



Course name: **Python programming and analytics by Rahul Sir**

Topic name: **logistic regression (telecom case study) Manav batch**

Video name: **telecom case study - logistic regression**

Video length: **1 hour 19 minutes 42 seconds**

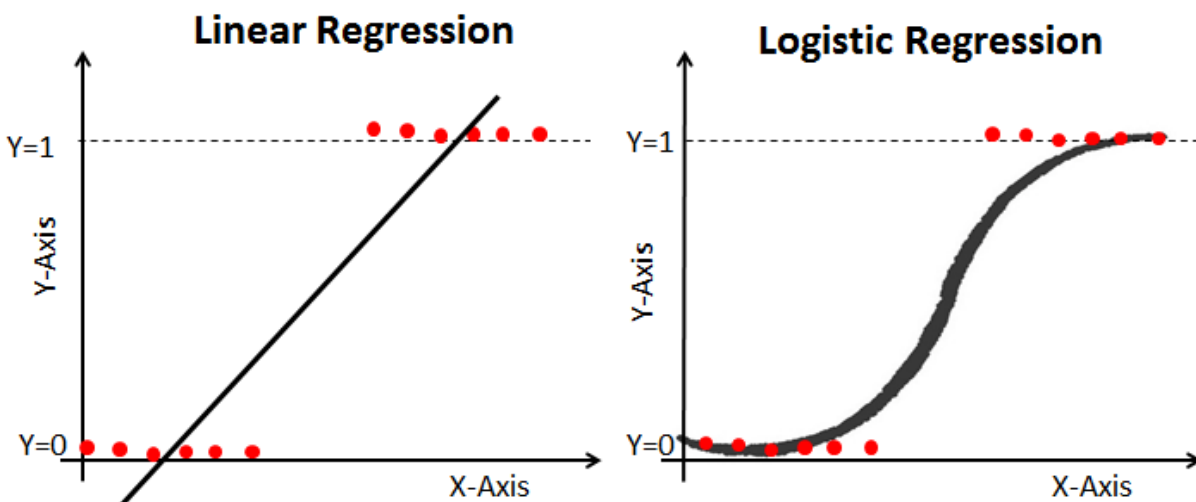
[Logistic Regression](#) is a Machine Learning classification algorithm that is used to predict the probability of a categorical dependent variable. In logistic regression, the dependent variable is a binary variable that contains data coded as 1 (yes, success, etc.) or 0 (no, failure, etc.). In other words, the logistic regression model predicts $P(Y=1)$ as a function of X . Despite the word **Regression** in Logistic Regression, Logistic Regression is a supervised machine learning algorithm used in binary classification. I say **binary** because one of the limitations of Logistic Regression is the fact that it can only categorize data with **two** distinct classes. At a high level, Logistic Regression fits a line to a dataset and then returns the probability that a new sample belongs to one of the two classes according to its location with respect to the line. Suppose we wanted to build a Logistic Regression model to predict whether a student would pass or fail given certain variables such as the number of hours studied. To be exact, we want a model that outputs the probability (a number between 0 and 1) that a student will pass. A value of 1 implies that the student is guaranteed to pass whereas a value of 0 implies that the student will fail.

Logistic Regression can be used for various classification problems such as spam detection. Diabetes prediction, if a given customer will purchase a particular product or will they churn another competitor,

whether the user will click on a given advertisement link or not, and many more examples are in the bucket.

Linear Regression Vs. Logistic Regression

Linear regression gives you a continuous output, but logistic regression provides a constant output. An example of the continuous output is house price and stock price. Example of the discrete output is predicting whether a patient has cancer or not, predicting whether the customer will churn.



Confusion matrix

A confusion matrix is a table that is used to evaluate the performance of a classification model













In the field of machine learning and specifically the problem of statistical classification, a confusion matrix, also known as an error matrix.

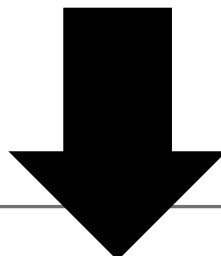
A confusion matrix is a table that is often used to describe the performance of a classification model (or "classifier") on a set of test

data for which the true values are known. It allows the visualization of the performance of an algorithm.

