PART A – Message Length Analysis

Exercise

After the HDFS breaks down the data (Olympictweets2016rio) into input splits (blocks), it then splits it into records of roughly 1 line per text. The arrList further divides the records into 4 separate chunks [0] epoch_time, [1] tweetId, [2] tweet and [3] device, respectfully.

```
String[] arrList = input.toString().split(";");
```

The if statement checks if the length of the arrList is equal to 4 to avoid a NullPointerException. It then checks if the length of the tweet is less than or equal to 140 to filter out foreign languages that may result in a high level of length tweets.

```
if (arrList.length == 4 && arrList[2].length() <= FILTER)
```

Although linear, efficiency is fairly essential when dealing with MapReduce. So instead of hard coding the ranges in a bunch of if statements, two was only needed. The ranges are formed in two different units. The first if the tweet length is a multiple of 5 and the other if its not.

```
int tweetLength = arrList[2].length();

if (tweetLength % 5 == 0) {
    start = tweetLength - 4;
    end = start + 4;
} else {
    start = tweetLength - (tweetLength % 5) + 1;
    end = start+ 4;
}
```

The ranges and values are then aggregated into bins.

```
6-10    11447
11-15   10244
16-20   31926
21-25   83224
26-30   222092
31-35   314410
36-40   356555
```

Figure 1a – A snippet of the output bin

Lastly we emit the range [1-5, 6-10…] of type Text and value of type IntWritable to the reducer.

```
range.set(start + "-" + end);
context.write(range, value);
```

To increase optimization and fluidity of the MapReduce job, a text object was created and initialized outside the map method instead of inside – context.write(new Text(start + "-" + end), value).

After the shuffle and sorting phase the results from the mapper are passed to the reducer. The reducer takes in the emitted Text and an IntWritable value associated to the variables from the mapper class. Then we aggregate the summation of the values in the for-each loop.

```
int sum = 0;
for (IntWritable value : values) {
    sum = sum + value.get();
}
result.set(sum);
```

After mapping and reducing the final output from the reducer writes and saves on HDFS.

```
context.write(key, result);
```

The only modification made in the main class is:

```
job.setNumReduceTasks(1);
```

The reducer is set to one so the output is in a single bin.

Results

Even though twitter only allows 140 characters, some tweets may exceed it via foreign characters like Kanji, and emojis like ☄☀☂. This is shown in figure 1b when results exceed the maximum tweet length. I deliberately withheld filtering the tweets so I could compare both scenarios.

