p2

October 24, 2024

1 P2 - Statistics + Visualization

This project is intended to give you experience with statistics and working with a large data set.

1.1 Project Setup

You should use the following modules in this assignment and not need any additional modules.

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import datetime
import os
import re
import platform
import sys
import importlib
from packaging.version import Version, parse

import otter
grader = otter.Notebook()
```

2 Flight Delays

Suppose the Houghton County Airport (CMX) is ready to renew its passenger airplane contract. Airport officials and interested passengers would like to select an airline and connecting city that has reliable service. *Currently, CMX is being served by United Airlines through Chicago-O'hare (ORD); it has been served by Delta in the past with connections to Minneapolis-St. Paul (MSP).*

In order to make an informed decision, you have been tasked to look at flight delay statistics for two potential connector airports:

- Chicago - O'hare (ORD)
- Detroit (DTW)

The data you are provided comes from the US Department of Transportation's Bureau of Transportation Statistics (BTS). In particular, it comes from the Reporting Carrier On-Time

Performance data tables:
https://www.transtats.bts.gov/TableInfo.asp?gnoyr_VQ=FGJ&QO_fu146_anzr=b0-gvzr&V0s1_b0yB=D

Information on the variables can be found in the Field Information:
https://www.transtats.bts.gov/Fields.asp?gnoyr_VQ=FGJ

The data is structured with the following elements:

- "YEAR"
- "MONTH"
- "DAY_OF_MONTH"
- "OP_UNIQUE_CARRIER" - Unique Carrier Code
- "TAIL_NUM" - Tail number
- "OP_CARRIER_FL_NUM" - Flight number
- "ORIGIN" - Origin Airport code
- "DEST" - Destination Airport code
- "CRS_DEP_TIME" - Computer Reservation System Departure Time (local time: hhmm)
- "DEP_TIME" - Actual Departure Time (local time: hhmm)
- "DEP_NEXT_DAY" - the flight departed on the: (scheduled day = 0), (next day = 1), (following day = 2), or (day before = -1)
- "CRS_ARR_TIME" - Computer Reservation System Arrival Time (local time: hhmm)
- "ARR_TIME" - Actual Arrival Time
- "ARR_NEXT_DAY" - the flight arrived on the: (scheduled day = 0), (next day = 1), (following day = 2), or (day before = -1)
- "CANCELLED" - Cancelled Flight Indicator (1 = Yes)
- "DIVERTED" - Diverted Flight Indicator (1 = Yes)

You have access to 2 years of data Jan 2022 - Dec 2023.

The data has also been restricted to several major carriers and their regional services:

Airline Code	Airline Name	Regional Partner
UA	United Airlines	NA
DL	Delta Airlines	NA
OO	Skywest Airlines	United
9E	Endeavor Air	Delta

2.1 Access the Data

The data is available on Kaggle

Once you download the data make sure it is saved in your project directory:

```
class-folder/
    p2/
        flight-final.csv
        img/
        p2.ipynb
```

2.2 Q0 - Setup

The following code checks to make sure your notebook is running on Gradescope, the cloud, or linux lab machines.

While you may develop on your own machines or places like Colab or Github, you should run your final submission on the campus Linux machines using the `un5550fa24` environment.

Type your answer here, replacing this text.

[2]:
```
# DO NOT CHANGE THIS CODE

# flag if notebook is running on Gradescope
if re.search(r'amzn', platform.uname().release):
    GS = True
else:
    GS = False

# flag if notebook is running on Colaboratory
try:
  import google.colab
  COLAB = True
except:
  COLAB = False

# flag if running on Linux lab machines.
cname = platform.uname().node
if re.search(r'(guardian|colossus|c28|coc-15954-m)', cname):
    LLM = True
else:
    LLM = False

print("System: GS - %s, COLAB - %s, LLM - %s" % (GS, COLAB, LLM))
```

```
System: GS - False, COLAB - False, LLM - False
```

[3]:
```
# DO NOT CHANGE THIS CODE
pver = sys.version
print(pver)
```

```
3.10.11 | packaged by conda-forge | (main, May 10 2023, 18:58:44) [GCC 11.3.0]
```

[4]:
```
# DO NOT CHANGE THIS CODE
env2 =!conda info | grep 'active env'
print(env2)
```

```
['     active environment : None']
```

```
[5]: # DO NOT CHANGE THIS CODE
OK = '\x1b[42m[ OK ]\x1b[0m'
FAIL = "x1b[41m[FAIL]\x1b[0m"

def import_version(pkg, req_ver, fail_msg=""):
    mod = None
    try:
        mod = importlib.import_module(pkg)
        ver = mod.__version__
        if Version(ver) != req_ver:
            print(FAIL, "%s version %s required, but %s installed."
                  % (lib, req_ver, ver))
        else:
            print(OK, '%s version %s' % (pkg, ver))
    except ImportError:
        print(FAIL, '%s not installed. %s' % (pkg, fail_msg))
    return (mod, Version(ver), req_ver)

requirements = {'numpy': parse("2.1.0"), 'scipy': parse("1.14.1"),
                'matplotlib': parse("3.9.2"), 'pandas': parse("2.2.2"),
                'otter': parse("5.7.0"), 'seaborn': parse('0.13.2'),
                'dill': parse('0.3.7'), 'sklearn': parse("1.5.1")
 }


pks = []
for lib, required_version in list(requirements.items()):
    pks.append(import_version(lib, required_version))
```

