MIE1628: BIG DATA SCIENCE

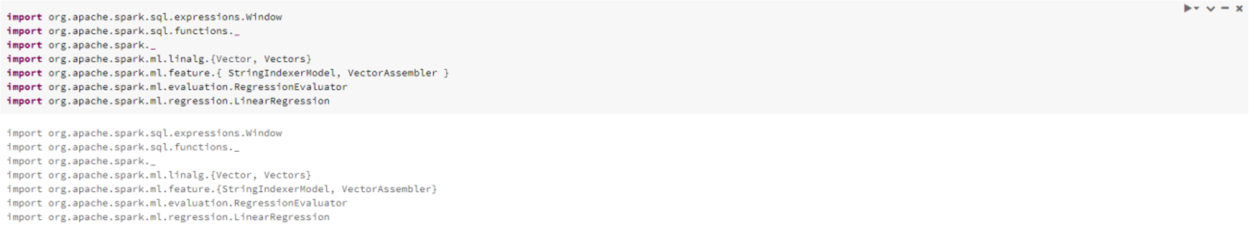
ASSIGNMENT 2: Spark

Bowen Xu 1006411786

Due on: November 11th

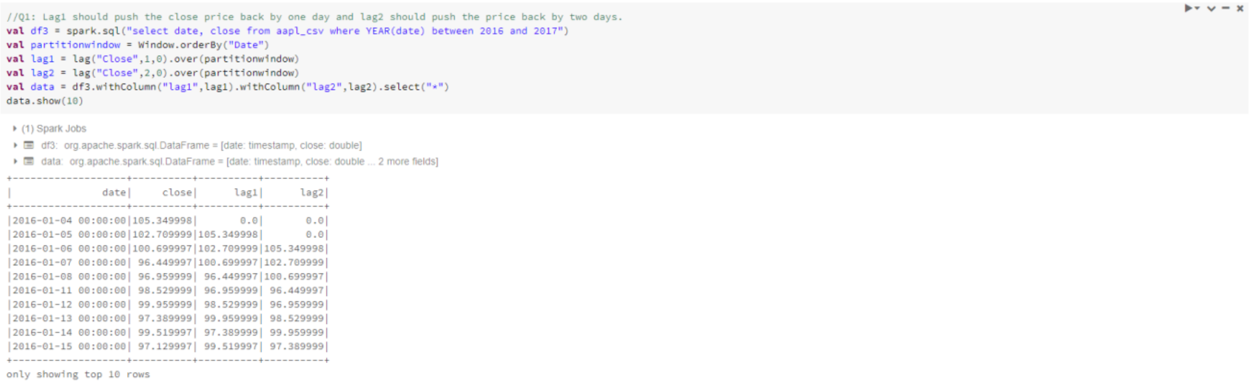
Section 1: Machine Learning and Time Series Prediction (25 points)
Given Data: APPLE daily close price from 2016-2017 (2 years inclusive)
Download it from Yahoo finance: *https://finance.yahoo.com/quote/AAPL/history?p=AAPL*

Q1: Use apple close price data and create two lag features, lag1 and lag2. Lag1 should push the close price back by one day and lag2 should push the price back by two days. Show your code and data in the dataframe.



```
import org.apache.spark.sql.expressions.Window
import org.apache.spark.sql.functions._
import org.apache.spark._
import org.apache.spark.ml.linalg.{Vector, Vectors}
import org.apache.spark.ml.feature.{ StringIndexerModel, VectorAssembler }
import org.apache.spark.ml.evaluation.RegressionEvaluator
import org.apache.spark.ml.regression.LinearRegression
```

