Brandon Shumin

ECE 8540 Lab 1

Part 1

The calculated fitted line for this section was $\begin{bmatrix} 1 \\ -4.6 \end{bmatrix}$ as shown in Figure 4 from my Python program. This represents a line with a slope of 1 and a y-intercept point of -4.6 as shown overlaid on the relevant data points in Figure 1. The Python code for determining these values is shown in Figure 7.

Part 2

The calculated fitted line for this section was $\begin{bmatrix} 1.81538462 \\ -8.67692308 \end{bmatrix}$ as shown in Figure 5 from my Python program. This represents a line with a slope of 1.81538462 and a y-intercept point of -8.67692308 as shown overlaid on the relevant data points in Figure 2. The Python code for determining these values is shown in Figure 7.

The added data point of (8, 14) alters the slope and intercept of the original data so much due to the fact that it does not fit the general form of the previous data well. This is easy to see when observing the other point (8, 3) also included in the data. These two points lie on the same x-axis point but are 11 units apart on the y-axis and this newly added point appears to be an outlier even simply by looking at the data in Figure 2. This point thus pulled the linear regression line towards it and reduced how well the line fit for the overall model for Part 2 as a result.

Part 3

For this section I decided to model the data as a function of $y = \frac{1}{x}$ as it appeared to most closely resemble a scatter of data similar to this linear model. The calculated fitted line for this section was $\begin{bmatrix} 131.60745051 \\ 10.46374461 \end{bmatrix}$, as shown in Figure 6, which translates to constants in the function $y = m * \frac{1}{x} + b$ where $m = 131.60745051$ and $b = 10.46374461$. The fitted line can be seen overlaid on the relevant data in Figure 3. The Python code for determining these values is shown in Figure 8.

I chose the model $y = m * \frac{1}{x} + b$ because it not only looked like the data just from a graphing perspective, but the way the data is being graphed would tend towards this relationship, as the fewer bites a subject took, the more calories they would consume per bite, and vice versa. This model breaks down as the subjects take more bites, since the model approaches an asymptote of 0 on the x-axis as $\lim_{x \to \infty} x$ and taking a bite with 0 Kcal of nutrition makes no realistic sense. At small amounts of bites, the model still makes reasonable sense, however, as the bites are measured as integers, and thus the largest y-value achievable is when x=1 which would make $y = m + b$ or, theoretically, the total calorie count of the meal in a single bite. This is not extremely likely for a subject to do, but would hold mathematically at least, as that one bite would contain the entire caloric content of the meal.

