Module: Programming for Data Science

Name: Sim Kit Ling, Erika

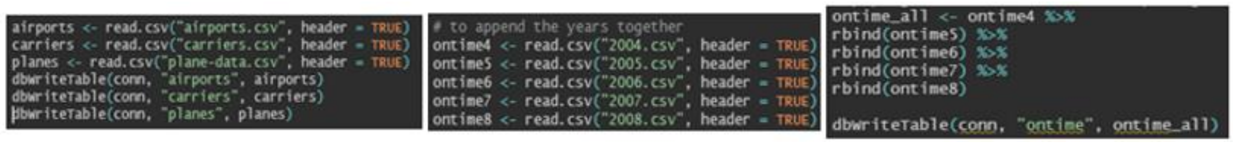
Student ID: 10229612

Date/Time: 23/02/2022 15:56 hours

Table of Contents

Raw Data

The years I have decided to extract the data are from 2004 to 2008 as I wanted the latest data for my project. When downloaded, all the data files were in BZ2 format, so I went to a website to convert it to a zip file. After which, I extracted and opened the csv file which allowed me to append the 5 excel files together and named it ontime. The rest of the files did not need to be appended as they stored variables to manipulate together with the ontime table as seen in Image 1.



```r
airports <- read.csv("airports.csv", header = TRUE)
carriers <- read.csv("carriers.csv", header = TRUE)
planes <- read.csv("plane-data.csv", header = TRUE)
dbwriteTable(conn, "airports", airports)
dbwriteTable(conn, "carriers", carriers)
dbwriteTable(conn, "planes", planes)
```

```r
# to append the years together
ontime4 <- read.csv("2004.csv", header = TRUE)
ontime5 <- read.csv("2005.csv", header = TRUE)
ontime6 <- read.csv("2006.csv", header = TRUE)
ontime7 <- read.csv("2007.csv", header = TRUE)
ontime8 <- read.csv("2008.csv", header = TRUE)
```

```r
ontime_all <- ontime4 %>%
rbind(ontime5) %>%
rbind(ontime6) %>%
rbind(ontime7) %>%
rbind(ontime8)

dbwriteTable(conn, "ontime", ontime_all)
```

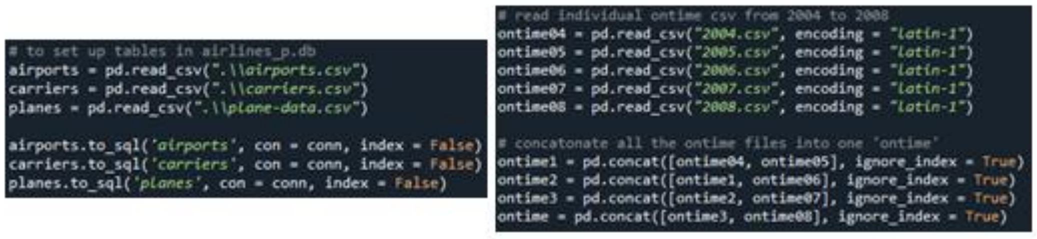
Image 1. Codes to append files together in R and writing them to database.

The content inside each of the files are the following:

i. Airports have everything on the airport's location, state, city, and IATA code.
ii. Carriers stores the airline's company name and unique code.
iii. Planes are about the plane's statistics: manufacturer, issue date, model etc.
iv. Ontime have everything about flights: origin, destination, time, day, week etc.

After appending and writing the files in R, I double checked that the files were uploaded correctly by checking the tables in database, getting the attributes from each table, and using DB Browser to ensure the tables are how it should be. Opening DB Browser allowed me to browse through the data with ease and to reference back to it when writing my code.

For Python, I created a brand-new database and re-coded the ways to append my csv files together. I used pd.read_csv(), pd.concat() to concatenate my 'ontime' files and to_sql() to write the tables into the second database 'airline_p.db' seen in Image 2.



```python
# to set up tables in airlines_p.db
airports = pd.read_csv(".\\airports.csv")
carriers = pd.read_csv(".\\carriers.csv")
planes = pd.read_csv(".\\plane-data.csv")

airports.to_sql('airports', con = conn, index = False)
carriers.to_sql('carriers', con = conn, index = False)
planes.to_sql('planes', con = conn, index = False)
```

```python
# read individual ontime csv from 2004 to 2008
ontime04 = pd.read_csv("2004.csv", encoding = "latin-1")
ontime05 = pd.read_csv("2005.csv", encoding = "latin-1")
ontime06 = pd.read_csv("2006.csv", encoding = "latin-1")
ontime07 = pd.read_csv("2007.csv", encoding = "latin-1")
ontime08 = pd.read_csv("2008.csv", encoding = "latin-1")

# concatonate all the ontime files into one 'ontime'
ontime1 = pd.concat([ontime04, ontime05], ignore_index = True)
ontime2 = pd.concat([ontime1, ontime06], ignore_index = True)
ontime3 = pd.concat([ontime2, ontime07], ignore_index = True)
ontime = pd.concat([ontime3, ontime08], ignore_index = True)
```

Image 2. Codes to append files together and sending to sqlite3 in Python.

Initial Setups

In the R script, I activated DBI using the library. My working directory was set up to the appropriate folder with my data stored by using the setwd() function. After which, I created a connection to the database named airlines_r.db. I had to activate dplyr as well to use some of its functions (piping) to assist me with certain situations such as appending the csv files with the years together into one.

Since I used DBI for my R codes, it was easier to bring over my codes to python to cross check my errors and code. The difference between Python and R is that I had to import my packages, os, sqlite3 and pandas. Changing of work directory is slightly different from R as well. Instead of using setwd(), I used os.chdir(). I activated the connection to my database by using conn.cursor() and attach it to a variable named 'c' to execute my commands for sqlite3.

