# COMP9417 - Homework1

Faiyam Islam - z5258151

## Question 1. Gradient Based Optimization

### (a)

Equation outlining gradient update:

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

$$= \frac{1}{2} \times (Ax-b)^T \times (Ax-b) + \frac{\gamma}{2} \times x^T x$$

$$\nabla f(x) = \frac{1}{2} \times [(Ax-b)^{T\prime} \times (Ax-b) + (Ax-b)^T \times (Ax-b)'] + \frac{\gamma}{2} \times 2x \quad \left(\frac{d}{dx} x^T x = 2x\right)$$

$$= \frac{1}{2} \times [(Ax-b)^{T\prime} \times (Ax-b) + (Ax-b)^{T\prime} \times (Ax-b)] + \gamma x$$

$$= \frac{1}{2} \times 2 \times (Ax-b)^{T\prime} \times (Ax-b) + \gamma x$$

$$= A^T \times (Ax-b) + \gamma x$$

$$x^{(1)} = x^{(0)} - 0.1 \times A^T \times (Ax^{(0)} - b) + \gamma x^{(0)} = [1\ 1\ 1\ 1]^T$$

$$x^{(2)} = x^{(1)} - 0.1 \times A^T \times (Ax^{(1)} - b) \times \gamma x^{(1)} = [0.998\ 0.998\ 0.998\ 0.998]^T$$

$$\vdots$$

$$x^{(k+1)} = x^{(k)} - 0.1 \times A^T \times (Ax^{(k)} - b) \times \gamma x^{(k)}$$

Code and screenshot of first 5 and last 5 rows of iterations:

```python
A = np.array([[1, 2, 1, -1], [-1, 1, 0, 2], [0, -1, -2, 1]])
b = np.array([3, 2, -2])

alpha = 0.1
gamma = 0.2

def gradient_function(x, A, b):
    return -1 * A.T @ b + A.T @ A @ x + gamma * np.eye(4) @ x

x0 = [0, 0, 0, 0]
x1 = [1, 1, 1, 1]

norm2 = 1
k = 0
while norm2 >= 0.001:
    delta = gradient_function(x1, A, b)
    norm2 = np.linalg.norm(delta, ord = 2)
    x0 = x1 - alpha * delta
    print(k, np.round(x1, 4))
    x1 = x0
    k = k + 1
```

```
0 [1 1 1 1]
1 [0.98 0.98 0.98 0.98]
2 [0.9624 0.9804 0.9744 0.9584]
3 [0.9427 0.9824 0.9668 0.9433]
4 [0.9234 0.9866 0.9598 0.9295]

272 [0.0666 1.3366 0.4928 0.3251]
273 [0.0666 1.3366 0.4928 0.325 ]
274 [0.0665 1.3366 0.4927 0.325 ]
275 [0.0664 1.3367 0.4927 0.3249]
276 [0.0663 1.3367 0.4927 0.3249]
```

(b) The termination condition: $\nabla f(x(k))_2 < 0.001$ means that the delta norm 2 of the gradient function has to be less than 0.001. In terms of convergence, as the k increases and we have more iterations, the algorithm will get to a minimiser of f close to 0.001. If we change the right hand side to 0.0001 (or smaller) then the iterations will approach close to 0.0001 or whatever number is chosen on the right hand side.

(c) Using PyTorch, the value for x_hat is: [0.0663, 1.3367, 0.4926, 0.3248]

