Monash University

FIT5202 - Data processing for Big Data

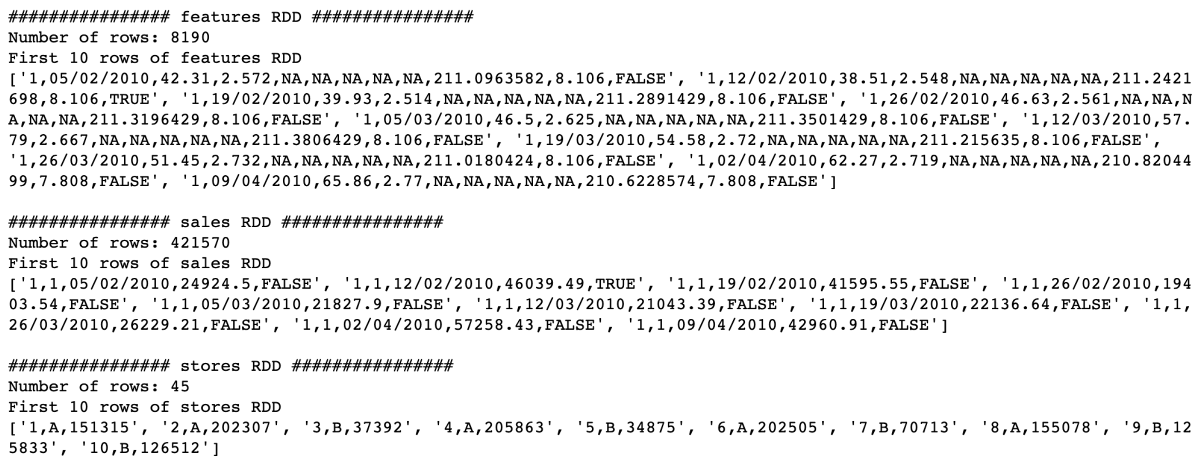
Assignment 1: Analysing Retail Data

Name : Aparna Suresh
ID : 32263546
username : asur0015@student.monash.edu

Part 1: Working with RDDs (30%)

1.1 Data Preparation and Loading (5%)

Firstly, the required libraries are loaded. SparkConf class is loaded along with the SparkContext and SparkSession classes from pyspark into the program. Spark applications applications run on a cluster through SparkContext which can connect to other clusters. SparkSession on the other hand is an object that is used to create data frames. Thus, loading all these classes in to program is vital to execute the code. The second step is to load the data (Features, sales, and stores) using SparkContext. Since we have been asked to remove the header for all the three data files, filter() is used to remove the header rows in all the three datasets and the first 10 records have been printed using take(n) function which will print the required n records. The output looks like the following image:



