Atividade em Laboratório Virtual 02

Disciplina	FAM – Fundamentos de Aprendizado de Máquina

Objetivos

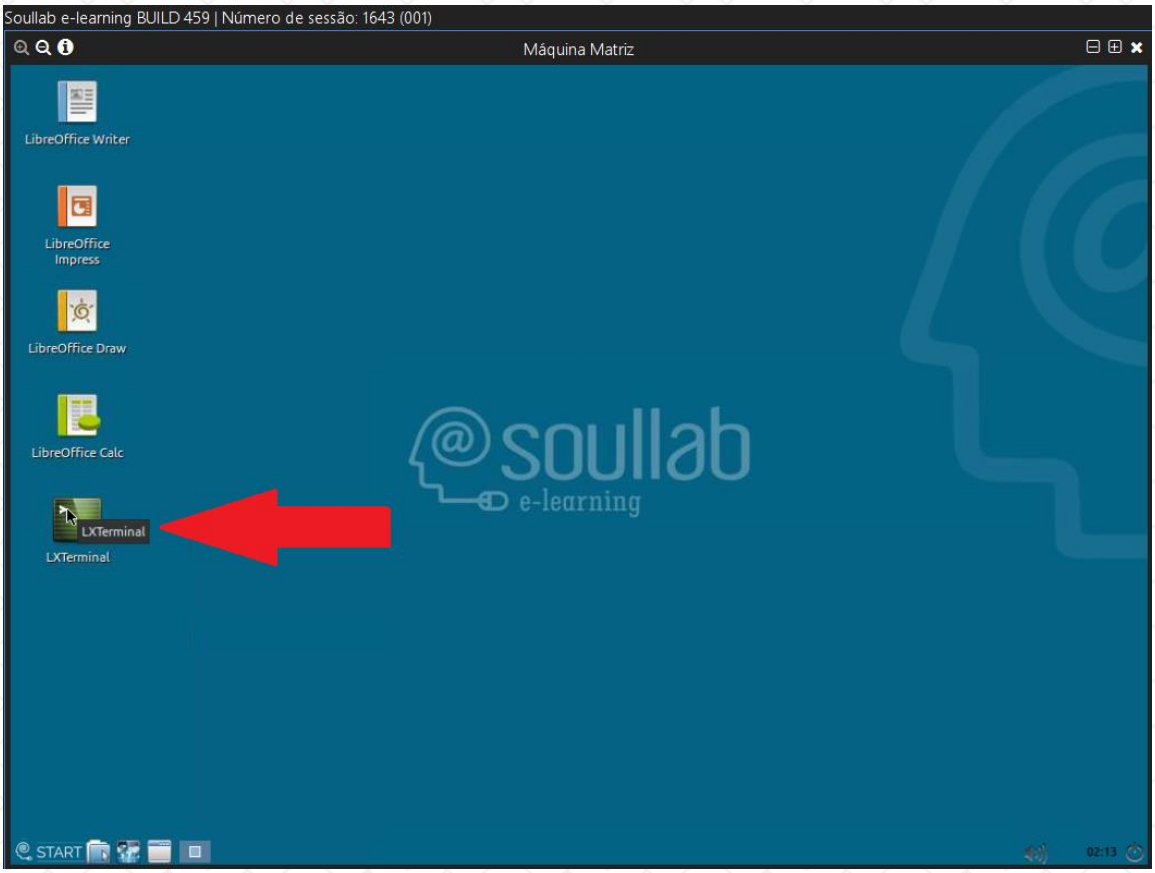
A segunda atividade prática em laboratório virtual possui como objetivo principal o uso do algoritmo de Regressão Linear por meio do framework Apache Spark e da biblioteca de Machine Learning MLlib.

Ao final desta atividade, o aluno deverá ser capaz de executar um experimento com o algoritmo de Regressão Linear utilizando o Apache Spark e a linguagem Scala.

Enunciado

No momento de iniciar esta atividade é esperado que o aluno já tenha recebido seus dados de login para acesso ao laboratório virtual.

