upGrad

Summary

Introduction to Linear Regression

In this session, you understood the importance of linear models and learnt about the most commonly used linear model: linear regression. You began by understanding the internals of the algorithm and looking at a basic implementation of linear regression using the NumPy library. Furthermore, you learnt how to build regression models using the most commonly used libraries: statsmodels and scikit-learn. Here, you learnt about the different steps involved in model building and understood how to fine-tune your model. In this session, you covered the following topics:

- Introduction to Linear Models
- Linear Models in Practice
- Mathematics behind Linear Regression
- Linear Regression with NumPy
- Assumptions of Linear Regression
- Evaluation Metrics
- Linear Models as Benchmarks

Introduction to Linear Models

Linearity is a property between a pair of variables that determines whether or not their relationship can be represented by a straight line.

y = mx + c

In this segment, you learnt how to represent a linear relationship among more than two variables. Consider the following relationship.

$$y = m_1 x_1 + m_2 x_2 + m_3 x_3 \ldots m_n x_n + c$$

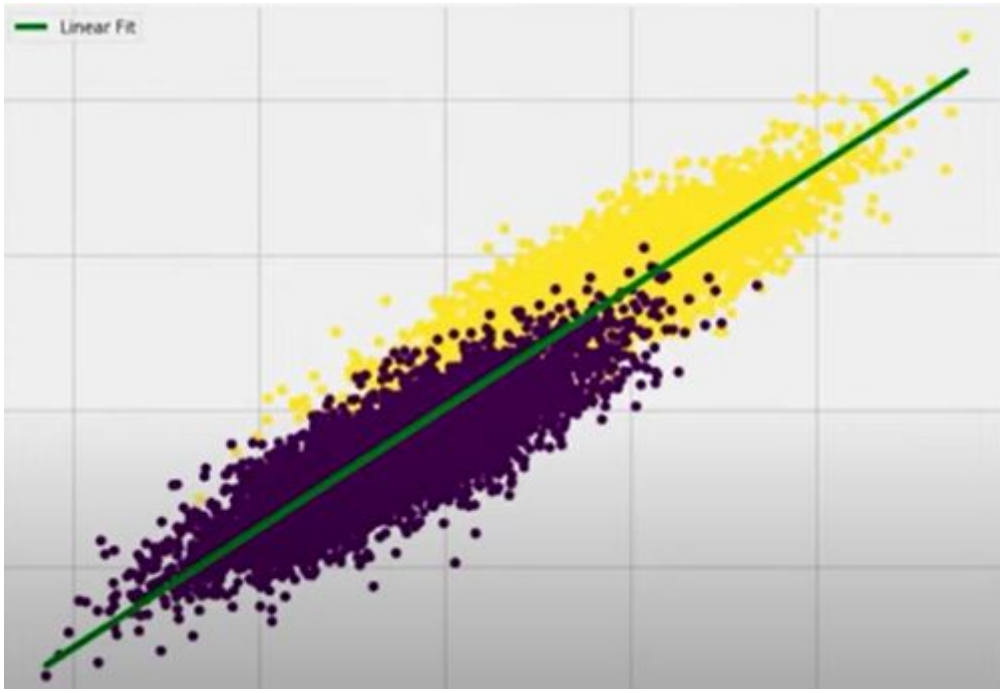
Here, y is the dependent variable (predictor variable) and ‘x1, x2, ..., xn are independent variables (output variables). In this segment, you also learnt about overfitting, which is a common phenomenon in machine learning, that occurs when a model becomes too specific to the data on which it is trained on and fails to generalise to other unseen data points. A model that has become too specific to a training data set has ‘learnt’ not only the hidden patterns in the data, but also the noise and inconsistencies in it. Note that noise is something that a model cannot learn, as it is random and does not follow a pattern. In a typical case of overfitting, the model performs quite well on the training data set, but performs quite poorly on the test data set.

upGrad

Linear Regression

A system of linear equations can be converted to a simple matrix form, which can be used later to identify the unknown variables m and c in the equation, y = mx+c.

Unfortunately, in the real world, all the data points may not fit any linear line perfectly, but they can be approximated to fit a certain line as shown in the plot given below.



Two-point form: If the two points are (x1, y1) and (x2, y2), then the formula for the line is as follows:

y - y1 = $\frac{y2-y1}{x2-x1}$ (x - x1).

If the two points are (x_1, x_2) and (y_1, y_2), then

$$y_1 = mx_1 + c$$
$$y_2 = mx_2 + c$$

Which can be written as:

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} x_1 & 1 \\ x_2 & 1 \end{bmatrix} \begin{bmatrix} m \\ c \end{bmatrix}$$

Or,

$$\boldsymbol{d = Ab}$$

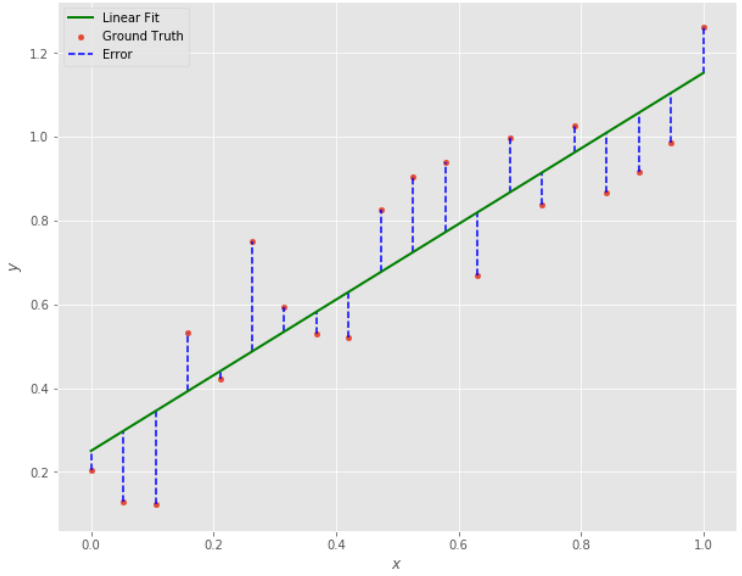
Systems of equations are of the following three types:

1. **Ideal system**: Number of equations = Number of unknowns
   Such a system will have unique solutions
2. **Underdetermined systems**: Number of equations < Number of unknowns
   Such a system will have ∞ solutions (or no solution)
3. **Overdetermined system**: Number of equations > Number of unknowns
   Such a system will have no unique solution

In machine learning, you will mostly be dealing with the third criterion, wherein the number of equations, or, in other words, the number of data points, is greater than the number of unknowns. There would be no unique solution in this case. All you need to do is try to find the best line that would take the values x1, x2, ..., xn and give a value of y that is closest to the actual value.

Cost Function

An error or a residual is the difference between the predicted value and the actual value. For example, if y = mx is the final assumed line, then for a given point (x1, y1), the error associated would be y1 − mx1.



This overall error is also known as the sum of squared errors (SSE) or the residual sum of squares (RSS). It is given by $\Sigma(y_i - mx_i)^2$. Here, the objective is to minimise the SSEs. Assuming the line y = mx best represents the set of points (x1, y1) (x2, y2) ... (xn, yn), you try to solve the equation in order to minimise the SSEs. This would provide an analytical solution that a line with slope $m = \frac{y^T x}{x^T x}$ is the ideal line in this case.

Remember the following points:

- Though the error is (y1 - mx1), (y2 - mx2) and so on for the different points, we do not minimise the sum of errors, i.e., (y1 - mx1) + (y2 - mx2) + …; we minimise the SSEs. We square it because the errors for the different points can nullify one another; hence, giving an incorrect result.
- All machine learning algorithms work towards minimising an objective function. Here, the objective function is either the RSS or the SSE. It varies from model to model. The objective function is also known as the cost function.
- Generally, we minimise the cost function/objective function, because it results in the minimisation of errors.

