<https://drive.google.com/open?id=0ByJLBTmJojjzRm5vX1E2cURtTTQ>

**id,season,city,date,team1,team2,toss\_winner,toss\_decision,result,dl\_applied,winner,win\_by\_runs,win\_by\_wickets,player\_of\_match,venue,umpire1,umpire2,umpire3**

**1.Which stadium is best suitable for first batting**

Here we evaluate that which stadium is most suitable for first batting. Here are the details how can we do that.

win\_by\_runs means – Team batted first and won the Match by margin of some runs.

win\_by\_wickets means – Team batted second and chased the target successfully.

So we will take out the columns **toss\_decision, won\_by\_runs, won\_by\_wickets, venue**. From this we will filter out the columns which are having **won\_by\_runs**value as **0**so that we can get the teams which won by batting first. Here is the scala code to do that.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **val** data = sc.textFile("file:///home/kiran/Documents/datasets/matches.csv")    **val** filtering\_bad\_records = data.map(line**=>**line.split(",")).filter(x**=>**x.length<19)    **val** extracting\_columns = filtering\_bad\_records.map(x=>(x(7),x(11),x(12),x(14)))    **val** bat\_first\_won = extracting\_columns.filter(x**=>**x.\_2!="0").map(x=>(x.\_4,1)) .reduceByKey(**\_**+**\_**).map(item => item.swap).sortByKey(**false**).collect.**foreach**(println) |

**Code Explanation**

In the first line of code we are loading the data from the local file system.

In the second line we are filtering the bad records if any are there i.e., the total number of columns are 19 if any record having less than 19 columns are filtered out.

In the third line we are extracting the columns that are required for our analysis i.e., **toss\_decision, won\_by\_runs, won\_by\_wickets, venue.**

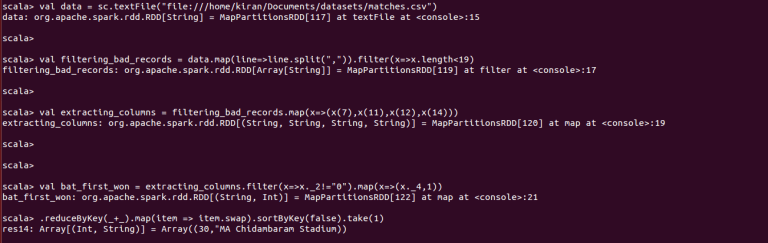
In the fourth line we are filtering the **won\_by\_run**column having more than 0 runs and we are preparing a key-value pair with the Venue column and a numeric 1 has been added to it so as to count the number of first\_bat\_wons in that stadium and finally we are sorting the records and printing all of them.

**Output**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65 | (30,"MA Chidambaram Stadium)    (25,Wankhede Stadium)    (24,M Chinnaswamy Stadium)    (24,Feroz Shah Kotla)    (22,Eden Gardens)    (15,"Punjab Cricket Association Stadium)    (14,"Rajiv Gandhi International Stadium)    (11,Subrata Roy Sahara Stadium)    (10,Sawai Mansingh Stadium)    (9,Kingsmead)    (7,Dr DY Patil Sports Academy)    (7,Dr. Y.S. Rajasekhara Reddy ACA-VDCA Cricket Stadium)    (6,"Sardar Patel Stadium)    (6,Brabourne Stadium)    (5,Himachal Pradesh Cricket Association Stadium)    (4,Newlands)    (4,SuperSport Park)    (4,Barabati Stadium)    (3,St George's Park)    (3,Sheikh Zayed Stadium)    (3,New Wanderers Stadium)    (3,Nehru Stadium)    (3,Maharashtra Cricket Association Stadium)    (3,"Punjab Cricket Association IS Bindra Stadium)    (3,Dubai International Cricket Stadium)    (2,Shaheed Veer Narayan Singh International Stadium)    (2,"Vidarbha Cricket Association Stadium)    (2,JSCA International Stadium Complex)    (2,Buffalo Park)    (2,Sharjah Cricket Stadium)    (1,De Beers Diamond Oval)    (1,OUTsurance Oval)    (1,Saurashtra Cricket Association Stadium) |

From this analysis as of now, we have got ***(30,”MA Chidambaram Stadium)***

Here is the screen shot of the whole stack trace.



But this is not the final result, we need to evaluate the total number of matches that chidambaram stadium has been venue.

Let us see how many matches that each stadium has been venued. Here is the code to do that

|  |  |
| --- | --- |
| 1  2  3  4  5 | **val** data = sc.textFile("file:///home/kiran/Documents/datasets/matches.csv")    **val** filtering\_bad\_records1 = data.map(line**=>**line.split(",")).filter(x**=>**x.length<19)    **val** total\_matches\_per\_venue = filtering\_bad\_records.map(x=>(x(14),1)).reduceByKey(**\_**+**\_**).map(item => item.swap).sortByKey(**false**).collect.**foreach**(println) |

Here are the total number of matches each stadium has been venued.

**Output**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69 | (58,M Chinnaswamy Stadium)    (54,Eden Gardens)    (53,Feroz Shah Kotla)    (49,Wankhede Stadium)    (48,"MA Chidambaram Stadium)    (41,"Rajiv Gandhi International Stadium)    (35,"Punjab Cricket Association Stadium)    (33,Sawai Mansingh Stadium)    (17,Dr DY Patil Sports Academy)    (17,Subrata Roy Sahara Stadium)    (15,Kingsmead)    (12,"Sardar Patel Stadium)    (12,SuperSport Park)    (11,Dr. Y.S. Rajasekhara Reddy ACA-VDCA Cricket Stadium)    (11,Brabourne Stadium)    (9,Himachal Pradesh Cricket Association Stadium)    (8,New Wanderers Stadium)    (8,Maharashtra Cricket Association Stadium)    (7,Newlands)    (7,St George's Park)    (7,Sheikh Zayed Stadium)    (7,JSCA International Stadium Complex)    (7,"Punjab Cricket Association IS Bindra Stadium)    (7,Dubai International Cricket Stadium)    (7,Barabati Stadium)    (6,Shaheed Veer Narayan Singh International Stadium)    (6,Sharjah Cricket Stadium)    (5,Nehru Stadium)    (5,Saurashtra Cricket Association Stadium)    (3,De Beers Diamond Oval)    (3,"Vidarbha Cricket Association Stadium)    (3,Buffalo Park)    (2,OUTsurance Oval)    (2,Green Park)    (2,Holkar Cricket Stadium) |

Chidambaram stadium has venued 48 matches in-total, in that 30 teams won by batting first.

So we will now see the winning percentage of each stadium for first\_bat\_won. Here is the code to do that.

|  |  |
| --- | --- |
| 1 | **val** join = bat\_first\_won.join(total\_matches\_per\_venue).map(x=>(x.\_1,(x.\_2.\_1\*100/x.\_2.\_2))).map(item => item.swap).sortByKey(**false**).collect.**foreach**(println) |

Here we have joined the two RDD’s i.e., **bat\_first\_won**and **total\_matches\_per\_venue**and we have drawn out the percentage of first\_bat\_won venues by dividing the **number of matches won by batting first**and the **total number of matches in that venue.**

Here is the result of the percentages of each venue for first\_bat\_won

**Output**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65 | (66,"Vidarbha Cricket Association Stadium)    (66,Buffalo Park)    (64,Subrata Roy Sahara Stadium)    (63,Dr. Y.S. Rajasekhara Reddy ACA-VDCA Cricket Stadium)    (62,"MA Chidambaram Stadium)    (60,Kingsmead)    (60,Nehru Stadium)    (57,Newlands)    (57,Barabati Stadium)    (55,Himachal Pradesh Cricket Association Stadium)    (54,Brabourne Stadium)    (51,Wankhede Stadium)    (50,"Sardar Patel Stadium)    (50,OUTsurance Oval)    (45,Feroz Shah Kotla)    (42,St George's Park)    (42,Sheikh Zayed Stadium)    (42,"Punjab Cricket Association IS Bindra Stadium)    (42,Dubai International Cricket Stadium)    (42,"Punjab Cricket Association Stadium)    (41,Dr DY Patil Sports Academy)    (41,M Chinnaswamy Stadium)    (40,Eden Gardens)    (37,New Wanderers Stadium)    (37,Maharashtra Cricket Association Stadium)    (34,"Rajiv Gandhi International Stadium)    (33,Shaheed Veer Narayan Singh International Stadium)    (33,De Beers Diamond Oval)    (33,SuperSport Park)    (33,Sharjah Cricket Stadium)    (30,Sawai Mansingh Stadium)    (28,JSCA International Stadium Complex)    (20,Saurashtra Cricket Association Stadium) |

***Vidarbha Cricket Association Stadium****stands in the first place, but the total number of matches held there was only****3.***

When we take the list of stadiums the top 4 stadiums with the highest first bat win percentage venued below 20 matches. But in Chidambaram stadium there held total 48 matches. When we take that streak, we can deduce that Ma Chidambaram Stadium is most suitable for first batting in the IPL.

