Chapter 2 - Implementing Gradient Descent

Solution Workbook for Student Practical Classes

Dr Rob Collins 2023

2025-01-11

Table of contents

List of Figures

Figure 1: Famed explorer Liu Qiaoxuan calculating route down Yunwushan (c. 1783)

2 Gradient Descent

2.1 Introduction

In this workshop you will first build code that fits a straight line to a data-set. Later in the tutorial I will show how the same algorithm can be used to fit a hyper-planes to higher-dimensional data.

The method used to achieve this is called 'Gradient Descent' - and this is covered in the technical lectures for this course.

In practice, when building Machine Learning models you are unlikely to need to build your own data fitting algorithms. The various Machine Learning libraries include a range of data fitting algorithms. In the vast majority of cases you will choose to use those since they are much easier to use and almost always faster than the implementation below.

However, it is important to have some understanding of how the various fitting algorithms actually work. This is core material for Machine Learning. Without understanding how data fitting actually works many of the algorithms will appear simply as 'magic' - things that seem to work without any real scientific understanding of how they work.

You are thus advised to study this section carefully. Also, consider reading around this subject to learn about other data fitting algorithms.

2.2 Instructions for Students

In this workbook there are regular 'callout' blocks indicating where you should add your own code. They look like this:

In those cases you are required to create the code for the block based on your learning on this course. In some cases the tutor has provided 'clues' towards the code you need to write and in a few places the complete code block.

In working through the following notebook, please do the following:

1. Create an empty Jupyter notebook on your own machine
2. Enter all of the **Python code** from this notebook into **code blocks** in your notebook
3. **Execute each of the code blocks** to check your understanding
4. You **do not need to** replicate all of the explanatory / tutorial text in text (markdown) blocks
5. You **may** add your own comments and description into text (markdown) blocks if it helps you remember what the commands do
6. You **may** add further code blocks to experiment with the commands and try out other things
7. Enter and run as many of the code blocks as you can within the time available during class
8. **After class**, enter and run any remaining code blocks that you have not been able to complete in class

2.3 Load the required libraries

For this practical session we will need the following libraries:

- 'pandas' : Conventionally given the reference name 'pd'
- 'numpy' : Conventionally given the reference name 'np'
- 'random'
- 'seaborne' : Conventionally given the reference name 'sns'
- 'matplotlib.pyplot' : Conventionally given the reference name 'plt'
- 'Axes3D' : from mpl_toolkits.mplot3d

Create a cell to import all of these libraries.

```
import pandas as pd
import numpy as np
import random
import seaborn as sns
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

2.4 Generating some data which we can use to experiment with fitting

The following function generates a set x and y data. The y data has a slope defined by 'A', a constants term (a 'bias' defined by 'K' and a level of random noise defined by 'varience'.

There is one thing to note about the following function which may surprise you. We actually generate two 'columns' (features) of x data. The first column will be populated only with '1's, rather than 'real' (variable / measured) x data. This is a common 'trick' that arises for the following reason:

We will fitting a straight line of the form y = A.x + k. Note that there are two terms to be identified in this case - the 'A' (slope of the line) and 'k' - the constant term or 'bias'. Only one of these ('A') is multiplied by the x values.

As we evaluate trial (hypothesis) versions of fitted values, it will be very convenient to use library-based matrix multiplication operations - which are very fast and easy to use. Creating the 'x' table as two columns enables us to simply perform a matrix operations between our hypothesised values for 'A' and 'k' and the table of 'x' data. In that case, the 'A' will be multiplied by real (variable / measured) x values and our hypothesised 'k' values will all be multiplied by 1 - the value in the first column.

You will see in many lectures and implementations of data-fitting algorithms use this 'trick'. By adding a vector of '1's in the first column, we simply enable more succinct matrix operations. You will see this used later in the workshop.

At this point, create a function to generate data in the form described above:

```
def genData(numPoints, A, K, variance):
    x = np.zeros(shape=(numPoints, 2))
    y = np.zeros(shape=numPoints)
    for i in range(0, numPoints):
        x[i][0] = 1
        x[i][1] = i
        y[i] = (A * i + K) + random.uniform(0, 1) * variance
    return x, y
```

Use the 'genData' function to build a data-set of the form:

$$y = A.x + K + noise$$

Where in this specific case, the values are:

$$y = 2.x + 100 + random.uniform(30)$$

Remember that 'genData' has two return values.

```
x, y = genData(numPoints = 100, A = 2, K = 100, variance = 30)
```

We can now look at the data graphically using a 'scatterplot'. There are several ways of plotting scatterplot in Python but my suggestion here is to use the 'seaborn' and 'matplotlib.pyplot' libraries.

Create a scatterplot chart that displays the data just created.

Note : You may wish to include the '%matplotlib inline' command in your code. This forces the chart to appear as part of your Jupyter notebook rather than in a separate window

```
%matplotlib inline

fig = plt.figure(figsize=(5, 5))
fig = sns.scatterplot(x = x[:,1], y=y)
fig.set(xlabel ="x",
        ylabel = "y",
        title ='Graph of y = 2.x + 100 + random.uniform(30)')

plt.plot()
plt.show();
```

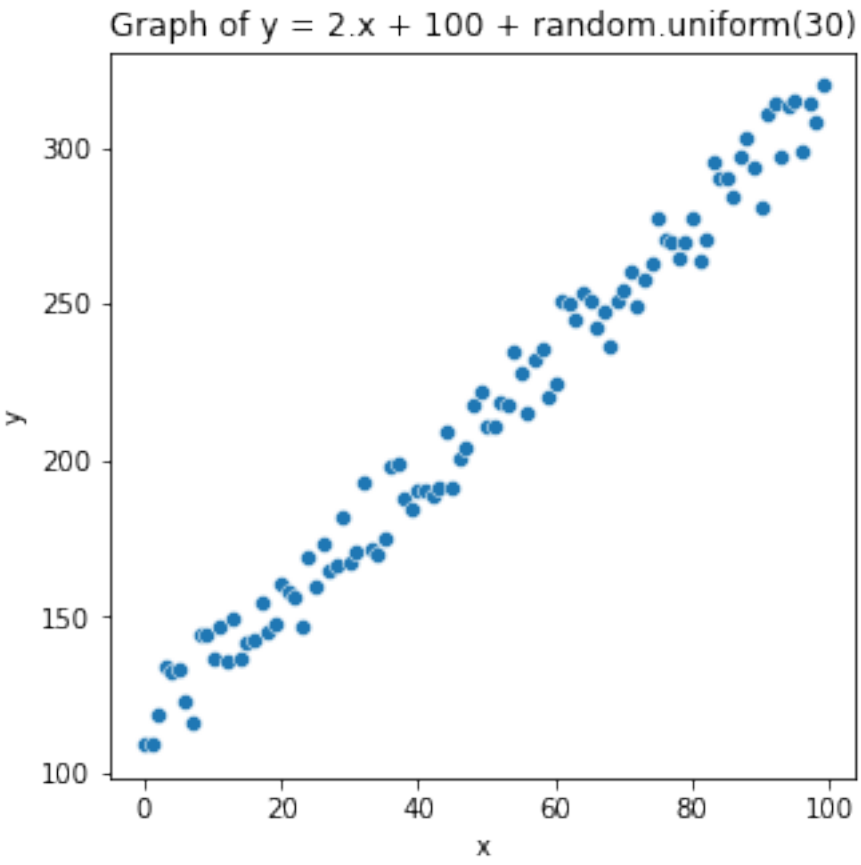

Figure 2: Scatter-plot of Synthetic linear data with some Gaussian noise

..and as explained above, the 'shape' of x is 'm' rows and 2 columns. You can demonstrate that using the numpy 'shape' function:

```
# The length and width of the source dataframe
m, n = np.shape(x)
print(f"m (Length)= {m}, and n (width)= {n}")
```

```
m (Length)= 100, and n (width)= 2
```

Add code to display the first few rows of x and note that the first column of x data contains only '1's

```
print(x[0:5])
```

```
[[1. 0.]
 [1. 1.]
```

```
[1. 2.]
[1. 3.]
[1. 4.]]
```

2.5 Fitting a model to the data

2.5.1 Initial parameter values

We set up the basic parameters for the data fitting. 'alpha' is the learning rate - the step size taken on each iteration.

'theta' is an array that will contain the parameter values we are searching for. theta[0] will contain the constant term ('k') we are searching for. theta[1] will contain the value of 'A'.

Now, in the following I could have chosen to use the variables 'k' and 'A' directly in the code. That would have made this tutorial somewhat easier to follow! However, it is not very extensible. Shortly, we will want to consider multi-dimensional linear models .. that is, models with multiple 'x' values:

$$y = k + A.x_1 + B.x_2 + C.x_3 + ...$$

You can see that very quickly, all of these different variable names would become confusing and difficult to manage. Far better to give those constants (A, B, C ..) a uniform name and store them in an array. This reflects a common way of writing the above equation, which you will see in many text-books:

$$y = \theta_0.x_0 + \theta_1.x_1 + \theta_2.x_2 + \theta_3.x_3 + ... + \theta_n.x_n$$

Where $x_0 = 1$

This formula also helps explain why the first column of our 'x' matrix was all set to the value 1.

Define a variable 'alpha' to represent the initial learning rate and set it so some small value.

Create a numpy array called 'theta' that will contain the learnt parameters of your model. The length of this array has to equal the number of parameters in the model - in this case 2.

Initially, set every element of theta equal to '1'.

(**note** Another common choice for starting values for parameters is to set the elements of the theta vector to a random number. It really makes no difference - we will be continually updating the values in the theta vector until they represent the best fit.)

```
alpha = 0.0005           # The learning rate.
                         # How large the steps are towards the solution

theta = np.ones(n)       # 'theta' will be the solution
                         # - a set of paramaters describing the linear
                         # function. Theta starts as an array of '1's
                         # - but will be updated during the algorithm to
                         # converge on the solution
                         # In this specific case we will be attempting
                         # to fit 2 parameters .. the 'A' and 'k' of the
                         # original equation.
print(f"alpha = {alpha}")
print(f"theta = {theta}")
```

```
alpha = 0.0005
theta = [1. 1.]
```

2.6 Calculating the Hypothesis function

We are going to attempt to calculate a 'Hypothesis' function. This will be a current, 'guess' at the best solution to the problem.

We will update the values of 'theta' step-by-step to improve the hypothesis. Thus, the hypothesis will become a better and better approximation to the 'correct' solution.

In order to evaluate the 'current' version of our Hypothesis function, the x data needs to be the correct shape. You will often see this in Machine Learning books as computing the Transpose of the data: x^T. Using Numpy this can be written x.T:

```
print(f"x.T = {x.T}")
```

```
x.T = [[ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
   1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
   1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
   1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
   1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
   1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
 [ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11. 12. 13. 14. 15. 16. 17.
  18. 19. 20. 21. 22. 23. 24. 25. 26. 27. 28. 29. 30. 31. 32. 33. 34. 35.
  36. 37. 38. 39. 40. 41. 42. 43. 44. 45. 46. 47. 48. 49. 50. 51. 52. 53.
  54. 55. 56. 57. 58. 59. 60. 61. 62. 63. 64. 65. 66. 67. 68. 69. 70. 71.
  72. 73. 74. 75. 76. 77. 78. 79. 80. 81. 82. 83. 84. 85. 86. 87. 88. 89.
  90. 91. 92. 93. 94. 95. 96. 97. 98. 99.]]
```

This means that we can 'multiply' our source data (x) with the current best-guess parameters (theta) using the numpy 'dot' operator, to calculate the hypothesis.

Using the mathematical formulation above we are calculating:

$$\vec{y} = \theta_0.\vec{x}_0 + \theta_1.\vec{x}_1$$

Where I have indicated that the x's and y's in this formula are actualy vectors (arrays) of numbers by using the common vector notation $\vec{y}$

So add a block of code that uses the numpy 'dot' operator to calculate a 'best_guess' by computing the dot–product of our thetas and the transposed data array x.

```
1 best_guess = np.dot(theta, x.T)
2 best_guess
```

```
array([  1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.,  11.,
        12.,  13.,  14.,  15.,  16.,  17.,  18.,  19.,  20.,  21.,  22.,
        23.,  24.,  25.,  26.,  27.,  28.,  29.,  30.,  31.,  32.,  33.,
        34.,  35.,  36.,  37.,  38.,  39.,  40.,  41.,  42.,  43.,  44.,
        45.,  46.,  47.,  48.,  49.,  50.,  51.,  52.,  53.,  54.,  55.,
        56.,  57.,  58.,  59.,  60.,  61.,  62.,  63.,  64.,  65.,  66.,
        67.,  68.,  69.,  70.,  71.,  72.,  73.,  74.,  75.,  76.,  77.,
        78.,  79.,  80.,  81.,  82.,  83.,  84.,  85.,  86.,  87.,  88.,
        89.,  90.,  91.,  92.,  93.,  94.,  95.,  96.,  97.,  98.,  99.,
       100.])
```

It is worth taking a moment to work out what the above 'np.dot' is achieving. Essentially, it is applying our current best guess parameters (our 'A' and 'k') to each of the x values in turn. It is saying: What would our y look like if we accepted those parameters as our solution?

In this first iteration, our approximation to the correct solution is rather poor (of course it is, we simply guessed at '1' for each of our parameters!)

2.7 Comparing the Hypothesis with our source data

We can compare this first 'guess' with the real y data by plotting them both on the same graph. Add some code to plot both the original data and the current model.

Define a function that plots two scatter plots on the same chart with the following signature:

```
def show_model(x, y, y_model, caption = "Model scatter plot"):
```

```
def show_model(x, y, y_model, caption = "Model scatter plot"):
        fig = plt.figure(figsize=(5, 5))
        fig = sns.scatterplot(x = x, y=y_model, label='Model')
        fig = sns.scatterplot(x = x, y=y, label='Original data')
        fig.set(xlabel ="x",
                ylabel = "y & y model",
                title =caption)
        plt.legend()
        plt.show();
```

Use that function to plot a chart showing your data and your model.

hint : Remember that your x data is stored in an array with 2 columns (see above). You will need to extract the second column (column 1!) from x to pass to the function for plotting.