Multiple linear regression

For a generalised case, wherein you have n features (a number of rooms, sizes, floors, etc), you can represent the individual values as shown in the equation given below:

$$\boldsymbol{X} = \begin{bmatrix} x_{11} & x_{12} \cdots & x_{1n} \\ x_{21} & \ddots & \vdots \\ \vdots & & \\ x_{m1} & x_{m2} \cdots & x_{mn} \end{bmatrix}$$

Thus, the final equation for y (price) can be represented as follows:

$$\hat{y}_i = x_{i1}\beta_1 + x_{i2}\beta_2 + \cdots x_{in}\beta_n$$

$$\therefore \boldsymbol{y} = \boldsymbol{X\beta}$$

The objective is to find β. Note that β in the equation y = Xβ given above is a vector as shown below:

$$\beta = \begin{bmatrix} \beta_1 \\ \beta_2 \\ . \\ . \\ \beta_n \end{bmatrix}$$

Here, you consider the cost function, which is nothing but the sum of squares, and minimise it to find the vector β. The solution to β is an analytical one, which is represented below.

$$\hat{\beta} = (X^T X)^{-1} X^T Y$$

After finding out β using the given values x_i, you can find out the value of $\hat{yi}$. Hence,

$$\hat{y} = X.\hat{\beta}$$

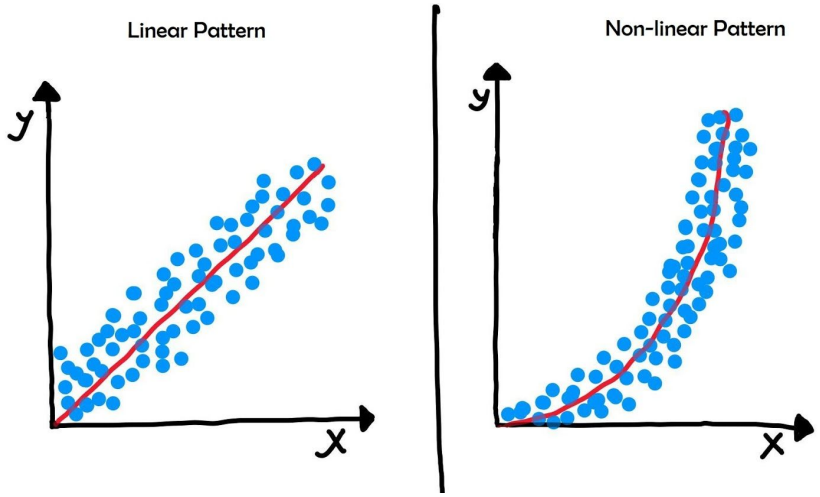
This is quite similar to the case wherein you have one independent variable. Now, the difference between the value $\hat{y}$ predicted above and the actual value y is the error ε. The MSE (Mean Square Error) is defined as follows:

$$MSE = ||\epsilon||^2 = ||(y - X\hat{\beta}||^2$$

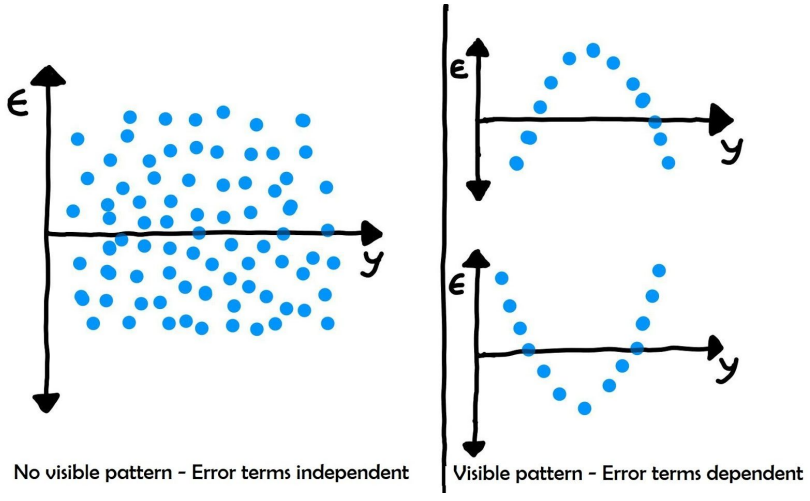
Assumptions of Linear Regression

The assumptions that are made while fitting a linear model between X and Y are as follows:

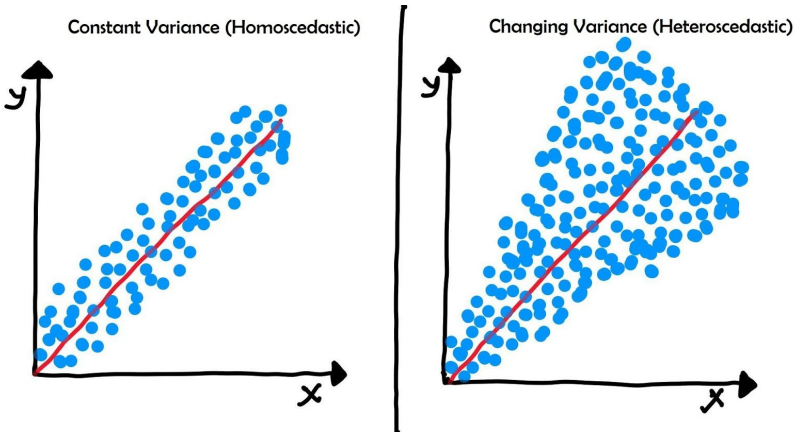
1. A linear relationship between X and Y: X and Y should display some form of linear relationship (as shown in the plots given below); otherwise, it is fruitless fitting a linear model between them.



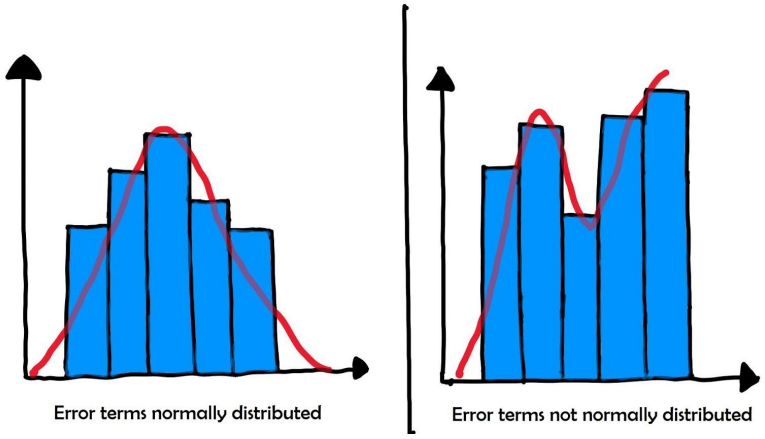
2. Independent error terms: The error terms should not be dependent on one another (as shown in the plots given below)



3. Constant variance (homoscedasticity) in error terms:
    - The variance should not increase (or decrease) with change in the error values.
    - Also, the variance should not follow any pattern as the error terms change.



4. Error terms distributed normally with a mean of 0 (not X, Y):
    - You need not worry if the error terms are not distributed normally in case you only want to fit a line and not make any further interpretations.
    - However, if you are willing to make some inferences on the model that you have built (you will learn about this in the upcoming segments), then you need to have an understanding of the distribution of error terms. One particular repercussion of error terms not being distributed normally is that the p-values that are obtained during the hypothesis test to determine the significance of the coefficients become unreliable. (You will learn about this in a subsequent segment.)
    - The assumption of normality, since it has been observed that the error terms generally follow a normal distribution, is made with the mean equal to 0 in most cases.