It allows easy identification of confusion between classes e.g. one class is commonly mislabeled as the other. Most performance measures are computed from the confusion matrix.

- Positive (P): Observation is positive (for example: is an apple).
- Negative (N): Observation is not positive (for example: is not an apple).
- True Positive (TP): Observation is positive, and is predicted to be positive.
- False Negative (FN): Observation is positive, but is predicted negative.
- True Negative (TN): Observation is negative, and is predicted to be negative.
- False Positive (FP): Observation is negative, but is predicted positive.

	Predicted class POSITIVE (spam )	Predicted class NEGATIVE (normal )
Actual class POSITIVE (spam )	TRUE POSITIVE (TP)   <div>320</div>	FALSE NEGATIVE (FN)   <div>43</div>
Actual class NEGATIVE (normal )	FALSE POSITIVE (FP)   <div>20</div>	TRUE NEGATIVE (TN)   <div>538</div>





True Positive:

Interpretation: You predicted positive and it's true.

You predicted that it is a spam mail and it actually is.

True Negative:

Interpretation: You predicted negative and it's true.

You predicted that the mail is normal and it actually is; or you predicted that the mail is not spam and it actually is not.

False Positive: (Type 1 Error)

Interpretation: You predicted positive and it's false.

You predicted that the mail is spam but it actually is normal.

False Negative: (Type 2 Error)

Interpretation: You predicted negative and it's false.

You predicted that the mail is normal but it actually is spam.

Recall: it answers the question:

“When it is actually the positive result, how often does it predict correctly?”

Recall can be defined as the ratio of the total number of correctly classified positive examples divide to the total number of positive examples. High Recall indicates the class is correctly recognized (small number of FN).

Recall is given by the relation:

$$\text{Recall} = \frac{TP}{TP + FN}$$

Recall is usually used when the goal is to *limit the number of false negatives* (FN). In our example, that would correspond to minimizing the number of spam emails that are classified as real emails. Recall is also known as “sensitivity” and “true positive rate” (TPR).

- **Precision:** it answers the question:

“When it predicts the positive result, how often is it correct?”

To get the value of precision we divide the total number of correctly classified positive examples by the total number of predicted positive examples. High Precision indicates an example labeled as positive is indeed positive (small number of FP).



Precision is given by the relation:

$$\text{Precision} = \frac{TP}{TP + FP}$$

Precision is usually used when the goal is to *limit the number of false positives* (FP). For example, this would be the metric to focus on if our goal with the spam filtering algorithm is to minimize the number of real emails that are classified as spam.

High recall, low precision: This means that most of the positive examples are correctly recognized (low FN) but there are a lot of false positives.

Low recall, high precision: This shows that we miss a lot of positive examples (high FN) but those we predict as positive are indeed positive (low FP)

F-measure:

Since we have two measures (Precision and Recall) it helps to have a measurement that represents both of them. We calculate an F-measure which uses Harmonic Mean in place of Arithmetic Mean as it punishes the extreme values more.

The F-Measure will always be nearer to the smaller value of Precision or Recall.

$$F - \text{measure} = \frac{2 * \text{Recall} * \text{Precision}}{\text{Recall} + \text{Precision}}$$

It considers both the Precision and Recall of the procedure to compute the score.

