ST2195 Coursework Report 2024

Student Number: 190341259

2024-03-09

Environment Setup

To set up the environment for running my code, please follow these steps:

1. Clone the Repository:
   - Open your terminal or command prompt and navigate to a directory of your choice using `cd`
   - Execute the following command to clone the repository to your local machine: `git clone https://github.com/inc0gn1k0/submission_final_st2195.git`
   - Alternatively, you can download the repository directly from GitHub by following the link above.
2. Navigate to the Repository:
   - Open the submission_final folder that you have just cloned or downloaded.
   - Run the `environment.yml` file using `conda env create -f environment.yml` to install the required packages.
3. Review the README.md File:
   - Inside the submission_final folder, please refer to the README.md file for detailed instructions on setting up the environment and running the code
4. Download and Populate Raw Data:
   - Please note that the raw_data folder within the repository is currently empty.
   - Follow the instructions provided in the raw_data folder to download and organize the data appropriately.
   - To populate the raw_data folder with the necessary data, download the dataset from this link.

Relevant Notebooks

Question	Python Notebook Paths	R Notebook Paths
Part 1 (A)	./submission_final/python_notebooks/1A_python.ipynb	./submission_final/r_notebooks/1A_R_notebook.Rmd
Part 1 (B)	./submission_final/python_notebooks/1B_python.ipynb	./submission_final/r_notebooks/1B_R_notebook.Rmd
Part 2 (A)	./submission_final/python_notebooks/2A_python.ipynb	./submission_final/r_notebooks/2A_R_notebook.Rmd
Part 2 (B)	./submission_final/python_notebooks/2B_python.ipynb	./submission_final/r_notebooks/2B_R_notebook.Rmd
Part 2 (C)	./submission_final/python_notebooks/2C_python.ipynb	./submission_final/r_notebooks/2C_R_notebook.Rmd

Table of Contents

Part 1

Background

- In their seminal 1996 work, Markov Chain Monte Carlo in Practice, Gilks, Richardson, and Spiegelhalter define a Markov chain as a "series of random events where the probability of each event depends on the state attained in the previous event."
- Using code, and initial conditions of sample size *N*, and standard deviation (steps) *s*, I am going to generate a proposal distribution, accept or reject the proposed moves using an acceptance criterion, and then based on what was accepted, generate a sequence of samples that converges to having the target distribution.
- This creates a Markov chain whose equilibrium distribution matches the target distribution. The samples from this chain of values is then used to approximate properties about the target distribution (which in our case is f(x)), such as a **Monte Carlo Mean and Standard deviation**, and to construct **Histograms and Kernel Density plots** that approximate the target distribution's shape.
- The equation below is the terminal distribution we have to plot, in the last stage of our program, using the values generated by our algorithm. The resulting line should ideally closely resemble our KDE and histogram, showing that our algorithm effectively approximates our target distribution through the samples it is generating.
  $f(x) = \frac{1}{2} exp(-|x|)$ → The inputs (x) of the pdf above are to be generated using our program.

Part 1A

[1] In this section, using Python, I show how we can apply the Random Walk Metropolis Algorithm using N=10000 and s=1

Python Code

```
# Here I apply the Random Walk Metropolis Algorithm using N=10000 and s=1
import numpy as np
from scipy.stats import uniform, norm
import matplotlib.pyplot as plt
import seaborn as sns
def f(x):
    return 0.5 * np.exp(-np.abs(x))
def log_f(x):
    return np.log(0.5) - np.abs(x)

# Random walk Metropolis algorithm
def random_walk_metropolis(N, s, x0):
    samples = np.zeros(N) #generates a sample
    samples[0] = x0  # initial value of my sample

    for i in range(1, N):
        x_star = norm.rvs(loc=samples[i-1], scale=s)  # propose a new state
        log_r = log_f(x_star) - log_f(samples[i-1])  # compute log of ratio
        log_u = np.log(uniform.rvs())  # log of uniform random number for comparison

        if log_u < log_r:
            samples[i] = x_star  # accept the new state
        else:
            samples[i] = samples[i-1]  # reject the new state

    return samples

N = 10000
s = 1
x0 = 0

samples = random_walk_metropolis(N, s, x0) #Please see 1A_R_notebook.Rmd for the equivalent R code!
```

Justification

Dissecting our function: `random_walk_metropolis()`

1. First I generate a candidate for state change x_* (x_star)

```
for i in range(1, N):
    x_star = norm.rvs(loc=samples[i-1], scale=s)
```

Here we simulate a random number from the normal distribution with mean (loc) equal to prev step (x_i)and standard deviation s.

2. Next we compute an acceptance ratio of a move given the acceptance criterion $u < r(x_*, x_{i-1})$
   - Expressing the acceptance criterion in log terms makes our analysis more numerically stable $log_u < log_r(x_*, x_{i-1})$, where $log_r(x_*, x_{i-1}) = logf(x_*) - logf(x_{i-1})$…this equation is represented in the code below

```
for i in range(1, N):
    log_r = log_f(x_star) - log_f(samples[i-1])
```

3. u is a uniformly distributed random number between 0 and 1 (and it is randomly drawn), and I use the code below to express this:

```
log_u = np.log(uniform.rvs())
```

4. We accept a move to the proposed step (and so set $x_i = x_*$) if $log_u < log_r(x_*, x_{i-1})$ and "stay" on current value (by setting $x_i = x_{i-1}$ where x_{i-1} is the previous step) if $log_u > log_r(x_*, x_{i-1})$, so we use a conditional statement in R and python such as the one below:

```
if log_u < log_r:
    samples[i] = x_star  # accept the new state
else:
    samples[i] = samples[i-1]  # reject the new state
```

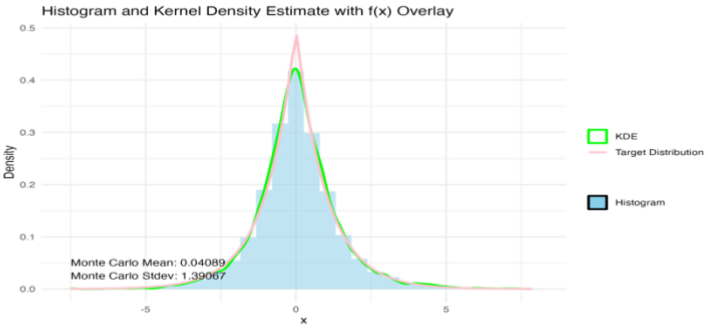
[2] We can then use the generated samples $(\boldsymbol{x1}, \dots \boldsymbol{xN})$ to construct a histogram (blue) and kernel density plot (green) in the same figure, and [3] overlay a graph of f(x) on this figure (red) to visualize the quality of the estimates. [4] Finally the Monte Carlo Mean and Standard deviations are also reported for the generated samples.

Note that the mean and stdev cannot be exactly the same whenever we re-run the code because we are dealing with stochastic inputs.