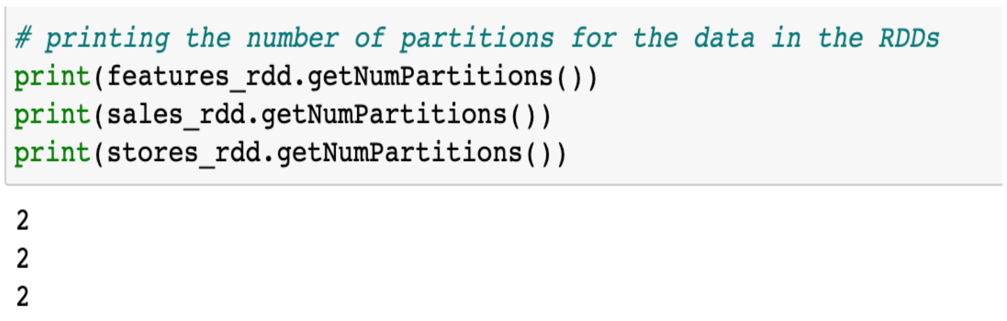
```
################ features RDD ################
Number of rows: 8190
First 10 rows of features RDD
['1,05/02/2010,42.31,2.572,NA,NA,NA,NA,NA,211.0963582,8.106,FALSE', '1,12/02/2010,38.51,2.548,NA,NA,NA,NA,NA,211.2421
698,8.106,TRUE', '1,19/02/2010,39.93,2.514,NA,NA,NA,NA,NA,211.2891429,8.106,FALSE', '1,26/02/2010,46.63,2.561,NA,NA,N
A,NA,NA,211.3196429,8.106,FALSE', '1,05/03/2010,46.5,2.625,NA,NA,NA,NA,NA,211.3501429,8.106,FALSE', '1,12/03/2010,57.
79,2.667,NA,NA,NA,NA,NA,211.3806429,8.106,FALSE', '1,19/03/2010,54.58,2.72,NA,NA,NA,NA,NA,211.215635,8.106,FALSE',
'1,26/03/2010,51.45,2.732,NA,NA,NA,NA,NA,211.0180424,8.106,FALSE', '1,02/04/2010,62.27,2.719,NA,NA,NA,NA,NA,210.82044
99,7.808,FALSE', '1,09/04/2010,65.86,2.77,NA,NA,NA,NA,NA,210.6228574,7.808,FALSE']

################ sales RDD ################
Number of rows: 421570
First 10 rows of sales RDD
['1,1,05/02/2010,24924.5,FALSE', '1,1,12/02/2010,46039.49,TRUE', '1,1,19/02/2010,41595.55,FALSE', '1,1,26/02/2010,194
03.54,FALSE', '1,1,05/03/2010,21827.9,FALSE', '1,1,12/03/2010,21043.39,FALSE', '1,1,19/03/2010,22136.64,FALSE', '1,1,
26/03/2010,26229.21,FALSE', '1,1,02/04/2010,57258.43,FALSE', '1,1,09/04/2010,42960.91,FALSE']

################ stores RDD ################
Number of rows: 45
First 10 rows of stores RDD
['1,A,151315', '2,A,202307', '3,B,37392', '4,A,205863', '5,B,34875', '6,A,202505', '7,B,70713', '8,A,155078', '9,B,12
5833', '10,B,126512']
```

1.2 Data Partitioning in RDD (15%)

1.2.1 The spark is set to run in the local mode using the local[*]. Local[*] helps spark run locally with as many processors in your machine. When you assign the Local[*] and use the .getNumPartitions() function, the default number of partitions obtained is 2. If we mention only 1 core in the local[ ] for processing, then only 1 partition is created. By default, the number of partitions is 2 if nothing is mentioned. This default condition is possible only when the system has more than 1 core. Here, the output number of partitions is 2 for the given data in the RDDs because we did not mention how many partitions we want. So, by default, 2 partitions are made. The following image shows the number of partitions made:



```
# printing the number of partitions for the data in the RDDs
print(features_rdd.getNumPartitions())
print(sales_rdd.getNumPartitions())
print(stores_rdd.getNumPartitions())
```

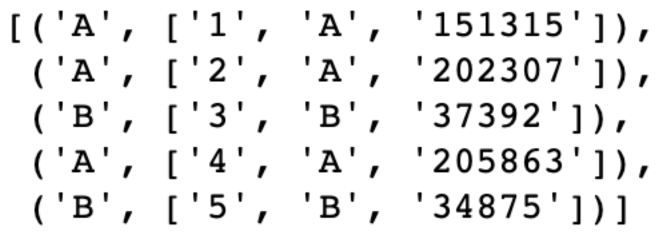
```
2
2
2
```

1.2.2 Here, the key value is created by splitting the second element which is the store type (A,B,C) and assigning it as the key element. This is achieved by the split() and map(). The data is in the format

Store, Type, Size

1, A, 151315

From this, the first element (counting from 0) is the type. We are splitting in the first element which is the type and making it as the key using the map(). Thus, the key value is the store type as shown below:



```
[('A', ['1', 'A', '151315']),
 ('A', ['2', 'A', '202307']),
 ('B', ['3', 'B', '37392']),
 ('A', ['4', 'A', '205863']),
 ('B', ['5', 'B', '34875'])]
```

The output is in the order [(store type(KEY), [store ID, type, size]),.]

