The University of Manchester

MSc Data Science; Business and Management

Data 70121: Statistics and Machine Learning 1

Report Title: Maven Rail EDA

Student ID:

14161710

Date: 15/11/2025

Introduction

The data in the MavenRail_cleaned2.csv file contains rail travel data consisting of 13 columns detailing travel routes, times and service performance information. In total there are 30,686 rows, each being the equivalent of one rail journey/ticket purchase. Professor David Spiegelhalter (Cambridge Professor, OBE FRS) suggests giving data a 'star rating': This data is fictitious and was made for a contest, it does contain missing values but is generally well structured but it may require some data wrangling on date fields to make them more user friendly for querying. However, based on it being a fabricated data set I would have to give it a 0-star rating as it is unlikely that the values here line up with real world train statistics but for the purpose of a test data set it is very robust.

Exploratory Data Analysis

The aim of the exploratory data analysis (EDA) was to gain further insight into the passengers' journeys, operational performance of the rail network and investigate ticketing and pricing patterns. Upon first inspection, the dataset was found to have a clean consistent structure with only a few unexplainable missing values in the "Departure" and "Scheduled Arrival" columns likely from human error in data entry while the missing values in the "Actual Arrival" and "Reason for Delay" columns are expected due to cancelled trains or trains not being delayed respectively. Table 1 below shows the total count of missing values for the table. These missing values were either imputed or handled through filtering depending on analytical needs.

Column	Number of missing Values
Departure	3
Scheduled Arrival	4
Actual Arrival	1835
Reason for Delay	26565

Table 1

Next investigations were done into the tickets and pricing. This analysis showed a substantial variation in ticket prices, driven by the ticket type and the class interestingly no strong correlation between how busy a route was and price could be found. Overall, the distribution of prices was skewed to the right, reflecting many low-cost fares and fewer high-value first-class or flexible tickets.

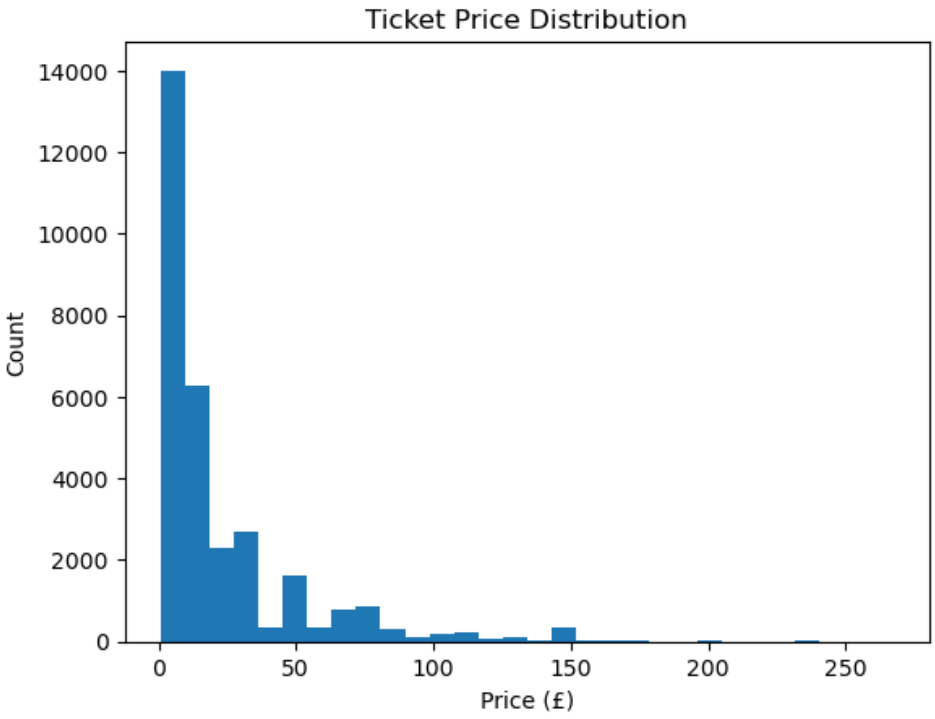


Fig. 1

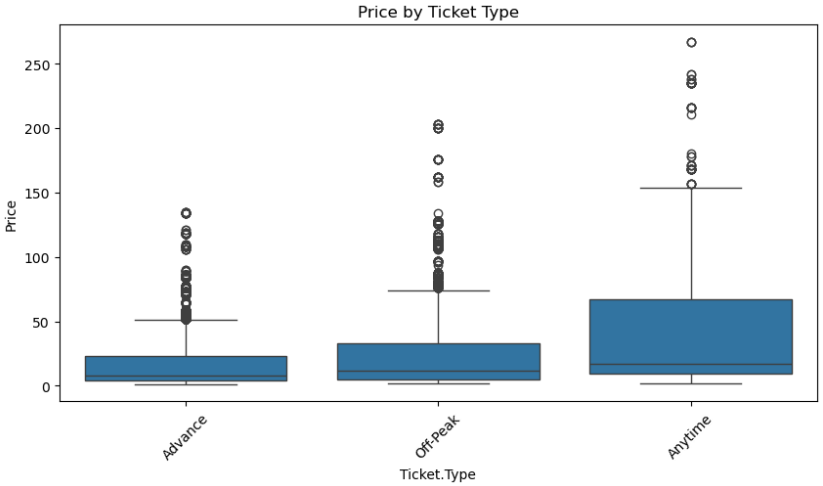


Fig. 2

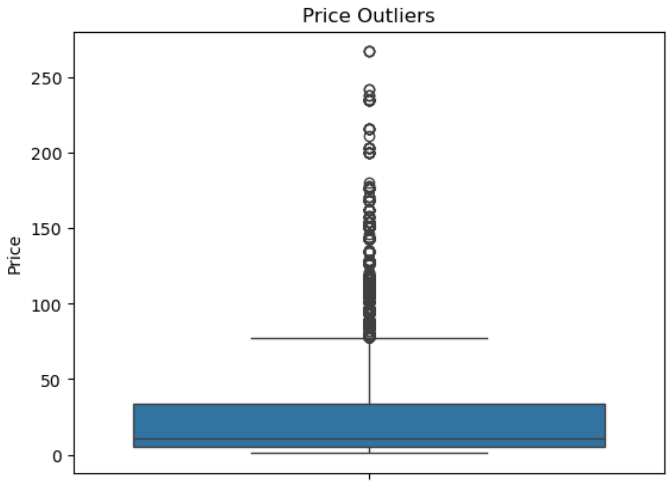


Fig. 3

Next the service performance was investigated. This analysis highlighted that typically services operated on time, while a smaller but still substantial proportion of journeys experienced cancellations or delays, a major driver behind refunds.