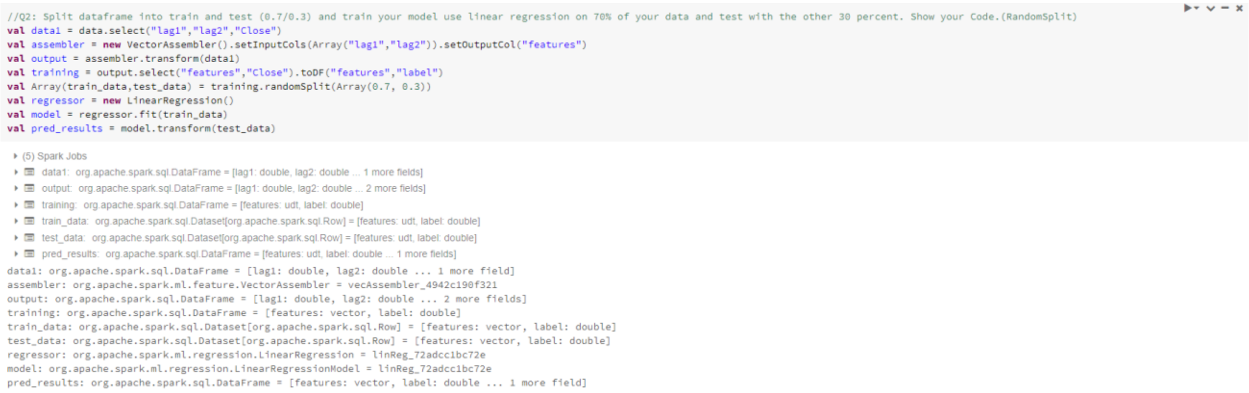
```
import org.apache.spark.sql.expressions.Window
import org.apache.spark.sql.functions._
import org.apache.spark._
import org.apache.spark.ml.linalg.{Vector, Vectors}
import org.apache.spark.ml.feature.{StringIndexerModel, VectorAssembler}
import org.apache.spark.ml.evaluation.RegressionEvaluator
import org.apache.spark.ml.regression.LinearRegression
```



```
//Q1: Lag1 should push the close price back by one day and lag2 should push the price back by two days.
val df3 = spark.sql("select date, close from aapl_csv where YEAR(date) between 2016 and 2017")
val partitionwindow = Window.orderBy("Date")
val lag1 = lag("Close",1,0).over(partitionwindow)
val lag2 = lag("Close",2,0).over(partitionwindow)
val data = df3.withColumn("lag1",lag1).withColumn("lag2",lag2).select("*")
data.show(10)
```

```
▸ (1) Spark Jobs
▸ df3:  org.apache.spark.sql.DataFrame = [date: timestamp, close: double]
▸ data:  org.apache.spark.sql.DataFrame = [date: timestamp, close: double ... 2 more fields]
+-------------------+----------+----------+----------+
|               date|     close|      lag1|      lag2|
+-------------------+----------+----------+----------+
|2016-01-04 00:00:00|105.349998|       0.0|       0.0|
|2016-01-05 00:00:00|102.709999|105.349998|       0.0|
|2016-01-06 00:00:00|100.699997|102.709999|105.349998|
|2016-01-07 00:00:00| 96.449997|100.699997|102.709999|
|2016-01-08 00:00:00| 96.959999| 96.449997|100.699997|
|2016-01-11 00:00:00| 98.529999| 96.959999| 96.449997|
|2016-01-12 00:00:00| 99.959999| 98.529999| 96.959999|
|2016-01-13 00:00:00| 97.389999| 99.959999| 98.529999|
|2016-01-14 00:00:00| 99.519997| 97.389999| 99.959999|
|2016-01-15 00:00:00| 97.129997| 99.519997| 97.389999|
+-------------------+----------+----------+----------+
only showing top 10 rows
```

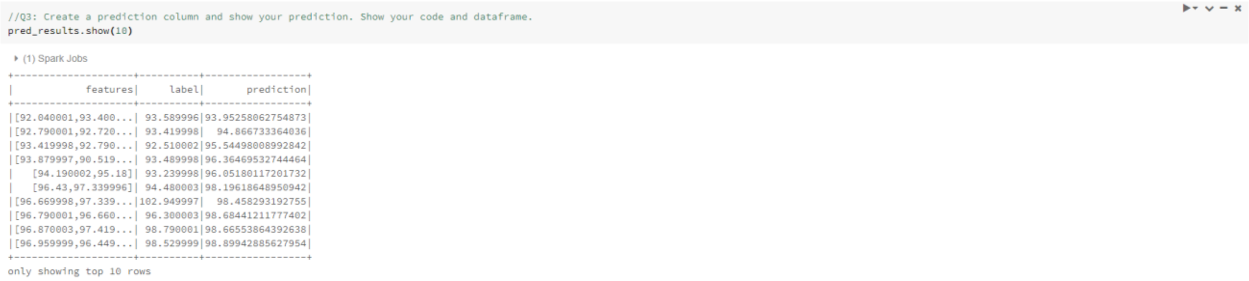
Q2: Split dataframe into train and test (0.7/0.3) and train your model use linear regression on 70% of your data and test with the other 30 percent. Show your Code.



```
//Q2: Split dataframe into train and test (0.7/0.3) and train your model use linear regression on 70% of your data and test with the other 30 percent. Show your Code.(RandomSplit)
val data1 = data.select("lag1","lag2","Close")
val assembler = new VectorAssembler().setInputCols(Array("lag1","lag2")).setOutputCol("features")
val output = assembler.transform(data1)
val training = output.select("features","Close").toDF("features","label")
val Array(train_data,test_data) = training.randomSplit(Array(0.7, 0.3))
val regressor = new LinearRegression()
val model = regressor.fit(train_data)
val pred_results = model.transform(test_data)
```

```
▸ (5) Spark Jobs
▸ data1:  org.apache.spark.sql.DataFrame = [lag1: double, lag2: double ... 1 more fields]
▸ output:  org.apache.spark.sql.DataFrame = [lag1: double, lag2: double ... 2 more fields]
▸ training:  org.apache.spark.sql.DataFrame = [features: udt, label: double]
▸ train_data:  org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [features: udt, label: double]
▸ test_data:  org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [features: udt, label: double]
▸ pred_results:  org.apache.spark.sql.DataFrame = [features: udt, label: double ... 1 more fields]
data1: org.apache.spark.sql.DataFrame = [lag1: double, lag2: double ... 1 more field]
assembler: org.apache.spark.ml.feature.VectorAssembler = vecAssembler_4942c190f321
output: org.apache.spark.sql.DataFrame = [lag1: double, lag2: double ... 2 more fields]
training: org.apache.spark.sql.DataFrame = [features: vector, label: double]
train_data: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [features: vector, label: double]
test_data: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [features: vector, label: double]
regressor: org.apache.spark.ml.regression.LinearRegression = linReg_72adcc1bc72e
model: org.apache.spark.ml.regression.LinearRegressionModel = linReg_72adcc1bc72e
pred_results: org.apache.spark.sql.DataFrame = [features: vector, label: double ... 1 more field]
```

Q3: Create a prediction column and show your prediction. Show your code and dataframe.

