Published in Towards Data Science

Animesh Agarwal Follow

Oct 9, 2018 · 6 min read · Listen

Polynomial Regression

This is my third blog in the Machine Learning series. This blog requires prior knowledge of Linear Regression. If you don't know about Linear Regression or need a brush-up, please go through the previous articles in this series.

- Linear Regression using Python
- Linear Regression on Boston Housing Dataset

> *Linear regression requires the relation between the dependent variable and the independent variable to be linear. What if the distribution of the data was more complex as shown in the below figure? Can linear models be used to fit non-linear data? How can we generate a curve that best captures the data as shown below? Well, we will answer these questions in this blog.*

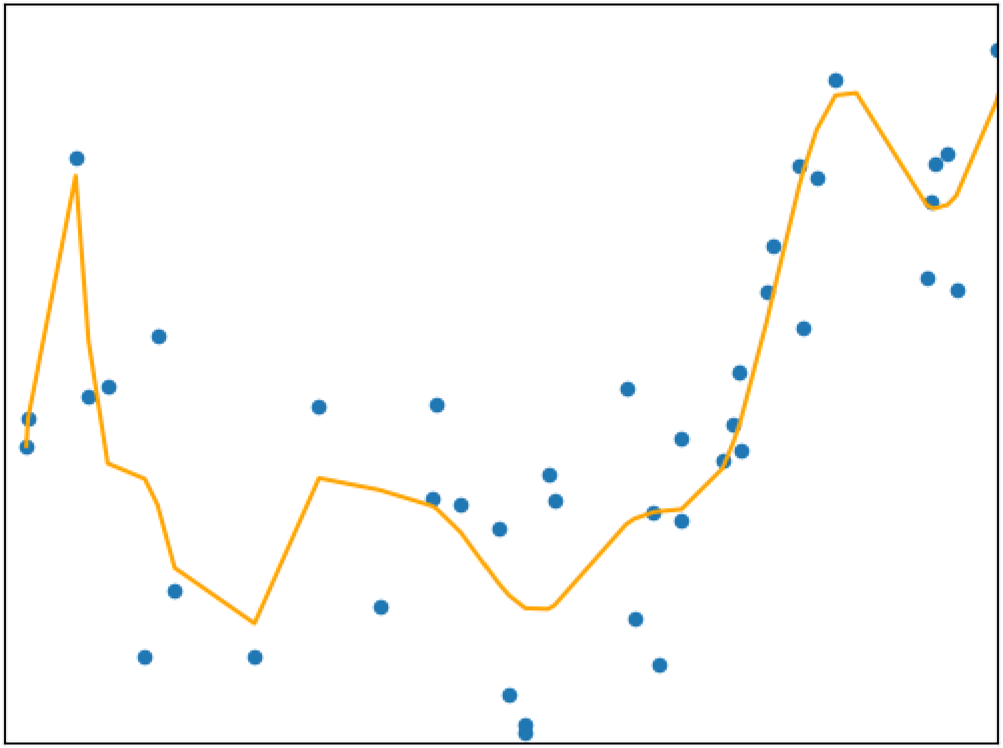


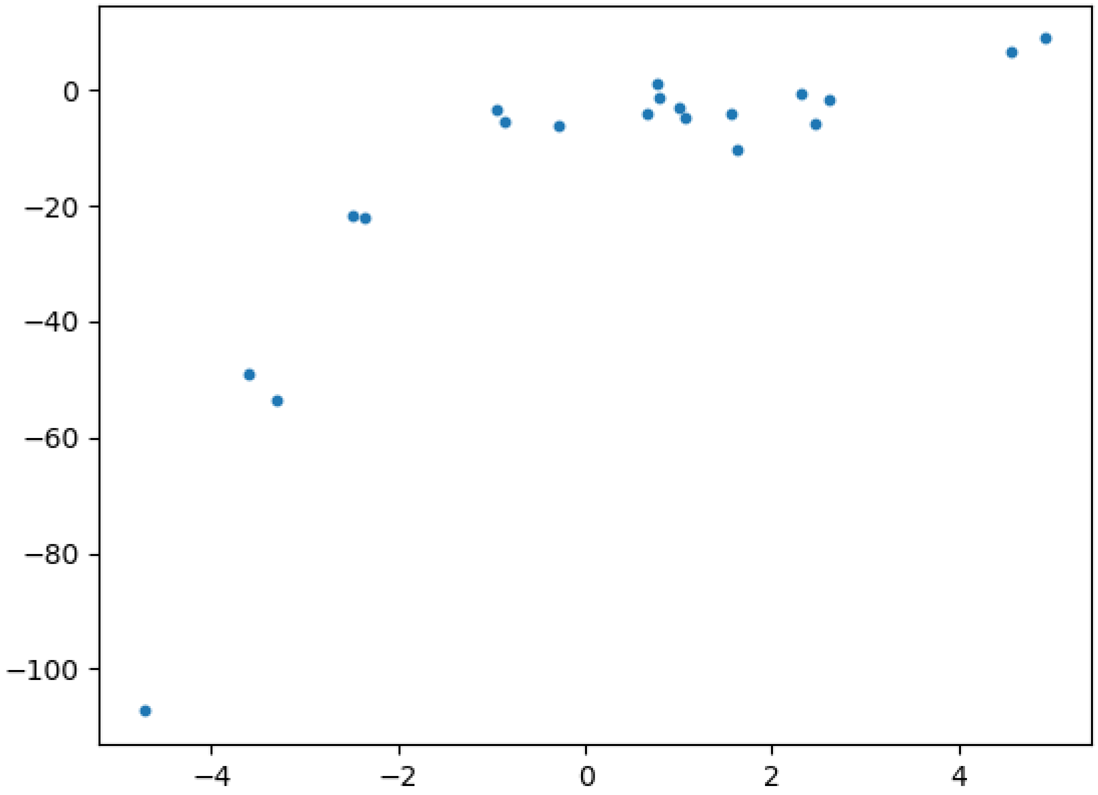
Table of Contents

Why Polynomial Regression?

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

```python
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



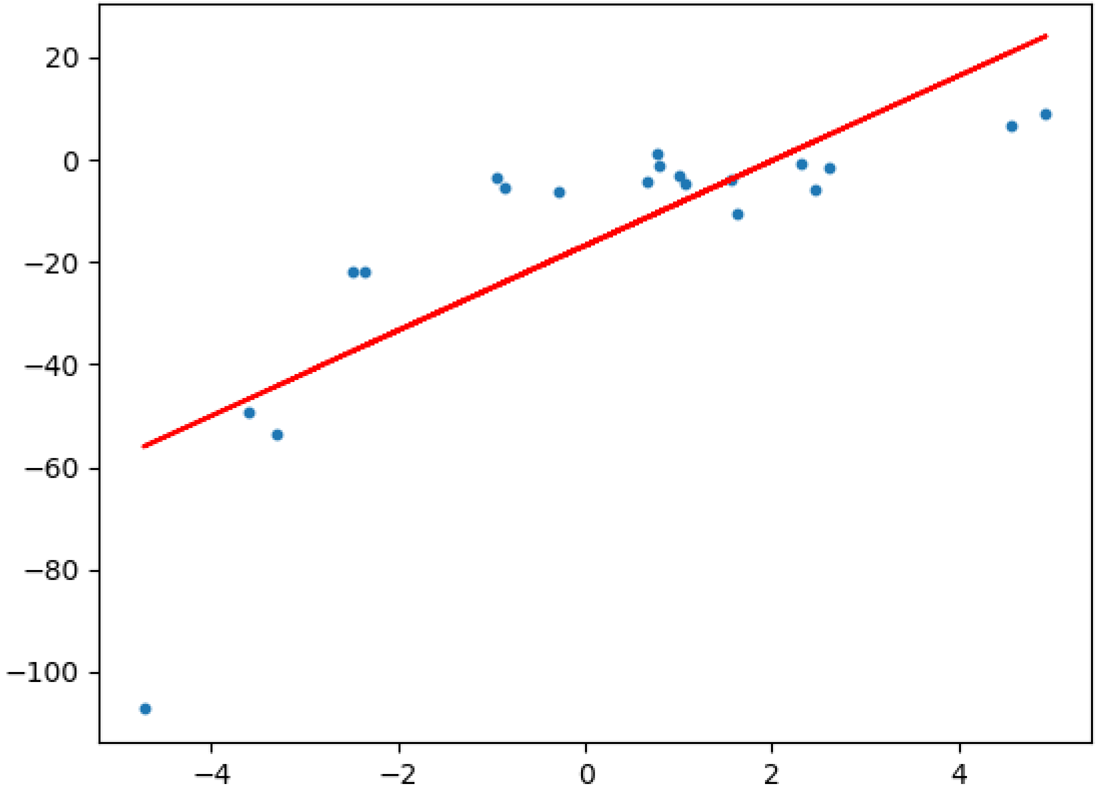
Let’s apply a linear regression model to this dataset.

