CMP3749M Big Data Assessment Report

A logo of a university

Description automatically generated

Mathews Joy

[25186202@students.lincoln.ac.uk](mailto:25186202@students.lincoln.ac.uk)

JOY20779936

University of Lincoln

School of Computer Science

When finished when ever I mention a specific function e.g. “toPandas()” include a reference next to it of the link to that functions documentation.

**Task 1 – Analysis of Nuclear Plants dataset**

**Part 1.1**

In the dataset we analysed for task 1 ”nuclear\_plants\_small\_dataset.csv” we used the pyspark “.isNull()” function to confirm there was no missing data points in the dataset. This function “is used to check if the current expression is NULL/None or column contains a NULL/None value”1. In our scenario we identified that there was no missing values in our dataset, as a result no further action was required. In the case a column contained 1 or more empty values, it is important to identify it and take necessary steps to fix the missing values, this is to avoid skewing the results from analysis you might perform. First it is important to identify what type of missing data you have; “Missing completely at random (MCAR) data is just randomly missing, missing at random (MAR) meaning missing conditionally at random based on another observation or missing not at random (MNAR) which is missing as part of how it is collected (deliberately missing)”2. Based on the previous statement by identifying these types helps in choosing the correct method to handle the missing data. For instance, if the data is classified as MCAR, it might be best to simply exclude those cases from your dataset. Although simply deleting data is not recommend, but as the missing data is MCAR, removing the data would not bias the analysis performed. This is because the missingness is not dependent on observed or unobserved data. But if it’s MAR or MNAR, more sophisticated techniques might be needed to avoid bias in your analysis. These could include imputation, which involves filling in missing values with substituted values such as mean of the column with missing data. Other options include acceptance/ignore them, “this is the most conservative option involves accepting your missing data: you simply leave these cells blank. It’s best to do this when you believe you’re dealing with MCAR or MAR values”3. Although the data seemed quite clean and structured, upon inspection with the “.printSchema()” function (which displays the data frame columns and there datatypes), we can see the column names are inconsistent. For example, in the pressure sensor column names, they have a random space value in the column names, which the other columns do no have. Due to this inconsistent it is best to keep the column names in uniform so we used the “.replace()” function on each column name to check for random empty spaces in the column names and get remove them. This means in future analysis no unexpected errors will be through when trying to access specific columns.

**Part 1.2**

For this task we are asked to find the summary statistics (minimum, maximum, mean, and median) for each group of subjects (normal and abnormal), and then for each group plot the boxplots for each feature. We can first approach this task by converting the pyspark data frame to “pandas” data frame; “pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool” 4. The reason we convert to pandas is it allows cleaner syntax for performing aggregate functions in general and integrates with graphical visualisations easier, however these calculations can also be performed directly on the pyspark data frame. We can convert our current data frame to pandas via the “.toPandas()” function. Next, we can create a for loop that loop through the 2 statuses (normal and abornal) and finds the min,max mean and median using the “.describe()” function.

A computer screen shot of text

Description automatically generated

We then get the below result. Some key findings we can pick out are, the pressure sensors overall have the highest max values for both normal and abnormal status e.g. (56 for normal and 67 for abnormal). The mean values are all around the similar levels for all features across both normal and abnormal status, one exception being “vibration\_sensor\_3” (normal) displaying a higher overall mean of 19.

A screenshot of a computer

Description automatically generated

The next part of this task revolved around creating boxplots for all features for both Status groups. Benefit of using a boxplot is they are “very effective and easy to read, as they can summarize data from multiple sources and display the results in a single graph” 8. We initially format the data frame so that we can pass it into a seaborn “catplot which is a figure-level interface, which is a more flexible tool that can be used to draw various types of categorical plots, including boxplot” 9. We use the catplot and set the y axis as the features values and the x axis for the feature labels.