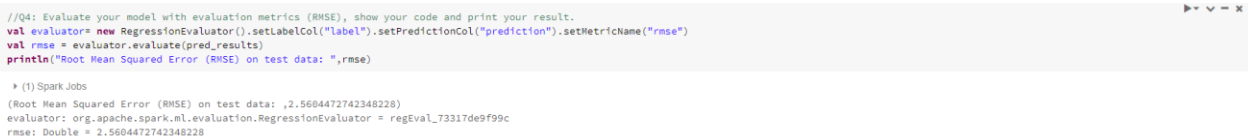


```
//Q3: Create a prediction column and show your prediction. Show your code and dataframe.
pred_results.show(10)
```

▸ (1) Spark Jobs

```
+--------------------+---------+-----------------+
|            features|    label|       prediction|
+--------------------+---------+-----------------+
|[92.040001,93.400...|93.589996|93.95258062754873|
|[92.790001,92.720...|93.419998| 94.866733364036|
|[93.419998,92.790...|92.510002|95.54498008992842|
|[93.879997,90.519...|93.489998|96.36469532744464|
|   [94.190002,95.18]|93.239998|96.05180117201732|
|   [96.43,97.339996]|94.480003|98.19618648950942|
|[96.669998,97.339...|102.949997| 98.458293192755|
|[96.790001,96.660...|96.300003|98.68441211777402|
|[96.870003,97.419...|98.790001|98.66553864392638|
|[96.959999,96.449...|98.529999|98.89942885627954|
+--------------------+---------+-----------------+
only showing top 10 rows
```

Q4: Evaluate your model with evaluation metrics (RMSE), show your code and print your result.



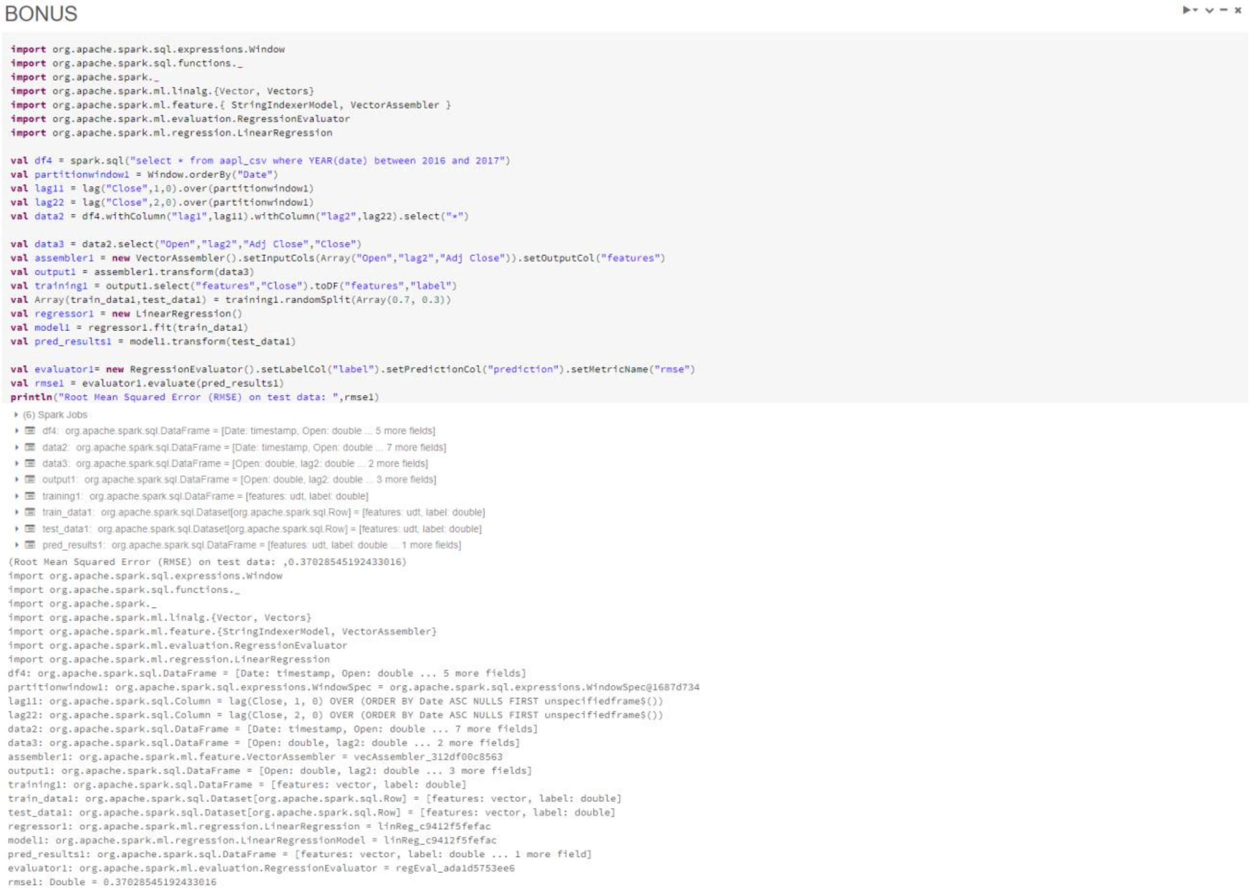
```
//Q4: Evaluate your model with evaluation metrics (RMSE), show your code and print your result.
val evaluator= new RegressionEvaluator().setLabelCol("label").setPredictionCol("prediction").setMetricName("rmse")
val rmse = evaluator.evaluate(pred_results)
println("Root Mean Squared Error (RMSE) on test data: ",rmse)
```

▸ (1) Spark Jobs

```
(Root Mean Squared Error (RMSE) on test data: ,2.5604472742348228)
evaluator: org.apache.spark.ml.evaluation.RegressionEvaluator = regEval_73317de9f99c
rmse: Double = 2.5604472742348228
```

Question5 - Bonus: Play around with features(Only) and try to see if you can get better result(split 0.7/0.3 and linear regression), if you can get top 5 result in the class, you will be reward with 100 percent on this assignment.

