Uber_Fares_Prediction

December 13, 2024

1 Uber ride fare prediction

Problem description

This project focuses on analyzing Uber ride fares, including exploratory data analysis (EDA) with hypothesis testing, and building a model to predict future ride costs. The data is sourced from Kaggle: Uber Fares Dataset.

Project objective: - Understand the structure of the data and key factors affecting fares. - Build a predictive model that estimates future ride costs based on features like location, distance, and ride time.

Significance of the project: Predicting ride costs can be beneficial for Uber customers who want to better plan their expenses and for operators to optimize services and implement dynamic pricing effectively.

The dataset includes the following columns: - `key` – a unique identifier for each trip. - `fare_amount` – the cost of each trip in USD (target variable). - `pickup_datetime` – the date and time when the meter was engaged. - `passenger_count` – the number of passengers in the vehicle (entered by the driver). - `pickup_longitude` – the longitude where the meter was engaged. - `pickup_latitude` – the latitude where the meter was engaged. - `dropoff_longitude` – the longitude where the meter was disengaged. - `dropoff_latitude` – the latitude where the meter was disengaged.

1.1 Action plan

2 Import libraries

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import opendatasets as od
from scipy.stats import kstest, zscore, kruskal
import calendar
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import MinMaxScaler, OneHotEncoder
from sklearn.metrics import root_mean_squared_error, r2_score
from sklearn.linear_model import LinearRegression, Ridge, ElasticNet
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from xgboost import XGBRegressor
```

3 Download and load data

3.1 Download data

[2]:
```
dataset_url = 'https://www.kaggle.com/datasets/yasserh/uber-fares-dataset/data'
```

[3]:
```
od.download(dataset_url)
```

```
Skipping, found downloaded files in ".\uber-fares-dataset" (use force=True to
force download)
```

[4]:
```
data_dir = './uber-fares-dataset'
```

3.2 Loading training set

- Ignore the key column
- Parse pickup datetime while loading data

[5]:
```
selected_cols =␣
 ↪'fare_amount,pickup_datetime,pickup_longitude,pickup_latitude,dropoff_longitude,dropoff_lat
 ↪split(',')

df = pd.read_csv(data_dir+'/uber.csv',
                 usecols=selected_cols,
                 parse_dates=["pickup_datetime"])
```

[6]:
```
df.duplicated().sum()
```

[6]:
```
np.int64(0)
```

[7]:
```
df.shape
```

[7]: (200000, 7)

[8]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 200000 entries, 0 to 199999
Data columns (total 7 columns):
 #   Column             Non-Null Count   Dtype
---  ------             --------------   -----
 0   fare_amount        200000 non-null  float64
 1   pickup_datetime    200000 non-null  datetime64[ns, UTC]
 2   pickup_longitude   200000 non-null  float64
 3   pickup_latitude    200000 non-null  float64
 4   dropoff_longitude  199999 non-null  float64
 5   dropoff_latitude   199999 non-null  float64
 6   passenger_count    200000 non-null  int64
dtypes: datetime64[ns, UTC](1), float64(5), int64(1)
memory usage: 10.7 MB
```

[9]:
```
df.describe()
```

[9]:
```
         fare_amount  pickup_longitude  pickup_latitude  dropoff_longitude  \
count  200000.000000     200000.000000    200000.000000      199999.000000
mean       11.359955        -72.527638        39.935885         -72.525292
std         9.901776         11.437787         7.720539          13.117408
min       -52.000000      -1340.648410       -74.015515       -3356.666300
25%         6.000000        -73.992065        40.734796         -73.991407
50%         8.500000        -73.981823        40.752592         -73.980093
75%        12.500000        -73.967154        40.767158         -73.963658
max       499.000000         57.418457      1644.421482        1153.572603

       dropoff_latitude  passenger_count
count     199999.000000    200000.000000
mean          39.923890         1.684535
std            6.794829         1.385997
min         -881.985513         0.000000
25%           40.733823         1.000000
50%           40.753042         1.000000
75%           40.768001         2.000000
max          872.697628       208.000000
```

[10]:
```
df.pickup_datetime.min(), df.pickup_datetime.max()
```

[10]:
```
(Timestamp('2009-01-01 01:15:22+0000', tz='UTC'),
 Timestamp('2015-06-30 23:40:39+0000', tz='UTC'))
```

- **Data size:** the dataset contains 200,000 rows and 7 columns.
- **Data anomalies:**

  - `fare_amount`: minimum value is -52 (illogical), maximum is 499 (potential outlier).
  - `passenger_count`: minimum is 0, maximum is 208, which requires verification.
  - Geographic coordinates contain values outside realistic ranges.
- **Pickup dates:** the pickup dates range from January 1st, 2009, to June 30th, 2015.

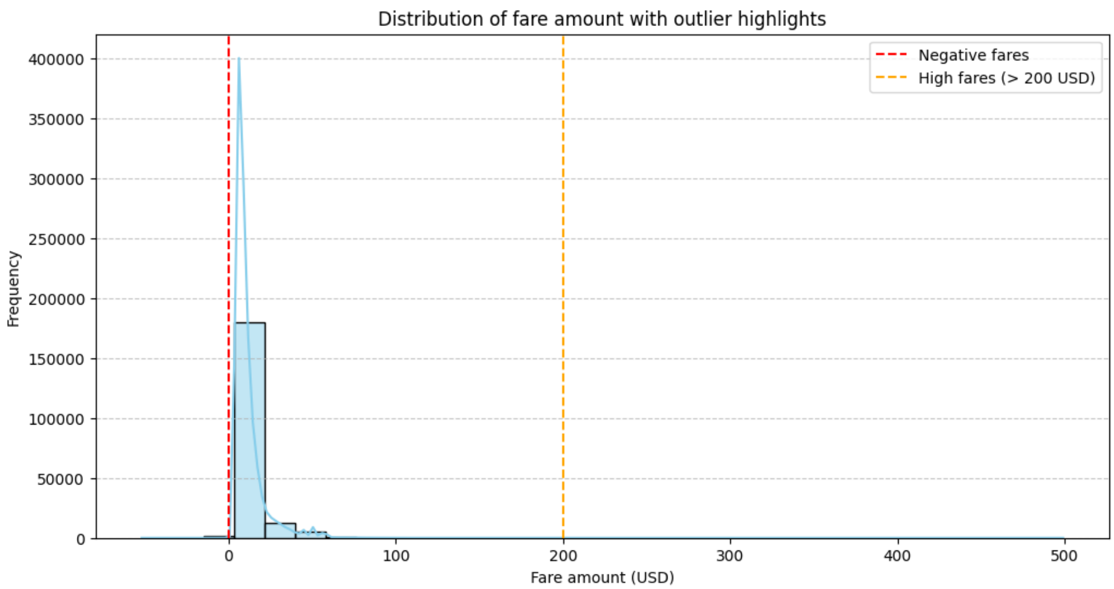
4 Exploratory data analysis (EDA) and visualization

[11]:
```
df
```

[11]:
```
        fare_amount           pickup_datetime  pickup_longitude  \
0               7.5 2015-05-07 19:52:06+00:00        -73.999817
1               7.7 2009-07-17 20:04:56+00:00        -73.994355
2              12.9 2009-08-24 21:45:00+00:00        -74.005043
3               5.3 2009-06-26 08:22:21+00:00        -73.976124
4              16.0 2014-08-28 17:47:00+00:00        -73.925023
…               …                         …               …
199995          3.0 2012-10-28 10:49:00+00:00        -73.987042
199996          7.5 2014-03-14 01:09:00+00:00        -73.984722
199997         30.9 2009-06-29 00:42:00+00:00        -73.986017
199998         14.5 2015-05-20 14:56:25+00:00        -73.997124
199999         14.1 2010-05-15 04:08:00+00:00        -73.984395

        pickup_latitude  dropoff_longitude  dropoff_latitude  passenger_count
0             40.738354         -73.999512         40.723217                1
1             40.728225         -73.994710         40.750325                1
2             40.740770         -73.962565         40.772647                1
3             40.790844         -73.965316         40.803349                3
4             40.744085         -73.973082         40.761247                5
…                   …                  …                 …              …
199995        40.739367         -73.986525         40.740297                1
199996        40.736837         -74.006672         40.739620                1
199997        40.756487         -73.858957         40.692588                2
199998        40.725452         -73.983215         40.695415                1
199999        40.720077         -73.985508         40.768793                1

[200000 rows x 7 columns]
```

[12]:
```
plt.figure(figsize=(12, 6))
sns.histplot(df['fare_amount'], bins=30, kde=True, color='skyblue',␣
 ↪edgecolor='black')

plt.axvline(x=0, color='red', linestyle='--', label='Negative fares')
plt.axvline(x=200, color='orange', linestyle='--', label='High fares (> 200␣
 ↪USD)')
```

```python
plt.title('Distribution of fare amount with outlier highlights')
plt.xlabel('Fare amount (USD)')
plt.ylabel('Frequency')
plt.legend()
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```



[13]:
```python
negative_fares = df[df['fare_amount'] < 0]
high_fares = df[df['fare_amount'] > 200]

negative_fare_count = negative_fares.shape[0]
high_fare_count = high_fares.shape[0]

print(f"The number of negative values in 'fare_amount': {negative_fare_count}")
print(f"The number of extreme high values in 'fare_amount': {high_fare_count}")
```

```
The number of negative values in 'fare_amount': 17
The number of extreme high values in 'fare_amount': 7
```

[14]:
```python
df.fare_amount.describe()
```

[14]:
```
count    200000.000000
mean         11.359955
std           9.901776
min         -52.000000
25%           6.000000
50%           8.500000
```

```
75%           12.500000
max          499.000000
Name: fare_amount, dtype: float64
```

Conclusion

1. Average fare: the mean value of `fare_amount` is **11.36 USD**.

2. Invalid values: the dataset contains 17 instances with negative values, which are logically inconsistent with the concept of fares.

3. 75th percentile: 75% of trips cost 12.50 USD or less, indicating that the majority of fares fall within a reasonable range.

4. Extreme values: there are 7 values exceeding 200 USD in the dataset, suggesting the presence of outliers.

5. Distribution: the data distribution is right-skewed (mean > median), indicating that most fare amounts are relatively low, with a few extreme high values.

Kolmogorov-Smirnov test

- Significance level ( ): 0.05
- H (Null Hypothesis): the `fare_amount` data follows a normal distribution.
- H (Alternative Hypothesis): the `fare_amount` data does not follow a normal distribution.

[15]:
```
standardized_fare = zscore(df['fare_amount'])
ks_stat, ks_p_value = kstest(standardized_fare, 'norm')

print(f'KS Test Statistic: {ks_stat}')
print(f'P-value: {ks_p_value}')
```

```
KS Test Statistic: 0.20732879525075143
P-value: 0.0
```

Results
- KS Test Statistic: 0.2073
- P-value: 0.0

Conclusion
Since **p-value < 0.05**, we reject the null hypothesis (H) at the 5% significance level. This means the `fare_amount` data does not follow a normal distribution.

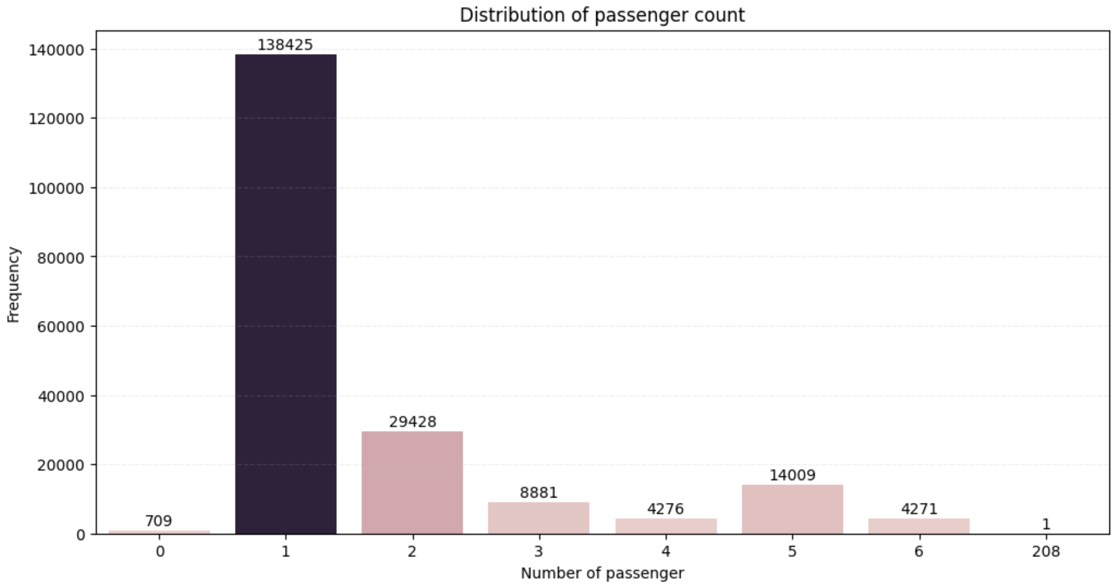
[16]:
```
df.passenger_count.describe()
```

[16]:
```
count    200000.000000
mean          1.684535
std           1.385997
min           0.000000
```

