Berliner Hochschule für Technik (BHT)

Course Name: Machine Learning-II

Project Title: NYC Bus Delay Time Prediction

Submission Date: 09th January 2024

Submitted By

Ahmed Dider Rahat	Tania Sultana	Rezaul Karim Mamun
ID: 916146	ID: 931031	ID: 915278

Introduction

In this report, we present our exploration into predicting bus delays in New York City using machine learning techniques. Our project began with a rich dataset from Kaggle, capturing the information of NYC's traffic. From this, we carefully selected a representative sample of 2000 rows. After preprocessing the data to refine and structure it, we conducted an exploratory data analysis to uncover underlying patterns and relationships. Our approach included splitting the data into an 80/20 train-test ratio and applying target encoding to the categorical features. The heart of our analysis involved developing and tuning two machine learning models: Lasso regression and Generalized Additive Models (GAM), aimed at accurately predicting bus delay times.

Data set

Introduce the Dataset

The New York City Bus Dataset is a rich source of real-time and historical data related to the movement of buses across the city's extensive public transportation network. This dataset, provided by the Metropolitan Transportation Authority (MTA), offers a unique opportunity for machine learning practitioners to develop innovative solutions for improving public transportation efficiency and passenger experience.

Sample Data

The dataset has been taken from Kaggle.com. There are a total number of 26,524,144 rows in the original dataset. So, we only pick 2000 observations randomly and store the new data for our further analysis.

Structure and key features

The final dataset comprises 2000 observations and 17 variables that capture various aspects of bus operations, including:

Column name	Description	Data type
RecordedAtTime	Timestamp indicating the time when the bus data was recorded.	DateTime

Column name	Description	Data type
DirectionRef	Indicates the direction of travel for the bus, either northbound, southbound, eastbound, or westbound.	string
PublishedLineName	Name of the bus route.	string
OriginName	Name of the origin stop for the bus.	string
OriginLat	Latitude of the origin stops.	Numeric
OriginLong	Longitude of the origin stops.	Numeric
DestinationName	Name of the destination stop for the bus.	string
DestinationLat	Latitude of the destination stop.	Numeric
DestinationLong	Longitude of the bus.	Numeric
VehicleRef	Unique identifier for the bus.	Numeric
VehicleLocation.Latitude	Current latitude of the bus.	Numeric
VehicleLocation.Longitude	Current longitude of the bus.	Numeric
NextStopPointName	Name of the next stop for the bus.	string
ArrivalProximityText	Indicates the proximity of the bus to the next stop, such as "arriving soon," "at stop," or "approaching."	string
DistanceFromStop	Distance between the bus and the next stop.	Numeric
ExpectedArrivalTime	Estimated time at which the bus is expected to reach the next stop.	DateTime
ScheduledArrivalTime	Scheduled time at which the bus is expected to reach the next stop.	DateTime

Data Preparation

Data preparation, also known as data preprocessing, is a crucial step in the machine learning (ML) process. It involves cleaning, transforming, and organizing raw data to make it suitable for training and evaluating ML models. The quality and accuracy of the data directly impact the performance of the ML model.

The specific data preparation steps for the New York City Bus Dataset will vary depending on the specific machine learning task and the desired outcome. However, some general data preparation steps that may be applicable include.

Handling Missing Values

First, calculate the percentage of missing values in each column of the data frame. From the following missing values we visualize the columns which have missing values.

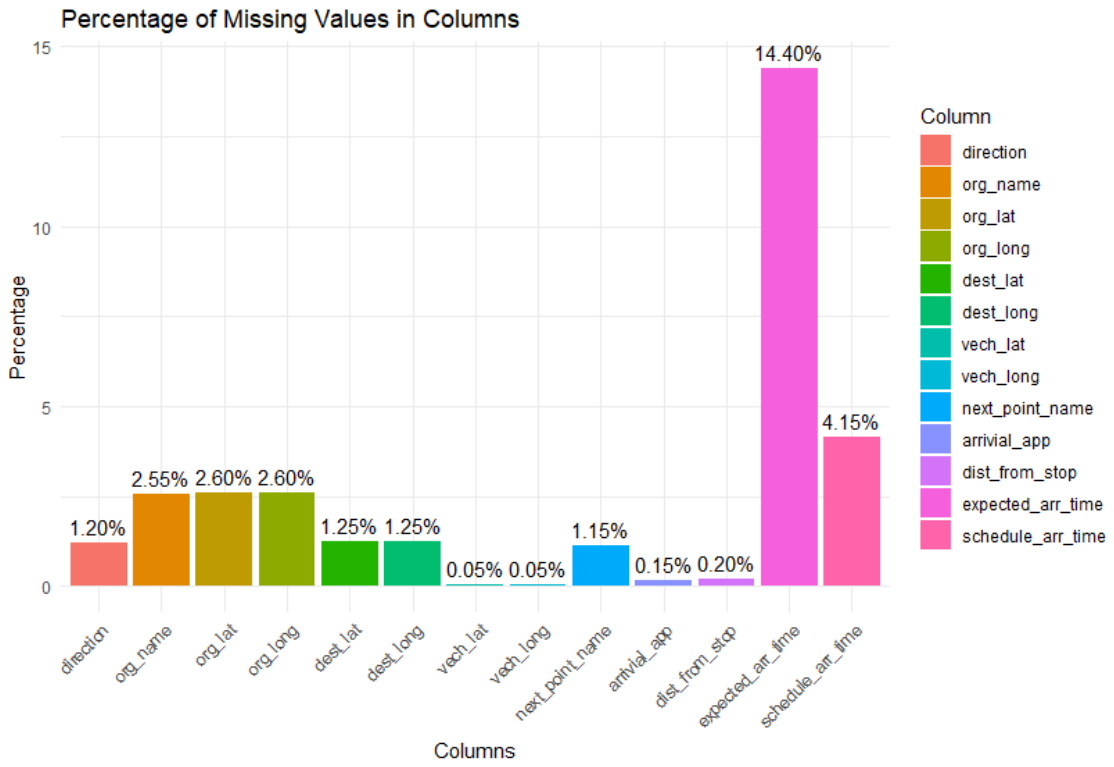


Fig-01: Percentage of missing values in different columns

In the dataset there are a total 332 numbers of missing values found. From the visualization we can see that, In "expected_arr_time" and "schedule_arr_time" columns we get a maximum number of missing values 14.40% and 4.15% respectively. These two of them need to calculate our predicted value delay_time. So, we decided to remove them. After removing the null values we see that 331 rows are eliminated and we have a total 1,669 rows to perform our further operation. As only one row remains with the missing values, we also remove it from our dataset to get a perfect dataset without any missing value. So, our final dataset contains **1,668 rows**.

Calculate Delay Time

Basically, the delay time is the time difference between "schedule_arr_time" and "expected_arr_time". We do this calculation using the following steps:

1. Firstly, we convert the "expected_arr_time" into date-time format and it's converted without any problem.

```
 [1] "2017-12-14 14:29:53 CET"  "2017-08-17 05:40:44 CEST" "2017-12-12 11:28:05 CET"
 [5] "2017-12-22 16:53:55 CET"  "2017-08-04 08:49:22 CEST" "2017-12-01 22:09:17 CET"
 [9] "2017-10-01 18:35:52 CEST" "2017-08-22 12:21:41 CEST" "2017-12-11 15:35:49 CET"
[13] "2017-10-24 16:51:18 CEST" "2017-10-27 07:45:23 CEST" "2017-10-18 07:58:20 CEST"
[17] "2017-12-07 18:00:08 CET"  "2017-06-08 15:26:49 CEST" "2017-06-30 16:03:45 CEST"
[21] "2017-06-09 14:07:48 CEST" "2017-06-27 08:06:52 CEST" "2017-08-19 19:06:44 CEST"
[25] "2017-08-05 16:11:52 CEST" "2017-10-28 11:58:11 CEST" "2017-10-23 14:31:11 CEST"
```

Fig-02: Snapshot of expected_arr_time after converting

2. Secondly, we try to convert the "schedule_arr_time" into date-time format and got error while perform the operation:

```
  [1] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
 [39] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
 [77] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
[115] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
[153] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
[191] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
[229] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
```

Fig-03: Snapshot of schedule_arr_time after converting

3. We then further investigate the issue. We see some of the rows have different kinds of values in the hour position. The hour should be between 00-23 but in some cases it becomes 24 or 25.

	expected_arr_time	schedule_arr_time
10	2017-08-07 00:32:54	24:31:00
31	2017-06-20 00:35:08	24:25:28
173	2017-10-24 00:16:46	24:18:12
211	2017-12-27 00:27:48	24:29:53
296	2017-10-05 00:15:03	24:10:41
370	2017-06-23 00:49:29	24:43:23
385	2017-12-30 00:12:55	24:09:29
420	2017-12-04 00:21:34	24:15:40
457	2017-08-03 01:16:17	25:16:40
480	2017-08-09 00:06:35	24:06:45
523	2017-12-09 00:11:41	24:04:00
524	2017-10-17 01:56:45	25:21:01

Fig-04: Snapshot of erroneous schedule_arr_time

4. We then modify those shcedule_arr_time 24 to 00 and 25 to 1. After that, we calculate the delay time in minutes. From the below visualization we can see that there are some extreme outliers found where the delay time is less than -1000 minutes. So, we then again investigate the issue.

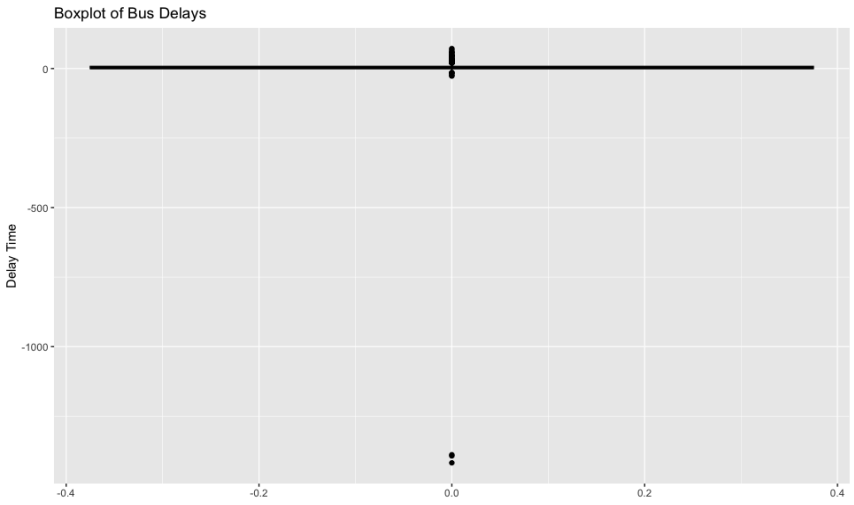


Fig-05: Distribution of Delay-time

5. For this analysis, we pick the rows which have a delay time < -100. The huge negative values occurred because of the following issue:

    a. We don't have a date for the scheduled arrival time, so we get the date from the expected time.