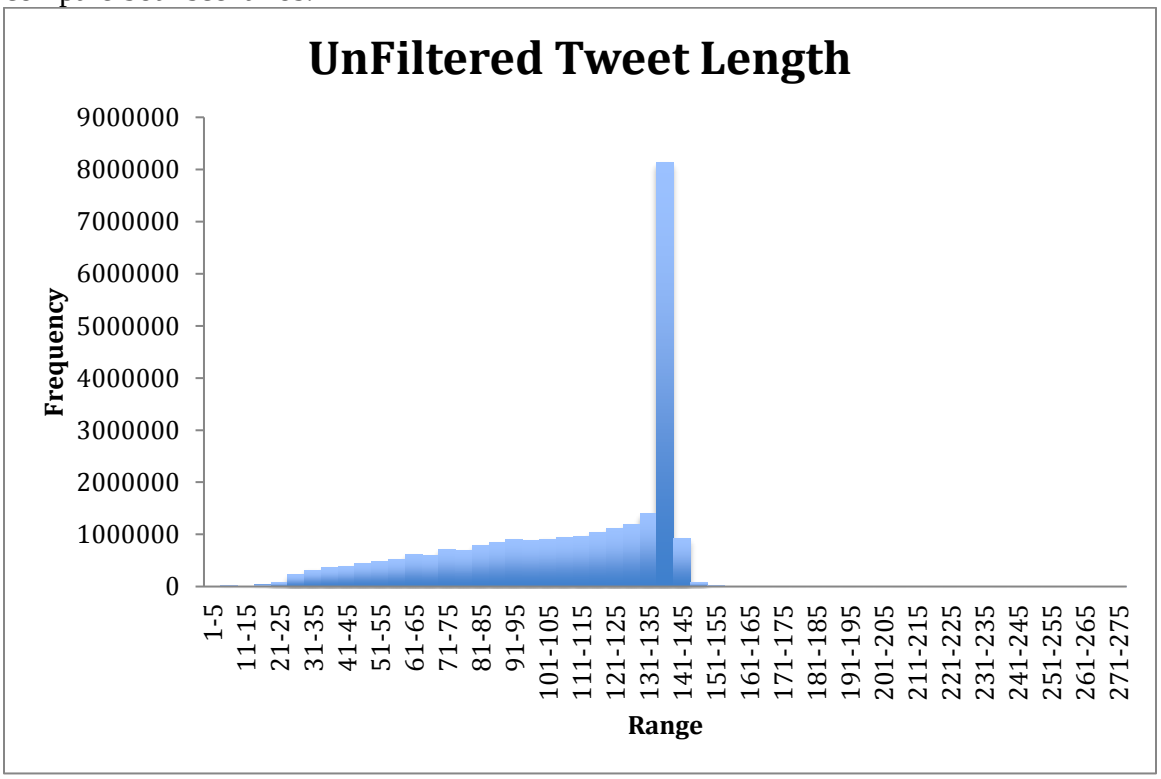

Figure 1b – unfiltered tweets

Figure 1c on the other hand shows a more accurate description of tweets. It displays tweets that are less than 140 characters; hence filters out foreign characters, emojis and other than them that exceeds the maximum tweet length.