R Code

```
#Here I apply the Random Walk Metropolis Algorithm using N=10000 and s=1

samples <- random_walk_metropolis(N, s, x0)
sample_mean <- mean(samples)
sample_std <- sd(samples)

plot <- ggplot(data.frame(x = samples), aes(x = x)) +
  geom_histogram(aes(y = ..density.., fill = "Histogram"), bins = 30, alpha = 0.5) +
  geom_density(aes(color = "KDE"), size = 1) +
  stat_function(fun = f, aes(color = "Target Distribution"), size = 1) +
  scale_fill_manual(name = "", values = "skyblue", labels = "Histogram") +
  scale_color_manual(name = "", values = c("KDE" = "green", "Target Distribution" = "pink"), labels = c("KDE", "Target Dist
ribution")) + # Manual scale for color
  labs(title = "Histogram and Kernel Density Estimate with f(x) Overlay",
       x = "x", y = "Density") +
  annotate("text", x = min(samples), y = max(0.5 * exp(-abs(min(samples)))), label = sprintf("Monte Carlo Mean: %.5f\nMonte
Carlo Stdev: %.5f", sample_mean, sample_std), hjust = 0, vjust = -0.5, size = 3.5) +
  theme_minimal() +
  guides(fill = guide_legend(override.aes = list(alpha = 1)))

print(plot)

#Please see 1A_python.ipynb for the equivalent python code!
```

Final output



Part 1B

[1] In this section I use R to calculate $\boldsymbol{\widehat{R}}$ for the random walk Metropolis algorithm with N = 2000, s=0.001 and J=4

The goal of computing this value of $\boldsymbol{\widehat{R}}$ is to assess whether the variances within the chains are comparable to the variance between the chains, indicating that the chains are sampling from the same distribution and have likely converged. In general values of $\boldsymbol{\widehat{R}}$ close to 1 indicate convergence, and it is usually desired for $\boldsymbol{\widehat{R}}$ to be lower than 1.05

R Code

```
log_f() #previously defined in Question 1A
random_walk_metropolis() #previously defined in Question 1A
compute_R_hat() #New Function!
compute_R_hat <- function(N, s, J, x0_initial_values) {
  chains <- sapply(x0_initial_values, function(x0) random_walk_metropolis(N, s, x0))
  Mj <- colMeans(chains)
  Vj <- apply(chains, 2, function(chain) var(chain))
  W <- mean(Vj)
  B <- N * var(Mj)
  var_hat_plus <- ((N-1)/N) * W + (1/N) * B #This is the Gelman and Rubbin 1992 adjustment…
  R_hat <- sqrt(var_hat_plus / W)

  R_hat
}

R_hat_specimen <- compute_R_hat(N, s, J, x0_initial_values)
print(paste("Caclculated R_hat:", R_hat_specimen)) #Calculate R_hat for RWMA with N=2000, s=0.001 and J=4
#Please see 1B_python.ipynb for the equivalent python code!
```

Justification

The theory behind the code

i. Enumerate a number (J) of sequences (N) of $x_0, \dots, x_N$, using different initial values x0.
Each chain should be denoted by $(x_0^j, x_1^j, \dots, x_N^j)$ for $j = 1,2, \dots, J$
If `J = 4`, then `x0_initial_values = np.random.randn(J)` captures initial random values of our 4 chains. Within our function we then combine our chains in an array data structure and store it in the variable named `chains`
`chains = np.array([random_walk_metropolis(N, s, x0) for x0 in x0_initial_values])`

ii. $M_j = \frac{1}{N}\sum * i = 1^N x * i^{(j)}$ This expression is a simple mean for each chain in our array "chains": `Mj = chains.mean(axis=1)`. Here we calculate the mean for each chain (axis one goes row by row, and we have 4 rows in our numpy array).

iii. The within sample variance of chain j is a simple variance calculation `var()`for each chain $V_j = \frac{1}{N}\sum_{i=1}^N (x_i^{(j)} - M_j)^2$
so we calculate the variance of each chain using `Vj = chains.var(axis=1)`

iv. The overall within sample variance W is a point estimate of the variance of the variances of each chain $W = \frac{1}{J}\sum * j = 1^J V * j$
`W = Vj.mean()`

v. Define and compute the overall sample mean M as the mean of the means of our chains…
$M = \frac{1}{J}\sum_{j=1}^J M_j$
`M = Mj.mean()`remember that chains is an array, and Mj stores an array of means.

vi. and the between sample variance B is the variance of each chain's mean from the overall mean of all the chains $B = \frac{1}{J}\sum * j = 1^J (M * j - M)^2$
`B = N/(J-1) * np.var(Mj, ddof=1)` note that we have to scale the between sample variance by N in order to get B (variance estimate based on means of each chain) to the same order of magnitude as W (variance based on individual observations).

vii. Compute the R_hat value as$\hat{R} = \sqrt{\frac{B+W}{W}}$
With a small adjustment according to the work of Gelman and Rubin 1992. We include the $\widehat{Var}^+$ statistic, and use that as our numerator instead of B+W

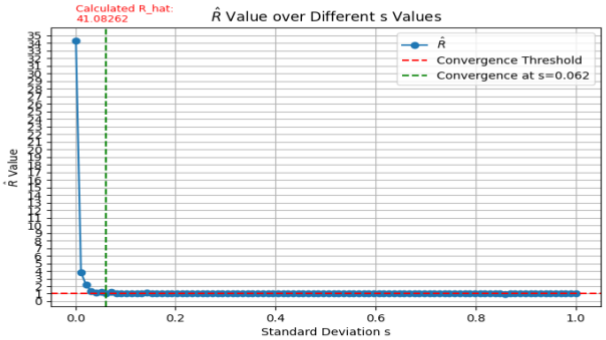
[2] Now, keeping N and J fixed, I generate a plot of the values of $\boldsymbol{\widehat{R}}$ over a grid of s values in the interval between 0.001 and 1 (this will show us convergence behaviour visually)

Though the equations looked complex for this task, they are actually simple mean and variance calculations that Python and R have preinstalled by default.

Python Code

```
# Parameters
N = 2000 #the question asks that I set this to 2000
J = 4 #the question asks that I set this to 4
s = 0.001 #the question asks that I set this to 0.001…notice that this value is well below our Convergence point!
s_values = np.linspace(0.001, 1, 100) #First arg specifies start, second arg represents stop and the third argument, 100, specifies that this interval shoul
d be divided into 99 equal segments, creating 100 evenly spaced points within and including the bounds.
x0_initial_values = np.random.randn(J) #Random initial values for the chains...so if J is 4 it is 4 random intial values for the chains
# Calculate R_hat for each s value
R_hats = [compute_R_hat(N, s, J, x0_initial_values) for s in s_values]
R_hat_value = compute_R_hat(N, s, J, x0_initial_values) # R_hat forRWMA with N=2000, s=0.001 and J=4
# Plot R_hat over s values
plt.figure(figsize=(8, 5))
plt.plot(s_values, R_hats, '-o', label='$\hat{R}$')
plt.xlabel('Standard Deviation s')
plt.ylabel('$\hat{R}$ Value')
plt.yticks(np.arange(np.floor(min(R_hats)), np.ceil(max(R_hats))+1, 1))  # Adjusting y-axis increments to 1
plt.title('$\hat{R}$ Value over Different s Values')
plt.axhline(1.05, color='red', linestyle='--', label='Convergence Threshold')
plt.text(min(s_values), max(R_hats)*1.07, f'Calculated R_hat:\n{R_hat_value:.5f}', fontsize=9, color='red', ha='left')
# Identify and plot the convergence point
convergence_points = [s for s, R_hat in zip(s_values, R_hats) if R_hat <= 1.05]
if convergence_points:
    plt.axvline(convergence_points[0], color='green', linestyle='--', label='Convergence at s=%.3f' % convergence_points[0])
plt.grid(True)
plt.legend()
plt.show() #Please see 1B_R_notebook.Rmd for the equivalent R code!
```

Final output



Why did I include var_hat_plus before computing R_hat?