Manipulating and Querying Databases in R and Python

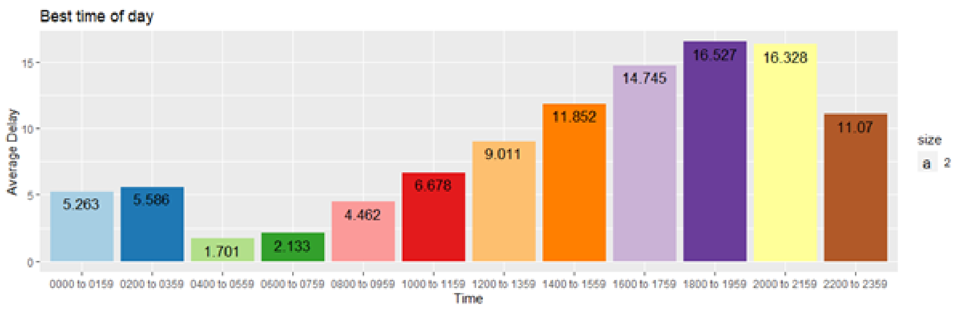
1. When is the best time of day, day of the week, and time of year to fly to minimise delays?

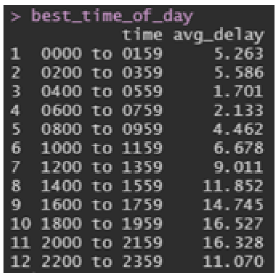
I decided to split this question into three different parts: best time of day, day of week and time of year. I used DepDelay as I felt most passengers will hate leaving late for their scheduled flight.

1.1 Using R

My assumption for best time of day was to get the time frame with the lowest average delay using DepDelay. The variables that I used were CRSDepTime and DepDelay from ontime table. CRSDepTime was to allow me to filter the time intervals by 2 hours when I used the SELECT CASE WHEN clause. I then calculated the average departure delay using DepDelay following the time intervals (0000 to 0159 etc). Using the results, I plotted a bar graph that allowed me to make an efficient comparison between the timings as seen in Image 3.

The graph shows me that best time of day to fly to minimize delays would be **at 0400 hours to 0559 hours** since it has the **lowest average departure delay of 1.701 minutes**.



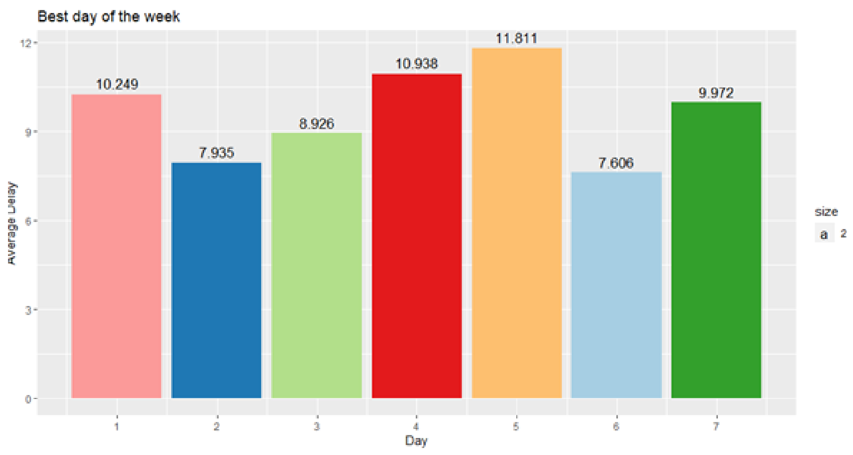


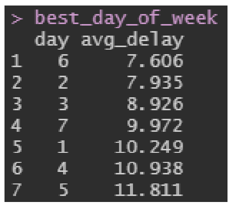
```
> best_time_of_day
           time avg_delay
1  0000 to 0159     5.263
2  0200 to 0359     5.586
3  0400 to 0559     1.701
4  0600 to 0759     2.133
5  0800 to 0959     4.462
6  1000 to 1159     6.678
7  1200 to 1359     9.011
8  1400 to 1559    11.852
9  1600 to 1759    14.745
10 1800 to 1959    16.527
11 2000 to 2159    16.328
12 2200 to 2359    11.070
```

Image 3. Best time of day.

Next, I made no assumptions for day of week. The variables I used were DayofWeek and DepDelay from ontime table. I calculated the total average delay of each day in a week using DepDelay and grouped it by day using the values in DayofWeek.

The output returned showed that **Day 6** is the best day of the week to fly with the **lowest average delay of 7.606 minutes** seen in Image 4.



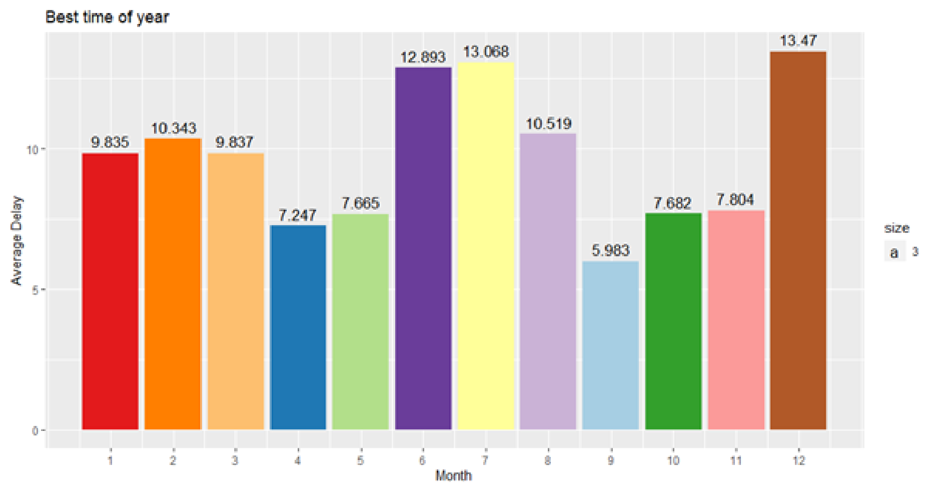


```
> best_day_of_week
  day avg_delay
1   6     7.606
2   2     7.935
3   3     8.926
4   7     9.972
5   1    10.249
6   4    10.938
7   5    11.811
```

Image 4. Best day of week.

For the last part of the question, best time of the year, my assumptions were the time frame it wanted to be was in months. The variables I used were Month and DepDelay from ontime table. The values in DepDelay were used to calculate the total average delay for each month. From there, grouping by months helped me to organize the average delay accordingly and I could analyze the data frame by each month instead of duplicate months in a data frame.

