ID2223 – Lab 1
Linear Regression with SparkML

Introduction

Goals

This lab has the following goals:

- Learn how to work with Spark RDDs and the Dataframe API
- Learn how to program a scalable machine learning pipeline in Spark
- Use Spark's implementation of Linear Regression in a pipeline
- Implement your own spark abstraction - Linear Regression with Gradient Descent
- Learn how to run your program against a small data set for testing your code as well as a very big dataset in a cluster.

Requirements

For being able to do this lab completely you have to have an IDE environment setup. We recommend you to use Scala IDE because we can support it better. Apache Zeppelin is an easy alternative but you cannot use it to program the entire assignment. Throughout the lab we use Spark version 2.0 with its new ML API. Therefore, make sure all the classes you use are from the *ml* package and not from the older *mlib* package (which is now deprecated). We provide the boiler-plate code in this github repo to help you get started.

Million Song Dataset

For this lab we will use a dataset called *million songs*. You can read up more on it at here. While you are developing your ML pipeline, we recommend that you can use a tiny version of the dataset which is 1.2MB big from this link. The full 200 GB dataset and a 1.9 GB subset of it are available as public datasets on www.hops.site.

Spark Terminology

Data Types A *RDD* is a resilient distributed collection of records. Spark applies higher order functions (e.g map, reduce, flatmap, filter) in parallel on a RDD and generates a new RDD in a consistent manner. A *Dataset* is a strongly typed RDD that is used in SparkSQL for supporting sql like queries and query optimizations. A *DataFrame* is a Dataset organized into named columns. A Dataframe is conceptually equivalent to a table in a relational database.

Operations A *Transformer* is a lazy operation on a RDD that generates one or many new RDDs. An *Estimator* is an abstraction of learning algorithms that fits a model on a dataset. Typically, an estimator produces a model for a given dataframe. A *Model* is a transformer itself, generating as output a new dataframe that adds a new prediction column to the input dataframe. A *Pipeline* forms a workflow (Directed Acyclic Graph) out of many transformers and estimators. A pipeline is itself an Estimator, so it will create a model based on the input DataFrame and then the model can generate a new output DataFrame.

1 Load and Inspect the Data

In this task you should load the provided tiny dataset and inspect different aspects of it. To do so, first you need to read the dataset in a RDD and DataFrame. Then you should write higher order functions or sql queris to answer the questions about the data. Use the code skeleton in Listing 1 and complete the annotated sections step by step. For the first six tasks, you will run the tasks on only three features to ensure that the experiments are fast.

Listing 1: "Load and Inspect Skeleton"

```
1  case class Song(year: Double, f1: Double, f2: Double, f3: Double)
2  def main(args: Array[String]) {
3    val sqlContext = new SQLContext(sc)
4
5    import sqlContext.implicits._
6    import sqlContext._
7
8    val rdd = sc.textFile(filePath)
9
10 //Step1: print the first 5 rows, what is the delimiter, number of features ↙
       ↳ and the data types?
11   println(rdd.???)
12
13 //Step2: split each row into an array of features
14   val recordsRdd = rdd.map(???)
15
16 //Step3: map each row into a Song object by using the year label and the ↙
       ↳ first three features
17   val songsRdd = recordsRdd.map(???)
```

```
18
19 //Step4: convert your rdd into a datafram
20   val songsDf = songsRdd.???
21 }
```

Questions: After you have successfully loaded the raw data into the DataFrame of songs, explore the DataFrame's contents to answer the following questions. Try to answer each question both by writing a higher order function (eg: map, reduce, filter and etc.) and SQL query (eg: select, group by).

1. How many songs there are in the DataFrame?
2. How many songs were released between the years 1998 and 2000?
3. What is the min, max and mean value of the year column?
4. Show the number of songs per year between the years 2000 and 2010?