```
x1b[41m[FAIL] numpy version 2.1.0 required, but 1.24.3 installed.
x1b[41m[FAIL] scipy version 1.14.1 required, but 1.10.1 installed.
x1b[41m[FAIL] matplotlib version 3.9.2 required, but 3.7.1 installed.
x1b[41m[FAIL] pandas version 2.2.2 required, but 2.0.1 installed.
x1b[41m[FAIL] otter version 5.7.0 required, but 4.3.2 installed.
x1b[41m[FAIL] seaborn version 0.13.2 required, but 0.12.2 installed.
x1b[41m[FAIL] dill version 0.3.7 required, but 0.3.6 installed.
x1b[41m[FAIL] sklearn version 1.5.1 required, but 1.2.2 installed.
```

2.3 Q1 - Load Data

Read in the data.

Set the column names to be: `year`, `month`, `day`, `carrier`, `tailNum`, `flNum`, `origin`, `dest`, `crsDepTime`, `actDepTime`, `depNextDay`, `crsArrTime`, `actArrTime`, `arrNextDay`, `cancelled`, `diverted`

You do not need to do any special preprocessing or changing of the data types, we will be during specific processing of the data in the coming questions.

[6]:
```python
# Read in data, set DataFrame columns to names above
column_names = [
    'year', 'month', 'day', 'carrier', 'tailNum', 'flNum', 'origin', 'dest',
    'crsDepTime', 'actDepTime', 'depNextDay', 'crsArrTime', 'actArrTime',
    'arrNextDay', 'cancelled', 'diverted'
]

flights = pd.read_csv("flight-final.csv", header=None, names=column_names)

flights.head()
```

[6]:
```
   year  month  day carrier tailNum  flNum origin dest  crsDepTime  \
0  2022     12    1      9E  N133EV   4774    DTW  TYS        1745
1  2022     12    1      9E  N133EV   4915    DSM  DTW         620
2  2022     12    1      9E  N133EV   5015    DSM  DTW        1405
3  2022     12    1      9E  N133EV   5042    DTW  DSM        1215
4  2022     12    1      9E  N134EV   4883    DTW  DSM        1950

   actDepTime  depNextDay  crsArrTime  actArrTime  arrNextDay  cancelled  \
0      1915.0           0        1927      2038.0           0        0.0
1       616.0           0         909       902.0           0        0.0
2      1401.0           0        1651      1635.0           0        0.0
3      1207.0           0        1320      1259.0           0        0.0
4      1958.0           0        2054      2051.0           0        0.0

   diverted
0       0.0
1       0.0
2       0.0
3       0.0
4       0.0
```

[7]:
```python
grader.check("q1")
```

[7]:
```
q1 results: All test cases passed!
```

2.4 Aside : Style

In the following questions you are going to be chaining together many expressions, perhaps creating code that is longer than 79 characters long (the maximum suggested by the PEP-8 style guides).

`The preferred way of wrapping long lines is by using Python's implied line continuation inside parentheses, brackets and braces. Long lines can be broken over multiple lines by wrapping expressions in parentheses. These should be used in preference to using a backslash for line continuation.`

Using parenthesis surrounding code make it possible to break the code into multiple lines for readability.

Here are some examples:

[8]:
```
temp = (flights[(flights['dest'] == 'CMX') |
                (flights['origin'] == 'CMX')])
```

[9]:
```
temp = (
    flights[(flights['dest'] == 'CMX') |
            (flights['origin'] == 'CMX')]
)
```

[10]:
```
temp = (flights
        .groupby('month')['diverted']
        .sum())
```

[11]:
```
temp = (
    flights
    .groupby('month')['diverted']
    .sum()
)
```

2.5 Q2 - Explore Data

Let's start to explore the data.

[12]:
```
print("Flight data: %8d rows,  %d columns" % (flights.shape[0], flights.
 ↪shape[1]))
```

```
Flight data:   919840 rows,  16 columns
```

We want to remove flights that were cancelled or diverted from the rest of our analysis.

But, first let's examine cancelled or diverted flights.

2.5.1 Q2a - Diverted and Cancelled Flights

You will need to determine the number of diverted flights, `num_divert` and the number of cancelled flights, `num_cancel`.

Next, determine the destination airport(s), by their code, with the highest proportion of diverted and cancelled flights `airport_highest_prop_divert` and `airport_highest_prop_cancel`.

[13]:
```
# num_divert - number of flights that were diverted.
# num_cancel - number of flights that were cancelled
num_divert = int(flights['diverted'].sum())
num_cancel = int(flights['cancelled'].sum())
print('Num. of diverted flights: %7d\nNum. of cancelled flights: %7d\n' % \
 (num_divert, num_cancel))
```

```
Num. of diverted flights:    1872
Num. of cancelled flights:   15264
```

```
[14]: airport_highest_prop_divert = (flights[flights['diverted'] == 1]['dest'].
       ↪value_counts() / flights['dest'].value_counts()).idxmax()
      airport_highest_prop_cancel = (flights[flights['cancelled'] == 1]['dest'].
       ↪value_counts() / flights['dest'].value_counts()).idxmax()



      print('Airport with highest proportion of flights diverted:  %s' %␣
       ↪airport_highest_prop_divert)
      print('Airport with highest proportion of flights cancelled: %s' %␣
       ↪airport_highest_prop_cancel)
```

```
Airport with highest proportion of flights diverted:  ASE
Airport with highest proportion of flights cancelled: CMX
```

```
[15]: grader.check("q2a")
```

```
[15]: q2a results: All test cases passed!
```

Do these values make sense? You can lookup airport codes on Google.

Note the local airport in Houghton / Hancock is `CMX`.

2.5.2 Q2b - Filter Cancelled and Diverted Flights

Now, let's remove the cancelled and diverted flights from further analysis.

After removing both flights, reset the index on the remaining flights in the `flights` DataFrame.