Features which can get lowest RMSE are "Open","lag2","Adj Close", results are:



BONUS

```
import org.apache.spark.sql.expressions.Window
import org.apache.spark.sql.functions._
import org.apache.spark._
import org.apache.spark.ml.linalg.{Vector, Vectors}
import org.apache.spark.ml.feature.{ StringIndexerModel, VectorAssembler }
import org.apache.spark.ml.evaluation.RegressionEvaluator
import org.apache.spark.ml.regression.LinearRegression

val df4 = spark.sql("select * from aapl_csv where YEAR(date) between 2016 and 2017")
val partitionwindow1 = Window.orderBy("Date")
val lag11 = lag("Close",1,0).over(partitionwindow1)
val lag22 = lag("Close",2,0).over(partitionwindow1)
val data2 = df4.withColumn("lag1",lag11).withColumn("lag2",lag22).select("*")

val data3 = data2.select("Open","lag2","Adj Close","Close")
val assembler1 = new VectorAssembler().setInputCols(Array("Open","lag2","Adj Close")).setOutputCol("features")
val output1 = assembler1.transform(data3)
val training1 = output1.select("features","Close").toDF("features","label")
val Array(train_data1,test_data1) = training1.randomSplit(Array(0.7, 0.3))
val regressor1 = new LinearRegression()
val model1 = regressor1.fit(train_data1)
val pred_results1 = model1.transform(test_data1)

val evaluator1= new RegressionEvaluator().setLabelCol("label").setPredictionCol("prediction").setMetricName("rmse")
val rmse1 = evaluator1.evaluate(pred_results1)
println("Root Mean Squared Error (RMSE) on test data: ",rmse1)
```

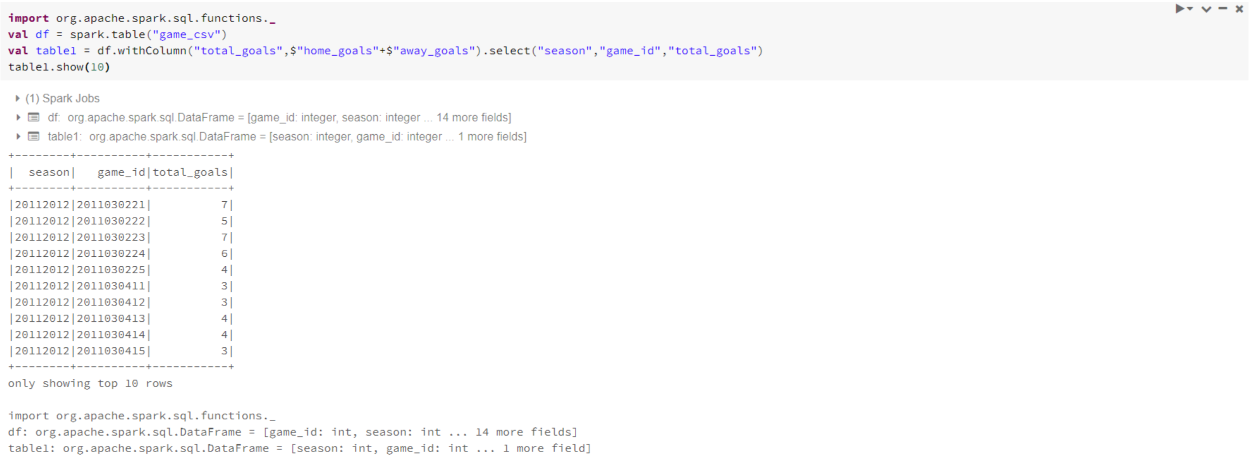
▸ (6) Spark Jobs
▸ df4: org.apache.spark.sql.DataFrame = [Date: timestamp, Open: double ... 5 more fields]
▸ data2: org.apache.spark.sql.DataFrame = [Date: timestamp, Open: double ... 7 more fields]
▸ data3: org.apache.spark.sql.DataFrame = [Open: double, lag2: double ... 2 more fields]
▸ output1: org.apache.spark.sql.DataFrame = [Open: double, lag2: double ... 3 more fields]
▸ training1: org.apache.spark.sql.DataFrame = [features: udt, label: double]
▸ train_data1: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [features: udt, label: double]
▸ test_data1: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [features: udt, label: double]
▸ pred_results1: org.apache.spark.sql.DataFrame = [features: udt, label: double ... 1 more fields]

```
(Root Mean Squared Error (RMSE) on test data: ,0.37028545192433016)
import org.apache.spark.sql.expressions.Window
import org.apache.spark.sql.functions._
import org.apache.spark._
import org.apache.spark.ml.linalg.{Vector, Vectors}
import org.apache.spark.ml.feature.{StringIndexerModel, VectorAssembler}
import org.apache.spark.ml.evaluation.RegressionEvaluator
import org.apache.spark.ml.regression.LinearRegression
df4: org.apache.spark.sql.DataFrame = [Date: timestamp, Open: double ... 5 more fields]
partitionwindow1: org.apache.spark.sql.expressions.WindowSpec = org.apache.spark.sql.expressions.WindowSpec@1687d734
lag11: org.apache.spark.sql.Column = lag(Close, 1, 0) OVER (ORDER BY Date ASC NULLS FIRST unspecifiedframe$())
lag22: org.apache.spark.sql.Column = lag(Close, 2, 0) OVER (ORDER BY Date ASC NULLS FIRST unspecifiedframe$())
data2: org.apache.spark.sql.DataFrame = [Date: timestamp, Open: double ... 7 more fields]
data3: org.apache.spark.sql.DataFrame = [Open: double, lag2: double ... 2 more fields]
assembler1: org.apache.spark.ml.feature.VectorAssembler = vecAssembler_312df00c8563
output1: org.apache.spark.sql.DataFrame = [Open: double, lag2: double ... 3 more fields]
training1: org.apache.spark.sql.DataFrame = [features: vector, label: double]
train_data1: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [features: vector, label: double]
test_data1: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [features: vector, label: double]
regressor1: org.apache.spark.ml.regression.LinearRegression = linReg_c9412f5fefac
model1: org.apache.spark.ml.regression.LinearRegressionModel = linReg_c9412f5fefac
pred_results1: org.apache.spark.sql.DataFrame = [features: vector, label: double ... 1 more field]
evaluator1: org.apache.spark.ml.evaluation.RegressionEvaluator = regEval_ada1d5753ee6
rmse1: Double = 0.37028545192433016
```

