databricks Telecom Customer Churn Prediction

(https://databricks.com)

Customer Attrition

Customer attrition, also known as customer churn, customer turnover, or customer defection, is the loss of clients or customers.

Telephone service companies, Internet service providers, pay TV companies, insurance firms, and alarm monitoring services, often use customer attrition analysis and customer attrition rates as one of their key business metrics because the cost of retaining an existing customer is far less than acquiring a new one. Companies from these sectors often have customer service branches which attempt to win back defecting clients, because recovered long-term customers can be worth much more to a company than newly recruited clients.

Companies usually make a distinction between voluntary churn and involuntary churn. Voluntary churn occurs due to a decision by the customer to switch to another company or service provider, involuntary churn occurs due to circumstances such as a customer's relocation to a long-term care facility, death, or the relocation to a distant location. In most applications, involuntary reasons for churn are excluded from the analytical models. Analysts tend to concentrate on voluntary churn, because it typically occurs due to factors of the company-customer relationship which companies control, such as how billing interactions are handled or how after-sales help is provided.

Predictive analytics use churn prediction models that predict customer churn by assessing their propensity of risk to churn. Since these models generate a small prioritized list of potential defectors, they are effective at focusing customer retention marketing programs on the subset of the customer base who are most vulnerable to churn.

Context

Predict behavior to retain customers. You can analyze all relevant customer data and develop focused customer retention programs.

Content

Each row represents a customer, each column contains customer's attributes described on the column Metadata.

The data set includes information about:

- Customers who left within the last month – the column is called Churn
- Services that each customer has signed up for – phone, multiple lines, internet, online security, online backup, device protection, tech support, and streaming TV and movies
- Customer account information – how long they've been a customer, contract, payment method, paperless billing, monthly charges, and total charges
- Demographic info about customers – gender, age range, and if they have partners and dependents

Inspiration

To explore this type of models and learn more about the subject.

```
%scala

import org.apache.spark.sql.Encoders;

case class telecom(customerID: String,
                   gender: String,
                   SeniorCitizen: Int,
                   Partner: String,
                   Dependents: String,
                   tenure: Int,
                   PhoneService: String,
                   MultipleLines: String,
                   InternetService: String,
                   OnlineSecurity: String,
                   OnlineBackup: String,
                   DeviceProtection: String,
                   TechSupport: String,
                   StreamingTV: String,
                   StreamingMovies: String,
                   Contract: String,
                   PaperlessBilling: String,
                   PaymentMethod: String,
                   MonthlyCharges: Double,
                   TotalCharges: Double,
                   Churn: String )

val telecomSchema = Encoders.product[telecom].schema

val telecomDF = spark.read.schema(telecomSchema).option("header", "true").csv("/FileStore/tables/TelcoCustomerChurn.csv")

display(telecomDF)
```

Table

	customerID	gender	SeniorCitizen	Partner	Dependents	tenure	PhoneService	MultipleLines
1	7590-VHVEG	Female	0	Yes	No	1	No	No phone service
2	5575-GNVDE	Male	0	No	No	34	Yes	No
3	3668-QPYBK	Male	0	No	No	2	Yes	No
4	7795-CFOCW	Male	0	No	No	45	No	No phone service
5	9237-HQITU	Female	0	No	No	2	Yes	No
6	9305-CDSKC	Female	0	No	No	8	Yes	Yes

Truncated results, showing first 1,000 rows.

Printing Schema

```
telecomDF.printSchema()
```

```
root
 |-- customerID: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- SeniorCitizen: integer (nullable = true)
 |-- Partner: string (nullable = true)
 |-- Dependents: string (nullable = true)
 |-- tenure: integer (nullable = true)
 |-- PhoneService: string (nullable = true)
 |-- MultipleLines: string (nullable = true)
 |-- InternetService: string (nullable = true)
 |-- OnlineSecurity: string (nullable = true)
 |-- OnlineBackup: string (nullable = true)
 |-- DeviceProtection: string (nullable = true)
 |-- TechSupport: string (nullable = true)
 |-- StreamingTV: string (nullable = true)
 |-- StreamingMovies: string (nullable = true)
```

```
|-- Contract: string (nullable = true)
|-- PaperlessBilling: string (nullable = true)
|-- PaymentMethod: string (nullable = true)
|-- MonthlyCharges: double (nullable = true)
|-- TotalCharges: double (nullable = true)
```

Creating Temp View from Dataframe

```
%scala

telecomDF.createOrReplaceTempView("TelecomData")
```

Exploratory Data Analysis

- What is EDA?
- Exploratory Data Analysis (EDA) is an approach/philosophy for data analysis that employs a variety of techniques (mostly graphical) to

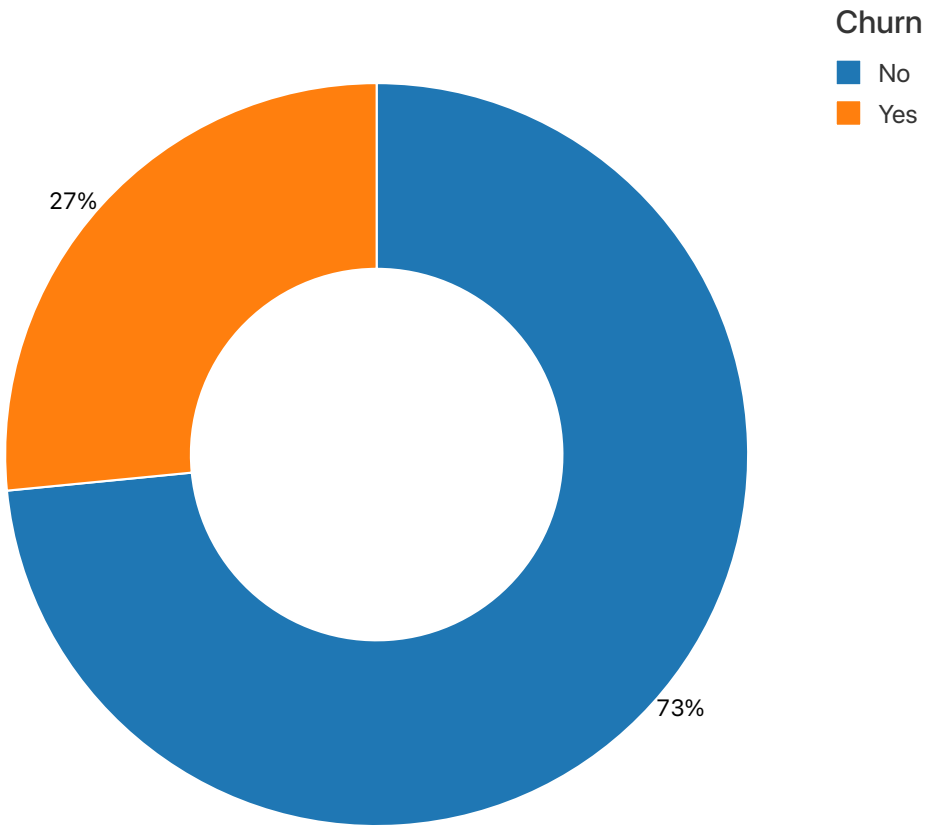
1. maximize insight into a data set;
2. uncover underlying structure:
3. extract important variables;
4. detect outliers and anomalies;
5. test underlying assumptions;
6. develop parsimo-tous models; and
7. determine optima factor settings.

Customer Attrition in Data

```
%sql

select Churn, count(Churn)
from TelecomData
group by Churn;
```

Visualization



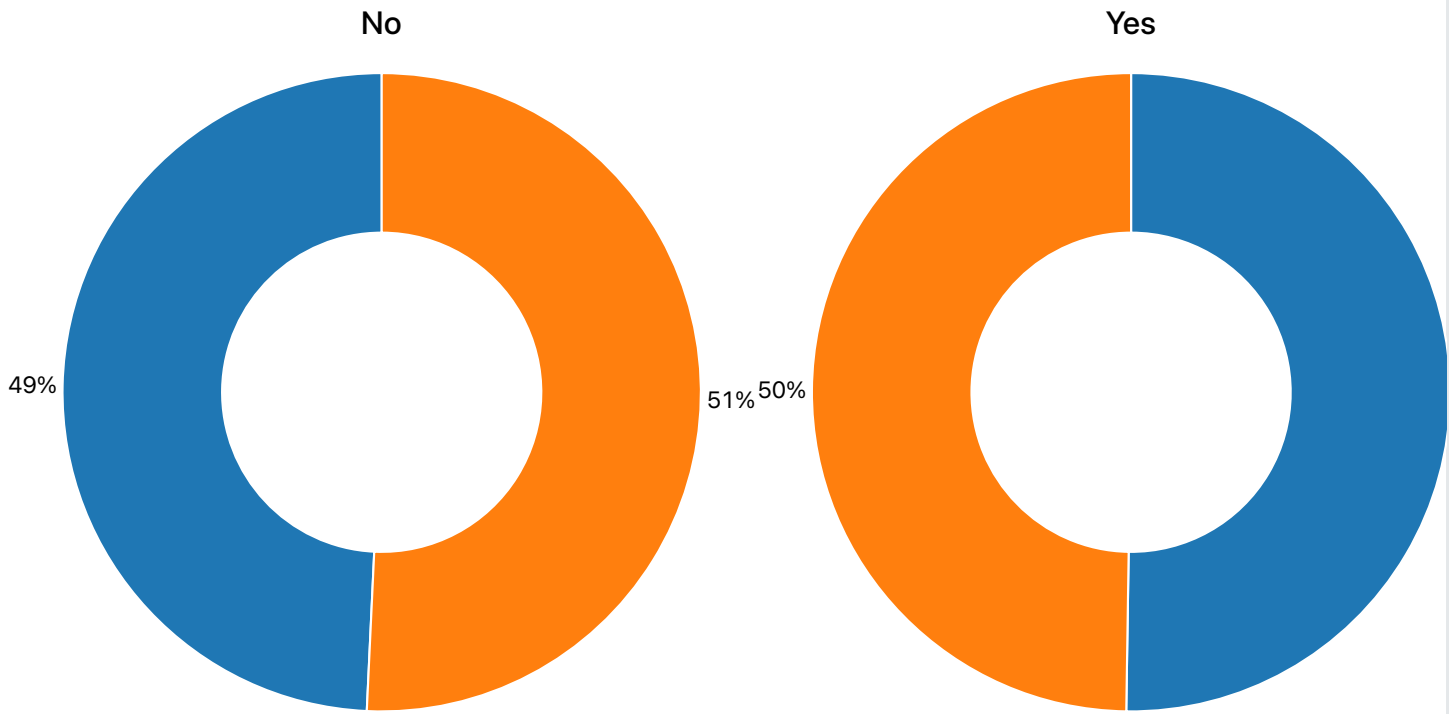
Showing all 2 rows.

Gender Distribution in Customer Attrition

```
%sql

select gender,count(gender), Churn
from TelecomData
group by Churn,gender;
```

Visualization

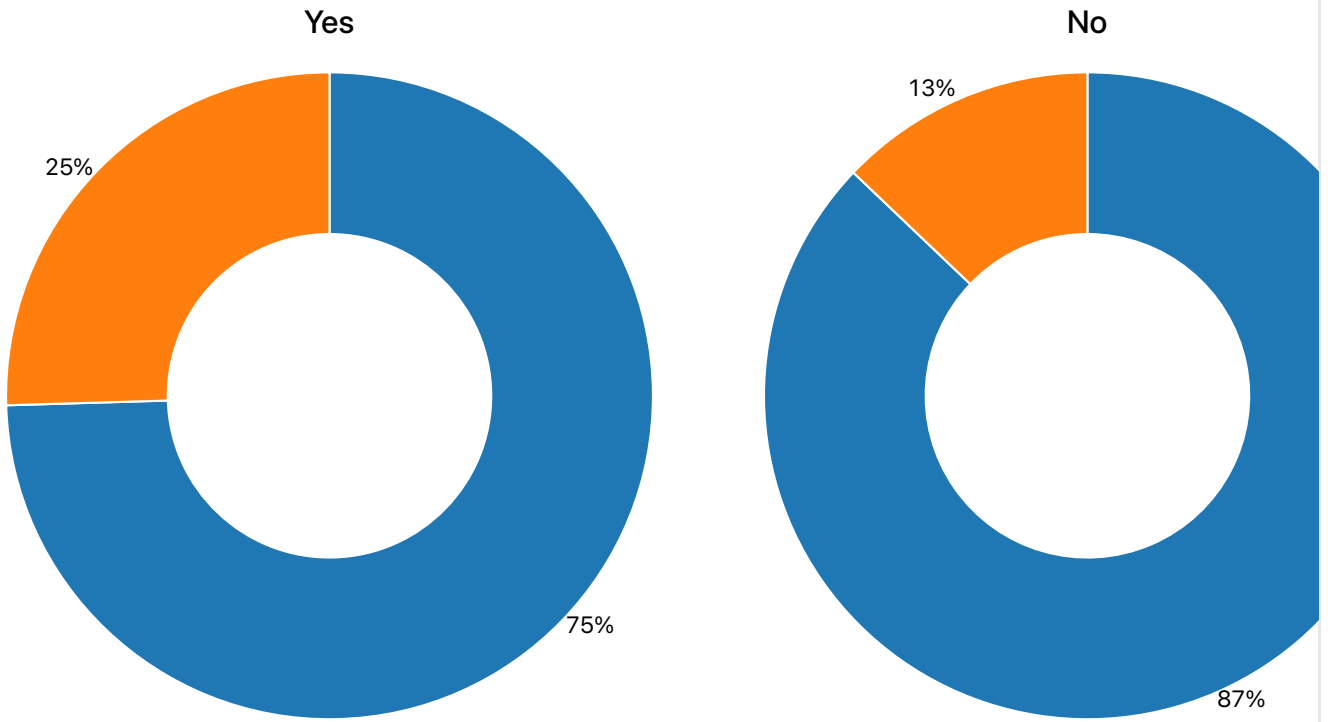


SeniorCitizen Distribution in Customer Attrition

```
%sql

select SeniorCitizen,count(SeniorCitizen), Churn
from TelecomData
group by Churn,SeniorCitizen;
```

