**Life Expectancy Linear Regression**

1a)

As age is independent and the expectation of life at age is dependent on the age, I made age the x-axis and expectation the y-axis.

A graph with a blue line

Description automatically generated

Equation: Expectancy of life at age x = 0.3429 \* age\_x

1b)

From the information given, we can conclude that .

Using the error formula, ,

Subbing in

Expanding the equation from summation:

Next, using Chain Rule:

In this equation, I have also replaced n with 500, which is the number of samples available in the dataset as shown in my code.

1c)

b = 10

alpha = 0.00005

epsilon = 0.001

max\_iters = 1000

def E(b):

    summation = 0

    for rownum in np.arange(len(df)):

        x = cleaned\_data.loc[rownum, 'age\_x']

        y = cleaned\_data.loc[rownum, 'Expectation of life at age x']

        summation += (y - b\*x)\*\*2

    return 1/len(cleaned\_data) \* summation

def Eprime(b):

    summation = 0

    for rownum in np.arange(len(df)):

        x = cleaned\_data.loc[rownum, 'age\_x']

        y = cleaned\_data.loc[rownum, 'Expectation of life at age x']

        summation += x \* (y - b\*x)

    return -2/len(cleaned\_data) \* summation

model1\_gradientdesc = pd.DataFrame(columns=['b', 'E(b)'])

for iter in range(max\_iters):

    new\_b = b - alpha \* Eprime(b)

    print(f"Iteration {iter + 1}: b-value = {new\_b}, E(b) = {E(new\_b)}")

    diff = abs(E(new\_b) - E(b))

    model1\_gradientdesc = pd.concat([model1\_gradientdesc, pd.DataFrame({'b': [new\_b], 'E(b)': [E(new\_b)]})], ignore\_index=True)

    if diff < epsilon:

        print(f"The local minimum occurs at b = {new\_b}")

        break

    b = new\_b

print(f"In {iter} iterations, the local minimum occurs at b = {new\_b} with a minimum error of E(b) = {E(new\_b)}")

In 24 iterations, the local minimum occurs at b = 0.3434083492907686 with a minimum error of E(b) = 1640.8324998112982

After plotting, I also tried improving the algorithm by using grid search to obtain the best parameters.

def model1hyperparam(b, alpha, epsilon, max\_iters):

    b\_val\_array = b

    alpha\_array = alpha

    epsilon = epsilon

    max\_iters = max\_iters

    first\_test = pd.DataFrame(columns=['alpha', 'Starting b-value', 'Iterations', 'b', 'E(b)'])

    final\_test = pd.DataFrame(columns=['alpha', 'Starting b-value', 'Iterations', 'b', 'E(b)'])

    for alpha in alpha\_array:

        for starting\_b in b\_val\_array:

            b = starting\_b

            for iter in range(max\_iters):

                b\_new = b - alpha \* E\_prime(b)

                diff = abs(E(b\_new) - E(b)) # stopping criterion

                if diff < epsilon: # break if converge

                    first\_test.loc[len(first\_test)] = [alpha, starting\_b, iter + 1, b\_new, E(b\_new)]

                    break

                b = b\_new

    first\_test.sort\_values(by=['E(b)'], inplace=True)

    finalTestVal = first\_test.iloc[0]['Starting b-value'] # Fetches

    b\_FinalTest = np.arange(finalTestVal/2, finalTestVal \* 2, finalTestVal/10) # Creates a range of b values around the best b value

    for starting\_b in b\_FinalTest: # loop through b values

            b = starting\_b

            for iter in range(max\_iters):

                b\_new = b - alpha \* E\_prime(b)

                diff = abs(E(b\_new) - E(b)) # stopping criterion

                if diff < epsilon: # break if converge

                    final\_test.loc[len(final\_test)] = [alpha, starting\_b, iter + 1, b\_new, E(b\_new)]

                    break

                b = b\_new

    final\_test.sort\_values(by=['E(b)'], inplace=True)

    return final\_test

results\_df = model1hyperparam([-1000, -100, -1, 0, 1, 100, 1000], [0.00005], 0.001, 1000)

# results\_df.sort\_values(by=['E(b)'], inplace=True) # sort results by error function

print(results\_df)

