flight-price-predicition_final

May 27, 2024

1 Flight Price Prediction

In this notebook, we will consider the problem of modelling flight price predicition based on the data from Kaggle website.

1.1 Data Loading and Preparation

This section outlines the process of loading datasets, calculating distances between cities, and preparing the data for further analysis.

1.1.1 Step-by-Step Process

1. **Load Datasets**:
    - Load `business.csv`, `economy.csv`, `Clean_Dataset.csv`, and `Clean_Dataset_Updated.csv`.
2. **Print DataFrame Heads**:
    - Display the first few rows of each loaded DataFrame to understand their structure.
3. **Define City Coordinates**:
    - Create a dictionary containing latitude and longitude information for major cities.
4. **Identify Missing Cities**:
    - Identify cities present in the dataset but missing from the `locations` dictionary.
5. **Calculate Distance Using Haversine Formula**:
    - Define the Haversine formula to calculate the distance between two geographical points.
    - Apply this formula to each row in the dataset to calculate the distance between `source_city` and `destination_city`.
6. **Update and Inspect DataFrame**:
    - Add a `distance` column to `Clean_Dataset_Updated.csv`.
    - Inspect the updated DataFrame.

1.2 Import Necessary Libraries

First, we need to import the libraries that will be used throughout this notebook.

[ ]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import sklearn as skl
from sklearn import datasets
```

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error,mean_absolute_error, r2_score
from sklearn.preprocessing import PolynomialFeatures
from sklearn.feature_selection import mutual_info_regression
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_predict
from sklearn.model_selection import cross_validate
import sys
```

[ ]:
```
# Load datasets
business_df = pd.read_csv('../datasets/business.csv')
economy_df = pd.read_csv('../datasets/economy.csv')
clean_dataset = pd.read_csv('../datasets/Clean_Dataset.csv')
clean_dataset_updated = pd.read_csv('../datasets/Clean_Dataset_Updated.csv')
business_df.head()
economy_df.head()
clean_dataset.head()
clean_dataset_updated.head()
```

[ ]:
```
   Unnamed: 0   airline   flight source_city departure_time stops  \
0           0  SpiceJet  SG-8709       Delhi        Evening  zero
1           1  SpiceJet  SG-8157       Delhi  Early_Morning  zero
2           2   AirAsia   I5-764       Delhi  Early_Morning  zero
3           3   Vistara   UK-995       Delhi        Morning  zero
4           4   Vistara   UK-963       Delhi        Morning  zero

    arrival_time destination_city    class  duration  days_left  price  \
0          Night           Mumbai  Economy      2.17          1   5953
1        Morning           Mumbai  Economy      2.33          1   5953
2  Early_Morning           Mumbai  Economy      2.17          1   5956
3      Afternoon           Mumbai  Economy      2.25          1   5955
4        Morning           Mumbai  Economy      2.33          1   5955

  combined_date
0    2022-02-11
1    2022-02-11
2    2022-02-11
3    2022-02-11
4    2022-02-11
```

[ ]:
```
import pandas as pd
import numpy as np

# Create a dictionary containing city information
locations = {
```

```
    'Delhi': (28.7041, 77.1025),
    'Mumbai': (19.0760, 72.8777),
    'Bangalore': (12.9716, 77.5946),
    'Hyderabad': (17.3850, 78.4867),
    'Kolkata': (22.5726, 88.3639),
    'Chennai': (13.0827, 80.2707)
}

df = pd.read_csv('../datasets/Clean_Dataset_Updated.csv')
# Find the unique city names in the DataFrame
source_cities = set(df['source_city'].unique())
destination_cities = set(df['destination_city'].unique())
all_cities = source_cities.union(destination_cities)

# Find the missing cities in the locations dictionary
missing_cities = [city for city in all_cities if city not in locations]
print("The city lost in dictionary:", missing_cities)

def haversine(lat1, lon1, lat2, lon2):
    # Convert angles to radians
    lon1, lat1, lon2, lat2 = map(np.radians, [lon1, lat1, lon2, lat2])

    # Calculate the difference in latitude and longitude
    dlon = lon2 - lon1
    dlat = lat2 - lat1

    # Apply the Haversine formula
    a = np.sin(dlat/2.0)**2 + np.cos(lat1) * np.cos(lat2) * np.sin(dlon/2.0)**2
    c = 2 * np.arcsin(np.sqrt(a))
    # The Earth's radius is approximately 6371 kilometers
    km = 6371 * c
    return km

# Read the DataFrame
# clean_dataset = pd.read_csv('../datasets/Clean_Dataset.csv')
clean_dataset_updated = pd.read_csv('../datasets/Clean_Dataset_Updated.csv')
# Calculate the distance for each row and add it to a new column
clean_dataset_updated['distance'] = clean_dataset_updated.apply(lambda row:␣
↪haversine(locations[row['source_city']][0],
                                                                           ␣
↪locations[row['source_city']][1],
                                                                           ␣
↪locations[row['destination_city']][0],
                                                                           ␣
↪locations[row['destination_city']][1]), axis=1)

# View the updated DataFrame
```

```
clean_dataset_updated.head()
```

```
The city lost in dictionary: []
```

[ ]:
```
   Unnamed: 0   airline   flight source_city departure_time stops  \
0           0  SpiceJet  SG-8709       Delhi        Evening  zero
1           1  SpiceJet  SG-8157       Delhi  Early_Morning  zero
2           2   AirAsia   I5-764       Delhi  Early_Morning  zero
3           3   Vistara   UK-995       Delhi        Morning  zero
4           4   Vistara   UK-963       Delhi        Morning  zero

   arrival_time destination_city    class  duration  days_left  price  \
0         Night           Mumbai  Economy      2.17          1   5953
1       Morning           Mumbai  Economy      2.33          1   5953
2 Early_Morning           Mumbai  Economy      2.17          1   5956
3     Afternoon           Mumbai  Economy      2.25          1   5955
4       Morning           Mumbai  Economy      2.33          1   5955

  combined_date     distance
0    2022-02-11  1153.241291
1    2022-02-11  1153.241291
2    2022-02-11  1153.241291
3    2022-02-11  1153.241291
4    2022-02-11  1153.241291
```

[ ]:
```
clean_dataset.shape
clean_dataset.describe(include='all')
```

[ ]:
```
           Unnamed: 0  airline  flight source_city departure_time   stops  \
count   300153.000000   300153  300153      300153         300153  300153
unique            NaN        6    1561           6              6       3
top               NaN  Vistara  UK-706       Delhi        Morning     one
freq              NaN   127859    3235       61343          71146  250863
mean    150076.000000      NaN     NaN         NaN            NaN     NaN
std      86646.852011      NaN     NaN         NaN            NaN     NaN
min          0.000000      NaN     NaN         NaN            NaN     NaN
25%      75038.000000      NaN     NaN         NaN            NaN     NaN
50%     150076.000000      NaN     NaN         NaN            NaN     NaN
75%     225114.000000      NaN     NaN         NaN            NaN     NaN
max     300152.000000      NaN     NaN         NaN            NaN     NaN

       arrival_time destination_city    class       duration      days_left  \
count        300153           300153   300153  300153.000000  300153.000000
unique            6                6        2            NaN            NaN
top           Night           Mumbai  Economy            NaN            NaN
freq          91538            59097   206666            NaN            NaN
mean            NaN              NaN      NaN      12.221021      26.004751
std             NaN              NaN      NaN       7.191997      13.561004
```

```
min               NaN              NaN       NaN       0.830000       1.000000
25%               NaN              NaN       NaN       6.830000      15.000000
50%               NaN              NaN       NaN      11.250000      26.000000
75%               NaN              NaN       NaN      16.170000      38.000000
max               NaN              NaN       NaN      49.830000      49.000000

                price
count   300153.000000
unique            NaN
top               NaN
freq              NaN
mean     20889.660523
std      22697.767366
min       1105.000000
25%       4783.000000
50%       7425.000000
75%      42521.000000
max     123071.000000
```

[ ]:
```
clean_dataset.dropna(inplace=True)
clean_dataset.shape
```

[ ]:
```
(300153, 12)
```

[ ]:
```
clean_dataset.isnull().sum()
```

[ ]:
```
Unnamed: 0          0
airline             0
flight              0
source_city         0
departure_time      0
stops               0
arrival_time        0
destination_city    0
class               0
duration            0
days_left           0
price               0
dtype: int64
```

1.3 Integrating Dates from Business and Economy Datasets

In this section, we aim to update the `Clean_Dataset.csv` by incorporating dates from the `business.csv` and `economy.csv` datasets. The steps are as follows:

1. **Load the Datasets**: Load `business.csv`, `Clean_Dataset.csv`, and `economy.csv`.
2. **Format Dates Consistently**: Ensure the date columns in `business.csv` and `economy.csv` are in a consistent datetime format (`%d-%m-%Y`).

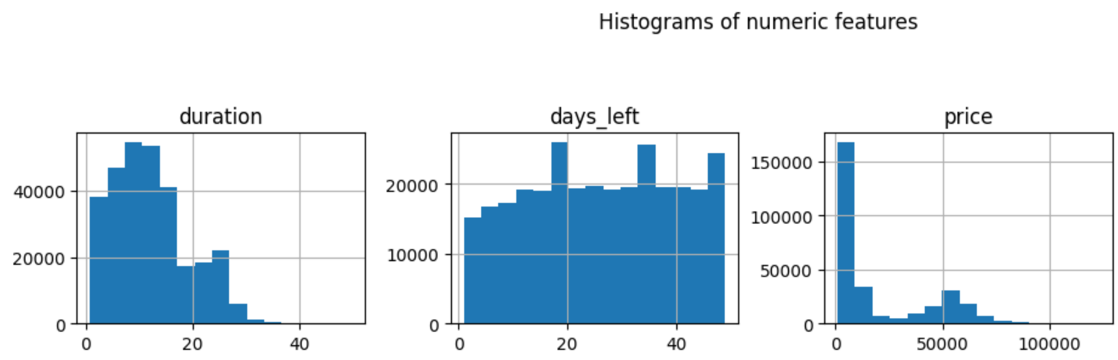
3. **Combine Date Columns**: Concatenate the date columns from `economy.csv` and `business.csv` into a single series.
4. **Truncate Combined Dates**: Truncate the combined dates to match the length of `Clean_Dataset.csv`.
5. **Update Clean Dataset**: Add the combined dates as a new column, `combined_date`, to `Clean_Dataset.csv`.
6. **Save the Updated DataFrame**: Save the updated DataFrame to `Clean_Dataset_Updated.csv`.

[ ]:
```python
import pandas as pd

# Load the datasets
business_df = pd.read_csv('../datasets/business.csv')
clean_dataset_df = pd.read_csv('../datasets/Clean_Dataset.csv')
economy_df = pd.read_csv('../datasets/economy.csv')

# Ensure the dates are in a consistent format
business_df['date'] = pd.to_datetime(business_df['date'], format='%d-%m-%Y')
economy_df['date'] = pd.to_datetime(economy_df['date'], format='%d-%m-%Y')

# Concatenate the date columns from economy and business
combined_dates = pd.concat([economy_df['date'], business_df['date']],␣
↪ignore_index=True)

# Ensure the combined_dates has the same length as cleandataset
combined_dates = combined_dates[:len(clean_dataset_df)]

# Add the combined dates to cleandataset
clean_dataset_df['combined_date'] = combined_dates

# Save the updated dataframe to a new CSV
updated_file_path = '../datasets/Clean_Dataset_Updated.csv'
clean_dataset_df.to_csv(updated_file_path, index=False)

print(f"Updated file saved to {updated_file_path}")
```

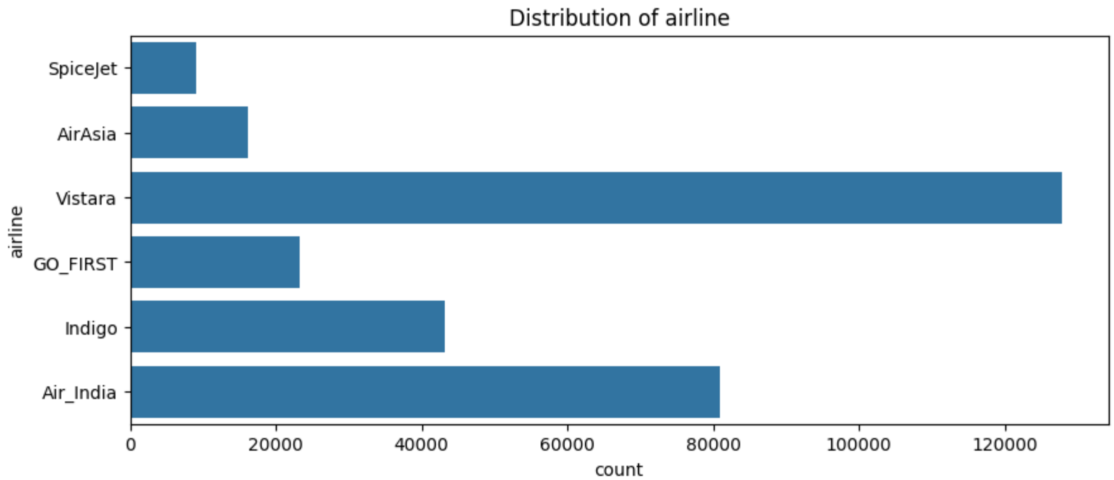
```
Updated file saved to ../datasets/Clean_Dataset_Updated.csv
```

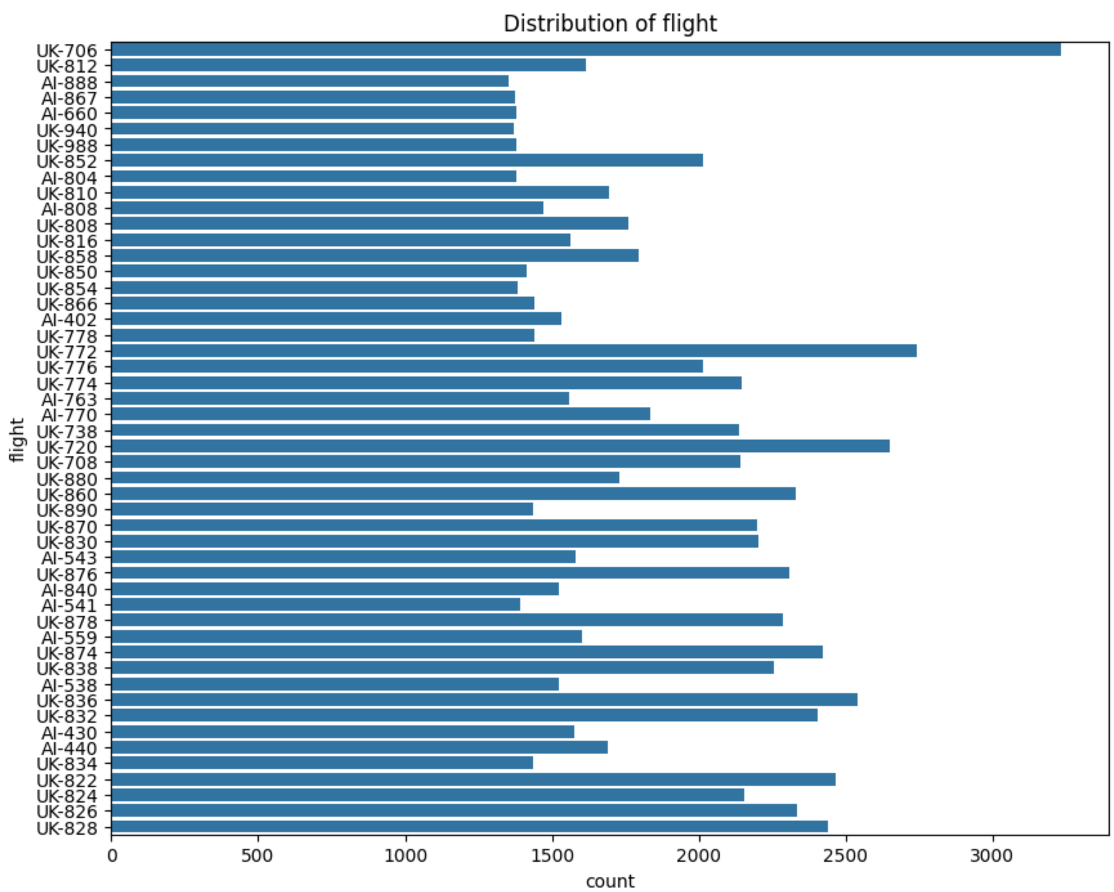
2 Let's visualize the first few rows of the dataset

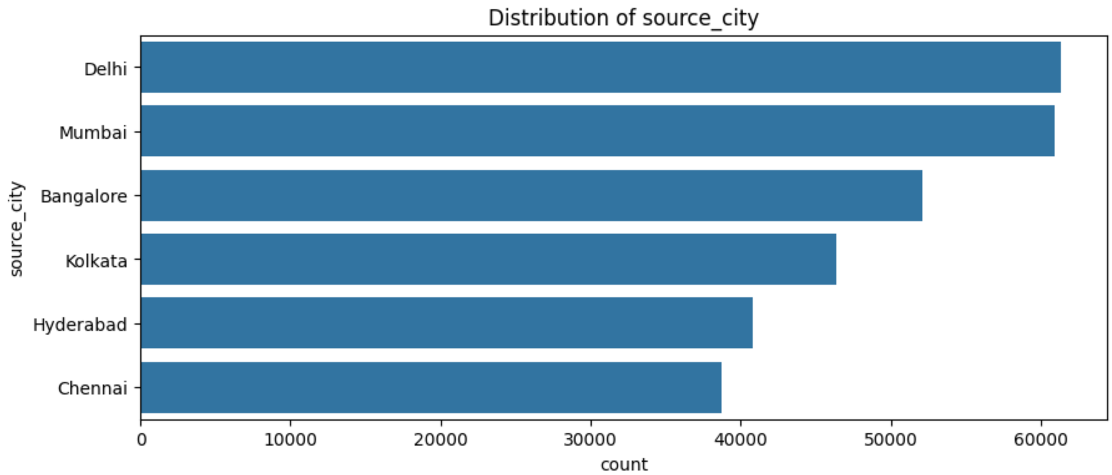
[ ]:
```python
# Plotting histograms for all numeric features to understand distributions
# exclude the unnamed column
clean_dataset.drop('Unnamed: 0', axis=1, inplace=True)
clean_dataset.hist(bins=15, figsize=(15, 10), layout=(4, 4))
plt.suptitle('Histograms of numeric features')
plt.show()
```

