Apache Spark And Scala Certification Training

Certification Project Solution

edureka!

Bicycle Sharing Demand

Domain – Transportation Industry

Business challenge/requirement

With the spike in pollution levels and the fuel prices, many Bicycle Sharing Programs are running around the world. Bicycle sharing systems are a means of renting bicycles where the process of obtaining membership, rental and bike return is automated via a network of joint locations throughout the city. Using this system people can rent a bike from one location and return it to a different place as and when needed.

Data Set

Data contains hourly rental data spanning two years. Training set comprised of the first 19 days of each month while the test set is the 20th to the end of month.

Considerations

You are building a Bicycle Sharing demand forecasting service that combines historical usage patterns with weather data to forecast the Bicycle rental demand in real-time. To develop this system, 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 Spark Streaming request.

Data Exploration and Transformation

Explore the data and develop the model in Spark Shell

1. Read dataset in Spark
2. Get summary of data and variable types
3. Decide which columns should be categorical and then convert them accordingly
4. Check for any missing value in dataset and treat it
5. Explode season column into separate columns such as season_<val> and drop season
6. Execute the same for weather as weather_<val> and drop weather
7. Split datetime into meaning columns such as hour, day, month, year, etc.
8. Explore how count varies with different features such as hour, month, etc

Model Development

1. Split the dataset into train and train_test.
2. Try different regression algorithms such as linear regression, random forest, etc. and note accuracy.
3. Select the best model and persist it

Model Implementation and Prediction

Application Development for Model Generation

For the above steps wrote write an application to:

1. Clean and Transform the data
2. Develop the model and persist it.

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

Application for Streaming Data

Write an application to predict demand on streaming data:

1. Setup flume to push data into spark flume sink.
2. Configure spark streaming to pull data from spark flume sink using receivers and predict the demand using model and persist the result to RDBMS.
3. Push messages from flume to test the application. Here application should process and persist the result to RDBMS

Data Exploration and Transformation

Load data in HDFS

Transfer data to server using ftp

```
unzip all.zip

hdfs dfs -mkdir use_cases/bike_sharing

hdfs dfs -put train.csv use_cases/bike_sharing
```

Read dataset in Spark

```
val raw = spark.read.option(“header",true).option("inferSchema",true).csv("use_cases/bike_sharing/train.csv")
```

Get summary of data and variable types

```
raw.describe().show
```

Decide which columns should be categorical and then convert them accordingly

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

val discretizer_t = new QuantileDiscretizer().setInputCol("atemp").setOutputCol("atempbin").setNumBuckets(4)

val df_t = discretizer_t.fit(raw).transform(raw).drop("atemp")

val discretizer_w = new QuantileDiscretizer().setInputCol("windspeed").setOutputCol("windspeedbin").setNumBuckets(4)

val df_w = discretizer_w.fit(df_t).transform(df_t).drop("windspeed")

import org.apache.spark.sql.types.DoubleType

val discretizer_h = new QuantileDiscretizer().setInputCol("humidity").setOutputCol("humiditybin").setNumBuckets(4)

val casted = df_w.withColumn("humidity", $"humidity".cast(DoubleType))

val df = discretizer_h.fit(casted).transform(casted).drop(“humidity")
```

Check for any missing value in dataset and treat it

```
val filterCond = df.columns.map(x=>col(x).isNotNull).reduce(_ && _)
val filtered = df.filter(filterCond)

filtered.count
```

Explode season column into separate columns such as season_<val> and drop season

```
var df1 = df

df1.cache

df.select("season").distinct.collect.foreach{s=>df1 = df1.withColumn("season_" + s(0), when($"season" === s(0),1).otherwise(0)
```

Execute the same for weather as weather_<val> and drop weather

```
df1.select("weather").distinct.collect.foreach{s=>df1 = df1.withColumn("weather_" + s(0), when($"weather" === s(0),1).otherwise(0))}
```

Split datetime into meaning columns such as hour, day, month, year, etc.

```
val df3 = df1.withColumn(“year",year($"datetime")).withColumn("month",month($"datetime")).withColumn("hour",hour($"datetime"))
```

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

```
df3.groupBy("workingday").pivot("humiditybin").sum("count").show()

df3.groupBy(“hour").pivot("humiditybin").sum("count").show()
```

Model Development

Assemble features

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

import org.apache.spark.ml.linalg.Vectors

val assembler = new VectorAssembler().setInputCols(Array("holiday","workingday","temp","atempbin","windspeedbin","humiditybin","month","hour")).setOutputCol("features")

val output = assembler.transform(df3).withColumn(“label",$"count")
```

Split the dataset into train and train_test.

```
import org.apache.spark.ml.tuning.{ParamGridBuilder, TrainValidationSplit}

val Array(training, test) = output.randomSplit(Array(0.7, 0.3), seed = 12345)
```

Try different regression algorithms such as linear regression, random forest, etc. and note accuracy.

