Bicycle Sharing Demand

Domain – Transportation Industry

Requirement

Building a Bicycle Sharing demand forecasting service that combines historical usage patterns with weather data to forecast the Bicycle rental demand in realtime.

To develop thissystem, you must first explore the dataset and build a model. Once it's done you must persist the model and then on each request run a Spark job to load the model and make predictions on each SparkStreaming request.

* Importing all the required Packages

```
import org.apache.spark.rdd.RDD
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.feature.StringIndexer
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.util.IntParam
import org.apache.spark.sql.SQLContext
import org.apache.spark.sql.functions._
import org.apache.spark.sql._
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.types._
import org.apache.log4j._
import org.apache.spark.sql.functions.to_timestamp
import org.apache.spark.ml.regression.LinearRegression
import org.apache.spark.ml.evaluation.RegressionEvaluator
import org.apache.spark.ml.feature.OneHotEncoder
import org.apache.spark.ml.regression.{GBTRegressionModel, GBTRegressor}
import org.apache.spark.ml.regression.DecisionTreeRegressor
import org.apache.spark.ml.regression.RandomForestRegressor
```

Data Exploration and Transformation

* Read the dataset in Spark

Note : Databricks community edition notebook has been used with the default language as Scala

```
val trainDF = spark.read.format("csv").option("inferSchema",true).option("header",true).load("/FileStore/tables/edureka/train.csv")
trainDF.show(10)
```

▸ (3) Spark Jobs

▸ trainDF: org.apache.spark.sql.DataFrame = [datetime: string, season: integer ... 10 more fields]

```
+----------------+------+-------+----------+-------+-----+------+--------+---------+------+----------+-----+
|        datetime|season|holiday|workingday|weather| temp| atemp|humidity|windspeed|casual|registered|count|
+----------------+------+-------+----------+-------+-----+------+--------+---------+------+----------+-----+
|01-01-2011 00:00|     1|      0|         0|      1| 9.84|14.395|      81|      0.0|     3|        13|   16|
|01-01-2011 01:00|     1|      0|         0|      1| 9.02|13.635|      80|      0.0|     8|        32|   40|
|01-01-2011 02:00|     1|      0|         0|      1| 9.02|13.635|      80|      0.0|     5|        27|   32|
|01-01-2011 03:00|     1|      0|         0|      1| 9.84|14.395|      75|      0.0|     3|        10|   13|
|01-01-2011 04:00|     1|      0|         0|      1| 9.84|14.395|      75|      0.0|     0|         1|    1|
|01-01-2011 05:00|     1|      0|         0|      2| 9.84| 12.88|      75|   6.0032|     0|         1|    1|
|01-01-2011 06:00|     1|      0|         0|      1| 9.02|13.635|      80|      0.0|     2|         0|    2|
|01-01-2011 07:00|     1|      0|         0|      1|  8.2| 12.88|      86|      0.0|     1|         2|    3|
|01-01-2011 08:00|     1|      0|         0|      1| 9.84|14.395|      75|      0.0|     1|         7|    8|
|01-01-2011 09:00|     1|      0|         0|      1|13.12|17.425|      76|      0.0|     8|         6|   14|
+----------------+------+-------+----------+-------+-----+------+--------+---------+------+----------+-----+
only showing top 10 rows

trainDF: org.apache.spark.sql.DataFrame = [datetime: string, season: int ... 10 more fields]
```


* Get summary of data and variable types

```
trainDF.printSchema
```

```
root
 |-- datetime: string (nullable = true)
 |-- season: integer (nullable = true)
 |-- holiday: integer (nullable = true)
 |-- workingday: integer (nullable = true)
 |-- weather: integer (nullable = true)
 |-- temp: double (nullable = true)
 |-- atemp: double (nullable = true)
 |-- humidity: integer (nullable = true)
 |-- windspeed: double (nullable = true)
 |-- casual: integer (nullable = true)
 |-- registered: integer (nullable = true)
 |-- count: integer (nullable = true)
```

```
display(trainDF.describe())
```

▸ (2) Spark Jobs

	summary	datetime	season	holiday	workingday	weather	temp	atemp
1	count	10886	10886	10886	10886	10886	10886	10886
2	mean	null	2.5066139996325556	0.02856880396839978	0.6808745177291935	1.418427337865148	20.230859819952173	23.65508405291192
3	stddev	null	1.1161743093443237	0.16659885062470944	0.4661591687997361	0.6338385858190968	7.791589843987573	8.47460062648494
4	min	01-01-2011 00:00	1	0	0	1	0.82	0.76
5	max	19-12-2012 23:00	4	1	1	4	41.0	45.455

Showing all 5 rows.

* Decide which columns should be categorical and convert them accordingly

```
//Cheking unique value In each column
val exprs = trainDF.schema.fields.filter(x => x.dataType != StringType).map(x=>x.name ->"approx_count_distinct").toMap
//data.agg(exprs).show(false)
```

```
exprs: scala.collection.immutable.Map[String,String] = Map(workingday -> approx_count_distinct, windspeed -> approx_count_distinct, registered -> approx_count_distinct, count -> approx_count_distinct, atemp -> approx_count_distinct, season -> approx_count_distinct, casual -> approx_count_distinct, humidity -> approx_count_distinct, temp -> approx_count_distinct, holiday -> approx_count_distinct, weather -> approx_count_distinct)
```

Command took 1.50 seconds -- by aj08.mufc@outlook.com at 22/09/2021, 12:52:08 on Custom

Cmd 7

```
display(trainDF.agg(exprs))
```

▸ (2) Spark Jobs

	approx_count_distinct(workingday)	approx_count_distinct(windspeed)	approx_count_distinct(registered)	approx_count_distinct(count)	approx_count_distinct(atemp)
1	2	27	726	802	60

Showing all 1 rows.

```
//Here we are considering "workingday,holiday,season, and wether column" as a categorical column and applying onehotencoder on column with values > 2
val indexer = Array("season","weather").map(c=>new OneHotEncoder().setInputCol(c).setOutputCol(c + "_Vec"))
val pipeline = new Pipeline().setStages(indexer)
val df_r = pipeline.fit(trainDF).transform(trainDF).drop("season","weather")
```

▸ (2) Spark Jobs

▸ df_r: org.apache.spark.sql.DataFrame = [datetime: string, holiday: integer ... 10 more fields]

```
indexer: Array[org.apache.spark.ml.feature.OneHotEncoder] = Array(oneHotEncoder_eea6d5da393a, oneHotEncoder_164d1d227465)
pipeline: org.apache.spark.ml.Pipeline = pipeline_aa1617d47dcf
df_r: org.apache.spark.sql.DataFrame = [datetime: string, holiday: int ... 10 more fields]
```

Command took 2.84 seconds -- by aj08.mufc@outlook.com at 22/09/2021, 12:53:12 on Custom

Cmd 9

```
df_r.show(5)
```

▸ (1) Spark Jobs

```
+----------------+-------+----------+----+------+--------+---------+------+----------+-----+-------------+-------------+
|        datetime|holiday|workingday|temp| atemp|humidity|windspeed|casual|registered|count|   season_Vec|  weather_Vec|
+----------------+-------+----------+----+------+--------+---------+------+----------+-----+-------------+-------------+
|01-01-2011 00:00|      0|         0|9.84|14.395|      81|      0.0|     3|        13|   16|(4,[1],[1.0])|(4,[1],[1.0])|
|01-01-2011 01:00|      0|         0|9.02|13.635|      80|      0.0|     8|        32|   40|(4,[1],[1.0])|(4,[1],[1.0])|
|01-01-2011 02:00|      0|         0|9.02|13.635|      80|      0.0|     5|        27|   32|(4,[1],[1.0])|(4,[1],[1.0])|
|01-01-2011 03:00|      0|         0|9.84|14.395|      75|      0.0|     3|        10|   13|(4,[1],[1.0])|(4,[1],[1.0])|
|01-01-2011 04:00|      0|         0|9.84|14.395|      75|      0.0|     0|         1|    1|(4,[1],[1.0])|(4,[1],[1.0])|
+----------------+-------+----------+----+------+--------+---------+------+----------+-----+-------------+-------------+
only showing top 5 rows
```


continued down below.......

