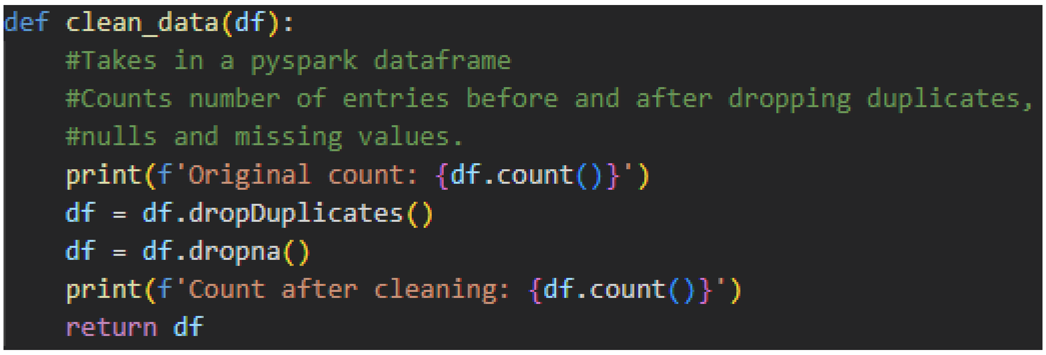
CMP3749M Big Data Assessment Item 1

1 Task 1 – PySpark Analysis of Nuclear Plants dataset

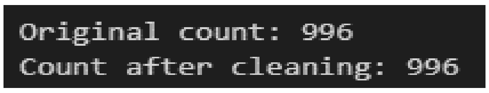
1.1 Part 1

Firstly, the required libraries and modules were imported, and a Spark session was created. A function named ‘clean_data’ was created, which takes in a data frame for cleaning. It checks for duplicate rows or missing/null values and drops the rows on which they occur, as shown in Figure 1. The dataset was read into a Spark data frame and then cleaned using the aforementioned function. It was found that there were no duplicate or missing/null values in the dataset, with the number of samples being 996, before and after cleaning as shown in Figure 2.



```
def clean_data(df):
    #Takes in a pyspark dataframe
    #Counts number of entries before and after dropping duplicates,
    #nulls and missing values.
    print(f'Original count: {df.count()}')
    df = df.dropDuplicates()
    df = df.dropna()
    print(f'Count after cleaning: {df.count()}')
    return df
```

Figure 1. Code for ’clean_data’ function.



```
Original count: 996
Count after cleaning: 996
```

Figure 2. Showing the number of dataset samples before and after cleaning.

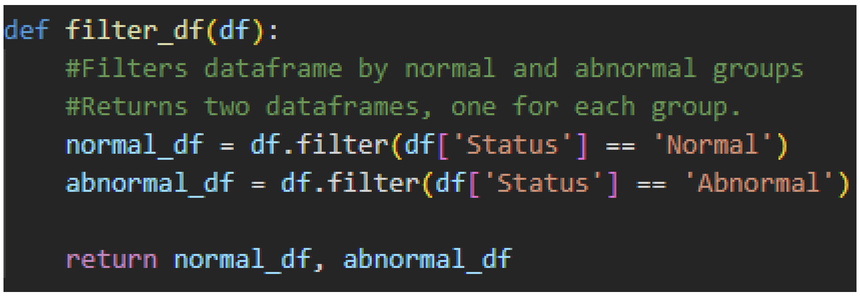
Incomplete datasets can have negative effects on the reliability of ML predictions. Dropping rows with missing/null or duplicated values is only one way of handling this task. Other methods, such as imputing data or data prediction using machine learning (ML), could have been used in the analysis of the dataset. Imputation refers to filling missing values with substituted data (Li et al., 2015). Simple strategies include replacing the values with the mean, median, or mode of that column/row. However, while this method is easy to implement, it can introduce bias into the applications of the data (Jäger et al., 2021).

More sophisticated ML methods, such as k-nearest neighbours, can be used for imputation. These methods can predict qualitative data, such as the ‘Status’ column in the nuclear plant's dataset, and quantitative data by changing the parameter of the distance metric. This method can be slow, as it must go through the whole dataset (Batista and Monard, 2003). This can be a huge drawback when dealing with billions of rows and columns within big datasets, due to the computational cost of processing the entire dataset.

1.2 Part 2

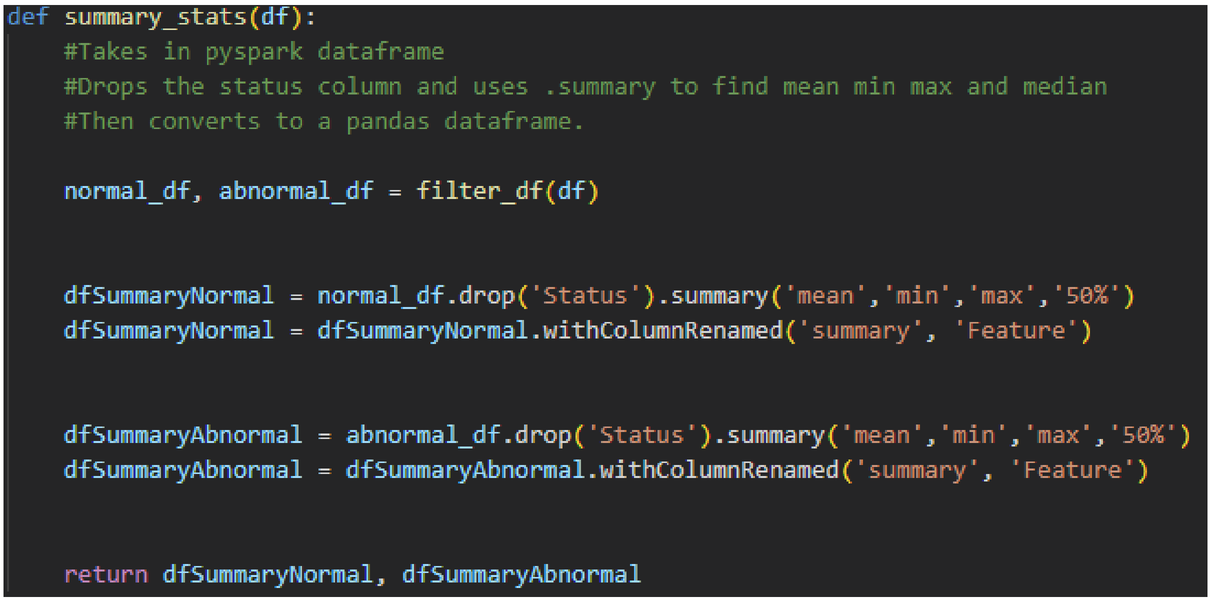
Two functions were created to complete the task. A function named ‘filter_df’ takes in a data frame and filters it based on the ‘Status’ column within the data frame, shown in Figure 3.