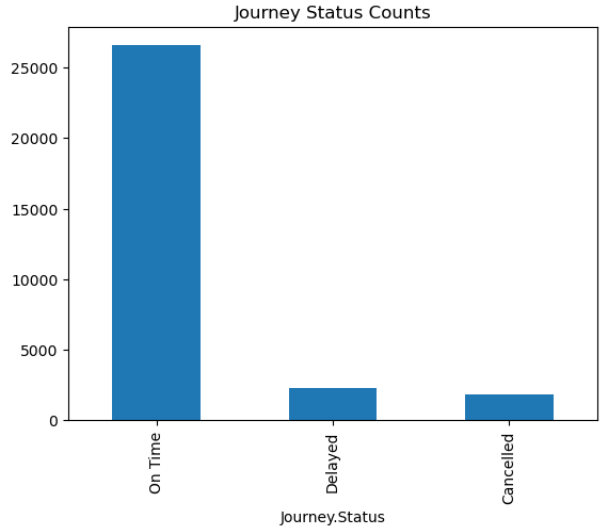


Fig. 4

To further analyse the data, route-level analysis was done grouping the records by the different station combinations. This showed that certain routes consistently carried higher volumes of passengers, intercity routes to and from London dominated the list of busiest journeys and non-coincidentally, all major London stations appeared in the top 10 busiest stations.

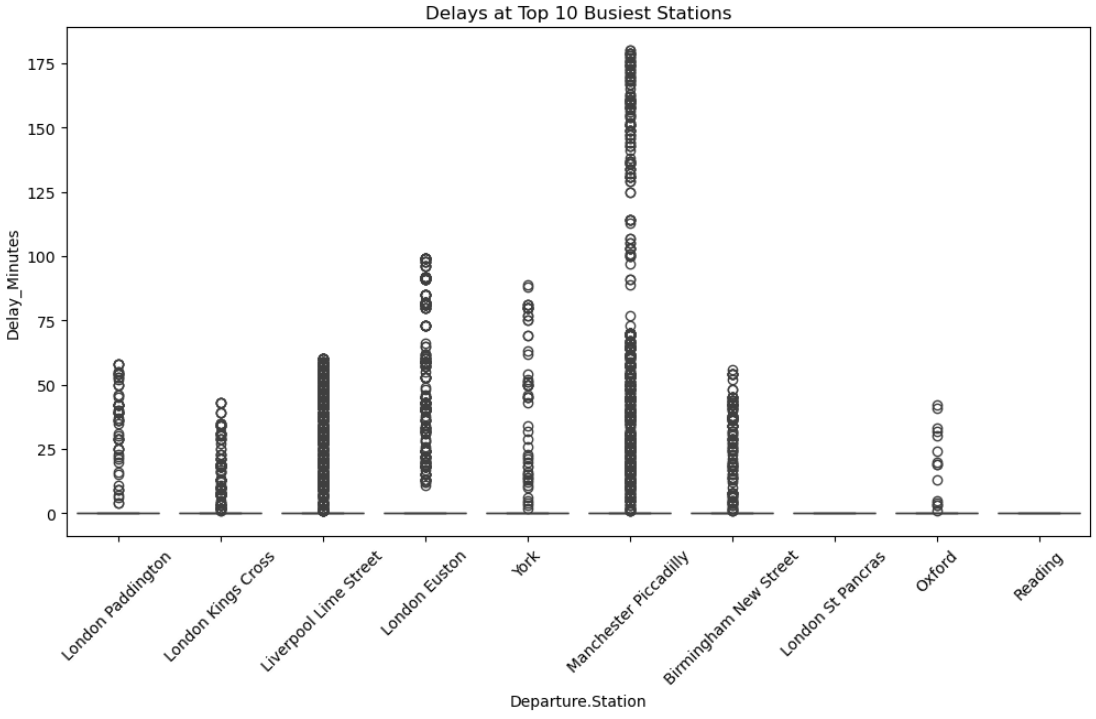


Fig. 5

Next the Delay patterns were examined. This revealed station-specific performance variations, a heatmap of average delay by departure and arrival station showed that stations in major cities had higher delays, suggesting underlying operational constraints. While London stations were in the top busiest stations only Euston was top 5 for delays with the other top 5 stations belonging to major hubs and could be identified as the “main” station in their respective cities it could be suggested that as London has multiple intercity stations the operational pressure is alleviated and shared across the various stations where as other major cities do not have the same level of infrastructure thus any delays at their stations results in greater knock on effects.