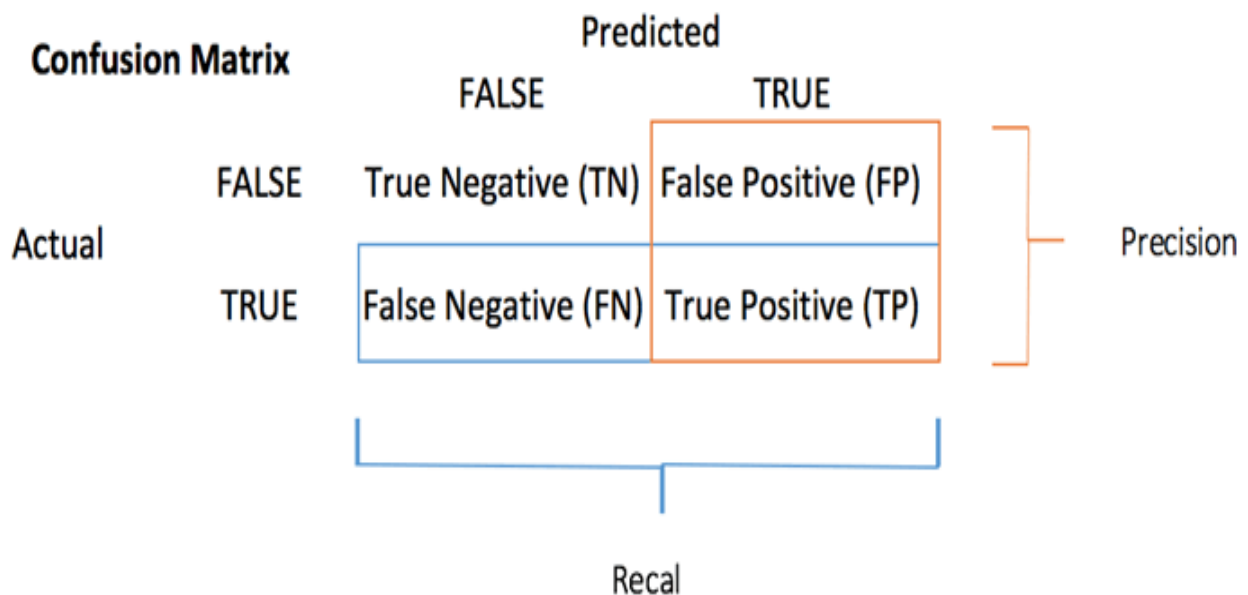
Higher the F-score, the better will be the predictive power of the classification procedure.

A score of 1 means the classification procedure is perfect. Lowest possible F-score is 0.

Classification Rate or Accuracy is given by the relation:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

The confusion matrix visualizes the accuracy of a classifier by comparing the actual and predicted classes. The binary confusion matrix is composed of squares:





First, we need to load the packages as well as the data

```
In [28]: import pandas as pd
```

```
In [29]: data = pd.read_csv("C:/Users/Sahibjot/Desktop/Data_set_telecom_churn.csv")
```

```
In [30]: data.describe()
```

Out[30]:

	Account Length	VMail Message	Day Mins	Eve Mins	Night Mins	Intl Mins	CustServ Calls	Int'l Plan	VMail Plan	Day Calls	Day Charge	Ev
count	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000
mean	101.064806	8.099010	179.775098	200.980348	200.872037	10.237294	1.562856	0.096910	0.276628	100.435644	30.562307	100.435644
std	39.822106	13.688365	54.467389	50.713844	50.573847	2.791840	1.315491	0.295879	0.447398	20.069084	9.259435	19.999999
min	1.000000	0.000000	0.000000	0.000000	23.200000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	74.000000	0.000000	143.700000	166.600000	167.000000	8.500000	1.000000	0.000000	0.000000	87.000000	24.430000	87.000000
50%	101.000000	0.000000	179.400000	201.400000	201.200000	10.300000	1.000000	0.000000	0.000000	101.000000	30.500000	100.435644
75%	127.000000	20.000000	216.400000	235.300000	235.300000	12.100000	2.000000	0.000000	1.000000	114.000000	36.790000	114.000000
max	243.000000	51.000000	350.800000	363.700000	395.000000	20.000000	9.000000	1.000000	1.000000	165.000000	59.640000	170.000000

to see some of the core statistics about a particular column, you can use the ['describe'](#) function.

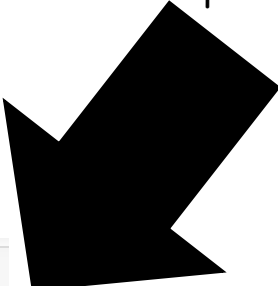
- For numeric columns, [describe\(\)](#) returns [basic statistics](#): the value count, mean, standard deviation, minimum, maximum, and 25th, 50th, and 75th quantiles for the data in a column.
- For string columns, [describe\(\)](#) returns the value count, the number of unique entries, the most frequently occurring value ('top'), and the number of times the top value occurs ('freq')

Select a column to describe using a string inside the [] braces, and call `describe()`

Many DataFrames have mixed data types, that is, some columns are numbers, some are strings, and some are dates etc. Internally, CSV files do not contain information on what data types are contained in each column; all of the data is just characters. Pandas infers the data


types when loading the data, e.g. if a column contains only numbers, pandas will set that column's data type to numeric: integer or float.