```
25%           1.000000
50%           1.000000
75%           2.000000
max         208.000000
Name: passenger_count, dtype: float64
```

[17]:

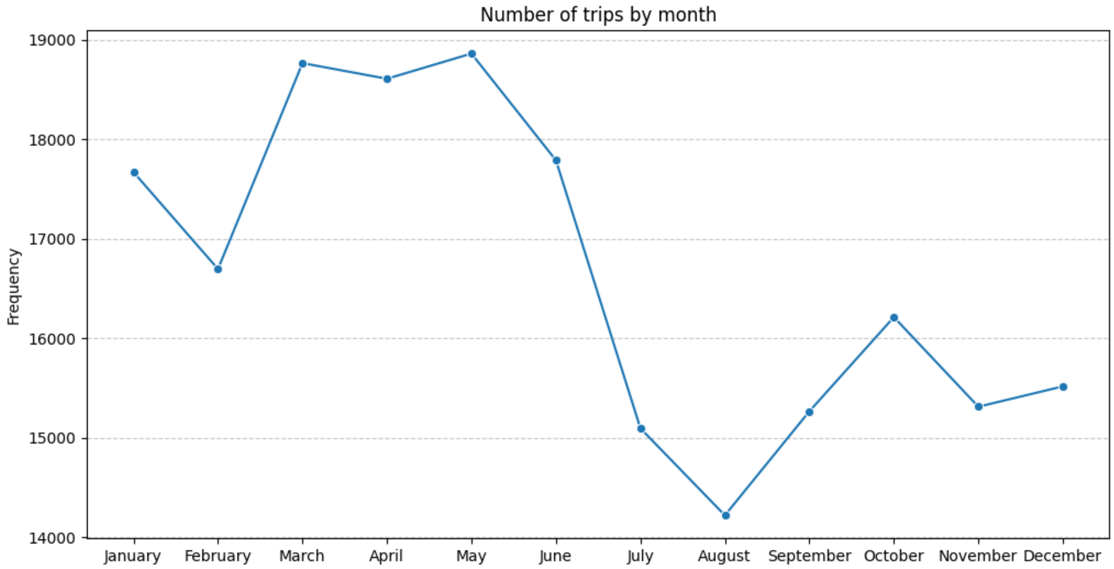
```python
passenger_counts = df['passenger_count'].value_counts().sort_index()
df_passengers = df.copy()
df_passengers['passangerFreq'] = df['passenger_count'].
 ↪map(df['passenger_count'].value_counts())

plt.figure(figsize=(12,6))
sns.countplot(
    x='passenger_count',
    data=df_passengers,
    hue='passangerFreq',
    legend=False
    )

for index, value in enumerate(passenger_counts.values):
    plt.text(
        x=index,
        y=value + 1500,
        s=str(value),
        ha='center',
        fontsize=10
    )

plt.title('Distribution of passenger count')
plt.xlabel('Number of passenger')
plt.ylabel('Frequency')
plt.grid(axis='y', linestyle='--', alpha=0.2)
plt.show()
```



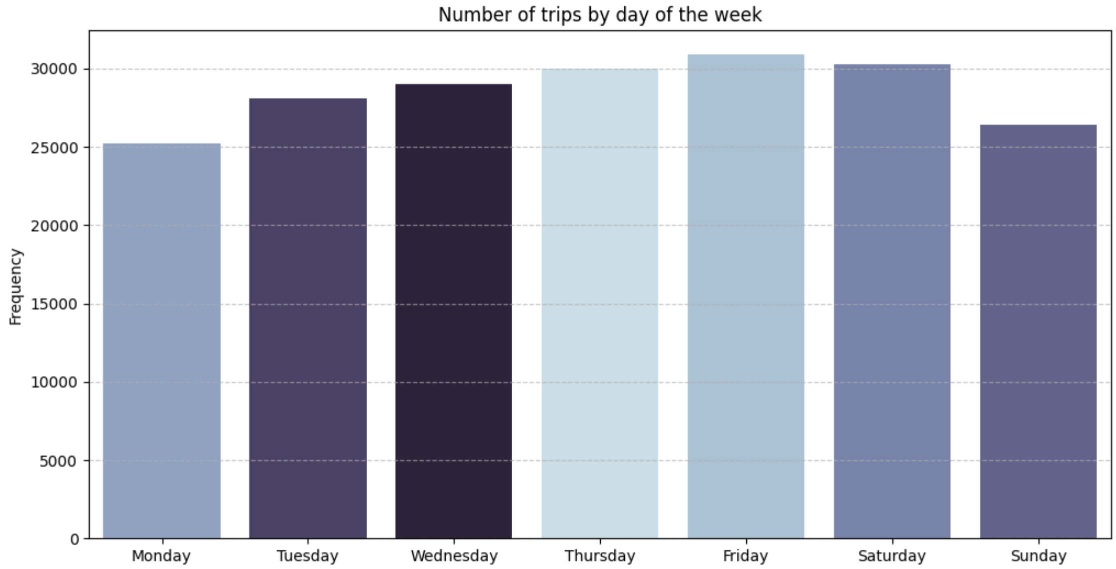
Conclusion

1. Most trips are taken with one passenger (138,425 cases), which is the dominant value.
2. There are atypical data points, such as 0 passengers (709 cases) and an extreme value of 208 passengers, which require cleaning.

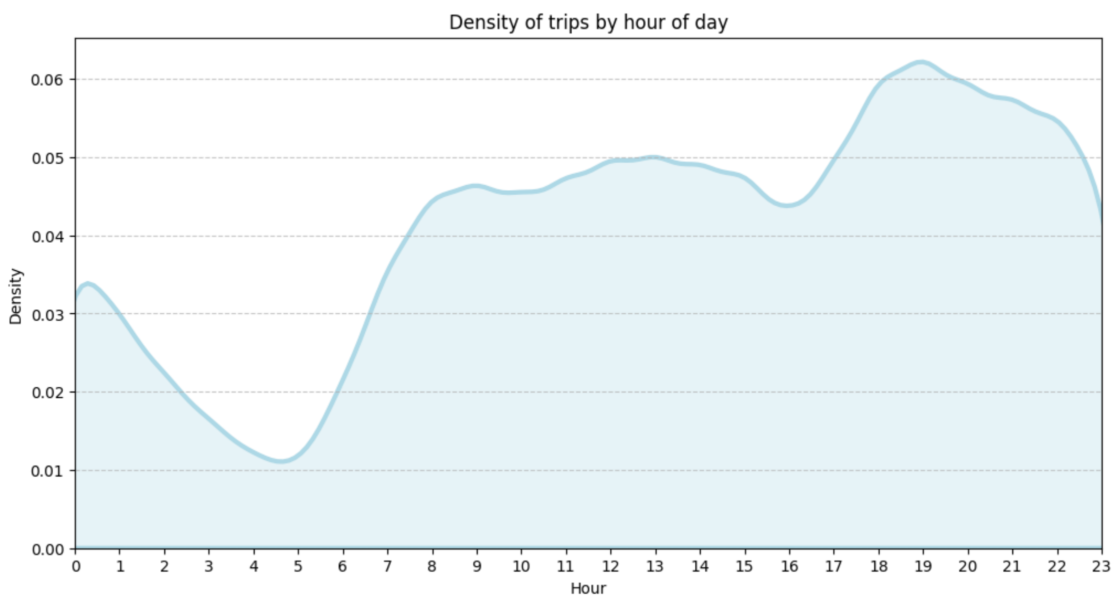
[18]:
```
monthly_data = df['pickup_datetime'].dt.month.value_counts().sort_index()
month_names = [calendar.month_name[i] for i in monthly_data.index]

plt.figure(figsize=(12,6))
sns.lineplot(x=month_names, y=monthly_data.values, marker='o')
plt.title('Number of trips by month')
plt.xlabel('')
plt.ylabel('Frequency')
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```



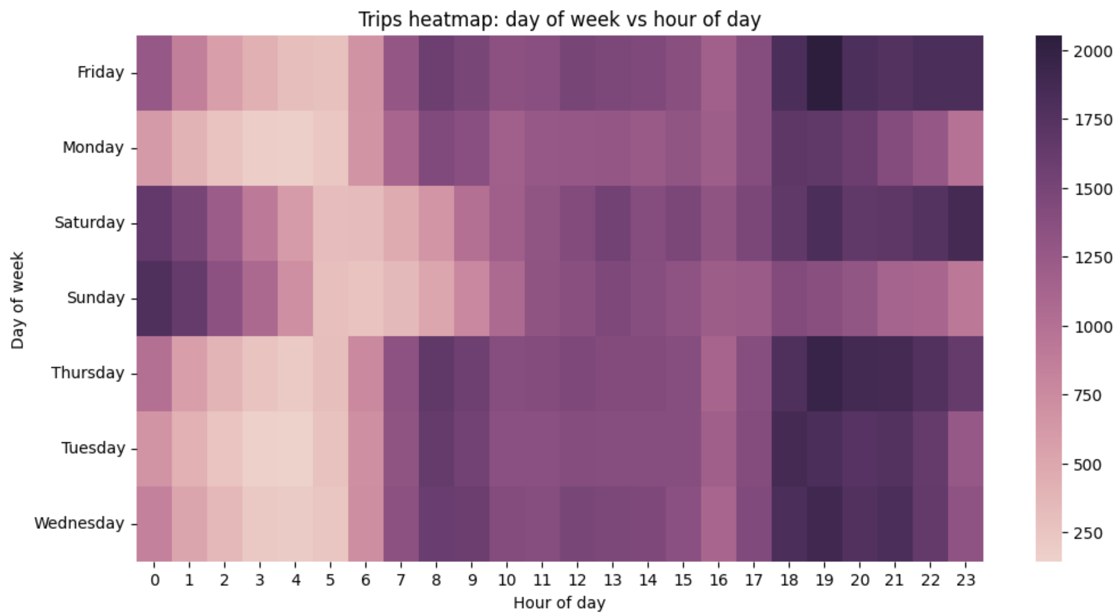
[19]:
```
plt.figure(figsize=(12,6))
sns.countplot(data=df,
              x=df.pickup_datetime.dt.day_name(),
              order=['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',␣
 ↪'Saturday', 'Sunday'],
              hue=df.pickup_datetime.dt.day_name(),
              palette='ch:s=.25,rot=-.25',
              dodge=False
              )
plt.title('Number of trips by day of the week')
plt.xlabel('')
plt.ylabel('Frequency')
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```



[20]:
```
plt.figure(figsize=(12, 6))
sns.kdeplot(df['pickup_datetime'].dt.hour, fill=True, color='lightblue',␣
↪alpha=0.3, linewidths=3)
plt.title('Density of trips by hour of day')
plt.xlabel('Hour')
plt.xlim(0, 23)
plt.xticks(range(0, 24))
plt.ylabel('Density')
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```



[21]:
```
heatmap_data = pd.crosstab(df['pickup_datetime'].dt.day_name(),
    df['pickup_datetime'].dt.hour)

plt.figure(figsize=(12, 6))
sns.heatmap(heatmap_data, cmap=sns.cubehelix_palette(as_cmap=True), annot=False)
plt.title('Trips heatmap: day of week vs hour of day')
plt.xlabel('Hour of day')
plt.ylabel('Day of week')
plt.show()
```



Conclusion

- **Seasonality:** the data shows a clear spring seasonality, with the highest number of trips in march, april, and may, and reduced activity during july and august.
- **Weekly Activity:** higher activity is observed at the end of the week (thursday–saturday), with fridays being the busiest day, while sundays and mondays have the lowest activity.
- **Hourly Trends:** peak activity occurs during evening hours (6:00 PM–9:00 PM), while the fewest trips are recorded during nighttime (4:00 AM–5:00 AM).