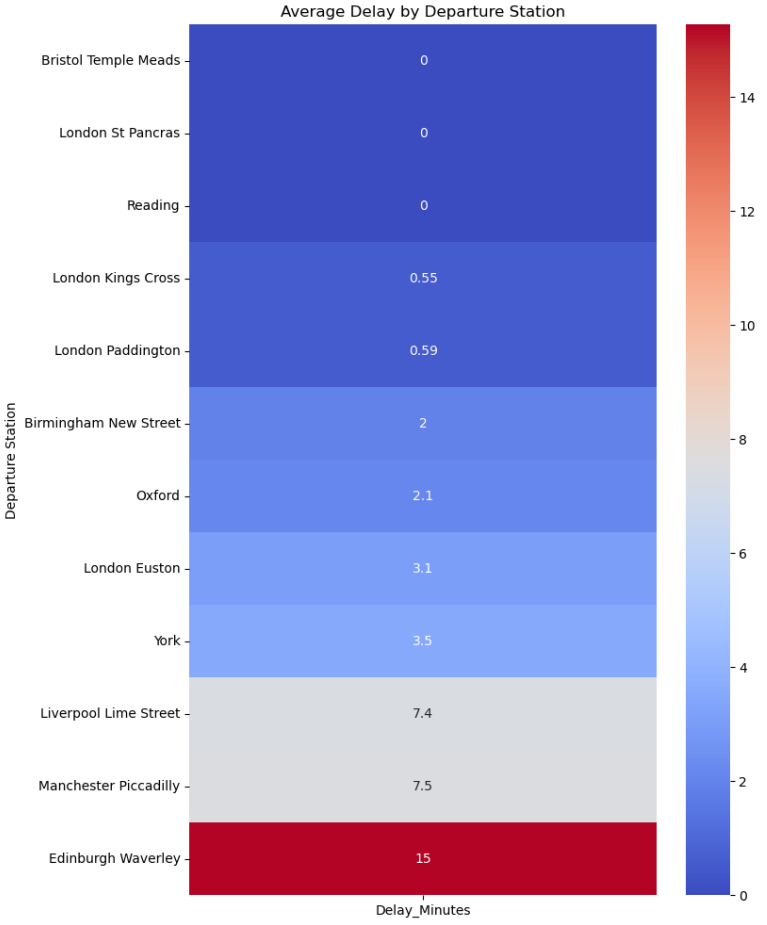


Fig. 6

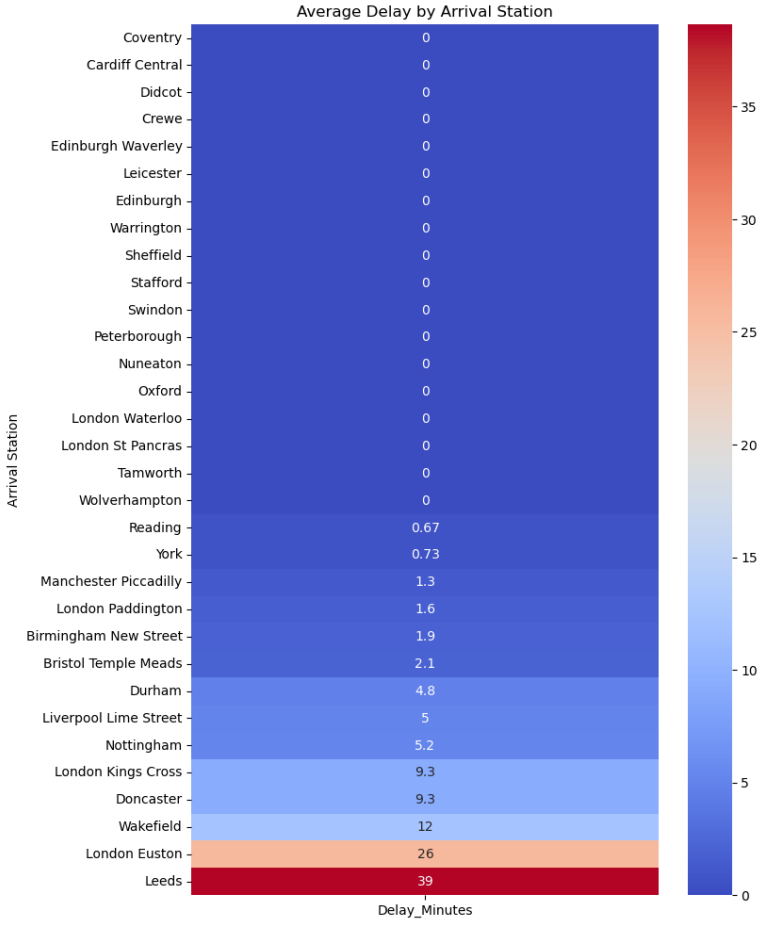


Fig. 7

Finally, binary variables such as Medium Price and the derived Delay Minutes attribute provided useful inputs for further analysis and Logistic regression modelling detailed later in this report. Through this EDA, the dataset’s structure, patterns, and key drivers were identified, informing subsequent modelling and decision-making.

Refund Forecast Analysis

To predict the likelihood of a passenger requesting a refund, A Binary logistic regression model was utilised. This model was selected based on the variables available and constraints detailed in the brief.
Firstly, the outcome variable "**Refund.Request"** is categorical with two possibilities: "Yes" or "No" making it a binary outcome and not continuous ruling out a linear regression model. The brief also specifies to use only a single predictor Medium Price which we can also take as a binary variable i.e. true when price is between £10 and £30 and false when it is not. As only a single predictor is being used we can rule out multi-variable regression models Based on these two variables and the task being to model probabilities a logistic regression was the most prudent option.

```
#set Refund.Request: Yes = 1, No = 0
df_late['Refund_Flag'] = df_late['Refund.Request'].map({'Yes': 1, 'No': 0})


# 2. Fit logistic regression model


# Predictor (MediumPrice) needs to be numeric (0/1)
X = df_late['MediumPrice'].astype(int)
X = sm.add_constant(X)  # add intercept

# Target variable
y = df_late['Refund_Flag']

# Fit model
model = sm.Logit(y, X).fit()

print(model.summary())
```

Fig. 8

The code above fits a **binary logistic regression model** to predict whether a passenger requests a refund using a single predictor, **Medium Price**. The model learns how being in this price range affects the likelihood of a refund.

```python
# 3. Use fitted model to predict probabilities


# Extract coefficients
b0 = model.params['const']
b1 = model.params['MediumPrice']

def logistic(z):
    return 1 / (1 + np.exp(-z))

# CASE 1: Ticket price = £5
# MediumPrice = False → 0
z_5 = b0 + b1 * 0
p_5 = logistic(z_5)

# CASE 2: Ticket price = £25
# MediumPrice = True → 1
z_25 = b0 + b1 * 1
p_25 = logistic(z_25)

print("\nPredicted probability of refund request:")
print(f"• For £5 ticket (MediumPrice = 0): {p_5:.4f}")
print(f"• For £25 ticket (MediumPrice = 1): {p_25:.4f}")
```

Fig. 9

After fitting the model, the script extracts the coefficients and uses them to calculate the refund probability for the two ticket prices then using the logistic formula, the model outputs two probabilities:

- **For £5 ticket (Medium Price = 0): 0.2527 ~25%**
- **For £25 ticket (Medium Price = 1): 0.3252 ~ 33%**

This indicates that passengers with mid-priced tickets (£10–£30) are more **l**ikely to request a refund than those with very cheap tickets.

With the first model returning viable results, another more expansive model was built to answer question 5 of the brief. While the target variable remains the same, the major difference here was the removal of the single predictor constraint which opened more viable model options such as a decision tree or random forest classifier models. However, I settled on a logistic regression as:

1. Coefficients are interpretable through log odds and odds ratios
2. It's a strong model for binary problems

Logistic Regression Formula

- The model estimates $\log(p/(1-p)) = \beta 0 + \beta 1X1 + ... + \beta kXk$.
- Probability is computed as $p = 1 / (1 + \exp(-(\beta 0 + \beta 1X1 + ...)))$.

Modelling this way allows the effect of each feature on the refund likelihood to be easily interpretable.

Feature Selection

After reviewing the ToPredict_with_refund.csv file, 6 meaningful and easily interpretable predictors were selected:

1. Ticket Price

- Continuous
- High correlation (higher prices ~ more likely to refund)

2. Ticket Type

- Categorical
- Some types have higher refund eligibility

3. Ticket Class

- Categorical
- passengers may behave differently based on class.

4. Railcard

- Categorical
- Discount holders may purchase differently

5. Journey Status

- Categorical
- Logical predictor: passengers likely to request refund for poor service

6. Delay Minutes

- Categorical encoded as: Actual or Scheduled

Building the model

- With the predictors selected the next step was to encode the target variable; Refund_Request **Yes" =** 1**, "No" =** 0
- Categorical variables were encoded.
- Numerical variables were used directly.
- Model fitted using maximum likelihood estimation (MLE).

