Polynomial Regression in Python

What is Polynomial Regression?

Polynomial Regression is a process by which given a set of inputs and their corresponding outputs, we find an nth degree polynomial f(x) which converts the inputs into the outputs.

This f(x) is of the form:

Polynomial regression has several advantages over linear regression because it can be used to identify patterns that linear regression cannot. For example, if a ball is thrown upwards, we apply a quadratic function to calculate the height of the ball over time. Also, cubic equations are used to calculate planetary motion. These patterns cannot be identified using linear regression.

Generating a random dataset

To do any Polynomial Regression, the first thing we need is data.

In the first part of this tutorial, we perform polynomial regression on a random, generated dataset to understand the concepts. Then we will do the same on some real data.

**Part 1: Using generated dataset**

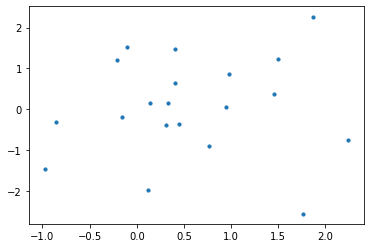
<https://colab.research.google.com/drive/1_Xa5QG-HLPV8yxIOd5vD-dA6PHYAfvd8>

We start by importing some libraries that we will be using in this tutorial.

1. **import** numpy as np
2. **import** matplotlib.pyplot as plt
3. **import** tensorflow as tf
4. **import** operator
5. **from** sklearn.metrics **import** mean\_squared\_error, r2\_score
6. **import** pandas as pd

As you expect this creates random points with random coordinates. We can visualise this using a scatter plot.

1. np.random.seed(0)
2. x = np.random.normal(0, 1, 20)
3. y = np.random.normal(0, 1, 20)
5. plt.scatter(x,y, s=10)
6. plt.show()



Doing Polynomial Regression

We are doing Polynomial Regression using Tensorflow. We have to feed in the degree of the polynomial that we want and the x data for this. The degree is an important feature that we will be covering later. First, we have to modify the data so that it can be accepted by tensorflow. Then we have to set some parameters like the optimizer and the loss function. Finally, we train the model for 12000 steps / epochs.

1. deg=3
2. W = tf.Variable(tf.random\_normal([deg,1]), name='weight')
3. #bias
4. b = tf.Variable(tf.random\_normal([1]), name='bias')
6. x\_=tf.placeholder(tf.float32,shape=[None,deg])
7. y\_=tf.placeholder(tf.float32,shape=[None, 1])
9. **def** modify\_input(x,x\_size,n\_value):
10. x\_new=np.zeros([x\_size,n\_value])
11. **for** i **in** range(deg):
12. x\_new[:,i]=np.power(x,(i+1))
13. x\_new[:,i]=x\_new[:,i]/np.max(x\_new[:,i])
14. **return** x\_new
16. x\_modified=modify\_input(x,x.size,deg)
17. Y\_pred=tf.add(tf.matmul(x\_,W),b)
19. #algortihm
20. loss = tf.reduce\_mean(tf.square(Y\_pred -y\_ ))
21. #training algorithm
22. optimizer = tf.train.GradientDescentOptimizer(0.05).minimize(loss)
23. #initializing the variables
24. init = tf.global\_variables\_initializer()
26. #starting the session session
27. sess = tf.Session()
28. sess.run(init)
30. epoch=12000
32. **for** step **in** range(epoch):
33. \_, c=sess.run([optimizer, loss], feed\_dict={x\_: x\_modified, y\_: y})
34. **if** step%1000==0 :
35. **print** ("loss: " + str(c))
37. y\_test=sess.run(Y\_pred, feed\_dict={x\_:x\_modified})

Finally we calculate the errors.

1. mse = np.sqrt(mean\_squared\_error(y,y\_poly\_pred))
2. r2 = r2\_score(y,y\_poly\_pred)
3. **print**(mse)
4. **print**(r2)

1.1507521092081143

0.061440511342737425

Loss functions

We need to calculate how efficient our model is at capturing the patterns in the data. There are 2 common ways of doing this:

1. Mean Square Error
2. R square score (R2 score)

Let us understand the math behind these two:

Mean Square Error:

For every value of x, we have the actual value of y and the value of y that our line predicts. We find the difference between the two. Then we add the differences for each value of x. Finally we divide this by the number of values of x.

This equation has a problem though. Some times the difference will be positive and other times it will be negative. These values can cancel out and even though there may be large errors the output will show that there is no error. So to tackle this problem, we square each difference.