1.2.3 We are using if conditional to check if the key is A, B, or C. based on that we are returning 0,1 or 2. Then we are partitioning the data based on the keys. Thus by using RDD, all the data with A as its key will come under one partition, all the data having B as its key will come under another partition and respectively with C. Thus, we have made three partitions with A, B, and C as their corresponding key elements. After partitioning, we are printing the number of records in each partition and the data within each partition. The output looks as follows:

```
Number of partitions: 3
Number of records in each partition: [22, 17, 6]
[[('A', ['1', 'A', '151315']), ('A', ['2', 'A', '202307']), ('A', ['4', 'A', '205863']), ('A', ['6', 'A', '202505']),
('A', ['8', 'A', '155078']), ('A', ['11', 'A', '207499']), ('A', ['13', 'A', '219622']), ('A', ['14', 'A', '20089
8']), ('A', ['19', 'A', '203819']), ('A', ['20', 'A', '203742']), ('A', ['24', 'A', '203819']), ('A', ['26', 'A', '15
2513']), ('A', ['27', 'A', '204184']), ('A', ['28', 'A', '206302']), ('A', ['31', 'A', '203750']), ('A', ['32', 'A',
'203007']), ('A', ['33', 'A', '39690']), ('A', ['34', 'A', '158114']), ('A', ['36', 'A', '39910']), ('A', ['39', 'A',
'184109']), ('A', ['40', 'A', '155083']), ('A', ['41', 'A', '196321'])], [('B', ['3', 'B', '37392']), ('B', ['5',
'B', '34875']), ('B', ['7', 'B', '70713']), ('B', ['9', 'B', '125833']), ('B', ['10', 'B', '126512']), ('B', ['12',
'B', '112238']), ('B', ['15', 'B', '123737']), ('B', ['16', 'B', '57197']), ('B', ['17', 'B', '93188']), ('B', ['18',
'B', '120653']), ('B', ['21', 'B', '140167']), ('B', ['22', 'B', '119557']), ('B', ['23', 'B', '114533']), ('B', ['2
5', 'B', '128107']), ('B', ['29', 'B', '93638']), ('B', ['35', 'B', '103681']), ('B', ['45', 'B', '118221'])], [('C',
['30', 'C', '42988']), ('C', ['37', 'C', '39910']), ('C', ['38', 'C', '39690']), ('C', ['42', 'C', '39690']), ('C',
['43', 'C', '41062']), ('C', ['44', 'C', '39910'])]]
```

1.3 Query/Analysis (10%)

1.3.1 To calculate the average weekly sales for each year, the RDD functions split() and mapValues() are used. We are first defining the key values pair using the year of the date and the weekly sales in the sales RDD data. Then using groupby() on the year , the average sales is calculated for each year. The average of each store type in years 2010, 2011 and 2012 is obtained and shown below as follows:

```
[('2010', 16270.275737033313),
 ('2011', 15954.070675386392),
 ('2012', 15694.948597357718)]
```

1.3.2 To find the highest temperature recorded in 2011 in type B store, we are joining the features RDD and the stores RDD based on the store number (1,2,3,.) and filtering where the type is B and the year is 2011 and getting the maximum temperature. The store ID, the date, the highest temperature and the store type is printed as shown below.