Results

Record number	Refund Probability
1	0.000201094
2	0.000208822
3	0.352582095
4	0.784001039
5	0.393095834
6	0.317300491
7	0.665762921
8	0.297569892

Table 2

The model outputs Refund Probability for each passenger, with probabilities closer to 1 indicating a higher likelihood of requesting a refund.

References

Hosmer, D.W., Lemeshow, S. & Sturdivant, R.X., 2013. *Applied Logistic Regression*. 3rd ed. Hoboken: Wiley.

James, G., Witten, D., Hastie, T. & Tibshirani, R., 2021. *An Introduction to Statistical Learning with Applications in R*. 2nd ed. New York: Springer.

Field, A.P., Miles, J. & Field, Z., 2012. *Discovering Statistics Using R*. London: SAGE Publications.

Kutner, M.H., Nachtsheim, C.J., Neter, J. & Li, W., 2005. *Applied Linear Statistical Models*. 5th ed. New York: McGraw-Hill Irwin.

Gelman, A. & Hill, J., 2007. *Data Analysis Using Regression and Multilevel/Hierarchical Models*. Cambridge: Cambridge University Press.

Appendix

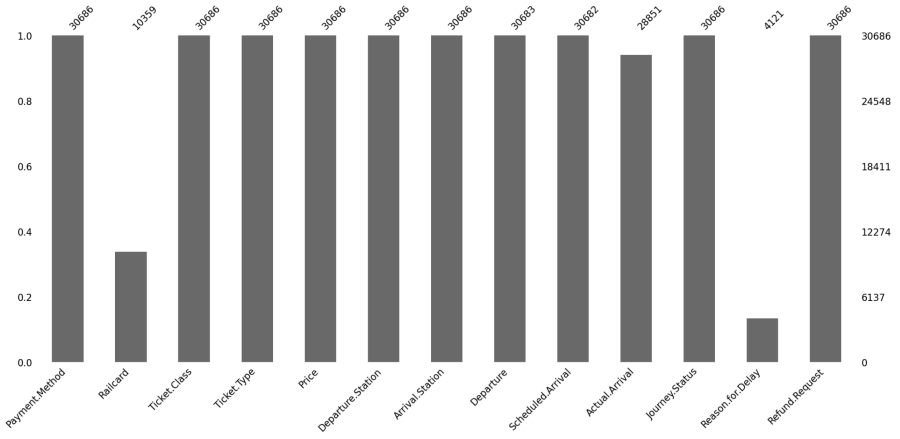
```
[25]: import pandas as pd
      import numpy as np
      import matplotlib.pyplot as plt
      import seaborn as sns
      import missingno as msno # Our hero ^_^
      import statsmodels.api as sm

      df = pd.read_csv('MavenRail_cleaned2.csv')

      #Data structure#
      df.head(10)
      df.info()
      df.shape
      df.columns
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30686 entries, 0 to 30685
Data columns (total 13 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   Payment.Method     30686 non-null  object
 1   Railcard           10359 non-null  object
 2   Ticket.Class       30686 non-null  object
 3   Ticket.Type        30686 non-null  object
 4   Price              30686 non-null  int64
 5   Departure.Station  30686 non-null  object
 6   Arrival.Station    30686 non-null  object
 7   Departure          30683 non-null  object
 8   Scheduled.Arrival  30682 non-null  object
 9   Actual.Arrival     28851 non-null  object
 10  Journey.Status     30686 non-null  object
 11  Reason.for.Delay   4121 non-null   object
 12  Refund.Request     30686 non-null  object
dtypes: int64(1), object(12)
memory usage: 3.0+ MB
[25]: Index(['Payment.Method', 'Railcard', 'Ticket.Class', 'Ticket.Type', 'Price',
       'Departure.Station', 'Arrival.Station', 'Departure',
       'Scheduled.Arrival', 'Actual.Arrival', 'Journey.Status',
       'Reason.for.Delay', 'Refund.Request'],
      dtype='object')
```

Appendix 1- library initialisation and data structure investigation

```
[26]: #Missing Values

      df.isna().sum()
      msno.bar(df)
```

[26]: <Axes: >



Appendix 2- missing valiues visualisation1

```
[27]: #Data Summary Stats
      df.describe()
      df.describe(include='object')
```

[27]:

	Payment.Method	Railcard	Ticket.Class	Ticket.Type	Departure.Station	Arrival.Station	Departure	Scheduled.Arrival	Actual.Arrival	Journey.Status	Reas
count	30686	10359	30686	30686	30686	30686	30683	30682	28851	30686	
unique	3	3	2	3	12	32	8422	13403	13544	3	
top	Credit Card	Adult	Standard	Advance	Manchester Piccadilly	Birmingham New Street	2024-02-13 18:45	2024-02-21 19:05	2024-04-27 19:05	On Time	
freq	18583	4653	27724	17039	5599	7607	34	21	18	26565	

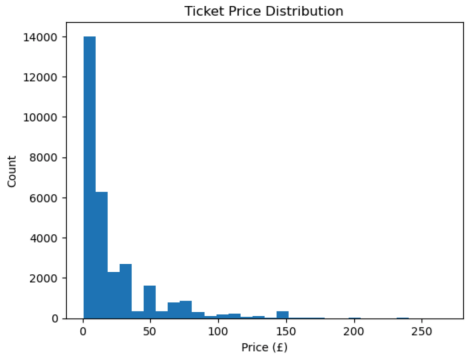
Appendix 3- Summary stats

```
[28]: #convert dates
df["Departure"] = pd.to_datetime( df["Departure"], format='%Y-%m-%d %H:%M')
df["Scheduled.Arrival"] = pd.to_datetime( df["Scheduled.Arrival"], format='%Y-%m-%d %H:%M')
df["Actual.Arrival"] = pd.to_datetime( df["Actual.Arrival"], format='%Y-%m-%d %H:%M')
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30686 entries, 0 to 30685
Data columns (total 13 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   Payment.Method     30686 non-null  object
 1   Railcard           10359 non-null  object
 2   Ticket.Class       30686 non-null  object
 3   Ticket.Type        30686 non-null  object
 4   Price              30686 non-null  int64
 5   Departure.Station  30686 non-null  object
 6   Arrival.Station    30686 non-null  object
 7   Departure          30683 non-null  datetime64[ns]
 8   Scheduled.Arrival  30682 non-null  datetime64[ns]
 9   Actual.Arrival     28851 non-null  datetime64[ns]
 10  Journey.Status     30686 non-null  object
 11  Reason.for.Delay   4121 non-null   object
 12  Refund.Request     30686 non-null  object
dtypes: datetime64[ns](3), int64(1), object(9)
memory usage: 3.0+ MB
```

Appendix 4- Date conversions

```
[29]: ### Univariate Analysis

#Price Distribution
plt.hist(df['Price'], bins=30)
plt.title("Ticket Price Distribution")
plt.xlabel("Price (£)")
plt.ylabel("Count")
plt.show()
```

