upGrad

Summary

Introduction to Logistic Regression

In this session, you learnt about the classification problem, and an algorithm that is used to solve such problems: logistic regression. The session covered the mathematics behind building as well as optimising the algorithm to predict the class of a data point as well as optimising the algorithm. And finally, you learnt about the metrics that can be used to measure the performance of classification algorithms.

The Classification Function

Unlike regression, the objective of a classification problem is to predict the class to which a particular data point belongs. The output of a classification problem is the class of the object; it can be binary, for instance, whether or not a customer is about to leave a network, or multinomial, for instance, predicting the blood group of a person.

When the given data is simple to visualise, and there is an obvious separation between the data points, you can use a step function to classify the data points. But as the complexity of the data increases, along with an increase in the number of features on which the class label depends, a simple step function is not enough to classify the data points. In such cases, the sigmoid function is used to classify the data. This sigmoid function [represented by σ(x)] is expressed as follows:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

The following properties make the sigmoid function ideal for use as a classification function:

1. It is bound between **0** and **1**.
   Regardless of the value of x, the output of y will lie between 1 and 0. This makes it ideal for predicting probabilities.

2. The function is **asymptotic**, that is, it can never take a value of 0 or 1.
   The function can be 0 only if the numerator is 0, which is not possible, and it can be 1 only when e^(-x) = 0, which is also not possible. This adheres to the probabilistic approach, which states that no data point can be classified with 100% confidence. The probability of a data point belonging to a particular class is **high**.

3. The **sigmoid function** is a good classifier, like the step function.
   The sigmoid function resembles the step function, that is, the slope of the curve is low close to the ends, whereas it is high in the middle. So, the sigmoid function will have high separation for the

data points that fall in the middle region.

4. It is easy to set a **threshold** for the sigmoid function.
   At x = 0, the output of the sigmoid function equals 0.5. If 0.5 is set as the threshold, then the data point is classified as 1 if the value of x is positive, and it is classified as 0 if the value of x is negative.

To use the sigmoid function as a classification function in logistic regression, the input to the sigmoid function is a combination of features instead of an independent variable x. In this case, the output of the sigmoid function represents the probability of a data point belonging to a particular class. Substituting the combination of features in the sigmoid function gives the following probability:

$$P = \sigma(x_i) = \frac{1}{1 + e^{-(w_1 x_{1i} + w_2 x_{2i} .. w_n x_{ni})}}$$

Odds and Log Odds

It is quite difficult to develop an intuition about the change in the output of the sigmoid function based on the change in its input. This is because the actual combination of features is raised to the power of e. To make interpretation of the logistic equation easier, the concepts of odds and log odds are introduced.

Odds is the ratio between the probability of an event occurring and the probability of that event not occurring. It can be described by the equation given below:

$$odds = \frac{p}{1-p}$$

Here, P represents the probability of the event occurring. Simply taking the log of the odds ratio gives the log odds; this is described by the equation given below:

$$log(odds) = log(\frac{p}{1-p})$$

As the probability of an event increases, its odds ratio increases as well, along with its log odds. Unlike probabilities, odds are not bound between 0 and 1. They can take any value between 0 and + ∞. You can verify this by substituting the minimum value of probability (0) and the maximum value of probability (1) in the formula for odds ratio. And since odds are not bound between 0 and 1, log odds can take any value between - ∞ and + ∞. So, you can equate the log odds to the combination of features. Interpreting these features using log odds makes it easy to interpret the change in the features.

In logistic regression, the probability of a class depends on the features and their weights. Hence, as you can see in the example above, for a vanilla logistic regression, where the threshold is 0.5, when a data point

is called a positive class, it means the sum of the weighted features [∑(Weights * Features + Bias)] of the data point is positive, and the data point will be classified as true.

Normalisation: 1NF and 2NF

In the case of continuous variables, such as children's heights, the distribution is normal. And to define a normal distribution, you need the following two parameters of the distribution: mean (μ) and standard deviation (σ). Similarly, for a random variable that has two outcomes, we define the distribution by the **Bernoulli distribution**.

Any experiment or question whose outcome or answer is a Boolean value, such as a success or a failure, or 0 or 1, follows a Bernoulli distribution. For instance, what are the possible outcomes of an experiment where you toss a fair coin? Heads and tails. So, this is an example of a Bernoulli process, and the outcome will follow a Bernoulli distribution. To define this distribution completely, only one parameter is enough: the bias of the experiment.

The probability of getting a specific output from an experiment with discrete outputs is called the probability mass function (PMF) or the discrete density function. In the above example of the fair coin toss experiment, the PMF of getting a head from a toss is a Bernoulli distribution that is described by the following equation:

$$f(p:k) = p^k.(1-p)^{(1-k)}$$, where k can be either 0 or 1.

The PMF obtained above can help you calculate the probability of a single trial resulting in a specific outcome. However, for more than one trial, you need to use the binomial distribution. A binomial experiment is a series of Bernoulli trials, with each trial producing a binary output. The probability of getting a positive class in each trial should remain the same. And the results of all the trials are independent of each other. So, in other words, the Bernoulli distribution represents the success or the failure of a single Bernoulli trial. Binomial distribution represents the number of successes and failures in *n independent* Bernoulli trials for some given value of n.

Consider a binomial experiment with n trials and k successes. If each of the trials had a probability p of producing a favourable outcome, then the overall probability would be given by the following equation:

$$\binom{n}{k} p^k.(1-p)^{(n-k)}$$,

where $\binom{n}{k}$ represents k items picked from n identical items, or ${}^{n}C_{k}$.

Maximum Likelihood Function

Now you have a means to relate the features of a data set to the probability of a data point belonging to a particular class. You can do this using the sigmoid function. The Bernoulli distribution models the distribution of variables that have only two outputs. Moreover, the Bernoulli distribution can be defined completely by a single parameter. So, the task now is to make sure the Bernoulli distribution can model the distribution of the class labels. This distribution fitting is achieved using the maximum likelihood approach.

A likelihood function is a statistical tool that is used to measure the **goodness of fit** of a statistical model. The likelihood function gives the probability of a random guess being equal to the actual data point in the distribution. It can be described by the following equation:

$$P\left(y=y_i \right) \forall\, i \in \left[1, n\right],$$

where y is the random guess and y_i is the actual data point. If you maximise this probability, then the actual distribution will 'fit' the actual data.

You should remember that y here is a vector of all the data points. So, upon expanding the above equation, you will get:

$$P(y) = \prod_{i=0}^{n} p(y = y_i),$$

where the $p(y = y_i)$ is given as:

$$p(y = y_i) = p_i^{y_i} \times (1 - p_i)^{(1-y_i)}$$

By substituting this in the previous equation, you will get:

$$P(y) = \prod_{i=0}^{n} p_i^{y_i} \times (1 - p_i)^{(1-y_i)}$$

The probability P(y) needs to be maximised to make sure the random guess is equal to the actual class label. But before maximising P(y), the equation above can be simplified a lot by taking log, as shown below:

$$log(P(y)) = \sum_{i=0}^{n} log(p_i^{y_i} \times (1 - p_i)^{(1-y_i)}) = \sum_{i=0}^{n} log(p_i^{y_i}) + \sum_{i=0}^{n} log(1 - p_i)^{(1-y_i)}$$

This gives the log-likelihood function, which is shown below:

$$P(y:p) = \sum_{i=0}^{n} y_i log(p_i) + (1-y_i) log(1-p_i)$$

Note that we have used P(y:p) instead of P(y) in the above expression for the log-likelihood function. P(y:p) is read as the log-likelihood of the variable y given the parameter p. This process is also known as the **maximum likelihood estimation** (MLE).

In machine learning, it is a good practice to minimise a function instead of maximising it. So, you can simply flip the sign of the log-likelihood function, as shown below:

$$\hat{L}(\boldsymbol{y}; p) = -\frac{1}{n}\sum_{i=1}^{n} [y_i \log(p_i) + (1-y_i)\log(1-p_i)]$$

This is the log loss function, which represents the average loss in predicting the class labels. This needs to be minimised.

Understanding Gradient Descent Optimisation

In gradient descent optimisation, the parameter being optimised is iterated in the direction of reducing the cost/loss according to the following rule:

$$W_{new} = W_{old} - \eta . \frac{\partial L}{\partial W}.$$

Let's try to understand this by understanding what gradient descent is. Gradient descent is an optimisation algorithm that is used to find the minimum of a function. Here, the basic idea is to use the gradient of the function to find **the direction of steepest descent**, i.e., the direction in which the value of the function decreases the most rapidly, and move towards the minima iteratively.

Let's take an example of a one-dimensional (univariate) function. Say, you have a loss function, L, that depends on only one variable, w. The minimum of this function is at w = 0.

The algorithm starts with an initial arbitrary guess of w, computes the gradient at that point, and then updates w iteratively according to the following rule:

$$w_{new} = w_{old} - \eta . \frac{\partial L}{\partial w}$$

For example, let's take an arbitrary initial value of $w_0 = 5$. The gradient is 2w and so, the value of the gradient (the slope) at $w_0 = 5$ is $2 * 5 = 10$, as represented by the blue line in the figure given above. The gradient has the following **two critical pieces of information**:

- The **sign of the gradient** (positive here) indicates the 'direction' in which the function's value increases, and, thus, a negative sign points to the **direction of decrease**.
- The **value of the gradient** (10 here) represents how steeply the function's value increases or decreases at that point.

The step size for moving towards the solution is determined by the learning rate, η. If η is large, then you might possibly overshoot the minima. Conversely, if it is small, then you might never reach the minima. Therefore, it is important that you select the right η.