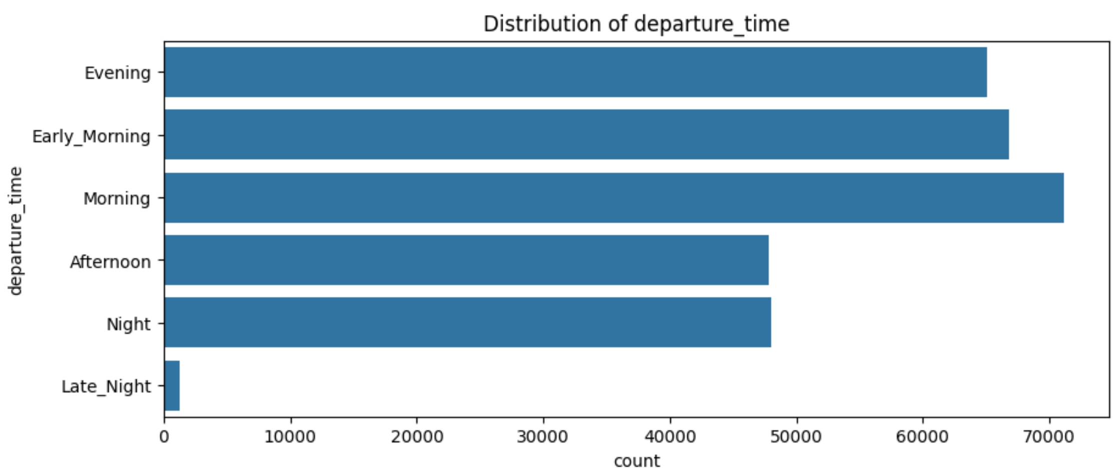


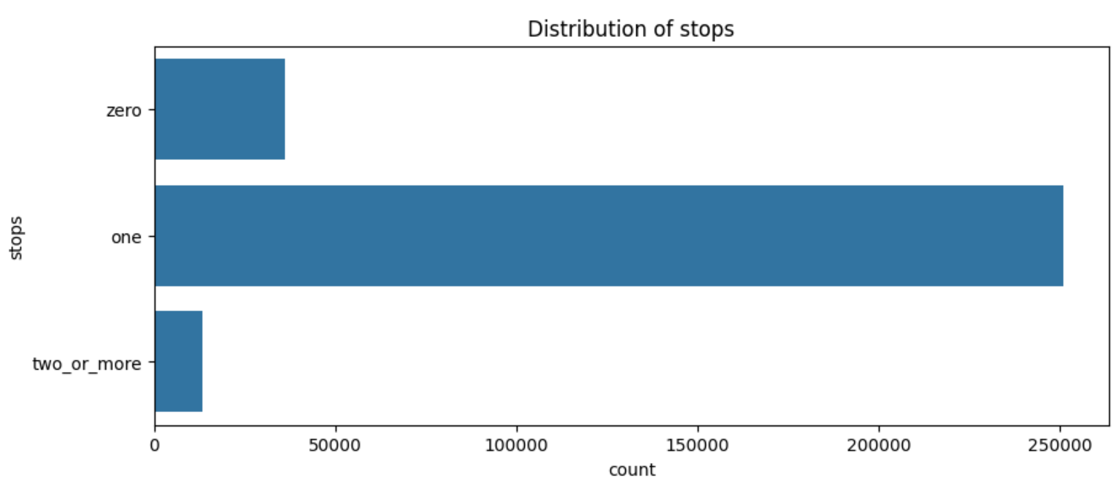

```
[ ]: # For categorical data, we can use count plots to understand the distribution␣
     ↪of categories
     for column in clean_dataset.select_dtypes(include=['object']).columns:
         # Plotting count plots for all categorical features
         # If the number of categories is too high,e.g., flight, we can filter the␣
     ↪top 50 categories to make the plot more readable
         if column != 'flight':
             plt.figure(figsize=(10, 4))
             sns.countplot(y=column, data=clean_dataset)
             plt.title(f'Distribution of {column}')
             plt.show()
         else:
             top_categories = clean_dataset[column].value_counts().index[:50]  # Get␣
     ↪top 50 categories
             filtered_data = clean_dataset[clean_dataset[column].
     ↪isin(top_categories)]
             plt.figure(figsize=(10, 8))
             sns.countplot(y=column, data=filtered_data)
             plt.yticks(fontsize=10)
             plt.title(f'Distribution of {column}')
             plt.show()
```

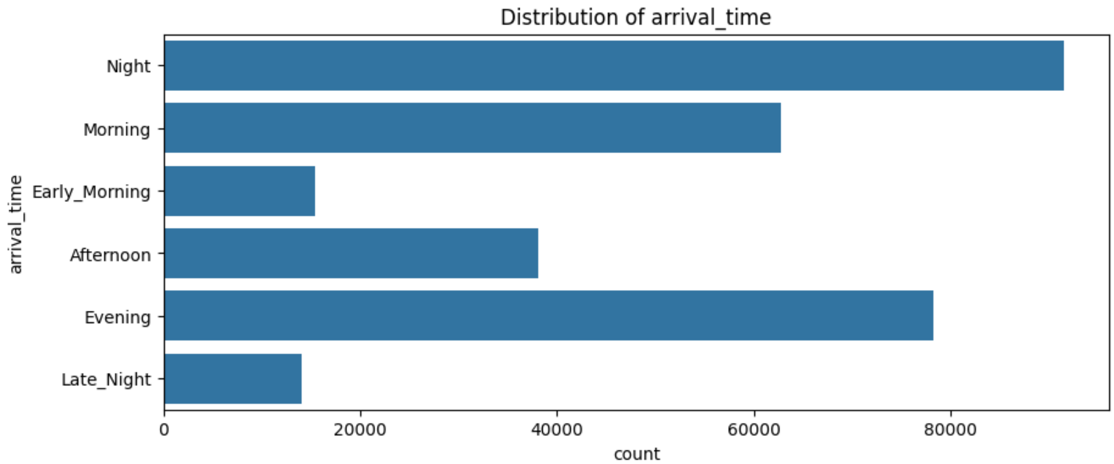

Distribution of airline
SpiceJet
AirAsia
Vistara
GO_FIRST
Indigo
Air_India
airline
0
20000
40000
60000
80000
100000
120000
count


Distribution of flight
UK-706
UK-812
AI-888
AI-867
AI-660
UK-940
UK-988
UK-852
AI-804
UK-810
AI-808
UK-808
UK-816
UK-858
UK-850
UK-854
UK-866
AI-402
UK-778
UK-772
UK-776
UK-774
AI-763
AI-770
UK-738
UK-720
UK-708
UK-880
UK-860
UK-890
UK-870
UK-830
AI-543
UK-876
AI-840
AI-541
UK-878
AI-559
UK-874
UK-838
AI-538
UK-836
UK-832
AI-430
AI-440
UK-834
UK-822
UK-824
UK-826
UK-828
flight
0
500
1000
1500
2000
2500
3000
count

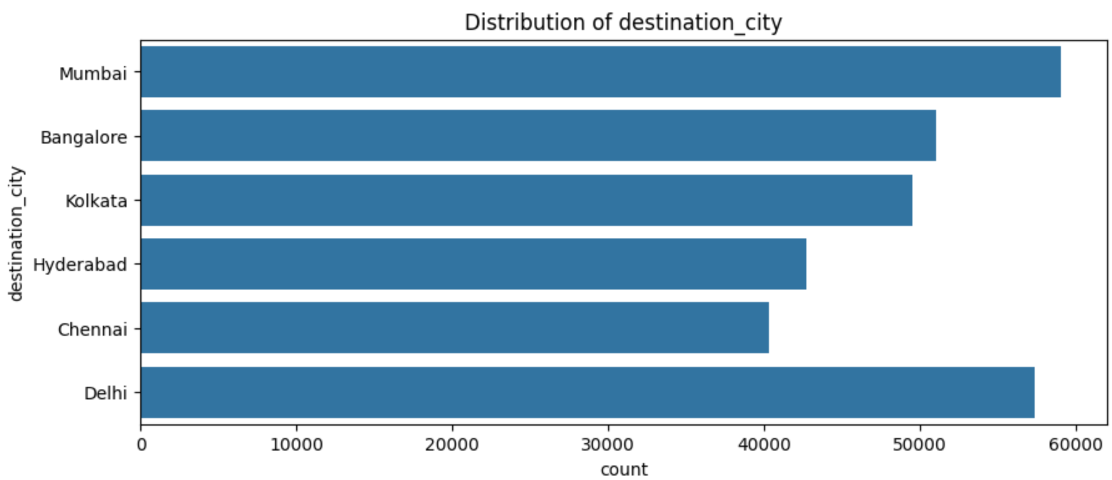

Distribution of source_city
Delhi
Mumbai
Bangalore
Kolkata
Hyderabad
Chennai
source_city
0
10000
20000
30000
40000
50000
60000
count


Distribution of departure_time
Evening
Early_Morning
Morning
Afternoon
Night
Late_Night
departure_time
0
10000
20000
30000
40000
50000
60000
70000
count

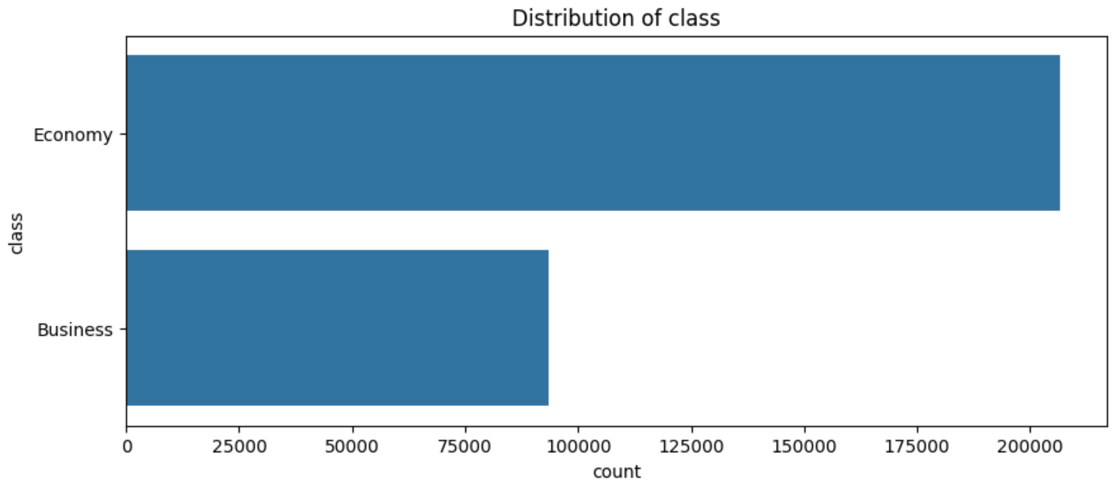

Distribution of stops
zero
one
two_or_more
stops
0
50000
100000
150000
200000
250000
count



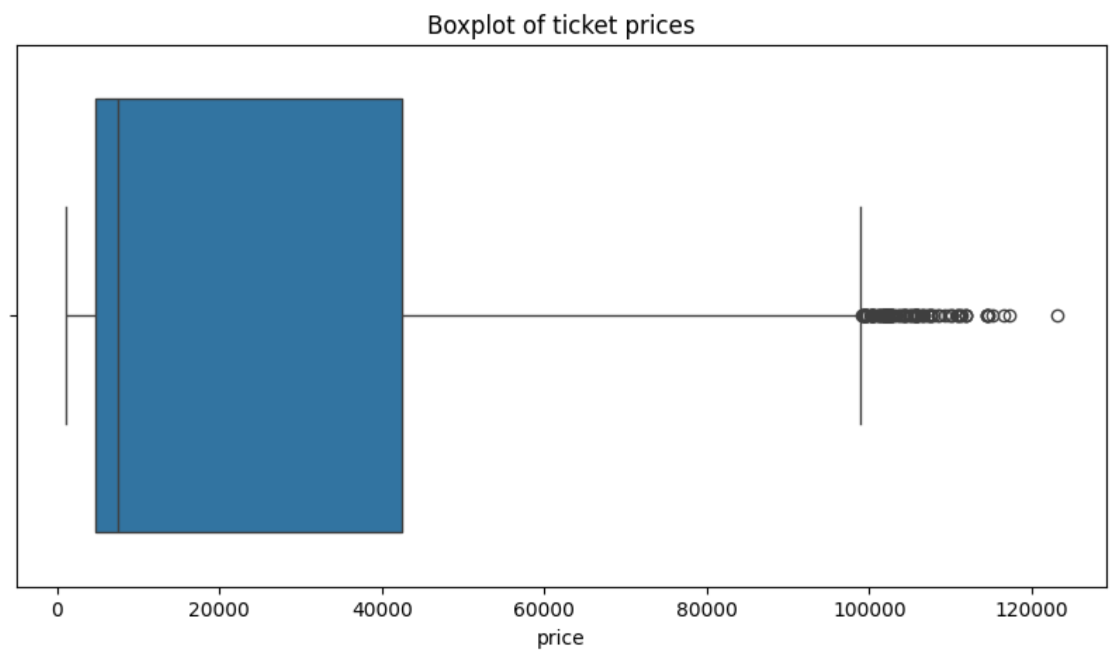
Distribution of arrival_time
Night
Morning
Early_Morning
Afternoon
Evening
Late_Night
arrival_time
0
20000
40000
60000
80000
count



Distribution of destination_city
Mumbai
Bangalore
Kolkata
Hyderabad
Chennai
Delhi
destination_city
0
10000
20000
30000
40000
50000
60000
count

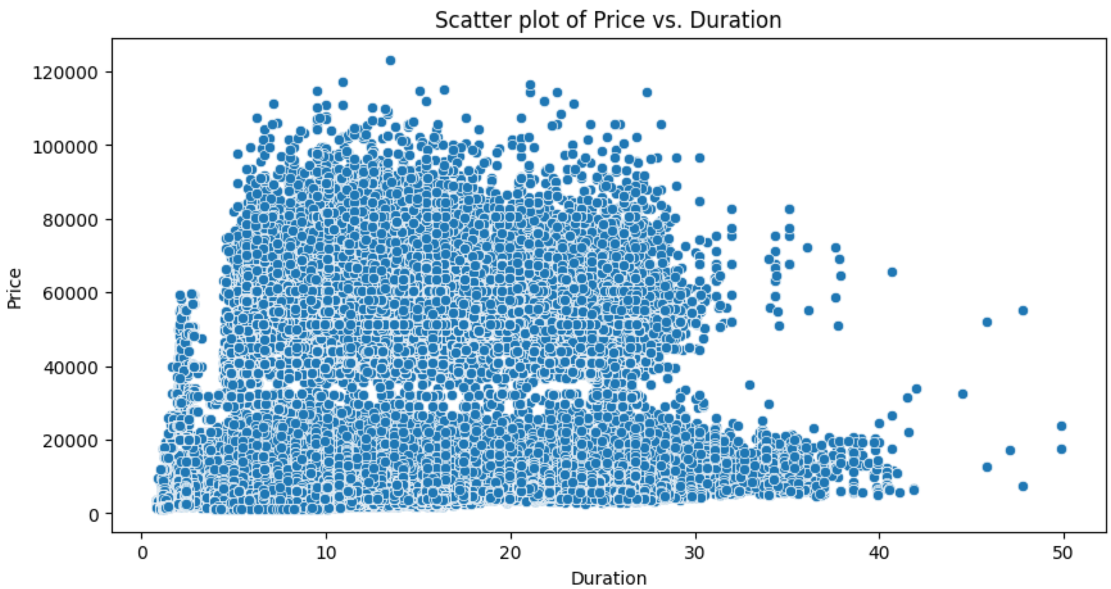


[ ]:
```
# Boxplot for the price column to see its distribution and spot any outliers
plt.figure(figsize=(10, 5))
sns.boxplot(x=clean_dataset['price'])
plt.title('Boxplot of ticket prices')
plt.show()
```



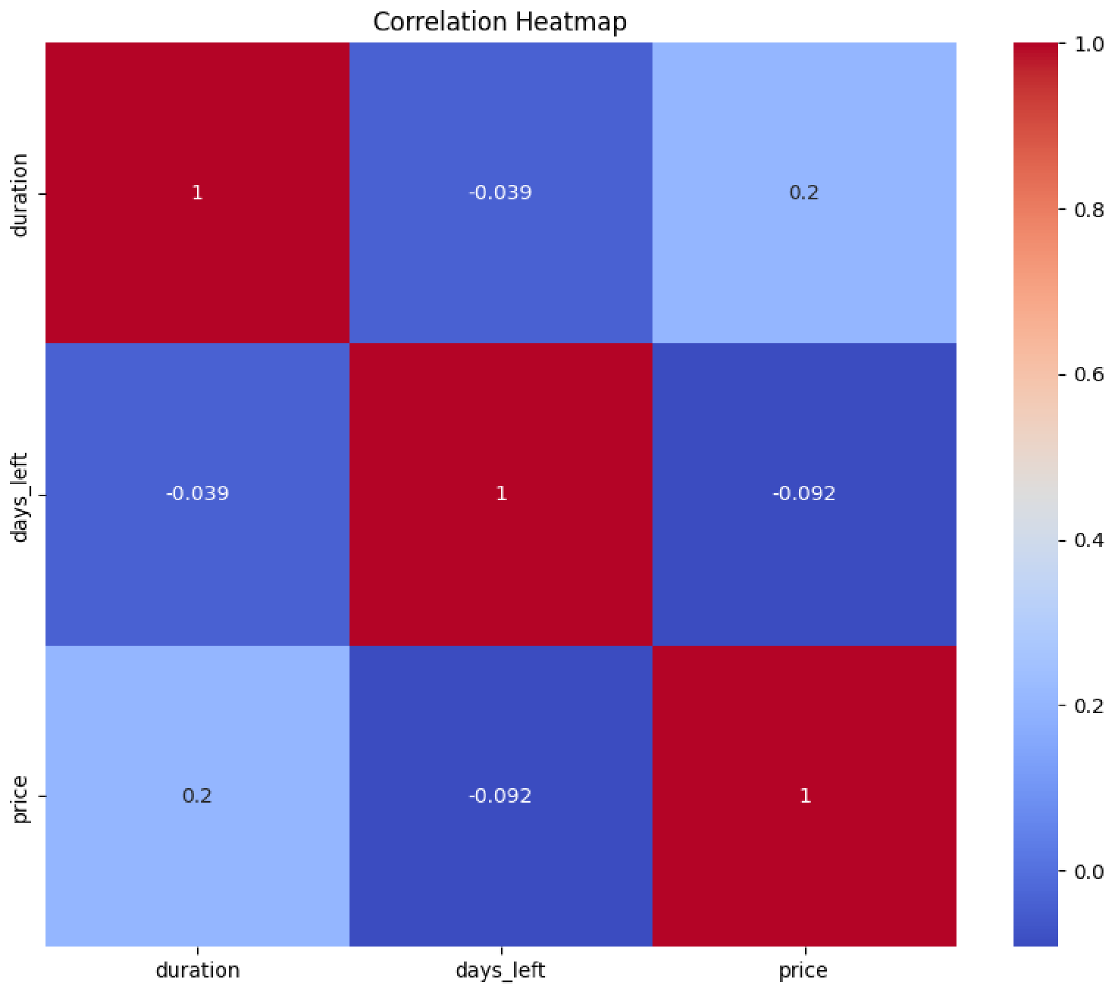
[ ]:
```
# A scatter plot to visualize the relationship between two variables, for␣
↪example, price and duration
```

```python
plt.figure(figsize=(10, 5))
sns.scatterplot(x=clean_dataset['duration'], y=clean_dataset['price'])
plt.title('Scatter plot of Price vs. Duration')
plt.xlabel('Duration')
plt.ylabel('Price')
plt.show()
```



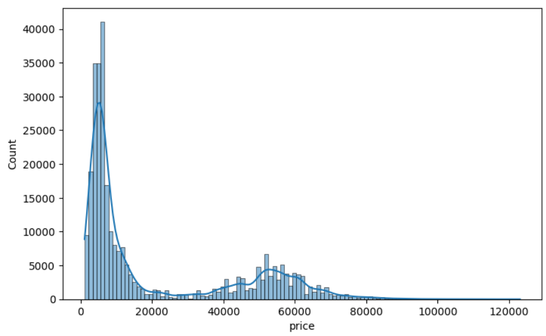
[ ]:
```python
# Correlation heatmap to understand the relationships between variables
# Select only the numeric columns for correlation
numeric_dataset = clean_dataset.select_dtypes(include=[np.number])
correlation_matrix = numeric_dataset.corr()
```

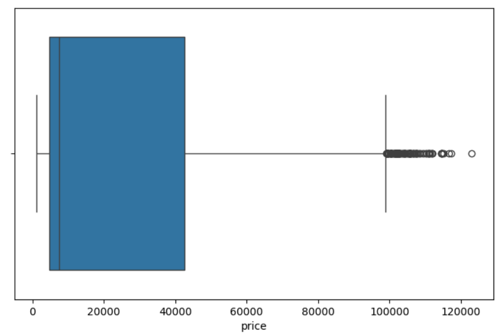
[ ]:
```python
# Visualize the correlation matrix
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm')
plt.title('Correlation Heatmap')
plt.show()
```



[ ]:
```
plt.figure(figsize = (18,5))
plt.subplot(1,2,1)
sns.histplot(x = 'price', data = clean_dataset, kde = True)
plt.subplot(1,2,2)
sns.boxplot(x = 'price', data = clean_dataset)
```

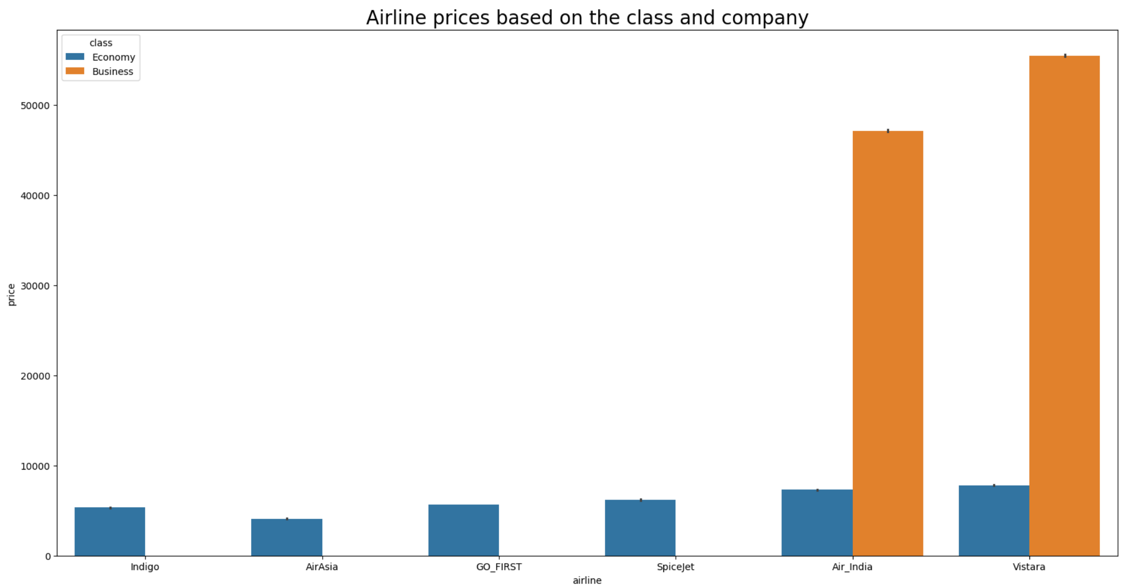
[ ]: <Axes: xlabel='price'>





[ ]:
```
plt.figure(figsize=(20, 10))
sns.barplot(x='airline',y='price',hue="class",data=clean_dataset.
  ↪sort_values("price")).set_title('Airline prices based on the class and␣
  ↪company',fontsize=20)
```

[ ]:
```
Text(0.5, 1.0, 'Airline prices based on the class and company')
```



3 Make data transformation

This section describes the steps taken to transform the `clean_dataset` to prepare it for further analysis or modeling. The transformation includes adding new features, encoding categorical variables, and dropping unnecessary columns. The steps are as follows:

1. **Create a Copy of the Dataset**: A copy of `clean_dataset` is made to ensure the original dataset remains unchanged.