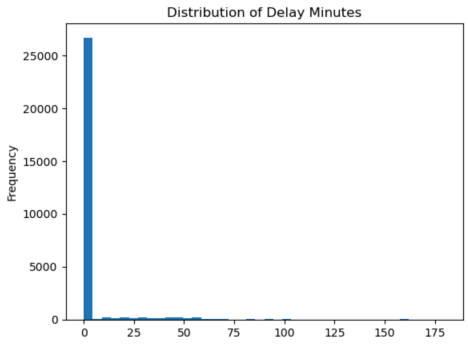



```
[ ]: #Add delay in minutes column
df['Delay_Minutes'] = (df['Actual.Arrival'] - df['Scheduled.Arrival']).dt.total_seconds() / 60

df.loc[df['Journey.Status'] == "On Time", 'DelayInMinutes'] = np.nan
```

Appendix 5- Price Distribution

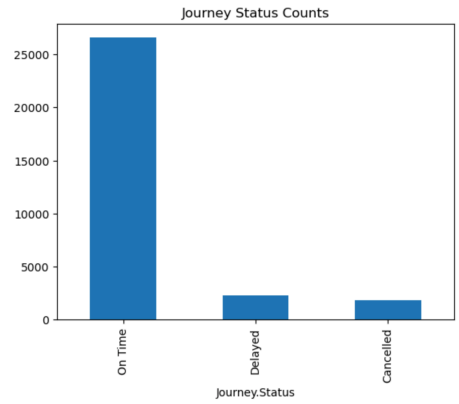
```
[ ]: #Add delay in minutes column
df['Delay_Minutes'] = (df['Actual.Arrival'] - df['Scheduled.Arrival']).dt.total_seconds() / 60

df.loc[df['Journey.Status'] == "On Time", 'DelayInMinutes'] = np.nan
```

```
[31]: #Delay Distribution
df['Delay_Minutes'].dropna().plot(kind='hist', bins=40)
plt.title("Distribution of Delay Minutes")
plt.xlabel("Minutes")
plt.show()
```



Appendix 6 – Addition of Delay in minutes column and Delay distribution

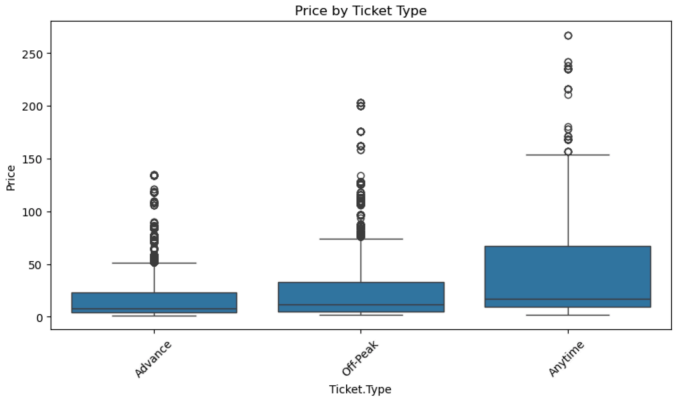
```
[32]: #Journey Status Counts
df['Journey.Status'].value_counts().plot(kind='bar')
plt.title("Journey Status Counts")
plt.show()
```




```
[33]: #Price by Ticket Type
```

Appendix 7 – Journey Status grouped counts

```
[33]: #Price by Ticket Type
plt.figure(figsize=(10,5))
sns.boxplot(data=df, x='Ticket.Type', y='Price')
plt.xticks(rotation=45)
plt.title("Price by Ticket Type")
plt.show()
```



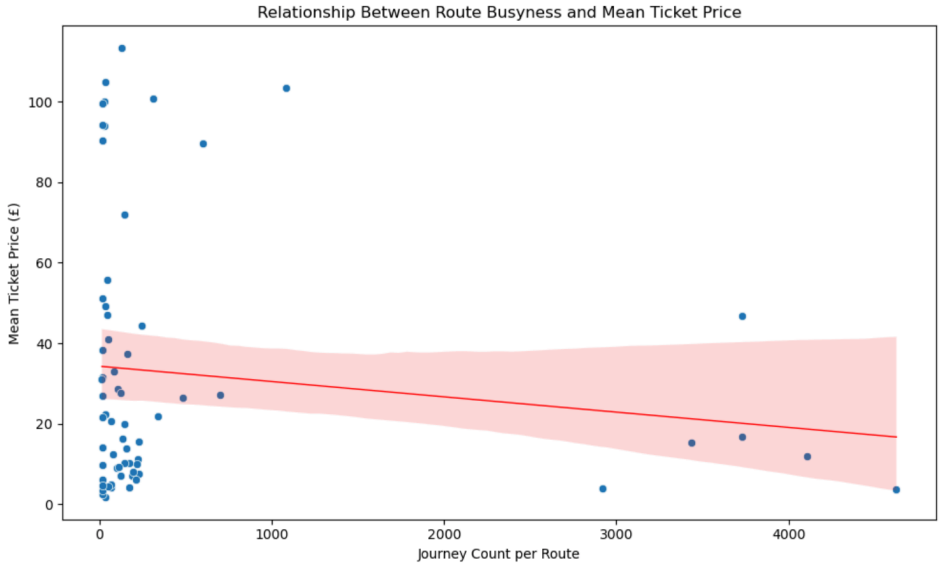
Appendix 8 – Price By Ticket Type

```
[52]: df["Route"] = df["Departure.Station"] + " → " + df["Arrival.Station"]
route_price = df.groupby("Route")["Price"].mean().sort_values(ascending=False)

print("Mean ticket price by route:")
print(route_price.head(10))


plt.figure(figsize=(12,6))
route_price.head(15).plot(kind="bar")
plt.title("Average Ticket Price by Route (Top 15)")
plt.xlabel("Route")
plt.ylabel("Average Ticket Price (£)")
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()
```

```
Mean ticket price by route:
Route
Manchester Piccadilly → London Paddington     113.220472
Liverpool Lime Street → London St Pancras     104.774194
Liverpool Lime Street → London Euston         103.338889
Manchester Piccadilly → London Euston         100.715655
Liverpool Lime Street → London Paddington      99.962963
Manchester Piccadilly → London St Pancras      99.562500
Reading → Liverpool Lime Street                94.125000
London Paddington → Manchester Piccadilly      93.933333
Manchester Piccadilly → London Kings Cross     90.250000
London Euston → Manchester Piccadilly          89.600334
Name: Price, dtype: float64
```