Gradient descent can be easily extended to multivariate functions, i.e., functions that depend on multiple variables. Let's take the bivariate function $L(w_1,w_2) = w_1^2+w_2^2$. The minimum value of this function is 0 at the point $(w_1,w_2) = (0,0)$. For convenience, let's represent the two variables as $W = (w_1,w_2)$ together. The iterative procedure is the same: start with an arbitrary initial guess of $W_0 = (w_1,w_2)_0$ and proceed in the direction of decreasing function value. The only change is that the **gradient** $\frac{\partial L}{\partial W}$ **is now a vector**:

$$\frac{\partial L}{\partial W} = \begin{bmatrix} \frac{\partial L}{\partial w_1} \\ \frac{\partial L}{\partial w_2} \end{bmatrix} = \begin{bmatrix} 2w_1 \\ 2w_2 \end{bmatrix}$$

You can now extend this idea to any number of variables. For instance, say, your machine learning model has k weights. You can represent these weights with a large column vector as shown below:

$$\begin{bmatrix} w_1 \\ w_2 \\ \cdots \\ w_k \end{bmatrix}$$

The gradient vector will also be a k-dimensional vector, and each of its elements will capture the following two pieces of information, the direction and the rate of change of the function with respect to the weight w_i, as shown below:

$$\frac{\partial L}{\partial w} = \begin{bmatrix} \frac{\partial L}{\partial w_1} \\ \frac{\partial L}{\partial w_2} \\ \cdots \\ \cdots \\ \frac{\partial L}{\partial w_k} \end{bmatrix}$$

upGrad

Minimising the Log Loss Function

To minimise the log loss, you will need the derivative of the sigmoid. Hence, it would be helpful if you can find the derivative of the sigmoid (denoted by σ in the video) beforehand, as you will just need to substitute it while minimising the log loss function, thus avoiding a lot of complications.

The sigmoid function is given by:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

Now, let the denominator of the sigmoid function be g(x); $g(x) = 1 + e^{-x}$.

Therefore,

$$\sigma(x) = \frac{1}{g(x)}$$

Now, differentiating σ with respect to x, you will get:

$$\frac{d\sigma}{dx} = \frac{d\sigma}{dg} \times \frac{dg}{dx} = -\frac{1}{g(x)^2} \times (-e^{-x}) = \frac{e^{-x}}{g^2(x)} = \frac{e^{-x}}{(1+e^{-x})^2}$$

To simplify this fraction even further, add and subtract 1 to and from the numerator, respectively:

$$\frac{d\sigma}{dx} = \frac{1+e^{-x}-1}{(1+e^{-x})^2}$$

$$\frac{d\sigma}{dx} = \frac{1+e^{-x}}{(1+e^{-x})^2} - \frac{1}{(1+e^{-x})^2} = \frac{1}{(1+e^{-x})} - \frac{1}{(1+e^{-x})^2}$$

$$\frac{d\sigma}{dx} = \sigma(x) - \sigma(x)^2$$

Therefore, the derivative of the sigmoid function is given by:

$$\frac{d\sigma}{dx} = \sigma(x)(1-\sigma(x))$$

So, now that you have the derivative of sigmoid, we'll next move on to finding out the derivative of the loss/cost function for logistic regression. If you substitute the sigmoid of the combination of the features, along with their weights [∑(Weights * Features + Bias)] into the loss function, then you will get a loss function in terms of the actual labels (y_i) and the predicted labels [$\sigma(x_i) = \sigma_i$ for simplicity], as shown below:

$$\hat{L}(y;p) = -\frac{1}{n}\sum_{i=0}^{n}[y_i log(\sigma_i) + (1-y_i)log(1-\sigma_i)]$$

,

where $\sigma_i = \sigma(x_i)$ represents the sigmoid of the i^{th} data point.

To simplify, let's consider the loss of i^{th} data point:

$$l_i(y_i;p) = -[y_i log(p_i) + (1-y_i)log(1-p_i)]$$

In this equation, p_i is the probability of the datapoint belonging to the positive class, and the sigmoid function gives the following equation:

$$p_i = \sigma_i = \sigma(w^T x_i) = \frac{1}{1+e^{-w^T x_i}}$$

Here, **w** is a column vector that represents the weights to be assigned to the feature vector **x**. Both **w** and **x** are columnar vectors. Hence,

$$w^T x_i = \begin{bmatrix} w_1 & w_2 & . & . & w_k \end{bmatrix} . \begin{bmatrix} x_{i1} \\ x_{i2} \\ . \\ . \\ x_{ik} \end{bmatrix} = w_1 x_{1i} + w_2 x_{2i} . . . . + w_k x_{ki}.$$

Now, substitute the value of the sigmoid function in the log loss function, as shown below:

$$l_i(y_i;w,x_i) = -[y_i log(\sigma_i) + (1-y_i)log(1-\sigma_i)]$$

We are differentiating this equation with respect to w to get $\nabla_w l_i$, as we intend to perform gradient descent optimisation on the weights. Note that as yi and xi are, respectively, the label and the features of the data point, and are already given, their differentials with respect to w will be 0. Therefore, you get the following equation:

$$\begin{aligned}\nabla_w l_i &= -\frac{\partial}{\partial w}[y_i log(\sigma_i) + (1-y_i) log(1-\sigma_i)] \\ &= -[y_i \frac{\partial}{\partial w} log(\sigma_i) + (1-y_i) \frac{\partial}{\partial w} log(1-\sigma_i)] \\ &= -[y_i \frac{1}{\sigma_i} \frac{\partial \sigma_i}{\partial w} + \frac{(1-y_i)}{(1-\sigma_i)} \frac{\partial}{\partial w}(1-\sigma_i)] \\ &= -\left[\frac{y_i}{\sigma_i} \frac{\partial \sigma_i}{\partial w} - \frac{(1-y_i)}{(1-\sigma_i)} \frac{\partial \sigma_i}{\partial w}\right] \\ &= -\frac{\partial \sigma_i}{\partial w}\left[\frac{y_i}{\sigma_i} - \frac{1-y_i}{1-\sigma_i}\right]\end{aligned}$$

Now, before moving further, let's find out the value of $\frac{\partial \sigma_i}{\partial w}$. As discussed earlier, the value of $\frac{\partial \sigma_i}{\partial w}$ is:

$$\frac{\partial \sigma_i}{\partial w} = \frac{\partial}{\partial w}\sigma(w^T x_i) = \frac{\partial \sigma(w^T x_i)}{\partial (w^T x_i)} \frac{\partial}{\partial w}(w^T x_i)$$

The value of the derivative of the sigmoid function has been calculated earlier. Substituting this value in the above equation, you will get:

$$\frac{\partial \sigma_i}{\partial w} = \sigma_i(1-\sigma_i) \times \frac{\partial}{\partial w}(w^T x_i)$$

Now, it is a bit critical to determine the value of the term $\frac{\partial}{\partial w}(w^T x_i)$. As discussed earlier, $w^T x_i = w_1 x_{1i} + w_2 x_{2i} + ... + w_n x_{ni}$, and you want to differentiate this with respect to the weight vector. You can perform this differentiation element-wise, as shown below:

$$\frac{\partial}{\partial w}(w^T x_i) = \begin{bmatrix} \frac{\partial (w_1 x_{1i} + w_2 x_{2i} .... + w_n x_{ni})}{\partial w_1} \\ \frac{\partial (w_1 x_{1i} + w_2 x_{2i} .... + w_n x_{ni})}{\partial w_2} \\ . \\ . \\ \frac{\partial (w_1 x_{1i} + w_2 x_{2i} .... + w_n x_{ni})}{\partial w_n} \end{bmatrix} = \begin{bmatrix} x_{i1} \\ x_{i2} \\ . \\ . \\ x_{ik} \end{bmatrix} = x_i$$

Note that all the weights are independent of each other and, hence, we get the above result. And the above equation can be simplified to:

$$\frac{\partial}{\partial w}(w^T x_i) = x_i.$$

Substituting this in the original gradient equation for the i[th] data point, you will get:

$$\nabla_w l_i = -\sigma_i(1-\sigma_i)x_i\left[\frac{y_i}{\sigma_i} - \frac{1-y_i}{1-\sigma_i}\right]$$

Now, let's solve the second half of the equation, which is the part within brackets:

$$\frac{y_i}{\sigma_i} - \frac{1-y_i}{1-\sigma_i} = \frac{y_i(1-\sigma_i)-\sigma_i(1-y_i)}{\sigma_i(1-\sigma_i)} = \frac{y_i - y_i\sigma_i - \sigma_i + \sigma_i y_i}{\sigma_i(1-\sigma_i)} = \frac{y_i-\sigma_i}{\sigma_i(1-\sigma_i)}$$

Substituting this in the original gradient equation, you will get:

$$\nabla_w l_i = -\sigma_i(1-\sigma_i)x_i\frac{y_i-\sigma_i}{\sigma_i(1-\sigma_i)}$$

Therefore,

$$\nabla_w l_i = x_i(\sigma_i - y_i)$$

Taking the average over all the data points, you will get:

$$\nabla_w \hat{L}(y;p) = -\frac{1}{n}\sum_{i=0}^{n}\frac{\partial l_i}{\partial w} = \frac{1}{n}\sum_{i=0}^{n} x_i(\sigma_i - y_i)$$

Gradient Descent Optimisation for Log Loss

The gradient of loss for all the points is given by:

$$\nabla_w \hat{L}(y;p) = -\frac{1}{n}\sum_{i=0}^{n}\frac{\partial l_i}{\partial w} = \frac{1}{n}\sum_{i=0}^{n} x_i(\sigma_i - y_i)$$

Here are the steps that you need to follow while minimising the equation above using gradient descent:

1. Initialise the weights with some random values.
2. Calculate the sigmoid of ${w_t^T x}_i$ using these values of w for all the data points:
   $$\sigma\left(w_t^T x_i\right) = \frac{1}{1+e^{-\left(w_t^T x_i\right)}}$$
3. Use this to calculate the log loss as shown below:
   $$\hat{L}(y;p) = -\frac{1}{n}\sum_{i=0}^{n}[y_i log(\sigma_i) + (1-y_i)log(1-\sigma_i)]$$

You should monitor that this log loss keeps reducing with every iteration.