```
[16]: # Remove the cancelled and diverted flights.
      # Then, reset the index of the flights DataFrame
      flights = (flights[(flights['cancelled'] == 0) & (flights['diverted'] == 0)].
       ↪reset_index(drop=True))
      print("Flight data: %8d rows" % flights.shape[0])
```

```
Flight data:   902704 rows
```

```
[17]: grader.check("q2b")
```

```
[17]: q2b results: All test cases passed!
```

2.5.3 Q2c - Flights at `DTW` and `ORD`

Report the number of flights departing and arriving from each of the two airports under study: `DTW` and `ORD`.

[18]:
```
# numArrDTW - number of flights arriving at DTW
numArrDTW = flights[flights['dest'] == 'DTW'].shape[0]
# numDepDTW - number of flights departing DTW
numDepDTW = flights[flights['origin'] == 'DTW'].shape[0]
print("DTW flights: %7d arrivals, %7d departures" % (numArrDTW, numDepDTW))
```

```
DTW flights:  192689 arrivals,  192578 departures
```

[19]:
```
# numArrORD - number of flights arriving at ORD
numArrORD = flights[flights['dest'] == 'ORD'].shape[0]
# numDepORD - number of flights departing ORD
numDepORD = flights[flights['origin'] == 'ORD'].shape[0]
print("ORD flights: %7d arrivals, %7d departures" % (numArrORD, numDepORD))
```

```
ORD flights:  266762 arrivals,  266971 departures
```

Note: The number of flights arriving and departing from the two airports exceeds the total number of flights, because flights between the 2 airports are counted twice.

[20]:
```
grader.check("q2c")
```

[20]:
```
q2c results: All test cases passed!
```

2.6 Q3 - Time Information and Flight Delays.

Both the departure and arrival times were read in as integers or floating-point numbers in local time format: `hhmm`.

2.6.1 Question 3a - Extract Hour and Minutes

Write two functions, `extract_hour` and `extract_mins` that converts the local time to hours and minutes, respectively.

Hint: You may want to use modular arithmetic and integer division.

Notes:

- The function should work on a single value passed or a Series passed in as an argument.
- The function should not have a `for` loop, if it does you will lose points!

[21]:
```
def extract_hour(time):
    """
    Extract hour information from the time given in hhmm format.

    Input:
        time (float64 or int64): Series of time given in hhmm format.
          Takes on values in 0.0-2400.0 in float64 representation or
          values in 0-2400 in int64 representation

    Returns:
```

```
        array (float64 or int64): Series of same dimension as input of hours
          values should be 0-24, i.e., 24-hour day format

    Example: 1303 should return 13
    >>> extract_hour(1303.0)
    13
    """
    time = pd.Series(time)
    hours = (time // 100).astype(int)
    return hours.iloc[0] if len(hours) == 1 else hours
```

[22]:
```
def extract_min(time):
    """
    Extract minute information from the time given in hhmm time.

    Input:
        time (float64 or int64): Series of time given in hhmm format.
          Takes on values in 0.0-2400.0 in float64 representation or
          values in 0-2400 in int64 representation

    Returns:
        array (float64 or int64): Series of same dimension as input of minutes.
          values should be 0-59

    Example: 1303 should return 3
    >>> extract_mins(1303.0)
    3
    """
    time = pd.Series(time)
    mins = (time % 100).astype(int)
    return mins.iloc[0] if len(mins) == 1 else mins
```

Here are some examples of usage:

```
>>> extract_hour(1450)
>>> extract_hour(flights['actDepTime'][100:200])
>>> extract_hour(flights['crsDepTime'])
>>> extract_min(1450)
>>> extract_min(flights['actDepTime'][100:200])
>>> extract_min(flights['crsDepTime'])
```

[23]:
```
print(extract_hour(1450))
print(extract_min(1450))
print(extract_hour(flights['actDepTime'][100:200]).head())
print(extract_min(flights['actDepTime'][100:200]).head())
print(extract_hour(flights['crsDepTime']).head())
print(extract_min(flights['crsDepTime']).head())
```

```
14
50
100    16
101    20
102    18
103    15
104    12
Name: actDepTime, dtype: int64
100    36
101    28
102    43
103    47
104    49
Name: actDepTime, dtype: int64
0    17
1     6
2    14
3    12
4    19
Name: crsDepTime, dtype: int64
0    45
1    20
2     5
3    15
4    50
Name: crsDepTime, dtype: int64
```

[24]:
```python
grader.check("q3a")
```

[24]:
```
q3a results: All test cases passed!
```

2.6.2 Q3b - Calculate Time in Minutes

Convert a time formatted in `hhmm` time in the number of minutes.

For example, the following times in `hhmm` format converted to minutes are: `1005` is 605 minutes and `837` is 517 minutes and `1524` is 924 minutes.

This function `convert_to_minutes` function, should make use of the two functions you wrote above: `extract_hour` and `extract_min`.

Comment:

- The function should work on a single value passed or a Series passed in as an argument.
- The function should not have a `for` loop, if it does you will lose points!
- The function should use the functions `extract_hour` and `extract_min`.

[25]:
```python
def convert_to_minutes(time):
    """
    Converts time in hhmm format to number of minutes
```

```
    Input:
       time (float64 or int64): Series of time given in hhmm format.
          Takes on values in 0.0-2400.0 in float64 representation or
          values in 0-2400 in int64 representation

    Returns:
        array (float64 or int64): Series of same dimension as input with
           total mins


    Example: 1303 is converted to 783
    >>> convert_to_minutes(1303.0)
    783.0
    """
     # make use of the functions you wrote above: extract_hour, extract_min
    if isinstance(time, pd.Series):
        return (extract_hour(time) * 60 + extract_min(time)).astype(float)
    else:
        return float(extract_hour(time) * 60 + extract_min(time))
```

[26]:
```
print(convert_to_minutes(1524))
print(convert_to_minutes(flights['crsDepTime'].head()))
```

```
924.0
0    1065.0
1     380.0
2     845.0
3     735.0
4    1190.0
Name: crsDepTime, dtype: float64
```

[27]:
```
grader.check("q3b")
```

[27]: q3b results: All test cases passed!

2.6.3 Q3c - Calculate Delayed Flights

You will add two new columns to the `flights` DataFrame that will contain the departure delay `depDelay` and arrival delay `arrDelay`.

To help answer this question, implement the helper functions, `calc_time_diff`. Make use of the functions above, e.g., `convert_to_minutes`.

Be careful for handling flights that that did not leave on their scheduled day indicated in the `depNextDay` and `arrNextDay` columns.
These two variables are encoded as: * 0, left/arrived on scheduled day * -1, left/arrived on the day before scheduled day * 1, left/arrived 1 day after scheduled day * 2, left/arrived 2 days after

scheduled day

Note, you can have *negative* flight delays for flights that leave / arrive early.

Notes:

- The function should work on Series passed in as an argument.
- The function should not have a `for` loop, if it does you will lose points!

[28]:
```
def calc_time_diff(x, y, nextDay):
    """
    Calculate the time difference, y - x, accounting for nextDay changes.

