Gradient Descent

Dr Rob Collins

Version 3, 3rd August 2022

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.

Instructions for Students

This workbook includes some empty code-blocks. In those cases you are required to create the code for the block based on your previous learning on this course. This activity is indicated in the workbook below with a '**Student Task**' indicator.

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

The numbers shown in the 'In [n]' text on your Jupyter notebook are likely to be different to the ones shown here because they are updated each time a block is executed.

1. Generating some data which we can use to experiment with fitting

As always we start by importing any required libraries. In this case, we are going to generate some data that essentially falls on a straight line, but also has a small amount of 'noise' (randomness).

```
In [85]: import numpy as np
         import random
```

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.

```
In [86]: 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)

```
In [87]: x, y = genData(numPoints = 100, A = 2, K = 100, variance = 30)
```

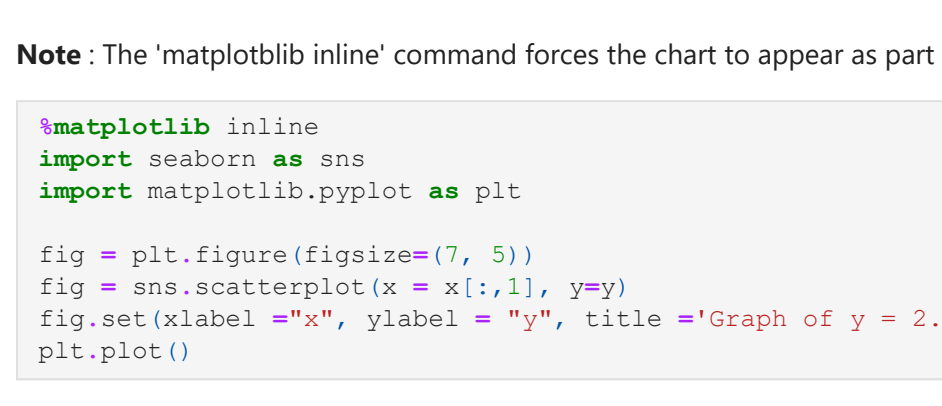
We can now look at the data graphically:

Note : The 'matplotlib inline' command forces the chart to appear as part of your Jupyter notebook rather than in a separate window

```
In [88]: %matplotlib inline
         import seaborn as sns
         import matplotlib.pyplot as plt

         fig = plt.figure(figsize=(7, 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()

Out[88]: []
```



..and as explained above, the 'shape' of x is 'm' rows and 2 columns:

```
In [89]: m, n = np.shape(x)                    # The length and width of the source dataframe
         print("m (Length)= ", m, "and n (width)=", n)

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

.. and the first column of x data contains only '1's

```
In [90]: print(x[0:5])

[[1. 0.]
 [1. 1.]
 [1. 2.]
 [1. 3.]
 [1. 4.]]
```

2. Fitting a model to the data

2.1 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 many different 'y' values:

y = k + A.x1 + B.x2 + C.x3 + ...

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.

In the following code I have set the theta vector to be equal to '1' in both columns. Another common choice here is to set 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.

```
In [91]: 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(theta)

[1. 1.]
```

2.2 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:

```
In [92]: print(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 computing our best-guess version of y, we get:

```
In [93]: best_guess = np.dot(theta, x.T)
         best_guess

Out[93]: 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.