4. Plug in the value of the sigmoid equation in the gradient function to calculate the derivative of the loss of the i^{th} data point:

$$\nabla_{w_t} l_i = x_i(\sigma(w_t^T x_i) - y_i)$$

5. Also calculate the gradient of the log loss, as shown below:

$$\nabla_{w_t} \hat{L}(y; p) = -\frac{1}{n} \sum_{i=0}^{n} \nabla_{w_t} l_i.$$

6. Using the values calculated above, update the weights according to the following rule:

$$w_{t+1} = w_t - \eta \nabla_{w_t} l_i$$

7. If you calculate the updated weights using the derivative of the log loss of a single data point, then it is known as **stochastic gradient descent**.

8. If you update the weights using the average log loss of all the data points, then a lot of time would be spent calculating the loss for all the data points and then updating the weights. Represented in the equation below:

$$w_{t+1} = w_t - \eta \nabla_{w_t} \hat{L}$$

This is known as **batch-gradient descent**.

9. Check whether there is a significant difference between the new weights and the weights of the earlier generation. If they don't differ significantly, then proceed to the next step; else, go back to Step 2 with the newly updated weights.
10. Use these converged weights to predict the class label for unseen data. Once you have the converged weights, you can predict the class label for unseen data points, as shown below:

$$p_i = \frac{1}{1 = e^{-(w_1 x_{1i} + w_2 x_{2i})}}$$

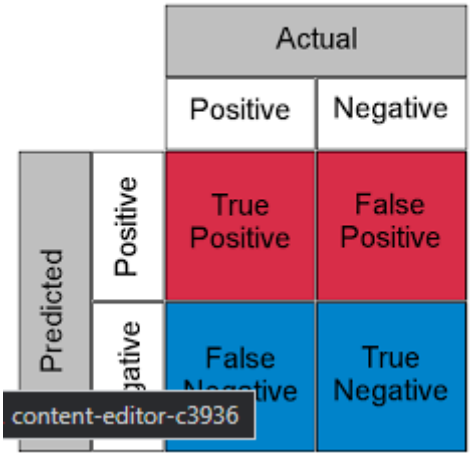
This is the process of finding the weights that will give the best prediction of class label for unseen data points.

Evaluation Metrics for Classification Problems

The metrics used for classification problems are different from the metrics used for regression problems, because in classification, you measure the number of predictions that you got right, instead of measuring the degree of wrongness. So, the most obvious evaluation metric is the ratio of the number of correct predictions and the total number of predictions. This metric is called accuracy. It is expressed as shown in the equation given below:

$$accuracy = \frac{correct\ guess}{total\ guess}$$

To measure the performance of classification problems, we use quite a handy tool called confusion matrix.



Each row of the confusion matrix corresponds to the predicted values, whereas each column corresponds to the actual values. The figure given above shows the confusion matrix for a binary class classification problem. A similar matrix can be developed for multiclass problems as well. The difference between the two matrices would be that the number of rows and columns would increase.

The terms inside the matrix represent the following conditions:

True positives (TP): Data points that actually belong to the positive class and were predicted to be positive.

True negatives (TN): Data points that actually belong to the negative class and were predicted to be negative.

False positives (FP): Data points that actually belong to the negative class but were predicted positive.

False negatives (FN): Data points that actually belong to the positive class but were labeled as negative.

The confusion matrix can be used to develop several other matrics, which are listed below:

1. **Sensitivity**
   It is the fraction of correctly identified positives among all the positive labels in a data set. It is exactly the same as recall. It is expressed as follows:

   Sensitivity = (TP)/(TP + FN)

2. **Prevalence**
   It is the fraction of positive classes in the complete data set. For instance, if there are 7 positive

classes in a data set of 10, then the prevalence is 0.7. It is expressed as follows:

Prevalence = (TP + FN)/(TP + TN + FP + FN)

3. **Specificity**
   It is the fraction of correctly predicted negative classes among all the negative classes of a data set. It is expressed as follows:

   Specificity = (TN)/(TN + FP)

4. **True accuracy**
   True accuracy = Sensitivity × Prevalence + Specificity × Probability of negative classes

   This is better than accuracy but still you can not use this alone.

5. **Precision**
   This gives the fraction of correctly predicted positives among all the data points that were predicted to be positive. For instance, if 10 data points were predicted to be positive, and of these, only 9 are actually positive, then the precision is 0.9. It is expressed as follows:

   Precision= TP/(TP + FP)

6. **Recall**
   It is the fraction of correctly predicted positives among all the actual positive data points. Continuing from the earlier example, if 18 data points were actually positive, then the recall would be 9/18 or 0.5. Recall is expressed as follows:

   Recall = TP/(TP + FN)

   It is the same as the sensitivity of a model.

In most real-world cases, there will be a trade-off between precision and recall. So, you need to find a balance between the two. F1 score can help you do that.

F1 score is the harmonic mean of precision and recall. It is expressed as follows:

$$F1 = \left(\frac{(Recall)^{-1} + (Precision)^{-1}}{2} \right) = 2 \times \frac{Recall \times Precision}{Recall + Precision}$$

Here are some properties of F1 score:

Observations

1. The value of F1 score always lies between precision and recall. This means F1, like precision and recall, is also bound between 0 and 1.
2. A high F1 score for a model means that it has both high precision and high recall, and, hence, it is good at identifying positive classes.
3. F1 score falls if either precision or recall is low.

So, a model with a high F1 score identifies most of the positive data points, and it does so with accuracy.

Receiver Operating Characteristic

All the metrics discussed earlier have used a standard threshold value. However, changing the threshold values also leads to many changes to the classification.

The receiver operating characteristic (ROC) curve does exactly the same. It varies the threshold from 0 to 1 and plots the true positive rate versus the false positive rate for each threshold.

True positive rate (TPR) is the same as sensitivity and recall, and it measures the proportion of correctly identified actual positives. It is expressed as follows:

TPR = TP/(TP + FN)

In contrast, **false positive rate** (FPR) measures the fraction of incorrectly identified positive cases among all the actual negative cases; this is nothing but (1 - Specificity). It is expressed as follows:

FPR = FP/(FP + TN)

Summary

Logistic Regression - Model Building with Python

upGrad

In this session, you learnt how to prepare the data and then use it to build a logistic regression model. You built the model using the sklearn and stats model libraries. This session focused on the preprocessing steps and feature selection.

Multivariate Logistic Regression - Telecom Churn Example

Telecom Churn Problem Statement

You have a telecom firm that has collected data pertaining to all of its customers. The main types of attributes in this data set include the following:

1. Demographics (age, gender, etc.)
2. Services availed (internet packs purchased, special offers availed, etc.)
3. Expenses (amount of recharge done per month)

Based on all of this past information, you want to build a model that will predict whether or not a particular customer will churn, i.e., whether or not they will switch to a different service provider. So, the variable of interest, i.e., the target variable, here is 'Churn', which will tell you whether or not or not a particular customer has churned. It is a binary variable: 1 means the customer has churned and 0 means the customer has not churned. You can apply logistic regression to try and solve this problem.

Data Cleaning and Preparation - I

The data given is divided into three separate files, each having a column named 'customer ID'. These files can be joined using customer id as the primary key to create a data frame with all the tables combined. As you can see, the data frame contains some binary columns: 'PhoneService', 'PaperlessBilling', 'Churn', 'Partner' and 'Dependents' are represented by yes or no. You need to convert them to 0 and 1, and to do that, you can use the following piece of code:

```
# List of variables to map

varlist =  ['PhoneService', 'PaperlessBilling', 'Churn', 'Partner',
'Dependents']

# Defining the map function
def binary_map(x):
    return x.map({'Yes': 1, "No": 0})

# Applying the function to the housing list
telecom[varlist] = telecom[varlist].apply(binary_map)
```

After converting the aforementioned columns to 'integer' form, you need to convert all the other columns as well. If you notice, most of the columns in the data frame are categorical and so, one hot encoding is used to convert them to integer form. Pandas library provides a simple function to one hot encode the categorical columns. Here is the command.

```
# Creating a dummy variable for some of the categorical variables and
dropping the first one.
dummy1 = pd.get_dummies(telecom[['Contract', 'PaymentMethod', 'gender',
'InternetService']], drop_first=True)

# Adding the results to the master dataframe
telecom = pd.concat([telecom, dummy1], axis=1)
```

As you can see from the code given above, after one hot encoding, one column from the encoded columns is dropped, and this is done to reduce the computational effort. Also, in doing so, no information is lost. The row with all zeros obviously represents the column that was dropped. After dropping the one column from the encoded columns, the rest of the columns for which we had created dummy columns are also dropped because the information present in them is not in a simpler format.

Similarly, we handle all the categorical columns. One last column that needs special attention is the 'TotalCharges' column. Although it is a continuous column, it is represented as a categorical column. So, the values in this column need to be converted to numerical values, and you can do this using the pd.to_neumeric function.

After all the columns have been converted to numerical form, you can check them for the presence of any redundancy in the information that they hold. For instance, for the 'MultipleLines' column, you dropped the level 'MultipleLines_No phone service' manually instead of simply using 'drop_first = True', which would have dropped the first level present in the 'MultipleLines' column. The reason we did this is that if you check the column 'MultipleLines', then you will see that this column has the following three levels:

```
No                  3390
Yes                 2971
No phone service     682
Name: MultipleLines, dtype: int64
```

Now, out of these levels, it is best that you drop 'No phone service' since it is of no consequence, as it is anyway being indicated by the variable 'PhoneService' that is already present in the data frame.

The next step is to deal with empty cells, if any. There are a few methods to deal with missing values. For instance, you can drop all the rows that have missing values. However, this can introduce bias in the data set. You can fill in the missing data. The missing data can be filled in (also called imputed) using different methods, such as mean or median. You can also leave the missing values as is. Some algorithms can work with missing data, but this is not a recommended practice.

In the given data frame, only 'TotalCharges' has 11 missing values. Now, since this is not a big number compared with the number of rows present in the data set, we decided to drop the missing values as we would not lose much data. The next step in data cleaning is to check for outliers. Since there are no outliers in the telecom churn data set, we do not need to worry about them here. As discussed already, you never test a model on the same data set on which it is trained. So, it is important to split the data into a training set and a test set. Pandas has a function that can help you do this directly.