A screen shot of a computer program

Description automatically generated

We are then presented the chart in figure 4. Some key takeaways from figure 4 are we can see for 9 of the 12 features there are an abundance of outliers as indicated by the diamond points above and below the whiskers of the boxplots. This is something to consider as we might want to potentially remove these values as it could affect any further analysis or choose to keep them as they can provide valuable insight into the data. We can also highlight vibration sensors tend to have a greater range of data point compared to power and pressure sensors, only exception being pressure sensor 1.

A graph of different colored squares

Description automatically generated

Figure 4. Multi Boxplot comparing all features grouped by status.

**Part 1.3**

When you produce a correlation matrix of some data, you can identify highly correlated features by looking for pairs of features that have a high correlation coefficient. “A correlation coefficient of 1 indicates a perfect positive correlation, while a correlation coefficient of -1 indicates a perfect negative correlation. A correlation coefficient of 0 indicates no correlation between the two features” 5.

Highly correlated features can be problematic for further processing, such as data classification, because they can introduce multicollinearity, which can lead to overfitting and unstable models. One way to deal with highly correlated features is to remove one of the features from the dataset. Another way is to use feature selection algorithms to select a subset of features that are most relevant to the classification task. Principal component analysis (PCA) can also be used to reduce the dimensionality of the dataset by transforming the correlated features into a set of uncorrelated features 6.

In our scenario we have constructed a correlation matrix using the seaborn package and the “.heatmap()” function. We pass in our data frame for the input to the “.heatmap” function ensuring we drop the “Status” column as this is binary text column and would not bring much value, and then applying the “.corr()” function. “.corr() calculates the correlation matrix whose elements range is [-1, 1], by default it uses Pearson Correlation coefficient. sns.heatmap is just a way to display using colours how strong the correlations are, where a lighter colour cream indicates a positive correlation and a stronger darker colour black highlights a negative colleration.”7.

**A screenshot of a computer

Description automatically generated**After observing the correlation matrix, we can see that there are some features with high positive correlation with one another for example “Power\_range\_sensor\_4” and “Pressure\_sensor\_4” with a correlation of 0.82. In general features with high correlation like our example can increase the complexity of the model, and will pose no benefit, so it could be a beneficial idea to remove one of the features. On the contrary, features such as “Power\_sensor\_3” and “Pressure\_sensor\_1” have a correlation of -0.26, this is a moderate negative correlation and shows as one feature increases the other slightly decreases. So, this pair of features for example should be kept in the data frame for further processing and data insights, as they will bring a less likely chance of multicollinearity and overfitting.

Figure 5. Correlation matrix of features

**Task 2 - MapReduce for Margie Travel dataset**

**Task 2.1**

For the first task we have we to determine the number of flights from each airport in a table, and show a list of any unused airports. We first need to set up a new spark context for our map reduce code. We can do this by first stopping our task 1 context and setting up the environment requirements and a new spark context using “SparkContext()” and “SparkConf()” as shown in figure 6.

A screen shot of a computer code

Description automatically generated

Figure 6. New spark context creation

We can then load our 2 datasets for the our problem scenario including the passenger and airport data set using the “.textFile()” function on our spark context object to read in the dataset. The function will return a Resilient Distributed Dataset (RDD) of strings, one for each dataset we wanted to read. A RDD is essentially “a fault-tolerant, immutable distributed collections of objects” 10. With our 2 datasets now being in RDD format, it allows us to perform useful “MapReduce” functions. MapReduce refers to two separate and distinct tasks. “The first is the map job, which takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key/value pairs). The reduce job takes the output from a map as input and combines those data tuples into a smaller set of tuples” 11.

Remove any missing values in both passenger and airport datasets using the “.filter()” function on both datasets along with a lambda expression. The filter function allows us to filter the dataset based on a condition in our instance we want to check if each sample in the dataset is not “None” and not empty. We use lambda along with the filter function as it allows us to define a anonymous function in line, allowing cleaner more concise code. We also continue to use lambda expressions in the rest of our code and apply to function such as “.map()” which is discussed later. After removing any missing values we can split both datasets by comma in order to allow for indexing of the data. This is performed by applying the “.map()” function to both datasets and defining a lambda function that takes each sample in the dataset and splits it on “.”. The map function is used to apply a function to each element of an RDD and returns a new RDD with the results.