* Check for any missing values in the data set

```
trainDF.select(trainDF.columns.map(c => sum(col(c).isNull.cast("int")).alias(c)): _*).show
```

▸ (2) Spark Jobs

```
+--------+------+-------+----------+-------+----+-----+--------+---------+------+----------+-----+
|datetime|season|holiday|workingday|weather|temp|atemp|humidity|windspeed|casual|registered|count|
+--------+------+-------+----------+-------+----+-----+--------+---------+------+----------+-----+
|       0|     0|      0|         0|      0|   0|    0|       0|        0|     0|         0|    0|
+--------+------+-------+----------+-------+----+-----+--------+---------+------+----------+-----+
```

* Explode season column into separate columns such as season_ and drop season
* Execute the same for weather as weather_ and drop weather

There is no need to explode the season column and weather column as we have already applied one-hot-encoder for categorical columns in the dataset with values > 2

* Split data time into meaningful columns such as hour, day month, year

```
//Converting datetime string column to timestamp column
val df_time = df_r.withColumn("datetime", to_timestamp(col("datetime"),"d-M-y H:m"))

//Now Spliting date time into meaning columns such as year,month,day,hour
val datetime_trainDF = df_time.
withColumn("year", year(col("datetime"))).
withColumn("month", month(col("datetime"))).
withColumn("day", dayofmonth(col("datetime"))).
withColumn("hour", hour(col("datetime"))).
withColumn("minute",minute(col("datetime")))
```

▸ df_time: org.apache.spark.sql.DataFrame = [datetime: timestamp, holiday: integer ... 10 more fields]

▸ datetime_trainDF: org.apache.spark.sql.DataFrame = [datetime: timestamp, holiday: integer ... 15 more fields]

```
df_time: org.apache.spark.sql.DataFrame = [datetime: timestamp, holiday: int ... 10 more fields]
datetime_trainDF: org.apache.spark.sql.DataFrame = [datetime: timestamp, holiday: int ... 15 more fields]
```

Command took 0.92 seconds -- by aj08.mufc@outlook.com at 22/09/2021, 12:57:42 on Custom

* Explore how count varies with different features such as hour, month,etc

```
datetime_trainDF.groupBy("year").count.show()
datetime_trainDF.groupBy("month").count.show()
datetime_trainDF.groupBy("day").count.show()
datetime_trainDF.groupBy("hour").count.show()
datetime_trainDF.groupBy("minute").count.show()

```

```
+----+-----+
|year|count|
+----+-----+
|2012| 5464|
|2011| 5422|
+----+-----+

+-----+-----+
|month|count|
+-----+-----+
|   12|  912|
|    1|  884|
|    6|  912|
|    3|  901|
|    5|  912|
|    9|  909|
|    4|  909|
|    8|  912|
|    7|  912|
|   10|  911|
|   11|  911|
```

* Model Development

Split the data into train and test set

```
val splitSeed = 123
val Array(train,train_test) = datetime_trainDF.randomSplit(Array(0.7,0.3),splitSeed)
```

▸ train: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [datetime: timestamp, holiday: integer ... 15 more fields]

▸ train_test: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [datetime: timestamp, holiday: integer ... 15 more fields]

```
splitSeed: Int = 123
train: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [datetime: timestamp, holiday: int ... 15 more fields]
train_test: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [datetime: timestamp, holiday: int ... 15 more fields]
```

* Try different Regression Algorithms and note down the accuracy

```
//Generate Feature Column
val feature = Array("holiday","workingday","temp","atemp","humidity","windspeed","season_Vec","weather_Vec","year","month","day","hour","minute")
//Assemble Feature Column
val assembler = new VectorAssembler().setInputCols(feature).setOutputCol("features")
```

```
feature: Array[String] = Array(holiday, workingday, temp, atemp, humidity, windspeed, season_Vec, weather_Vec, year, month, day, hour, minute)
assembler: org.apache.spark.ml.feature.VectorAssembler = VectorAssembler: uid=vecAssembler_9f981c065826, handleInvalid=error, numInputCols=13
```

continued below......

* Linear Regression Model

```
//Model Building
val lr = new LinearRegression().setLabelCol("count").setFeaturesCol("features")

//Creating Pipeline
val pipeline = new Pipeline().setStages(Array(assembler,lr))

//Training Model
val lrModel = pipeline.fit(train)
val predictions = lrModel.transform(train_test)

//Model Summary
val evaluator = new RegressionEvaluator().setLabelCol("count").setPredictionCol("prediction").setMetricName("rmse")
val rmse = evaluator.evaluate(predictions)
println("Linear Regression Root Mean Squared Error (RMSE) on train_test data = " + rmse)
```

▸ (3) Spark Jobs

▸ predictions: org.apache.spark.sql.DataFrame = [datetime: timestamp, holiday: integer ... 17 more fields]

```
Linear Regression Root Mean Squared Error (RMSE) on train_test data = 143.53570193575268
lr: org.apache.spark.ml.regression.LinearRegression = linReg_541ae3c313c1
pipeline: org.apache.spark.ml.Pipeline = pipeline_52d5bafbcef6
lrModel: org.apache.spark.ml.PipelineModel = pipeline_52d5bafbcef6
predictions: org.apache.spark.sql.DataFrame = [datetime: timestamp, holiday: int ... 17 more fields]
evaluator: org.apache.spark.ml.evaluation.RegressionEvaluator = RegressionEvaluator: uid=regEval_8ae8bfbce78c, metricName=rmse, throughOrigin=false
rmse: Double = 143.53570193575268
```


* GBT Regressor

```
//Model Building
val gbt = new GBTRegressor().setLabelCol("count").setFeaturesCol("features")

//Creating pipeline
val pipeline = new Pipeline().setStages(Array(assembler,gbt))

//Training Model
val gbtModel = pipeline.fit(train)
val predictions = gbtModel.transform(train_test)

//Model Summary
val evaluator = new RegressionEvaluator().setLabelCol("count").setPredictionCol("prediction").setMetricName("rmse")
val rmse = evaluator.evaluate(predictions)
println("GBT Regressor Root Mean Squared Error (RMSE) on train_test data = " + rmse)
```

▸ (51) Spark Jobs

▸ predictions: org.apache.spark.sql.DataFrame = [datetime: timestamp, holiday: integer ... 17 more fields]

```
GBT Regressor Root Mean Squared Error (RMSE) on train_test data = 60.13502303606433
gbt: org.apache.spark.ml.regression.GBTRegressor = gbtr_ee12e982664d
pipeline: org.apache.spark.ml.Pipeline = pipeline_d0d442d379b3
gbtModel: org.apache.spark.ml.PipelineModel = pipeline_d0d442d379b3
predictions: org.apache.spark.sql.DataFrame = [datetime: timestamp, holiday: int ... 17 more fields]
evaluator: org.apache.spark.ml.evaluation.RegressionEvaluator = RegressionEvaluator: uid=regEval_7971c6b176e6, metricName=rmse, throughOrigin=false
rmse: Double = 60.13502303606433
```


continued below.....

