Big Data Assignment

Kyriacos Xanthos
CID: 01389741

April 25, 2022

MD5 Sums

- `prox-fixed.csv`: d04a9027e74db1ce2408b65e8d6b7a03
- `prox-mobile.csv`: a3d2c735e7cf9e8b6ceade46290c35bd
- `bldg-measurements.csv`: 3f7577041d6590f8bccb58d0ad18c92f
- `f2z2-haz.csv`: 477cbae9ae42ca72e39304637ca5e83f

Question 1

For this question we are using one mapper and one reducer. The mapper creates key-value pairs where the key is the date (not including the time) and the id of the staff member in the building. The value is always 1 to each of these key value pairs in the mapping phase, ie. (date id, 1). Before these key-value pairs are passed to the reducer they are sorted using `HADOOP`'s shuffle sort algorithm. The reducer counts how many unique id's are identified on each date and they are aggregated to create a total count of unique prox-ids on each date.

The code for the mapper and reducer is shown below:

```
> cd bd-sp-2017/coursework
> nano q1_mapper.py
```

```
#!/usr/bin/env python
import sys
# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    words = line.split(',')
    # make sure we skip header
    if 'timestamp' not in words:
        words = [word.strip() for word in words]
        # get the date
        date = words[0][:-9]
        id = words[2]
        key = date + ' ' + id
        print ('%s\t%s' % (key, 1))
```

```
> nano q1_reducer.py
```

```
#!/usr/bin/env python
import sys

current_id = None
```

```
current_date = None
current_count = 0

for line in sys.stdin:
    line = line.strip()

    # parse the input we got from mapper.py
    date_id, count = line.split('\t')
    date , id = date_id.split()

    # convert count (currently a string) to int
    try:
            count = int(count)
    except ValueError:
            # count was not a number, so silently
            # ignore/discard this line
            continue

    if id != current_id:
        current_count += 1
        current_id = id

    if date != current_date:
        if current_date != None:
            print ('%s\t%s' % (current_date, current_count))
        current_date = date
        # initialize all ids since we are in a diffferent date
        current_id = None
        current_count = 0

# do not forget to output the last word if needed!
if current_id == id:
    print ('%s\t%s' % (current_date, current_count))
```

We will be using both prox-fixed and prox-mobile datasets since they have the same format for the dates and the ids. The following command is used to run the Hadoop streaming command on Athena:

```
hadoop jar $HADOOP_STR/hadoop-streaming-2.7.0-mapr-1808.jar \
-libjars $HADOOP_STR/hadoop-streaming-2.7.0-mapr-1808.jar \
-input coursework/prox-*.csv -output q1_output \
-mapper "python q1_mapper.py" -file q1_mapper.py \
-reducer "python q1_reducer.py" -file q1_reducer.py
```

The results of this command are shown on table 1 and figure 1 and we can observe that about 110 staff members are present in the building on weekdays and only about 2-3 in the weekends (on the 4^{th} of June no one appears to be in the building. We can inspect the output of this execution using:

```
hadoop fs -cat q1_output/part* | less
```

Date	Number of staff members present
2016-05-31	115
2016-06-01	114
2016-06-02	116
2016-06-03	115
2016-06-05	2
2016-06-06	116
2016-06-07	116
2016-06-08	116
2016-06-09	117
2016-06-10	115
2016-06-11	3
2016-06-12	2
2016-06-13	115

Table 1: Diagram of the number of unique prox-ids on each day in the prox-fixed and prox-mobile datasets.

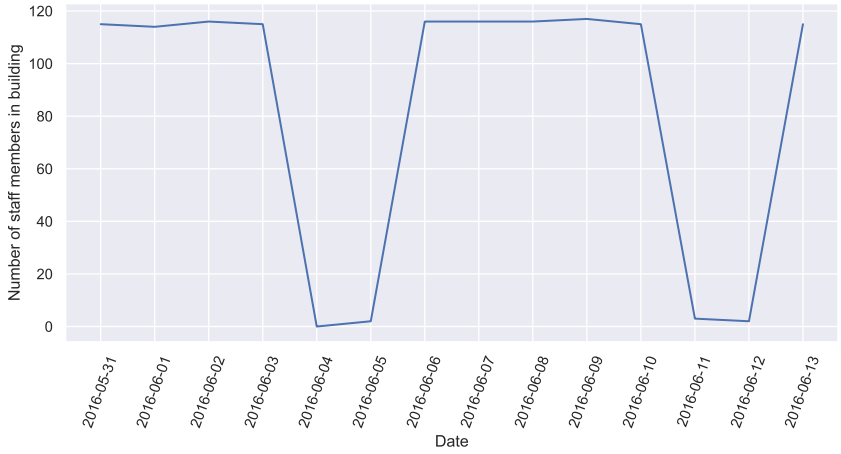


Figure 1: Amount of employees in the building on each day

Question 2

To find the (floor, zone) of the most visited location in the building we need to first create key value pairs using a mapper program. The keys will be the floor, zone and the value will always be one for each entry in the prox-fixed dataset. The reducer will then take the sorted output from the mapper and for each floor-zone increment the count. When the current floor-zone is finished, the count is checked with the previous maximum count and if it is larger then the max count, the corresponding floor-zone is updated. The code for both of the mapper and reducer is shown below:

```
> nano q2_mapper.py
```

```
#!/usr/bin/env python
import sys
# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    words = line.split(',')
    # make sure we skip header
    if 'timestamp' not in words:
        words = [word.strip() for word in words]
        floor = words[3]
        zone = words[4]
```

```
        key = floor + ' ' + zone
        print ('%s\t%s' % (key, 1))
```

```
> nano q2_reducer.py
```

```
#!/usr/bin/env python
import sys

current_id = None
current_floor_zone = None
current_count = 0
max_count = 0
max_fz = None

for line in sys.stdin:
    line = line.strip()

    # parse the input we got from mapper.py
    floor_zone, count = line.split('\t')

    # convert count (currently a string) to int
    try:
            count = int(count)
    except ValueError:
            # count was not a number, so silently
            # ignore/discard this line
            continue


    if current_floor_zone == floor_zone:
        # if the same fz, we increment count
        current_count += 1

    else:
        # make sure this is not none
        if current_floor_zone:
            # update max count if it is larger than the previous
            if current_count >= max_count:
                max_count = current_count
                max_fz = current_floor_zone
        current_count = count
        current_floor_zone = floor_zone

# print the final count
print('MAX FZ {} with count {}'.format(max_fz, max_count))
```

To run the streaming command on Athena we use:

```
> hadoop jar $HADOOP_STR/hadoop-streaming-2.7.0-mapr-1808.jar \
-libjars $HADOOP_STR/hadoop-streaming-2.7.0-mapr-1808.jar \
-input coursework/prox-fixed.csv \
-output q2_output \
-mapper "python q2_mapper.py" \
-file q2_mapper.py \
-reducer "python q2_reducer.py" \
-file q2_reducer.py
```

To inspect the output we run:

```
> hadoop fs -cat q2_output/part*

Floor Zone of the most visited location: 2 1 with prox-fixed ID counts 6154
```

The printout shows that the most visited location in the building is floor 2, zone 1 with 6,154 prox-id readings. This is the Break Room in the middle of the building which makes sense: Most of the staff members use this area for their breaks and therefore it is the most visited location in the building. It is also next to the stairs which means that anyone moving from floor 3 to floor 1 or vice versa would need to pass from that Break Room.