Section 2: PySpark/Scala syntax

Question 1 (15 points)

1. Select two columns - games and seasons - and add a column with total goals (sum of home and away goals). Suggestion: use df.withColumn() function.

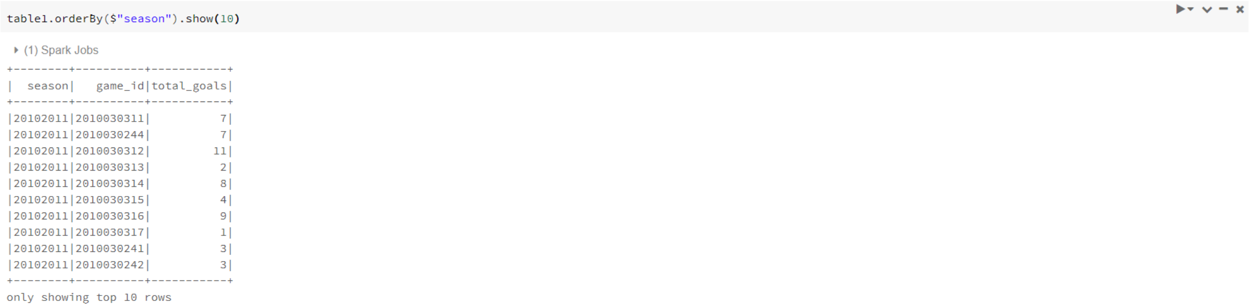


```
import org.apache.spark.sql.functions._
val df = spark.table("game_csv")
val table1 = df.withColumn("total_goals",$"home_goals"+$"away_goals").select("season","game_id","total_goals")
table1.show(10)
```

```
▸ (1) Spark Jobs
▸ df: org.apache.spark.sql.DataFrame = [game_id: integer, season: integer ... 14 more fields]
▸ table1: org.apache.spark.sql.DataFrame = [season: integer, game_id: integer ... 1 more fields]
+--------+----------+-----------+
|  season|   game_id|total_goals|
+--------+----------+-----------+
|20112012|2011030221|          7|
|20112012|2011030222|          5|
|20112012|2011030223|          7|
|20112012|2011030224|          6|
|20112012|2011030225|          4|
|20112012|2011030411|          3|
|20112012|2011030412|          3|
|20112012|2011030413|          4|
|20112012|2011030414|          4|
|20112012|2011030415|          3|
+--------+----------+-----------+
only showing top 10 rows

import org.apache.spark.sql.functions._
df: org.apache.spark.sql.DataFrame = [game_id: int, season: int ... 14 more fields]
table1: org.apache.spark.sql.DataFrame = [season: int, game_id: int ... 1 more field]
```

