
22 plt.title(f"L(beta_hat) = {loss(beta_hat, y, W, L)}")
23 plt.savefig("figures/closed_form_fit.png")
24 plt.show()
25
```

(c) Write out each of the two terms that make up the loss function ($\frac{1}{2p}\|y-\beta\|_2^2$ and $\lambda\|W\beta\|_2^2$) explicitly using summations. Use this representation to explain the role played by each of the two terms. Be as specific as possible. *What to submit: your answer, and any working either typed or handwritten.*

Solution:
We get that

$$\frac{1}{2p}\sum_{j=1}^{p}(y_j-\beta_j)^2+\lambda\sum_{j=2}^{p-1}(\beta_{j-1}-2\beta_j+\beta_{j+1})^2.$$

The first term is the data-fit term, and forces β to be close to the signal y. The second sum is a penalty that enforces smoothness of β, since deviations of β_p from its preceding and succeeding terms will cause the penalty to be higher. The penalty is zero only when the three points $\beta_{j-1}, \beta_j, \beta_{j+1}$ are on a line (maximal smoothness).

(d) Show that we can write (2) in the following way:

$$L(\beta)=\frac{1}{p}\sum_{j=1}^{p}L_j(\beta),$$

where $L_j(\beta)$ depends on the data $y_1,\ldots,y_p$ only through y_j. Further, show that

$$\nabla L_j(\beta)=\begin{bmatrix}0\\ \vdots\\ 0\\ -(y_j-\beta_j)\\ 0\\ \vdots\\ 0\end{bmatrix}+2\lambda W^TW\beta,\qquad j=1,\ldots,p.$$

Note that the first vector is the p-dimensional vector with zero everywhere except for the j-th index. Take a look at the supplementary material if you are confused by the notation. *What to submit: your answer, and any working either typed or handwritten.*

