

Atividade em Laboratório Virtual 01

Disciplina	FAM – Fundamentos de Aprendizagem de Máquina
-------------------	---

Objetivos

A primeira atividade prática em laboratório virtual possui como objetivos principais:

- ✓ Apresentar as ferramentas utilizadas nesse curso: Spark, MLlib e a linguagem Scala;
- ✓ Propiciar ao aluno a oportunidade de conhecer o laboratório virtual;

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

O que é o Spark?

De forma geral, o Spark é um *Framework* para processamento de grandes volumes de dados em um cluster de computadores. Inicialmente, o Spark oferece um framework unificado e de fácil compreensão para gerenciar e processar Big Data com uma variedade de conjuntos de dados de diversas naturezas (por exemplo: texto, grafos, etc.), bem como de diferentes origens (*batch* ou *streaming* de dados em tempo real).

O Spark pode acessar dados de diferentes fontes, como por exemplo do HDFS (Hadoop Distributed File System). Outra possibilidade é executar o Spark local sem processamento distribuído. O modo mais comum, entretanto, é utilizar o Spark em um *cluster* de computadores, podendo ser controlado por diferentes gerenciadores.

MLlib

A biblioteca MLlib é uma funcionalidade para Machine Learning no Spark e possui diversos tipos de algoritmos, incluindo classificação, regressão, agrupamento (*clustering*) e filtragem colaborativa, assim como permite avaliação de modelos e importação de dados. Tudo isso foi criado para trabalhar muito bem através de cluster de servidores.

Os principais algoritmos suportados em MLlib são:

Classificação: Regressão Logística, Naive Bayes, etc.

Regressão: Regressão Linear

Árvores: Árvores de Decisão, Random Forests e Gradient-Boosted Trees

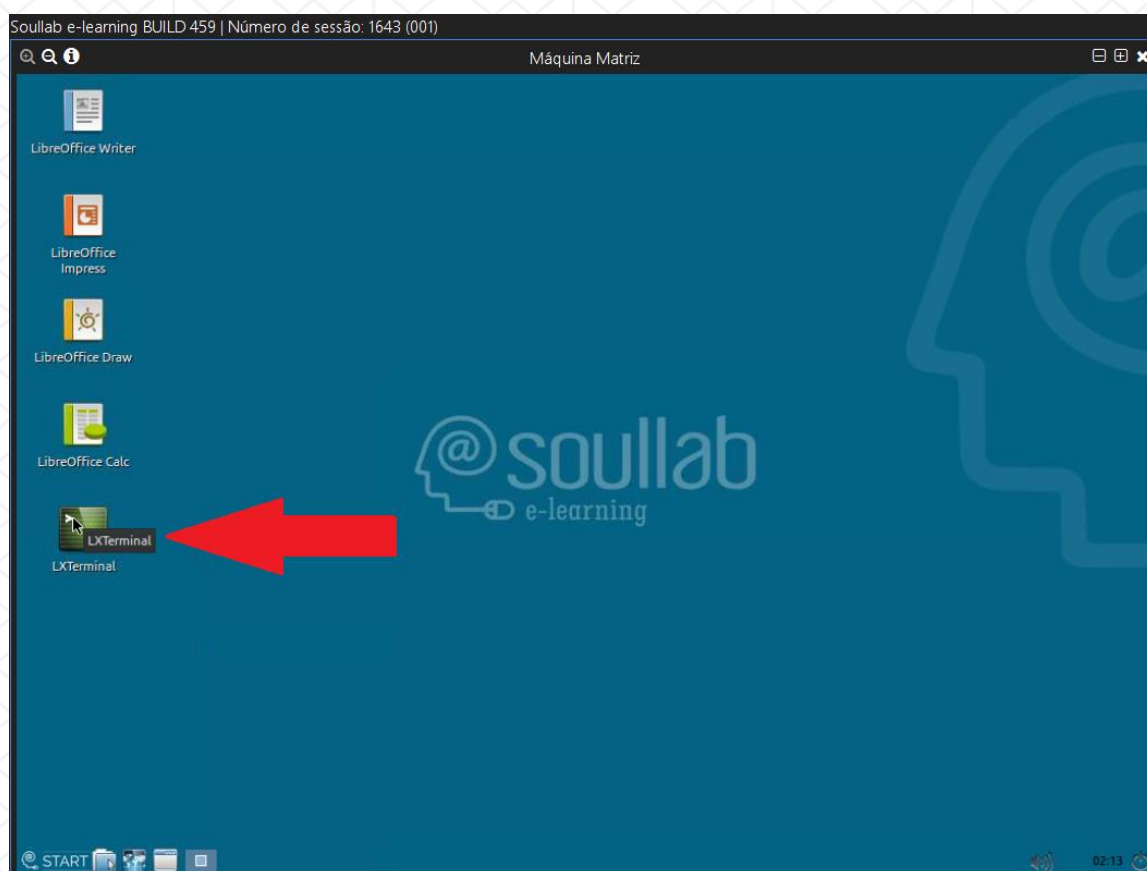
Recomendação: Alternating Least Squares (Als) – Filtragem Colaborativa

Clustering: K-means, Gaussian mixtures (GMMs), etc.