These are the best results:

alpha Starting b-value Iterations b E(b)

0.00005 1.0 19.0 0.343289 1640.832186

1cii) Therefore, model 1 equation is: (5s.f)

2a)

A graph of a line graph

Description automatically generated

2b) Equations given: ,

From both equations,

Deriving

Using Summation:

Deriving ,

Using summation:

2ci) a = 80

b = 0

alpha = 0.00001

epsilon = 0.001

max\_iters = 1000

def E(a, b):

    summation = 0

    for rownum in np.arange(len(cleaned\_data)):

        x = cleaned\_data.loc[rownum, 'age\_x']

        y = cleaned\_data.loc[rownum, 'Expectation of life at age x']

        summation += (y - a - b\*x)\*\*2

    return 1/len(cleaned\_data) \* summation

def partial\_E\_a(a, b): # wrt a

    summation = 0

    for rownum in np.arange(len(cleaned\_data)):

        x = cleaned\_data.loc[rownum, 'age\_x']

        y = cleaned\_data.loc[rownum, 'Expectation of life at age x']

        summation += y - a - b\*x

    return -2/len(cleaned\_data) \* summation

def partial\_E\_b(a, b): # wrt b

    summation = 0

    for rownum in np.arange(len(cleaned\_data)):

        x = cleaned\_data.loc[rownum, 'age\_x']

        y = cleaned\_data.loc[rownum, 'Expectation of life at age x']

        summation += x \* (y - a - b\*x)

    return -2/len(cleaned\_data) \* summation

model2\_gradientdesc = pd.DataFrame(columns=['a-value', 'b-value', 'E(a,b)'])

for iter in range(max\_iters):

    new\_a = a - alpha \* partial\_E\_a(a, b)

    new\_b = b - alpha \* partial\_E\_b(a, b)

    diff = abs(E(new\_a, new\_b) - E(a, b))

    print(f"Iteration {iter+1}: a = {new\_a}, b = {new\_b}, E(a,b) = {E(new\_a, new\_b)}")

    model2\_gradientdesc.loc[len(model2\_gradientdesc)] = [new\_a, new\_b, E(new\_a, new\_b)]

    if diff < epsilon:

        print(f"Iteration {iter + 1}: a = {new\_a}, b = {new\_b}, E(a,b) = {E(new\_a, new\_b)}")

        print(f"The local minimum occurs at a = {new\_a}, b = {new\_b}")

        break

    a = new\_a

    b = new\_b

print(f"In {iter} iterations, the local minimum occurs at a = {new\_a} b = {new\_b} with a minimum error of E(b) = {E(new\_a, new\_b)}")

In 94 iterations, the local minimum occurs at a = 79.98713663579773 b = -0.8615302138751975 with a minimum error of E(b) = 6.673488572848425

2cii) Expectancy of life at age x + 79.987(5s.f)

2d)

* I set the initial value for b to be an arbitrary number such as 2 and after a few iterations I noticed E(b) would always converge near the -1 to 0 range. Because the initial value was close to the convergence point, I left it as it is.
* Initially the value of A was set to 0, but later on as I saw that it was converging near 79-80. Hence, I changed the initial value to 80.
* I set the learning rate, α, to be a small value like 0.001 as the values got too large and close to inf when I set it to a high value such as 0.1 when I first tried. Even though it would take longer, it would have a higher chance of finding the minima.
* I chose to perform gradient descent with 1000 iterations as I thought it would take longer to converge due to the small learning rate. However, this was not the case as convergence was reached at the 94th iteration.

3a) I used all the columns, but these are the examples of the values that I inserted.

|  |  |  |  |
| --- | --- | --- | --- |
| Year | age\_x | Expectation of life at age x | LERatio |
| 2022 | 0 | 83 | 83 |
| 2022 | 1 | 82.1 | 82.1 |
| 2022 | 2 | 81.1 | 40.55 |
| 2022 | 3 | 80.1 | 26.7 |
| 2022 | 4 | 79.1 | 19.775 |
| 2022 | 5 | 78.1 | 15.62 |
| 2022 | 6 | 77.1 | 12.85 |
| 2022 | 7 | 76.1 | 10.87143 |
| 2022 | 8 | 75.2 | 9.4 |
| 2022 | 9 | 74.2 | 8.244444 |