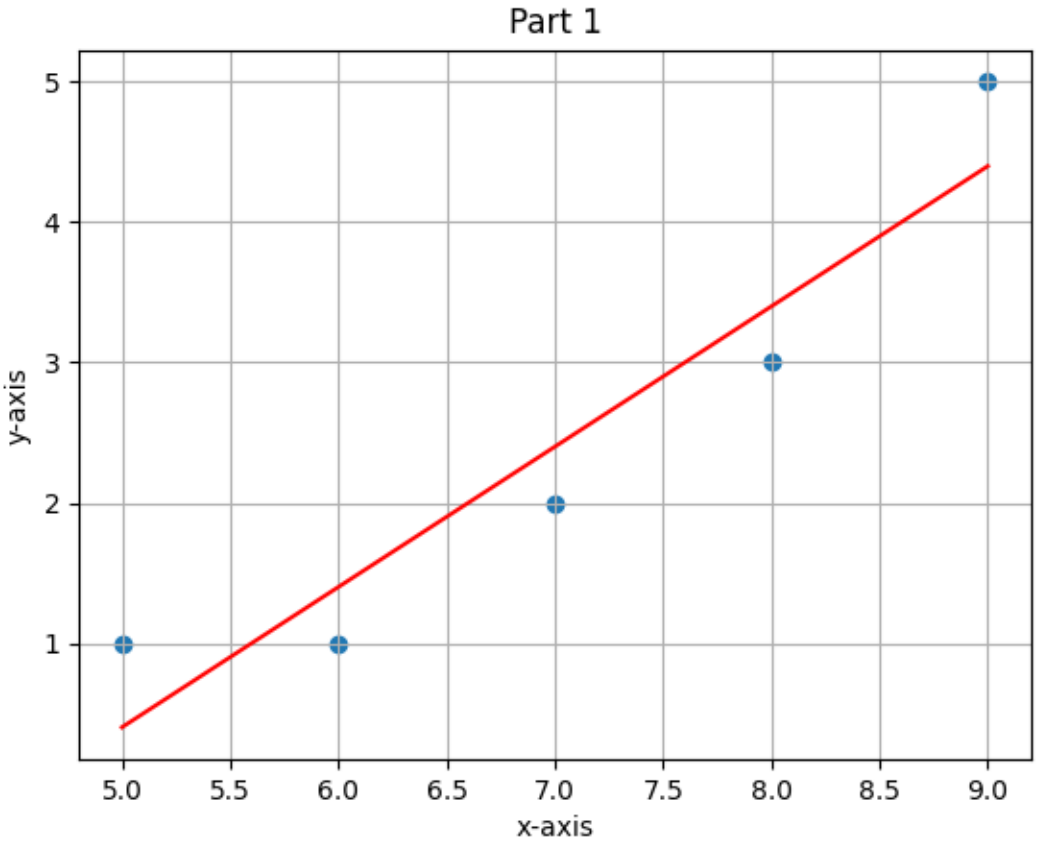


Figure 1: Graphed data from Part 1 including the calculated fitted line

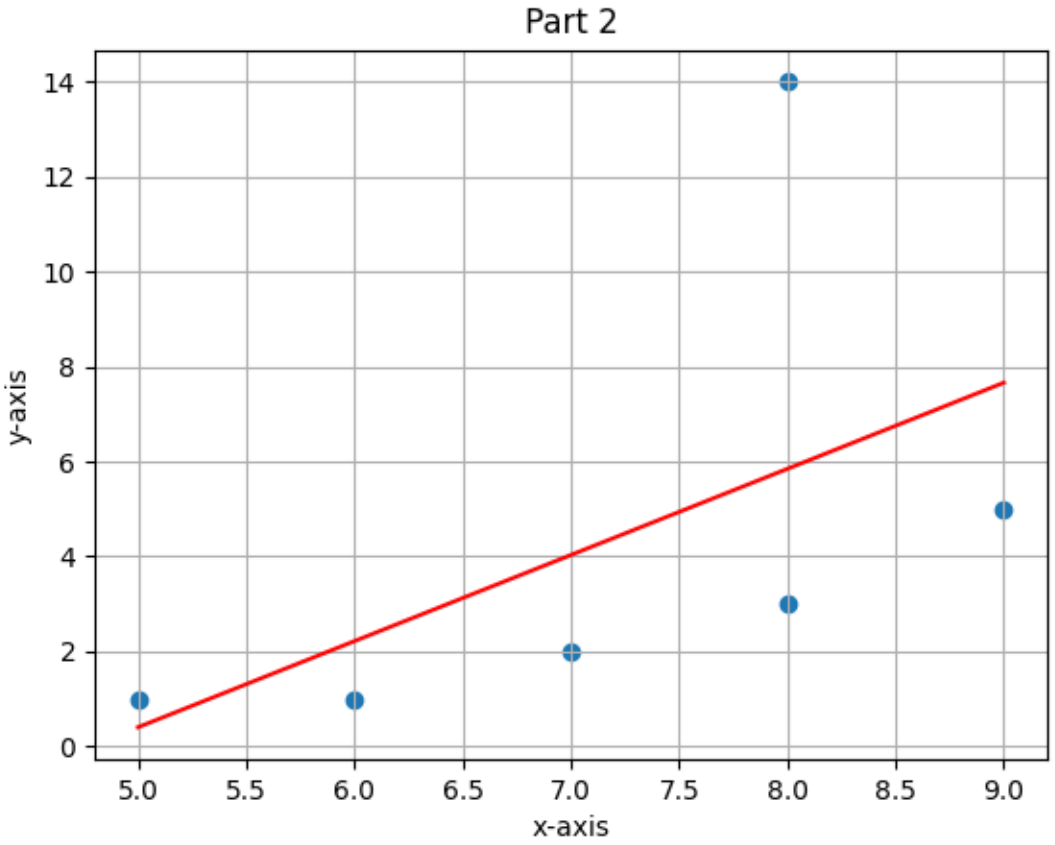


Figure 2: Graphed data from Part 2 including the calculated fitted line