Enunciado do trabalho

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`. Sendo assim, devemos ir até o diretório de instalação do Spark utilizando o seguinte comando:

```
cd /usr/local/spark
```

A **Spark Shell** é a aplicação do Apache Spark que possibilita a exploração interativa dos dados. Para acessar o Spark Shell, utilize o seguinte comando:

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

Os exemplos que seguem abaixo são básicos e visam apenas o contato inicial com a linguagem Scala. Devem ser executados no próprio shell do Spark.

Exemplo 1: no exemplo abaixo é criada uma *tupla* com um inteiro e uma *string*, note que não é preciso declarar o tipo da variável. Usando **getClass** é possível verificar que Scala sabe exatamente os tipos de cada variável, sendo a *tupla* uma variável com um conjunto de tipos.

```
scala> val tup1 = (0, "zero")
scala> tup1.getClass
res1: Class[_ <: (Int, String)] = class scala.Tuple2

scala> val tup2 = (1, "um", 1.0)
scala> tup2.getClass
res2: Class[_ <: (Int, String, Double)] = class scala.Tuple3

scala> val tup2 = (1, "um", 1.0)
scala> tup2.getClass
res2: Class[_ <: (Int, String, Double)] = class scala.Tuple3
```

É possível acessar os valores de cada *tupla* utilizando a sintaxe sublinhado ("_").


```
scala> println(tup1._1)
0

scala> println(tup1._2)
zero

scala> println(tup2._2)
um

scala> println(tup2._3)
1.0
```

Exemplo 2: o exemplo abaixo cria a função **sum** que realiza a soma de dois inteiros. Para definir uma função em Scala é preciso dizer o tipo dos parâmetros e o tipo do retorno. Em Scala é possível definir funções dentro de funções. Além disso, Scala suporta nomes de funções com os caracteres +, ++, ~, &-, --, \, /, : etc.

```
scala> def sum (x: Int, y: Int) : Int = { return x + y }
sum: (x: Int, y: Int)Int

scala> sum (1,2)
res0: Int = 3
```

Exemplo 3: Abstração de dados RDD

No Apache Spark realizamos o processamento dos dados por meio de operações em coleções distribuídas que são automaticamente paralelizadas no *cluster*. Essas coleções são chamadas de RDD (*Resilient Distributed Datasets*).

Toda aplicação Spark consiste de um **driver** que executa uma função **main** escrita pelo usuário e executa também várias operações em paralelo no cluster. Cada uma destas operações é executada sobre uma coleção de dados, representada internamente no Spark por uma abstração de dados.

Existem duas maneiras de criar um RDD: (1) paralelizando uma coleção já existente dentro do programa, ou (2) referenciando dados em um sistema externo ao programa (HDFS, HBase, ou qualquer fonte de dados).

Vamos carregar um novo RDD chamado “linhas”, com os dados do arquivo README (em maiúsculo mesmo!) que encontra-se no sistema de arquivos do Linux. Em seguida vamos verificar a quantidade de linhas que temos no RDD “linhas”. Por fim vamos exibir o primeiro elemento do RDD linhas por meio da ação *first()*.



```
version 2.1.0

Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_121)
Type in expressions to have them evaluated.
Type :help for more information.

scala> val linhas = sc.textFile("/usr/local/spark/README.md")
linhas: org.apache.spark.rdd.RDD[String] = /usr/local/spark/README.md MapPartitionsRDD[1] at textFile at <console>:24

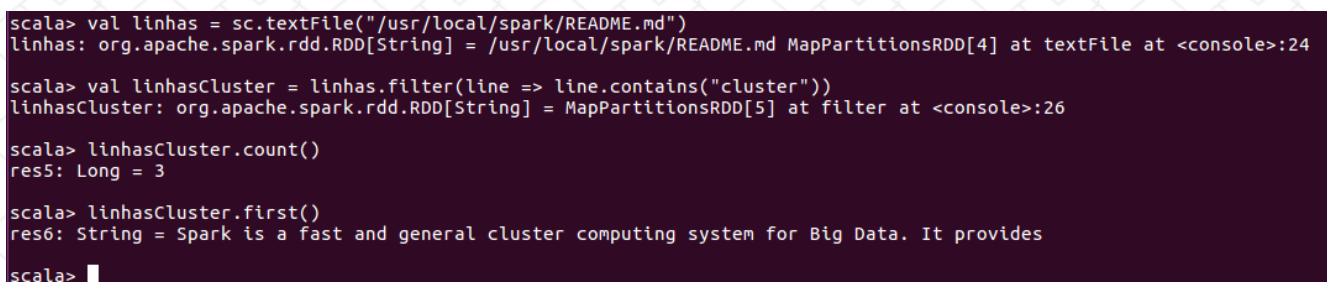
scala> linhas.count()
res0: Long = 104

scala> linhas.first()
res1: String = # Apache Spark

scala>
```