    Input:
        x,y (float64 or int64): Series of scheduled time given in hhmm format.
          Takes on values in 0.0-2400.0 due to float64 representation or
          values in 0-2400 in int64 representation
        nextDay (int): Series of next day indicator, takes on values: -1, 0, 1,␣
↪2

    Returns:
        array (float64): Series of input dimension with delay time


    Example: 1303 is converted to 783
             1305 is converted to 785
    >>> calc_time_diff(pd.Series([1303]), pd.Series([1305]), pd.Series([0]))
    2

    Example: 2320.0 is converted to 1400.0
               37.0 is converted to    37.0
    >>> calc_time_diff(pd.Series([2320.0]), pd.Series([37.0]), pd.Series([1]))
    77.0
    """
    # make use of the convert_to_minutes function
    dep_minutes = convert_to_minutes(x)
    arr_minutes = convert_to_minutes(y)
    diff = arr_minutes - dep_minutes + (nextDay * 1440)

    return diff
```

[29]:
```
flights['depDelay'] = calc_time_diff(
 flights['crsDepTime'],
 flights['actDepTime'],
 flights['depNextDay']
)
flights['arrDelay'] = calc_time_diff(
 flights['crsArrTime'],
 flights['actArrTime'],
```

```
 flights['arrNextDay']
)
print(flights['depDelay'].head(5))
print(flights['arrDelay'].head(5))
```

```
0    90.0
1    -4.0
2    -4.0
3    -8.0
4     8.0
Name: depDelay, dtype: float64
0    71.0
1    -7.0
2   -16.0
3   -21.0
4    -3.0
Name: arrDelay, dtype: float64
```

[30]:
```
grader.check("q3c")
```

[30]: q3c results: All test cases passed!

2.7 Q4 - Patterns of Delays

Next, you will examine if there are any patterns in the delays.

2.7.1 Q4a - Delay Statistics

Present the mean, standard deviation, and median for departure and arrival delays for each airport, `DTW` and `ORD` as a destination in a DataFrame.

Repeat the results where each airport, `DTW` and `ORD`, is the origin.

NOTE: all the meaningful autograded tests are hidden for this problem.

Hint You should consider using `groupby`

The DataFrame you create should look something like (note the values are not correct):

[31]:
```
# Calculate the mean, std, and median Depature Delay for flights leaving DTW/ORD
# Calculate the mean, std, and median Delay for flights arriving at DTW/ORD
# Report in requested DataFrames

destDelays = (flights[flights['dest'].isin(['DTW', 'ORD'])].groupby('dest').agg(
     mean_arr_delay=('arrDelay', 'mean'),
     std_arr_delay=('arrDelay', 'std'),
     median_arr_delay=('arrDelay', 'median'),
     mean_dep_delay=('depDelay', 'mean'),
     std_dep_delay=('depDelay', 'std'),
     median_dep_delay=('depDelay', 'median'),)
```