- We had to make a small adjustment according to the work of **Gelman and Rubin 1992** by including the $\widehat{Var}^+$ statistic, wherein we use that as our numerator instead of B+W.
- As it turns out, our first visualization of $\hat{R}$ vs stdev showed that $\hat{R}$ was well below 1 (please see 1B_python.ipynb or 1B_R_notebook.Rmd), this would mean that chains are more similar to each other than to themselves over time, which does not fit the logic underpinning $\hat{R}$ (as I expected $\hat{R}$ to remain stable at around 1 as stated in the question). Further literature review showed that our formula for $\hat{R}$ could be improved, especially when it came to calculating $\hat{R}$ using B and W (we were not achieving a convergence at 1, our convergence idled close to zero with increasing standard deviations). This is because after scaling B and W appropriately, we obtained the correct $\hat{R}$ convergence. If all chains are sampling from the same target distribution, the between-chain and within-chain variances should be similar, making $\hat{R}$ close to 1.

Part 2

Database Setup

In Part 2, I analyze a subset of 10 consecutive years (1998 to 2007) of IATA data from the Harvard Dataverse. I produced an SQL database (‘comp97to07.db’) composed of 15 tables.

Please note: For all questions, I use a list in python and a vector in R to select the years of interest. It is possible to use any subset of the 10 years I selected by adjusting this list before running each code block, thereby speeding up kernel execution time.

```
#Please open ‘./r_notebooks/2DBsetup_R_notebook.Rmd’ and open ‘./python_notebooks/2DBSetup_python.ipynb’ to see how I setup the primary reference Database (‘comp97to07.db’), that is described below, in R and Python

Tables
Data: 1997.csv, 1998.csv, 1999.csv, 2000.csv, 2001.csv, 2002.csv, 2003.csv, 2004.csv, 2005.csv, 2006.csv, 2007.csv, Supplementary Data: airports.csv, carriers.csv, plane-data.csv, variable-descriptions.csv
```

Objectives of both the Python and R notebook code blocks at this stage

- **Use of Relative Paths for reproducibility:** By using os.path.join(current_dir,'..', 'raw_data') in Python and file.path(getwd(), '..', 'raw_data') in R, I ensured that the scripts can dynamically locate the raw_data directory relative to the script's execution location. This makes my code more portable, as it doesn't rely on hard-coded absolute paths, which can vary across different users' environments.
- **Iterating Over Years:** I created a list (in python) and a vector (in R) of CSV file paths for the years 1997 through 2007. This iterative approach streamlines the process of handling multiple data files without manually specifying each file, making the code more efficient and easier to maintain.
- **Handling Supplementary Data:** Similar to the yearly data, I created lists (and vectors) that map supplementary data files to their corresponding table names. This ensures that additional relevant data, like plane information, airports, carriers, and variable descriptions, is also included in the database, providing a comprehensive data structure.
- **Creating and Populating the SQLite Database:** Both scripts connect to (and create) a SQLite database file, comp97to07.db, in the raw_data directory. I then iterate over the CSV files to create tables and populate them with the CSV data. This approach centralizes the data in a single database, facilitating easier access and analysis.
- **Memory Management:** After each table is populated, I explicitly free memory by deleting the temporary dataframe variable (del(df) in Python and rm(df) in R) and invoking garbage collection (gc.collect() in Python and gc() in R). This is particularly important for large datasets or when running the script on machines with limited memory resources.

Final steps to create the database:

```
#Using Python to connect to DB with sqlite3
#To see the preliminary processing steps, please see Jupyter and Rmarkdown notebooks)
....
# This is how I connected to the SQLite database using Python’s sqlite3 (if DB did not exist before it will be automatically created)
conn = sqlite3.connect('comp97to07.db')

#Using R to connect to DB with DBI and RSQLite
#To see the preliminary processing steps, please see Jupyter and Rmarkdown notebooks)
....
# This is how I connected to the SQLite database using RSQLite (if DB did not exist before it will be automatically created)
conn <- dbConnect(RSQLite::SQLite(), dbname = 'comp97to07.db')
```

Part 2A

When is the best time of day, day of the week, and time of year to fly to minimize delays?

Libraries

```
#python
import sqlite3
import pandas as pd
import dataframe_image as dfi
#r
library(DBI)
library(RSQLite)
library(readr)
```

Data Source

comp97to07.db

Data Cleaning and Shaping

1. After creating our comp97to07.db we check the variable_description.csv to understand the naming system of the columns
2. We want to know the CRSDepTime, DayOfWeek and Month with the **most frequent incidence of ArrDelay and DepDelay of 0** for our entire date range.
3. We use the following query format using `dbGetQuery()` in R and `read_sql_query()` in python to extract this information for each year (using a loop) and store the top most row returned by each query in one of 3 lists:

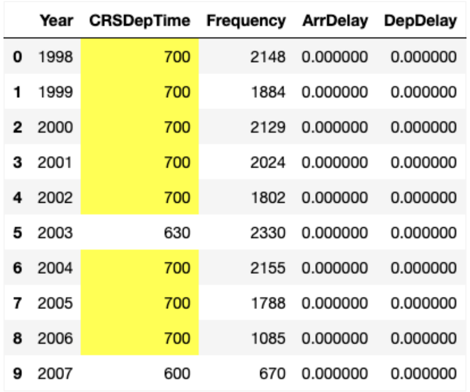
Python	R
`top_rows_time = []`	`top _rows_time <- list()`
`top_rows_week = []`	`top _rows_week <- list()`
`top_rows_month = []`	`top _ rows_month <- list()`

```
#Query1 to obtain best time of day
"SELECT Year, CRSDepTime, COUNT(CRSDepTime) AS Frequency, ArrDelay, DepDelay
                         FROM Y%d
                         WHERE DepDelay = 0 AND ArrDelay = 0
                         GROUP BY CRSDepTime
                         ORDER BY Frequency DESC
                         LIMIT 1"

#Query2 to obtain best day of week
"SELECT Year, DayOfWeek, COUNT(DayOfWeek) AS Frequency, ArrDelay, DepDelay
                         FROM Y%d
                         WHERE DepDelay = 0 AND ArrDelay = 0
                         GROUP BY DayOfWeek
                         ORDER BY Frequency DESC
                         LIMIT 1"

#Query3 to obtain best time of year
"SELECT Year, Month, COUNT(Month) AS Frequency, ArrDelay, DepDelay
                          FROM Y%d
                          WHERE DepDelay = 0 AND ArrDelay = 0
                          GROUP BY Month
                          ORDER BY Frequency DESC
                          LIMIT 1"
```

3. Notice that I SELECT only variables of interest to answer each part of the question, and that the GROUP BY Statement and COUNT() are critical for aggregating the variable of interest. I then ORDER BY Frequency in Descending order.
4. By limiting each query to only produce the top most value for each year, I improve performance of my queries, and allow the loop to proceed quickly for my chosen subset of years. **I also used two dictionaries to map the numeric representation of Days of the week and Months of the year to their string equivalents.**
5. I added a feature that highlights the mode for each variable of interest using a custom python function `highlight_mode()`, for all the years in our date range, and display our results in a table (see the images below). Please note, that at the time of writing, I did not know how to build the same color logic into R, so this has been left out of my R markdown notebook. However, the tables produced in R are otherwise **identical** to those pictured below.

