**Polynomial Regression**

**Table of Contents**

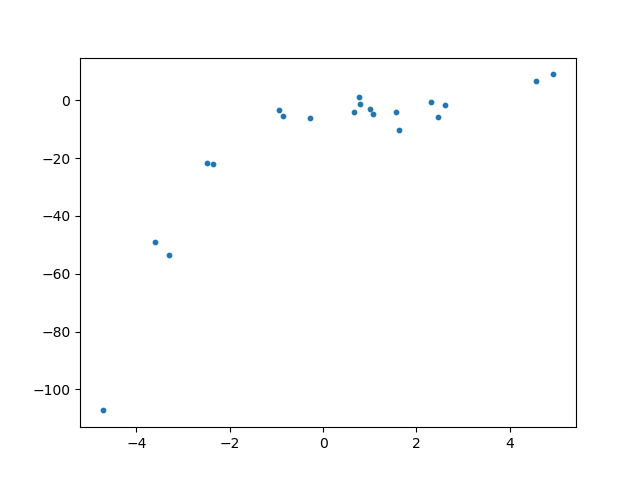
* Why Polynomial Regression
* Over-fitting vs Under-fitting
* Bias vs Variance trade-offs
* Applying polynomial regression to the Boston housing dataset.

**Why Polynomial Regression?**

To understand the need for polynomial regression, let’s generate some random dataset first.

|  |
| --- |
|  |
|  | import numpy as np  import matplotlib.pyplot as plt |
|  |  |
|  | np.random.seed(0) |
|  | x = 2 - 3 \* np.random.normal(0, 1, 20) |
|  | y = x - 2 \* (x \*\* 2) + 0.5 \* (x \*\* 3) + np.random.normal(-3, 3, 20) |
|  | plt.scatter(x,y, s=10) |
|  | plt.show() |

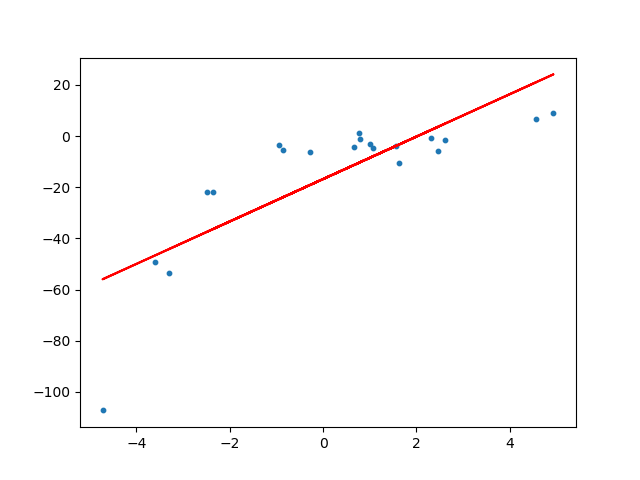
The data generated looks like



Let’s apply a linear regression model to this dataset.

|  |
| --- |
|  |
|  | import numpy as np  import matplotlib.pyplot as plt |
|  |  |
|  | from sklearn.linear\_model import LinearRegression |
|  |  |
|  | np.random.seed(0) |
|  | x = 2 - 3 \* np.random.normal(0, 1, 20) |
|  | y = x - 2 \* (x \*\* 2) + 0.5 \* (x \*\* 3) + np.random.normal(-3, 3, 20) |
|  |  |
|  | # transforming the data to include another axis |
|  | x = x[:, np.newaxis] |
|  | y = y[:, np.newaxis] |
|  |  |
|  | model = LinearRegression() |
|  | model.fit(x, y) |
|  | y\_pred = model.predict(x) |
|  |  |
|  | plt.scatter(x, y, s=10) |
|  | plt.plot(x, y\_pred, color='r') |
|  | plt.show() |

The plot of the best fit line is



We can see that the straight line is unable to capture the patterns in the data. This is an example of **under-fitting**. Computing the RMSE and R²-score of the linear line gives:

RMSE of linear regression is **15.908242501429998**.  
R2 score of linear regression is **0.6386750054827146**

***To overcome under-fitting, we need to increase the complexity of the model.***

To generate a higher order equation we can add powers of the original features as new features. The linear model,

Image for post

can be transformed to

Image for post

*This is still considered to be****linear model****as the coefficients/weights associated with the features are still linear. x² is only a feature. However the curve that we are fitting is****quadratic****in nature.*

To convert the original features into their higher order terms we will use the PolynomialFeatures class provided by scikit-learn. Next, we train the model using Linear Regression.