Another function, ‘summary_stats’, shown in Figure 4, takes in a data frame and uses the previously mentioned function to filter the ‘normal’ and ‘abnormal’ statuses, calculating the summary statistics using a built-in PySpark function. There is a large difference in mean for ‘vibration_sensor_3’, with the ‘Normal’ value being ~19.44 and ‘Abnormal’ being ~10.94, shown in Figures 5 and 6. Finally, a function named ‘boxplot_df’, shown in Figure. 7, takes in the data frame and filters it using the ‘filter_df’ function, creating two boxplots, one for each group: ‘normal’ and ‘abnormal’.



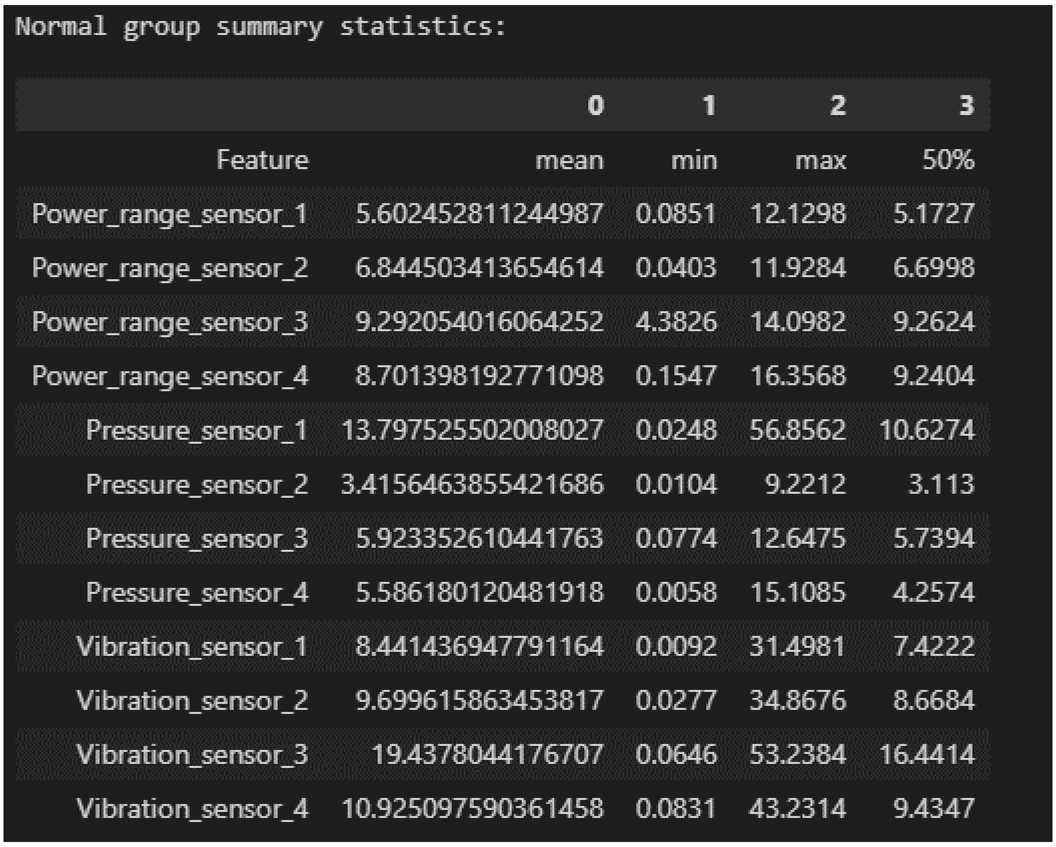
```python
def filter_df(df):
    #Filters dataframe by normal and abnormal groups
    #Returns two dataframes, one for each group.
    normal_df = df.filter(df['Status'] == 'Normal')
    abnormal_df = df.filter(df['Status'] == 'Abnormal')

    return normal_df, abnormal_df
```

Figure 3. Code showing ‘filter_df’ function.



```python
def summary_stats(df):
    #Takes in pyspark dataframe
    #Drops the status column and uses .summary to find mean min max and median
    #Then converts to a pandas dataframe.

    normal_df, abnormal_df = filter_df(df)


    dfSummaryNormal = normal_df.drop('Status').summary('mean','min','max','50%')
    dfSummaryNormal = dfSummaryNormal.withColumnRenamed('summary', 'Feature')


    dfSummaryAbnormal = abnormal_df.drop('Status').summary('mean','min','max','50%')
    dfSummaryAbnormal = dfSummaryAbnormal.withColumnRenamed('summary', 'Feature')


    return dfSummaryNormal, dfSummaryAbnormal
```