The result showed that **Month 9, which is September,** is the best time of year to fly to minimize delays with the **lowest average delay of 5.983 minutes** seen in Image 5.



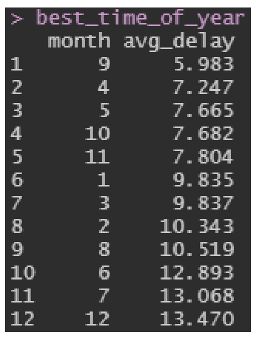
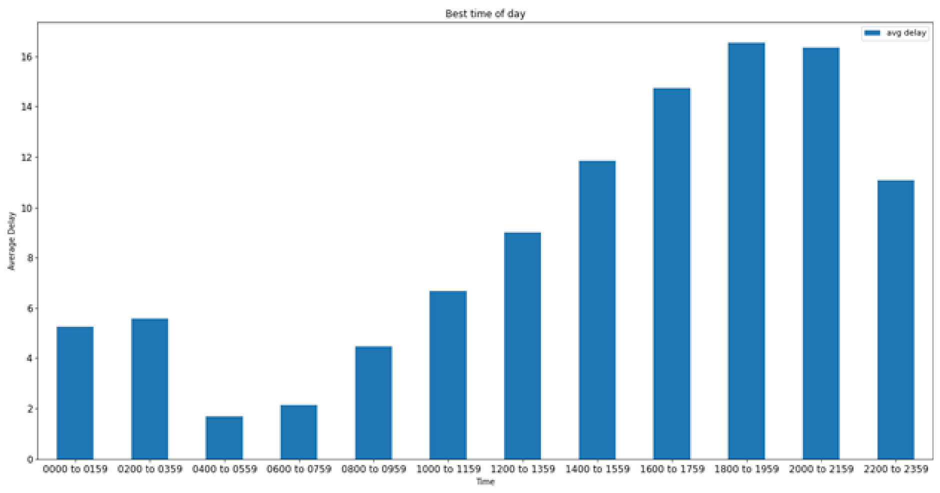
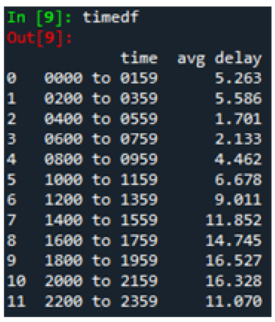

```
> best_time_of_year
   month avg_delay
1      9     5.983
2      4     7.247
3      5     7.665
4     10     7.682
5     11     7.804
6      1     9.835
7      3     9.837
8      2    10.343
9      8    10.519
10     6    12.893
11     7    13.068
12    12    13.470
```

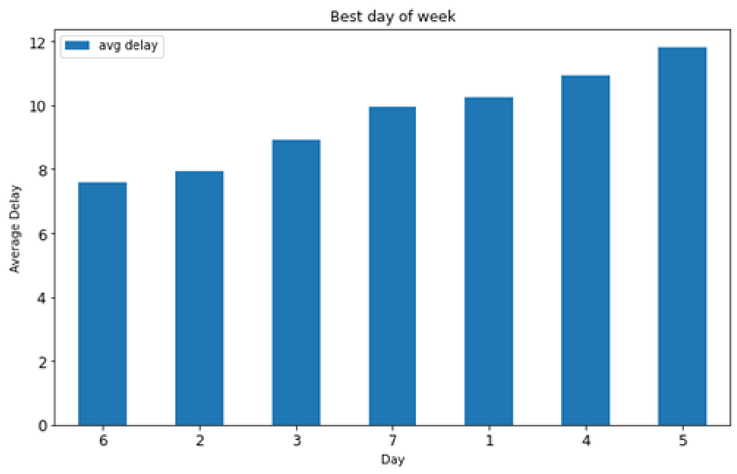
Image 5. Best time of year.

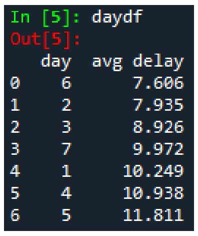
1.2 Using Python

My assumptions were the same as my R script for the 3 different parts. However, the command to execute the queries from the database in Python were different to what R used. Instead of dbGetQuery, Python uses the cursor followed by the execute() command to execute the queries from the database. At the end of it, I had to write fetchall() or there would not be any output shown. I used normal plot functions that was already provided in Python itself and plotted bar plots for the data frames. The output returned were the same that I had in the R shown in Image 6 together with their bar plots.

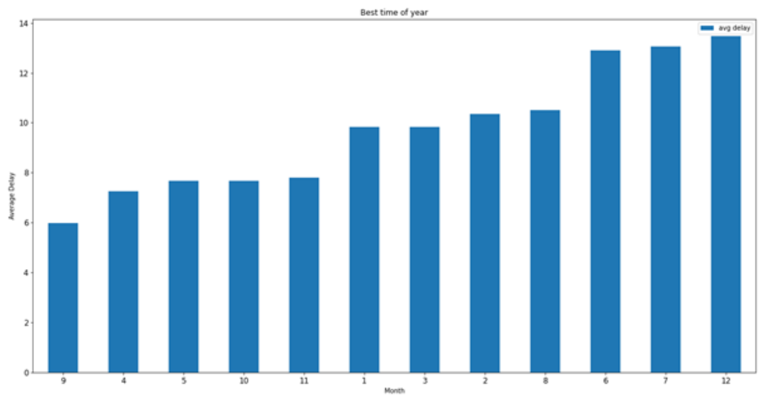


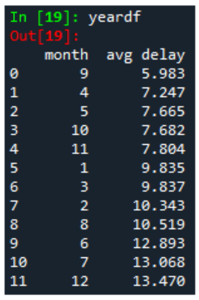

```
In [9]: timedf
Out[9]:
            time  avg delay
0   0000 to 0159      5.263
1   0200 to 0359      5.586
2   0400 to 0559      1.701
3   0600 to 0759      2.133
4   0800 to 0959      4.462
5   1000 to 1159      6.678
6   1200 to 1359      9.011
7   1400 to 1559     11.852
8   1600 to 1759     14.745
9   1800 to 1959     16.527
10  2000 to 2159     16.328
11  2200 to 2359     11.070
```





```
In [5]: daydf
Out[5]:
   day  avg delay
0    6      7.606
1    2      7.935
2    3      8.926
3    7      9.972
4    1     10.249
5    4     10.938
6    5     11.811
```





```
In [19]: yeardf
Out[19]:
    month  avg delay
0       9      5.983
1       4      7.247
2       5      7.665
3      10      7.682
4      11      7.804
5       1      9.835
6       3      9.837
7       2     10.343
8       8     10.519
9       6     12.893
10      7     13.068
11     12     13.470
```

Image 6. Bar plots and outputs for best time of day, day of week and time of year.