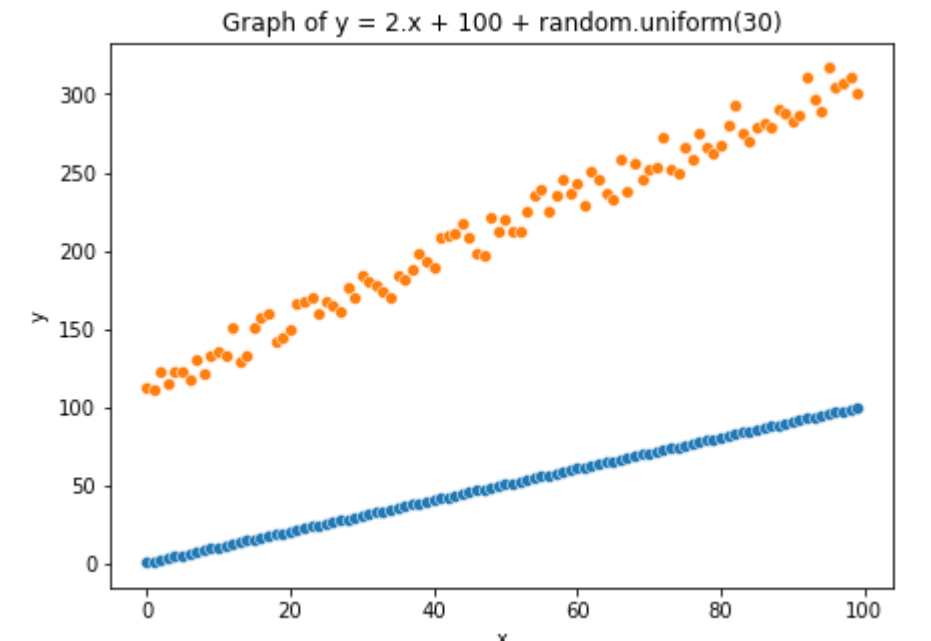
In this first iteration, our approximation to the correct solution is rather poor (of course it is, we simply guessed at '1' for each of our parameters!

2.3 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:

```
In [94]: fig = plt.figure(figsize=(7, 5))
         fig = sns.scatterplot(x = x[:,1], y=best_guess)
         fig = sns.scatterplot(x = x[:,1], y=y)
         fig.set(xlabel ="x", ylabel = "y", title ='Graph of y = 2.x + 100 + random.uniform(30)')

Out[94]: [Text(0.5, 0, 'x'),
          Text(0, 0.5, 'y'),
          Text(0.5, 1.0, 'Graph of y = 2.x + 100 + random.uniform(30)')]
```



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 'loss'.

```
In [95]: loss = best_guess - y
         loss

Out[95]: array([-112.14875241, -109.26968461, -120.49947685, -111.69969503,
         -118.40905711, -117.21782709, -110.40325958, -123.1224127 ,
         -113.09618468, -122.90029068, -125.20454847, -121.407177  ,
         -138.51196172, -115.34953656, -118.77874601, -135.36032699,
         -140.47099148, -142.37677296, -123.61046782, -124.13033277,
         -129.08393855, -144.8959505 , -145.27504207, -146.65579246,
         -134.68940191, -141.06488195, -137.87900587, -133.47868249,
         -147.56519187, -140.4445072 , -152.89999423, -148.43456489,
         -144.66190775, -139.87961769, -135.2112033 , -148.15620823,
         -144.2166674 , -150.26786064, -159.58121925, -153.10771572,
         -149.00019856, -166.13583152, -166.79783405, -167.28823187,
         -172.14383773, -162.10907319, -150.91375143, -148.95200613,
         -172.24949268, -162.5555269 , -168.74172462, -160.00966176,
         -159.46638108, -171.60338857, -180.06443644, -182.96558242,
         -167.53592538, -177.9542166 , -186.26917368, -176.90090081,
         -182.61949754, -167.22646076, -187.79801681, -181.4334935 ,
         -172.24864189, -166.53053565, -191.94219594, -170.02263113,
         -186.98447381, -175.17870218, -180.95803411, -181.12167729,
         -199.6287694 , -177.87435897, -174.07500185, -189.51816197,
         -181.41753953, -196.83089601, -187.1942707 , -181.73280989,
         -186.14398083, -197.57500267, -209.43452798, -190.76311551,
         -184.52450477, -192.72418907, -194.74299855, -190.80231036,
         -200.88397538, -197.5365717 , -192.146677  , -194.78159199,
         -217.34197894, -202.80926277, -194.07562659, -220.67606737,
         -207.65868407, -208.96275356, -212.18589821, -200.15014855])
```

.. and the 'cost' as the sum of the squares of this cost:

```
In [96]: cost = np.sum(loss ** 2) / (2 * m)
         cost

Out[96]: 13701.76657882209
```

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 $theta_1$.