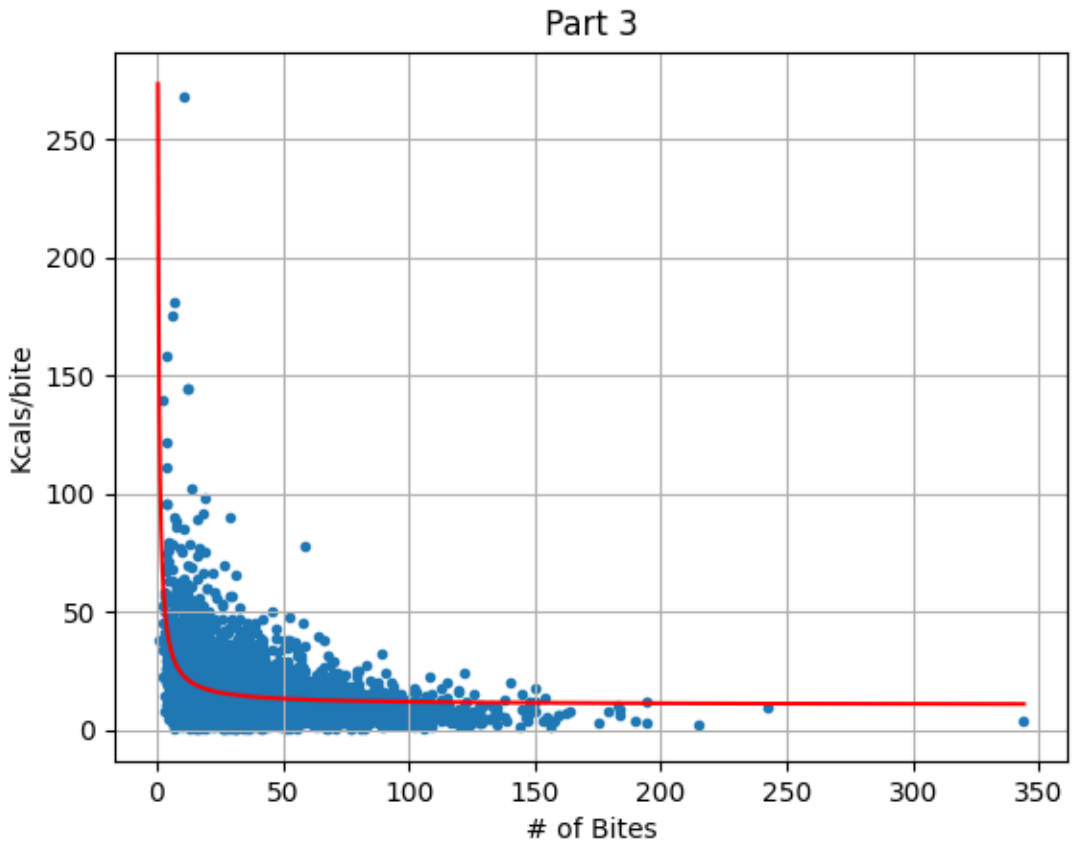


Figure 3: Graphed data from Part 3 including the calculated fitted line

```
Solution to normal equation in part 1:
[[ 1. ]
 [-4.6]]
```