    b. Now if we pick the date from the expected time, sometimes the bus came earlier like 11.50 PM on the same day but its scheduled time was 12.05 AM. So, this means the bus came 15 minutes earlier but in the previous day

c. So, in that case we need to pick one day before the expected date to get the original date.

	expected_arr_time	schedule_arr_time	expected_arr_time_tt	schedule_arr_time_tt	delay_mins
748	2017-12-17 00:31:49	23:40:29	2017-12-17 00:31:49	2017-12-17 23:40:29	-1388.667
878	2017-10-08 00:00:52	23:13:54	2017-10-08 00:00:52	2017-10-08 23:13:54	-1393.033
1668	2017-06-11 00:00:46	23:38:42	2017-06-11 00:00:46	2017-06-11 23:38:42	-1417.933

Fig-06: sample of wrong calculated data

6. Finally, we modify those values and calculate the delay again and have the below distribution of delay time.

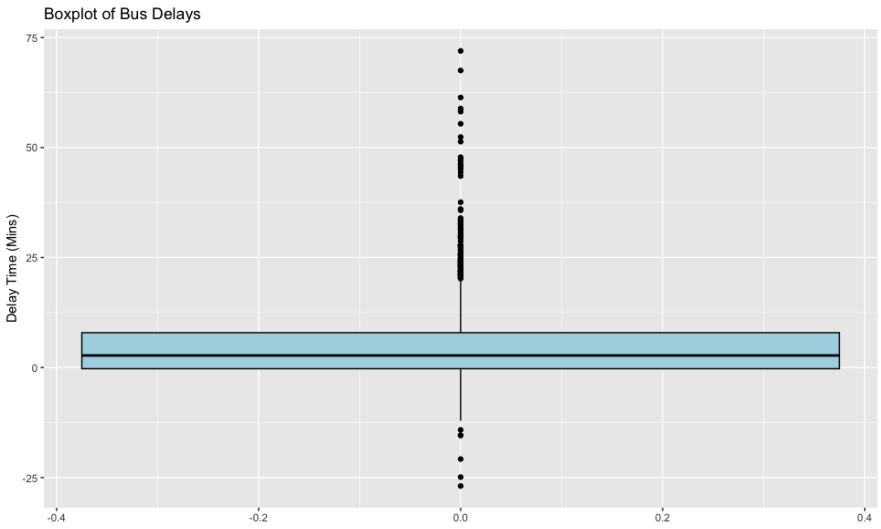


Fig-07: Final distribution of delay time

Calculate Other Time Relevant information

From the scheduled arrival time we derived some of the important time based information.

1. Weekend status: Define the schedule day is weekend or not.
2. Day of Year: Contains an integer (1-365) representing the day of year.
3. Time of Day: It's the integer time frame in minutes [0-1400].

Final Choice of Columns

We don't need the time based features as we have the derived features out of them like delay_mins, day_of_year, weekend_status, and time_of_day. So, we can remove the time

based features like "recorded_at", "expected_arr_time", and "schedule_arr_time" from the dataset. At the end the columns of the dataset:

```
 [1] "direction"      "line_name"      "org_name"       "org_lat"        "org_long"       "dest_name"
 [7] "dest_lat"       "dest_long"      "vech_name"      "vech_lat"       "vech_long"      "next_point_name"
[13] "arrivial_app"   "dist_from_stop" "weekend_status" "day_of_year"    "time_of_day"    "delay_mins"
```

Fig-08: Final columns of the dataset

Exploratory Data Analysis(EDA)

Exploratory Data Analysis (EDA) is an analysis approach that identifies general patterns in the data. These patterns include outliers and features of the data that might be unexpected. In our report we did both Univariate and Bivariate exploratory data analysis.

Univariate Analysis

Univariate analysis is a form of analysis that only involves a single variable. In a practical setting, a univariate analysis means the analysis of a single variable (or column) in a dataset. It is typically the first step to understanding a dataset.

In our project, we thoroughly examine each feature, and now we incorporate univariate analysis into our approach.

1. **Analysis Variable Direction:** The column contains two values representing the directions of the bus route: **"Outbound"** and **"Inbound."** Outbound, denoted by **0,** signifies that the bus is moving away from the center, while Inbound, represented by **1,** indicates the bus is moving towards the center.

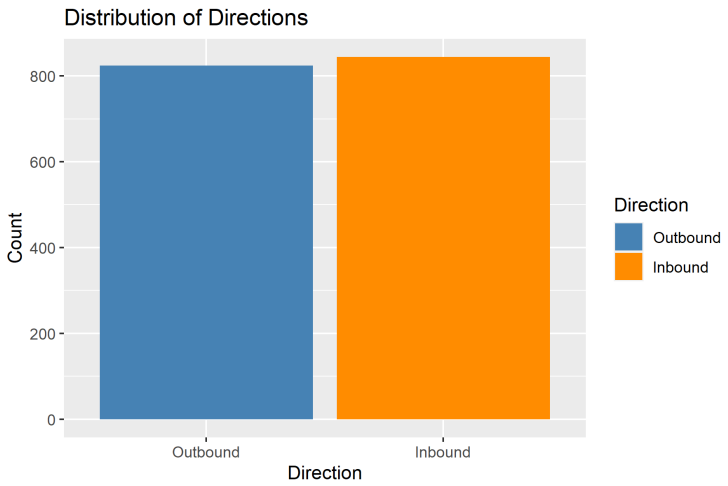


Fig-09: Distribution of Direction

The dataset exhibits an equal distribution between the two distinct directions. There are **824** instances of outbound and **844** instanc- es of inbound.

2. **Line Name:** There are **220** unique line names in total. The top **20** lines account for a frequency percentage of **26.01%**.

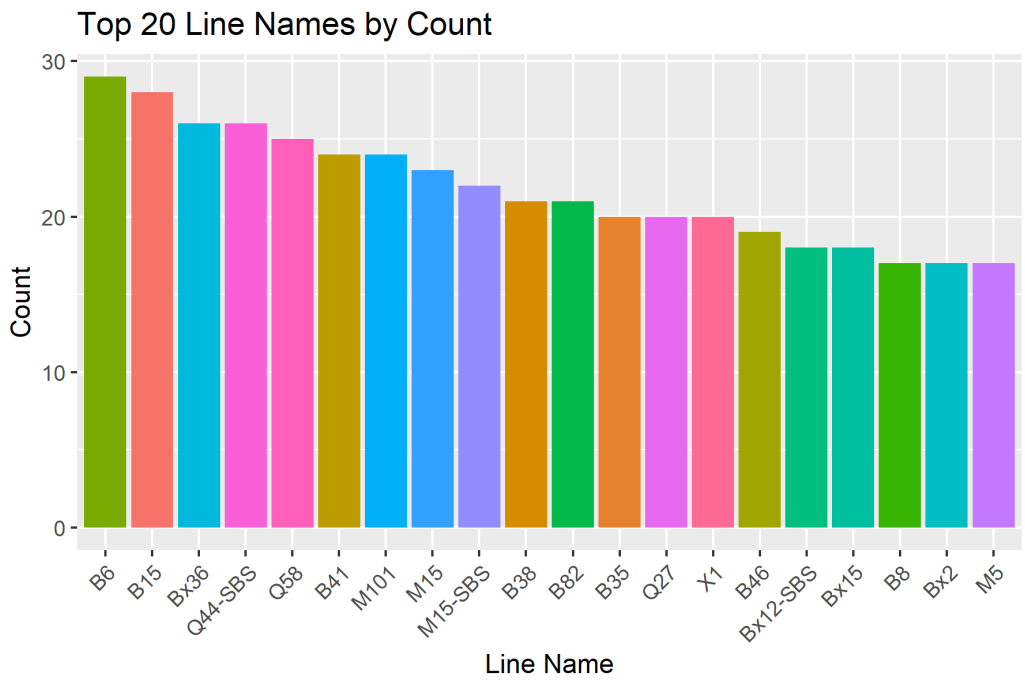


Fig-10: Distribution of the Top 20 Line

Distribution of Counts:

- The average frequency per line name is **7.58**.
- There are **88** line names with counts exceeding this average.

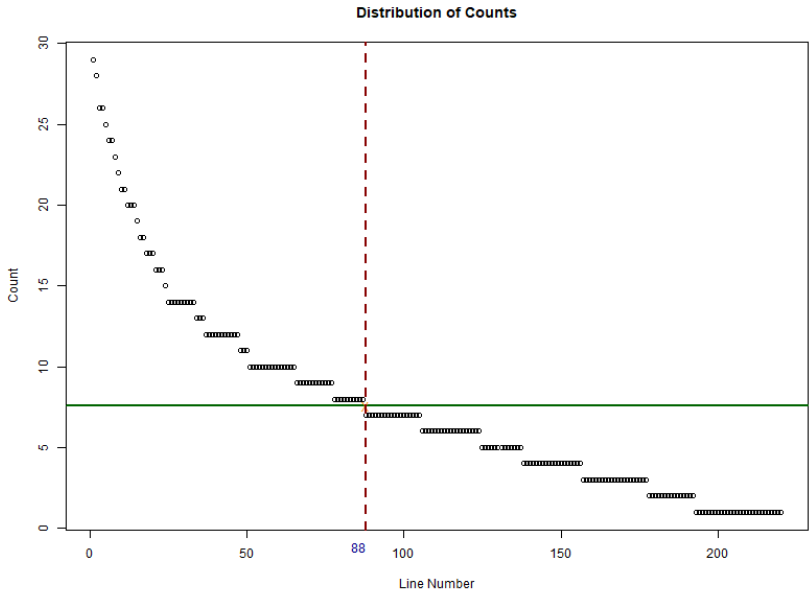


Fig-11: Distribution of the Lines

3. **Origin/Source Name:** There are **350** unique origin names in total. The top 20 origins account for **20.02398%** of the total frequency.

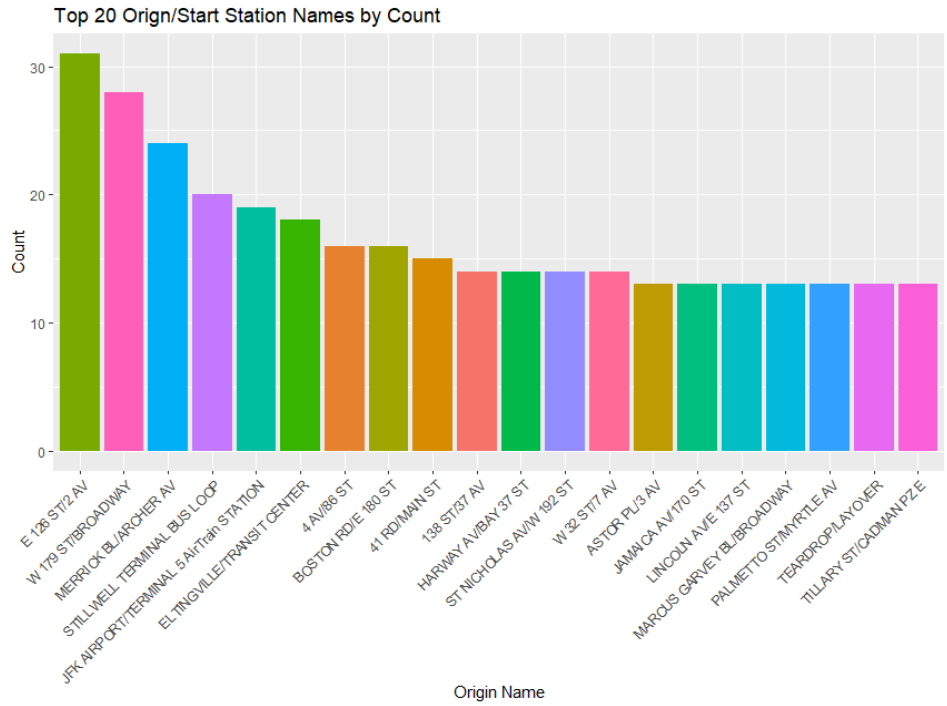


Fig-12: Distribution of the Top 20 origin name

Distribution of Counts:

- On average, each origin name has a frequency of **4.765714**.
- **136** origin names exceed the average count for their occurrences.