2. Do older planes suffer more delays?

My assumption was to compare the older and newer planes by their total average delay to see if older planes do suffer more delays.

2.1 Using R

I went to query the range of manufacturing years to find which years would be considered as older models. This data was extracted from year variable from planes table. I found the median by filtering the data frame and the result came out as 1983. Therefore, I assumed years before that were considered as older planes while those after were considered as newer planes.

I used the year and DepDelay variables to get the average delay on both planes. The planes and ontime tables were joined together using tailnum. As the tailnum had common values, the code could combine the data from the two tables into one data frame. There were empty rows and anomaly in the data frames, so I omitted them by using na.omit() which removes the rows with NA. From there, I rbind() both data frames to plot a line graph to see the trend between the older and newer planes shown in Image 7. I could visually see that there were lower delays among the older planes before 1983.

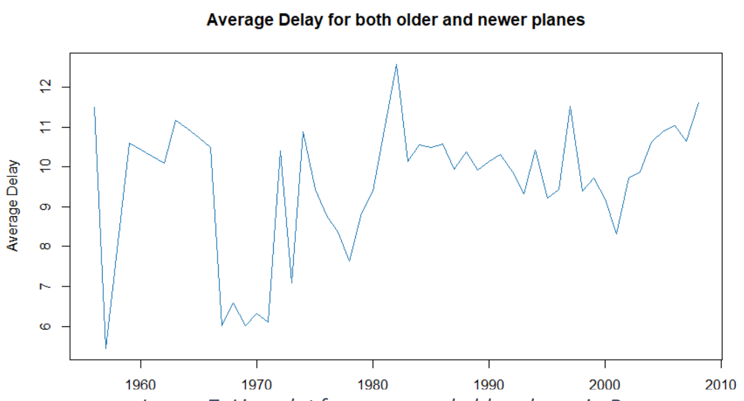
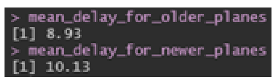
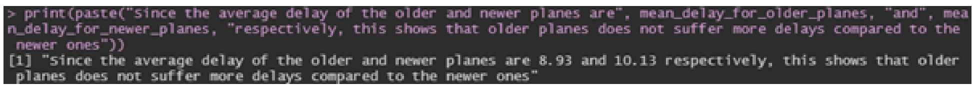


Image 7. Line plot for newer and older planes in R.

To prove my observation, I calculated the mean of the average delay for both older and newer planes by subsetting and putting na.rm = TRUE. NA had to be removed as the result would show NA as the output. After comparing the two mean values, I pasted my results and concluded that **older models do not suffer more delays** compared to newer models shown in Image 8.



```
> mean_delay_for_older_planes
[1] 8.93
> mean_delay_for_newer_planes
[1] 10.13
```

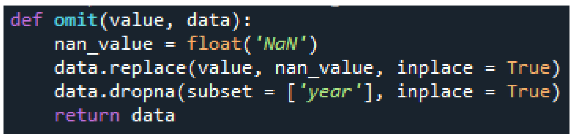


```
> print(paste("since the average delay of the older and newer planes are", mean_delay_for_older_planes, "and", mea
n_delay_for_newer_planes, "respectively, this shows that older planes does not suffer more delays compared to the
 newer ones"))
[1] "since the average delay of the older and newer planes are 8.93 and 10.13 respectively, this shows that older
 planes does not suffer more delays compared to the newer ones"
```

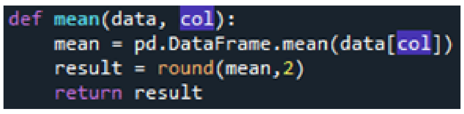
Image 8. Results for mean delay of older and newer planes for R.

2.2 Using Python

Similarly, the queries found were the same as R and I proceeded to get the output for Python. When the output of the query was shown, there was an invalid year in index 0. I made a function, omit(). It replaces the invalid row with NaN value and dropping it after that. I used the function to remove invalid rows for newer planes data frame as well. I also created another function to calculate the total mean in both older and newer planes to improve the efficiency of writing my codes. Both functions are seen in Image 9Image 9. The functions helped to reduce time wasted from copying and pasting the same code chunks. Instead, all I had to do was key in the values I wanted to omit or calculate, and the output would be given.



```
def omit(value, data):
    nan_value = float('NaN')
    data.replace(value, nan_value, inplace = True)
    data.dropna(subset = ['year'], inplace = True)
    return data
```



```
def mean(data, col):
    mean = pd.DataFrame.mean(data[col])
    result = round(mean,2)
    return result
```

Image 9. Omit and mean function I created.

I proceeded to plot the line graph for both newer and older planes as I did in R. The result is seen in Image 10. From here, I can also observe the same thing where the newer planes have a higher trend of more average delays from 1983 onwards.

Overall, I supported my observations by calculating the mean of both planes. The output was the same as R where it shows that **older planes do not suffer more delays** as the average departure delays of **8.93 minutes for older planes** was better than the average departure delays of **10.14 minutes for newer planes** shown in Image 11Image 11.