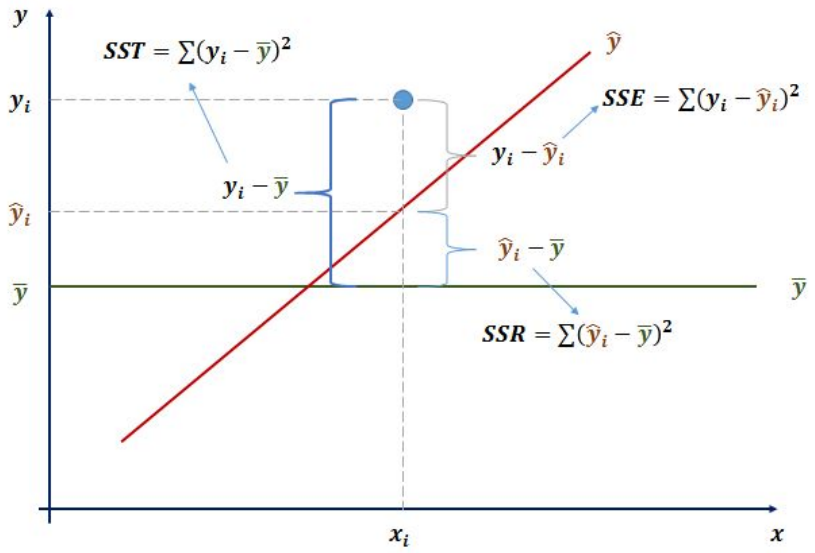


5. All the columns of the predictor variables should be linearly independent. This is essential because if any dependency exists between features (columns of X), then the determinant of X would be 0, which makes X inverse undefined. Hence, there should be no dependency among features. The existence of such dependencies is known as **multicollinearity.** Multicollinearity occurs when the

input data set contains predictor (independent) variables that are related to one another? In simple terms, in a model that has been built using several independent variables, some of the variables might be interrelated, and, therefore, redundant in the model

Evaluation Metrics

To understand how well the model is representing the data and how it performs on unseen data, you need to understand the plot given below.



Here, (xi, yi) represents your data point. $\hat{y}$ represented the predicted value in the case of (xi, yi). The sum of squared errors (SSE) or the residual sum of squares (RSS) is $\Sigma(y_i - \hat{y})^2$. The sum of squares defines the variance of the residuals as shown below.

RSS = $\Sigma(y_i - \hat{y})^2$

The strength of the line given above should be defined around this particular RSS because it is the value/objective function that you need to minimise. Now, the most simplistic or trivial linear regression model that you can build on a set of data points is the line y that passes through the mean of all the target values, where y = $\bar{y}$.

Hence, the total sum of squares (TSS) (i.e., the maximum variation in the data) equals $\Sigma(y_i - \bar{y})^2$.

Therefore, we define one of the commonly used evaluation metrics, R^2, as follows:

$$R^2 = \frac{TSS - RSS}{TSS} = 1 - \frac{RSS}{TSS}$$

R^2 is a measure of the ratio of the total variation captured by the model and the total variation present in the data. If $\hat{y}$ in the model $y = \hat{y}$ becomes $\bar{y}$, then RSS = TSS; hence, $R^2 = 0$. And if $\hat{y}$ in the model $y = \hat{y}$ becomes such that $\hat{y}_i = y_i$ for all i, then RSS = 0 and $R^2 = 1$. Hence, for the perfect model, $R^2 = 1$. Thus, as R^2 increases, the model fit improves. The coefficient of determination, or R^2, determines how well a model captures the variance in the data.

R^2 should be the go-to metric for you to make predictions, although there are certain issues with using this metric. The number of predictor variables associated with y should also be considered in the calculations. This is why the adjusted R-squared value is defined as follows:

$$Adjusted R^2 = 1 - \frac{(1-R^2)(N-1)}{N-p-1}$$

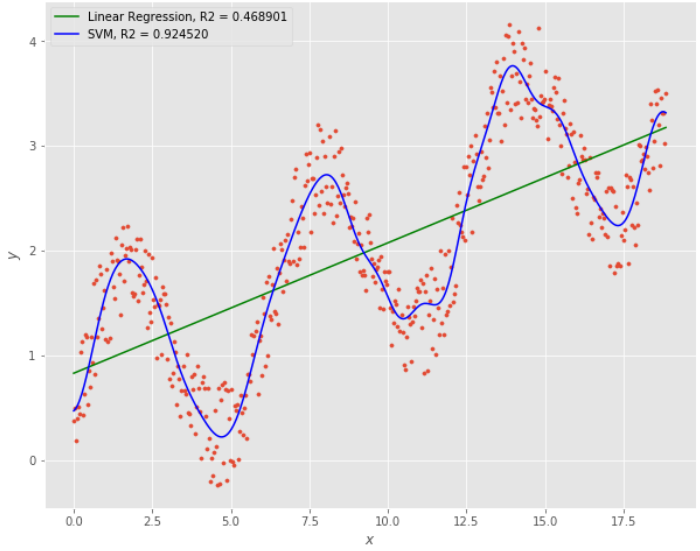
,

where
N = Number of points in the data and
p = Number of independent predictor variables, excluding the constant.

Important: Note that the r2 score, R^2 and R-squared are the same and can be used interchangeably.

Linear Models as Benchmarks

Linear models are the simplest models to build, and the performance of any non-linear model that you build should be better than that of a linear model. Linearity is a simple, naïve interpretation of data.



upGrad

Summary

Linear Regression: Model Building

In this session, you learnt about the steps that are required to build a linear regression model. You also learnt how to read and visualise the data set. Next, you learnt how to build a linear model using two different libraries: statsmodels and SKLearn. You also learnt about some of the key feature selection methods that are used for selecting relevant features for model building.

Simple Linear Regression with scikit-learn

In this segment, you learnt how to build a linear model using the scikit-learn API. The first step in building a linear regression model is to create an instance of the linear regression class by defining an object as shown below.

```
lr = LinearRegression()
```

The next step is to build the model using the .fit() method and specifying the independent and dependent variables as follows:

```
lr.fit(x, y)
```

Here, x is the input/independent variable and y is the output/dependent variable. The .fit() method is used for building a linear regression model. Now, the lr variable has all the information pertaining to the model.

Using the lr.predict() method, you can predict the values of y for given values of x. lr.coef_ and lr.intercept_ return the coefficients and the intercept of the regression model, respectively.

`help(lr)` is used to learn more about the other parameters that are defined in the function.

Note that the .fit() method expects the X parameter to be a 2D array, not a 1D array. In order to convert the 1D array to a 2D array, you need to use .reshape(-1,1). Using lr.fit(x,y) with a 1D array would give you the following error, if you do not reshape X:

```
ValueError: Expected 2D array, got 1D array instead:
```

You can read more about this in the link provided below.

Additional Link

Stackoverflow thread explaining the significance of - 1 in the numpy reshape

upGrad

Multiple Linear Regression With scikit-learn

In this segment, you learnt how to apply a regression model on the Boston Housing data set.

Note: The data set used in the demonstration comes loaded with the scikit-learn library, and to use it, you need to import the sklearn datasets first. Using .load_boston(), you can directly load the data set necessary for the demonstration.

The steps to create a dataframe have been shown in the code below.

```
# Importing pandas
Import pandas as pd

# Importing inbuilt data sets
from sklearn import datasets as data

# Using .load_boston() to load the data set
boston = data.load_boston()

# Creating a dataframe out of the loaded data set
df = pd.DataFrame(boston.data, columns=boston.feature_names)
df['value'] = boston.target
df.head()
```

As in the earlier case of simple linear regression, to build a multiple linear regression model, you need to define the X and Y parameters of the model.