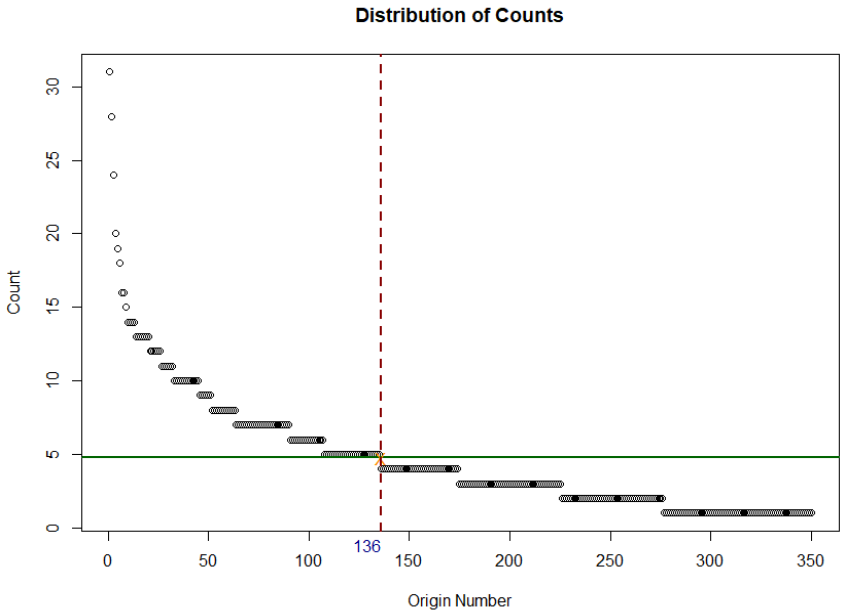


Fig-13: Distribution of the origin

4. **Destination/End station Name:** There are **438** unique destination names in total. The top 20 destinations contribute to **15.35%** of the total frequency.

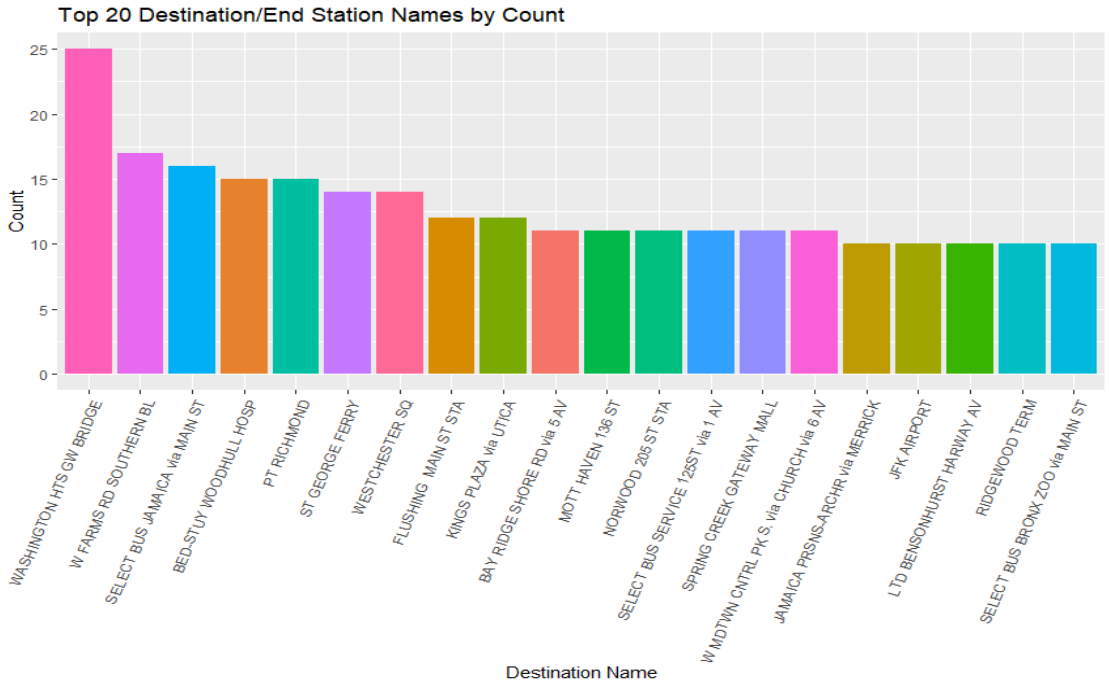


Fig-14: Distribution of the Top 20 destination name

Distribution of Counts:

- Each destination, on average, has a frequency of **3.808219**.
- For **190** destinations, the counts exceed the average frequency.

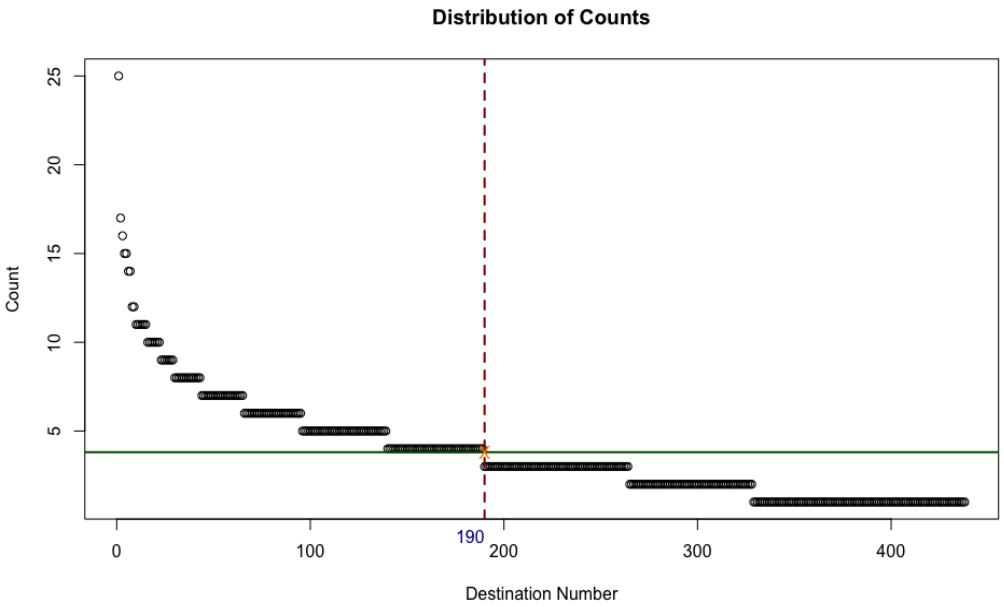


Fig-15: Distribution of the destination

5. **Map Visualization of each routes from source to destination:**

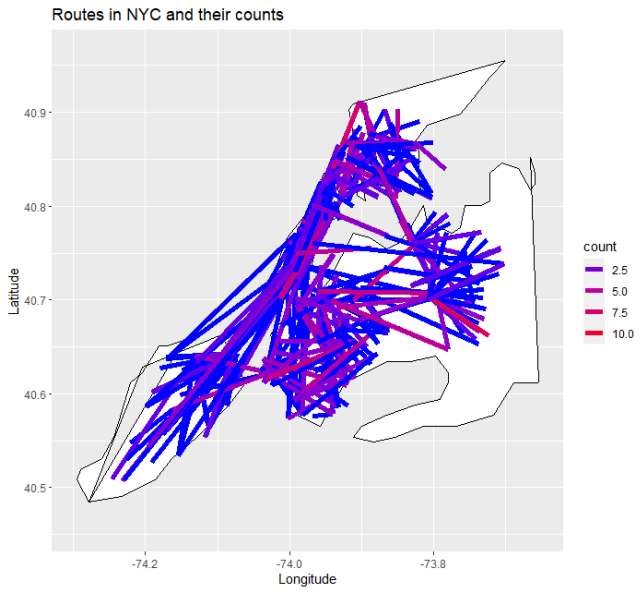


Fig-16: Visualize each route in map

Key Insights:

The visualization appears to be a map of New York City (NYC) showing various routes marked in blue, with their thickness and color intensity representing the count of each route.

6. **Vehicle Name:** The dataset contains **1381** unique vehicle names. The top 20 vehicles contribute to **3.78%** of the total frequency.

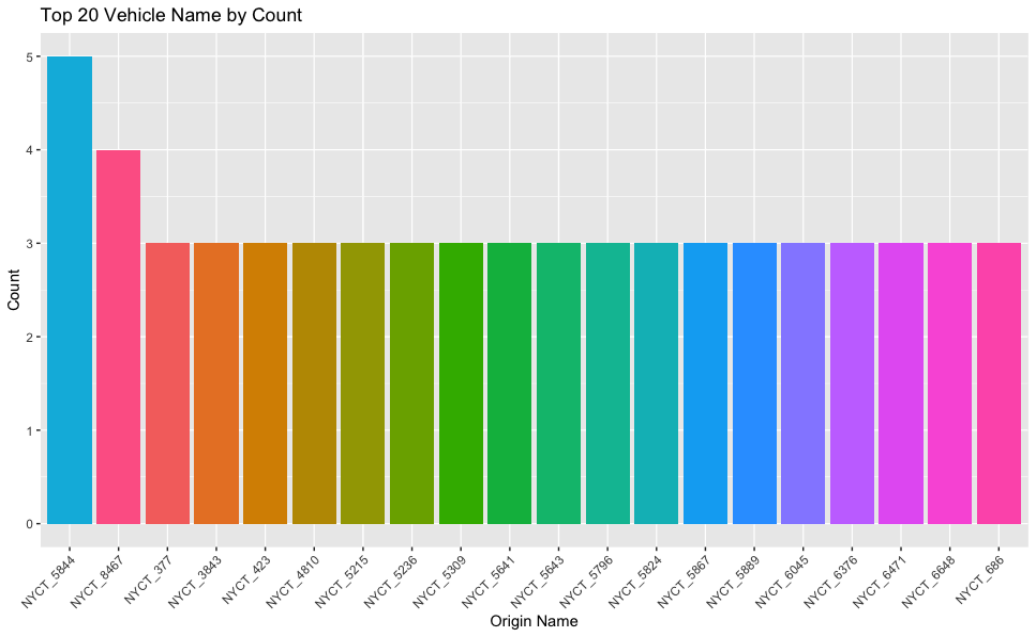


Fig-17: Distribution of the Top 20 vehicle name

Distribution of Counts:

- On average, each vehicle name has a frequency of **1.20782**.
- There are **251** vehicle names with counts exceeding the average frequency.