Solution:
This should be pretty straight-forward from the previous part, we can write

$$L(\beta) = \frac{1}{p}\sum_{j=1}^{p} L_j(p),$$

where

$$L_j(\beta) = \frac{1}{2}(y_j - \beta_j)^2 + \lambda\|W\beta\|_2^2.$$

From here, the gradient computation is also simple if we remember the definition of the gradient:

$$\nabla L_j(\beta) = \left[\frac{\partial L_j(\beta)}{\partial \beta_1}, \ldots, \frac{\partial L_j(\beta)}{\partial \beta_p}\right]^T.$$

Clearly then for $k = 1, \ldots, p$:

$$\frac{\partial L_j(\beta)}{\partial \beta_k} = \begin{cases} -(y_j - \beta_j) + 2\lambda W^T W\beta & \text{if} \quad j = k \\ 0 + 2\lambda W^T W\beta & \text{if} \quad j \neq k. \end{cases}$$

(e) In this question, you will implement (batch) GD from scratch to minimize the loss function (2). Use an initial estimate $\beta^{(0)} = 1_p$ (the p-dimensional vector of ones), and $\lambda = 0.001$ 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:

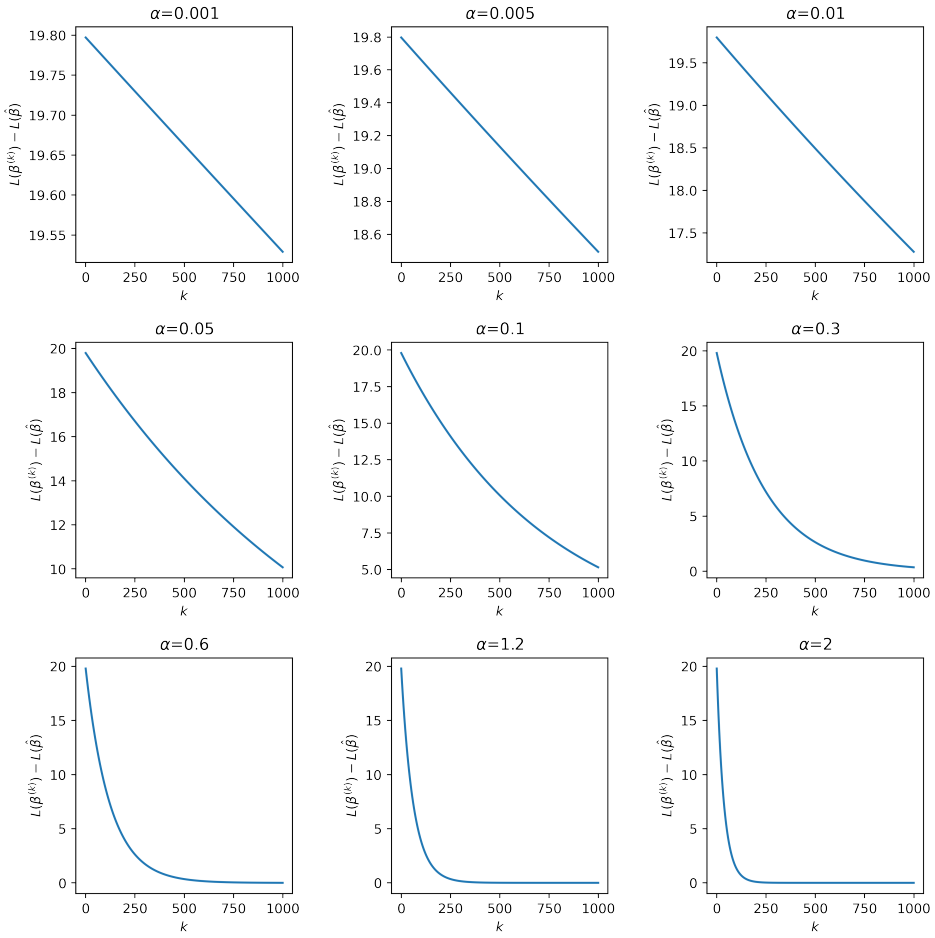
$$\alpha \in \{0.001, 0.005, 0.01, 0.05, 0.1, 0.3, 0.6, 1.2, 2\}$$

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) solution derived earlier (but with $\lambda = 0.001$ now for consistency). Present your results in a single 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 in terms of speed of convergence. *What to submit: a single plot. 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 $\alpha = 2$ since it results in the quickest convergence. The code used in this section is:

```
        n_epochs = 1000
        n_iter = n_epochs
        alphas = [0.001, 0.005, 0.01, 0.05, 0.1, 0.3, 0.6, 1.2, 2]
        L = 0.001

        def calc_grad(b, y, W, L):
            p = b.shape[0]
            return -(y - b)/p + 2 * L * W.T @ W @ b

        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], y, W, L) - loss(beta_hat, y, W, L)
            alpha = alphas[i]
```

```
            for j in range(1, n_iter):
                betas[j] = betas[j-1] - alpha * calc_grad(betas[j-1], y, W, L)
                loss_diffs[j] = loss(betas[j], y, W, L) - loss(beta_hat, y, W, L)
            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()

```

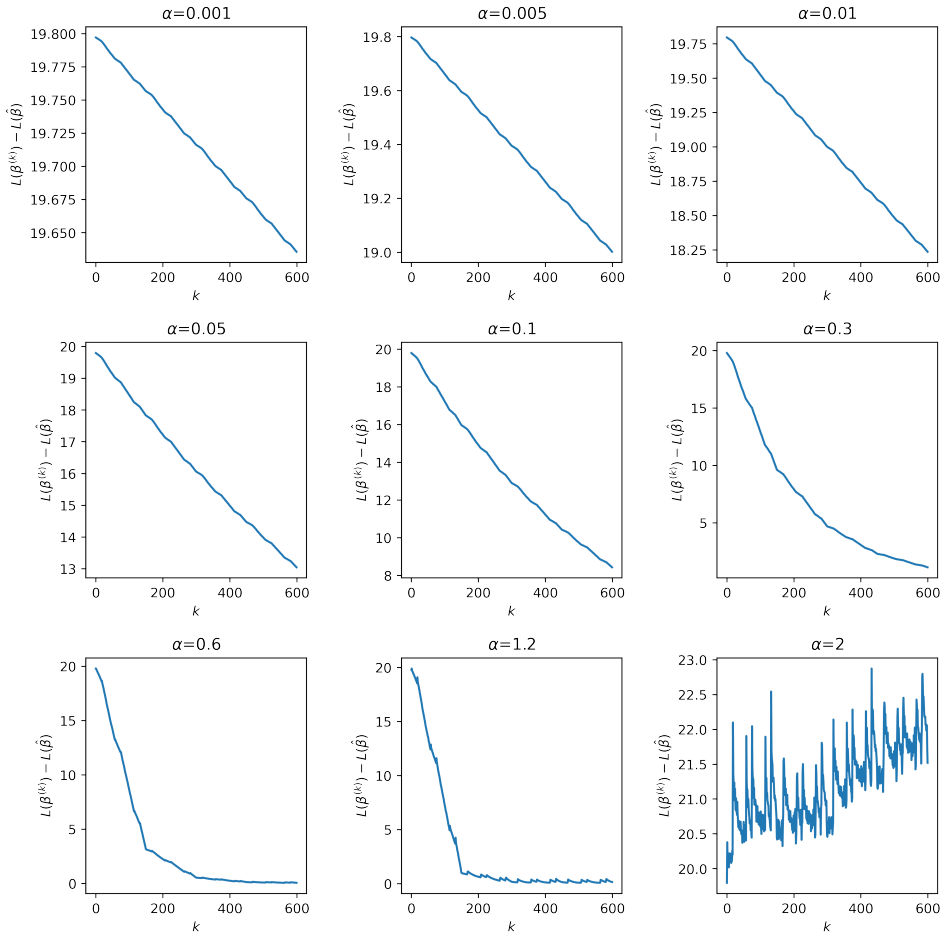
(f) We will now implement SGD from scratch to solve (2). Use an initial estimate $\beta^{(0)} = 1_p$ (the vector of ones) and $\lambda = 0.001$ and run the algorithm for 4 epochs (this means a total of $4p$ updates of β. Repeat this for the following step sizes:

$$\alpha \in \{0.001, 0.005, 0.01, 0.05, 0.1, 0.3, 0.6, 1.2, 2\}$$

Present an analogous single 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 y to ensure consistent results. Report the best step-size choice. 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 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 $\alpha = 1.2$ since it results in the quickest convergence. Note that for $\alpha = 2$ the SGD gives terrible results with the loss increasing. This indicates that estimating the full gradient by a single ∇L_j and taking a large step in that direction is a poor estimate.

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.Taking the step-size to be not too large minimizes the damage of going too far in the wrong direction though.

The code used in this section is:

```
        alphas = [0.001, 0.005, 0.01, 0.05, 0.1, 0.3, 0.6, 1.2, 2]
        L = 0.001
        p = y.shape[0]
        n_iter = 4 * p

```

```
        def calc_grad_i(i, b, y, W, L):
            p = b.shape[0]
            g_vec = np.zeros(p)
            g_vec[i] = -(y[i] - b[i])
            return g_vec + 2 * L * W.T @ W @ b