Code and screenshot of first 5 and last 5 rows of iterations (using PyTorch):

```
A = torch.tensor(A).float()
b = torch.tensor(b).float()

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):
        return self.X

model = MyModel()
optimizer = optim.SGD(model.parameters(), lr = alpha)
k = 1
while k >= 0.001:
    x = model.forward()
    x_hat = A @ x
    res = x_hat - b
    loss_f = ((res**2).sum())/2 + ((x**2).sum()) * (gamma / 2)

    # similar to part a except we use tensor
    loss = -1 * A.T @ b + A.T @ A @ x + gamma * torch.eye(4) @ x
    k = math.sqrt(torch.sum(loss**2))
    print(model.X.data)
    loss_f.backward()
    optimizer.step()
    optimizer.zero_grad()

print("x hat using PyTorch w model: ", model.X.data)
```

```
tensor([1., 1., 1., 1.])
tensor([0.9800, 0.9800, 0.9800, 0.9800])
tensor([0.9624, 0.9804, 0.9744, 0.9584])
tensor([0.9427, 0.9824, 0.9668, 0.9433])
tensor([0.9234, 0.9866, 0.9598, 0.9295])
```

```
tensor([0.0666, 1.3366, 0.4928, 0.3251])
tensor([0.0666, 1.3366, 0.4928, 0.3250])
tensor([0.0665, 1.3366, 0.4927, 0.3250])
tensor([0.0664, 1.3367, 0.4927, 0.3249])
tensor([0.0663, 1.3367, 0.4927, 0.3249])
```

### (d)

Feature means:

```
CompPrice      3.483325e-16
Income         5.162537e-17
Advertising   -6.161738e-17
Population     1.454392e-16
Price         -6.994405e-17
Age            1.786071e-16
Education     -2.534084e-16
```

Feature variances:

```
CompPrice      1.002506
Income         1.002506
Advertising    1.002506
Population     1.002506
Price          1.002506
Age            1.002506
Education      1.002506
```

Code for feature means, variances and the 4 objects:

```
car_data = pd.read_csv('CarSeats.csv')
categorical_variables = ['ShelveLoc', 'Urban', 'US']
numerical_data = car_data.drop(categorical_variables, axis = 1)
response = ['Sales']
predictors = [x for x in list(numerical_data.columns) if x not in response]
scaled_data = StandardScaler()
numerical_data[predictors] = scaled_data.fit_transform(numerical_data[predictors])
```

```
numerical_data[predictors].mean()
```

```
CompPrice      3.483325e-16
Income         5.162537e-17
Advertising   -6.161738e-17
Population     1.454392e-16
Price         -6.994405e-17
Age            1.786071e-16
Education     -2.534084e-16
dtype: float64
```

```
numerical_data[predictors].var()
```

```
CompPrice      1.002506
Income         1.002506
Advertising    1.002506
Population     1.002506
Price          1.002506
Age            1.002506
Education      1.002506
dtype: float64
```

```
response_mean = numerical_data[response].mean()
numerical_data[response] = numerical_data[response] - response_mean
```

```
X_train, X_test, Y_train, Y_test = train_test_split(numerical_data.iloc[:,1:], numerical_data.iloc[:,0],
                                                    test_size = 0.5, shuffle=False)
```

```
X_train.head(1)
```

| | CompPrice | Income | Advertising | Population | Price | Age | Education |
|---|---|---|---|---|---|---|---|
| **0** | 0.850455 | 0.155361 | 0.657177 | 0.075819 | 0.177823 | -0.699782 | 1.184449 |

```
X_train.tail(1)
```

| | CompPrice | Income | Advertising | Population | Price | Age | Education |
|---|---|---|---|---|---|---|---|
| **199** | -0.19425 | 0.692014 | -0.246159 | 0.476656 | 0.431555 | 0.659918 | 0.038208 |

```
X_test.head(1)
```

| | CompPrice | Income | Advertising | Population | Price | Age | Education |
|---|---|---|---|---|---|---|---|
| **200** | 1.242219 | 0.835121 | -0.998939 | 0.57177 | 1.277326 | 0.536309 | -0.725953 |

```
X_test.tail(1)
```

| | CompPrice | Income | Advertising | Population | Price | Age | Education |
|---|---|---|---|---|---|---|---|
| **399** | 0.589279 | -1.132606 | -0.998939 | -1.615848 | 0.177823 | -0.26715 | 0.802369 |

```
Y_train.head(1)
```

```
0    2.003675
Name: Sales, dtype: float64
```

```
Y_train.tail(1)
```

```
199   -1.076325
Name: Sales, dtype: float64
```

```
Y_test.head(1)
```

```
200   -1.936325
Name: Sales, dtype: float64
```

```
Y_test.tail(1)
```

```
399    2.213675
Name: Sales, dtype: float64
```