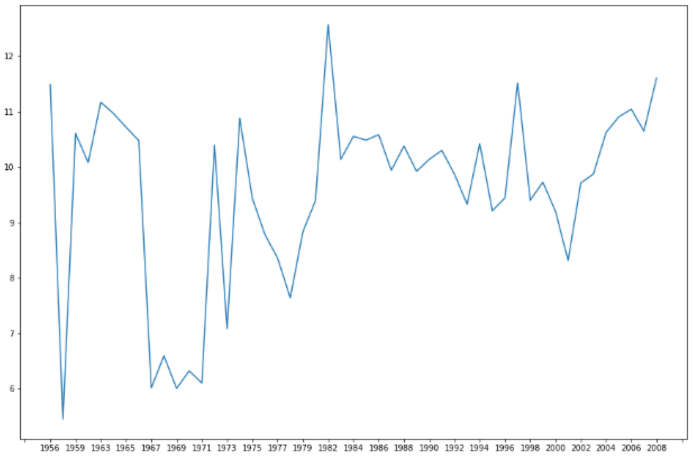
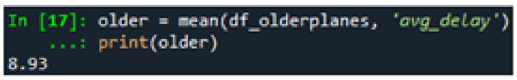
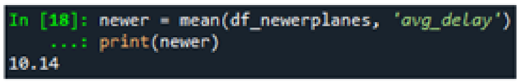


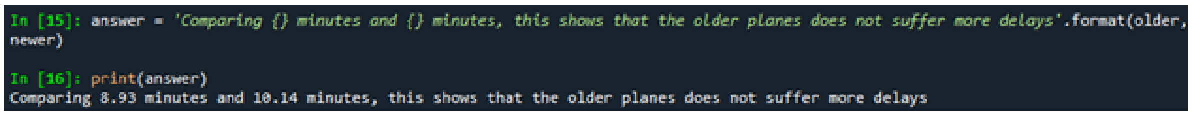
Image 10. Line plot for older and newer planes in Python.



```
In [17]: older = mean(df_olderplanes, 'avg_delay')
    ...: print(older)
8.93
```



```
In [18]: newer = mean(df_newerplanes, 'avg_delay')
    ...: print(newer)
10.14
```



```
In [15]: answer = 'Comparing {} minutes and {} minutes, this shows that the older planes does not suffer more delays'.format(older,
newer)

In [16]: print(answer)
Comparing 8.93 minutes and 10.14 minutes, this shows that the older planes does not suffer more delays
```

Image 11. Results from mean of older and newer planes for Python.

3. How does the number of people flying between different locations change over time?

I assumed that the number of trips to each state equals to number of people flying between different locations. Every flight to a particular state would be counted as total trips taken by people.

3.1 Using R

I did a query to count the number of times there were flights to a certain state for each of the five years. I divided it by a thousand as I realized that big values were unnecessary since I could not see the numbers on the y-axis after plotting. Next, I removed 12 unknown states and the state of DE in 2006 and 2007 from the data frames since other years did not have any flights from there. I bound the 5 data frames (upd2004, upd2005, upd2006 etc) together using rbind() and plotted the graph to analyse using ggplot2 shown in Image 12.

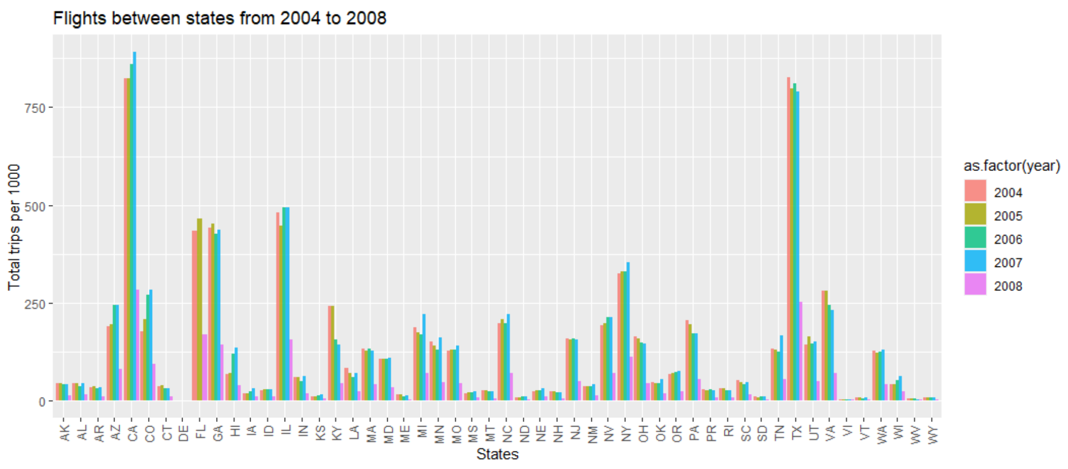


Image 12. Bar plot for flights over the 5 years in different states for R.

3.2 Using Python

The steps to retrieve the queries were like R. Some differences were that I had to subset my data frame into 5 parts to split the 50 states into groups of 5 and concatenate it together into a single variable. 10 states of each year were placed into a subplot as the visualization would be too small to compare if I fit all 50 into a single plot. I then bound the final 5 data frames together (plot1f, plot2f, plot3f etc.) using Pandas concatenate function. Lastly, I used matplotlib and ax to create the 5 empty subplots. After which, I assigned the final 5 data frames to the respective subplots, and it formed the result seen in Image 13Image 13.

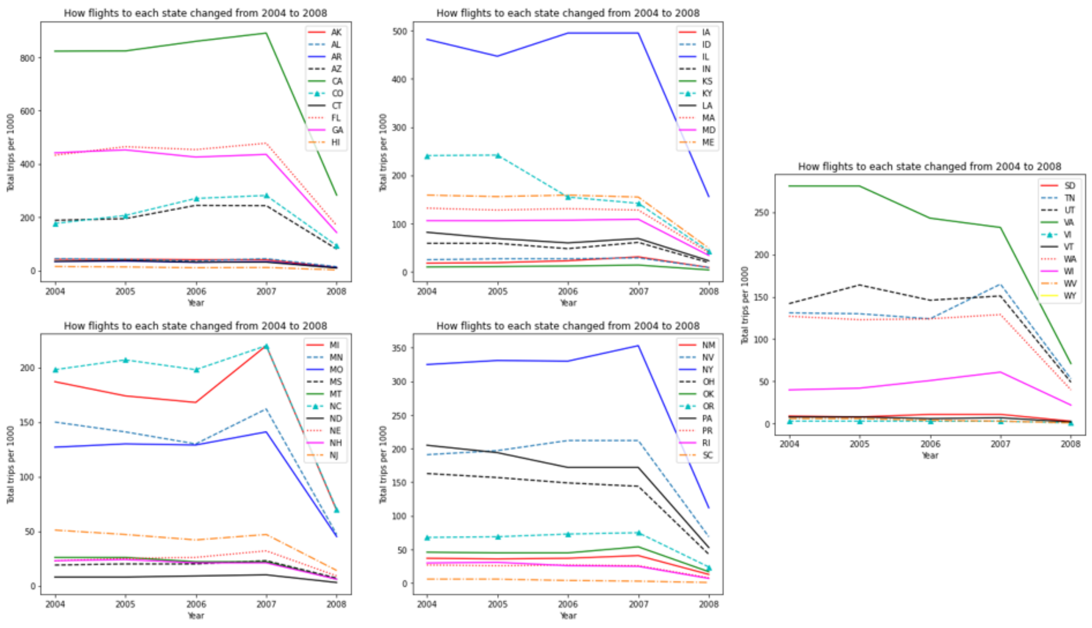


Image 13. Subplots with 10 states in each over the 5 years for Python.