        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], y, W, L) - loss(beta_hat, y, W, L)
            alpha = alphas[i]
            for j in range(1, n_iter):
                idx = j%p
                betas[j] = betas[j-1] - alpha * calc_grad_i(idx, betas[j-1], y, W, L)
                loss_diffs[j] = loss(betas[j], y, W, L) - loss(beta_hat, y, W, L)
            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()
```

An alternative Coordinate Based scheme: In GD, SGD and mini-batch GD, we always update the entire p-dimensional vector β at each iteration. An alternative approach is to update each of the p parameters individually. To make this idea more clear, we write the loss function of interest $L(\beta)$ 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 problem we have been working on (2).

(g) Derive closed-form expressions for $\hat{\beta}_1, \hat{\beta}_2, \ldots, \hat{\beta}_p$ where for $j = 1, 2, \ldots, p$:

$$\hat{\beta}_j = \arg\min_{\beta_j} L(\beta_1, \ldots, \beta_{j-1}, \beta_j, \beta_{j+1}, \ldots, \beta_p).$$

What to submit: a closed form expression along with your working.

Hint: Be careful, this is not as straight-forward as it might seem at first. It is recommended to choose a value for p, e.g. $p = 8$ and first write out the expression in terms of summations. Then take derivatives to get the closed form expressions.

Solution:
Grader: I imagine that a common mistake will be that students assume all the updates are of the same form, this should be penalized heavily. We have to be careful here since the $\hat{\beta}_j$'s will be of different forms. Writing it out in the case of $p = 8$ for example:

$$\begin{aligned} L(\beta) &= \frac{1}{2 \times 8} \sum_{j=1}^{8} (y_j - \beta_j)^2 + \lambda \sum_{j=2}^{7} (\beta_{j-1} - 2\beta_j + \beta_{j+1})^2 \\ &= \frac{1}{2 \times 8} \sum_{j=1}^{p} (y_j - \beta_j)^2 + \lambda[(\beta_1 - 2\beta_2 + \beta_3)^2 \\ &\qquad + (\beta_2 - 2\beta_3 + \beta_4)^2 \\ &\qquad + (\beta_3 - 2\beta_4 + \beta_5)^2 \\ &\qquad + (\beta_4 - 2\beta_5 + \beta_6)^2 \\ &\qquad + (\beta_5 - 2\beta_6 + \beta_7)^2 \\ &\qquad + (\beta_6 - 2\beta_7 + \beta_8)^2]. \end{aligned}$$

So we see that β_1 and β_p only appear in one of the terms each, whereas β_2 and β_{p-1} appear in two terms each, and the rest ($\beta_3, \ldots, \beta_{p-2}$) appear in 3 terms each.

1. *β_1 and β_p:* Differentiating with respect to β_1 and setting to zero yields

$$-\frac{1}{p}(y_1 - \beta_1) + 2\lambda(\beta_1 - 2\beta_2 + \beta_3) = 0 \iff \hat{\beta}_1 = \frac{y_1 - 2p\lambda(\hat{\beta}_3 - 2\hat{\beta}_2)}{1 + 2p\lambda}.$$

   By the symmetry of the problem, it follows also that

$$\hat{\beta}_p = \frac{y_p - 2p\lambda(\hat{\beta}_{p-2} - 2\hat{\beta}_{p-1})}{1 + 2p\lambda}$$