Visualization



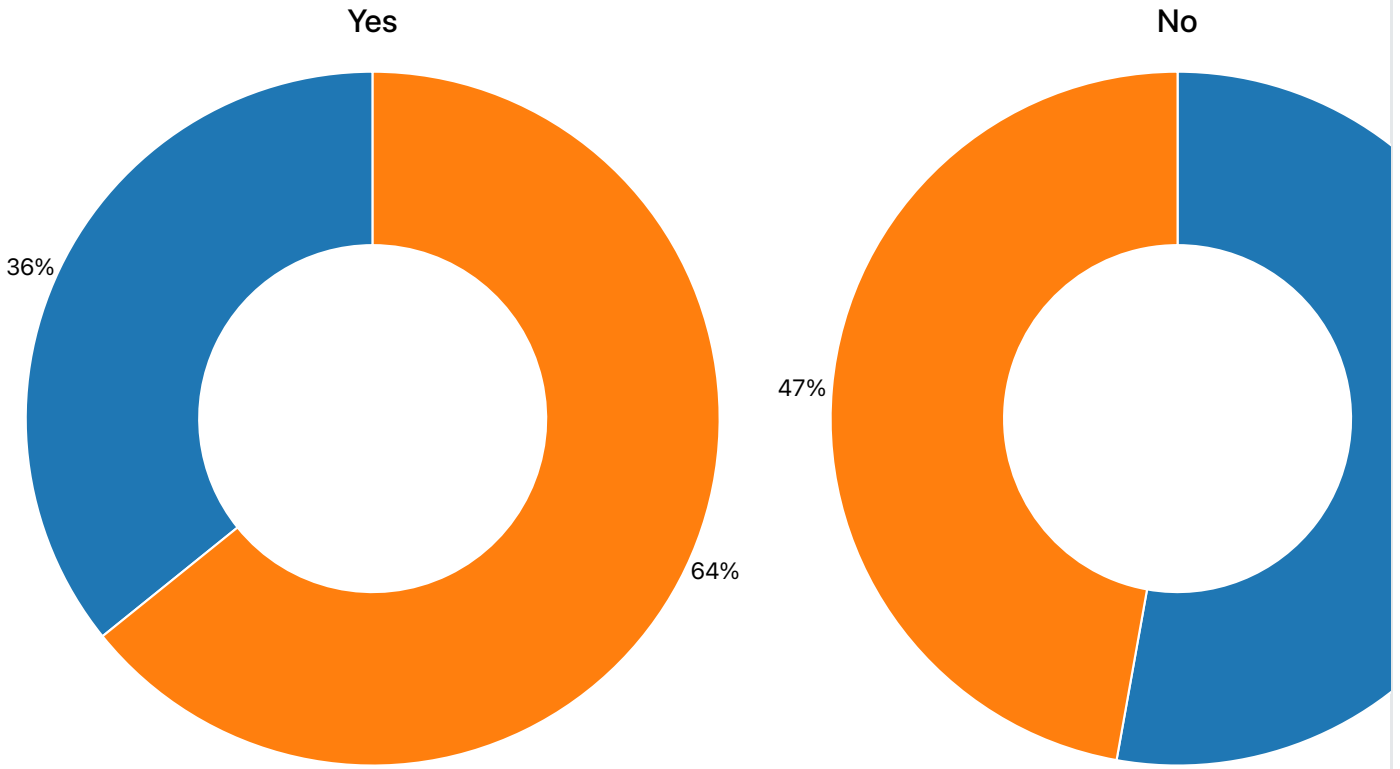
Showing all 4 rows.

Partner Distribution in Customer Attrition

```
%sql

select Partner,count(Partner), Churn
from TelecomData
group by Churn,Partner;
```

Visualization



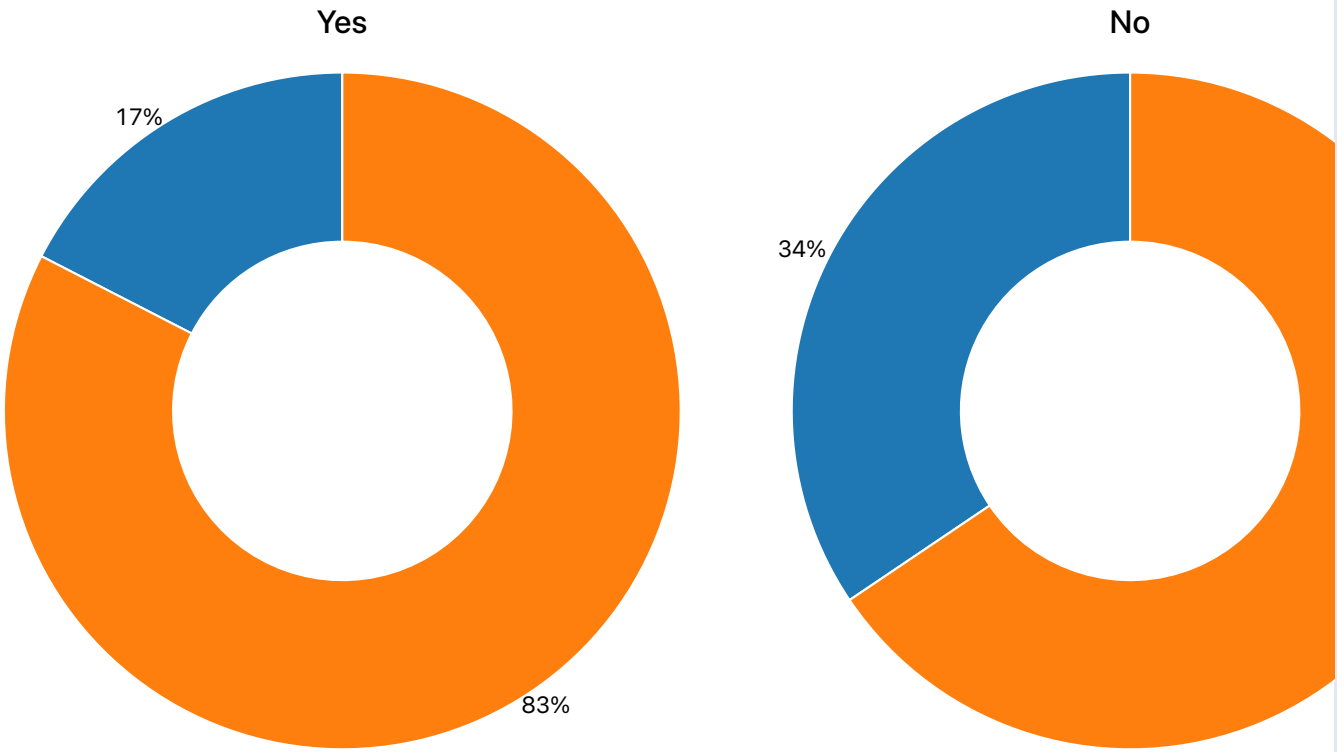
Showing all 4 rows.

Dependents Distribution in Customer Attrition

```
%sql

select Dependents,count(Dependents), Churn
from TelecomData
group by Churn,Dependents;
```

Visualization



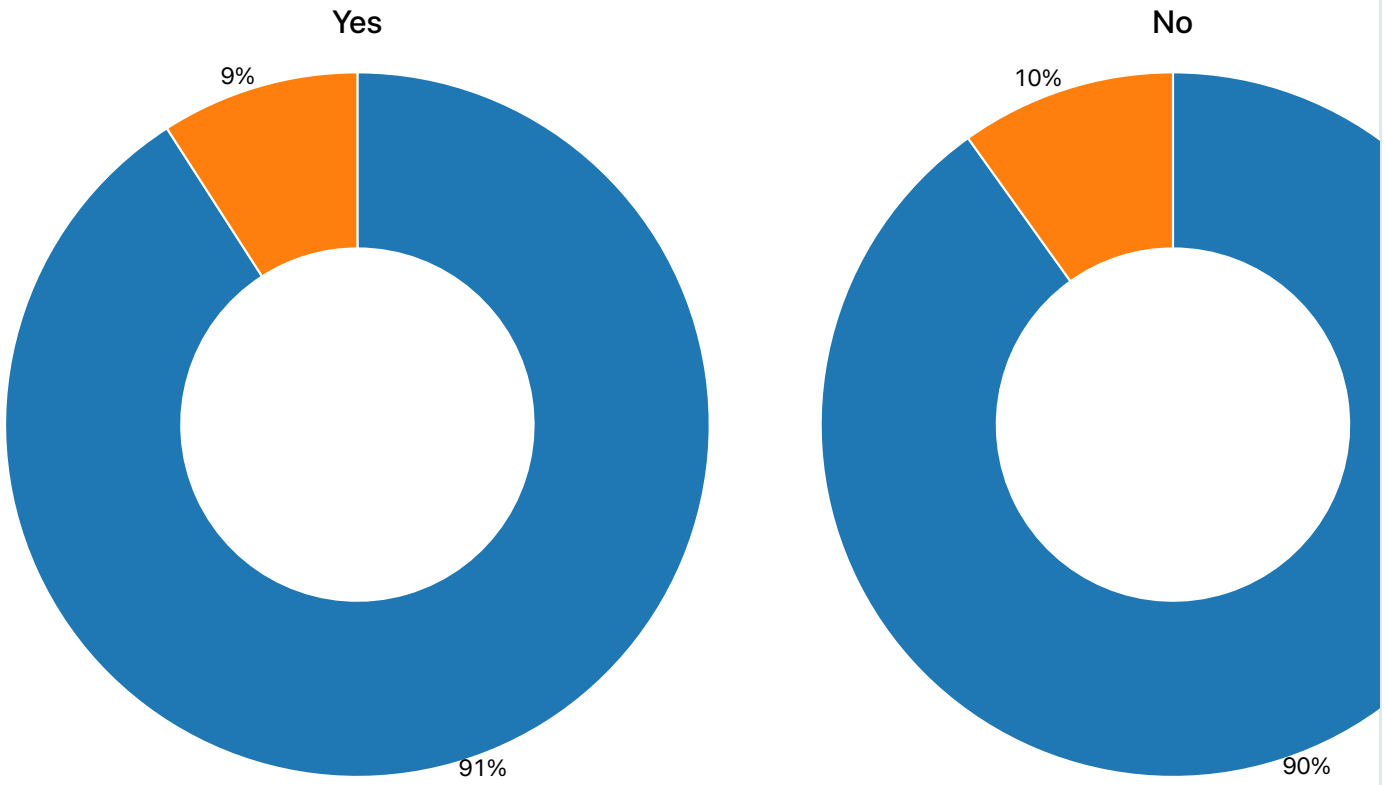
Showing all 4 rows.