```
)
destDelays.columns = pd.MultiIndex.from_tuples([
     ('arrDelay', 'mean'),
     ('arrDelay', 'std'),
     ('arrDelay', 'median'),
     ('depDelay', 'mean'),
     ('depDelay', 'std'),
     ('depDelay', 'median')]
)
originDelays = (flights[flights['origin'].isin(['DTW', 'ORD'])].
 ↪groupby('origin').agg(
    mean_arr_delay=('arrDelay', 'mean'),
     std_arr_delay=('arrDelay', 'std'),
     median_arr_delay=('arrDelay', 'median'),
     mean_dep_delay=('depDelay', 'mean'),
     std_dep_delay=('depDelay', 'std'),
     median_dep_delay=('depDelay', 'median'),)
)
originDelays.columns = pd.MultiIndex.from_tuples([
     ('arrDelay', 'mean'),
     ('arrDelay', 'std'),
     ('arrDelay', 'median'),
     ('depDelay', 'mean'),
     ('depDelay', 'std'),
     ('depDelay', 'median')])
```

[32]:
```
grader.check("q4a")
```

[32]: q4a results: All test cases passed!

2.7.2 Q4b - Distribution of Departure Delays

Let's look at the distribution of departure delays for flights leaving `DTW` and `ORD`.

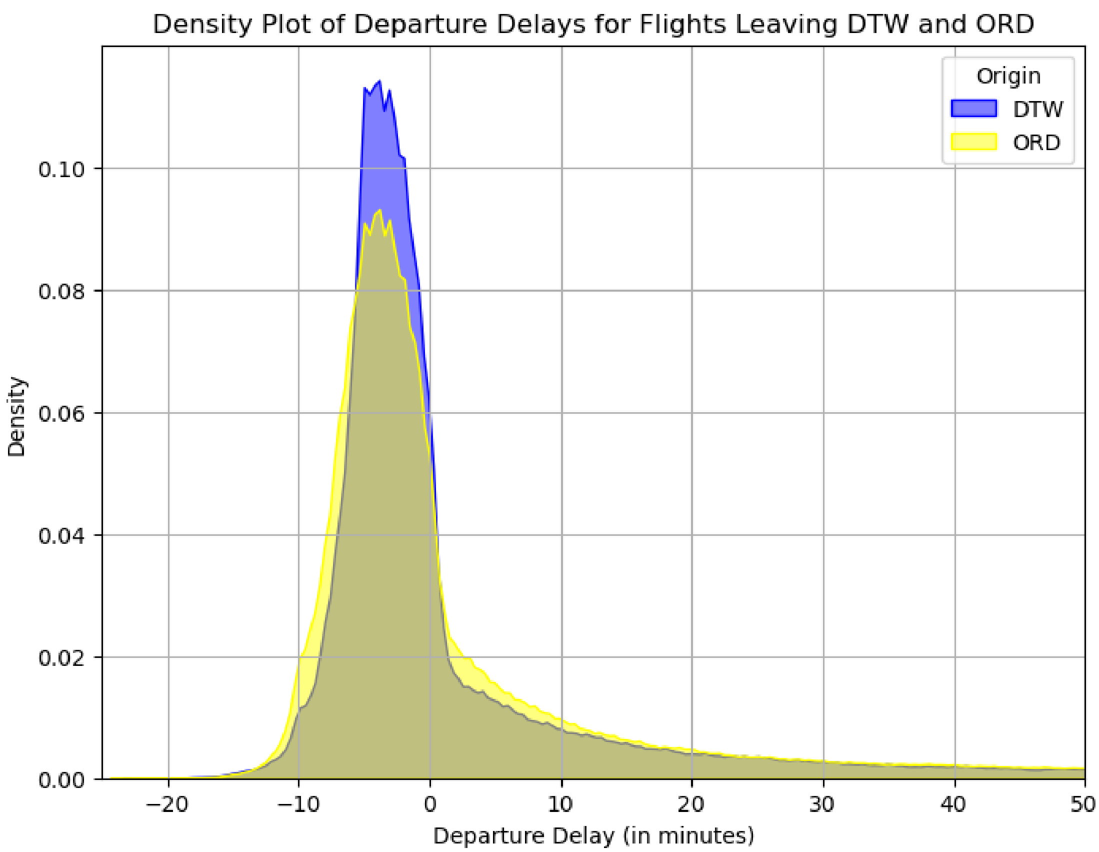
Create overlapping density plots to examine these distributions. Only show the delays from -25 to 50 minutes.

Hint Try using seaborns `kdeplot`

[33]:
```
departure_delays = flights[flights['origin'].isin(['DTW', 'ORD'])]
filtered_delays = departure_delays[(departure_delays['depDelay'] >= -25) &␣
 ↪(departure_delays['depDelay'] <= 50)]
plt.figure(figsize=(8, 6))

sns.kdeplot(data=filtered_delays[filtered_delays['origin'] == 'DTW'],␣
 ↪x='depDelay', label='DTW', color='blue', fill=True, alpha=0.5, bw_adjust=0.5)
```

```
sns.kdeplot(data=filtered_delays[filtered_delays['origin'] == 'ORD'],
 ↪x='depDelay', label='ORD', color='yellow', fill=True, alpha=0.5, bw_adjust=0.
 ↪5)
plt.title('Density Plot of Departure Delays for Flights Leaving DTW and ORD')
plt.xlabel('Departure Delay (in minutes)')
plt.ylabel('Density')
plt.xlim(-25, 50)
plt.legend(title='Origin')
plt.grid()
```



2.7.3 Q4c - Departure Delays by Day of Week

Let's examine if the departure delays differ by which day a flight is scheduled to leave.

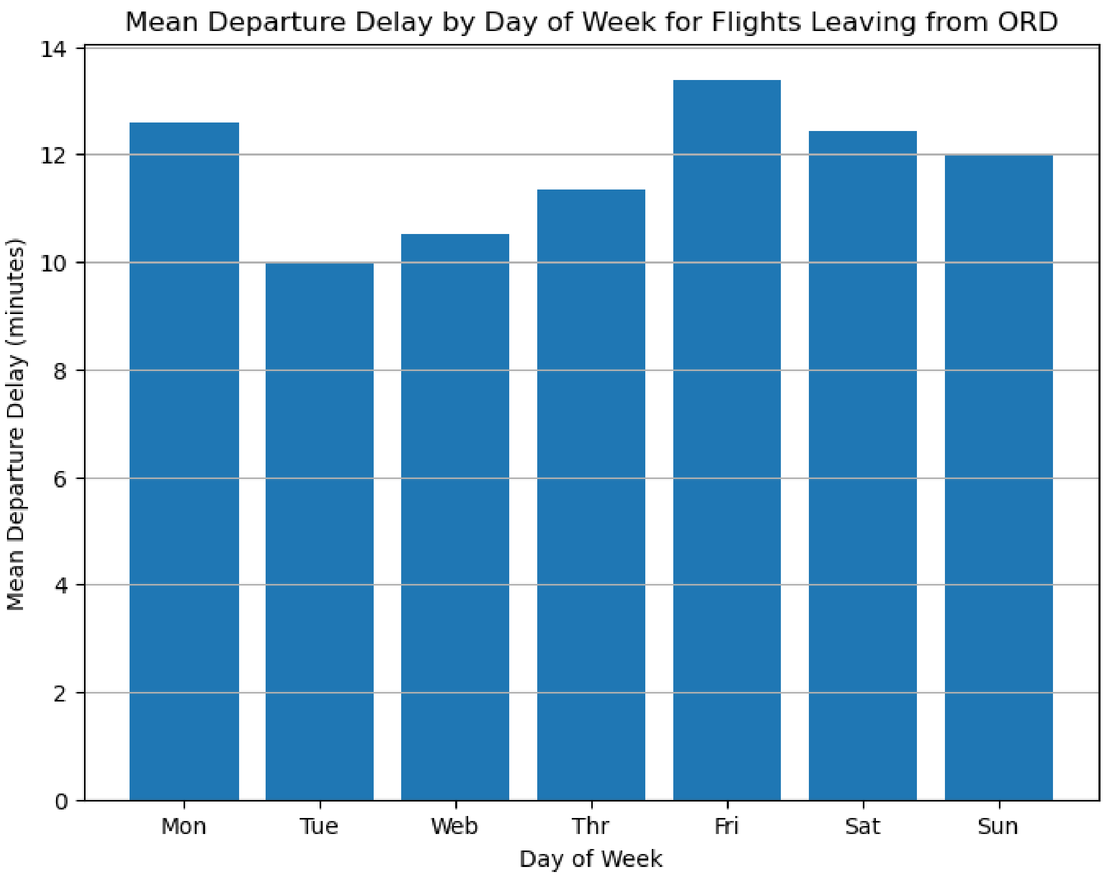
Create a bar chart showing the mean departure delay for each day of the week. Report mean delays for Sun., Mon., Tues., …

In order to get the day of the week, create a two new columns `dateVal` that is a datetime object of the date information of the flight. Also, create a column `dayOfWeek` that encodes the day of the week for the flight. You can make use of functions to convert datetime to day of the week.

We are looking at departure delays for flights leaving from ORD.

[34]:

```python
# Look at departure delays for flights leaving ORD
# Create a new column "dateVal" with a datetime object that may be useful
# for this questions. Also, add a column for "dayOfWeek"
# Create a bar chart of the mean `depDelay` by day of the week: Mon., Tues., ...
flights['dateVal'] = pd.to_datetime(flights[['year', 'month', 'day']])
flights['dayOfWeek'] = flights['dateVal'].dt.day_name()
ord_flights = flights[flights['origin'] == 'ORD']
mean_delays = ord_flights.groupby('dayOfWeek')['depDelay'].mean().reindex(
 ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
)
plt.figure(figsize=(8, 6))
plt.bar(mean_delays.index, mean_delays.values)
plt.title('Mean Departure Delay by Day of Week for Flights Leaving from ORD')
plt.xlabel('Day of Week')
plt.ylabel('Mean Departure Delay (minutes)')
plt.xticks(ticks=range(7), labels=['Mon', 'Tue', 'Web', 'Thr', 'Fri', 'Sat',␣
 ↪'Sun'])
plt.grid(axis='y')
```



[35]:
```python
grader.check("q4c")
```

[35]:
```
q4c results: All test cases passed!
```

2.7.4 Q4d - Dept. Delays by Month

Next, let's examine if the depature delays differ by month of the year.

Create a bar chart showing the mean departure delay for each month of the year (Jan, Feb, Mar, Apr, …)

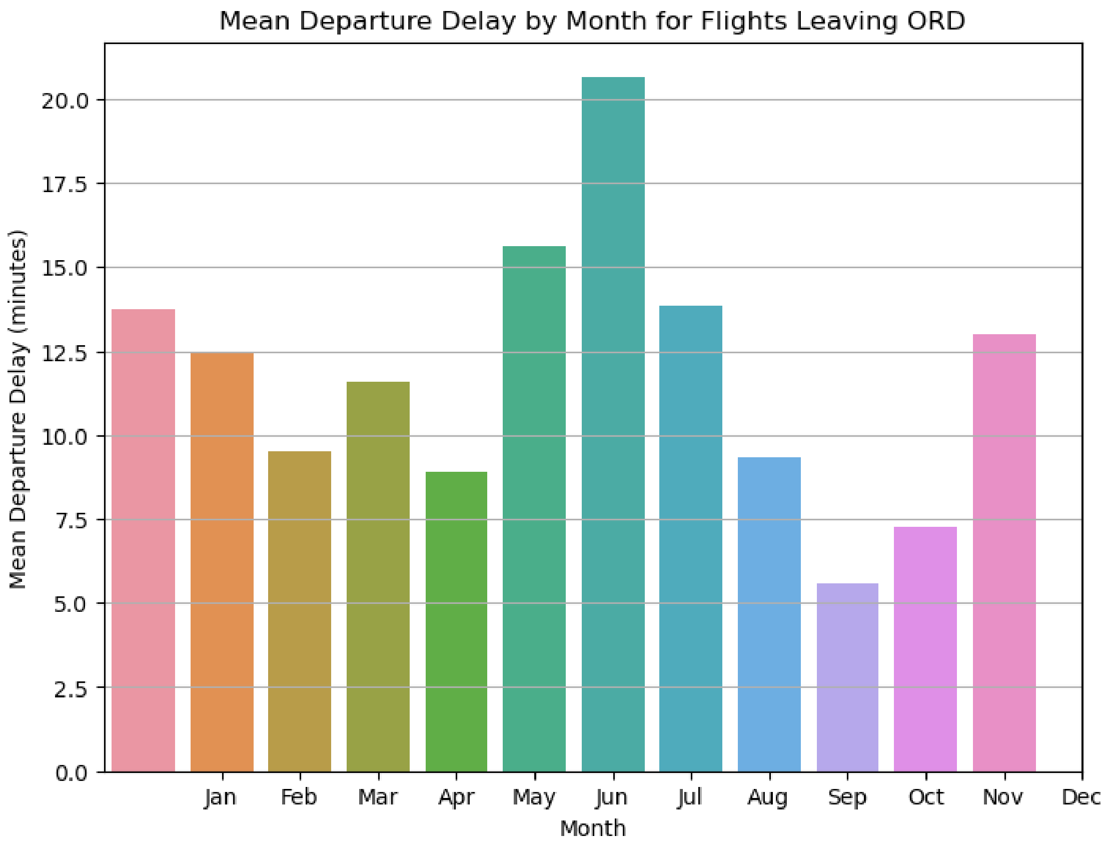
We again are looking at departure delays for flights leaving from ORD.

[36]:
```python
# Create a bar plot with the mean dep. delay for each month.
ord_flights = flights[flights['origin'] == 'ORD']

# Calculate mean departure delay by month
mean_delays_by_month = ord_flights.groupby('month')['depDelay'].mean()

# Create bar chart
plt.figure(figsize=(8, 6))  # Adjust figure size if needed
sns.barplot(x=mean_delays_by_month.index, y=mean_delays_by_month.values)