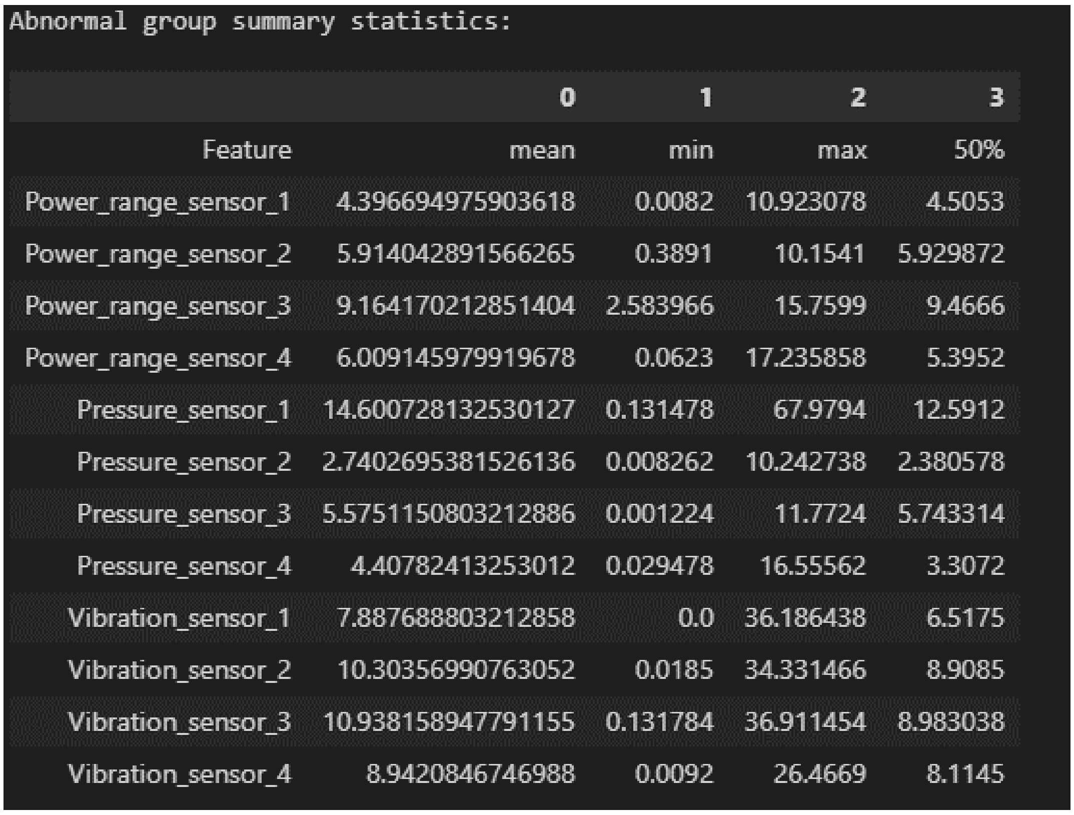
Figure 4. Code showing the 'summary_stats' function.



```
Normal group summary statistics:
```

	0	1	2	3
Feature	mean	min	max	50%
Power_range_sensor_1	5.602452811244987	0.0851	12.1298	5.1727
Power_range_sensor_2	6.844503413654614	0.0403	11.9284	6.6998
Power_range_sensor_3	9.292054016064252	4.3826	14.0982	9.2624
Power_range_sensor_4	8.701398192771098	0.1547	16.3568	9.2404
Pressure_sensor_1	13.797525502008027	0.0248	56.8562	10.6274
Pressure_sensor_2	3.4156463855421686	0.0104	9.2212	3.113
Pressure_sensor_3	5.923352610441763	0.0774	12.6475	5.7394
Pressure_sensor_4	5.586180120481918	0.0058	15.1085	4.2574
Vibration_sensor_1	8.441436947791164	0.0092	31.4981	7.4222
Vibration_sensor_2	9.699615863453817	0.0277	34.8676	8.6684
Vibration_sensor_3	19.4378044176707	0.0646	53.2384	16.4414
Vibration_sensor_4	10.925097590361458	0.0831	43.2314	9.4347

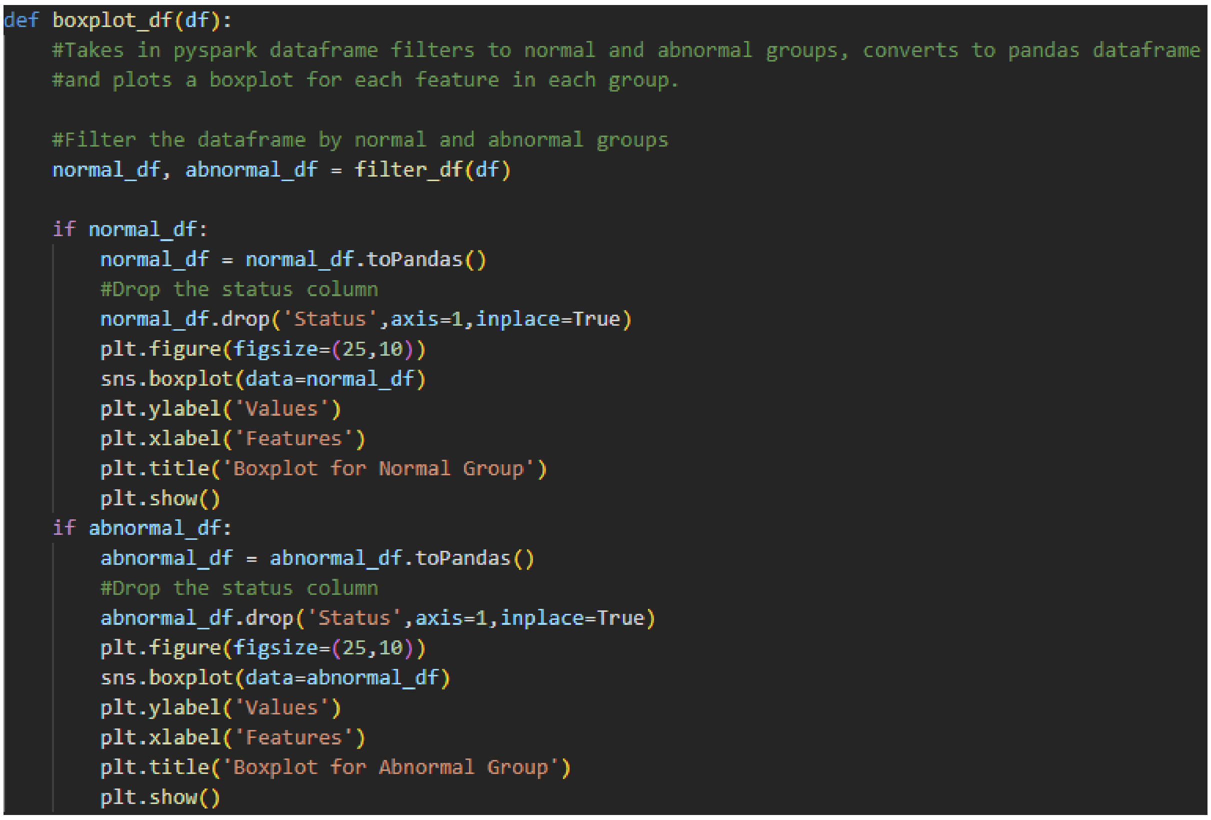
Figure 5. Summary statistics for the 'Normal' group.



```
Abnormal group summary statistics:
```