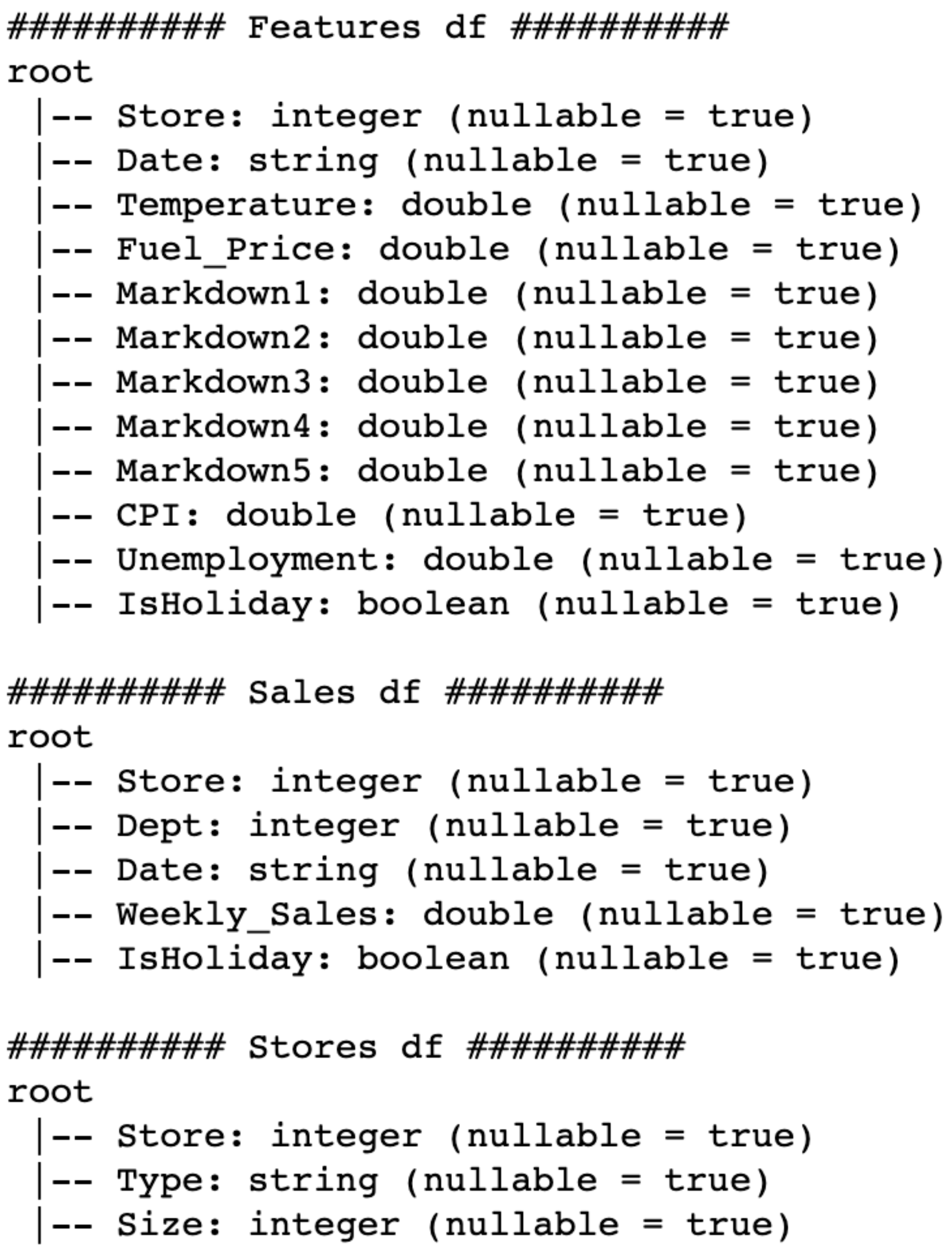
```
('10', ['01/07/2011', '95.36', 'B'])
```

Here, store 10 of type B marked the highest temperature of 95.36 in the year 2011.

Part2. Working with DataFrames (50%)

2.1 Data Preparation and Loading (5%)

The required libraries are imported and a structured type (Schema) is made for all the three data based on the metadata given. A Schema will help us execute all the DF codes efficiently with optimum time. Once the schema is made, the data is read and saved as a dataframe. The schema for all the three DFs looks like the following image when printed:



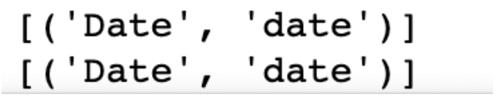
```
########## Features df ##########
root
 |-- Store: integer (nullable = true)
 |-- Date: string (nullable = true)
 |-- Temperature: double (nullable = true)
 |-- Fuel_Price: double (nullable = true)
 |-- Markdown1: double (nullable = true)
 |-- Markdown2: double (nullable = true)
 |-- Markdown3: double (nullable = true)
 |-- Markdown4: double (nullable = true)
 |-- Markdown5: double (nullable = true)
 |-- CPI: double (nullable = true)
 |-- Unemployment: double (nullable = true)
 |-- IsHoliday: boolean (nullable = true)

########## Sales df ##########
root
 |-- Store: integer (nullable = true)
 |-- Dept: integer (nullable = true)
 |-- Date: string (nullable = true)
 |-- Weekly_Sales: double (nullable = true)
 |-- IsHoliday: boolean (nullable = true)

########## Stores df ##########
root
 |-- Store: integer (nullable = true)
 |-- Type: string (nullable = true)
 |-- Size: integer (nullable = true)
```