[22]:
```
invalid_coords = df[
    (df.pickup_longitude < -180) | (df.pickup_longitude > 180) |
    (df['pickup_latitude'] < -90) | (df['pickup_latitude'] > 90) |
    (df['dropoff_longitude'] < -180) | (df['dropoff_longitude'] > 180) |
    (df['dropoff_latitude'] < -90) | (df['dropoff_latitude'] > 90)
]
```

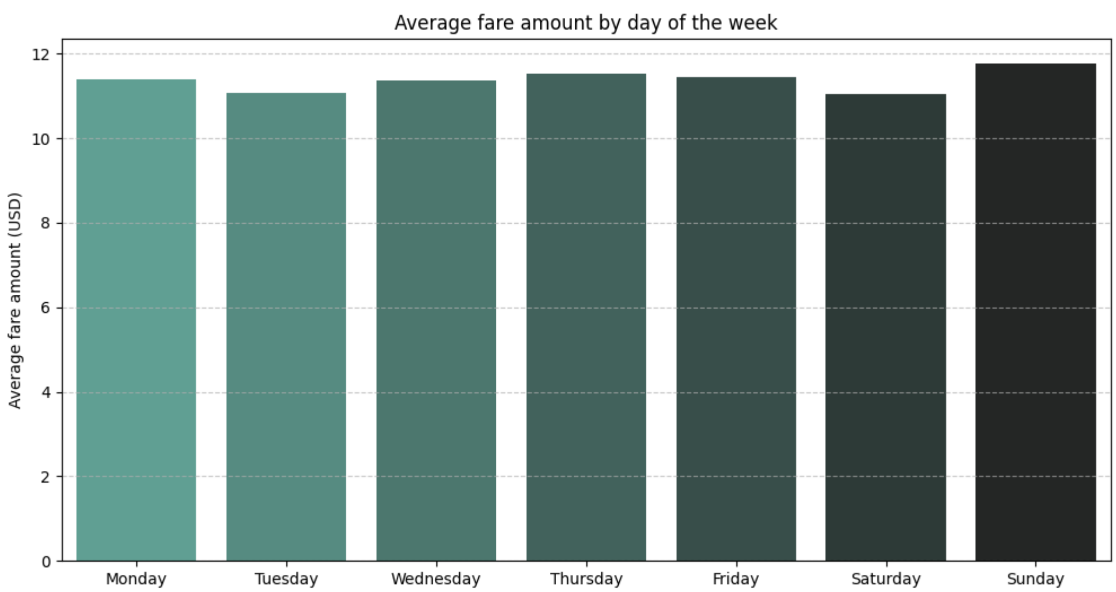
```python
print(f'Number of invalid coordinates: {len(invalid_coords)}')
```

```
Number of invalid coordinates: 12
```

[23]:
```python
fare_by_weekday = df.groupby(df['pickup_datetime'].dt.
    ↪day_name())['fare_amount'].mean()
fare_by_weekday = fare_by_weekday.reindex(['Monday', 'Tuesday', 'Wednesday',␣
    ↪'Thursday', 'Friday', 'Saturday', 'Sunday'])
fare_by_weekday
```

[23]:
```
pickup_datetime
Monday       11.378528
Tuesday      11.075793
Wednesday    11.351323
Thursday     11.517768
Friday       11.439793
Saturday     11.032273
Sunday       11.756463
Name: fare_amount, dtype: float64
```

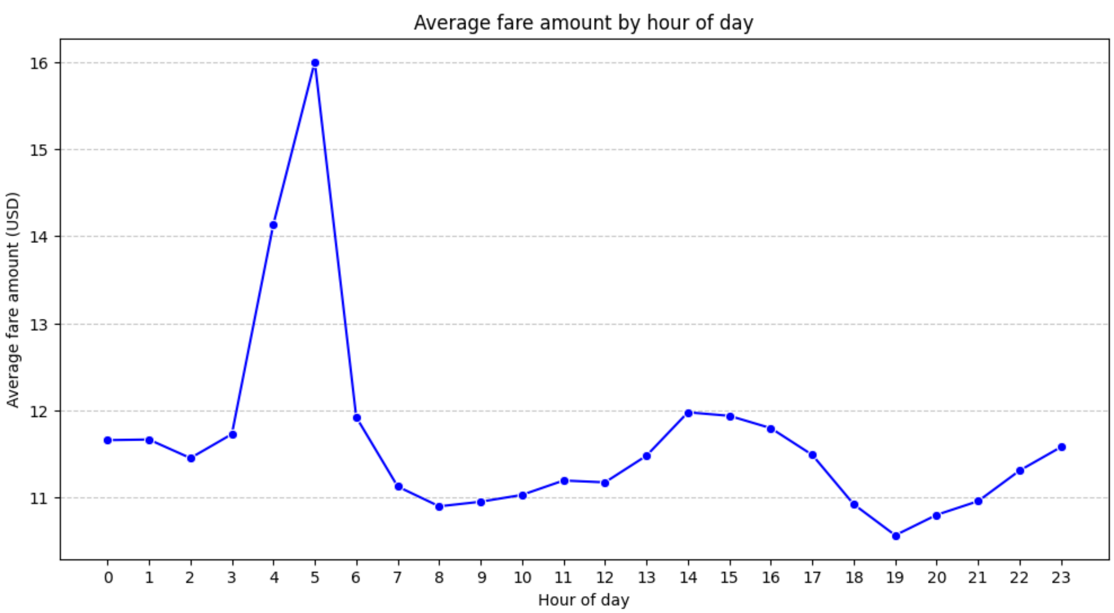
[24]:
```python
plt.figure(figsize=(12, 6))
sns.barplot(x=fare_by_weekday.index, y=fare_by_weekday.values, palette='dark:
    ↪#5A9_r', hue=fare_by_weekday.index)
plt.title('Average fare amount by day of the week')
plt.xlabel('')
plt.ylabel('Average fare amount (USD)')
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```



Conclusion

The average fare amount varies slightly across the days of the week, with sunday having the highest average fare ($11.76) and saturday the lowest ($11.03).

[25]:
```
fare_by_hour = df.groupby(df['pickup_datetime'].dt.hour)['fare_amount'].mean()

plt.figure(figsize=(12, 6))
sns.lineplot(x=fare_by_hour.index, y=fare_by_hour.values, marker='o',␣
 ↪color='blue')
plt.title('Average fare amount by hour of day')
plt.xlabel('Hour of day')
plt.ylabel('Average fare amount (USD)')
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.xticks(range(0, 24))
plt.show()
```



Conclusion

The average fare amount peaks significantly around 4-5 AM, likely due to limited availability or longer trips, and remains relatively stable throughout the day.

Kruskal-Wallis Test

- Significance level ( ): 0.05

- H (Null Hypothesis): the average fare amount is the same across all groups of passenger count.

- H (Alternative Hypothesis): at least one group of passenger count has a different average fare amount.

[26]:
```python
sample_df = df.sample(400, random_state=42)
groups = [sample_df[sample_df['passenger_count'] == p]['fare_amount'] for p in range(1, 7)]

h_stat, p_value = kruskal(*groups)
print(f"Kruskal-Wallis test: H-statistic={h_stat}, p-value={p_value}")
```

```
Kruskal-Wallis test: H-statistic=4.109764170530943, p-value=0.5337233805951217
```

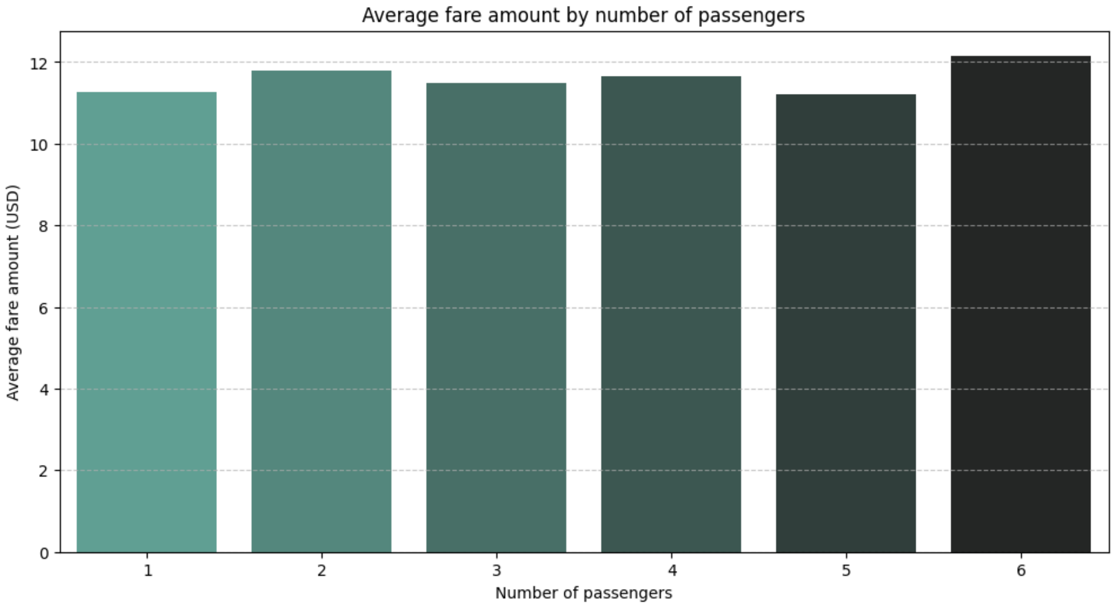
Interpretation

Since the p-value > 0.05, there is no basics to reject the null hypothesis.

This indicates that there is no statistically significant difference in the average fare amount across the different groups of passenger count.

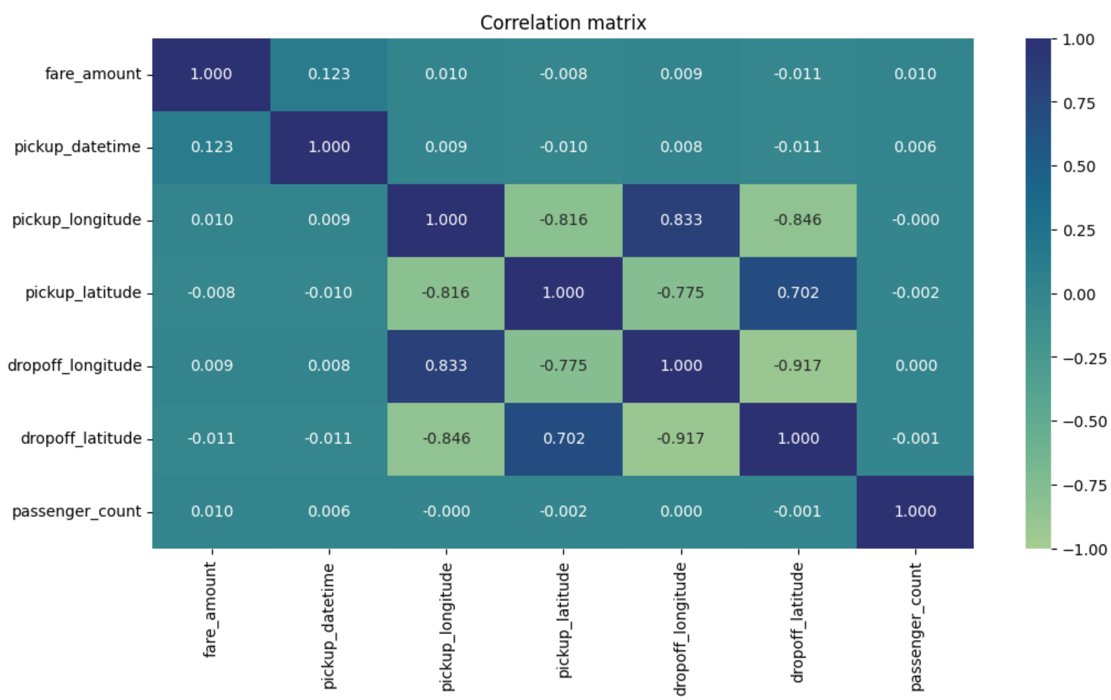
[27]:
```python
fare_by_passenger = df.groupby('passenger_count')['fare_amount'].mean()
fare_by_passenger = fare_by_passenger.loc[1:6,]
print(fare_by_passenger)