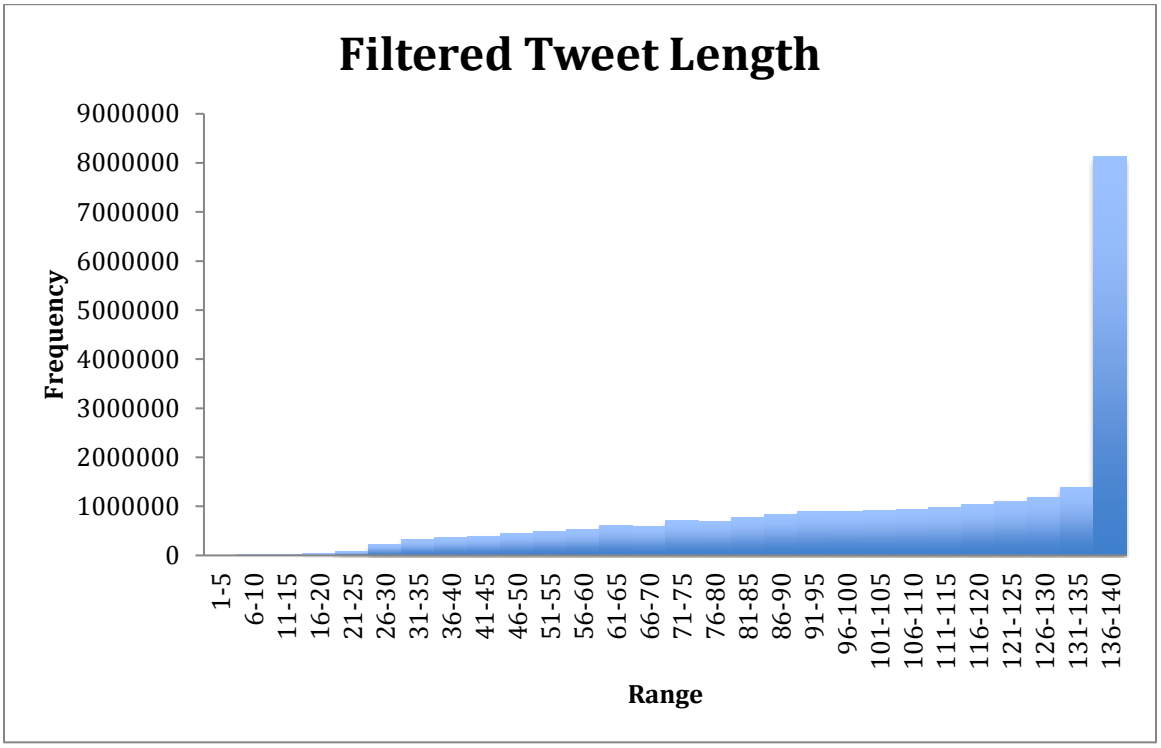

Figure 1c – filtered tweets

The highest frequency of tweets was between the 136-140-character ranges; this implies that most tweeters tweeted the maximum length that twitter allows. This illustrates how enthusiastic people are about the games.
Remarkably, not a single tweet was found in the data that ranged between 1-5 *(See bin in src/Question1/out filter).*

Pseudo code / relaxed explanation
Though you can access the code directly from the src folder it may be easier to read the pseudo code provided below. This shows only the Mapper since both the driver and reducer are fairly similar to the ones provided in the labs.

```
Split input
if: input length == 4 && input[2] length <=140
        if: input[2] length is multiple of 5
                compute start and end range
        else:
                compute start and end range
emit(range, value)
```

PART B (i) – Time Analysis

Assumption:

1. Tweets are filtered
2. Days are not considered

Exercise

DateFormat and SimpleDateFormat were used to extract the hours of the twitter file. At first glance the results looked correct, but later discovered that it gave the wrong description of time. This is because Date doesn't take time zones into consideration. Example, if we hard code '1469453965000L' through Data's arguments it gives 14 instead of 13.39 when using epochConverter *(See Epoch Converter 2017).* So in theory this would work, but it needs to be subtracted by -1.

```
if (arrList.length == 4) {
    try {
        DateFormat format = new SimpleDateFormat("HH");
        Date date = new Date(Long.parseLong(arrList[0]));
        String formatted = format.format(date);
        context.write(new Text(formatted), value);
    } catch (Exception e) {
    }
```

The modifier type LocalDateTime and the method epochSecond were later used to improve accuracy of tweet times. It passes 3 arguments. The first is the epoch time (arrList[0]) of type long since it exceeds Integer.MAX_VALUE. LocalDateTime returns a timestamp in milliseconds so it needs to be divided by 1000. The second is the nanoseconds, which was left at 0 (found no reason to calculate the nanosecond within the second). Lastly the time zone offset; a fully resolved offset from UTC/Greenwich that uses the same offset for all local date-times.

A try and catch is used to catch any invalid date times. More specifically DateTimeParseException.

```
if (arrList.length == 4 && arrList[2].length() <= FILTER) {
    try {
        LocalDateTime date = LocalDateTime.ofEpochSecond
        (Long.parseLong(arrList[0]) / 1000, 0, ZoneOffset.UTC);
        hour.set(date.getHour());
        context.write(hour, value);
    } catch (Exception e) {
    }
}
```

The hours were extracted from LocalDateTime by calling getHour() via date, and later emitted to the reducer as the feature key.

The reducer will then compute the tweets of every hour.

Results

The time analysis shows that there was a high level of activity in the initial hours of the Olympic games. The peak number of tweets was in the first hour of the games; reason for this maybe the frenzy when the games opened. It later dips and then increases hours later, a reason for this might be that tweeters where sleeping and then woke up.