From the results shown in both R and Python, I can conclude that in **2008** most of the flights had **a plunge in flights**. This could be due to a **spike in fuel prices**. (Goldman, 2008) Observing from the R plot, I could also tell which states had some of the highest number of flights (CA and TX) as well as the trend for each state of how flights were from 2004 to 2008.

4. Can you detect cascading failures as delays in one airport create delays in others?

I decided to create a Network visualisation only for 2007 as it had the highest number of flights among the five years. This would help me to see a relationship between states and delayed flights.

4.1 Using R

I queried the edges by using Origin, Dest and ArrDelay variables from ontime table. I chose these variables because I want to see the relationship between the states and their arrival delays from other states. I also decided to filter airports that has more than or equal to 1500 delayed arrival flights since the range of total delayed arrival flights was up to 4000+. Anything below that would be considered invalid. After that, I made a subset to ensure that my nodes and edges' list had the same origins as there was an error that prevented me from plotting the Network. I converted the data frames using an igraph network object and used a circle layout shown in Image 14.

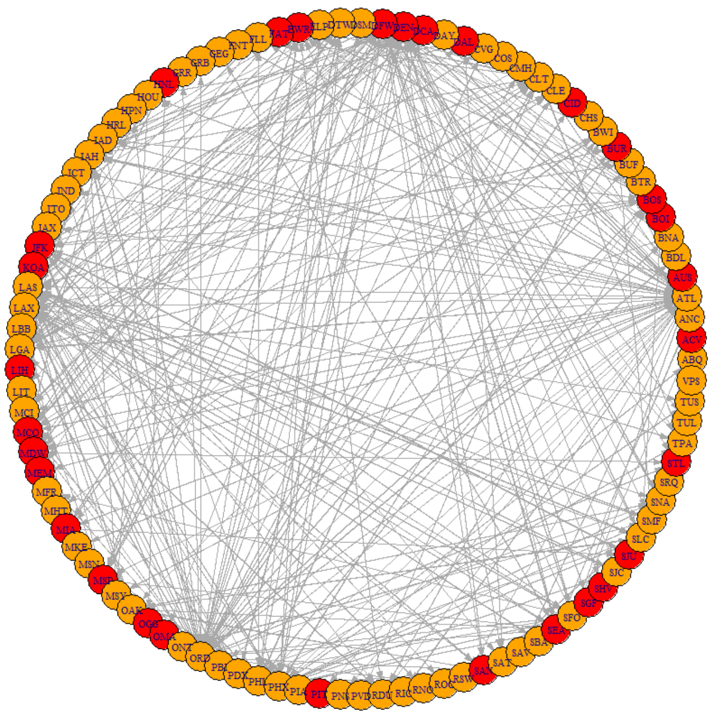
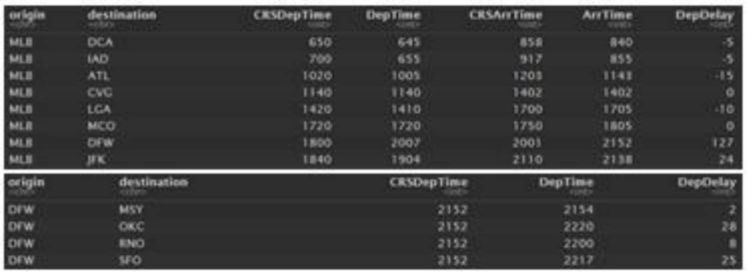


Image 14. Network visualisation for relationship between airports and delayed flights using R.

The red nodes represent airports that has more than 45 minutes of average departure delay. From here, I can also observe a concentration of some flights to different destinations: ORD, ATL, DCA, DFW and DEN. I chose to focus on DFW to proof of cascading delays among the 5 states that I have picked out. The next query helped me to decide which origin I should take that was going to DFW. The output showed MLB had the highest average delay of 127 minutes.

I focused on the schedule for MLB which showed the arrival time to DFW was 2152 hours when it was supposed to arrive at 2001 hours. The last code chunk showed the schedule for DFW and at 2152 hours of intended departure to MSY, it had a slight delay that pushed back to 2154 hours along with other flight delays shown in Image 15Image 15.



origin	destination	CRSDepTime	DepTime	CRSArrTime	ArrTime	DepDelay
MLB	DCA	650	645	858	840	-5
MLB	IAD	700	655	917	855	-5
MLB	ATL	1020	1005	1203	1141	-15
MLB	CVG	1140	1140	1402	1402	0
MLB	LGA	1420	1410	1700	1705	-10
MLB	MCO	1720	1720	1750	1805	0
MLB	DFW	1800	2007	2001	2152	127
MLB	JFK	1840	1904	2110	2138	24

origin	destination	CRSDepTime	DepTime	DepDelay
DFW	MSY	2152	2154	2
DFW	OKC	2152	2220	28
DFW	RNO	2152	2200	8
DFW	SFO	2152	2217	25

Image 15. Schedule from MLB and DFW.

4.2 Using Python

The queries made were based on what I found in R. Instead of igraph, Python uses network. I started off by forming an empty graph first followed by adding the edges which I used from the relationdf data frame. I did some data wrangling to remove isolated vertices and ensure that the nodes and edges were there. Next, I set up the options for the size and colours and then plot by adjusting figure size, the layout, labels for the nodes etc. The result is shown in Image 16Image 16.