2.2 Query/Analysis (45%)

2.2.1 To transform the Date column in both the features and sales dataframe from string to date type, to_date() is used with the format “dd/MM/yyyy”. On using the function, the datatype of the “Date” column is changed to “date” as shown below:

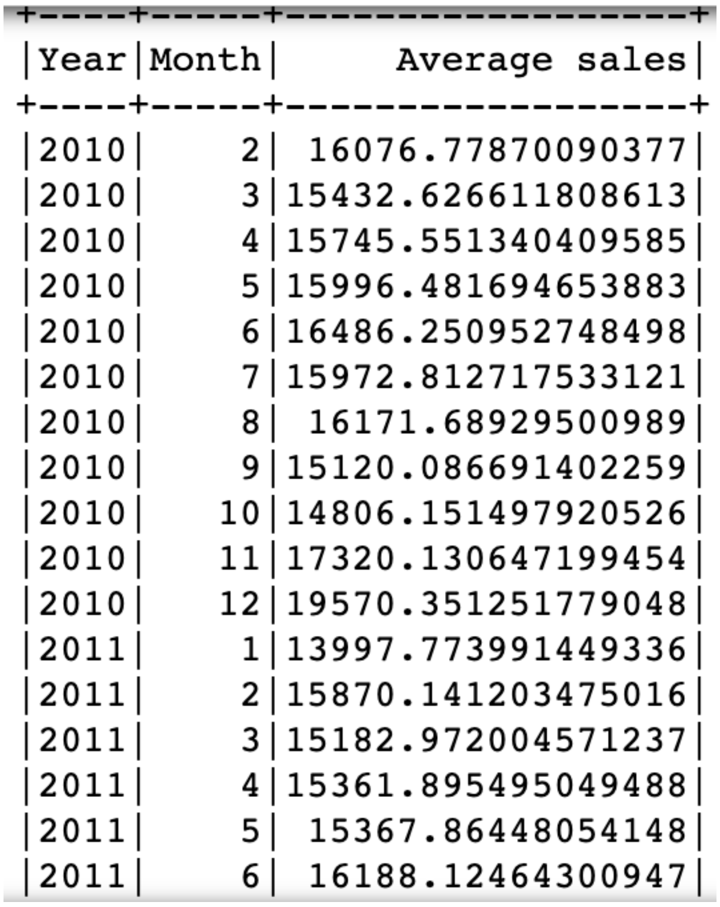


```
[('Date', 'date')]
[('Date', 'date')]
```

2.2.2 To calculate the average weekly sales for holiday week and non-holiday week, the column “IsHoliday” is grouped based on which the average of the weekly sales is aggregated and printed. orderBy() is used to show the average sales in descending order. **False** indicates that **it is not** a special holiday week and **True** indicates that **it is** a special holiday week. Thus, the output for average weekly sales looks like:

```
+---------+------------------+
|IsHoliday|     Average sales|
+---------+------------------+
|    false|15901.445069008767|
|     true| 17035.82318735039|
+---------+------------------+
```

2.2.3 To calculate the average weekly sales based on year and month, the data is grouped by year and month and the average of the weekly sales is aggregated and printed on the order of month, year.