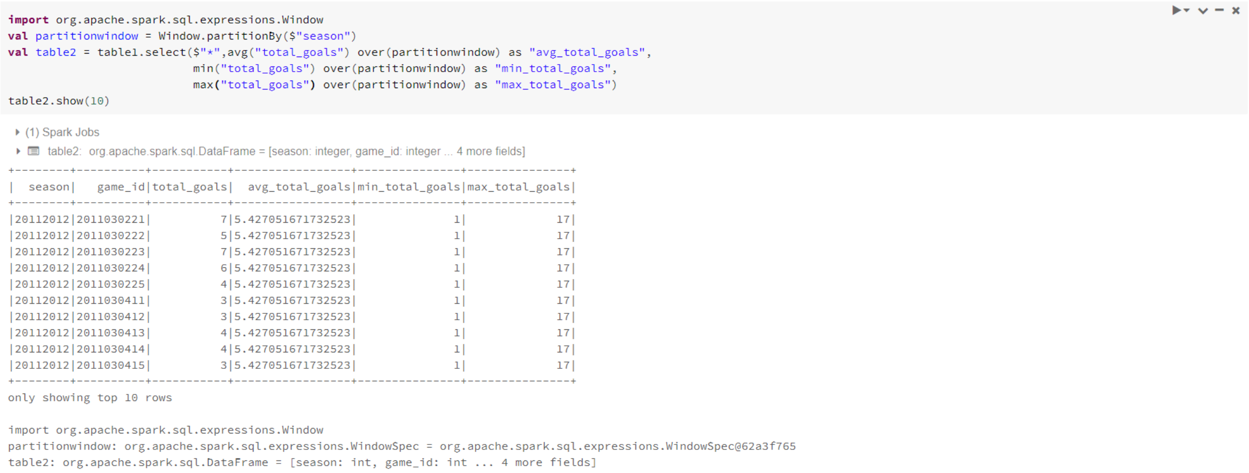
2. Organize records in ascending order (by season).



```
table1.orderBy($"season").show(10)
```

```
▸ (1) Spark Jobs
+--------+----------+-----------+
|  season|   game_id|total_goals|
+--------+----------+-----------+
|20102011|2010030311|          7|
|20102011|2010030244|          7|
|20102011|2010030312|         11|
|20102011|2010030313|          2|
|20102011|2010030314|          8|
|20102011|2010030315|          4|
|20102011|2010030316|          9|
|20102011|2010030317|          1|
|20102011|2010030241|          3|
|20102011|2010030242|          3|
+--------+----------+-----------+
only showing top 10 rows
```

3. Add a column with an average, min and max total score for each season. Suggestion: use Window function.

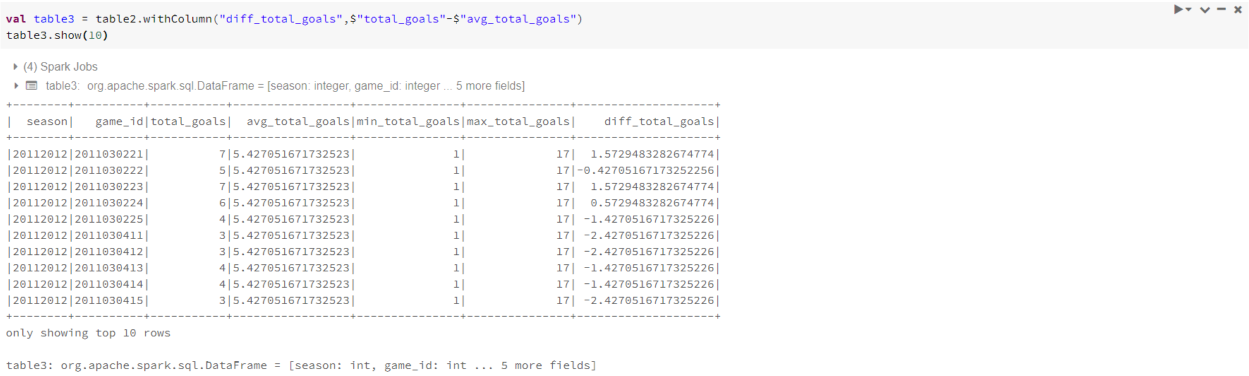


```
import org.apache.spark.sql.expressions.Window
val partitionwindow = Window.partitionBy($"season")
val table2 = table1.select($"*",avg("total_goals") over(partitionwindow) as "avg_total_goals",
                     min("total_goals") over(partitionwindow) as "min_total_goals",
                     max("total_goals") over(partitionwindow) as "max_total_goals")
table2.show(10)
```

```
▸ (1) Spark Jobs
▸ table2: org.apache.spark.sql.DataFrame = [season: integer, game_id: integer ... 4 more fields]
+--------+----------+-----------+-----------------+---------------+---------------+
|  season|   game_id|total_goals|  avg_total_goals|min_total_goals|max_total_goals|
+--------+----------+-----------+-----------------+---------------+---------------+
|20112012|2011030221|          7|5.427051671732523|              1|             17|
|20112012|2011030222|          5|5.427051671732523|              1|             17|
|20112012|2011030223|          7|5.427051671732523|              1|             17|
|20112012|2011030224|          6|5.427051671732523|              1|             17|
|20112012|2011030225|          4|5.427051671732523|              1|             17|
|20112012|2011030411|          3|5.427051671732523|              1|             17|
|20112012|2011030412|          3|5.427051671732523|              1|             17|
|20112012|2011030413|          4|5.427051671732523|              1|             17|
|20112012|2011030414|          4|5.427051671732523|              1|             17|
|20112012|2011030415|          3|5.427051671732523|              1|             17|
+--------+----------+-----------+-----------------+---------------+---------------+
only showing top 10 rows

import org.apache.spark.sql.expressions.Window
partitionwindow: org.apache.spark.sql.expressions.WindowSpec = org.apache.spark.sql.expressions.WindowSpec@62a3f765
table2: org.apache.spark.sql.DataFrame = [season: int, game_id: int ... 4 more fields]
```

4. Add a column that finds a difference between each game's total score and average for that season. Suggestion: use Window function.