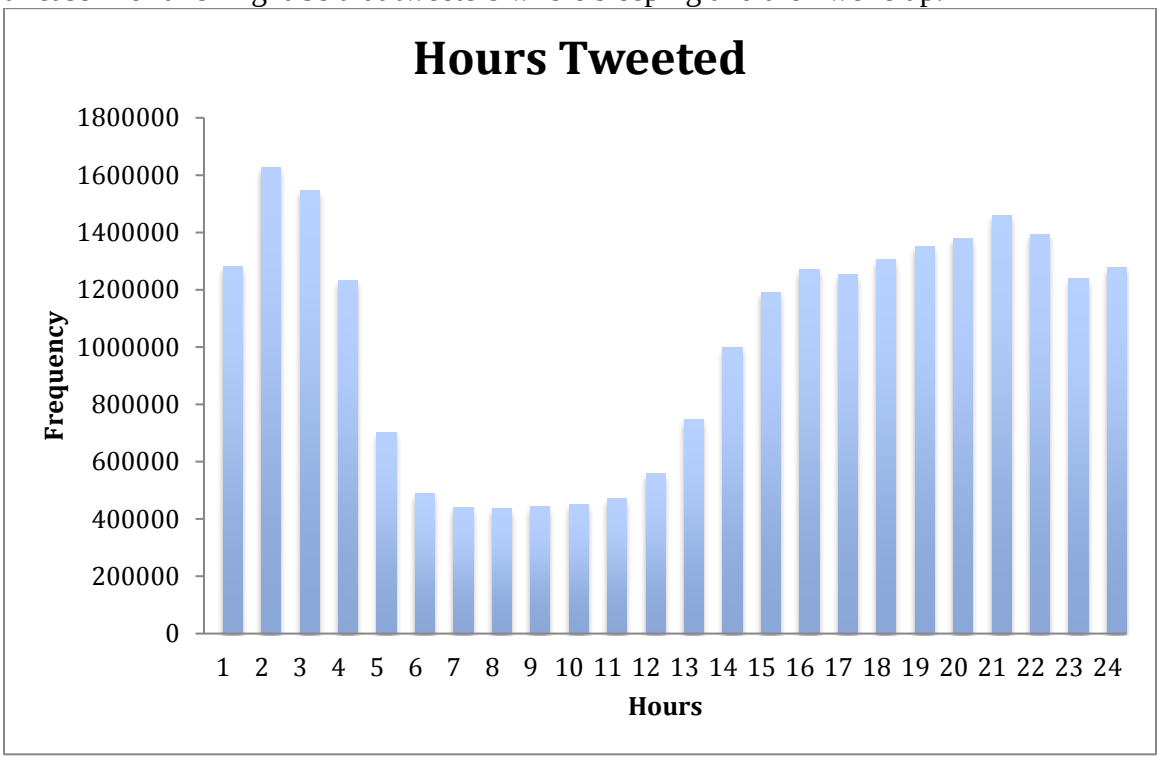

Figure 2a – relationship between tweets in given hours

Pseudo code / relaxed explanation
Get full date of epoch time
Extract only hour
emit(hour, value)

PART B (ii) – Time Analysis

Assumption:

1. A tweet that reads "###rio##2016##omg##" will not be computed
2. The most populated hour tweeted may seem like its 2 in the bar chart, but I assume hours begin at 0; hence the most populated hour is 1 with 1,626,026 and not 2.
3. Tweets are filtered

Exercise
Java.util.regex package offers a powerful and economical object model to implement regular expressions in Java; it is composed of three objects, the Pattern, Matcher and Pattern syntax exception *(See RegExr 2017).*

The Pattern passes a regular expression in its argument; this regular expression is used to check if the string matches the expression. The compile() is used multiple times to match the regular expression against multiple texts, in this case the tweets. So it almost acts like a 'filter' to block out the tweet and only catches the hashtags.

```
Pattern pattern = Pattern.compile("(#\\w+)");
```

The Matcher class matches the regular expression against the whole text; in this case the tweets.

```
Matcher matcher = pattern.matcher(arrList[2]);
```

The highest number of tweets in any given hour was 1. This was hard coded into the if statement to check if date equaled 1.

```
if (date.getHour() == POPULAR_HOUR)
```

Multiple matches can be found in the input text file and later collected in groups and then emitted to the reducer to calculate the summation of the hashtags.
For computation purposes a .toLowerCase() was used to include both capital and lower case characters. Equally .toUpperCase() could've been used.

```
while (matcher.find()) {
    for (int i = 1; i <= matcher.groupCount(); i++) {
        hashtag.set(matcher.group(i).toLowerCase());
        context.write(hashtag, value);
    }
```

