Lab 9

Objectives

- Run machine learning algorithms on big data using MLlib.
- Try out the different components of MLlib.

Prerequisites

- Download the following two files both made available from Kaggle. House prices file and Sentiment data.

Lab Work

I. Project Setup (20 minutes)

In this lab, we will use Scala as a programming language.

1. Follow the project setup part in Lab 5.
2. Add the following dependencies to your `pom.xml` file.

```
<!-- https://mvnrepository.com/artifact/org.apache.spark/spark-sql -->
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-sql_${scala.compat.version}</artifactId>
    <version>${spark.version}</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.apache.spark/spark-mllib -->
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-mllib_${scala.compat.version}</artifactId>
    <version>${spark.version}</version>
</dependency>
```

II. Load Input

First, we will load the input file that we will use throughout the lab. The input file name is passed as the first command line argument `args(0)`.

1. Add the following code stub and make sure it compiles.

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.SparkConf
```

```
import org.apache.spark.ml.{Pipeline, PipelineModel}
import org.apache.spark.ml.classification.{LogisticRegression, LogisticRegressionModel}
import org.apache.spark.ml.evaluation.{BinaryClassificationEvaluator, RegressionEvaluator}
import org.apache.spark.ml.feature.{HashingTF, StringIndexer, Tokenizer, VectorAssembler, Word2Vec}
import org.apache.spark.ml.regression.{LinearRegression, LinearRegressionModel}
import org.apache.spark.ml.stat.Correlation
import org.apache.spark.ml.tuning.{CrossValidator, ParamGridBuilder}

object App {

  def main(args : Array[String]) {
    if (args.length < 2) {
      println("Usage <input file> <algorithm>")
      println("<input file> path to a CSV file input")
      println("<algorithm> is either regression or classification")
    }
    val inputfile = args(0)
    val method = args(1)
    val conf = new SparkConf
    if (!conf.contains("spark.master"))
      conf.setMaster("local[*]")
    println(s"Using Spark master '${conf.get("spark.master")}'")

    val spark = SparkSession
      .builder()
      .appName("CS167 Lab9")
      .config(conf)
      .getOrCreate()

    val t1 = System.nanoTime
    try {
      if (method.equals("regression"))
      {
        // TODO apply regression model on house prices
      } else if (method.equals("classification")) {
        // TODO process the sentiment data
      } else {
        println(s"Unknown algorithm ${method}")
        System.exit(1)
      }
      val t2 = System.nanoTime
      println(s"Applied algorithm $method on input $inputfile in ${(t2 - t1) * 1E-9} seconds")
    } finally {
      spark.stop
    }
  }
}
```

III. Regression model on house pirces (60 minutes)

In the first case study, we will run a regression model on the house prices dataset to build a model that estimates house prices based on some house features. To activate this part of the code, set the command line arguments to:

```
kc_house_data.csv  regression
```

1. Open the file `kc_house_data.csv` in a spreadsheet file to get familiar with the attributes in the file and their values. The first few lines are included here for your reference.

id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	wa
7129300520	20141013T000000	221900	3	1	1180	5650	1	
6414100192	20141209T000000	538000	3	2.25	2570	7242	2	
5631500400	20150225T000000	180000	2	1	770	10000	1	
2487200875	20141209T000000	604000	4	3	1960	5000	1	

id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	wa
1954400510	20150218T000000	510000	3	2	1680	8080	1	
7237550310	20140512T000000	1.225E+06	4	4.5	5420	101930	1	

2. Load the input CSV file using `spark.read` method. Make sure to use the options `header=true` and `inferSchema=true`.
3. Use `printSchema` and `show` to make sure that the file was loaded as you expected.
4. We want to run a regression model that estimates the price based on other house features.
5. Initially, we will use the following features to estimate the model `bedrooms`, `bathrooms`, `sqft_living`, and `sqft_lot`.
6. Use `VectorAssembler` to combine the above features into a `Vector` column. We will call the created column `features`.
7. Create a `LinearRegression` model and the set the features column to `features` and the label column to `price`.
8. At this point, you can run your model by using something similar to the following code snippet.

```
val model = linearRegression.fit(vectorAssembler.transform(input))
```

This code snippet will run the model using the default parameters using the entire input as training data. 9. Once you get the model, you can print the coefficients of the parameters using the following code snippet.

```
println(model.coefficients)
```

10. Now, let us run the same model in a more systematic way by choosing good parameter models. First, create a pipeline that combines the two stages `VectorAssembler` and `LinearRegression`. THe code will look like

```
val pipeline = new Pipeline()
  .setStages(Array(vectorAssembler, linearRegression))
```

11. Next, we will define some parameters to let the model choose from. Create a `ParamGridBuilder` and configure it as follows.

Parameter	Values
lineRegression.regParam	Array(0.01, 0.1, 0.3)
linearRegression.elasticNetParam	Array(0.0, 0.3, 0.8, 1.0)

12. After that, create a `CrossValidator` and configure it as follows.

Method	Parameter
setEstimator	pipeline
setEvaluator	new RegressionEvaluator().setLabelCol("price")
setEstimatorParamMaps	paramGrid
setNumFolds	5
setParallelism	2

13. Now, we will separate the input dataset into training and test sets. The following code snippet will create 80% and 20% training and test sets, respectively.