2. **Encode Class Column**:
   - Add a new column `Economy` to indicate if the class is 'Economy'.
   - Drop the original `class` column.
3. **Map City Population Sizes**:
   - Replace `source_city` and `destination_city` with their respective population sizes (data from 2011).
   - Drop the original `source_city` and `destination_city` columns.
4. **One-Hot Encoding for Time Columns**:
   - Perform one-hot encoding on `departure_time` and `arrival_time` columns.
5. **Map Stops to Numerical Values**:
   - Replace `stops` with numerical values.
   - Drop the original `stops` column.
6. **One-Hot Encoding for Airline Column**:
   - Perform one-hot encoding on the `airline` column.

[ ]:
```
transformed_dataset = clean_dataset.copy()
transformed_dataset['Economy'] = clean_dataset['class'] == 'Economy'
transformed_dataset.drop('class', axis=1, inplace=True)
```

[ ]:
```
#transformed_dataset['source_city'].unique()
```

[ ]:
```
city_size = { # this is for year 2011 - https://en.wikipedia.org/wiki/
 ↪List_of_cities_in_India_by_population
    'Delhi': 110,
    'Mumbai': 124,
    'Bangalore': 84,
    'Kolkata': 44,
    'Hyderabad': 69,
    'Chennai' : 46
}
transformed_dataset['source_size'] = transformed_dataset['source_city'].
 ↪replace(city_size)
transformed_dataset.drop('source_city', axis=1, inplace=True)
transformed_dataset['destination_size'] =␣
 ↪transformed_dataset['destination_city'].replace(city_size)
transformed_dataset.drop('destination_city', axis=1, inplace=True)
```

```
/tmp/ipykernel_2570/2533689888.py:9: FutureWarning: Downcasting behavior in
`replace` is deprecated and will be removed in a future version. To retain the
old behavior, explicitly call `result.infer_objects(copy=False)`. To opt-in to
the future behavior, set `pd.set_option('future.no_silent_downcasting', True)`
  transformed_dataset['source_size'] =
transformed_dataset['source_city'].replace(city_size)
```

```
/tmp/ipykernel_2570/2533689888.py:11: FutureWarning: Downcasting behavior in
`replace` is deprecated and will be removed in a future version. To retain the
old behavior, explicitly call `result.infer_objects(copy=False)`. To opt-in to
the future behavior, set `pd.set_option('future.no_silent_downcasting', True)`
  transformed_dataset['destination_size'] =
transformed_dataset['destination_city'].replace(city_size)
```

[ ]:
```
transformed_dataset = pd.
 ↪get_dummies(transformed_dataset,columns=['departure_time','arrival_time'])
```

[ ]:
```
stops = {
    'zero': 0,
    'one': 1,
    'two_or_more': 2,
}
transformed_dataset['stops_num'] = transformed_dataset['stops'].replace(stops)
transformed_dataset.drop('stops', axis=1, inplace=True)
```

```
/tmp/ipykernel_2570/3828070998.py:6: FutureWarning: Downcasting behavior in
`replace` is deprecated and will be removed in a future version. To retain the
old behavior, explicitly call `result.infer_objects(copy=False)`. To opt-in to
the future behavior, set `pd.set_option('future.no_silent_downcasting', True)`
  transformed_dataset['stops_num'] = transformed_dataset['stops'].replace(stops)
```

[ ]:
```
transformed_dataset = pd.get_dummies(transformed_dataset,columns=['airline'])
```

[ ]:
```
transformed_dataset['flight_num'] = pd.
 ↪factorize(transformed_dataset['flight'])[0]
transformed_dataset.drop('flight', axis=1, inplace=True)
```

[ ]:
```
transformed_dataset.head()
```

[ ]:
```
   duration  days_left  price  Economy  source_size  destination_size  \
0      2.17          1   5953     True          110               124
1      2.33          1   5953     True          110               124
2      2.17          1   5956     True          110               124
3      2.25          1   5955     True          110               124
4      2.33          1   5955     True          110               124

   departure_time_Afternoon  departure_time_Early_Morning  \
0                     False                         False
1                     False                          True
2                     False                          True
3                     False                         False
4                     False                         False

   departure_time_Evening  departure_time_Late_Night  …  \
0                    True                      False  …
```

```
1                    False                          False  ...
2                    False                          False  ...
3                    False                          False  ...
4                    False                          False  ...

   arrival_time_Morning  arrival_time_Night  stops_num  airline_AirAsia  \
0                 False                True          0            False
1                  True               False          0            False
2                 False               False          0             True
3                 False               False          0            False
4                  True               False          0            False

   airline_Air_India  airline_GO_FIRST  airline_Indigo  airline_SpiceJet  \
0              False             False           False              True
1              False             False           False              True
2              False             False           False             False
3              False             False           False             False
4              False             False           False             False

   airline_Vistara  flight_num
0            False           0
1            False           1
2            False           2
3             True           3
4             True           4

[5 rows x 26 columns]
```

[ ]:
```
transformed_dataset.describe()
# output the transformed dataset to a new CSV file
transformed_dataset.to_csv('../datasets/Transformed_Dataset.csv', index=False)
```

3.1 MCMC Model Training and Evaluation

This section details the process of training and evaluating a model using the `clean_dataset_updated` dataset. The steps include preparing the dataset, encoding features, calculating transition matrices, and evaluating model accuracy using 5-fold cross-validation.

3.1.1 Data Preparation and Feature Engineering

1. **Ensure Date Column Type**:
   - Convert the `combined_date` column to datetime type.
   - Extract day of the week, week of the year, and month from `combined_date`.
   - Identify holidays in India and mark them in the dataset.
2. **Create Route-Class Identifier**:
   - Combine `source_city`, `destination_city`, and `class` into a single identifier column `route_class`.
3. **Discretize Price**:

- Use `KBinsDiscretizer` to divide the `price` column into 5 intervals (`price_bin`).

4. **Group Days Left**:
   - Bin the `days_left` column into 10 intervals (`days_left_bin`).

3.1.2 Cross-Validation Setup

- Initialize 5-fold cross-validation.
- Prepare dictionaries to store cross-validation results and accuracies for each route-class.

3.1.3 Transition Matrix Calculation

- For each training fold:
  - Calculate transition matrices for each route-class based on time features, holidays, and days left.
  - Determine the most common initial price state for each route-class.

3.1.4 Model Evaluation

- For each test fold:
  - Evaluate model accuracy for each route-class using the transition matrices.
  - Calculate overall fold accuracy and store results.

3.1.5 Output Results

- Print model accuracy for each fold and the average accuracy across all folds.
- Print model accuracy for each route-class.
- Calculate and print overall MSE, R^2, and RMSE.
- Display likely price predictions for each route-class based on transition matrices.

[ ]:
```
import pandas as pd
import numpy as np
from sklearn.preprocessing import KBinsDiscretizer
from sklearn.model_selection import train_test_split
import holidays

# Assume that the 'clean_dataset_updated' has a 'combined_date' column
clean_dataset_updated['combined_date'] = pd.
 ↪to_datetime(clean_dataset_updated['combined_date'])

# Ensure that the 'combined_date' column has been converted to datetime type
if clean_dataset_updated['combined_date'].dtype == '<M8[ns]':
    clean_dataset_updated['day_of_week'] =␣
 ↪clean_dataset_updated['combined_date'].dt.dayofweek
    clean_dataset_updated['week_of_year'] =␣
 ↪clean_dataset_updated['combined_date'].dt.isocalendar().week
    clean_dataset_updated['month'] = clean_dataset_updated['combined_date'].dt.
 ↪month

    # Get holidays in India
```

```
    india_holidays = holidays.country_holidays('IN',␣
↪years=clean_dataset_updated['combined_date'].dt.year.unique())
    clean_dataset_updated['is_holiday'] =␣
↪clean_dataset_updated['combined_date'].isin(india_holidays)
```

```
/tmp/ipykernel_2570/2493624178.py:18: FutureWarning: The behavior of 'isin' with
dtype=datetime64[ns] and castable values (e.g. strings) is deprecated. In a
future version, these will not be considered matching by isin. Explicitly cast
to the appropriate dtype before calling isin instead.
  clean_dataset_updated['is_holiday'] =
clean_dataset_updated['combined_date'].isin(india_holidays)
```

[ ]:
```
# Create a combination identifier column for route and class categories
clean_dataset_updated['route_class'] = clean_dataset_updated['source_city'] +␣
↪'-' + clean_dataset_updated['destination_city'] + '-' +␣
↪clean_dataset_updated['class']
```

[ ]:
```
# Use KBinsDiscretizer to divide the price into several intervals
n_bins = 5
binning = KBinsDiscretizer(n_bins=n_bins, encode='ordinal', strategy='uniform')
clean_dataset_updated['price_bin'] = binning.
↪fit_transform(clean_dataset_updated[['price']]).astype(int)

# Group the days_left to reduce the number of unique values
days_left_bins = 10
clean_dataset_updated['days_left_bin'] = pd.
↪cut(clean_dataset_updated['days_left'], bins=days_left_bins, labels=False)

max_bin = clean_dataset_updated['price_bin'].max() + 1  # Maximum number of␣
↪states in the entire dataset
```

```
/home/siyan/.local/lib/python3.10/site-
packages/sklearn/preprocessing/_discretization.py:248: FutureWarning: In version
1.5 onwards, subsample=200_000 will be used by default. Set subsample explicitly
to silence this warning in the mean time. Set subsample=None to disable
subsampling explicitly.
  warnings.warn(
```

[ ]:
```
# Use KFold for 5-fold cross-validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)

# Prepare a dictionary to store cross-validation results
cv_accuracies = []
route_class_accuracies = {route_class: [] for route_class in␣
↪clean_dataset_updated['route_class'].unique()}

# Initialize global lists to store all actual values and predictions
all_actuals = []
```

```
all_predictions = []
```

[ ]:
```
import time
start_time = time.time()

for train_index, test_index in kf.split(clean_dataset_updated):
    train_data = clean_dataset_updated.iloc[train_index]
    test_data = clean_dataset_updated.iloc[test_index]

    # Prepare two dictionaries to store the transition matrix and the most␣
↪common initial price state for each route
    route_class_matrices = {}
    route_class_common_initial_states = {}

    # Calculate the transition probability matrix for each route
    for route_class in train_data['route_class'].unique():
        sub_df = train_data[train_data['route_class'] == route_class]
        transition_matrix = np.zeros((max_bin, max_bin))

        # Calculate the transition probability matrix based on time features,␣
↪holidays, and days_left_bin
        for is_holiday in [True, False]:
            for day_of_week in range(7):
                for days_left_bin in range(days_left_bins):
                    filtered_data = sub_df[
                        (sub_df['is_holiday'] == is_holiday) &
                        (sub_df['day_of_week'] == day_of_week) &
                        (sub_df['days_left_bin'] == days_left_bin)
                    ]
                    for i in range(len(filtered_data) - 1):
                        current_state = int(filtered_data.iloc[i]['price_bin'])
                        next_state = int(filtered_data.iloc[i + 1]['price_bin'])
                        transition_matrix[current_state, next_state] += 1

        # Convert counts to probabilities
        row_sums = transition_matrix.sum(axis=1)
        for i in range(len(row_sums)):
            if row_sums[i] != 0:
                transition_matrix[i] /= row_sums[i]

        route_class_matrices[route_class] = transition_matrix
        # Find the most common initial price state for each route
        most_common_initial_state = np.argmax(sub_df['price_bin'].
↪value_counts().values)
        route_class_common_initial_states[route_class] =␣
↪most_common_initial_state
```

```
    # Testing phase: Evaluate the model fit for each route
    accuracies = {}

    for route_class in test_data['route_class'].unique():
        sub_df = test_data[test_data['route_class'] == route_class]
        if route_class in route_class_matrices:
            correct_predictions = 0
            total_predictions = 0

            for i in range(len(sub_df) - 1):
                current_state = int(sub_df.iloc[i]['price_bin'])
                next_state = int(sub_df.iloc[i + 1]['price_bin'])
                predicted_state = np.
↪argmax(route_class_matrices[route_class][current_state])

                if predicted_state == next_state:
                    correct_predictions += 1
                total_predictions += 1

                # Collect actual values and predicted values
                all_actuals.append(next_state)
                all_predictions.append(predicted_state)

            if total_predictions > 0:
                accuracies[route_class] = correct_predictions /␣
↪total_predictions
                route_class_accuracies[route_class].
↪append(accuracies[route_class])

    # Calculate the average accuracy of the current fold
    fold_accuracy = np.mean(list(accuracies.values()))
    cv_accuracies.append(fold_accuracy)
```

[ ]:

```
# Output the model accuracy for each fold
for i, accuracy in enumerate(cv_accuracies):
    print(f"Fold {i+1} model accuracy: {accuracy:.2f}")

# Output the average model accuracy across all folds
print(f"Average model accuracy over 5 folds: {np.mean(cv_accuracies):.2f}")
```

```
Fold 1 model accuracy: 0.91
Fold 2 model accuracy: 0.91
Fold 3 model accuracy: 0.91
Fold 4 model accuracy: 0.91
Fold 5 model accuracy: 0.91
Average model accuracy over 5 folds: 0.91
```

[ ]:

```python
# Output the model accuracy for each route class and cabin
for route_class, accuracies in route_class_accuracies.items():
    if len(accuracies) > 0:
        print(f"Average model accuracy for route and class {route_class}: {np.
 ↪mean(accuracies):.2f}")
    else:
        print(f"Not enough data to evaluate model accuracy for route and class␣
 ↪{route_class}")
```

```
Average model accuracy for route and class Delhi-Mumbai-Economy: 1.00
Average model accuracy for route and class Delhi-Bangalore-Economy: 1.00
Average model accuracy for route and class Delhi-Kolkata-Economy: 1.00
Average model accuracy for route and class Delhi-Hyderabad-Economy: 1.00
Average model accuracy for route and class Delhi-Chennai-Economy: 1.00
Average model accuracy for route and class Mumbai-Delhi-Economy: 1.00
Average model accuracy for route and class Mumbai-Bangalore-Economy: 1.00
Average model accuracy for route and class Mumbai-Kolkata-Economy: 1.00
Average model accuracy for route and class Mumbai-Hyderabad-Economy: 1.00
Average model accuracy for route and class Mumbai-Chennai-Economy: 1.00
Average model accuracy for route and class Bangalore-Delhi-Economy: 1.00
Average model accuracy for route and class Bangalore-Mumbai-Economy: 1.00
Average model accuracy for route and class Bangalore-Kolkata-Economy: 1.00
Average model accuracy for route and class Bangalore-Hyderabad-Economy: 1.00
Average model accuracy for route and class Bangalore-Chennai-Economy: 1.00
Average model accuracy for route and class Kolkata-Delhi-Economy: 1.00
Average model accuracy for route and class Kolkata-Mumbai-Economy: 1.00
Average model accuracy for route and class Kolkata-Bangalore-Economy: 1.00
Average model accuracy for route and class Kolkata-Hyderabad-Economy: 1.00
Average model accuracy for route and class Kolkata-Chennai-Economy: 1.00
Average model accuracy for route and class Hyderabad-Delhi-Economy: 1.00
Average model accuracy for route and class Hyderabad-Mumbai-Economy: 1.00
Average model accuracy for route and class Hyderabad-Bangalore-Economy: 1.00
Average model accuracy for route and class Hyderabad-Kolkata-Economy: 1.00
Average model accuracy for route and class Hyderabad-Chennai-Economy: 1.00
Average model accuracy for route and class Chennai-Delhi-Economy: 1.00
Average model accuracy for route and class Chennai-Mumbai-Economy: 1.00
Average model accuracy for route and class Chennai-Bangalore-Economy: 0.99
Average model accuracy for route and class Chennai-Kolkata-Economy: 0.99
Average model accuracy for route and class Chennai-Hyderabad-Economy: 1.00
Average model accuracy for route and class Delhi-Mumbai-Business: 0.86
Average model accuracy for route and class Delhi-Bangalore-Business: 0.87
Average model accuracy for route and class Delhi-Kolkata-Business: 0.81
Average model accuracy for route and class Delhi-Hyderabad-Business: 0.75
Average model accuracy for route and class Delhi-Chennai-Business: 0.81
Average model accuracy for route and class Mumbai-Delhi-Business: 0.85
Average model accuracy for route and class Mumbai-Bangalore-Business: 0.82
Average model accuracy for route and class Mumbai-Kolkata-Business: 0.84
Average model accuracy for route and class Mumbai-Hyderabad-Business: 0.81
```

```
Average model accuracy for route and class Mumbai-Chennai-Business: 0.78
Average model accuracy for route and class Bangalore-Delhi-Business: 0.87
Average model accuracy for route and class Bangalore-Mumbai-Business: 0.80
Average model accuracy for route and class Bangalore-Kolkata-Business: 0.91
Average model accuracy for route and class Bangalore-Hyderabad-Business: 0.83
Average model accuracy for route and class Bangalore-Chennai-Business: 0.80
Average model accuracy for route and class Kolkata-Delhi-Business: 0.77
Average model accuracy for route and class Kolkata-Mumbai-Business: 0.83
Average model accuracy for route and class Kolkata-Bangalore-Business: 0.92
Average model accuracy for route and class Kolkata-Hyderabad-Business: 0.76
Average model accuracy for route and class Kolkata-Chennai-Business: 0.94
Average model accuracy for route and class Hyderabad-Delhi-Business: 0.74
Average model accuracy for route and class Hyderabad-Mumbai-Business: 0.78
Average model accuracy for route and class Hyderabad-Bangalore-Business: 0.81
Average model accuracy for route and class Hyderabad-Kolkata-Business: 0.77
Average model accuracy for route and class Hyderabad-Chennai-Business: 0.76
Average model accuracy for route and class Chennai-Delhi-Business: 0.80
Average model accuracy for route and class Chennai-Mumbai-Business: 0.73
Average model accuracy for route and class Chennai-Bangalore-Business: 0.81
Average model accuracy for route and class Chennai-Kolkata-Business: 0.92
Average model accuracy for route and class Chennai-Hyderabad-Business: 0.73
```

[ ]:
```python
# Calculate and output overall MSE, R^2, RMSE
if all_actuals and all_predictions:
    mse = mean_squared_error(all_actuals, all_predictions)
    r2 = r2_score(all_actuals, all_predictions)
    rmse = np.sqrt(mse)
    print(f"Overall MSE: {mse:.2f}, R^2: {r2:.2f}, RMSE: {rmse:.2f}")

end_time = time.time()
print(f"Total running time: {end_time - start_time:.2f} seconds")
```

```
Overall MSE: 0.11, R^2: 0.85, RMSE: 0.32
Total running time: 318.79 seconds
```