To find the number of flights from each airport and get a list of not used airports, we first get a list of all flights from the passenger dataset and also show the passenger count. We can first use the map function along with a lambda to get the flight id and airport along with a count for each passenger, then use the “reduceByKey()”, to reduce the data by adding up the values of each key. Then use the “sortBy()” in order to sort the data by the passenger count in descending format, returning a RDD named airport\_flight\_passenger\_count . Refer to figure 7.

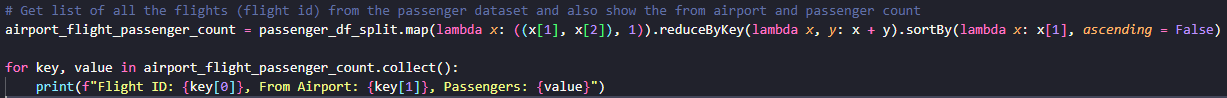


Figure 7. Code for getting list of all flights and passenger counts

Next get the number of flights from each airport. Once again using the map, reduceByKey and sortBy methods on the RDD air\_flight\_passenger\_count. This time we are passing in a tuple to the map function, to get only the airport code and flight count. Refer to figure 8.

**A screen shot of a computer

Description automatically generated**

Figure 8. Code for number of flights per airport

We then create 2 new RDD’s with the airport codes from the passenger and airport dataset, which we can then use the “union()” function on the airport and passenger codes RDD to merge both into 1 central RDD. Then apply the “.filter()” method to return airports with no departing flights by using the expression “lambda x:x[1] == 1” in the filter method, meaning the airport code only appears once in the RDD, so the airport is not used. Refer to figure 9.

A screen shot of a computer

Description automatically generated

Figure 9. Code for list of unused airports

**Task 2.2**

This task involved Creating a list of flights based on the Flight id; including number of passengers, relevant IATA/FAA codes, and departure and arrival times. We first approach this task by defining 2 custom functions that will allow us to work with the datetime problems of this task more effectively, refer to figure 10. The current departure time column in our RDD is in the Unix ‘epoch’ time format, however for this task we want the “HH:MM” format. We can define a function “convert\_unix\_time()”, that takes in a unix time and converts to our desired format, using the python time module. We also have no arrival time column in our dataset, instead departure time and the total flight time (in minutes) is provided, so we can get the arrival time by simply adding these 2 values. We can also define a function for this called “add\_time()”, that given a unix time and also a number of minutes to add, can calculate the final time by first converting the minutes to float (as the minutes in our dataframe is in string format originally) then multiply it by 60 to convert to unix time, adding our 2 unix times together, and finally using “our convert\_unix\_time” function to convert our final time to “HH:MM” format and return it.

**A computer screen shot of text

Description automatically generated**

Figure 10. Code definition for unix time manipulation

We can then apply the map function on our passenger RDD, and using lambda to map the flight id, departure and arrival airport codes, then use our convert\_unix\_time() method on the departure time column, to find the depatuhre time in HH:MM format, and finally pass in our departure time and total flight time in minutes to our add\_time() method and then pass a 1 to count the occurrence of passenger. We then use the reduceByKey() to add up the values for each key. Then sortBy() is used to sort the resulting RDD by the values (the counts from the reduce operation), in descending order. The flight list and data can then be viewed by apply a for loop for each key value in the RDD. Refer to figure 11.

