实验三 Spark机器学习

一、实验目标

本次实验旨在教会学生使用Spark处理数据并实现机器学习算法。具体如下：

- 学习使用Spark-shell基本命令
- 使用Spark实现词频统计、计算数据方差
- 使用Spark实现线性回归训练算法

二、学习Spark-shell常用指令

- 开启spark-shell：

```
2021214308@thumm01:~$ spark-shell
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
21/11/16 17:27:31 WARN Utils: Service 'SparkUI' could not bind on port 4040. Attempting port 4041.
21/11/16 17:27:31 WARN Utils: Service 'SparkUI' could not bind on port 4041. Attempting port 4042.
21/11/16 17:27:31 WARN Utils: Service 'SparkUI' could not bind on port 4042. Attempting port 4043.
Spark Context Web UI is available at Spark Master Public URL
Spark context available as 'sc' (master = spark://thumm01:7077, app id = app-20211116172731-0030).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.4.4
      /_/

Using Scala version 2.12.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_221)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

- 输入`:help`查看指令：

```
scala> :help
All commands can be abbreviated, e.g., :he instead of :help.
:completions <string>    output completions for the given string
:edit <id>|<line>        edit history
:help [command]          print this summary or command-specific help
:history [num]           show the history (optional num is commands to show)
:h? <string>             search the history
:imports [name name ...] show import history, identifying sources of names
:implicits [-v]          show the implicits in scope
:javap <path|class>      disassemble a file or class name
:line <id>|<line>        place line(s) at the end of history
:load <path>             interpret lines in a file
:paste [-raw] [path]     enter paste mode or paste a file
:power                   enable power user mode
:quit                    exit the interpreter
:replay [options]        reset the repl and replay all previous commands
:require <path>          add a jar to the classpath
:reset [options]         reset the repl to its initial state, forgetting all session entries
:save <path>             save replayable session to a file
:sh <command line>       run a shell command (result is implicitly => List[String])
:settings <options>      update compiler options, if possible; see reset
:silent                  disable/enable automatic printing of results
:type [-v] <expr>        display the type of an expression without evaluating it
:kind [-v] <type>        display the kind of a type. see also :help kind
:warnings                show the suppressed warnings from the most recent line which had any
```

- 在ubuntu自己的目录下新建一个Scala文件，写入输出`Hello world`的语句，进入`spark-shell`使用`:load`运行该文件：

  创建`Helloworld.scala`文件：

```
println("Hello world!")
```

进入`spark-shell`使用`:load`运行该文件:

```
scala> :load ./HelloWorld.scala
Loading ./HelloWorld.scala...
Hello world!
```

三、使用Spark进行词频统计

我们使用单词表数据集，其包含**536700000行**单词，大小为**2.6G**，存储路径为**/home/dsjxtjc/2021214308/word_count/wc_data.txt**

首先，将数据集传入Hadoop文件系统内:

```
hadoop fs -copyFromLocal word_count/wc_data.txt /dsjxtjc/2021214308/wc_data.txt
```

而后，进入spark-shell，并加载待统计词频的数据集:

```
val words = sc.textFile("/dsjxtjc/2021214308/wc_data.txt")
```

我们分别输入`words.first()`和`words.count()`查看words的内容:

```
scala> words.first()
res0: String = chapter

scala> words.count()
res1: Long = 536700000
```

最后，我们使用一行代码统计词频:

```
scala> val result = words.flatMap(l => l.split(" ")).map(w => (w, 1)).reduceByKey(_ + _)
result: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[4] at reduceByKey at <console>:25

scala> result.first()
res2: (String, Int) = (growls,20000)

scala> result.saveAsTextFile("/dsjxtjc/2021214308/wc_output")
```

hadoop查看生成结果:

```
2021214308@thumm01:/mnt/data/dsjxtjc/2021214308/word_count$ hadoop fs -ls /dsjxtjc/2021214308/
Found 4 items
-rw-r--r--   3 2021214308 dsjxtjc    6888896 2021-11-16 22:45 /dsjxtjc/2021214308/numbers.txt
-rw-r--r--   3 2021214308 dsjxtjc         13 2021-10-19 19:22 /dsjxtjc/2021214308/test.txt
-rw-r--r--   3 2021214308 dsjxtjc 2716098000 2021-11-17 08:44 /dsjxtjc/2021214308/wc_data.txt
drwxr-xr-x   - 2021214308 dsjxtjc          0 2021-11-17 08:52 /dsjxtjc/2021214308/wc_output
```

四、使用Spark计算均值与方差

首先，创建一个从1到1000000的数据集：