X = boston.data [predictor variables]
Y = boston.target [target variables]

Note: X and Y are not data frames but **arrays**. It is important to remember this whenever you are building a model.

To determine the performance of this model, you first need to import the necessary metrics from the sklearn library as shown below:

```
from sklearn.metrics import mean_squared_error, r2_score
```

After importing the metrics, you need to call the mean_squared_error and r2_score functions by passing the necessary functional parameters. The mean_squared_error is the sum of the squares of the residuals divided by the total data points, or, in other words, the number of y values. It is expressed as follows:

MSE = $\sum_{i=1}^{n} (yi - \widehat{yi})^2$ /n

Use the following function to call mean_squared_error:

```
mean_squared_error(y, yhat)
```

Similarly, use the following function to call r2_score:

```
r2_score(y, yhat)
```

Here, y is the actual value and $\widehat{yi}$ is the predictions made by the model. r2_score or the r squared metric explains how much of the variance of y is explained by the model. On executing the code, you found that the model had explained 74% of the variance of y, which means it is performing well.

It is extremely important to rescale variables so that they have comparable scales. If you do not have comparable scales, then some of the coefficients obtained by fitting the regression model might be very large or very small as compared with the others.

So, if you observe the scales of the values in the Boston Housing data set, then you will observe that the scale of TAX is in the range of 100s, whereas those of DIS and NOX are in the range of 10s, as shown in the table given below.

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000
mean	3.593761	11.363636	11.136779	0.069170	0.554695	6.284634	68.574901	3.795043	9.549407	408.237154	18.455534	356.674032
std	8.596783	23.322453	6.860353	0.253994	0.115878	0.702617	28.148861	2.105710	8.707259	168.537116	2.164946	91.294864
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000	2.900000	1.129600	1.000000	187.000000	12.600000	0.320000
25%	0.082045	0.000000	5.190000	0.000000	0.449000	5.885500	45.025000	2.100175	4.000000	279.000000	17.400000	375.377500
50%	0.256510	0.000000	9.690000	0.000000	0.538000	6.208500	77.500000	3.207450	5.000000	330.000000	19.050000	391.440000
75%	3.647423	12.500000	18.100000	0.000000	0.624000	6.623500	94.075000	5.188425	24.000000	666.000000	20.200000	396.225000
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	100.000000	12.126500	24.000000	711.000000	22.000000	396.900000

You can rescale this data using a min-max scalar feature transformer of the sklearn library. First, you need to import the MinMaxScalar from the sklearn preprocessing module using the following code:

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
```

Next, using the .fit() method, you can transform the input data set into a scaled data set as follows:

```
# Apply scaler() to all the columns except the 'yes-no' and 'dummy' variables
num_vars = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX',
       'PTRATIO', 'B', 'LSTAT', 'value']
```

```
df[num_vars] = scaler.fit_transform(df[num_vars])
```

In the unscaled data set, you will notice that some features are associated with high-value coefficients, whereas some others have coefficients of almost negligible values. A coefficient explains the effect of a particular independent variable on the dependent variable when all other variables remain constant.

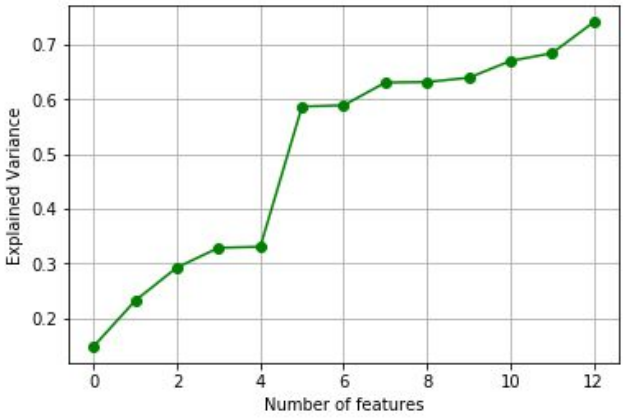
Cumulative Feature Variances

In this segment, you learnt about cumulative feature variances. Consider the following code where we have used the explained_variance_score to denote the proportion of variance that has been explained for the input columns. Note that the data used belongs to the boston housing dataset.

```
y = boston.target
X = boston.data
variances = []
for i in range(X.shape[1]):
    xx = X[:, :(i + 1)]
    lr.fit(xx, y)
    variances.append(explained_variance_score(y, lr.predict(X[:, :(i + 1)])))

plt.plot(variances, 'go-')
plt.xlabel('Number of features')
plt.ylabel('Explained Variance')
plt.grid()
```

On executing the code, you will get the plot as shown below.In this diagram, the explained variance gradually increases when you increase the features considered one by one.



At step number 5, there is a drastic increase in the explained variance. Now, if you check the Boston Housing data set, you will observe that the feature 'RM' is responsible for this increase in the explained variance.

One major observation is that as you add to the list of features, the R2 score increases. In order to maintain a balance between **keeping the model simple** and **explaining the highest variance** (which means that you would want to keep as many variables as possible), you need to penalise a model for keeping a large number of predictor variables.

Hence, you should measure the following two evaluation metrics in this scenario:

$$Adjusted\ R^2 = 1 - \frac{(1-R^2)(N-1)}{N-p-1}$$

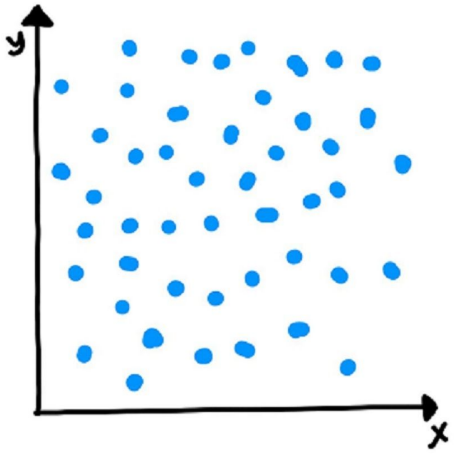
$$AIC = n \times log(\frac{RSS}{n}) + 2p$$

In the second formula, n is the sample size, which represents the number of rows that you would have in the data set, and p is the number of predictor variables. Unfortunately, the scikit-learn library does not provide a direct implementation to these metrics; however, the statsmodel package provides these metrics as part of summary statistics.

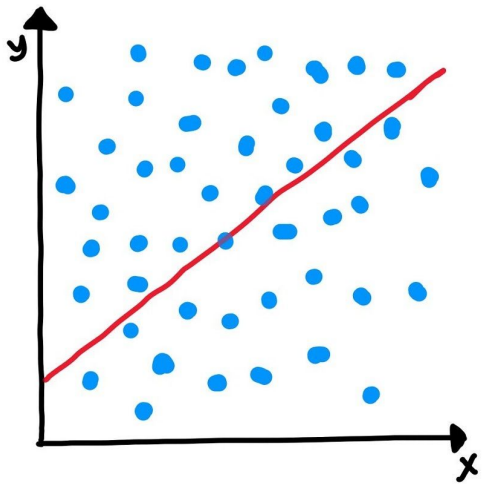
Hypothesis Testing in Linear Regression

In this segment, you learnt how to perform hypothesis testing on the estimated values of the coefficients, i.e., the betas. This, in turn, determines which independent variables are significant, so you can use them in your model.

Suppose you have a data set whose scatter plot looks like this:



Now, if you run a linear regression on this data set, then you will be able to fit a line on the data, which, say, looks like this:



You always fit a line through the data by applying linear regression using the least-square method. However, as you can see, the fitted line in the diagram given above is of no use in this case. Hence, hypothesis testing can be used to test whether or not the fitted line is a significant one.

Consider the simple model y = β1x + β0. In order to conduct hypothesis testing, you need to propose the null hypothesis that β1 is 0. The alternative hypothesis, thus, becomes β1 is not 0. The hypotheses are are as follows:

- Null Hypothesis (H0): β1 = 0
- Alternate Hypothesis (HA): β1 ≠ 0

Now, if you reject the null hypothesis, then it would mean that β1 is not 0 and the fitted line is significant.

The steps to conduct the hypothesis test are as follows:

- You need to compute the t-score (which is similar to the Z-score), which is given by:

  $$\frac{(X-\mu)}{(s/\sqrt{n})}$$

  where μ is the population mean and s is the sample standard deviation, which, when divided by √n, is also known as the standard error.
- The t-score for $\hat{\beta}_1$ would be (since the null hypothesis is that β1 is equal to 0) as follows:

  $$\frac{\hat{\beta}_1 - 0}{SE(\hat{\beta}_1)}$$

  with (n - 2) degrees of freedom.

Considering a significance level of 0.05, if the p-value turns out to be less than 0.05, then you can reject the null hypothesis and state that β1 is indeed significant.

After you import the **statsmodel.api**, you can create a simple linear regression model in just a few steps as shown below.