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
train_size=0.7, test_size=0.3, random_state=100)
```

In the code shown above, the last argument, 'random_state', is used to set a seed. Now, setting a seed makes sure the training and test data are actually the same. After performing the training–test split of the data, you need to make sure no variable has a disproportionate effect on the model predictability. To ensure this, you squeeze the values of a column into smaller values. Feature scaling can help us here. In the given case study, we have used the 'standard scaler'. After standardisation, the ranges of the variables changed to:

- Tenure = -1.28 to +1.61
- Monthly charges = -1.55 to +1.79
- Total charges = -0.99 to 2.83

Clearly, none of the variables will have a disproportionate effect on the model's results now.

In the problem above, you saw that the data has a churn rate of almost 27%. It is important to check the churn rate since you usually want to have a balance between the 0s and 1s in your data. Now, if one of the classes is more prevalent in the data set compared with the other, for instance, if 95% of the data points are 0s, then simply predicting all of the data points as 0 will give you an accuracy of 95%. This is called class imbalance. In the given problem, there are 27% 1s. So, it is not quite imbalanced.

Building your First Model

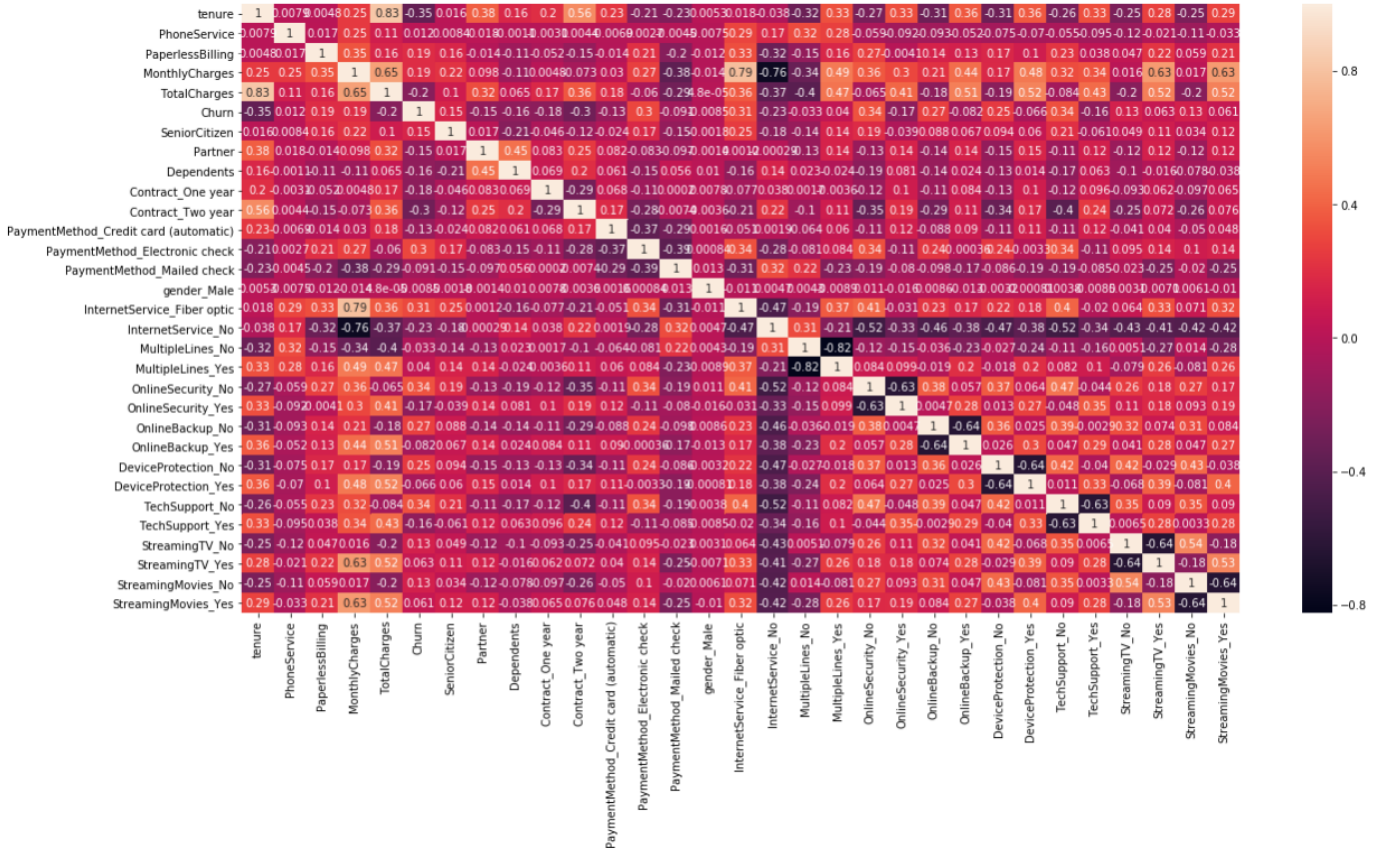
One last step before building a model is to check the correlation between all remaining variables. A strong correlation between any of the variables in the training data set will lead to a wastage of computational resources. A strong correlation implies that the information present in the two columns can also be stored in one column and so, there is no need to keep both the columns.

A combination of Matplotlib and Seaborn can help you get the correlation between all the variables in the data set. The code segment given here is how it is implemented

```
# Importing matplotlib and seaborn
```

```
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
# Let's see the correlation matrix
plt.figure(figsize = (20,10))        # Size of the figure
sns.heatmap(telecom.corr(),annot = True)
plt.show()
```

This command above gives the result plot shown below.



In this plot, the extremely dark and the extremely light colours represent high correlations. It is always a good practice to drop columns that have high correlations. Now, if two columns are strongly related to each other, then which one would you drop? The business context can help you make this decision. Keep the column that makes business sense.

So, after dropping the columns that have high correlations, you can build your first model. Instead of using the sklearn library, the case study has used the statsmodels library. The benefit of using the statsmodels library is that it has more statistical information, which can be used to improve the model further. We used the following command to train the model.

```
import statsmodels.api as sm
# Logistic regression model
logm1 = sm.GLM(y_train,(sm.add_constant(X_train)), family =
```

```
sm.families.Binomial())
logm1.fit().summary()
```

The output of the command above gives the summary of the model, the coefficients and the p-value of each feature. The key focus area is just the different **coefficients** and their respective **p-values**. As you can see, there are many variables whose p-values are high; this implies that such variables are statistically insignificant. So, you need to eliminate some of these variables to build a better model.

Feature Elimination using RFE

Based on the summary statistics, you inferred that many of the variables might be insignificant and, hence, you need to perform some feature elimination. Since the number of variables is quite large, the first technique that you used was RFE. RFE, or Recursive Feature Elimination, is an automated feature elimination algorithm. The RFE object takes two arguments: a machine learning (ML) algorithm object and an integer to represent the number of features that are to be considered. RFE algorithm trains a ML model of the mentioned ML type, finds the least important feature and eliminates it. Then, it retrains a new model and again finds the least important feature to eliminate. This process is repeated until the specified number of features are left.

Following are the steps to execute the RFE algorithm.

Step 1: Create an ML algorithm object

```
from sklearn.linear_model import LogisticRegression
logreg = LogisticRegression()
```

Step 2: Initialise the RFE object with relevant arguments and fit it on the data set

```
from sklearn.feature_selection import RFE
rfe = RFE(logreg, 15)
rfe = rfe.fit(X_train, y_train)
```

Then you can use the selected columns to build a model again; for this, you need to use the same set of commands as earlier. Now, if you observe the p-values of the features, then you will notice that they seem to be under control, except one feature, whose p-value is also not that high.

Now, you can use the probabilities calculated using the statsmodel to predict the class of a given data point. The following code snippets are helpful for this purpose.

```
# Getting the predicted values on the train set
y_train_pred = res.predict(X_train_sm)
```

```
y_train_pred[:10]
```

The code given above will get the probability predictions for the first 10 rows of the data frame. Now, create a data frame with the probabilities, the actual churn label and the customer ID. The following code will be helpful for this purpose.

```
y_train_pred_final = pd.DataFrame({'Churn':y_train.values,
'Churn_Prob':y_train_pred})
y_train_pred_final['CustID'] = y_train.index
y_train_pred_final.head()
```

Now, using the data frame created in the code above, assign the class labels to the data points. A data point with a probability more than 0.5 is assigned the positive class, whereas a data point with a probability less than 0.5 is assigned the negative class. The value of 0.5 is chosen arbitrarily, and it is called the threshold value. The following code will help achieve this classification.

```
y_train_pred_final['predicted'] =
y_train_pred_final.Churn_Prob.map(lambda x: 1 if x > 0.5 else 0)

# Let's see the head
y_train_pred_final.head()
```

Now you can use the prediction to evaluate the predictability of the model.

Confusion Matrix and Accuracy

Confusion matrix and accuracy are tools to measure the predictability of a classification model. The default cut-off of 0.5 is used to classify customers into 'Churn' and 'Non-Churn' categories. Now, since you are classifying customers into two classes, you will obviously have certain errors. Following are the two classes of errors that you would have:

- 'Churn' customers being classified (incorrectly) as 'Non-Churn' customers
- 'Non-Churn' being classified (incorrectly) as 'Churn' customers

To capture these errors and to evaluate how good the model is, you will use a tool known as the '**Confusion Matrix**'. A typical confusion matrix would look like this:

Actual	Predicted	
	No (Non-Churn)	Yes (Churn)
No (Non-Churn)	1406	143
Yes (Churn)	263	298

You can use the following command to get the confusion matrix in Python:

```
from sklearn import metrics
# Confusion matrix
confusion = metrics.confusion_matrix(y_train_pred_final.Churn,
y_train_pred_final.predicted )
print(confusion)
```

This command will print the actual confusion matrix, which can be used to determine the accuracy of the model. Now, if you have a confusion matrix, then either you can use that to determine the model accuracy, or you can determine the accuracy directly using the sklearn library. As given below,

```
# Let's check the overall accuracy.
print(metrics.accuracy_score(y_train_pred_final.Churn,
y_train_pred_final.predicted))
```

The accuracy of this model is given by:

$$Accuracy = \frac{1406+298}{1406+143+263+298} \approx 80.75$$

Manual Feature Elimination

Now that you are left with 13 features, you can eliminate some more features using a method called VIF (Variance Inflation Factor). Unlike RFE, VIF is a manual feature selection method. As you learnt under correlation matrix, even after eliminating high-correlation features, some features with collinearity still exist, i.e., there is still some multicollinearity among the features. So, you definitely need to check the VIFs as well to further eliminate redundant variables. VIF calculates how well one independent variable is explained by all the other independent variables combined, and it is expressed as shown below:

$$VIF_i = \frac{1}{1-{R_i}^2},$$

where 'i' refers to the i^{th} variable that is being represented as a combination of the rest of the independent variables. A regression model is run with x_i as the independent variable and all other features are dependent variables. This regression model will give an R^2 value, which is used to calculate the VIF of the feature x_i. The closer the R^2 value is to 1, the higher is the VIF value. A high VIF value signifies that x_i can be described by a linear combination of all other features. You might want to drop the feature with high VIF. The following code will help you get the VIF values for all the features.