[ ]:
```python
# Get the boundaries of the price bins
bin_edges = binning.bin_edges_[0]

# Output price predictions
for route_class, matrix in route_class_matrices.items():
    initial_state = route_class_common_initial_states[route_class]
    likely_next_state = np.argmax(matrix[initial_state])
    initial_price_range = f"{bin_edges[initial_state]:.2f} to␣
 ↪{bin_edges[initial_state + 1]:.2f}"
    next_price_range = f"{bin_edges[likely_next_state]:.2f} to␣
 ↪{bin_edges[likely_next_state + 1]:.2f}"
    print(f"Route and class {route_class}:")
```

```
    print(f"  Most common initial price state: {initial_state}␣
↪({initial_price_range})")
    print(f"  Predicted next most likely price state: {likely_next_state}␣
↪({next_price_range})")
    print()
```

```
Route and class Delhi-Mumbai-Economy:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Delhi-Bangalore-Economy:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Delhi-Kolkata-Economy:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Delhi-Hyderabad-Economy:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Delhi-Chennai-Economy:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Mumbai-Delhi-Economy:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Mumbai-Bangalore-Economy:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Mumbai-Kolkata-Economy:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Mumbai-Hyderabad-Economy:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Mumbai-Chennai-Economy:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Bangalore-Delhi-Economy:
```

```
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Bangalore-Mumbai-Economy:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Bangalore-Kolkata-Economy:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Bangalore-Hyderabad-Economy:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Bangalore-Chennai-Economy:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Kolkata-Delhi-Economy:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Kolkata-Mumbai-Economy:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Kolkata-Bangalore-Economy:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Kolkata-Hyderabad-Economy:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Kolkata-Chennai-Economy:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Hyderabad-Delhi-Economy:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Hyderabad-Mumbai-Economy:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Hyderabad-Bangalore-Economy:
```

```
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Hyderabad-Kolkata-Economy:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Hyderabad-Chennai-Economy:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Chennai-Delhi-Economy:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Chennai-Mumbai-Economy:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Chennai-Bangalore-Economy:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Chennai-Kolkata-Economy:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Chennai-Hyderabad-Economy:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Delhi-Mumbai-Business:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Delhi-Bangalore-Business:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Delhi-Kolkata-Business:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Delhi-Hyderabad-Business:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Delhi-Chennai-Business:
```

```
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Mumbai-Delhi-Business:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Mumbai-Bangalore-Business:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Mumbai-Kolkata-Business:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Mumbai-Hyderabad-Business:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Mumbai-Chennai-Business:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Bangalore-Delhi-Business:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Bangalore-Mumbai-Business:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Bangalore-Kolkata-Business:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 1 (25498.20 to 49891.40)

Route and class Bangalore-Hyderabad-Business:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 1 (25498.20 to 49891.40)

Route and class Bangalore-Chennai-Business:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Kolkata-Delhi-Business:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Kolkata-Mumbai-Business:
```

```
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Kolkata-Bangalore-Business:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 1 (25498.20 to 49891.40)

Route and class Kolkata-Hyderabad-Business:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 1 (25498.20 to 49891.40)

Route and class Kolkata-Chennai-Business:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 2 (49891.40 to 74284.60)

Route and class Hyderabad-Delhi-Business:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Hyderabad-Mumbai-Business:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Hyderabad-Bangalore-Business:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 1 (25498.20 to 49891.40)

Route and class Hyderabad-Kolkata-Business:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 1 (25498.20 to 49891.40)

Route and class Hyderabad-Chennai-Business:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 1 (25498.20 to 49891.40)

Route and class Chennai-Delhi-Business:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Chennai-Mumbai-Business:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 0 (1105.00 to 25498.20)

Route and class Chennai-Bangalore-Business:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 1 (25498.20 to 49891.40)

Route and class Chennai-Kolkata-Business:
```

```
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 2 (49891.40 to 74284.60)

Route and class Chennai-Hyderabad-Business:
  Most common initial price state: 0 (1105.00 to 25498.20)
  Predicted next most likely price state: 1 (25498.20 to 49891.40)
```

3.2 Combining Linear Regression with DNN and PGM Features for Price Prediction

This section describes the implementation of a model that combines linear regression, deep neural networks (DNN), and probabilistic graphical models (PGM) to predict flight prices. The process includes data preprocessing, feature engineering, model training, and evaluation.

3.2.1 Step-by-Step Process

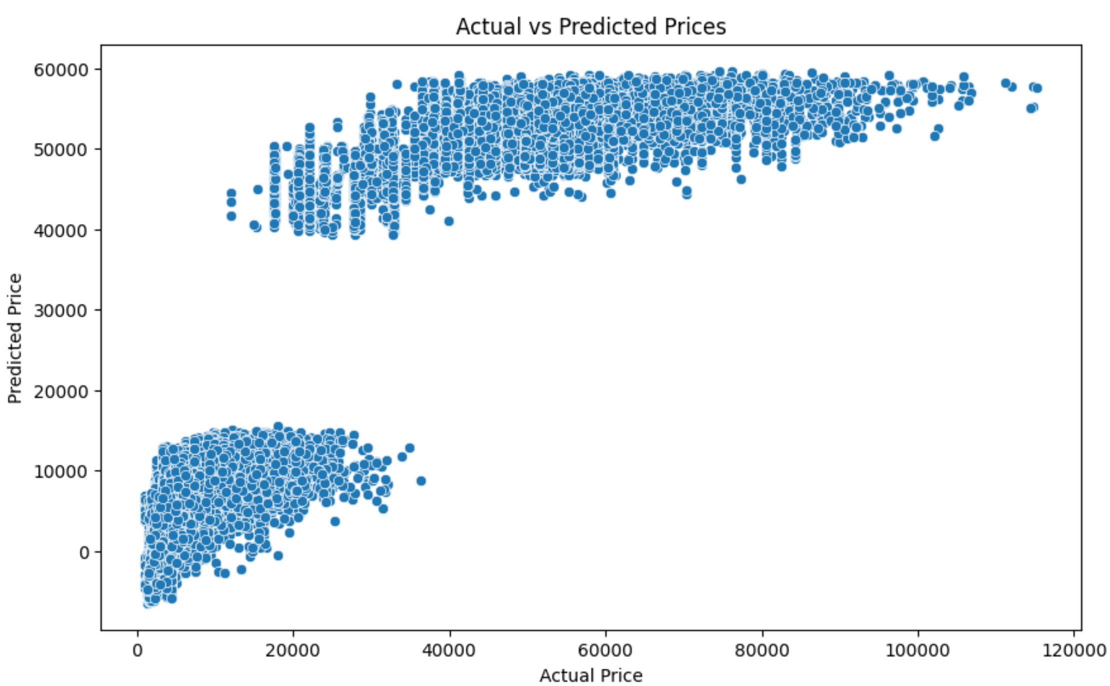
1. **Data Preprocessing**:
   - Read the dataset and encode categorical features using `OrdinalEncoder`.
   - Combine encoded categorical features with numerical features.
2. **Feature Selection**:
   - Separate features into linear and nonlinear categories.
   - Prepare data for Bayesian Linear Regression using PyStan.
3. **Bayesian Linear Regression (PGM)**:
   - Define and compile a Bayesian linear regression model.
   - Extract regression coefficients from the model and create a new feature for the DNN.
4. **Deep Neural Network (DNN)**:
   - Combine linear and nonlinear features along with the PGM output.
   - Standardize the data.
   - Build and train the DNN model.
5. **Model Evaluation**:
   - Evaluate the combined model on the test set.
   - Calculate metrics such as Mean Squared Error (MSE), Mean Absolute Error (MAE), R-squared (R^2), and Root Mean Squared Error (RMSE).
   - Visualize the results.

[ ]:
```python
# linear regression
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.preprocessing import OrdinalEncoder
import seaborn as sns
import pandas as pd
import numpy as np

clean_dataset = pd.read_csv('../datasets/Clean_Dataset.csv')
ordinalEncoder = OrdinalEncoder()
```

```
cate_features = ['airline', 'source_city', 'departure_time', 'stops',␣
 ↪'arrival_time', 'destination_city', 'class']
ordinalEncoder_features = ordinalEncoder.
 ↪fit_transform(clean_dataset[cate_features])
ordinalEncoder_features

# combine the ordinal encoded features with the numeric features
final_features = pd.concat([clean_dataset[['duration', 'days_left', 'price']],␣
 ↪pd.DataFrame(ordinalEncoder_features)], axis=1)
final_features

# feature and target variables
X = final_features.drop('price', axis = 1)
y = final_features['price']
print(X.shape, y.shape)
print(X[:5])
print(y[:5])

# split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)
# create a linear regression model
X_train.columns = X_train.columns.astype(str)
X_test.columns = X_test.columns.astype(str)


model = LinearRegression()
model.fit(X_train, y_train)

# make predictions
y_pred = model.predict(X_test)
# calculate the mean squared error
mse = mean_squared_error(y_test, y_pred)
print(f'Mean Squared Error: {mse:.2f}')

# calculate the mean absolute error
mae = mean_absolute_error(y_test, y_pred)
print(f'Mean Absolute Error: {mae:.2f}')

# calculate the R-squared value
r2 = r2_score(y_test, y_pred)
print(f'R-squared: {r2:.2f}')

# calculate the root mean squared error
rmse = np.sqrt(mse)
print(f'Root Mean Squared Error: {rmse:.2f}')
```

```python
# plot the predicted vs actual prices
plt.figure(figsize=(10, 6))
sns.scatterplot(x=y_test, y=y_pred)
plt.xlabel('Actual Price')
plt.ylabel('Predicted Price')
plt.title('Actual vs Predicted Prices')
plt.show()
```

```
(300153, 9) (300153,)
   duration  days_left    0    1    2    3    4    5    6
0      2.17          1  4.0  2.0  2.0  2.0  5.0  5.0  1.0
1      2.33          1  4.0  2.0  1.0  2.0  4.0  5.0  1.0
2      2.17          1  0.0  2.0  1.0  2.0  1.0  5.0  1.0
3      2.25          1  5.0  2.0  4.0  2.0  0.0  5.0  1.0
4      2.33          1  5.0  2.0  4.0  2.0  4.0  5.0  1.0
0    5953
1    5953
2    5956
3    5955
4    5955
Name: price, dtype: int64
Mean Squared Error: 49200540.29
Mean Absolute Error: 4624.99
R-squared: 0.90
Root Mean Squared Error: 7014.31
```