2.4 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.

```
In [97]: gradient = np.ones(n)  # Create an array to store the gradient
         gradient[0] = 0
         gradient[1] = 0
         for i in range(len(x)):
             #gradient of loss function with respect to k
             gradient[0] += - (1 / m) * (y[i] - ((theta[0] * x[i,1]) + theta[1]))

             #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

Out[97]: array([ -162.99330067, -8869.87162943])
```

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:

```
In [98]: gradient = np.dot(x.T, loss) / m
         gradient

Out[98]: array([ -162.99330067, -8869.87162943])
```

This gives us the 'slope' in the cost function at each point. 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):

```
In [99]: theta = theta - alpha * gradient
         theta

Out[99]: array([1.08149665, 5.43493581])
```

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 give us a hypothesis of:

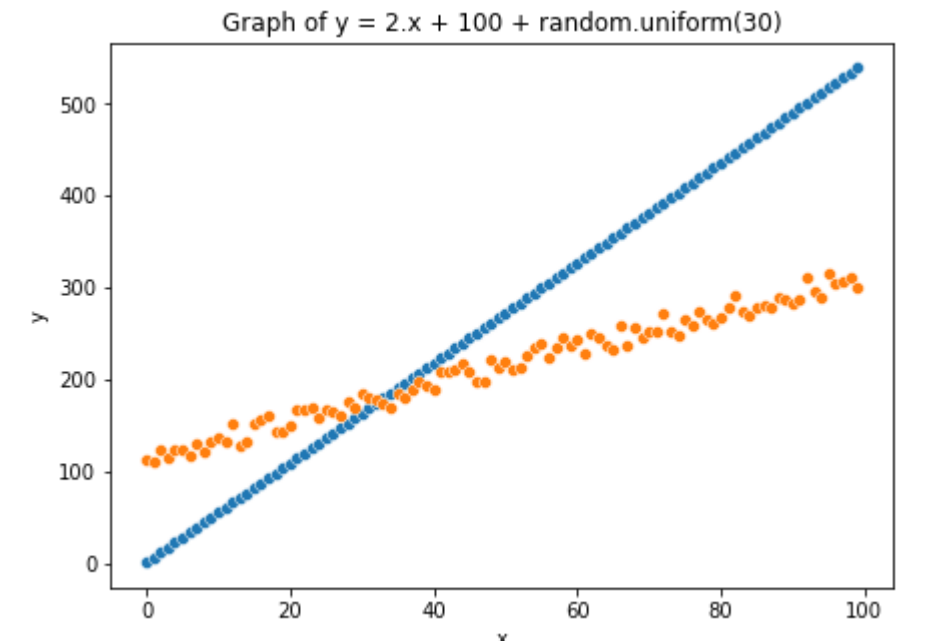
```
In [100...  best_guess = np.dot(theta, x.T)
```

2.5 Reviewing results of the first iteration

Which we can then review using a scatterplot that shows both the original y data and the data generated through our current best-guess hypothesis:

```
In [101...  fig2 = plt.figure(figsize=(7, 5))
            fig2 = sns.scatterplot(x = x[:,1], y=best_guess)
            fig2 = sns.scatterplot(x = x[:,1], y=y)
            fig2.set(xlabel ="x", ylabel = "y", title ='Graph of y = 2.x + 100 + random.uniform(30)')

Out[101...  [Text(0.5, 0, 'x'),
             Text(0, 0.5, 'y'),
             Text(0.5, 1.0, 'Graph of y = 2.x + 100 + random.uniform(30)')]
```