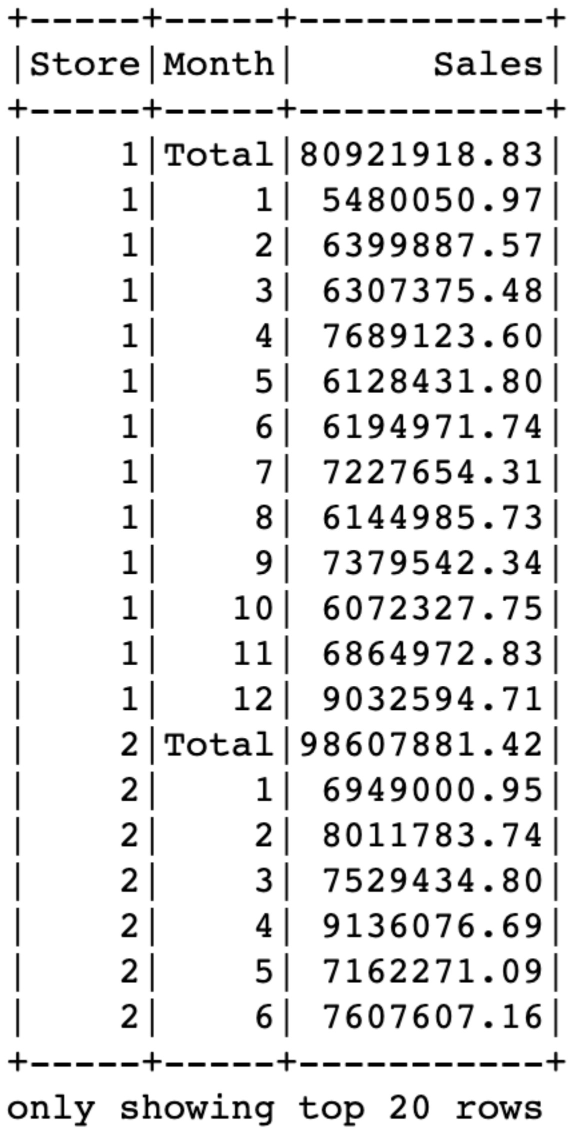
```
+----+-----+------------------+
|Year|Month|     Average sales|
+----+-----+------------------+
|2010|    2| 16076.77870090377|
|2010|    3|15432.626611808613|
|2010|    4|15745.551340409585|
|2010|    5|15996.481694653883|
|2010|    6|16486.250952748498|
|2010|    7|15972.812717533121|
|2010|    8| 16171.68929500989|
|2010|    9|15120.086691402259|
|2010|   10|14806.151497920526|
|2010|   11|17320.130647199454|
|2010|   12|19570.351251779048|
|2011|    1|13997.773991449336|
|2011|    2|15870.141203475016|
|2011|    3|15182.972004571237|
|2011|    4|15361.895495049488|
|2011|    5| 15367.86448054148|
|2011|    6| 16188.12464300947|
```

2.2.4 To calculate the average MarkDown1 value in holiday week for all the type C store, firstly we inner join both the features and store dataframe based on the Store ID (1,2,3,.). The second step is to filter the store type == "C" and IsHoliday == "True". Then the average MarkDown1 value is printed based on the filter applied above

```
[Row(avg(Markdown1)=778.2502439024388)]
```

2.2.5 To show all the stores total sales, based on each month and each store's yearly total for the year 2011, first we filter with year 2011 and then group by on the basis of Store ID

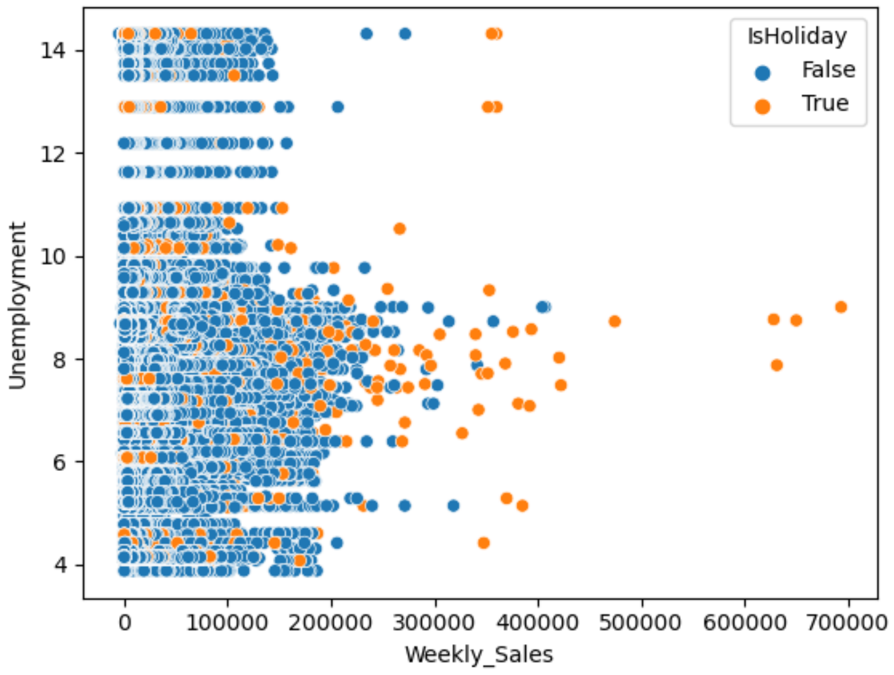
(1,2,3,.), and the month. Then based on the monthly result, groupby for each store in the year 2011 is created to get the total sales of the year. Both of these results are then appended, sorted and then rounded upto 2 decimal places.



```
+-----+-----+-----------+
|Store|Month|      Sales|
+-----+-----+-----------+
|    1|Total|80921918.83|
|    1|    1| 5480050.97|
|    1|    2| 6399887.57|
|    1|    3| 6307375.48|
|    1|    4| 7689123.60|
|    1|    5| 6128431.80|
|    1|    6| 6194971.74|
|    1|    7| 7227654.31|
|    1|    8| 6144985.73|
|    1|    9| 7379542.34|
|    1|   10| 6072327.75|
|    1|   11| 6864972.83|
|    1|   12| 9032594.71|
|    2|Total|98607881.42|
|    2|    1| 6949000.95|
|    2|    2| 8011783.74|
|    2|    3| 7529434.80|
|    2|    4| 9136076.69|
|    2|    5| 7162271.09|
|    2|    6| 7607607.16|
+-----+-----+-----------+
only showing top 20 rows
```