	0	1	2	3
Feature	mean	min	max	50%
Power_range_sensor_1	4.396694975903618	0.0082	10.923078	4.5053
Power_range_sensor_2	5.914042891566265	0.3891	10.1541	5.929872
Power_range_sensor_3	9.164170212851404	2.583966	15.7599	9.4666
Power_range_sensor_4	6.009145979919678	0.0623	17.235858	5.3952
Pressure_sensor_1	14.600728132530127	0.131478	67.9794	12.5912
Pressure_sensor_2	2.7402695381526136	0.008262	10.242738	2.380578
Pressure_sensor_3	5.5751150803212886	0.001224	11.7724	5.743314
Pressure_sensor_4	4.40782413253012	0.029478	16.55562	3.3072
Vibration_sensor_1	7.887688803212858	0.0	36.186438	6.5175
Vibration_sensor_2	10.30356990763052	0.0185	34.331466	8.9085
Vibration_sensor_3	10.938158947791155	0.131784	36.911454	8.983038
Vibration_sensor_4	8.9420846746988	0.0092	26.4669	8.1145

Figure 6. Summary statistics for the 'Abnormal' group.



```python
def boxplot_df(df):
    #Takes in pyspark dataframe filters to normal and abnormal groups, converts to pandas dataframe
    #and plots a boxplot for each feature in each group.

    #Filter the dataframe by normal and abnormal groups
    normal_df, abnormal_df = filter_df(df)

    if normal_df:
        normal_df = normal_df.toPandas()
        #Drop the status column
        normal_df.drop('Status',axis=1,inplace=True)
        plt.figure(figsize=(25,10))
        sns.boxplot(data=normal_df)
        plt.ylabel('Values')
        plt.xlabel('Features')
        plt.title('Boxplot for Normal Group')
        plt.show()
    if abnormal_df:
        abnormal_df = abnormal_df.toPandas()
        #Drop the status column
        abnormal_df.drop('Status',axis=1,inplace=True)
        plt.figure(figsize=(25,10))
        sns.boxplot(data=abnormal_df)
        plt.ylabel('Values')
        plt.xlabel('Features')
        plt.title('Boxplot for Abnormal Group')
        plt.show()
```

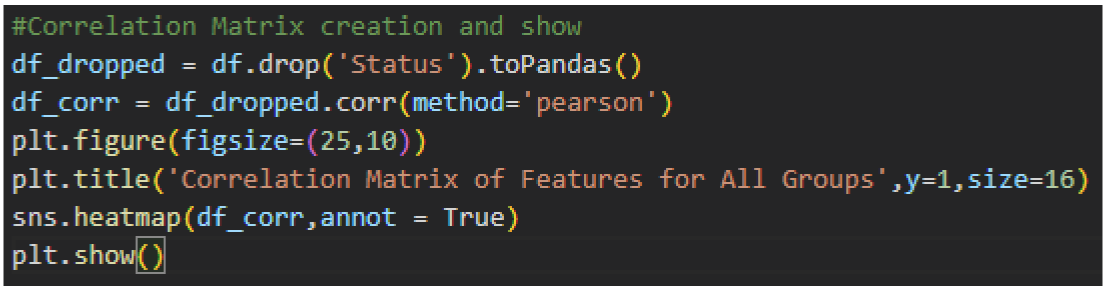
Figure 7. Code for 'boxplot_df' function.

Both boxplots give a good visual analysis of the data, showing outliers and clear differences in the values for 'vibration_sensor_3 for each group, "Normal" and "abnormal", possibly being the reason for the status changing. The outliers were not removed from the dataset as it was not required for the tasks but there are various methods to do so. Trimming, which removes the row where the outlier appears and winsorization which sets extreme values of data to a designated percentile within the dataset (Kwak and Kim, 2017). Trimming provides a simple implementation but leads to loss of information and can introduce bias whilst winsorization preserves the sample size and is less sensitive to extreme values but can distort the original distribution.

1.3 Part 3

A correlation matrix was created using the pandas library and visualised using the seaborn library, shown in Figure. 8. Two features, 'pressure_sensor_4' and 'power_range_sensor_4' have a Pearson coefficient of 0.82. This indicates a high positive correlation between the two features and there may be redundancy in using both features for analysis as they could carry similar information about the target class, this could mean multicollinearity and one of the variables could be removed, with the dataset being analysed without it. Further calculations could be computed to determine the variance inflation factor (VIF) to determine the extent of multicollinearity with values between 5 and 10 (Kim, 2019), but cannot be used to determine which features are multicollinear. Dimensionality reduction could have also been used to combine the two features into one singular feature, but it is challenging to evaluate the impact of individual variables on multicollinearity (Kim, 2019).

Ultimately, the dataset should be tested using a suitable machine learning algorithm and its performance with each method to ensure the best method is chosen, in this case, keeping the features could be good as they may contain unique information about other features that is useful for predicting the reactor status.



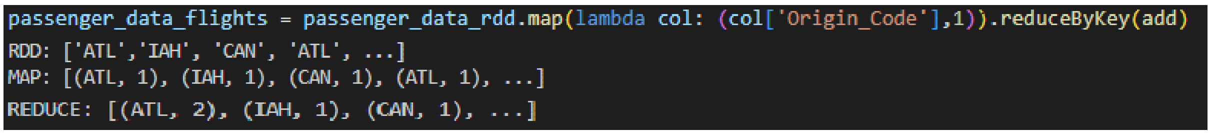
```
#Correlation Matrix creation and show
df_dropped = df.drop('Status').toPandas()
df_corr = df_dropped.corr(method='pearson')
plt.figure(figsize=(25,10))
plt.title('Correlation Matrix of Features for All Groups',y=1,size=16)
sns.heatmap(df_corr,annot = True)
plt.show()
```

Figure 8. Code to create a correlation matrix for the whole dataset.

2 Task 2 – MapReduce for Margie Travel dataset

2.1 Part 1

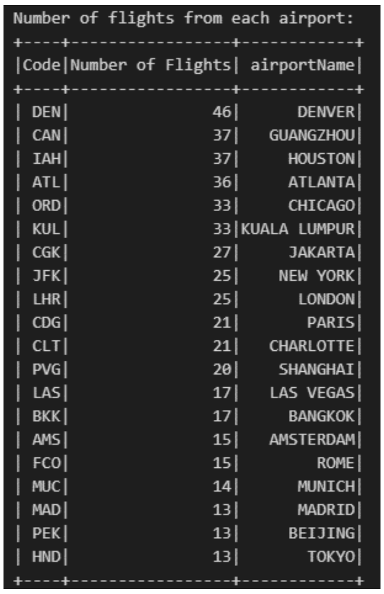
Initially, a 'SparkContext' is created which is used to work with resilient distributed datasets (RDD), both datasets were loaded into spark data frames with their columns renamed. The 'passenger_data' data frame is converted into an RDD so map and reduce functions can be used to count the number of flights from each airport. Using the 'map' function, each element in the 'Origin_Code' column is given a key-value pair, where the key is the airport code, and the value is 1. It is then reduced using the 'reduceByKey' function, and 'add' operator, which groups the key-value pairs by airport code and sums up the values for each key, counting the number of occurrences of each airport, thus the number of flights. An example is shown below in Figure 9.



```
passenger_data_flights = passenger_data_rdd.map(lambda col: (col['Origin_Code'],1)).reduceByKey(add)
RDD: ['ATL','IAH', 'CAN', 'ATL', ...]
MAP: [(ATL, 1), (IAH, 1), (CAN, 1), (ATL, 1), ...]
REDUCE: [(ATL, 2), (IAH, 1), (CAN, 1), ...]
```