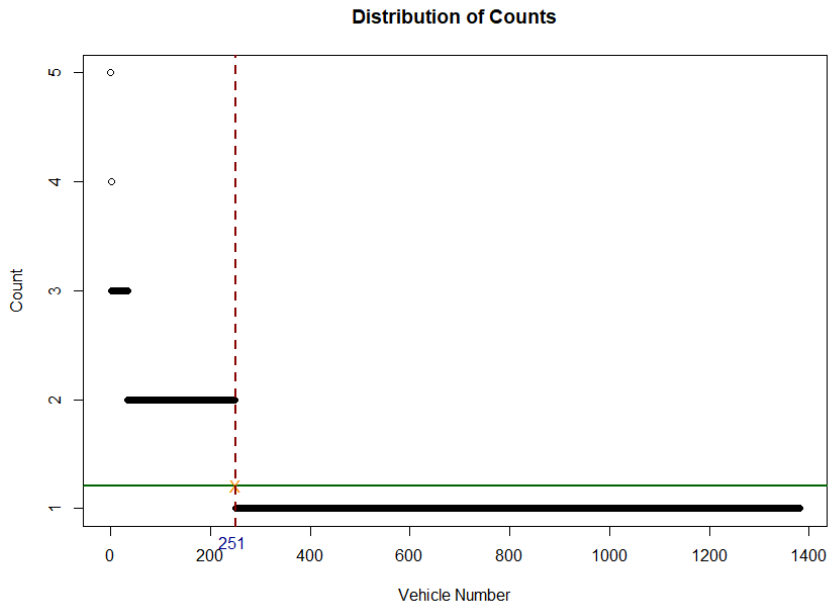


Fig-18: Distribution of the vehicle

7. **Weekend Status:** The "weekend status" variable has two values: **FALSE,** representing weekdays, and **TRUE,** representing weekends. Both categories are evenly distributed in the dataset, with weekdays accounting for nearly **80%** of the counts, while weekends make up the remaining **20%.**

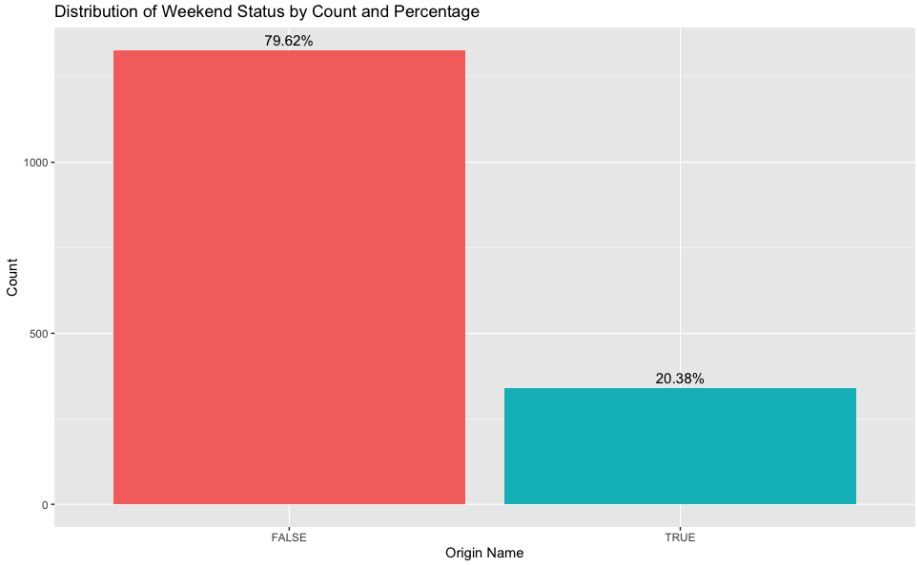
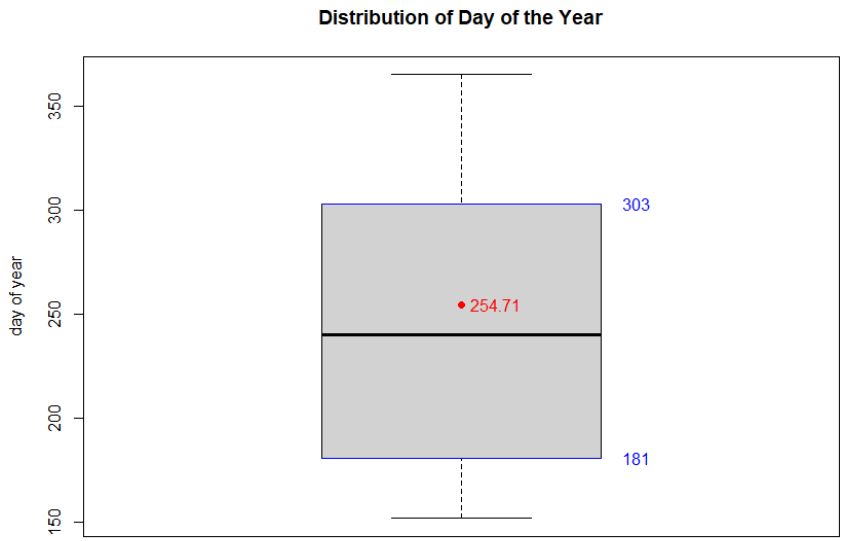


Fig-19: Distribution Weekend Status

8. **Day of the year:**

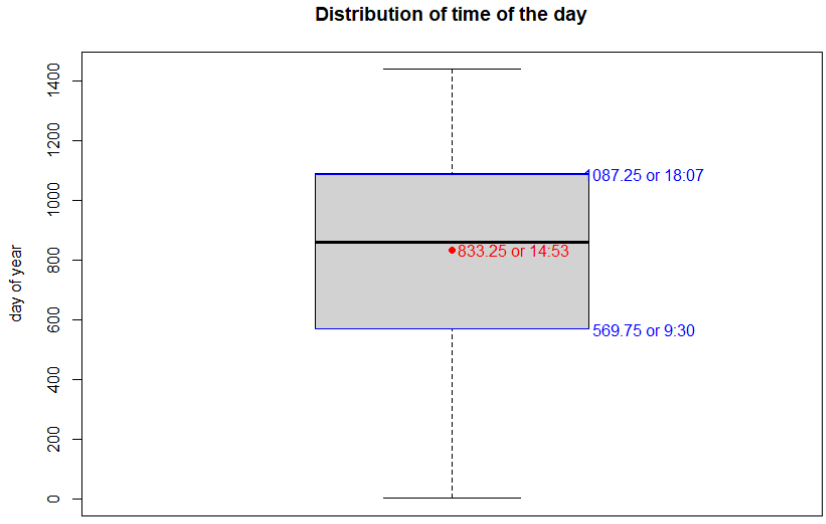


Key Insights:

- Average day of the year is **254.71**.
- **50%** of data lies between **181** to **303**.

Fig-20: Distribution of day of year

9. **Time of the Day:**



Key Insights:

- The peak activity occurs from **09:30** to **18:07**.
- The average time is **14:53**.

Fig-21: Distribution of the time of day

10. **Un-relevance features:** "**next_point_name**", "**arrivial_app**", and "**dist_from_stop**" have no relevance to our analysis so we remove these features.

11. Delay Mins:

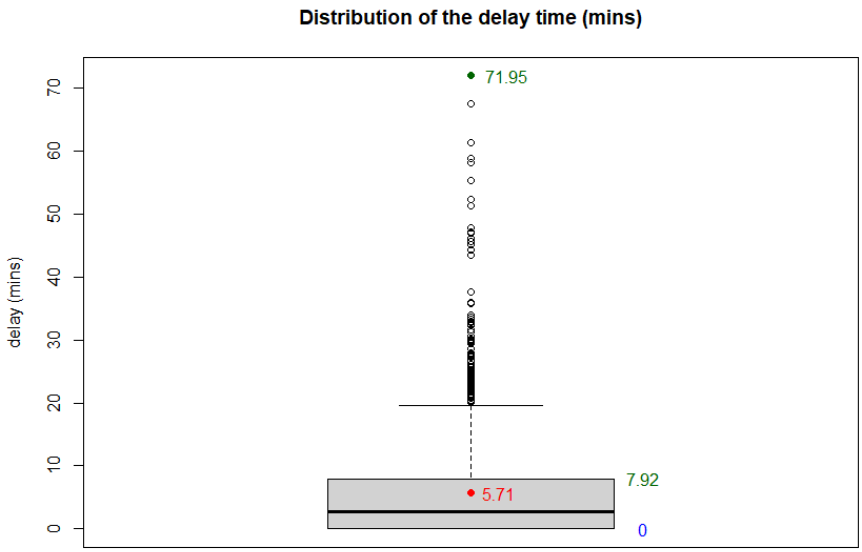


Fig-22: Distribution of delay times in minutes

Key Insights:

- The delay time is distributed from **-26.92 to 71.95**
- Here, the negative delay means the bus arrived early with respect to the scheduled time. So, we consider the early arrival as 0 delay time.
- So, the delay time is distributed from **0** to **71.95**.
- Mean delay time is **5.71**.
- **50%** of the data is concentrated into 0 minutes to **7.92** minutes.
- There are several outliers present in the data.

Bivariate Analysis

Bivariate analysis means the analysis of bivariate data. This is a single statistical analysis that is used to find out the relationship that exists between two value sets. **The variables that are involved are X and Y.**

1. **Direction Column vs Delay Time:**

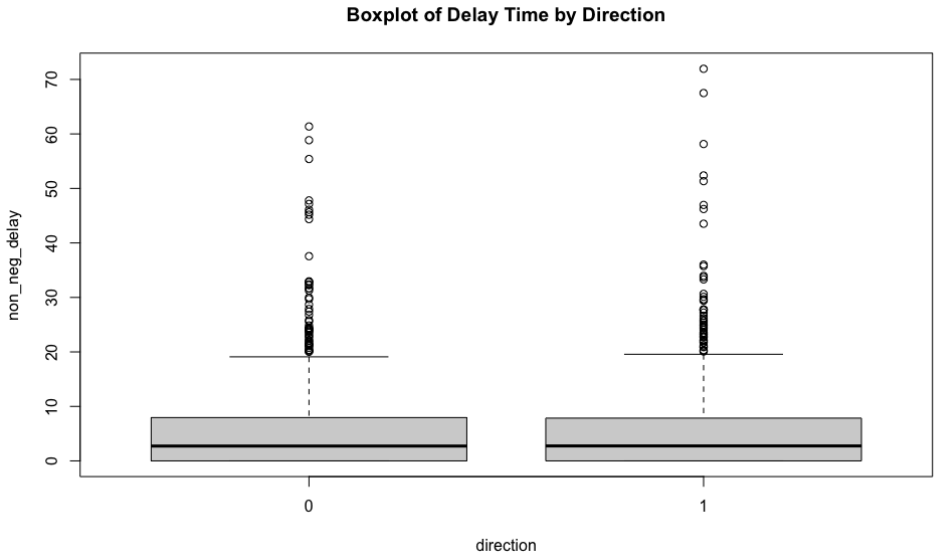


Fig-23: Distribution delay based on direction

Key Insights:

- Both directions exhibit a range of delay times with several outliers indicating instances of significant delay.
- The box for direction '**0**' appears slightly higher, suggesting that the median delay time for this direction is greater than that for direction '**1**'.
- The spread of delay times, as indicated by the interquartile range, seems similar for both directions, but direction '**0**' shows more variability with outliers extending further up.