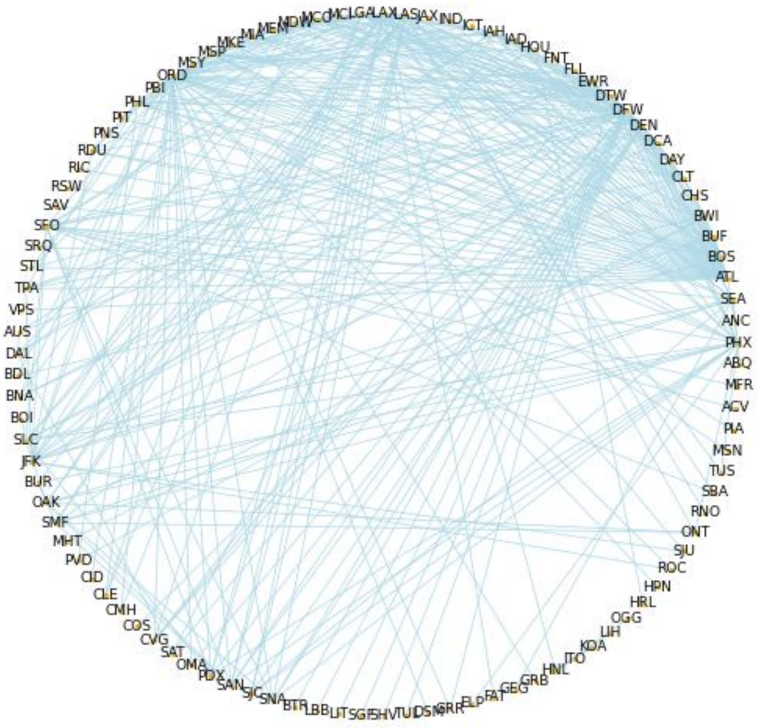
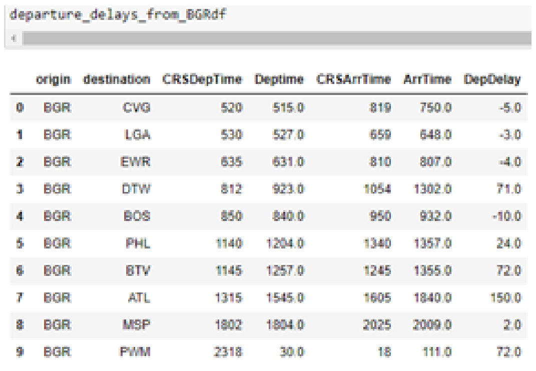
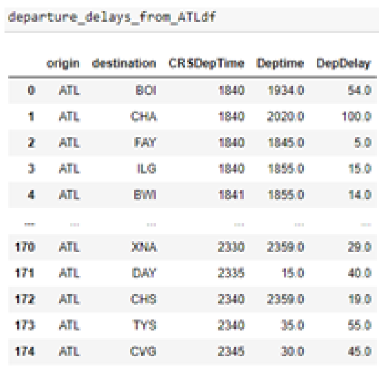


Image 16. Network visualisation for connections between airports using Python.

From the figure I can visually see that most flights go to ATL so I decided to focus on ATL for Python instead of DFW. After querying, I found out that the origin with the highest departure delays going to ATL was BGR of 25.24 minutes. I queried for the arrival timing from BGR to ATL and it was delayed by 150 minutes. Initial arrival time was 1605 hours but only arrived at 1840 hours. I moved on to the next code chunk to see the departure delay in ATL and observed that there were at least two flights which was delayed by 50 minutes or so as seen in Image 17. Overall conclusion after using R and Python with **two different sets of data**, shows that **there is cascading failures** as delays in one airport will usually create delays in another. I felt that one reason for such delays was most likely the time needed for airports to give clearance for the next flight.


```
departure_delays_from_BGRdf
```

	origin	destination	CRSDepTime	Deptime	CRSArrTime	ArrTime	DepDelay
0	BGR	CVG	520	515.0	819	750.0	-5.0
1	BGR	LGA	530	527.0	659	648.0	-3.0
2	BGR	EWR	635	631.0	810	807.0	-4.0
3	BGR	DTW	812	923.0	1054	1302.0	71.0
4	BGR	BOS	850	840.0	950	932.0	-10.0
5	BGR	PHL	1140	1204.0	1340	1357.0	24.0
6	BGR	BTV	1145	1257.0	1245	1355.0	72.0
7	BGR	ATL	1315	1545.0	1605	1840.0	150.0
8	BGR	MSP	1802	1804.0	2025	2009.0	2.0
9	BGR	PWM	2318	30.0	18	111.0	72.0


```
departure_delays_from_ATLdf
```

	origin	destination	CRSDepTime	Deptime	DepDelay
0	ATL	BOI	1840	1934.0	54.0
1	ATL	CHA	1840	2020.0	100.0
2	ATL	FAY	1840	1845.0	5.0
3	ATL	ILG	1840	1855.0	15.0
4	ATL	BWI	1841	1855.0	14.0
...	...	...	...	...	...
170	ATL	XNA	2330	2359.0	29.0
171	ATL	DAY	2335	15.0	40.0
172	ATL	CHS	2340	2359.0	19.0
173	ATL	TYS	2340	35.0	55.0
174	ATL	CVG	2345	30.0	45.0

Image 17. Schedule from BGR and ATL.

5. Use the available variables to construct a model that predicts delays.

I also decided to focus on 2007 data for this question since it has the greatest number of flights for machine learning. The more samples, the better for the machine to learn.

5.1 Using R

To use the machine learning tools, I had to import most of it from mlr3 via the library which included the learners, pipelines, tuning and visualization tools. I made a query to extract out the five origins with the highest average delay to train the data set. I went on to filter those five origins and changed the Origin, Dest and TailNum to be a factor. As 'DepDelay' is not a factor, the task had to be a Regression problem. Next, I made an encoder to convert some variables that are factors to numerical values as some models do not take factors to learn. I also made tuners and a terminator to tune the hyperparameters.