Figure 9. Map Reduce Example

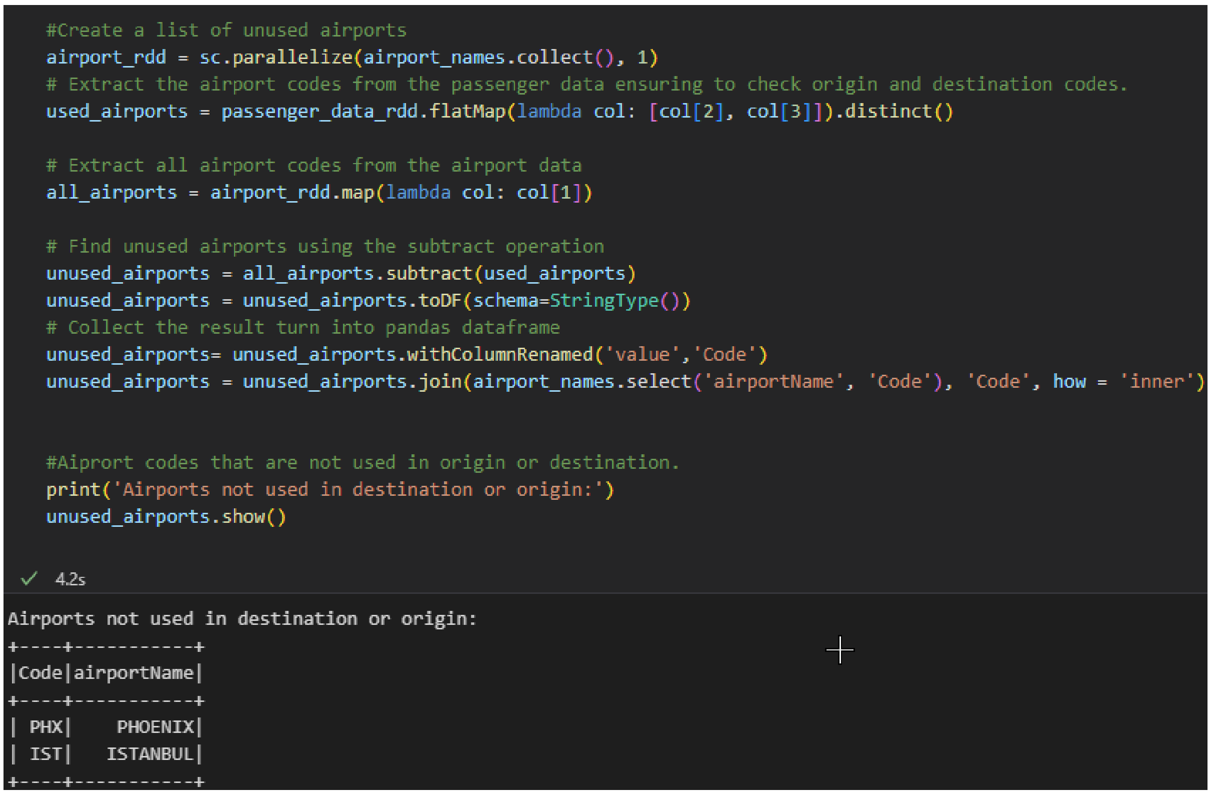
These functions are done in parallel, being sorted by the number of flights and shown to the user. The results for the task are shown in Figure 10.



```
Number of flights from each airport:
+----+-----------------+------------+
|Code|Number of Flights| airportName|
+----+-----------------+------------+
| DEN|               46|      DENVER|
| CAN|               37|   GUANGZHOU|
| IAH|               37|     HOUSTON|
| ATL|               36|     ATLANTA|
| ORD|               33|     CHICAGO|
| KUL|               33|KUALA LUMPUR|
| CGK|               27|     JAKARTA|
| JFK|               25|    NEW YORK|
| LHR|               25|      LONDON|
| CDG|               21|       PARIS|
| CLT|               21|   CHARLOTTE|
| PVG|               20|    SHANGHAI|
| LAS|               17|   LAS VEGAS|
| BKK|               17|     BANGKOK|
| AMS|               15|   AMSTERDAM|
| FCO|               15|        ROME|
| MUC|               14|      MUNICH|
| MAD|               13|      MADRID|
| PEK|               13|     BEIJING|
| HND|               13|       TOKYO|
+----+-----------------+------------+
```

Figure 10. Results for the number of flights from each airport.

The unused airports were computed using two different methods, using airports not used as an origin and airports not used as an origin or destination. The origin and destination codes were flat mapped, getting the distinct codes, it was subtracted from the airport RDD which had the airport code column mapped to obtain the airports that were not used in origin and destination, the results are shown in Figure. 11.