R2 score:

First we have to find the mean m of all the values of y:

Then we get the difference between each value of y and the mean. We square each difference and add them. Let this value be k.

Now we divide the mse by k and subtract the result from 1. This gives us the R2 score. The R2 score is a value between 0 and 1. A large R2 score means x correlates to y well and the line can predict the y value well.

Let us see how this looks in code. There are some inbuilt functions that handle the calculations for us:

1. mse = np.sqrt(mean\_squared\_error(y,y\_pred))
2. r2 = r2\_score(y,y\_pred)
3. **print**(mse)
4. **print**(r2)

1.1832766119182259z

0.007636444138149345

Visualising the results

Now let us try to visualise the results.

First we find, the coefficients and the intercept of the quadratic equation generated.

1. **print**("Model paramters:")
2. **print**(sess.run(W))
3. **print**("bias:%f" %sess.run(b))

Model paramters:

[[ 1.1229055 ]

 [-2.1566594 ]

 [ 0.67295593]]

bias:0.128522

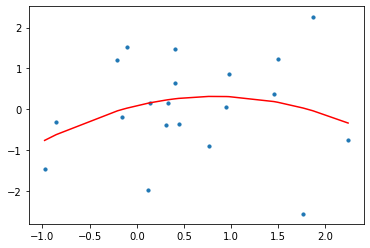
Using this we can find the equation itself

1. res = "y = f(x) = " + str(sess.run(b)[0])
3. **for** i, r **in** enumerate(sess.run(W)):
4. res = res + " + {}\*x^{}".format("%.2f" % r[0], i + 1)
6. **print** (res)

y = f(x) = 0.088324 + 1.23\*x^1 + -1.65\*x^2

Finally, we can visualise the function by plotting it. We plot a line graph of the equation.

1. plt.scatter(x, y, s=10)
2. # sort the values of x before line plot
3. sort\_axis = operator.itemgetter(0)
4. sorted\_zip = sorted(zip(x,y\_poly\_pred), key=sort\_axis)
5. x, y\_poly\_pred = zip(\*sorted\_zip)
6. plt.plot(x, y\_poly\_pred, color='red')
7. plt.show()



**Part 2: Using real data**

<https://colab.research.google.com/drive/1S0wz7xquJ5-6MaREEMnxx-_HA7r6BVZw>

In this part of the tutorial, we will be using some data (you can get it here: <https://github.com/SiddhantAttavar/PolynomialRegression/blob/master/Position_Salaries.csv> ) about position level and salary relationship in a company. As you can see as the level increases, so does the salary. However, the relationship is not linear.

First we import the data.

1. # Importing the dataset
2. url = 'https://raw.githubusercontent.com/SiddhantAttavar/PolynomialRegression/master/Position\_Salaries.csv'
3. datas = pd.read\_csv(url)
4. **print**(datas)



Position,Level,Salary

Business Analyst,1,45000

Junior Consultant,2,50000

Senior Consultant,3,60000

Manager,4,80000

Country Manager,5,110000

Region Manager,6,150000

Partner,7,200000

Senior Partner,8,300000

C-level,9,500000

CEO,10,1000000

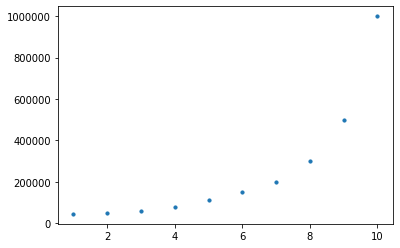
The data is stored as a csv (comma separated values) file. In this file, each column is separated by a comma, which makes it easy to read.

In this case the x values are the level column and the y values are the salary column. We create the arrays using some functions in the pandas library.

1. X = datas.iloc[:, 1].values
2. Y = datas.iloc[:, 2].values
3. Y = Y[:, np.newaxis]

Now we can plot the data using a scatter plot.

1. plt.scatter(X, Y, s=10)
2. plt.show()



We can do Polynomial Regression for this data with degree 2. We will modify this later in the course.