You can check the types of each column in our example with the `'dtypes'` property of the dataframe.



```
In [31]: data.dtypes
Out[31]: Phone                object
Account Length              int64
VMail Message               int64
Day Mins                    float64
Eve Mins                    float64
Night Mins                   float64
Intl Mins                    float64
CustServ Calls              int64
Int'l Plan                  int64
VMail Plan                  int64
Day Calls                   int64
Day Charge                  float64
Eve Calls                   int64
Eve Charge                  float64
Night Calls                 int64
Night Charge                float64
Intl Calls                  int64
Intl Charge                 float64
State                       object
AreaCode                    int64
Churn                       int64
dtype: object
```

In the next step, we are going to drop or remove the "phone" column using `.drop()` function because we don't need the phone column in our data, this is not necessary if you want to keep the column, you can keep it. It is just a preprocessing step.





```
In [32]: data1 = data.drop(["Phone"], axis=1)
```

```
In [33]: data1.head()
```

Out[33]:

	Account Length	VMail Message	Day Mins	Eve Mins	Night Mins	Intl Mins	CustServ Calls	Int'l Plan	VMail Plan	Day Calls	Day Charge	Eve Calls	Eve Charge	Night Calls	Night Charge	Intl Calls	Intl Charge	State	AreaCode	Churn
0	128	25	265.1	197.4	244.7	10.0	1	0	1	110	45.07	99	16.78	91	11.01	3	2.70	KS	415	0
1	107	26	161.6	195.5	254.4	13.7	1	0	1	123	27.47	103	16.62	103	11.45	3	3.70	OH	415	0
2	137	0	243.4	121.2	162.6	12.2	0	0	0	114	41.38	110	10.30	104	7.32	5	3.29	NJ	415	0
3	84	0	299.4	61.9	196.9	6.6	2	1	0	71	50.90	88	5.26	89	8.86	7	1.78	OH	408	0
4	75	0	166.7	148.3	186.9	10.1	3	1	0	113	28.34	122	12.61	121	8.41	3	2.73	OK	415	0

To delete rows and columns from DataFrames, Pandas uses the "[drop](#)" function.

To delete a column, or multiple columns, use the name of the column(s), and specify the "axis" as 1. The drop function returns a new Data Frame, with the columns removed.

You can see in output33 that the phone column is removed from the data.

Now, as you can see the area code column is denoted with integer data type, but the area code can't be an integer, it must be a character or string as you can't take the average of area code, so we need to change this integer column into string column, as follows:

```
In [34]: data1['AreaCode'] = data1.AreaCode.astype(str)
```

In some cases, the automated inferring of data types can give unexpected results. Note that strings are loaded as 'object' datatypes.

To change the datatype of a specific column, use the [astype\(\) function](#)

In statistics, especially in regression models, we deal with various kind of data. The data may be quantitative (numerical) or qualitative (categorical). The numerical data can be easily handled in regression



models but we can't use categorical data directly, it needs to be transformed in some way.

For transforming categorical attribute to numerical attribute, we can use label encoding procedure (label encoding assigns a unique integer to each category of data). But this procedure is not alone that much suitable, hence, **One hot encoding** is used in regression models following label encoding. This enables us to create new attributes according to the number of classes present in the categorical attribute i.e if there are n number of categories in categorical attribute, n new attributes will be created. These attributes created are called **Dummy Variables**. Hence, dummy variables are "proxy" variables for categorical data in regression models.

These dummy variables will be created with *one hot encoding* and each attribute will have value either 0 or 1, representing presence or absence of that attribute.