# Customize the plot
plt.xlabel('Month')
plt.ylabel('Mean Departure Delay (minutes)')
plt.title('Mean Departure Delay by Month for Flights Leaving ORD')
plt.xticks(ticks=range(1, 13), labels=['Jan', 'Feb', 'Mar', 'Apr', 'May',␣
 ↪'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'])
plt.grid(axis='y')
# Show the plot
plt.show()
```



2.7.5 Q4e - Most Delayed Routes

Let's examine the routes that are most delayed.

Report out a DataSeries, `most_delay_routeA`, with the airport as it's index and the value the mean arrival delay, only consider flights departing from `ORD` and routes that have at least 200 flights.

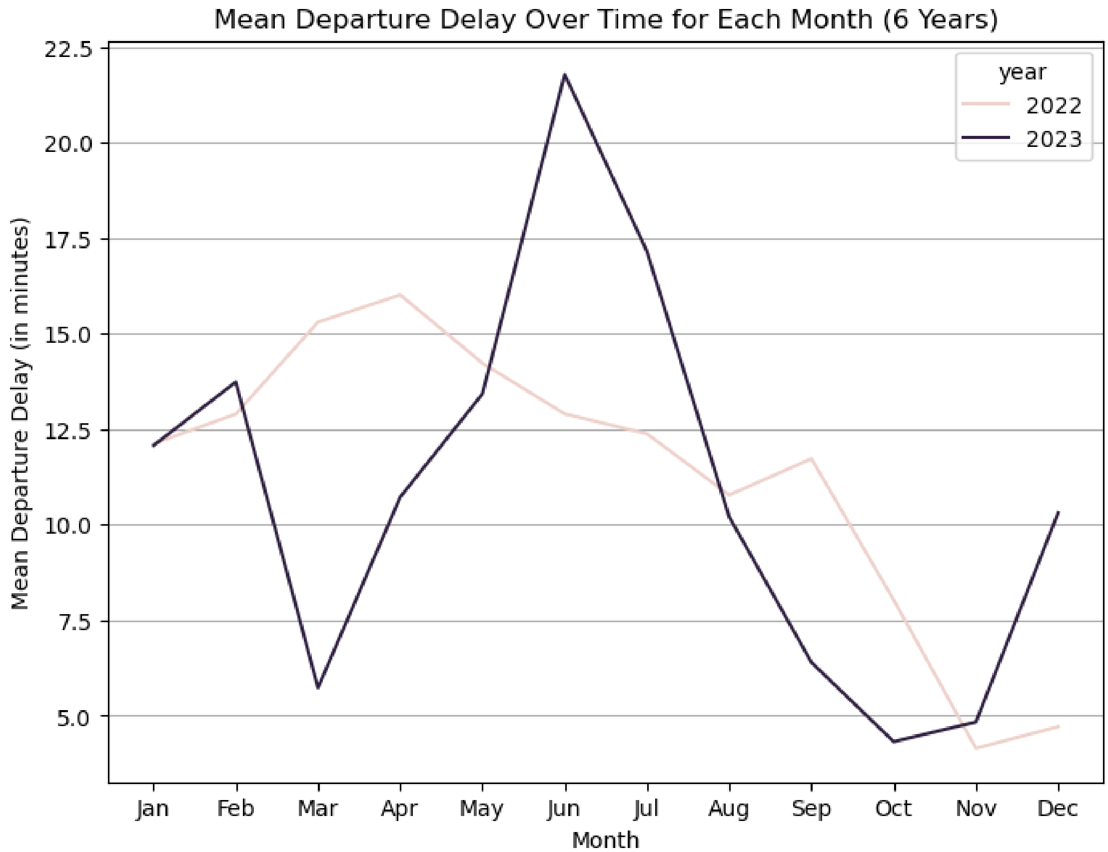
Sort the DataSeries in descending order.

Repeat the analysis for `DTW`, but use departure delay, report results in `most_delay_routeB` DataSeries.

[37]:
```
# Create a line chart plotting the mean dep. delay over time for each month
#  of the 6 years

flights['month'] = flights['dateVal'].dt.month_name()
mean_dep_delay_month = flights.groupby(['year', 'month'])['depDelay'].mean().
 ↪reset_index()
plt.figure(figsize=(8,6))
sns.lineplot(
 data=mean_dep_delay_month,
 x='month',
```

```
 y='depDelay',
 hue='year'
)
plt.title('Mean Departure Delay Over Time for Each Month (6 Years)')
plt.xlabel('Month')
plt.ylabel('Mean Departure Delay (in minutes)')
plt.xticks(ticks=range(12), labels=['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',␣
↪'Jul','Aug','Sep','Oct','Nov','Dec'])
plt.grid(axis='y')
```



[38]:
```
most_delay_routeA = (flights[flights['origin'] == 'ORD'].
↪groupby('dest')['arrDelay'].agg(mean_arr_delay='mean',count='count')
 .query('count >= 200')['mean_arr_delay']
 .sort_values(ascending=False)
)
most_delay_routeB = (flights[flights['origin'] == 'DTW'].
↪groupby('dest')['depDelay'].agg(mean_dep_delay='mean',count='count')
 .query('count >= 200')['mean_dep_delay']
 .sort_values(ascending=False)
```

```
)
most_delay_routeA.head(), most_delay_routeB.head()
```

[38]:
```
(dest
 JST    27.169935
 ASE    23.138562
 AVP    18.191318
 JLN    16.776632
 ABE    14.748982
 Name: mean_arr_delay, dtype: float64,
 dest
 SBN    23.317881
 FWA    22.978182
 CIU    21.326316
 MQT    21.047767
 HNL    18.811047
 Name: mean_dep_delay, dtype: float64)
```

[39]:
```
delay_routes = (
 flights[flights['origin'] == 'ORD']
 .groupby('dest')
 .agg(
 mean_dep_delay=('depDelay', 'mean'),
 mean_arr_delay=('arrDelay', 'mean'),
 count=('depDelay', 'count')
 )
 .reset_index()
)
# delay_routes.loc[(delay_routes[('depDelay', 'count')] > 200)].shape
print(delay_routes[delay_routes['count'] > 200].shape)
delay_routes.head()
```