* Decision Tree Regressor

```
//Model Building
val dt = new DecisionTreeRegressor().setLabelCol("count").setFeaturesCol("features")

//Creating Pipeline
val pipeline = new Pipeline().setStages(Array(assembler,dt))

//Training Model
val dtModel = pipeline.fit(train)
val predictions = dtModel.transform(train_test)

//Model Summary
val evaluator = new RegressionEvaluator().setLabelCol("count").setPredictionCol("prediction").setMetricName("rmse")
val rmse = evaluator.evaluate(predictions)
println("Decision Tree Regressor Root Mean Squared Error (RMSE) on train_test data = " + rmse)
```

▸ (9) Spark Jobs

▸ predictions: org.apache.spark.sql.DataFrame = [datetime: timestamp, holiday: integer ... 17 more fields]

```
Decision Tree Regressor Root Mean Squared Error (RMSE) on train_test data = 108.42151766658162
dt: org.apache.spark.ml.regression.DecisionTreeRegressor = dtr_5d1141349e57
pipeline: org.apache.spark.ml.Pipeline = pipeline_90f41cf62351
dtModel: org.apache.spark.ml.PipelineModel = pipeline_90f41cf62351
predictions: org.apache.spark.sql.DataFrame = [datetime: timestamp, holiday: int ... 17 more fields]
evaluator: org.apache.spark.ml.evaluation.RegressionEvaluator = RegressionEvaluator: uid=regEval_7736cf7d9129, metricName=rmse, throughOrigin=fa
rmse: Double = 108.42151766658162
```


* Random Forest Regressor

```
//Model Building
val rf = new RandomForestRegressor().setLabelCol("count").setFeaturesCol("features")

//Creating Pipeline
val pipeline = new Pipeline().setStages(Array(assembler,rf))

//Training Model
val rfModel = pipeline.fit(train)
val predictions = rfModel.transform(train_test)

//Model Summary
val evaluator = new RegressionEvaluator().setLabelCol("count").setPredictionCol("prediction").setMetricName("rmse")
val rmse = evaluator.evaluate(predictions)
println("Random Forest Regressor Root Mean Squared Error (RMSE) on train_test data = " + rmse)
```

▸ (9) Spark Jobs

▸ predictions: org.apache.spark.sql.DataFrame = [datetime: timestamp, holiday: integer ... 17 more fields]

```
Random Forest Regressor Root Mean Squared Error (RMSE) on train_test data = 113.05487428850965
rf: org.apache.spark.ml.regression.RandomForestRegressor = rfr_39f5471ad0a5
pipeline: org.apache.spark.ml.Pipeline = pipeline_682df01be52d
rfModel: org.apache.spark.ml.PipelineModel = pipeline_682df01be52d
predictions: org.apache.spark.sql.DataFrame = [datetime: timestamp, holiday: int ... 17 more fields]
evaluator: org.apache.spark.ml.evaluation.RegressionEvaluator = RegressionEvaluator: uid=regEval_12a9d475556e, metricName=rmse, throughOrigin=false
rmse: Double = 113.05487428850965
```


* Select the best model and persist it

```
// In this case the "GBT Regressor Model" has the best accuracy compared to other models

gbtModel.write.overwrite().save("/FileStore/tables/model/bicycle-model")
```

* Model Implementation

```scala
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.SparkContext._
import org.apache.spark.sql._
import org.apache.spark.sql.types._
import org.apache.spark.sql.functions._
import org.apache.spark.ml.regression.{GBTRegressionModel, GBTRegressor}
import org.apache.spark.ml.evaluation.RegressionEvaluator
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml._
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.feature.OneHotEncoder

object BicyclePredict{
  def main(args: Array[String]) {
    val sparkConf = new SparkConf().setAppName("ajay")
    val sc = new SparkContext(sparkConf)

    sc.setLogLevel("ERROR")

    val spark = new org.apache.spark.sql.SQLContext(sc)
    import spark.implicits._

    println("Reading training data..................")

    val trainDF = spark.read.format("csv").option("inferSchema",true).option("header",true).load("/FileStore/tables/edureka/train.csv")

    println("Cleaning data................")

    //Converting datetime string column to timestamp column
    val df_time = trainDF.withColumn("datetime", to_timestamp(col("datetime"),"d-M-y H:m"))

    //Now Spliting date time into meaning columns such as year,month,day,hour
    val datetime_trainDF = df_time.
    withColumn("year", year(col("datetime"))).
    withColumn("month", month(col("datetime"))).
    withColumn("day", dayofmonth(col("datetime"))).
    withColumn("hour", hour(col("datetime"))).
    withColumn("minute",minute(col("datetime")))

    //Onehot encoding on season and weather column.
    val indexer = Array("season","weather").map(c=>new OneHotEncoder().setInputCol(c).setOutputCol(c + "_Vec"))
    val pipeline = new Pipeline().setStages(indexer)
    val df_r = pipeline.fit(datetime_trainDF).transform(datetime_trainDF)

    //split data into train test
    val splitSeed =123
    val Array(train, train_test) = df_r.randomSplit(Array(0.7, 0.3), splitSeed)

    //Generate Feature Column
    val feature_cols = Array("holiday","workingday","temp","atemp","humidity","windspeed","season_Vec","weather_Vec","year","month","day","hour","minute")

    //Assemble Feature
    val assembler = new VectorAssembler().setInputCols(feature_cols).setOutputCol("features")

    //Model Building
    val gbt = new GBTRegressor().setLabelCol("count").setFeaturesCol("features")

    val pipeline2 = new Pipeline().setStages(Array(assembler,gbt))

    println("Training model................")
    val gbt_model = pipeline2.fit(train)
    val predictions = gbt_model.transform(train_test)

    val evaluator = new RegressionEvaluator().setLabelCol("count").setPredictionCol("prediction").setMetricName("rmse")
    val rmse = evaluator.evaluate(predictions)
    println("GBT Regressor Root Mean Squared Error (RMSE) on train_test data = " + rmse)

    println("Persisting the model...............")
    gbt_model.write.overwrite().save("/FileStore/tables/model/bicycle-model")
  }
}
```

```
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.SparkContext._
import org.apache.spark.sql._
import org.apache.spark.sql.types._
import org.apache.spark.sql.functions._
import org.apache.spark.ml.regression.{GBTRegressionModel, GBTRegressor}
import org.apache.spark.ml.evaluation.RegressionEvaluator
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml._
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.feature.OneHotEncoder
defined object BicyclePredict
```

Command took 1.71 seconds -- by aj08.mufc@outlook.com at 22/09/2021, 17:52:27 on Custom

md 24

```
//Application Execution
spark2-submit --class "BicyclePredict" --master yarn /mnt/home/edureka_1470433/BicycleProject/BicycleTrain/target/scala-2.11/bicycletrain_2.11-1.0.jar
```