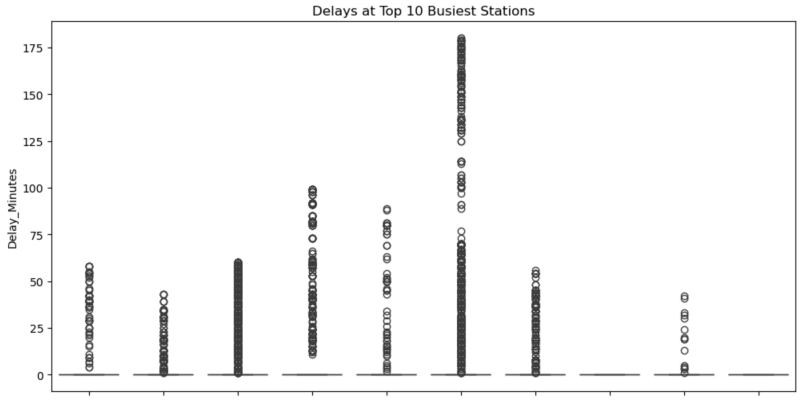
Appendix 9 – Ticket price by route

```
[53]: route_stats = df.groupby("Route").agg(
          MeanPrice=("Price", "mean"),
          JourneyCount=("Route", "count")
      ).reset_index()

      # ------------------------------------------------------------
      # 3. Compute correlation coefficient
      # ------------------------------------------------------------
      corr = route_stats["MeanPrice"].corr(route_stats["JourneyCount"])
      print(f"Correlation between route busyness and ticket price: {corr:.4f}")

      # ------------------------------------------------------------
      # 4. Scatter plot
      # ------------------------------------------------------------
      plt.figure(figsize=(10,6))
      sns.scatterplot(
          data=route_stats,
          x="JourneyCount",
          y="MeanPrice"
      )

      # Add trendline for visual correlation
      sns.regplot(
          data=route_stats,
          x="JourneyCount",
          y="MeanPrice",
          scatter=False,
          color="red",
          line_kws={"linewidth": 1}
      )

      plt.title("Relationship Between Route Busyness and Mean Ticket Price")
      plt.xlabel("Journey Count per Route")
      plt.ylabel("Mean Ticket Price (£)")
      plt.tight_layout()
      plt.show()

      Correlation between route busyness and ticket price: -0.1268
```



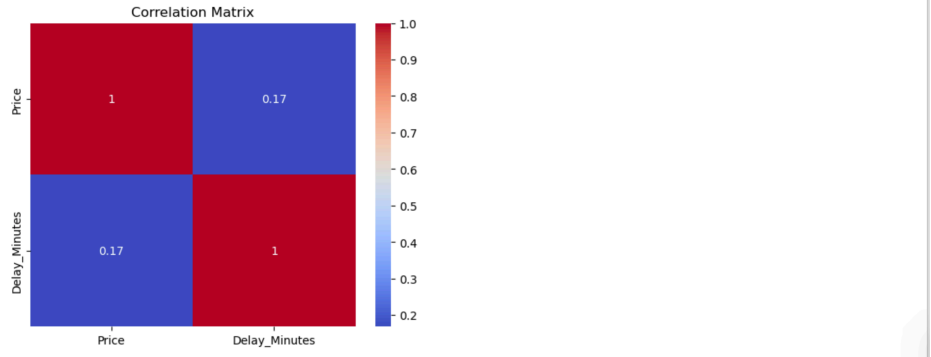
Appendix 10 – Correlation between route traffic and price

```
[34]: plt.figure(figsize=(12,6))
top_departures = df['Departure.Station'].value_counts().head(10).index

sns.boxplot(
    data=df[df['Departure.Station'].isin(top_departures)],
    x='Departure.Station', y='Delay_Minutes'
)
plt.xticks(rotation=45)
plt.title("Delays at Top 10 Busiest Stations")
plt.show()
```



Appendix 11 – Delays at busiest stations

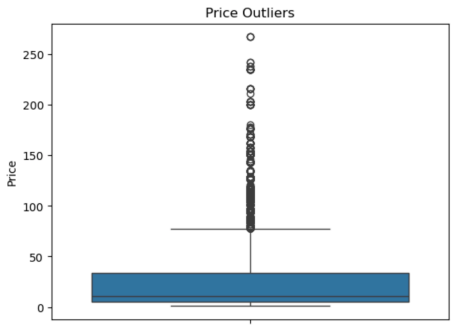
```
[35]: #Correlation Matrix price and delay
numeric_df = df.select_dtypes(include=['int64','float64'])
sns.heatmap(numeric_df.corr(), annot=True, cmap='coolwarm')
plt.title("Correlation Matrix")
plt.show()
```




```
[36]: #outliers
#price outliers
sns.boxplot(df['Price'])
plt.title("Price Outliers")
plt.show()
```

Appendix 12 – Correlation matrix of price and delay

[36]:
```
#outliers
#price outliers
sns.boxplot(df['Price'])
plt.title("Price Outliers")
plt.show()
```



[37]:
```
#delay outliers
sns.boxplot(df['Delay_Minutes'])
plt.title("Delay Outliers")
plt.show()
```

Appendix 13 – Price and Delay outlier's code

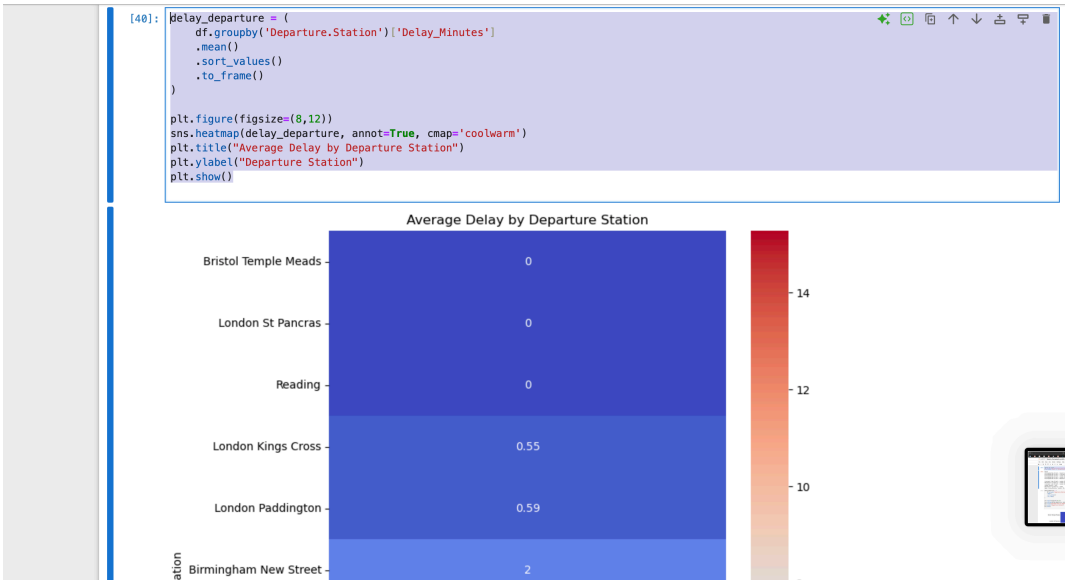
[38]:
```
#station level analysis
#most frequent routes
df['Route'] = df['Departure.Station'] + " → " + df['Arrival.Station']
df['Route'].value_counts().head(10)
```