**A screen shot of a computer

Description automatically generated**

Figure 11. Code for calculating and displaying list of flights based on the “Flight id”

**Task 2.3**

This task consisted of Calculating the line-of-sight (nautical) miles for each flight and the total travelled by each passenger, then output the passenger having earned the highest air miles. We initially define a function called “calculate\_distance()” that can help calculate the nautical miles given 2 pairs of latitude and longitude values. In the function to calculate the distance in nautical miles we need to define the radius of the earth in nautical miles, which is 3440.064 12 as of 30/11/2023. Then we convert our latitude and longitude values to radians to allow us to calculate the nautical distance. Then the “Harversine” formula is used to “determine the [great-circle distance](https://en.wikipedia.org/wiki/Great-circle_distance) between two points on a [sphere](https://en.wikipedia.org/wiki/Sphere) given their [longitudes](https://en.wikipedia.org/wiki/Longitude) and [latitudes](https://en.wikipedia.org/wiki/Latitude)” 13, which allow us to find the distance for each flight. Finally, the result from the haversine formula is multiplied by the earths radius we defined earlier to give the final distance in nautical miles. Refer to figure 12.

A computer screen shot of text

Description automatically generated

Figure 12. Code to calculate the nautical distance between 2 points

We then perform a series of map and keyBy() method on our airport and passenger datasets to get the passenger required features from each datatset including the flight id, passenger id, airport code, from airport code, latitude and longtidue values for each airport. We then use the “.join()” method, which is a form of inner join to join our 2 RDD based on a common key. Refer to figure 13.

A computer screen shot of a code

Description automatically generated

Figure 13. Code to get required features for task 2.3 from passenger and airport datasets and join them

Using the RDD created from the previously mentioned join, we can apply the map function and reduceByKey methods to map the flight id as the key and pass the airport arrival and departure location data to the custom function that calculates the nautical distance. We then sort by the nautical miles in descending order. This resulting RDD shows the nautical miles for each flight with the flight with the greatest nautical miles at the top. Refer to figure 14.

A screenshot of a computer

Description automatically generated

Figure 14. Code for calculating the nautical miles per flight id and displaying it in descending order

Then second part of task 2.3 was to show total travelled by each passenger, then output the passenger having earned the highest air miles. We can show the total travelled by each passenger by using the same RDD generated from the join method named “passenger\_airport\_data”, and instead of flight id like used in figure 13, we can apply the map function on passenger id, and sort by the nautical miles in descending order. Which we can now see the passenger with the highest nautical miles at the top, which is passenger id UESD9151GS5 with 131876 nautical miles in total (rounded to nearest whole number). Refer to figure 15.

A screenshot of a computer

Description automatically generated

Figure 15. Code for calculating the total nautical miles for each passenger id

**Task 3 - Big Data Tools and Technology Appraisal**

For task 1 pyspark was adopted as the tool of choice, and it served as an optimal tool for the task. Pyspark is a powerful tool for handling large-scale data processing. The benefit of using [PySpark “is it can scale from a single machine to thousands of nodes” 14, making it suitable for processing large-scale datasets](https://ieeexplore.ieee.org/document/10162348). In our scenario processing a dataset 996 rows, pyspark couldn’t show its full potential, however the code we used to solve task 1 can be easily used and be scaled if our dataset increased in size. [Compared to SQL for example, which generally runs on a single system, PySpark can handle big data workloads more effectively because of its scalability](https://ieeexplore.ieee.org/document/10162348). The limitations of pyspark in our task is that we are running the code on a local machine. As pyspark requires significant resources to run efficiently a local machine can limit the performance as the speed will be determined by the user’s personal hardware (“pyspark runs with as many worker threads as logical cores available on your machine”15). For our scenario with a smaller dataset, a tool such as pandas could have served a similar performance to pyspark and potentially a better user experience as pandas has a less steep learning curve. However, overall pyspark had an efficient and quick code execution, and no performance issues. In addition, pyspark had limited interaction with visualisations which was evident when attempting to create the correlation matrix for task 1, it was chosen to convert the pyspark data frame to pandas in the end, to allow a easier user experience. Pyspark is limited in its visualisation features and isn’t as feature rich as pandas. Pyspark only has access to basic charts however is a much better choice over pandas if the dataset cannot fit in memory, as this is where pandas struggle. Another key reason to utilise pyspark is it “integrates with a wide range of big data tools and technologies, including Hadoop, Hive, Cassandra, and HBase. This makes it easier to work with large datasets stored in distributed file systems and other big data stores” 16, and allows real-time data streaming, which cannot be used in tools such as pandas.