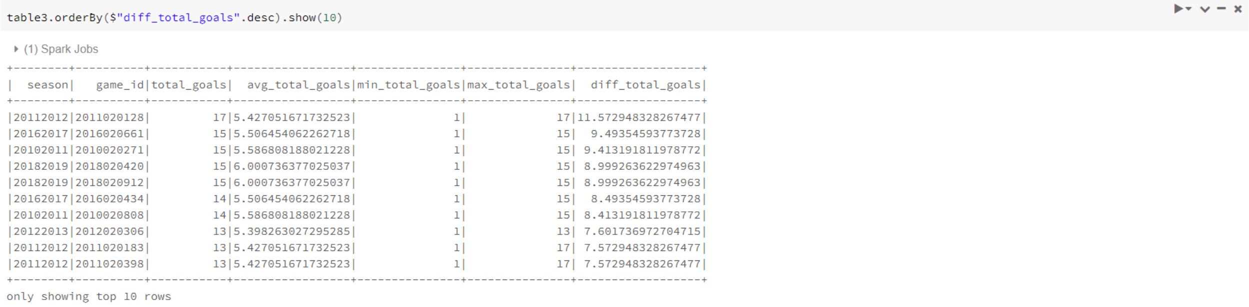


```
val table3 = table2.withColumn("diff_total_goals",$"total_goals"-$"avg_total_goals")
table3.show(10)
```

```
(4) Spark Jobs
table3:  org.apache.spark.sql.DataFrame = [season: integer, game_id: integer ... 5 more fields]
+--------+----------+-----------+-----------------+---------------+---------------+--------------------+
|  season|   game_id|total_goals|  avg_total_goals|min_total_goals|max_total_goals|    diff_total_goals|
+--------+----------+-----------+-----------------+---------------+---------------+--------------------+
|20112012|2011030221|          7|5.427051671732523|              1|             17|  1.5729483282674774|
|20112012|2011030222|          5|5.427051671732523|              1|             17|-0.42705167173252256|
|20112012|2011030223|          7|5.427051671732523|              1|             17|  1.5729483282674774|
|20112012|2011030224|          6|5.427051671732523|              1|             17|  0.5729483282674774|
|20112012|2011030225|          4|5.427051671732523|              1|             17| -1.4270516717325226|
|20112012|2011030411|          3|5.427051671732523|              1|             17| -2.4270516717325226|
|20112012|2011030412|          3|5.427051671732523|              1|             17| -2.4270516717325226|
|20112012|2011030413|          4|5.427051671732523|              1|             17| -1.4270516717325226|
|20112012|2011030414|          4|5.427051671732523|              1|             17| -1.4270516717325226|
|20112012|2011030415|          3|5.427051671732523|              1|             17| -2.4270516717325226|
+--------+----------+-----------+-----------------+---------------+---------------+--------------------+
only showing top 10 rows

table3: org.apache.spark.sql.DataFrame = [season: int, game_id: int ... 5 more fields]
```

5. Print top 10 records.

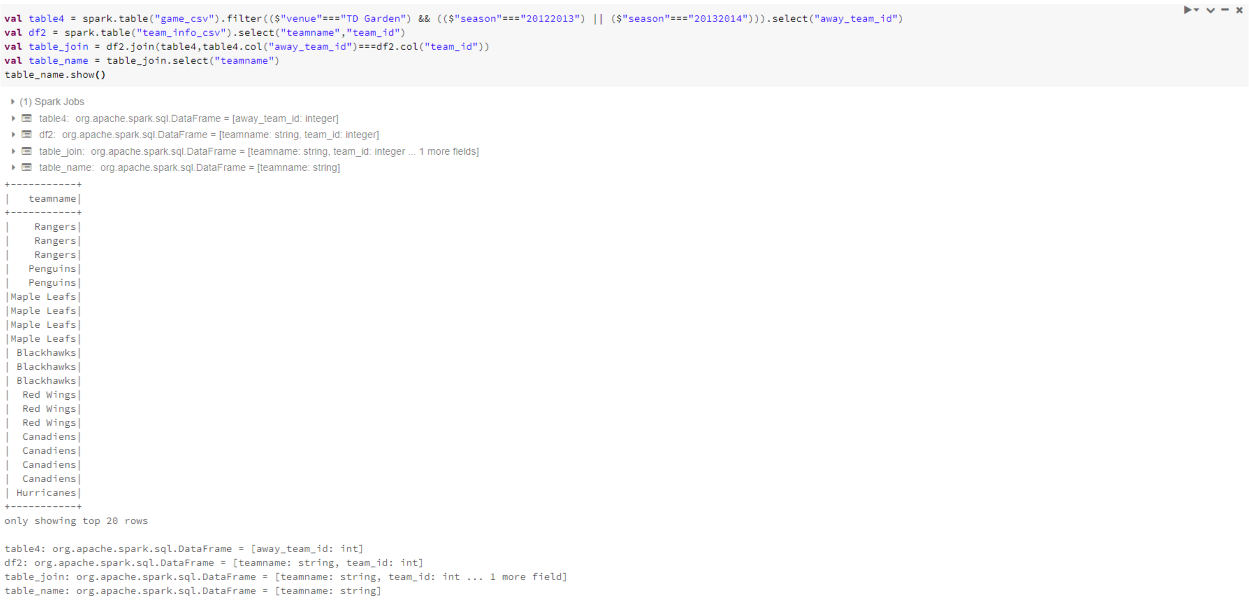


```
table3.orderBy($"diff_total_goals".desc).show(10)
```

```
(1) Spark Jobs
+--------+----------+-----------+-----------------+---------------+---------------+------------------+
|  season|   game_id|total_goals|  avg_total_goals|min_total_goals|max_total_goals|  diff_total_goals|
+--------+----------+-----------+-----------------+---------------+---------------+------------------+
|20112012|2011020128|         17|5.427051671732523|              1|             17|11.572948328267477|
|20162017|2016020661|         15|5.506454062262718|              1|             15|   9.49354593773728|
|20102011|2010020271|         15|5.586808188021228|              1|             15| 9.413191811978772|
|20182019|2018020420|         15|6.000736377025037|              1|             15| 8.999263622974963|
|20182019|2018020912|         15|6.000736377025037|              1|             15| 8.999263622974963|
|20162017|2016020434|         14|5.506454062262718|              1|             15|   8.49354593773728|
|20102011|2010020808|         14|5.586808188021228|              1|             15| 8.413191811978772|
|20122013|2012020306|         13|5.398263027295285|              1|             13| 7.601736972704715|
|20112012|2011020183|         13|5.427051671732523|              1|             17| 7.572948328267477|
|20112012|2011020398|         13|5.427051671732523|              1|             17| 7.572948328267477|
+--------+----------+-----------+-----------------+---------------+---------------+------------------+
only showing top 10 rows
```