2. Line Name Column vs Delay Time:

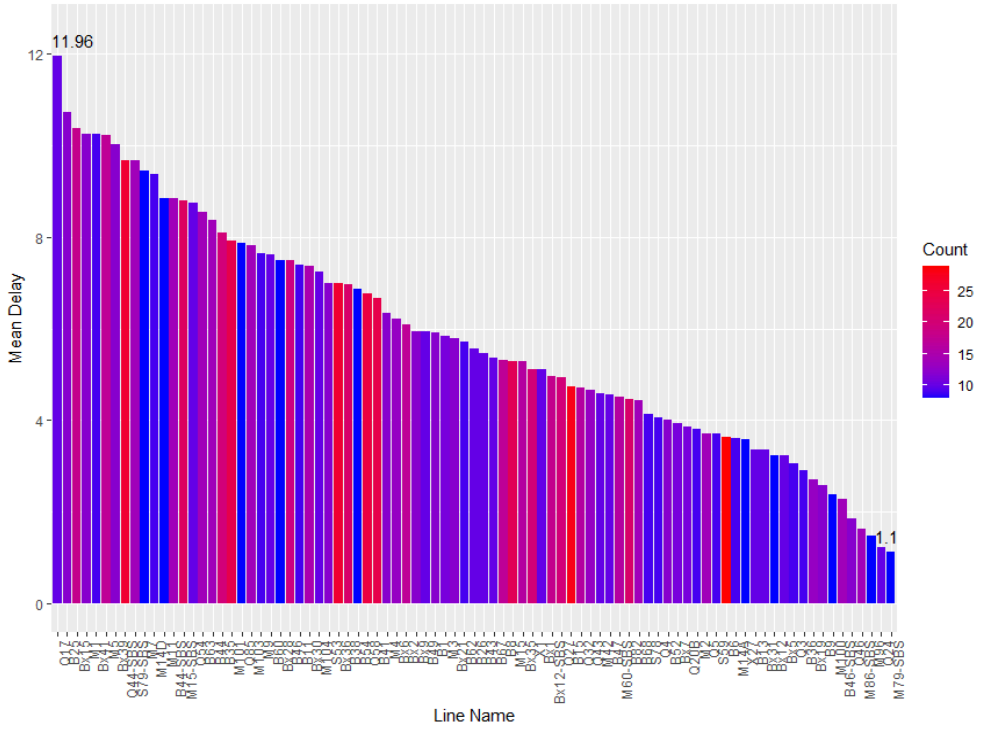


Fig-24: Distribution delay for frequent line

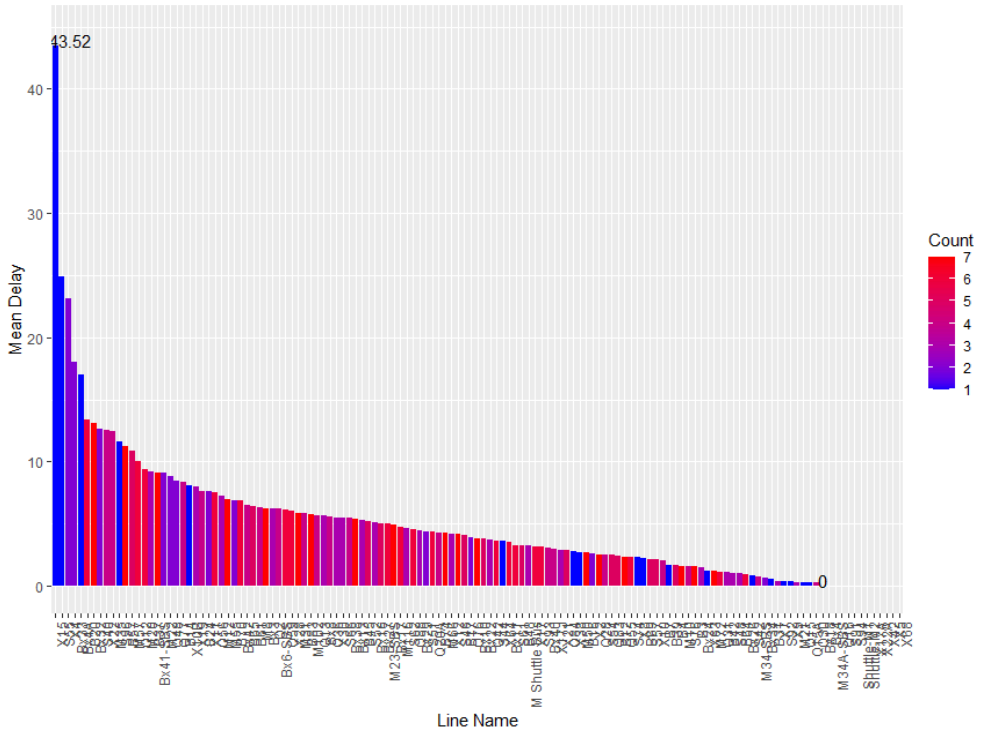


Fig-25: Distribution delay for infrequent line

Key Insights: To understand the behavior we divide the total dataset into two categories-Frequent line (which has frequency above to the avg. frequency) and infrequent lines.

- There are a total number of **87** frequent lines and **133** infrequent lines.
- Frequent bus lines have less variation in the average delay time (11.96 - 1.1).
- Where the less frequent lines have high variation in their average delay time (**43.52 - 0**).

3. Weekend Column vs Delay Time:

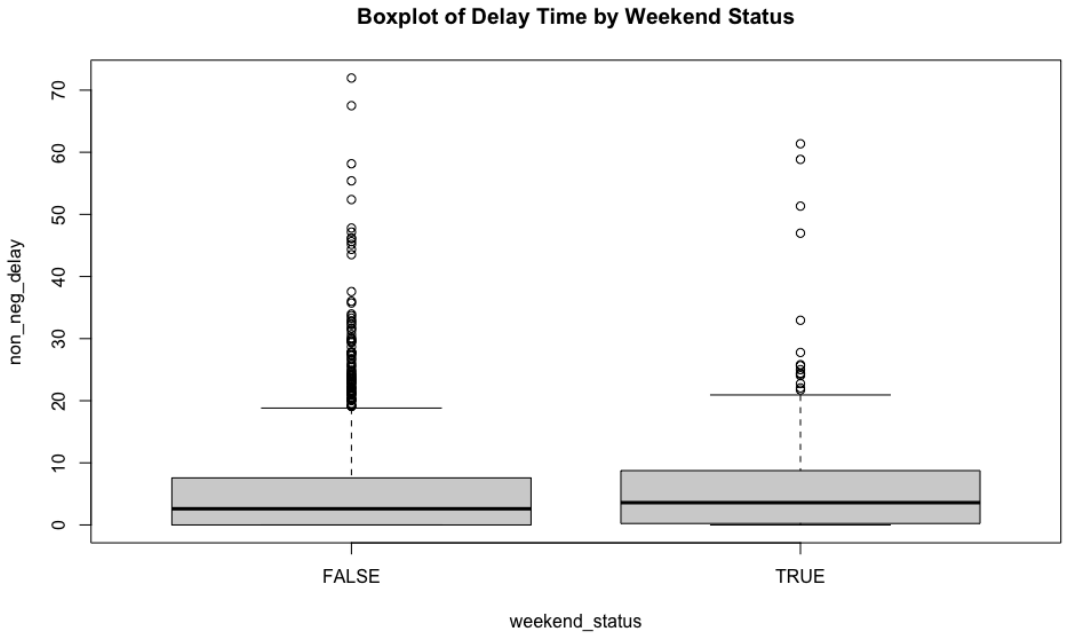


Fig-26: Distribution delay based on weekend status

Key Insights: The boxplot illustrates the distribution of non-negative delay times across two categories of 'weekend_status': FALSE (weekdays) and TRUE (weekends).

- Both categories exhibit a range of delays, with several outliers indicating occasional substantial delays.
- The median delay for weekends appears slightly higher than for weekdays.
- The spread of delays is similar between weekdays and weekends, with weekends showing more variability.

4. Day of the Year vs Delay Time:

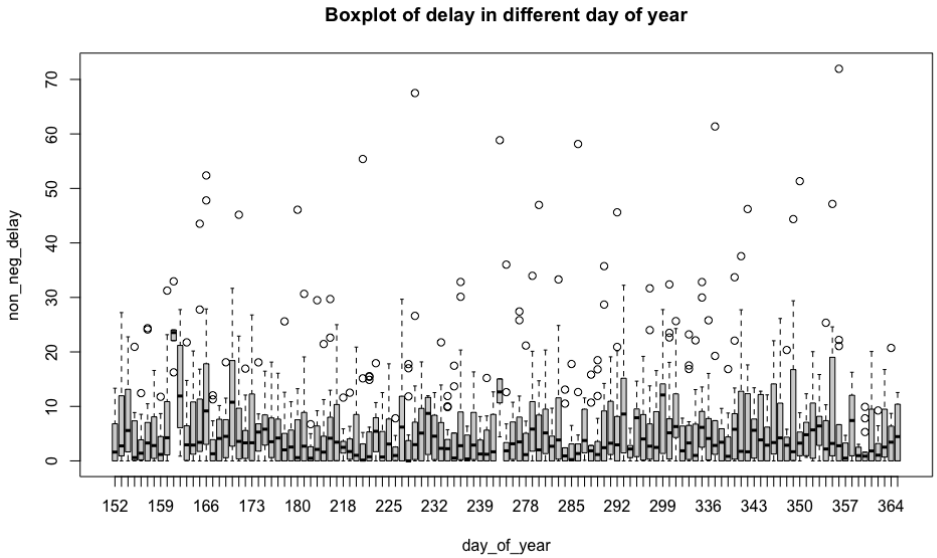


Fig-27: Distribution delay based on different day of the year

Key Insights: The boxplot presents the distribution of non-negative delay times across different days of the year.

- The plot reveals a consistent pattern of delays throughout the year, with the median values and interquartile ranges showing no dramatic fluctuations.
- Outliers are scattered across the entire year, suggesting sporadic occurrences of unusually high delays regardless of the season or specific day.