|  |
| --- |
|  |
|  |  |
|  | Import operator  import numpy as np |
|  | import matplotlib.pyplot as plt |
|  |  |
|  | from sklearn.linear\_model import LinearRegression |
|  | from sklearn.metrics import mean\_squared\_error, r2\_score |
|  | from sklearn.preprocessing import PolynomialFeatures |
|  |  |
|  | np.random.seed(0) |
|  | x = 2 - 3 \* np.random.normal(0, 1, 20) |
|  | y = x - 2 \* (x \*\* 2) + 0.5 \* (x \*\* 3) + np.random.normal(-3, 3, 20) |
|  |  |
|  |  |
|  | # transforming the data to include another axis |
|  | x = x[:, np.newaxis] |
|  | y = y[:, np.newaxis] |
|  |  |
|  | polynomial\_features= PolynomialFeatures(degree=2) |
|  | x\_poly = polynomial\_features.fit\_transform(x) |
|  |  |
|  | model = LinearRegression() |
|  | model.fit(x\_poly, y) |
|  | y\_poly\_pred = model.predict(x\_poly) |
|  |  |
|  | rmse = np.sqrt(mean\_squared\_error(y,y\_poly\_pred)) |
|  | r2 = r2\_score(y,y\_poly\_pred) |
|  | print(rmse) |
|  | print(r2) |
|  |  |
|  | plt.scatter(x, y, s=10) |
|  | # sort the values of x before line plot |
|  | sort\_axis = operator.itemgetter(0) |
|  | sorted\_zip = sorted(zip(x,y\_poly\_pred), key=sort\_axis) |
|  | x, y\_poly\_pred = zip(\*sorted\_zip) |
|  | plt.plot(x, y\_poly\_pred, color='m') |
|  | plt.show() |

To generate polynomial features (here 2nd degree polynomial)  
------------------------------------------------------------

polynomial\_features = PolynomialFeatures(degree=2)  
x\_poly = polynomial\_features.fit\_transform(x)

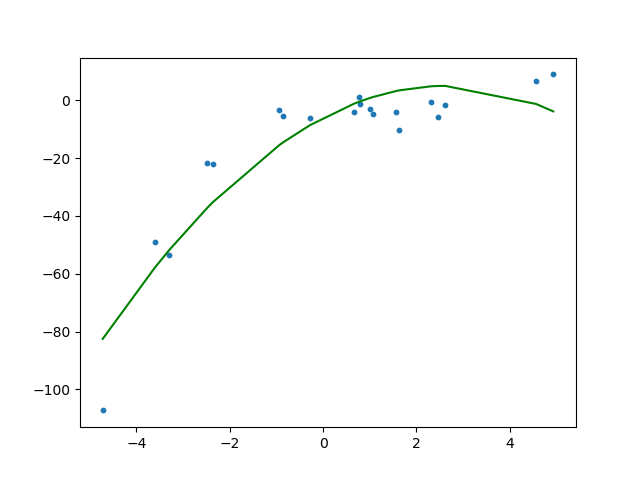
Explaination  
------------

Let's take the first three rows of X:   
[[-3.29215704]  
 [ 0.79952837]  
 [-0.93621395]]

If we apply polynomial transformation of degree 2, the feature vectors become

[[-3.29215704 10.83829796]  
 [ 0.79952837 0.63924562]  
 [-0.93621395 0.87649656]]

Fitting a linear regression model on the transformed features gives the below plot.

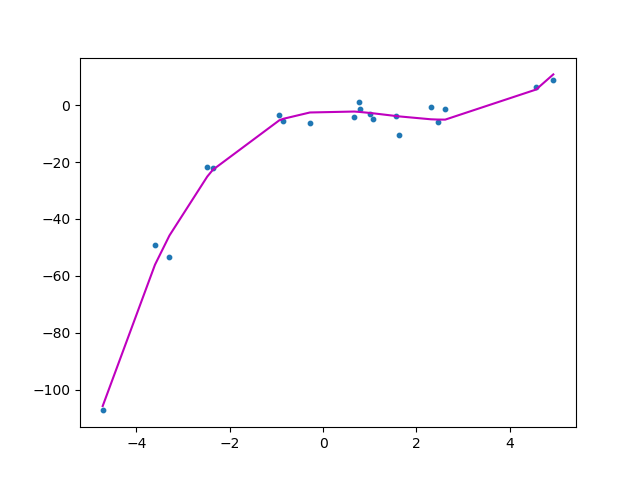


It is quite clear from the plot that the quadratic curve is able to fit the data better than the linear line. Computing the RMSE and R²-score of the quadratic plot gives:

RMSE of polynomial regression is **10.120437473614711**.  
R2 of polynomial regression is **0.8537647164420812**.