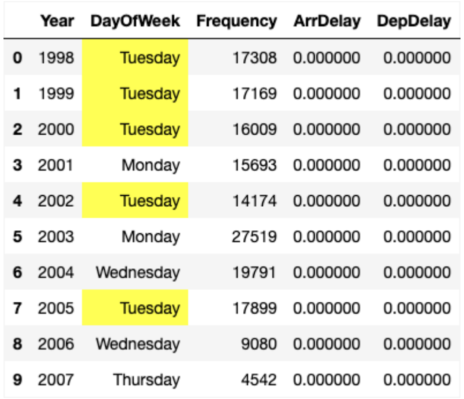
Visualization and results

The best time to fly to minimize delays is 7:00AM



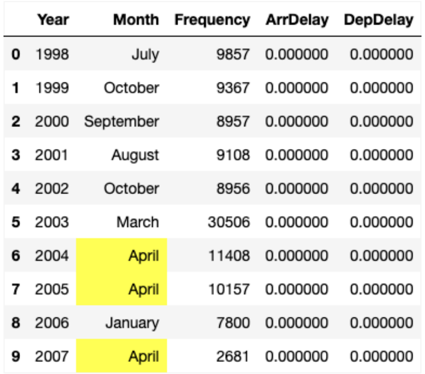
	Year	CRSDepTime	Frequency	ArrDelay	DepDelay
0	1998	700	2148	0.000000	0.000000
1	1999	700	1884	0.000000	0.000000
2	2000	700	2129	0.000000	0.000000
3	2001	700	2024	0.000000	0.000000
4	2002	700	1802	0.000000	0.000000
5	2003	630	2330	0.000000	0.000000
6	2004	700	2155	0.000000	0.000000
7	2005	700	1788	0.000000	0.000000
8	2006	700	1085	0.000000	0.000000
9	2007	600	670	0.000000	0.000000

The best day of the week to fly to minimize delays is Tuesday



	Year	DayOfWeek	Frequency	ArrDelay	DepDelay
0	1998	Tuesday	17308	0.000000	0.000000
1	1999	Tuesday	17169	0.000000	0.000000
2	2000	Tuesday	16009	0.000000	0.000000
3	2001	Monday	15693	0.000000	0.000000
4	2002	Tuesday	14174	0.000000	0.000000
5	2003	Monday	27519	0.000000	0.000000
6	2004	Wednesday	19791	0.000000	0.000000
7	2005	Tuesday	17899	0.000000	0.000000
8	2006	Wednesday	9080	0.000000	0.000000
9	2007	Thursday	4542	0.000000	0.000000

The best month to fly to minimize delays is April



	Year	Month	Frequency	ArrDelay	DepDelay
0	1998	July	9857	0.000000	0.000000
1	1999	October	9367	0.000000	0.000000
2	2000	September	8957	0.000000	0.000000
3	2001	August	9108	0.000000	0.000000
4	2002	October	8956	0.000000	0.000000
5	2003	March	30506	0.000000	0.000000
6	2004	April	11408	0.000000	0.000000
7	2005	April	10157	0.000000	0.000000
8	2006	January	7800	0.000000	0.000000
9	2007	April	2681	0.000000	0.000000

Conclusion

To conclude, my analysis over my chosen subset of years (1998 to 2007) indicates that to minimize flight delays, the optimal strategy involves flying at **7:00 AM**, preferably on a **Tuesday**, with **April** being the most favorable month. This approach combines the benefits of early morning travel, weekly scheduling efficiency, and seasonal advantages to increase the probability of travelling on-time in the USA.

Part 2B

Do older planes suffer more delays?

Libraries

```
#python
import sqlite3
import pandas as pd
import statsmodels.api as sm
import matplotlib.pyplot as plt
#r
library(DBI)
library(RSQLite)
library(dplyr)
library(kableExtra) #adds further styling and formatting options for tables to knitr::kable()
library(ggplot2)
```

Analysis Plan

We can use "year" of Launch of the Airplane (Year of Manufacture – YoM) as a proxy for Age, and use code to answer the following:

1. Is the delay, as a percentage of total flights, in any given year higher for older planes, meaning that delay incidence is higher for older planes?
   - We will produce two regressions for our date range showing: %ArrDelay vs YoM and %DepDelay vs YoM to answer this.

2. Is the delay length (WeightedAverageDelay) decreasing in proportion to plane age, meaning older planes suffer longer delays on average?
   - We will produce two regressions for our date range: WeightedAverageArrLength vs YoM and WeightedAverageDepLength vs YoM to answer this.

We can also use "Issue_Date" as a proxy for Age and use code to elucidate the following:

3. We assigned an "Age" to planes based on the [current year - issue year] and counted the aggregate delay incidents for each age, repeating the process over a ten-year period. This data was then expressed as a percentage of total flights for each age group each year. Utilizing regression analysis, we explored the relationship between plane age and flight delays, plotting the percentage of delays against plane age for a comprehensive visual and statistical analysis. Two scatter plots were created to display these results, with axes labeled for age and aggregate percentage delays, providing a clear visualization of how plane age affects delay rates.

Data Source

comp97to07.db

Data Cleaning and Shaping (For Analysis 1 and 2 using Year of Manufacture)

1. **Database Utilization**: Accessed `comp97to07.db`, focusing on flight data from 1998 to 2007 and corresponding plane information. The primary objective was to analyze flight delays concerning plane ages.
2. **Data Modeling Decisions**: Developed SQL queries for correlating flight delays with plane ages by joining flight data (Y1998 to Y2007 tables) with plane data (the 'planes' table) based on tail numbers. This allowed for an accurate matching of flights to their respective planes.
3. **Delay Definition and Filtering**: "The United States Federal Aviation Administration (FAA) considers a flight to be delayed when it is 15 minutes later than its scheduled time." - This means we have to set our filtering condition to only select rows where the delay is >=15. This involved filtering records to include only those with `ArrDelay` and `DepDelay` greater than or equal to 15 minutes.
4. **Handling of Cancelled Flights**: I opted against a separate cleaning step for cancelled flights, as our delay-based filtering inherently excluded such flights.
5. **Exclusion Criteria**: At first I excluded planes introduced post-1998 and ensured that the tail numbers corresponded to unique planes within the considered time frame to maintain data integrity and relevance. But following my normalization procedure (obtaining a percentage of total flights), I included even the most recent planes (Planes from 2007) which have less flights, because we are looking at rate of delay incidence **not the raw count of delays**.
6. **Taxi Time Consideration**: Excluded `TaxiIn` and `TaxiOut` times, focusing solely on scheduled departure and arrival times to precisely measure delays.
7. **Aggregation and Transformation**: Aggregated delay incidents across different years of manufacture (YoM) for a comparative analysis. Transformed SQL query outcomes into Pandas DataFrames, ensuring data quality through data type consistency and null value exclusion.
8. **Visualization and Statistical Interpretation**: Employed linear regression analyses to statistically evaluate the impact of plane age on the likelihood and extent of delays. This provided a quantified measure of the correlation between plane age and delay propensity.

Data Cleaning and Shaping (For Analysis 3 using issue_date as a proxy for Age)

1. **Database Utilization**: For each year in the subset (1998 to 2007), a SQL query selects planes based on their age (current year minus the year from the `issue_date` field), counts delay incidents (where `ArrDelay` or `DepDelay` >= 15), and counts the total number of flights. This step creates a detailed picture of delays related to plane age for each year.
2. **Data Aggregation**: After fetching data for each year, all individual year data frames are concatenated into a single dataframe. This unification is crucial for analyzing trends across multiple years instead of looking at each year in isolation.
3. **Calculating Aggregate Delays and Flights**: The combined data is then grouped by the plane's age, summing up delay incidents and the total number of flights for each age group across all years. This aggregation step is essential for understanding the overall impact of plane age on delay incidents, removing the yearly variance and focusing on age as the main variable.
4. **Calculating Delays as a % of Flights**: For the aggregated data, the percentage of delays relative to the total number of flights is calculated for each age group (delays%flights). This step transforms raw counts into a more interpretable metric that signifies the proportion of flights delayed for planes of each age, allowing for a clearer comparison across different ages.
5. **Preparation for Regression Analysis**: Before conducting the regression analysis, a constant term (for the intercept) is added to the age variable to prepare the data set. This step is a prerequisite for linear regression analysis, ensuring the model accounts for a baseline level of delays%flights that is not dependent on the age of the planes.
6. **Regression Analysis**: With the data prepared, a linear regression model is fitted to predict Delays%Flights based on the Age of planes. This analysis aims to uncover any statistically significant relationships between the age of planes and the incidence of delays.