## (e)

Closed form for ridge parameter:

$$\hat{\beta}_{ridge} = \underset{\beta}{\operatorname{argmin}} \frac{1}{n} \|y - X\beta\|_2^2 + \phi\|\beta\|_2^2$$

$$= \underset{\beta}{\operatorname{argmin}} \frac{1}{n} (y - X\beta)^T (y - X\beta) + \phi\beta^T\beta$$

$$= \underset{\beta}{\operatorname{argmin}} \frac{1}{n} (y^T - \beta^T X^T)(y - X\beta) + \phi\beta^T\beta$$

$$= \underset{\beta}{\operatorname{argmin}} \frac{1}{n} (y^T y - y^T X\beta - \beta^T X^T y + \beta^T X^T X\beta) + \phi\beta^T\beta$$

$$= \underset{\beta}{\operatorname{argmin}} \frac{1}{n} (y^T y - 2\beta^T X^T y + \beta^T X^T X\beta) + \phi\beta^T\beta$$

Now taking the derivative and setting the expression to 0:

$$\frac{1}{n}(-2X^T y + 2X^T X\beta) + 2\phi\beta = 0$$

$$-X^T y + X^T X\beta + \phi n\beta = 0$$

$$\therefore \beta = (X^T X + \phi n I)^{-1} X^T y$$

Ridge solution based on X_train and Y_train:

```
0    0.680673
1    0.282293
2    0.651570
3    0.008348
4   -1.171295
5   -0.400892
6   -0.100634
```

Code for ridge solution:

```
X_T = X_train.T
Ridge_parameter = np.linalg.inv(X_T @ X_train + 0.5 * 200 * np.identity(7)) @ X_T @ Y_train
print(Ridge_parameter)
```

```
0    0.680673
1    0.282293
2    0.651570
3    0.008348
4   -1.171295
5   -0.400892
6   -0.100634
dtype: float64
```

(f)

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

$$= \frac{1}{n}\left[\sum_{i=1}^{n}(y_i - x_i\beta)^2 + n\phi\|\beta\|_2^2\right]$$

$$= \frac{1}{n}\left[\sum_{i=1}^{n}(y_i - x_i\beta)^2 + \sum_{i=1}^{n}\phi\|\beta\|_2^2\right]$$

$$= \frac{1}{n}\sum_{i=1}^{n}\left[(y_i - x_i\beta)^2 + \phi\|\beta\|_2^2\right]$$

where $L_i(\beta) = (y_i - x_i\beta)^2 + \phi\|\beta\|_2^2$, $\beta \in \mathbb{R}^p$, $X \in \mathbb{R}^{n \times p}$, $y \in \mathbb{R}^n$, $\phi > 0$

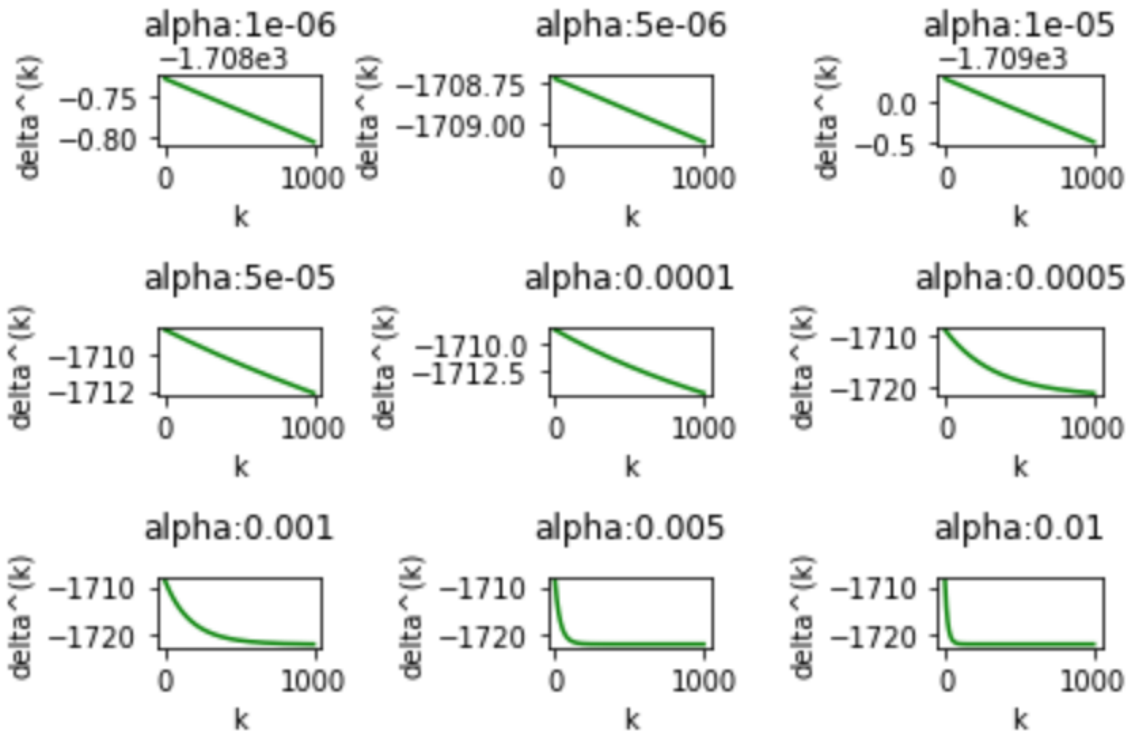
Hence $L_1(\beta), \ldots, L_n(\beta) = (y_1 - x_1\beta)^2 + \phi\|\beta\|_2^2, \ldots, (y_n - x_n\beta)^2 + \phi\|\beta\|_2^2$