PhoneService Distribution in Customer Attrition

```
%sql

select PhoneService,count(PhoneService), Churn
from TelecomData
group by Churn,PhoneService;
```

Visualization



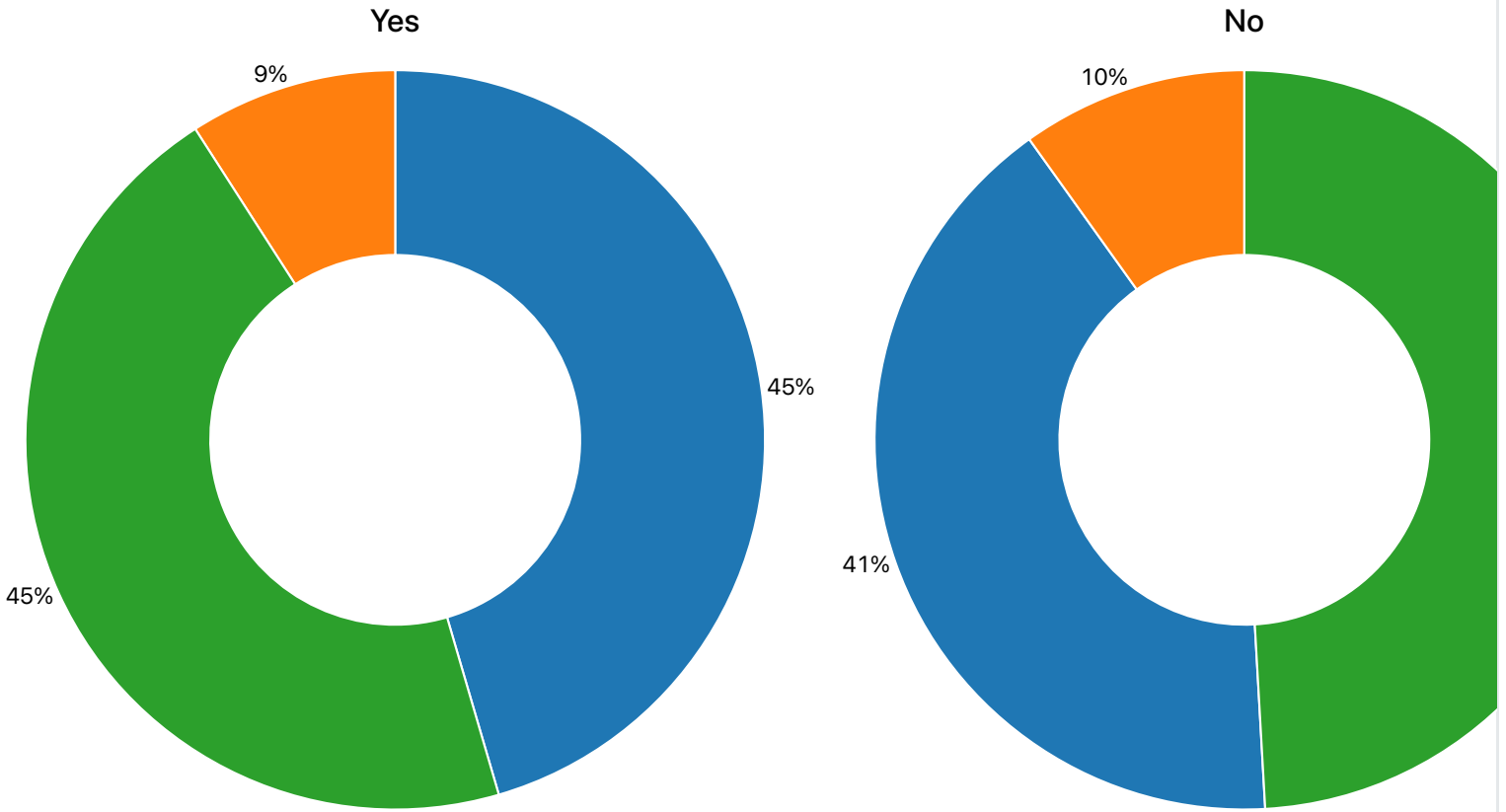
Showing all 4 rows.

MultipleLines Distribution in Customer Attrition

```
%sql

select MultipleLines,count(MultipleLines), Churn
from TelecomData
group by Churn,MultipleLines;
```

Visualization



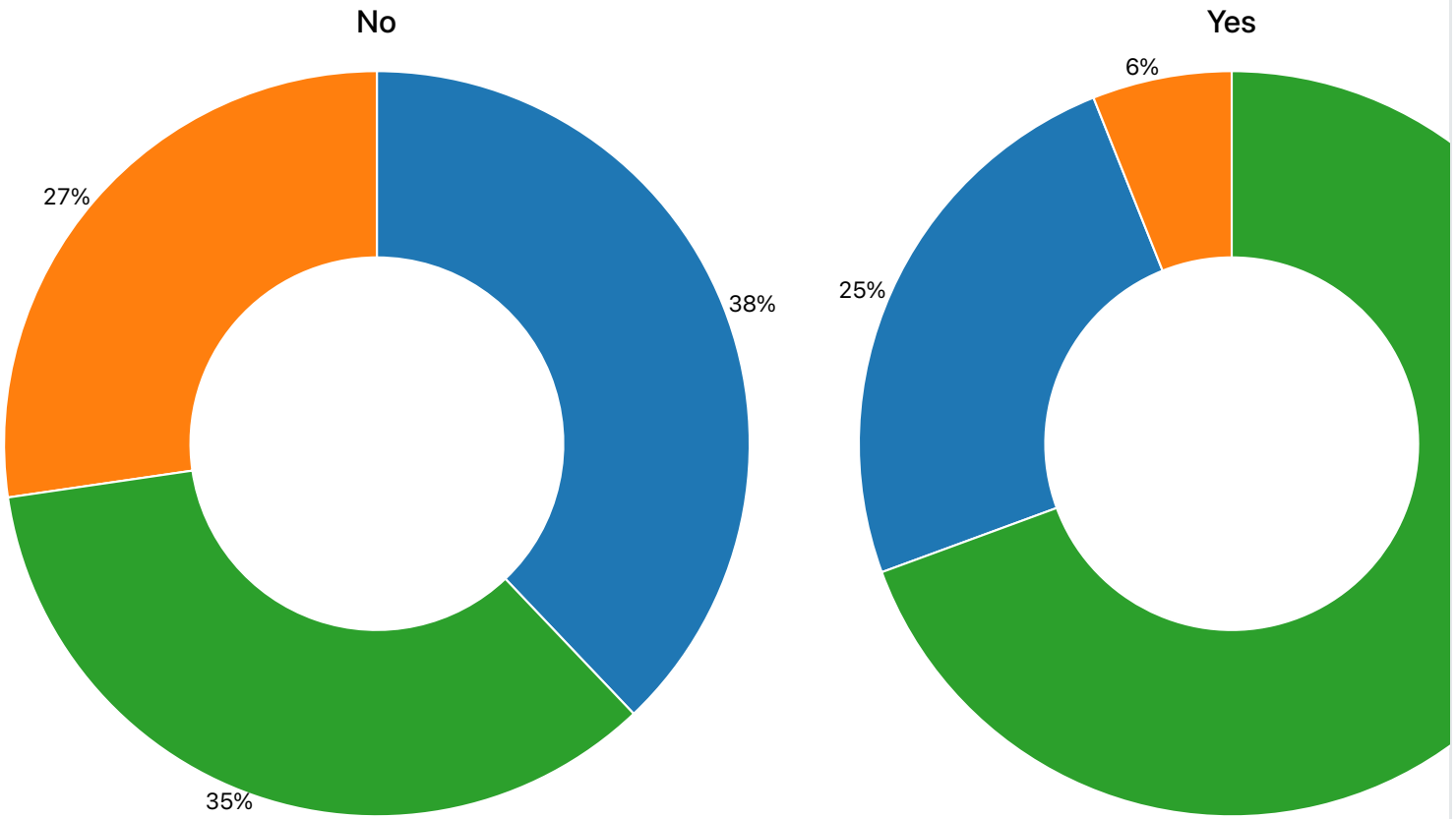
Showing all 6 rows.

InternetService Distribution in Customer Attrition

```
%sql

select InternetService,count(InternetService), Churn
from TelecomData
group by Churn,InternetService;
```

Visualization



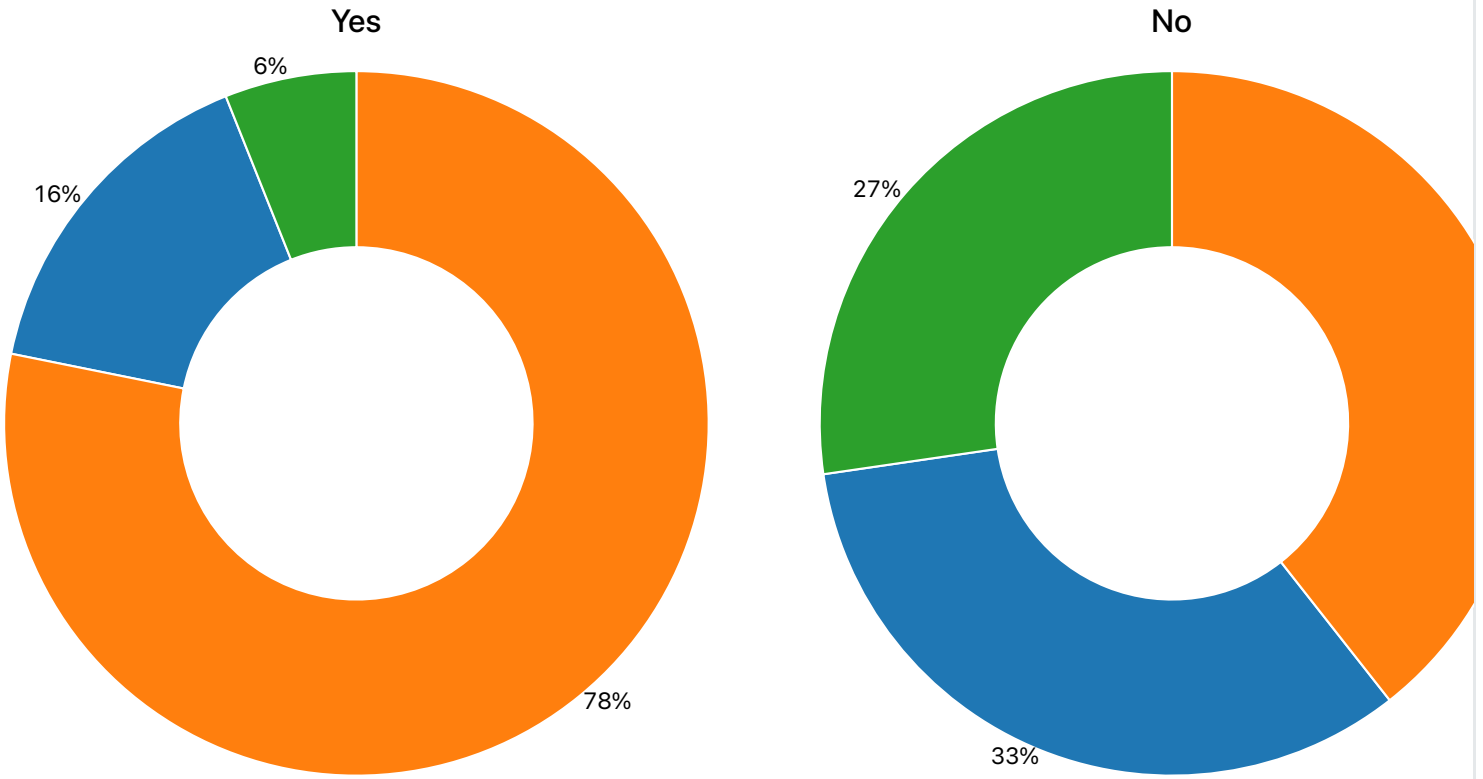
Showing all 6 rows.

OnlineSecurity Distribution in Customer Attrition

```
%sql

select OnlineSecurity,count(OnlineSecurity), Churn
from TelecomData
group by Churn,OnlineSecurity;
```

Visualization



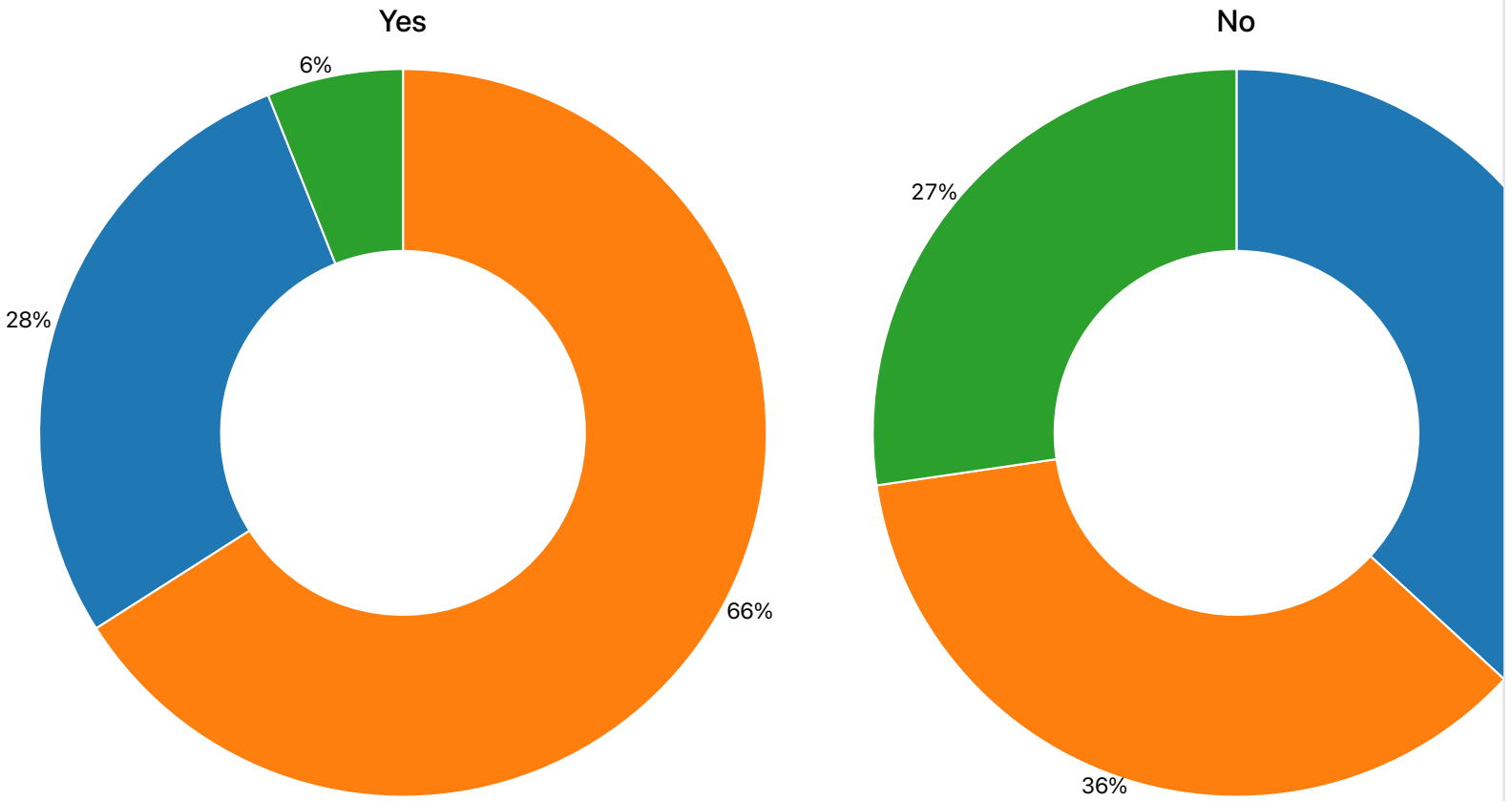
Showing all 6 rows.