2 Data Preparation Pipeline

Pipelines are the preferred way for data transformations as they allow Spark to perform more optimizations in running a DAG of transformers and estimators. In this task, you will develop a pipeline for transforming the DataFrame of raw data into a DataFrame of label and features vector. The code skeleton for this task is provided in Listing 2, which is a combination of spark's existing transformers with the user defined ones. The user defined transformers are provided in the lab skeleton.

A transformer is a function having one or more columns as input and adding one or more columns to a DataFrame as output. As you instantiate each transformer, test it individually and show a few rows of the generated DataFrame to make sure that it has performed correctly. When all of the transformers work, connect them together as pipeline stages and wire the output column of each transformer to the inputs of its subsequent transformers.

Almost all learning algorithms in spark work with Vector[Double] datatype. That's why, for instance, you have to convert array of tokens to a vector of tokens (line 11). In learning problems, it is often natural to shift labels such that they start from zero. That's why you should shift all labels (line 21) such that smallest label equals zero.

For simplicity and fast development, you are using only the first three features in your experiments (line 23). Later when you have the linear regression pipeline developed, you might want to improve your learning model by using all of the features. As an alternative you can use PolynomialExpansion to expand your feature set.

Listing 2: "Pipeline Skeleton"

```
1  //Step1: tokenize each row
2  val regexTokenizer = new RegexTokenizer()
```

```
3   .setInputCol(???)
4   .setOutputCol(???)
5   .setPattern(???)
6
7   //Step2: transform with tokenizer and show 5 rows
8   ???
9
10  //Step3: transform array of tokens to a vector of tokens (use our ⬋
        ↳ ArrayToVector)
11  val arr2Vect = new Array2Vector()
12  ???
13
14  //Step4: extract the label(year) into a new column
15   val lSlicer = ???
16
17  //Step5: convert type of the label from vector to double (use our ⬋
        ↳ Vector2Double)
18  val v2d = new Vector2Double(???)
19                    ???
20  //Step6: shift all labels by the value of minimum label such that the ⬋
        ↳ value of the smallest becomes 0 (use our DoubleUDF)
21  val lShifter = new DoubleUDF(???)
22                       ???
23  //Step7: extract just the 3 first features in a new vector column
24  val fSlicer = ???
25
26  //Step8: put everything together in a pipeline
27  val pipeline = new Pipeline()
28                .setStages(???)
29
30  //Step9: generate model by fitting the rawDf into the pipeline
31  val pipelineModel = pipeline.fit(rawDF)
32
33  //Step10: transform data with the model
34  ???
35
36  //Step11: drop all columns from the datafram other than label and features
37  ???
```

3 Basic Linear Regression

In this section you are introduced the following Spark abstractions:

- LinearRegression and LinearRegressionModel. The Linear Regression is an Estimator that will produce a LinearRegressionModel when its `fit` method is called. The `fit` method takes a DataFrame as a parameter - the testing data. The model will afterwards be used to make predictions on previously unseen data - the testing data, using its `transform` method. The `transform` method takes a DataFrame as input(the testing data) and returns a DataFrame that contains an extra column compared to the input, called predictions.

- LinearRegressionSummary. The LinearRegressionModel contains a `summary` field containing metrics that reflect the behaviour of the model on the training data.

Task

Create a LinearRegression instance and set the input columns:

- label column
- features column

as well as the basic parameters:

- maximum iterations - set it to 10 or 50
- regularization parameter - set it to 0.1 or 0.9
- elastic net parameter - set it to 0.1

Add the linear regression learning algorithm to your pipeline and use the testing data in order to generate a pipeline model. Afterwards, use the pipeline model to perform predictions on the testing data. Print the first five lines of the resulting DataFrame and compare the predictions with the actual labels of the data. Extract the summary and inspect the behaviour of the model on the training data by printing the RMSE. Run these steps with the LinearRegression set in the 4 configurations possible from the parameter description.