Question 3

To find the most active staff member on a particular day (2nd June 2016) we need to find the greatest number of prox-readings on that specific date. Using the prox-fixed dataset, we create a mapper that prints out key value pairs of (id, 1) only if the date is equal to 2nd June 2016. The reducer then takes the sorted (by id) output of the mapper and in a similar manner to question 2, it counts how many instances appear for the same id. Before moving to the next id, it checks whether the total count is larger than the last id counted, and if it is, it updates the maximum (id, count) pair. This can be seen from the code below.

```
> nano q3_mapper.py
```

```
#!/usr/bin/env python
import sys
# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    words = line.split(',')
    # make sure we skip header
    if 'timestamp' not in words:
        words = [word.strip() for word in words]
        # get the date
        date = words[0][:-9]
        id = words[2]
        if date == '2016-06-02':
            print ('%s\t%s' % (id, 1))
```

```
> nano q3_reducer.py
```

```
#!/usr/bin/env python
import sys

current_id = None
current_count = 0
max_count = 0
max_id = None

for line in sys.stdin:
    line = line.strip()

    # parse the input we got from mapper.py
    id, count = line.split('\t')
```

```
    # print('floor zone is {}'.format(floor_zone))

    # convert count (currently a string) to int
    try:
            count = int(count)
    except ValueError:
            # count was not a number, so silently
            # ignore/discard this line
            continue

    if current_id == id:
        current_count += 1

    else:
        if current_id:
            if current_count >= max_count:
                max_count = current_count
                max_id = current_id
        current_count = count
        current_id = id

print('id with greatest number of prox-ID reading {} with count
↪  {}'.format(max_id, max_count))
```

The streaming command in Athena is:

```
> hadoop jar $HADOOP_STR/hadoop-streaming-2.7.0-mapr-1808.jar \
-libjars $HADOOP_STR/hadoop-streaming-2.7.0-mapr-1808.jar \
-input coursework/prox-*.csv \
-output q3_output \
-mapper "python q3_mapper.py" \
-file q3_mapper.py \
-reducer "python q3_reducer.py" \
-file q3_reducer.py
```

```
> hadoop fs -cat q3_output/part*

id with greatest number of prox-ID reading fresumir001 with count 64
```

The id of the most active staff member is `fresumir001` with count 64.

Question 4

For this question we look at the `bldg-measurements.csv` file and create a time series plot of the average hourly "Total Electric Demand Power" (`tedp`). We first use a mapper to create a key value pair of (time, `tedp`). Then the reducer takes these sorted (by time) key-value pairs and while the time is the same, it sums all the `tedp` values. Before checking the next time stamp, it finds the average of the `tedp` values and prints the time and the average `tedp` in that hour.

```
> nano q4_mapper.py
```

```
#!/usr/bin/env python
import sys
# input comes from STDIN (standard input)
for line in sys.stdin:
```

```
    # remove leading and trailing whitespace
    line = line.strip()
    words = line.split(',')
    # make sure we skip header
    if 'Date/Time' not in words:
        words = [word.strip() for word in words]
        # get the time
        time = words[0][11:13]
        # total electric demand power
        tedp = words[8]
        print ('%s\t%s' % (time, tedp))
```

```
> nano q4_reducer.py
```

```
#!/usr/bin/env python
import sys
current_time = None
current_count = 0
current_sum = 0.

for line in sys.stdin:
    line = line.strip()

    # parse the input we got from mapper.py
    time, tedp = line.split('\t')

    if current_time == time:
        current_count += 1
        current_sum += float(tedp)

    else:
        if (current_time) and (current_count!=0):
            print(time, current_sum / current_count)
        current_count = 0
        current_sum = 0.
        current_time = time
```

To run the streaming command in hadoop:

```
> hadoop jar $HADOOP_STR/hadoop-streaming-2.7.0-mapr-1808.jar \
-libjars $HADOOP_STR/hadoop-streaming-2.7.0-mapr-1808.jar \
-input coursework/bldg-measurements.csv \
-output q4_output \
-mapper "python q4_mapper.py" \
-file q4_mapper.py \
-reducer "python q4_reducer.py" \
-file q4_reducer.py
```

We can inspect the time series data using:

```
> hadoop fs -cat q4_output/part* | less
```

Note that because the data only contains 24 entries corresponding to hours of the day, we just manually copied those values to our local computer. If the data was larger we would use the `scp` command like in Question 10. The time series plot can be seen in figure 3. The code for creating the plot can be found in appendix A.

We observe that `tedp` is on average higher during working hours (7 am to 6 pm) and it levels off at lower levels for the rest of the hours.

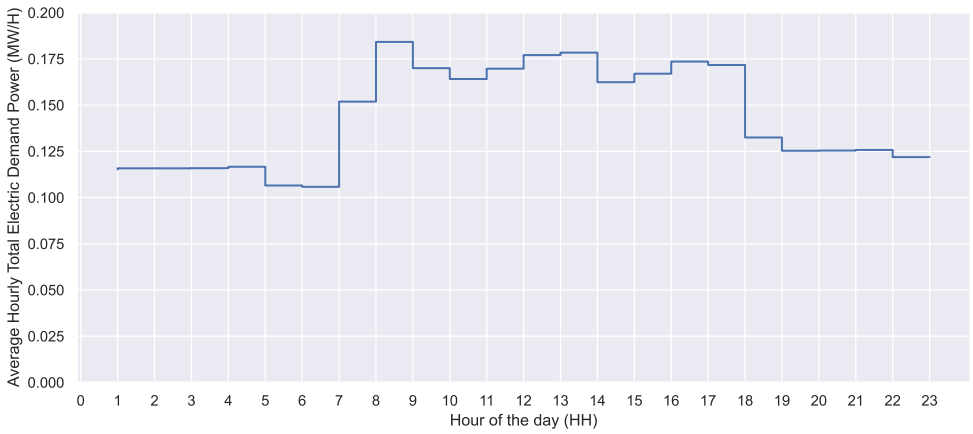


Figure 2: Time series plot of average hourly Total Electric Demand Power. The hours are presented as HH.

Question 5

We read the `prox-fixed.csv` file directly from HDFS so that the parallel storing structure present in HDFS is passed and replicated in the Spark shell. We create a `case class` which is immutable and contains the 4 required variables (timeStamp, id, floorNum and zone) with their corresponding formats. Then we use a parse function which reads each line of our csv file and breaks the string to the required components. We create the `isHeader` function to skip the header of the csv file and after using this filter we use a map function which creates the required `RDD[ProxReading]` variable. We then cache `pro_fixed` so that whenever we use this variable we will not need to re-compute it. It is also good practise because in the event of a node failure of a machine this variable will be safely stored.

```
import org.joda.time.DateTime

val pro_fixed_file =
↪  sc.textFile("hdfs:///user/user434/coursework/prox-fixed.csv")

case class ProxReading(timeStamp: org.joda.time.DateTime, id: String,
 floorNum: String, zone: String)


def isHeader(line: String): Boolean = {
 line.contains("timestamp")
}

def parse(line: String) = {
    val pieces = line.split(", ")
    val fmt = org.joda.time.format.DateTimeFormat.forPattern("yyyy-MM-dd
    ↪  HH:mm:ss")
    val timestamp_unformatted = pieces(0)
    val timestamp = fmt.parseDateTime(timestamp_unformatted)
    val id = pieces(2).trim()
    val floorNum = pieces(3).trim()
    val zone = pieces(4).trim()
    ProxReading(timestamp, id, floorNum, zone)
}

val pro_fixed = pro_fixed_file.filter(x => !isHeader(x)).map(parse)
pro_fixed.cache()
```