***We can see that RMSE has decreased and R²-score has increased as compared to the linear line.***

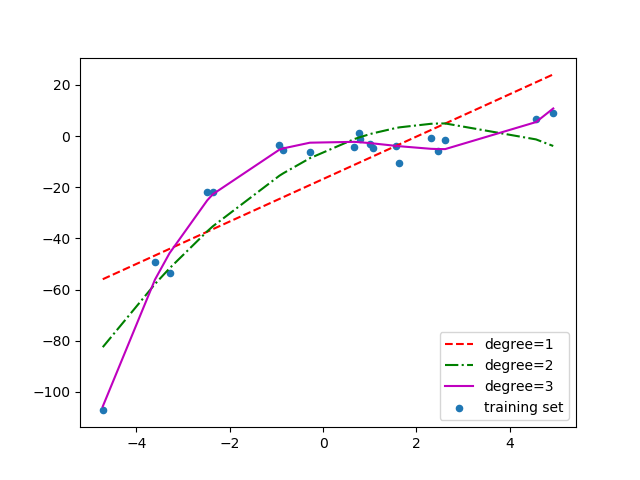
If we try to fit a cubic curve (degree=3) to the dataset, we can see that it passes through more data points than the quadratic and the linear plots.



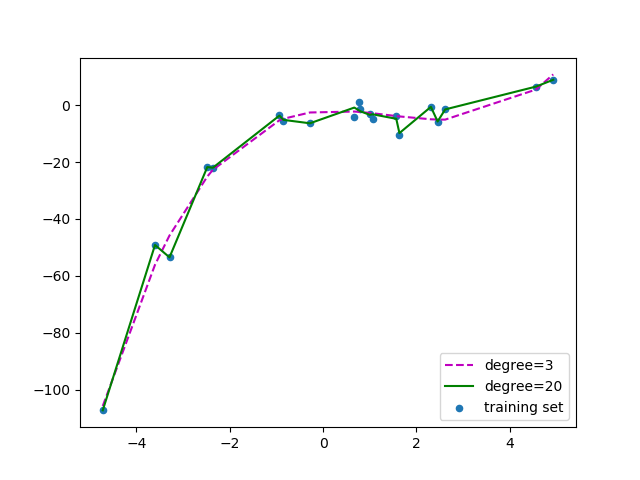
The metrics of the cubic curve is

RMSE is **3.449895507408725**  
R2 score is **0.9830071790386679**

Below is a comparison of fitting linear, quadratic and cubic curves on the dataset.



If we further increase the degree to 20, we can see that the curve passes through more data points. Below is a comparison of curves for degree 3 and 20.



For degree=20, the model is also capturing the noise in the data. This is an example of **over-fitting**. Even though this model passes through most of the data, it will fail to generalize on unseen data.

***To prevent over-fitting, we can add more training samples so that the algorithm doesn’t learn the noise in the system and can become more generalized.****( Note: adding more data can be an issue if the data is itself noise).*

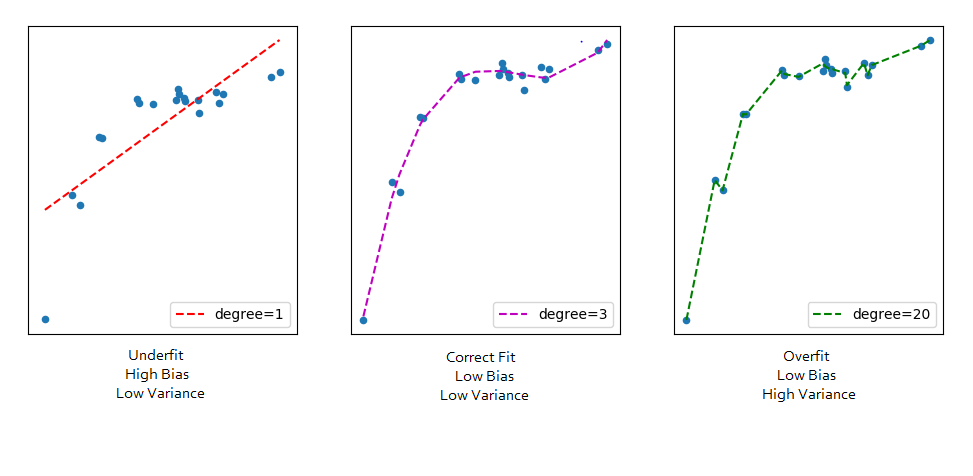
How do we choose an optimal model? To answer this question we need to understand the bias vs variance trade-off.

**The Bias vs Variance trade-off**

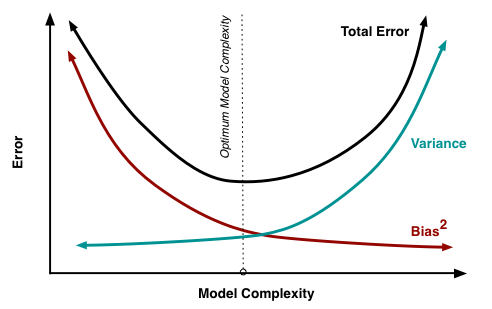
**Bias**refers to the error due to the model’s simplistic assumptions in fitting the data. A high bias means that the model is unable to capture the patterns in the data and this results in **under-fitting**.