plt.figure(figsize=(12, 6))
sns.barplot(x=fare_by_passenger.index, y=fare_by_passenger.values, palette='dark:#5A9_r', hue=fare_by_passenger.index, legend=False)
plt.title('Average fare amount by number of passengers')
plt.xlabel('Number of passengers')
plt.ylabel('Average fare amount (USD)')
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```

```
passenger_count
1    11.254158
2    11.784452
3    11.486731
4    11.642472
5    11.199698
6    12.158537
Name: fare_amount, dtype: float64
```



[28]:

```
corrMatrix = df.corr()
plt.figure(figsize=(12,6))
sns.heatmap(corrMatrix, annot=True, cmap="crest", fmt='.3f', vmin = -1, vmax =␣
↪1, center = 0)
plt.title('Correlation matrix')
plt.show()
```



The correlation matrix indicates that the existing features have relatively weak linear relationships with the fare amount. This suggests that the model may benefit from additional, more informative features. By introducing distance-based features (using the Haversine formula) and decomposing the pickup_datetime into granular components (year, month, day, hour, etc.), we aim to uncover stronger patterns and potentially enhance the predictive power of the model.

5 Data Preprocessing

5.1 Remove outliers and invalid data

Removing outliers ensures the dataset is clean and reliable, preventing extreme or incorrect values from skewing the analysis and model training. We will remove rows with:

- `fare_amount`: negative values, as they are not logical.
- `passenger_count`: values outside the valid range of 1 to 6 passengers.
- Geographic coordinates (`pickup_longitude`, `pickup_latitude`, `dropoff_longitude`, `dropoff_latitude`): invalid coordinates outside the NY.

[29]:
```
print(f"Original data size: {df.shape}")

def remove_outliers(df):
    return df[
        (df['fare_amount'] >= 1.0) &
        (df['passenger_count'].between(1, 6)) &
        (df['pickup_longitude'].between(-75, -72)) &
        (df['pickup_latitude'].between(40, 42)) &
        (df['dropoff_longitude'].between(-75, -72)) &
        (df['dropoff_latitude'].between(40, 42))
    ]

df = remove_outliers(df).copy()

print(f"Data size after outlier removal: {df.shape}")
```

```
Original data size: (200000, 7)
Data size after outlier removal: (195097, 7)
```

5.2 Impute missing numerical data

After analyzing the dataset, no missing values were detected in any of the columns.

[30]:
```
df.isna().sum()
```

[30]:
```
fare_amount          0
pickup_datetime      0
pickup_longitude     0
pickup_latitude      0
```

```
dropoff_longitude    0
dropoff_latitude     0
passenger_count      0
dtype: int64
```

5.3 Feature engineering

Feature engineering allows us to extract additional meaningful information from the existing dataset, which can improve the accuracy and interpretability of predictive models.

Planned Operations: - Extract parts of date: add columns for year, month, day, day of the week, and hour from the `pickup_datetime` column. - Add distance between pickup and drop: calculate the distance in kilometers between the pickup and drop-off locations using the Haversine formula. - Add distance from popular landmarks: calculate the distance from pickup and drop-off points to popular landmarks, such as. These landmarks are based on the list from Tourism in New York City: - Times Square - World Trade Center - Central Park - Statue of Liberty - JFK Airport - LGA Airport - EWR Airport

[31]:
```python
def add_dateparts(df, col):
    df['Year'] = df[col].dt.year
    df['Month'] = df[col].dt.month
    df['Day'] = df[col].dt.day
    df['DayOfWeek'] = df[col].dt.dayofweek
    df['Hour'] = df[col].dt.hour

add_dateparts(df, 'pickup_datetime')
```

[32]:
```python
def haversine_np(lon1, lat1, lon2, lat2):

    lon1, lat1, lon2, lat2 = map(np.radians, [lon1, lat1, lon2, lat2])

    dlon = lon2 - lon1
    dlat = lat2 - lat1

    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))
    km = 6367 * c
    return km


def add_trip_distance(df):
    df['distance_km'] = haversine_np(df['pickup_longitude'],␣
 ↪df['pickup_latitude'], df['dropoff_longitude'], df['dropoff_latitude'])

add_trip_distance(df)
```

[33]:
```python
landmarks = {
    "Times Square": (-73.9855, 40.7580),
    "World Trade Center": (-74.0131, 40.7115),
    "JFK Airport": (-73.7781, 40.6413),
    "LGA Airport": (-73.8740, 40.7769),
    "EWR Airport": (-74.1745, 40.6895),
    "Statue of Liberty": (-74.0445, 40.6892),
    "Central Park": (-73.9683, 40.7851)
}

def add_landmark_distances(df, landmarks):
    for name, (lon, lat) in landmarks.items():
        df[f"distance_to_{name.replace(' ', '_').lower()}"] = haversine_np(
            df['pickup_longitude'], df['pickup_latitude'], lon, lat
        )

add_landmark_distances(df, landmarks)
```

[34]:
```python
df.head()
```

[34]:
```
   fare_amount           pickup_datetime  pickup_longitude  pickup_latitude  \
0          7.5 2015-05-07 19:52:06+00:00        -73.999817        40.738354
1          7.7 2009-07-17 20:04:56+00:00        -73.994355        40.728225
2         12.9 2009-08-24 21:45:00+00:00        -74.005043        40.740770
3          5.3 2009-06-26 08:22:21+00:00        -73.976124        40.790844
4         16.0 2014-08-28 17:47:00+00:00        -73.925023        40.744085

   dropoff_longitude  dropoff_latitude  passenger_count  Year  Month  Day  \
0         -73.999512         40.723217                1  2015      5    7
1         -73.994710         40.750325                1  2009      7   17
2         -73.962565         40.772647                1  2009      8   24
3         -73.965316         40.803349                3  2009      6   26
4         -73.973082         40.761247                5  2014      8   28

   DayOfWeek  Hour  distance_km  distance_to_times_square  \
0          3    19     1.682266                  2.493806
1          4    20     2.456047                  3.391702
2          0    21     5.033215                  2.524443
3          4     8     1.660640                  3.734106
4          3    17     4.472640                  5.320792

   distance_to_world_trade_center  distance_to_jfk_airport  \
0                        3.186907                21.571634
1                        2.438593                20.624994
2                        3.322650                22.086949
3                        9.350436                23.544665
4                        8.253876                16.843654
```

```
   distance_to_lga_airport  distance_to_ewr_airport  \
0                11.424060                15.683180
1                11.485096                15.773168
2                11.738305                15.368224
3                 8.731469                20.144671
4                 5.633972                21.870565

   distance_to_statue_of_liberty  distance_to_central_park
0                       6.633326                  5.832831
1                       6.053859                  6.689963
2                       6.624671                  5.816470
3                      12.677749                  0.916934
4                      11.767161                  5.834464
```

5.4 Training, validation, and test sets

This step involves splitting the dataset into three subsets: - **Training set**: used to train the model. - **Validation set**: used to tune model parameters and avoid overfitting. - **Test set**: used to evaluate the final performance of the model on unseen data.

By separating these subsets, we ensure the model's performance is evaluated on data it has not been trained on, maintaining its generalization ability.

[35]:
```
train_val_df, test_df = train_test_split(df, test_size=0.2, random_state=42)
train_df, val_df = train_test_split(train_val_df, test_size=0.25,␣
 ↪random_state=42)
```

[36]:
```
print('train_df.shape :', train_df.shape)
print('val_df.shape :', val_df.shape)
print('test_df.shape :', test_df.shape)
```

```
train_df.shape : (117057, 20)
val_df.shape : (39020, 20)
test_df.shape : (39020, 20)
```

5.5 Identifying input and target columns

In this step, we separate the dataset into: - **Input columns (features):** variables that the model will use to make predictions - **Target column:** the variable we aim to predict, in this case, `fare_amount`.

[37]:
```
input_cols = [col for col in df.columns if col not in ['fare_amount',␣
 ↪'pickup_datetime']]
target_col = 'fare_amount'
```

[38]:
```
train_inputs = train_df[input_cols].copy()
train_targets = train_df[target_col].copy()
```

```
val_inputs = val_df[input_cols].copy()
val_targets = val_df[target_col].copy()

test_inputs = test_df[input_cols].copy()
test_targets = test_df[target_col].copy()
```

[39]:
```
train_inputs
```

[39]:
```
        pickup_longitude  pickup_latitude  dropoff_longitude  \
167725        -73.978657        40.756217         -73.982102
89538         -73.862652        40.769010         -73.987463
139145        -73.991367        40.750028         -73.974102
24145         -74.008415        40.745135         -73.967803
91963         -73.956500        40.766988         -73.978757
…                    …               …                  …
55846         -73.944257        40.788072         -73.953770
186827        -73.982840        40.744962         -73.994233
124517        -73.981367        40.784438         -73.966772
94676         -73.984227        40.725100         -73.978428
17854         -73.997769        40.741334         -73.988508

        dropoff_latitude  passenger_count  Year  Month  Day  DayOfWeek  Hour  \
167725         40.752237                3  2013      9    1          6    21
89538          40.732503                1  2010      4   30          4    13
139145         40.760328                4  2009      7   23          3    16
24145          40.710770                3  2010     11   17          2    17
91963          40.765180                1  2011      6   21          1     8
…                    …               …    …     …    …         …     …
55846          40.775090                1  2013      8   26          0     6
186827         40.746320                1  2012     12   13          3     0
124517         40.789953                4  2010     11    9          1     6
94676          40.724597                1  2014      3   20          3    23
17854          40.722683                1  2009      9   13          6    20

        distance_km  distance_to_times_square  distance_to_world_trade_center  \
167725     0.528874                  0.609138                        5.753637
89538     11.263022                 10.411930                       14.188072
139145     1.849934                  1.014257                        4.656172
24145      5.126400                  2.401048                        3.758458
91963      1.883985                  2.637335                        7.793072
…                 …                       …                               …
55846      1.649823                  4.818080                       10.295265
186827     0.970975                  1.466051                        4.507840
124517     1.372421                  2.958444                        8.534162
94676      0.491559                  3.657587                        2.863174
17854      2.214453                  2.120555                        3.557832
```

```
        distance_to_jfk_airport  distance_to_lga_airport  \
167725                21.179667                 9.103232
89538                 15.878935                 1.296439
139145                21.652920                10.319955
24145                 22.576844                11.851658
91963                 20.516972                 7.029758
…                           …                       …
55846                 21.491511                 6.040614
186827                20.743260                 9.824590
124517                23.369907                 9.073216
94676                 19.708864                10.919684
17854                 21.590641                11.142496

        distance_to_ewr_airport  distance_to_statue_of_liberty  \
167725                18.083346                       9.285013
89538                 27.707467                      17.696417
139145                16.826768                       8.106594
24145                 15.293950                       6.919052
91963                 20.277351                      11.385969
…                           …                             …
55846                 22.266767                      13.854860
186827                17.279279                       8.085084
124517                19.384386                      11.843371
94676                 16.509307                       6.457122
17854                 15.961315                       7.004011

        distance_to_central_park
167725                  3.325874
89538                   9.068321
139145                  4.354135
24145                   5.578786
91963                   2.244321
…                             …
55846                   2.049716
186827                  4.625174
124517                  1.101922
94676                   6.800966
17854                   5.459463

[117057 rows x 18 columns]
```

[40]:
```
train_targets
```

[40]:
```
167725     4.00
89538     28.27
139145     7.30
24145     15.30
```

```
91963      10.10
           …
55846       5.00
186827      6.00
124517      5.30
94676       4.00
17854      10.50
Name: fare_amount, Length: 117057, dtype: float64
```

Let's also identify which of the columns are numerical and which ones are categorical.