```
show_model(x[:,1],y, best_guess, "Graph of y = 2.x + 100 + random.uniform(30)" )
plt.show()
```

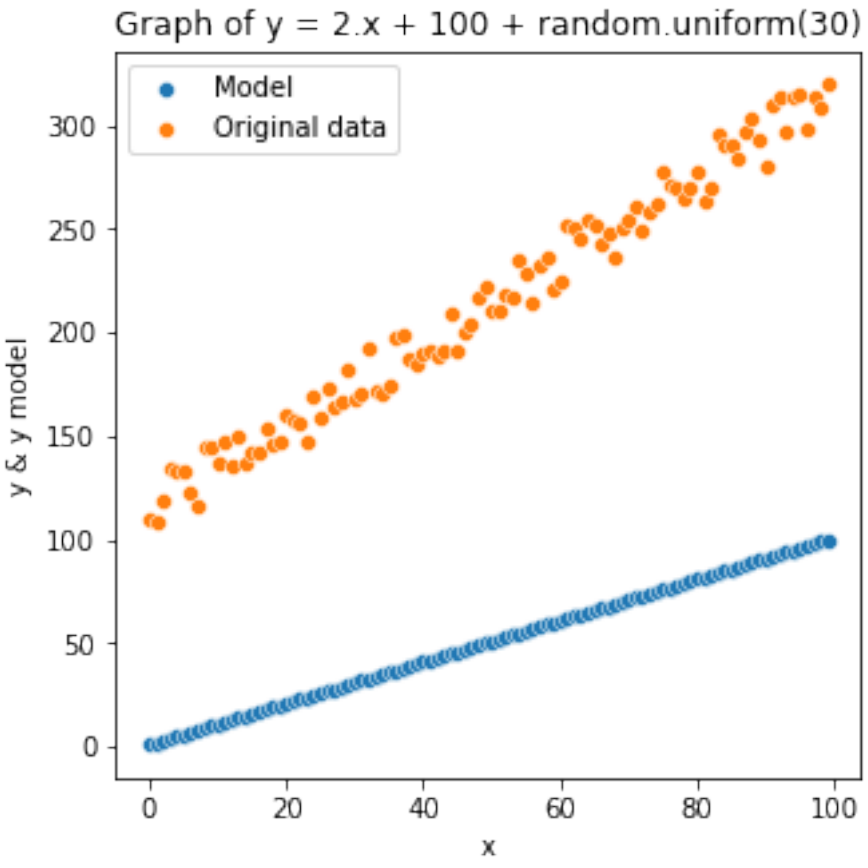

Figure 3: Scatter-plot of Synthetic linear data and the current model

An important question is .. how 'wrong' is our Hypopthesis? What is the 'distance' between our hypothesis and the true set of y values? In the following we are using the l2 norm .. or Euclidean distance between the hypothesis and the true y values.

The difference for each data point can be calculated as the 'epsilon'. Add code to create an array that represents the difference between the 'best_guess' and the true (measured) y values.

```
epsilon = best_guess - y
print(f"epsilon = {epsilon}")
```

```
epsilon = [-108.70157221 -107.0948408  -115.99201135 -130.56732698 -127.8489362
 -127.23029528 -115.85609288 -108.61043665 -135.77696985 -134.72337021
 -126.00356144 -134.88557525 -123.24656315 -135.85387184 -121.81194152
 -125.99733295 -125.85467545 -136.3130653  -126.54752974 -127.60365882
 -139.75629607 -135.7708841  -133.51464586 -122.92931543 -144.28493605
 -133.43906698 -146.08757919 -136.58412056 -137.81246313 -151.91942018
 -136.71199931 -138.53054836 -159.95462119 -138.0825547  -135.20738594
 -138.79243741 -160.84708554 -160.6782652  -148.51053169 -144.53039432
 -149.49598975 -148.74657826 -145.52300763 -147.17395909 -164.19156326
 -145.53827118 -153.68806031 -156.13274569 -168.36322925 -172.35173722
 -159.87606059 -158.78591507 -165.57403861 -163.4949577  -179.79340728
 -172.40915068 -157.82177757 -174.20869009 -176.83388814 -160.41633982
 -163.52943465 -189.25965435 -187.35444469 -181.20811825 -188.93712248
 -185.41506556 -175.39309852 -179.90245902 -167.83344129 -180.9188989
 -183.17975931 -188.75840502 -176.70044394 -183.7184172  -187.69343771
 -201.39628273 -193.7626343  -191.92076108 -185.61970204 -190.02254979
 -196.52514047 -181.50652973 -187.43475634 -211.32776695 -205.48981171
 -204.16180161 -197.4964076  -209.0216591  -214.45988153 -203.5861462
 -189.52571429 -218.56648479 -220.83390258 -203.42623296 -218.50493663
 -219.02534813 -201.72299212 -215.85973489 -209.47004679 -219.80490364]
```

The average 'cost' as the sum of the squares of this array divided by the number of items. Add code to produce a single value which is the sum of the squares of the loss.

```
cost = np.sum(epsilon ** 2)  / m
print(f"Cost = {cost:.2f}")
```

```
Cost = 27742.41
```

Again, this is worth considering for a moment. The above number represents a 'score' for our hypothesis. It is the is a measure of the difference between the given y values, and a set of hypothetical y values that are generated by our current best-guess parameters for $theta_0$ and $theta_1$. We need to minimize this value if we are to get a good fit.

2.8 Updating the parameters to improve the hypothesis

We now need to work out how to adjust $theta_0$ and $theta_1$ to improve our Hypothesis. As described in the lecture, this means determining the gradient of the cost function. The gradients are calculated using the following formula.

$$\frac{\partial}{\partial M} = \frac{2}{n}\sum_{i=1}^{n}(y_i - (k + M.x_i)).x_i$$

$$\frac{\partial}{\partial k} = \frac{2}{n}\sum_{i=1}^{n}(y_i - (k + M.x_i))$$

So thinking about this in terms of code … first create a numpy array called 'gradient' that will be used to store the calculated gradients.