Question 2 (10 points)

1. List all team names (teamName) for teams that played as away team at TD Garden during seasons 2012-2013 and 2013-2014.

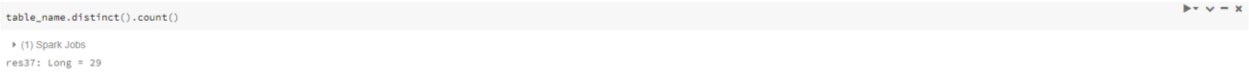


```
val table4 = spark.table("game_csv").filter(($"venue"==="TD Garden") && (($"season"==="20122013") || ($"season"==="20132014"))).select("away_team_id")
val df2 = spark.table("team_info_csv").select("teamname","team_id")
val table_join = df2.join(table4,table4.col("away_team_id")===df2.col("team_id"))
val table_name = table_join.select("teamname")
table_name.show()
```

```
(1) Spark Jobs
table4:  org.apache.spark.sql.DataFrame = [away_team_id: integer]
df2:  org.apache.spark.sql.DataFrame = [teamname: string, team_id: integer]
table_join:  org.apache.spark.sql.DataFrame = [teamname: string, team_id: integer ... 1 more fields]
table_name:  org.apache.spark.sql.DataFrame = [teamname: string]
+-----------+
|   teamname|
+-----------+
|    Rangers|
|    Rangers|
|    Rangers|
|   Penguins|
|   Penguins|
|Maple Leafs|
|Maple Leafs|
|Maple Leafs|
|Maple Leafs|
| Blackhawks|
| Blackhawks|
| Blackhawks|
|  Red Wings|
|  Red Wings|
|  Red Wings|
|  Canadiens|
|  Canadiens|
|  Canadiens|
|  Canadiens|
| Hurricanes|
+-----------+
only showing top 20 rows

table4: org.apache.spark.sql.DataFrame = [away_team_id: int]
df2: org.apache.spark.sql.DataFrame = [teamname: string, team_id: int]
table_join: org.apache.spark.sql.DataFrame = [teamname: string, team_id: int ... 1 more field]
table_name: org.apache.spark.sql.DataFrame = [teamname: string]
```

2. How many unique teams are on the list?



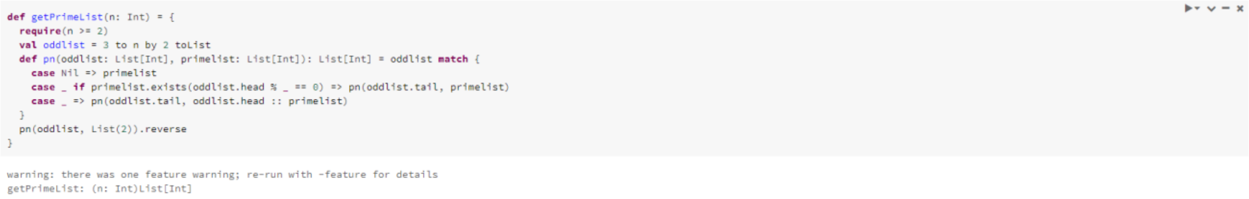
```
table_name.distinct().count()
```

```
▸ (1) Spark Jobs
res37: Long = 29
```

Question 3 (additional 15 points)

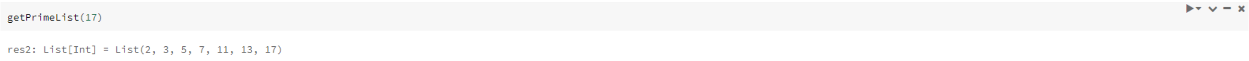
(Bonus Question (doesn’t have to be completed to get a full mark for this assignment, but if you complete it you will get additional 15 points)

- Create a function that when input a number n returns a list of prime numbers between 1 and n.
- Test your function with number 17.



```
def getPrimeList(n: Int) = {
  require(n >= 2)
  val oddlist = 3 to n by 2 toList
  def pn(oddlist: List[Int], primelist: List[Int]): List[Int] = oddlist match {
    case Nil => primelist
    case _ if primelist.exists(oddlist.head % _ == 0) => pn(oddlist.tail, primelist)
    case _ => pn(oddlist.tail, oddlist.head :: primelist)
  }
  pn(oddlist, List(2)).reverse
}
```

```
warning: there was one feature warning; re-run with -feature for details
getPrimeList: (n: Int)List[Int]
```



```
getPrimeList(17)
```

```
res2: List[Int] = List(2, 3, 5, 7, 11, 13, 17)
```