[41]:
```python
categorical_cols = ['DayOfWeek']
numeric_cols = [col for col in train_inputs.columns if col not in␣
 ↪categorical_cols]
```

5.6 Scaling numeric features

Another good practice is to scale numeric features to a small range of values. Scaling ensures that no particular feature has a disproportionate impact on the model's loss. In this project, we applied **MinMaxScaler** to normalize the numerical data within the range [0, 1].

[42]:
```python
scaler = MinMaxScaler().fit(df[numeric_cols])
```

[43]:
```python
train_inputs[numeric_cols] = scaler.transform(train_inputs[numeric_cols])
val_inputs[numeric_cols] = scaler.transform(val_inputs[numeric_cols])
test_inputs[numeric_cols] = scaler.transform(test_inputs[numeric_cols])
```

[44]:
```python
train_inputs[numeric_cols].describe()
```

[44]:

	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	\
count	117057.000000	117057.000000	117057.000000	117057.000000	
mean	0.378199	0.548881	0.382955	0.498991	
std	0.020286	0.022344	0.020285	0.022608	
min	0.000000	0.000000	0.000000	0.000000	
25%	0.369437	0.538153	0.374067	0.488333	
50%	0.374585	0.550621	0.379692	0.500633	
75%	0.381560	0.561134	0.387368	0.510430	
max	1.000000	1.000000	1.000000	0.910431	

	passenger_count	Year	Month	Day	\
count	117057.000000	117057.000000	117057.000000	117057.000000	
mean	0.138100	0.457793	0.479910	0.489350	
std	0.261332	0.310409	0.312659	0.289453	
min	0.000000	0.000000	0.000000	0.000000	
25%	0.000000	0.166667	0.181818	0.233333	
50%	0.000000	0.500000	0.454545	0.500000	
75%	0.200000	0.666667	0.727273	0.733333	
max	1.000000	1.000000	1.000000	1.000000	

```
                Hour    distance_km  distance_to_times_square  \
count  117057.000000  117057.000000             117057.000000
mean        0.586673       0.029065                  0.028903
std         0.283482       0.032680                  0.035277
min         0.000000       0.000000                  0.000000
25%         0.391304       0.010951                  0.011600
50%         0.608696       0.018783                  0.021824
75%         0.826087       0.034122                  0.034437
max         1.000000       1.000000                  1.000000

       distance_to_world_trade_center  distance_to_jfk_airport  \
count                   117057.000000            117057.000000
mean                         0.053999                 0.210914
std                          0.037010                 0.034585
min                          0.000000                 0.002688
25%                          0.032046                 0.207882
50%                          0.050121                 0.214301
75%                          0.068509                 0.221314
max                          0.988776                 0.983660

       distance_to_lga_airport  distance_to_ewr_airport  \
count            117057.000000            117057.000000
mean                  0.096314                 0.147219
std                   0.031127                 0.031123
min                   0.001425                 0.000000
25%                   0.083196                 0.130887
50%                   0.095573                 0.142752
75%                   0.109843                 0.156160
max                   1.000000                 0.989328

       distance_to_statue_of_liberty  distance_to_central_park
count                  117057.000000             117057.000000
mean                        0.077131                  0.043587
std                         0.035355                  0.036580
min                         0.000000                  0.000000
25%                         0.056181                  0.022381
50%                         0.073832                  0.037190
75%                         0.091811                  0.056999
max                         0.976679                  1.000000
```

5.7 Encoding Categorical Data

Since machine learning models can only be trained with numeric data, we need to convert categorical variables into numbers. One-hot encoding involves creating a new binary (0/1) column for each unique category in a categorical column. In this project, we applied one-hot encoding to the **`Weekday`** column to represent days of the week as binary vectors.

[45]:
```
encoder = OneHotEncoder(sparse_output=False, handle_unknown='ignore').
↪fit(df[categorical_cols])
```

[46]:
```
encoded_cols = list(encoder.get_feature_names_out(categorical_cols))
print(encoded_cols)
```

```
['DayOfWeek_0', 'DayOfWeek_1', 'DayOfWeek_2', 'DayOfWeek_3', 'DayOfWeek_4',
'DayOfWeek_5', 'DayOfWeek_6']
```

[47]:
```
train_inputs[encoded_cols] = encoder.transform(train_inputs[categorical_cols])
val_inputs[encoded_cols] = encoder.transform(val_inputs[categorical_cols])
test_inputs[encoded_cols] = encoder.transform(test_inputs[categorical_cols])
```

verify that the data was loaded properly

[48]:
```
print('train_inputs:', train_inputs.shape)
print('train_targets:', train_targets.shape)
print('val_inputs:', val_inputs.shape)
print('val_targets:', val_targets.shape)
print('test_inputs:', test_inputs.shape)
print('test_targets:', test_targets.shape)
```

```
train_inputs: (117057, 25)
train_targets: (117057,)
val_inputs: (39020, 25)
val_targets: (39020,)
test_inputs: (39020, 25)
test_targets: (39020,)
```

6 Train and evaluate different models

In this step, we apply several regression algorithms to our processed dataset. Each model is trained on the training set and evaluated on the validation set using a consistent set of metrics (RMSE and R^2). By comparing these models, we can identify which approaches are most promising before proceeding to hyperparameter tuning.

[49]:
```
def evaluate(model):
    """
    Evaluate the given model on training and validation sets.

    This function uses the trained model to generate predictions for both
    the training and validation datasets. It then computes the Root Mean␣
↪Squared Error (RMSE)
    and R² score for both sets, and returns these metrics in a dictionary.

    Parameters
    ----------
    model : estimator
```

```
        A trained scikit-learn estimator that supports the predict method.

    Returns
    -------
    dict
        A dictionary containing:
        - 'train_rmse' : float
            RMSE of the model on the training set.
        - 'train_r2' : float
            R² score of the model on the training set.
        - 'val_rmse' : float
            RMSE of the model on the validation set.
        - 'val_r2' : float
            R² score of the model on the validation set.
    """
    train_preds = model.predict(train_inputs)
    val_preds = model.predict(val_inputs)

    train_rmse = root_mean_squared_error(train_targets, train_preds)
    train_r2 = r2_score(train_targets, train_preds)
    val_rmse = root_mean_squared_error(val_targets, val_preds)
    val_r2 = r2_score(val_targets, val_preds)

    return {
        'train_rmse': round(float(train_rmse), 6),
        'train_r2': round(train_r2, 6),
        'val_rmse': round(float(val_rmse), 6),
        'val_r2': round(val_r2, 6)
    }
```

6.1 Linear regression

[50]:
```
model1 = LinearRegression().fit(train_inputs, train_targets)
```

[51]:
```
result1 = evaluate(model1)
```

6.2 Ridge regression

[52]:
```
model2 = Ridge(random_state=42).fit(train_inputs, train_targets)
```

[53]:
```
result2 = evaluate(model2)
```

6.3 ElasticNet

[54]:
```
model3 = ElasticNet(random_state=42).fit(train_inputs, train_targets)
```

[55]:
```
result3 = evaluate(model3)
```

6.4 Decision tree regressor

[56]:
```
model4 = DecisionTreeRegressor(max_depth=10, random_state=42).fit(train_inputs,␣
↪train_targets)
```

[57]:
```
result4 = evaluate(model4)
```

6.5 Random forest

[58]:
```
model5 = RandomForestRegressor(max_depth=10, n_jobs=-1, random_state=42).
↪fit(train_inputs, train_targets)
```

[59]:
```
result5 = evaluate(model5)
```

6.6 Gradient boosting

[60]:
```
model6 = XGBRegressor(random_state=42, n_jobs=-1, objective='reg:squarederror').
↪fit(train_inputs, train_targets)
```

[61]:
```
result6 = evaluate(model6)
```

6.7 Results

[62]:
```
results = pd.DataFrame(
    data={
        'Linear Regression': result1.values(),
        'Ridge Regression': result2.values(),
        'ElasticNet': result3.values(),
        'Decision Tree Regresor': result4.values(),
        'Random Forest': result5.values(),
        'Gradient Boosting': result6.values()
    },
    index=result1.keys()
)
```

[63]:
```
results
```

[63]:
```
            Linear Regression  Ridge Regression  ElasticNet  \
train_rmse           5.409391          5.424880    9.955497
train_r2             0.704763          0.703070    0.000000
val_rmse             5.013715          5.030713    9.484227
val_r2               0.720473          0.718574   -0.000250

            Decision Tree Regresor  Random Forest  Gradient Boosting
train_rmse                3.678757       3.434792           2.726046
train_r2                  0.863455       0.880965           0.925021
val_rmse                  4.219684       3.857935           3.781239
val_r2                    0.802000       0.834494           0.841009
```

6.8 Conclusion

After comparing various models, we observe that tree-based methods (Decision Tree, Random Forest, XGBoost) generally outperform linear regression models and their variants, particularly in terms of RMSE and R^2 on the validation set. Among these, Random Forest and XGBoost stand out, achieving the lowest errors and highest R^2 scores. These models appear to be a solid starting point for further hyperparameter tuning to improve prediction quality.

7 Tune hyperparameters

Hyperparameter tuning aims to optimize model performance by identifying the best combination of hyperparameters. In this step, we focus on improving the XGBoost model, which demonstrated the best results in previous evaluations. By carefully adjusting parameters such as learning rate, number of estimators, and tree depth, we aim to enhance model accuracy and reduce validation errors. Through iterative testing and analysis, we will identify the optimal parameter configuration that yields the best predictive performance.

[64]:
```python
def test_params(ModelType, **params):
    model = ModelType(**params).fit(train_inputs, train_targets)
    train_rmse = root_mean_squared_error(model.predict(train_inputs),␣
 ↪train_targets)
    val_rmse = root_mean_squared_error(model.predict(val_inputs), val_targets)
    return train_rmse, val_rmse

def test_param_and_plot(ModelType, param_name, param_values, **other_params):
    train_errors = []
    val_errors = []
    for value in param_values:
        params = dict(other_params)
        params[param_name] = value
        train_rmse, val_rmse = test_params(ModelType, **params)
        train_errors.append(train_rmse)
        val_errors.append(val_rmse)

    plt.figure(figsize=(6,6))
    plt.plot(param_values, train_errors, 'b-o')
    plt.plot(param_values, val_errors, 'r-o')
    plt.title('Overfitting curve: ' + param_name)
    plt.xlabel(param_name)
    plt.ylabel('RMSE')
    plt.legend(['Training', 'Validation'])
```

Below are the baseline hyperparameters for the XGBoost model, inherited from our previous best-performing setup. As we progress, we will incorporate the newly identified optimal values into this dictionary.