```
val Array(trainingData, testData) = input.randomSplit(Array(0.8, 0.2))
```

14. Finally, we can apply let the cross validator run on the training set to search for the best model.

```
val model = crossValidator.fit(trainingData)
```

Note: If you would like to find the parameters that were eventually chosen in the best model, using the following code snippet.

```
val chosenModel = model.bestModel.asInstanceOf[PipelineModel].stages(1).asInstanceOf[LinearRegressionModel]
println(s"ElasticNetParam: ${chosenModel.getElasticNetParam}")
println(s"regParam: ${chosenModel.getRegParam}")
println(s"coefficients: ${chosenModel.coefficients}")
```

15. Now, since we have the model, we can apply it on the test set to predict the prices as below.

```
val predictions = model.transform(testData)
predictions.select("price", "prediction").show(5)
```

The output might look like the following.

price	prediction
402500	372965.9079659164
520000	353853.8000358873
635000	966140.5817349589
872750	811021.4795956322
843000	894930.4252227778

16. At this point, we can evaluate the model using standard methods. Let us create an instance of `RegressionEvaluator` and use it to evaluate the model on the test data.

```
val rmse = new RegressionEvaluator()
  .setLabelCol("price")
  .setPredictionCol("prediction")
  .setMetricName("rmse")
  .evaluate(predictions)
println(s"RMSE on test set is $rmse")
```

17. As another metric, we can calculate the correlation between the `price` and `prediction` on the test set. This will use the statistics package from MLlib.

```
val corr = Correlation.corr(new VectorAssembler()
  .setInputCols(Array("prediction", "price"))
  .setOutputCol("features2")
  .transform(predictions), "features2").head
println(s"Correlation is $corr")
```

The output might look like the following:

```
Correlation is [1.0                 0.7045238033214848
0.7045238033214848  1.0                 ]
```

18. Bonus (+3 points). We would like to test if another set of features would provide a more accurate model. Modify your code to try all the following sets of features and choose the one that yields the best model. (Q1) In the README file,

explain how you did it and include a code snippet that highlights the code that you added to make it work. (Q2) Which one provided the best model for you?

```
Try the following sets of features.
Array("bedrooms", "bathrooms", "sqft_living", "sqft_lot"),
Array("bedrooms", "bathrooms", "sqft_living", "sqft_lot", "condition", "waterfront", "view"),
Array("sqft_living", "sqft_lot", "sqft_above", "sqft_basement", "sqft_living15", "sqft_lot15"),
Array("bedrooms", "bathrooms", "condition", "waterfront", "view", "sqft_living", "sqft_lot", "sqft_above",
"sqft_basement", "sqft_living15", "sqft_lot15"),
```

IV. Logistic model on sentimate data (60 minutes)

In this second task, we will build a model that estimates the sentiment (positive/negative) of textual data based on labeled data. To activate this part of the code, set the command line arguments to: `sentiment.csv classification`

1. Open the file `sentiment.csv` in a text editor and go through it to see some examples of the data. For your reference, the first few lines are included below.

text	sentiment
My daughter liked it but I was aghast, that a character in this movie smokes. ...	neg
I... No words. No words can describe this. I will try for the sake of those ...	neg
this film is basically a poor take on the old urban legend of the babysitter ...	neg
This is a terrible movie, and I'm not even sure why it's so terrible. It's ugly, ...	neg
First of all this movie is a piece of reality very well realized artistically. ...	pos

2. Load the file input file `sentiment.csv` as a CSV file using the following options.

option	value
header	true
quote	"
escape	"

Print the schema and the first few lines to double check that the file was loaded correctly. It should look like the following.