* Application Development for demand prediction

```
// Model Prediction Application - Write an application to predict the bike demand based on the input dataset from HDFS:

// 1. Load the persisted model.

// 2. Predict bike demand

// 3.Persist the result to RDBMS
```

```
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.SparkContext._
import org.apache.spark.sql._
import org.apache.spark.sql.types._
import org.apache.spark.sql.functions._
import org.apache.spark.ml.regression.{GBTRegressionModel, GBTRegressor}
import org.apache.spark.ml.evaluation.RegressionEvaluator
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml._
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.feature.OneHotEncoder

object BicyclePredict {
  def main(args: Array[String]) {
    val sparkConf = new SparkConf().setAppName("Telecom")
    val sc = new SparkContext(sparkConf)

    sc.setLogLevel("ERROR")

    val spark = new org.apache.spark.sql.SQLContext(sc)
    import spark.implicits._

    println("Reading Training data................")

    val testDF = spark.read.format("csv").option("inferSchema",true).option("header",true).load("/FileStore/tables/edureka/test.csv")

    println("Cleaning data................")
```


continued below......

```

    //Converting datetime string column to timestamp column
    val df_time = testDF.withColumn("datetime", to_timestamp(col("datetime"),"d-M-y H:m"))

    //Now Spliting date time into meaning columns such as year,month,day,hour
    val datetime_testDF = df_time.
    withColumn("year", year(col("datetime"))).
    withColumn("month", month(col("datetime"))).
    withColumn("day", dayofmonth(col("datetime"))).
    withColumn("hour", hour(col("datetime"))).
    withColumn("minute",minute(col("datetime")))

    //Onehot encoding on season and weather column.
    val indexer = Array("season","weather").map(c=>new OneHotEncoder().setInputCol(c).setOutputCol(c + "_Vec"))
    val pipeline = new Pipeline().setStages(indexer)
    val df_r = pipeline.fit(datetime_testDF).transform(datetime_testDF)

    println("Loading Trained Model.............")
    val gbt_model = PipelineModel.read.load("/FileStore/tables/model/bicycle-model")

    println("Making predictions...........")
    val predictions = gbt_model.transform(df_r).select($"datetime",$"prediction".as("count"))

    println("Persisting the result to RDBMS.............")
```

```
    predictions.write.format("jdbc").
      option("url", "jdbc:mysql://mysqldb.edu.cloudlab.com/ajay_bicycle").
      option("driver", "com.mysql.cj.jdbc.Driver").option("dbtable", "predictions").
      option("user", "labuser").
      option("password", "edureka").
      mode(SaveMode.Append).save
  }
}
```

* Application for streaming data

```
// Write an application to predict demand on streaming data:
// Setup flume to push data into spark flume sink.
```

Cmd 28

```
//Kafka topic creation:
kafka-topics --create --zookeeper ip-20-0-21-161.ec2.internal:2181 --replication-factor 1 --partitions 1 --topic edureka_1470433_bicycle_ajay
```

```
agent1.sources  = source1
agent1.channels = channel1
agent1.sinks = spark
agent1.sources.source1.type = org.apache.flume.source.kafka.KafkaSource
agent1.sources.source1.kafka.bootstrap.servers = ip-20-0-31-210.ec2.internal:9092
agent1.sources.source1.kafka.topics = edureka_1470433_bicycle_ajay
agent1.sources.source1.kafka.consumer.group.id = edureka_1470433_bicycle_ajay
agent1.sources.source1.channels = channel1
agent1.sources.source1.interceptors = i1
agent1.sources.source1.interceptors.i1.type = timestamp
agent1.sources.source1.kafka.consumer.timeout.ms = 100
agent1.channels.channel1.type = memory
agent1.channels.channel1.capacity = 10000
agent1.channels.channel1.transactionCapacity = 1000
agent1.sinks.spark.type = org.apache.spark.streaming.flume.sink.SparkSink
agent1.sinks.spark.hostname = ip-20-0-41-62.ec2.internal
agent1.sinks.spark.port = 4143
agent1.sinks.spark.channel = channel1
```

```
// Configure spark streaming to pulldata from spark flume sink using receivers
// and predict the demand using model and persist the result to RDBMS.
```

Cmd 32

```
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.SparkContext._
import org.apache.spark.sql._
import org.apache.spark.sql.types._
import org.apache.spark.sql.functions._
import org.apache.spark.ml.regression.{GBTRegressionModel, GBTRegressor}
import org.apache.spark.ml.feature.{StringIndexer, VectorAssembler}
import org.apache.spark.ml._
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.flume._
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.feature.OneHotEncoder

object BicycleStreaming {
  case class Bicycle(datetime: String, season: Int, holiday: Int, workingday: Int, weather: Int, temp: Double, atemp: Double, humidity: Int, windspeed: Double)

  def main(args: Array[String]) {
    val sparkConf = new SparkConf().setAppName("ajay")
    val sc = new SparkContext(sparkConf)
    val ssc = new StreamingContext(sc, Seconds(2))

    sc.setLogLevel("ERROR")

    val spark = new org.apache.spark.sql.SQLContext(sc)

    import spark.implicits._

    val flumeStream = FlumeUtils.createPollingStream(ssc, "ip-20-0-41-62.ec2.internal", 4143)

    println("Loading tained model............")
    val gbt_model = PipelineModel.read.load("/user/edureka_1470433/bicycle-model")


    val lines = flumeStream.map(event => new String(event.event.getBody().array(), "UTF-8"))

    lines.foreachRDD { rdd =>
      def row(line: List[String]): Bicycle = Bicycle(line(0), line(1).toInt, line(2).toInt,
              line(3).toInt, line(4).toInt, line(5).toDouble, line(6).toDouble, line(7).toInt,
              line(8).toDouble
              )

      val rows_rdd = rdd.map(_.split(",").to[List]).map(row)
      val rows_df = rows_rdd.toDF

      if(rows_df.count > 0) {

        val df_time = rows_df.withColumn("datetime",to_timestamp(col("datetime"),"d-M-y H:m"))
        val datetime_testDF = df_time.
```


continued below....

```
        withColumn("year", year(col("datetime"))).
        withColumn("month", month(col("datetime"))).
        withColumn("day", dayofmonth(col("datetime"))).
        withColumn("hour", hour(col("datetime"))).
        withColumn("minute",minute(col("datetime")))

        //Onehot encoding on season nd weather column.
        val indexer = Array("season","weather").map(c => new OneHotEncoder().setInputCol(c).setOutputCol(c + "_Vec"))
        val pipeline = new Pipeline().setStages(indexer)
        val df_r = pipeline.fit(datetime_testDF).transform(datetime_testDF)

        println("Making predictions..............")
        val predictions =  gbt_model.transform(df_r).select($"datetime",$"prediction".as("count"))

        println("Persisting the result to RDBMS.................")
        predictions.write.format("jdbc").
          option("url", "jdbc:mysql://mysqldb.edu.cloudlab.com/ajay64_bicycle").
          option("driver", "com.mysql.cj.jdbc.Driver").option("dbtable", "predictions").
          option("user", "labuser").
          option("password", "edureka").
          mode(SaveMode.Append).save
      }
    }

    ssc.start()
    ssc.awaitTermination()
  }
}
```

```
// Run the application
// Persist the result to RDBMS
```

md 34

```
spark2-submit --packages mysql:mysql-connector-java:8.0.13 --class "BicycleStreaming" --master yarn
/mnt/home/edureka_1470433/BicycleProject/BicycleStreaming/target/scala-2.11/bicyclestreaming_2.11-1.0.jar
```

md 35

```
kafka-console-producer --broker-list ip-20-0-31-210.ec2.internal:9092 --topic edureka_1470433_bicycle_ajay
```

continued below.......