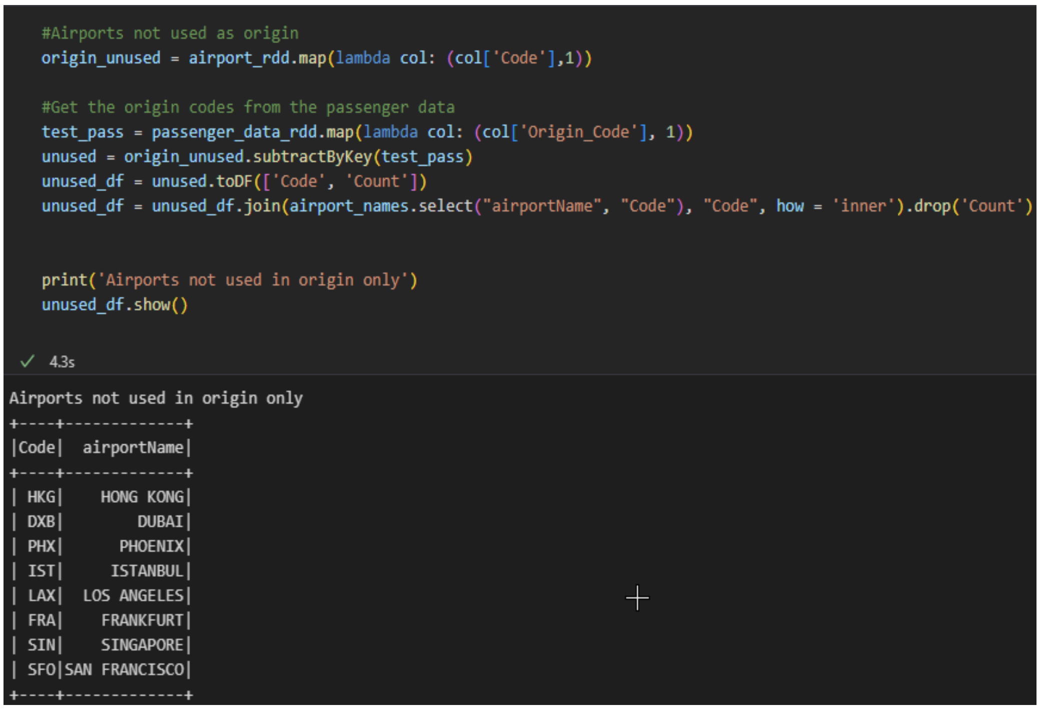
```
#Create a list of unused airports
airport_rdd = sc.parallelize(airport_names.collect(), 1)
# Extract the airport codes from the passenger data ensuring to check origin and destination codes.
used_airports = passenger_data_rdd.flatMap(lambda col: [col[2], col[3]]).distinct()

# Extract all airport codes from the airport data
all_airports = airport_rdd.map(lambda col: col[1])

# Find unused airports using the subtract operation
unused_airports = all_airports.subtract(used_airports)
unused_airports = unused_airports.toDF(schema=StringType())
# Collect the result turn into pandas dataframe
unused_airports= unused_airports.withColumnRenamed('value','Code')
unused_airports = unused_airports.join(airport_names.select('airportName', 'Code'), 'Code', how = 'inner')


#Aiprort codes that are not used in origin or destination.
print('Airports not used in destination or origin:')
unused_airports.show()
```

✓ 4.2s

```
Airports not used in destination or origin:
+----+-----------+
|Code|airportName|
+----+-----------+
| PHX|    PHOENIX|
| IST|   ISTANBUL|
+----+-----------+
```

Figure 11. Code and results for unused destination and origin airports.

For the airports not used as an origin, the code was mapped using the already existing airport rdd, with the code column from the passenger rdd mapped, which then has the function ‘subtractByKey’ used to gather the key-value pairs which have no matching pair. This is joined by the airport name from the original airport data frame, to show the airport code and name of those airports not being used which 7 different airports were not used to fly from, shown in Figure. 12.



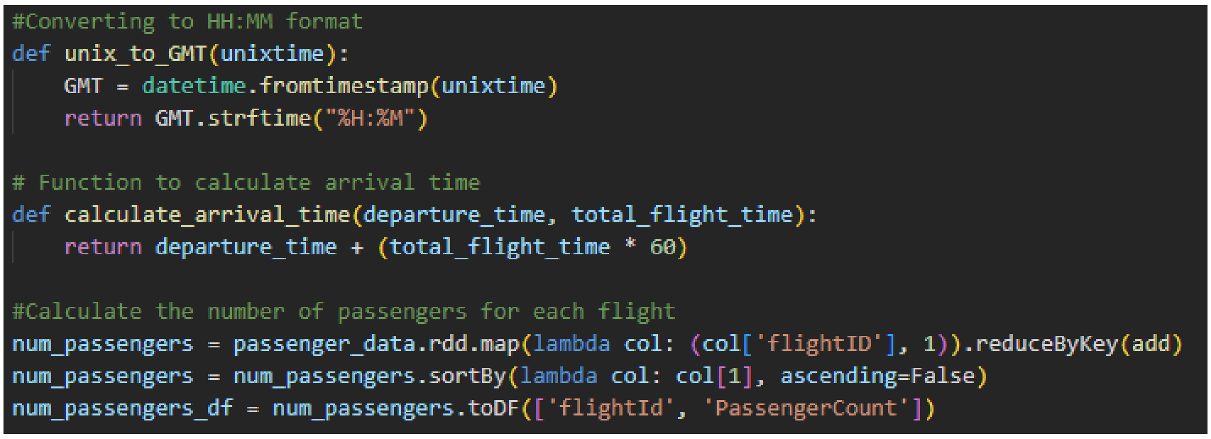
```
#Airports not used as origin
origin_unused = airport_rdd.map(lambda col: (col['Code'],1))

#Get the origin codes from the passenger data
test_pass = passenger_data_rdd.map(lambda col: (col['Origin_Code'], 1))
unused = origin_unused.subtractByKey(test_pass)
unused_df = unused.toDF(['Code', 'Count'])
unused_df = unused_df.join(airport_names.select("airportName", "Code"), "Code", how = 'inner').drop('Count')


print('Airports not used in origin only')
unused_df.show()

✓ 4.3s

Airports not used in origin only
+----+-------------+
|Code|  airportName|
+----+-------------+
| HKG|    HONG KONG|
| DXB|        DUBAI|
| PHX|      PHOENIX|
| IST|     ISTANBUL|
| LAX|  LOS ANGELES|
| FRA|    FRANKFURT|
| SIN|    SINGAPORE|
| SFO|SAN FRANCISCO|
+----+-------------+
```

Figure 12. Code and results for unused origin airports.

The implementation could have been improved by using the ‘Code’ from the number of flights data frame and using a left-anti join on the list of all airport names to find the set of unused airports. This would have reduced the number of newly declared variables used and reduced the number of times columns are mapped, possibly speeding up the runtime.

2.2 Part 2

Firstly, two functions were created, one to calculate GMT from the given Unix epoch time and another to calculate the arrival time based on the departure time and flight time. Using an RDD the ‘flightId’ column is mapped as a key with value 1, being reduced by the add operator, thus the number of flightId is the number of passengers on each aeroplane, which are stored in a separate data frame, shown in Figure. 13.