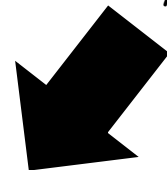
```
In [35]: features = pd.get_dummies(data1)
```

```
In [36]: features
```

```
Out[36]:
```

	Account Length	VMail Message	Day Mins	Eve Mins	Night Mins	Intl Mins	CustServ Calls	Int'l Plan	VMail Plan	Day Calls	...	State_UT	State_VA	State_VT	State_WA	State_WI	State_WV	State_
0	128	25	265.1	197.4	244.7	10.0	1	0	1	110	...	0	0	0	0	0	0	
1	107	26	161.6	195.5	254.4	13.7	1	0	1	123	...	0	0	0	0	0	0	
2	137	0	243.4	121.2	162.6	12.2	0	0	0	114	...	0	0	0	0	0	0	
3	84	0	299.4	61.9	196.9	6.6	2	1	0	71	...	0	0	0	0	0	0	
4	75	0	166.7	148.3	186.9	10.1	3	1	0	113	...	0	0	0	0	0	0	
5	118	0	223.4	220.6	203.9	6.3	0	1	0	98	...	0	0	0	0	0	0	
6	121	24	218.2	348.5	212.6	7.5	3	0	1	88	...	0	0	0	0	0	0	
7	147	0	157.0	103.1	211.8	7.1	0	1	0	79	...	0	0	0	0	0	0	
8	117	0	184.5	351.6	215.8	8.7	1	0	0	97	...	0	0	0	0	0	0	
9	141	37	258.6	222.0	326.4	11.2	0	1	1	84	...	0	0	0	0	0	0	1
10	65	0	129.1	228.5	208.8	12.7	4	0	0	137	...	0	0	0	0	0	0	

After creating dummy variables, now if you want to see that how many columns are there in your data and also the column names:





```
In [37]: a = features.columns.values.tolist()
```

```
In [38]: print(a)
```

```
['Account Length', 'VMail Message', 'Day Mins', 'Eve Mins', 'Night Mins', 'Intl Mins', 'CustServ Calls', 'Int'l Plan', 'VMail Plan', 'Day Calls', 'Day Charge', 'Eve Calls', 'Eve Charge', 'Night Calls', 'Night Charge', 'Intl Calls', 'Intl Charge', 'Churn', 'State_AK', 'State_AL', 'State_AR', 'State_AZ', 'State_CA', 'State_CO', 'State_CT', 'State_DC', 'State_DE', 'State_FL', 'State_GA', 'State_HI', 'State_IA', 'State_ID', 'State_IL', 'State_IN', 'State_KS', 'State_KY', 'State_LA', 'State_MA', 'State_MD', 'State_ME', 'State_MI', 'State_MN', 'State_MO', 'State_MS', 'State_MT', 'State_NC', 'State_ND', 'State_NE', 'State_NH', 'State_NJ', 'State_NM', 'State_NV', 'State_NY', 'State_OH', 'State_OK', 'State_OR', 'State_PA', 'State_RI', 'State_SC', 'State_SD', 'State_TN', 'State_TX', 'State_UT', 'State_VA', 'State_VT', 'State_WA', 'State_WI', 'State_WV', 'State_WY', 'AreaCode_408', 'AreaCode_415', 'AreaCode_510']
```

```
In [39]: len(a)
```

```
Out[39]: 72
```

```
In [40]: features.head()
```

```
Out[40]:
```

	Account Length	VMail Message	Day Mins	Eve Mins	Night Mins	Intl Mins	CustServ Calls	Int'l Plan	VMail Plan	Day Calls	...	State_UT	State_VA	State_VT	State_WA	State_WI	State_WV	State_WY
0	128	25	265.1	197.4	244.7	10.0	1	0	1	110	...	0	0	0	0	0	0	0
1	107	26	161.6	195.5	254.4	13.7	1	0	1	123	...	0	0	0	0	0	0	0
2	137	0	243.4	121.2	162.6	12.2	0	0	0	114	...	0	0	0	0	0	0	0
3	84	0	299.4	61.9	196.9	6.6	2	1	0	71	...	0	0	0	0	0	0	0
4	75	0	166.7	148.3	186.9	10.1	3	1	0	113	...	0	0	0	0	0	0	0