```
# Create a dataframe that will contain the names of all the feature
variables and their respective VIFs
vif = pd.DataFrame()
vif['Features'] = X_train[col].columns
vif['VIF'] = [variance_inflation_factor(X_train[col].values, i) for i
in range(X_train[col].shape[1])]
vif['VIF'] = round(vif['VIF'], 2)
vif = vif.sort_values(by = "VIF", ascending = False)
vif
```

Using the VIF process iteratively, we had dropped the features 'PhoneService' and 'TotalCharges' as part of manual feature elimination. The statsmodel summary also gives the coefficients of each of the features, and this helps make the model highly interpretable. Using the coefficient of each feature, you can calculate the value of log odds for a given observation. From the log odds of a data point, you can find the class of that data point given the threshold is set at 0.5.

Summary

Logistic Regression - Model Evaluation with Python

This session had a two-fold objective to improve the model built in the earlier session and learn the metrics that are used to evaluate a classification model. In this session, you learnt about the different evaluation metrics and the codes to implement them.

Sensitivity and Specificity in Python

You already know how to find the confusion matrix, using the confusion matrix sensitivity and specificity can be found. Now, let's recall what sensitivity and specificity mean.

1. **Sensitivity:**
   It is the fraction of correctly identified positives among all the positive labels in a data set. It is exactly the same as recall. It is expressed as follows:

   Sensitivity = (TP)/(TP + FN)

2. **Specificity:**
   It is the fraction of correctly predicted negative classes among all the negative classes of a data set.

It is expressed as follows:

Specificity = (TN)/(TN + FP)

To calculate sensitivity and specificity, you need to extract the values of TP, FP, TN and FN from the confusion matrix. You can do this by indexing out the elements from the matrix. You can use the following code for this purpose.

```
TP = confusion[1,1] # true positive
TN = confusion[0,0] # true negatives
FP = confusion[0,1] # false positives
FN = confusion[1,0] # false negatives
```

Using these values, you can calculate several other metrics. However, business context is necessary to decide which metric to observe, or what should the benchmark of each individual metric be. For example, for a disease detection model, you might want to detect all the positives; in fact, it might even be acceptable if you marked a few negatives as positives, as you will complement this model with some actual testing method, where the negatives will get detected. In this case, you need very high sensitivity. Similar is the case with telecom churn; for instance, how will you deal with customers who are predicted to churn? Perhaps extend some offers to them so they stick with you, right? So, it is important to capture as many positives as possible. In achieving this, it is acceptable to allow the number of false positives to increase only up to a certain limit; but if the false positive number exceeds a certain threshold, then it means you are simply wasting money. So, in this model, the budget will decide the acceptable number of false positives, although the model needs to have high sensitivity.

So, your model seems to have **high accuracy (~80.475%)** and **high specificity (~89.931%)**, but **low sensitivity (~53.768%)**, and since you are interested in identifying the customers which might churn the model is still insufficient.

ROC Curve

Until this point in the analysis, we have maintained a threshold of 0.5. This is an arbitrarily chosen value, and changing it can lead to a lot of changes to the model predictability. For low values of threshold, you would have a higher number of customers predicted as a 1 (Churn). This is because if the threshold is low, then it basically means that everything above the threshold would be 1 and everything below it would be 0. So, naturally, a lower threshold would mean a higher number of customers being identified as ‘Churn’. Similarly, for high values of threshold, you would have a higher number of customers predicted as a 0 (Non-Churn) and a lower number of customers predicted as a 1 (Churn).

The following terminology could prove helpful in analysing how the predictability changes with change of threshold.

True Positive Rate (TPR)

This is the ratio of the number of correctly predicted positives and the total number of positives. The formula shown in the video for TPR is as follows:

$$True\ Positive\ Rate\ (TPR) = \frac{True\ Positives}{True\ Positives + False\ Negatives} = \frac{TP}{TP+FN}$$

As you might remember, the formula above is nothing but the formula for **sensitivity**. Hence, the term true positive rate, which you just learnt, is nothing but sensitivity.

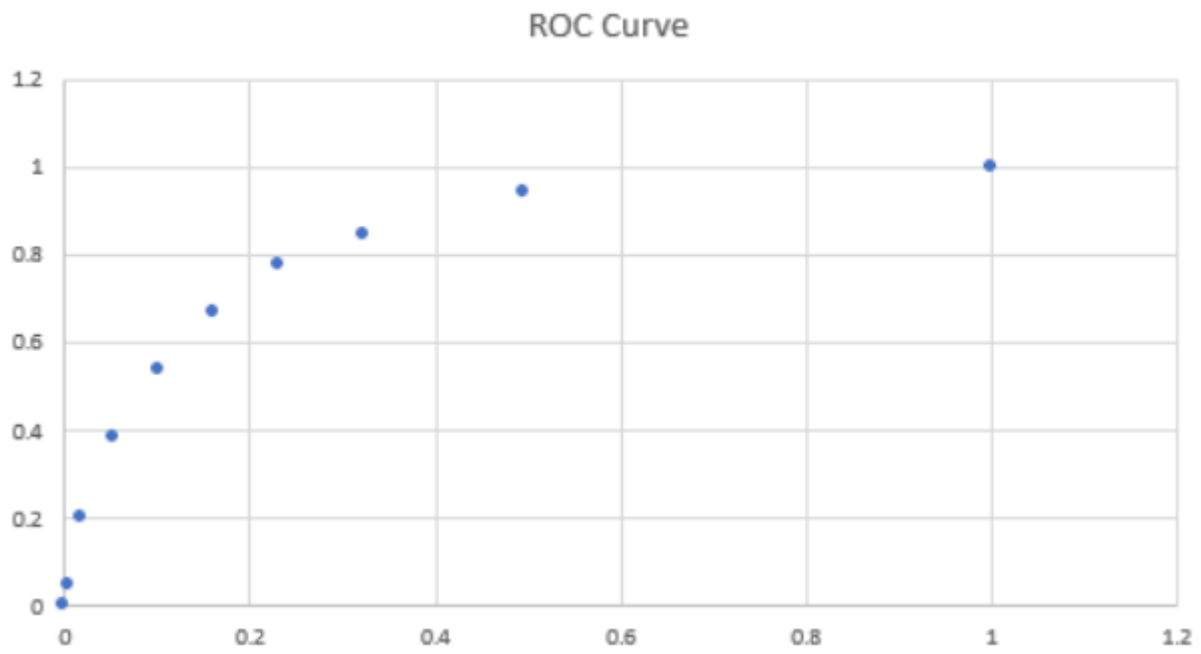
False Positive Rate (FPR)

It is the ratio of the number of false positives (0s predicted as 1s) and the total number of negatives. The formula for FPR formula is as follows:

$$False\ Positive\ Rate\ (FPR) = \frac{False\ Positives}{True\ Negatives + False\ Positives} = \frac{FP}{TN+FP}$$

So, now that you have understood these terms are, you will next learn about **ROC curves**, which depict the **trade-off between TPR and FPR**. And as we established from the above formulas that TPR and FPR are nothing but sensitivity and (1 - Specificity), respectively, so it can also be looked at as a trade-off between sensitivity and specificity. So, to analyse the change in model predictability with change in threshold, you need to find the TPR and FPR for each threshold value.

When you plot TPR against FPR, you get a graph that shows the trade-off between these two metrics. This graph is known as the ROC curve, or the receiver operating characteristic curve.



As you can see, for higher values of TPR, you will also have higher values of FPR, which might not be good. So, it is all about finding a balance between these two metrics and that's what the ROC curve helps you find.

ROC Curve in Python

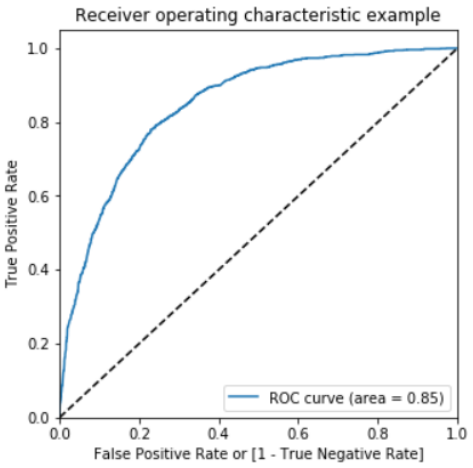
Earlier, you saw ROC curves in Excel. Now, you can use the following code to implement an ROC curve in Python.