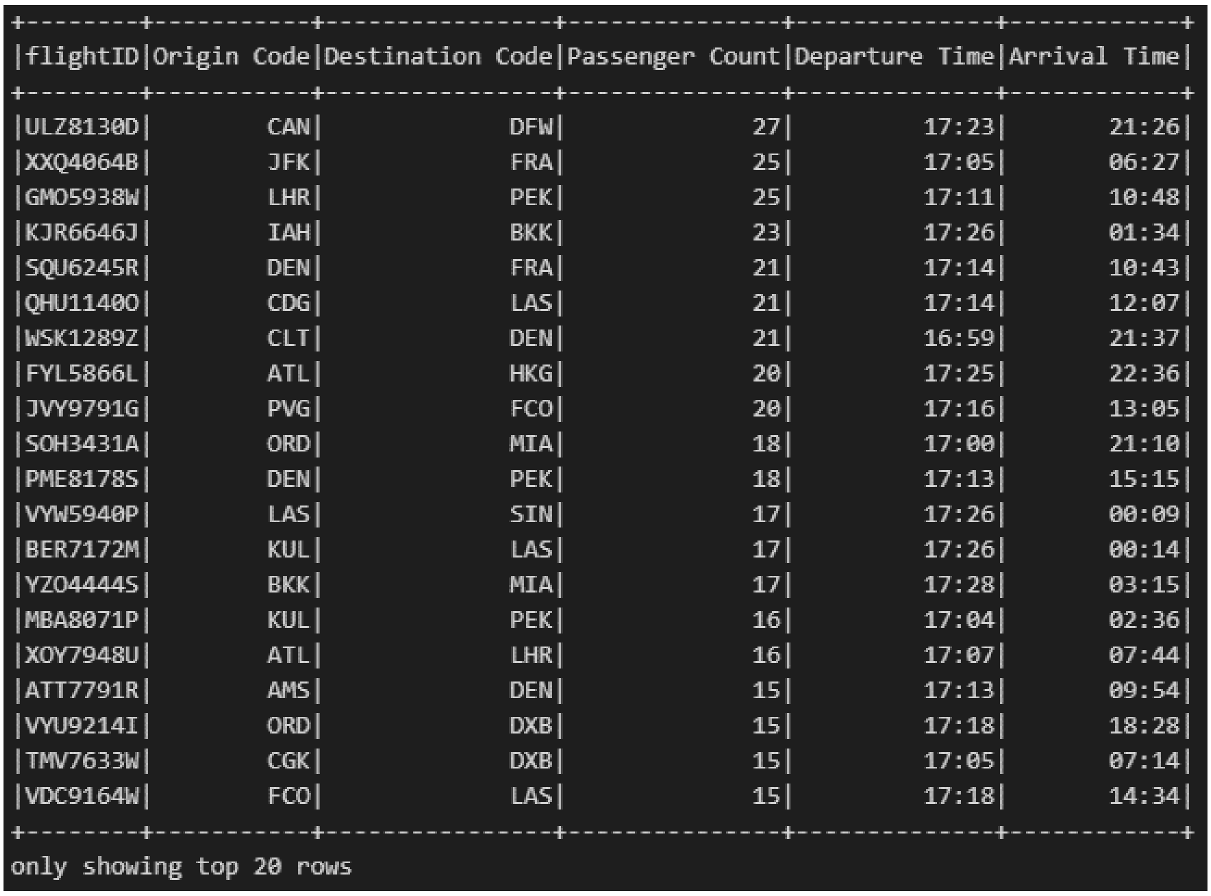


```
#Converting to HH:MM format
def unix_to_GMT(unixtime):
    GMT = datetime.fromtimestamp(unixtime)
    return GMT.strftime("%H:%M")

# Function to calculate arrival time
def calculate_arrival_time(departure_time, total_flight_time):
    return departure_time + (total_flight_time * 60)

#Calculate the number of passengers for each flight
num_passengers = passenger_data.rdd.map(lambda col: (col['flightID'], 1)).reduceByKey(add)
num_passengers = num_passengers.sortBy(lambda col: col[1], ascending=False)
num_passengers_df = num_passengers.toDF(['flightId', 'PassengerCount'])
```

Figure 13. Code showing the two functions for time conversion and MapReduce for the number of passengers on each flight.

Next, the original passenger data frame is grouped by ‘flightID’, origin code, and destination code, and aggregates using the first departure time and total flight time for each group. The “num_passengers_df” is joined with the aggregated data frame, which is converted to an

RDD, where the map function is applied to transform the data by using the previously defined functions on the ‘departureTime_GMT’ and ‘totalFlightTime_mins’ columns, being sorted by the number of passengers, finally being converted to a spark data frame, and shown. The results are shown in Figure. 14.



```
+--------+-----------+----------------+---------------+--------------+------------+
|flightID|Origin Code|Destination Code|Passenger Count|Departure Time|Arrival Time|
+--------+-----------+----------------+---------------+--------------+------------+
|ULZ8130D|        CAN|             DFW|             27|         17:23|       21:26|
|XXQ4064B|        JFK|             FRA|             25|         17:05|       06:27|
|GMO5938W|        LHR|             PEK|             25|         17:11|       10:48|
|KJR6646J|        IAH|             BKK|             23|         17:26|       01:34|
|SQU6245R|        DEN|             FRA|             21|         17:14|       10:43|
|QHU1140O|        CDG|             LAS|             21|         17:14|       12:07|
|WSK1289Z|        CLT|             DEN|             21|         16:59|       21:37|
|FYL5866L|        ATL|             HKG|             20|         17:25|       22:36|
|JVY9791G|        PVG|             FCO|             20|         17:16|       13:05|
|SOH3431A|        ORD|             MIA|             18|         17:00|       21:10|
|PME8178S|        DEN|             PEK|             18|         17:13|       15:15|
|VYW5940P|        LAS|             SIN|             17|         17:26|       00:09|
|BER7172M|        KUL|             LAS|             17|         17:26|       00:14|
|YZO4444S|        BKK|             MIA|             17|         17:28|       03:15|
|MBA8071P|        KUL|             PEK|             16|         17:04|       02:36|
|XOY7948U|        ATL|             LHR|             16|         17:07|       07:44|
|ATT7791R|        AMS|             DEN|             15|         17:13|       09:54|
|VYU9214I|        ORD|             DXB|             15|         17:18|       18:28|
|TMV7633W|        CGK|             DXB|             15|         17:05|       07:14|
|VDC9164W|        FCO|             LAS|             15|         17:18|       14:34|
+--------+-----------+----------------+---------------+--------------+------------+
only showing top 20 rows
```

Figure 14. Results for number of passengers on each flight with the departure and arrival time calculated.