2.6 Iterative improvement of the model

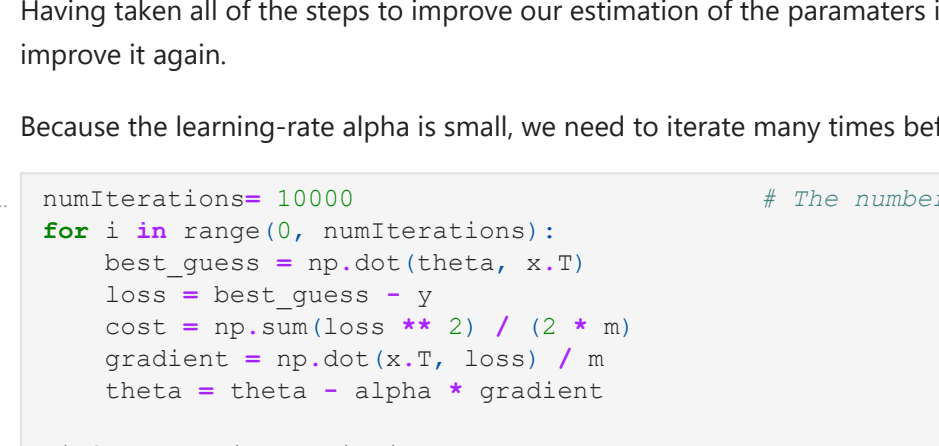
Having taken all of the steps to improve our estimation of the paramaters in one iteration, we can now apply exacly the same procedure to improve it again.

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

```
In [105...  numIterations= 10000                    # The number of times we will loop around looking for a better sc
            for i in range(0, numIterations):
                best_guess = np.dot(theta, x.T)
                loss = best_guess - y
                cost = np.sum(loss ** 2) / (2 * m)
                gradient = np.dot(x.T, loss) / m
                theta = theta - alpha * gradient

            fig2 = plt.figure(figsize=(7, 5))
            fig2 = sns.scatterplot(x = x[:,1], y=best_guess)
            fig2 = sns.scatterplot(x = x[:,1], y=y)
            fig2.set(xlabel ="x", ylabel = "y", title ='Graph of y = 2.x + 100 + random.uniform(30)')
            print ("cost = ", cost)

cost =  32.74912735206974
```



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:

```
In [106...  theta

Out[106...  array([115.64621215,   1.9730126 ])
```

When I ran this I saw:

```
array([116.94978094,   1.97198893])
```

(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).

3. Wrapping the above as a single function

In practice, it is much more convenient to wrap much of the above code into a re-usable function as follows:

```
In [107...  def gradientDescent(x, y, theta, alpha, numIterations):

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

                if (x_length != y.size):
                    raise Exception("Error in gradientDescent : Length of x and y data inputs are different")
                else:
                    m = x_length

                if (x_width < 2):
                    raise Exception("Error: x data must contain at least two columns with x=[0,n] = 1")


                # Compute the best fit
                for i in range(0, numIterations):
                    best_guess = np.dot(x, theta)
                    loss = best_guess - y
                    cost = np.sum(loss ** 2) / (2 * m)
                    gradient = np.dot(x.T, loss) / m
                    theta = theta - alpha * gradient
                return theta
```

We can trial this function by generating some new data:

```
In [108...  x, y = genData(numPoints = 100, A = 2, K = 100, variance = 30)
            theta = np.ones(n)
            theta

Out[108...  array([1., 1.])
```