```
(138, 4)
```

[39]:

	dest	mean_dep_delay	mean_arr_delay	count
0	ABE	16.877883	14.748982	737
1	ABQ	7.653034	1.664908	758
2	ALB	9.784349	5.932497	1674
3	ANC	12.219709	-0.812601	619
4	ASE	28.052941	23.138562	1530

[40]:
```
grader.check("q4e")
```

[40]:
```
q4e results: All test cases passed!
```

2.7.6 Q4f - Delays by Carrier

Because `ORD` is a United Hub and `DTW` is a Delta Hub we expect to see the delays grouped by airport may also be similar to the delays grouped by carrier. Let's examine this.

Consider all flights, not just those arriving or departing from `ORD` and `DTW`. Report out the mean, standard deviation, and median arrival and departure delays for each carrier in a DataFrame.

Label the carriers with their name, not code.

Airline Code	Airline Name	Regional Partner
UA	United Airlines	NA
DL	Delta Airlines	NA
OO	Skywest Airlines	United
9E	Endeavor Air	Delta

The DataFrame `carrierDelays` has columns: `carrier`, `arrDelayMean`, `arrDelayStd`, `arrDelayMedian`, `depDelayMean`, `depDelayStd`, `depDelayMedian`

Sort the DataFrame by increasing mean arrival delays.

[41]:
```
carrierDelays = (flights.groupby('carrier').agg(arrDelayMean=('arrDelay',␣
↪'mean'),arrDelayStd=('arrDelay', 'std'),arrDelayMedian=('arrDelay',␣
↪'median'),
 depDelayMean=('depDelay', 'mean'),
 depDelayStd=('depDelay', 'std'),
 depDelayMedian=('depDelay', 'median')
 ).reset_index())
carrierDelays['carrier'] = carrierDelays['carrier'].map({
 'UA': 'United Airlines',
 'DL': 'Delta Airlines',
 'OO': 'Skywest Airlines',
 '9E': 'Endeavor Air'
})
carrierDelays.sort_values(by='arrDelayMean', inplace=True)
carrierDelays
```

[41]:
```
            carrier  arrDelayMean  arrDelayStd  arrDelayMedian  depDelayMean  \
0      Endeavor Air     -0.639414    49.252149           -10.0      6.042961
1    Delta Airlines      1.954484    50.207973            -9.0      9.891461
3   United Airlines      4.177474    51.646764            -7.0     11.224120
2  Skywest Airlines      7.926424    68.812359            -7.0     13.294421

   depDelayStd  depDelayMedian
0    47.087847            -4.0
1    48.154017            -2.0
3    49.727298            -2.0
2    67.310415            -3.0
```

[42]:
```python
grader.check("q4f")
```

[42]:
```
q4f results: All test cases passed!
```

2.8 Bonus

Find the tail number of the top ten planes (identified by `tailNum`), with the worst departure delays (average delays). You may find `drop_duplicates`, `agg`, and `sort_values` helpful.

Consider only planes that have made at least 10 flights.

Report out results in a DataFrame, `topDelayed`, the tail number `tailNum`, number of flights `num`, and the mean delay `mnDelay`.

[43]:
```python
# Find the tail numbers of the planes with the worst mean dep. delays.
# Store the tail number "tailNum", number of flights "num", and
# mean delay "mnDelay" in a DataFrame
topDelayed = (flights.groupby('tailNum').agg(num=('tailNum',␣
 ↪'count'),mnDelay=('depDelay', 'mean')).reset_index().query('num >= 10').
 ↪sort_values(by='mnDelay', ascending=False)
)
topDelayed.shape
topDelayed.head(10)
```

[43]:

	tailNum	num	mnDelay
1271	N507DZ	11	129.727273
1535	N670UA	11	117.727273
878	N3730B	16	104.000000
1528	N67052	20	97.650000
1280	N510SY	69	94.202899
940	N3752	15	85.800000
1302	N521DT	10	77.100000
1001	N377DA	14	76.357143
1383	N59053	14	75.071429
1468	N641UA	36	72.611111

[44]:
```python
grader.check("b1")
```

[44]:
```
b1 results:
    b1 - 1 result:
        Test case passed

    b1 - 2 result:
        Test case failed
        Trying:
            topDelayed.shape == (2273, 3)
        Expecting:
            True
```

```
**********************************************************************
Line 1, in b1 1
Failed example:
    topDelayed.shape == (2273, 3)
Expected:
    True
Got:
    False

b1 - 3 result:
    Test case passed
```

2.9 Congratulations! You have finished P2!

2.9.1 Submission Instructions

Below, you will see a cell. Running this cell will automatically generate a zip file with your autograded answers. Once you submit this file to the P2 assignment on Gradescope, Gradescope will automatically submit a PDF file with your some of your answers to the P2 - Figures assignment (making them easier to grade).

Important: Please check that your responses were generated and submitted correctly to the P2 - Figures Assignment.

You are responsible for ensuring your submission follows our requirements and that the PDF for P2- Figures answers was generated/submitted correctly. We will not be granting regrade requests nor extensions to submissions that don't follow instructions. If you encounter any difficulties with the submission, contact course staff well-ahead of the deadline.

Make sure you have run all cells in your notebook **in order** before running the cell below, so that all images/graphs appear in the output. The cell below will generate a zip file for you to submit. **Please save before exporting!**

2.10 Submission

Make sure you have run all cells in your notebook in order before running the cell below, so that all images/graphs appear in the output. The cell below will generate a zip file for you to submit. **Please save before exporting!**

[ ]:
```
# Save your notebook first, then run this cell to export your submission.
grader.export()
```