and $\nabla L_i(\beta) = \frac{\partial L_i(\beta)}{\partial \beta} = -2(y_i - x_i^T\beta) + 2\phi\beta$

thus $\nabla L_1(\beta), \ldots, \nabla L_n(\beta) = -2(y_1 - x_1^T\beta) + 2\phi\beta, \ldots, -2(y_n - x_n^T\beta) + 2\phi\beta$

(g)

3 x 3 gridplot for batch GD implementation:




I believe the best step-size is $k = 8$, i.e. when $\alpha = 0.01$, because we have a smooth curve with no noise and the delta value is the lowest compared to the rest.

MSE_train = 4.558906724365393

MSE_test = 4.38042918327184

Code for batch GD implementation:

```
# Dataframe with all features
B0 = pd.DataFrame({'CompPrice': [1], 'Income': [1], 'Advertising':[1], 'Population':[1],
                   'Price':[1], 'Age':[1], 'Education':[1]})

# betas containing 7 variables and 1 array
b0 = B0.values.reshape(7, 1)
x = X_train.values.reshape(200, 7)
# print(x)

# X transppose with 7 variables and sample = 200
xt = x.T.reshape(7, 200)
y = Y_train.values.reshape(200, 1)
# print(y)

# Step sizes for alpha
alpha_values = [0.000001, 0.000005, 0.00001, 0.00005, 0.0001, 0.0005, 0.001, 0.005, 0.01]

# phi = 0.5
loss_hat = np.linalg.norm(y - x @ Ridge_parameter, ord = 2)**2 / 200 + 0.5 * np.linalg.norm(Ridge_parameter, ord = 2)**2

# Loss function for Ridge Regression
def ridge_loss(y, x, B):
    return np.linalg.norm(y - x @ B, ord = 2)**2/200 + 0.5 * np.linalg.norm(B, ord = 2)**2
```