```
# Defining the function to plot the ROC curve
def draw_roc( actual, probs ):
    fpr, tpr, thresholds = metrics.roc_curve(actual, probs,
                                             drop_intermediate = False)
    auc_score = metrics.roc_auc_score( actual, probs )
    plt.figure(figsize=(5, 5))
    plt.plot( fpr, tpr, label='ROC curve (area = %0.2f)' % auc_score )
    plt.plot([0, 1], [0, 1], 'k--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate or [1 - True Negative Rate]')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver operating characteristic example')
    plt.legend(loc="lower right")
    plt.show()
```

```
    return None

# Calling the function
draw_roc(y_train_pred_final.Churn, y_train_pred_final.Churn_Prob)
```

Interpreting an ROC Curve

So, here is the ROC curve that you obtained in the notebook. Note that this is the same curve that you obtained in Excel as well, although there, you had used a scatter plot to represent the discrete points, whereas you are using a continuous line here.



The 45₀ Diagonal

For a completely random model, the ROC curve will pass through the 45-degree line, which has been shown in the graph above, and in the best case, it would pass through the upper left corner of the graph. So, the least area of an ROC curve is 0.5 and the highest area can be 1.

The Sensitivity–Specificity Trade-Off

As you saw in the previous segment as well, the ROC curve shows the trade-off between TPR and FPR, which can, essentially, also be viewed as a trade-off between sensitivity and specificity. As you can see, on the y-axis, you have the values of sensitivity and on the x-axis, you have the (1 - Specificity) values. Observe that in the curve, when sensitivity is increasing, (1 - Specificity) is also increasing. And since (1 - Specificity) is increasing, it simply means specificity is decreasing.

Area Under the Curve

By calculating the Area under the Curve (AUC) of an ROC curve, you can determine how good your model is. If the ROC curve is more to the top-left corner, then it means that the model is quite good, and if it is more to the 45-degree diagonal, then it means that the model is almost completely random. So, the larger the AUC, the better would your model be; you saw this in the previous segment as well.

upGrad

Finding the Optimal Threshold

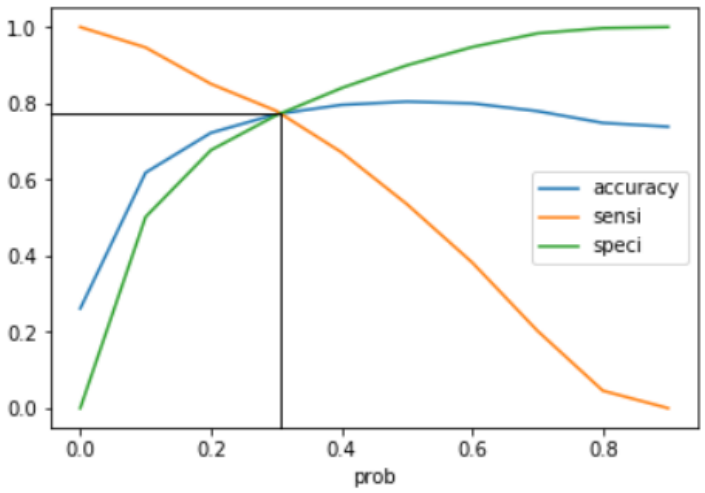
Previously, you learnt that there is a trade-off between sensitivity and specificity. Now, let's use this information to find the optimal cut-off. You can calculate the sensitivity, specificity and accuracy for each threshold point using the following code:

```
# Now, let's calculate the accuracy, sensitivity and specificity for
various probability cut-offs.
cutoff_df = pd.DataFrame( columns =
['prob','accuracy','sensi','speci'])
from sklearn.metrics import confusion_matrix

# TP = confusion[1,1] # true positive
# TN = confusion[0,0] # true negatives
# FP = confusion[0,1] # false positives
# FN = confusion[1,0] # false negatives

num = [0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9]
for i in num:
    cm1 = metrics.confusion_matrix(y_train_pred_final.Churn,
y_train_pred_final[i] )
    total1=sum(sum(cm1))
    accuracy = (cm1[0,0]+cm1[1,1])/total1

    speci = cm1[0,0]/(cm1[0,0]+cm1[0,1])
    sensi = cm1[1,1]/(cm1[1,0]+cm1[1,1])
    cutoff_df.loc[i] =[ i ,accuracy,sensi,speci]
print(cutoff_df)
```

The output of this line of code will be a data frame with sensitivity, specificity and accuracy for all the threshold values, as provided below.

	prob	accuracy	sensi	speci
0.0	0.0	0.261479	1.000000	0.000000
0.1	0.1	0.619667	0.946387	0.503989
0.2	0.2	0.722674	0.850039	0.677579
0.3	0.3	0.771434	0.780109	0.768363
0.4	0.4	0.795002	0.671329	0.838790
0.5	0.5	0.804754	0.537685	0.899312
0.6	0.6	0.800284	0.385392	0.947180
0.7	0.7	0.779764	0.205128	0.983219
0.8	0.8	0.749289	0.050505	0.996699
0.9	0.9	0.738521	0.000000	1.000000

The data frame shown above can be plotted to analyse the most suitable threshold values. As you can see, when the probability thresholds are very low, the sensitivity is very high and the specificity is very low. Similarly, for larger probability thresholds, the sensitivity values are very low, but the specificity values are very high. At the threshold values of approximately 0.3, the three metrics (accuracy, sensitivity and specificity) seem to be almost equal with high enough values, and hence, we choose 0.3 as the optimal cut-off point. The following graph showcases that at approximately 0.3, the three metrics intersect.



As you can see, at a threshold of approximately 0.3, the curves of accuracy, sensitivity and specificity intersect, and they all take a value of about 77–78%. You could have chosen any other cut-off point as well based on which of these metrics you want to be high. If you wanted to capture the 'Churns' better, then you could compromise on accuracy and choose a lower cut-off and if you wanted a higher accuracy you would choose a higher cut off. This completely depends on the situation you are in. In this case, we are choosing the 'optimal' cut-off point to give you a fair idea of how the threshold should be chosen.

Precision and Recall

Apart from sensitivity and specificity, you should know about **'Precision'** and **'Recall',** which are two more metrics that are widely used in the industry. These metrics are quite similar to sensitivity and specificity; knowing these exact terminologies will be helpful, as both of these pairs of metrics are often used in the industry.

Let's go through the definitions of precision and recall once again.

- **Precision:** It is the probability that a predicted 'Yes' is actually a 'Yes'.

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

Remember that 'Precision' is the same as the 'Positive Predictive Value' that you learnt about earlier. Going forward, we will call it ‘precision’.

- **Recall:** It is the probability that an actual 'Yes' case is predicted correctly.

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

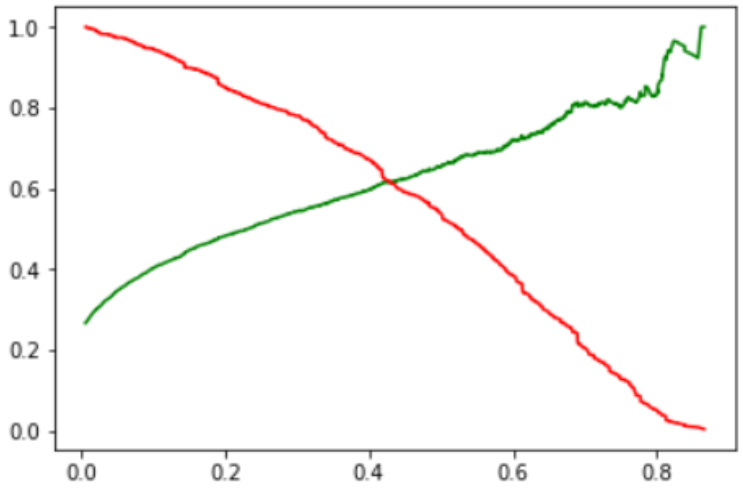
Remember that 'recall' is exactly the same as sensitivity. Do not get confused between these two. You can calculate the precision and recall of any model in Python using the following code:

```
from sklearn.metrics import precision_score, recall_score
precision_score(y_train_pred_final.Churn, y_train_pred_final.predicted)
recall_score(y_train_pred_final.Churn, y_train_pred_final.predicted)
```

Similar to sensitivity and specificity, precision and recall also have a trade-off. You can either have a model with high precision or one with high recall. It is rare to find a model with both high precision and high recall. The following code will help you plot the variation of precision and recall with a change in threshold:

```
from sklearn.metrics import precision_recall_curve
p, r, thresholds = precision_recall_curve(y_train_pred_final.Churn,
y_train_pred_final.Churn_Prob)
plt.plot(thresholds, p[:-1], "g-")
plt.plot(thresholds, r[:-1], "r-")
plt.show()
```

The set of commands given above will give a plot similar to the one shown below.



As you can see, the curve is similar to the one that you had for sensitivity and specificity, except now, the curve for precision is quite jumpy towards the end. This is because the denominator of precision, i.e., (True Positives + False positives), is not constant, as TP and TP depend on the predicted values of 1s. Since the predicted values can swing wildly, you get a very jumpy curve.

Both the ROC curve and the precision and recall curve are evaluation metrics for the same model; you can select either of these based on your preference and as the situation demands. Some industries prefer one to the other, but you will get similar results with either.

Making Predictions

You have done all kinds of tuning to improve the model's predictability. But remember that all the model improvement that you have achieved so far has been on the training data set. To actually test a model's predictability, you need to use an unseen data set, and we have set aside the data set for this purpose, that is, the test data set.

Extract the necessary columns, and apply the model on them to predict the class labels. The code for this can be found in the notebook provided with the session. The prediction can then be made using the threshold that you derived from the specificity and sensitivity graph. After the predictions are made, you can get the confusion matrix and evaluate the model in terms of the same metrics used earlier.

You will observe that the metrics seem to hold on to the test data set as well. So, it looks like you have created a decent model for the churn data set, as the metrics are similar for both the training and the test data sets.

Summary

Logistic Regression - Model Evaluation With Python

In this session, you learnt about the use of logistic regression in PySpark. The use case was a problem to determine whether an advertisement will be clicked on or not. This is called the click-through rate (CTR) prediction. This was solved in the EMR cluster on the AWS platform.

Click-Through Rate Prediction

Most of the web pages that you visit display some ads. The online advertising industry is huge, and players such as Google, Amazon and Facebook generate a lot of revenue by targeting their advertisements to the

correct audience. Most of the decisions pertaining to the ads involve data-driven solutions such as the following:

1. How does one decide which ad is to be shown to whom?
2. Of the multiple companies that apply for their ads of products falling in the same category, how does one decide whose ad is to be shown?
3. Which ad should be placed in which section of a page?
4. Should a particular ad be pushed on a mobile device or a desktop/laptop?

These decisions depend on numerous factors, such as the time of the ad, the site on which the ad is shown, characteristics of the users looking at the ad, and its demographics.

An important exercise that marketing companies need to do before taking a decision is the click-through rate (CTR) prediction exercise. The objective of this exercise is to predict whether the audience will click on an ad or not, which will help the marketing teams to answer all the ad placement-related questions. The data set used in this case study is taken from a kaggle competition that was hosted a few years ago.

After this, simple EDA was done on the data. Some observations of the data are as follows:

1. The data types of most of the columns in the data set are **string** and **integer** except for the label column, which is Boolean.
2. The data set is **imbalanced**; the percentage of data points with the label '0' is **87%** and that with the label '1' is **13%**.
3. C1–C20 are **anonymised categorical** columns in the string form.
4. Certain columns such as the **banner positions** clearly show that one position is clicked on more often than the others.
5. **True clicks** are present in all the levels of certain columns such as C1. Some levels such as 1002, 1005 and 1012 have a high CTR.

Data Sampling

The actual data set given on Kaggle has **40M** rows. It is not wise to start building a prediction model with such a large data set because the iterative process of model building will utilise a lot of resources. So, sampling needs to be done on the data set.

This exercise uses stratified random sampling. While taking a stratified sample, the data is divided into strata, and then, a proportionate random sample is drawn from each stratum. The greatest advantage of stratified sampling is that the drawn sample represents the actual data set. This ensures that the model gives relevant weightage to all the classes. The code used to obtain the stratified data set is given below.