Question 6

For this question we create a map reduce pipe that firstly creates key value pairs of the form (floor zone, 1) and then use the `reduceByKey` command to aggregate the sum of each key. This command uses distributed sorting to sort the input by key and then accumulates the counts for each key. We then sort the list of (floor zone, counts) by the number of counts in ascending order and then use the `collect()` method to collect all of the results in one machine and print out the last element. As expected, we get the same result as question 2, that the floor 2 zone 1 is the most visited location in the building with 6,154 counts.

```
val most_visited_location = pro_fixed.map(x => (x.floorNum + " " + x.zone,
↪  1)).reduceByKey(_+_).sortBy(_._2)
```

```
scala> most_visited_location.collect().last
res0: (String, Int) = (2 1,6154)
```

Question 7

We read the files again directly from HDFS. We create a case class `ProxReading_Mobile` which only takes one argument, the id of the staff employee. We create a function that picks out the date we want to check (7^{th} of June) and create a parse function that takes the id from the files we read. We then create two `RDD[ProxReading_Mobile]` variables corresponding to the prox-mobile and prox-fixed datasets. After caching these variables we create key value pairs of (id, 1), then union both datasets together and finally reduce them using the `reduceByKey` function so that each unique id is collected together with how many times it appeared in the datasets. Sorting by the count, we print the last element that is the same as our solution in question 3: fresumir with 64 counts.

```
val pro_mobile_file =
↪  sc.textFile("hdfs:///user/user434/coursework/prox-mobile.csv")

val pro_fixed_file =
↪  sc.textFile("hdfs:///user/user434/coursework/prox-fixed.csv")

case class ProxReading_Mobile(id: String)

def isHeader(line: String): Boolean = {
 line.contains("timestamp")
}


def is_correct_date(line: String): Boolean = {
    line.contains("2016-06-07")
}

def parse2(line: String) = {
    val pieces = line.split(", ")
    val id = pieces(2).trim()
    ProxReading_Mobile(id)
}

val pro_mobile = pro_mobile_file.filter(x => !isHeader(x)).filter(x =>
↪  is_correct_date(x)).map(parse2)

val pro_fixed_id = pro_fixed_file.filter(x => !isHeader(x)).filter(x =>
↪  is_correct_date(x)).map(parse2)
```

```
pro_mobile.cache()

pro_fixed_id.cache()

val total_counts = pro_mobile.map(x => (x.id , 1)).union(pro_fixed_id.map(x
↪  => (x.id , 1))).reduceByKey(_+_).sortBy(_._2)
```

```
scala> total_counts.collect().last
res3: (String, Int) = (fresumir001,64)
```

Question 8

The two computational platforms perform identical operations in different ways. The mapper program we use for Hadoop is the same as the `map` function we use in Spark. The reducer program we use for Hadoop is similar to the `reduceByKey` function we use in Spark, depending on how we want to reduce the input. Both of them make use of HDFS distributed data storage and they are both fast and effective for performing easy map-reduce tasks.

Using Hadoop, there is more freedom into what the mapper and reducer can do, because one can explicitly add if conditions, change types of variables and have more control of what is printed out of the program. These things can also be coded in Spark in different ways, for example instead of using an if statement to check that we only look at one particular date, we add a filter that basically does the same job.

It is also very noticeable that the amount of code needed for the same operations using both platforms is very different. Mapper operations in Hadoop for this assignment were usually about 15-20 lines of code whereas in spark they were 5-10 lines depending on the parse function. Similarly, reduce programs in Hadoop were roughly 50 lines of code whereas for Spark the reduce operation was just 1 line of code.

Overall, Hadoop does give more freedom into what exactly happens in the map-reduce operations but Spark requires much fewer lines of code for the same commands. Both use HDFS's distributed data storage so by taking into consideration the wider applicability of spark for more high level problems (like Machine Learning tools) and the more user friendly error messages, the Spark platform gives a better overall experience for Big Data analysis.

Question 9

In this question we use the `bldg-measurements.csv` file to find the date and time of the first occurrence of the F_2_Z_1 VAV Reheat Damper Position being fully opened. To do this, we create a case class `BLDG` which takes as arguments the timestamp and damper position. The parse function takes the timestamp and the damper position from the csv file and using the map operation we create a `bldg_measurements` variable which includes all measurements. We then sort the variable by date and then using a filter we find the first instance of when the damper position is equal to 1 (ie. fully opened). The date and time found is 2016-06-02, 15:20:00.

```
val bldg_measurements_file =
↪  sc.textFile("hdfs:///user/user434/coursework/bldg-measurements.csv")

case class BLDG(date: String, damper_position: Float)

def isHeader(line: String): Boolean = {
 line.contains("Date/Time")
}

def parse(line: String) = {
    val pieces = line.split(", ")
```

```
    val timestamp_unformatted = pieces(0)
    val damper_position = pieces(192).trim().toFloat
    BLDG(timestamp_unformatted, damper_position)
}

val bldg_measurements = bldg_measurements_file.filter(x =>
↪  !isHeader(x)).map(parse).sortBy(_.date)
```

```
scala> bldg_measurements.filter(x => x.damper_position == 1).first
res4: BLDG = BLDG(2016-06-02 15:20:00,1.0)
```

Question 10

To understand whether there is an association between Hazium concentration and the damper position we first plotted both variables on the same axis, which can be seen in figure 3. We can see that the hazium concentration goes to very high levels at the timestamp 1×10^6 which is equivalent to the $10th$ of June. We also observe that the damper position fluctuates between 0 and 1 multiple times before 0.4×10^6 and afterwards an unusual pattern appears. We explored this further in figure 4a where we see the cumulative plot of when the damper position to returns to 0. At first, the damper position was returning to 0 every 250 timestamp points but then between 1300-2000 it takes an unusually long time to return to 0, same for 3500-4000.

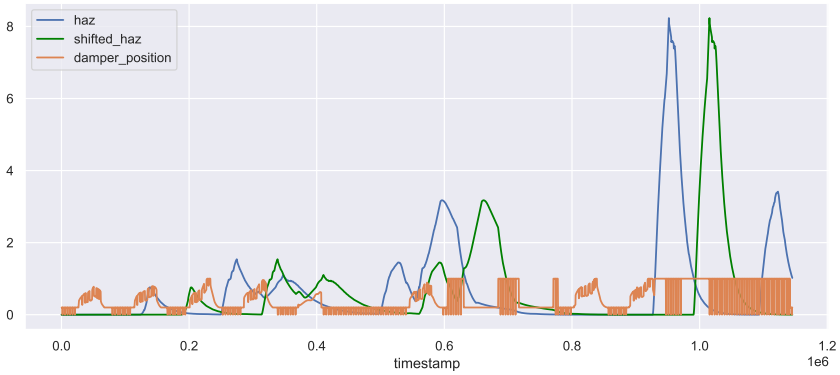
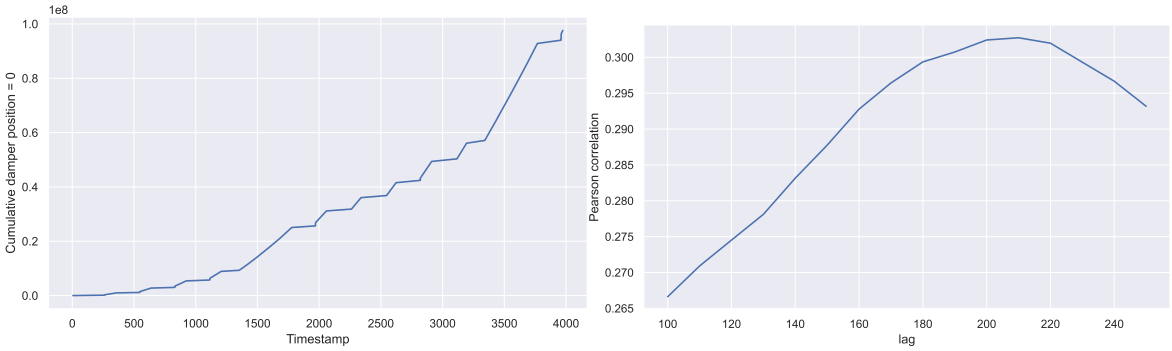


Figure 3: Time series plot of hazium concentration, hazium concentration with lag = 210 and damper position.