[38]:
```
Route
Manchester Piccadilly → Liverpool Lime Street    4626
London Euston → Birmingham New Street            4109
London Kings Cross → York                        3731
London Paddington → Reading                      3728
London St Pancras → Birmingham New Street        3436
Liverpool Lime Street → Manchester Piccadilly    2918
Liverpool Lime Street → London Euston            1080
Birmingham New Street → London St Pancras         701
London Euston → Manchester Piccadilly             598
London Paddington → Oxford                        485
Name: count, dtype: int64
```

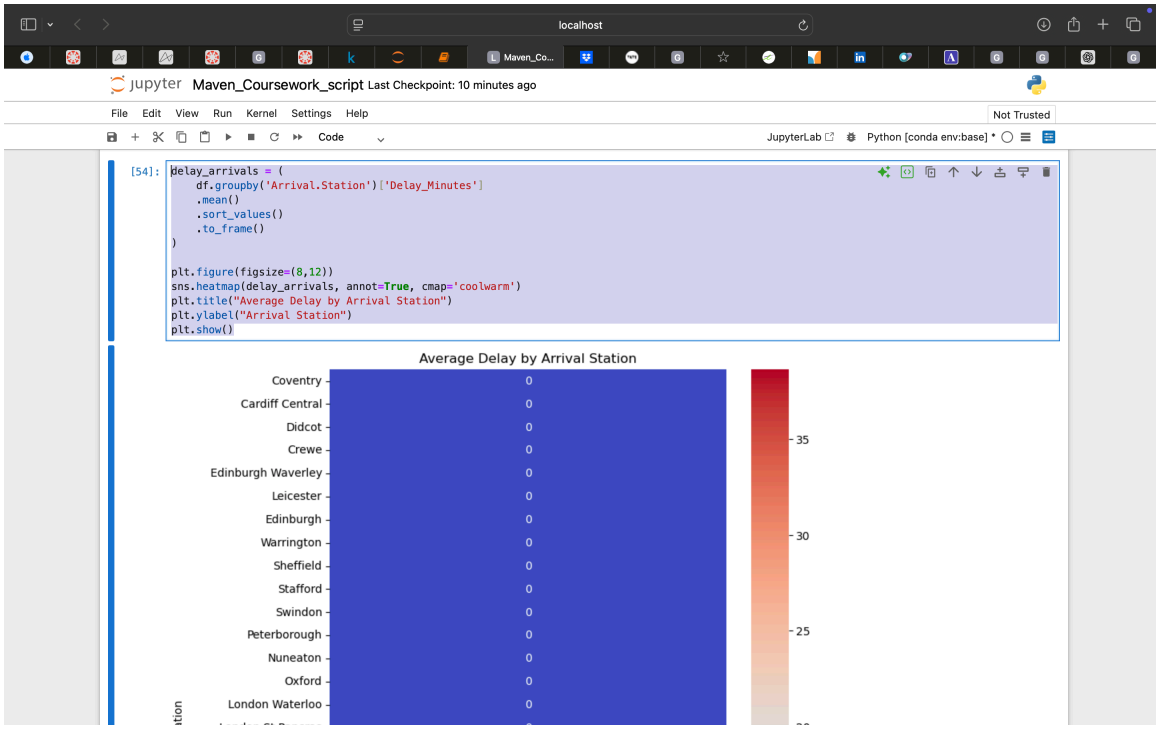
[39]:
```
#delay by route
df.groupby('Route')['Delay_Minutes'].mean().sort_values().head(100)
```

[39]:
```
Route
Birmingham New Street → Coventry                  0.000000
Birmingham New Street → Liverpool Lime Street     0.000000
Birmingham New Street → London Kings Cross        0.000000
Birmingham New Street → London Paddington         0.000000
Birmingham New Street → Nuneaton                  0.000000
                                                   ...
Liverpool Lime Street → London Paddington        18.481481
Manchester Piccadilly → London Euston            18.503205
Liverpool Lime Street → London Euston            29.061162
London Euston → York                             35.125000
Manchester Piccadilly → Leeds                    65.241135
Name: Delay_Minutes, Length: 64, dtype: float64
```