Pseudo code / relaxed explanation

Get full date of epoch time
Pass regex expression through pattern argument
Pass tweets through matcher argument
if: date,getHour = 1
    while(tweet)
        loop: 1-number of capturing groups in tweet
            emit(hashtag, value)

Results

Figure 2b shows the top 10 hashtags in the most popular hour of the Olympic games. The Olympic games happened in Rio Brazil in the year 2016, so there's no surprise that the most trending hashtag is rio2016, olympic and bra. The most active twitter users in 2016 where from the states, hence why usa is also trending *(See Tweets 2017)*. One of the most popular sports in the Olympic games is futebol, translated from Portuguese to English is Football. Which comes in at number 5 in the most popular tweets.

Number	Hashtags	Frequency
1	#rio2016	1449254
2	#olympics	91756
3	#gold	68144
4	#bra	50263
5	#futebol	49366

6	#usa	42756
7	#oro	40899
8	#swimming	36649
9	#cerimoniadeabertura	36499
10	#openingceremony	35974

Figure 2b – top 10 hashtags from tweets

The hour that generated the highest amount of tweets was 1 with a frequency of 1,626,026 total tweets, and within that hour the highest hashtag frequency was rio2016 with 1,449,254. Meaning on average rio2016 appeared in 89% of all tweets within that hour.

PART C (i) – Support Analysis

Assumption:

1. A tweet that reads "@athlete @athlete @athlete" will count as one
2. Tweets are filtered
3. The small data set is never empty

Exercise

Since the dataset is relatively small to be kept in memory, a replication map side join is used. It takes a small dataset and replicates it across the cluster (medalistrio.csv).

```
job.addCacheFile(new Path("/data/medalistsrio.csv").toUri());
```

In replication join there's no need for a reducer, but since we are computing the frequency of both athlete names and sports; we need one.

The medalistrio.csv text contains 11 columns separated by a comer, one of which that we need; fields[1] contains the names of the athletes. The athleteName Map <String, String> takes in key and value of all athletes name. This is the loading process of the join.

```
while ((line = br.readLine()) != null) {
    String[] fields = line.split(",");
    if (fields.length == 11)
        this.athletesName.put(fields[1], fields[1]);
}
```

Technically all computations can be done by most Collections from the java framework, but in this case a HashTable Map was used.

A for each loop is used to iterate through the athletes name via key. The names of the athletes are matched with tweets mentioned by users. If the twitter message contains the athlete name it will be emitted to the reducer with feature representing the athlete name, and value calculating how many times the athlete was mentioned.

```
for (String athlete : athletesName.keySet()) {
    if (arrList[2].contains(athlete)) {
        allAthletes.set(athletesName.get(athlete));
        context.write(allAthletes, value);
    }
}
```

Since the names of the athletes are stored in the data set as a capital letters, there's no need to make all characters lower case or upper case.

Pseudo code / relaxed explanation

for(Map<athletes name, athletes name >):
    if: tweet has athlete name
        emit(Map.get(athletes name), value)

Setup()
Map.put(athlete name, athlete name)

Results

Figure 3a shows the top 30 athletes in the small dataset that were mentioned in a tweet. Michael Phelps was the most mentioned name, given that most tweets where from the states and he himself is an American it seems plausible. Michael Phelps won 5 gold medals in Rio. This shows a strong correlation between #usa, #gold, #swimming and the most mentioned, Michael Phelps *(See MichaelWiki 2017).*