There is not a very clear relationship between the two variables when looking at the figure 3. In fact, when using the `mllib.stat` package in Spark to calculate the Pearson correlation coefficient between the two variables, the correlation was only 018. This shows that there is very little statistical association between the two variables, although it is not exactly zero.

To explore this further, we thought that maybe there is a lagged relationship between the two variables. This makes intuitive sense, because the damper position might not have an immediate effect in the hazium concentration, instead it might take some time for the hazium to be released and left uncontrolled to hit the dangerous levels (more than 6). We calculated the correlation between the two variables between a lag of 100 and 250 time-points, and the results can be seen in figure 4b. The correlation seems to be the maximum at lag 210 with correlation 0.30 which is significantly larger that the 0.18 before, and it hints that there actually is a statistical association between the two variables. The 210 lag equals 17.5 hours $((210 \times 5)/60 = 17.5)$ which suggests that there are almost 18 hours needed for the damper position to have an effect in the hazium concentration. We added the logged hazium time series plot in figure 3 where there is a more clear relationship between the two variables.

We now look at who was in the server room on the day before the concentration of hazium went to dangerous levels, ie. June the 10^{th}. By looking at all the ids that visit the Server room, it seems that all of the ids have multiple visits to the server room spread within the two week period, except id `ncalixto001` that only has one visit for the two week period.



(a) Cumulative plot of instances when the damper position was eqaul to 0.

(b) Correlation between Hazium concentration andd damper position as we increase the lag of hazium concentration

Figure 4: Plots exploring relationship between hazium concentration and damper position.

Looking at when that visit was, it was on the 10^{th} of June, a few hours before the haziumm concentration went to dangerous levels. We therefore conclude that an investigation should take place, as to why `ncalixto001` was in the server room on that day.

The code for this question is included in appendix B.

Question 11

We create a case class `ProxReading` taking arguments timeStamp, id, floor number, zone. We then filter the variable storing all the above arguments so that the zone only corresponds to the Server Room. We then filter again to make sure that the timeStamp is equal to the date we want to check (2016-06-10), and collect all the observations that are filtered. The employee id's that were in the server room that day are: pyoung001, sflecha001, ncalixto001, lbennett001, csolos001.

```
import org.joda.time.DateTime

val pro_fixed_file =
↪  sc.textFile("hdfs:///user/user434/coursework/prox-fixed.csv")

val haz_measurements_file =
↪  sc.textFile("hdfs:///user/user434/coursework/f2z2-haz.csv")


case class ProxReading(timeStamp: String, id: String,
 floorNum: String, zone: String)


def isHeader(line: String): Boolean = {
 line.contains("timestamp")
}

def parse(line: String) = {
    val pieces = line.split(", ")
    // val fmt = org.joda.time.format.DateTimeFormat.forPattern("yyyy-MM-dd
    ↪  HH:mm:ss")
    val timestamp_unformatted = pieces(0).substring(0, 10).trim()
    // val timestamp = fmt.parseDateTime(timestamp_unformatted)
    val id = pieces(2).trim()
    val floorNum = pieces(3).trim()
    val zone = pieces(4).trim()
```

```
    ProxReading(timestamp_unformatted, id, floorNum, zone)
}

val pro_fixed = pro_fixed_file.filter(x => !isHeader(x)).map(parse)
```

```
scala> pro_fixed.filter(x => x.zone == "Server Room").filter(x => \
x.timeStamp == "2016-06-10").collect().foreach(println)
ProxReading(2016-06-10,pyoung001,3,Server Room)
ProxReading(2016-06-10,sflecha001,3,Server Room)
ProxReading(2016-06-10,ncalixto001,3,Server Room)
ProxReading(2016-06-10,lbennett001,3,Server Room)
ProxReading(2016-06-10,csolos001,3,Server Room)
```

Question 12

Boris Johnson has decided he wants to increase the efficiency of Satander Cycles in London. He has noticed that there are a lot of complaints that sometimes people are not able to find any bikes at certain stations, and sometimes the stations are full and cannot park the cycles they have rented. He wants you to conduct an analysis behind how many bikes are usually used and how this is affected by weather and holidays.

You are given access to a Bike Sharing dataset [1] that includes the hourly and daily count of rental bikes for the years 2011 and 2012 in Capital bikeshare system together with the weather information of each day and hour. More information about the dataset and what each column represents can be found here.

Assume the dataset is large enough that only Big Data technologies can handle the amount of data in the dataset. For the next 4 questions (1-4) write Map-Reduce programs to compute the answers. Include all the code used and the commands needed to run the Map-Reduce programs.

1. Using the `hour.csv` file produce a diagram of the total rental bikes (including both casual and registered) used each month.
2. Find the date from the two year period that most of the casual users rented a bike.
3. Find the first date that the weather conditions included Heavy Rain, Ice Pallets, Thunderstorm, Mist, Snow and Fog and how many total rental bikes were used. (Instances where `weathersit` was equal to 4).
4. Create a time series plot of the average hourly counts of rented bikes by registered users. Add another line that includes the average hourly counts of casual users.

Figure 5: Image downloaded from Compaign Live (2015)

For the remaining questions write a Spark sequence of commands to compute the answer. Include all code for used in the spark shell in your answers.

5. Parse the `hour.csv` data file into an RDD[BikesCounts] where BikesCounts is defined as:

```
case class BikesCounts(timeStamp: org.joda.time.DateTime, season:
↪ Double, holiday: Double, workingdate: Double, weathersit:
↪ Double, atemp: Double, humidity: Double, windspeed: Int, count:
↪ Double)
```

6. Solve questions 2-3 now using Spark. You can create a new RDD if needed but this is not necessary.

7. Now create a new RDD that takes the windspeed variable and maps all the values that are equal to 0 to the value 0 and anything more than 0 equal to one. This will act as a boolean variable indicating if there is wind or not, and not incorporate the strength of the wind. All the other variables in the `BikesCounts` should stay the same.

8. Inspect the correlation between the counts and the rest of the variables in the RDD. Which variable has the largest correlation? Does this make sense?

9. Create a `parseddata:org.apache.spark.mllib.regression.LabeledPoint` object that has the count variable as the feature and the rest of the variables in the RDD as the covariates.

10. Create a Linear Regression Model that is able to predict the counts of rental bikes based on the rest of the variables. Report some regression metrics like MSE, R^2.

11. Boris asks you to predict how many bikes will be needed next weekend based on the Linear Regression Model you have built. You can find the weather forecast for the next weekend from BBC Weather.

12. Read the following paper and produce a short summary of the key points raised: [2]

13. The paper raises the point that Big Data are often of heterogeneous nature. Provide an example for this, perform an analysis using either spark or Hadoop and comment on how you would tackle heterogeneity.

