Homework 2

Abigail Wright

STA-4364

$$f(x_1, x_2) = 2x_1^2 + 2x_1x_2 + x_2^2 + x_1 - x_2$$

Problem 1

1.

$$f(x_1, x_2) = \frac{1}{2}\begin{pmatrix} x_1 & x_2 \end{pmatrix}\begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} b_1 & b_2 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

Expanding

$$f(x_1, x_2) = \frac{1}{2}\begin{pmatrix} x_1 & x_2 \end{pmatrix}\begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} b_1 & b_2 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

$$f(x_1, x_2) = \frac{1}{2}\left(Q_{11}x_1^2 + 2Q_{12}x_1x_2 + Q_{22}x_2^2\right) + b_1x_1 - b_2x_2$$

Thus:

$$2x_1^2 + 2x_1x_2 + x_2^2 + x_1 - x_2$$

$$Q_{11} = 4$$

$$Q_{12} = 1$$

$$Q_{22} = 2$$

Thus,

$$Q = \begin{bmatrix} 4 & 1 \\ 1 & 2 \end{bmatrix}$$

$$b = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

Defined as:

```
def compute_f():
    x = np.array([1, 1])
    x_T = np.array([[1],
                    [1]])

    Q11 = 4
    Q12 = 1
    Q21 = 1
    Q22 = 2
    Q = np.array([[Q11, Q12], [Q21, Q22]])

    b1 = 1
    b2 = -1
    b = np.array([b1, b2])
    return Q, b
```

2.

The gradient of $f(x_1, x_2)$ is given by: $\nabla f = Qx + b$
The first derivative with respect to x_1:

$$f'(x_1, x_2) = \frac{d}{dx_1}(2x_1^2 + 2x_1x_2 + x_2^2 + x_1 - x_2) = 4x_1 + 2x_2 + 1 \qquad (1)$$

The first derivative with respect to x_2:

$$f'(x_1, x_2) = \frac{d}{dx_2}(2x_1^2 + 2x_1x_2 + x_2^2 + x_1 - x_2) = 2x_1 + 2x_2 - 1 \qquad (2)$$

Where $Qx + b = \begin{bmatrix} 4x_1 + 2x_2 + 1 \\ 2x_1 + 2x_2 - 1 \end{bmatrix}$

Utilizing the code:

```
def f(x1,x2):
    return 2 * x1**2 + 2 * x1 * x2 + x2**2 + x1 - x2

def gradient_f(x1,x2):
    return np.array([4 * x1 + 2 * x2 + 1, 2 * x1 + 2 * x2 -1])
```

3.

The formula for the gradient requires us to define our original function and first derivative of the function. Which is defined in our code as:

```
def f(x1,x2):
    return 2 * x1**2 + 2 * x1 * x2 + x2**2 + x1 - x2

def gradient_f(x1,x2):
    return np.array([4 * x1 + 2 * x2 + 1, 2 * x1 + 2 * x2 - 1])
```

Where $f(x_1, x_2) = 2x_1^2+2x_1x_2+x_2^2+x_1-x_2$ and $\nabla(f(x_1, x_2) = \begin{bmatrix} 4x_1 + 2x_2 + 1 \\ 2x_1 + 2x_2 - 1 \end{bmatrix}$

4.

The steps used for the Steepest Descent are as follows:

- Set counter $k = 0$ iterations.
- Compute $\nabla f(x_k)$.
- Choose λ_k.
- Update $x_{k+1} = x_k - \lambda_k \nabla f(x_k)$.
- Continue until $||\nabla f(x_k)|| <$ tolerance.

To find the minimizer of f using gradient descent with exact line search, we define our function for each iteration by:

```
def g(lambda_k, x, r):
    return f(x - lambda_k * r)
```

We also define our steepest descent function by:

```
def exact_line(f, gradient_f, x0, tol=1.e-8, maxit=100):
    x = np.array(x0)
    r = gradient_f(x)
    k = 0
    while np.abs(npl.norm(r)) > tol and k < maxit:
```

```
        lambda_k = spo.golden(lambda l: g(l, x, r))
        x = x - lambda_k * r
        r = gradient_f(x)
        k += 1
    return x, k
```

In this case I selected $x_0 = [4.0, 2.0]$ which resulted in minimizer $x : [-1.0, 1.49999999]$ with 19 iterations before convergence.

5.