This was an important cleaning step in my query logic for all 3 analyses for this question: `WHERE p.year <= %d AND p.year IS NOT NULL AND p.year != '0000' AND p.year != '' AND p.year != 'None' AND y.Cancelled != 1`

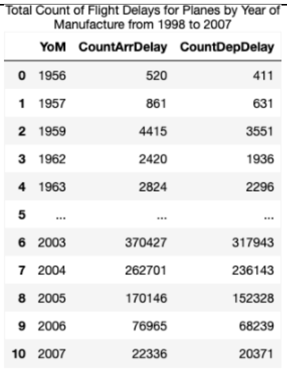
Preliminary Results from my Exploratory Analysis

Our simple COUNT() query returned the following Dataframes (I displayed the head(5) and tail(5) for simplicity) that underpin Analysis 1 and Analysis 2.

Table 1 (The Problem)

Count of Flight delays aggregated by Year of Manufacture

Head(5) and Tail(5) displayed for simplicity…



Total Count of Flight Delays for Planes by Year of Manufacture from 1998 to 2007

	YoM	CountArrDelay	CountDepDelay
0	1956	520	411
1	1957	861	631
2	1959	4415	3551
3	1962	2420	1936
4	1963	2824	2296
5	...	...	...
6	2003	370427	317943
7	2004	262701	236143
8	2005	170146	152328
9	2006	76965	68239
10	2007	22336	20371

Table 2 (The Solution)

Count of Total Flights, Flight Delays and Delays as a percentage of Total Flights, aggregated by Year of Manufacture

Head(5) and Tail(5) displayed for simplicity…

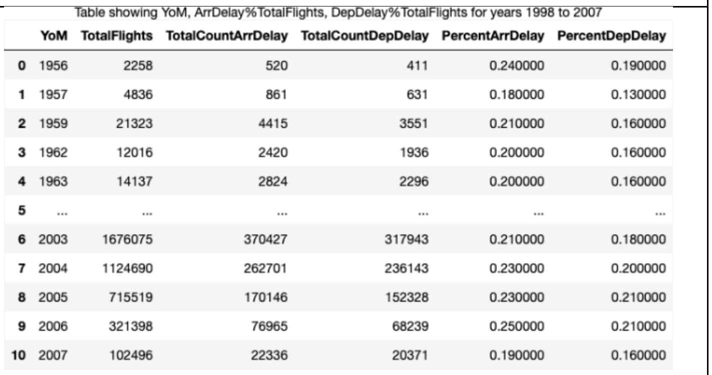


Table showing YoM, ArrDelay%TotalFlights, DepDelay%TotalFlights for years 1998 to 2007

	YoM	TotalFlights	TotalCountArrDelay	TotalCountDepDelay	PercentArrDelay	PercentDepDelay
0	1956	2258	520	411	0.240000	0.190000
1	1957	4836	861	631	0.180000	0.130000
2	1959	21323	4415	3551	0.210000	0.160000
3	1962	12016	2420	1936	0.200000	0.160000
4	1963	14137	2824	2296	0.200000	0.160000
5	...	...	...	...	...	...
6	2003	1676075	370427	317943	0.210000	0.180000
7	2004	1124690	262701	236143	0.230000	0.200000
8	2005	715519	170146	152328	0.230000	0.210000
9	2006	321398	76965	68239	0.250000	0.210000
10	2007	102496	22336	20371	0.190000	0.160000

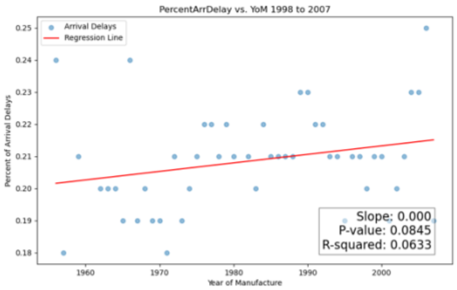
The question asks if older planes suffer more delays, and whilst it is tempting to just accept that older planes have a lower COUNT of the number of delays, the lower COUNT is because in our period of 1998 to 2007 the older planes are less common (fewer flights), and so feature less in our date range. So we needed to perform some data wrangling to normalize our data (hence we produce Table 2). A plane from 1956 is likely to have less flights, just like a plane from 2007 is likely to have less flights, so really old planes, and really young planes are not well represented in our dataset just going by raw scores. You can see this bell-shaped trend as flights are increasing from 1959 to 1963 and decreasing from 1999 to 2003.

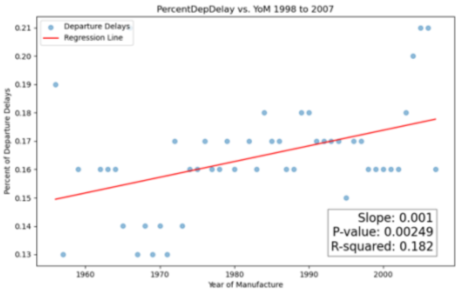
We calculated the delay frequency for each plane, identified by its Year of Manufacture (YoM) and tail number, as a proportion of its total flights during the specified period **to normalize our data, so that it is more comparable**. This approach quantifies delay occurrences relative to the number of flights for each unique plane, ensuring our analysis specifically examines the rate at which planes experience delays **based on their age.**

Visualizations and Results

Analysis 1: Using our data from Table 2, we then regress the Percentage columns against the Year of Manufacture (See Below)

Python



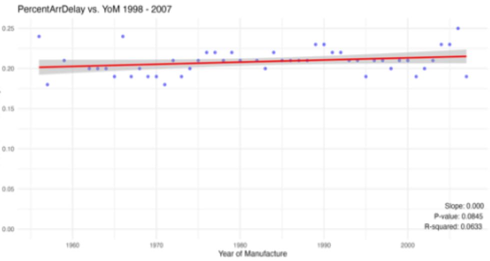


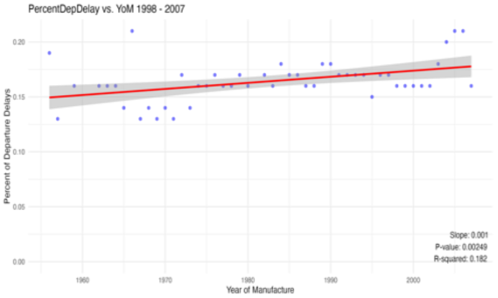
For **PercentArrDelay vs YoM,** the slope of zero, high p-value, and extremely low R-squared value collectively indicate that the **Year of Manufacture does not significantly affect the percent of arrival delays**, and it accounts for almost none of the variation in arrival delays.

For **PercentDepDelay vs YoM,** the results indicate a statistically significant relationship between the independent and dependent variables, as evidenced by the low p-value but the slope suggests that while the relationship is statistically significant, the effect size is minimal. This means that even though changes in X are associated with changes in Y, these changes are very slight.

