Activity_Course 6 Automatidata project lab

November 2, 2023

1 Automatidata project

Course 6 - The Nuts and bolts of machine learning

You are a data professional in a data analytics firm called Automatidata. Their client, the New York City Taxi & Limousine Commission (New York City TLC), was impressed with the work you have done and has requested that you build a machine learning model to predict if a customer will not leave a tip. They want to use the model in an app that will alert taxi drivers to customers who are unlikely to tip, since drivers depend on tips.

A notebook was structured and prepared to help you in this project. Please complete the following questions.

2 Course 6 End-of-course project: Build a machine learning model

In this activity, you will practice using tree-based modeling techniques to predict on a binary target class.

The purpose of this model is to find ways to generate more revenue for taxi cab drivers.

The goal of this model is to predict whether or not a customer is a generous tipper.

This activity has three parts:

Part 1: Ethical considerations * Consider the ethical implications of the request

- Should the objective of the model be adjusted?

Part 2: Feature engineering

- Perform feature selection, extraction, and transformation to prepare the data for modeling

Part 3: Modeling

- Build the models, evaluate them, and advise on next steps

Follow the instructions and answer the questions below to complete the activity. Then, complete an Executive Summary using the questions listed on the PACE Strategy Document.

Be sure to complete this activity before moving on. The next course item will provide you with a completed exemplar to compare to your own work.

3 Build a machine learning model

4 PACE stages

Throughout these project notebooks, you'll see references to the problem-solving framework PACE. The following notebook components are labeled with the respective PACE stage: Plan, Analyze, Construct, and Execute.

4.1 PACE: Plan

Consider the questions in your PACE Strategy Document to reflect on the Plan stage.

In this stage, consider the following questions:

1. What are you being asked to do?
2. What are the ethical implications of the model? What are the consequences of your model making errors?

- What is the likely effect of the model when it predicts a false negative (i.e., when the model says a customer will give a tip, but they actually won't)?
- What is the likely effect of the model when it predicts a false positive (i.e., when the model says a customer will not give a tip, but they actually will)?

3. Do the benefits of such a model outweigh the potential problems?
4. Would you proceed with the request to build this model? Why or why not?
5. Can the objective be modified to make it less problematic?

Suppose you were to modify the modeling objective so, instead of predicting people who won't tip at all, you predicted people who are particularly generous—those who will tip 20% or more? Consider the following questions:

1. What features do you need to make this prediction?
2. What would be the target variable?
3. What metric should you use to evaluate your model? Do you have enough information to decide this now?

Complete the following steps to begin:

4.1.1 Task 1. Imports and data loading

Import packages and libraries needed to build and evaluate random forest and XGBoost classification models.

[1]:
```
# Import packages and libraries
import pandas as pd
import numpy as np
```

```python
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.metrics import roc_auc_score, roc_curve, RocCurveDisplay,␣
↪confusion_matrix, ConfusionMatrixDisplay,\
                            accuracy_score, precision_score, recall_score,␣
↪f1_score
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier, plot_importance
```

[2]:
```python
# RUN THIS CELL TO SEE ALL COLUMNS
pd.set_option('display.max_columns', None)
```

Begin by reading in the data. There are two dataframes: one containing the original data, the other containing the mean durations, mean distances, and predicted fares from the previous course's project called nyc_preds_means.csv.

Note: `Pandas` reads in the dataset as `df0`, now inspect the first five rows. As shown in this cell, the dataset has been automatically loaded in for you. You do not need to download the .csv file, or provide more code, in order to access the dataset and proceed with this lab. Please continue with this activity by completing the following instructions.

[3]:
```python
# Load dataset into dataframe
df0 = pd.read_csv('2017_Yellow_Taxi_Trip_Data.csv')

# Import predicted fares and mean distance and duration from previous course
nyc_preds_means = pd.read_csv('nyc_preds_means.csv')
```

Inspect the first few rows of `df0`.