Appendix 14 – Route analysis



Appendix 15 – Delay by departure station

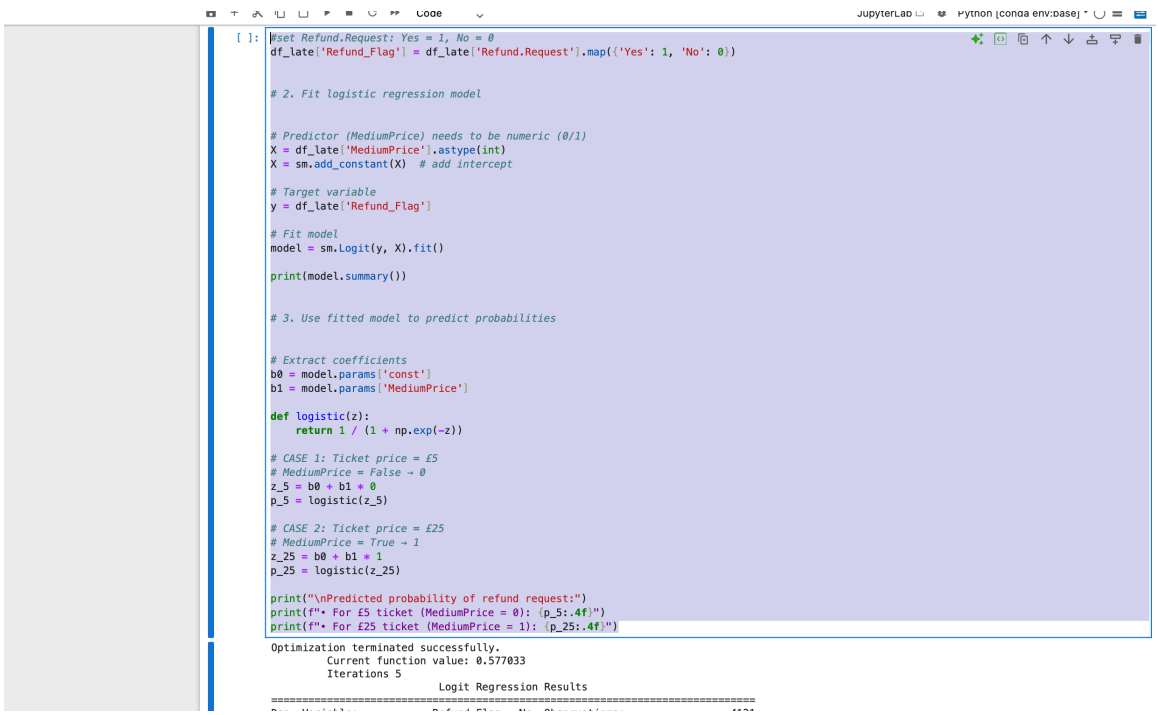


Appendix 16 – Delay by arrival station

```
[41]: #Q4

# Restrict to journeys that are NOT "On Time"
df_late = df[df['Journey.Status'] != "On Time"].copy()

# Add "MediumPrice"
#median_price = df['Price'].median()

df_late['MediumPrice'] = df_late['Price'].apply(lambda x: 10 < x <= 30)

# Save filtered dataset
df_late.to_csv('/Users/tomiwaodulaja/Desktop/Masters/DATA70121/Assessment1/MavenRail_filtered_with_MediumPrice.csv', index=False)

print("Filtered dataset created with MediumPrice column.")
print(df_late.head())
```

```
Filtered dataset created with MediumPrice column.
   Payment.Method Railcard Ticket.Class Ticket.Type  Price  \
1     Credit Card    Adult     Standard     Advance     23
8     Credit Card      NaN     Standard     Advance     37
20     Debit Card    Adult     Standard     Advance      7
26    Credit Card   Senior  First Class     Advance     34
38    Credit Card      NaN     Standard     Advance      7

        Departure.Station        Arrival.Station           Departure  \
1       London Kings Cross                 York 2024-01-01 09:45:00
8            London Euston                 York 2024-01-01 00:00:00
20  Birmingham New Street  Manchester Piccadilly 2024-01-01 11:15:00
26                 Oxford   Bristol Temple Meads 2024-01-01 14:15:00
38          London Euston  Birmingham New Street 2024-01-02 02:15:00

     Scheduled.Arrival      Actual.Arrival Journey.Status Reason.for.Delay  \
1  2024-01-01 11:35:00 2024-01-01 11:40:00        Delayed   Signal Failure
8  2024-01-01 01:50:00 2024-01-01 02:07:00        Delayed   Signal Failure
20 2024-01-01 12:35:00 2024-01-01 13:06:00        Delayed  Technical Issue
26 2024-01-01 15:30:00 2024-01-01 15:54:00        Delayed   Signal Failure
38 2024-01-02 03:35:00                NaT      Cancelled  Technical Issue

   Refund.Request  Delay_Minutes  \
```

Appendix 17 – Medium price dataset




```
[ ]: #set Refund.Request: Yes = 1, No = 0
df_late['Refund_Flag'] = df_late['Refund.Request'].map({'Yes': 1, 'No': 0})


# 2. Fit logistic regression model


# Predictor (MediumPrice) needs to be numeric (0/1)
X = df_late['MediumPrice'].astype(int)
X = sm.add_constant(X)  # add intercept

# Target variable
y = df_late['Refund_Flag']

# Fit model
model = sm.Logit(y, X).fit()

print(model.summary())


# 3. Use fitted model to predict probabilities


# Extract coefficients
b0 = model.params['const']
b1 = model.params['MediumPrice']

def logistic(z):
    return 1 / (1 + np.exp(-z))

# CASE 1: Ticket price = £5
# MediumPrice = False → 0
z_5 = b0 + b1 * 0
p_5 = logistic(z_5)

# CASE 2: Ticket price = £25
# MediumPrice = True → 1
z_25 = b0 + b1 * 1
p_25 = logistic(z_25)

print("\nPredicted probability of refund request:")
print(f"• For £5 ticket (MediumPrice = 0): {p_5:.4f}")
print(f"• For £25 ticket (MediumPrice = 1): {p_25:.4f}")
```

```
Optimization terminated successfully.
         Current function value: 0.577033
         Iterations 5
                           Logit Regression Results
==============================================================================
```

Appendix 18 – Refund probability single predictor regression code

```
[ ]:
```

```python
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.impute import SimpleImputer


# 1. Load datasets

df = pd.read_csv("MavenRail_cleaned2.csv")
df_pred = pd.read_csv("ToPredict_with_refund.csv")


# 2. Feature engineering

# Encode refund variable
df['Refund_Flag'] = df['Refund.Request'].map({'Yes': 1, 'No': 0})

# Convert times + calculate delays
for d in [df, df_pred]:
    d['Scheduled.Arrival'] = pd.to_datetime(d['Scheduled.Arrival'], errors='coerce')
    d['Actual.Arrival'] = pd.to_datetime(d['Actual.Arrival'], errors='coerce')
    d['Delay_Minutes'] = (d['Actual.Arrival'] - d['Scheduled.Arrival']).dt.total_seconds() / 60


# 3. Select explanatory variables

features = [
    "Price",
    "Ticket.Type",
    "Ticket.Class",
    "Railcard",
    "Journey.Status",
    "Delay_Minutes"
]

X = df[features]
y = df['Refund_Flag']
X_pred = df_pred[features]


# 4. Preprocessing with IMPUTATION
```

```python
# 4. Preprocessing with IMPUTATION


categorical_features = ["Ticket.Type", "Ticket.Class", "Railcard", "Journey.Status"]
numeric_features = ["Price", "Delay_Minutes"]

preprocess = ColumnTransformer(
    transformers=[
        ("categorical",
         Pipeline(steps=[
             ("imputer", SimpleImputer(strategy="most_frequent")),
             ("onehot", OneHotEncoder(handle_unknown="ignore"))
         ]),
         categorical_features
        ),
        ("numeric",
         Pipeline(steps=[
             ("imputer", SimpleImputer(strategy="median"))
         ]),
         numeric_features
        )
    ]
)


# 5. Logistic Regression Model

log_reg = LogisticRegression(max_iter=2000)

pipeline = Pipeline(steps=[
    ("preprocess", preprocess),
    ("model", log_reg)
])

# Train model
pipeline.fit(X, y)


# 6. Predict refund probabilities

df_pred["Refund_Probability"] = pipeline.predict_proba(X_pred)[:, 1]

# Save output
output_path = "Refund_Predictions_Logistic.csv"
df_pred.to_csv(output_path, index=False)

print("Predictions saved to:", output_path)
print(df_pred[["Refund_Probability"]].head())
```

```
Predictions saved to: Refund_Predictions_Logistic.csv
   Refund_Probability
0            0.000201
1            0.000209
2            0.352582
3            0.784001
4            0.393096
```

Appendix 19 – Question 5 model code