5. Time of the Day Column vs Delay Time:

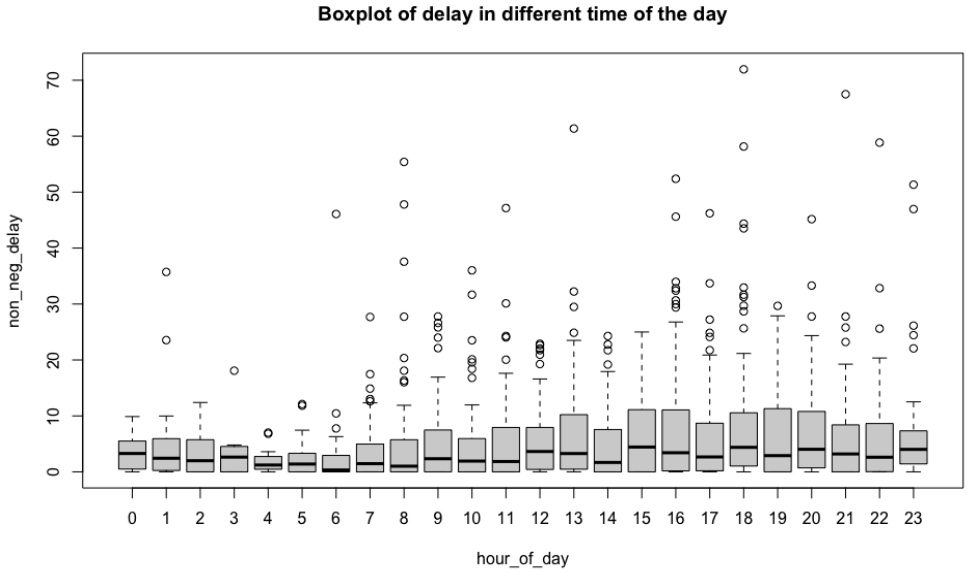


Fig-28: Distribution delay based on different day of the year

Key Insights: The boxplot depicts the variation in non-negative delay times across different hours of the day.

- There's a consistent presence of outliers throughout, indicating that high delays can occur at any hour.
- The medians and interquartile ranges appear relatively uniform across the hours, suggesting that the central tendency of delays does not vary significantly throughout the day.
- It's noticeable that the late-night and early-morning hours (**0-5**) show slightly less variability in delay times compared to the daytime hours.

Feature Selection

In this portion we decide which feature we need to use for our machine learning algorithm. All the current features are given below:

```
 [1] "direction"      "line_name"      "org_name"       "org_lat"        "org_long"       "dest_name"
 [7] "dest_lat"       "dest_long"      "vech_name"      "vech_lat"       "vech_long"      "next_point_name"
[13] "arrivial_app"   "dist_from_stop" "weekend_status" "day_of_year"    "time_of_day"    "non_neg_delay"
```

Fig-29: All the columns of the dataset

For our model training we don't need the latitude and longitude data, we also eliminate the "next_point_name" as we have source and destination station names. "Arrival_app" and "dist_from_stop" are also removed because these columns have no effects on the delay prediction. So, after that the dataset has the following columns.

```
[1] "direction"     "line_name"     "org_name"      "dest_name"     "vech_name"     "weekend_status"
[7] "day_of_year"   "time_of_day"   "non_neg_delay"
```

Fig-30: Final dataset columns

Test-Train Split

In this portion we randomly split our dataset into two subsets: train-set, and test-set. The train-set contains **80%** of total data and the test-set contains **20%** of the data.

Encoding

The summary statistics of our dataset is given below:

```
'data.frame':   1334 obs. of  9 variables:
 $ direction     : int  0 1 0 1 0 0 1 0 0 0 ...
 $ line_name     : chr  "Q27" "Bx38" "Q44-SBS" "M4" ...
 $ org_name      : chr  "SPRINGFIELD BL/JAMAICA AV" "BAY PZ/BARTOW AV" "MERRICK BL/ARCHER AV" "FT WASHINGTON AV/CABRINI BL"
...
 $ dest_name     : chr  "FLUSHING  MAIN ST STA" "NORWOOD 205 ST STATION via GUNHILL" "SELECT BUS BRONX ZOO via MAIN ST" "BWAY
- 135 ST" ...
 $ vech_name     : chr  "NYCT_6845" "NYCT_4411" "NYCT_5909" "NYCT_6652" ...
 $ weekend_status: logi  TRUE FALSE FALSE FALSE FALSE FALSE ...
 $ day_of_year   : int  224 347 181 221 283 233 181 170 240 303 ...
 $ time_of_day   : int  1012 631 995 1342 902 1206 962 1199 1063 586 ...
 $ non_neg_delay : num  0.717 0 18.633 0 12.767 ...
```

Fig-31: Summary of the dataset

In the above figure we can see that, among 8 features, 6 of them are categorical variables. As we want to implement a regression model out of the data, we need to convert them into numeric data.

In this section we convert our categorical features into numeric data. There are a couple of ways to convert them **like one-hot encoding**, **label encoding**, and **target encoding**. But we consider target encoding.

Why not one-hot encoding

- **Basic Concept:** One-hot encoding transforms each categorical value into a binary vector representing the presence (1) or absence (0) of the category. Each category becomes a new binary feature in the dataset.

- **Example:** In our dataset consider the feature direction: 0, and 1. One-hot encoding of this feature will create two new features: "weekend_statusTRUE", and "weekend_status-FALSE". A "0" item is encoded as [0,1], and a "1" item as [1, 0].

- **Problem:** For the direction and weekeend_status column we can perform one-hot encoding but it's kind of impossible to convert other categorical variables (with large cardinality) using one-hot encoding. From the below snapshot we can see that there are a total **(212+332+415+1148+121+789+625) = 3642** different categories for all the categorical features. As we only have 1334 observations, the number of columns will become 3 times more than the number of rows. So, we avoid one-hot encoding.

	(unique_counts)
direction	2
line_name	212
org_name	332
dest_name	415
vech_name	1148
weekend_status	2
day_of_year	121
time_of_day	789
non_neg_delay	625

Fig-32: Unique categories in categorical features

Why not Label Encoding

Limitation of label encoding:

- **Ordinal Relationship Imposition:** Label encoding assigns each unique category in the column with a numerical value in a sequence (like 0, 1, 2, 3,...). This introduces an artificial ordinal relationship. For example, if categories 'Red', 'Blue', 'Green' are encoded as 0, 1, 2, respectively, the model might interpret Green as higher or more important than Blue and Red, which is often not the case.
- **Model Misinterpretation:** Linear models assume that higher numerical values carry more weight. This can lead to incorrect interpretations and poor performance if the numerical values do not have an ordinal relationship.
- **Limited to Nominal Variables:** Label encoding is generally not suitable for nominal categorical variables where no ordinal relationship exists. For instance, encoding countries or gender with numbers might mislead the model.

As our categorical variables are not nominal and we tried a linear model, we don't use label encoding.

Target Encoding

The main idea behind the target encoder is to encode the categories by replacing them for a measurement of the effect they might have on the target.

In our project we use categorical mean encoding. Where we take the mean delay of each category and assign it instead of that category.

	direction	line_name	org_name	dest_name	vech_name	weekend_status	day_of_year	time_of_day
1	0	5.96	0.3111111	0.3666667	4.683333	TRUE	224	1012
2	1	5.96	0.3111111	3.0600000	3.566667	FALSE	347	631
3	0	5.96	0.3111111	3.0600000	16.850000	FALSE	181	995
4	1	5.96	31.6666667	3.0600000	5.691667	FALSE	221	1342
5	0	5.96	3.2250000	3.0600000	5.691667	FALSE	283	902
6	0	5.96	3.2250000	3.0600000	8.816667	FALSE	233	1206

Fig-33: Snapshot of the training data after target mean encoding

Target encoding for Test Data

As we randomly choose the train-test set, in the test set some of the categories are present which are missing in the training dataset. In that case we picked the mean of overall delay of train data for that category.

	direction	line_name	org_name	dest_name	vech_name	weekend_status	day_of_year	time_of_day
1	0	7.3781250	4.0000000	4.000000	4.000000	FALSE	346	672
2	0	0.1055556	0.1583333	5.431052	5.431052	FALSE	335	1326
3	0	16.5666667	5.5853164	6.087500	6.087500	FALSE	291	478
4	1	5.9600000	8.2666667	8.266667	8.266667	FALSE	160	851
5	0	5.5504577	3.0533333	5.431052	5.431052	FALSE	178	488
6	1	10.1712121	14.4666667	14.466667	14.466667	FALSE	296	852

Fig-34: Snapshot of the test data after target mean encoding

Machine Learning Model

We choose Lasso regression, and Generalized Additive Model (GAM) as our two machine learning models.

Lasso regression

Lasso regression, short for Least Absolute Shrinkage and Selection Operator regression, is a type of linear regression that uses shrinkage. Shrinkage is where data values are shrunk towards a central point, like the mean.

Why we choose Lasso regression

1. **Regularization:** The lasso regression adds a regularization term to the cost function. This term is the absolute value of the magnitude of coefficients, multiplied by a lambda parameter. The larger the value of lambda, the more the coefficients are shrunk towards zero.

2. **Variable Selection:** This approach inherently performs variable selection by shrinking less important variables' coefficients to zero. Variables with a coefficient of zero are

effectively excluded from the model. This leads to models that are easier to interpret and reveal the most important features of your data.

Lasso Implementation

For the lasso model at first we train a very simple lasso model:

```
lasso_model <- glmnet(as.matrix(trainX), trainY$non_neg_delay, alpha=1)
```

By using this basic lasso model the train RMSE value was: **8.234974** and test RMSE was: **8.1422**. Here the test RMSE is a bit less than the train RMSE. So, we investigate the scenario. Our hypothesis for that:

1. Maybe the RMSE is affected by outliers. The mean of train and test delay is **5.735307** and **5.615669** respectively. Also the variance of both sets is: **69.76785** and **66.78679**.
2. The train set has more outliers than the test set. In train, **6.52%** of data points are outlier where in test data **5.69%** are outlier.

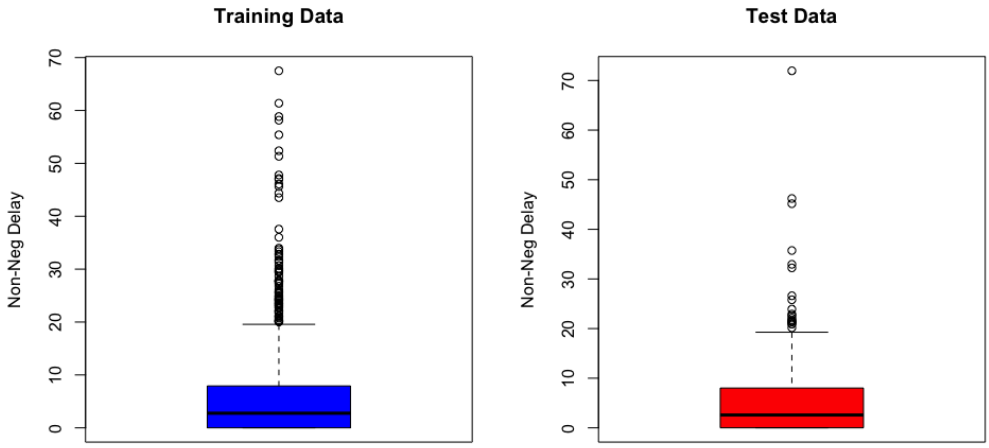


Fig-35: Delay time in train and test sets

Hyperparameter Tuning

In this portion, we use k-fold cross validation to choose the best Lasso model. We use a 10-fold cross validation. And then use the optimal value of Lambda to re-train the model.