[4]:
```python
# Inspect the first few rows of df0
df0.head(5)
```

[4]:
```
   Unnamed: 0  VendorID     tpep_pickup_datetime    tpep_dropoff_datetime  \
0    24870114         2   03/25/2017 8:55:43 AM   03/25/2017 9:09:47 AM
1    35634249         1   04/11/2017 2:53:28 PM   04/11/2017 3:19:58 PM
2   106203690         1   12/15/2017 7:26:56 AM   12/15/2017 7:34:08 AM
3    38942136         2   05/07/2017 1:17:59 PM   05/07/2017 1:48:14 PM
4    30841670         2  04/15/2017 11:32:20 PM  04/15/2017 11:49:03 PM

   passenger_count  trip_distance  RatecodeID store_and_fwd_flag  \
0                6           3.34           1                  N
1                1           1.80           1                  N
2                1           1.00           1                  N
3                1           3.70           1                  N
4                1           4.37           1                  N
```

```
   PULocationID  DOLocationID  payment_type  fare_amount  extra  mta_tax  \
0           100           231             1         13.0    0.0      0.5
1           186            43             1         16.0    0.0      0.5
2           262           236             1          6.5    0.0      0.5
3           188            97             1         20.5    0.0      0.5
4             4           112             2         16.5    0.5      0.5

   tip_amount  tolls_amount  improvement_surcharge  total_amount
0        2.76           0.0                    0.3         16.56
1        4.00           0.0                    0.3         20.80
2        1.45           0.0                    0.3          8.75
3        6.39           0.0                    0.3         27.69
4        0.00           0.0                    0.3         17.80
```

[5]:
```python
df0.shape
```

[5]:
```
(22699, 18)
```

Inspect the first few rows of `nyc_preds_means`.

[6]:
```python
# Inspect the first few rows of `nyc_preds_means`
nyc_preds_means.head(5)
```

[6]:
```
   mean_duration  mean_distance  predicted_fare
0      22.847222       3.521667       16.434245
1      24.470370       3.108889       16.052218
2       7.250000       0.881429        7.053706
3      30.250000       3.700000       18.731650
4      14.616667       4.435000       15.845642
```

[7]:
```python
nyc_preds_means.shape
```

[7]:
```
(22699, 3)
```

Join the two dataframes Join the two dataframes using a method of your choice.

[8]:
```python
# Merge datasets
df0 = df0.merge(nyc_preds_means, left_index= True, right_index= True)
df0.head(5)
```

[8]:
```
   Unnamed: 0  VendorID    tpep_pickup_datetime   tpep_dropoff_datetime  \
0    24870114         2   03/25/2017 8:55:43 AM   03/25/2017 9:09:47 AM
1    35634249         1   04/11/2017 2:53:28 PM   04/11/2017 3:19:58 PM
2   106203690         1   12/15/2017 7:26:56 AM   12/15/2017 7:34:08 AM
3    38942136         2   05/07/2017 1:17:59 PM   05/07/2017 1:48:14 PM
4    30841670         2  04/15/2017 11:32:20 PM  04/15/2017 11:49:03 PM
```

```
   passenger_count  trip_distance  RatecodeID store_and_fwd_flag  \
0                6           3.34           1                  N
1                1           1.80           1                  N
2                1           1.00           1                  N
3                1           3.70           1                  N
4                1           4.37           1                  N

   PULocationID  DOLocationID  payment_type  fare_amount  extra  mta_tax  \
0           100           231             1         13.0    0.0      0.5
1           186            43             1         16.0    0.0      0.5
2           262           236             1          6.5    0.0      0.5
3           188            97             1         20.5    0.0      0.5
4             4           112             2         16.5    0.5      0.5

   tip_amount  tolls_amount  improvement_surcharge  total_amount  \
0        2.76           0.0                    0.3         16.56
1        4.00           0.0                    0.3         20.80
2        1.45           0.0                    0.3          8.75
3        6.39           0.0                    0.3         27.69
4        0.00           0.0                    0.3         17.80

   mean_duration  mean_distance  predicted_fare
0      22.847222       3.521667       16.434245
1      24.470370       3.108889       16.052218
2       7.250000       0.881429        7.053706
3      30.250000       3.700000       18.731650
4      14.616667       4.435000       15.845642
```

[9]:
```
df0.shape
```

[9]:
```
(22699, 21)
```

4.2 PACE: Analyze

Consider the questions in your PACE Strategy Documentto reflect on the Analyze stage.

4.2.1 Task 2. Feature engineering

You have already prepared much of this data and performed exploratory data analysis (EDA) in previous courses.

Call `info()` on the new combined dataframe.