OnlineBackup Distribution in Customer Attrition

```
%sql

select OnlineBackup,count(OnlineBackup), Churn
from TelecomData
group by Churn,OnlineBackup;
```

Visualization



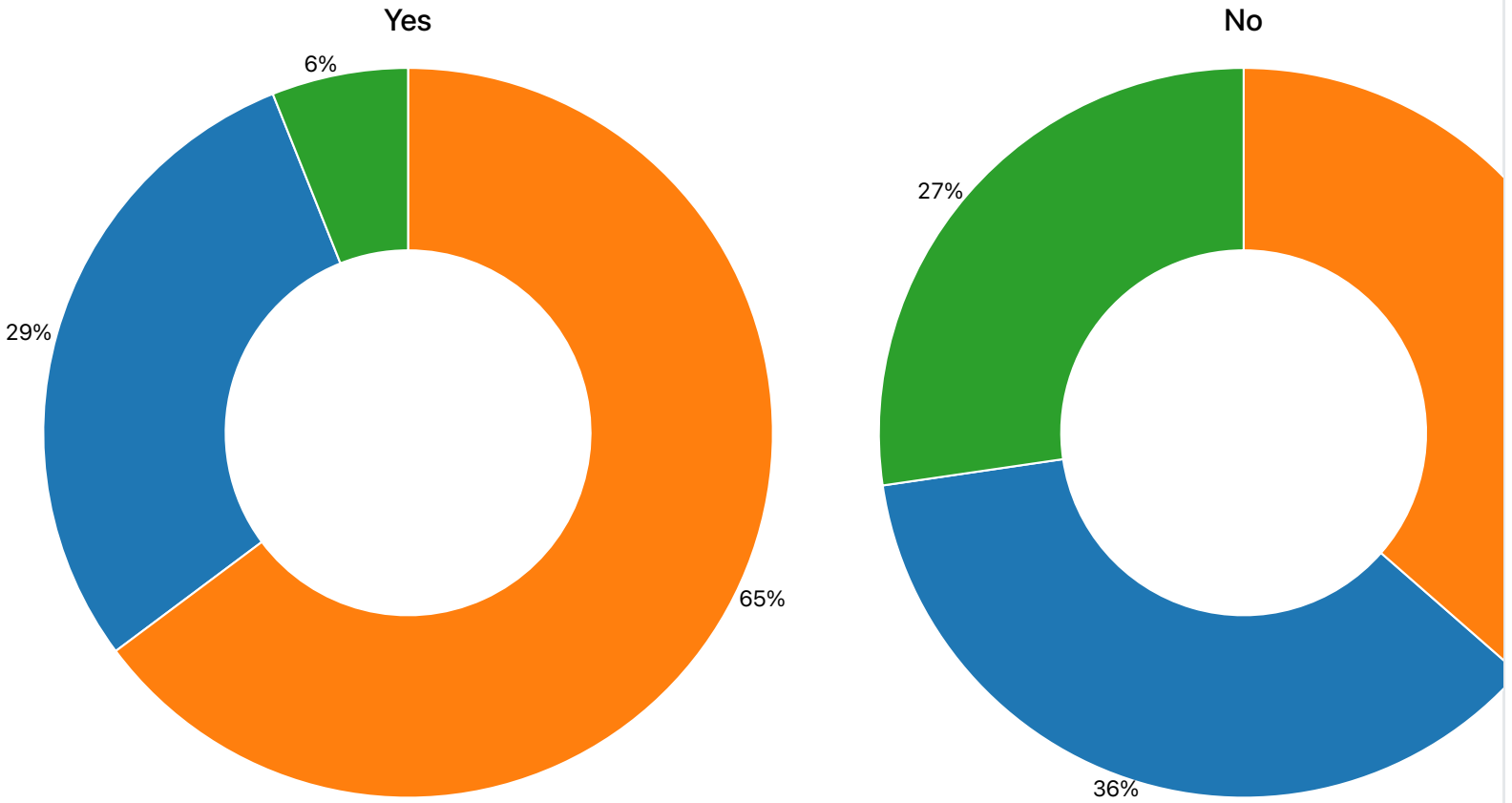
Showing all 6 rows.

DeviceProtection Distribution in Customer Attrition

```
%sql

select DeviceProtection,count(DeviceProtection), Churn
from TelecomData
group by Churn,DeviceProtection;
```

Visualization



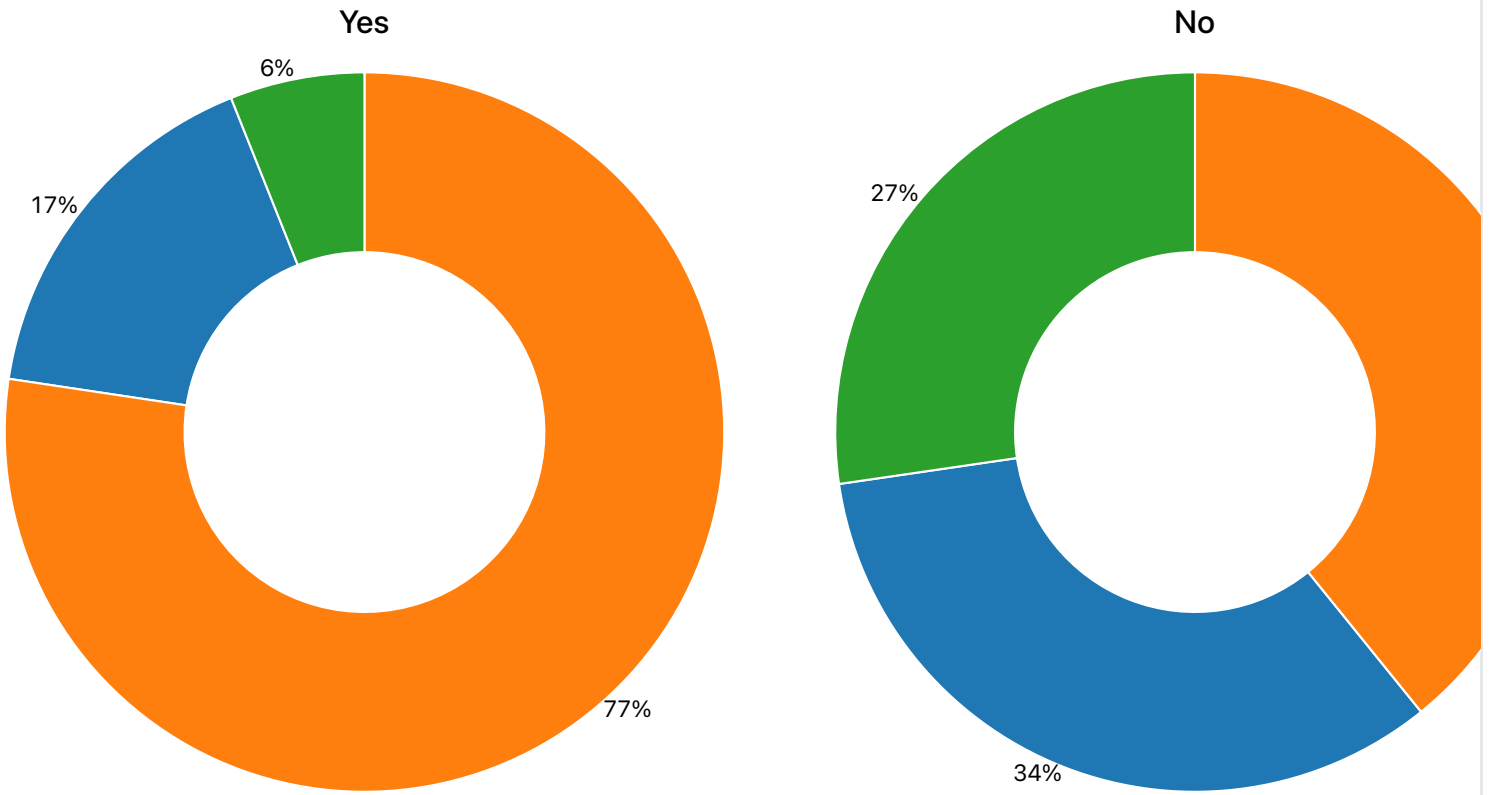
Showing all 6 rows.

TechSupport Distribution in Customer Attrition

```
%sql

select TechSupport,count(TechSupport), Churn
from TelecomData
group by Churn,TechSupport;
```

Visualization



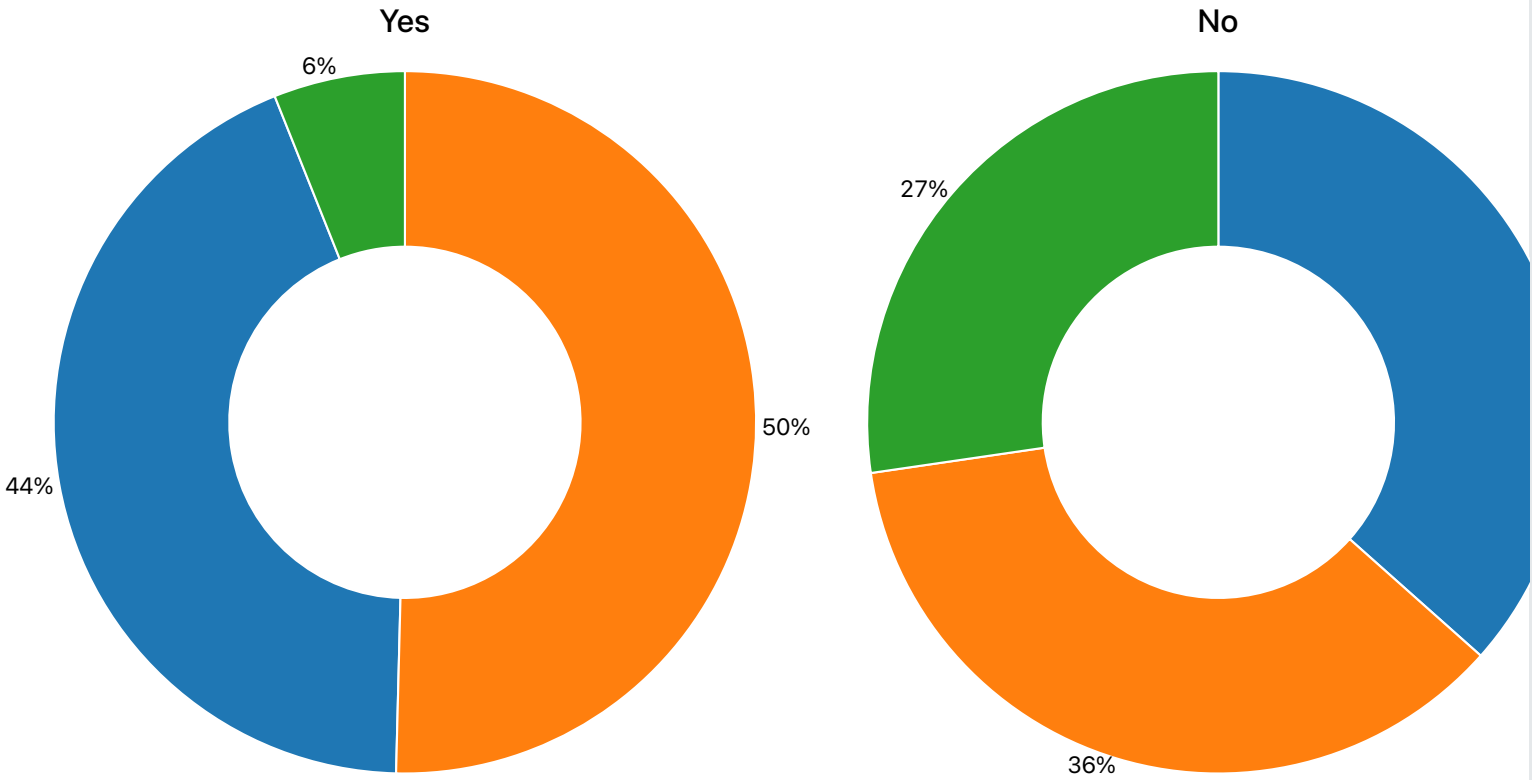
Showing all 6 rows.