Quesstion 13

The UK automotive industry is huge, with about 2 million cars registered every year [3]. The average age of cars on the road in the UK is about 8 years, which shows that people in the UK constantly look to buy new cars. This motivates the dataset we decided to use for this question, which is the 100,000 UK Used Car Data set. This dataset was created by scrapping used car listings from the web, and contains 100,000 cars with 9 different brands. We preprocessed the data so that all of the listings are included in one csv file. This is a relatively small dataset but we can assume the dataset is large enough that Big Data technologies are needed to analyze it.

We begin by using Hadoop to find some key statistics for the dataset at hand. We first create a map reduce program that finds the most popular car by first creating key value pairs of the form (brand model, 1) and then using a reducer to find the brand, model of the most popular car. The most popular car listed in UK dealerships was Ford Fiesta with 6,557 entries.

Next, we wanted to see the amount of Ford Fiesta models listed corresponding to their manufacturing year. This would give an insight as to what the average age of the cars was at the time of listing. We did this using a mapper that only selects Ford Fiesta models and then a reducer that prints the year of manufacturing and the amount of cars. This part also helped us to identify an outlier observation (a listing corresponding to year 2060) which was later removed from the dataset.

We can see from table 2 that most of the cars in the listings were cars manufactured in 2017-2018. Since the dataset was created in 2021, this means that most of the Ford Fiesta cars listed online were 3-4 years old.

1998	2000	2004	2005	2006	2007	2008	2009	2010	2011
1	1	3	1	4	8	19	33	18	25
2012	2013	2014	2015	2016	2017	2018	2019	2020	
39	279	379	498	849	1575	1822	917	85	

Table 2: Number of Ford Fiesta Cars manufactured in each year (1998-2020).

We then uploaded the file in Spark and created a `CarsFeatures` RDD that includes the price, year, milage, engineSize, brand, transmission and fueltype as variables. We filtered out the Header and the Outliers and created a `cars` RDD with all the variables. We then checked to see if there were any strong correlations between the features of the listed car and the price. We found a 0.63 correlation between price and engine size using a Pearson correlation coefficient. This was computed using the `org.apache.spark.mllib.stat` library. Moreover, we found a 0.5 correlation with the manufacturing year and a -0.42 correlation with the milage of the car. All of these correlations make intuitive sense because we expect the price to be lower as the milage of the car is higher since the car's parts *worn off*. Also, the smaller the age of the car, the higher the price and the larger the engine of the car the more expensive it is to manufacture and the price is therefore higher.

Moving forward we wanted to create One-Hot Encoding features for the categorical variables in our dataset, so that a Linear Regression model can be built later. We created these encodings for model brand, transmission and fuel type. They were then stored in the RDD `encode_cars` and created a `parsedData:LabeledPoint` which included the target (price) and all of the other columns as features (24 features including the hot-endoded ones).

We then used the `mllib` library to make a linear regression with the `LinearRegressionModel` function. To quantify the quality of fit of our model we are going to use R^2 defined as:

$$R^2 = 1 - \frac{\text{RSS}}{\sum_{i=1}^{n} \left(Y_i - \bar{Y}\right)^2} \tag{1}$$

where RSS is the residual sum of squares (quantifies the departure of the data from the model) Y are the true values and $\hat{Y}$ the predicted values for the price.

Initially our fit was not very good ($R^2 = 0.67$) but we noticed by plotting the price variable that it was heavily right skewed. Of course in the very big data setting we might have not been able to plot the data, so transforming the variables when we get a bad fit would be something to always check. We log-transformed the price and performed the fit again which gave an $R^2 = 0.78$ and a root mean squared error (RMSE) of 0.25. We transferred the actual data from HDFS to a local computer and plotted the results to understand the fit.

Figure 6: Linear Regression fit with price as the target variable.

As seen from figure 6 our model is able to successfully predict most of the prices very accurately. It is clear that we could spend some time cleaning the data and removing outliers with measures like Cook's distances but this goes beyond the scope of the question.

This analysis has shown that we can easily do a linear regression on a Big Data framework and that tools like Pearson correlation, one-hot encoding can easily be programmed in Spark. Other models were also explored like Ridge Regression , Random Forrest Regression but we decided not to include them in this summary. The code for this question is included in appendix C.

Question 14

Part A

The paper "Statistical paradises and paradoxes in big data (i): law of large populations, big data paradox, and the 2016 US presidential election" by Meng tackles two key subjects: The use of Big Data for Population Inference and use of Big Data for Individualised Inference/Prediction. The former uses metrics to show that data quality is far more important than data quantity, that there are other strategies other than probabilistic sampling to ensure data quality, and to overcome data quality issues we need a huge fraction of the data. The latter is presented in a way that suggests that statistics becomes an approximation scheme, where individuals are approximated by their proxy population, that the bias-variance tradeoff should be used to determine the appropriate level of approximation and that the concept of bias is more important than the variance in the data, especially in the large scale. The paper presents statistical concepts for dealing with big data and applies them to the 2016 US Presidential election.

Section 1 of the paper motivates the reason we need to address the problem of Big Data and how our traditional methods for assessing probabilistic uncertainty should move from the standard error ($\sigma/\sqrt{n}$) to the relative bias ($\rho\sqrt{N}$) where n the sample size and N to be the population size.

Section 2 begins with the motivating question "Is an 80% non-random sample 'better' than a 5% random sample in measurable terms?". This is addressed by inference the population mean of some quantity G. The equation is for assessing the quality of the sample suggested to be:

$$\bar{G}_n - \bar{G}_N = \underbrace{\rho_{R,G}}_{\text{Data Quality}} \times \underbrace{\sqrt{\frac{1-f}{f}}}_{\text{Data Quantity}} \times \underbrace{\sigma_G}_{\text{Problem Difficulty}}, \quad \bar{G}_n = \frac{1}{n}\sum_{j\in I_n} G_j = \frac{\sum_{j=1}^{N} R_j G_j}{\sum_{j=1}^{N} R_j} \tag{2}$$

This equation suggests that three factors affect our estimation error. The data quality is captured by the correlation between the G and R (where R represents the response mechanism) as it measures the sign and the degree of the selection bias. The Data Quantity is captured using the sampling rate $f = n/N$. Finally, the problem difficulty is measured through the variation of G, since the more variation, the larger the σ, the more difficult it is to estimate G_N accurately. It is possible to measure quantitatively how well the sample size captures the population using the mean squared error:

$$\text{MSE}_R\left(\bar{G}_n\right) = \text{E}_R\left[\rho^2_{R,G}\right] \times \left(\frac{1-f}{f}\right) \times \sigma^2_G \equiv D_I \times D_O \times D_U \tag{3}$$

This means we can increase the data quality by reducing D_I, increase the data quantity by reducing D_O and reduce the difficulty of estimation by reducing D_U. These might not always be possible but they give us a intuitive way of addressing some problems in our chosen samples.

Section 3 explores equations 2 and 3 in more detail and captures the problems with low-quality Big Data through the Law of Large Populations (the error of $\bar{G}_j$ grows at the rate of $\sqrt{N}$ for the same $E_{\boldsymbol{R}}(\rho_{R,G})$). Meng derives an upper bound for the effective sample size $n_{\text{eff}} \leq \frac{n}{1-f}\frac{1}{ND_I}$ which demonstrates that small biases in the sampling mechanism become

disastrous as N increases. The section is concluded with the Big Data Paradox: The bigger the data, the surer we fool ourselves.