[10]:
```
df0.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 22699 entries, 0 to 22698
Data columns (total 21 columns):
 #   Column                 Non-Null Count  Dtype  
---  ------                 --------------  -----  
 0   Unnamed: 0             22699 non-null  int64  
 1   VendorID               22699 non-null  int64  
 2   tpep_pickup_datetime   22699 non-null  object 
 3   tpep_dropoff_datetime  22699 non-null  object 
 4   passenger_count        22699 non-null  int64  
 5   trip_distance          22699 non-null  float64
 6   RatecodeID             22699 non-null  int64  
 7   store_and_fwd_flag     22699 non-null  object 
 8   PULocationID           22699 non-null  int64  
 9   DOLocationID           22699 non-null  int64  
 10  payment_type           22699 non-null  int64  
 11  fare_amount            22699 non-null  float64
 12  extra                  22699 non-null  float64
 13  mta_tax                22699 non-null  float64
 14  tip_amount             22699 non-null  float64
 15  tolls_amount           22699 non-null  float64
 16  improvement_surcharge  22699 non-null  float64
 17  total_amount           22699 non-null  float64
 18  mean_duration          22699 non-null  float64
 19  mean_distance          22699 non-null  float64
 20  predicted_fare         22699 non-null  float64
dtypes: float64(11), int64(7), object(3)
memory usage: 3.6+ MB
```

[11]:
```
df0.groupby(["payment_type"])["tip_amount"].mean()
```

[11]:
```
payment_type
1    2.7298
2    0.0000
3    0.0000
4    0.0000
Name: tip_amount, dtype: float64
```

You know from your EDA that customers who pay cash generally have a tip amount of $0. To meet the modeling objective, you'll need to sample the data to select only the customers who pay with credit card.

Copy `df0` and assign the result to a variable called `df1`. Then, use a Boolean mask to filter `df1` so it contains only customers who paid with credit card.

[12]:
```
# Subset the data to isolate only customers who paid by credit card
df1 = df0[df0["payment_type"] == 1]
df1.reset_index(drop= True, inplace= True)
```

```
df1.shape
```

[12]:
```
(15265, 21)
```

[13]:
```
df1["tip_amount"].describe()
```

[13]:
```
count    15265.000000
mean         2.729800
std          3.036917
min          0.000000
25%          1.350000
50%          2.000000
75%          3.050000
max        200.000000
Name: tip_amount, dtype: float64
```

Target Notice that there isn't a column that indicates tip percent, which is what you need to create the target variable. You'll have to engineer it.

Add a `tip_percent` column to the dataframe by performing the following calculation:

$$tip\ percent = \frac{tip\ amount}{total\ amount - tip\ amount}$$

Round the result to three places beyond the decimal. **This is an important step.** It affects how many customers are labeled as generous tippers. In fact, without performing this step, approximately 1,800 people who do tip 20% would be labeled as not generous.

To understand why, you must consider how floats work. Computers make their calculations using floating-point arithmetic (hence the word "float"). Floating-point arithmetic is a system that allows computers to express both very large numbers and very small numbers with a high degree of precision, encoded in binary. However, precision is limited by the number of bits used to represent a number, which is generally 32 or 64, depending on the capabilities of your operating system.

This comes with limitations in that sometimes calculations that should result in clean, precise values end up being encoded as very long decimals. Take, for example, the following calculation:

[14]:
```
# Run this cell
1.1 + 2.2
```

[14]:
```
3.3000000000000003
```

Notice the three that is 16 places to the right of the decimal. As a consequence, if you were to then have a step in your code that identifies values 3.3, this would not be included in the result. Therefore, whenever you perform a calculation to compute a number that is then used to make an important decision or filtration, round the number. How many degrees of precision you round to is your decision, which should be based on your use case.

Refer to this guide for more information related to floating-point arithmetic.
Refer to this guide for more information related to fixed-point arithmetic, which is an alternative to floating-point arithmetic used in certain cases.

[15]:
```
# Create tip % col
df1["tip_percent"] = round(df1["tip_amount"] / (df1["total_amount"] -
 ↪df1["tip_amount"]), 3)
df1["tip_percent"].head(10)
```

[15]:
```
0    0.200
1    0.238
2    0.199
3    0.300
4    0.200
5    0.200
6    0.100
7    0.199
8    0.200
9    0.250
Name: tip_percent, dtype: float64
```

Now create another column called `generous`. This will be the target variable. The column should be a binary indicator of whether or not a customer tipped 20% (0=no, 1=yes).

1. Begin by making the `generous` column a copy of the `tip_percent` column.
2. Reassign the column by converting it to Boolean (True/False).
3. Reassign the column by converting Boolean to binary (1/0).

[16]:
```
# Create 'generous' col (target)
df1["generous"] = df1["tip_percent"]
df1["generous"] = df1["generous"] >= 0.200
df1["generous"] = df1["generous"].astype("int64")
df1["generous"].describe()
```

[16]:
```
count    15265.000000
mean         0.526368
std          0.499321
min          0.000000
25%          0.000000
50%          1.000000
75%          1.000000
max          1.000000
Name: generous, dtype: float64
```

HINT

To convert from Boolean to binary, use `.astype(int)` on the column.