The optimal value of Lambda is: **0.1406666** and the total non-zero coefficient is 7 (out of 9). After using the optimal Lasso model we achieved the train RMSE as **8.22499** and test RMSE as **8.131431**.

Generalized Additive Models (GAMs)

Generalized Additive Models (GAMs) are a type of statistical model that extend linear models by allowing non-linear relationships between the independent variables and the dependent variable through the use of smooth functions.

Model Structure: A typical GAM has the form:

```
g(E[Y]) = β0 + f1(X1) + f2(X2) + ... + fp(Xp)
```

Here, $g(\cdot)$ is the link function, E[Y] is the expected value of the dependent variable Y, β_0 is the intercept, and $f_i(X_i)$ are smooth functions of predictor variables X_i.

Smooth Functions: The functions $f_i(X_i)$ are typically constructed using spline bases, but other bases can be used as well. These functions are "smooth" in the sense that they vary smoothly with the predictor variables, allowing the model to capture non-linear trends.

Regularization: To prevent overfitting, GAMs include a penalty on the wiggliness of the smooth functions.

Implementation GAM

At first we implement a basic GAM model:

```
# fit a general GAM model with smoothing spine
gam_ss <- gam(non_neg_delay ~
            s(log(time_of_day), by = factor(weekend_status), k = 5) +
            s(org_name, by=factor(direction), k = 5) +
            s(dest_name, by=factor(direction), k = 5) +
            s(vech_name, k = 5) +
            s(line_name, k = 5) +
            s(day_of_year, k = 5),
            data = train_data,
            method = "REML")
```

In the model, we train a simple GAM with the features. The key-takeaway of the model is given below:

1. We use a smooth term of the logarithm of "time_of_day" based on "weekend_status".
2. We also apply the same technique for "org_name" and "dest_name" based on their "direction".
3. For the other three variables we used a simple smoothing spine.

4. In all the cases we keep, **k = 5** means that up to four degrees of freedom are allowed for the smooth terms (one degree of freedom is typically used for the intercept).
5. The model uses the REML (Restricted Maximum Likelihood) method for estimating smoothness. This is a way of making sure that the model is complex enough to fit the data well but not so complex that it just fits random noise.

```
Formula:
non_neg_delay ~ s(log(time_of_day), by = factor(weekend_status),
    k = 5) + s(org_name, by = factor(direction), k = 5) + s(dest_name,
    by = factor(direction), k = 5) + s(vech_name, k = 5) + s(line_name,
    k = 5) + s(day_of_year, k = 5)

Parametric coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   5.7117     0.2254   25.34   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Approximate significance of smooth terms:
                                                  edf Ref.df     F  p-value
s(log(time_of_day)):factor(weekend_status)FALSE 2.228  2.680 7.293 0.000242 ***
s(log(time_of_day)):factor(weekend_status)TRUE  2.567  3.086 5.340 0.001030 **
s(org_name):factor(direction)0                  1.001  1.002 2.249 0.133920
s(org_name):factor(direction)1                  1.001  1.002 1.420 0.233489
s(dest_name):factor(direction)0                 1.270  1.494 0.805 0.538782
s(dest_name):factor(direction)1                 1.003  1.005 0.037 0.850481
s(vech_name)                                    1.001  1.002 0.403 0.525748
s(line_name)                                    2.428  2.948 1.465 0.193991
s(day_of_year)                                  2.884  3.301 2.835 0.055022 .
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

R-sq.(adj) =  0.0342   Deviance explained = 4.54%
-REML = 4705.9  Scale est. = 67.379    n = 1334
```

Fig-36: Model summary of GAM

Model Summary: The GAM has identified significant non-linear relationships for time_of_day across different weekend_status categories. Other variables, particularly when interacting with direction, do not show strong non-linear relationships with the response. The overall model fit is modest (adjusted R-squared value is **0.0342**), capturing a small portion of the variability in the response variable (**3.42%**).

Training GAM with different types of function:

```
# fit a GAM model with s(), te(), ti()
gam_te <- gam(non_neg_delay ~
            ti(log(time_of_day), by = factor(weekend_status), k = 5) +
```

```
            te(org_name, dest_name, by = factor(direction), k = c(5, 5)) +
            te(vech_name, line_name, k = c(5, 5)) +
            s(day_of_year, k = 5),
            data = train_data,
            method = "REML")
```

In the model, we train a simple GAM with the features. The key-takeaway of the model is given below:

1. Here we modify the smoothing function for some of the variables. For "time_of_day" we use Tensor Interaction Smoothing Splines (ti) based on the categorical feature weekend_status with k = 5.
2. On the other hand, for "org_name", and "dest_name" we use Tensor Product Smoothing Splines based on the categorical feature direction with k = 5.
3. The model uses the REML (Restricted Maximum Likelihood) method for estimating smoothness. This is a way of making sure that the model is complex enough to fit the data well but not so complex that it just fits random noise.
4. The model summary is given below:

```
Formula:
non_neg_delay ~ ti(log(time_of_day), by = factor(weekend_status),
    k = 5) + te(org_name, dest_name, by = factor(direction),
    k = c(5, 5)) + te(vech_name, line_name, k = c(5, 5)) + s(day_of_year,
    k = 5)

Parametric coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   5.7132     0.2259   25.29   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Approximate significance of smooth terms:
                                                  edf Ref.df     F p-value
ti(log(time_of_day)):factor(weekend_status)FALSE 2.056  2.352 8.019 0.00015 ***
ti(log(time_of_day)):factor(weekend_status)TRUE  2.355  2.689 6.851 0.00110 **
te(org_name,dest_name):factor(direction)0        3.400  3.693 0.805 0.45141
te(org_name,dest_name):factor(direction)1        3.005  3.010 0.552 0.64785
te(vech_name,line_name)                          3.231  3.423 0.839 0.53762
s(day_of_year)                                   2.907  3.322 2.903 0.05163 .
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

R-sq.(adj) =  0.0309   Deviance explained = 4.33%
-REML = 4670.6  Scale est. = 67.61     n = 1334
```

Fig-37: Model summary of GAM

Model summary: The model captures some complex and non-linear relationships between the predictors and non_neg_delay, with significant effects observed particularly for the "time_of_day" interaction with "weekend_status". However, the overall explanatory power of the model is relatively low (adjusted R-squared = 0.0309 and deviance explained = 4.33%), suggesting that there may be other factors or complexities not captured by the current model structure.

Choosing better model: We apply ANOVA table for the comparison of the model. The summary of the anova table is given below:

```
Analysis of Deviance Table

Model 1: non_neg_delay ~ ti(log(time_of_day), by = factor(weekend_status),
    k = 5) + te(org_name, dest_name, by = factor(direction),
    k = c(5, 5)) + te(vech_name, line_name, k = c(5, 5)) + s(day_of_year,
    k = 5)
Model 2: non_neg_delay ~ s(log(time_of_day), by = factor(weekend_status),
    k = 5) + s(org_name, by = factor(direction), k = 5) + s(dest_name,
    by = factor(direction), k = 5) + s(vech_name, k = 5) + s(line_name,
    k = 5) + s(day_of_year, k = 5)
  Resid. Df Resid. Dev       Df Deviance
1    1313.0      88977
2    1313.3      88779 -0.36994   198.22
```

Fig-38: ANOVA table of two GAM model

Summary: The key insight of the anova table is given below:

1. **In the ANODEV table:**

   Model 1 (gam_te) has a residual deviance of 88,977.
   Model 2 (gam_ss) has a residual deviance of 88,779.

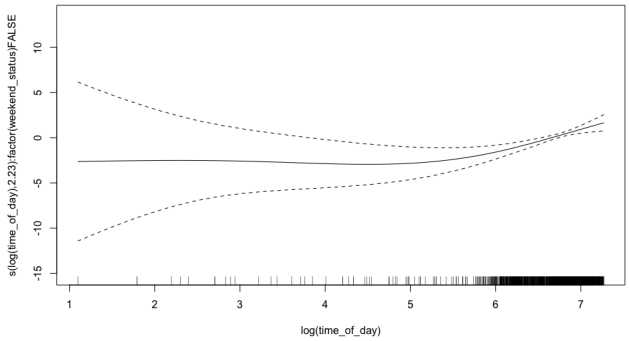
2. The model with the lower residual deviance is generally considered to have a better fit, as it explains more of the variability in the data. In this case, Model 2 (gam_ss) has a lower residual deviance compared to Model 1 (gam_te), suggesting it fits the data better.

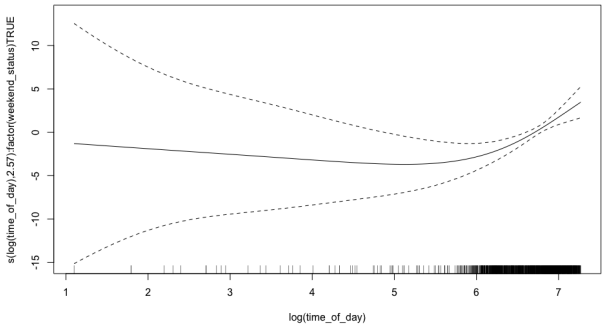
3. **Degrees of Freedom (Df):** The difference in degrees of freedom between the models can indicate their complexity. Here, the degrees of freedom are almost the same (**1313.0** vs. **1313.3**), indicating similar model complexities.

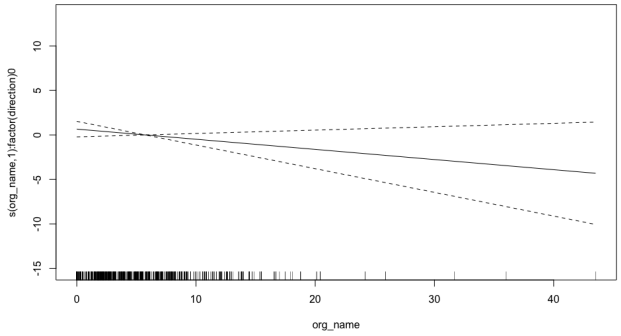
4. **Deviance Difference:** The change in deviance is 198.22 with a decrease in degrees of freedom of 0.36994 when moving from Model 1 (gam_te) to Model 2 (gam_ss).. This suggests that Model 2 (gam_ss) provides a slightly better fit without adding unnecessary complexity.

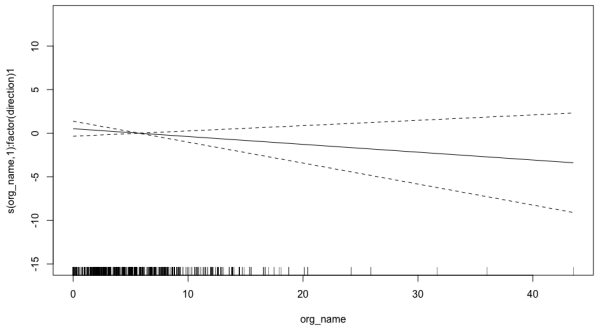
5. Finally we choose Model 2 (gam_ss) for the next level of our analysis and do hyper-parameter tuning with it.