```
gradient = np.zeros(n)
print(f"gradient = {gradient}")
```

```
gradient = [0. 0.]
```

Now iterate through the x and y arrays in each case applying the above partial differential to calculate the gradients

```
for i in range(len(x)):

    #gradient of loss function with respect to A
    gradient[1] += ( - (1 / m) * x[i,1]
                    * (y[i] - (( theta[0] * x[i,1]) + theta[1])))

    #gradient of loss function with respect to k
    gradient[0] += (- (1 / m) *
                        (y[i] - ((theta[0] * x[i,1]) + theta[1])))


print(F"gradient = {gradient}")
```

```
gradient = [ -163.71157875 -8956.62147806]
```

2.8.1 A short-cut method for calculating the gradients

In many examples of demonstration code for Gradient Descent you will see a 'short-cut' formula for the gradient. This formula gives exactly the same result as above, but is shorter to write and much faster.

An explanation for this formula is outside of the scope of the current course. However, if you wish to read more, about this **after class** then there is a nice explanation here:

https://stats.stackexchange.com/questions/396735/intuition-behind-computing-gradient-for-a-model.

Here, I print the results again, simply to demonstrate that the short-cut version gives the same answer as above:

```
gradient = np.dot(x.T, epsilon) / m
print(F"gradient = {gradient}")
```

```
gradient = [ -163.71157875 -8956.62147806]
```

This gives us the 'slope' in the cost function for a specific set of thetas (hypothesis parameters). We can therefore use this to update our set of hypothesised parameters (remembering that each of the variables in the following formula is actually a vector).

Add code to update your array of thetas using the gradient value you just calculated. Remember to scale your gradient by 'alpha' your learning rate.

```
theta = theta - alpha * gradient
print(f"theta = {theta}")
```

```
theta = [1.08185579 5.47831074]
```

The two values above are a closer approximation to the 'best fit' values than were our originals (which were just [1,1].

These new parameters are the basis of our updated model.

Add code to re-calculate the 'best_guess' array based on those new thetas. **hint** this is just a copy of code you have already used above. Then print out the first few values of the 'best_guess'.

```
best_guess = np.dot(theta, x.T)
print(f"Best guess[0:10] = {best_guess[0:10]}")
```

```
Best guess[0:10] = [ 1.08185579  6.56016653 12.03847727 17.51678801 22.99509875 28.4734094
 33.95172022 39.43003096 44.9083417  50.38665244]
```

2.9 Reviewing results of the first iteration

Now review the results of this iteration by calling the function 'show_model' defined previously.

```
show_model(x[:,1],y, best_guess, "Graph of y = 2.x + 100 + random.uniform(30)" )
```

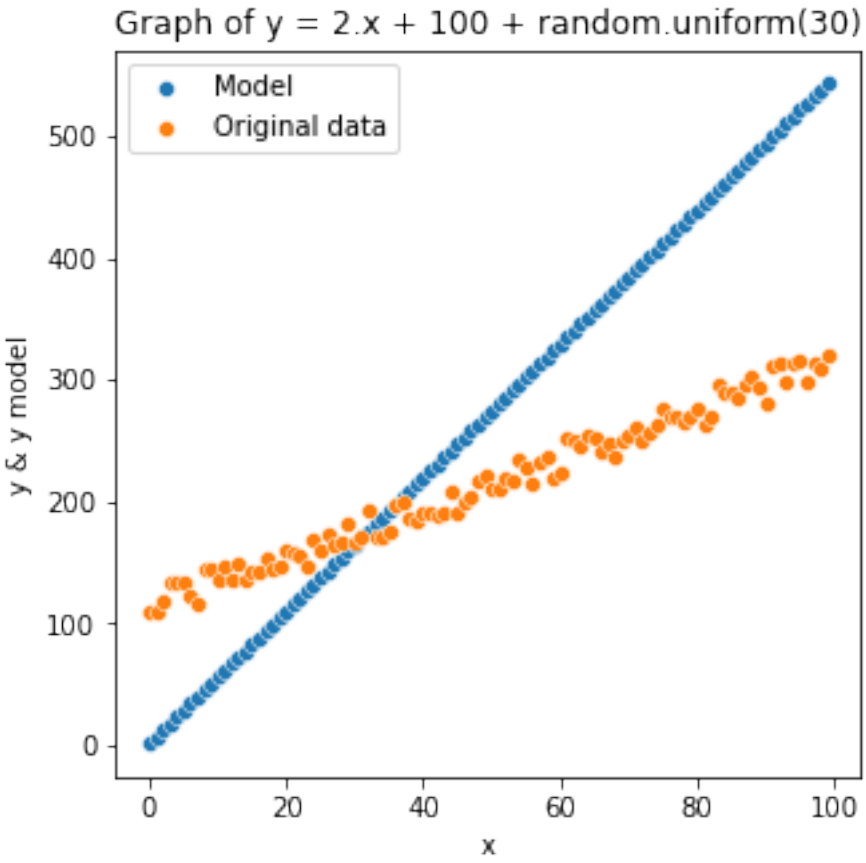

Figure 4: Scatter-plot of data vs model after model improvement

2.10 Iterative improvement of the model

Having taken all of the steps to improve our estimation of the paramaters in one iteration, we can now apply exacly the same procedure to improve it again.

Because the learning-rate alpha is small, we need to iterate many times before the solution gets closer to the best fit value.

Add code that iterates over the 'model improvement' code multiple times then plots a chart of the updated model.

```
# Define the number of improvement steps
numIterations= 10000
for i in range(0, numIterations):
    best_guess = np.dot(theta, x.T)
    epsilon = best_guess - y
    gradient = np.dot(x.T, epsilon) / m
    theta = theta - alpha * gradient


show_model(x[:,1],y, best_guess, "Graph of y = 2.x + 100 + random.uniform(30)" )
```

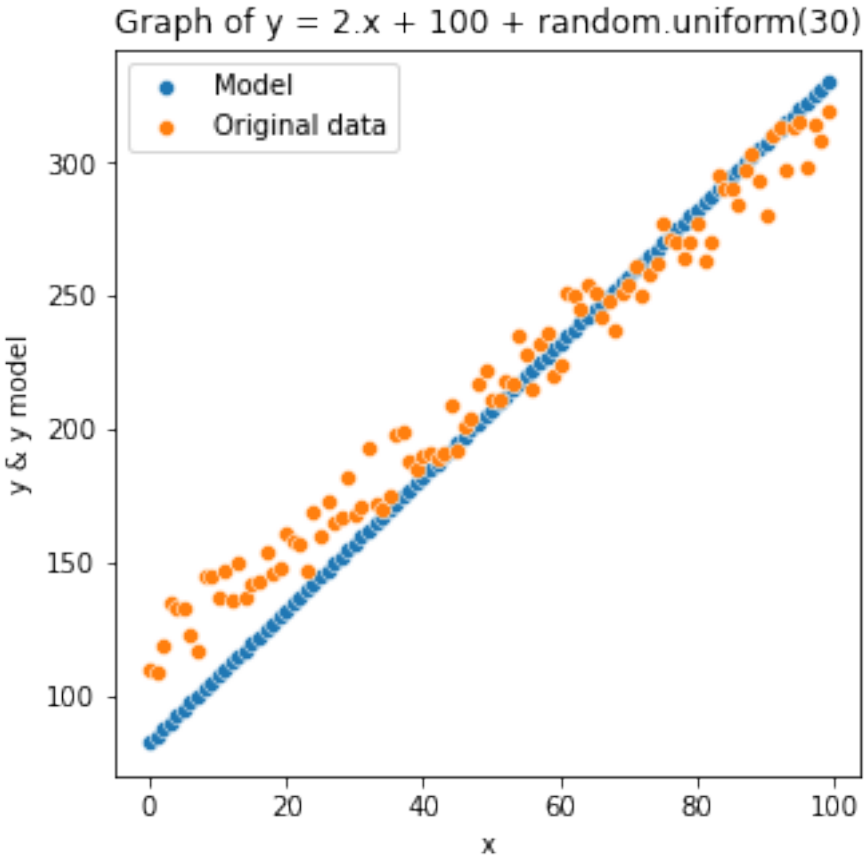

Figure 5: Scatter-plot of data vs model after many iterations

Student task : Now run the above code block several times and watch how each time the cost should reduce, and the fitted line should get closer to the data-set.

Eventually you will see that the rate of improvement slows down and the fit is as good as it is going to get.

At this point you can print the values of the parameters you have discovered for your model:

```
print(f"theta = {theta}, A = {theta[1]:.2f}, k = {theta[0]:.2f}")
```

```
theta = [82.27019693  2.50262472], A = 2.50, k = 82.27
```

When I ran this I saw:

```
theta = [85.70479873  2.44094766], A = 2.44, k = 85.70
```

(You may see something slightly different since the data matrix is randomised each time this code is run)

The values above are a fairly good match to the original values of A=2 and k=100).

2.11 Wrapping Gradient Descent as a single function

In practice, it is much more convenient to wrap much of the above code into a re-usable function. Define a function with the following signature:

```
def gradientDescent(x, y, theta, alpha, numIterations):
```

Reproduce the code from the above sections into a single, re-usable function.

In your code, include some checking of the parameters:

1. That the x and y parameters are the same length
2. Check that the x parameter is a matrix with at least 2 columns

Your function should return the theta values for the updated model:

```
def gradientDescent(x, y, theta, alpha, numIterations):

    # string run over two lines here simple for
    # formatting in workbook
    length_error_msg = ("Error in gradientDescent :"
        "Length of x and y data inputs are different")

    x_dims_error_msg = ("Error: x data must contain at least two "
                        "columns with x=[0,n] = 1")

    # Check x and y shape
    x_length, x_width  = np.shape(x)

    if (x_length != y.size):
```

```
        raise Exception(length_error_msg)
    else:
        m = x_length

    if (x_width < 2):
        raise Exception(x_dims_error_msg)


    # Compute the best fit
    for i in range(0, numIterations):
        best_guess = np.dot(x, theta)
        epsilon = best_guess - y
        gradient = np.dot(x.T, epsilon) / m
        theta = theta - alpha * gradient

    total_cost = np.sum(epsilon**2)      # Sum of squares of errors
    print(f"A = {theta[1]:.2f}, k = {theta[0]:.2f}")
    print(f"total_cost = {total_cost:.2f}")

    return theta
```

We can trial this function by generating some new data. Re-use the 'genData' function defined above. Experiment using different values for A, K and the variance

```
numPoints = 100
A = 2
K = 100
variance = 30

x, y = genData(numPoints, A, K , variance)
theta = np.ones(n)
print(f"A = {A:.1f}, K = {K:.1f}, variance = {variance:.1f}")
# print(f"y = {y}")
```

```
A = 2.0, K = 100.0, variance = 30.0
```

Repeatedly call the 'gradientDescent' function to improve the model

```
theta = gradientDescent(x, y, theta, alpha=0.0005, numIterations = 200)
```

```
A = 3.66, k = 3.92
total_cost = 326102.39
```

Which we can again plot as a chart:

```
best_guess = np.dot(theta, x.T)
show_model(x[:,1],y, best_guess, "Graph of y = 2.x + 100 + random.uniform(30)" )
```

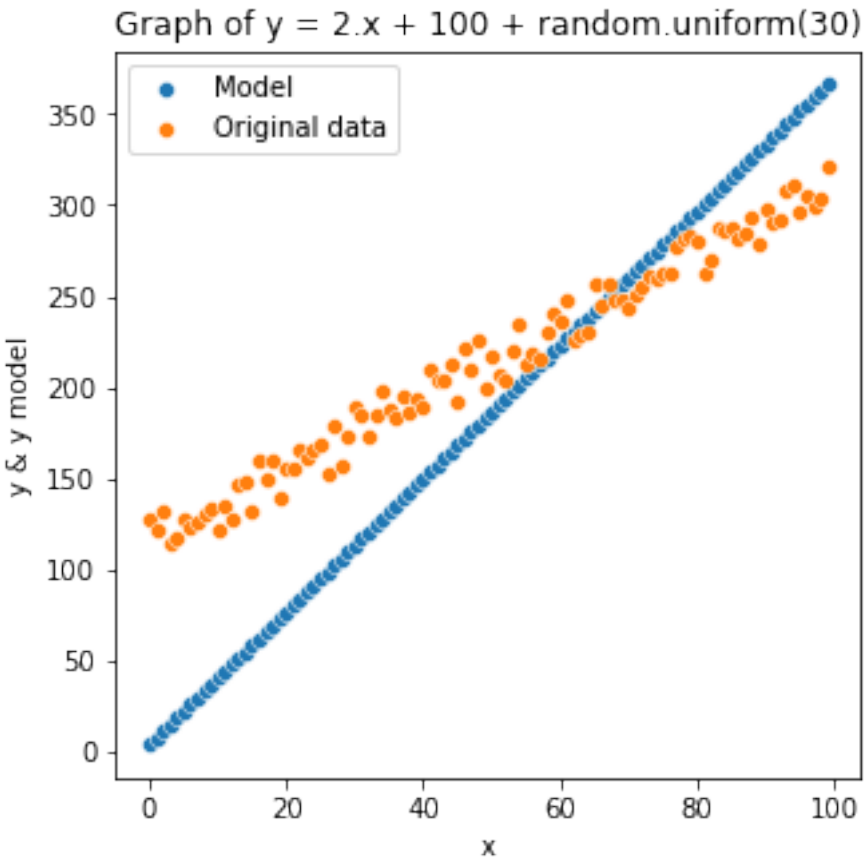

Figure 6: Scatter-plot of data vs model after many iterations (2)

2.12 Extending to multiple dimensions

One of the neat aspects of this algorithm is that it is easily extended to data-sets with multiple features (dimensions) .. creating a model of the form:

$$y = \theta_0.x_0 + \theta_1.x_1 + \theta_2.x_2 + \theta_3.x_3 + ... + \theta_n.x_n$$

Where $x_0 = 1$

We can demonstrate this by creating a new version of our data-set generator that creates multi-dimensional data. In this code we are passing an array of 'theta' values that will be used to create our synthetic data.