```
for ((i=1; i<=1000000; i=i+1)); do echo $i >> numbers.txt; done
tail -n 3 numbers.txt
```

```
2021214308@thumm01:~$ tail -n 3 numbers.txt
999998
999999
1000000
```

而后将数据集上次到HDFS：

```
2021214308@thumm01:~$ hadoop fs -put numbers.txt /dsjxtjc/2021214308/
2021-11-16 22:45:33,986 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2021214308@thumm01:~$ hadoop fs -tail /dsjxtjc/2021214308/numbers.txt
2021-11-16 22:45:59,581 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false

999855
999856
999857
999858
999859
999860
999861
999862
999863
999864
999865
999866
999867
999868
999869
999870
999871
999872
999873
999874
999875
999876
```

运行spark-shell并加载`numbers.txt`：

```
scala> val numbers = sc.textFile("/dsjxtjc/2021214308/numbers.txt")
numbers: org.apache.spark.rdd.RDD[String] = /dsjxtjc/2021214308/numbers.txt MapPartitionsRDD[1] at textFile at <console>:24
```

将加载的字符串形式的数字转为`double`类型的数值：

```
scala> val numbers_double = numbers.map(num => num.toDouble)
numbers_double: org.apache.spark.rdd.RDD[Double] = MapPartitionsRDD[3] at map at <console>:25
```

统计数字个数：

```
scala> val n_num = numbers_double.count()
n_num: Long = 1000000
```

计算均值：

```
scala> val mean = numbers_double.reduce(_ + _) / n_num
mean: Double = 500000.5
```

计算方差：

```
scala> val variance = numbers_double.map(num => num - mean).map(num => num * num).reduce(_ + _) / n_num
variance: Double = 8.333333333359143E10
```

计算标准差：

```
scala> import scala.math._
import scala.math._

scala> val std = sqrt(variance)
std: Double = 288675.1345952599
```

五、Spark机器学习（n元线性回归矩阵形式）

5.1 数据准备

在本小节中，首先我们准备了一个可用于n元线性回归的数据集。数据集中每条数据的格式遵照 $(y, \quad x_1 \quad x_2 \cdots x_{n-1} \quad x_n)$的格式：

```
-0.4307829,-1.63735562648104 -2.00621178480549 -1.86242597251066 -1.02470580167082 -0.522940888712441 -0.863171185425945 -1.04215728919298 -0.864466507337306
-0.1625189,-1.98898046126935 -0.722008756122123 -0.787896192088153 -1.02470580167082 -0.522940888712441 -0.863171185425945 -1.04215728919298 -0.864466507337306
-0.1625189,-1.57881887548545 -2.1887840293994 1.36116336875686 -1.02470580167082 -0.522940888712441 -0.863171185425945 0.342627053981254 -0.155348103855541
-0.1625189,-2.16691708463163 -0.807993896938655 -0.787896192088153 -1.02470580167082 -0.522940888712441 -0.863171185425945 -1.04215728919298 -0.864466507337306
0.3715636,-0.507874475300631 -0.458834049396776 -0.250631301876899 -1.02470580167082 -0.522940888712441 -0.863171185425945 -1.04215728919298 -0.864466507337306
0.7654678,-2.03612849966376 -0.933954647105133 -1.86242597251066 -1.02470580167082 -0.522940888712441 -0.863171185425945 -1.04215728919298 -0.864466507337306
0.8544153,-0.557312518810673 -0.208756571683607 -0.787896192088153 0.990146852537193 -0.522940888712441 -0.863171185425945 -1.04215728919298 -0.864466507337306
1.2669476,-0.929360463147704 -0.0578991819441687 0.152317365781542 -1.02470580167082 -0.522940888712441 -0.863171185425945 -1.04215728919298 -0.864466507337306
1.2669476,-2.28833047634983 -0.0706369432557794 -0.116315079324086 0.80409888772376 -0.522940888712441 -0.863171185425945 -1.04215728919298 -0.864466507337306
1.2669476,0.223498042876113 -1.41471935455355 -0.116315079324086 -1.02470580167082 -0.522940888712441 -0.29928234305568 0.342627053981254 0.199211097885341
1.3480731,0.107785900236813 -1.47221551299731 0.420949810887169 -1.02470580167082 -0.522940888712441 -0.863171185425945 0.342627053981254 -0.687186906466865
1.446919,0.162180092313795 -1.32557369901905 0.286633588334355 -1.02470580167082 -0.522940888712441 -0.863171185425945 -1.04215728919298 -0.864466507337306
1.4701758,-1.49795329918548 -0.263601072284232 0.823898478545609 0.788388310173035 -0.522940888712441 -0.29928234305568 0.342627053981254 0.199211097885341
1.4929041,0.796247055396743 0.0476559407005752 0.286633588334355 -1.02470580167082 -0.522940888712441 0.394013435896129 -1.04215728919298 -0.864466507337306
1.5581446,-1.62233848461465 -0.843294091975396 -3.07127197548598 -1.02470580167082 -0.522940888712441 -0.863171185425945 -1.04215728919298 -0.864466507337306
1.5993876,-0.990720665490831 0.458513517212311 0.823898478545609 1.07379746308195 -0.522940888712441 -0.863171185425945 -1.04215728919298 -0.864466507337306
```

我们的数据保存在**/home/dsjxtjc/2021214308/linear_regression/linear_data.txt**路径下。并将其上传至hadoop：