Linear Regression

```
import org.apache.spark.ml.regression.LinearRegression

val lr = new LinearRegression().setMaxIter(10).setRegParam(0.3).setElasticNetParam(0.8)

val lrModel = lr.fit(training)

val trainingSummary = lrModel.summary

trainingSummary.residuals.show()

println(s"RMSE: ${trainingSummary.rootMeanSquaredError}")

println(s"r2: ${trainingSummary.r2}”)
```

RandomForest

```
import org.apache.spark.ml.Pipeline

import org.apache.spark.ml.evaluation.RegressionEvaluator

import org.apache.spark.ml.feature.VectorIndexer

import org.apache.spark.ml.regression.{RandomForestRegressionModel, RandomForestRegressor}

// Train a RandomForest model.

val rf = new RandomForestRegressor().setLabelCol("label").setFeaturesCol("features")

val model = rf.fit(output)

val predictions = model.transform(test)

val evaluator = new RegressionEvaluator().setLabelCol("label").setPredictionCol("prediction").setMetricName("rmse")

val rmse = evaluator.evaluate(predictions)
```

Select the best model and persist it

```
model.save(“use_cases/model4.4”)
```

Model Implementation and Prediction

Application Development for Model Generation

```
package com.edureka.training.bikesharing
import org.apache.spark.sql.SparkSession
import org.apache.spark.ml.feature.QuantileDiscretizer
import org.apache.spark.sql.types._
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml.linalg.Vectors
import org.apache.spark.ml.Pipeline
import org.apache.spark.sql.functions._
import org.apache.spark.ml.evaluation.RegressionEvaluator
import org.apache.spark.ml.regression.{RandomForestRegressionModel, RandomForestRegressor}
object ModelGenerator {
  def main(args: Array[String]) {
    if (args.length < 1) {
      System.err.println("Usage: ModelGenerator <data path> <model persitence path>")
      System.exit(1)
    }
    val spark = SparkSession
      .builder
      .appName("ModelGenerator")
      .getOrCreate()
    // data and model path
    val dataPath = args(0)
    val modelPath = args(1)
    // load data

    val raw = spark.read.option("header",true).option("inferSchema",true).csv(dataPath)

    // cast column to double

    val casted = raw.withColumn("humidity", col("humidity").cast(DoubleType))

    // explode year into hour and month

    val df =
casted.withColumn("year",year(col("datetime"))).withColumn("month",month(col("datetime"))).withColu
mn("hour",ho$

"))).withColumnRenamed("count","label")

    // rename count to label

    //val output = assembler.transform(df).withColumnRenamed("label","count")

    // categorize columns

val discretizer_t = new
QuantileDiscretizer().setInputCol("atemp").setOutputCol("atempbin").setNumBuckets(4)

    val discretizer_w = new
QuantileDiscretizer().setInputCol("windspeed").setOutputCol("windspeedbin").setNumBuckets(4)

    val discretizer_h = new
QuantileDiscretizer().setInputCol("humidity").setOutputCol("humiditybin").setNumBuckets(4)

    // assemble features

    val assembler = new
VectorAssembler().setInputCols(Array("holiday","workingday","atempbin","windspeedbin","humidit

ybin","month","hour")).setOutputCol("features")

   // model

   val rf = new RandomForestRegressor().setLabelCol("label").setFeaturesCol("features")

   val pipeline = new Pipeline().setStages(Array(discretizer_t, discretizer_w, discretizer_h, assembler, rf))

   // Train model. This also runs the other steps in pipeline

   val model = pipeline.fit(df)

   // save model

   model.write.overwrite().save(modelPath)

   spark.stop()

  }

}
```

Application Development for Demand Prediction

Load Test data in HDFS

hdfs dfs -put test.csv use_cases/bike_sharing

Download Driver jar and include it in spark-shell class path

1. Download the driver jar

- wget https://dev.mysql.com/get/Downloads/Connector-J/mysql-connector-java-8.0.12.tar.gz

2. Extract the tar.gz

tar -zxvf mysql-connector-java-8.0.12.tar.gz

mv mysql-connector-java-8.0.12.jar ../

3. Run spark-shell

spark2-shell --jars mysql-connector-java-8.0.12.jar

Load data in Spark

val test = spark.read.option(“header",true).option("inferSchema",true).csv("use_cases/bike_sharing/test.csv")

Load the persisted model

val casted = test.withColumn("humidity", col("humidity").cast(DoubleType))

// explode year into hour and month

val df = casted.withColumn("year",year(col("datetime"))).withColumn("month",month(col("datetime"))).withColumn("hour",hour(col("datetime")))

// load pipeline model

val pipeline = PipelineModel.read.load(“use_cases/model4.4")

Predict bike demand

val predictions = pipeline.transform(df)