The visualisation of the data frame could have been improved by converting it into a pandas data frame and printing the result to display all the data instead of a truncated result.

2.3 Part 3

The required functions from the ‘math’ library were imported to be used in the ‘nautical_miles’ function which utilises the haversine formula to calculate the line-of-sight/nautical miles. The function, takes in the latitudes and longitudes of both the origin and destination, using the formulas shown below.

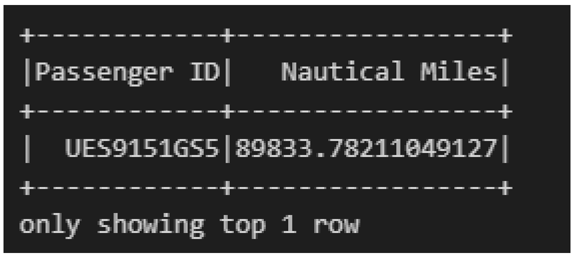
$$a = sin^2\left(\frac{\Delta\varphi}{2}\right) + cos\varphi_1 \times cos\varphi_2 \times sin^2(\frac{\Delta\lambda}{2})$$

$$c = 2\ \times atan2(\sqrt{a}, \sqrt{1-a})$$

$$d = R\ \times c$$

Where φ is latitude, λ is longitude, R is earth’s radius (mean radius = 6,371km), the angles are converted to radians, and the calculation is computed in kilometres and then converted to nautical miles.

The data is prepared with 3 data frames containing the ‘flightID’ and ‘Origin_Code’, another containing ‘flightID’ and ‘Destination_Code’, and the last containing the ‘Code’, ‘Latitude’ and ‘Longitude’ for the airport. The two data frames containing origin and destination codes are used to join their respective latitude and longitudes into two new data frames for calculation, where they are combined into one large data frame. This combined data frame is converted to an RDD where the ‘flightID’, ‘Origin_Code’, ‘Destination_Code’, and number of nautical miles are calculated using the latitude and longitudes of each origin and destination airport, thus each ‘flightID ‘now has a value of nautical miles corresponding to it. The combined data frame is then split, to map the ‘flightID’ and ‘Nautical Miles’ as a tuple, and then the ‘flightID’ and ‘PassengerID’ are mapped in a separate data frame for joining on ‘flightID’. Finally, the ‘PassengerID’ and ‘Nautical Miles’ are used as a key-value pair so the total nautical miles can be calculated by reducing using the add operator, being sorted by the number of nautical miles with the top passenger being shown in Figure. 15.



```
+-----------+----------------+
|Passenger ID|   Nautical Miles|
+-----------+----------------+
|  UES9151GS5|89833.78211049127|
+-----------+----------------+
only showing top 1 row
```

Figure 15. Top passenger for total nautical miles travelled.

3 Task 3 – Big Data Tools and Technology Appraisal

In this section, big data tools and techniques applied in Tasks 1 and 2 will be critically evaluated, reflecting on their effectiveness, limitations, and potential improvements. The scalability of PySpark and MapReduce for large-scale data processing will also be discussed.

Regarding Task 1, PySpark was effective in handling the small nuclear plant's dataset, effectively loading, and using built-in functions to drop missing or null values and provide statistics, however, due to the relatively small size of the dataset its distributed nature providing scalability for much larger datasets would not be fully utilised. While being effective for basic analysis, use of a simpler library with a less complex setup such as pandas could have been used and exploring more advanced machine learning techniques for imputation, or anomaly detection could provide more valuable information about the data if missing values were present.

Task 2, using MapReduce with RDDs, effectively handled basic data aggregation and manipulation, and if the dataset was large enough it could have handled size due to its ability to distribute components across multiple nodes and compute processes in parallel, accelerating computations (Dean and Ghemawat, 2008). The simplicity of the map-reduce code also could mean it’s easier to learn, however, its debugging process is extremely complicated due to the parallel errors in Java being written into the Python terminal, mostly being given as a ‘Py4J error’. Therefore, for more complex tasks MapReduce could be less intuitive compared to the Spark SQL which offers higher-level abstractions and clear error codes. In a real-world scenario such as with real-time flight analysis, Spark streaming within MapReduce could be more appropriate.

Both PySpark and MapReduce are evident in their ability to process big data with their distributed computing and parallel processing capabilities, however, in small dataset scenarios simpler tools may be more appropriate as they provide easier, and efficient solutions without the need for the overhead of setup. Their suitability depends on the specific needs and dataset size. PySpark is suited for more complex tasks with its high flexibility in operations and scalability whereas MapReduce is suited towards simpler big data tasks, where parallel processing and computational efficiency are priorities.

4 References

Batista, G.E.A.P.A. and Monard, M.C. (2003) An analysis of four missing data treatment methods for supervised learning. *Applied Artificial Intelligence*, 17(5-6) 519-533. Available from https://doi.org/10.1080/713827181 [accessed 06/01/2024].

Dean, J. and Ghemawat, S. (2008) MapReduce: Simplified Data Processing on Large Clusters. *Commun.ACM*, 51(1) 107–113. Available from https://doi.org/10.1145/1327452.1327492. [accessed 09/01/2024].

Jäger, S., Allhorn, A. and Bießmann, F. (2021) *A Benchmark for Data Imputation Methods.* Frontiers Media SA. Available from https://doi.org/10.3389/fdata.2021.693674 [accessed 06/01/2024].

Kim, J.H. (2019) Multicollinearity and misleading statistical results. *Korean journal of anesthesiology*, 72(6) 558-569. Available from https://doi.org/10.4097/kja.19087 [accessed 09/01/2024].

Kwak, S.K. and Kim, J.H. (2017) Statistical data preparation: management of missing values and outliers. *Korean journal of anesthesiology*, 70(4) 407-411. Available from https://doi.org/10.4097%2Fkjae.2017.70.4.407 [accessed 07/01/2024].

Li, P., Stuart, E.A. and Allison, D.B. (2015) Multiple Imputation: A Flexible Tool for Handling Missing Data. *JAMA*, 314(18) 1966-1967. Available from https://doi.org/10.1001/jama.2015.15281 [accessed 06/01/2024].