```
2021214308@thumm01:/mnt/data/dsjxtjc/2021214308/linear_regression$ hadoop fs -ls /dsjxtjc/2021214308/
Found 5 items
-rw-r--r--   3 2021214308 dsjxtjc 2251853460 2021-11-17 10:13 /dsjxtjc/2021214308/linear_data.txt
-rw-r--r--   3 2021214308 dsjxtjc    6888896 2021-11-16 22:45 /dsjxtjc/2021214308/numbers.txt
-rw-r--r--   3 2021214308 dsjxtjc         13 2021-10-19 19:22 /dsjxtjc/2021214308/test.txt
-rw-r--r--   3 2021214308 dsjxtjc 2716098000 2021-11-17 08:44 /dsjxtjc/2021214308/wc_data.txt
drwxr-xr-x   - 2021214308 dsjxtjc          0 2021-11-17 08:52 /dsjxtjc/2021214308/wc_output
```

5.2 n元线性回归算法

针对一元线性回归的情况，设数据集为$\{(x_i, y_i)\}_{i=1}^{n}$，且一元线性回归函数为$f(x) = wx + b$，我们需要得到一组参数(w^*, b^*)，使作为目标函数的均方根$E_{(w,b)}$误差取到最小值：

$$
\begin{aligned}
E_{(w,b)} &= \sum_{i=1}^{m} (f(x_i) - y_i)^2 \\
&= \sum_{i=1}^{m} (y_i - wx_i - b)^2 \\
(w^*, b^*) &= \arg\min_{(w,b)} \sum_{i=1}^{m} (f(x_i) - y_i)^2 \\
&= \arg\min_{(w,b)} \sum_{i=1}^{m} (y_i - wx_i - b)^2
\end{aligned}
$$

$E_{(w,b)}$对w, b求偏导可得:

$$\frac{\partial E_{(w,b)}}{\partial w} = 2\left(w\sum_{i=1}^{m} x_i^2 - \sum_{i=1}^{m}(y_i - b)x_i\right)$$
$$\frac{\partial E_{(w,b)}}{\partial b} = 2\left(mb - \sum_{i=1}^{m}(y_i - wx_i)\right)$$

由于$E_{(w,b)}$对w, b均为凸函数，故令偏导等于零，我们可以得到我们所需的(w^*, b^*):

$$w^* = \frac{\sum_{i=1}^{m} y_i(x_i - \bar{x})}{\sum_{i=1}^{m} x_i^2 - \frac{1}{m}\left(\sum_{i=1}^{m} x_i\right)^2}$$
$$b^* = \frac{1}{m}\sum_{i=1}^{m}(y_i - wx_i)$$

我们可以将一元线性回归拓展为多元线性回归。设样本数为m，特征数为n:

$$\mathbf{W} = (b, w_1, w_2, \ldots, w_n)^T$$
$$\mathbf{X} = (1, x_1, x_2, \ldots, x_n)^T$$
$$\mathbf{y} = (y_1, y_2, \ldots, y_m)^T$$

类似地，我们可以使用梯度下降法来优化多元线性回归的损失函数，其中优化目标函数为:

$$\mathbf{W} = \arg\min_{\mathbf{W}}(\mathbf{XW} - \mathbf{y})^T(\mathbf{XW} - \mathbf{y})$$

梯度为:

$$\frac{\partial E_{\mathbf{w}}}{\partial \mathbf{W}} = 2\mathbf{X}^T(\mathbf{XW} - \mathbf{y})$$

5.3 代码实现

根据上述的算法，我们可以使用scala代码对其进行实现。

首先，我们设计一个矩阵类，作为我们进行矩阵运算的接口。这个矩阵类中包含矩阵乘法（内积）、矩阵加减法、以及矩阵与常数的乘法:

```
// 用于执行矩阵运算的矩阵类
class Matrix(private val data:Array[Double], val rownum:Int){
    val colnum = (data.length.toDouble/rownum).ceil.toInt
    val matrix:Array[Array[Double]]={
        val matrix:Array[Array[Double]] = Array.ofDim[Double](rownum,colnum)
        for(i <- 0 until rownum){
            for(j <- 0 until colnum){
                val index = i * colnum + j
                matrix(i)(j) = if(data.isDefinedAt(index)) data(index) else 0
            }
        }
        matrix
    }

    override def toString = {
        var str = ""
```

```
        matrix.map((p:Array[Double]) => {p.mkString(" ")}).mkString("\n")
    }

    def mat(row:Int,col:Int) = {
        matrix(row - 1)(col - 1)
    }

    // 矩阵与矩阵乘法
    def *(a : Matrix) : Matrix = {
        if(this.colnum != a.rownum){
            println("Wrong!")
            var ans = new Matrix(this.data,this.rownum)
            ans.asInstanceOf[Matrix]
        }else{
            val data:ArrayBuffer[Double] = ArrayBuffer()
            for(i <- 0 until this.rownum){
                for(j <- 0 until a.colnum){
                    var num = 0.0
                    for(k <- 0 until this.colnum){
                        num += this.matrix(i)(k) * a.matrix(k)(j)
                    }
                data += num
                }
            }
            var ans = new Matrix(data.toArray,this.rownum)
            ans.asInstanceOf[Matrix]
        }
    }

    // 矩阵乘常数
    def *(a:Double) : Matrix = {
        val data:ArrayBuffer[Double] = ArrayBuffer()
        for(i <- 0 until this.rownum){
            for(j <- 0 until this.colnum){
                data += this.matrix(i)(j) * a
            }
        }
        var ans = new Matrix(data.toArray,this.rownum)
        ans.asInstanceOf[Matrix]
    }

    // 矩阵间加法
    def +(a : Matrix) : Matrix = {
        if(this.rownum != a.rownum || this.colnum != a.colnum){
            println("Wrong!")
            var ans = new Matrix(this.data,this.rownum)
            ans.asInstanceOf[Matrix]
        }else{
            val data:ArrayBuffer[Double] = ArrayBuffer()
            for(i <- 0 until this.rownum){
                for(j <- 0 until this.colnum){
                    data += this.matrix(i)(j) + a.matrix(i)(j)
                }
            }
            var ans = new Matrix(data.toArray,this.rownum)
            ans.asInstanceOf[Matrix]
        }
    }
```

```
    // 矩阵间减法
    def -(a : Matrix) : Matrix = {
        if(this.rownum != a.rownum || this.colnum != a.colnum){
            println("Wrong!")
            var ans = new Matrix(this.data,this.rownum)
            ans.asInstanceOf[Matrix]
        }else{
            val data:ArrayBuffer[Double] = ArrayBuffer()
            for(i <- 0 until this.rownum){
                for(j <- 0 until this.colnum){
                    data += this.matrix(i)(j) - a.matrix(i)(j)
                }
            }
            var ans = new Matrix(data.toArray,this.rownum)
            ans.asInstanceOf[Matrix]
        }
    }


    // 矩阵转置
    def transpose() : Matrix = {
        val transposeMatrix = for (i <- Array.range(0,colnum)) yield {
             for (rowArray <- this.matrix) yield rowArray(i)
            }
        var ans = new Matrix(transposeMatrix.flatten,colnum)
        ans.asInstanceOf[Matrix]
    }