In the similar way, let us see for the stadium which supports bowling

**2.Which stadium is best suitable for first bowling**

Here we evaluate that which stadium is most suitable for first batting. Here are the details how can we do that.

win\_by\_runs means – first bat won or second bowl

win\_by\_wickets means – second bat won or first bowl

So we will take out the columns **toss\_decision, won\_by\_runs, won\_by\_wickets, venue**. From this we will filter out the columns which are having **won\_by\_wickets**value as **0**so that we can get the teams which won by batting first. Here is the scala code to do that.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **val** data = sc.textFile("file:///home/kiran/Documents/datasets/matches.csv")    **val** filtering\_bad\_records = data.map(line**=>**line.split(",")).filter(x**=>**x.length<19)    **val** extracting\_columns = filtering\_bad\_records.map(x=>(x(7),x(11),x(12),x(14)))    **val** bowl\_first\_won = extracting\_columns.filter(x**=>**x.\_3!="0").map(x=>(x.\_4,1)) .reduceByKey(**\_**+**\_**).map(item => item.swap).sortByKey(**false**).collect.**foreach**(println) |

**Code Explanation**

In the first line of code we are loading the data from the local file system.

In the second line we are filtering the bad records if any are there i.e., the total number of columns are 19 if any record having less than 19 columns are filtered out.

In the thrid line we are extracting the columns that are required for our analysis i.e., **toss\_decision, won\_by\_runs, won\_by\_wickets, venue.**

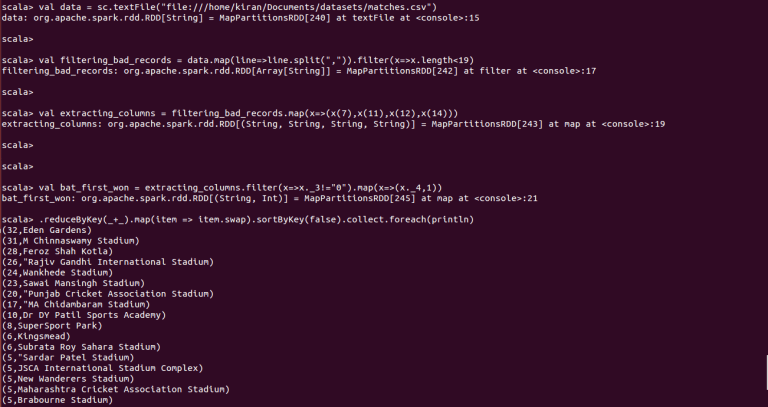
In the fourth line we are filtering the **won\_by\_wickets**column having more than 0 wickets and we are preparing a key-value pair with the Venue column and a numeric 1 has been added to it so as to count the number of first\_bowl\_wons in that stadium and finally we are sorting the records and printing all of them.

Here is the result of this analysis

**Output**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69 | (32,Eden Gardens)    (31,M Chinnaswamy Stadium)    (28,Feroz Shah Kotla)    (26,"Rajiv Gandhi International Stadium)    (24,Wankhede Stadium)    (23,Sawai Mansingh Stadium)    (20,"Punjab Cricket Association Stadium)    (17,"MA Chidambaram Stadium)    (10,Dr DY Patil Sports Academy)    (8,SuperSport Park)    (6,Kingsmead)    (6,Subrata Roy Sahara Stadium)    (5,"Sardar Patel Stadium)    (5,JSCA International Stadium Complex)    (5,New Wanderers Stadium)    (5,Maharashtra Cricket Association Stadium)    (5,Brabourne Stadium)    (4,Shaheed Veer Narayan Singh International Stadium)    (4,St George's Park)    (4,Himachal Pradesh Cricket Association Stadium)    (4,Sharjah Cricket Stadium)    (4,Saurashtra Cricket Association Stadium)    (4,Dr. Y.S. Rajasekhara Reddy ACA-VDCA Cricket Stadium)    (4,"Punjab Cricket Association IS Bindra Stadium)    (4,Dubai International Cricket Stadium)    (3,Sheikh Zayed Stadium)    (3,Barabati Stadium)    (2,Newlands)    (2,De Beers Diamond Oval)    (2,Nehru Stadium)    (2,Green Park)    (2,Holkar Cricket Stadium)    (1,"Vidarbha Cricket Association Stadium)    (1,OUTsurance Oval)    (1,Buffalo Park) |

We can see that Eden Gardens stands in the first place. Here is the screen shot of the whole stack trace.



Now we will see the percentage of first\_bowl\_won by taking the percentage of first\_bowl\_won and the total number of matches held in that stadium.

This code will find out the total of number of matches each stadium has venued.

|  |  |
| --- | --- |
| 1  2  3  4  5 | **val** data = sc.textFile("file:///home/kiran/Documents/datasets/matches.csv")    **val** filtering\_bad\_records1 = data.map(line**=>**line.split(",")).filter(x**=>**x.length<19)    **val** total\_matches\_per\_venue = filtering\_bad\_records.map(x=>(x(14),1)).reduceByKey(**\_**+**\_**).map(item => item.swap).sortByKey(**false**).collect.**foreach**(println) |

Now we will perform a join operation on the total number of matches in that venue and bowl\_first\_won and it can be done as follows:

|  |  |
| --- | --- |
| 1 | **val** join1 = bowl\_first\_won.join(total\_matches\_per\_venue).map(x=>(x.\_1,(x.\_2.\_1\*100/x.\_2.\_2))).map(item => item.swap).sortByKey(**false**).collect.**foreach**(println) |

Here is the percentage of first\_bowl\_won matches for each stadium

**Output**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69 | (100,Green Park)    (100,Holkar Cricket Stadium)    (80,Saurashtra Cricket Association Stadium)    (71,JSCA International Stadium Complex)    (69,Sawai Mansingh Stadium)    (66,Shaheed Veer Narayan Singh International Stadium)    (66,De Beers Diamond Oval)    (66,SuperSport Park)    (66,Sharjah Cricket Stadium)    (63,"Rajiv Gandhi International Stadium)    (62,New Wanderers Stadium)    (62,Maharashtra Cricket Association Stadium)    (59,Eden Gardens)    (58,Dr DY Patil Sports Academy)    (57,St George's Park)    (57,"Punjab Cricket Association IS Bindra Stadium)    (57,Dubai International Cricket Stadium)    (57,"Punjab Cricket Association Stadium)    (53,M Chinnaswamy Stadium)    (52,Feroz Shah Kotla)    (50,OUTsurance Oval)    (48,Wankhede Stadium)    (45,Brabourne Stadium)    (44,Himachal Pradesh Cricket Association Stadium)    (42,Sheikh Zayed Stadium)    (42,Barabati Stadium)    (41,"Sardar Patel Stadium)    (40,Kingsmead)    (40,Nehru Stadium)    (36,Dr. Y.S. Rajasekhara Reddy ACA-VDCA Cricket Stadium)    (35,"MA Chidambaram Stadium)    (35,Subrata Roy Sahara Stadium)    (33,"Vidarbha Cricket Association Stadium)    (33,Buffalo Park)    (28,Newlands) |

Green park stands in the first place but the number of matches held there was only 2 but the total number of matches held at Eden gardens were 40, out of 40 matches 32 matches won by bowling first. So if we take that winning streak, Eden gardens is most suitable for bowling first.

This post will help you to understand how to handle data sets that does not have proper structure and how to sort the output of reducer.

**Data Set Description**

**Column 1:** Video id of 11 characters.

**Column 2:** Uploader of the video.

**Column 3:** Interval between day of establishment of YouTube and the date of uploading of the video.

**Column 4:** Category of the video.

**Column 5:** Length of the video.

**Column 6:** Number of views for the video.

**Column 7:** Rating on the video.

**Column 8:** Number of ratings given for the video

**Column 9:** Number of comments on the videos.

**Column 10:** Related video ids with the uploaded video.

You can download the data set [from here](https://drive.google.com/file/d/0ByJLBTmJojjzR2x0MzVpc2Z6enM/view?usp=drive_web)

**Problem Statement 1:**

Here, we will find out what are the top five categories with maximum number of videos uploaded.

**Source Code:**

|  |  |
| --- | --- |
| 1  2  3  4 | **val** textFile = sc.textFile("hdfs://localhost:9000/youtubedata.txt")  **val** counts = textFile.map(line=>{**var** YoutubeRecord = "";  **val** temp**=**line.split("\t");  ;**if**(temp.length >= 3) {  YoutubeRecord**=**temp(3)  };YoutubeRecord})  **val** test**=**counts.map ( x => (x,1) )  **val** res**=**test.reduceByKey(**\_**+**\_**).map(item => item.swap).sortByKey(**false**).take(5) |

**Walk Through of the Above Program:**

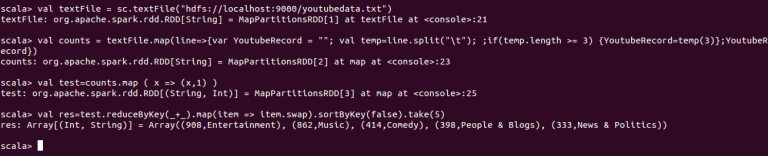
* In **line 1,** we are creating an [RDD](https://acadgild.com/big-data/apache-spark-training-certification)with the existing dataset, which is inside HDFS.
* In **line 2,** we are taking each record as input using the map method and extracting the 4th column, which is the category of the video.
* In **line 3,**we are creating a pair of category\_name,1(count) which is used to calculate how many times that the category is present.