Executando transformações nos RDDs

Ainda no RDD “linhas” que foi criado anteriormente, vamos aplicar uma transformação do tipo *filter* e, com o resultado, vamos criar um novo RDD chamado “*linhasCluster*”. Esse RDD vai conter apenas os elementos do RDD “linhas” que possuem a palavra “cluster”. Em seguida vamos contar a quantidade de elementos (Ação *count()*) e exibir o primeiro elemento (Ação *first()*) do RDD “linhasCluster”. O resultado é apresentado na figura abaixo:



```
scala> val linhas = sc.textFile("/usr/local/spark/README.md")
linhas: org.apache.spark.rdd.RDD[String] = /usr/local/spark/README.md MapPartitionsRDD[4] at textFile at <console>:24

scala> val linhasCluster = linhas.filter(line => line.contains("cluster"))
linhasCluster: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[5] at filter at <console>:26

scala> linhasCluster.count()
res5: Long = 3

scala> linhasCluster.first()
res6: String = Spark is a fast and general cluster computing system for Big Data. It provides

scala>
```

Já criamos um RDD baseado em um arquivo do sistema de arquivos do sistema operacional e agora vamos carregar um novo RDD com dados de uma lista de valores inteiros. Em seguida vamos aplicar uma função *Map* a cada um dos elementos do RDD criado e essa função vai multiplicar cada elemento por ele mesmo, conforme a figura abaixo. Vamos usar a ação *take(9)* para buscar os 9 elementos do RDD e o *foreach* para “navegar” por todos eles.

```
scala> val entrada = sc.parallelize(List(1, 2, 4, 5, 6, 7, 8, 3, 0))
entrada: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[5] at parallelize at <console>:24

scala> val resultado = entrada.map(x => x * x)
resultado: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[6] at map at <console>:26

scala> resultado.count()
res10: Long = 9

scala> resultado.take(9).foreach(println)
1
4
16
25
36
49
64
9
0

scala>
```

Vamos carregar um novo RDD chamado “numeros”. Nesse novo RDD iremos criar uma lista de números inteiros com os seguintes valores 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 e 12. Em seguida iremos aplicar uma Transformação Map multiplicando cada elemento por -1. Vamos também exibir todos os valores do RDD “numeros” com a ação collect() e separando os valores com vírgulas usando mkString(“,”). A figura abaixo destaca o resultado da execução.

```
scala> val numeros = sc.parallelize(List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12))
numeros: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[3] at parallelize at <console>:24

scala> val resultado = numeros.map(x => x*(-1))
resultado: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[4] at map at <console>:26

scala> println(resultado.collect().mkString(","))
-1,-2,-3,-4,-5,-6,-7,-8,-9,-10,-11,-12

scala>
```

Agora vamos usar uma transformação Filter e selecionar apenas os elementos do RDD que são maiores que 10.

```
scala> val numeros = sc.parallelize(List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12))
numeros: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[5] at parallelize at <console>:24

scala> val resultado = numeros.filter(x => x>10)
resultado: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[6] at filter at <console>:26

scala> println(resultado.collect().mkString(","))
11,12

scala>
```


Nesse momento vamos recarregar o RDD “numeros” com os valores 1, 2, 3, 4 e 5. Em seguida vamos aplicar uma transformação Reduce, atribuindo ao RDD “aux” o resultado. Primeiramente vamos somar os elementos e depois multiplicá-los.

```
scala> val numeros = sc.parallelize(List(1, 2, 3, 4, 5))
numeros: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[35] at parallelize at <console>:24

scala> val aux = numeros.reduce((x, y) => x + y)
aux: Int = 15

scala> val aux = numeros.reduce((x, y) => x * y)
aux: Int = 120

scala>
```

Vamos carregar novamente o RDD “numeros” com novos valores e aplicar o countByValue e o top.

```
scala> val numeros = sc.parallelize(List(3, 2, 3, 4, 5, 7, 8, 6, 4, 3, 2, 1, 0, 2, 1, 2, 7, 8, 0, 0, 2, 1))
numeros: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[45] at parallelize at <console>:24

scala> numeros.countByValue()
res36: scala.collection.Map[Int,Long] = Map(0 -> 3, 5 -> 1, 1 -> 3, 6 -> 1, 2 -> 5, 7 -> 2, 3 -> 3, 8 -> 2, 4 -> 2)

scala> numeros.top(1)
res37: Array[Int] = Array(8)
```

Conclusão:

Nessa atividade prática executamos o spark-shell e manipulamos RDDs por meio de criação, transformações e ações. O aluno deverá se sentir motivado a executar os outros exemplos que constam na apostila e slides das aulas gravadas.