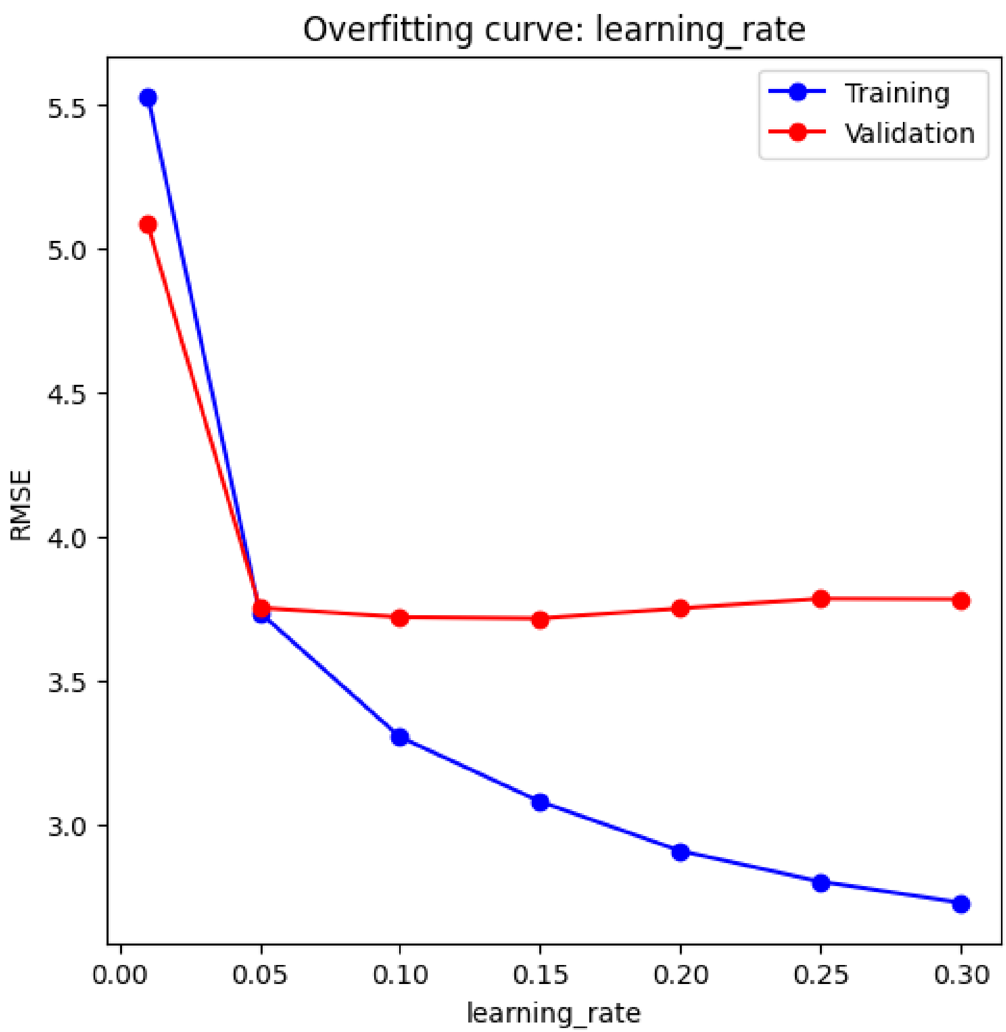
[65]:
```python
standard_params = {
    'random_state': 42,
```

```
    'n_jobs': 1,
    'objective': 'reg:squarederror'
}
```

7.1 learning_rate

[67]:
```
%%time
test_param_and_plot(XGBRegressor, 'learning_rate', [0.01, 0.05, 0.1, 0.15, 0.2,␣
 ↪0.25, 0.3], **standard_params)
```

```
CPU times: total: 15.6 s
Wall time: 15.4 s
```

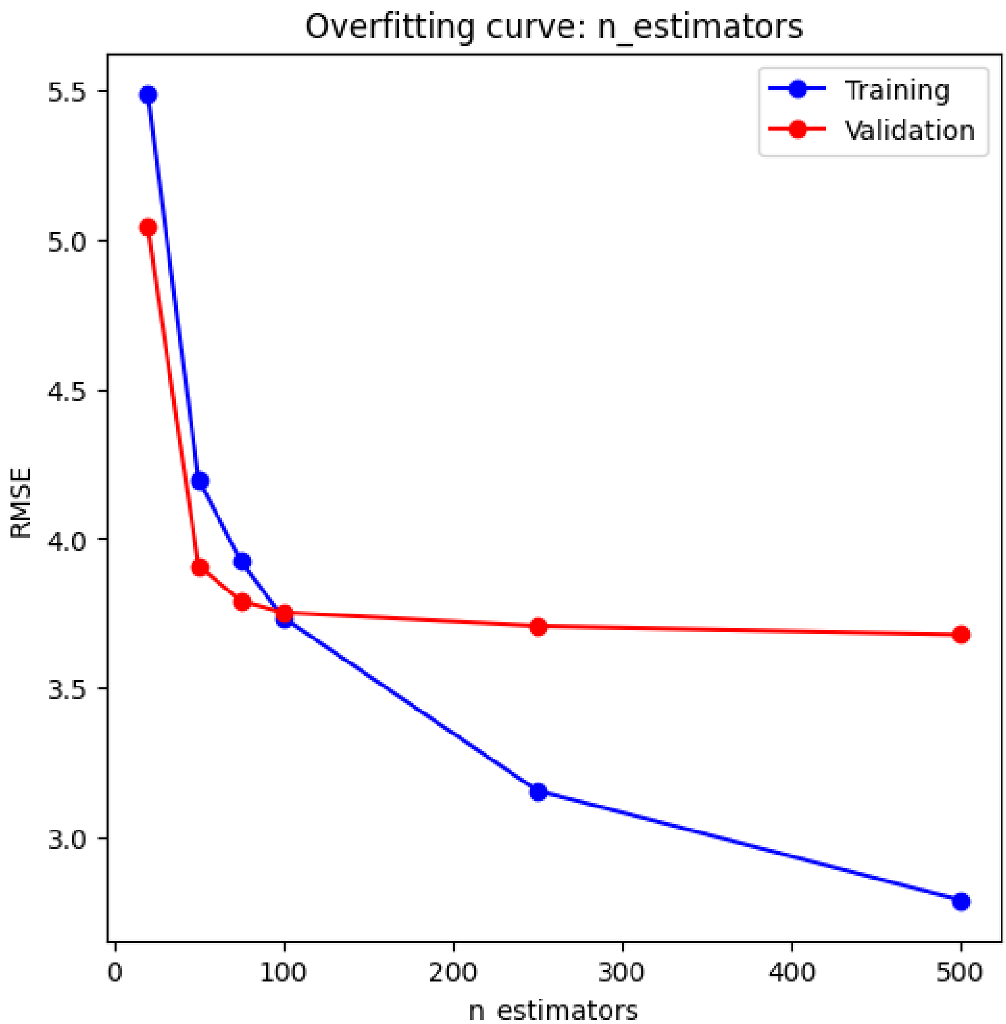


[67]:
```
standard_params['learning_rate'] = 0.05
```

7.2 n_estimators

[68]:
```
%%time
test_param_and_plot(XGBRegressor, 'n_estimators', [20, 50, 75, 100, 250, 500],
 ↪**standard_params)
```

```
CPU times: total: 18.2 s
Wall time: 18 s
```

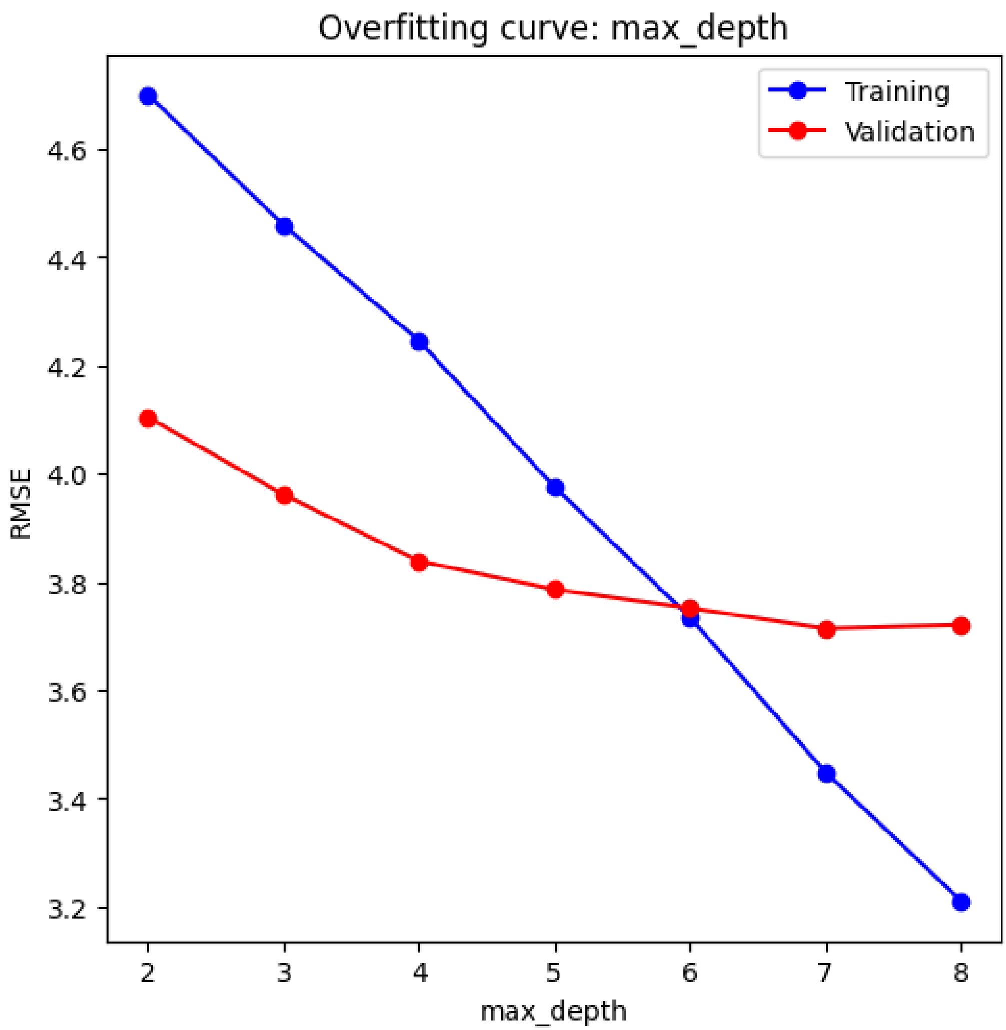


[69]:
```
standard_params['n_estimators'] = 100
```

7.3 max_depth

[70]:
```
%%time
test_param_and_plot(XGBRegressor, 'max_depth', [2, 3, 4, 5, 6, 7, 8],␣
↪**standard_params)
```

```
CPU times: total: 13.8 s
Wall time: 13.6 s
```