2. *β_2 and β_{p-1}:* Differentiating with respect to β_2 and setting to zero yields

$$\begin{aligned} &-\frac{1}{p}(y_2 - \beta_2) + 2\lambda(-2)(\beta_1 - 2\beta_2 + \beta_3) + 2\lambda(\beta_2 - 2\beta_3 + \beta_4) \\ &\iff \hat{\beta}_2 = \frac{y_2 - 2p\lambda(\hat{\beta}_4 - 2\hat{\beta}_1 - 4\hat{\beta}_3)}{1 + 10p\lambda}. \end{aligned}$$

   By symmetry then, it follows that

$$\hat{\beta}_{p-1} = \frac{y_{p-1} - 2p\lambda(\hat{\beta}_{p-3} - 2\hat{\beta}_p - 4\hat{\beta}_{p-2})}{1 + 10p\lambda}.$$

3. β_3: Differentiating with respect to β_3 and setting to zero yields

$$-\frac{1}{p}(y_3-\beta_3)+\lambda[2(\beta_1-2\beta_2+\beta_3)-4(\beta_2-2\beta_3+\beta_4)+2(\beta_3-2\beta_4+\beta_5)]$$
$$\iff \hat{\beta}_3=\frac{y_3-2p\lambda(\hat{\beta}_1-4\hat{\beta}_2-4\hat{\beta}_4+\hat{\beta}_5)}{1+12p\lambda}.$$

By symmetry then, it follows that for $j=3,4,\ldots,p-2$

$$\hat{\beta}_j=\frac{y_j-2p\lambda(\hat{\beta}_{j-2}-4\hat{\beta}_{j-1}-4\hat{\beta}_{j+1}+\hat{\beta}_{j+2})}{1+12p\lambda}.$$

In summary, we have

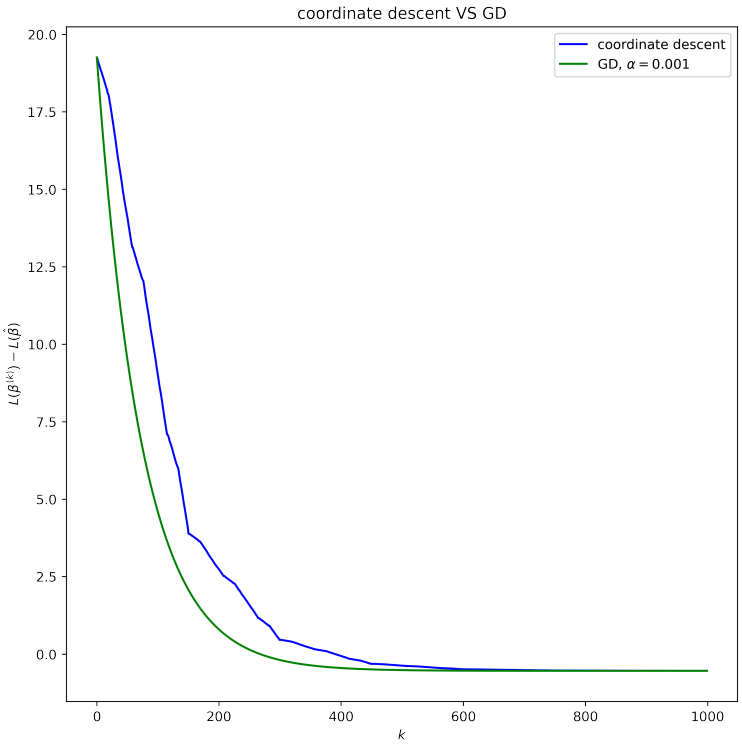
$$\hat{\beta}_j=\begin{cases}\frac{y_1-2p\lambda(\hat{\beta}_3-2\hat{\beta}_2)}{1+2p\lambda} & \text{if } j=1\\ \frac{y_2-2p\lambda(\hat{\beta}_4-2\hat{\beta}_1-4\hat{\beta}_3)}{1+10p\lambda} & \text{if } j=2\\ \frac{y_j-2p\lambda(\hat{\beta}_{j-2}-4\hat{\beta}_{j-1}-4\hat{\beta}_{j+1}+\hat{\beta}_{j+2})}{1+12p\lambda} & \text{if } j=3,4,\ldots,p-2\\ \frac{y_{p-1}-2p\lambda(\hat{\beta}_{p-3}-2\hat{\beta}_p-4\hat{\beta}_{p-2})}{1+10p\lambda} & \text{if } j=p-1\\ \frac{y_p-2p\lambda(\hat{\beta}_{p-2}-2\hat{\beta}_{p-1})}{1+2p\lambda} & \text{if } j=p\end{cases}$$

(h) Implement both gradient descent and the coordinate scheme in code (from scratch) and apply it to the provided data. In your implementation:

- Use $\lambda=0.001$ for the coordinate scheme, and step-size $\alpha=1$ for your gradient descent scheme.
- Initialize both algorithms with $\beta=1_p$, the p-dimensional vector of ones.
- For the coordinate scheme, be sure to update the β_j's in order (i.e. 1,2,3,...)
- For your coordinate scheme, terminate the algorithm after 1000 updates (each time you update a single coordinate, that counts as an update.)
- For your GD scheme, terminate the algoirthm after 1000 epochs.
- Create a single plot of k vs $\Delta^{(k)}=L(\beta^{(k)})-L(\hat{\beta})$, where $\hat{\beta}$ is the closed form expression derived earlier. Your plot should have both the coordinate scheme (blue) and GD (green) displayed and should start from $k=0$. Your plot should have a legend.

What to submit: a single plot and a screen shot of your code along with a copy of your code in your .py file.

Solution:
The plot is:



The code used here is:

```
def beta_j_update(j, b, y, L):
p = y.shape[0]
if j==0:          # first element
    F = b[2]-2*b[1]
    num = y[0] - 2 * p * L * F
    den = 1 + 2*L *p
elif j==1:        # second element
    F = b[3]-2*b[0]-4*b[2]
    num = y[1] - 2 * p * L * F
    den = 1 + 10*L*p
elif j==p-1:        # last element
    F = b[-3]-2*b[-2]
    num = y[-1] - 2 * p * L * F
    den = 1 + 2*L*p
elif j==p-2:      # second last element
    F = b[-4]-2*b[-1]-4*b[-3]
    num = y[-2] - 2 * p * L * F
    den = 1 + 10*L*p
else:
    F = b[j-2]-4*b[j-1]-4*b[j+1]+b[j+2]
    num = y[j] - 2 * p * L * F
    den = 1 + 12*L *p
return num/den


```

```
            n_iter = 1000
            L = 0.001
            betas = np.ones((n_iter, p))
            loss_diffs = np.ones(n_iter) * np.inf
            loss_diffs[0] = loss(betas[0], y, W, L) - loss(beta_hat, y, W, L)

            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], y, L)
                loss_diffs[i+1] = loss(betas[i+1], y, W, L) - loss(beta_hat, y, W, L)

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

            # GD
            alpha = 1


            betas = np.ones((n_iter, p))
            loss_diffs_gd = np.ones(n_iter) * np.inf
            loss_diffs_gd[0] = loss(betas[0], y, W, L) - loss(beta_hat, y, W, L)

            for j in range(1, n_iter):
                betas[j] = betas[j-1] - alpha * calc_grad(betas[j-1], y, W, L)
                loss_diffs_gd[j] = loss(betas[j], y, W, L) - loss(beta_hat, y, W, L)

            plt.plot(np.arange(n_iter), loss_diffs_gd, color='green',
                     label='GD, $\\alpha=1$')


            plt.legend()
            plt.title(rf'coordinate descent VS GD')
            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()
```

(i) Assume the probabilistic model:

$$y|\beta \sim N(\beta, \frac{1}{p} I_p)$$

and place a prior on β with density $\pi(\beta) = c\exp(-\lambda\|W\beta\|_2^2)$, where c is the normalizing constant. The MAP estimator, $\hat{\beta}_{\text{MAP}}$, maximizes the posterior density

$$p(\beta|y) = \frac{p(y|\beta)\pi(\beta)}{p(y)},$$

where $p(y)$ is the distribution of y. Derive $\hat{\beta}_{\text{MAP}}$. How does it compare to $\hat{\beta}$? *What to submit: Some working and commentary.*

Solution:
We are given

$$y \mid \beta \sim \mathcal{N}\left(\beta, \frac{1}{p} I_p\right), \qquad \pi(\beta) = c \exp\big(-\lambda \|W\beta\|_2^2\big).$$

The posterior density (up to proportionality) is

$$p(\beta \mid y) \propto p(y \mid \beta)\, \pi(\beta) \propto \exp\left(-\frac{p}{2}\|y-\beta\|_2^2 - \lambda\|W\beta\|_2^2\right).$$

Maximizing the posterior is equivalent to minimizing the negative log–posterior:

$$J(\beta) = \frac{p}{2}\|y-\beta\|_2^2 + \lambda\|W\beta\|_2^2.$$

Setting the gradient to zero gives

$$p(\beta - y) + 2\lambda W^\top W\beta = 0 \quad \Longrightarrow \quad (pI_p + 2\lambda W^\top W)\beta = py.$$

Hence, the MAP estimator is

$$\hat{\beta}_{\mathrm{MAP}} = (pI_p + 2\lambda W^\top W)^{-1}(py) = \left(I_p + \tfrac{2\lambda}{p} W^\top W\right)^{-1} y.$$

So this differs from $\hat{\beta}$ only in the coefficient of the $W^T W$ term.

Supplementary: Background on Gradient Descent 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 labelled L_i, and each L_i depends on the data only through (x_i, y_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.