[ ]:

```
# Import necessary libraries
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OrdinalEncoder, StandardScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
import matplotlib.pyplot as plt
import seaborn as sns
import pystan
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout

# Read data
clean_dataset = pd.read_csv('../datasets/Clean_Dataset.csv')

# Encode categorical features using OrdinalEncoder
ordinalEncoder = OrdinalEncoder()
cate_features = ['airline', 'source_city', 'departure_time', 'stops',␣
 ↪'arrival_time', 'destination_city', 'class']
ordinalEncoder_features = ordinalEncoder.
 ↪fit_transform(clean_dataset[cate_features])

# Combine encoded categorical features with numerical features
encoded_df = pd.DataFrame(ordinalEncoder_features, columns=cate_features)
final_features = pd.concat([clean_dataset[['duration', 'days_left', 'price']],␣
 ↪encoded_df], axis=1)

# Ensure all column names are of string type
final_features.columns = final_features.columns.astype(str)

# Features and target variable
X = final_features.drop('price', axis=1)
y = final_features['price']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

# Linear and non-linear features
linear_features = ['duration', 'days_left']
nonlinear_features = [col for col in X.columns if col not in linear_features]

# Parameter inference using Bayesian Linear Regression Model (PGM)
stan_code = """
data {
  int<lower=0> N; // Data size
```

```
  int<lower=0> K; // Number of features
  matrix[N, K] X; // Feature matrix
  vector[N] y; // Target variable
}
parameters {
  vector[K] beta; // Regression coefficients
  real alpha; // Intercept
  real<lower=0> sigma; // Noise standard deviation
}
model {
  y ~ normal(X * beta + alpha, sigma); // Normal distribution
}
"""

# Compile model
stan_model = pystan.StanModel(model_code=stan_code)

# Prepare data
data = {'N': X_train[linear_features].shape[0], 'K': X_train[linear_features].
 ↪shape[1], 'X': X_train[linear_features], 'y': y_train}

# Sampling
fit = stan_model.sampling(data=data, iter=100, chains=4)

# Extract parameters from PGMs
params = fit.extract()
beta_mean = np.mean(params['beta'], axis=0)
alpha_mean = np.mean(params['alpha'])

# Add PGM outputs as features to DNN input
pgm_feature_train = (X_train[linear_features].dot(beta_mean) + alpha_mean).
 ↪values.reshape(-1, 1)
pgm_feature_test = (X_test[linear_features].dot(beta_mean) + alpha_mean).values.
 ↪reshape(-1, 1)

X_train_pgm = np.hstack([X_train, pgm_feature_train])
X_test_pgm = np.hstack([X_test, pgm_feature_test])

# Data standardization
scaler = StandardScaler()
X_train_nonlin = scaler.fit_transform(X_train[nonlinear_features])
X_test_nonlin = scaler.transform(X_test[nonlinear_features])

X_train_combined = np.hstack([X_train_nonlin, pgm_feature_train])
X_test_combined = np.hstack([X_test_nonlin, pgm_feature_test])

# Create DNN model
```

```python
def create_dnn_model(input_dim):
    model = Sequential()
    model.add(Dense(128, input_dim=input_dim, activation='relu'))
    model.add(Dropout(0.2))
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.2))
    model.add(Dense(32, activation='relu'))
    model.add(Dense(1))  # Output layer
    model.compile(optimizer='adam', loss='mse', metrics=['mae'])
    return model

# Initialize and train DNN model
dnn_model = create_dnn_model(X_train_combined.shape[1])
history = dnn_model.fit(X_train_combined, y_train, epochs=50, batch_size=32,
 ↪validation_split=0.2, verbose=1)

# Make predictions
y_pred = dnn_model.predict(X_test_combined).flatten()

# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
rmse = np.sqrt(mse)

print(f'Mean Squared Error: {mse:.2f}')
print(f'Mean Absolute Error: {mae:.2f}')
print(f'R-squared: {r2:.2f}')
print(f'Root Mean Squared Error: {rmse:.2f}')

# Plot actual vs predicted prices scatter plot
plt.figure(figsize=(10, 6))
sns.scatterplot(x=y_test, y=y_pred)
plt.xlabel('Actual Price')
plt.ylabel('Predicted Price')
plt.title('Actual vs Predicted Prices using Linear Regression + DNN with PGM
 ↪features')
plt.show()
```

```
2024-05-27 11:08:46.543637: I tensorflow/core/util/port.cc:113] oneDNN custom
operations are on. You may see slightly different numerical results due to
floating-point round-off errors from different computation orders. To turn them
off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2024-05-27 11:08:48.834568: I tensorflow/core/platform/cpu_feature_guard.cc:210]
This TensorFlow binary is optimized to use available CPU instructions in
performance-critical operations.
To enable the following instructions: AVX2 AVX_VNNI FMA, in other operations,
```

```
rebuild TensorFlow with the appropriate compiler flags.
2024-05-27 11:08:50.760149: W
tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warning: Could not
find TensorRT
INFO:pystan:COMPILING THE C++ CODE FOR MODEL
anon_model_6480e1d1f319fa39f3dae7fd542ccee9 NOW.
/home/siyan/.local/lib/python3.10/site-packages/Cython/Compiler/Main.py:381:
FutureWarning: Cython directive 'language_level' not set, using '3str' for now
(Py3). This has changed from earlier releases! File: /tmp/tmpqn66zojf/stanfit4an
on_model_6480e1d1f319fa39f3dae7fd542ccee9_5381600317920641068.pyx
  tree = Parsing.p_module(s, pxd, full_module_name)
In file included from /home/siyan/.local/lib/python3.10/site-
packages/numpy/core/include/numpy/ndarraytypes.h:1929,
                 from /home/siyan/.local/lib/python3.10/site-
packages/numpy/core/include/numpy/ndarrayobject.h:12,
                 from /home/siyan/.local/lib/python3.10/site-
packages/numpy/core/include/numpy/arrayobject.h:5,
                 from /tmp/tmpqn66zojf/stanfit4anon_model_6480e1d1f319fa39f3dae7
fd542ccee9_5381600317920641068.cpp:1280:
/home/siyan/.local/lib/python3.10/site-
packages/numpy/core/include/numpy/npy_1_7_deprecated_api.h:17:2: warning:
#warning "Using deprecated NumPy API, disable it with " "#define
NPY_NO_DEPRECATED_API NPY_1_7_API_VERSION" [-Wcpp]
   17 | #warning "Using deprecated NumPy API, disable it with " \
      |  ^~~~~~~
In file included from /home/siyan/.local/lib/python3.10/site-packages/pystan/sta
n/lib/stan_math/lib/boost_1.66.0/boost/mpl/aux_/na_assert.hpp:23,
                 from /home/siyan/.local/lib/python3.10/site-
packages/pystan/stan/lib/stan_math/lib/boost_1.66.0/boost/mpl/arg.hpp:25,
                 from /home/siyan/.local/lib/python3.10/site-packages/pystan/sta
n/lib/stan_math/lib/boost_1.66.0/boost/mpl/placeholders.hpp:24,
                 from /home/siyan/.local/lib/python3.10/site-
packages/pystan/stan/lib/stan_math/lib/boost_1.66.0/boost/mpl/apply.hpp:24,
                 from /home/siyan/.local/lib/python3.10/site-packages/pystan/sta
n/lib/stan_math/lib/boost_1.66.0/boost/mpl/aux_/iter_apply.hpp:17,
                 from /home/siyan/.local/lib/python3.10/site-packages/pystan/sta
n/lib/stan_math/lib/boost_1.66.0/boost/mpl/aux_/find_if_pred.hpp:14,
                 from /home/siyan/.local/lib/python3.10/site-
packages/pystan/stan/lib/stan_math/lib/boost_1.66.0/boost/mpl/find_if.hpp:17,
                 from /home/siyan/.local/lib/python3.10/site-
packages/pystan/stan/lib/stan_math/lib/boost_1.66.0/boost/mpl/find.hpp:17,
                 from /home/siyan/.local/lib/python3.10/site-packages/pystan/sta
n/lib/stan_math/lib/boost_1.66.0/boost/mpl/aux_/contains_impl.hpp:20,
                 from /home/siyan/.local/lib/python3.10/site-
packages/pystan/stan/lib/stan_math/lib/boost_1.66.0/boost/mpl/contains.hpp:20,
                 from /home/siyan/.local/lib/python3.10/site-packages/pystan/sta
n/lib/stan_math/lib/boost_1.66.0/boost/math/policies/policy.hpp:10,
                 from /home/siyan/.local/lib/python3.10/site-packages/pystan/sta
```

```
   86 |     MayLinearVectorize = bool(MightVectorize) && MayLinearize &&
DstHasDirectAccess
      |
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~
In file included from /home/siyan/.local/lib/python3.10/site-
packages/pystan/stan/lib/stan_math/lib/eigen_3.3.3/Eigen/Core:420,
                 from /home/siyan/.local/lib/python3.10/site-
packages/pystan/stan/lib/stan_math/lib/eigen_3.3.3/Eigen/Dense:1,
                 from /home/siyan/.local/lib/python3.10/site-
packages/pystan/stan/lib/stan_math/stan/math/prim/mat/fun/Eigen.hpp:4,
                 from /home/siyan/.local/lib/python3.10/site-
packages/pystan/stan/lib/stan_math/stan/math/rev/mat/fun/Eigen_NumTraits.hpp:4,
                 from /home/siyan/.local/lib/python3.10/site-
packages/pystan/stan/lib/stan_math/stan/math/rev/core/matrix_vari.hpp:4,
                 from /home/siyan/.local/lib/python3.10/site-
packages/pystan/stan/lib/stan_math/stan/math/rev/core.hpp:14,
                 from /home/siyan/.local/lib/python3.10/site-
packages/pystan/stan/lib/stan_math/stan/math/rev/mat.hpp:4,
                 from /home/siyan/.local/lib/python3.10/site-
packages/pystan/stan/src/stan/model/log_prob_grad.hpp:4,
                 from /home/siyan/.local/lib/python3.10/site-
packages/pystan/stan/src/stan/model/test_gradients.hpp:7,
                 from /home/siyan/.local/lib/python3.10/site-
packages/pystan/stan/src/stan/services/diagnose/diagnose.hpp:10,
                 from /home/siyan/.local/lib/python3.10/site-
packages/pystan/stan_fit.hpp:22,
                 from /tmp/tmpqn66zojf/stanfit4anon_model_6480e1d1f319fa39f3dae7
fd542ccee9_5381600317920641068.cpp:1287:
/home/siyan/.local/lib/python3.10/site-packages/pystan/stan/lib/stan_math/lib/ei
gen_3.3.3/Eigen/src/Core/AssignEvaluator.h:90:55: warning: enum constant in
boolean context [-Wint-in-bool-context]
   90 |     MaySliceVectorize  = bool(MightVectorize) &&
bool(DstHasDirectAccess)
      |                                                       ^~~~~~~~~~~~~~~~~~
In file included from /home/siyan/.local/lib/python3.10/site-
packages/pystan/stan/lib/stan_math/lib/eigen_3.3.3/Eigen/Core:420,
                 from /home/siyan/.local/lib/python3.10/site-
packages/pystan/stan/lib/stan_math/lib/eigen_3.3.3/Eigen/Dense:1,
                 from /home/siyan/.local/lib/python3.10/site-
packages/pystan/stan/lib/stan_math/stan/math/prim/mat/fun/Eigen.hpp:4,
                 from /home/siyan/.local/lib/python3.10/site-
packages/pystan/stan/lib/stan_math/stan/math/rev/mat/fun/Eigen_NumTraits.hpp:4,
                 from /home/siyan/.local/lib/python3.10/site-
packages/pystan/stan/lib/stan_math/stan/math/rev/core/matrix_vari.hpp:4,
                 from /home/siyan/.local/lib/python3.10/site-
packages/pystan/stan/lib/stan_math/stan/math/rev/core.hpp:14,
                 from /home/siyan/.local/lib/python3.10/site-
packages/pystan/stan/lib/stan_math/stan/math/rev/mat.hpp:4,
```

```
boolean context [-Wint-in-bool-context]
   90 |     MaySliceVectorize  = bool(MightVectorize) &&
bool(DstHasDirectAccess)
      |                                                      ^~~~~~~~~~~~~~~~~~


Gradient evaluation took 0.029532 seconds
1000 transitions using 10 leapfrog steps per transition would take 295.32
seconds.
Adjust your expectations accordingly!


WARNING: There aren't enough warmup iterations to fit the
         three stages of adaptation as currently configured.
         Reducing each adaptation stage to 15%/75%/10% of
         the given number of warmup iterations:
           init_buffer = 7
           adapt_window = 38
           term_buffer = 5


Gradient evaluation took 0.030637 seconds
1000 transitions using 10 leapfrog steps per transition would take 306.37
seconds.
Adjust your expectations accordingly!


WARNING: There aren't enough warmup iterations to fit the
         three stages of adaptation as currently configured.
         Reducing each adaptation stage to 15%/75%/10% of
         the given number of warmup iterations:
           init_buffer = 7
           adapt_window = 38
           term_buffer = 5

Iteration:  1 / 100 [  1%]  (Warmup)
Iteration: 10 / 100 [ 10%]  (Warmup)
Iteration:  1 / 100 [  1%]  (Warmup)
Iteration: 10 / 100 [ 10%]  (Warmup)
Iteration: 20 / 100 [ 20%]  (Warmup)
Iteration: 30 / 100 [ 30%]  (Warmup)
Iteration: 20 / 100 [ 20%]  (Warmup)
Iteration: 30 / 100 [ 30%]  (Warmup)
Iteration: 40 / 100 [ 40%]  (Warmup)
Iteration: 40 / 100 [ 40%]  (Warmup)
Iteration: 50 / 100 [ 50%]  (Warmup)
Iteration: 51 / 100 [ 51%]  (Sampling)
Iteration: 50 / 100 [ 50%]  (Warmup)
```

```
Iteration: 51 / 100 [ 51%]  (Sampling)
Iteration: 60 / 100 [ 60%]  (Sampling)
Iteration: 60 / 100 [ 60%]  (Sampling)
Iteration: 70 / 100 [ 70%]  (Sampling)
Iteration: 70 / 100 [ 70%]  (Sampling)
Iteration: 80 / 100 [ 80%]  (Sampling)
Iteration: 80 / 100 [ 80%]  (Sampling)
Iteration: 90 / 100 [ 90%]  (Sampling)
Iteration: 90 / 100 [ 90%]  (Sampling)
Iteration: 100 / 100 [100%]  (Sampling)
Iteration: 100 / 100 [100%]  (Sampling)

 Elapsed Time: 20.446 seconds (Warm-up)
               54.8753 seconds (Sampling)
               75.3213 seconds (Total)


Gradient evaluation took 0.01197 seconds
1000 transitions using 10 leapfrog steps per transition would take 119.7
seconds.
Adjust your expectations accordingly!


WARNING: There aren't enough warmup iterations to fit the
         three stages of adaptation as currently configured.
         Reducing each adaptation stage to 15%/75%/10% of
         the given number of warmup iterations:
           init_buffer = 7
           adapt_window = 38
           term_buffer = 5

Iteration:  1 / 100 [  1%]  (Warmup)
Iteration: 10 / 100 [ 10%]  (Warmup)

 Elapsed Time: 13.7674 seconds (Warm-up)
               65.9777 seconds (Sampling)
               79.7452 seconds (Total)


Gradient evaluation took 0.012487 seconds
1000 transitions using 10 leapfrog steps per transition would take 124.87
seconds.
Adjust your expectations accordingly!


WARNING: There aren't enough warmup iterations to fit the
         three stages of adaptation as currently configured.
         Reducing each adaptation stage to 15%/75%/10% of
```

```
         the given number of warmup iterations:
           init_buffer = 7
           adapt_window = 38
           term_buffer = 5

Iteration:  1 / 100 [  1%]  (Warmup)
Iteration: 10 / 100 [ 10%]  (Warmup)
Iteration: 20 / 100 [ 20%]  (Warmup)
Iteration: 30 / 100 [ 30%]  (Warmup)
Iteration: 40 / 100 [ 40%]  (Warmup)
Iteration: 50 / 100 [ 50%]  (Warmup)
Iteration: 51 / 100 [ 51%]  (Sampling)
Iteration: 60 / 100 [ 60%]  (Sampling)
Iteration: 70 / 100 [ 70%]  (Sampling)
Iteration: 80 / 100 [ 80%]  (Sampling)
Iteration: 90 / 100 [ 90%]  (Sampling)
Iteration: 20 / 100 [ 20%]  (Warmup)
Iteration: 30 / 100 [ 30%]  (Warmup)
Iteration: 100 / 100 [100%]  (Sampling)

 Elapsed Time: 11.1007 seconds (Warm-up)
               35.7596 seconds (Sampling)
               46.8604 seconds (Total)

Iteration: 40 / 100 [ 40%]  (Warmup)
Iteration: 50 / 100 [ 50%]  (Warmup)
Iteration: 51 / 100 [ 51%]  (Sampling)
Iteration: 60 / 100 [ 60%]  (Sampling)
Iteration: 70 / 100 [ 70%]  (Sampling)
Iteration: 80 / 100 [ 80%]  (Sampling)
Iteration: 90 / 100 [ 90%]  (Sampling)
Iteration: 100 / 100 [100%]  (Sampling)

WARNING:pystan:Rhat for parameter beta[1] is 1.633223718279847!
WARNING:pystan:Rhat for parameter beta[2] is 1.7384104126035809!
WARNING:pystan:Rhat for parameter alpha is 5.9975803845317905!
WARNING:pystan:Rhat for parameter sigma is 1.4681878616509982!
WARNING:pystan:Rhat for parameter lp__ is 1.6689490236452358!
WARNING:pystan:Rhat above 1.1 or below 0.9 indicates that the chains very likely
have not mixed
WARNING:pystan:1 of 200 iterations saturated the maximum tree depth of 10 (0.5%)
WARNING:pystan:Run again with max_treedepth larger than 10 to avoid saturation
WARNING:pystan:Chain 1: E-BFMI = 0.0247110640184885
WARNING:pystan:Chain 2: E-BFMI = 0.09329297206902808
WARNING:pystan:Chain 3: E-BFMI = 0.020096063442197255
WARNING:pystan:Chain 4: E-BFMI = 0.02551020285389375
WARNING:pystan:E-BFMI below 0.2 indicates you may need to reparameterize your
model
```

```
 Elapsed Time: 56.1792 seconds (Warm-up)
               28.2692 seconds (Sampling)
               84.4484 seconds (Total)
```

```
/home/siyan/.local/lib/python3.10/site-
packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential models,
prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

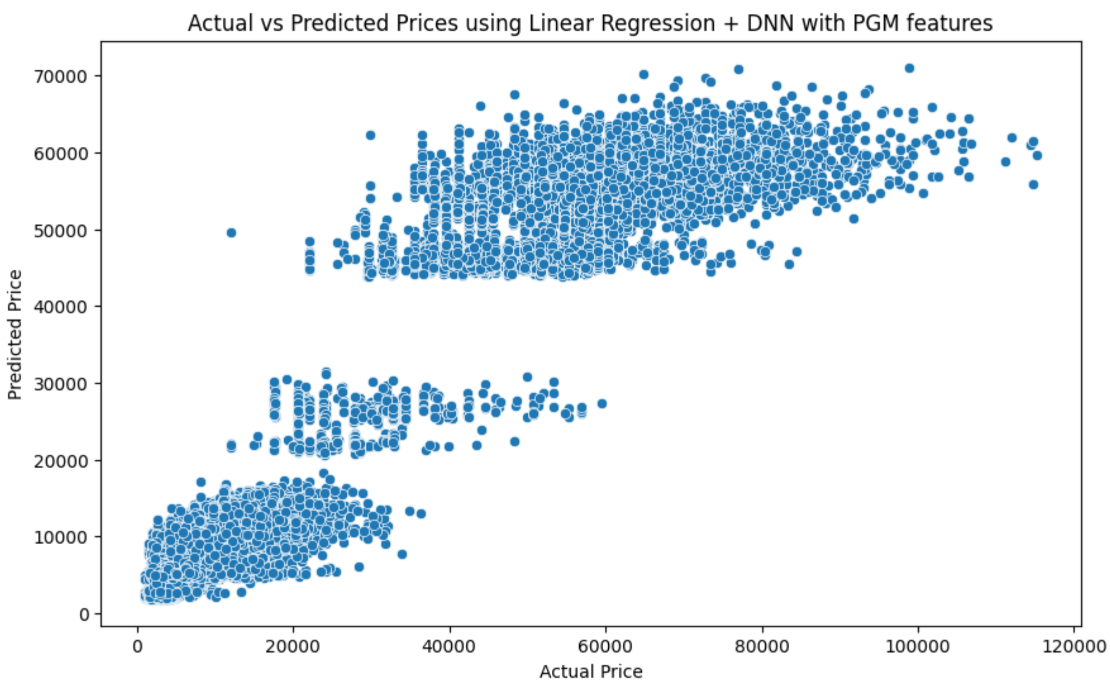
```
Epoch 1/50
6004/6004           19s 3ms/step -
loss: 365749248.0000 - mae: 13073.1260 - val_loss: 52561604.0000 - val_mae:
4659.9834
Epoch 2/50
6004/6004           18s 2ms/step -
loss: 48502496.0000 - mae: 4465.6455 - val_loss: 43136812.0000 - val_mae:
4008.1938
Epoch 3/50
6004/6004           28s 3ms/step -
loss: 47018840.0000 - mae: 4327.7871 - val_loss: 42804552.0000 - val_mae:
4434.1704
Epoch 4/50
6004/6004           40s 3ms/step -
loss: 46103360.0000 - mae: 4286.6113 - val_loss: 40681664.0000 - val_mae:
4338.7168
Epoch 5/50
6004/6004           20s 3ms/step -
loss: 46140688.0000 - mae: 4263.8213 - val_loss: 36633464.0000 - val_mae:
3816.5642
Epoch 6/50
6004/6004           20s 3ms/step -
loss: 45286808.0000 - mae: 4217.4819 - val_loss: 37530192.0000 - val_mae:
3965.4048
Epoch 7/50
6004/6004           20s 3ms/step -
loss: 44790028.0000 - mae: 4192.9927 - val_loss: 36173936.0000 - val_mae:
3695.9714
Epoch 8/50
6004/6004           20s 3ms/step -
loss: 44476088.0000 - mae: 4170.0762 - val_loss: 41258668.0000 - val_mae:
3936.4614
Epoch 9/50
6004/6004           22s 3ms/step -
loss: 44493024.0000 - mae: 4163.6597 - val_loss: 34824404.0000 - val_mae:
3655.3069
Epoch 10/50
```

```
6004/6004              20s 3ms/step -
loss: 43563496.0000 - mae: 4128.2329 - val_loss: 35466656.0000 - val_mae:
3677.0403
Epoch 11/50
6004/6004              21s 4ms/step -
loss: 43076656.0000 - mae: 4107.5410 - val_loss: 37137636.0000 - val_mae:
3817.7642
Epoch 12/50
6004/6004              20s 3ms/step -
loss: 43364344.0000 - mae: 4126.5737 - val_loss: 33928752.0000 - val_mae:
3636.9260
Epoch 13/50
6004/6004              20s 3ms/step -
loss: 43562420.0000 - mae: 4128.2603 - val_loss: 35176120.0000 - val_mae:
3665.1921
Epoch 14/50
6004/6004              20s 3ms/step -
loss: 43327664.0000 - mae: 4102.0356 - val_loss: 34537040.0000 - val_mae:
3620.0166
Epoch 15/50
6004/6004              21s 3ms/step -
loss: 42567680.0000 - mae: 4072.3235 - val_loss: 33516616.0000 - val_mae:
3611.1711
Epoch 16/50
6004/6004              40s 3ms/step -
loss: 42845884.0000 - mae: 4093.1050 - val_loss: 36151308.0000 - val_mae:
3729.1575
Epoch 17/50
6004/6004              21s 3ms/step -
loss: 42418784.0000 - mae: 4070.7651 - val_loss: 34484216.0000 - val_mae:
3721.9666
Epoch 18/50
6004/6004              21s 4ms/step -
loss: 41963660.0000 - mae: 4046.4438 - val_loss: 33451078.0000 - val_mae:
3703.1013
Epoch 19/50
6004/6004              21s 4ms/step -
loss: 42130356.0000 - mae: 4038.9106 - val_loss: 32679360.0000 - val_mae:
3541.6843
Epoch 20/50
6004/6004              22s 4ms/step -
loss: 41921048.0000 - mae: 4034.4951 - val_loss: 32624378.0000 - val_mae:
3525.3772
Epoch 21/50
6004/6004              21s 4ms/step -
loss: 41804748.0000 - mae: 4026.5105 - val_loss: 33573036.0000 - val_mae:
3623.1619
Epoch 22/50
```

```
6004/6004              20s 3ms/step -
loss: 41868836.0000 - mae: 4035.0906 - val_loss: 32410284.0000 - val_mae:
3534.3606
Epoch 23/50
6004/6004              19s 3ms/step -
loss: 42031732.0000 - mae: 4036.7600 - val_loss: 32610460.0000 - val_mae:
3562.6765
Epoch 24/50
6004/6004              19s 3ms/step -
loss: 41874900.0000 - mae: 4041.3655 - val_loss: 35915540.0000 - val_mae:
3704.5342
Epoch 25/50
6004/6004              22s 3ms/step -
loss: 41292648.0000 - mae: 4007.4788 - val_loss: 35236560.0000 - val_mae:
3641.9514
Epoch 26/50
6004/6004              20s 3ms/step -
loss: 41098484.0000 - mae: 3993.3037 - val_loss: 32199342.0000 - val_mae:
3539.4221
Epoch 27/50
6004/6004              20s 3ms/step -
loss: 41275864.0000 - mae: 3998.2317 - val_loss: 32453798.0000 - val_mae:
3550.1926
Epoch 28/50
6004/6004              19s 3ms/step -
loss: 41065920.0000 - mae: 3997.3621 - val_loss: 32295672.0000 - val_mae:
3557.3032
Epoch 29/50
6004/6004              20s 3ms/step -
loss: 41270964.0000 - mae: 4000.6711 - val_loss: 31459018.0000 - val_mae:
3476.8877
Epoch 30/50
6004/6004              20s 3ms/step -
loss: 41230132.0000 - mae: 3989.7463 - val_loss: 31201610.0000 - val_mae:
3454.5142
Epoch 31/50
6004/6004              18s 3ms/step -
loss: 40767856.0000 - mae: 3958.4656 - val_loss: 34266828.0000 - val_mae:
3612.9324
Epoch 32/50
6004/6004              18s 3ms/step -
loss: 40645892.0000 - mae: 3960.6453 - val_loss: 31218984.0000 - val_mae:
3439.8977
Epoch 33/50
6004/6004              19s 3ms/step -
loss: 39884524.0000 - mae: 3924.6948 - val_loss: 32929342.0000 - val_mae:
3573.8220
Epoch 34/50
```

```
6004/6004             20s 3ms/step -
loss: 40822744.0000 - mae: 3973.4580 - val_loss: 32311218.0000 - val_mae:
3594.2759
Epoch 35/50
6004/6004             18s 3ms/step -
loss: 40547764.0000 - mae: 3965.3333 - val_loss: 31220660.0000 - val_mae:
3442.4961
Epoch 36/50
6004/6004             19s 3ms/step -
loss: 40074884.0000 - mae: 3920.0349 - val_loss: 32865538.0000 - val_mae:
3567.9084
Epoch 37/50
6004/6004             19s 3ms/step -
loss: 39704328.0000 - mae: 3913.0068 - val_loss: 33393306.0000 - val_mae:
3545.7219
Epoch 38/50
6004/6004             22s 3ms/step -
loss: 39411944.0000 - mae: 3900.3411 - val_loss: 31065084.0000 - val_mae:
3453.7236
Epoch 39/50
6004/6004             20s 3ms/step -
loss: 39551716.0000 - mae: 3896.0144 - val_loss: 31728846.0000 - val_mae:
3469.3213
Epoch 40/50
6004/6004             20s 3ms/step -
loss: 39196020.0000 - mae: 3875.5813 - val_loss: 30409348.0000 - val_mae:
3385.5105
Epoch 41/50
6004/6004             20s 3ms/step -
loss: 39039460.0000 - mae: 3868.4290 - val_loss: 30888992.0000 - val_mae:
3435.6819
Epoch 42/50
6004/6004             19s 3ms/step -
loss: 39330680.0000 - mae: 3881.1108 - val_loss: 30887360.0000 - val_mae:
3399.8345
Epoch 43/50
6004/6004             20s 3ms/step -
loss: 39234012.0000 - mae: 3873.6770 - val_loss: 31770126.0000 - val_mae:
3457.9263
Epoch 44/50
6004/6004             19s 3ms/step -
loss: 38427044.0000 - mae: 3832.0393 - val_loss: 31137780.0000 - val_mae:
3429.9253
Epoch 45/50
6004/6004             20s 3ms/step -
loss: 39076240.0000 - mae: 3855.5298 - val_loss: 30373172.0000 - val_mae:
3391.1411
Epoch 46/50
```