Persist the result to RDBMS

1. Create table in Mysql under database use_cases

create table bike_sharing (

datetime timestamp,

season int,

holiday int,

```
workingday int,

weather int,

temp double,

atemp double,

humidity double,

windspeed double,

year int,

month int,

hour int,

atempbin double,

windspeedbin double,

humiditybin double,

prediction double);
```

2. Persist dataframe to mysql

```
val prop = new java.util.Properties

prop.put("driver", "com.mysql.jdbc.Driver");

prop.put("url", "jdbc:mysql://mysqldb.edu.cloudlab.com/use_cases");

prop.put("user", "labuser");

prop.put("password", "edureka");

predictions.drop("features").write.mode("append").jdbc(prop.getProperty("url"), "bike_sharing", prop);
```

Application for Streaming Data

Setup flume to push data into spark flume sink

For testing purpose we will be using the necat as the source of flume .

1. Create file wh.conf

```
wh.sources = ws

wh.channels = mem

wh.sinks = hd

wh.sources.ws.type = netcat

wh.sources.ws.bind = ip-20-0-21-161.ec2.internal

wh.sources.ws.port = 44444
```

```
# Each sink's type must be defined

wh.sinks.hd.type = hdfs

wh.sinks.hd.hdfs.writeFormat = Text

wh.sinks.hd.hdfs.fileType = DataStream

wh.sinks.hd.hdfs.filePrefix = flumedemo

wh.sinks.hd.hdfs.useLocalTimeStamp = true

wh.sinks.hd.hdfs.path = use_cases/bike_sharing/

wh.sinks.hd.hdfs.rollCount=100

wh.sinks.hd.hdfs.rollSize=0

# Each channel's type is defined.

wh.channels.mem.type = memory

wh.channels.mem.capacity = 1000

wh.channels.mem.transactionCapacity = 100

# Bind source and sink to channel

wh.sinks.hd.channel = mem

wh.sources.ws.channels = mem

wh.channels.mem.capacity = 100
```

2. Run flume

```
flume-ng agent -n wh -c conf -f netcat.conf -    Dflume.root.logger=INFO,console
```

Streaming Application

1. Create new directory for package in the training_project
2. Create new application

```
package com.edureka.training.bikesharing

import org.apache.spark.sql.SparkSession

import org.apache.spark.ml.feature.QuantileDiscretizer

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

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

import org.apache.spark.ml.linalg.Vectors

import org.apache.spark.ml.Pipeline

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

import org.apache.spark.ml.evaluation.RegressionEvaluator
import org.apache.spark.ml.regression.{RandomForestRegressionModel, RandomForestRegressor}

import org.apache.spark._

import org.apache.spark.streaming._

import org.apache.spark.sql.Encoders

import org.apache.spark.ml._

case class Bike(datetime:String,season:Int, holiday:Int, workingday:Int, weather:Int, temp:Double,

atemp:Double,humidity:Double, windspeed:Double)

object BikeStreaming {

  def main(args: Array[String]) {

    val conf = new SparkConf().setAppName("BikeStreaming")

    val ssc = new StreamingContext(conf, Seconds(10))

    val lines = ssc.textFileStream("use_cases/bike_sharing/flume")

    lines.foreachRDD { rdd =>

        val spark=SparkSession.builder().getOrCreate()

        import spark.implicits._

        val rawRdd = rdd.map(_.split(",")).

                map(d=>Bike(d(0).toString,d(1).toInt, d(2).toInt, d(3).toInt,
d(4).toInt,d(5).toDouble,d(6).toDouble, d(7).toDouble, d(8).toDouble))

        val raw = spark.createDataFrame(rawRdd)

        val casted = raw.withColumn("humidity", col("humidity").cast(DoubleType))

        val df =
casted.withColumn("year",year(col("datetime"))).withColumn("month",month(col("datetime"))).

            withColumn("hour",hour(col("datetime")))

        val pipeline = PipelineModel.read.load("use_cases/model4.4")

        val predictions = pipeline.transform(df)

        val prop = new java.util.Properties

        prop.put("driver", "com.mysql.jdbc.Driver");

        prop.put("url", "jdbc:mysql://mysqldb.edu.cloudlab.com/use_cases");

        prop.put("user", "labuser");

        prop.put("password", "edureka");

        predictions.drop("features").write.mode("append").jdbc(

            prop.getProperty("url"), "bike_sharing", prop)
    }
    ssc.start()
    ssc.awaitTermination()
  }
}
```

3. Compile and run the program

```
spark2-submit --jzars mysql-connector-java-8.0.12.jar --class com.edureka.training.bikesharing.BikeStreaming  --deploy-mode client target/scala-2.11/sparkme-project_2.11-1.0.jar
```

Push messages from flume to test the application. Here application should process and persist the result to RDBMS

Run netcat to send messages. Once connected, anything you write in it will be treated as data.

```
nc -lk 44444
```

In case of having any port issues with the server, the app can be tested by putting the files in the streaming HDFS path, where only new files will be considered.