Analysis 1 suggests that **older planes do not suffer more delays than younger planes.**

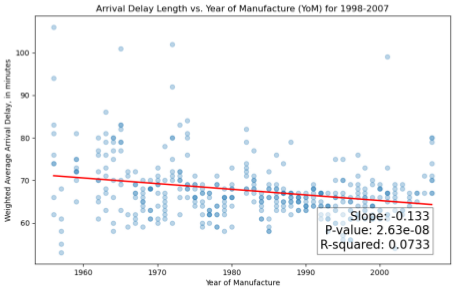
R

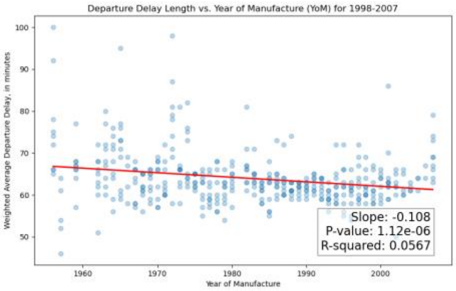




Analysis 2: Using our data from Table 2, we can also regress a measure of delay severity against YoM (See Below) where our measure is Average Delay Length weighted by number of flights

Python

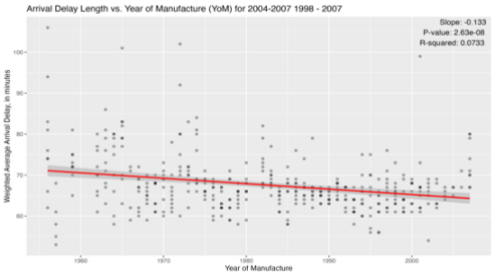


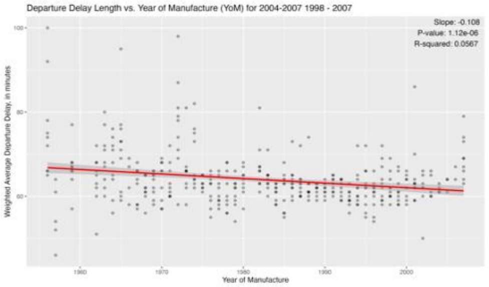


The data for both graphs suggests that newer planes (higher Year of Manufacture) tend to have shorter Arrival and Departure Delay Lengths, and **this relationship is statistically significant**. Though the Year of Manufacture alone does not explain the majority of the variance in Departure Delay Lengths (given the low R-squared values). There are other factors that explain Delay Length that need further investigation and discovery.

Analysis 2 suggests that **older planes suffer longer delays.**

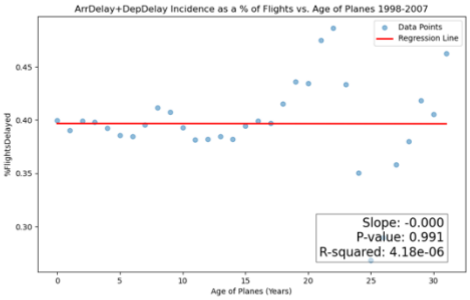
R





Analysis 3: Finally, we also use the issue_date variable from the provided 'planes' table to work out a grouping of planes by 'Age' in years (see Python and R notebooks Part 5)

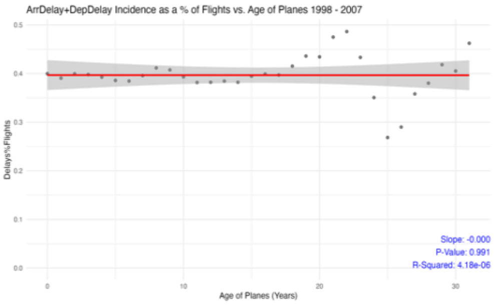
Python



The regression analysis of AggregateDelayIncidence as a percentage of Flights versus Age of Planes reveals that there is no statistically significant relationship between the age of planes and the percentage of flight delays, as indicated by a slope of 0 and a p-value of 0.991. Furthermore, the R-squared value of **$4.18*10^{-6}$** suggests that the age of planes explains little to no variation in delay incidence, underscoring the minimal impact of plane age on delay frequencies.

Analysis 3 suggests that **older planes do not suffer more delays than younger planes**

R



Conclusion

Overall, considering all the evidence we have gathered from Analyses 1, 2 and 3, older planes do not suffer more delays. However, the slight increase in departure delays for younger planes might be due to newer technology requiring additional maintenance checks for improved safety of the newer planes. This is supported by research by Panish, Shea and Ravipudi LLP who show that aviation crashes have been decreasing year on year from 1982 to 2019. They found that majority of accidents happen during takeoff and landing, hence the need for robust departure checks in newer, larger capacity aircraft. Our findings suggest planes are becoming safer but this comes at the expense of more frequent departure delays (but shorter delays overall - as per Analysis 2). Delays in minutes (Severity of Delays) are inversely proportional to YoM, which suggests that over time, airports, airlines and airplanes are becoming more efficient at handling delays when they do occur. This could be due to technological improvement over time. Improved technology means that errors, when they do occur can be fixed more quickly, albeit tighter regulations and attention to safety means that younger planes are delayed more frequently. More data is needed to test the fidelity of our hypotheses.

Part 2C

Fitting a logistic regression model for the probability of diverted US flights, for each year

Libraries

```
#python
import sqlite3
import pandas as pd #optimized for working with large data sets
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, roc_curve, auc
from sklearn.preprocessing import OneHotEncoder, MaxAbsScaler
import matplotlib.pyplot as plt
from scipy.sparse import csr_matrix, hstack
import gc  # Garbage collector which helped me with memory management on my local machine
import numpy as np


#r
library(DBI)
library(RSQLite)
library(dplyr)
library(data.table) #fread() is very useful reading large data quickly
library(fastDummies) # for one-hot encoding
library(caret) # for training models
library(pROC)
```

Analysis Plan

Our objective is to model the likelihood of flight diversions in the US for each year within our dataset using a logistic regression model - glm() in R and LogisticRegression() in Python

- The **target variable** for our model is: 'Diverted', a binary indicator (0/1) signifying whether a flight was diverted. Which is well suited for a logistic function tbat can be used to measure the likelihood of a binary outcome.

The analysis utilizes the following **predictors**:

- **Temporal Features:** Month, Day of Week, which help capture seasonal and weekly patterns in flight diversions.
- **Scheduled Times:** CRSArrTime and CRSDepTime, indicating scheduled arrival and departure times, respectively, to understand if timing influences diversion probability.
- **Carrier Influence (Custom Feature):** %CarrierDiverted, a measure reflecting the historical diversion frequency associated with each carrier.
- **Flight Distance:** Distance between departure and arrival airports, to examine the effect of flight length on diversion likelihood.
- **Airport-Specific Diversion and Delay History:**
    - **Airport DivertedCount_Dest (%) (Custom Feature):** The proportion of flights diverted at the destination airport, providing insight into the destination airport's role in diversions.
    - **DepDelayCount_Dest (%) (Custom Feature):** The percentage of departure delays at the destination airport, which may influence diversion decisions.
    - **ArrDelayCount_Dest (%) (Custom Feature):** The percentage of arrival delays at the destination airport, further contextualizing the destination airport's impact.