Após executar o login no laboratório virtual, inicie o Terminal do Linux, conforme a imagem abaixo:



Foi criado, no ambiente virtual, um usuário exclusivo para trabalhar com o Spark. O nome desse usuário é **spuser** e a sua senha é **spark**. Você deve fazer login com o usuário **spuser** e, para isso, execute o comando abaixo:

su **spuser**

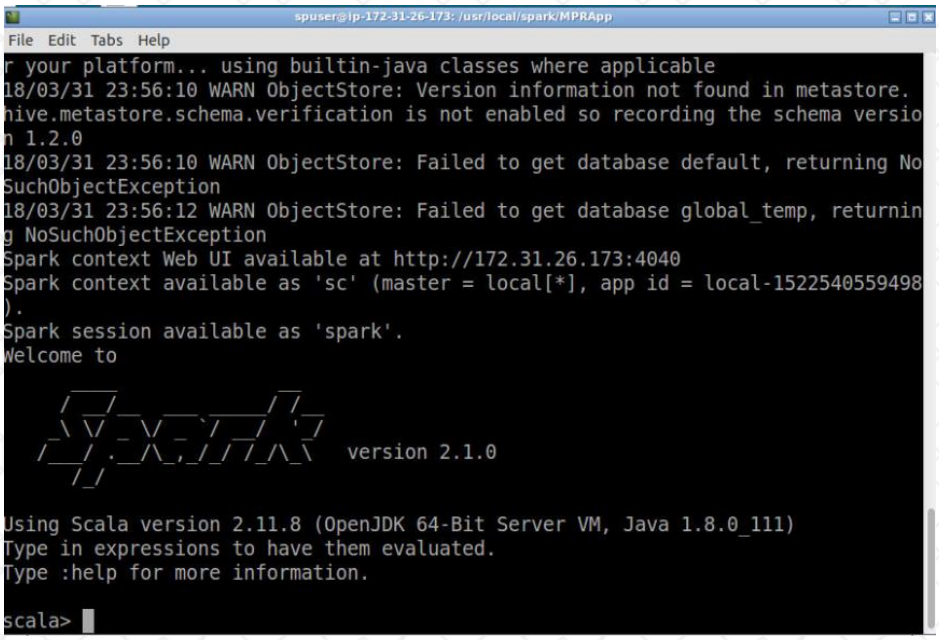
Após digitar o comando acima, o sistema operacional irá solicitar a digitação da senha. Informe a senha "**spark**" (desconsidere as aspas).

A ferramenta Spark já foi previamente instalada no ambiente virtual e encontra-se no diretório `/usr/local/spark`.

Em seguida você deverá executar o Spark Shell. Para isso utilize o seguinte comando:

```
/usr/local/spark/bin/spark-shell
```

Nesse momento a seguinte o Spark Shell será carregado e o seguinte resultado aparecerá na tela:

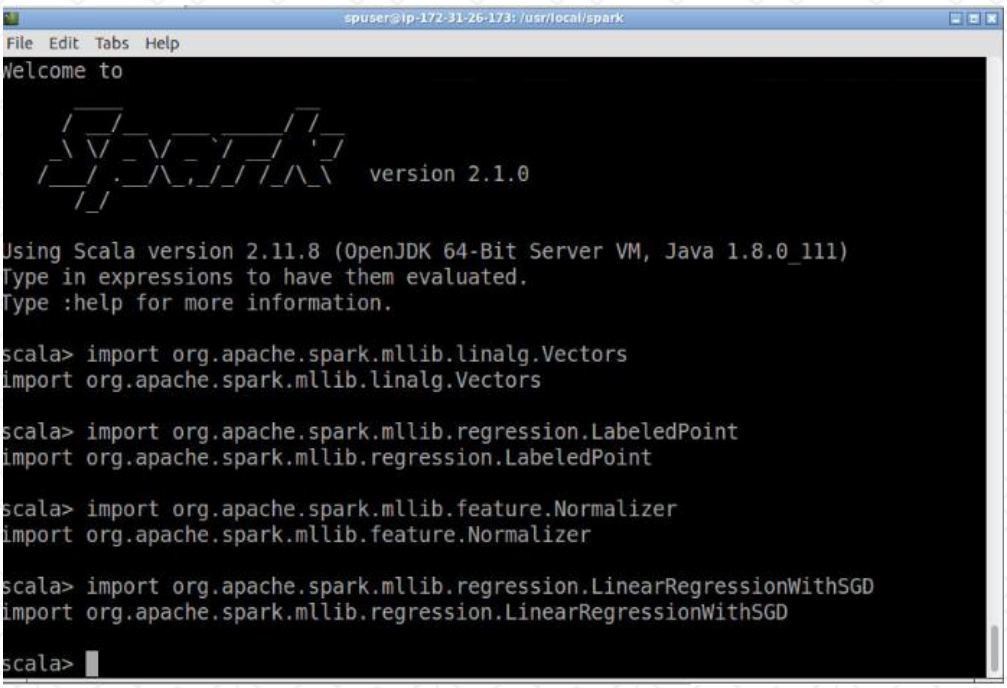


Os dados que serão utilizados no experimento já foram baixados e se encontram no diretório `/usr/local/exemplosml/Atividade2/Dados/lpsa.dat`. Nesse momento o significado desses dados não tem tanta importância, pois nós queremos apresentar como é feita a Regressão Linear e precisamos de alguns dados que são correlacionados.

Primeiramente, já dentro do Spark Shell, vamos incluir as bibliotecas necessárias para execução do experimento. São elas:

```
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.feature.Normalizer
import org.apache.spark.mllib.regression.LinearRegressionWithSGD
```

Cada uma dessas linhas, que representam as bibliotecas do MLLib, deverão ser digitadas no Spark Shell, conforme a figura abaixo:



Nesse momento nós vamos utilizar o método *textFile* para ler o arquivo lpsa.dat, cujas primeiras linhas são apresentadas na figura abaixo. Nesse conjunto de dados, *y* é o valor que queremos calcular (a variável dependente) e é o **primeiro campo do arquivo**. Os outros campos são as variáveis independentes. Portanto, ao invés de termos apenas *y* = *mx* + *b*, onde x é uma variável de entrada, aqui teremos muitas variáveis *x*.

```
-0.4307829,-1.63735562648104 -2.00621178480549 -1.86242597251066
-1.02470580167082 -0.522940888712441 -0.863171185425945
-1.04215728919298 -0.864466507337306
```

Para criar um RDD através do arquivo lpsa.dat, utilize no Spark Shell o seguinte comando:

```
val data = sc.textFile("/usr/local/exemplosml/Atividade2/Dados/lpsa.dat")
```

A figura abaixo apresenta o resultado do comando *textFile*(...) que criou um RDD[String]. Além disso, vamos contar as linhas do RDD que foi criado (*data*), por meio do comando *count*() e também vamos exibir o conteúdo da primeira linha do arquivo por meio do comando *first*().

```
scala> val data = sc.textFile("/usr/local/exemplosml/Atividade2/Dados/lpsa.dat")

data: org.apache.spark.rdd.RDD[String] = /usr/local/exemplosml/Atividade2/Dados/
lpsa.dat MapPartitionsRDD[7] at textFile at <console>:28

scala> data.count()
res3: Long = 67

scala> data.first()
res4: String = -0.4307829,-1.63735562648104 -2.00621178480549 -1.86242597251066
-1.02470580167082 -0.522940888712441 -0.863171185425945 -1.04215728919298 -0.864
466507337306

scala>
```

Iremos utilizar agora o código abaixo para percorrer a *string* dos dados, quebrando os registros pelos espaços em branco. Em seguida vamos converter os valores *string* em *double*, pois o algoritmo requer um vetor de *doubles*.