* PySpark

In [1]:
```
#Read dataset in Spark
df = sqlContext.read.load("dbfs:/databricks-datasets/bikeSharing/data-001/day.csv",
                          format='com.databricks.spark.csv',
                          header='true',
                          inferSchema='true')
#This is a databrick dataset of the bike-sharing demand prediction, and the test-train dateset is the given dataset
```

In [2]:
```
#2. Get summary of data and variable types
df.printSchema()
```

root -- instant: integer (nullable = true) -- dteday: timestamp (nullable = true) -- season: integer (nullable = true) -- yr: integer (nullable = true) -- mnth: integer (nullable = true) -- holiday: integer (nullable = true) -- weekday: integer (nullable = true) -- workingday: integer (nullable = true) -- weathersit: integer (nullable = true) -- temp: double (nullable = true) -- atemp: double (nullable = true) -- hum: double (nullable = true) -- windspeed: double (nullable = true) -- casual: integer (nullable = true) -- registered: integer (nullable = true) -- cnt: integer (nullable = true)

In [3]:
```
#df.show(5)
display(df.take(5))
```

instant	dteday	season	yr	mnth	holiday	weekday	workingday	weathersit	temp	atemp	hum	windspeed	casual	registered	cnt
1	2011-01-01T00:00:00.000+0000	1	0	1	0	6	0	2	0.344167	0.363625	0.805833	0.160446	331	654	985
2	2011-01-02T00:00:00.000+0000	1	0	1	0	0	0	2	0.363478	0.353739	0.696087	0.248539	131	670	801
3	2011-01-03T00:00:00.000+0000	1	0	1	0	1	1	1	0.196364	0.189405	0.437273	0.248309	120	1229	1349

In [4]:
```
#Given Train file from which data frame is generated
bs_df = spark.sql("select * from bike_sharing_train_csv")
display(bs_df.take(5))
```

datetime	season	holiday	workingday	weather	temp	atemp	humidity	windspeed	casual	registered	count
01-01-2011 00:00	1	0	0	1	9.84	14.395	81	0.0	3	13	16
01-01-2011 01:00	1	0	0	1	9.02	13.635	80	0.0	8	32	40
01-01-2011 02:00	1	0	0	1	9.02	13.635	80	0.0	5	27	32
01-01-2011 03:00	1	0	0	1	9.84	14.395	75	0.0	3	10	13
01-01-2011 04:00	1	0	0	1	9.84	14.395	75	0.0	0	1	1

In [5]:
```
bs_df.printSchema()
```

root -- datetime: string (nullable = true) -- season: integer (nullable = true) -- holiday: integer (nullable = true) -- workingday: integer (nullable = true) -- weather: integer (nullable = true) -- temp: double (nullable = true) -- atemp: double (nullable = true) -- humidity: integer (nullable = true) -- windspeed: double (nullable = true) -- casual: integer (nullable = true) -- registered: integer (nullable = true) -- count: integer (nullable = true)

In [6]:
```
bs_df.describe().show()
```

```
+-------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----
------------+-----------------+ summary| datetime| season| holiday| workingday| weather| temp| atemp| humidity| windspeed| casual| registered| count| +-------+-----
-----------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-
-----------------+ count| 10886| 10886| 10886| 10886| 10886| 10886| 10886| 10886| 10886| 10886| 10886| 10886| mean|
null|2.5066139996325556|0.02856880396839978|0.6808745177291935| 1.418427337865148|20.23085981995217|23.65508405291192|
61.88645967297446|12.799395406945093|36.02195480433584| 155.5521771082124|191.57413191254824| stddev|
```


continued below.......

In [7]:

```
bs_df.explain()
```

== Physical Plan == *(1) FileScan csv default.bike_sharing_train_csv[datetime#254,season#255,holiday#256,workingday#257,weather#258,temp#259,atemp#260,humidity#261,windspeed#262,casu Batched: false, DataFilters: [], Format: CSV, Location: InMemoryFileIndex[dbfs:/FileStore/tables/train.csv], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<datetime:string,season:int,holiday:int,workingday:int,weather:int,temp:double,atemp:double...

In [8]:

```
#Check for any missing value in dataset and treat it
print(bs_df.count())
df_no_null = bs_df.na.drop()
print(df_no_null.count())
```

10886 10886

In [9]:

```
#Check what are the distinct seasons present to explode them
display(bs_df.select('season').distinct())
```

season
1
3
4
2

In [10]:

```
#user defined function to help creat new columns
def valueToCategory(value, encoding_index):
   if(value == encoding_index):
      return 1
   else:
     return 0
```

In [11]:

```
#Explode season column into separate columns such as season_<val> and drop season
from pyspark.sql.functions import udf
from pyspark.sql.functions import lit
from pyspark.sql.types import *
from pyspark.sql.functions import col
udfValueToCategory = udf(valueToCategory, IntegerType())
bs_df_encoded = (bs_df.withColumn("season_1", udfValueToCategory(col('season'),lit(1)))
                      .withColumn("season_2", udfValueToCategory(col('season'),lit(2)))
                      .withColumn("season_3", udfValueToCategory(col('season'),lit(3)))
                      .withColumn("season_4", udfValueToCategory(col('season'),lit(4))))
bs_df_encoded = bs_df_encoded.drop('season')
```

In [12]:

```
display(bs_df_encoded.take(5))
```

datetime	holiday	workingday	weather	temp	atemp	humidity	windspeed	casual	registered	count	season_1	season_2	season_3	season_4
01-01-2011 00:00	0	0	1	9.84	14.395	81	0.0	3	13	16	1	0	0	0
01-01-2011 01:00	0	0	1	9.02	13.635	80	0.0	8	32	40	1	0	0	0
01-01-2011 02:00	0	0	1	9.02	13.635	80	0.0	5	27	32	1	0	0	0

In [13]:

```
#Execute the same for weather as weather_<val> and drop weather
display(bs_df.select('weather').distinct())
```

weather
1
3
4
2

```
In [14]: bs_df_encoded = (bs_df_encoded.withColumn("weather_1", udfValueToCategory(col('weather'),lit(1)))
                          .withColumn("weather_2", udfValueToCategory(col('weather'),lit(2)))
                          .withColumn("weather_3", udfValueToCategory(col('weather'),lit(3)))
                          .withColumn("weather_4", udfValueToCategory(col('weather'),lit(4))))
         bs_df_encoded = bs_df_encoded.drop('weather')
```

```
In [15]: display(bs_df_encoded.take(5))
```

datetime	holiday	workingday	temp	atemp	humidity	windspeed	casual	registered	count	season_1	season_2	season_3	season_4	weather_1	weather_2
01-01-2011 00:00	0	0	9.84	14.395	81	0.0	3	13	16	1	0	0	0	1	0
01-01-2011 01:00	0	0	9.02	13.635	80	0.0	8	32	40	1	0	0	0	1	0
01-01-2011 02:00	0	0	9.02	13.635	80	0.0	5	27	32	1	0	0	0	1	0
01-01-2011 03:00	0	0	9.84	14.395	75	0.0	3	10	13	1	0	0	0	1	0
01-01-2011 04:00	0	0	9.84	14.395	75	0.0	0	1	1	1	0	0	0	1	0


```
In [16]: # Split datetime into meaningful columns such as hour,day,month,year,etc
         from pyspark.sql.functions import split
         from pyspark.sql.functions import *
         from pyspark.sql.types import *
         bs_df_encoded = bs_df_encoded.withColumn('hour',  split(split(bs_df_encoded['datetime'], ' ')[1], ':')[0].cast('int'))
         bs_df_encoded = bs_df_encoded.withColumn('month', split(split(bs_df_encoded['datetime'], ' ')[0], '-')[0].cast('int'))
         bs_df_encoded = bs_df_encoded.withColumn('day', split(split(bs_df_encoded['datetime'], ' ')[0], '-')[1].cast('int'))
         bs_df_encoded = bs_df_encoded.withColumn('year', split(split(bs_df_encoded['datetime'], ' ')[0], '-')[2].cast('int'))
```

```
In [17]: display(bs_df_encoded.take(5))
```

datetime	holiday	workingday	temp	atemp	humidity	windspeed	casual	registered	count	season_1	season_2	season_3	season_4	weather_1	weather_2
01-01-2011 00:00	0	0	9.84	14.395	81	0.0	3	13	16	1	0	0	0	1	0
01-01-2011 01:00	0	0	9.02	13.635	80	0.0	8	32	40	1	0	0	0	1	0
01-01-2011 02:00	0	0	9.02	13.635	80	0.0	5	27	32	1	0	0	0	1	0
01-01-2011 03:00	0	0	9.84	14.395	75	0.0	3	10	13	1	0	0	0	1	0
01-01-2011 04:00	0	0	9.84	14.395	75	0.0	0	1	1	1	0	0	0	1	0