```
import statsmodels.api as sm
X_C= sm.add_constant(X)
model = sm.OLS(y, X_C)
result = model.fit()
result.summary()
```

Remember to use the command 'add_constant' so that statsmodels also fits an intercept. If you do not use this command, then it will fit a line passing through the origin by default.

Summary Statistics

Let's understand summary statistics with the help of the table given below.

OLS Regression Results

Dep. Variable:	y	R-squared:	0.741
Model:	OLS	Adj. R-squared:	0.734
Method:	Least Squares	F-statistic:	108.1
Date:	Sat, 23 May 2020	Prob (F-statistic):	6.95e-135
Time:	14:18:28	Log-Likelihood:	-1498.8
No. Observations:	506	AIC:	3026.
Df Residuals:	492	BIC:	3085.
Df Model:	13		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	36.4911	5.104	7.149	0.000	26.462	46.520
x1	-0.1072	0.033	-3.276	0.001	-0.171	-0.043
x2	0.0464	0.014	3.380	0.001	0.019	0.073

F-statistic

The heuristic for F-statistic is similar to what you learnt in the normal p-value calculation. If **'Prob (F-statistic)'** is less than **0.05**, then you can conclude that the overall model fit is significant. If it is greater than 0.05, then you will need to review your model, as the fit might have occurred by chance, i.e., the line may have fit the data by luck. In the image given above, you can see that the p-value of the F-statistic is

1.52e-52, which is practically a zero value. This means that the model for which this was calculated is definitely significant since the F-statistic for it is less than 0.05.

R-squared

In the previous diagram, the R-squared value is about 0.741, indicating the model is able to explain 74% of the variance, which is quite good.

Coefficients and p-Values

The image given below shows the warnings that appear along with the summary statistics.

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 1.51e+04. This might indicate that there are
strong multicollinearity or other numerical problems.

The second point in the image given above highlights the existence of multicollinearity. Multicollinearity occurs when the input data set contains predictor (independent) variables that are related to one another.

Variance Inflation Factor

The two ways to deal with multicollinearity. They are as follows:

1. Looking at **pairwise correlations**
    - You can look at the correlation between different pairs of independent variables.
2. Checking the **variance inflation factor** (VIF)
    - Sometimes pairwise correlations are not enough.
    - Instead of just one variable, the independent variable may be dependent on a combination of other variables.
    - VIF is used to calculate how well one independent variable is explained by all the other independent variables combined.

VIF is given by:

$$\text{VIF} = \frac{1}{1-R_i^2}$$

Here, *'i'* refers to the i-th variable, which is represented as a linear combination of the rest of the independent variables, and R_i^2 is the R2 score of the model when the linear regression model is fit for the variable i against the other independent variables.

The common heuristic that we follow for VIF values is as follows:
> 10: The VIF value is definitely high, and the variable should be eliminated.
> 5: The VIF value is satisfactory, but it is worth inspecting.

< 5: The VIF value is good, and there is no need to eliminate this variable.

Using the VIF value, you can identify features that best define the model and remove features that are redundant in the data set. The VIF can be calculated using variance_inflation_factor function of the statsmodels library.

To find an optimal model, you can try all possible combinations of the independent variables and check which model fits best. However, this method is time-consuming and infeasible. Therefore, you need another method to find a decent model. This is where the manual feature elimination method comes into the picture, wherein you:

- Build the model with all the features;
- Drop the features that are the least helpful in making predictions (high p-value);
- Drop the features that are redundant (multicollinearity); and
- Rebuild the model and repeat the process.

Note that you should never drop all the insignificant variables and the variables with high VIFs in a single step. You must do it sequentially, as there is interdependence among the variables and you would not want to drop an important variable on the basis of the initial result only.

Feature Selection: Variable Thresholding

As per the general norm, you use automated approaches such as RFE (recursive feature elimination) or regularisation. When you are down to a few features, you can start looking at the p-values and the VIF, and proceed with a manual improvement by considering the business requirements and other validity checks.

In this segment, you learnt about the most basic method of feature selection: variance thresholding. Variance thresholding involves removing all the features that show extremely low variance, or, in other words, columns whose values remain more or less the same for the data points in the data set. You should be careful while using a thresholding value, as it may eliminate important features in the model.

Consider the features and their individual variances given in the image below.

```
CRIM          73.904671
ZN           543.936814
INDUS         47.064442
CHAS           0.064513
NOX            0.013428
RM             0.493671
AGE          792.358399
DIS            4.434015
RAD           75.816366
TAX        28404.759488
PTRATIO        4.686989
B           8334.752263
LSTAT         50.994760
price         84.586724
dtype: float64
```

You need to scale your data using a min-max scalar to bring all the features to a similar scale, and then apply variance thresholding. This will help you make a better comparison between all the features involved in your experiment.

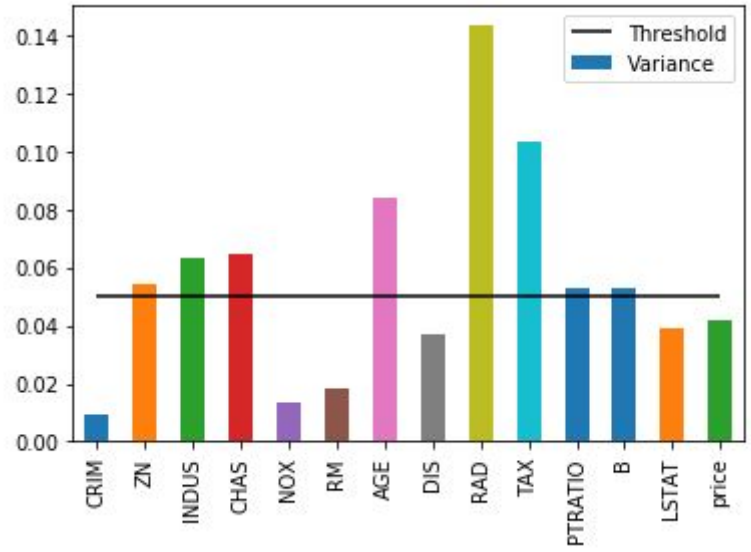
The code to apply MinMaxScalar and plotting individual variances is as follows:

```
#import the MinMaxScalar
from sklearn.preprocessing import MinMaxScaler
scalar = MinMaxScaler()

# Apply scaler() to all the columns except the 'yes-no' and 'dummy' variables
num_vars = ['CRIM', 'ZN', 'INDUS', 'RM', 'AGE', 'DIS', 'RAD', 'TAX','PTRATIO', 'B', 'LSTAT','price']
df[num_vars] = scalar.fit_transform(df[num_vars])