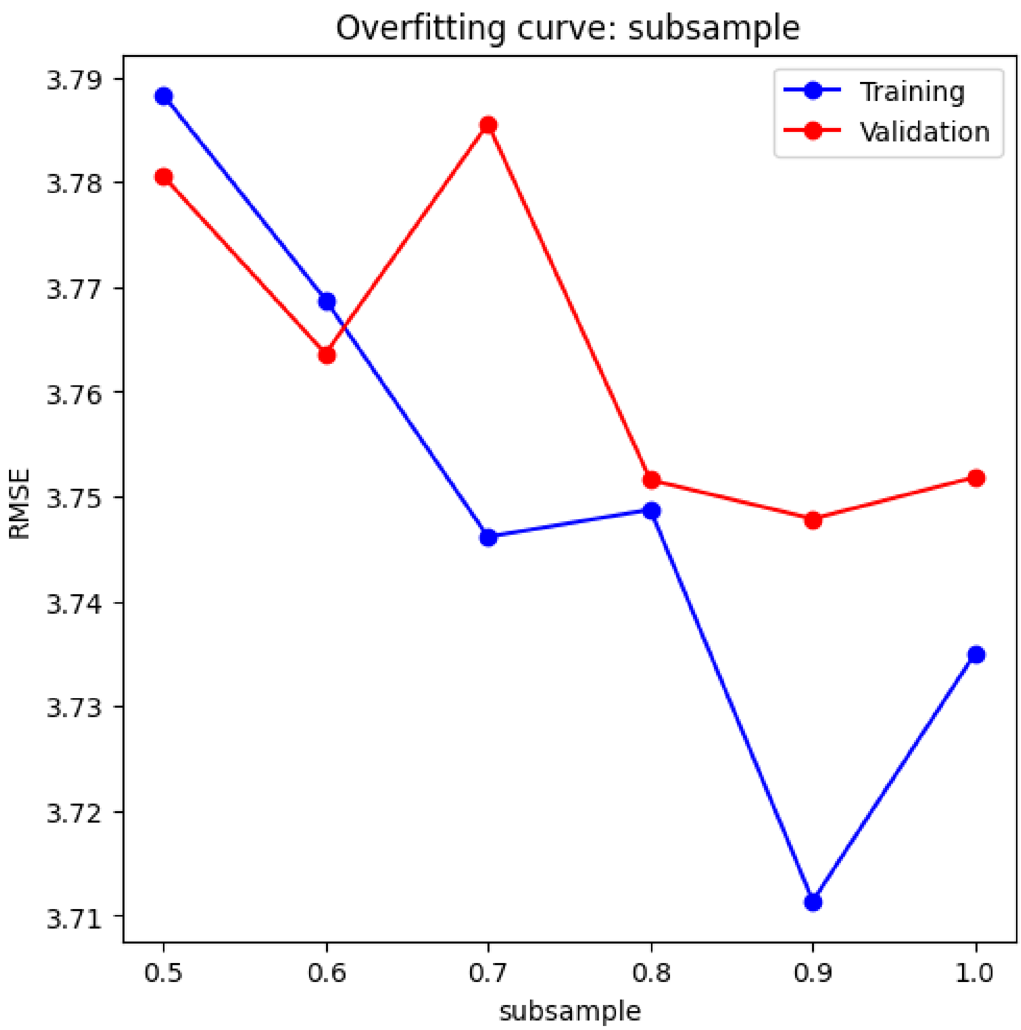


[71]:
```
standard_params['max_depth'] = 6
```

7.4 subsample

[72]:
```
%%time
test_param_and_plot(XGBRegressor, 'subsample', [0.5, 0.6, 0.7, 0.8, 0.9, 1.0],␣
↪**standard_params)
```

```
CPU times: total: 14.5 s
Wall time: 14.1 s
```

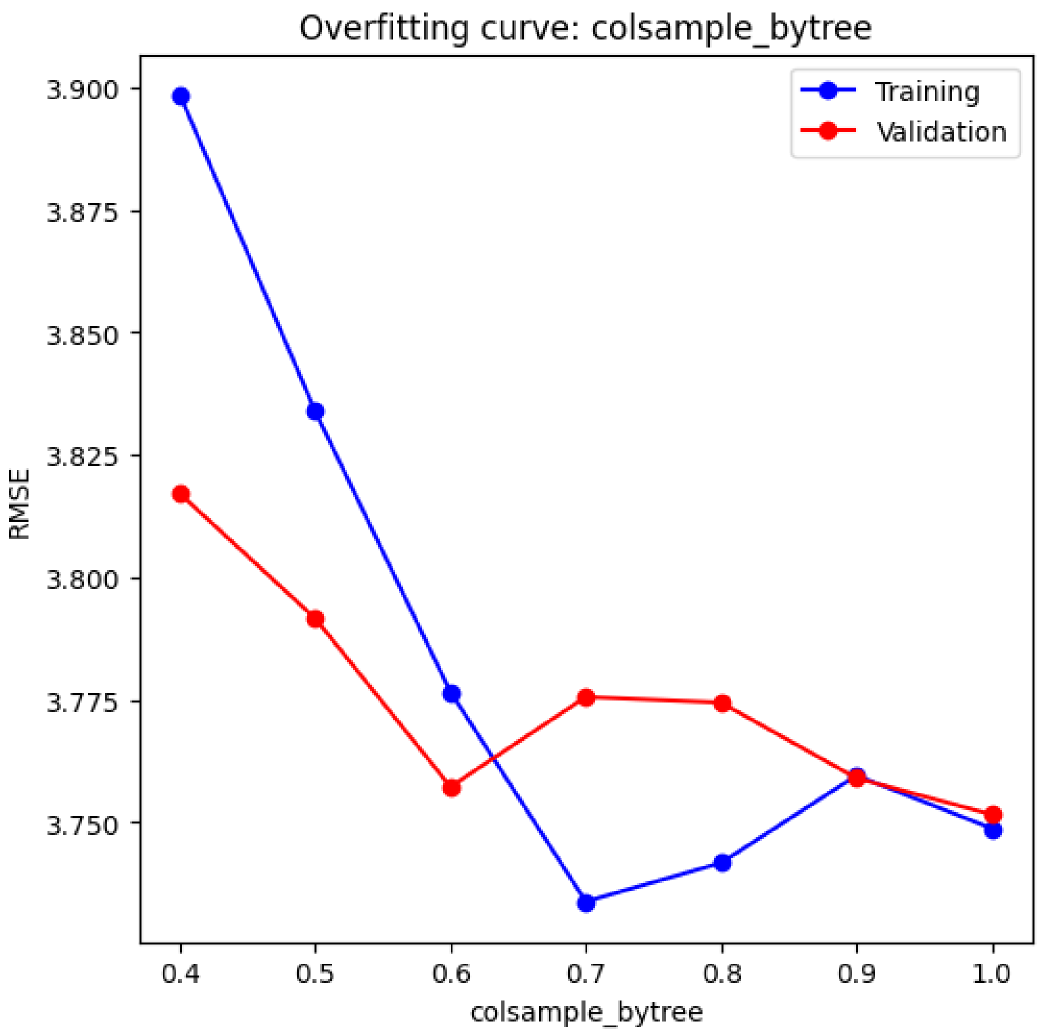


[73]:
```
standard_params['subsample'] = 0.8
```

7.5 colsample_bytree

[74]:
```
%%time
test_param_and_plot(XGBRegressor, 'colsample_bytree', [0.4, 0.5, 0.6, 0.7, 0.8,␣
↪0.9, 1.0], **standard_params)
```

```
CPU times: total: 16.6 s
Wall time: 16 s
```

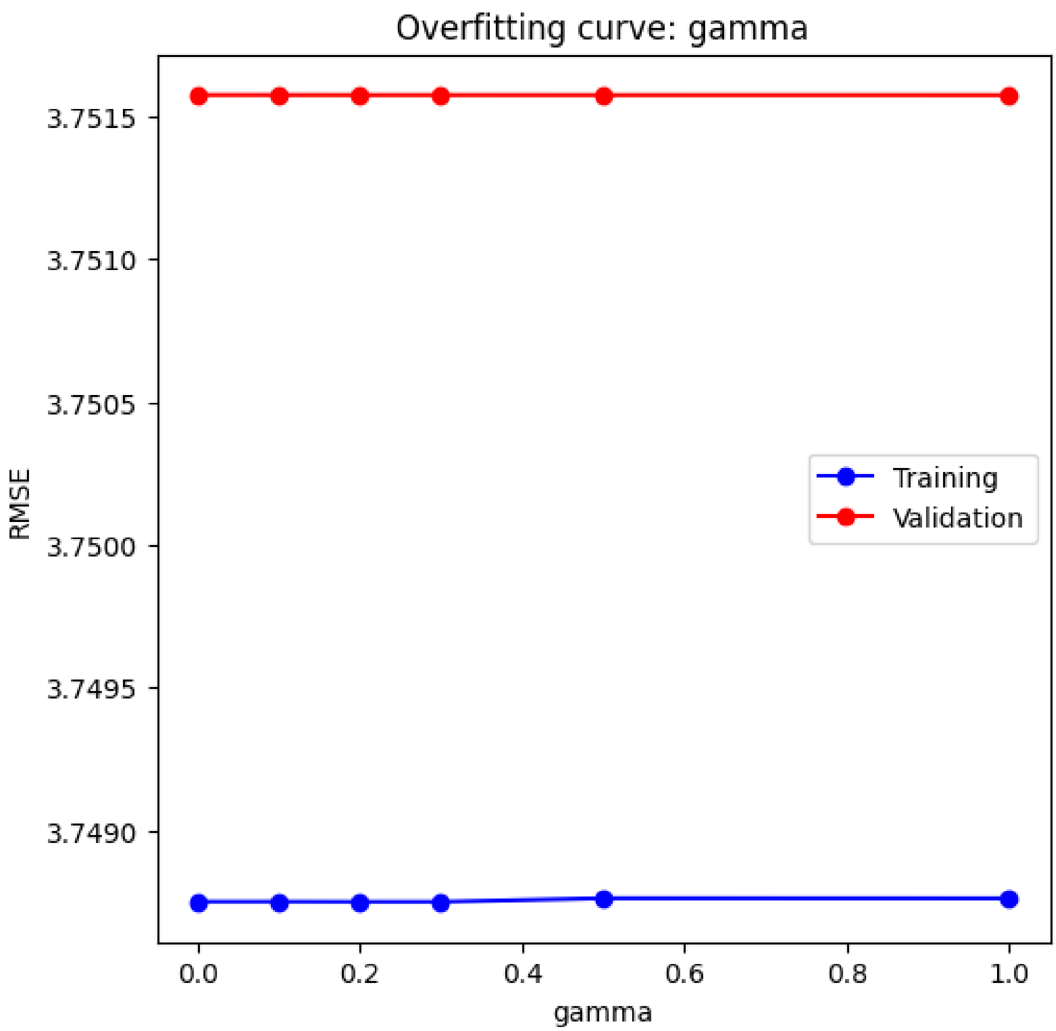


[75]:
```
standard_params['colsample_bytree'] = 1
```

7.6 gamma

[76]:
```
%%time
test_param_and_plot(XGBRegressor, 'gamma', [0, 0.1, 0.2, 0.3, 0.5, 1],␣
↪**standard_params)
```

```
CPU times: total: 15.2 s
Wall time: 14.7 s
```