}
```

而后，在已经有了可调用的矩阵接口的条件下，我们可以对多元线性回归进行代码实现。在代码中，我们首先读取数据集中的数据，并对其进行格式上的预处理后将其构造为矩阵；而后，我们初始化权重矩阵，将其全部初始化为1；最后，我们按照上述公式对用向量表示的n元线性回归梯度下降法进行迭代，在每轮对权重矩阵进行更新。

```
object Linear_Regressuion{
    def main(args:Int) : Unit = {
        var alpha = 0.001   // 学习率alpha
        var x = ArrayBuffer[Double]()
        var y = ArrayBuffer[Double]()
        val data =
Source.fromFile("/home/dsjxtjc/2021214308/linear_regression/linear_data.txt")
        var cnt =  0    //   矩阵行数计数器
        for(line <- data.getLines)
        {
            cnt += 1
            var parts = line.split(",")
            // 逗号前一部分为y
            y.append(parts(0).toDouble)
            // 逗号后半部分为多元x，并以空格分开
            var x_part : Array[Double] = parts(1).split(" ").map(_.toDouble)
            x.append(1.0)
            for(i <- 0 until x_part.size)
                x.append(x_part(i))
        }
        data.close
```

```scala
        // 使用矩阵接口构造X、Y的矩阵
        var X_matrix = new Matrix(x.toArray, cnt)
        var Y_matrix = new Matrix(y.toArray, cnt)

        // 初始化权重矩阵(初始化为1)
        var w_array = ArrayBuffer[Double]()
        for(k <- 0 until X_matrix.colnum)
            w_array.append(1)
        var w = new Matrix(w_array.toArray, X_matrix.colnum)

        for(i <- 0 until 150){
            // 对用向量表示的n元线性回归梯度下降法进行迭代
            var J : Matrix = ( ((X_matrix*w) - Y_matrix).transpose) * ((X_matrix*w) - Y_matrix) * (1.0/(2 * X_matrix.rownum))
            println("iteration " + i + ":  " + J)
            // 更新权重矩阵
            w = w - (X_matrix.transpose) * ((X_matrix * w) - Y_matrix) * (alpha) * 2
        }

        println("w:")
        println(w)
    }
}
```

5.4 结果

在spark-shell中，我们load相关的scala文件后并执行相关的main函数，进行梯度下降法。可以看到，随着迭代次数的逐渐增加，我们的目标损失函数的值逐渐下降，并趋于稳定：

```
scala> Linear_Regressuion.main(1)
iteration 0:  10.15798321667408
iteration 1:  3.788752152716596
iteration 2:  1.804842304918377
iteration 3:  1.1029638364885836
iteration 4:  0.8003501535348875
iteration 5:  0.6383195571251372
iteration 6:  0.5360858085586169
iteration 7:  0.46517864674631765
iteration 8:  0.4135505846292794
iteration 9:  0.37497035226230224
iteration 10:  0.345668526924697 9
iteration 11:  0.3231395056833352
iteration 12:  0.30563211780662236
iteration 13:  0.29188909610138497
iteration 14:  0.2809932333750507
iteration 15:  0.27226806928505193
iteration 16:  0.2652104892874571
iteration 17:  0.259443650781925
iteration 18:  0.254683506343676
iteration 19:  0.2507146915096903
iteration 20:  0.24737298719506665
iteration 21:  0.244532463224282
iteration 22:  0.2420959917206987
iteration 23:  0.23998820829866965
```

```
iteration 134:  0.21963354700191737
iteration 135:  0.21963194129613647
iteration 136:  0.21963041305964542
iteration 137:  0.21962895848842315
iteration 138:  0.21962757397029636
iteration 139:  0.21962625607487757
iteration 140:  0.2196250015440604
iteration 141:  0.21962380728303824
iteration 142:  0.21962267035181693
iteration 143:  0.21962158795719156
iteration 144:  0.2196205574451607
iteration 145:  0.21961957629375167
iteration 146:  0.21961864210623372
iteration 147:  0.21961775260469796
iteration 148:  0.21961690562398006
iteration 149:  0.21961609910590915
```

最终，在损失函数值趋于稳定后，我们可以得到最终所需的权重矩阵$\mathbf{W}$：

```
w:
2.4659080033126006
0.6755741430070558
0.263696796514O857
-0.14159035933066322
0.2103272219495O615
0.3049545198332462
-0.2812855429405669
-0.015248501981444092
0.25871873623106456
```