StreamingTV Distribution in Customer Attrition

```
%sql

select StreamingTV,count(StreamingTV), Churn
from TelecomData
group by Churn,StreamingTV;
```

Visualization



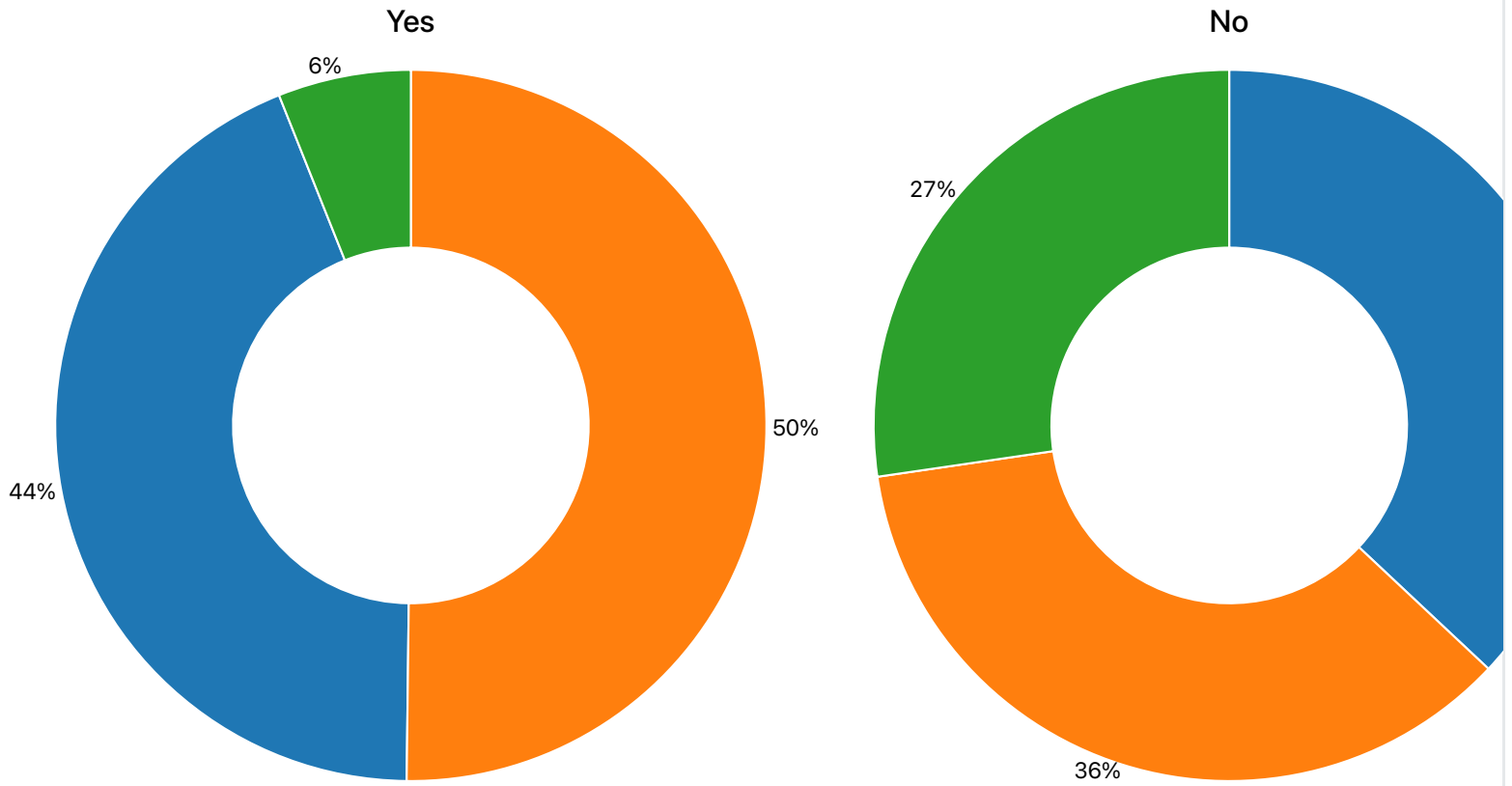
Showing all 6 rows.

StreamingMovies Distribution in Customer Attrition

```
%sql

select StreamingMovies,count(StreamingMovies), Churn
from TelecomData
group by Churn,StreamingMovies;
```

Visualization



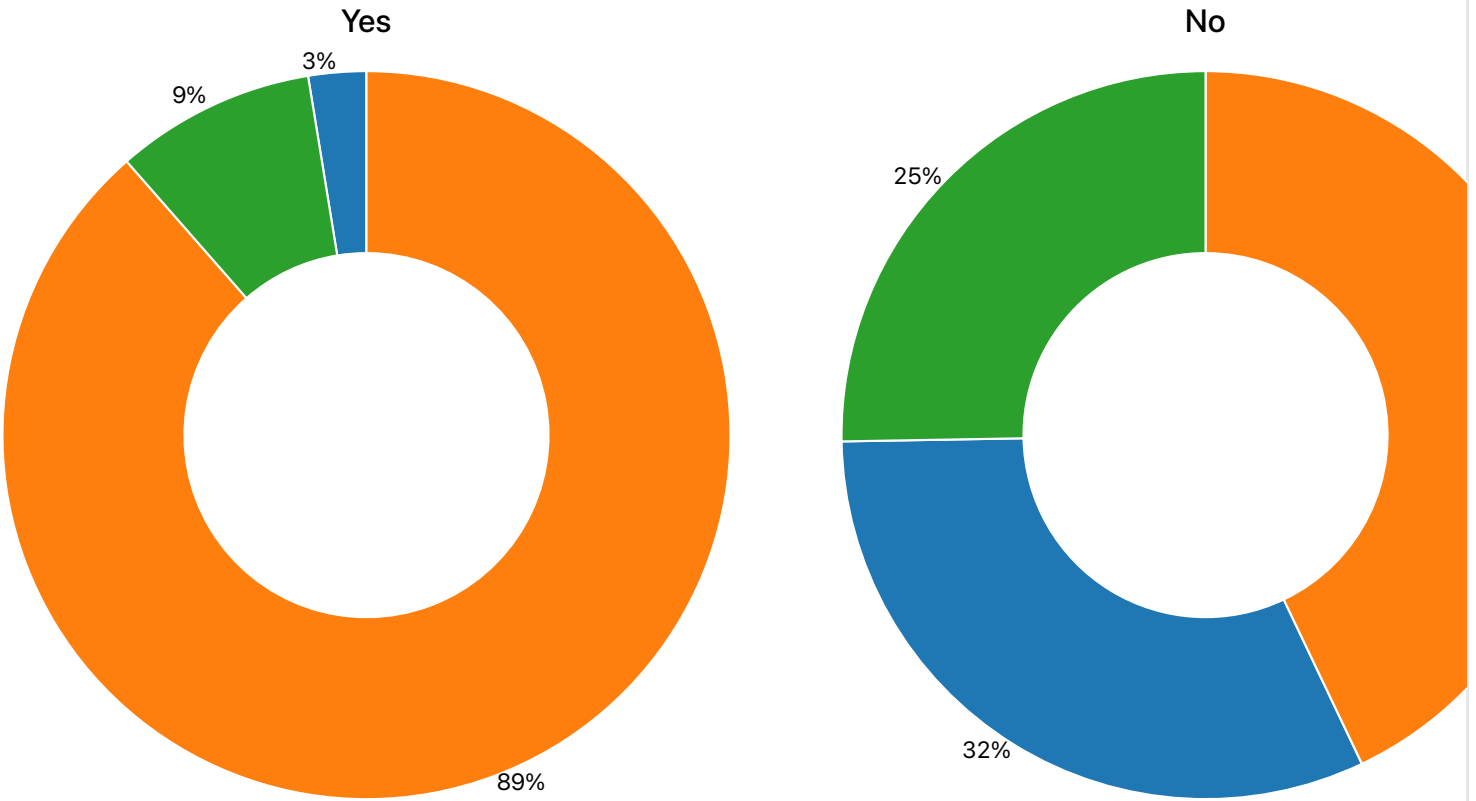
Showing all 6 rows.

Contract Distribution in Customer Attrition

```
%sql

select Contract,count(Contract), Churn
from TelecomData
group by Churn,Contract;
```

Visualization



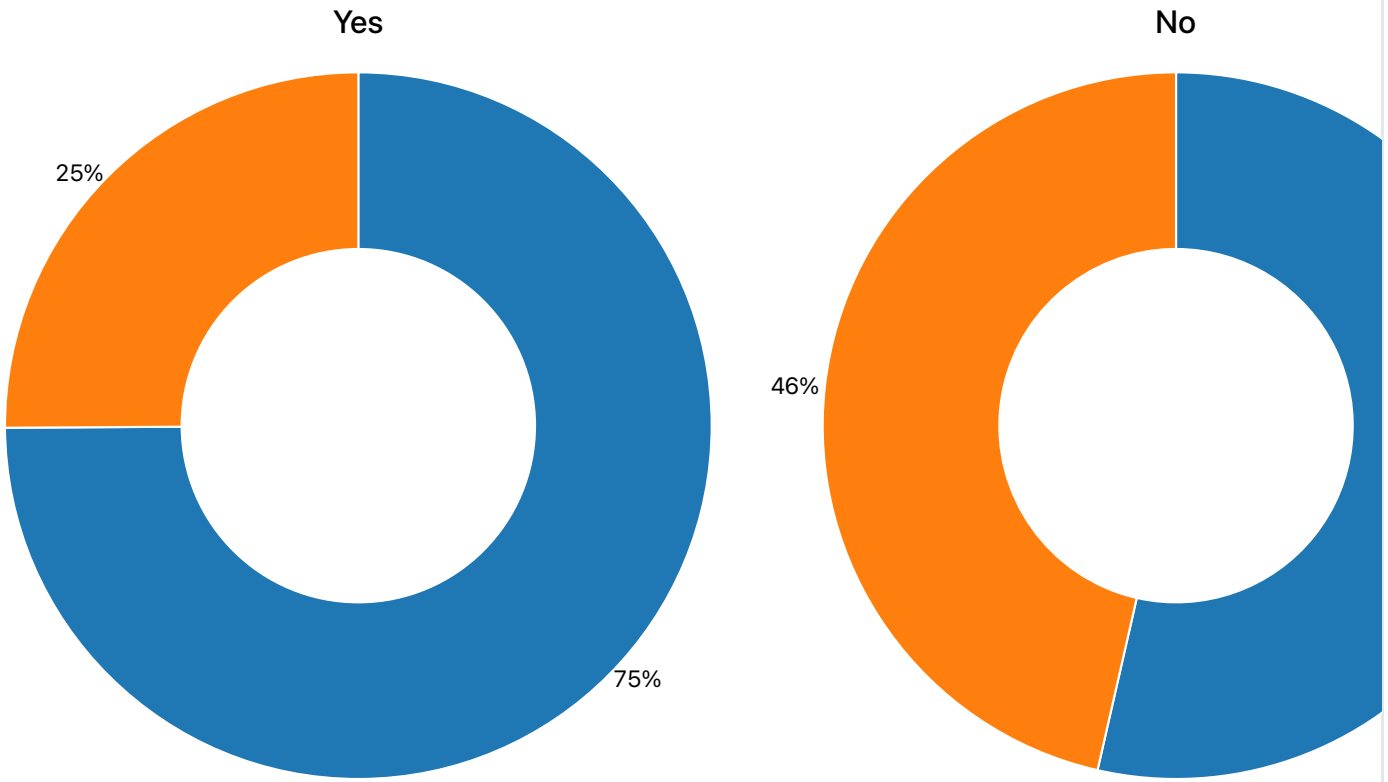
Showing all 6 rows.

PaperlessBilling Distribution in Customer Attrition

```
%sql

select PaperlessBilling,count(PaperlessBilling), Churn
from TelecomData
group by Churn,PaperlessBilling;
```

Visualization



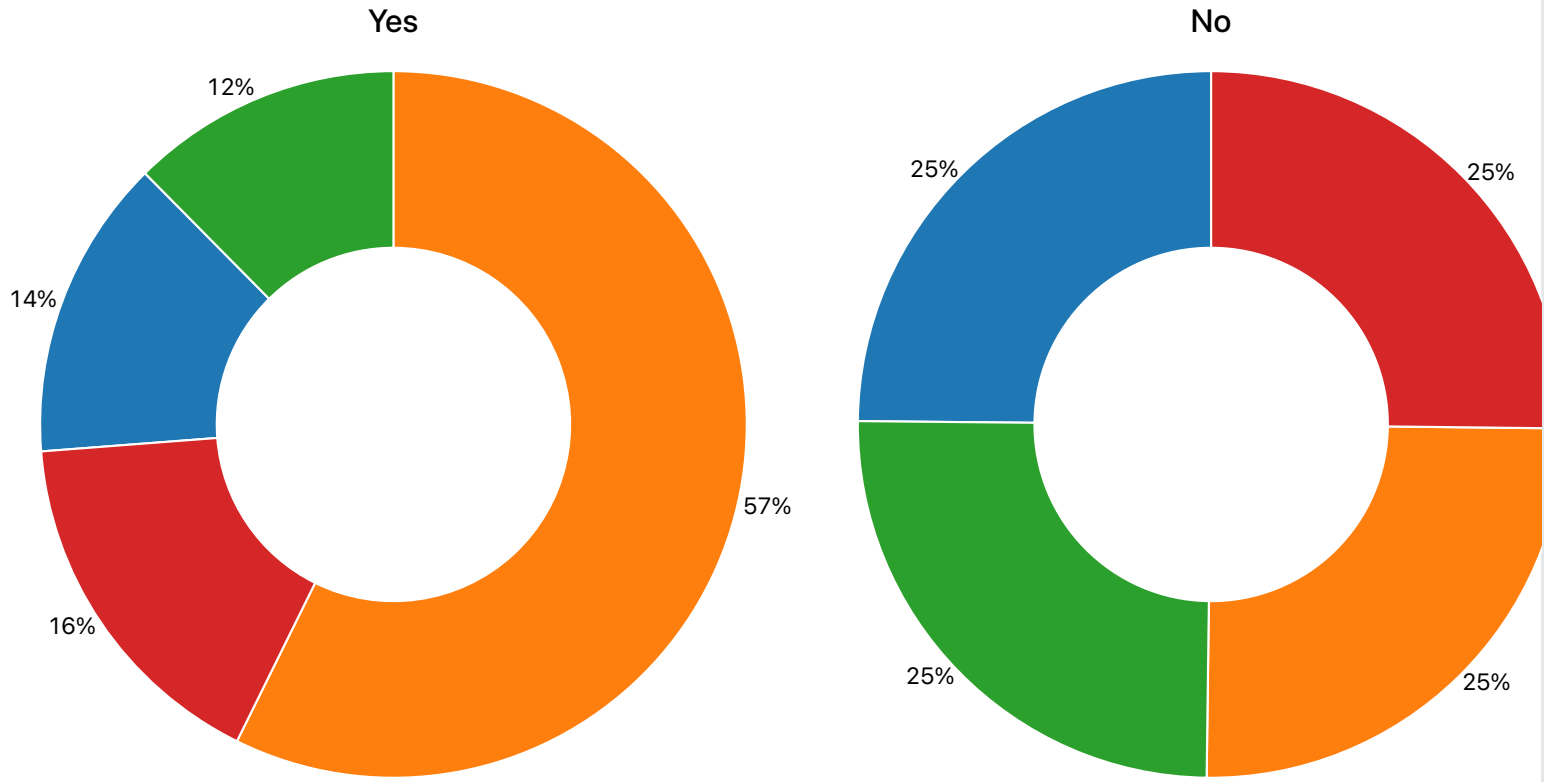
Showing all 4 rows.