Data Source

comp97to07.db

Data Cleaning and Shaping

Feature Preparation

7. Utilized an SQLite database **`comp97to07.db`** to manage and access flight data spanning 1998 to 2007, incorporating tables for each year alongside relevant flight attributes.
8. Several data wrangling and cleaning operations were performed to prepare the dataset for logistic regression modeling.
    - **Data Cleaning:** DataFrames were concatenated across years, null values were dropped, and numeric conversions are applied to ensure data type consistency. I cleaned and processed the combined data set, converting appropriate fields to numeric types to ensure data quality.
    - **Retaining relevant data:** I made sure to only include rows that had at least one flight with a confirmed diversion. We did not track flights that did not have a diversion in our years of interest.
    - **It was discovered that the total data size of our dataset after cleaning is** 55,767,925 rows of data! Diverted incidents numbered at only 134,793! (See Part 5 in notebook 2C for R and Python on why this is a valid case for down sampling)
        - The scarcity of the target variable in our dataset means that our logistic regression model has limited examples of the minority class to learn from. This scarcity challenges the model's ability to accurately predict rare events, such as flight diversions. Hence, we determined it important to down sample rows where Diverted=0 to balance classes.
9. **Feature Engineering**
    - I selected **9 features** for our logistic regression model, the task specified that I should use 'as many features as possible', but at some stage, **if features are too many this leads to overfitting the data.**
    - We chose the following features and applied the following pre-processing steps to scale the data appropriately, given the sensitivity of logistic regression to scale issues:
        - **Month** (raw score…1,2,3………,12) …`remains as is`
        - **DayofWeek**…`OneHotEncoder in Python, but data processed raw in R.`
        - **CRSDepTime and CRSArrTime** converted to a decimal (0.000 is midnight, 0.0100 is 1 AM, 0.13 is 1 PM, 0.2359 IS 23:59PM) …

          ```
          #We used this SQL query in python and R…
          CAST(y.CRSArrTime AS REAL)/10000 AS CRSArrTime,
          CAST(y.CRSDepTime AS REAL)/10000 AS CRSDepTime
          ```
        - **Distance,** we determined that the information contained in coordinates is actually already embedded in the distance parameter, so I decided to leave out Coordinates data points, as well as Origin and Destination data points. Distance was standardized using MinMaxScaler.

          ```
          features_to_normalize = ['Distance']
          scaler = MinMaxScaler(feature_range=(0, 1))
          df[features_to_normalize] = scaler.fit_transform(df[features_to_normalize])
          ```
        - **Carrier Diversion Count** (as a percentage of total diversions), Carrier information was initially one hot encoded, but this added to the dimensionality of our program so we changed the approach to include a rate of incidence of diversions for each carrier.
        - Airport **DivertedCount_Dest** (as a percentage), where we took the rate of incidence of diversions per airport.
        - **DepDelayCount_Dest**(as a percentage), where we took departure delay count as a fraction of total flights.
        - **ArrDelayCount_Dest** (as a percentage), where we took arrival delay count as a fraction of total flights.
    - The target range of values of our features was between 0 and 1 (except for Month and DayofWeek), this is so that we have appropriately scaled our features.
10. **Merging and Renaming:** The resulting custom features were merged with the main data set based on airport codes. This is to enrich the main dataset with detailed diversion and delay attributes. Columns were renamed for clarity and unnecessary columns (e.g., 'Origin', 'Dest') were dropped after their information was integrated into new features.

Model training and visualizing the coefficients across years

11. Upon fitting the logistic regression model for each year, I aggregated and **displayed the model coefficients using bar plots**. It needs to be noted that the approaches for Python and R to achieve the same outputs is slightly different. This is because R and Python have different packages that are optimized for working with large data sets.
12. In python, I tweaked hyperparameters such as **`max_iter`** and **`tol`** to find a balance between computational efficiency and prediction accuracy.
13. Treatment of DayOfWeek
    - In python we used One-Hot Encoding and obtained an average coefficient from the resulting columns
    - In R we did not use One-Hot-Encoding as the increased dimensionality presents diminishing returns to complexity, since R is already comparatively computationally inefficient when it comes to machine learning.
14. In python
    - Downsampled the dataset to fix the class imbalance
    - Set a **tolerance (tol)** of 0.1 to expedite the model's convergence, effectively decreasing training time without substantially impacting accuracy.
    - Chose the **'saga' solver** for its efficiency on large datasets, benefiting from its ability to handle 'balanced' class weights for my logistic regression model.
    - Applied a **'balanced' class weight** to correct for the imbalanced distribution of our target variable, ensuring equitable representation of both classes in the model training process.
15. In R
    - Hyperparameters like tol, solver and balanced were not available in glm() in R, so we only used down sampling in R to correct the class imbalance.

Impact of Scale on Model Interpretability

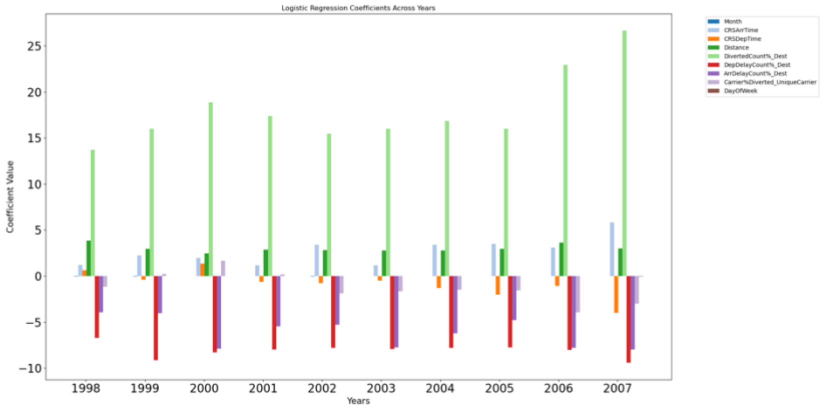
- In models that use gradient descent as the optimization technique, including logistic regressions with **`solver='saga'`**, features with different scales can make the training process less stable and slower. Large feature values might dominate the gradient updates, causing smaller feature values to have less influence on the model.
- I realized that normalizing or standardizing features can help mitigate these issues by ensuring that all features contribute equally to the model training process. So, I set about choosing features based on whether I could get them to be as standardized as possible. For features like distance, applying a normalization or standardization technique helped align their scales with the rest of my features. This prevented any single feature from disproportionately influencing the model due to its scale, which happened in an earlier version of my code (this version was commented out in my notebooks)
- My logistic regression model, as configured, is using l2 regularization (by default) and SAGA optimization in my python notebooks. Standardizing my features improves optimization efficiency, regularization effectiveness, and the interpretability of feature importance, contributing to better overall model performance and reliability.

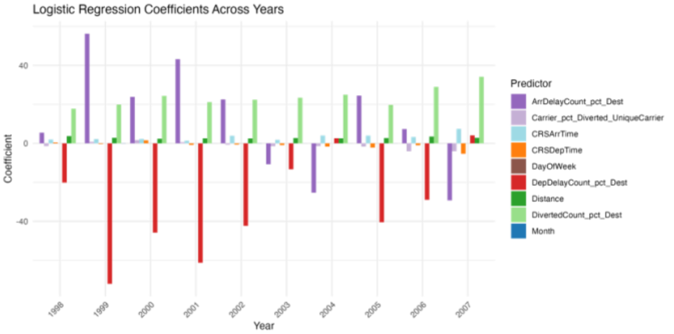
Some important questions and answers

- **Why did we use fread() instead of read() in R?** Because fread is faster at reading large datasets!
- **Why is there a slight difference in naming convention in our R notebook?** R does not take kindly to `%` characters in the names of variables, so we used the shorthand `pct` to mean "percentage". I had to create a separate `.csv` file, so that when I repeat running my code, R and Python do not get confused!