```
import numpy as np
```

```
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()
```

Linear Regression on non linear data.py hosted with ❤ by GitHub view raw

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:

> ***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,

$$Y = \theta_0 + \theta_1 x$$

can be transformed to

$$Y = \theta_0 + \theta_1 x + \theta_2 x^2$$

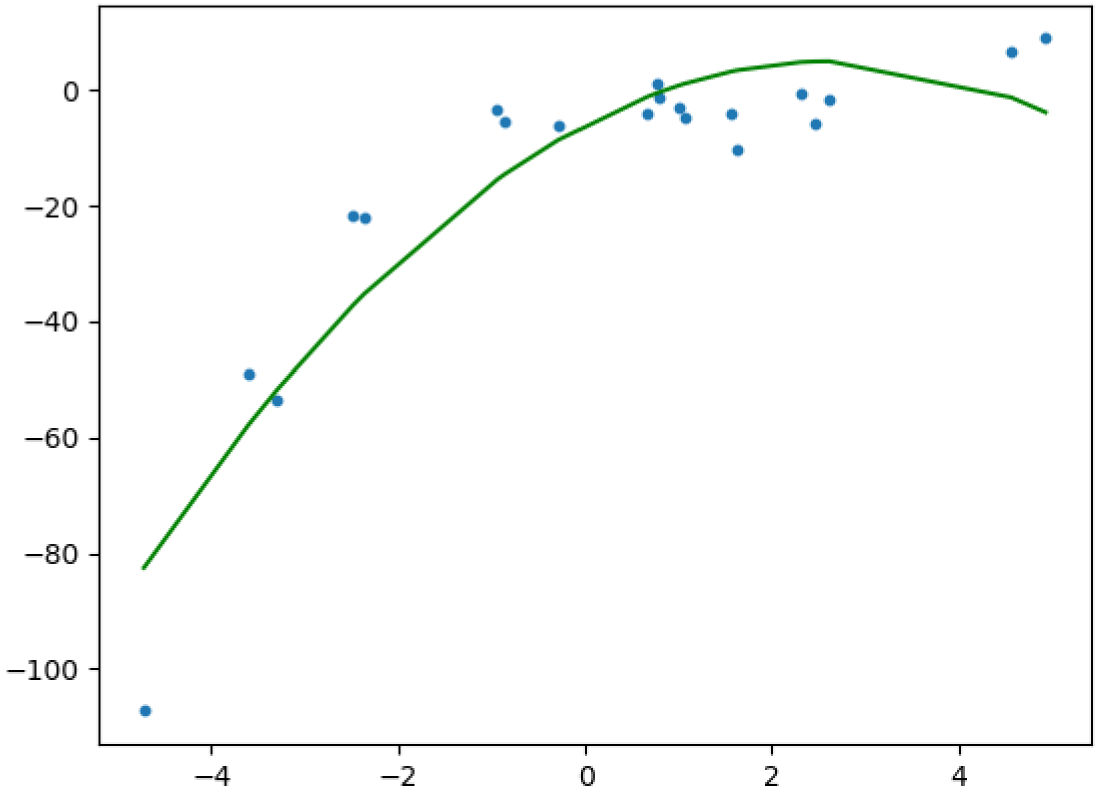
> *This is still considered to be* ***linear model*** *as the coefficients/weights associated with the features are still linear. x^2 is only a feature. However the curve that we are fitting is* ***quadratic*** *in nature.*

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 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
```

```
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]]
```