```
root
 |-- text: string (nullable = true)
 |-- sentiment: string (nullable = true)

+--------------------+---------+
|                text|sentiment|
+--------------------+---------+
|My daughter liked...|      neg|
|I... No words. No...|      neg|
|this film is basi...|      neg|
|This is a terribl...|      neg|
|First of all this...|      pos|
+--------------------+---------+
```

3. We will build a pipeline that consists of three transformations and one estimator described below.
4. First, we use a `Tokenizer`, which splits the text into words, and configure it as below.

method	parameter
setInputCol	text

method	parameter
setOutputCol	words

5. Second, we use a `HashingTF`, which maps the words in each review to a vector of TF-IDF, and configure it as below.

method	parameter
setInputCol	words
setOutputCol	features

6. Third, we use a `StringIndexer`, which maps the two sentiments `pos` and `neg` to integer values, and configure it as below.

method	parameter
setInputCol	sentiment
setOutputCol	label
setHandleInvalid	skip

7. Fourth, we run the data through a `LogisticRegression` model. The default configuration is fine as it uses the `features` and `label` columns as features and label columns, respectively.

8. Create a pipeline that combines the four stages in the above order.

9. Create a parameter grid that tries the following parameters.

parameter	values
hashingTF.numFeatures	Array(10, 100, 1000)
logisticRegression.regParam	Array(0.01, 0.1, 0.3, 0.8)

10. Working with this dataset will take more time because it is bigger and more complex. To speed up the training process, we will use a `TrainValidationSplit` validator. We will create and configure it as below:

```
val cv = new TrainValidationSplit()
  .setEstimator(pipeline)
  .setEvaluator(new BinaryClassificationEvaluator())
  .setEstimatorParamMaps(paramGrid)
  .setTrainRatio(0.8)
  .setParallelism(2)
```

11. Split the input data into training and test sets as below.

```
val Array(trainingData, testData) = sentimentData.randomSplit(Array(0.8, 0.2))
```

12. Run through the data and get the best model as follows.

```
val logisticModel = cv.fit(trainingData)
```

Note: If you want to get the parameters used by the best model, run the following code snippet.

```
val numFeatures = logisticModel.bestModel.asInstanceOf[PipelineModel].stages(1).asInstanceOf[HashingTF].getNumFeat
val regParam = logisticModel.bestModel.asInstanceOf[PipelineModel].stages(3).asInstanceOf[LogisticRegressionModel]
```

13. Now, apply the the model on the test set and print out a few cases for demonstration.

```
val predictions = logisticModel.transform(testData)
predictions.select("text", "label", "prediction", "probability").show()
```

The output should look similar to the following:

```
+--------------------+-----+----------+--------------------+
|                text|label|prediction|         probability|
+--------------------+-----+----------+--------------------+
|"Air Bud 2: Golde...|  1.0|       0.0|[0.68380935570389...|
|"Cover Girl" is t...|  0.0|       0.0|[0.72511721634649...|
|"Dominion" is a g...|  0.0|       0.0|[0.55437404786371...|
|"Dungeon of Harro...|  1.0|       1.0|[0.43665553614488...|
|"Everything is Il...|  0.0|       0.0|[0.92100407854036...|
|"Hoods" doesn't d...|  1.0|       1.0|[0.07458240427025...|
|"I haven't laughe...|  1.0|       1.0|[0.48317031993762...|
|"Jaded" offers a ...|  1.0|       1.0|[0.30711688918009...|
|"One True Thing" ...|  0.0|       0.0|[0.62604014511659...|
|"Problem Child" w...|  1.0|       1.0|[0.07038002571342...|
|"Summer of My Ger...|  0.0|       0.0|[0.85532203845878...|
|"Tart" is a pathe...|  1.0|       1.0|[0.47321342214712...|
|"The Ballad of th...|  1.0|       1.0|[0.45337525076680...|
|"The Deer Hunter'...|  0.0|       0.0|[0.57247042379411...|
|"The Phenix City ...|  0.0|       0.0|[0.85034976542746...|
|"The Spirit of St...|  0.0|       0.0|[0.90987968937279...|
|"The missing star...|  0.0|       0.0|[0.51433765691704...|
|"Three" is a seri...|  1.0|       1.0|[0.37648809982384...|
|"Tourist Trap" is...|  0.0|       0.0|[0.61016012355390...|
|"War is in your b...|  1.0|       1.0|[0.26858342080044...|
+--------------------+-----+----------+--------------------+
```

14. Let us also calculate the accuracy of the result.

```
val binaryClassificationEvaluator = new BinaryClassificationEvaluator()
  .setLabelCol("label")
  .setRawPredictionCol("prediction")

val accuracy = binaryClassificationEvaluator.evaluate(predictions)
println(s"Accuracy of the test set is $accuracy")
```

V. Submission (15 minutes)

1. Add a README that contains your information.
2. Add a run script file that compiles your code and runs both examples on the two input files.
3. Package your `README` file, run script, and code (without test files or binary files) and submit as a compressed file.

Hints

Spark tends to be very chatty on the console. If you would like to reduce the amount of logs written by Spark, create a file named `log4j.properties` under `src/main/resources` and include the following configuraiton in it.

```
# Set root logger level to INFO and its only appender to A1.
log4j.rootLogger=ERROR, A1

# A1 is set to be a ConsoleAppender.
log4j.appender.A1=org.apache.log4j.ConsoleAppender

# A1 uses PatternLayout.
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-4r [%t] %-5p %c %x - %m%n
```