continued below....

```
In [18]: bs_df_encoded.printSchema()
         bs_df_encoded = bs_df_encoded.drop('datetime')
         bs_df_encoded = bs_df_encoded.withColumnRenamed("count", "label")
```

root -- datetime: string (nullable = true) -- holiday: integer (nullable = true) -- workingday: integer (nullable = true) -- temp: double (nullable = true) -- atemp: double (nullable = true) -- humidity: integer (nullable = true) -- windspeed: double (nullable = true) -- casual: integer (nullable = true) -- registered: integer (nullable = true) -- count: integer (nullable = true) -- season_1: integer (nullable = true) -- season_2: integer (nullable = true) -- season_3: integer (nullable = true) -- season_4: integer (nullable = true) -- weather_1: integer (nullable = true) -- weather_2: integer (nullable = true) -- weather_3: integer (nullable = true) -- weather_4: integer (nullable = true) -- hour: integer (nullable = true) -- month: integer (nullable = true) -- day: integer (nullable = true) -- year: integer (nullable = true)

```
In [19]: #Split the dataset into train and train_test
         from pyspark.ml.tuning import ParamGridBuilder, TrainValidationSplit
         train, test = bs_df_encoded.randomSplit([0.9, 0.1], seed=12345)
```

```
In [20]: #The features are assembled to send it to model
         from pyspark.ml.linalg import Vectors
         from pyspark.ml.feature import VectorAssembler

         assembler = VectorAssembler(
             inputCols=["holiday","workingday","temp","atemp","humidity","windspeed","casual","registered","label","season_1","season_2",'
             outputCol="features")

         output = assembler.transform(train)
         print("Assembled columns 'hour', 'day' etc  to vector column 'features'")
         display(output.take(5))
         print(output.count())
```



```
train_output = output.na.drop()
print(train_output.count())
```

holiday	workingday	temp	atemp	humidity	windspeed	casual	registered	label	season_1	season_2	season_3	season_4	weather_1	weather_2	weather_3
0	0	3.28	2.275	79	31.0009	0	24	24	1	0	0	0	0	0	1
0	0	3.28	3.79	53	16.9979	0	26	26	1	0	0	0	1	0	0

continued below....

```
In [21]: test_output = assembler.transform(test)
print(test_output.count())
train_output = test_output.na.drop()
print(test_output.count())
print("Assembled columns 'hour', 'day' etc  to vector column 'features'")
#.select("features", "clicked")
```

1089 1089 Assembled columns 'hour', 'day' etc to vector column 'features'

```
In [22]: from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.regression import LinearRegression
lr = LinearRegression(maxIter=10)

# Fit the model
lrModel = lr.fit(train_output)
```

```
In [23]: # Print the coefficients and intercept for logistic regression
print("Coefficients: " + str(lrModel.coefficients))
print("Intercept: " + str(lrModel.intercept))
```

Coefficients:
[0.241007000329,0.0240300559307,-0.00329772606512,-0.0038201221511,-0.00311898556861,0.00062578456448,0.563257347413,0.563189130437,0.4367
Intercept: 173.7435412550812


```
In [24]: import pyspark.sql.functions
predictions = lrModel.transform(test_output)\
    .select("features", "label", "prediction")\
    .take(10)
display(predictions)

from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.mllib.evaluation import BinaryClassificationMetrics
# testRDD = test.rdd
# predictionAndLabels = testRDD.map(lambda lp: (float(model.predict(lp.features)), lp.label))
# # Evaluate model
# metrics = BinaryClassificationMetrics(predictionAndLabels)
# f1Score = metrics.fMeasure()
# print(f1Score)
from pyspark.ml.evaluation import RegressionEvaluator
lr_evaluator = RegressionEvaluator(predictionCol="prediction", labelCol="label",metricName="r2")
# print("R Squared (R2) on test data = %g" % lr_evaluator.evaluate(predictions))
```

features	label	prediction
List(0, 21, List(2, 3, 4, 5, 7, 8, 9, 13, 17, 18, 19, 20), List(3.28, 4.545, 53.0, 12.998, 18.0, 18.0, 1.0, 1.0, 7.0, 12.0, 2.0, 2012.0))	18	17.977026052947167
List(0, 21, List(2, 3, 4, 5, 7, 8, 9, 13, 17, 18, 19, 20), List(4.1, 3.03, 39.0, 30.0026, 22.0, 22.0, 1.0, 1.0, 23.0, 8.0, 1.0, 2011.0))	22	22.015787070326525
List(0, 21, List(2, 3, 4, 5, 7, 8, 9, 13, 17, 18, 19, 20), List(5.74, 7.575, 43.0, 11.0014, 28.0, 28.0, 1.0, 1.0, 22.0, 12.0, 2.0, 2012.0))	28	28.058417248633106
List(1, 21, List(), List(0.0, 0.0, 6.56, 6.06, 40.0, 31.0009, 4.0, 92.0, 96.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 18.0, 12.0, 2.0, 2012.0))	96	96.06176876485841
List(1, 21, List(), List(0.0, 0.0, 6.56, 6.82, 40.0, 22.0028, 4.0, 44.0, 48.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 18.0, 9.0, 1.0, 2011.0))	48	47.96614804695298
List(1, 21, List(), List(0.0, 0.0, 6.56, 6.82, 47.0, 19.0012, 5.0, 38.0, 43.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 2.0, 15.0, 1.0, 2012.0))	43	42.88936084849334
List(1, 21, List(), List(0.0, 0.0, 6.56, 6.82, 48.0, 26.0027, 1.0, 24.0, 25.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 3.0, 15.0, 1.0, 2012.0))	25	24.89586081959351
List(1, 21, List(), List(0.0, 0.0, 6.56, 9.85, 59.0, 6.0032, 2.0, 18.0, 20.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 15.0, 1.0, 2011.0))	20	19.91497309035597
List(0, 21, List(2, 3, 4, 5, 6, 7, 8, 9, 13, 18, 19, 20), List(6.56, 9.85, 69.0, 6.0032, 3.0, 27.0, 30.0, 1.0, 1.0, 12.0, 2.0, 2011.0))	30	29.944761523582343
List(0, 21, List(2, 3, 4, 7, 8, 9, 13, 17, 18, 19, 20), List(6.56, 11.365, 59.0, 1.0, 1.0, 1.0, 1.0, 5.0, 15.0, 1.0, 2011.0))	1	0.8497462836939462

In [25]:
```
# Parameter grid search for best parameters to give good predictions
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.regression import LinearRegression
from pyspark.ml.tuning import ParamGridBuilder, TrainValidationSplit
# We use a ParamGridBuilder to construct a grid of parameters to search over.
# TrainValidationSplit will try all combinations of values and determine best model using
# the evaluator.
paramGrid = ParamGridBuilder()\
    .addGrid(lr.regParam, [0.1, 0.01]) \
    .addGrid(lr.fitIntercept, [False, True])\
    .addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0])\
    .build()

# In this case the estimator is simply the linear regression.
# A TrainValidationSplit requires an Estimator, a set of Estimator ParamMaps, and an Evaluator.
tvs = TrainValidationSplit(estimator=lr,
                           estimatorParamMaps=paramGrid,
                           evaluator=RegressionEvaluator(),
                           # 80% of the data will be used for training, 20% for validation.
                           trainRatio=0.8)

# Run TrainValidationSplit, and choose the best set of parameters.
model = tvs.fit(train_output)

# Make predictions on test data. model is the model with combination of parameters
# that performed best.
display(model.transform(test_output)\
    .select("features", "label", "prediction")\
    .take(5))
```