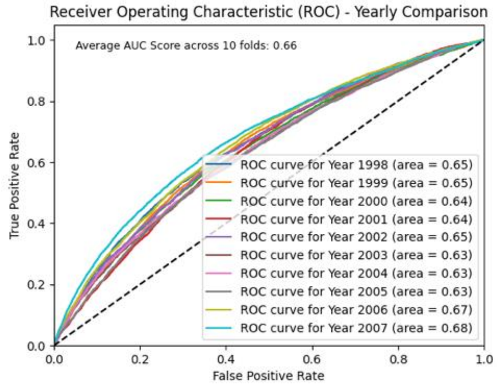
Results: Visualization of the Logistic Regression Coefficients and ROC across years

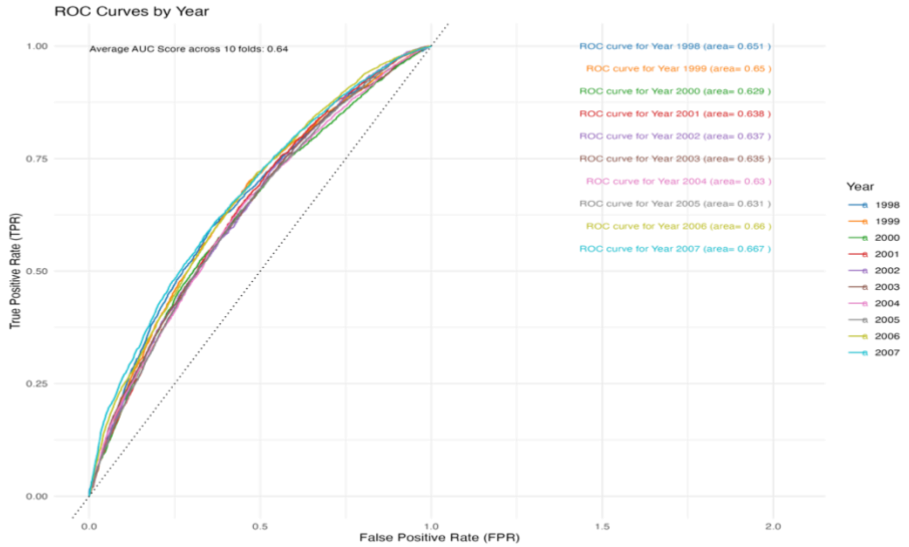
Python **R**





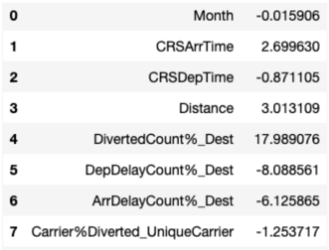
Python **R**



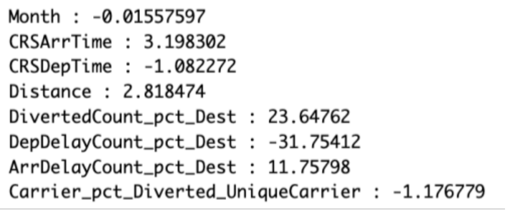


Conclusion

Python – Average Coefficients (see Part 7 of the 2C_python.ipynb notebook) **R – Average Coefficients** (see Part 7 of the R_2C_notebook.Rmd notebook)



0	Month	-0.015906
1	CRSArrTime	2.699630
2	CRSDepTime	-0.871105
3	Distance	3.013109
4	DivertedCount%_Dest	17.989076
5	DepDelayCount%_Dest	-8.088561
6	ArrDelayCount%_Dest	-6.125865
7	Carrier%Diverted_UniqueCarrier	-1.253717



```
Month : -0.01557597
CRSArrTime : 3.198302
CRSDepTime : -1.082272
Distance : 2.818474
DivertedCount_pct_Dest : 23.64762
DepDelayCount_pct_Dest : -31.75412
ArrDelayCount_pct_Dest : 11.75798
Carrier_pct_Diverted_UniqueCarrier : -1.176779
```

The Average values of our Coefficients lead us to draw the following conclusions:

- **DivertedCount%_Dest** (Python) / **DivertedCount_pct_Dest** (R) stands out as the most influential predictor, with higher coefficients indicating a strong positive relationship. This suggests that flights heading to destinations with a higher historical diversion rate are more likely to be diverted.
- **DepDelayCount%_Dest** (Python) / **DepDelayCount_pct_Dest** (R) shows a notable negative coefficient in the R model, contrary to its positive impact in Python, indicating differences in model sensitivity or data handling. This might imply that, under certain circumstances, departure delays could reduce the likelihood of diversion, potentially due to more cautious planning in delay-prone situations.
- **Distance** shows a positive coefficient in both models, suggesting longer flights have a slightly higher likelihood of diversion, possibly due to the increased chance of encountering variable conditions over longer routes.
- **Coefficients for time-related features** (**CRSArrTime**, **CRSDepTime**) and **Carrier%Diverted_UniqueCarrier** are relatively consistent across both models, underscoring their stable influence on the likelihood of diversion. The Month feature shows minimal influence in both models, indicating the time of year may not be a significant predictor of flight diversion when other factors are considered.
- This is supported by Part 7 in the Python and R notebooks (2C_python.ipynb and 2C_R_notebook.Rmd) where I calculated an average of all the coefficients of interest in my model.

ROC Curves and K-Fold Validation: The ROC curves for all the years, shows a higher true positive rate, which suggests the logistic regressions models are better than chance (AUC=0.5) at predicting flight diversions using our selected Features. Moreover, the results of the cross validations, summarized by the average AUC scores (0.66 in python and 0.64 in R) suggest that the models have moderate predictive power, but neither is particularly close to perfect prediction (a perfect prediction would be equal to 1). The slight difference in AUC might be due to implementation details between the logistic regression models in Python and R.

Why is there a difference in the coefficients reported in R and Python?
This is because Pandas and Scikit-learn utilize vectorized operations and efficient C++ backends (through Cython), which can lead to more memory-efficient execution compared to certain R operations that fall back to less efficient methods when dealing with very large datasets. We had to shrink down the dataset size, to be able to train our model, test it, and plot the coefficients as planned. Furthermore glm() in R lacks some of the hyperparameters that we have available in LogisticRegression() in Python.

Can we improve the performance of our models?
We can use a GridSearchCV() in python to obtain the best hyperparameter combination to take our model to the next level (See Part 8 of the python notebook). In R we can use glmnet(), in combination to the 'trainControl' and 'train' functions instead of glm(). The glmnet function itself fits a sequence of models for different values of the regularization parameter, λ (lambda). When combined with cv.glmnet, which performs cross-validation, it's effectively tuning the hyperparameter (λ) to find the model that generalizes best (See Part 8).

Final Remarks and Challenges

```
Recurring Issues:
{In R notebooks} Error: vector memory exhausted (limit reached?)
And
{In Python notebooks} Kernel Crashed: …

The main cause was running out of local system memory. Future solutions are using libraries that support job parallelization, or uploading the project to Google Colab.
```

References

Objective 1A…Gilks, W.R., Richardson, S., and Spiegelhalter, D.J., eds. (1996). *Markov Chain Monte Carlo in Practice*. Chapman & Hall/CRC….”Metropolis-Hastings generates a proposal distribution from which we then start to piece together the complete target distribution”

Objective 1B….Gelman, A., & Rubin, D. B. (1992). Inference from Iterative Simulation Using Multiple Sequences. Statistical Science, 7(4), 457-472….

Objective 1B….Dootika Vats, Christina Knudson “Revisiting the Gelman–Rubin Diagnostic,” Statistical Science, Statist. Sci. 36(4), 518-529, (November 2021)

Objective 2B…https://www.panish.law/aviation_accident_statistics.html - The Aviation Crashes and Injuries Statistics show that aviation crashes have been decreasing year on year from 1982 to 2019. This suggests planes are becoming safer, even though flights are increasing.

Objective 2B…https://en.wikipedia.org/wiki/Flight_cancellation_and_delay#- “The United States Federal Aviation Administration (FAA) considers a flight to be delayed when it is 15 minutes later than its scheduled time.”

Objective 2C…https://docs.python.org/3/index.html - for when my code kept running into kernel errors in python.