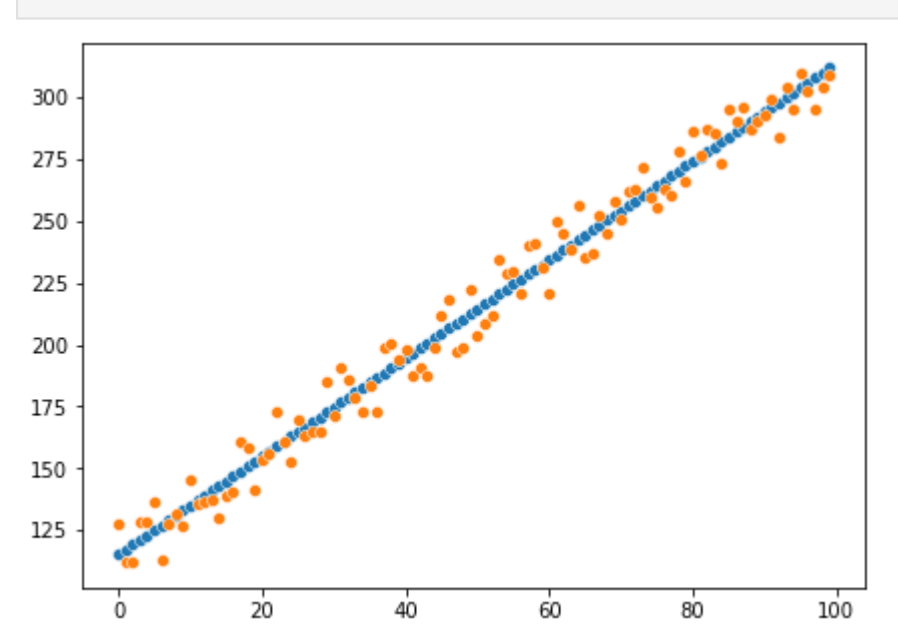
Then repeatedly running the function:

```
In [111...  theta = gradientDescent(x, y, theta, alpha=0.0005, numIterations = 90000)
            theta

Out[111...  array([115.09682679,   1.98969794])
```

Which we can again plot as a chart:

```
In [112...  best_guess = np.dot(theta, x.T)
            fig = plt.figure(figsize=(7, 5))
            fig = sns.scatterplot(x = x[:,1], y=best_guess)
            fig = sns.scatterplot(x = x[:,1], y=y)
```



4 (Optional Section) Extending to multiple dimensions

One of the neat aspects of this algorithm is that it is easily extended to multiple dimensional data sets .. 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 [113...  def genDataM(numPoints, numDimensions, x_upper, 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
```

```
In [114...  # Step 1
            n = 3  # number of dimensions
            x,y = genDataM(numPoints= 500, numDimensions = n, x_upper = 100, thetas=[1,5,3], variance = 5)
            theta = np.ones(n)                    # re-set the model paramters
            x[0:5]

Out[114...  array([[1.        , 0.18054352, 0.04747812],
                   [1.        , 0.37812555, 0.50619993],
                   [1.        , 0.67320517, 0.43361142],
                   [1.        , 0.15042351, 0.13457878],
                   [1.        , 0.27627481, 0.45395328]])
```

Plot this as a 3D chart:

```
In [115...  # Step 2
            from mpl_toolkits.mplot3d import Axes3D
            import matplotlib.pyplot as plt
            %matplotlib
            fig = plt.figure(figsize=(12,12))
            ax = fig.add_subplot(111, projection='3d')
            ax.scatter(x[:,1],x[:,2],y)

Using matplotlib backend: Qt5Agg
Out[115...  <mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x13fb28e2760>
```

Then use our same fitting function to determine a best fit linear model for this data

```
In [116...  # Step 3
            theta = gradientDescent(x, y, theta, alpha=0.0005, numIterations = 50000)
            theta

Out[116...  array([3.77874723, 4.3390216 , 2.91069536])
```

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

```
In [117...  # Step 4
            best_guess = np.dot(theta, x.T)
            ax.scatter(x[:,1],x[:,2],best_guess)

Out[117...  <mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x13fb28aaeb0>
```

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
- 1. If the chart gets too cluttered, re-run the code block labelled 'Step 3' - this will re-set the chart
- 1. 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

5 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.


```
In [ ]:
```