```
#import the larger data set and draw a stratified sample from it.
```

```
df = sqlContext.read.csv('s3a://.../10M.csv', header=True,
inferSchema=True)

from pyspark.sql.types import IntegerType
from math import floor
from pyspark.sql.functions import rand
from pyspark.sql.functions import col

def stratifiedSample(df, N, labelCol="y"):
    ctx = df.groupby(labelCol).count()
    ctx = ctx.withColumn('frac', col("count") / df.count())
    frac = ctx.select("y", "frac").rdd.collectAsMap()
    pos = int(floor(frac[1] * N))
    neg = int(floor(frac[0] * N))
    posDF = df.filter(col(labelCol) == 1).orderBy(rand()).limit(pos)
    negDF = df.filter(col(labelCol) == 0).orderBy(rand()).limit(neg)
    return posDF.unionAll(negDF).orderBy(rand())

df = df.withColumn("y", df["click"].cast(IntegerType()))
xdf = stratifiedSample(df, 1_000_000)
```

The function given above can draw a stratified sample from a large data set. The same job can be done by an inbuilt function called sampleByKey. The following code example is an of its use.

```
df.sampleByKey(False , {a:0.4, b:0.6} , seed = 0)
```

Following are the arguments of this function:

False: A Boolean value representing whether the sample is drawn with or without replacing. Here, 'False' denotes 'without replacement'.

{a:0.4, b:0.6}: This is a dictionary that contains information of the proportion to be drawn from a class.

Seed = 0: The number represents the sample drawn, i.e., if the seed is the same, then the resulting sample will also be the same.

Data Preparation I

Some columns such as the 'banner_position' are integers, but they are categorical variables, not continuous variables. Representing them with integers gives them an order that does not exist. For instance, the banner position 3 is not three times the banner position 1. So, you need to find a way to

communicate only the category of that particular data point using integers and simultaneously make sure that no other relation or order is communicated.

Similarly, different types of variables need to be handled differently to make sure that the right amount of information is being conveyed. Before learning about encoding variables, let's understand the different types of categorical variables, which are as follows:

1. **Nominal variables**: These are categories with no relation among them, and you can think of them as tags. For example, which hand do you write with, right or left? They have no relationship between them.
2. **Ordinal variables**: These are categorical variables with an order among them. For example, people's level of education: graduate, postgraduate and doctorate. These are categories, but they have an order among them. A person with a doctorate degree is more educated than a graduate. These variables can sometimes be input as integers because there is an order among the categories.

Then, there are variables depicting values such as temperature, height and weight. They do not need to be encoded and can be input directly into machine learning (ML) algorithms. So, variables such as the banner position are nominal variables. Hence, you will one-hot encode them.

One-hot encoding

As discussed earlier, you need to communicate the category without sharing any extra information. One-hot encoding creates K columns for K categories. Each category is represented by a vector of 0s, and each vector has only one 'hot value' or '1'. The ML library in PySpark has a prebuilt transformer for one-hot encoding, that is, the OneHotEncoder(). To implement it, you need to simply call the library and pass the column to encode. The estimator will provide the encoded output.

The output of OneHotEncoder() in PySpark is called a '**sparse matrix**'. Let's interpret one value of the matrix given below to understand how the sparse matrix represents the same amount of information as a one-hot encoded vector.

(7,[0],[1.0])

Here, 7 represents the number of columns in the matrix.

0 is the position with a non-zero value.

1 is the non-zero value at the 0th position.

In the expanded form, (7,[0],[1.0]) will be represented as 1,0,0,0,0,0,0.

String indexing

The string indexer takes in a column, finds all the unique strings in that column, assigns unique integers to them and gives back a column with integers representing the categories, which is similar to the 'banner_pos' column. Then, this can be passed to the one-hot encoder to obtain an encoded sparse matrix.

Generalising your learnings so far, in any given data set, what are the columns that you will encounter most often? You will encounter continuous variables in the numerical form, categorical variables in the numerical and string forms, Boolean variables and date/time information. You should be able to convert all this information into the numerical form so that the ML algorithm can process them.

Data Preparation II

A learning algorithm takes only the following two columns as input: The first one contains all the features combined as a list, and the second one is the class label. So, you need to combine all the features in a single column, for which a transformer called **VectorAssembler** is used. The code that you need to implement this is given below.

```
# Combine all the feature columns into a single vector called features.

assembler = VectorAssembler(inputCols=trainCols, outputCol='features')
xdf = assembler.transform(xdf)
```

The output of the VectorAssembler can be pushed into logistic regression. Let's revise the steps followed to make the data ready for ML.

1. **Continuous** columns with numerical data: No processing required
2. **Nominal** categorical variables represented with integers: One-hot encoding
3. **Ordinal** categorical variables with integer representation: No processing required
4. **Nominal** categorical variables represented with strings: String indexer + One-hot encoding
5. **Ordinal** categorical variables represented with strings: String indexer (Make sure the correct categories get the appropriate integer.)

Instead of following these steps one by one, a parallel way to achieve the same output is the pipeline API in the ML library.

Before learning about the pipelines, let's revise the APIs available in the **SparkML library**. These APIs are built on the DataFrame structure and are as follows:

1. **Transformers**: These transform the DataFrame from one form to another, usually by adding a few columns or manipulating the values in the columns. The transform() method is used to transform a data frame. Any object that makes changes in a DataFrame is called a transformer.

2. **Estimator**: Estimators are learning algorithms, and fit() is used to call an estimator into action. Fit() will take a DataFrame as input and create a model object. This model itself is a transformer. The job of a transformer is to transform a DataFrame; for example, a transformer created by a logistic model uses weights to find probabilities, and using these probabilities, it predicts the class of the data point. The actual transformation is creating two new columns, 'probability' and 'prediction'. The transformation and the transformation object will differ across algorithms.

The pipeline API combines all the transformers and estimators into one object so that the ML process becomes streamlined and more interpretable. The pipeline object has a parameter called stages, which takes as input a list of transformers and estimator objects that will be executed when the pipeline object is executed. The following code shows its implementation.

```
from pyspark.ml.feature import
(VectorAssembler,VectorIndexer,OneHotEncoder,StringIndexer)

gender_indexer = StringIndexer(inputCol='Sex',outputCol='SexIndex')

gender_encoder = OneHotEncoder(inputCol='SexIndex',outputCol='SexVec')

embark_indexer =
StringIndexer(inputCol='Embarked',outputCol='EmbarkIndex')

embark_encoder =
OneHotEncoder(inputCol='EmbarkIndex',outputCol='EmbarkVec')

cols=['Pclass','SexVec','Age','SibSp','Parch','Fare','EmbarkVec']

assembler = VectorAssembler(inputCols=cols,outputCol='features')

from pyspark.ml import Pipeline
pipeline =
Pipeline(stages=[gender_indexer,embark_indexer,gender_encoder,embark_en
coder,assembler])

fit_model = pipeline.fit(df)
```

In the code given above, a few transformer objects are created, such as gender_indexer and embark_indexer. All these objects are put together in a pipeline to create a pipeline object. When the pipeline object is executed, all the internal objects are executed in the order that they are specified.

Model Building

In this segment, you built your first logistic model in PySpark. A logistic regression object has a few settings that can change the type of logistic model that you are building. One example of the logistic model is given in the following code.