```
6004/6004                19s 3ms/step -
loss: 38896476.0000 - mae: 3848.7749 - val_loss: 30274802.0000 - val_mae:
3389.5149
Epoch 47/50
6004/6004                19s 3ms/step -
loss: 37919836.0000 - mae: 3806.2861 - val_loss: 29626476.0000 - val_mae:
3376.6514
Epoch 48/50
6004/6004                22s 3ms/step -
loss: 37663664.0000 - mae: 3793.5464 - val_loss: 29961246.0000 - val_mae:
3369.3389
Epoch 49/50
6004/6004                20s 3ms/step -
loss: 37400512.0000 - mae: 3791.6680 - val_loss: 30967924.0000 - val_mae:
3429.1924
Epoch 50/50
6004/6004                18s 3ms/step -
loss: 37651120.0000 - mae: 3787.2373 - val_loss: 31407144.0000 - val_mae:
3462.3911
1876/1876                3s 2ms/step
Mean Squared Error: 31611055.71
Mean Absolute Error: 3452.47
R-squared: 0.94
Root Mean Squared Error: 5622.37
```



3.3 PCA Analysis on Cleaned Dataset

In this section, we perform Principal Component Analysis (PCA) on a portion of the `clean_dataset_updated` dataset. The steps include sampling the data, preprocessing features, running PCA, and visualizing the results.

3.3.1 Step-by-Step Process

1. **Print Dataset Columns**:
   - Confirm the columns present in the dataset.
2. **Sample the Data**:
   - Select 10% of the data for testing to make the computations manageable.
3. **Select Features for PCA**:
   - Identify the features to be used for PCA: `airline`, `source_city`, `destination_city`, `class`, `duration`, `days_left`, and `distance`.
   - Ensure the target variable `price` is included.
4. **Check for Missing Features**:
   - Ensure all selected features and the target variable are present in the sampled dataset.
5. **Preprocess Data**:
   - Use `ColumnTransformer` to standardize numerical features (`duration`, `days_left`, `distance`) and one-hot encode categorical features (`airline`, `source_city`, `destination_city`, `class`).
   - Extract features and target variable from the dataset.
   - Fit and transform the features using the preprocessor.
6. **Standardize the Preprocessed Data**:
   - Standardize the data to ensure it has a mean of 0 and standard deviation of 1.
7. **Run PCA**:
   - Create a PCA instance.
   - Fit and transform the standardized data with PCA.
8. **Visualize Results**:
   - Plot the Scree Plot to show the explained variance ratio of each principal component.
   - Plot the cumulative explained variance ratio to determine the number of principal components required to explain a significant portion of the variance.

[ ]:
```
import pandas as pd
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.decomposition import PCA
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
import matplotlib.pyplot as plt
```

[ ]:
```
# Print the column names of the dataset to confirm
print("Columns in the dataset:", clean_dataset_updated.columns)
```

```
Columns in the dataset: Index(['Unnamed: 0', 'airline', 'flight', 'source_city',
'departure_time',
       'stops', 'arrival_time', 'destination_city', 'class', 'duration',
       'days_left', 'price', 'combined_date', 'distance', 'day_of_week',
       'week_of_year', 'month', 'is_holiday', 'route_class', 'price_bin',
```

```
       'days_left_bin'],
      dtype='object')
```

[ ]:
```
# Select a portion of the data for testing
sampled_data = clean_dataset_updated.sample(frac=0.1, random_state=42)  #␣
 ↪Select 10% of the data
```

[ ]:
```
# Select the features for PCA
selected_features = ['airline', 'source_city', 'destination_city', 'class',␣
 ↪'duration', 'days_left', 'distance']
target = 'price'
```

[ ]:
```
# Ensure all required columns are present
for feature in selected_features + [target]:
    if feature not in sampled_data.columns:
        raise ValueError(f"Missing feature: {feature}")
```

[ ]:
```
# One-hot encode categorical variables
categorical_features = ['airline', 'source_city', 'destination_city', 'class']
numerical_features = ['duration', 'days_left', 'distance']

# Create a preprocessor: standardize numerical features and one-hot encode␣
 ↪categorical features
preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numerical_features),
        ('cat', OneHotEncoder(sparse_output=False), categorical_features)
    ])
```

[ ]:
```
# extract the features and target variable
X = sampled_data[selected_features]
y = sampled_data[target]
```

[ ]:
```
# Data preprocessing
X_preprocessed = preprocessor.fit_transform(X)

# Check the shape of the preprocessed data
print("Shape of preprocessed data:", X_preprocessed.shape)
```

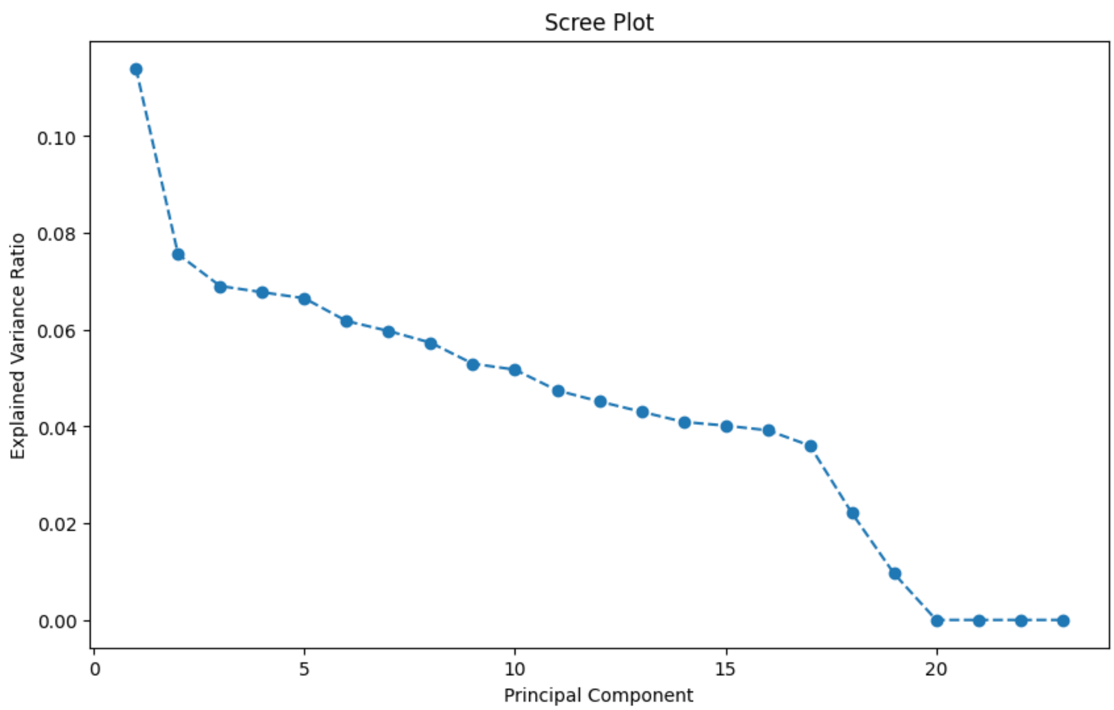
```
Shape of preprocessed data: (30015, 23)
```

[ ]:
```
# Standardize the data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_preprocessed)
```

[ ]:
```
# Create a PCA instance
pca = PCA(n_components=None)
```

```
# Run PCA
X_pca = pca.fit_transform(X_scaled)
```

[ ]:
```
# Plot the Scree plot
explained_variance = pca.explained_variance_ratio_
plt.figure(figsize=(10, 6))
plt.plot(range(1, len(explained_variance) + 1), explained_variance, marker='o',␣
↪linestyle='--')
plt.xlabel('Principal Component')
plt.ylabel('Explained Variance Ratio')
plt.title('Scree Plot')
plt.show()
```



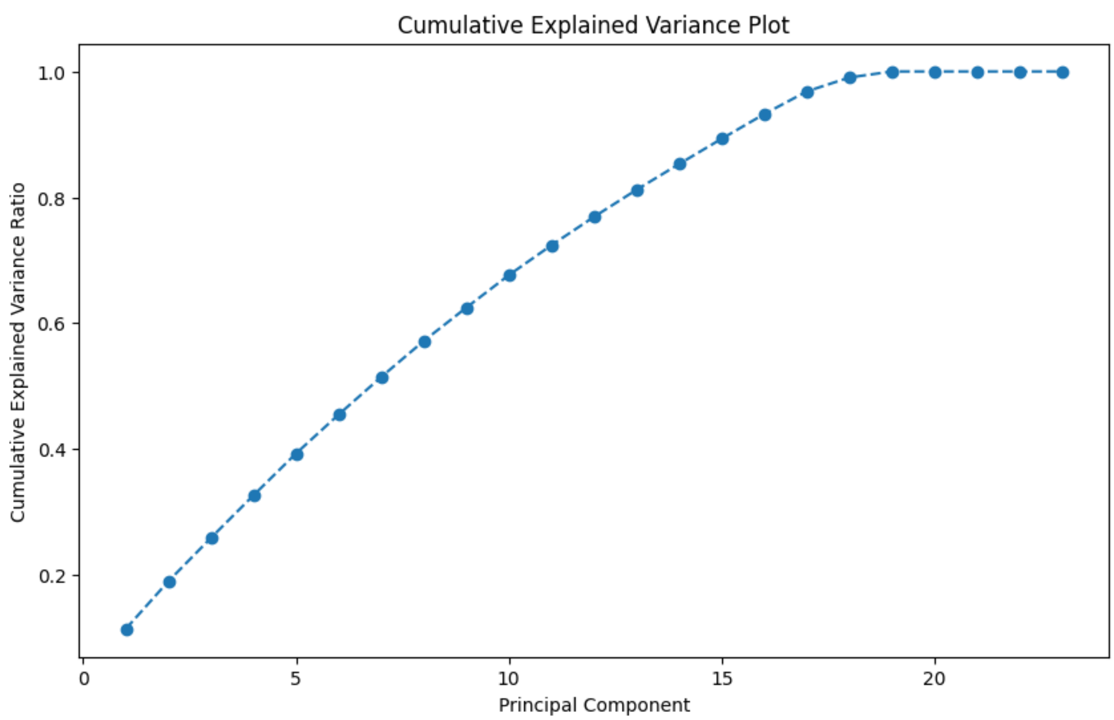
[ ]:
```
# Output the explained variance ratio of each principal component
print(f'Explained variance by each principal component: {explained_variance}')
print(f'Total explained variance: {sum(explained_variance)}')
```

```
Explained variance by each principal component: [1.14035211e-01 7.56039336e-02
6.90165573e-02 6.77327648e-02
 6.64899571e-02 6.17862214e-02 5.97153844e-02 5.72980050e-02
 5.29563086e-02 5.17358682e-02 4.74469999e-02 4.51641078e-02
 4.30332257e-02 4.09375639e-02 4.01638791e-02 3.92007037e-02
 3.60336976e-02 2.20789518e-02 9.57065851e-03 1.21044185e-31
 7.22936217e-33 3.90294770e-33 2.13944202e-33]
```

```
Total explained variance: 1.0000000000000002
```

[ ]:
```python
# Calculate cumulative explained variance ratio
cumulative_variance = explained_variance.cumsum()
print(f'Cumulative explained variance: {cumulative_variance}')


# Plot the cumulative explained variance ratio
plt.figure(figsize=(10, 6))
plt.plot(range(1, len(cumulative_variance) + 1), cumulative_variance,␣
 ↪marker='o', linestyle='--')
plt.xlabel('Principal Component')
plt.ylabel('Cumulative Explained Variance Ratio')
plt.title('Cumulative Explained Variance Plot')
plt.show()
```

```
Cumulative explained variance: [0.11403521 0.18963915 0.2586557  0.32638847
0.39287842 0.45466465
 0.51438003 0.57167804 0.62463434 0.67637021 0.72381721 0.76898132
 0.81201455 0.85295211 0.89311599 0.93231669 0.96835039 0.99042934
 1.         1.         1.         1.         1.        ]
```



3.4 Model Selection and Feature Importance Analysis

This section describes the steps taken to select the best regression model for predicting flight prices from the `clean_dataset_updated` dataset. The analysis includes data preprocessing, model training, hyperparameter tuning, and evaluation. Additionally, we visualize the feature importance and analyze the correlation matrix of the features.

3.4.1 Step-by-Step Process

1. **Print Dataset Columns**:
   - Confirm the columns present in the dataset.
2. **Select Features and Target Variable**:
   - Identify the features to be used for the analysis: `airline`, `source_city`, `destination_city`, `class`, `duration`, `days_left`, and `distance`.
   - The target variable is `price`.
3. **Preprocess Data**:
   - Standardize numerical features (`duration`, `days_left`, `distance`) and one-hot encode categorical features (`airline`, `source_city`, `destination_city`, `class`) using `ColumnTransformer`.
4. **Split the Data**:
   - Split the dataset into training and test sets.
   - Further split the training set to create a smaller sample for initial experimentation.
5. **Define Models and Hyperparameters**:
   - Define four models: Linear Regression, Ridge Regression, Decision Tree Regressor, and Random Forest Regressor.
   - Specify hyperparameters for GridSearchCV.
6. **Perform Grid Search**:
   - Use GridSearchCV to find the best model based on the negative mean squared error (MSE).
   - Select the best model and hyperparameters based on the grid search results.
7. **Evaluate the Best Model**:
   - Evaluate the best model on the test set.
   - Calculate the mean squared error (MSE) of the predictions.
   - Visualize the actual vs predicted prices.
8. **Analyze Feature Importance**:
   - Extract and plot feature importances from the best model.
   - Visualize the correlation matrix of the features.

[ ]:
```
import pandas as pd
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.linear_model import LinearRegression, Ridge
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
```

[ ]:
```
# Assuming clean_dataset_updated has been loaded as a DataFrame
# Confirm the columns included in the dataset
print("Columns in the dataset:", clean_dataset_updated.columns)

# Select the features and target variable for analysis
features = ['airline', 'source_city', 'destination_city', 'class', 'duration',␣
↪'days_left', 'distance']
target = 'price'
```

```
Columns in the dataset: Index(['Unnamed: 0', 'airline', 'flight', 'source_city',
'departure_time',
       'stops', 'arrival_time', 'destination_city', 'class', 'duration',
       'days_left', 'price', 'combined_date', 'distance', 'day_of_week',
       'week_of_year', 'month', 'is_holiday', 'route_class', 'price_bin',
       'days_left_bin'],
      dtype='object')
```

[ ]:
```
# One-hot encode categorical variables
categorical_features = ['airline', 'source_city', 'destination_city', 'class']
numerical_features = ['duration', 'days_left', 'distance']

# Create a preprocessor: standardize numerical features and one-hot encode␣
↪categorical features
preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numerical_features),
        ('cat', OneHotEncoder(sparse_output=False), categorical_features)
    ])

# Extract features and target
X = clean_dataset_updated[features]
y = clean_dataset_updated[target]
```

[ ]:
```
# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)

# use a small sample of the data for initial experimentation
X_train_sample, _, y_train_sample, _ = train_test_split(X_train, y_train,␣
↪test_size=0.5, random_state=42)
```

[ ]:
```
# Define the models and hyperparameters to try
models = [
    ('Linear Regression', LinearRegression(), {}),
    ('Ridge Regression', Ridge(), {'regressor__alpha': [0.1, 1.0]}),
    ('Decision Tree', DecisionTreeRegressor(), {'regressor__max_depth': [10,␣
↪20]}),
```

```
    ('Random Forest', RandomForestRegressor(), {'regressor__n_estimators': [50,␣
↪100], 'regressor__max_depth': [10, 20]})
]
```

[ ]:
```
# Perform grid search to find the best model
best_model = None
best_score = float('inf')
best_params = None

for name, model, params in models:
    pipeline = Pipeline(steps=[
        ('preprocessor', preprocessor),
        ('regressor', model)
    ])
    grid_search = GridSearchCV(pipeline, param_grid=params, cv=3,␣
↪scoring='neg_mean_squared_error', n_jobs=-1)
    grid_search.fit(X_train_sample, y_train_sample)
    score = -grid_search.best_score_
    if score < best_score:
        best_score = score
        best_model = grid_search.best_estimator_
        best_params = grid_search.best_params_

# Print the best model and parameters
print(f"Best Model: {best_model}")
print(f"Best Parameters: {best_params}")
```

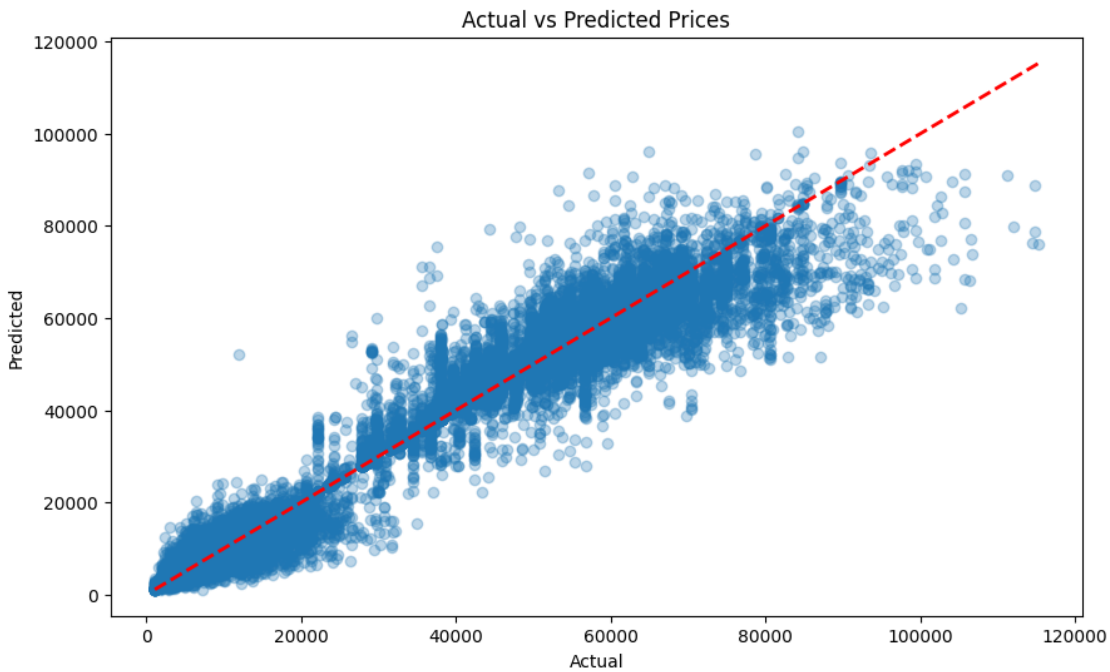
```
Best Model: Pipeline(steps=[('preprocessor',
                 ColumnTransformer(transformers=[('num', StandardScaler(),
                                                  ['duration', 'days_left',
                                                   'distance']),
                                                 ('cat',
OneHotEncoder(sparse_output=False),
                                                  ['airline', 'source_city',
                                                   'destination_city',
                                                   'class'])])),
                ('regressor', RandomForestRegressor(max_depth=20))])
Best Parameters: {'regressor__max_depth': 20, 'regressor__n_estimators': 100}
```

[ ]:
```
# Evaluate the best model on the test set
y_pred = best_model.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error on Test Set: {mse}")
```

```
Mean Squared Error on Test Set: 10959913.231577694
```

[ ]:
```
# visualize the actual vs predicted prices
plt.figure(figsize=(10, 6))
```

```
plt.scatter(y_test, y_pred, alpha=0.3)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], '--r',␣
↪linewidth=2)
plt.xlabel('Actual')
plt.ylabel('Predicted')
plt.title('Actual vs Predicted Prices')
plt.show()
```



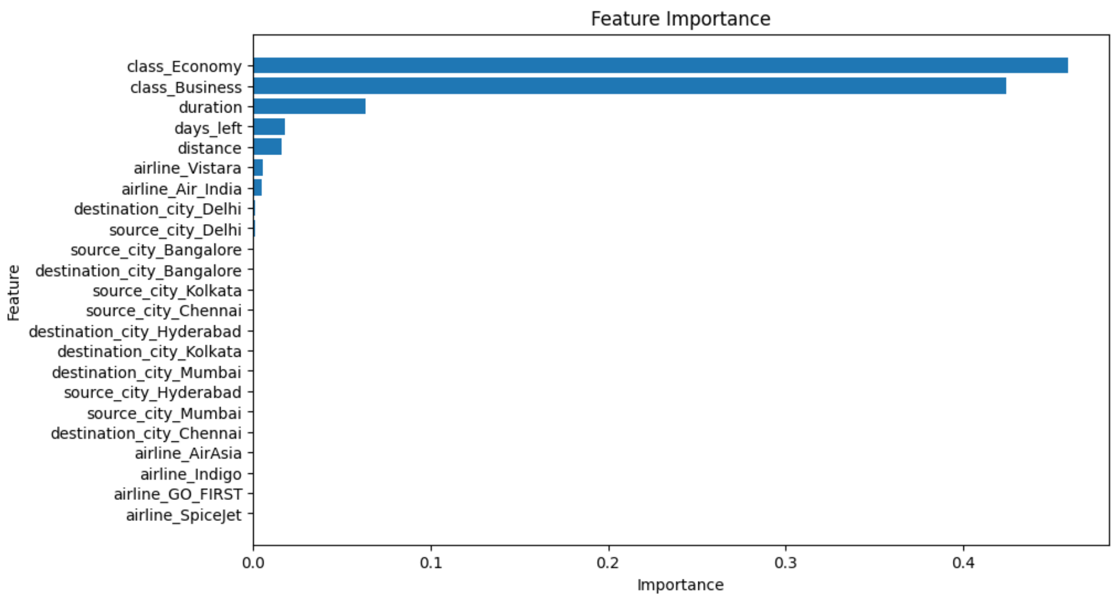
[ ]:
```
# Influencing factors, correlation analysis
# Extract regression coefficients