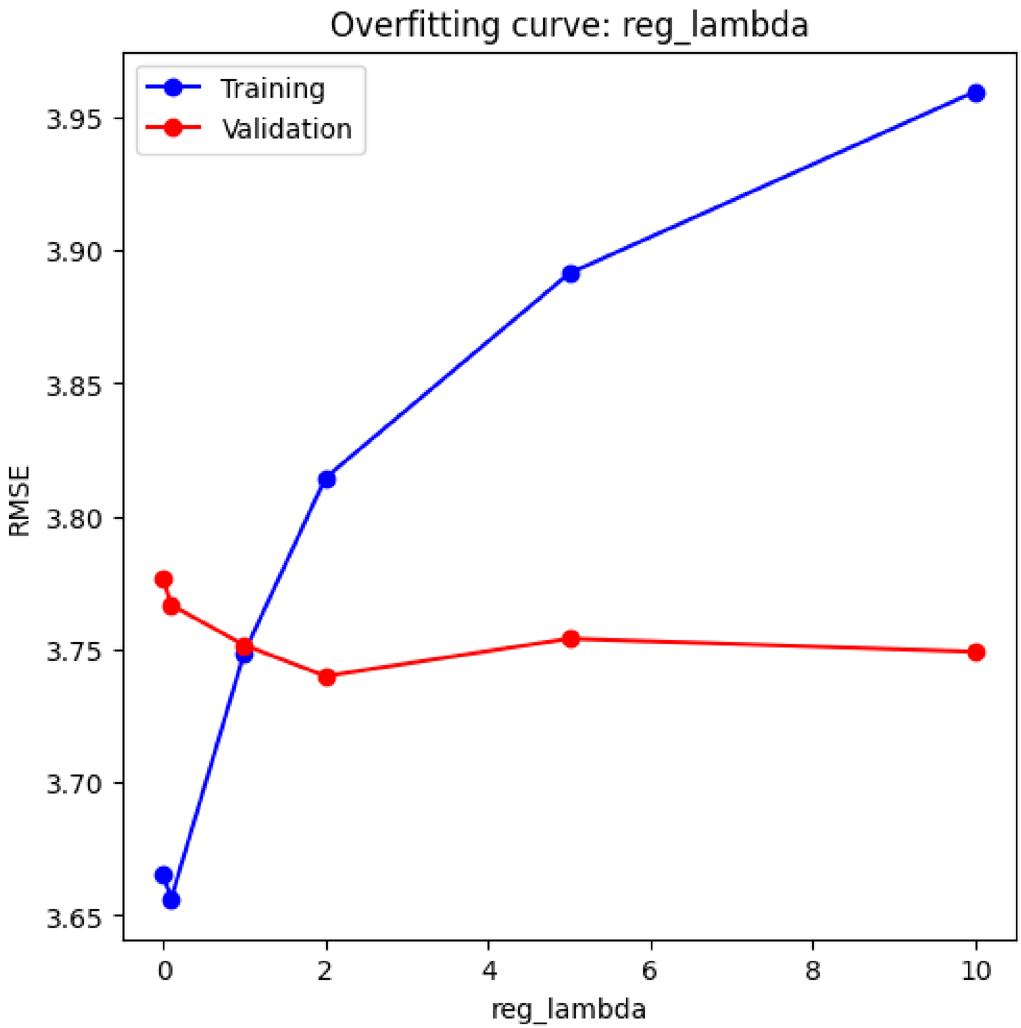


[77]:
```
standard_params['gamma'] = 0
```

7.7 reg_lambda

[78]:
```
%%time
test_param_and_plot(XGBRegressor, 'reg_lambda', [0, 0.1, 1, 2, 5, 10],␣
↪**standard_params)
```

```
CPU times: total: 14.8 s
Wall time: 14.3 s
```

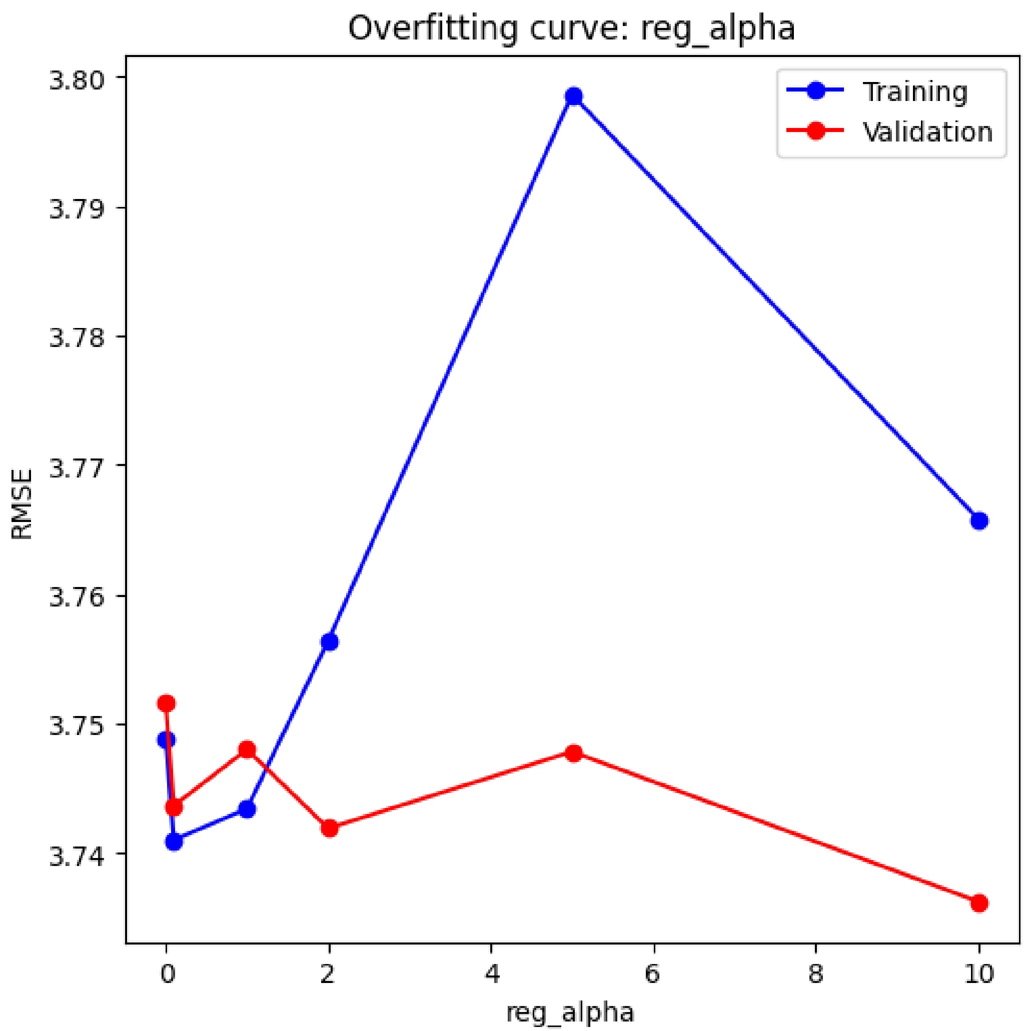


[79]:
```
standard_params['reg_lambda'] = 1
```

7.8 reg_alpha

[80]:
```
%%time
test_param_and_plot(XGBRegressor, 'reg_alpha', [0, 0.1, 1, 2, 5, 10],␣
↪**standard_params)
```

```
CPU times: total: 14.9 s
Wall time: 14.3 s
```



[81]:
```
standard_params['reg_alpha'] = 0.1
```

7.9 hyperparameters selection

The hyperparameters listed below are the result of our step-by-step tuning process. They represent the best combination found so far, after multiple rounds of testing and refinement.

[82]:
```
standard_params
```

[82]:
```
{'random_state': 42,
 'n_jobs': 1,
 'objective': 'reg:squarederror',
 'learning_rate': 0.05,
 'n_estimators': 100,
 'max_depth': 6,
 'subsample': 0.8,
 'colsample_bytree': 1,
 'gamma': 0,
 'reg_lambda': 1,
 'reg_alpha': 0.1}
```

[83]:
```
model7 = XGBRegressor(**standard_params).fit(train_inputs, train_targets)
```

[84]:
```
result7 = evaluate(model7)
```

[85]:
```
result7 = pd.DataFrame(result7.values(), columns=['XGBoost Tuning'],␣
↪index=result7.keys())

results = pd.concat([results, result7], axis=1)
```

[86]:
```
results
```

[86]:
```
            Linear Regression  Ridge Regression  ElasticNet  \
train_rmse           5.409391          5.424880    9.955497
train_r2             0.704763          0.703070    0.000000
val_rmse             5.013715          5.030713    9.484227
val_r2               0.720473          0.718574   -0.000250

            Decision Tree Regresor  Random Forest  Gradient Boosting  \
train_rmse                3.678757       3.434792           2.726046
train_r2                  0.863455       0.880965           0.925021
val_rmse                  4.219684       3.857935           3.781239
val_r2                    0.802000       0.834494           0.841009

            XGBoost Tuning
train_rmse        3.741015
train_r2          0.858794
val_rmse          3.743664
val_r2            0.844153
```

Evaluating the final XGBoost model shows only a slight improvement over the baseline gradient boosting approach. Therefore, i will now attempt a more systematic approach using GridSearchCV to further refine the hyperparameters and potentially achieve a more significant performance gain.

7.10 GridSearchCV

I'll apply GridSearchCV to carefully investigate a targeted selection of hyperparameters, building on the insights gained so far. This systematic search will help us refine our parameter choices and potentially boost the model's predictive accuracy beyond what we've achieved with manual tuning.

[87]:
```
param_grid = {
    'learning_rate': [0.05, 0.06, 0.07],
    'n_estimators': [330, 350, 400],
    'max_depth': [6, 7],
    'subsample': [0.9, 0.95, 1.0],
    'colsample_bytree': [0.75, 0.8, 0.85],
}
```

[88]:
```
model = XGBRegressor(random_state=42)

grid_search = GridSearchCV(
    estimator=model,
    param_grid=param_grid,
    scoring='neg_root_mean_squared_error',
    cv=3,
    verbose=2,
    n_jobs=-1
)
```

[89]:
```
grid_search.fit(train_inputs, train_targets)

print("Best params:", grid_search.best_params_)
print("Best score:", -grid_search.best_score_)
```

```
Fitting 3 folds for each of 162 candidates, totalling 486 fits
Best params: {'colsample_bytree': 0.8, 'learning_rate': 0.05, 'max_depth': 7,
'n_estimators': 400, 'subsample': 0.9}
Best score: 4.194516400532325
```

[90]:
```
param_grid = grid_search.best_params_
```

[91]:
```
final = XGBRegressor(random_state=42, **param_grid).fit(train_inputs,␣
 ↪train_targets)
```

[92]:
```
result8 = evaluate(final)
```

[93]:
```
result8 = pd.DataFrame(result8.values(), columns=['GridSearchCV'],␣
 ↪index=result8.keys())

results = pd.concat([results, result8], axis=1)
```

[94]:
```
results
```

[94]:

	Linear Regression	Ridge Regression	ElasticNet
train_rmse	5.409391	5.424880	9.955497
train_r2	0.704763	0.703070	0.000000
val_rmse	5.013715	5.030713	9.484227
val_r2	0.720473	0.718574	-0.000250

	Decision Tree Regresor	Random Forest	Gradient Boosting
train_rmse	3.678757	3.434792	2.726046
train_r2	0.863455	0.880965	0.925021
val_rmse	4.219684	3.857935	3.781239
val_r2	0.802000	0.834494	0.841009

	XGBoost Tuning	GridSearchCV
train_rmse	3.741015	2.547374
train_r2	0.858794	0.934527
val_rmse	3.743664	3.638614
val_r2	0.844153	0.852777

After applying GridSearchCV to optimize the hyperparameters of the XGBoost model, there was a modest improvement in performance compared to earlier methods.

8 Model testing

[95]:
```
test_preds = final.predict(test_inputs)
test_rmse = root_mean_squared_error(test_targets, test_preds)
test_r2 = r2_score(test_targets, test_preds)

print("Test RMSE:", test_rmse)
print("Test R²:", test_r2)
```

```
Test RMSE: 3.8053163400228147
Test R²: 0.8446119720212818
```

The final evaluation of the XGBoost model on the unseen test dataset yielded the following results:

- Test RMSE: **3.8053**
- Test R^2: **0.8446**

These results are consistent with the validation performance, confirming that the model generalizes well to new data. The relatively low RMSE and high R^2 indicate that the model effectively captures the underlying patterns in the data while maintaining good predictive accuracy.

9 Summary and insights

Data Understanding and Preparation: Initial EDA and feature engineering steps were key to uncovering patterns (e.g., the impact of distance on fare) and refining the dataset for modeling.

Model Selection and Comparison: While baseline models like Linear Regression provided a starting point, tree-based methods—especially Gradient Boosting—offered superior predictive performance.

Hyperparameter Tuning: Systematic tuning (via GridSearchCV) further improved the Gradient Boosting model, finding a balanced set of parameters that enhanced accuracy and reduced overfitting.

Robust Generalization: Consistent results across validation and test sets confirmed the model's reliability and suitability for real-world fare predictions.

A clear, stepwise workflow ensured each stage contributed to overall improvements, ultimately leading to a well-tuned, high-performing model.