Listing 3: "Basic Linear Regression Skeleton"

```
1  //set the required paramaters
2  val learningAlg = new LinearRegression().(???)
3  //set appropriate stages
4  val task3Pipeline = new Pipeline().(???)
5  //fit on the training data
6  val task3PipelineModel = task3Pipeline.(???)
7  //get model summary and print RMSE
8  val task3ModelSummary = ↙
       ↳ task3Pipeline.stage(???).asInstanceOf[LinearRegressionModel].(???)
9  println(???)
10 //make predictions on testing data
11 val predictions = task3PipelineModel.(???)
12 //print predictions
13 ???
```

4 Model selection and hyper-parameter tuning

Hyper-parameters are parameters that are not directly learnt within estimators. Such parameters are the number of iterations or the regularization parameter that are used to

configure the learning algorithm. It is possible and recommended to search the hyper-parameter space for the combination of hyper-parameters that produces the best learning model from your training data.

You can read here a base description of spark's model selection and hyperparameter tuning.

In this section you are introduced the following spark abstractions:

- Evaluator - An evaluator is a transformation that maps a DataFrame into a metric indicating how good a model is. In particular we will use the RegressionEvaluator. Reference
- ParamGridBuilder - sets the grid in which the search for the hyper-parameter will take place
- The CrossValidator searches the hyper-parameter grid space and trains a model that is assessed by the Evaluator. When calling the `fit` method on the CrossValidator, a CrossValidatorModel is produced.This model has a field `bestmodel` containing, as the name suggests the best model the validator found. Reference

Task

Use the linear regression instance from the previous task with the same base values. Create a ParameterGridBuilder and add grids for the *maxIter* and *regParam* of the estimator. For each of the parameter chose three values above the base value and three bellow. Create a CrossValidator instance and set your pipeline as the estimator and also set the evaluator and the estimatorParamMaps(which is the search grid space). Inspect the new, best model summary and print again the RMSE and compare with the previous results.

Listing 4: "Hyper-parameter tuning skeleton"

```
//create pipeline
???
//build the parameter grid by setting the values for maxIter and regParam
val paramGrid = new ParamGridBuilder().(???)
val evaluator = new RegressionEvaluator
//create the cross validator and set estimator, evaluator, paramGrid
val cv = new CrossValidator().(???)
val cvModel = cv.(???)
val bestModelSummary = cvModel.bestModel.asInstanceOf[PipelineModel].
   stages(???).asInstanceOf[LinearRegressionModel]
//print best model RMSE to compare to previous
println(???)
```

5 Add 2-way interactions

So far, we've used the features as they were provided. Now, we will add features that capture the two-way interactions between our existing features in order to see if we detect new patterns. If the initial features are $\{x, y, z\}$, the two way interaction set would be $\{x^2, xy, y^2, yz, z^2, zx\}$.

In this section you will introduce the following spark abstractions:

- PolynomialExpansion - this transformer will be used, with a degree of 2 to generate the two way interactions of our features.

Task

Add the PolynomialExpansion transformer in you pipeline and use its 2-way interaction features to generate a new best model using the cross validator setup from before. Print the RMSE of your best model and compare it with previous.

Listing 5: "2-way interactions skeleton"

```
1 //create the polynomial expansion transformer
2 val polynomialExpansionT = new ↙
      ↳ PolynomialExpansion().setInputCol(???).setOutputCol(???).setDegree(2)
3 //create pipeline and set stages
4 ???
5 //setup cross validator
6 ???
7 //print best model RMSE to compare to previous
8 println(???)
```

6 Linear Regression with Gradient Descent

In this section you will develop your own linear regression with gradient descent algorithm as a brand new Estimator. The gradient descent update for linear regression is:

$$w_{i+1} = w_i - \alpha_i \sum_{j=1}^{n} (w_i^\top x_j - y_j) x_j$$

where i is the iteration number of the gradient descent algorithm, and j identifies the observation. w represents an array of weights that is the same size as the array of features and provides a weight for each of the features when finally computing the label prediction in the form:

$$prediction = w^\top \cdot x$$

where w is the final array of weights computed by the gradient descent, x is the array of features of the observation point and *prediction* is the label we predict should be associated to this observation.