Section 4 applies these concepts to the outcomes of the 2016 US presidential election. Specifically it measures the D_I from equation 3 and the author quantitatively proves the serious under reporting for Trump's supporters. It is also shown that there are very different bench-marking standard errors for larger states and that led to the overconfidence for the election results in 2016.

Finally, section 5 illustrates the importance of 2 in the big picture of statistics. It explores these measures for multiple populations and explores applications of Quasi Monte Carlo. The author also illustrates the problems that sometimes arise with data confidentiality and concludes with the optimistic view that the more young people explore the realm of Big Data Statistics, the more we can utilise its potential as scientists.

The key statistical contribution of the paper is that very small correlations between response and the covariates can have huge impact to the statistical analysis for Big Data.

Part B

The dataset used in question 13 was web-scraped from the internet. It could also be constructed by visiting dealerships in the UK and collecting the data through a survey. In fact, many dealerships might not have websites which means they were all excluded from the sample used in Question 13. If the data were collected through in-person surveys, then the key points raised in Meng's paper are relevant for our analysis and we can proceed with this assumption forward.

The sample size $n = 100,000$, and we estimate our total population (the total number of listed cars at the time of the survey) to be be $N \approx 1,000,000$. This is a rough estimate based on the fact that 7.9 million *used* cars were sold in 2019 [3] which means that on average about $20,000$ used cars are sold each day, and the average time for a car to be sold from the day it is listed is about 45 days [4].

The quantity G in our example is the price of the car, shown at a particular listing. The response mechanism R is a binary variable indicating whether a listed car appears in our dataset. There are multiple reasons why our sample estimate for G might be biased. Some examples are outlined below:

- Day or Month of visit at the dealership might affect the price of the car:
  - A large shipment of cars from a different country could decrease the car prices of the specific brand/model since dealerships will want to get ahead of competition.
  - New regulations for CO_2 emissions might drive the car prices of conventional engine cars lower.
  - New incentives from the government for electric cars might drive their price up.
  - Current Inflation of the economy.
- Not all dealerships might be willing to disclose all of their prices
- Some dealerships could have different discounting schemes that would drive the actual price of the car lower than what is listed.

These biases can be mitigated and increase the data quality by taking some measures in acquiring the data. For example, we could make sure that the data are collected in a short period of time and take random samples from different dealerships rather than using all of the data provided by the dealerships.

For our analysis in question 13, we use a sampling rate $f = 100,000/1,000,000 = 0.1$ which is significantly larger than what Wang proposes through his equation 2. This suggests that we could spend more time cleaning the data, trying to remove biases in multiple ways like randomly sampling from the websites, or decreasing the amount of the same cars appearing in the dataset (like Ford Fiesta), and a sample size of $10,000$ or even less could give equally good (and probably more accurate) results.

The paper by Wang focuses on how we assess the performance of the sample mean when we want to estimate the population mean. Our analysis in question 13 involved a linear regression model, so the most applicable statistic for our analysis would be a measure of how the coefficients of the regression model we built can fit the entire population of all cars available. This problem will be completely different to how the estimate for the sample mean was tackled by Wang, and to the best of our knowledge there is no existing literature tackling this issue. In the Big Data world, linear regression is the most interpretable and easy model to implement. However, the fact that a study of how sample size affects the coefficients for the whole population has not been implemented yet, shows how much more work should go in the field of understanding the statistics of Big Data.

Many recent papers also tackle the issue of data quality in Big Data. Bradley *et al* [5] have recently looked at significant over estimations of US Vaccine uptake in the US. They conclude similarly to Wang that a random sample size of 10 could give the same (inaccurate) results as the survey by Delphi-Facebook for 250,000 people.

Kauss *et al* [6] propose 10 rules for effective statistical practice that are very applicable to this day and age of Big Data. They emphasize that we can understand a lot from missing data and understanding the patterns with which missing data arises can have massive implications for dealing with data quality. Unfortunately the dataset we used did not have any missing data therefore we could not easily assess the quality of the data, but we understand that the biases we mentioned in this section could give an insight into how the data quality of the cars dataset could have affected our analysis in question 13.

References

[1] Hadi Fanaee-T and Joao Gama. Event labeling combining ensemble detectors and background knowledge. *Progress in Artificial Intelligence*, pages 1–15, 2013. ISSN 2192-6352. doi: 10.1007/s13748-013-0040-3. URL [WebLink].

[2] Peter Bühlmann and Sara van de Geer. Statistics for big data: A perspective. *Statistics Probability Letters*, 136:37–41, 2018. ISSN 0167-7152. doi: https://doi.org/10.1016/j.spl.2018.02.016. URL https://www.sciencedirect.com/science/article/pii/S0167715218300610. The role of Statistics in the era of big data.

[3] Society of Motor Manufacturers and Traders. Smmt motor industry facts 2020. https://www.smmt.co.uk/wp-content/uploads/sites/2/SMMT-Motor-Industry-Facts-JUNE-2020-FINAL.pdf, 2020. [Online; accessed 13-April-2022].

[4] James Baggott. Fastest selling used cars in april revealed as car dealers slash time models are in stock on forecourts. https://cardealermagazine.co.uk/publish/fastest-selling-used-cars-in-april-revealed-as-car-dealers-slash-time-models-are-in-s223020, 2021. [Online; accessed 18-April-2022].

[5] Valerie C. Bradley, Shiro Kuriwaki, Michael Isakov, Dino Sejdinovic, Xiao-Li Meng, and Seth Flaxman. Unrepresentative big surveys significantly overestimated US vaccine uptake. *Nature*, 600(7890):695–700, dec 2021. doi: 10.1038/s41586-021-04198-4. URL https://doi.org/10.1038%2Fs41586-021-04198-4.

[6] Robert E. Kass, Brian S. Caffo, Marie Davidian, Xiao-Li Meng, Bin Yu, and Nancy Reid. Ten simple rules for effective statistical practice. *PLOS Computational Biology*, 12 (6):1–8, 06 2016. doi: 10.1371/journal.pcbi.1004961. URL https://doi.org/10.1371/journal.pcbi.1004961.

A Code for Question 4

```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import seaborn as sns
sns.set_theme()
plt.rcParams['figure.figsize'] = (12,5)
plt.rcParams['figure.dpi'] = 80


df = pd.DataFrame(data={

'01': 115217.34262530117,
'02': 115811.11016347304,
'03': 115801.70906242428,
'04': 115880.79261856287,
'05': 116648.94543734947,
'06': 106514.02059940119,
'07': 105799.95474790418,
'08': 151882.14258484848,
'09': 184176.85309277117,
'10': 169984.75482891573,
'11': 164121.39139212118,
'12': 169704.57312574852,
'13': 177061.38221437126,
'14': 178386.57954311377,
'15': 162398.93439573172,
'16': 166989.15328614446,
'17': 173546.53333353295,
'18': 171684.80188614456,
'19': 132489.87956848484,
'20': 125311.93179090915,
'21': 125420.91426325304,
'22': 125730.92196969697,
'23': 121866.72329515149}, index=[1] ).T
df = df / 10**6

# df.plot()
plt.step(x=np.arange(1,24,1), y=df[1])
plt.xlabel('Hour of the day (HH)')
plt.ylabel('Average Hourly Total Electric Demand Power (MW/H)')
plt.xticks(np.arange(0,24,1))
plt.ylim((0,0.2))
plt.savefig('q4_plot.pdf')
plt.show()
```