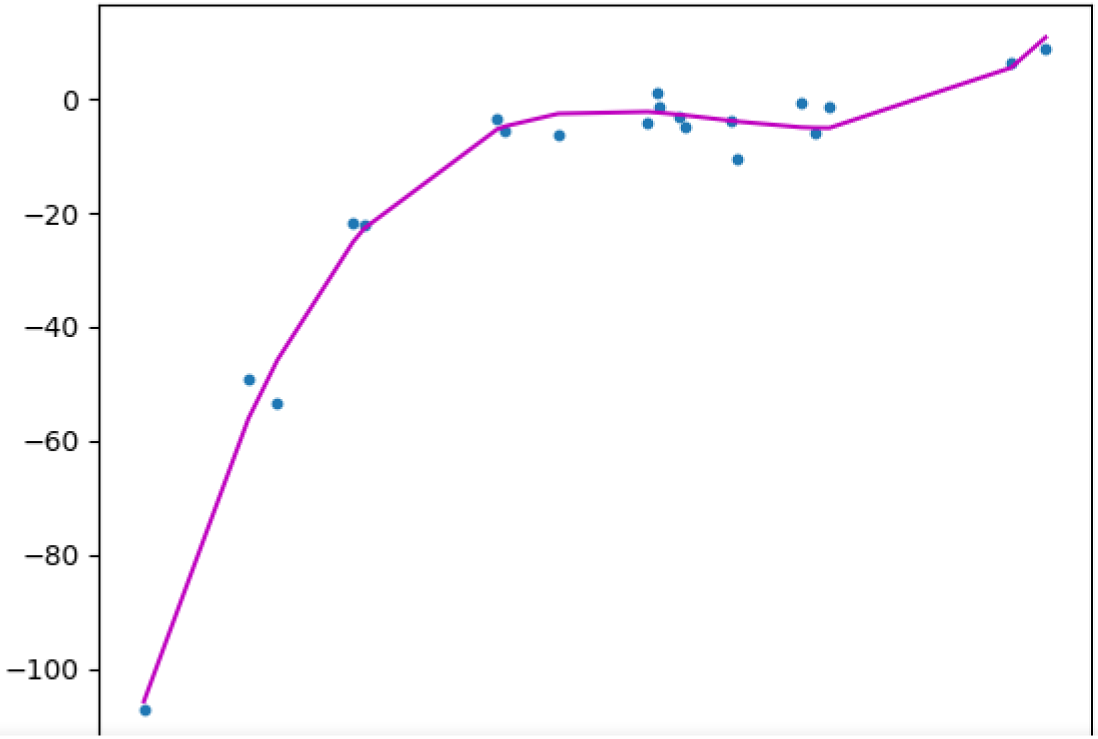
Fitting a linear regression model on the transformed features gives the below



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 R^2-score of the quadratic plot gives:

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

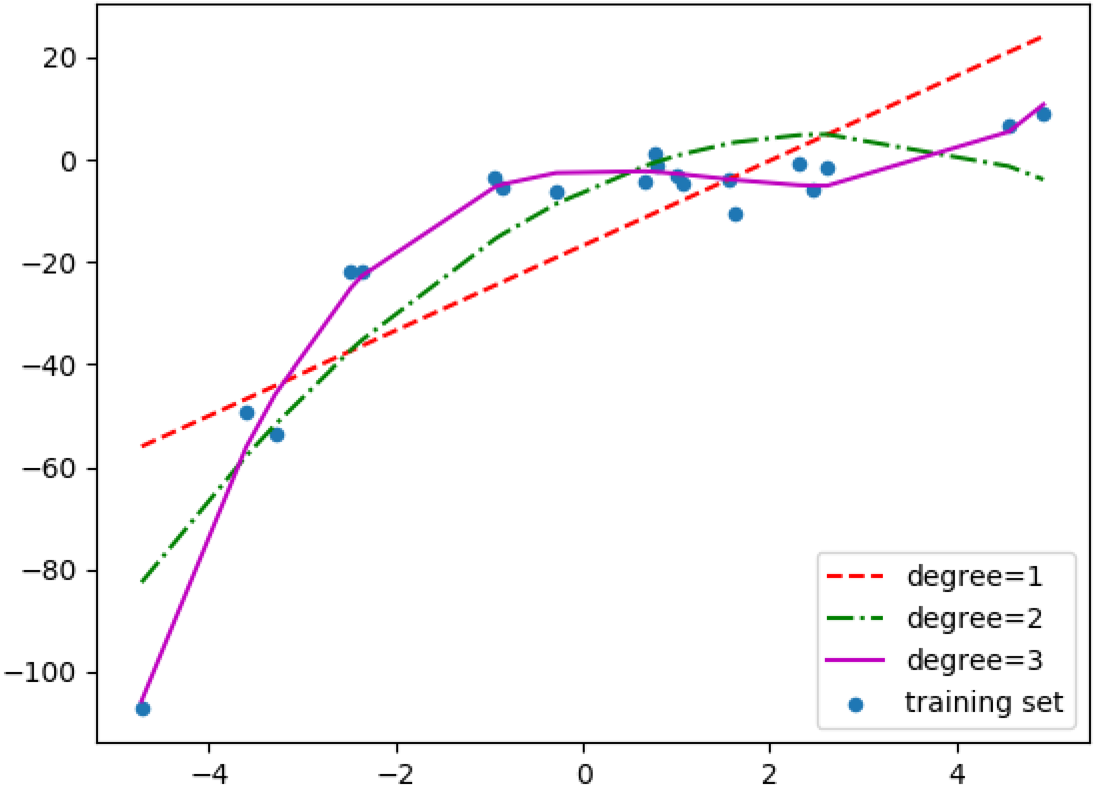
If we try to fit a cubic curve (degree=3) to the dataset, we can see that it passes through more data points than the quadratic and the linear plots.