Implementation

In order to simplify things a bit we will use a pre-existing User Defined Vector Type - VectorUDT from spark. However this one is available only with spark (it is spark protected), so we will use create an alias(VectorType) for it in a scala package object. This is a well known hack and is provided for you in the project:

```
1 package org.apache.spark
2
3 package object hacks {
4   type VectorType = org.apache.spark.ml.linalg.VectorUDT
5 }
```

Provided files that do not require any change:

- org.apache.spark.hack.package.scala - hack import of private type VectoryUDT
- se.kth.spark.task6.MyLinearRegresion - interfaces for MyLinearRegression and MyLinearModel

The parts where you will have to fill in blanks are:

- se.kth.spark.task6.VectorHelper - scala object with vector helper methods
- se.kth.spark.task6.MyLinearRegressionImpl - your linear regression with gradient descent implementation
- se.kth.spark.task6.TaskDriver - main class of this task creates and runs the pipeline with your algorithm implementation

Tasks

6.1 Vector helper methods

In class se.kth.spark.task6.VectorHelper implement all the helper methods:

- dot - implement the dot product of two vectors and the dot product of a vector and a scalar
- sum - implement addition of two vectors
- fill - create a vector of predefined size and initialize it with the predefined value

The interface of Vector is quite limiting, so a solution where you call the .toArray() to get an array out of it and use functional combinators like map on this array and create a new DenseVector as result will be accepted.

6.2 Root Mean Square Error

Implement the `rmse` method that computes the Root Mean Square Error of a given RDD of tuples of (label, prediction) using the formula:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}(label - prediction)^2}{n}}$$

6.3 Prediction

This method receives as input a RDD of observation instances and for each instance should apply the predict formula below and in the end return an RDD of (label, prediction).

$$prediction = w^{\top} \cdot x_j$$

where x is the feature set of each observation and w is the provided weights

6.4 Gradient summand

To simplify the computation of gradient descent, you should fill in the `gradientSummand` method which computes the formula:

$$gs_{ij} = (w_i^{\top} x_j - y_j) x_j$$

6.5 Gradient

In the `gradient` method you should now compute the simplified formula:

$$gradient = \sum_{j=1}^{n} gs_{ij}$$

6.6 Linear Regression

The linear regression method should be to put together the other smaller helper methods you have written so far.

Listing 6: "Provided Linear Regression"

```
for (i <- 0 until numIters) {
   //compute this iterations set of predictions based on our current weights
   val labelAndPredictRDD = predict(w, trainingData)
   //compute this iterations RMSE
   errorTrain(i) = rmse(labelAndPredictRDD)
   //compute the gradient
   val gradient = gradient(trainingData, w)
   //update the descent direction - the alpha
```

```
9     val alpha_i = alpha / (n * scala.math.sqrt(i + 1))
10    //update weigths based on gradient and alpha
11    val wAux = VectorHelper.dot(gradient, (-1) * alpha_i)
12    w = VectorHelper.sum(w, wAux)
13  }
14  (w, errorTrain)
```

Task summary:

- implement VectorHelper methods
- implement rmse method
- implement predict method
- implement gradientSummand and gradient method
- replace the spark LinearRegresion estimator with your new MyLinearRegression estimator in the task 3 pipeline and inspect the RMSE of our naive model.

7 Run Spark pipeline on Big Data - bonus task (25% extra)

Based on task 4 with hyperparameters you evaluated as being best (iterNum and regParam), run the learning pipeline on the tiny dataset, with all twelve features, the small sized (1.8 GB) dataset (millionsongsubset.tar.gz) and then on the big (200 GB) dataset (consisting of files A.tar.gz to Z.tar.gz). You will need to re-write your code for the big dataset to handle all 26 different input files.

8 Acknowledgements

Thanks to Ali Ghodsi, CEO of Databricks, for help with clarifying issues regarding this lab.