```
def genDataM(numPoints, numDimensions, thetas, variance):
    x = np.random.rand(numPoints, numDimensions)
    y = np.zeros(shape=numPoints)
    for i in range(0, numPoints):
        x[i][0] = 1
        y[i] = 0
        for j in range(0,len(thetas)):
            y[i] += x[i,j] * thetas[j]
        y[i] += random.uniform(0, 1) * variance
    return x, y
```

2.12.1 Step 1 : Generate synthetic data for testing our algorithm

Call 'genDataM' with the following parameters:

- 'n = 3' : the number of dimensions for this synthetic data
- thetas=[1,5,3] : The gradients in each dimension of the model
- variance = 5 : The spread of the Gaussian noise on each data-point

Create a new parameter array 'theta' with length 'n'

```
n = 3  # number of dimensions
x,y = genDataM(numPoints= 500, numDimensions = n, thetas=[1,5,3],
                    variance = 5)
theta = np.ones(n)                    # re-set the model parameters

print(f"n = {n:.0f}")
print(f"theta = {theta}")
print(f"x[0:5] = \n{x[0:5]}")
```

```
n = 3
theta = [1. 1. 1.]
x[0:5] =
[[1.         0.49244429 0.1816613 ]
 [1.         0.49500228 0.56293045]
 [1.         0.59238729 0.74236173]
 [1.         0.60532583 0.41132928]
 [1.         0.73105483 0.90212802]]
```

2.12.2 Step 2 : Visualise

Plot this as a 3D chart:

note : You may use of the 'magic' directive '%matplot' without a parameter to enable the output as a pop-out, rotatable, 3D chart in a window.

```
%matplotlib
fig = plt.figure(figsize=(12,12))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(x[:,1],x[:,2],y)
plt.show(),
```

```
Using matplotlib backend: Qt5Agg
```

```
(None,)
```

3D Scatterplot of data to be modelled

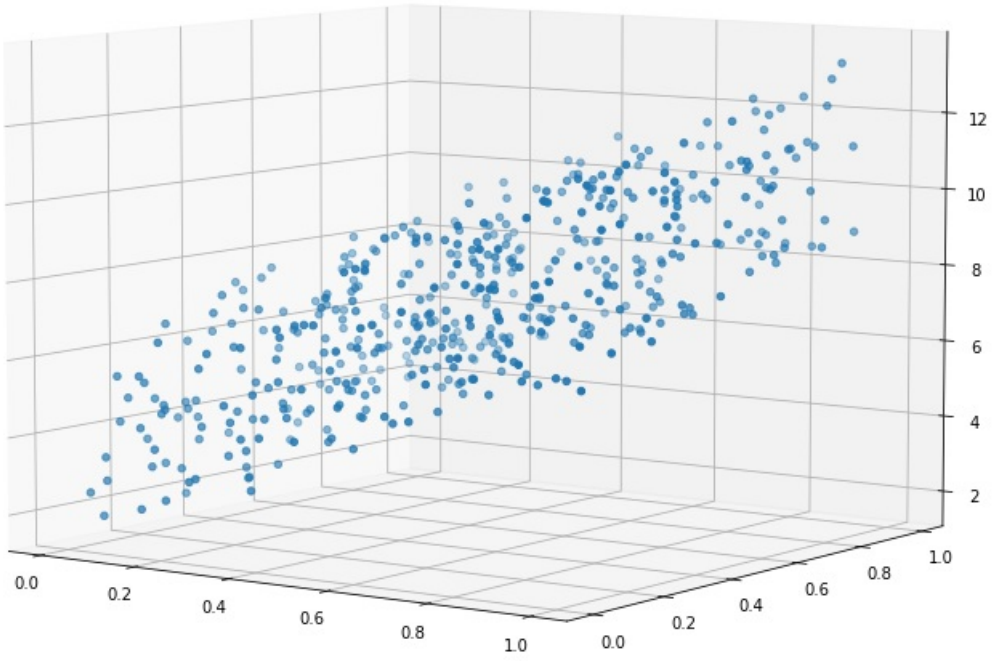

Figure 7: 3D plot of Synthetic Data before model fitting

2.12.3 Step 3 : Apply gradient descent to improve the model

Then use the 'gradientDescent' function to determine a best fit linear model for this data

```
theta = gradientDescent(x, y, theta, alpha=0.0005, numIterations = 1000)
theta
```

```
A = 2.09, k = 2.91
total_cost = 4659.31

array([2.91094967, 2.08543584, 2.02760835])
```

2.12.4 Step 4 : Visualize the model

Then finally, plot a chart of the best fit model:

```
# Step 4
#| fig-cap: '3D Scatterplot of fitted data'
best_guess = np.dot(theta, x.T)
ax.scatter(x[:,1],x[:,2],best_guess)
plt.show()
```

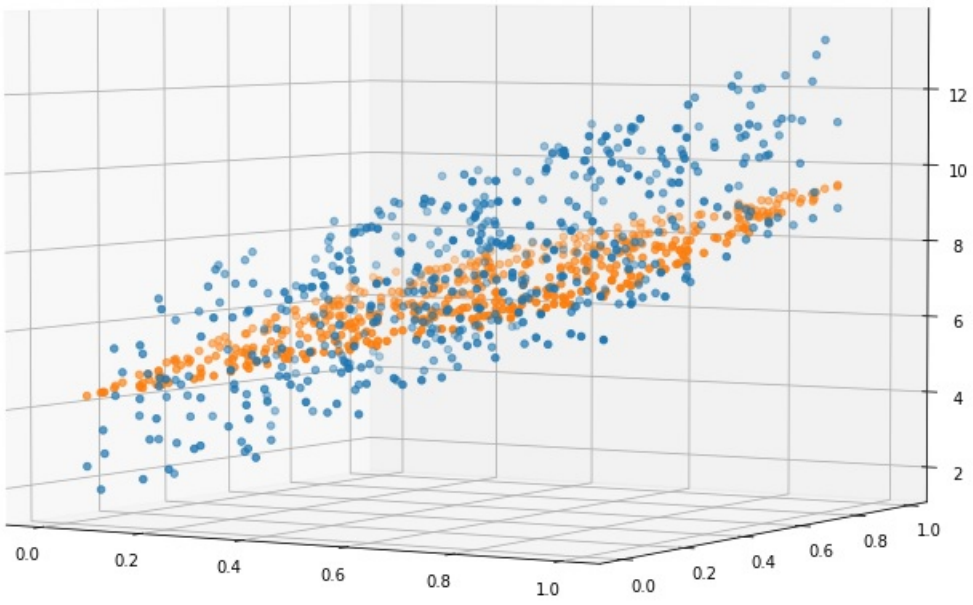

Figure 8: 3D plot of Synthetic Data and Corresponding Model

2.12.5 *Student Task* : Experiment with the above:

- 1. Repeatedly run the code blocks labelled 'Step 3' and 'Step 4'. Each time you do this, a new set of coloured points should appear. Each of these represents a closer and closer approximation to the best fit model
- 2. If the chart gets too cluttered, re-run the code block labelled 'Step 3' - this will re-set the chart

- 3. Edit the code-block labelled 'Step 1' and change the theta parameters and the variance. Re-run code blocks 2,3 and 4 to look at the results

2.13 Conclusion

The 'genDataM' and the 'gradientDescent' functions above, can generate and fit a model to data with higher dimensionality. Of course, the fitting algorithm will get slower as the data set and the model becomes more complex. But nevertheless, it will work successfully over a wide range of data.

The algorithm demonstrated in this workshop is absolutely fundamental to an understanding of Machine Learning. It shows one important method for fitting a model to data. This core algorithm is then applicable to a range of other problems - as we shall see in later workshops.

2.14 Stochastic Gradient Descent

The algorithm for gradient descent described in the previous chapter requires that the model be evaluated for every data-point for every item of data. If the data-set is large, then this represents a significant amount of computation for each update.

Stochastic Gradient Descent (SGD) is a technique that achieves similar results, using far fewer computing resources. It is perhaps a surprising fact that a the model can be improved by iteratively 'sampling' from a data set and updating the model after every sample. In this case, the error for only the sample needs to be calculated for each update. Since the sample size is typically much smaller than the total data size this process is typically much more efficient than the full Gradient Descent algorithm described above.

There is also another reason to use SGD although that reason does not really apply in this simple case of linear curve fitting. However, in more complex modelling situations on of the issues may be that the cost function is not convex; there may be local minima in the cost surface and simple gradient descent may converge to one of those local minima. SGD is rather less susceptible to this phenomena since sampled data points introduce a level of 'noise' into the descent process than can help escape from local minima. So whilst not being relevant in this specific case, it is useful to understand SGD because of the variety of benefits it provides.

You should now attempt to implement your own Stochastic Gradient Descent function based on this description and the code you have already developed above. Work on the single dimension problem to start with, rather that trying to develop a full multi-dimensional model (my example code is the single dimensional case). In my solution is also use a sample size of 1 for each iteration. This is only a small simplification and you may optionally decide to iterate over larger samples.

It is interesting to record the evolving values of your model parameters (thetas) - so for each iteration of your model, append the values of theta to a list called 'theta_list'.

The signature for your function should be this:

`def sgd(x, y, theta, alpha, numIterations):`

the return value should be:

`return theta, theta_hist`

This is a rather harder exercise than those given previously .. but actually you have most of the code you will need in the above sections. The only 'trick' here is that, rather than calculating epsilon for a whole array of model_y vs y values, you only need to do this for a sample from the input training set on each iteration. In my solution I selected a single specific item using code something like this:

```
sample_index = np.random.randint(0, y.size)
x_item = x[sample_index][1]
y_item = y[sample_index]
```

```
def sgd(x, y, theta, alpha, numIterations):

    # string run over two lines here simple for
    # formatting in workbook
    length_error_msg = ("Error in gradientDescent :"
        "Length of x and y data inputs are different")

    x_dims_error_msg = ("Error: x data must contain at least two "
                        "columns with x=[0,n] = 1")

    # Check x and y shape
    x_length, x_width  = np.shape(x)

    if (x_length != y.size):
        raise Exception(length_error_msg)
    else:
        m = x_length

    if (x_width < 2):
        raise Exception(x_dims_error_msg)

    # record the costs for each iteration
    theta_hist = []  # record the evolution of thetas

    # Compute the best fit
    for i in range(0, numIterations):
        # select a random x,y data pair to use for the sample
        sample_index = np.random.randint(0, y.size)
```

```
        x_item = x[sample_index][1]
        y_item = y[sample_index]

        # calculate the model y for this data-point
        best_guess = theta[0] + theta[1] * x_item

        # Then the error between model and true value
        epsilon = best_guess - y_item

        # Small optimization - I omitted the normalization factor
        # 'm' - it is a constant throughout and makes no difference
        # to the result
        theta[0] = theta[0] - alpha * epsilon           # Constant term
        theta[1] = theta[1] - alpha * epsilon * x_item  # Linear term

        # Record the parameters for charting purposes
        theta_hist.append(theta.copy())


    # For reference, let's compute the distance between
    # the new model and our original data
    model_ys = np.dot(x, theta)
    squared_distance = np.mean((model_ys - y)**2)
    print(f"k = {theta[0]:.2f},")
    print(f"A = {theta[1]:.2f}, mean squared distance = {squared_distance:.2f}")

    return theta, theta_hist
```

2.14.1 Testing the Stochastic Gradient Descent Function

Now we can test the above SGD function by generating new test data using your original genData function

```
x, y = genData(numPoints = 100, A = -10, K = 100, variance = 200)
```

You will need to define some parameters for your model:

- alpha : learning rate. Suggested value, 0.0001
- n : the number of dimensions, in this case - just 2
- theta : A numpy array of dimension 'n', initialized with ones
- number_of_iterations : Suggested value 100,000

Remember that, in experimenting with alpha, the larger the value of alpha the faster the model will converge. **But** : With a larger alpha:

1. The model may become unstable
2. In the case of SGD, the final results will be more 'noisy'

```
alpha = 0.0001
n = 2
theta = np.array([ 1.0, 1.0 ])
number_of_iterations = 100000
```

Finally, call your sgd function:

```
theta, theta_hist = sgd(x, y, theta, alpha, number_of_iterations)
```

```
k = 174.15,
A = -9.38, mean squared distance = 3659.58
```

It is also interesting to plot the evolving values for 'k' and 'A' (theta[0] and theta[1]).

Write some code to plot two line charts - one for values of k and the other for values of A as returned in the 'theta_hist' list

```
k_history = [ theta[0] for theta in theta_hist ]
A_history = [ theta[1] for theta in theta_hist ]
```

```
%matplotlib inline
# So that we can show just part of the data ..
plt_start = 0
plt_end = 100000

plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)  # 1 row, 2 columns, subplot 1
ax.yaxis.get_major_locator().set_params(integer=True)

plt.plot(k_history[plt_start:plt_end])
plt.title('k')
plt.xlabel('Iteration')
plt.ylabel('Model value')

plt.subplot(1, 2, 2)  # 1 row, 2 columns, subplot 2

plt.plot(A_history[plt_start:plt_end])
plt.title('A')
plt.xlabel('Iteration')
plt.ylabel('Model value')
```