Number	Athlete	Frequency			
1	Michael Phelps	178286	**16**	Liliyana Natsir	19666
2	Usain Bolt	166471	**17**	Wayde van Niekerk	18194
3	Neymar	98094	**18**	Penny Oleksiak	17266
4	Simone Biles	77093	**19**	Monica Puig	16003
5	William	50510	**20**	Rafael Nadal	15530
6	Ryan Lochte	40363	**21**	Laura Trott	15367
7	Katie Ledecky	37582	**22**	Ruth Beitia	13487
8	Yulimar Rojas	33611	**23**	Teddy Riner	13351
9	Joseph Schooling	25295	**24**	Lilly King	13055
10	Sakshi Malik	24565	**25**	Shaunae Miller	12094
11	Simone Manuel	23234	**26**	Jason Kenny	11575
12	Rafaela Silva	22576	**27**	Allyson Felix	10912
13	Andy Murray	21086	**28**	Caster Semenya	10799
14	Kevin Durant	20947	**29**	Almaz Ayana	10490
15	Tontowi Ahmad	20187	**30**	Elaine Thompson	10385

Figure 3a – top 30 athlete medalists

PART C (ii) – Support Analysis

Assumption:

1. Get all sports via the mentions of tweeters
2. Tweets are filtered

Exercise

Instead of one java collection framework being used, we have two. The athletesName is an arrayList that adds all athletes name. The athletesSport is a Map that adds athletes name as its key and sports as its value.

```
if (fields.length == 11) {
    this.athletesName.add(fields[1]);
    this.athletesSport.put(fields[1], fields[7]);
}
```

The for each loop iterates through the athletesName. The tweets are compared by the athlete's name. The Map athletesSport's value (sports) is emitted to the reducer if the key of athletesSport matches the athlete's name. The reducer will then sum up the frequencies of sports.

```
for (String athlete : athletesName) {
    if (arrList[2].contains(athlete)) {
        allSports.set(athletesSport.get(athlete));
        context.write(allSports, value);
    }
}
```

A noteworthy mention; in part 3i, a Hashtable was used, but in 3ii a HashMap was used. The reason for this is change is that a Hashtable takes slightly longer time to process than a HashMap.

Results

Figure 3b shows the results of all sports aggregated from users mentioning athletes.
For example Mo Farah and Usain Bolt mentions both contribute to athletics.

Number	Athlete	Frequency			
1	athletics	443018	**11**	shooting	22700
2	aquatics	436922	**12**	canoe	22604
3	football	201956	**13**	sailing	22376
4	gymnastics	124207	**14**	weightlifting	22095
5	judo	95058	**15**	equestrian	21797
6	tennis	77380	**16**	boxing	20350
7	basketball	71675	**17**	volleyball	16783
8	cycling	64557	**18**	rowing	15810
9	badminton	60372	**19**	taekwondo	15325
10	wrestling	33358	**20**	fencing	11943

Figure 3b – top 20 sports

Pseudo code / relaxed explanation

for(ArrayList<athlete name>):
    if: tweet has athlete name

emit(Map.get(athlete name), value) //what's being emitted is the sport

Setup()
ArrayList.add(athlete name)
Map.add(athlete name, sport)

Bibliography

Epoch Converter (2017). *Unix Timestamp Converter* [online] Epoch Converter. Available at: https://www.epochconverter.com/ [Accessed 1 Nov. 2017].

RegExr. (2017). *Learn, Build, & Test RegEx* [online] RegExr.com. Available at: https://regexr.com/ [Accessed 1 Nov. 2017].

Matcher (2017). *Matcher (Java Platform SE 7)* [online] Available at: https://docs.oracle.com/javase/7/docs/api/java/util/regex/Matcher.html. [Accessed 1 Nov. 2017].

Tutorials Jenkov (2017). Jenkov [online] Available at: http://tutorials.jenkov.com/java-regex/matcher.html. [Accessed 1 Nov. 2017].

*Michael*Wiki (2017). *Michael Phelps* [online] Available at: https://en.wikipedia.org/wiki/Michael_Phelps [Accessed 1 Nov. 2017].

Tweets (2017). *Stats* [online] Available at: https://www.statista.com/statistics/242606/number-of-active-twitter-users-in-selected-countries/ [Accessed 1 Nov. 2017].