#plot of individual variance after scaling
ax = df.var(0).plot(kind='bar', label='Variance')
ax.hlines(0.05,0,13 ,label='Threshold')
plt.legend()
```

As you can observe in the image given below, the variances in NOX and RM are not as insignificant as shown in the earlier plot. This is because you have scaled your data and brought all the features into a comparable range. It is essential to do this when your data involves features of different scales.



Variable Thresholding can directly be applied using VarianceThreshold function of sklearn.

Feature Selection: selectKbest

In this segment, you learnt about the selectKbest feature estimator of the sklearn library, which is used for feature selection. selectKbest is used for finding the correlation between a set of input features and the output feature, and based on this correlation, it chooses all the features that show the highest correlation.

Note that we use f-regression as a parameter while using selectKbest for feature selection in a regression problem. You can read more about f_regression here.

You can directly use the SelectKBest function of the sklearn library

Feature Selection: RFE

Recursive feature elimination (RFE) is one of the most widely used approaches for feature selection. It is a backward stepwise approach, wherein all the features are considered for model building, and after each iteration, identifies the least significant feature that can be dropped.

You can use the RFE function from the sklearn library to apply this technique.

Summary

Linear Regression Using PySpark

In this session, you first explored the Spark ML library API and then learnt how to perform basic exploratory data analysis on a data set. You also learnt about certain components such as feature transformers and feature estimator pipelines, and their usage while writing your code in PySpark. Finally, you learnt how to build regression and classification models.

Scalability in Linear Regression

In the previous course, you learned that in the case of linear regression, the entire algorithm comes down to a simple matrix representation of β ,where

$$\hat{\beta} = (X^T X)^{-1} X^T Y$$

You can see that the matrix given above is composed of the following four operations:

1. Multiplication X.T and X,
2. Inverse of X.T * X,
3. Multiplication X.T and Y, and
4. Multiplication of inv(X.T * X) and (X.T * Y).

All the operations above are multiplications, which can be easily parallelized in Spark, and, hence, most of the machine learning algorithms are parallelizable in Spark, as they are usually a series of matrix operations.

upGrad

MLlib Overview

The first step in data preparation is to interact with Spark and read the data into a DataFrame. You can do this using the read() method, and by mentioning the file type and file path in Spark.

The read() method is available through the SparkSession class and it also supports various optional methods for indicating the header and schema. By setting the header to 'true', Spark treats the first row in the DataFrame as the header and the rest of the rows as samples. Similarly, by setting 'inferschema' to 'true', Spark automatically infers the schema of the data set.

The code for creating a Spark session and reading data from a csv file is given below.

```
# Creating Spark Session

from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('MLlib_Overview').getOrCreate()

#Reading Data from a CSV file
#Inferring Schema and Setting Header as True

df = spark.read.csv('auto-miles-per-gallon-Raw.csv', header=True, inferSchema=False)
```

Imputer

As the next step in data cleaning, you can either remove records containing incomplete or garbage values, or replace missing values with approximate values. Generally, the median or the mean of the complete column variable serves as a good approximate value. Removing records often leads to loss of some valuable information and so, you may want to impute those values, instead. In this segment, you learnt about the following two methods for handling missing values:

1. In the first method, you remove records with missing values by using the na() method available in Spark. This method drops all the rows that may contain a missing value.

```
df2 = df1.dropna()
```

2. In the second method, you can replace missing values with the mean of their respective features using the Imputer() transformer that is present in the Spark ML library. It is an extension of the transformer class.

```
from pyspark.ml.feature import Imputer
imputer = Imputer(inputCols = ["MPG","HORSEPOWER"], outputCols =
```

```
["MPG-Out","HORSEPOWER-Out"])
imputeModel = imputer.fit(df1)
df3=imputeModel.transform(df1)
```

Feature Transformer: Vector Assembler

A feature transformer transforms the data stored in a data frame and stores the data back as a new data frame. This transformation generally takes place by appending one or more columns to the existing data frame. It can be broken down into the following sequence: DataFrame = [transform] = > DataFrame.

The VectorAssembler() transformer is a feature transformer that takes a set of individual column features as input and returns a vector that contains all the column features. It is an extension of the transformer class and supports the .transform() method.

It is a common practice to scale all the data variables within the range [0, 1]. You can perform scaling using transformers, such as MaxAbsScaler(), MinMaxScaler(), etc.

To perform scaling, you need to create a scalar object followed by a scalar model. This model will then transform any input DataFrame into a scaled DataFrame.

The code snippet for using the VectorAssembler function is as follows:

```
from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import VectorAssembler


assembler = VectorAssembler(
   inputCols=[ ‘’’ INPUT FEATURES ’’’  ],
   outputCol=‘’’ TARGET VARIABLE ’’’)

output = assembler.transform(df)
```

Pipeline

Instead of executing a transform method at each data preparation step, a pipeline clubs all the steps of data, such as cleansing, scaling, normalising, etc., in the desired sequence. By creating a pipeline, you can skip multiple steps and make your code more efficient. Also, once you have designed a pipeline with all the required steps, it can be reused for various data sets without the need to severely alter the nature of the code for each data set.

A pipeline can be built by declaring the ‘Pipeline’ object available in the Spark ML library. Furthermore, you need to build a PipelineModel by fitting the pipeline object on the data. Unlike the steps involved in the

previous segments, the PipelineModel will take only one data frame as input and output the final prepared DataFrame.

```
#import Pipeline and setting the stages of the pipeline
from pyspark.ml import Pipeline
pipeline =  Pipeline(stages = [imputer,assembler,scaler])
model = pipeline.fit(data)
Final_output = model.transform(data)
```

In short, a Pipeline is specified as a sequence of stages, and each stage is either a Transformer or an Estimator. These stages are run in a specific order, and the input dataFrame is transformed as it passes through each stage. As you have learnt previously, no intermediate dataframe is created at any stage.

Additional Reading

1. Pipelines Spark Documentation
2. The Elegance of the Spark ML Pipeline

Regression Using Spark MLlib

Machine learning models require input in the form of a vector. If this criterion is not satisfied, then you will get an error saying 'A linear regression object expects an input features column'.

Using VectorAssembler, a feature transformer, you can assemble the features in the form of a vector as shown below.

```
assembler = VectorAssembler(inputCols=[c for c in sdf.columns if c != 'price'], outputCol='features')
dataset = assembler.transform(sdf)
```

After assembling the features to form a new features column, you can build a linear regression model by specifying the two columns: featuresCol and labelCol as shown in the code snippet below.