```
22
23 plt.tight_layout()
24
25 # Display the plot
26 plt.show();
```

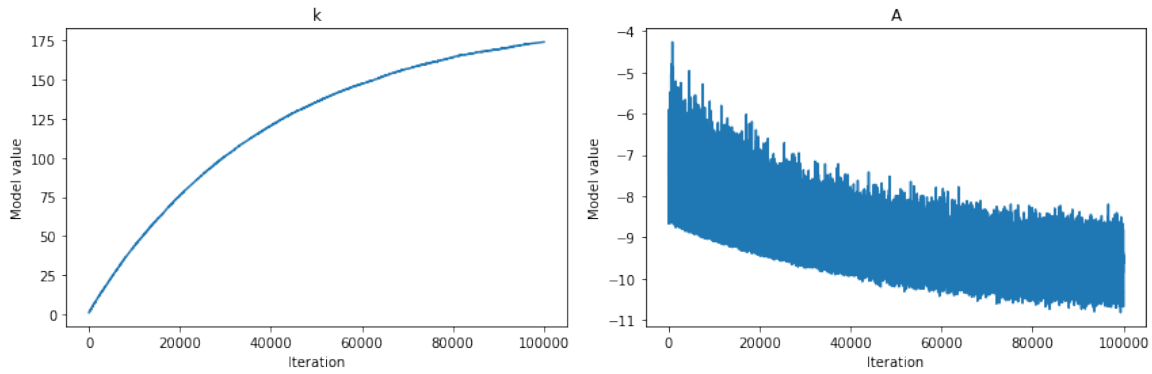

Figure 9: Line charts showing evolution of k and A towards best fit

If you wish, you can plot the actual model using the 'show_model' function developed above.

```
1 best_guess = np.dot(x, theta)
2 show_model(x[:,1],y, best_guess, "Scatter plot of original y and model" )
```

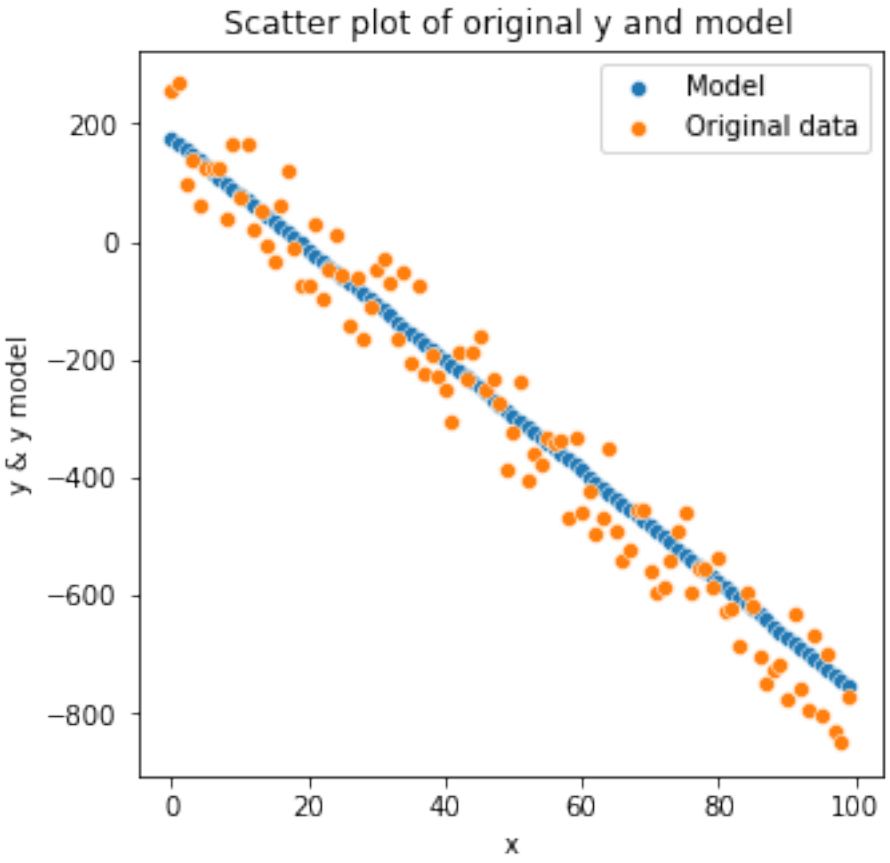

Figure 10: Scatter-plot of model generated using SGD

2.14.2 A more direct chart display using Pandas

An alternative, and somewhat simpler method for showing the history of theta values is included below

```
%matplotlib inline
for_plotting_df = pd.DataFrame(theta_hist)

# Plot theta history as a line chart
for_plotting_df.plot()

plt.xlabel('Iteration')
plt.ylabel('Thetas')
plt.title('Line Chart of Values over Time')

plt.show()
```

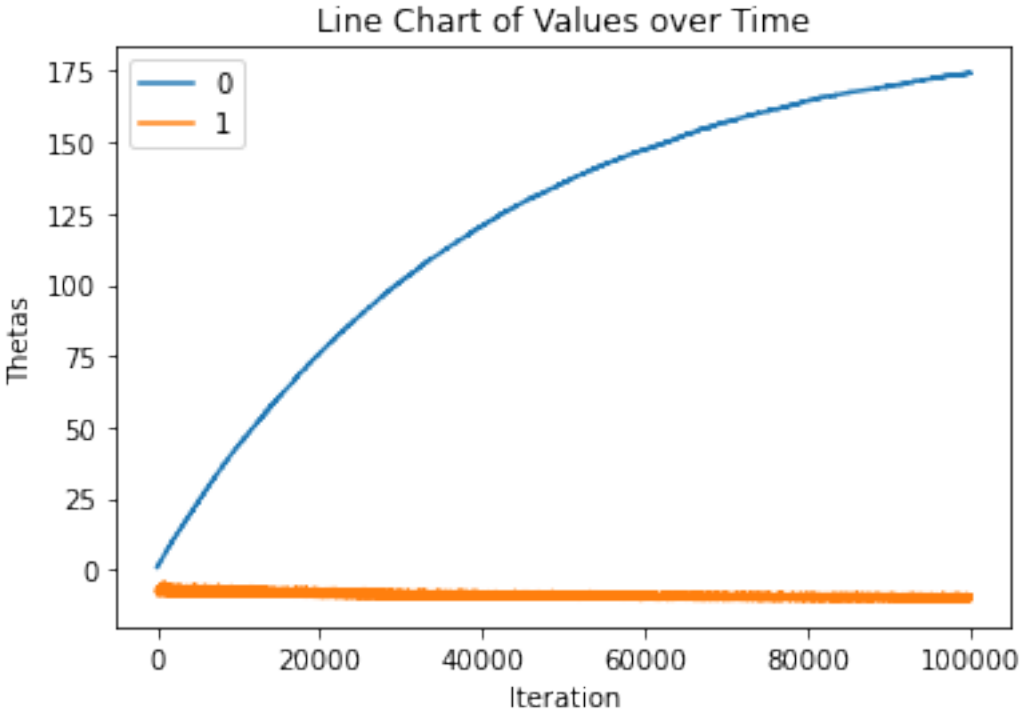

Figure 11: Pandas version of line chart showing evolution of thetas