B Code for question 10

```
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.stat.{MultivariateStatisticalSummary,
↪  Statistics}
import spark.implicits._

val bldg_measurements_file =
↪  sc.textFile("hdfs:///user/user434/coursework/bldg-measurements.csv")
```

```
val haz_measurements_file =
↪  sc.textFile("hdfs:///user/user434/coursework/f2z2-haz.csv")

case class BLDG_HAZ(time: String, pos_conc: Double)

def isHeader(line: String): Boolean = {
 line.contains("Date/Time")
}

def parse_bldg(line: String) = {
    val pieces = line.split(", ")
    val timestamp_unformatted = pieces(0)
    val damper_position = pieces(192).trim().toDouble
    BLDG_HAZ(timestamp_unformatted, damper_position)
    // (timestamp_unformatted, damper_position)
}

def parse_haz(line: String) = {
    val pieces = line.split(", ")
    val timestamp_unformatted = pieces(0)
    val haz_conc = pieces(1).trim().toDouble
    BLDG_HAZ(timestamp_unformatted, haz_conc)
    // (timestamp_unformatted, haz_conc)
}

val damper_position = bldg_measurements_file.filter(x =>
↪  !isHeader(x)).map(parse_bldg)

val haz_conc = haz_measurements_file.filter(x =>
↪  !isHeader(x)).map(parse_haz)

val damper_data = damper_position.map(x => (x.time, x.pos_conc))

val haz_data = haz_conc.map(x => (x.time, x.pos_conc))

val combined_position_haz = damper_data.join(haz_data)

val cph_observations = combined_position_haz.map(_._2._1).map(x =>
↪  Vectors.dense(x.toDouble)) // type:
↪  org.apache.spark.rdd.RDD[org.apache.spark.mllib.linalg.Vector]

val cph_summary: MultivariateStatisticalSummary =
↪  Statistics.colStats(cph_observations)

val correlation =
↪  Statistics.corr(combined_position_haz.map(_._2._1),combined_position_haz.map(_._2._2)

> correlation: Double = 0.17852920930396365


val df1 = damper_data.toDF("date", "damper_position")
val df2 = haz_data.toDF("date", "haz")

val df_all = df1.join(df2, Seq("date"), "inner").orderBy("date")
df_all.show(false)
```

```
// add a unique timestamp
val df_trial = df_all.withColumn("unix_timestamp", unix_timestamp($"date",
↪  "yyyy-MM-dd HH:mm:ss"))
df_trial.show(false)


// to single file: this actually works but it saves the folder in HDFS!
df_trial.coalesce(1).write.format("com.databricks.spark.csv").save("my_data")

// then from terminal we do: hadoop fs -get my_data

// and then from local scp -r
↪  user434@athena.ma.ic.ac.uk://home/user434/bd-sp-2017/coursework/trial_data
↪  results/


// now we add lag
import org.apache.spark.sql.functions.lag
import scala.collection.mutable.ListBuffer

val w = org.apache.spark.sql.expressions.Window.orderBy("date")

val leadDf = df_all.withColumn("new_col", lag("haz", -210, 0).over(w))


// this is same as before
leadDf.stat.corr("damper_position", "haz", "pearson")

//better (0.302)
leadDf.stat.corr("damper_position", "new_col", "pearson")

val my_list = new ListBuffer[Double]()

 for(a<-100 to 250 if a %10 == 0)
{
        val leadDf = df_all.withColumn("new_col", lag("haz", -a, 0).over(w))
        val cor = leadDf.stat.corr("damper_position", "new_col", "pearson")
        my_list +=  cor
}

// saved to python file for plotting
my_list


// now see who was in the server room

val pro_fixed_file =
↪  sc.textFile("hdfs:///user/user434/coursework/prox-fixed.csv")


case class ProxReading(timeStamp: org.joda.time.DateTime, id: String,
 floorNum: String, zone: String)


def isHeader(line: String): Boolean = {
 line.contains("timestamp")
}
```

```
def parse(line: String) = {
    val pieces = line.split(", ")
    val fmt = org.joda.time.format.DateTimeFormat.forPattern("yyyy-MM-dd
    ↪  HH:mm:ss")
    val timestamp_unformatted = pieces(0)
    val timestamp = fmt.parseDateTime(timestamp_unformatted)
    val id = pieces(2).trim()
    val floorNum = pieces(3).trim()
    val zone = pieces(4).trim()
    ProxReading(timestamp, id, floorNum, zone)
}

 val server_room_visits = pro_fixed_file.filter(x =>
 ↪  !isHeader(x)).map(parse).filter(x => x.zone == "Server Room")

server_room_visits.collect().foreach(println)

server_room_visits.map(x => (x.id,
↪  1)).reduceByKey(_+_).collect().foreach(println)
```

```
scala> server_room_visits
(csolos001,6)
(ncalixto001,1)
(clais001,6)
(pyoung001,4)
(sflecha001,7)
(lbennett001,5)
```

Python code for plotting

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme()

plt.rcParams['figure.figsize'] = (9,5)
plt.rcParams['figure.dpi'] = 80

df =
↪  pd.read_csv("/Users/lysi2/Documents/UNI_Imperial/Big_Data/CW/results/trial_data/sampl
↪  header=None)
df = df.rename(columns={0:'datetime', 1:'damp', 2:'haz', 3:'timestamp'})

df['timestamp'] -= 1464649000

# plot when the damper is equal to zero
df_zeros = df[df['damp'] ==0]
df_zeros['cum_timestamp'] = df_zeros['timestamp'].cumsum()
plt.plot(df_zeros['cum_timestamp'])
plt.xlabel('Timestamp')
plt.ylabel('Cumulative damper position = 0')
plt.savefig('cum.pdf')
plt.show()
```

```
df_new = pd.concat([df['timestamp'], df['haz'], df['damp'],
↪  df['haz'].shift(-210)], axis=1, keys=['timestamp', 'haz', 'damp',
↪  'shifted_haz']).dropna()

# now plot timeseries
plt.plot(df_new['timestamp'], df_new['shifted_haz'], label = 'haz')
plt.plot(df_new['timestamp'], df_new['haz'], label = 'shifted_haz', color =
↪  'green')
plt.plot(df_new['timestamp'], df_new['damp'], label = 'damper_position')
plt.legend()
plt.xlabel('timestamp')
plt.savefig('shifted_haz.pdf')
plt.show()

shift = np.arange(100, 260, 10)
lst = np.array([0.26661869844530883, 0.2709075522082924,
↪  0.27451899115016276, 0.2781068554746894, 0.28315302692523525,
↪  0.2877515355894161, 0.29277531373589594, 0.29642891641589064,
↪  0.29936321202845884, 0.300740064897844, 0.3024166142781724,
↪  0.30272683424746455, 0.3019709148899516, 0.29931759781199707,
↪  0.2966697222308417, 0.29317569242504754])

# find optimum shift
plt.plot(shift, lst)
plt.xlabel('lag')
plt.ylabel('Pearson correlation')
plt.savefig('pearcorr.pdf')
plt.show()
```

C Code for question 13

C.1 Map-Reduce Part

```
nano q13_mapper1.py
```

```
import sys
# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    words = line.split(',')
    # make sure we skip header
    if 'model' not in words:
        words = [word.strip() for word in words]
        # get the model of the car
        model = words[1]
        # get the brand of the car
        brand = words[-1]
        key = brand + ' ' + model
        print ('%s\t%s' % (key, 1))
```

```
nano q13_reducer1.py
```

```
#!/usr/bin/env python
import sys
```

```
current_id = None
current_brand_model = None
current_count = 0
max_count = 0
max_bm = None

for line in sys.stdin:
    line = line.strip()