In **line 4,**we are using the reduceByKey method so that all the values of that key are aggregated. Then we are swapping the category\_name and its count, and sorting the result with this we will get the sorted records of category\_name and its count in descending order. Finally, we are taking the top five from the list.

**Output:**

***(908,Entertainment), (862,Music), (414,Comedy), (398,People & Blogs), (333,News & Politics)***

You can this result in the below screenshot.



**Problem Statement 2:**

In this problem statement, we will find the top 10 rated videos in YouTube.

|  |  |
| --- | --- |
| 1  2  3  4 | **val** textFile = sc.textFile("hdfs://localhost:9000/youtubedata.txt")  **val** counts = textFile.filter { x => {**if**(x.toString().split("\t").length >= 6) **true else false**} }.map(line=>{line.toString().split("\t")})  **val** pairs = counts.map(x => {(x(0),x(6).toDouble)})  **val** res**=**pairs.reduceByKey(**\_**+**\_**).map(item => item.swap).sortByKey(**false**).take(10) |

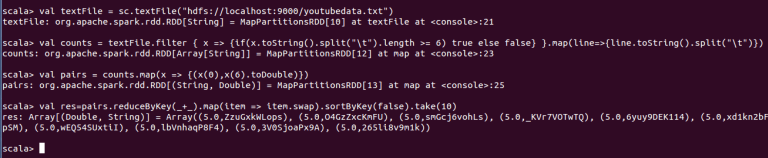
**Walk Through of the Above Program:**

* In **line 1,** we are creating an RDD with the existing dataset, which is inside HDFS.
* In **line2,** we are first filtering the lines with more than six elements to avoid ***ArrayIndexOutOfBounds Exception*** and then we are using **map**method to pass the split line as output to the next RDD.
* In **line 3,**we are creating a pair of key and value by using the video id, which is the first column and 7thcolumn 7 respectively.
* In **line 4,**we are using the reduceByKey method to find the ratings of the video, and to sort them by value, we are using the map method and swapping the key and value. Now, values become the keys and we are performing the ***sortByKey***method and sorting the videos based on their rating and taking the top 10 videos.

**Output:**

***(5.0,ZzuGxkWLops), (5.0,O4GzZxcKmFU), (5.0,smGcj6vohLs), (5.0,\_KVr7VOTwTQ), (5.0,6yuy9DEK114), (5.0,xd1kn2bFpSM), (5.0,wEQ54SUxtiI), (5.0,lbVnhaqP8F4), (5.0,3V0SjoaPx9A), (5.0,265li8v9m1k)***

This output can be seen in the below screenshot.



Hope this post has been helpful in understanding how to perform simple data analysis using Spark and Scala.

*From <*[*https://acadgild.com/blog/spark-use-case-youtube-data-analysis/*](https://acadgild.com/blog/spark-use-case-youtube-data-analysis/)*>*

There have been huge disasters throughout the history of mankind, but the magnitude of the Titanic’s disaster ranks as one of the highest. So much so that the subsequent disasters have always been described as “Titanic in proportion,” implying huge losses.

Anyone who has ever read about the Titanic knows that a perfect combination of natural events and human errors had led to the sinking of the Titanic on its fateful maiden journey from Southampton to New York on April 14, 1912.

There have been several questions put forward to understand the cause(s) of the tragedy; foremost among them is, what made it sink and even more intriguing, how can a 46,000-ton ship sink to the depth of 13,000 feet in a matter of 3 hours? This is a mind-boggling question indeed!

There have been as many investigations as there have and is still poses too many questions and an equal types of analysis methods have been applied to arrive at a conclusion. This post is not about analyzing why or what made the Titanic sink; it is about analysing the data present about the Titanic. This Titanic data is public-ally available and the titanic data set is described below under the heading Data Set Description.

Using this dataset, we will perform some analysis and will draw out some insights, like finding the average age of male and females who died in the Titanic, and the number of males and females who died in each compartment.

**Data Set Description:**

**Column 1**: PassengerId

**Column 2**: Survived  (survived=0 & died=1)

**Column 3**: Pclass

**Column 4**: Name

**Column 5**: Sex

**Column 6**: Age

**Column 7**: SibSp

**Column 8**: Parch

**Column 9**: Ticket

**Column 10**: Fare

**Column 11**: Cabin

**Column 12**: Embarked

You can download the data set [from here](https://drive.google.com/open?id=0ByJLBTmJojjzNmV0dk1EMmwwQ1U)

**Problem Statement 1:**

In this problem statement, we will find the average age of males and females who died in the Titanic tragedy.

**Source Code:**

|  |  |
| --- | --- |
| 1  2  3  4 | **val** textFile = sc.textFile("hdfs://localhost:9000/TitanicData.txt")  **val** split = textFile.filter { x => {**if**(x.toString().split(",").length >= 6) **true else false**} }.map(line=>{line.toString().split(",")})  **val** key\_value = split.filter{x**=>if**((x(1)=="1")&&(x(5).matches(("\\d+"))))**true else false**}.map(x => {(x(4),x(5).toInt)})  key\_value.mapValues((**\_**, 1)).reduceByKey((x, y) => (x.\_1 + y.\_1, x.\_2 + y.\_2)).mapValues{ **case** (sum, count) => (1.0 \* sum)/count}.collectAsMap() |

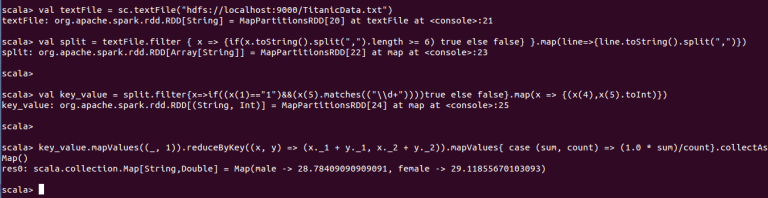
**Walk Through of the Above Code:**

* In line 1, we are creating a new [RDD](https://acadgild.com/big-data/apache-spark-training-certification) by loading a new dataset which is in HDFS.
* In line 2, we are filtering the lines which has more than 7 columns so as to avoid ArrayIndexOutOfBound Exception after the filter we are using the map method to split the line as the dataset is coma ‘,’ delimited we are splitting the line by the parameter ‘,’.
* In line 3, we are filtering the splitted by using two conditions. First condition is that the person should be dead (as per the data description survived means 0 and died means 1 in the 2nd column) and the second condition is that the data in the 6th column should be numerical (we are achieving it by the regular expression [\\d+](file:///\\d+)). If the two conditions are satisfied, then we are creating the key and values. Key is the person’s gender, which is in 5th column and his/her age is in 6th column.
* In line 4, we are performing the average of the values for each unique key so that we will get the average age of male and female who died. We have achieved it by using the Map Values method. The MapValues method pass each value in the key-value pair RDD through a map function without changing the keys. Now, we are passing the values to the reduceByKey method so that it will add all the values as per their keys. In addition, at the last, we are again using the Map Values method to calculate the average value, by calculating the sum/count and finally, we are collecting the value’s result as map.

**Output:**

***male -> 28.78409090909091, female -> 29.11855670103093***

You can see the achieved result in the below screen shot.



**Problem Statement 2:**

In this problem statement, we will find the number of people who died or survived in each class, along with their gender and age.

In the Titanic, there are 3 classes; 1,2,3, which is in the 3rd column and the information about their mortality (alive or dead during the tragedy) is present in the 2nd column (0 for survived and 1 for dead). The details about the gender is present in the 5th column and their age is present in the 6th column.

Here, we will club all the four columns as a key, so that we will get the details about the mortality, in each class along with their age and gender.

**Source Code:**

|  |  |
| --- | --- |
| 1  2  3 | **val** textFile = sc.textFile("hdfs://localhost:9000/TitanicData.txt")  **val** split = textFile.filter { x => {**if**(x.toString().split(",").length >= 6) **true else false**} }.map(line=>{line.toString().split(",")})  **val** count**=**split.map(x=>(x(1)+" "+x(4)+" "+x(6)+" "+x(2),1)).reduceByKey(**\_**+**\_**).collect |

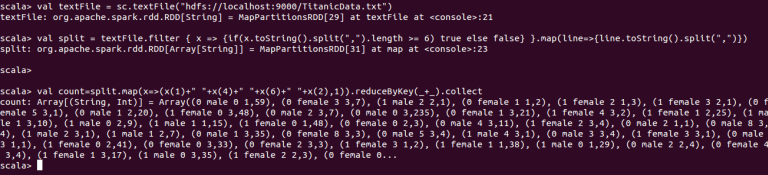
**Walk Through of the Above Code:**

* In **line 1,**we are creating a new RDD by loading a new dataset, which is in HDFS.
* In **line 2,**we are filtering the lines, which has more than 7 columns, so as to avoid ArrayIndexOutOfBound Exception after the filter. We are using the map method to split the line as the dataset is coma ‘,’ delimited and we are splitting the line by the parameter ‘,’.
* In **line 3,**we are clubbing the columns 2,5,6,3, which contains Survived, Gender, Age, Passenger class respectively, and we are making it as key. Here, we are giving ‘1’ as value so as to count them and we are passing this key and value to the reduceByKey method so that it will count and return the required result.