features	label	prediction
List(0, 21, List(2, 3, 4, 5, 7, 8, 9, 13, 17, 18, 19, 20), List(3.28, 4.545, 53.0, 12.998, 18.0, 18.0, 1.0, 1.0, 7.0, 12.0, 2.0, 2012.0))	18	17.99775672379881
List(0, 21, List(2, 3, 4, 5, 7, 8, 9, 13, 17, 18, 19, 20), List(4.1, 3.03, 39.0, 30.0026, 22.0, 22.0, 1.0, 1.0, 23.0, 8.0, 1.0, 2011.0))	22	22.002205520528005
List(0, 21, List(2, 3, 4, 5, 7, 8, 9, 13, 17, 18, 19, 20), List(5.74, 7.575, 43.0, 11.0014, 28.0, 28.0, 1.0, 1.0, 22.0, 12.0, 2.0, 2012.0))	28	28.002288892258296
List(1, 21, List(), List(0.0, 0.0, 6.56, 6.06, 40.0, 31.0009, 4.0, 92.0, 96.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 18.0, 12.0, 2.0, 2012.0))	96	95.99923333047171
List(1, 21, List(), List(0.0, 0.0, 6.56, 6.82, 40.0, 22.0028, 4.0, 44.0, 48.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 18.0, 9.0, 1.0, 2011.0))	48	48.00084264442458

In [26]:
```
# Random Forest Classifier model
from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml.regression import RandomForestRegressor
from pyspark.ml.feature import VectorIndexer
from pyspark.ml.evaluation import RegressionEvaluator
rf = RandomForestRegressor(labelCol="label", featuresCol="features", numTrees=100)
# Train model.  This also runs the indexers.
rf_model = rf.fit(train_output)
# rf_model.persist()
# Make predictions.
predictions = rf_model.transform(test_output)

# Select example rows to display.
display(predictions.select("prediction", "label", "features").take(5))

# Select (prediction, true label) and compute test error
evaluator = RegressionEvaluator(
    labelCol="label", predictionCol="prediction", metricName="rmse")
rmse = evaluator.evaluate(predictions)
print("Root Mean Squared Error (RMSE) on test data = %g" % rmse)
```

prediction	label	features
26.980856309621263	18	List(0, 21, List(2, 3, 4, 5, 7, 8, 9, 13, 17, 18, 19, 20), List(3.28, 4.545, 53.0, 12.998, 18.0, 18.0, 1.0, 1.0, 7.0, 12.0, 2.0, 2012.0))
33.05357800429468	22	List(0, 21, List(2, 3, 4, 5, 7, 8, 9, 13, 17, 18, 19, 20), List(4.1, 3.03, 39.0, 30.0026, 22.0, 22.0, 1.0, 1.0, 23.0, 8.0, 1.0, 2011.0))
36.28714272390532	28	List(0, 21, List(2, 3, 4, 5, 7, 8, 9, 13, 17, 18, 19, 20), List(5.74, 7.575, 43.0, 11.0014, 28.0, 28.0, 1.0, 1.0, 22.0, 12.0, 2.0, 2012.0))
90.99638717240707	96	List(1, 21, List(), List(0.0, 0.0, 6.56, 6.06, 40.0, 31.0009, 4.0, 92.0, 96.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 18.0, 12.0, 2.0, 2012.0))
53.02938717787259	48	List(1, 21, List(), List(0.0, 0.0, 6.56, 6.82, 40.0, 22.0028, 4.0, 44.0, 48.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 18.0, 9.0, 1.0, 2011.0))

```
In [27]: # GBT Regressor model
         from pyspark.ml.regression import GBTRegressor
         gbt = GBTRegressor(featuresCol="features", maxIter=10)

         gbt_model = gbt.fit(train_output)
         # Make predictions.
         predictions = gbt_model.transform(test_output)


         gbt_model.write().overwrite().save("bike_sharing_gbt.model")
         # Select example rows to display.
         display(predictions.select("prediction", "label", "features").take(5))

         # Select (prediction, true label) and compute test error
         evaluator = RegressionEvaluator(
             labelCol="label", predictionCol="prediction", metricName="rmse")
         rmse = evaluator.evaluate(predictions)
         print("Root Mean Squared Error (RMSE) on test data = %g" % rmse)
         #Gave root mean square error
```

prediction	label	features
16.889233655915504	18	List(0, 21, List(2, 3, 4, 5, 7, 8, 9, 13, 17, 18, 19, 20), List(3.28, 4.545, 53.0, 12.998, 18.0, 18.0, 1.0, 1.0, 7.0, 12.0, 2.0, 2012.0))
16.92511614166162	22	List(0, 21, List(2, 3, 4, 5, 7, 8, 9, 13, 17, 18, 19, 20), List(4.1, 3.03, 39.0, 30.0026, 22.0, 22.0, 1.0, 1.0, 23.0, 8.0, 1.0, 2011.0))
30.32523546505164	28	List(0, 21, List(2, 3, 4, 5, 7, 8, 9, 13, 17, 18, 19, 20), List(5.74, 7.575, 43.0, 11.0014, 28.0, 28.0, 1.0, 1.0, 22.0, 12.0, 2.0, 2012.0))
95.83936857912708	96	List(1, 21, List(), List(0.0, 0.0, 6.56, 6.06, 40.0, 31.0009, 4.0, 92.0, 96.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 18.0, 12.0, 2.0, 2012.0))
46.025330660611566	48	List(1, 21, List(), List(0.0, 0.0, 6.56, 6.82, 40.0, 22.0028, 4.0, 44.0, 48.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 18.0, 9.0, 1.0, 2011.0))

* bs_model_generation

This file is used to give a trained model taking training files as input by cleaning them and using one of the algorithms which gave best results GBTRegressor which gave least root mean square error.

```
import pyspark.sql.functions as func
from pyspark.sql.types import *
import os
import json
import pandas as pd
import numpy as np
from datetime import datetime, timedelta, date
from functools import reduce
from pyspark import SparkContext, SparkConf
from pyspark.sql import HiveContext, SQLContext, DataFrame
# from camp_revamp import turingBatch as tb
from pyspark.sql.functions import rand,when

######################################################## Function definitions ########################################################

try:
                ######################################################## Define variables ########################################################
                sc = SparkContext()
                sqlContext = HiveContext(sc)
                bs_df = sqlContext.read.load("train.csv",
                                format='com.databricks.spark.csv',
                                header='true',
                                inferSchema='true')
                bs_df.show()
                print(bs_df.printSchema())

```

```python
def valueToCategory(value, encoding_index):
   if(value == encoding_index):
     return 1
   else:
     return 0
#Explode season column into separate columns such as season_<val> and drop season
from pyspark.sql.functions import udf
from pyspark.sql.functions import lit
from pyspark.sql.types import *
from pyspark.sql.functions import col
udfValueToCategory = udf(valueToCategory, IntegerType())
bs_df_encoded = (bs_df.withColumn("season_1", udfValueToCategory(col('season'),lit(1)))
                     .withColumn("season_2", udfValueToCategory(col('season'),lit(2)))
                     .withColumn("season_3", udfValueToCategory(col('season'),lit(3)))
                     .withColumn("season_4", udfValueToCategory(col('season'),lit(4))))
bs_df_encoded = bs_df_encoded.drop('season')
#https://stackoverflow.com/questions/40161879/pyspark-withcolumn-with-two-conditions-and-three-outcomes

bs_df_encoded = (bs_df_encoded.withColumn("weather_1", udfValueToCategory(col('weather'),lit(1)))
     .withColumn("weather_2", udfValueToCategory(col('weather'),lit(2)))
     .withColumn("weather_3", udfValueToCategory(col('weather'),lit(3)))
     .withColumn("weather_4", udfValueToCategory(col('weather'),lit(4))))
bs_df_encoded = bs_df_encoded.drop('weather')