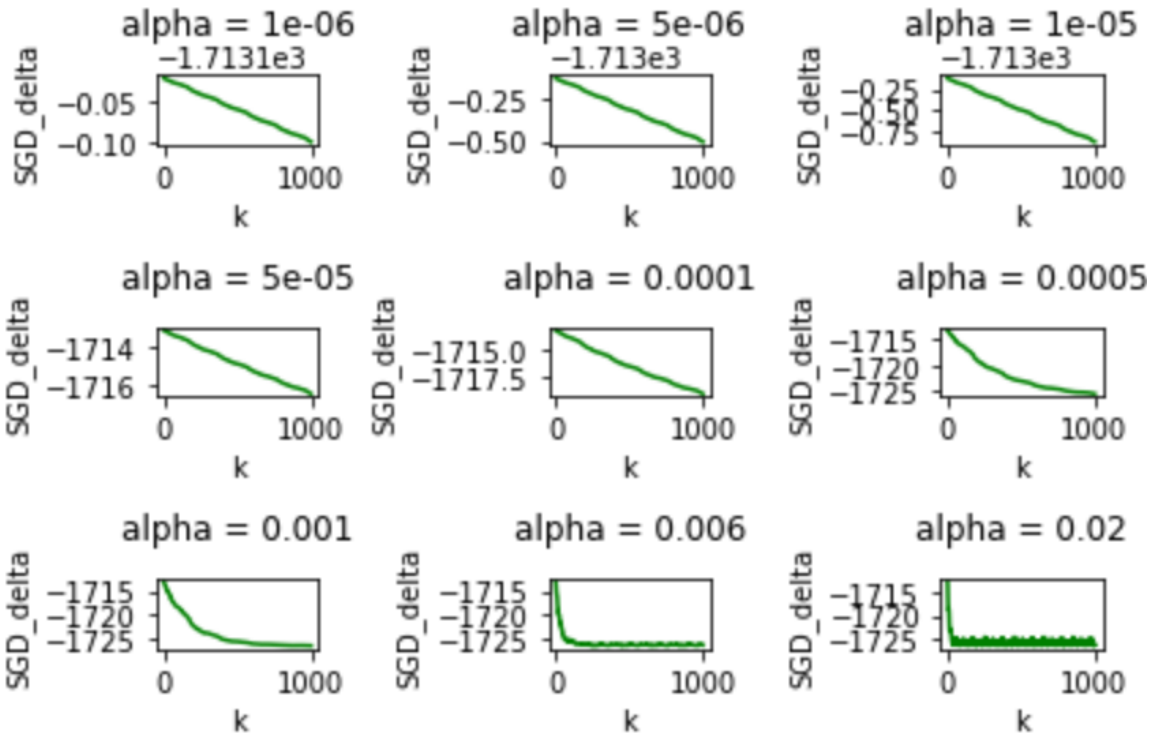
```
# beta values for GD steps
def betas(b0, x, y, xt, alpha):
    b = b0
    Betas = []
    Betas.append(b)
    # we run the algorithm for 1000 epochs
    for i in range(1, 1001):
        B = b - alpha*(-2 * xt @ (y - x @ b) + 200 * b)/200
        b = B
        Betas.append(B)
    return Betas

# delta values for GD steps
def deltas(b0, x, y, xt, loss_value, alpha):
    Deltas = []
    Betas = betas(b0, x, y, xt, alpha)

    for B in Betas:
        loss = np.linalg.norm(y - x @ B, ord = 2)**2/200 + 0.5 * np.linalg.norm(B, ord = 2)**2
        d = loss - loss_value
        Deltas.append(d)
    return Deltas

# function to generate 3 x 3 grid plot
def grid_plot(b0, x, y, xt, loss_value, alpha):
    D = deltas(b0, x, y, xt, loss_value, alpha)
    plt.plot(np.arange(0, len(D)), D, color = 'green')
    plt.xlabel("k")
    plt.ylabel("delta^(k)")
    plt.title("alpha:" + str(alpha), pad = 15)

i = 1
for alpha in alpha_values:
    plt.subplot(3, 3, i)
    grid_plot(b0, x, y, xt, loss_hat, alpha)
    i += 1

plt.tight_layout()
plt.show()
```

MSE train and test code for GD:

```
alpha = 0.01
b = betas(b0, x, y, xt, alpha)
beta = b[1000] # epochs = 1000

MSE_train = (np.linalg.norm(y - x @ beta, ord = 2)**2) / 200
Y_test_values = Y_test.values.reshape(200, 1)
X_test_values = X_test.values.reshape(200, 7)
MSE_test = (np.linalg.norm(Y_test_values - X_test_values @ beta, ord = 2)**2) / 200

# Train MSE
print(MSE_train)

# Test MSE
print(MSE_test)
```

## (h)

3 x 3 gridplot for SGD implementation:



I believe the best step-size is when k = 7, i.e. when $\alpha = 0.006$**.** This is because for α = 0.02, which may produce the lowest SGD delta values, there seems to be some noticeable noise, hence the optimal step-size is when $\alpha = 0.006$.