**Output:**

***(0 male 0 1,59), (0 female 3 3,7), (1 male 2 2,1), (0 female 1 1,2), (1 female 2 1,3), (1 female 3 2,1), (0 female 5 3,1), (0 male 1 2,20), (1 female 0 3,48), (0 male 2 3,7), (0 male 0 3,235), (0 female 1 3,21), (1 female 4 3,2), (1 female 1 2,25), (1 male 1 3,10), (1 male 0 2,9), (1 male 1 1,15), (1 female 0 1,48), (0 female 0 2,3), (0 male 4 3,11), (1 female 2 3,4), (0 male 2 1,1), (0 male 8 3,4), (1 male 2 3,1), (1 male 1 2,7), (0 male 1 3,35), (0 female 8 3,3), (0 male 5 3,4), (1 male 4 3,1), (0 male 3 3,4), (1 female 3 3,1), (0 male 3 1,1), (1 female 0 2,41), (0 female 0 3,33), (0 female 2 3,3), (1 female 3 1,2), (1 female 1 1,38), (1 male 0 1,29), (0 male 2 2,4), (0 female 4 3,4), (1 female 1 3,17), (1 male 0 3,35), (1 female 2 2,3)***

The same output can be seen in the below screen shot.



Hope this blog has been helpful in understanding how to perform simple data analysis using [Spark and Scala.](https://acadgild.com/big-data/apache-spark-training-certification)

*From <*[*https://acadgild.com/blog/spark-use-case-titanic-data-analysis/*](https://acadgild.com/blog/spark-use-case-titanic-data-analysis/)*>*

Spark Use Case – Popular Movie Analysis

Friday, September 23, 2016

3:20 PM

In this blog, we will work on a case study to find the list of most popular movies. Spark use Case

We will perform various transformations and actions to display a list of movies with maximum occurrence in the given data set.

Let’s  start our discussion with the data definition by considering a sample of four records.

|  |  |  |  |
| --- | --- | --- | --- |
| 196 | 242 | 3 | 881250949 |
| 186 | 302 | 3 | 891717742 |
| 22 | 377 | 1 | 878887116 |
| 244 | 51 | 2 | 880606923 |

**Data Definition:**

**Column 1:**  User ID

**Column 2:**  Movie ID

**Column 3:** Rating

**Column 4:**Time stamp

The input file can be downloaded from [here.](https://drive.google.com/file/d/0Bxr27gVaXO5sbGdjUWFSRUxUMXM/view?usp=sharing)

The statement in the screenshot below, loads the data file by creating the [RDD through](https://acadgild.com/big-data/apache-spark-training-certification)**sc.textFile**method. The data file is loaded into RDD**my\_lines** and the textFile property breaks every line of text into a value in the RDD.



Map method is called on my\_lines RDD and  key value pair of movie ID and 1 appended to it as value is pulled out and stored in my\_movies RDD.

C:\Users\KAMAKS~1\AppData\Local\Temp\msohtmlclip1\02\clip_image002.png

To see the first 10 records of the my\_lines RDD, the take action has been called on my\_linesRDD.

C:\Users\KAMAKS~1\AppData\Local\Temp\msohtmlclip1\02\clip_image003.png

C:\Users\KAMAKS~1\AppData\Local\Temp\msohtmlclip1\02\clip_image004.png

In this step we call reduceByKey transformation on my\_movies RDD  which group together and aggregate all the values seen  for each individual Movie ID and add all 1’s associated with that MOVIE ID  and we are able to calculate how many times each movie ID occurs  in the my\_movies RDD.

C:\Users\KAMAKS~1\AppData\Local\Temp\msohtmlclip1\02\clip_image005.png

To see the first 10 records of the movie\_Counts RDD, the take action has been called.

C:\Users\KAMAKS~1\AppData\Local\Temp\msohtmlclip1\02\clip_image006.png

C:\Users\KAMAKS~1\AppData\Local\Temp\msohtmlclip1\02\clip_image007.png

Movie\_Counts RDD has entries in key value format where  key is the movie ID and the value is the number of occurrence for that movie ID.

In order to sort the entries by value we make keys i.e movies\_Counts as value and value i.e count of occurrence of each movie ID as key by using the script given in the below screenshot.

C:\Users\KAMAKS~1\AppData\Local\Temp\msohtmlclip1\02\clip_image008.png

C:\Users\KAMAKS~1\AppData\Local\Temp\msohtmlclip1\02\clip_image009.png

C:\Users\KAMAKS~1\AppData\Local\Temp\msohtmlclip1\02\clip_image010.png

In this step we perform sortByKey on flipped\_op to sort the RDD by key i.e count of the occurrence of each movie ID.

C:\Users\KAMAKS~1\AppData\Local\Temp\msohtmlclip1\02\clip_image011.png

C:\Users\KAMAKS~1\AppData\Local\Temp\msohtmlclip1\02\clip_image012.png

C:\Users\KAMAKS~1\AppData\Local\Temp\msohtmlclip1\02\clip_image013.png

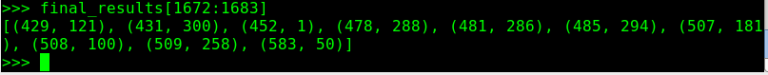
In this step we collect the sorted results in a python list named as final\_results.

C:\Users\KAMAKS~1\AppData\Local\Temp\msohtmlclip1\02\clip_image014.png

In this step we need to find the number of key value pairs present in the python object  final\_results.

C:\Users\KAMAKS~1\AppData\Local\Temp\msohtmlclip1\02\clip_image015.png

We extract out the last 10 pairs of key value pairs and find that the most popular movie with  movie ID 50 has 583 occurrences in the data set.



We hope this post has been helpful in understanding this [Spark use case](https://acadgild.com/big-data/apache-spark-training-certification) using Python. In case of any queries, feel free to comment below and we will get back to you at the earliest.

*From <*[*https://acadgild.com/blog/spark-use-case-popular-movie-analysis/*](https://acadgild.com/blog/spark-use-case-popular-movie-analysis/)*>*

Spark Use Case – Analyzing MovieLens Dataset

Friday, September 23, 2016

3:25 PM

In this blog, we will discuss a use case involving MovieLens dataset and try to analyze how the movies fare on a rating scale of 1 to 5.

We will start our discussion with the data definition by considering a sample of four records.

|  |  |  |  |
| --- | --- | --- | --- |
| 196 | 242 | 3 | 881250949 |
| 186 | 302 | 3 | 891717742 |
| 22 | 377 | 1 | 878887116 |
| 244 | 51 | 2 | 880606923 |

**Data Definition**

Column 1: User ID

Column 2: Movie ID

Column 3: Rating

Column 4: Timestamp

You can download the input file [from here](https://drive.google.com/file/d/0Bxr27gVaXO5sRUZnMjBQR0lqNDA/view?usp=sharing).

Below is the code that is used to calculate the number of movies that are rated on a scale of 1 to 5.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | **from** pyspark **import** SparkConf,SparkContext    **import collections**    my\_lines = sc.textFile('/home/acadgild/ml-100k/u.data')    ratings = lines.**map**(**lambda** x : x.split()[2])    res = ratings.countByValue()    my\_sortedres = **collections**.OrderedDict(**sorted**(res.items()))    **for** key,value **in** sortedres.items():    **print** ("%s %i" %(key,value)) |

The first two lines of the code import SparkConf ,[SparkContext](https://acadgild.com/big-data/apache-spark-training-certification) from pyspark libraries that are present in Spark.

SparkContext is the fundamental starting point in Spark that enables us to [create RDDs.](https://acadgild.com/big-data/apache-spark-training-certification)

Once the collections are imported, the final results have to be sorted out.

C:\Users\KAMAKS~1\AppData\Local\Temp\msohtmlclip1\02\clip_image001.png

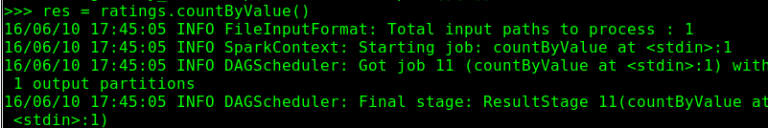
The statement in the screenshot below, loads the data file by creating the RDD through sc.textFile method. The data file is loaded into RDD my\_lines and the textFile property breaks every line of text into a value in the RDD.



In the screenshot below, expression X is passed on to split function. The 3rd column is extracted and new RDD is created with the new results.

C:\Users\KAMAKS~1\AppData\Local\Temp\msohtmlclip1\02\clip_image003.png

In the screenshot below, the statement countByValue() is executed on ratings RDD to calculate the occurrence of each and every rating starting from 1 to 5 and results are stored in a new Python object *res.*

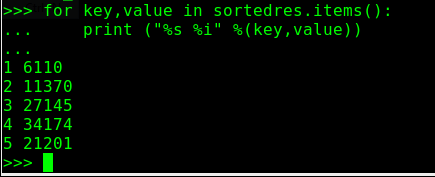


Below is the sample output of the above action applied on ratings RDD.

The code below creates an ordered Dictionary to sort the results based on the key which is rating.