PaymentMethod Distribution in Customer Attrition

```
%sql

select PaymentMethod,count(PaymentMethod), Churn
from TelecomData
group by Churn,PaymentMethod;
```

Visualization



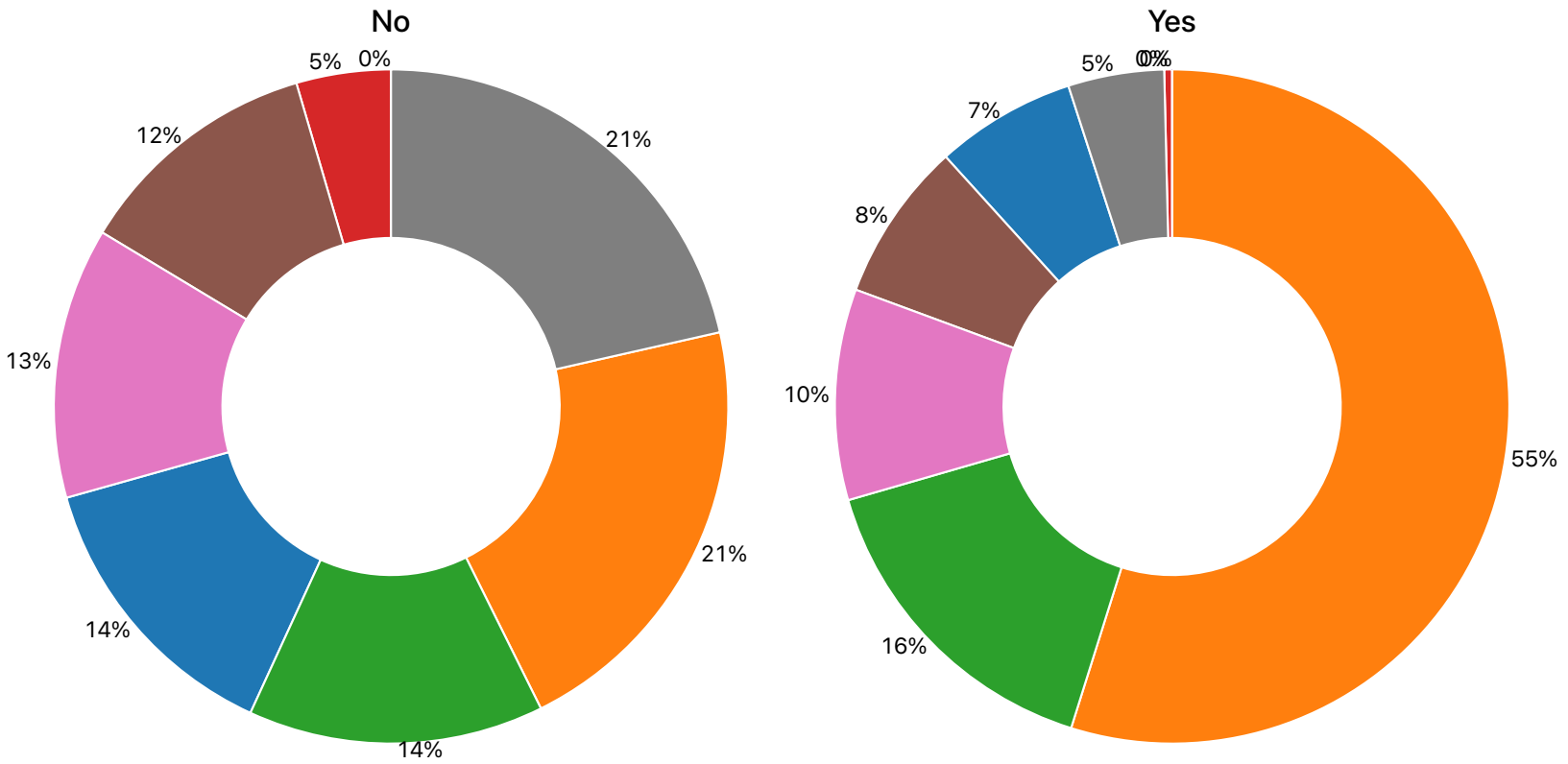
Showing all 8 rows.

Tenure Group Distribution in Customer Attrition

```
%sql

select cast ((TotalCharges/MonthlyCharges)/12 as Int) as Tenure, count(cast ((TotalCharges/MonthlyCharges)/12 as Int)),
Churn
from TelecomData
group by Churn, cast ((TotalCharges/MonthlyCharges)/12 as Int);
```

Visualization

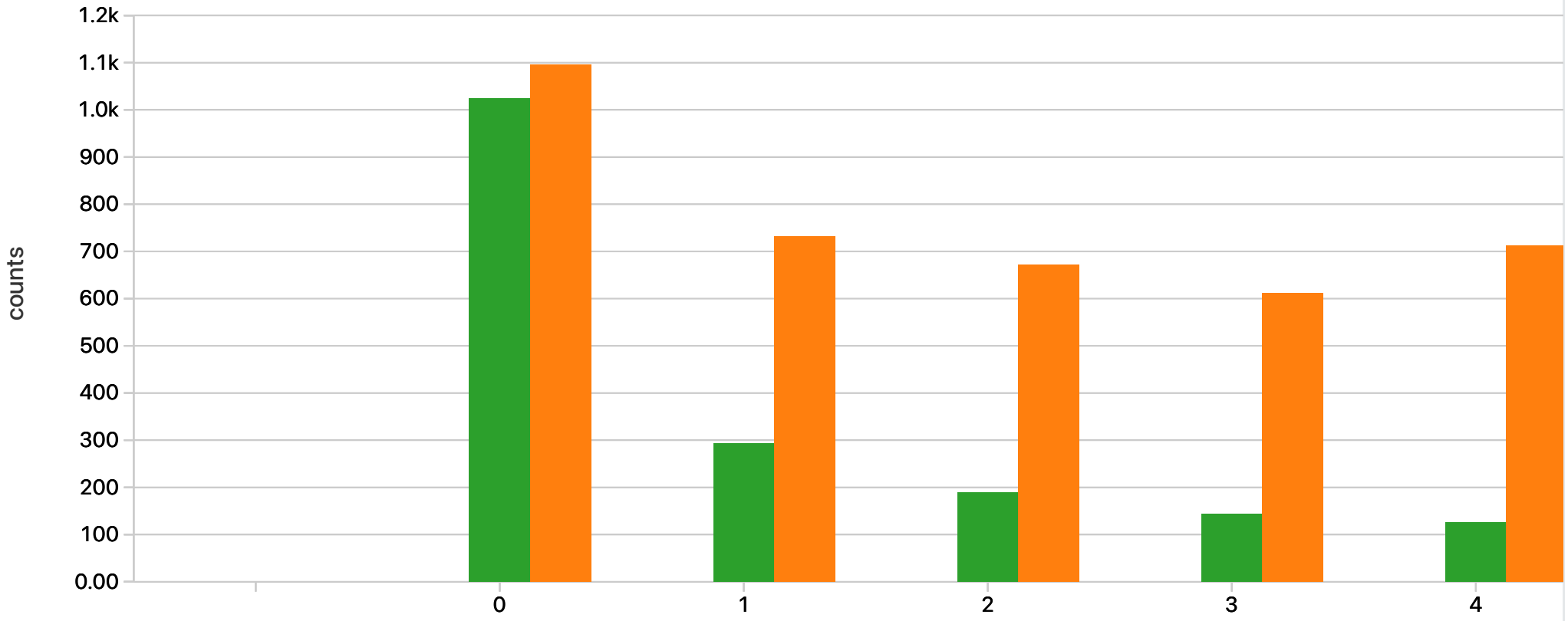


Tenure Group Distribution in Customer Attrition

```
%sql

select cast ((TotalCharges/MonthlyCharges)/12 as Int) as Tenure, count(cast ((TotalCharges/MonthlyCharges)/12 as Int)) as
counts, Churn
from TelecomData
group by Churn,cast ((TotalCharges/MonthlyCharges)/12 as Int)
order by Tenure;
```

Visualization



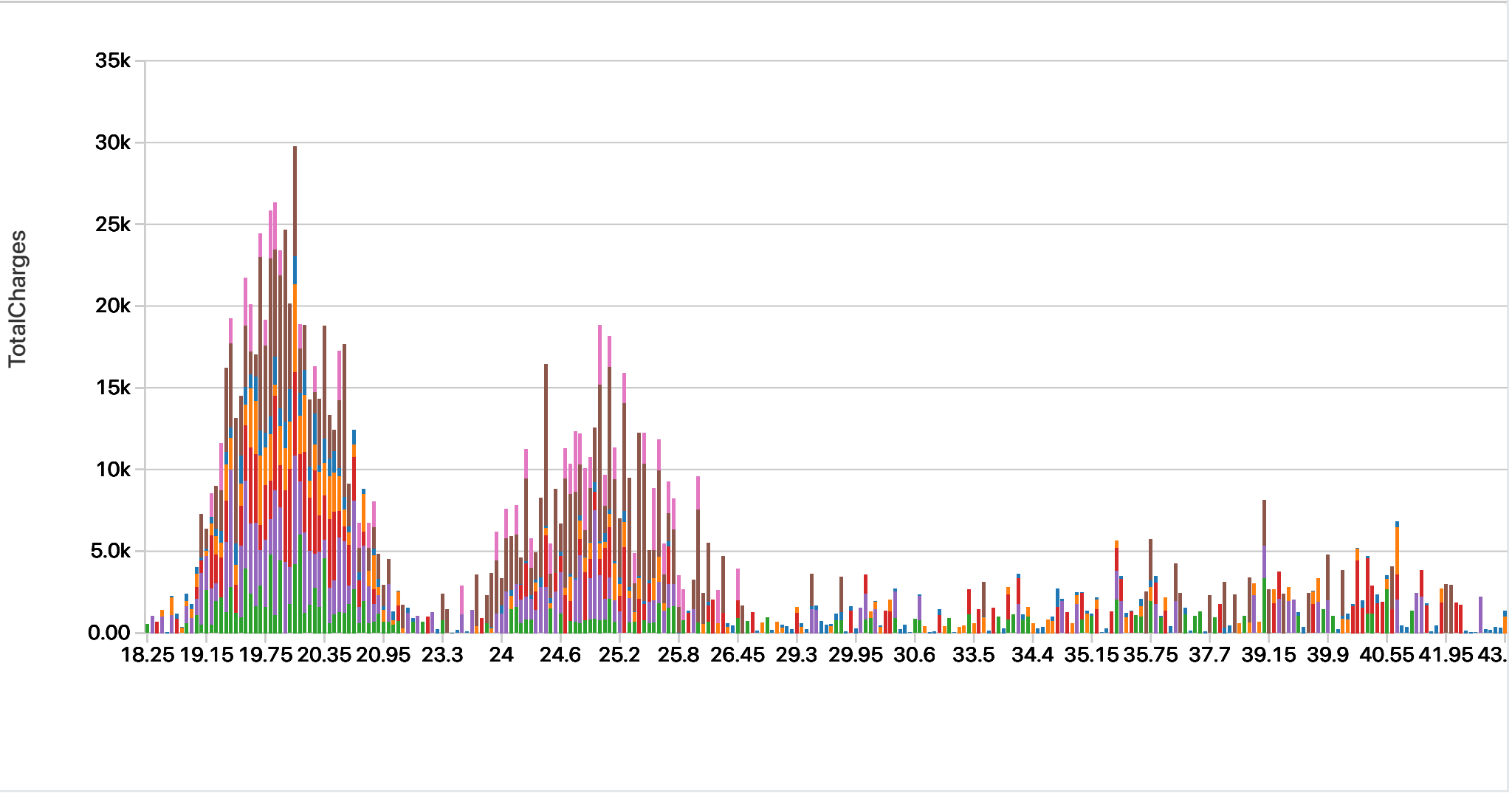
Showing all 15 rows.