# extract the regressor from the best model
regressor = best_model.named_steps['regressor']


# extract feature importances
importances = regressor.feature_importances_


# get the encoded feature names
encoded_features = best_model.named_steps['preprocessor'].transformers_[1][1].
↪get_feature_names_out(categorical_features)
all_features = np.concatenate([numerical_features, encoded_features])
```

```python
# Create a DataFrame for feature importances
importances_df = pd.DataFrame({'Feature': all_features, 'Importance':␣
↪importances})
importances_df = importances_df.sort_values(by='Importance', ascending=False)


# output the feature importances DataFrame
print(importances_df)

# Evaluate the best model on the test set
y_pred = best_model.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error on Test Set: {mse}")
```

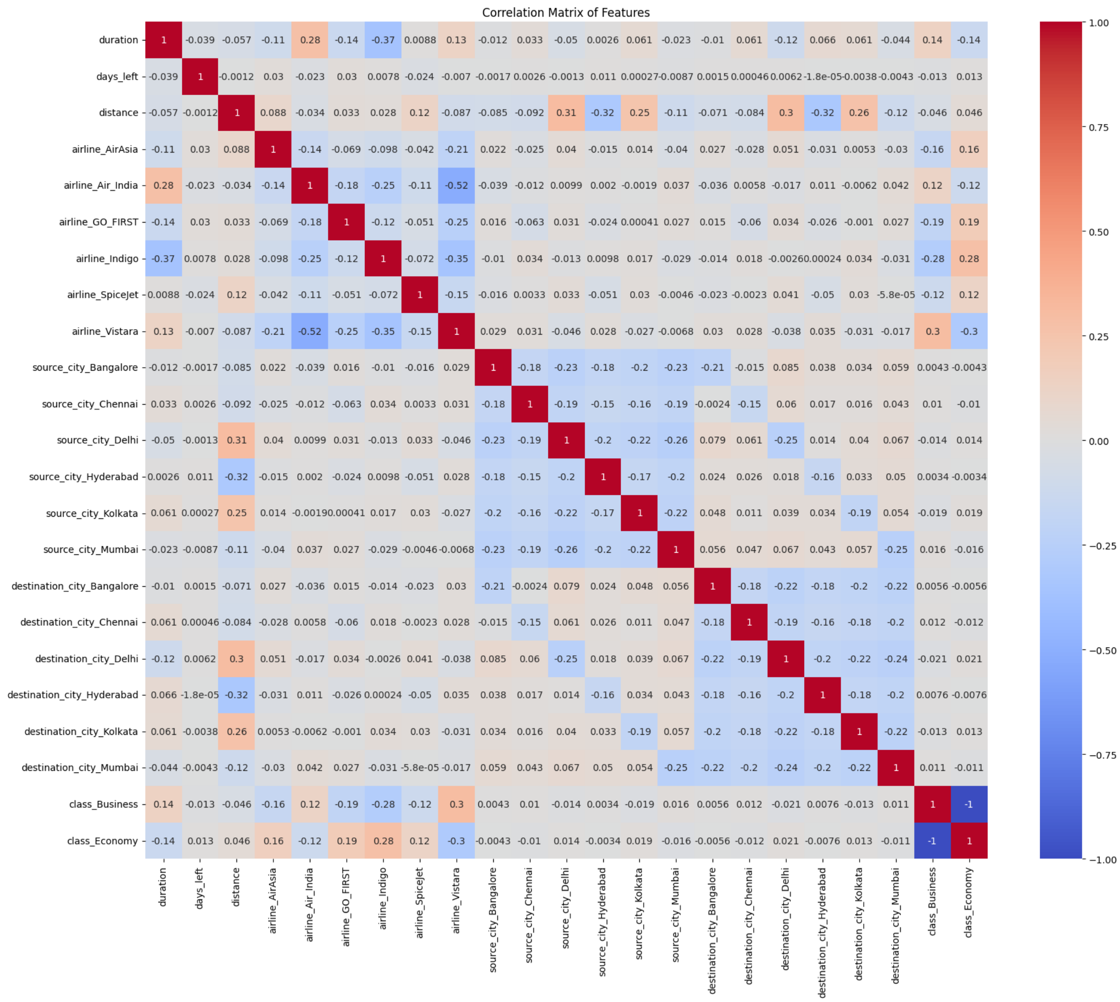
```
                       Feature  Importance
22               class_Economy    0.459372
21              class_Business    0.424111
0                     duration    0.063127
1                    days_left    0.017855
2                     distance    0.015988
8              airline_Vistara    0.005409
4            airline_Air_India    0.004620
17      destination_city_Delhi    0.001252
11           source_city_Delhi    0.001103
9        source_city_Bangalore    0.000755
15  destination_city_Bangalore    0.000716
13         source_city_Kolkata    0.000700
10         source_city_Chennai    0.000634
18  destination_city_Hyderabad    0.000617
19    destination_city_Kolkata    0.000570
20     destination_city_Mumbai    0.000569
12       source_city_Hyderabad    0.000523
14          source_city_Mumbai    0.000507
16    destination_city_Chennai    0.000491
3              airline_AirAsia    0.000488
6               airline_Indigo    0.000388
5             airline_GO_FIRST    0.000131
7             airline_SpiceJet    0.000075
Mean Squared Error on Test Set: 10959913.231577694
```

[ ]:
```python
# Plot the regression coefficients bar chart

plt.figure(figsize=(10, 6))
plt.barh(importances_df['Feature'], importances_df['Importance'])
plt.xlabel('Importance')
plt.ylabel('Feature')
```

```
plt.title('Feature Importance')
plt.gca().invert_yaxis()
plt.show()
```



[ ]:
```
features = ['airline', 'source_city', 'destination_city', 'class', 'duration',␣
↪'days_left', 'distance']

# Perform one-hot encoding for categorical variables
encoded_data = pd.get_dummies(clean_dataset_updated[features])

# Calculate the correlation matrix
correlation_matrix = encoded_data.corr()

# Plot the heatmap
plt.figure(figsize=(20, 16))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', center=0)
plt.title('Correlation Matrix of Features')
plt.show()
```



3.5 Deep Learning Model for Price Prediction

In this section, we develop a deep learning model using TensorFlow to predict flight prices. The process includes data preprocessing, model construction, training, and evaluation. The model incorporates feature interactions to improve predictive accuracy.

3.5.1 Step-by-Step Process

1. **Print Dataset Columns**:
   - Confirm the columns present in the dataset.
2. **Select Features and Target Variable**:
   - Identify the features to be used for the analysis: `airline`, `source_city`, `destination_city`, `class`, `duration`, `days_left`, and `distance`.
   - The target variable is `price`.
3. **Preprocess Data**:
   - Standardize numerical features (`duration`, `days_left`, `distance`) and one-hot encode categorical features (`airline`, `source_city`, `destination_city`, `class`) using

ColumnTransformer.
    - Extract features and target from the dataset.
    - Convert preprocessed features to a DataFrame.
4. **Split Data**:
    - Split the preprocessed DataFrame into training and testing sets.
    - Separate input features for the model.
5. **Build the Model**:
    - Define input layers for each feature set.
    - Create intermediate layers to capture feature interactions.
    - Construct hidden layers and an output layer.
6. **Compile and Train the Model**:
    - Compile the model with Adam optimizer and mean squared error loss function.
    - Train the model for 50 epochs with a batch size of 32.
7. **Evaluate the Model**:
    - Plot the change in loss during training.
    - Predict prices on the test set.

[ ]:
```
import pandas as pd
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, Input, concatenate
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
import numpy as np
import matplotlib.pyplot as plt
```

[ ]:
```
# Assuming clean_dataset_updated has been loaded as a DataFrame
print("Columns in the dataset:", clean_dataset_updated.columns)

# Select the features and target variable for analysis
features = ['airline', 'source_city', 'destination_city', 'class', 'duration',␣
↪'days_left', 'distance']
target = 'price'

# One-hot encode categorical variables
categorical_features = ['airline', 'source_city', 'destination_city', 'class']
numerical_features = ['duration', 'days_left', 'distance']

# Create a preprocessor: standardize numerical features and one-hot encode␣
↪categorical features
preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numerical_features),
        ('cat', OneHotEncoder(sparse_output=False), categorical_features)
    ])
```

```python
# Extract features and target
X = clean_dataset_updated[features]
y = clean_dataset_updated[target]

# Data preprocessing
X_preprocessed = preprocessor.fit_transform(X)

# Convert preprocessed features to DataFrame
encoded_features = preprocessor.named_transformers_['cat'].
 ↪get_feature_names_out(categorical_features)
all_features = np.concatenate([numerical_features, encoded_features])
X_preprocessed_df = pd.DataFrame(X_preprocessed, columns=all_features)
```

```
Columns in the dataset: Index(['Unnamed: 0', 'airline', 'flight', 'source_city',
'departure_time',
       'stops', 'arrival_time', 'destination_city', 'class', 'duration',
       'days_left', 'price', 'combined_date', 'distance', 'day_of_week',
       'week_of_year', 'month', 'is_holiday', 'route_class', 'price_bin',
       'days_left_bin'],
      dtype='object')
```

[ ]:
```python
# Split the preprocessed DataFrame into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_preprocessed_df, y,␣
 ↪test_size=0.2, random_state=42)

# Separate input features
train_airline = X_train.filter(like='airline')
train_source_city = X_train.filter(like='source_city')
train_destination_city = X_train.filter(like='destination_city')
train_class = X_train.filter(like='class')
train_duration = X_train[['duration']]
train_days_left = X_train[['days_left']]
train_distance = X_train[['distance']]

test_airline = X_test.filter(like='airline')
test_source_city = X_test.filter(like='source_city')
test_destination_city = X_test.filter(like='destination_city')
test_class = X_test.filter(like='class')
test_duration = X_test[['duration']]
test_days_left = X_test[['days_left']]
test_distance = X_test[['distance']]
```

[ ]:
```python
# Split the preprocessed DataFrame into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_preprocessed_df, y,␣
 ↪test_size=0.2, random_state=42)
```

```python
# Separate input features
train_airline = X_train.filter(like='airline')
train_source_city = X_train.filter(like='source_city')
train_destination_city = X_train.filter(like='destination_city')
train_class = X_train.filter(like='class')
train_duration = X_train[['duration']]
train_days_left = X_train[['days_left']]
train_distance = X_train[['distance']]

test_airline = X_test.filter(like='airline')
test_source_city = X_test.filter(like='source_city')
test_destination_city = X_test.filter(like='destination_city')
test_class = X_test.filter(like='class')
test_duration = X_test[['duration']]
test_days_left = X_test[['days_left']]
test_distance = X_test[['distance']]
```

[ ]:
```python
# Input layers
input_airline = Input(shape=(train_airline.shape[1],), name='airline_input')
input_source_city = Input(shape=(train_source_city.shape[1],),␣
 ↪name='source_city_input')
input_destination_city = Input(shape=(train_destination_city.shape[1],),␣
 ↪name='destination_city_input')
input_class = Input(shape=(train_class.shape[1],), name='class_input')
input_duration = Input(shape=(1,), name='duration_input')
input_days_left = Input(shape=(1,), name='days_left_input')
input_distance = Input(shape=(1,), name='distance_input')

# Feature interactions
# Combine source_city and destination_city to generate an intermediate layer␣
 ↪affecting duration and distance
combined_city = concatenate([input_source_city, input_destination_city])
duration_distance = concatenate([combined_city, input_duration, input_distance])

# Airline affects source_city and destination_city
combined_airline_city = concatenate([input_airline, combined_city])

# Combine all features to form the final input layer
combined_features = concatenate([input_airline, input_class,␣
 ↪combined_airline_city, input_days_left, duration_distance])

# Construct hidden layers and output layer
dense1 = Dense(128, activation='relu')(combined_features)
dense2 = Dense(64, activation='relu')(dense1)
dense3 = Dense(32, activation='relu')(dense2)
output = Dense(1, activation='linear')(dense3)
```

```
# Build the model
model = Model(inputs=[input_airline, input_source_city, input_destination_city,␣
 ↪input_class, input_duration, input_days_left, input_distance],␣
 ↪outputs=output)

# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error',␣
 ↪metrics=['mean_squared_error'])

# View the model structure
model.summary()
```

```
Model: "functional_7"


 Layer (type)             Output Shape          Param #  Connected to

 source_city_input        (None, 6)                   0  -
 (InputLayer)

 destination_city_i…      (None, 6)                   0  -
 (InputLayer)

 airline_input            (None, 6)                   0  -
 (InputLayer)

 concatenate              (None, 12)                  0  source_city_inpu…
 (Concatenate)                                           destination_city…

 duration_input           (None, 1)                   0  -
 (InputLayer)

 distance_input           (None, 1)                   0  -
 (InputLayer)

 class_input              (None, 2)                   0  -
 (InputLayer)

 concatenate_2            (None, 18)                  0  airline_input[0]…
 (Concatenate)                                           concatenate[0][0]

 days_left_input          (None, 1)                   0  -
 (InputLayer)

 concatenate_1            (None, 14)                  0  concatenate[0][0…
 (Concatenate)                                           duration_input[0…
                                                         distance_input[0…
```

concatenate_3 (Concatenate)	(None, 41)	0	airline_input[0]… class_input[0][0… concatenate_2[0]… days_left_input[… concatenate_1[0]…
dense_4 (Dense)	(None, 128)	5,376	concatenate_3[0]…
dense_5 (Dense)	(None, 64)	8,256	dense_4[0][0]
dense_6 (Dense)	(None, 32)	2,080	dense_5[0][0]
dense_7 (Dense)	(None, 1)	33	dense_6[0][0]

Total params: 15,745 (61.50 KB)

Trainable params: 15,745 (61.50 KB)

Non-trainable params: 0 (0.00 B)

[ ]:
```
# Train the model
history = model.fit(
    [train_airline, train_source_city, train_destination_city, train_class,␣
 ↪train_duration, train_days_left, train_distance],
    y_train,
    epochs=50,
    batch_size=32,
    validation_data=([test_airline, test_source_city, test_destination_city,␣
 ↪test_class, test_duration, test_days_left, test_distance], y_test)
)

# Plot the change in loss during training
plt.plot(history.history['loss'], label='train_loss')
plt.plot(history.history['val_loss'], label='val_loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Model Loss')
plt.legend()
plt.show()
```

```
Epoch 1/50
```

```
7504/7504              19s 2ms/step -
loss: 192505648.0000 - mean_squared_error: 192505648.0000 - val_loss:
```

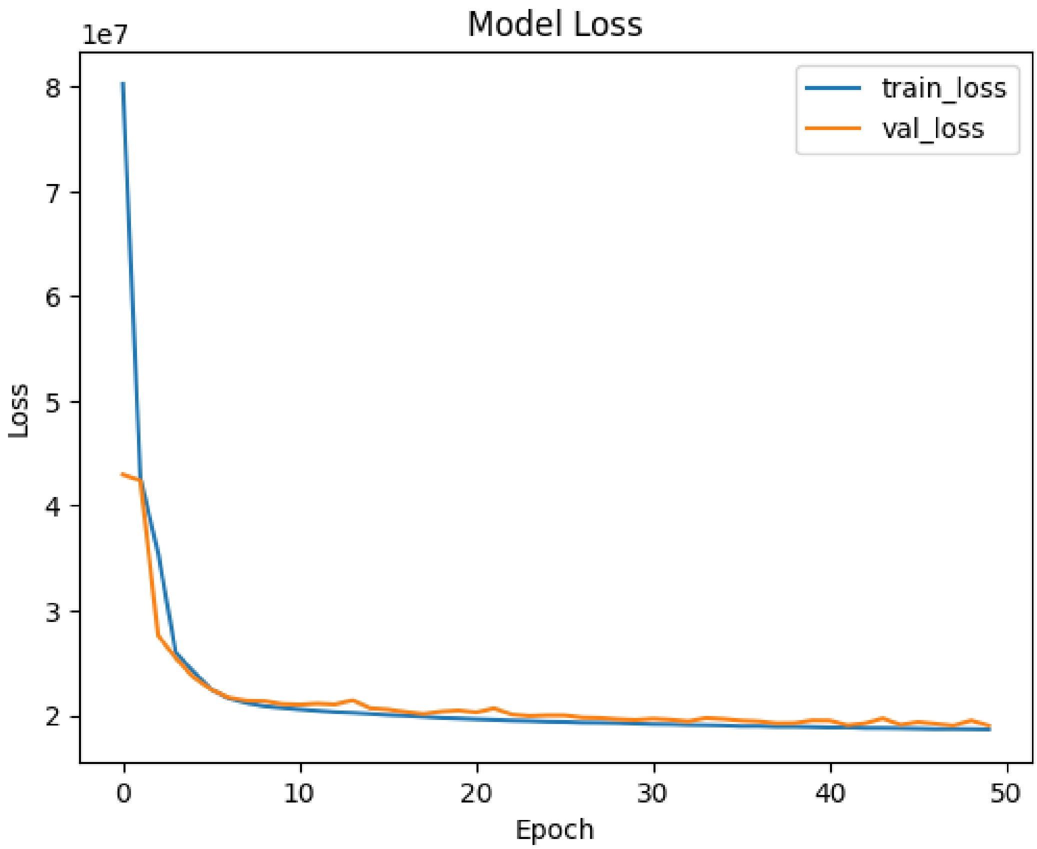
```
42964484.0000 - val_mean_squared_error: 42964484.0000
Epoch 2/50
7504/7504            19s 2ms/step -
loss: 42311652.0000 - mean_squared_error: 42311652.0000 - val_loss:
42399536.0000 - val_mean_squared_error: 42399536.0000
Epoch 3/50
7504/7504            16s 2ms/step -
loss: 39908180.0000 - mean_squared_error: 39908180.0000 - val_loss:
27604640.0000 - val_mean_squared_error: 27604640.0000
Epoch 4/50
7504/7504            17s 2ms/step -
loss: 26737412.0000 - mean_squared_error: 26737412.0000 - val_loss:
25444582.0000 - val_mean_squared_error: 25444582.0000
Epoch 5/50
7504/7504            16s 2ms/step -
loss: 24686468.0000 - mean_squared_error: 24686468.0000 - val_loss:
23614482.0000 - val_mean_squared_error: 23614482.0000
Epoch 6/50
7504/7504            21s 2ms/step -
loss: 22767270.0000 - mean_squared_error: 22767270.0000 - val_loss:
22454296.0000 - val_mean_squared_error: 22454296.0000
Epoch 7/50
7504/7504            22s 2ms/step -
loss: 21673888.0000 - mean_squared_error: 21673888.0000 - val_loss:
21677222.0000 - val_mean_squared_error: 21677222.0000
Epoch 8/50
7504/7504            21s 2ms/step -
loss: 21278714.0000 - mean_squared_error: 21278714.0000 - val_loss:
21391498.0000 - val_mean_squared_error: 21391498.0000
Epoch 9/50
7504/7504            20s 2ms/step -
loss: 20618222.0000 - mean_squared_error: 20618222.0000 - val_loss:
21351416.0000 - val_mean_squared_error: 21351416.0000
Epoch 10/50
7504/7504            22s 3ms/step -
loss: 20701380.0000 - mean_squared_error: 20701380.0000 - val_loss:
21057570.0000 - val_mean_squared_error: 21057570.0000
Epoch 11/50
7504/7504            20s 2ms/step -
loss: 20444308.0000 - mean_squared_error: 20444308.0000 - val_loss:
20998998.0000 - val_mean_squared_error: 20998998.0000
Epoch 12/50
7504/7504            18s 2ms/step -
loss: 20285142.0000 - mean_squared_error: 20285142.0000 - val_loss:
21094054.0000 - val_mean_squared_error: 21094054.0000
Epoch 13/50
7504/7504            20s 3ms/step -
loss: 20287518.0000 - mean_squared_error: 20287518.0000 - val_loss:
```