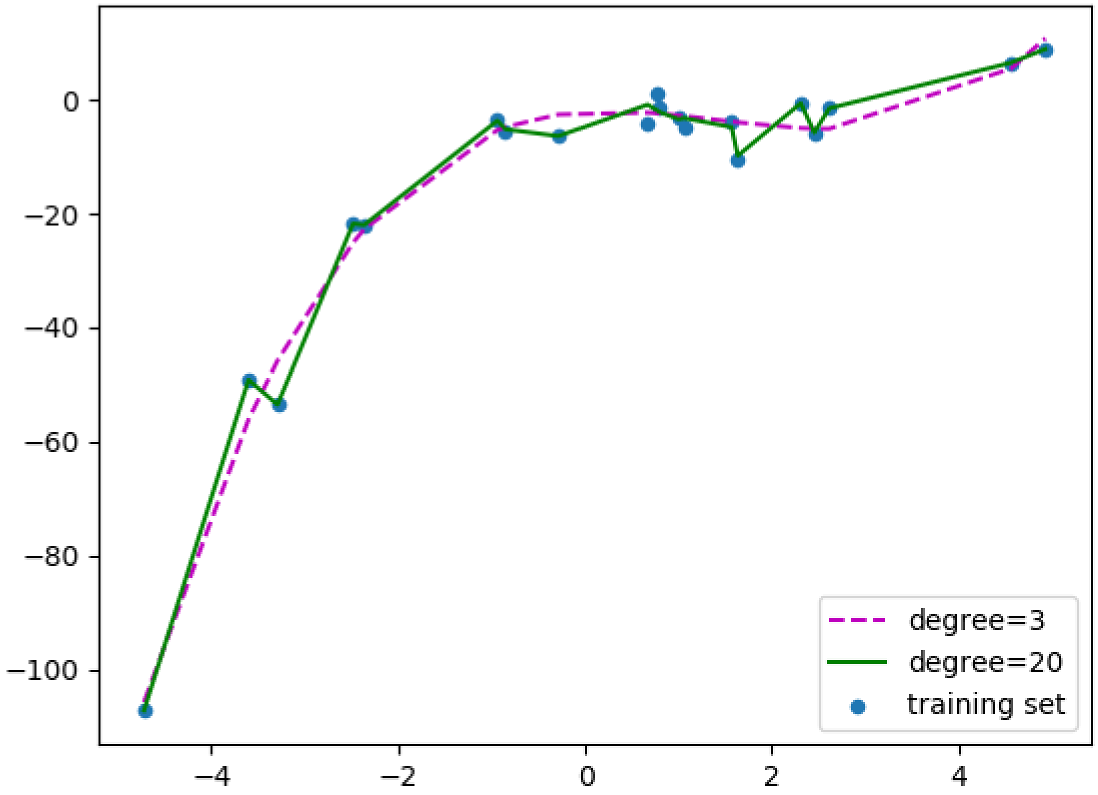
The metrics of the cubic curve is

```
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.

> ***generalized.*** *(Note: adding more data can be an issue if the data is itself noise).*

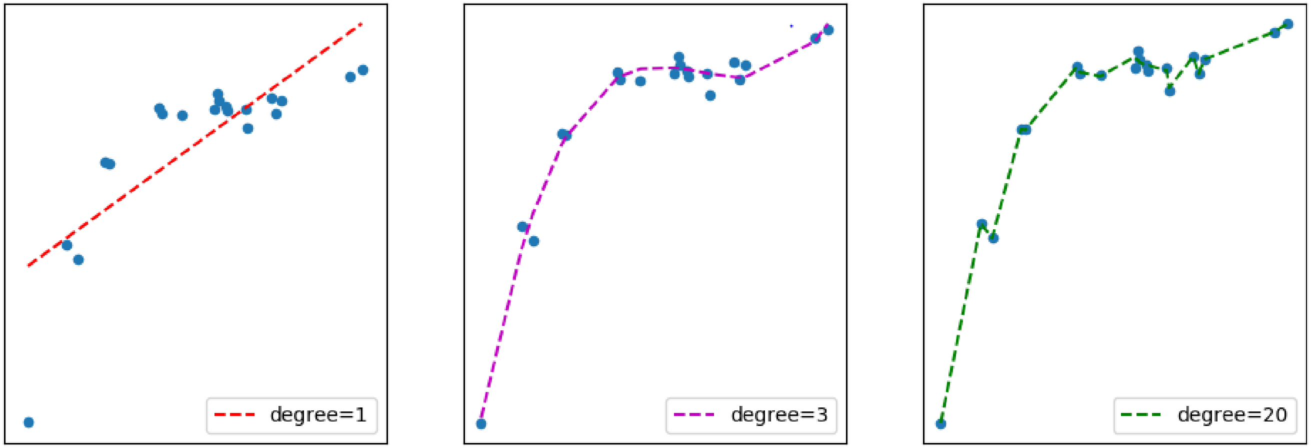
How do we choose an optimal model? To answer this question we need to understand the bias vs variance trade-off.

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.

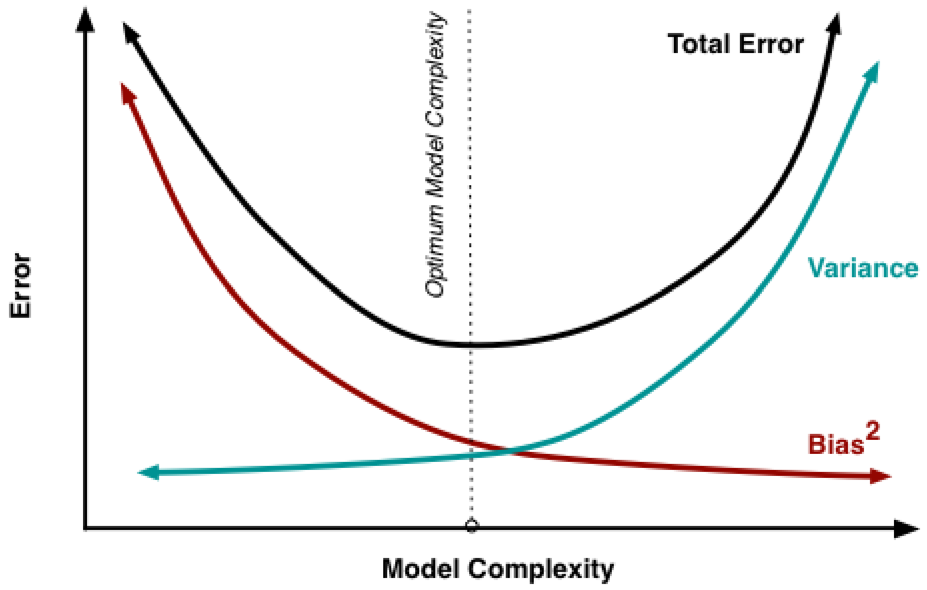


Underfit
High Bias
Low Variance

Correct Fit
Low Bias
Low Variance

Overfit
Low Bias
High Variance

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.

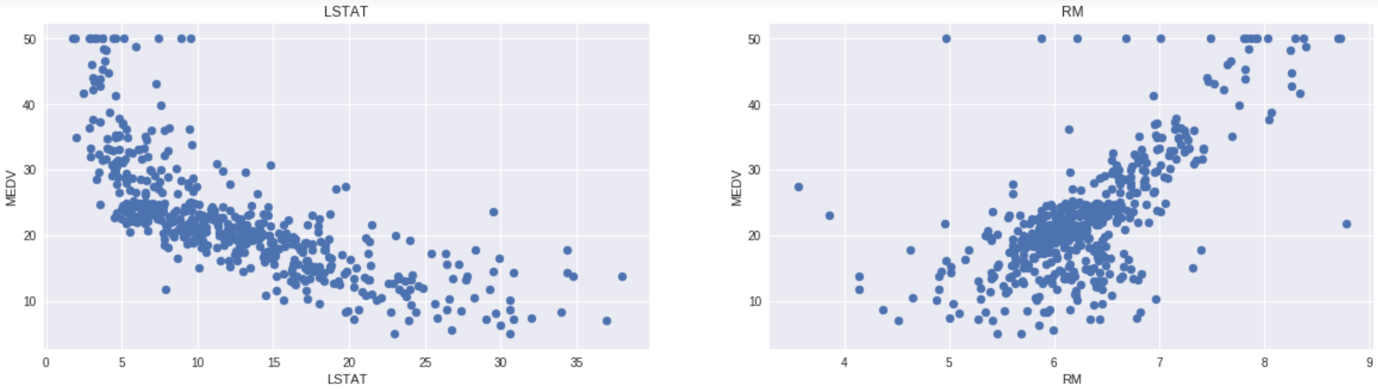


Source: http://scott.fortmann-roe.com/docs/BiasVariance.html

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 blog.

Applying Polynomial Regression to the Housing dataset