    # parse the input we got from mapper.py
    brand_model, count = line.split('\t')

    # convert count (currently a string) to int
    try:
            count = int(count)
    except ValueError:
            # count was not a number, so silently
            # ignore/discard this line
            continue

    if current_brand_model == brand_model:
        current_count += 1

    else:
        if current_brand_model:
            if current_count >= max_count:
                # update max count if the current most popular car
                # is more popular than the previous one
                max_count = current_count
                max_bm = current_brand_model
        # initialise count again
        current_count = count
        current_brand_model = brand_model


print('Brand and Model of most popular car: {} with count {}'.format(max_bm,
     max_count))
```

```
nano q13_mapper2.py
```

```
import sys
# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    words = line.split(',')
    # make sure we skip header
    if 'model' not in words:
        # get rid of all other models and outlier
        if ((' Fiesta' in words) and ('2060' not in words)):
            words = [word.strip() for word in words]
            # get the date
            year = words[2]
            key = year
            print ('%s\t%s' % (key, 1))
```

```
nano q13_reducer2.py
```

```
import sys

current_date = None
current_count = 0

for line in sys.stdin:
    line = line.strip()

    # parse the input we got from mapper.py
    date, count = line.split('\t')
    # convert count (currently a string) to int
    try:
            count = int(count)
    except ValueError:
            # count was not a number, so silently
            # ignore/discard this line
            continue


    if date != current_date:
        if current_date != None:
            print ('%s\t%s' % (current_date, current_count))
        current_date = date
        current_count = 1
    else:
        current_count += 1

# do not forget to output the last word if needed!
if current_date == date:
    print ('%s\t%s' % (current_date, current_count))
```

```
nano Q13.scala
```

```
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.regression.LinearRegressionModel
import org.apache.spark.mllib.regression.LinearRegressionWithSGD
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.evaluation.RegressionMetrics
import org.apache.spark.mllib.stat.{MultivariateStatisticalSummary,
↪  Statistics}

val cars_file = sc.textFile("hdfs:///user/user434/coursework/car_data.csv")

def isHeader(line: String): Boolean = {
 line.contains("model")
}

def isOutlier(line: String): Boolean = {
    line.contains("2060")
}

case class CarsFeatures(price: Double, year: Double,
 milage: Double, engineSize: Double, brand: String, transmission: String,
 ↪  fueltype: String)

def parse(line: String) = {
    val pieces = line.split(",")
```

```
    val price = pieces(3).trim().toDouble
    val year = pieces(2).trim().toDouble
    val milage = pieces(5).trim().toDouble
    val engineSize = pieces(7).trim().toDouble
    val brand = pieces(8).trim()
    val transmission = pieces(4).trim()
    val fueltype = pieces(6).trim()
    CarsFeatures(price, year, milage, engineSize, brand, transmission,
    ↪  fueltype)
}

val cars = cars_file.filter(x => !isHeader(x)).filter(x =>
↪  !isOutlier(x)).map(parse)

// explore some correlations
val check_cars = cars.map(x => (x.price ,x.year, x.engineSize, x.milage))
val correlation =
↪  Statistics.corr(check_cars.map(_._1),check_cars.map(_._2),"pearson")
val correlation =
↪  Statistics.corr(check_cars.map(_._1),check_cars.map(_._3),"pearson")
val correlation =
↪  Statistics.corr(check_cars.map(_._1),check_cars.map(_._4),"pearson")


// one hot encodings
val indexed_brand =
↪  cars.map(x=>x.brand).distinct().sortBy[String](x=>x).zipWithIndex().collectAsMap()

val indexed_transmission =
↪  cars.map(x=>x.transmission).distinct().sortBy[String](x=>x)
.zipWithIndex().collectAsMap()

val indexed_fuel = cars.map(x=>x.fueltype).distinct().sortBy[String](x=>x)
.zipWithIndex().collectAsMap()


def encode_brand(x: String) =
 {
    var encodeArray = Array.fill(12)(0)
    encodeArray(indexed_brand.get(x).get.toInt)=1
    encodeArray
 }

def encode_transmission(x: String) =
 {
    var encodeArray = Array.fill(5)(0)
    encodeArray(indexed_transmission.get(x).get.toInt)=1
    encodeArray
 }

def encode_fuel(x: String) =
 {
    var encodeArray = Array.fill(6)(0)
    encodeArray(indexed_fuel.get(x).get.toInt)=1
    encodeArray
 }
```

```
val encode_cars = cars.map{ x => (x.price,x.year,x.milage,x.engineSize ,
                                  encode_brand(x.brand),
                                  ↪  encode_transmission(x.transmission),
                                  encode_fuel(x.fueltype))}


val new_encode_cars = encode_cars.map(x => ((log(x._1)), Seq(x._2, x._3,
↪  x._4, x._5(1), x._5(1)
                                            , x._5(2), x._5(3), x._5(4),
                                            ↪  x._5(5), x._5(6)
                                            , x._5(7), x._5(8), x._5(9),
                                            ↪  x._5(10), x._5(11)
                                            , x._6(1), x._6(2), x._6(3),
                                            ↪  x._6(4)
                                            , x._7(1), x._7(2), x._7(3),
                                            ↪  x._7(4), x._7(5) )))

val parsedData = new_encode_cars.map{case (k, vs) => LabeledPoint(k,
↪  Vectors.dense(vs.toArray))}.cache()

// step size should be small enough
val numIterations = 500
val stepSize = 0.00000001

// define linear regression
val algorithm = new LinearRegressionWithSGD()

// use intercept
algorithm.setIntercept(true).optimizer.setNumIterations(numIterations)
.setStepSize(stepSize)

// fit model
val model = algorithm.run(parsedData)

val pred_actual = parsedData.map { labeledPoint =>
    // make predictions using the model
    val prediction = model.predict(labeledPoint.features)
    (prediction, labeledPoint.label)
}

val metrics = new RegressionMetrics(pred_actual)

// Root Mean Squared error
println(s"RMSE = ${metrics.rootMeanSquaredError}")
// RMSE = 0.2505

println(s"R-squared = ${metrics.r2}")
// R-squared = 0.78

// create dataframe to move data to local
val pred_df = pred_actual.toDF()
// write dataframe
pred_df.coalesce(1).write.format("com.databricks.spark.csv").save("my_pred_data")
```

```
// to get data to local
> hadoop fs -get my_pred_data
```

```
> scp -r user434@athena.ma.ic.ac.uk://home/user434/bd-sp-2017/data/ \
my_pred_data results/
```

```
nano plot_results.py
```

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import *
sns.set_theme()

y_hdfs = pd.read_csv('../../results/my_pred_data/predictions.csv', names =
↪  ['y', 'y_pred'])

y= y_hdfs['y']
y_hat = y_hdfs['y_pred']

rmse = mean_squared_error(y, y_hat, squared=False)

res = pd.DataFrame(
            data = {'y': y, 'y_hat': y_hat, 'resid': y - y_hat}
    )

plt.figure(figsize=(12, 6))
plt.subplot(121)
sns.lineplot(x='y', y='y_hat', color="grey",
data = pd.DataFrame(data={'y': [min(y),max(y)], 'y_hat': [min(y),max(y)]}))
sns.scatterplot(x='y', y='y_hat', data=res, alpha=0.5).set_title("Fit plot")
plt.subplot(122)
sns.scatterplot(x='y', y='resid', data=res, alpha=0.5).set_title("Residual
↪  plot")
plt.suptitle("Model rmse = " + str(round(rmse, 4)), fontsize=16)
plt.savefig('my_fit.pdf')
plt.show()
```