```
21014250.0000 - val_mean_squared_error: 21014250.0000
Epoch 14/50
7504/7504            17s 2ms/step -
loss: 20113172.0000 - mean_squared_error: 20113172.0000 - val_loss:
21424532.0000 - val_mean_squared_error: 21424532.0000
Epoch 15/50
7504/7504            23s 3ms/step -
loss: 20003584.0000 - mean_squared_error: 20003584.0000 - val_loss:
20667574.0000 - val_mean_squared_error: 20667574.0000
Epoch 16/50
7504/7504            27s 4ms/step -
loss: 20055872.0000 - mean_squared_error: 20055872.0000 - val_loss:
20549198.0000 - val_mean_squared_error: 20549198.0000
Epoch 17/50
7504/7504            31s 4ms/step -
loss: 19805788.0000 - mean_squared_error: 19805788.0000 - val_loss:
20315112.0000 - val_mean_squared_error: 20315112.0000
Epoch 18/50
7504/7504            31s 4ms/step -
loss: 19788478.0000 - mean_squared_error: 19788478.0000 - val_loss:
20132210.0000 - val_mean_squared_error: 20132210.0000
Epoch 19/50
7504/7504            27s 4ms/step -
loss: 19641516.0000 - mean_squared_error: 19641516.0000 - val_loss:
20344578.0000 - val_mean_squared_error: 20344578.0000
Epoch 20/50
7504/7504            39s 3ms/step -
loss: 19568130.0000 - mean_squared_error: 19568130.0000 - val_loss:
20457836.0000 - val_mean_squared_error: 20457836.0000
Epoch 21/50
7504/7504            41s 3ms/step -
loss: 19474474.0000 - mean_squared_error: 19474474.0000 - val_loss:
20265036.0000 - val_mean_squared_error: 20265036.0000
Epoch 22/50
7504/7504            43s 4ms/step -
loss: 19447924.0000 - mean_squared_error: 19447924.0000 - val_loss:
20664588.0000 - val_mean_squared_error: 20664588.0000
Epoch 23/50
7504/7504            39s 3ms/step -
loss: 19450354.0000 - mean_squared_error: 19450354.0000 - val_loss:
20074736.0000 - val_mean_squared_error: 20074736.0000
Epoch 24/50
7504/7504            47s 4ms/step -
loss: 19315552.0000 - mean_squared_error: 19315552.0000 - val_loss:
19923842.0000 - val_mean_squared_error: 19923842.0000
Epoch 25/50
7504/7504            39s 4ms/step -
loss: 19203420.0000 - mean_squared_error: 19203420.0000 - val_loss:
```

```
19986274.0000 - val_mean_squared_error: 19986274.0000
Epoch 26/50
7504/7504            43s 4ms/step -
loss: 19441760.0000 - mean_squared_error: 19441760.0000 - val_loss:
19982298.0000 - val_mean_squared_error: 19982298.0000
Epoch 27/50
7504/7504            27s 4ms/step -
loss: 19137204.0000 - mean_squared_error: 19137204.0000 - val_loss:
19768984.0000 - val_mean_squared_error: 19768984.0000
Epoch 28/50
7504/7504            44s 4ms/step -
loss: 19347832.0000 - mean_squared_error: 19347832.0000 - val_loss:
19732436.0000 - val_mean_squared_error: 19732436.0000
Epoch 29/50
7504/7504            37s 4ms/step -
loss: 19395234.0000 - mean_squared_error: 19395234.0000 - val_loss:
19615552.0000 - val_mean_squared_error: 19615552.0000
Epoch 30/50
7504/7504            26s 3ms/step -
loss: 18995330.0000 - mean_squared_error: 18995330.0000 - val_loss:
19542950.0000 - val_mean_squared_error: 19542950.0000
Epoch 31/50
7504/7504            25s 3ms/step -
loss: 18965998.0000 - mean_squared_error: 18965998.0000 - val_loss:
19657232.0000 - val_mean_squared_error: 19657232.0000
Epoch 32/50
7504/7504            24s 3ms/step -
loss: 19301024.0000 - mean_squared_error: 19301024.0000 - val_loss:
19568566.0000 - val_mean_squared_error: 19568566.0000
Epoch 33/50
7504/7504            44s 4ms/step -
loss: 19141302.0000 - mean_squared_error: 19141302.0000 - val_loss:
19418592.0000 - val_mean_squared_error: 19418592.0000
Epoch 34/50
7504/7504            42s 4ms/step -
loss: 19051838.0000 - mean_squared_error: 19051838.0000 - val_loss:
19732684.0000 - val_mean_squared_error: 19732684.0000
Epoch 35/50
7504/7504            26s 3ms/step -
loss: 18722286.0000 - mean_squared_error: 18722286.0000 - val_loss:
19635448.0000 - val_mean_squared_error: 19635448.0000
Epoch 36/50
7504/7504            25s 3ms/step -
loss: 18782306.0000 - mean_squared_error: 18782306.0000 - val_loss:
19482494.0000 - val_mean_squared_error: 19482494.0000
Epoch 37/50
7504/7504            27s 4ms/step -
loss: 19038224.0000 - mean_squared_error: 19038224.0000 - val_loss:
```

```
19410890.0000 - val_mean_squared_error: 19410890.0000
Epoch 38/50
7504/7504            41s 4ms/step -
loss: 18791342.0000 - mean_squared_error: 18791342.0000 - val_loss:
19228758.0000 - val_mean_squared_error: 19228758.0000
Epoch 39/50
7504/7504            42s 4ms/step -
loss: 18813490.0000 - mean_squared_error: 18813490.0000 - val_loss:
19242768.0000 - val_mean_squared_error: 19242768.0000
Epoch 40/50
7504/7504            31s 4ms/step -
loss: 18920764.0000 - mean_squared_error: 18920764.0000 - val_loss:
19504446.0000 - val_mean_squared_error: 19504446.0000
Epoch 41/50
7504/7504            36s 4ms/step -
loss: 18794576.0000 - mean_squared_error: 18794576.0000 - val_loss:
19490458.0000 - val_mean_squared_error: 19490458.0000
Epoch 42/50
7504/7504            41s 4ms/step -
loss: 18858050.0000 - mean_squared_error: 18858050.0000 - val_loss:
19053316.0000 - val_mean_squared_error: 19053316.0000
Epoch 43/50
7504/7504            23s 3ms/step -
loss: 18730820.0000 - mean_squared_error: 18730820.0000 - val_loss:
19227026.0000 - val_mean_squared_error: 19227026.0000
Epoch 44/50
7504/7504            23s 3ms/step -
loss: 18761082.0000 - mean_squared_error: 18761082.0000 - val_loss:
19695136.0000 - val_mean_squared_error: 19695136.0000
Epoch 45/50
7504/7504            22s 3ms/step -
loss: 18819152.0000 - mean_squared_error: 18819152.0000 - val_loss:
19089574.0000 - val_mean_squared_error: 19089574.0000
Epoch 46/50
7504/7504            25s 3ms/step -
loss: 18766210.0000 - mean_squared_error: 18766210.0000 - val_loss:
19347996.0000 - val_mean_squared_error: 19347996.0000
Epoch 47/50
7504/7504            29s 4ms/step -
loss: 18316112.0000 - mean_squared_error: 18316112.0000 - val_loss:
19181938.0000 - val_mean_squared_error: 19181938.0000
Epoch 48/50
7504/7504            26s 3ms/step -
loss: 18712088.0000 - mean_squared_error: 18712088.0000 - val_loss:
19009966.0000 - val_mean_squared_error: 19009966.0000
Epoch 49/50
7504/7504            40s 3ms/step -
loss: 18760308.0000 - mean_squared_error: 18760308.0000 - val_loss:
```

```
19493580.0000 - val_mean_squared_error: 19493580.0000
Epoch 50/50
7504/7504             24s 3ms/step -
loss: 18476988.0000 - mean_squared_error: 18476988.0000 - val_loss:
19014326.0000 - val_mean_squared_error: 19014326.0000
```




```
[ ]: # Prediction
y_pred = model.predict([test_airline, test_source_city, test_destination_city,
 ↪test_class, test_duration, test_days_left, test_distance])

# Calculate MSE
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error: {mse}")

# Calculate RMSE
rmse = np.sqrt(mse)
print(f"Root Mean Squared Error: {rmse}")

# Calculate R-Squared
r2 = r2_score(y_test, y_pred)
```

```
print(f"R-Squared: {r2}")
```

```
  17/1876              5s 3ms/step

1876/1876              3s 2ms/step
Mean Squared Error: 19014324.84346018
Root Mean Squared Error: 4360.541806181908
R-Squared: 0.963113523357812
```

3.6 Bayesian Neural Network for Price Prediction

This section outlines the process of developing a Bayesian Neural Network (BNN) using Pyro and PyTorch for predicting flight prices. The steps include data preprocessing, model construction, training, and evaluation.

3.6.1 Step-by-Step Process

1. **Print Dataset Columns**:
   - Confirm the columns present in the dataset.
2. **Select Features and Target Variable**:
   - Identify the features to be used for the analysis: `airline`, `source_city`, `destination_city`, `class`, `duration`, `days_left`, `distance`, and `stops`.
   - The target variable is `price`.
3. **Preprocess Data**:
   - Standardize numerical features (`duration`, `days_left`, `distance`) and one-hot encode categorical features (`airline`, `source_city`, `destination_city`, `class`, `stops`) using `ColumnTransformer`.
   - Extract features and target from the dataset.
   - Convert preprocessed features to a DataFrame.
4. **Split Data**:
   - Split the preprocessed DataFrame into training and testing sets.
   - Convert the data to torch tensors.
5. **Define Bayesian Neural Network**:
   - Define input layers for each feature set.
   - Create intermediate layers to capture feature interactions.
   - Construct hidden layers and an output layer.
6. **Compile and Train the Model**:
   - Define optimizer and loss function.
   - Train the model using Stochastic Variational Inference (SVI).
7. **Evaluate the Model**:
   - Use the trained model to make predictions on the test set.
   - Calculate and print Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and R-squared (R^2) values.

[ ]:
```
import pandas as pd
import torch
import torch.nn as nn
import pyro
import pyro.distributions as dist
```

```python
from pyro.nn import PyroModule, PyroSample
from pyro.infer import SVI, Trace_ELBO
from pyro.optim import Adam
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
import numpy as np
```

[ ]:
```python
# clean_dataset_updated = pd.read_csv('../datasets/Clean_Dataset_Updated.csv')
# Assuming clean_dataset_updated has been loaded as a DataFrame
print("Columns in the dataset:", clean_dataset_updated.columns)

# Select the features and target variable for analysis
features = ['airline', 'source_city', 'destination_city', 'class', 'duration',␣
↪'days_left', 'distance', 'stops']
target = 'price'

# One-hot encode categorical variables
categorical_features = ['airline', 'source_city', 'destination_city', 'class',␣
↪'stops']
numerical_features = ['duration', 'days_left', 'distance']

# Create a preprocessor: standardize numerical features and one-hot encode␣
↪categorical features
preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numerical_features),
        ('cat', OneHotEncoder(sparse_output=False), categorical_features)
    ])

# Extract features and target
X = clean_dataset_updated[features]
y = clean_dataset_updated[target]

# Data preprocessing
X_preprocessed = preprocessor.fit_transform(X)

# Convert preprocessed features to DataFrame
encoded_features = preprocessor.named_transformers_['cat'].
↪get_feature_names_out()
all_features = np.concatenate([numerical_features, encoded_features])
X_preprocessed_df = pd.DataFrame(X_preprocessed, columns=all_features)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_preprocessed_df, y,␣
↪test_size=0.2, random_state=42)
```

```
# Convert the data to torch tensors
X_train = torch.tensor(X_train.values, dtype=torch.float32)
X_test = torch.tensor(X_test.values, dtype=torch.float32)
y_train = torch.tensor(y_train.values, dtype=torch.float32)
y_test = torch.tensor(y_test.values, dtype=torch.float32)

print(f"X_train shape: {X_train.shape}")
print(f"X_test shape: {X_test.shape}")
print(f"Columns after preprocessing: {all_features}")
print(f"Number of features after preprocessing: {len(all_features)}")
```

```
Columns in the dataset: Index(['Unnamed: 0', 'airline', 'flight', 'source_city',
'departure_time',
       'stops', 'arrival_time', 'destination_city', 'class', 'duration',
       'days_left', 'price', 'combined_date', 'distance', 'day_of_week',
       'week_of_year', 'month', 'is_holiday', 'route_class', 'price_bin',
       'days_left_bin'],
      dtype='object')
X_train shape: torch.Size([240122, 26])
X_test shape: torch.Size([60031, 26])
Columns after preprocessing: ['duration' 'days_left' 'distance'
'airline_AirAsia' 'airline_Air_India'
 'airline_GO_FIRST' 'airline_Indigo' 'airline_SpiceJet' 'airline_Vistara'
 'source_city_Bangalore' 'source_city_Chennai' 'source_city_Delhi'
 'source_city_Hyderabad' 'source_city_Kolkata' 'source_city_Mumbai'
 'destination_city_Bangalore' 'destination_city_Chennai'
 'destination_city_Delhi' 'destination_city_Hyderabad'
 'destination_city_Kolkata' 'destination_city_Mumbai' 'class_Business'
 'class_Economy' 'stops_one' 'stops_two_or_more' 'stops_zero']
Number of features after preprocessing: 26
```

[ ]:
```
import torch
import torch.nn as nn
import pyro
import pyro.distributions as dist
from pyro.nn import PyroModule, PyroSample
from pyro.infer import SVI, Trace_ELBO
from pyro.optim import Adam
from sklearn.metrics import mean_squared_error, r2_score

# Define feature dimensions
input_dim_airline = X_train[:, :6].shape[1]  # One-hot encoded airline feature␣
↪dimension
input_dim_city = X_train[:, 6:18].shape[1]  # One-hot encoded city feature␣
↪dimension
input_dim_numerical = X_train[:, 18:].shape[1]  # Numerical feature dimension
```

```
class BayesianNN(PyroModule):
    def __init__(self, input_dim_airline, input_dim_city, input_dim_numerical):
        super().__init__()

        self.fc_airline = PyroModule[nn.Linear](input_dim_airline, 10)
        self.fc_airline.weight = PyroSample(dist.Normal(0., 1.).expand([10,␣
↪input_dim_airline]).to_event(2))
        self.fc_airline.bias = PyroSample(dist.Normal(0., 1.).expand([10]).
↪to_event(1))

        self.fc_city = PyroModule[nn.Linear](input_dim_city, 12)
        self.fc_city.weight = PyroSample(dist.Normal(0., 1.).expand([12,␣
↪input_dim_city]).to_event(2))
        self.fc_city.bias = PyroSample(dist.Normal(0., 1.).expand([12]).
↪to_event(1))

        combined_input_dim = 10 + 12 + input_dim_numerical
        self.fc_combined = PyroModule[nn.Linear](combined_input_dim, 30)
        self.fc_combined.weight = PyroSample(dist.Normal(0., 1.).expand([30,␣
↪combined_input_dim]).to_event(2))
        self.fc_combined.bias = PyroSample(dist.Normal(0., 1.).expand([30]).
↪to_event(1))

        self.fc_out = PyroModule[nn.Linear](30, 1)
        self.fc_out.weight = PyroSample(dist.Normal(0., 1.).expand([1, 30]).
↪to_event(2))
        self.fc_out.bias = PyroSample(dist.Normal(0., 1.).expand([1]).
↪to_event(1))

        self.relu = nn.ReLU()

    def forward(self, x_airline, x_city, x_numerical, y=None):
        x_airline = self.relu(self.fc_airline(x_airline))
        x_city = self.relu(self.fc_city(x_city))
        x_combined = torch.cat((x_airline, x_city, x_numerical), dim=1)
        x_combined = self.relu(self.fc_combined(x_combined))
        mean = self.fc_out(x_combined).squeeze(-1)
        sigma = pyro.sample("sigma", dist.Uniform(0., 10.))

        with pyro.plate("data", x_airline.shape[0]):
            obs = pyro.sample("obs", dist.Normal(mean, sigma), obs=y)

        return mean

# Instantiate model and guide
```

```python
bnn = BayesianNN(input_dim_airline, input_dim_city, input_dim_numerical)
guide = pyro.infer.autoguide.AutoDiagonalNormal(bnn)

# Define optimizer and loss function
optimizer = Adam({"lr": 0.01})
svi = SVI(bnn, guide, optimizer, loss=Trace_ELBO())

# Train the model (example code, specific training loop may need adjustment␣
 ↪based on actual case)
#num_iterations = 1000
num_iterations = 4500
for j in range(num_iterations):
    loss = svi.step(X_train[:, :input_dim_airline], X_train[:,␣
 ↪input_dim_airline:input_dim_airline + input_dim_city], X_train[:,␣
 ↪input_dim_airline + input_dim_city:], y_train)
    if j % 100 == 0:
        print(f"Step {j} : loss = {loss}")

# Evaluate the model
bnn.eval()
predictive = pyro.infer.Predictive(bnn, guide=guide, num_samples=1000)
samples = predictive(
    X_test[:, :input_dim_airline],
    X_test[:, input_dim_airline:input_dim_airline + input_dim_city],
    X_test[:, input_dim_airline + input_dim_city:]
)

# Get the mean of the predicted values
predictions = samples["obs"].mean(0).detach().numpy()

# Calculate MSE, RMSE, and R^2
mse = mean_squared_error(y_test, predictions)
rmse = np.sqrt(mse)
r2 = r2_score(y_test, predictions)

print(f"Mean Squared Error (MSE): {mse}")
print(f"Root Mean Squared Error (RMSE): {rmse}")
print(f"R^2 Score: {r2}")
```

```
Step 0 : loss = 2222693812396.671
Step 100 : loss = 1302426684299.5872
Step 200 : loss = 685733384797.0276
Step 300 : loss = 615829476180.835
Step 400 : loss = 566101021905.154
Step 500 : loss = 534166968318.87006
Step 600 : loss = 488293866420.3417
Step 700 : loss = 486128328460.40717
```

```
Step 800 : loss = 467087860390.3056
Step 900 : loss = 454955640427.21454
Step 1000 : loss = 439983581823.83075
Step 1100 : loss = 430337896151.9171
Step 1200 : loss = 419360256896.29926
Step 1300 : loss = 409382926425.0242
Step 1400 : loss = 391748894075.89056
Step 1500 : loss = 377884230616.1282
Step 1600 : loss = 360814824061.7006
Step 1700 : loss = 343856206277.7754
Step 1800 : loss = 324533246185.7589
Step 1900 : loss = 301976150172.6165
Step 2000 : loss = 281477671037.46277
Step 2100 : loss = 260345130929.2907
Step 2200 : loss = 238112929576.12643
Step 2300 : loss = 216904924664.19608
Step 2400 : loss = 196362601639.5636
Step 2500 : loss = 176463186756.64307
Step 2600 : loss = 158094659899.28873
Step 2700 : loss = 140689053689.24045
Step 2800 : loss = 123530944084.31726
Step 2900 : loss = 109996778682.18759
Step 3000 : loss = 97129294878.08228
Step 3100 : loss = 86496506500.29697
Step 3200 : loss = 76145474051.34634
Step 3300 : loss = 67033850168.089355
Step 3400 : loss = 60661254241.06781
Step 3500 : loss = 54518792032.825676
Step 3600 : loss = 50033775952.16765
Step 3700 : loss = 46269446950.91878
Step 3800 : loss = 43556434069.94115
Step 3900 : loss = 41470490340.50438
Step 4000 : loss = 39637535992.626945
Step 4100 : loss = 38124700314.70492
Step 4200 : loss = 36788406510.15152
Step 4300 : loss = 35717078437.57699
Step 4400 : loss = 34837742224.513626
Mean Squared Error (MSE): 28540378.0
Root Mean Squared Error (RMSE): 5342.3193359375
R^2 Score: 0.944633638293368
```

[ ]:
```
mae = mean_absolute_error(y_test, predictions)
print(f"Mean Absolute Error (MAE): {mae}")

# make a prediction
X_new = X_test[:1]
y_new = y_test[:1]
```

```
prediction = predictive(X_new[:, :input_dim_airline], X_new[:,␣
 ↪input_dim_airline:input_dim_airline + input_dim_city], X_new[:,␣
 ↪input_dim_airline + input_dim_city:])
print("Predicted price:", prediction["obs"].mean().item())
print("Actual price:", y_new.item())


# Plot the actual vs predicted prices
plt.figure(figsize=(10, 6))
plt.scatter(y_test, predictions, alpha=0.3)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], '--r',␣
 ↪linewidth=2)
plt.xlabel('Actual Price')
plt.ylabel('Predicted Price')
plt.title('Actual vs Predicted Prices')
plt.show()
```

```
Mean Absolute Error (MAE): 3337.95947265625

Predicted price: 7230.78369140625
Actual price: 7366.0
```