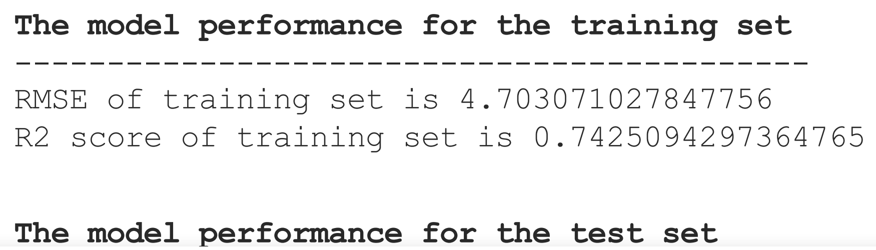
It can be seen from the below figure that `LSTAT` has a slight non-linear variation with the target variable `MEDV` . We will transform the original features into higher



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.

Next, we call the above function with the degree as 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
```

This is better than what we achieved using Linear Regression in the previous blog.

That’s all for this story. This Github repo contains all the code for this blog and the complete Jupyter Notebook used for Boston housing dataset can be found here.

Conclusion

In this Machine Learning series, we have covered Linear Regression, Polynomial Regression and implemented both these models on the Boston Housing dataset.

We will cover Logistic Regression in the next blog.

Thanks for Reading !!