databricksBicycle Sharing Demand Prediction

```
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.RandomForestRegressorv
```

```
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 dataset in Spark


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

```
+-------------------+------+-------+----------+-------+-----+------+--------+---------+------+----------+-----+
|           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())
```

	summary	datetime	season	holiday	workingday	weather	temp	atemp	humidity
1	count	10886	10886	10886	10886	10886	10886	10886	10886
2	mean	null	2.5066139996325556	0.02856880396839978	0.6808745177291935	1.418427337865148	20.230859819952173	23.65508405291192	61.88645
3	stddev	null	1.1161743093443237	0.16659885062470944	0.4661591687997361	0.6338385858190968	7.791589843987573	8.47460062648494	19.24503
4	min	01-01-2011 00:00	1	0	0	1	0.82	0.76	0
5	max	19-12-2012 23:00	4	1	1	4	41.0	45.455	100

Showing all 5 rows.

Decide which columns should be categorical and then 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)
```

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

	approx_count_distinct(workingday)	approx_count_distinct(windspeed)	approx_count_distinct(registered)	approx_count_distinct(count)	approx_count_distinct(atemp)	approx
1	2	27	726	802	60	4

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")
```

```
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]
```

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

```
+----------------+-------+----------+----+------+--------+---------+------+----------+-----+-------------+-------------+
|        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
```

Check for any missing value in data set and treat it

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

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

```
|       0|     0|      0|         0|      0|   0|    0|       0|        0|     0|         0|    0|
+--------+------+-------+----------+-------+----+-----+--------+---------+------+----------+-----+
```

Untitled

```
//Explode season column into separate columns such as season_and drop season
//Execute the same for weather as weather_ and drop weather

// There's no need to explode season and weather column as we have already applied one-hot-encoder for categorical columns in the dataser with values > 2
```

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

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 set into train and train_test


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

```
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 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
```

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)
```

```
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)
```

```
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
```

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)
```

```
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=false
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)
```

```
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")
```

```
command-439559543007772:3: error: not found: value gbtModel
gbtModel.write.overwrite().save("/FileStore/tables/model/bicycle-model")
^
```

Model Implementation and Prediction

```
// Application Development for Model Generation

// 1. Clean and Transform the data

// 2. Develop the model and persist it.
```

```
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")
  }
}
```

```
command-4395595430077775:20: warning: constructor SQLContext in class SQLContext is deprecated (since 2.0.0): Use SparkSession.builder instead
    val spark = new org.apache.spark.sql.SQLContext(sc)
                ^
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
```

```
//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................")

    //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
  }
}
```

```
command-3715568042741804:20: warning: constructor SQLContext in class SQLContext is deprecated (since 2.0.0): Use SparkSession.builder instead
    val spark = new org.apache.spark.sql.SQLContext(sc)
                ^
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
```

Application for Streaming Data

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


//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


flume-ng agent --conf conf --conf-file bicycle.conf --name agent1 -Dflume.root.logger=DEBUG,console


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


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
```