C:\Users\KAMAKS~1\AppData\Local\Temp\msohtmlclip1\02\clip_image005.png

The Python code below iterates through the pair of key and value and prints the total number of occurrence of movies falling under one particular rating category.



*From <*[*https://acadgild.com/blog/spark-use-case-analyzing-movielens-dataset/*](https://acadgild.com/blog/spark-use-case-analyzing-movielens-dataset/)*>*

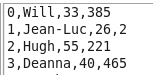
Spark Use Case – Social Media Analysis

Friday, September 23, 2016

3:30 PM

In this post spark, we will work on a case study to calculate the average number of friends based on their age, on a social media website.

Let’s begin by considering a sample of four records.



**Column 1:** User ID

**Column 2:**User Name

**Column 3:** Age of the User

**Column 4:** Number of Friends with that User

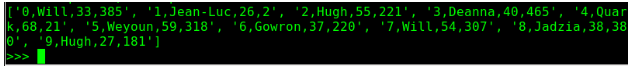
You can download the input file from [here.](https://drive.google.com/file/d/0Bxr27gVaXO5sMXc3aXo2WVJQNE0/view?usp=sharing)

The new RDD, **my\_lines**, is created by calling the textFile function on the Spark Context with our source data, where every individual line of that comma separated source data is passed as individual [entries in the RDD.](https://acadgild.com/big-data/apache-spark-training-certification)



To see the first 10 records of the **my\_lines** RDD, the take action has been called on **my\_lines**RDD.

C:\Users\KAMAKS~1\AppData\Local\Temp\msohtmlclip1\02\clip_image003.png

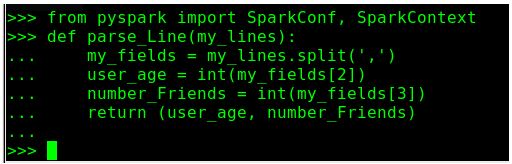


We are going to transform our **my\_lines** RDD into new RDD named as **my\_rdd** by calling map on it and then passing it to the parse\_line function, which could actually perform that mapping.

Hence, every record from **my\_lines** RDD is passed on to parse\_line function one by one and then parsed out.

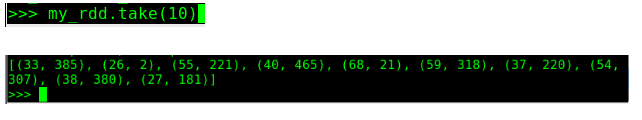
C:\Users\KAMAKS~1\AppData\Local\Temp\msohtmlclip1\02\clip_image005.png

**my\_lines** RDD is first split by comma and then the required fields, i.e. age of the user and number of friends that user is having and those two fields are extracted from third and fourth fields respectively and then returned and stored in the new key-value RDD named as **my\_rdd.**



To see the first 10 records of the RDD named as **my\_rdd**, take action has been called on **my\_rdd**.

The results are the key-value pairs with the age of individual user as key and number of friends for that particular age as value.



We have simplified the below complex script by breaking it into multiple statements to achieve the results.

C:\Users\KAMAKS~1\AppData\Local\Temp\msohtmlclip1\02\clip_image008.png

We need to take the RDD **my\_rdd** containing the key value pairs of the age of individual user as the key, and the number of friends for that particular age as value ,and then call the map values on it.

This transforms every value in key value pair of age and number of friends in from the above RDD.

Every value from RDD is passed on to map function, and the new output comprising of number of friends for a particular user as key and 1 as value.

C:\Users\KAMAKS~1\AppData\Local\Temp\msohtmlclip1\02\clip_image009.png

The first 10 records of the new RDD can be displayed by passing take function on x RDD.

C:\Users\KAMAKS~1\AppData\Local\Temp\msohtmlclip1\02\clip_image010.png

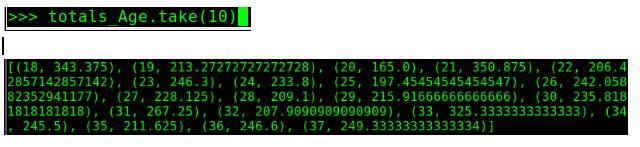
C:\Users\KAMAKS~1\AppData\Local\Temp\msohtmlclip1\02\clip_image011.png

This step involves summing up of the total number of friends for one particular age group as key and the number of users in that age group as value.

This is done by passing reduceByKey transformation on x RDD.

C:\Users\KAMAKS~1\AppData\Local\Temp\msohtmlclip1\02\clip_image012.png

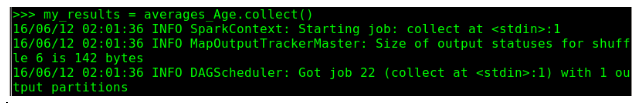
The first 10 records of the new RDD can be displayed by passing take function on totals\_Age RDD.



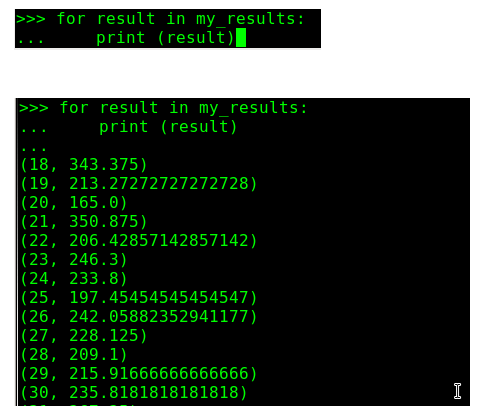
This step includes calculating the average number of friends for every age group by passing a formula in the lambda function to divide the key of previous RDD i.e. total number of friends for one particular age group by value in the previous RDD i.e. number of users in that age group.

C:\Users\KAMAKS~1\AppData\Local\Temp\msohtmlclip1\02\clip_image014.png

The results of averages, \_Age RDD is collected in **my\_results** RDD.



The final results are displayed by using for loop statement in Python to print the age of the user as key and the average number of friends in that age group as value.



*From <*[*https://acadgild.com/blog/spark-use-case-social-media-analysis/*](https://acadgild.com/blog/spark-use-case-social-media-analysis/)*>*

Spark Use Case – The Daily Show

Friday, September 23, 2016

3:45 PM

In this blog we will be  taking a famous Tv show dataset i.e., The Daily show and we will be performing analysis on the guests who came to the show.

Before going ahead we recommend readers to go through our previous blogs on various publicly available datasets.

[Youtube Data Analysis](https://acadgild.com/blog/spark-use-case-youtube-data-analysis/)

[Titanic Data Analysis](https://acadgild.com/blog/spark-use-case-titanic-data-analysis/)

[Olympic Data Analysis](https://acadgild.com/blog/spark-use-case-olympics-data-analysis/)

We have a historical data of the daily show guests from 1999 to 2004. The dataset can be downloaded from [here](https://drive.google.com/open?id=0ByJLBTmJojjzaXlsVWl0WEhPZDQ).

Please find the the dataset description below:

**Dataset Description:**

**YEAR** –  The year the episode aired

**GoogleKnowlege\_Occupation** -Their occupation or office, according to Google’s Knowledge Graph or, if they’re not in there, how Stewart introduced them on the program.

**Show –**Air date of episode. Not unique, as some shows had more than one guest

**Group –**A larger group designation for the occupation. For instance, us senators, us presidents, and former presidents are all under “politicians”

**Raw\_Guest\_List –**The person or list of people who appeared on the show, according to Wikipedia. The GoogleKnowlege\_Occupation only refers to one of them in a given row.

**Problem Statement:**

Find the top 5 kinds of GoogleKnowlege\_Occupation people gusted the show in a particular time period.

**Source Code:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **val** file = sc.textFile("/home/kiran/dialy\_show\_guests")  **val** split = file.map(line => line.split(","))  **val** format = **new** java.text.SimpleDateFormat("MM/dd/yy")  **val** pair = split.map(line => (line(1),format.parse(line(2))))  **val** fil = pair.filter(x => {**if**(x.\_2.after(format.parse("1/11/99")) && x.\_2.before(format.parse("6/11/99"))) **true else false**})  **val** cnt = fil.map(x => (x.\_1,1)).reduceByKey(**\_**+**\_**).map(item => item.swap).sortByKey(**false**).take(5) |

**Walk through of the above code:**

In **line 1**we are creating a new [RDD](https://acadgild.com/big-data/apache-spark-training-certification) by loading the dataset which is in local file system.

In **line 2** we are splitting the records by using the delimiter as ‘,’ since the data is delimited by ‘,’.

In **line 3** we are declaring the date format by using the java library java.text.SimpleDateFormat. In the dataset the data format is “MM/dd/YY”.

In **line 4** we are creating a pair of GoogleKnowlege\_Occupation and Show(date of the show). Here date of the show is taken as a string and we are converting this string to date format using the *parse*method available in java.text.SimpleDateFormat.