5 rows x 72 columns

Now, the next step is defining X and Y variables:

```
In [48]: cols = ['Account Length', 'VMail Message', 'Day Mins', 'Eve Mins', 'Night Mins', 'Intl Mins', 'CustServ Calls', 'Int'l Plan', 'VMail Plan', 'Day Calls', 'Day Charge', 'Eve Calls', 'Eve Charge', 'Night Calls', 'Night Charge', 'Intl Calls', 'Intl Charge', 'Churn']
```

```
In [49]: x=features[cols]
        y=features['Churn']
```

```
In [50]: x.shape
```

```
Out[50]: (3333, 71)
```

If you want to know how many rows and columns are there in the variables.

Now, the next step is to split the data:

To understand model performance, dividing the dataset into a training set and a test set is a good strategy. Let's split dataset by using function `train_test_split()`.

Machine learning is about learning some properties of a data set and then testing those properties against another data set. A common practice in machine learning is to evaluate an algorithm by splitting a data set into two. We call one of those sets the **training set**, on which we learn some properties; we call the other set the **testing set**, on which we test the learned properties.

```
In [57]: import sklearn
         from sklearn.linear_model import LogisticRegression
         from sklearn import metrics
         from sklearn.model_selection import train_test_split

In [58]: #LR model

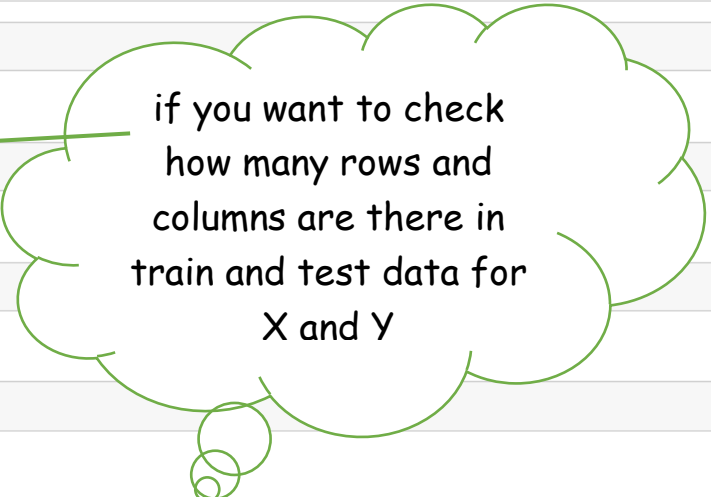
         X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.3, random_state=0)

In [59]: X_train.shape
Out[59]: (2333, 71)

In [60]: X_test.shape
Out[60]: (1000, 71)

In [76]: y_train.shape
Out[76]: (2333,)

In [77]: y_test.shape
Out[77]: (1000,)
```



if you want to check
how many rows and
columns are there in
train and test data for
X and Y

You can't possibly manually split the dataset into two. And you also have to make sure you split the data in a random manner. To help us with this task, the SciKit library provides a tool, called the Model Selection library. There's a class in the library which is, aptly, named 'train_test_split.' Using this we can easily split the dataset into the training and the testing datasets in various proportions.

- **test_size** — This parameter decides the size of the data that has to be split as the test dataset. This is given as a fraction. For example, if you pass 0.2 as the value, the dataset will be split 20% as the test dataset.
- **train_size** — You have to specify this parameter only if you're not specifying the test_size. This is the same as test_size, but instead you tell the class what percent of the dataset you want to split as the training set.
- **random_state** — Here you pass an integer, which will act as the seed for the random number generator during the split.

After this fit your model on the train set using fit()

```
In [61]: LRC = LogisticRegression()  
         LRC.fit(X_train, y_train)
```

```
Out[61]: LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,  
                             intercept_scaling=1, max_iter=100, multi_class='warn',  
                             n_jobs=None, penalty='l2', random_state=None, solver='warn',  
                             tol=0.0001, verbose=0, warm_start=False)
```



Now, perform prediction on the test set using `predict()`

```
In [71]: Preds= LRC.predict(X_test)
```

```
In [72]: Preds
```

[illegible]