MSE_train = 4.661749176041872

MSE_test = 4.447228989660328

Code for SGD implementation:

```
alpha_values = [0.000001, 0.000005, 0.00001, 0.00005, 0.0001, 0.0005, 0.001, 0.006, 0.02]

def betas(b0, X_train, Y_train, alpha):
    b = b0
    SGD_beta = []
    SGD_beta.append(b)
    for j in range(1, 1001):
        j = j % 200
        if j == 0:
            j = 200
        X = X_train.loc[j - 1]
        Y = Y_train.loc[j - 1]

        x = X.values.reshape(7, 1)
        x_t = x.T

        B = b - alpha * (-2 * x @ (Y - x_t @ b) + b)
        SGD_beta.append(B)
        b = B
    return SGD_beta

def deltas(b0, X_train, Y_train, loss_hat, alpha):
    y = Y_train.values.reshape(200, 1)
    x = X_train.values
    SGD_delta = []
    SGD_beta = betas(b0, X_train, Y_train, alpha)
    for B in SGD_beta:
        loss = np.linalg.norm(y - x @ B, ord = 2)**2 / 200 + 0.5 * np.linalg.norm(B, ord = 2)**2
        delta = loss - loss_hat
        SGD_delta.append(delta)
    return SGD_delta

def SGD_plot(b0, X_train, Y_train, loss_hat, alpha):
    SGD_delta = deltas(b0, X_train, Y_train, loss_hat, alpha)
    plt.plot(np.arange(0, len(SGD_delta)), SGD_delta, color = 'green')
    plt.xlabel("k")
    plt.ylabel("SGD_delta")
    plt.title("alpha = " + str(alpha), pad = 15)

z = 1
for alpha in alpha_values:
    plt.subplot(3, 3, z)
    SGD_plot(b0, X_train, Y_train, loss_hat, alpha)
    z = z + 1

plt.tight_layout()
plt.show()
```

MSE train and test code for SGD:

```
SGD_Beta = betas(b0, X_train, Y_train, 0.006)
beta_values = SGD_Beta[1000]

MSE_train = (np.linalg.norm(y - x @ beta_values, ord = 2)**2) / 200
MSE_test = (np.linalg.norm(Y_test_values - X_test_values @ beta_values, ord = 2)**2) / 200

# Train MSE
print(MSE_train)

# Test MSE
print(MSE_test)
```

(i)

**GD (Train MSE) = 4.558906724365393**

**GD (Test MSE) = 4.38042918327184**

**SGD (Train MSE) = 4.661749176041872**

**SGD (Test MSE) = 4.447228989660328**

From the results of the train and test mean squared error of GD and SGD, we see that overall the GD results have lower mse, therefore I prefer the GD algorithm compared to SGD. However, SGD reaches convergence much quicker than GD so we might prefer to use SGD if we want a faster algorithm. On the contrary, SGD is computationally expensive to compute the gradient for the entire dataset. However, in SGD the algorithm calculates an approximation of the gradient, not the actual gradient like that of GD, thus supporting the results of the train and test mse, where GD mse is lower than SGD.

(j)

$$\hat{\beta}_1 = \underset{\beta_1}{\operatorname{argmin}}\, L(\beta_1, \dots, \beta_p)$$

Let $\beta = (\beta_1, \beta_2, \dots, \beta_p)$

Using the ridge regression loss function:

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

from hint

$$= \frac{1}{n}\|y - X_j\beta_j - X_{-j}\beta_{-j}\|_2^2 + \Phi\|\beta\|_2^2$$

This is because all values of $X_{-j}\beta_{-j}$ are known to us, thus achieving a constant vector.

$$= \frac{1}{n}\|y - X_j\beta_j - a\|_2^2 + \Phi\|\beta\|_2^2 \quad \text{where } a \text{ is a constant and } a \in \mathbb{R}^n$$

$$= \frac{1}{n}\sum_{i=1}^{n}[(y_i - X_{ij}\beta_{ij} - a_i)^2 + \Phi(\beta_j^2 + c)] \quad \text{where } c \text{ is a constant and } c \in \mathbb{R}^n$$

$$= \frac{1}{n}\sum_{i=1}^{n}[y_i - X_{i1}\beta_i - a_i)^2 + \Phi(\beta_1^2 + c)]$$