Tuning helps to maximize the model's performance without creating too high of a variance (RiskSpan, 2017). As for po('imputemean') and po('imputemode'), they help to replace any missing values that could be in the train set by either setting the mean of variables, mode, the most frequent or a constant value used in the feature. Since the process is automated, I do not have to look through the data frame for missing values and the code would run on its own. I chose to use two different regression learner models, Random Forests and Support Vector Machine. Lastly, I benchmarked the models to compare the best model for prediction and it clearly showed that it was **Random Forests** as seen in Image 18.

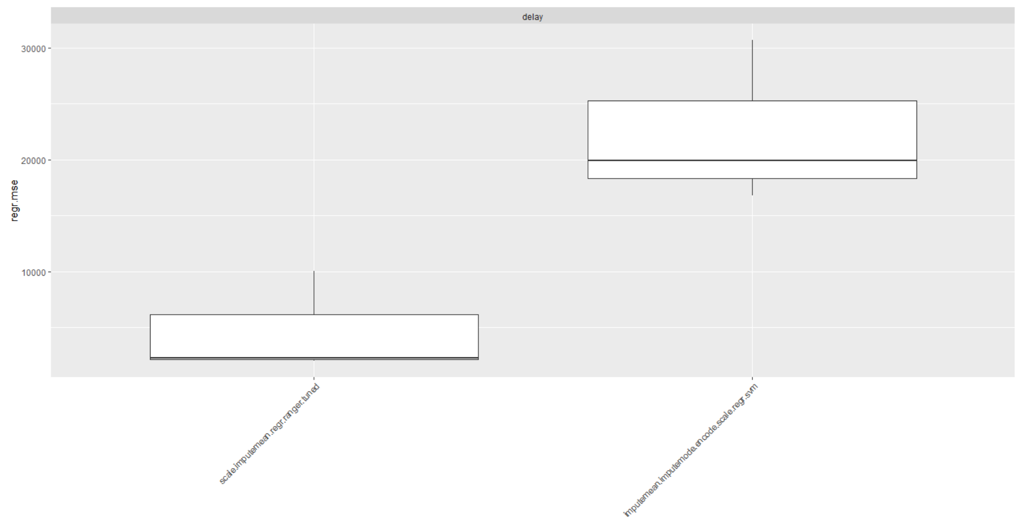


Image 18. Boxplot for model comparison in R.

5.2 Using Python

The tools for Python were slightly different in terms of command and I extracted them from sklearn. They included model_selection for me to split my data into train and test sets, ensemble for RandomForestClassifier, pipeline, impute, preprocessing and compose for the column transformer.

I started off by loading the 2007 csv file and filtered against the top 5 origins found previously in R. I chose the features I wanted and the output to be 'DepDelay'. This was followed by sorting the features into their numerical and categorical transformer. The SimpleImputer, Standard Scalar and OneHotEncoder are the same concepts as po('imputemean') and po('imputemode') in R where they replace any missing values in the selected features and output chosen.

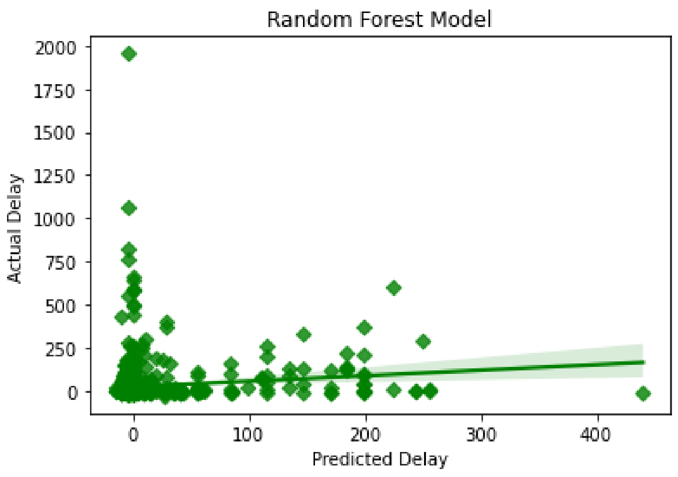


Image 19. Random Forest Model for Python.

I also decided to use more samples for training (60%) so that our accuracy will be slightly better than halving it. In Python, GridSearchCV() helps with tuning the hyperparameters by doing cross validation among the train and test set. Once done, I printed the best score and parameter used for the Random Forest Model for checking. Lastly, I proceeded to place the test set into the pipeline and did a prediction. Using **seaborn regression plot** to get the visualisation for the **Random Forest Model** and the output is shown in Image 19.

References

Brownlee, J. (2020, April 8). *4 types of classification tasks in machine learning*. Machine Learning Mastery. https://machinelearningmastery.com/types-of-classification-in-machine-learning/

Data Independent. (2022, January 25). *Pandas bar plot – DataFrame.plot.bar()*. https://dataindependent.com/pandas/pandas-bar-plot-dataframe-plot-bar/

Goldman, D. (2008, July 15). *Airlines may face bankruptcies - report - JUL. 15, 2008*. Business News - Latest Headlines on CNN Business - CNN. https://money.cnn.com/2008/07/15/news/economy/airlines/

Grepper. (2020, July 24). *How to insert a value in a print statement Python code example*. https://www.codegrepper.com/code-examples/python/how+to+insert+a+value+in+a+print+statement+python

Kite. (n.d.). *Code faster with line-of-Code completions, cloudless processing*. https://www.kite.com/python/answers/how-to-drop-empty-rows-from-a-pandas-dataframe-in-python

Kumar, S. (2020, May 6). *Format function in Python*. Medium. https://towardsdatascience.com/format-function-in-python-98ed34e0a70e

Net-Informations.Com. (n.d.). *TypeError: 'NoneType' object is not subscriptable*. https://net-informations.com/python/err/nonetype.htm

Pandas. (n.d.). *Pandas.DataFrame.median — pandas 1.4.1 documentation*. https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.median.html

SQL Shack. (2019, July 18). *Understanding the SQL server CASE statement*. Articles about database auditing, server performance, data recovery, and more. https://www.sqlshack.com/understanding-sql-server-case-statement/

stackoverflow. (2020, April 6). *Error: Stat_count() can only have an X or Y aesthetic*. https://stackoverflow.com/questions/61068031/error-stat-count-can-only-have-an-x-or-y-aesthetic

Tuning machine learning models. (2017, November 7). RiskSpan. https://riskspan.com/tuning-machine-learning-models/