c)

a = 90

b = 0

c = 0

alpha = 0.000025

epsilon = 0.001

max\_iters = 1000

def E(a, b, c):

    summation = 0

    for rownum in np.arange(len(cleaned\_data)):

        x = cleaned\_data.loc[rownum, 'age\_x']

        y = cleaned\_data.loc[rownum, 'Expectation of life at age x']

        w = cleaned\_data.loc[rownum, 'LERatio']

        summation += (y - a - b\*x - c\*w)\*\*2

    return 1/len(cleaned\_data) \* summation

def partialE\_a(a, b, c): # 1st derivative of E(a, b, c) with respect to a

    summation = 0

    for rownum in np.arange(len(cleaned\_data)):

        x = cleaned\_data.loc[rownum, 'age\_x']

        y = cleaned\_data.loc[rownum, 'Expectation of life at age x']

        w = cleaned\_data.loc[rownum, 'LERatio']

        summation += y - a - b\*x - c\*w

    return -2/len(cleaned\_data) \* summation

def partialE\_b(a, b, c): # 1st derivative of E(a, b, c) with respect to b

    summation = 0

    for rownum in np.arange(len(cleaned\_data)):

        x = cleaned\_data.loc[rownum, 'age\_x']

        y = cleaned\_data.loc[rownum, 'Expectation of life at age x']

        w = cleaned\_data.loc[rownum, 'LERatio']

        summation += x \* (y - a - b\*x - c\*w)

    return -2/len(cleaned\_data) \* summation

def partialE\_c(a, b, c): # 1st derivative of E(a, b, c) with respect to c

    summation = 0

    for rownum in np.arange(len(cleaned\_data)):

        x = cleaned\_data.loc[rownum, 'age\_x']

        y = cleaned\_data.loc[rownum, 'Expectation of life at age x']

        w = cleaned\_data.loc[rownum, 'LERatio']

        summation += w \* (y - a - b\*x - c\*w)

    return -2/len(cleaned\_data) \* summation

for iter in range(max\_iters): # Gradient Descent

    new\_a = a - alpha \* partialE\_a(a, b, c) # update of a

    new\_b = b - alpha \* partialE\_b(a, b, c) # update of b

    new\_c = c - alpha \* partialE\_c(a, b, c) # update of c

    diff = abs(E(new\_a, new\_b, new\_c) - E(a, b, c)) # stopping criterion

    print(f"Iteration {iter+1}: a = {new\_a}, b = {new\_b}, c= {new\_c}, E(a,b,c) = {E(new\_a, new\_b, new\_c)}")

    if diff < epsilon:

        print(f"The local minimum occurs at a = {new\_a}, b = {new\_b}, c={new\_c}")

        break

    a = new\_a

    b = new\_b

    c = new\_c

print(f"In {iter} iterations, the local minimum occurs at a = {new\_a} b = {new\_b} c = {new\_c} with a minimum error of E(b) = {E(new\_a, new\_b, new\_c)}")

In 256 iterations, the local minimum occurs at a = 89.95746251596272 b = -1.0117203724215524 c = -0.1342391928057935 with a minimum error of E(b) = 26.753217935000535

a\_model1 = 89.957

b\_model1 = -1.0117

w\_model1 = -0.13424

x\_model1 = cleaned\_data['age\_x']

y\_model1 = a\_model1 + b\_model1 \* x\_model1 + w\_model1 \* cleaned\_data['LERatio']

a\_model2 = new\_a

b\_model2 = new\_b

x\_model2 = cleaned\_data['age\_x']

y\_model2 = a\_model2 + b\_model2 \* x\_model2

# Plotting both models

plt.figure(figsize=(10, 6))

# Plot actual values in blue

plt.scatter(x\_model1, cleaned\_data['Expectation of life at age x'], color='blue', label='Actual')

# Plot regression line of Model 1 in red

plt.scatter(x\_model1, y\_model1, color='red', label='Model 2')

# Plot regression line of Model 2 in green

plt.plot(x\_model2, y\_model2, color='green', label='Model 3')

plt.xlabel('Age\_x')

plt.ylabel('Expectation of life at age x')

plt.title('Comparison of Regression Models')

plt.legend()

plt.show()