Here when it comes to fixed step-size we will follow similar steps as the exact line search with some updates:

- Choose a set value for λ_k.
- Compute $\nabla f(x_k)$.
- Update $x_{k+1} = x_k - \lambda_k \nabla f(x_k)$.
- Continue until convergence is reached

The code used to find the minimizer of f using gradient descent with fixed step-size:

```
    def g(lambda_k,x,r):
    return f(x - lambda_k*r)

def fixed_step(f, gradient_f, x0, tol=1.e-6, maxit=100):
    step = [x0]
    x = x0
    r = gradient_f(x)
    lambda_k = 0.2

    for i in range(maxit):
        diff = lambda_k * r
        if npl.norm(diff)<tol:
            break
        x = x - diff
        r = gradient_f(x)
```

```
        step.append(x) ## tracking
    return step, i + 1
```

In this case I selected $x_0 = [2.0, 1.0]$ with $\lambda = 0.2$ which resulted in minimizer $x : [-0.99999997, 1.49999994]$ with 105 iterations before convergence.

6.

Find the minimizer of f using backtracking we must follow the steps:

- Set counter $k = 0$ iterations.
- Make an initial guess for $\boldsymbol{x_0}$
- Choose an initial $\alpha = 1$.
- Update $\lambda_{k+1} = \beta\lambda - k$
- Go until $f(x_{k+1} - \lambda_k \nabla f(x_k)) \leq f(x_{k+1} - \alpha\lambda_k ||\nabla f(x_k)||^2)$
- Calculate $x_{k+1} = x_k - \lambda_k \nabla f(x_k)$ and update $k = k + 1$
- Continue until $||\nabla f(x_k) <$ tolerance $||$

The code used to find the minimizer of f using gradient descent with backtrack line search included defining step-size, gradient descent, and backtracking functions:

```
def step_size(f, gradient_f, x):
alpha = 1.0
beta = 0.8
r = gradient_f(x)
while f(x - alpha*r) > (f(x) - 0.5*alpha*lp.norm(r)**2):
    alpha *= beta
return alpha

def g(lambda_k,x,r):
    return f(x - lambda_k*r)

def back_track(f, gradient_f, x0, tol=1.e-8, maxit=100):
    x = np.array(x0)
```

```
    r = gradient_f(x)
    k = 0
    while npl.norm(r) > tol and k < maxit:
        lambda_k = step_size(f, gradient_f, x)
        x = x - lambda_k * r
        r = gradient_f(x)
        k += 1
    return x, k
```

In the case of backtracking line search, our convergence was reached much faster when compared to other methods. For x_0 the point $[2.0, 1.0]$ was selected with $\alpha = 1$ and $\beta = 0.08$. Which resulted in minimizer $x : [-1.0, 1.5]$ with 61 iterations before convergence.

Problem 2

1.

First, we must find the first derivative with respect to β_0 and β_1:
$g(\beta_0, \beta_1) = \begin{bmatrix} \frac{\partial l}{\partial \beta_0} \\ \frac{\partial l}{\partial \beta_1} \end{bmatrix}$ where $y = \begin{bmatrix} 3.8 \\ 6.5 \\ 11.5 \end{bmatrix}$ and $x = \begin{bmatrix} 1 & 5 \\ 1 & 6 \\ 1 & 7 \end{bmatrix}$

The derivative with respect to β_0 is: $\frac{\partial l}{\partial \beta_0}$

$= -\frac{1}{3}\left[(3.8 - \beta_0 - 5\beta_1) + (6.5 - \beta_0 - 6\beta_1) + (11.5 - \beta_0 - 7\beta_1)\right]$

The derivative with respect to β_1 is: $\frac{\partial l}{\partial \beta_1}$

$= -\frac{1}{3}[(5)(3.8 - \beta_0 - 5\beta_1) + (6)(6.5 - \beta_0 - 6\beta_1) + (7)(11.5 - \beta_0 - 7\beta_1)]$

The array of the y-intercept and slope is defined as: $\boldsymbol{\beta} = \begin{bmatrix} \beta_0 & \beta_1 \end{bmatrix}^T$

The gradient is: $g(\boldsymbol{\beta}) = \frac{1}{N} x^T e$

Here we must create a function for $g(\beta)$:

```
def grad_beta(x, y, B_0, B_1):
```

```
n = len(y)
residuals = (y - (B_0 + (B_1 * x)))

df_yint = (1/n) * np.sum(residuals)
df_slope = (1/n) * np.sum((residuals) * x)
g_B = np.array([[df_yint],
           [df_slope]])
return g_B
```

Here we see the gradient is $[7.26666667, 46.16666667]$.

2.

To find the minimizer of L using gradient descent with a fixed step-size:

- $g(\beta_0, beta_1) = \begin{bmatrix} \frac{dl}{d\beta_0} \\ \frac{dl}{d\beta_1} \end{bmatrix}$
- $\beta_{=}(\beta_0, \beta_1)^T$
- $\beta^{k+1} = \beta^{(k)} - \lambda \nabla g(\beta^k)$

Here using fixed step-size we are choosing a constant value for λ in order to move in direction of the gradient $= 0$.