```
lr = LinearRegression(featuresCol='features', labelCol='price')
model = lr.fit(dataset)
```

You can find the r2 score associated with the model by using model.evaluate(). You will find that the model gives the same r2 score as given by the statsmodel api and scikit-learn library.

There are many classification algorithms, such as logistic regression and Naive Bayes classifiers. For building a simple logistic regression model, the first step involves loading the necessary data sets into a new data frame. Then, VectorAssembler is used to create a feature column out of all the features excluding the predictor column. Next, besides defining the input and output features, which you do in regression, you also need to convert the output column, which is categorical, into numerical features. You can do so using StringIndexer, a feature transformer that is available in the Spark MLlib library.

StringIndexer Feature Transformer

You can convert a column of string values in your data frame to a column of numeric values using the **StringIndexer** transformer. It assigns index values into numerical values based on their corresponding string frequencies. for example, consider your input data has the following strings:

String
High
Low
High
High
Low
Medium

You can use StringIndexer to convert these strings into indices. You can refer to the table given below to understand this conversion.

String	Frequency	IndexedString
High	3	0
Low	2	1
High	3	0
High	3	0
Low	2	1
Medium	1	2

Here, as an intermediary step, the frequency of each string value is calculated and the highest frequency is assigned an index value of 0. This way, a string value is converted to a numerical value.

Finally, by using the .transform() method, you can transform the input data frame into a new indexed column. Now, once you have applied the StringIndexer feature transformer to the output column, you can now build a logistic regression model on the top of it by specifying the input, feature column, and the output, label column.

In order to evaluate a model, you can perform a train–test split and check how the model performs on unseen data. Using the .evaluate() method on the test data set, you can print the summary statistics of the model.

Note: The implementation of Naive Bayes classifier is the same as compared to the logistic regression classifier; the only difference being the use of MulticlassClassificationEvaluator.

Linear Regression Using Spark MLlib

You can perform the following steps to fit a linear regression model:

Step 1: Create a Spark DataFrame and prepare the data set

```
# Step 1
diabetes = datasets.load_diabetes()

df = None
assembler = None
dataset = None
```

Step 2: Find the R2 score and explained variance

```
# Step 2
lr = None
model = None
summary = None
print(summary.r2, summary.explainedVariance)
```

Step 3: Find a subset of features with the highest absolute coefficients (by plotting)

Step 4: Train a new model on this subset, and find the R2 score and explained variance

```
subset = None
assembler = None
small_dataset = None
```

```
lr = None
model = None
summary = None
print(summary.r2, summary.explainedVariance)
```

Cross-Validation

In cross-validation, you begin by splitting the input data set into training and testing data sets. This is done to understand the performance of your model on an unseen data set. The heuristic is that you do either a **70–30%** or an **80–20%** split of the input data set into training and test data sets. Once you have the training data sets, you can then build models of the training datasets and check their performance on the testing dataset.

Cross-validation helps you identify the issues that are present in your model or in the data set, such as overfitting. Setting the seed value will reduce the random behaviour; however, upon splitting the data set may not have so much variability that the training R2 score is greater than the test R2 score and vice versa. This might be possible because the data set is ill-conditioned, meaning it has no underlying pattern that can be modelled.

The code snippet for applying cross validation on Spark is as follows.

```
train, test = dataset.randomSplit([0.5, 0.5], 25)
lr = LinearRegression(featuresCol='features', labelCol='y')
model = lr.fit(train)