Regarding task 2 we opted to complete the given task with map reduce methods combined with pyspark. PySpark uses Resilient Distributed Datasets (RDDs) as its primary data structure. RDDs are “immutable distributed collections of objects” 17 that can be easily processed in parallel across a cluster of servers. RDDs are designed to be fault-tolerant, so that they can recover any node failures. However, pySpark’s implementation of MapReduce with RDDs has some restraints. First, RDDs are immutable, which means that they cannot be updated after creation. This is a problem when dealing with iterative algorithms that require updating the data, for our task this was not an issue as no iterative algorithms where involved. Secondly, RDDs can be slow when dealing with small datasets because of the overhead of distributed processing and setup. The scalability of MapReduce for handling large-scale data processing is one of its key strengths. MapReduce is designed to handle large datasets by distributing the data across a cluster of machines. This allows for parallel processing of the data, which can significantly improve performance.The performance boost of using RDD’s was not able to be captured in our specific problem case, due to the relatively small dataset size and the fact we are running the code on a local machine, instead for example on a cluster using Hadoop where we can scale our data across “hadoop clusters ranging from a few nodes to extremely large clusters with thousands of nodes” 18.In the future for, task 2 we could try to implement the “sc.parallelize()” 19 method to chunk the data to allow for parallelisation, however we didn’t use this as the dataset was not large enough in dimension and the setup up making the data ready to be chunked and be able to perform parallelisation on it isn’t worth the performance trade off in our particular case.

**References**

1. [PySpark isNull() & isNotNull() - Spark By {Examples} (sparkbyexamples.com)](https://sparkbyexamples.com/pyspark/pyspark-isnull/)
2. Week 3 big data workshop
3. [Missing Data | Types, Explanation, & Imputation (scribbr.co.uk)](https://www.scribbr.co.uk/stats/missing-values/)
4. <https://pandas.pydata.org/>
5. [How to Read a Correlation Matrix - Statology](https://www.statology.org/how-to-read-a-correlation-matrix/)
6. [linear regression - What we should do with highly correlated features? - Stack Overflow](https://stackoverflow.com/questions/65302136/what-we-should-do-with-highly-correlated-features)
7. <https://stackoverflow.com/questions/61281035/what-does-the-corr-method-do-in-pandas-and-how-does-it-relate-it-to-the-heatm>
8. <https://asq.org/quality-resources/box-whisker-plot>
9. <https://seaborn.pydata.org/generated/seaborn.catplot.html>
10. [PySpark RDD Tutorial | Learn with Examples - Spark By {Examples} (sparkbyexamples.com)](https://sparkbyexamples.com/pyspark-rdd/)
11. [What is Apache MapReduce? | IBM](https://www.ibm.com/topics/mapreduce)
12. [Earth radius to Nautical miles Conversion - Length Measurement | TrustConverter](https://trustconverter.com/en/length-conversion/earth-radius/earth-radius-to-nautical-miles.html#:~:text=1%20earth%20radius,%3D%203%2C440.0647948%20nautical%20miles)
13. [Haversine formula - Wikipedia](https://en.wikipedia.org/wiki/Haversine_formula)
14. [Essential PySpark for Scalable Data Analytics: A beginner's guide to harnessing the power and ease of PySpark 3 | Packt Publishing books | IEEE Xplore](https://ieeexplore.ieee.org/document/10162348)
15. [PySpark - Local system performance - Stack Overflow](https://stackoverflow.com/questions/61129686/pyspark-local-system-performance)
16. [PySpark vs Pandas: Performance, Memory Consumption and Use Cases - Code Conquest](https://www.codeconquest.com/blog/pyspark-vs-pandas-performance-memory-consumption-and-use-cases/)
17. <https://www.tutorialspoint.com/apache_spark/apache_spark_rdd.htm>
18. [Apache Hadoop 3.3.6 – Hadoop Cluster Setup](https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/ClusterSetup.html)
19. (Reference week 10 workshop)

GO through all screenshots and add figure subtitles with quick description maybe