```

```python
#  hour, day, month, year
from pyspark.sql.functions import split
from pyspark.sql.functions import *
from pyspark.sql.types import *
bs_df_encoded = bs_df_encoded.withColumn('hour',  split(split(bs_df_encoded['datetime'], ' ')[1], ':')[0].cast('int'))
bs_df_encoded = bs_df_encoded.withColumn('year', split(split(bs_df_encoded['datetime'], ' ')[0], '-')[0].cast('int'))
bs_df_encoded = bs_df_encoded.withColumn('month', split(split(bs_df_encoded['datetime'], ' ')[0], '-')[1].cast('int'))
bs_df_encoded = bs_df_encoded.withColumn('day', split(split(bs_df_encoded['datetime'], ' ')[0], '-')[2].cast('int'))
bs_df_encoded.show(10)

bs_df_encoded = bs_df_encoded.drop('datetime')
bs_df_encoded = bs_df_encoded.withColumnRenamed("count", "label")

#Split the dataset into train and train_test
from pyspark.ml.tuning import ParamGridBuilder, TrainValidationSplit
train, test = bs_df_encoded.randomSplit([0.9, 0.1], seed=12345)

from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import VectorAssembler
assembler = VectorAssembler(inputCols=["holiday","workingday","temp","atemp","humidity","windspeed","label","season_1","season_2","season_3","season_4","weather_1","weather_2","weather_3","weather_4", "hour", "year", "month", "day"],outputCol="features")

```

```python
assembler = VectorAssembler(inputCols=["holiday","workingday","temp","atemp","humidity","windspeed","casual","registered","label","season_1","season_2","season_3","season_4","weather_1","weather_2","weather_3","weather_4", "hour", "month", "day", "year"],outputCol="features")

output = assembler.transform(train)
print("Assembled columns 'hour', 'minute' .. to vector column 'features'")
output.show(truncate=False)#.select("features", "clicked")
print(output.count())
train_output = output.na.drop()
print(train_output.count())

test_output = assembler.transform(test)
print(test_output.count())
train_output = test_output.na.drop()
print(test_output.count())
print("Assembled columns 'hour', 'minute' .. to vector column 'features'")
test_output.show(truncate=False)#.select("features", "clicked")

from pyspark.ml.regression import GBTRegressor
gbt = GBTRegressor(featuresCol="features", maxIter=10)

gbt_model = gbt.fit(train_output)
# Make predictions.
predictions = gbt_model.transform(test_output)
path = "bike_sharing_gbt_file.model"
gbt_model.write().overwrite().save(path)
# Select example rows to display.
predictions.select("prediction", "label", "features").show(5)

```

```
            # Select (prediction, true label) and compute test error

            from pyspark.ml.evaluation import RegressionEvaluator
            evaluator = RegressionEvaluator(labelCol="label", predictionCol="prediction", metricName="rmse")
            rmse = evaluator.evaluate(predictions)
            print("Root Mean Squared Error (RMSE) on test data = %g" % rmse)
except Exception as e:
        print(e)
```


* bs_prediction_generator

This file is uses the model generated out of previous file and predicts the bike sharing demand on the test files given. Then finally outputs the predictions as csv file with name prediction.csv.

After importing all the necessary packages

```
                sc = SparkContext()
                sqlContext = HiveContext(sc)
                bs_df = sqlContext.read.load("test.csv",
                                  format='com.databricks.spark.csv',
                                  header='true',
                                  inferSchema='true')
                print("Test data features")
                bs_df.show()
                print(bs_df.printSchema())

                def valueToCategory(value, encoding_index):
                   if(value == encoding_index):
                     return 1
                   else:
                     return 0
                #Explode season column into separate columns such as season_<val> and drop season
                from pyspark.sql.functions import udf
                from pyspark.sql.functions import lit
                from pyspark.sql.types import *
                from pyspark.sql.functions import col
                udfValueToCategory = udf(valueToCategory, IntegerType())
                bs_df_encoded = (bs_df.withColumn("season_1", udfValueToCategory(col('season'),lit(1)))
                                      .withColumn("season_2", udfValueToCategory(col('season'),lit(2)))
                                      .withColumn("season_3", udfValueToCategory(col('season'),lit(3)))
                                      .withColumn("season_4", udfValueToCategory(col('season'),lit(4))))
                bs_df_encoded = bs_df_encoded.drop('season')

```



```
                bs_df_encoded = (bs_df_encoded.withColumn("weather_1", udfValueToCategory(col('weather'),lit(1)))
                     .withColumn("weather_2", udfValueToCategory(col('weather'),lit(2)))
                     .withColumn("weather_3", udfValueToCategory(col('weather'),lit(3)))
                     .withColumn("weather_4", udfValueToCategory(col('weather'),lit(4))))
                bs_df_encoded = bs_df_encoded.drop('weather')

                #  hour, day, month, year
                from pyspark.sql.functions import split
                from pyspark.sql.functions import *
                from pyspark.sql.types import *
                bs_df_encoded = bs_df_encoded.withColumn('hour',  split(split(bs_df_encoded['datetime'], ' ')[1], ':')[0].cast('int'))
                bs_df_encoded = bs_df_encoded.withColumn('year', split(split(bs_df_encoded['datetime'], ' ')[0], '-')[0].cast('int'))
                bs_df_encoded = bs_df_encoded.withColumn('month', split(split(bs_df_encoded['datetime'], ' ')[0], '-')[1].cast('int'))
                bs_df_encoded = bs_df_encoded.withColumn('day', split(split(bs_df_encoded['datetime'], ' ')[0], '-')[2].cast('int'))
                print("Test data features encoded")
                bs_df_encoded.show(10)

                bs_df_encoded = bs_df_encoded.drop('datetime')
                # bs_df_encoded = bs_df_encoded.withColumnRenamed("count", "label")

                #Split the dataset into train and train_test
                from pyspark.ml.tuning import ParamGridBuilder, TrainValidationSplit
                #train, test = bs_df_encoded.randomSplit([0.9, 0.1], seed=12345)

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

```
assembler = VectorAssembler(inputCols=["holiday","workingday","temp","atemp","humidity","windspeed","season_1","season_2","season_3","season_4","weather_1","weather_2","weather_3","weather_4", "hour", "year", "month", "day"],outputCol="features")

output = assembler.transform(bs_df_encoded)
output.show(truncate=False)#.select("features", "clicked")
print(output.count())
test_features = output.na.drop()
print(test_features.count())

test_output = assembler.transform(test)
print(test_output.count())
train_output = test_output.na.drop()
print(test_output.count())
print("Assembled columns 'hour', 'minute' .. to vector column 'features'")
test_output.show(truncate=False)#.select("features", "clicked")

```

```
from pyspark.ml.regression import GBTRegressor, GBTRegressionModel
# gbt = GBTRegressor(featuresCol="features", maxIter=10)
path = "bike_sharing_gbt_file.model"

gbt_model = GBTRegressionModel.load(path)
# Make predictions.

print("Before model creation")
predictions = gbt_model.transform(test_features)
print("After model creation")
predictions.printSchema()
predictions.show()
gbt_model.write().overwrite().save(path)
# Select example rows to display.
from pyspark.sql.functions import col, lit, concat
bs_df.show()
predictions = predictions.withColumn("datetime", bs_df.select("datetime"))
predictions = predictions.withColumn("datetime",concat(col("year"),lit("-"),col("month"),lit("-"),col("day"),lit(" "),col("hour"),lit(":00:00")))
predictions.show()

pred_file =  predictions.select("prediction", "datetime")
spark_df.write.fxormat('com.databricks.spark.csv') \
pred_file.coalesce(1).write.mode("overwrite").csv("prediction.csv")
print("file saved")
```