Nós utilizamos a palavra *cache* ao final, pois o Spark é um sistema de processamento distribuído. Para realizar os cálculos, nós precisamos recuperar dados de cada um dos nós de um *cluster*, e o comando *cache* reúne os elementos através dos nós. Para esse experimento nós não vamos utilizar o Spark em formato *cluster*.

```
val parsedData = data.map { line =>
val x : Array[String] = line.replace(",", " ").split(" ")
val y = x.map{ (a => a.toDouble)}
val d = y.size - 1
val c = Vectors.dense(y(0),y(d))
LabeledPoint(y(0), c)}.cache()
```

A figura abaixo apresenta o resultado da execução do comando. Foi criado um novo RDD com os dados normalizados, chamado *parsedData*. Após a normalização dos dados, executamos o comando *count*() para contar o número de elementos do novo RDD.

```
scala> val parsedData = data.map { line =>
     | val x : Array[String] = line.replace(",", " ").split(" ")
     | val y = x.map{ (a => a.toDouble)}
     | val d = y.size - 1
     | val c = Vectors.dense(y(0),y(d))
     | LabeledPoint(y(0), c)
     | }.cache()
parsedData: org.apache.spark.rdd.RDD[org.apache.spark.mllib.regression.LabeledPo
int] = MapPartitionsRDD[9] at map at <console>:30

scala> parsedData.count
res11: Long = 67
```

Nesse exemplo utilizaremos o mesmo conjunto de dados para treinamento e para testes. Isso é aceitável para fins de exemplo. Na vida real, você treinará com os dados de treinamento (como o arquivo lpsa.dat) e depois realizará previsões com os novos dados que serão recebidos, que são chamados de dados de testes.

Para a execução do algoritmo de Regressão Linear no Apache Spark/MlLib, precisaremos de dois parâmetros que são: o número de iterações (numIterations) e o stepSize que é um fator de ajuste. Esses parâmetros são importantes, pois determinam a condição de parada do algoritmo. Basicamente, eles determinam quantas vezes o algoritmo pode manter o loop para ajustar sua estimativa, ou seja, aprimorar o ponto em que o erro chega o mais próximo de zero possível. Isso é chamado de convergência.

A figura abaixo apresenta a atribuição de valores às respectivas variáveis que representam o número de iterações (*numIterations*) e o fator de ajuste (*stepSize*).

```
scala> val numIterations = 100
numIterations: Int = 100

scala> val stepSize = 0.00000001
stepSize: Double = 1.0E-8
```

Em seguida vamos **treinar** o modelo de Regressão Linear, por meio do método *Train*(...), apresentado abaixo:

```
val model = LinearRegressionWithSGD.train(parsedData,
numIterations, stepSize)
```

Criamos um RDD chamado "*model*" que irá receber o resultado do método *TRAIN*. Esse método recebe como parâmetros os dados normalizados (*parsedData*), o número máximo de iterações do algoritmo (*numIterations*) e o fator de ajuste (*stepSize*). Após a execução do treinamento você verá o seguinte resultado:

```
scala> val model = LinearRegressionWithSGD.train(parsedData, numIterations, step
Size)
warning: there was one deprecation warning; re-run with -deprecation for details
18/04/01 13:58:49 WARN Executor: 1 block locks were not released by TID = 7:
[rdd_2_0]
model: org.apache.spark.mllib.regression.LinearRegressionModel = org.apache.spar
k.mllib.regression.LinearRegressionModel: intercept = 0.0, numFeatures = 2
```

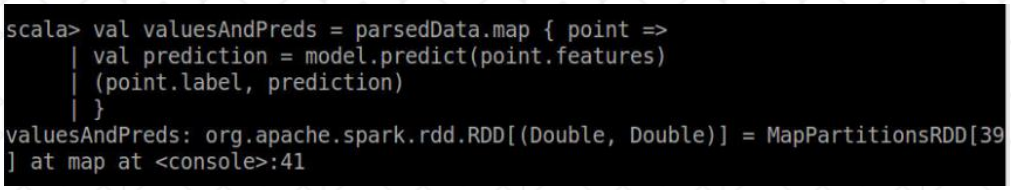
Na figura acima foi informado que existe um método que está em desuso, mas isso não é problema nesse momento. Para mais informações sobre isso, consulte:

https://issues.apache.org/jira/browse/SPARK-14829

Agora poderemos finalmente testar o nosso modelo. Para isso vamos usar o método *predict*() sobre cada um dos dados, criando um novo RDD (*double, double*) de nome *prediction,* com os rótulos (*labels*) y como variável independente significando o resultado final observado e a previsão que é a estimativa baseada na fórmula de regressão determinada no algoritmo. O comando para executar o *predict* é o seguinte:

```
val valuesAndPreds = parsedData.map { point =>
val prediction = model.predict(point.features)
(point.label, prediction)}
```

Após a execução da predição a seguinte tela será exibida:



Em seguida, vamos exibir os valores das previsões que foram feitas. Para isso, execute o comando abaixo que irá apresentar o valor da previsão e o valor atual do rótulo (*label*). Observe que foi dado um *loop* no RDD *valuesAndPreds* exibindo com *println* o valor da predição (*result._1*) e o valor atual (*result._2*). O resultado pode ser conferido na tela a seguir:

```
valuesAndPreds.foreach((result) => println(s"predicted label:
${result._1}, actual label: ${result._2}"))
```

```
scala> valuesAndPreds.foreach((result) => println(s"predicted label: ${result._1
}, actual label: ${result._2}"))
predicted label: -0.4307829, actual label: -6.538896995115336E-8
predicted label: -0.1625189, actual label: -3.126657138282193E-8
predicted label: -0.1625189, actual label: -2.257582977070669E-8
predicted label: -0.1625189, actual label: -3.126657138282193E-8
predicted label: 0.3715636, actual label: 3.6667165284680464E-8
predicted label: 0.7654678, actual label: 8.677063093843847E-8
predicted label: 0.8544153, actual label: 9.808449357620074E-8
predicted label: 1.2669476, actual label: 1.5055739923223532E-7
predicted label: 1.2669476, actual label: 1.5055739923223532E-7
predicted label: 1.2669476, actual label: 1.6359351165040817E-7
predicted label: 1.3480731, actual label: 1.6304901168121577E-7
predicted label: 1.446919, actual label: 1.7344923653317994E-7
predicted label: 1.4701758, actual label: 1.894435460967915E-7
predicted label: 1.4929041, actual label: 1.7929840716511183E-7
predicted label: 1.5581446, actual label: 1.875968085301747E-7
predicted label: 1.5993876, actual label: 1.9284279786960784E-7
predicted label: 1.6389967, actual label: 1.9788095989333408E-7
predicted label: 1.6956156, actual label: 2.0508271887814195E-7
predicted label: 1.7137979, actual label: 2.3346767925667365E-7
predicted label: 1.8000583, actual label: 2.4878512141581507E-7
predicted label: 1.8484548, actual label: 2.592863859023973E-7
```

Por último devemos realizar o cálculo do erro total que é a média da diferença entre o valor real e o valor de predição elevado ao quadrado. Se o erro é zero, isso significa que o nosso modelo é perfeito. Como você pode ver abaixo, o trabalho realizado pelo modelo não foi muito bom, pois a diferença está alta.

Para realizar o cálculo e exibir o erro total, utilize o seguinte código:

```
val MSE = valuesAndPreds.map{ case(v, p) => math.pow((v - p), 2) }.mean()
println("training Mean Squared Error = " + MSE)
```

```
scala> val MSE = valuesAndPreds.map{ case(v, p) => math.pow((v - p), 2) }.mean()

MSE: Double = 7.451030996001438

scala> println("training Mean Squared Error = " + MSE)
training Mean Squared Error = 7.451030996001438
```

Referências:

Rowe. Walker; SGD Linear Regression Example with Apache Spark. Disponível em: https://www.bmc.com/blogs/sgd-linear-regression-example-apache-spark/ Acesso em: 01/03/2018.