The **Total** rows show the yearly sales and the store ID (1,2,3,.) The **month** column represents the months January to December in number format with their corresponding monthly sales and store ID .

2.2.6 Relationship between weekly sales and unemployment based on the special holiday weeks



The scatterplot shows the relationship between the weekly sales and the unemployment rate. From the graph we can infer that whenever the unemployment rate is less than 10%, the weekly sales are higher even when the IsHoliday is False. But when the unemployment is greater than 10%, the weekly sales are less in comparison when the unemployment is less then 10%. When IsHoliday is True and the unemployment is less than 10%, the weekly sales are high in comparison when the unemployment rate is more than 10%.

Part3: RDDs vs DataFrame vs Spark SQL (20%)

Why are dataframes faster than RDDs?

Dataframes are faster than RDDs in Spark. Dataframes support processing of large structured data (Schema). Dataframes are better than RDD in terms of memory management, optimized execution, and improves the scalability and performance of Spark. Although RDDs share some of the features, memory management and efficient execution is better in terms of Dataframes in Spark. That is the reason why DataFrame is faster than RDD.

Query: Calculate the average weekly fuel price for all stores' size larger than 150000.

To implement the queries through Spark-SQL, temporary views are created firstly for all the data. Then using the “%%time” built-in command, the total processing time along with the average weekly fuel price for all stores with size greater than 150000 are calculated in RDD, dataframes and Spark-SQL .

Although the average weekly fuel price is same for all the three methods, there is a variation in the execution time.

For RDDs, the Features data lines are split based on comma. This gives us lists of each line. Then we take the index 0 and 3 which is the Store Id and fuel price as a key-value pair. In the stores data, we split based on the “,” and then filter where the store size is greater than 150000. This gives us a data of store ID and store size as a key value pair. Then we join both these data based on the store ID as the key. Now, we have the joined data:

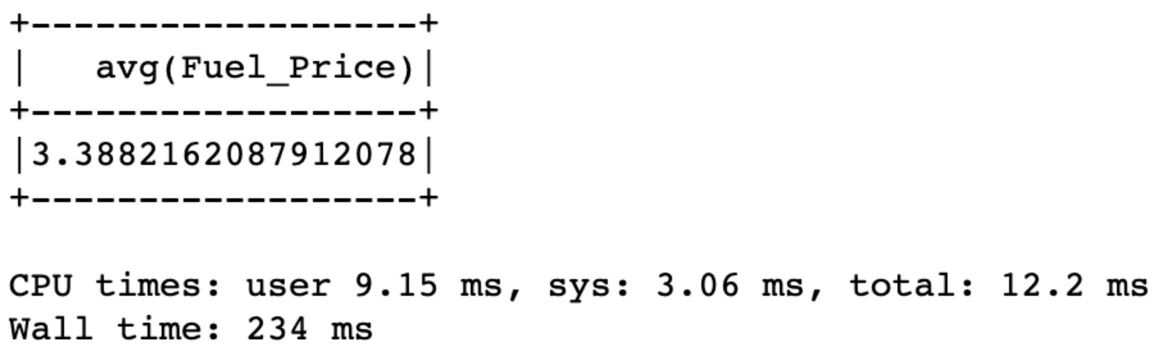
```
[('4', ('2.598', '205863')),
 ('4', ('2.573', '205863')),
 ('4', ('2.54', '205863')),
 ('4', ('2.59', '205863')),
 ('4', ('2.654', '205863')),
 ('4', ('2.704', '205863')),
 ('4', ('2.743', '205863')),
 ('4', ('2.752', '205863')),
 ('4', ('2.74', '205863')),
 ('4', ('2.773', '205863')),
 ('4', ('2.81', '205863')),
 ('4', ('2.805', '205863')),
 ('4', ('2.787', '205863')),
 ('4', ('2.836', '205863')),
 ('4', ('2.845', '205863')),
 ('4', ('2.82', '205863')),
```

This gives us a list of tuples with a key and the corresponding tuple. The first element is the key (Store ID) and the second element (tuple) (contains the fuel price and the size) is the value. Then we take the key value of the tuple within (the fuel price) and calculate the mean of it. By this, we get the mean of the fuel price which is 3.388216208..

```
CPU times: user 31.4 ms, sys: 10.4 ms, total: 41.8 ms
Wall time: 271 ms

3.3882162087912087
```

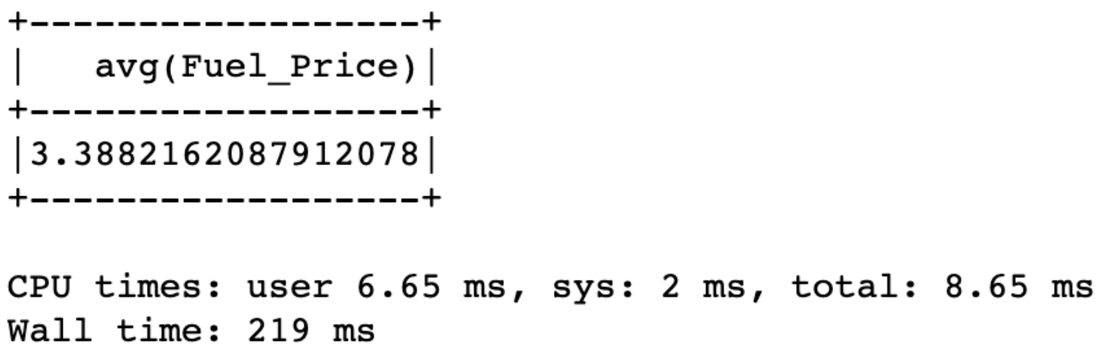
For DataFrames, we basically filter where the size is greater than 150000 and inner join the features and store df based on the store ID and calculate the mean of the fuel price.



```
+------------------+
|   avg(Fuel_Price)|
+------------------+
|3.3882162087912078|
+------------------+

CPU times: user 9.15 ms, sys: 3.06 ms, total: 12.2 ms
Wall time: 234 ms
```

For Spark-SQL, we inner join the features and store data and select the average fuel price where the size is greater than 150000.



```
+------------------+
|   avg(Fuel_Price)|
+------------------+
|3.3882162087912078|
+------------------+

CPU times: user 6.65 ms, sys: 2 ms, total: 8.65 ms
Wall time: 219 ms
```

Overall, SparkSQL is the fastest with an execution wall time of 219ms. The second fastest is Dataframes with 235ms wall time and the last is the RDD with a wall time of 271ms.

Also talk about this question in a detailed answer, “Why is RDD faster than DF?”

Dataframes are faster than RDDs. The speed of execution based on time is ordered as

- Spark-SQL
- DataFrames
- RDDs.

Spark SQL is the fastest among all the three while dataframes is the second fastest when it comes to execution. RDDs is slow when compared to these two. Thus, RDDs are not faster than DF. The time might differ when running on different machines. Overall, the DataFrame and SQL queries run faster compared with RDD.