1. deg = 2 #@param {type:"slider", min:1, max:20, step:1}
2. W = tf.Variable(tf.random\_normal([deg,1]), name='weight')
3. #bias
4. b = tf.Variable(tf.random\_normal([1]), name='bias')
6. X\_=tf.placeholder(tf.float32,shape=[None,deg])
7. Y\_=tf.placeholder(tf.float32,shape=[None, 1])
9. X\_modified=modify\_input(X,X.size,deg)
10. Y\_pred=tf.add(tf.matmul(X\_,W),b)
12. #algortihm
13. loss = tf.reduce\_mean(tf.square(Y\_pred -Y\_ ))
14. #training algorithm
15. optimizer = tf.train.GradientDescentOptimizer(0.05).minimize(loss)
16. #initializing the variables
17. init = tf.global\_variables\_initializer()
19. #starting the session session
20. sess = tf.Session()
21. sess.run(init)
23. epoch=12000
25. **for** step **in** range(epoch):
26. \_, c=sess.run([optimizer, loss], feed\_dict={X\_: X\_modified, Y\_: Y})
27. **if** step%1000==0 :
28. **print** ("loss: " + str(c))
29. Y\_test=sess.run(Y\_pred, feed\_dict={X\_:X\_modified})

Now, we can find how well our model is performing

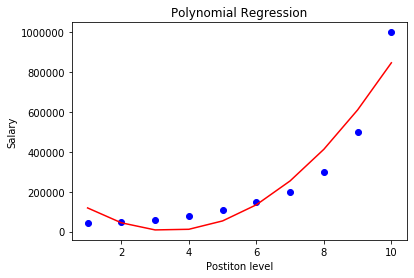
1. rmse = np.sqrt(mean\_squared\_error(Y,lin\_poly.predict(poly.fit\_transform(X))))
2. r2 = r2\_score(y,y\_poly\_pred)
3. **print**(rmse)
4. **print**(r2)

1.1507521216059198

0.06144049111930627

After this we visualise the results. First, we get the coefficients and print the formula and then, we plot the equation

1. **print**("Model paramters:")
2. **print**(sess.run(W))
3. **print**("bias:%f" %sess.run(b))
5. res = "y = f(x) = " + str(sess.run(b)[0])
7. **for** i, r **in** enumerate(sess.run(W)):
8. res = res + " + {}\*x^{}".format("%.2f" % r[0], i + 1)
10. **print** (res)
12. plt.scatter(X, Y, s=10)
13. # sort the values of x before line plot
14. sort\_axis = operator.itemgetter(0)
15. sorted\_zip = sorted(zip(X,Y\_test), key=sort\_axis)
16. X, Y\_poly\_pred = zip(\*sorted\_zip)
17. plt.plot(X, Y\_poly\_pred, color='red')
18. plt.show()



Lastly we predict what the salary for level 11 would be.

1. # Predicting a new result with Polynomial Regression
2. lin\_poly.predict(poly.fit\_transform([[11.0]]))[0]
3. 1121833.333333334

Overfitting

Under-fitting and over-fitting are 2 things that you must always try to avoid.

Under-fitting is when your model is not able to recognise the relationship between the 2 quantities. For example it may be found when trying to apply a linear model to a quadratic relationship. Common symptoms of this are high MSE and low R2 score.

On the other hand, overfitting is also a common issue. The model performs very well on the training data, but fails to perform on new, unseen data. In this case, the curve generated passes through all or nearly all the datapoints. The model fails to understand the overall pattern and cannot generalize.

There are 2 ways of eliminating these problems:

1. Providing more data: If you provide more data, the model is more likely to identify the general pattern
2. Finding the correct degree for the polynomial

Finding the correct degree

In our position vs salary example, we have very limited data, so adding more data is not possible. We have to find the correct degree. Here is a table of some degrees, their graph, their MSE, and their R2 score

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Degree | Graph | MSE | R2 score | Equation |
| 1 |  | 163388.73 | 0.66 |  |
| 2 |  | 82212.12 | 0.91 |  |
| 3 |  | 38931.5 | 0.9812097727913367 |  |
| 5 |  | 4047.5 | 0.9997969027099755 |  |
| 10 |  | 0.0008 | 1.0 |  |

Over here the linear polynomial is an underfit since it fails to capture the pattern. It also has a high MSE and a low R2 score.

The degree 5 and degree 10 polynomials overfit the data. They have high scores. In fact the degree 10 polynomial has a R2 score of 1, which is the best possible. However, given data slightly off the curve, they will not be able to generalize.

The degree 2 and degree 3 polynomials are a good fit as they capture the pattern but do not overfit. Note that in general the best fits usually do not have a degree greater than 3.

Summary

In this tutorial, we learnt the following concepts:

1. Linear Regression
2. Generating datasets
3. Mean Square Error
4. R2 score
5. Polynomial Regression
6. Plotting scatter plots and line graphs
7. Importing datasets from csv files
8. Overfitting and underfitting

Hope you are able to use these concepts in your own projects.