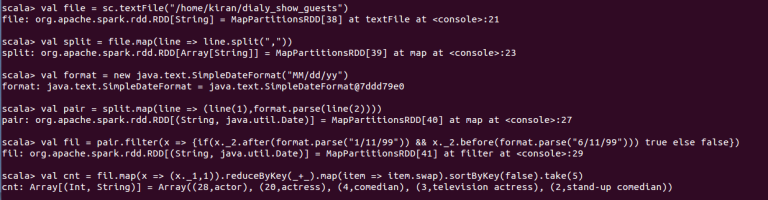
In **line 5**we are using the filter method to filter out the records which doesn’t match our requirement. Here we are giving the range of data explicitly in between we need to count the GoogleKnowlege\_Occupation people gusted. Here we have given the range as 6 months i.e., from 1/11/99 to 6/11/99.

In **line 6**we will get the data which is in specified range from that we are creating a pair of*GoogleKnowlege\_Occupation* and *1*as key value pairs respectively. After that we are performing reduceByKey action on the RDD which will count all the values for each unique key. Then we are swapping the *GoogleKnowlege\_Occupation* and its *count*, and sorting the result by *sortByKey*operation with this we will get the sorted records of *GoogleKnowlege\_Occupation* and its *count*in descending order. Finally, we are taking the top five from the list.

**Output:**

***(28,actor), (20,actress), (4,comedian), (3,television actress), (2,stand-up comedian)***

The same is displayed in the below screen shot.



*From <*[*https://acadgild.com/blog/spark-use-case-daily-show/*](https://acadgild.com/blog/spark-use-case-daily-show/)*>*

Spark Use Case – Olympics Data Analysis

Friday, September 23, 2016

3:46 PM

In this blog, we will discuss on the analysis of Olympics dataset using Apache Spark in Scala.

Olympics data set is a publically available data. Using this dataset, we will evaluate some problem statements such as, finding the number of medals won by each country in swimming, finding the number of medals won by India etc.

**Data Set Description**

The data set consists of the following fields:

**Athlete**: Name of the athlete

**Age**: Age of the athlete

**Country**: The name of the country participating in Olympics

**Year**: The year in which Olympics is conducted

**Closing** **Date**: Closing date of Olympics

**Sport**: Sports name

**Gold Medals**: No. of gold medals

**Silver Medals**: No. of silver medals

**Bronze Medals**: No. of bronze medals

**Total Medals**: Total no. of medals

**Dataset Link**

<https://drive.google.com/drive/folders/0ByJLBTmJojjzVGNsWmpUUUxTZDA>

**Problem Statement 1**

Find the total number of medals won by each country in swimming.

**Source code**

|  |  |
| --- | --- |
| 1  2  3  4  5 | **val** textFile = sc.textFile("hdfs://localhost:9000/olympix\_data.csv")  **val** counts = textFile.filter { x => {**if**(x.toString().split("\t").length >= 10) **true else false**} }.map(line=>{line.toString().split("\t")})  **val** fil = counts.filter(x=>{**if**(x(5).equalsIgnoreCase("swimming")&&(x(9).matches(("\\d+")))) **true else false** })  **val** pairs: RDD[(**String**, **Int**)] = fil.map(x => (x(2),x(9).toInt))  **val** cnt = pairs.reduceByKey(**\_** + **\_**).collect() |

**Description of the Above Code**

**Line1:** We are creating an [RDD](https://acadgild.com/big-data/apache-spark-training-certification) with the existing dataset which is inside HDFS.

**Line 2:**We are taking each record as input and filtering the records which do not have 11 columns. This is useful in eliminating ArrayIndexOutofBound exception.

**Line 3:**We will get the records which have 11 columns and here, we are again filtering the records under the sport ‘swimming’ because we need to find out the number of medals won by countries in swimming. We are also checking whether the 10th column has a digit or not.

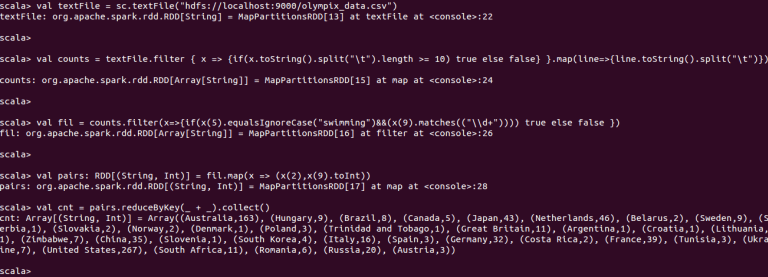
**Line 4:**We are creating a pair RDD (String,Int) where the key is country name and value is the number of medals it won in swimming.

**Line 5:**We are counting the number of medals that each country won in swimming by using the reduceByKey method and finally we are displaying it by using collect() method.

**Output**

***(Australia,163), (Hungary,9), (Brazil,8), (Canada,5), (Japan,43), (Netherlands,46), (Belarus,2), (Sweden,9), (Serbia,1), (Slovakia,2), (Norway,2), (Denmark,1), (Poland,3), (Trinidad and Tobago,1), (Great Britain,11), (Argentina,1), (Croatia,1), (Lithuania,1), (Zimbabwe,7), (China,35), (Slovenia,1), (South Korea,4), (Italy,16), (Spain,3), (Germany,32), (Costa Rica,2), (France,39), (Tunisia,3), (Ukraine,7), (United States,267), (South Africa,11), (Romania,6), (Russia,20), (Austria,3)***

You can see the same in the below screen shot.



*Figure 1*

**Problem Statement 2**

Find the number of medals that India won year wise.

**Source code**

|  |  |
| --- | --- |
| 1  2  3  4  5 | **val** textFile = sc.textFile("hdfs://localhost:9000/olympix\_data.csv")  **val** counts = textFile.filter { x => {**if**(x.toString().split("\t").length >= 10) **true else false**} }.map(line=>{line.toString().split("\t")})  **val** fil = counts.filter(x=>{**if**(x(2).equalsIgnoreCase("india")&&(x(9).matches(("\\d+")))) **true else false** })  **val** pairs: RDD[(**String**, **Int**)] = fil.map(x => (x(3),x(9).toInt))  **val** cnt = pairs.reduceByKey(**\_** + **\_**).collect() |

**Description of the Above Code**

**Line1:** We are creating an RDD with the existing dataset which is inside HDFS.

**Line 2:**We are taking each record as input and filtering the records which do not have 11 columns. This is useful in eliminating ArrayIndexOutofBound exception.

**Line 3:** We will get the records which have 11 columns and here, we are again filtering the records for the country ‘India’ as we need to find out the number of medals won by India. Also, we are checking whether the 10th column has a digit or not.

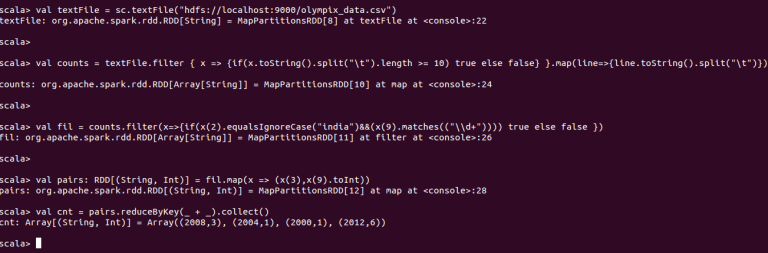
**Line 4:**We are creating a pair RDD(String,Int) where key is the year and value is the number of medals won in that year.

**Line 5:**We are counting the number of medals won by India, year wise by using the reduceByKey method and finally we are displaying it by using collect() method.

**Output**

***(2008,3), (2004,1), (2000,1), (2012,6)***

You can see the same in the below screen shot.



*Figure 2*

**Problem Statement 3**

Find the total number of medals won by each country.

**Source Code**

|  |  |
| --- | --- |
| 1  2  3  4  5 | **val** textFile = sc.textFile("hdfs://localhost:9000/olympix\_data.csv")  **val** counts = textFile.filter { x => {**if**(x.toString().split("\t").length >= 10) **true else false**} }.map(line=>{line.toString().split("\t")})  **val** fil = counts.filter(x=>{**if**((x(9).matches(("\\d+")))) **true else false** })  **val** pairs: RDD[(**String**, **Int**)] = fil.map(x => (x(2),x(9).toInt))  **val** cnt = pairs.reduceByKey(**\_** + **\_**).collect() |

**Description of the Above Code**

**Line1:** We are creating an RDD with the existing dataset which is inside HDFS.

**Line 2:**We are taking each record as input and filtering the lines which do not have 11 columns. This is useful in eliminating ArrayIndexOutofBound exception.

**Line 3:**We will get the records which have 11 columns and we are again filtering the records which has a digit in 10th column.

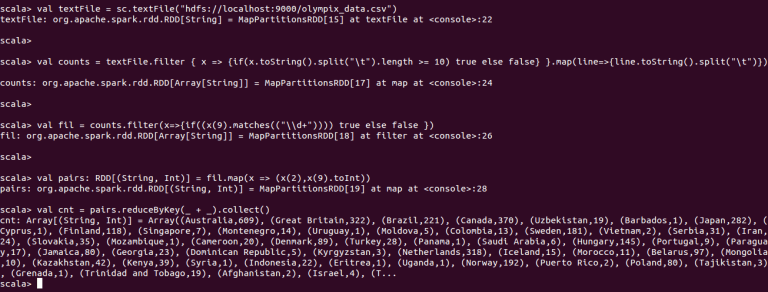
**Line 4:**We are creating a pair RDD(String,Int) where key is the country and value is the number of medals won by the country.