Monthly Charges & Total Charges by Tenure group

```
%sql

select TotalCharges, MonthlyCharges, cast ((TotalCharges/MonthlyCharges)/12 as Int) as Tenure
from TelecomData;
```

Visualization



Creating a Classification Model

In this Project, you will implement a classification model **(Logistic Regression)** that uses features of telecom details of customer and we will predict it is Churn or Not

Import Spark SQL and Spark ML Libraries

First, import the libraries you will need:

```
import org.apache.spark.sql.types._
import org.apache.spark.sql.functions._

import org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.ml.feature.VectorAssembler
```

```
import org.apache.spark.sql.types._
import org.apache.spark.sql.functions._
import org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.ml.feature.VectorAssembler
```

Prepare the Training Data

To train the classification model, you need a training data set that includes a vector of numeric features, and a label column. In this project, you will use the **VectorAssembler** class to transform the feature columns into a vector, and then rename the **Churn** column to **label**.

VectorAssembler()

VectorAssembler(): is a transformer that combines a given list of columns into a single vector column. It is useful for combining raw features and features generated by different feature transformers into a single feature vector, in order to train ML models like logistic regression and decision trees.

VectorAssembler accepts the following input column types: **all numeric types, boolean type, and vector type.**

In each row, the **values of the input columns will be concatenated into a vector** in the specified order.

```
%scala

var StringfeatureCol = Array("customerID", "gender", "Partner", "Dependents", "PhoneService", "MultipleLines",
"InternetService", "OnlineSecurity", "OnlineBackup", "DeviceProtection", "TechSupport", "StreamingTV", "StreamingMovies",
"Contract", "PaperlessBilling", "PaymentMethod", "Churn")

StringfeatureCol: Array[String] = Array(customerID, gender, Partner, Dependents, PhoneService, MultipleLines, InternetServ
ice, OnlineSecurity, OnlineBackup, DeviceProtection, TechSupport, StreamingTV, StreamingMovies, Contract, PaperlessBilling
, PaymentMethod, Churn)
```

StringIndexer

StringIndexer encodes a string column of labels to a column of label indices.

Example of StringIndexer

```
import org.apache.spark.ml.feature.StringIndexer

val df = spark.createDataFrame(
  Seq((0, "a"), (1, "b"), (2, "c"), (3, "a"), (4, "a"), (5, "c"))
).toDF("id", "category")

val indexer = new StringIndexer()
  .setInputCol("category")
  .setOutputCol("categoryIndex")

val indexed = indexer.fit(df).transform(df)

display(indexed)
```

Table

	id	category	categoryIndex
1	0	a	0
2	1	b	2
3	2	c	1
4	3	a	0
5	4	a	0
6	5	c	1

Showing all 6 rows.

Define the Pipeline

A predictive model often requires multiple stages of feature preparation.

A pipeline consists of a series of *transformer* and *estimator* stages that typically prepare a DataFrame for modeling and then train a predictive model.

In this case, you will create a pipeline with stages:

- A **StringIndexer** estimator that converts string values to indexes for categorical features
- A **VectorAssembler** that combines categorical features into a single vector

```
%scala

import org.apache.spark.ml.attribute.Attribute
import org.apache.spark.ml.feature.{IndexToString, StringIndexer}
import org.apache.spark.ml.{Pipeline, PipelineModel}

val indexers = StringfeatureCol.map { colName =>
  new StringIndexer().setInputCol(colName).setHandleInvalid("skip").setOutputCol(colName + "_indexed")
}

val pipeline = new Pipeline()
                    .setStages(indexers)

val TelDF = pipeline.fit(telecomDF).transform(telecomDF)
```

```
import org.apache.spark.ml.attribute.Attribute
import org.apache.spark.ml.feature.{IndexToString, StringIndexer}
import org.apache.spark.ml.{Pipeline, PipelineModel}
```

```
indexers: Array[org.apache.spark.ml.feature.StringIndexer] = Array(strIdx_ba8aa3b65f3d, strIdx_b42eb4120c4b, strIdx_1231e69f817d, strIdx_1327ee19a4fe, strIdx_c544d8c415bd, strIdx_ed24432d5c81, strIdx_b840697f17ff, strIdx_1e1fd0ad5f9d, strIdx_6283369b4f15, strIdx_98507bc4119c, strIdx_324152c4251e, strIdx_2d30a5e0dac7, strIdx_03d280cb5f44, strIdx_46c7860964a5, strIdx_40805db919e5, strIdx_9c3ad819eacd, strIdx_5f15159ff52f)
pipeline: org.apache.spark.ml.Pipeline = pipeline_3b2599d7d066
TelDF: org.apache.spark.sql.DataFrame = [customerID: string, gender: string ... 36 more fields]
```

Printing Schema

```
TelDF.printSchema()
```

```
root
 |-- customerID: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- SeniorCitizen: integer (nullable = true)
 |-- Partner: string (nullable = true)
 |-- Dependents: string (nullable = true)
 |-- tenure: integer (nullable = true)
 |-- PhoneService: string (nullable = true)
 |-- MultipleLines: string (nullable = true)
 |-- InternetService: string (nullable = true)
 |-- OnlineSecurity: string (nullable = true)
 |-- OnlineBackup: string (nullable = true)
 |-- DeviceProtection: string (nullable = true)
 |-- TechSupport: string (nullable = true)
 |-- StreamingTV: string (nullable = true)
 |-- StreamingMovies: string (nullable = true)
 |-- Contract: string (nullable = true)
 |-- PaperlessBilling: string (nullable = true)
 |-- PaymentMethod: string (nullable = true)
 |-- MonthlyCharges: double (nullable = true)
 |-- TotalCharges: double (nullable = true)
```

Data Display

```
TelDF.show()
```

```
+----------+------+-------------+-------+----------+------+------------+----------------+---------------+----------------
--+------------------+------------------+------------------+------------------+------------------+-------------+----
-----------+--------------------+-------------+------------+-----+------------------+-------------+---------------+----
-------------+--------------------+---------------------+----------------------+----------------------+----------------
---+----------------------+------------------+------------------+----------------------+---------------+-----------
-----------+--------------------+-------------+
|customerID|gender|SeniorCitizen|Partner|Dependents|tenure|PhoneService|   MultipleLines|InternetService|     OnlineSecuri
ty|       OnlineBackup|   DeviceProtection|        TechSupport|        StreamingTV|    StreamingMovies|      Contract|Pape
rlessBilling|       PaymentMethod|MonthlyCharges|TotalCharges|Churn|customerID_indexed|gender_indexed|Partner_indexed|Depe
ndents_indexed|PhoneService_indexed|MultipleLines_indexed|InternetService_indexed|OnlineSecurity_indexed|OnlineBackup_inde
xed|DeviceProtection_indexed|TechSupport_indexed|StreamingTV_indexed|StreamingMovies_indexed|Contract_indexed|PaperlessBil
ling_indexed|PaymentMethod_indexed|Churn_indexed|
+----------+------+-------------+-------+----------+------+------------+----------------+---------------+----------------
--+------------------+------------------+------------------+------------------+------------------+-------------+----
-----------+--------------------+-------------+------------+-----+------------------+-------------+---------------+----
-------------+--------------------+---------------------+----------------------+----------------------+----------------
---+----------------------+------------------+------------------+----------------------+---------------+-----------
-----------+--------------------+-------------+
|7590-VHVEG|Female|            0|    Yes|        No|     1|          No|No phone service|            DSL|
No|                Yes|                 No|                 No|                 No|                No|Month-to-month|
Yes|    Electronic check|         29.85|       29.85|   No|            1952.0|           1.0|            1.0|
```

Count of Records

```
%scala

TelDF.count()
```

```
res9: Long = 7040
```

Split the Data

It is common practice when building supervised machine learning models to split the source data, using some of it to train the model and reserving some to test the trained model. In this project, you will use 70% of the data for training, and reserve 30% for testing.

```
%scala

val splits = TelDF.randomSplit(Array(0.7, 0.3))
val train = splits(0)
val test = splits(1)
val train_rows = train.count()
val test_rows = test.count()
println("Training Rows: " + train_rows + " Testing Rows: " + test_rows)
```

```
Training Rows: 4835 Testing Rows: 2197
splits: Array[org.apache.spark.sql.Dataset[org.apache.spark.sql.Row]] = Array([customerID: string, gender: string ... 36 m
ore fields], [customerID: string, gender: string ... 36 more fields])
train: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [customerID: string, gender: string ... 36 more fields]
test: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [customerID: string, gender: string ... 36 more fields]
train_rows: Long = 4835
test_rows: Long = 2197
```

```
%scala

import org.apache.spark.ml.feature.VectorAssembler

val assembler = new VectorAssembler().setInputCols(Array("customerID_indexed", "gender_indexed", "SeniorCitizen",
"Partner_indexed", "Dependents_indexed", "PhoneService_indexed", "MultipleLines_indexed", "InternetService_indexed",
"OnlineSecurity_indexed", "OnlineBackup_indexed", "DeviceProtection_indexed", "TechSupport_indexed", "StreamingTV_indexed",
"StreamingMovies_indexed", "Contract_indexed", "PaperlessBilling_indexed", "PaymentMethod_indexed", "tenure",
"MonthlyCharges", "TotalCharges" )).setOutputCol("features")
val training = assembler.transform(train).select($"features", $"Churn_indexed".alias("label"))
training.show()
```

```
+--------------------+-----+
|            features|label|
+--------------------+-----+
|(20,[0,10,17,18,1...|  1.0|
|(20,[0,1,4,7,11,1...|  0.0|
|[4555.0,1.0,0.0,1...|  0.0|
|[594.0,1.0,0.0,1....|  0.0|
|[1531.0,1.0,0.0,0...|  0.0|
|[1518.0,1.0,0.0,1...|  0.0|
|(20,[0,1,2,6,17,1...|  0.0|
|(20,[0,2,7,8,10,1...|  1.0|
|(20,[0,2,5,6,7,17...|  1.0|
|[2711.0,1.0,1.0,1...|  0.0|
|(20,[0,1,3,6,9,10...|  1.0|
|[2481.0,1.0,0.0,0...|  0.0|
|[3164.0,1.0,0.0,1...|  1.0|
|[3671.0,1.0,0.0,1...|  1.0|
|[4521.0,0.0,0.0,1...|  0.0|
|(20,[0,3,4,12,13,...|  0.0|
|[4663.0,1.0,0.0,1...|  0.0|
|[4566.0,1.0,0.0,0...|  0.0|
```

Train a Classification Model (Logistic Regression)

Next, you need to train a Classification Model using the training data. To do this, create an instance of the Logistic regression algorithm you want to use and use its **fit** method to train a model based on the training DataFrame. In this Project, you will use a *Logistic Regression* algorithm

```
%scala

import org.apache.spark.ml.classification.LogisticRegression

val lr = new LogisticRegression().setLabelCol("label").setFeaturesCol("features").setMaxIter(10).setRegParam(0.3)
val model = lr.fit(training)
println("Model Trained!")
```

```
Model Trained!
import org.apache.spark.ml.classification.LogisticRegression
lr: org.apache.spark.ml.classification.LogisticRegression = logreg_ecd57c79a4d4
model: org.apache.spark.ml.classification.LogisticRegressionModel = LogisticRegressionModel: uid = logreg_ecd57c79a4d4, nu
mClasses = 2, numFeatures = 20
```

Prepare the Testing Data

Now that you have a trained model, you can test it using the testing data you reserved previously. First, you need to prepare the testing data in the same way as you did the training data by transforming the feature columns into a vector. This time you'll rename the **Churn_indexed** column to **trueLabel**.

```
%scala

val testing = assembler.transform(test).select($"features", $"Churn_indexed".alias("trueLabel"))
testing.show()
```

```
+--------------------+---------+
|            features|trueLabel|
+--------------------+---------+
|[1592.0,1.0,0.0,1...|      0.0|
|(20,[0,6,7,13,15,...|      0.0|
|(20,[0,2,3,9,10,1...|      1.0|
|(20,[0,1,2,3,11,1...|      1.0|
|[4096.0,1.0,1.0,1...|      0.0|
|(20,[0,3,6,8,11,1...|      0.0|
|(20,[0,1,2,7,8,17...|      0.0|
|[5827.0,0.0,0.0,0...|      0.0|
|(20,[0,1,3,17,18,...|      0.0|
|(20,[0,1,6,8,9,10...|      0.0|
|[3584.0,1.0,0.0,1...|      0.0|
|(20,[0,1,3,4,6,12...|      0.0|
|[5389.0,1.0,0.0,1...|      0.0|
|[5340.0,0.0,0.0,0...|      0.0|
|[4165.0,1.0,0.0,1...|      0.0|
|[5414.0,0.0,0.0,1...|      0.0|
|[4530.0,1.0,0.0,0...|      0.0|
|(20,[0,1,3,6,8,9,...|      0.0|
```

Test the Model

Now you're ready to use the **transform** method of the model to generate some predictions. But in this case you are using the test data which includes a known true label value, so you can compare the predicted Churn.

```
%scala

val prediction = model.transform(testing)
val predicted = prediction.select("features", "prediction", "trueLabel")
predicted.show(200)
```

```
+--------------------+----------+---------+
|            features|prediction|trueLabel|
+--------------------+----------+---------+
|[1592.0,1.0,0.0,1...|       0.0|      0.0|
|(20,[0,6,7,13,15,...|       0.0|      0.0|
|(20,[0,2,3,9,10,1...|       0.0|      1.0|
|(20,[0,1,2,3,11,1...|       1.0|      1.0|
|[4096.0,1.0,1.0,1...|       0.0|      0.0|
|(20,[0,3,6,8,11,1...|       0.0|      0.0|
|(20,[0,1,2,7,8,17...|       0.0|      0.0|
|[5827.0,0.0,0.0,0...|       0.0|      0.0|
|(20,[0,1,3,17,18,...|       1.0|      0.0|
|(20,[0,1,6,8,9,10...|       0.0|      0.0|
|[3584.0,1.0,0.0,1...|       0.0|      0.0|
|(20,[0,1,3,4,6,12...|       0.0|      0.0|
|[5389.0,1.0,0.0,1...|       0.0|      0.0|
|[5340.0,0.0,0.0,0...|       0.0|      0.0|
|[4165.0,1.0,0.0,1...|       0.0|      0.0|
|[5414.0,0.0,0.0,1...|       0.0|      0.0|
|[4530.0,1.0,0.0,0...|       0.0|      0.0|
|(20,[0,1,3,6,8,9,...|       0.0|      0.0|
```

Looking at the result, the **prediction** column contains the predicted value for the label, and the **trueLabel** column contains the actual known value from the testing data. It looks like there is some variance between the predictions and the actual values (the individual differences are referred to as *residuals*) you'll learn how to measure the accuracy of a model.