$\beta_1$ is unknown for $\beta$

Taking the derivative of this expression wrt $\beta$ and setting to 0:

$$\nabla L(\beta) = \frac{1}{n}\sum_{i=1}^{n}[-2x_i(y_i - X_{i1}\beta_1 - a_i) + 2\beta_1\Phi]$$

$$= \sum_{i=1}^{n}[-2x_{i1}(y_i - X_{i1}\beta_1 - a_i)] + 2n\beta_1\Phi = 0$$

$$= \sum_{i=1}^{n} 2x_{i1}^2\beta_1 + \sum_{i=1}^{n}(-2x_{i1}y_i + 2x_i a_i) + 2n\beta_1\Phi = 0$$

$$= \beta_1\left(\sum_{i=1}^{n} x_{i1}^2 + n\Phi\right) + \sum_{i=1}^{n}(-x_{i1}y_i + x_i a_i) = 0$$

Now, $\beta_1 = \dfrac{-\sum_{i=1}^{n}(-2x_{i1}y_i + 2x_{i1}a_i)}{\sum_{i=1}^{n} 2x_{i1}^2 + 2n\Phi}$

$$\beta_1 = \frac{\sum_{i=1}^{n}(y_i - a_i)}{\sum_{i=1}^{n} x_{i1}^2 + n\Phi}$$

Now we take the double derivative and prove that this expression has a minimum

$$\nabla L^2(\beta) = \frac{1}{n}\sum_{i=1}^{n}(2x_{i1}^2 + 2\Phi)$$ and since $\Phi > 0$, we can confirm that $\nabla L^2(\beta) > 0$ for all $\beta \in \mathbb{R}^n$

Therefore $\hat{\beta}_1 = \underset{\beta_1}{\operatorname{argmin}}\, L(\beta_1, \beta_2, \dots, \beta_p) = \dfrac{\sum_{i=1}^{n}(y_i - (X_{-1}\beta_{-1})_i)}{\sum_{i=1}^{n} x_{i1} + n\Phi}$

and following from this we get:

$$\hat{\beta}_j = \underset{\beta_j}{\operatorname{argmin}}\, L(\beta_1, \beta_2, \dots, \beta_p) = \frac{\sum_{i=1}^{n}(y_i - (X_{-j}\beta_{-j})_i}{\sum_{i=1}^{n} x_{ij} + n\Phi}$$

(k)

Couldn't do this question, but this was roughly my idea:

```
def total_betas(X, y, i, c, size):
    sum = 0
    for i in range(size):
        sum = X[i][y] * # not sure what to do here


def minimisation(X, y, size):
    betas = np.ones(7)
    Betas = []
    Deltas = []
    # Terminate the algorithm after 10 cycles
    for i in range(10):
        for i in range(7):
            # Matrix multiplication, but delete the first row
            constant = np.matmul(np.delete(X, i, 1), np.delete(beta, i, 0))
            betas[i] = total_betas(X, y, i, c, size)
            Betas.append(betas)
            Deltas.append()
            # not sure what to do after this

# Intuition: Produce grid plots indicating the batch GD and SGD and their comparisons. Then using the implementation from
# the previous question and generate MSE train and MSE test which will take in the total sum of the betas (updating).
```

### (l)

[ Reference: Lab 1, Q2: Extended Exercise: LASSO vs. Ridge ]

The standardisation for the entire data set then splitting the data set into train and test sets will create more reliable results for parts (e) - (k). The objective of Ridge regression is to place a penalty on the size of the weight vector $\Phi||\beta||^2$. A coefficient in ridge regression can be arbitrarily large because of the magnitude of the feature $j$ that affects $\beta_j$, despite the feature not being imperative. If we want the comparison across features to be less arbitrary, we need to standardise them on the same scale, thus creating more consistent results across each implementation of GD and SGD which can be seen from (e) - (k). If we did not standardise the data set in (d), then we would see $\beta$coefficients which may be arbitrarily large.