# Evaluate on training data
summary_train = model.evaluate(train)
print('R2 (training): ', summary_train.r2)

# Evaluate on training data
summary_test = model.evaluate(test)
print('R2 (testing): ', summary_test.r2)
```

Bias-Variance Trade-Off

As you have learnt, any data set that you come across in real life will have some noise. Also, any model that you create is to model the true underlying function. Now, assuming that the true function that fits the data set is denoted by f(x) and the model that you built is denoted by f′(x), the relationship between the response values, Y (which are, essentially, the data points that are available for modelling), and the predictor variables, X, is given by:

$$Y = f(X) + \epsilon,$$

where ϵ is the irreducible error term or noise that cannot be modelled. Note that the values belonging to the irreducible error will be distributed normally with mean 0, since it is noise and cannot be modelled.

Hence, the mean square error (MSE) of an unknown data set is the expected value of

$$(Y - f'(x))^2$$

Then, MSE can be decomposed as shown below:

$$MSE = (E[f'(x)] - E[f(x)])^2 + (E[f'(x)^2] - E[f'(x)]^2) + \sigma_\epsilon^2$$

The first term is the bias of the model and the second term is the variance of f'(x) that the model used for prediction, while the third term is the variance of the irreducible error. Thus we can write the MSE as follows:

$$MSE = Bias(f'(x))^2 + Variance(f'(x)) + \sigma_\epsilon^2$$

or,

$$MSE = Bias^2 + Variance + Irreducible\ error$$

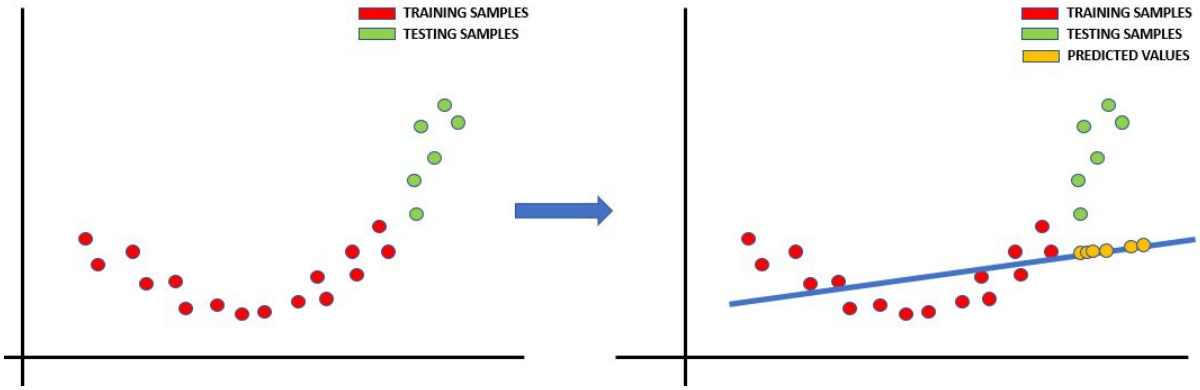
As you can see in the equations given above, for the same MSE, if the bias decreases, the variance increases, and vice versa. Hence, there is a trade-off between bias and variance in the process of building a machine learning model.

Bias

A bias error is the difference between the predicted value and the true value of the data. It is important to understand that the predicted value depends on the assumptions made by the model. Hence, if the model makes a large number of assumptions about the data (like it does in the case of linear regression), then the bias error will be high, and vice versa.

Note that due to the presence of the irreducible error, the true value may differ from the actual value available to you. However, for the purpose of evaluation of your model, you can consider the training error as a representative of the bias error. Hence, if the training error is high, then there is high bias, and vice versa.

Bias quantifies how accurate the model is likely to be on the future/test data. Extremely simple models are likely to fail in predicting complex real-world phenomena. Hence, simplicity has its own disadvantages in machine learning. Refer to the model given below as an example.



Since the model given above is found to poorly fit the training data, such a model is said to be **underfitting**.

High bias suggests that the model has been built by making many assumptions, thus simplifying and making the algorithm less complex. Linear regression is an example of a high-bias algorithm.

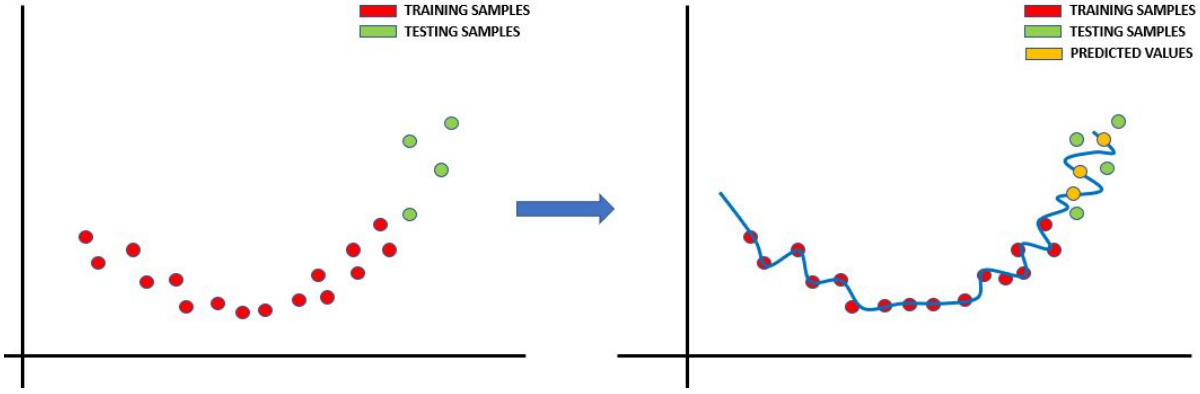
Variance

A variance error is an error that is generated when you pay too much attention to the training data.

The 'variance' of a model is the variance in its output on the same test data with respect to the changes to the training data. In other words, variance refers to the degree of changes to the model itself with respect to the changes in the training data.

For example, consider the model (given below) that memorises the entire training data set. Even if you made a small change to the data set, the model would change drastically. The model is, therefore, unstable and is sensitive to the changes to the training data, and this is called high variance. Also, this has happened because the model has also modelled the irreducible error, which cannot be modelled.

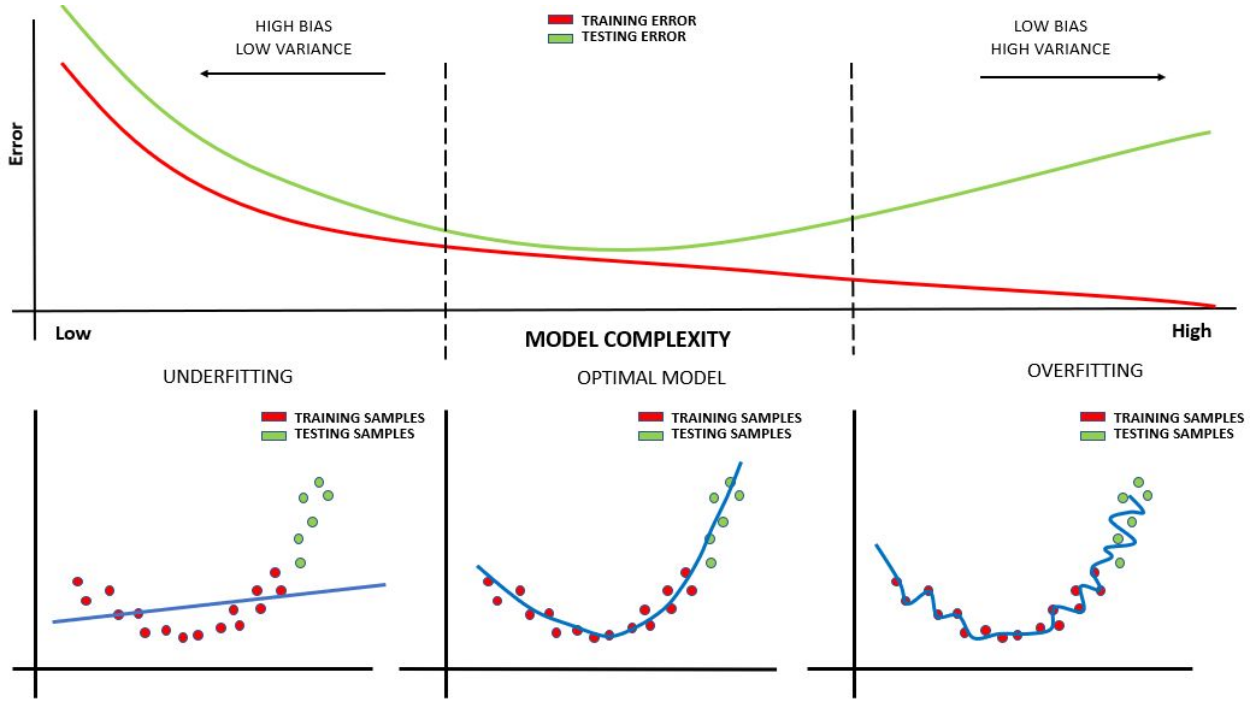
As you can see, the model has been fitted perfectly on the training data; however, it poorly fits the test data. Hence, the testing error is often considered a representation of the model variance.



Since this model is found to capture the unwanted noise within the training data, and is overly sensitive to it, such a model is said to be **overfitting**. Overfitting is a phenomenon where a model becomes too specific to the data on which it is trained, and it fails to generalise to other unseen data points in the larger domain. An example of a high-variance algorithm is decision trees.

Bias-Variance Trade-Off

In an ideal case, you would want to reduce both bias and variance, because the expected total error of a model is the sum of the errors in bias and variance. In practice, you cannot have a model that has low bias as well as low variance. With increase in the model's complexity, the bias decreases but the variance increases, resulting in a trade-off. You can refer to the following model to understand this better.



Summary

Linear Regression: Case Study

In this case study, you learnt how to solve a practical problem using the concepts that you have learnt so far. For this, you used the data set of New York City taxi fares. The objective of this case study is to build a **pricing model** wherein the fare for a particular ride is predicted based on a set of attributes, such as date-time, and the coordinates for pickup and drop-off.

1. **Data Exploration**

   The first part of this case study was data exploration, wherein you performed the following steps:

   - First, you invoked the Spark session context and read the data from s3.
   - Then you imported the VectorAssembler and LinearRegression functions from the MLlib library. Next, you created a linear regression model by providing the input and the output features.
   - On fitting the model on the data set, you created a summary instance using the evaluate() function.

- Next, you computed the r2 value.

Note: At this point, the r2 value works out to be 0.0002, meaning there is very little correlation between the target variable and the independent features. Hence, you need to explore and transform the data to get a better fit.

- On exploring the feature 'pickup_latitude', you found several inconsistencies, such as negative values and non-viable values. Hence, it is important to remove such values and outliers before fitting the model.

2. Outlier Treatment

You identified many outliers in the data set that might have led to a low r2 score, as linear regression is quite sensitive to outliers. Here, you performed the following steps:

- You performed a simple Google search and found that New York City lies between 73 and 75 degrees West, and 40 and 42 degrees North. Using this information, you filtered out the values in the pickup and drop off locations that lie outside the given coordinates of latitude and longitude.
- On exploring the data further, you found that certain rides had zero passengers. Since these rides must not be considered for building the pricing model, you filtered them out and considered only those rides that had at least one passenger.
- You also found that the pricing data consisted of some rows with negative fare values. Hence, you removed these rows and considered only those rows that had a fare value greater than 0.
- Once all the data was cleaned, you saved this new data frame and fit the linear regression model.
- You computed the new r2 value as 0.25. However, this value is still less, and the data set needs to be explored further.

3. Feature Engineering

Here, you extracted some additional features from the data set to improve the r2 score of the model. These features are as follows:

- The date-time feature is modified by removing the 'UTC' substring and converting it to the timestamp data type for further analysis.
- An interesting point to note is that all the timestamps correspond to the UTC timestamps, which differ from the New York timestamps by 5 hours. Hence, you used the Spark API to convert the timestamp to the EST time and saved it in a new column.
- Using the pyspark.sql.functions, you extracted the year, month, day and hour, and added these columns to the data frame.
- The abs function is imported from the PySpark API to compute the l1 norm distance between two latitudes or two longitudes. Using this api, the horizontal and vertical distances are computed from the coordinates of the pickup and drop-off locations.
- Using the 'l1' column as an additional feature, you fit the regression model and calculated the r2 score to be 0.679.

4. **Model Validation**

In this part, you split the data set into a training and a test data set. You trained the model on the training data and evaluated it on the test data. The steps that you performed are as follows:

- You began by creating a train–test split of the data set. For this, you used the randomSplit function and performed a 2:1 split. Then you stored these data frames in the ‘trainDataset‘ and ‘testDataset‘ data frames.
- Using the features that you created after cleaning and modifying the data, you fit the training data using the LinearRegression library.
- You computed the summary by using the model .evaluate function on the test data set.
- You computed the r2 value as 0.681, which is similar to the r2 value that was computed on the entire data set. This indicates that the model was not overfitting the train data.

upGrad