Compute Confusion Matrix Metrics

Classifiers are typically evaluated by creating a *confusion matrix*, which indicates the number of:

- True Positives
- True Negatives
- False Positives
- False Negatives

From these core measures, other evaluation metrics such as *precision* and *recall* can be calculated.

```
val tp = predicted.filter("prediction == 1 AND truelabel == 1").count().toFloat
val fp = predicted.filter("prediction == 1 AND truelabel == 0").count().toFloat
val tn = predicted.filter("prediction == 0 AND truelabel == 0").count().toFloat
val fn = predicted.filter("prediction == 0 AND truelabel == 1").count().toFloat
val metrics = spark.createDataFrame(Seq(
 ("TP", tp),
 ("FP", fp),
 ("TN", tn),
 ("FN", fn),
 ("Precision", tp / (tp + fp)),
 ("Recall", tp / (tp + fn)))).toDF("metric", "value")
metrics.show()
```

```
+---------+----------+
|   metric|     value|
+---------+----------+
|       TP|     143.0|
|       FP|      44.0|
|       TN|    1542.0|
|       FN|     468.0|
|Precision| 0.7647059|
```

```
|   Recall|0.23404256|
+---------+----------+

tp: Float = 143.0
fp: Float = 44.0
tn: Float = 1542.0
fn: Float = 468.0
metrics: org.apache.spark.sql.DataFrame = [metric: string, value: float]
```

View the Raw Prediction and Probability

The prediction is based on a raw prediction score that describes a labelled point in a logistic function. This raw prediction is then converted to a predicted label of 0 or 1 based on a probability vector that indicates the confidence for each possible label value (in this case, 0 and 1). The value with the highest confidence is selected as the prediction.

```
prediction.select("rawPrediction", "probability", "prediction", "trueLabel").show(100, truncate=false)
```

```
+----------------------------------------------+----------------------------------------+----------+---------+
|rawPrediction                                 |probability                             |prediction|trueLabel|
+----------------------------------------------+----------------------------------------+----------+---------+
|[1.1304381342112333,-1.1304381342112333]      |[0.7559197461478987,0.24408025385210122]|0.0       |0.0      |
|[0.4253744452547032,-0.4253744452547032]      |[0.6047685894555563,0.39523141054444366]|0.0       |0.0      |
|[0.07981260697469093,-0.07981260697469093]    |[0.5199425666015123,0.4800574333984877] |0.0       |1.0      |
|[-0.0030250274566276797,0.0030250274566276797]|[0.49924374371253827,0.5007562562874618]|1.0       |1.0      |
|[1.7128739568424858,-1.7128739568424858]      |[0.847208678178602,0.15279132182139807] |0.0       |0.0      |
|[1.5820315950865826,-1.5820315950865826]      |[0.8294920489804423,0.1705079510195577] |0.0       |0.0      |
|[0.06442224523675126,-0.06442224523675126]    |[0.5160999934771059,0.4839000065228941] |0.0       |0.0      |
|[1.737672283491884,-1.737672283491884]        |[0.8503911608053444,0.14960883919465556]|0.0       |0.0      |
|[-0.05333307084631335,0.05333307084631335]    |[0.4866698918368806,0.5133301081631194] |1.0       |0.0      |
|[1.5378236376061276,-1.5378236376061276]      |[0.8231481234205636,0.1768518765794363] |0.0       |0.0      |
|[1.8279540785810944,-1.8279540785810944]      |[0.8615178189339568,0.13848218106604315]|0.0       |0.0      |
|[0.3245001689023171,-0.3245001689023171]      |[0.5804205845843676,0.4195794154156324] |0.0       |0.0      |
```

```
|[1.8464475913455813,-1.8464475913455813]      |[0.8637094697484022,0.13629053025159776] |0.0       |0.0      |
|[2.328903225979352,-2.328903225979352]        |[0.9112426702490206,0.08875732975097934] |0.0       |0.0      |
|[2.9809591177222563,-2.9809591177222563]      |[0.9517064726446443,0.048293527355355706]|0.0       |0.0      |
|[2.241713756086338,-2.241713756086338]        |[0.9039333801930787,0.09606661980692134] |0.0       |0.0      |
|[2.1821588014398827,-2.1821588014398827]      |[0.8986358853932024,0.10136411460679773] |0.0       |0.0      |
|[0.9141098078796176,-0.9141098078796176]      |[0.713840419828425,0.286159580171575]    |0.0       |0.0      |
```

Note that the results include rows where the probability for 0 (the first value in the probability vector) is only slightly higher than the probability for 1 (the second value in the probability vector). The default discrimination threshold (the boundary that decides whether a probability is predicted as a 1 or a 0) is set to 0.5; so the prediction with the highest probability is always used, no matter how close to the threshold.

Review the Area Under ROC

Another way to assess the performance of a classification model is to measure the area under a ROC curve for the model. the spark.ml library includes a **BinaryClassificationEvaluator** class that you can use to compute this. The ROC curve shows the True Positive and False Positive rates plotted for varying thresholds.

```
import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator

val evaluator = new
BinaryClassificationEvaluator().setLabelCol("trueLabel").setRawPredictionCol("rawPrediction").setMetricName("areaUnderROC")
val auc = evaluator.evaluate(prediction)
println("AUC = " + (auc))
```

```
AUC = 0.818327509736379
import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator
evaluator: org.apache.spark.ml.evaluation.BinaryClassificationEvaluator = binEval_34d1e7066870
auc: Double = 0.818327509736379
```

Train a Naive Bayes Model

Naive Bayes can be trained very efficiently. With a single pass over the training data, it computes the conditional probability distribution of each feature given each label. For prediction, it applies Bayes' theorem to compute the conditional probability distribution of each label given an observation.

```
%scala

import org.apache.spark.ml.classification.NaiveBayes
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator

import org.apache.spark.ml.feature.VectorAssembler

val assembler = new VectorAssembler().setInputCols(Array("customerID_indexed", "gender_indexed", "SeniorCitizen",
"Partner_indexed", "Dependents_indexed", "PhoneService_indexed", "MultipleLines_indexed", "InternetService_indexed",
"OnlineSecurity_indexed", "OnlineBackup_indexed", "DeviceProtection_indexed", "TechSupport_indexed", "StreamingTV_indexed",
"StreamingMovies_indexed", "Contract_indexed", "PaperlessBilling_indexed", "PaymentMethod_indexed", "tenure",
"MonthlyCharges", "TotalCharges" )).setOutputCol("features")

val training = assembler.transform(TelDF).select($"features", $"Churn_indexed".alias("label"))

// Split the data into training and test sets (30% held out for testing)
val Array(trainingData, testData) = training.randomSplit(Array(0.9, 0.1), seed = 1234L)

// Train a NaiveBayes model.
val model = new NaiveBayes()
  .fit(trainingData)

// Select example rows to display.
val predictions = model.transform(testData)

val predicted = predictions.select("features", "prediction", "label")
predicted.show()
```

```
+--------------------+----------+-----+
|            features|prediction|label|
+--------------------+----------+-----+
|(20,[0,1,2,3,4,5,...|       1.0|  0.0|
```

```
|(20,[0,1,2,3,6,9,...|       1.0|  1.0|
|(20,[0,1,2,3,6,9,...|       0.0|  0.0|
|(20,[0,1,2,3,6,9,...|       0.0|  1.0|
|(20,[0,1,2,3,7,9,...|       0.0|  1.0|
|(20,[0,1,2,3,10,1...|       0.0|  0.0|
|(20,[0,1,2,5,6,7,...|       1.0|  0.0|
|(20,[0,1,2,5,6,7,...|       1.0|  0.0|
|(20,[0,1,2,6,7,8,...|       1.0|  0.0|
|(20,[0,1,2,6,8,9,...|       0.0|  0.0|
|(20,[0,1,2,6,8,9,...|       0.0|  0.0|
|(20,[0,1,2,6,8,12...|       1.0|  1.0|
|(20,[0,1,2,6,9,17...|       0.0|  1.0|
|(20,[0,1,2,6,10,1...|       0.0|  1.0|
|(20,[0,1,2,6,10,1...|       0.0|  0.0|
|(20,[0,1,2,6,10,1...|       0.0|  1.0|
|(20,[0,1,2,6,10,1...|       0.0|  1.0|
```

Train a One-vs-Rest classifier (a.k.a. One-vs-All) Model

OneVsRest is an example of a machine learning reduction for performing multiclass classification given a base classifier that can perform binary classification efficiently. It is also known as “One-vs-All.”

```
import org.apache.spark.ml.classification.{LogisticRegression, OneVsRest}
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator

// load data file.
import org.apache.spark.ml.feature.VectorAssembler
```

```
val assembler = new VectorAssembler().setInputCols(Array("customerID_indexed", "gender_indexed", "SeniorCitizen",
"Partner_indexed", "Dependents_indexed", "PhoneService_indexed", "MultipleLines_indexed", "InternetService_indexed",
"OnlineSecurity_indexed", "OnlineBackup_indexed", "DeviceProtection_indexed", "TechSupport_indexed", "StreamingTV_indexed",
"StreamingMovies_indexed", "Contract_indexed", "PaperlessBilling_indexed", "PaymentMethod_indexed", "tenure",
"MonthlyCharges", "TotalCharges" )).setOutputCol("features")

val training = assembler.transform(TelDF).select($"features", $"Churn_indexed".alias("label"))

// generate the train/test split.
val Array(train, test) = training.randomSplit(Array(0.8, 0.2))

// instantiate the base classifier
val classifier = new LogisticRegression()
  .setMaxIter(10)
  .setTol(1E-6)
  .setFitIntercept(true)

// instantiate the One Vs Rest Classifier.
val ovr = new OneVsRest().setClassifier(classifier)

// train the multiclass model.
val ovrModel = ovr.fit(train)

// score the model on test data.
val predictions = ovrModel.transform(test)

val predicted = predictions.select("features", "prediction", "label")
predicted.show()
```

```
+--------------------+----------+-----+
|            features|prediction|label|
+--------------------+----------+-----+
|(20,[0,1,2,3,4,6,...|       1.0|  0.0|
|(20,[0,1,2,3,4,6,...|       1.0|  0.0|
```

```
|(20,[0,1,2,3,4,6,...|       0.0|  1.0|
|(20,[0,1,2,3,5,6,...|       0.0|  1.0|
|(20,[0,1,2,3,6,8,...|       1.0|  1.0|
|(20,[0,1,2,3,6,9,...|       0.0|  0.0|
|(20,[0,1,2,3,6,9,...|       0.0|  0.0|
|(20,[0,1,2,3,6,9,...|       0.0|  1.0|
|(20,[0,1,2,3,6,9,...|       1.0|  1.0|
|(20,[0,1,2,3,6,9,...|       0.0|  1.0|
|(20,[0,1,2,3,6,9,...|       0.0|  1.0|
|(20,[0,1,2,3,6,9,...|       1.0|  1.0|
|(20,[0,1,2,3,6,9,...|       0.0|  1.0|
|(20,[0,1,2,3,6,10...|       0.0|  0.0|
|(20,[0,1,2,3,6,12...|       1.0|  0.0|
|(20,[0,1,2,3,6,12...|       1.0|  1.0|
|(20,[0,1,2,3,6,13...|       1.0|  1.0|
|(20,[0,1,2,3,6,13...|       1.0|  1.0|
```

```
Test Error = 0.18593314763231195
evaluator: org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator = mcEval_c4f2eec5d58d
accuracy: Double = 0.814066852367688
```