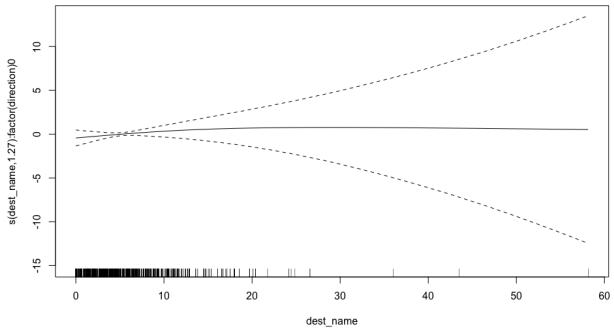
Visualization of the effect of different features in the GAM model (gam_ss):

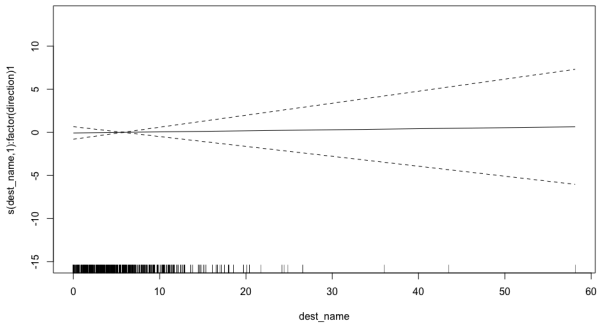


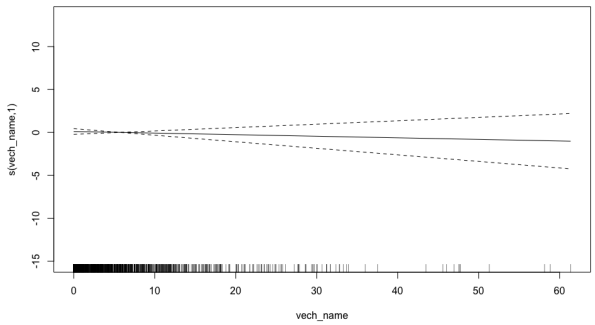


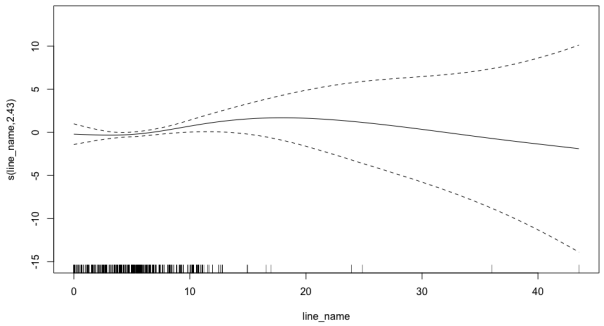


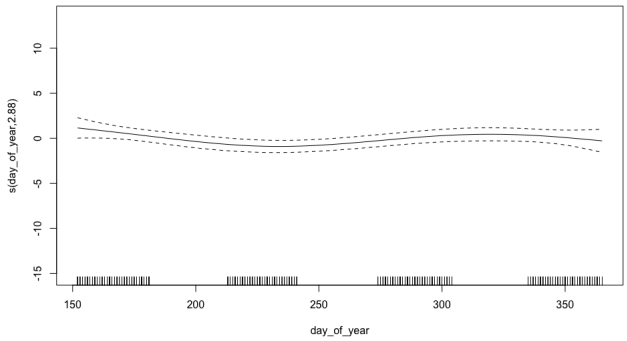












RMSE value of the GAM model spine smoothing:

For training dataset: **8.157889**

For test dataset: **8.106507**

Fig-39: Visualization of the effect of different features in the GAM model

Hyperparameter Tuning

We again apply k-fold cross validation to get the optimal value of K. We choose 10-fold cross validation and for each iteration we get the RMSE score, and calculate the average RMSE score each of the iteration. After that we compare the avg. RMSE score of each value of K and choose the K which has least avg. RMSE score. Initially, we choose the value of K = (**5, 6, 7, 8, 9, 10, 11, 12**).

From the below figure we can conclude that the RMSE score of 10-fold cross validation increases a bit when we choose K=5 to K=6, but suddenly drops with the increase of K until we reach at K=12. And we got the lowest RMSE in K=11.

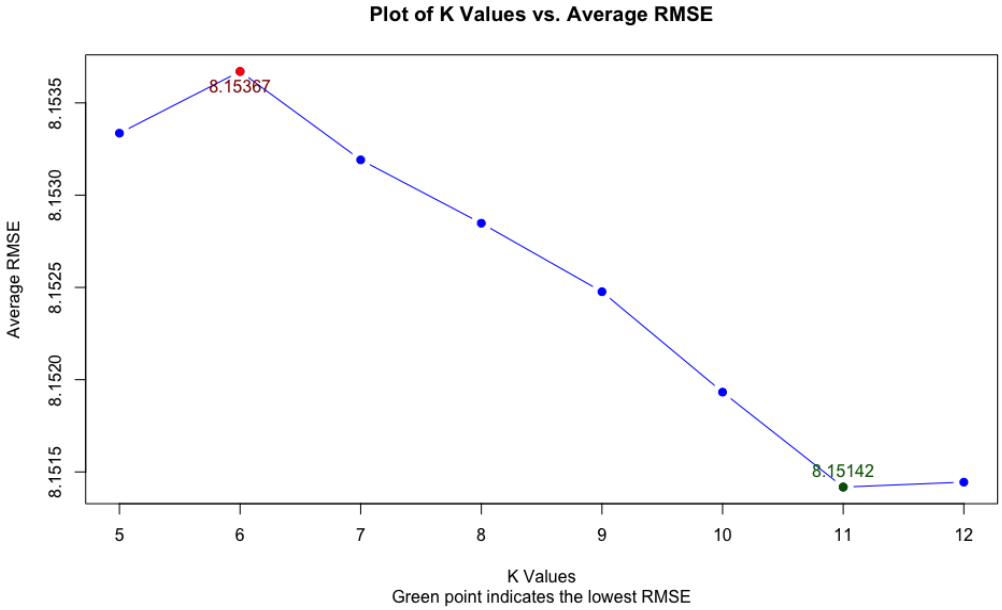


Fig-40: Choosing Optimal Value of K

Retrain the model with the optimal number of K: In this section, we retrain our GAM with K=11 and calculate the total train RMSE and test RMSE.

Final GAM:

```
final_model <- gam(non_neg_delay ~
      s(log(time_of_day), by = factor(weekend_status), k = best_k) +
      s(org_name, by=factor(direction), k = best_k) +
      s(dest_name, by=factor(direction), k = best_k) +
      s(vech_name, k = best_k) +
      s(line_name, k = best_k) +
      s(day_of_year, k = best_k) +
      weekend_status + direction,
      data = train_data, method = "REML")
#Here, best_k = 11
```

The summary statistics of the final model:

```
Formula:
non_neg_delay ~ s(log(time_of_day), by = factor(weekend_status),
    k = best_k) + s(org_name, by = factor(direction), k = best_k) +
    s(dest_name, by = factor(direction), k = best_k) + s(vech_name,
    k = best_k) + s(line_name, k = best_k) + s(day_of_year, k = best_k) +
    weekend_status + direction

Parametric coefficients:
                  Estimate Std. Error t value Pr(>|t|)
(Intercept)         5.9092     0.3440  17.176   <2e-16 ***
weekend_statusTRUE  0.2751     0.5653   0.487    0.627
direction          -0.4899     0.4528  -1.082    0.279
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Approximate significance of smooth terms:
                                                    edf Ref.df     F  p-value
s(log(time_of_day)):factor(weekend_status)FALSE 2.369  2.935 6.770 0.000217 ***
s(log(time_of_day)):factor(weekend_status)TRUE  2.726  3.395 4.745 0.001787 **
s(org_name):factor(direction)0                  1.002  1.004 2.096 0.147947
s(org_name):factor(direction)1                  1.002  1.003 1.404 0.235928
s(dest_name):factor(direction)0                 1.204  1.383 0.776 0.520306
s(dest_name):factor(direction)1                 1.006  1.012 0.046 0.838847
s(vech_name)                                    1.001  1.002 0.447 0.504001
s(line_name)                                    2.395  3.021 1.402 0.231146
s(day_of_year)                                  3.064  3.709 1.981 0.074987 .
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

R-sq.(adj) =  0.0337   Deviance explained = 4.66%
-REML = 4704.8  Scale est. = 67.418    n = 1334
```

Fig-41: Summary of the GAm with optimal K

Model Summary: The model performance is not significantly higher than the baseline GAM model though the adj. R-square and deviance explained increase slightly.

Accuracy of the final model:

Accuracy of the final GAM on train data: **8.152894**

Accuracy of the final GAM on test data: **8.129277**

Result Analysis

Model Name	Train RMSE	Test RMSE
Basic Lasso Model	8.234974	8.1422
Tuned Lasso Model (Lamda = 0.1406666)	8.22499	8.131431
Default GAM using spline smoothing K = 5	8.157889	8.106507
Tuned GAM (K = 11)	8.152894	8.129277

Among all the methods, GAM using spline smoothing performed best accuracy on the test set although the lowest train RMSE was found using tuned GAM of K = 11.

Conclusion

In conclusion, our NYC Bus Delay Time Prediction project applied machine learning techniques to develop models capable of forecasting bus delays. We implement two predictive models: Lasso regression and Generalized Additive Models (GAM). Through hyperparameter tuning, we refined our Lasso and Generalized Additive Models (GAM). The GAM, particularly with K set to 5 for spline smoothing, yielded the best performance on the test data. Notably, while a more complex GAM with K = 11 achieved the lowest training RMSE, it did not translate to the lowest test RMSE, highlighting the importance of model generalization overfitting to the training set.

Resources

1. **Code Repository:** *https://github.com/AhmedDiderRahat/nyc_bus_delay_prediction*
2. *https://www.kaggle.com/datasets/stoney71/new-york-city-transport-statistics*
3. *https://www.epa.gov/caddis-vol4/exploratory-data-analysis#:~:text=Exploratory%20Data%20Analysis%20(EDA)%20is,step%20in%20any%20data%20analysis*.

4. *https://www.analytixlabs.co.in/blog/univariate-analysis/*
5. *https://u-next.com/blogs/business-analytics/bivariate-analysis/*
6. *https://towardsdatascience.com/dealing-with-categorical-variables-by-using-target-encoder-a0f1733a4c69*
7. *https://www.kaggle.com/code/vprokopev/mean-likelihood-encodings-a-comprehensive-study/notebook*
8. *https://medium.com/@aryamohapatra/target-encoding-create-some-relation-between-target-variable-and-the-encoded-labels-2ed0d172fceb*
9. *https://m-clark.github.io/generalized-additive-models/application.html#multiple-features*
10. *https://www.analyticsvidhya.com/blog/2023/09/understanding-generalized-additive-models-gams-a-comprehensive-guide/*