**Variance**refers to the error due to the complex model trying to fit the data. High variance means the model passes through most of the data points and it results in **over-fitting** the data.

The below picture summarizes our learning.



From the below picture we can observe that as the model complexity increases, the bias decreases and the variance increases and vice-versa. Ideally, a machine learning model should have **low variance and low bias**. But practically it’s impossible to have both. Therefore to achieve a good model that performs well both on the train and unseen data, a **trade-off** is made.

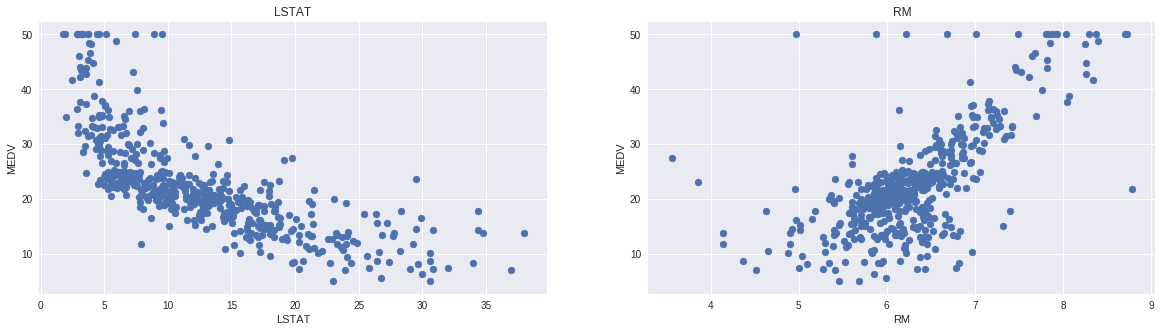


Source: <http://scott.fortmann-roe.com/docs/BiasVariance.html>

Till now, we have covered most of the theory behind Polynomial Regression. Now, let’s implement these concepts on the Boston Housing dataset we analyzed in the [previous](https://towardsdatascience.com/linear-regression-on-boston-housing-dataset-f409b7e4a155) blog.

**Applying Polynomial Regression to the Housing dataset**

It can be seen from the below figure that LSTAT has a slight non-linear variation with the target variable MEDV. We will transform the original features into higher degree polynomials before training the model.



Let’s define a function which will transform the original features into polynomial features of a given degree and then apply Linear Regression on it.

|  |
| --- |
|  |
|  |  |
|  | from sklearn.preprocessing import PolynomialFeatures  def create\_polynomial\_regression\_model(degree): |
|  | "Creates a polynomial regression model for the given degree" |
|  |  |
|  | poly\_features = PolynomialFeatures(degree=degree) |
|  |  |
|  | # transforms the existing features to higher degree features. |
|  | X\_train\_poly = poly\_features.fit\_transform(X\_train) |
|  |  |
|  | # fit the transformed features to Linear Regression |
|  | poly\_model = LinearRegression() |
|  | poly\_model.fit(X\_train\_poly, Y\_train) |
|  |  |
|  | # predicting on training data-set |
|  | y\_train\_predicted = poly\_model.predict(X\_train\_poly) |
|  |  |
|  | # predicting on test data-set |
|  | y\_test\_predict = poly\_model.predict(poly\_features.fit\_transform(X\_test)) |
|  |  |
|  | # evaluating the model on training dataset |
|  | rmse\_train = np.sqrt(mean\_squared\_error(Y\_train, y\_train\_predicted)) |
|  | r2\_train = r2\_score(Y\_train, y\_train\_predicted) |
|  |  |
|  | # evaluating the model on test dataset |
|  | rmse\_test = np.sqrt(mean\_squared\_error(Y\_test, y\_test\_predict)) |
|  | r2\_test = r2\_score(Y\_test, y\_test\_predict) |
|  |  |
|  | print("The model performance for the training set") |
|  | print("-------------------------------------------") |
|  | print("RMSE of training set is {}".format(rmse\_train)) |
|  | print("R2 score of training set is {}".format(r2\_train)) |
|  |  |
|  | print("\n") |
|  |  |
|  | print("The model performance for the test set") |
|  | print("-------------------------------------------") |
|  | print("RMSE of test set is {}".format(rmse\_test)) |
|  | print("R2 score of test set is {}".format(r2\_test)) |

Next, we call the above function with the degree as 2.

create\_polynomial\_regression\_model(2)

The model’s performance using Polynomial Regression:

**The model performance for the training set**   
-------------------------------------------   
RMSE of training set is 4.703071027847756   
R2 score of training set is 0.7425094297364765

**The model performance for the test set**  
-------------------------------------------   
RMSE of test set is 3.784819884545044   
R2 score of test set is 0.8170372495892174