**Line 5:**We are counting the number of medals won by each country by using the reduceByKeymethod and finally we are displaying it by using collect() method.

**Output**

***(Australia,609), (Great Britain,322), (Brazil,221), (Canada,370), (Uzbekistan,19), (Barbados,1), (Japan,282), (Cyprus,1), (Finland,118), (Singapore,7), (Montenegro,14), (Uruguay,1), (Moldova,5), (Colombia,13), (Sweden,181), (Vietnam,2), (Serbia,31), (Iran,24), (Slovakia,35), (Mozambique,1), (Cameroon,20), (Denmark,89), (Turkey,28), (Panama,1), (Saudi Arabia,6), (Hungary,145), (Portugal,9), (Paraguay,17), (Jamaica,80), (Georgia,23), (Dominican Republic,5), (Kyrgyzstan,3), (Netherlands,318), (Iceland,15), (Morocco,11), (Belarus,97), (Mongolia,10), (Kazakhstan,42), (Kenya,39), (Syria,1), (Indonesia,22), (Eritrea,1), (Uganda,1), (Norway,192), (Puerto Rico,2), (Poland,80), (Tajikistan,3), (Grenada,1), (Trinidad and Tobago,19), (Afghanistan,2), (Israel,4)***

You can see the same in the below screen shot.

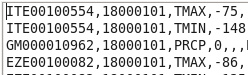


*Figure 3*

*From <*[*https://acadgild.com/blog/spark-use-case-olympics-data-analysis/*](https://acadgild.com/blog/spark-use-case-olympics-data-analysis/)*>*

In this post of spark, we will work on a case study to find the minimum temperature observed in a given weather station in a particular year.

Let’s begin by considering a sample of four records.



**Data Definition:**

Column 1: Weather Station

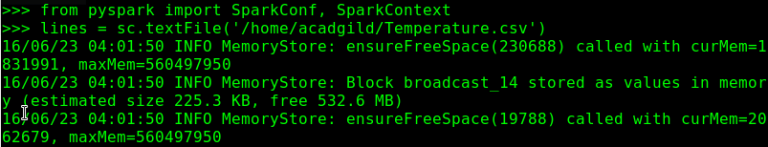
Column 2: Date(year/Month/Day)

Column 3: Observation Type

Column 4: Temperature

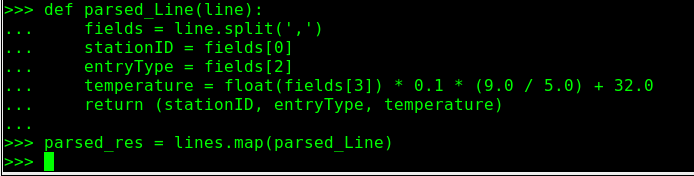
You can download the input file from [here](https://drive.google.com/file/d/0Bxr27gVaXO5sTVExTUp5NE9TRXM/view?usp=sharing)[.](https://drive.google.com/file/d/0Bxr27gVaXO5sMXc3aXo2WVJQNE0/view?usp=sharing)

The new RDD,lines, is created by calling the textFile function on the [Spark Context](https://acadgild.com/big-data/apache-spark-training-certification) with our source data, where every individual line of that comma separated source data is passed as individual entries in the RDD.



We are going to transform our lines [RDD into new](https://acadgild.com/big-data/apache-spark-training-certification) RDD named as parsed\_res by calling map on it and then passing it to the parse\_Line function, which could actually perform that mapping.

Hence, every record from lines RDD is passed on to parse\_Line function one by one and then parsed out.



Our raw weather data includes information like minimum temperature,maximum temperature and amount of precipitation but since we need to find out the minimum temperature in a weather station so we discard other information like maximum temperature and precipitation details and only pass out the records with minimum temperature details.

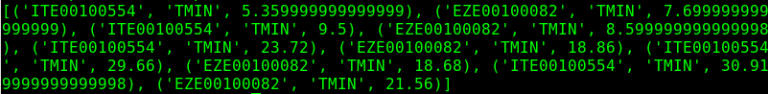
C:\Users\KAMAKS~1\AppData\Local\Temp\msohtmlclip1\02\clip_image004.png

In the below step we strip out the second column i.e the observation type and extracts out just weather station ID and minimum temperature.

To see the first 10 records of the temp\_station RDD,  take action has been called on my\_rdd.

The results are the key-value pairs with the weather station ID as key and minimum temperature in that weather station as value.

C:\Users\KAMAKS~1\AppData\Local\Temp\msohtmlclip1\02\clip_image005.png



In the below step reduceByKey action is called which is going to aggregate every minimum temperature observed  for every weather station ID and lambda function determines that how we do that aggregation.

Two observations for minimum temperature for a given weather station ID is combined and the Min function takes the minimum temperature between two observations  for a given weather station and this process continues as more and more observations are fed in and in the last only minimum temperature for a weather station ID survives.

C:\Users\KAMAKS~1\AppData\Local\Temp\msohtmlclip1\02\clip_image007.png

To display the first 10 records of the temp\_min RDD, take action has been called.

The results are the key-value pairs with the weather station ID as key and minimum temperature observed in that weather station throughout the year as value.

C:\Users\KAMAKS~1\AppData\Local\Temp\msohtmlclip1\02\clip_image008.png

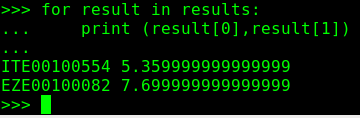
C:\Users\KAMAKS~1\AppData\Local\Temp\msohtmlclip1\02\clip_image009.png

The results of averages, \_Age RDD is collected in my\_results which is a python list.

C:\Users\KAMAKS~1\AppData\Local\Temp\msohtmlclip1\02\clip_image010.png

C:\Users\KAMAKS~1\AppData\Local\Temp\msohtmlclip1\02\clip_image011.png

The final results are displayed by using for loop statement in Python to print the age of the weather station  as key and the minimum temperature for that year as value.



We hope this post has been helpful in understanding this Spark use case using Python. In case of any queries, feel free to comment below and we will get back to you at the earliest.

*From <*[*https://acadgild.com/blog/spark-use-case-weather-data-analysis/*](https://acadgild.com/blog/spark-use-case-weather-data-analysis/)*>*

Spark Use Case – Uber Data Analysis

Friday, September 23, 2016

4:14 PM

In this post, we will be performing analysis on the Uber dataset  in Apache spark using Scala.

The Uber dataset consists of 4 columns. They are dispatching\_base\_number, date, active\_vehicles and trips. You can download the dataset from the below link:

<https://drive.google.com/open?id=0ByJLBTmJojjzS2c2UktqLW5uRG8>

**Problem Statement:**

Find the days on which each basement has more trips.

**Source Code:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | **val** dataset = sc.textFile("/home/kiran/Desktop/uber")  **val** header = dataset.first()  **val** format = **new** java.text.SimpleDateFormat("MM/dd/yyyy")  **var** days =**Array**("Sun","Mon","Tue","Wed","Thu","Fri","Sat")  **val** eliminate = dataset.filter(line => line != header)  **val** split = eliminate.map(line => line.split(",")).map { x => (x(0),format.parse(x(1)),x(3)) }  **val** combine = split.map(x => (x.\_1+" "+days(x.\_2.getDay),x.\_3.toInt))  **val** arrange = combine.reduceByKey(**\_**+**\_**).map(item => item.swap).sortByKey(**false**).collect.**foreach**(println) |

*Here’s the explanation of the above code:*

* In **line 1,** we are loading the dataset in our local system, using the textFile method.
* In **line 2,** we are creating a variable **header,**which holds the first line of the dataset (In this dataset the first line is header line).
* In **line 3,** we are declaring the date format using SimpleDateFormat in Java. Here the date is in the format of MM/dd/YYYY.
* In **line 4,**we are declaring an array, which will hold the days of the week from Sunday to Saturday.
* In **line 5,**we are filtering the header line from the dataset using the filter [RDD](https://acadgild.com/big-data/apache-spark-training-certification).

* In **line 6,**we are splitting the dataset using the delimiter as ***coma and*** taking out the three columns; *dispatching\_base\_number,*which is in the 1st column, datewhich is in the second column and the *number of* *trips,*which is in the fourth column. While extracting the columns, we are parsing the date, which is in string format to date format.
* After this step, we will get the records as *B02512, Thu Jan 01 00:00:00 IST 2015, 1132.*
* In **line 7,**we are adding the two columns, *dispatching\_base\_number*and *formatted date.*To get the *day*from the formatted date, we need to use the **getDay**method of java.util.Date package. Here, we will get the day number of the week and pass the day number into the array consisting of the names of the days. Finally, we will get the combination of *dispatching\_base\_number*and*day* of the week and the number of weeks.These are like keys and values.
* In **line 8,**we are using the reduceByKey RDD to combine all the values for each unique key, where key is the combination of *dispatching\_base\_number*and *day* of the week. After this, we are swapping the keys and values and then perform sortByKey action on the RDD, which will sort the records by values in the descending order. Finally, we are printing the result using the**collect**action.