Utilizing the defined code:

```
def gradient_descent_fixed_step(x, y, B_0_init,
B_1_init, lambda_k, tol=1e-8, maxit=200):
B = np.array([B_0_init, B_1_init])
steps = [B.copy()]
k = 0

for _ in range(maxit):
    gradient = grad_beta(x, y, B[0], B[1])
    B_new = B - lambda_k * gradient

    steps.append(B_new.copy())

    if np.linalg.norm(gradient) < tol:
```

```
            break

        B = B_new
        k += 1

    return B, k
```

3.

Here we will define the backtracking function by:

```
    def gradient_descent_backtracking(x, y, B_0_init,
    B_1_init, tol=1e-8, maxit=200):
        B = np.array([B_0_init, B_1_init])
        step = [B.copy()]
        k = 0

        for _ in range(maxit):
            gradient = grad_beta(x, y, B[0], B[1])
            lambda_k = backtracking_line_search(x, y, B, gradient)
            B_new = B - lambda_k * gradient

            step.append(B_new.copy())

            if np.linalg.norm(gradient) < tol:
                break

            B = B_new
            k += 1

        return B, k
```

Problem 3

1.

Write a function for the gradient $g(\beta)$
Here we are going to find the $g(\beta)$ by utilizing matrix multiplication:

Where $y = \begin{bmatrix} 3.8 \\ 6.5 \\ 11.5 \end{bmatrix}$, $x = \begin{bmatrix} 1 & 5 \\ 1 & 6 \\ 1 & 7 \end{bmatrix}$, and $\boldsymbol{\beta} = \begin{bmatrix} \beta_0 & \beta_1 \end{bmatrix}^T$

To find the gradient using matrix multiplication: $g(\boldsymbol{\beta}) = \frac{1}{n}\mathbf{X}^T\boldsymbol{e}$

Utilizing $\boldsymbol{e} = \boldsymbol{y} - \boldsymbol{x\beta}$ Here we must create a function for $g(\beta)$:

```
def grad_beta(x, y, B):
    B = B_0, B_1
    n = len(y)
    e = y - np.dot(x, B)
    g_B = (1/n) * np.dot(x.T, e)
    return g_B
```

Here we see the gradient is $[7.26666667, 46.16666667]$.

2.

To find the minimizer of L using gradient descent with fixed step-size with matrix multiplication:

- $g(\beta_0, \beta 1) = \begin{bmatrix} \frac{dl}{d\beta 0} \\ \frac{dl}{d\beta 1} \end{bmatrix}$
- $\boldsymbol{\beta} = (\beta_0, \beta_1)^T$
- $e = y - x\beta$
- $g(\beta) = \frac{1}{n}x^T e$

The code for fixed step-size with matrix multiplication:

```
def gradient_descent_fixed_step(x, y, B_0_init, B_1_init,
lambda_k, tol=1.e-8, maxit=10000):
B = np.array([B_0_init, B_1_init])
steps = [B.copy()]
```

```
    losses = [loss_function(x, y, B[0], B[1])]
    k = 0

    for _ in range(maxit):
        gradient = grad_beta(x, y, B[0], B[1])
        B_new = B - lambda_k * gradient

        steps.append(B_new.copy())
        loss = loss_function(x, y, B_new[0], B_new[1])
        losses.append(loss)

        if np.abs(losses[-1] - losses[-2]) < tol:
            break

        B = B_new
        k += 1

    return steps, losses, k
```

Resulting in $\beta_0 = -15.831477$ and $\beta_1 = 3.849696$. Which was reached after 17058 iterations utilizing fixed step size.

3.

To find the minimizer of L using gradient descent with backtracking we will choose an α between 0.01 and 0.3 and a β between 0.1 and 0.3.
To define our required functions for backtracking is as follows:

```
def backtracking_line_search(x, y, B, gradient, alpha=0.5, beta=0.8):
    t = 1.0
    while np.linalg.norm(grad_beta(x, y, B - t * gradient))
    > (1 - alpha * t) * np.linalg.norm(gradient):
        t *= beta
    return t

def gradient_descent_backtracking(x, y, B_init, tol=1e-6, max_iter=10000):
    B = B_init.copy()
    steps = [B.copy()]
```

```
    for i in range(max_iter):
        gradient = grad_beta(x, y, B)
        step_size = backtracking_line_search(x, y, B, gradient)
        B_new = B - step_size * gradient
        steps.append(B_new.copy())

        if np.linalg.norm(B_new - B) < tol:
            break

        B = B_new

    return steps, i + 1
```

Resulting in $\beta_0 = -0.005954$ and $\beta_1 = 1.258902$. Which was reached after 18 iterations utilizing backtrackihg exact line search.