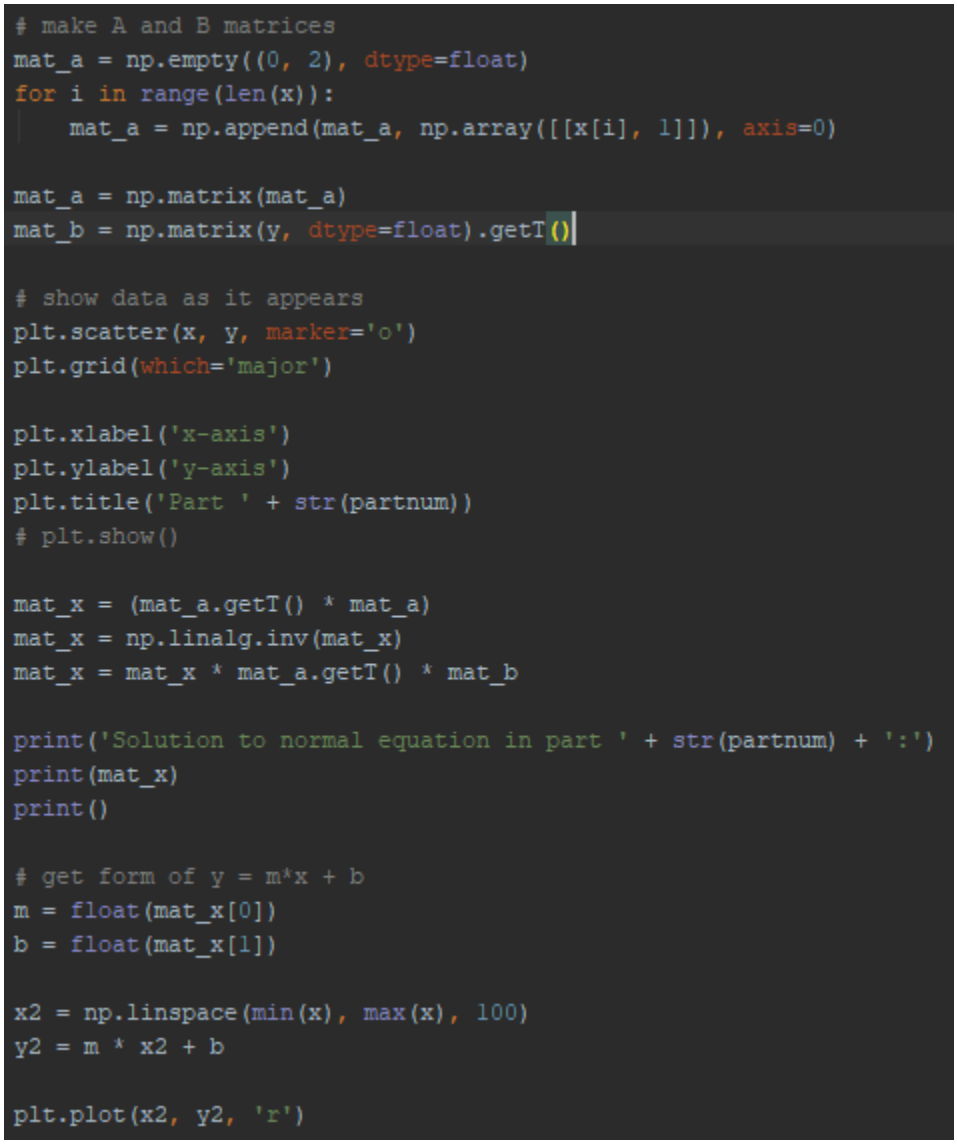
Figure 4: Image of displayed solutions to normal equation from program for part 1

```
Solution to normal equation in part 2:
[[ 1.81538462]
 [-8.67692308]]
```

Figure 5: Image of displayed solutions to normal equation from program for part 2

```
Solution to normal equation in part 3:
[[131.60745051]
 [ 10.46374461]]
```

Figure 6: Image of displayed solutions to normal equation from program for part 3



```
# make A and B matrices
mat_a = np.empty((0, 2), dtype=float)
for i in range(len(x)):
    mat_a = np.append(mat_a, np.array([[x[i], 1]]), axis=0)

mat_a = np.matrix(mat_a)
mat_b = np.matrix(y, dtype=float).getT()

# show data as it appears
plt.scatter(x, y, marker='o')
plt.grid(which='major')

plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.title('Part ' + str(partnum))
# plt.show()

mat_x = (mat_a.getT() * mat_a)
mat_x = np.linalg.inv(mat_x)
mat_x = mat_x * mat_a.getT() * mat_b

print('Solution to normal equation in part ' + str(partnum) + ':')
print(mat_x)
print()

# get form of y = m*x + b
m = float(mat_x[0])
b = float(mat_x[1])

x2 = np.linspace(min(x), max(x), 100)
y2 = m * x2 + b

plt.plot(x2, y2, 'r')
```

Figure 7: Code to determine best fit line for model with data in parts 1 and 2

```
# make A and B matrices
mat_a = np.empty((0, 2), dtype=float)
for i in range(len(x)):
    mat_a = np.append(mat_a, np.array([[1 / x[i], 1]]), axis=0)

mat_a = np.matrix(mat_a)
mat_b = np.matrix(y, dtype=float).getT()

# show data as it appears
plt.scatter(x, y, marker='.')
plt.grid()

plt.xlabel('# of Bites')
plt.ylabel('Kcals/bite')
plt.title('Part ' + str(partnum))

mat_x = (mat_a.getT() * mat_a)
mat_x = np.linalg.inv(mat_x)
mat_x = mat_x * mat_a.getT() * mat_b

print('Solution to normal equation in part ' + str(partnum) + ':')
print(mat_x)
print()

# get form of y = m*(1/x) + b
m = float(mat_x[0])
b = float(mat_x[1])

x2 = np.linspace(0.5, max(x), 10000)
y2 = m * (1 / x2) + b

plt.plot(x2, y2, 'r')
```

Figure 8: Code to determine best-fit line for model with data in part 3