**Output:**

*(356789,B02764 Sat)*

*(326968,B02764 Fri)*

*(304200,B02764 Thu)*

*(249896,B02764 Sun)*

*(241137,B02764 Wed)*

*(221343,B02764 Tue)*

*(214116,B02764 Mon)*

*(127902,B02617 Sat)*

*(125067,B02617 Fri)*

*(120283,B02682 Sat)*

*(118254,B02617 Thu)*

*(114662,B02682 Fri)*

*(106643,B02682 Thu)*

*(94887,B02617 Wed)*

*(94588,B02598 Sat)*

*(93126,B02598 Fri)*

*(91722,B02617 Sun)*

*(90333,B02598 Thu)*

*(86602,B02617 Tue)*

*(86252,B02682 Wed)*

*(82825,B02682 Sun)*

*(80591,B02617 Mon)*

*(76905,B02682 Tue)*

*(74939,B02682 Mon)*

*(71956,B02598 Wed)*

*(66477,B02598 Sun)*

*(63429,B02598 Tue)*

*(60882,B02598 Mon)*

*(36737,B02765 Sat)*

*(34934,B02765 Fri)*

*(30408,B02765 Thu)*

*(24340,B02765 Wed)*

*(22741,B02765 Tue)*

*(22536,B02765 Sun)*

*(21974,B02765 Mon)*

*(16435,B02512 Fri)*

*(15809,B02512 Thu)*

*(15026,B02512 Sat)*

*(12691,B02512 Wed)*

*(12041,B02512 Tue)*

*(11297,B02512 Mon)*

*(10487,B02512 Sun)*

The same is as displayed in the below screenshot.

*From <*[*https://acadgild.com/blog/spark-use-case-uber-data-analysis/*](https://acadgild.com/blog/spark-use-case-uber-data-analysis/)*>*

Spark Use Case – Travel Data Analysis

Friday, September 23, 2016

4:19 PM

In this blog, we will discuss on the analysis of travel dataset and gain insights from the dataset using [Apache Spark.](https://acadgild.com/big-data/apache-spark-training-certification)

The travel dataset is publically available and the contents are detailed under the heading, ‘Travel Sector Dataset Description’.

Based on the data, we will find the top 20 destination people travel the most, top 20 locations from where people travel the most, top 20 cities that generate high airline revenues for travel, based on booked trip count.

**Travel Sector Dataset Description**

Column 1: City pair (Combination of *from* and *to*): String

Column 2: From location: String

Column 3: To Location: String

Column 4: Product type: Integer (1=Air, 2=Car, 3 =Air+Car, 4 =Hotel, 5=Air+Hotel, 6=Hotel +Car, 7 =Air+Hotel+Car)

Column 5: Adults traveling: Integer

Column 6: Seniors traveling: Integer

Column 7: Children traveling: Integer

Column 8: Youth traveling: Integer

Column 9: Infant traveling: Integer

Column 10: Date of travel: String

Column 11: Time of travel: String

Column 12: Date of Return: String

Column 13: Time of Return: String

Column 14: Price of booking: Float

Column 15: Hotel name: String

You can download the dataset from the link below:

<https://drive.google.com/open?id=0ByJLBTmJojjzZEg2bXpYa0dyd1k>

**Problem Statement 1**

Top 20 destination people travel the most: Based on the given data, we can find the most popular destination that people travel frequently. There are many destinations out of which we will find only first 20, based on trips booked for particular destinations.

**Source Code**

|  |  |
| --- | --- |
| 1  2  3 | **val** textFile = sc.textFile("hdfs://localhost:9000/TravelData.txt")    **val** split = textFile.map(lines**=>**lines.split('\t')).map(x=>(x(2),1)).reduceByKey(**\_**+**\_**).map(item => item.swap).sortByKey(**false**).take(20) |

**Description of the above code**

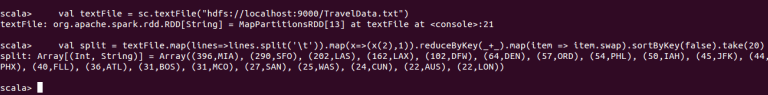
**Line 1:**We are creating an [RDD](https://acadgild.com/big-data/apache-spark-training-certification)by loading a new dataset which is in HDFS.

**Line 2:**We have split each record by taking the delimiter as *tab*because the data is tab separated. We are creating the key-value pair, where key is the *destination* that is in 3rd column and the value is 1. Since we need to count the cities which are popular, we are using the reduceByKey method to count them. After counting the destinations, we are swapping the key-value pairs. The sortByKey method sorts the data with keys and *false*stands for descending order. Once the sorting is complete, we are considering the top 20 destinations.

**Output**

***(396,MIA), (290,SFO), (202,LAS), (162,LAX), (102,DFW), (64,DEN), (57,ORD), (54,PHL), (50,IAH), (45,JFK), (44,PHX), (40,FLL), (36,ATL), (31,BOS), (31,MCO), (27,SAN), (25,WAS), (24,CUN), (22,AUS), (22,LON)***

You can see the same in the below screen shot.



**Problem Statement 2**

Top 20 locations from where people travel the most: We can find the places from where most of the trips are undertaken, based on the booked trip count.

**Source Code**

|  |  |
| --- | --- |
| 1  2  3 | **val** textFile = sc.textFile("hdfs://localhost:9000/TravelData.txt")    **val** split = textFile.map(lines**=>**lines.split('\t')).map(x=>(x(1),1)).reduceByKey(**\_**+**\_**).map(item => item.swap).sortByKey(**false**).take(20) |

**Description of the above code**

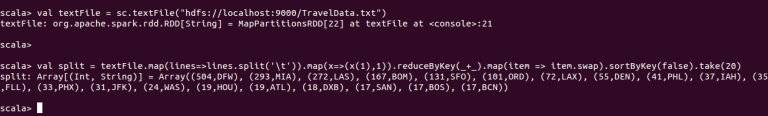
**Line 1:**We are creating an RDD by loading a new dataset which is in HDFS.

**Line 2:**We have split each record by taking the delimiter as *tab*since the data is tab separated. We are creating the key-value pair, where key is the *location from where people start,* that is in the 2nd column and the value is 1. Since we need to count the cities which are popular locations from where people undertake the trips, we are using the reduceByKey method to count them. After counting the locations, we are swapping the key-value pairs. We are using the sortByKey method which sorts the data with keys where *false*stands for descending order. Once the sorting is complete, we are taking the top 20 locations from where people undertake the trips.

**Output**

***(504,DFW), (293,MIA), (272,LAS), (167,BOM), (131,SFO), (101,ORD), (72,LAX), (55,DEN), (41,PHL), (37,IAH), (35,FLL), (33,PHX), (31,JFK), (24,WAS), (19,HOU), (19,ATL), (18,DXB), (17,SAN), (17,BOS), (17,BCN)***

You can see the same in the below screen shot.



**Problem Statement 3**

Top 20 cities that generate high airline revenues for travel, so that the site can concentrate on offering discount on booking, to those cities to attract more bookings.

**Source Code**

|  |  |  |
| --- | --- | --- |
| 1  2  3  4  5 | **val** textFile = sc.textFile("hdfs://localhost:9000/TravelData.txt")    **val** fil = textFile.map(x**=>**x.split('\t')).filter(x=>{**if**((x(3).matches(("1")))) **true else false** })    **val** cnt = fil.map(x=>(x(2),1)).reduceByKey(**\_**+**\_**).map(item => item.swap).sortByKey(**false**).take(20) |  |

**Description of the above code**

**Line 1:**We are creating an RDD by loading a new dataset which is in HDFS.

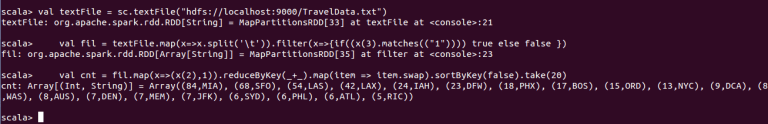
**Line 2:**We are splitting each record based on the delimiter tab as the data is tab separated. From this, we are filtering the records based on the mode of travel. Here, we need the count of people who travelled by flight which is denoted by **1**(1=Air, 2=Car, 3 =Air+Car, 4 =Hotel, 5=Air+Hotel, 6=Hotel +Car, 7 =Air+Hotel+Car).

**Line 3:**We are creating the key-value pairs for those people who traveled by air, where key is the *destination*which is in 3rd column and value is 1. Since we need to count the popular cities, we are counting them by using the reduceByKey method. After counting the destinations, we are swapping the key-value pairs. We are using the sortByKey method to sort the data with keys where *false*stands for descending order. Once sorting is completed, we are considering top 20 cities that generate high airline revenues for travel.

**Output:**

***(84,MIA), (68,SFO), (54,LAS), (42,LAX), (24,IAH), (23,DFW), (18,PHX), (17,BOS), (15,ORD), (13,NYC), (9,DCA), (8,WAS), (8,AUS), (7,DEN), (7,MEM), (7,JFK), (6,SYD), (6,PHL), (6,ATL), (5,RIC)***

You can see the same in the below screen shot.



*From <*[*https://acadgild.com/blog/spark-use-case-travel-data-analysis/*](https://acadgild.com/blog/spark-use-case-travel-data-analysis/)*>*