```
# Create a logistic regression with vanilla settings and run it over
the data frame.

from pyspark.ml.classification import LogisticRegression

lr = LogisticRegression(featuresCol='features', labelCol='y')
model = lr.fit(xdf)
result = model.evaluate(xdf)
```

The parameters passed to it were featureCol and labelCol, which are used to specify the names of the features and the label columns, and by default, the values of these parameters are set to 'features' and 'label', respectively. These columns need to be in the DataFrame, and if they are not, then the compiler will throw an error.

A few important parameters of a logistic object are as follows:

threshold: float between 0 and 1; used to set the threshold/cut-off for classification into classes 0 and 1

fitIntercept: Boolean; used to specify whether the weighted feature column should have an intercept or not

family: [auto, binomial, multinomial]; used to specify the type of regression to use

The following two steps are involved in implementing logistic regression:

1. **Creating the logistic regression object**: First, create an object with all the properties and attributes that you need.

   ```
   lr = LogisticRegression(featuresCol='features', labelCol='label')
   ```

2. **Training the model**: Then, use the object that you created to train the data.

   ```
   model = lr.fit(encoded)
   ```

   The .fit() method is used to begin the training process. It works with all the estimators, and the argument passed to the .fit() method is the DataFrame to be used to train the model.

The evaluate() method is used to evaluate a model's performance. The evaluate() is called on a trained model, and it takes the DataFrame on which the evaluation needs to be performed.

A model is evaluated on the following metrics:

1. **Accuracy**: The accuracy of the model is 0.83. So, the model is able to correctly guess the class of 83% data points. But recall that this data set has a heavy class imbalance, and so, other metrics are needed to correctly evaluate the model's performance.
2. **Area under the ROC curve (ROC AUC)**: The ROC AUC of the model is 0.73. It is not intuitive, and the judgement is based on the magnitude of the AUC. The models are said to be good if they have an ROC AUC value closer to 1.
3. **Recall by label**: A better metric than 'accuracy' is 'recall by label'. As the name suggests, it finds the recall with respect to each class label. The model is able to correctly identify 99% of the negative classes and only 6% of the positive classes. So, this model does not perform well in terms of predicting which ads are likely to be clicked on, which is important information, and correctly predicting only 6% of the true cases is not helpful.

Model Improvement I

The performance of a model refers to how well it can predict class labels and how fast it can do so. In the next couple of segments, we will focus on the predictability aspect of a model's performance.

Using more data

The most brute-force way to improve a model's predictability is to use more training data. The higher the number of data points that a model trains on, the better will it be at predicting. However, this involves certain issues. Often, the improvement in a model's performance that is achieved by training on large amounts of data is not worth the computational resources spent on it. Sometimes, abundant data might not be available. So, even though this method improves the predictability of the model, similar improvements with the same data can be achieved in other ways.

Using more features

A model is only as good as the data on which it trains; having the optimum number of and the most relevant features improves a model's performance significantly. In the previous segment, the model that Jaidev built used only a few features from the data, which were selected based on domain knowledge. Other anonymised categorical variables in the data set were not considered for building the model.

In this segment, a brute-force way to improve the model's predictability was demonstrated. Simply increasing the features might not necessarily increase a model's predictability. You also learnt about the method to use the model that you trained on unseen data.

As expected, adding more features did not result in an increase in the model's predictability.

Model Improvement II

A model's predictability can be improved in a few ways. If you recall the earlier case study, then you will notice that a lot of feature selection was carried out to improve the model, and this is a good starting point. The CTR data set is a 'real-world' data set, and so, to maintain privacy, it has been encoded or masked. The values in each column refer to real world properties.

Feature engineering

Upon examining the data set, you can recognise the following feature selection steps:

1. Combine the fields site_id, site_category, site_domain, and app_id, app_category and app_domain.
2. Numerous entries in the column user_id are encoded with a single value; this repeating value should depict null values. Wherever the user_id is null, replace it with a combination of device_type and device_ip.
3. Find the frequency with which each user_id repeats.
4. Combine the user and the hour to find the frequency with which ads are shown to users within a particular hour.
5. Drop all the columns that:
   a. Are redundant,
   b. Have a lot of categories, and
   c. Do not contribute to the learning process.

After performing feature engineering, you will notice that the data has become 'lean' or that the number of columns has decreased a lot. Now, you can use this data set to train a logistic model and compare its predictability with that of the earlier one. The results of this model are still not promising. To improve this model's predictability even further, you can select a more suitable threshold. Selecting the correct threshold does not mean simply picking the threshold that gives the highest recall or precision. A lot of business groundwork goes into making this decision.

In the process of selecting a new threshold, a few new concepts are introduced, which are as follows:

Creating a dataframe from a list:

You can create a data frame from any list as long as it has a structure. The result list in the video was a list of lists, and all the internal lists have the same order. If such a structure exists, then you can create a schema using the StructType and StructField commands. The argument to StructType is a list of StructFields. Each StructField has the following three arguments: the name of the column, the data type of the column and a Boolean representing whether or not the column can have empty values. Then, you create an RDD from the list and convert it into a data frame by applying the schema to it.

Installing packages to the EMR cluster:

You can find the Python packages that are already present on EMR by running the following command:

```
sc.list_packages()
```

If a package that you need is not on the list, then you can use the following command to install them.

```
sc.install_pypi_package("pandas==0.25.1")

sc.install_pypi_package("matplotlib", "https://pypi.org/simple")
```

You can vision the web address mentioned in the command to check which packages are available on that page. All of them can be installed in the same way. Then, you can use these libraries similar to the way in which you use them in Python.

You have a model that has been trained on the data. You also have a test data set, which you can use to predict the class labels, and the transform() function. It adds a column named probability, which is a dense vector with two values because the problem being solved here is binomial. These probabilities represent the probability of a negative and positive class, and their sum in each row is 1. By default, the probability shown first is for the negative class, and the one shown second is for the positive class. Using the actual label and the probability we tried a few different thresholds. We also created a few new columns to help analyse each of the thresholds.

The most business-relevant columns that were created in the final data frame were ad_spend, conversions and spend per conversion. Finding the correct balance among these columns will help in selecting the correct threshold.

Consider a few different cases and thresholds you may want to select. If the product you are selling is niche and has a huge profit margin, then you might be tempted to pick 0.1 as the threshold. In this case, you do not want to lose potential customers, and the money being spent is not an issue. If the money being spent is a constraint and the product is universal, then losing a few customers will not be consequential; you will easily get new customers. In that case, you may want to pick a higher threshold such as 0.4. If you are looking to optimise the cost spent on getting the most conversions, then a threshold such as 0.2 or 0.25 may be ideal.

Cluster Management I

SparkUI is the window used to visualise the internal working of Spark, and exploring it can help you understand the bottleneck in your code. The SparkUI page has a few tabs at the top, which provide specific information about the Spark application, such as what jobs are running and what is the environment set-up.

Environment tab:

The environment tab contains a lot of information about different environments and configuration variables. The variables give information pertaining to the versions of Java, Scala, etc. You can also find Spark application-related information in the environment tab, such as the application name, the executors and the memory. You can use this tab to check the properties that are set.

Jobs tab:

You can find the summaries of the jobs assigned to the Spark application. These jobs are grouped together based on their execution status. The details of each job can be found by clicking on the job. In the detailed view, you will observe the stages of the job, DAG visualisation, the event timelines, etc.

A lot of other information is present in SparkUI. Most of it is relevant in specific cases. Although we have not covered all the tabs in SparkUI in detail in this module, it is highly recommended that you spend some time and explore SparkUI.

Cluster Management II

Now you can use SparkUI to figure out which jobs take longer to execute than expected to and then make an attempt to tweak the parameters of the cluster such that the job can be optimised. To do this, configure magic commands can be used.

Whenever a new Spark app is created on an EMR, it allocates certain default resources to the app. This limited allocation of resources is done to make sure that some resources are free in case you create more apps. But since you are using only one app, you can increase the resource allocation.

This allocation of resources can be changed manually by the magic configuration tool. It can set the environment of Spark from the Jupyter Notebook itself.

```
%%configure -f
 {"executorCores": 8}
```

Using the same tool, you can set a few other variables, such as driverMemory (string format), driverCores (integer format), executorMemory (string format), executorCores(integer format) and numExecutors(integer format).

Please remember to run this code first before performing other activities because the -f command will create a new Spark application, causing the current application progress to be lost. Also, the actual configuration information should be in the JSON format; otherwise, the cell will throw an error.

```
%%configure -f
 {"executorMemory": "3072M", "driverMemory ": "2000M"}
```

When you run the configure commands now, a new Spark app is created, and you can check its SparkUI to confirm the resource allocation.

Multiclass Classification

The foundation of logistic regression is built on the PMF of variables that can have one of the two possible values either true or false. Using this PMF, a likelihood function is derived and then maximised to find the coefficients of the features that can achieve the best separation. This whole algorithm is built for binary classification, but it can also be used to solve multiclass classification problems. These problems have more than two categorical outcomes, such as the blood group of a person, the mode of transport taken by a person to reach their workplace, and the political party a person is likely to vote for.

The logistic algorithm can be used in the following two ways:

1. **Algorithm 1 cascading models**: The first model is used to distinguish one class from the rest, and subsequent models are used to identify the next class from the remaining labels. This should be continued until the penultimate label. This approach seems logical and efficient, but it is not because the accuracy keeps on decreasing with every new model that you build. Moreover, the first label on which you want to build the model can be selected in multiple ways.
2. **Algorithm 2 individual models**: For K variables, K models are trained, with each one to predict one class over the rest. For each data point, the label of the model with the best probability is selected.

The second one is actually used in multiclass prediction problems, as it is less prone to errors. Before applying the algorithm, you need to be aware of an assumption that is made while applying the logistic regression to multiclass problems.

The independence of irrelevant alternatives

This means that the presence or absence of any alternative does not affect the occurrence rate of any class. For example, consider a model that predicts the blood group of people. If the features of a data point are such that the label is predicted to be B+, then this prediction will not change based on the presence of a different class such as AB+.

Why is this assumption important? Take a look at how the training takes place; k models will train to predict k labels/classes. Each model will predict the probability of a data point belonging to the positive class of that model. At this stage, the probabilities of the models are compared, and the label is decided. The model built for any label does not consider other labels or their model parameters at any point during the training process. This is why the labels cannot affect each other.

The modification in the training process also changes the coefficient vector. A binary logistic model produces one vector of coefficients. Since multiple such models are being trained, multiple vectors of coefficients will be generated. These multiple vectors form a matrix of coefficients. In the prediction phase, the label of each data point will be predicted using all the models, and a vector of probabilities will be

formed in which each element of the vector will be the probability of a data point belonging to that particular label/class.

The softmax function is used to simultaneously train all the models. The softmax function takes in the $W^T x_i$ the weighted feature matrix and gives out a vector of probabilities, which can be used to predict the class label.

Regardless of the input given to the softmax function, the output vector has the following specific properties:

1. Each element on the output is between 0 and 1.
2. The sum of all the elements in the output equals 1.

The application of multinomial regression does not differ from that of the normal logistic regression model that you trained. The aforementioned changes happen in the background. In PySpark, you can simply set the family of the regression object to multinomial.