Create day column Next, you're going to be working with the pickup and dropoff columns.

Convert the `tpep_pickup_datetime` and `tpep_dropoff_datetime` columns to datetime.

[17]:
```
# Convert pickup and dropoff cols to datetime
df1["tpep_pickup_datetime"] = df1["tpep_pickup_datetime"].astype("datetime64")
df1["tpep_dropoff_datetime"] = df1["tpep_dropoff_datetime"].astype("datetime64")
df1.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 15265 entries, 0 to 15264
Data columns (total 23 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   Unnamed: 0             15265 non-null  int64
 1   VendorID               15265 non-null  int64
 2   tpep_pickup_datetime   15265 non-null  datetime64[ns]
 3   tpep_dropoff_datetime  15265 non-null  datetime64[ns]
 4   passenger_count        15265 non-null  int64
 5   trip_distance          15265 non-null  float64
 6   RatecodeID             15265 non-null  int64
 7   store_and_fwd_flag     15265 non-null  object
 8   PULocationID           15265 non-null  int64
 9   DOLocationID           15265 non-null  int64
 10  payment_type           15265 non-null  int64
 11  fare_amount            15265 non-null  float64
 12  extra                  15265 non-null  float64
 13  mta_tax                15265 non-null  float64
 14  tip_amount             15265 non-null  float64
 15  tolls_amount           15265 non-null  float64
 16  improvement_surcharge  15265 non-null  float64
 17  total_amount           15265 non-null  float64
 18  mean_duration          15265 non-null  float64
 19  mean_distance          15265 non-null  float64
 20  predicted_fare         15265 non-null  float64
 21  tip_percent            15262 non-null  float64
 22  generous               15265 non-null  int64
dtypes: datetime64[ns](2), float64(12), int64(8), object(1)
memory usage: 2.7+ MB
```

Create a `day` column that contains only the day of the week when each passenger was picked up. Then, convert the values to lowercase.

[18]:
```
# Create a 'day' col
df1["day"] = df1["tpep_pickup_datetime"].dt.day_name().str.lower()
df1["day"].unique()
```

[18]:
```
array(['saturday', 'tuesday', 'friday', 'sunday', 'wednesday', 'monday',
       'thursday'], dtype=object)
```

HINT

To convert to day name, use `dt.day_name()` on the column.

Create time of day columns Next, engineer four new columns that represent time of day bins. Each column should contain binary values (0=no, 1=yes) that indicate whether a trip began (picked up) during the following times:

`am_rush` = [06:00–10:00)
`daytime` = [10:00–16:00)
`pm_rush` = [16:00–20:00)
`nighttime` = [20:00–06:00)

To do this, first create the four columns. For now, each new column should be identical and contain the same information: the hour (only) from the `tpep_pickup_datetime` column.

[19]:
```
# Create 'am_rush' col
df1["am_rush"] = df1["tpep_pickup_datetime"].dt.hour

# Create 'daytime' col
df1["daytime"] = df1["tpep_pickup_datetime"].dt.hour

# Create 'pm_rush' col
df1["pm_rush"] = df1["tpep_pickup_datetime"].dt.hour

# Create 'nighttime' col
df1["nighttime"] = df1["tpep_pickup_datetime"].dt.hour
```

You'll need to write four functions to convert each new column to binary (0/1). Begin with `am_rush`. Complete the function so if the hour is between [06:00–10:00), it returns 1, otherwise, it returns 0.

[20]:
```
# Define 'am_rush()' conversion function [06:00-10:00)
def am_rush(hour):
    if 6 < hour["am_rush"] < 10:
        result = 1
    else:
        result = 0
    return result
```

Now, apply the `am_rush()` function to the `am_rush` series to perform the conversion. Print the first five values of the column to make sure it did what you expected it to do.

Note: Be careful! If you run this cell twice, the function will be reapplied and the values will all be changed to 0.

[21]:
```
# Apply 'am_rush' function to the 'am_rush' series
df1["am_rush"] = df1.apply(am_rush, axis= 1)
df1["am_rush"].head(5)
```

[21]:
```
0    1
1    0
```

```
2    1
3    0
4    0
Name: am_rush, dtype: int64
```

Write functions to convert the three remaining columns and apply them to their respective series.

[22]:
```python
# Define 'daytime()' conversion function [10:00-16:00)
def daytime(hour):
    if 10 < hour["daytime"] < 16:
        result = 1
    else:
        result = 0
    return result
```

[23]:
```python
# Apply 'daytime()' function to the 'daytime' series
df1["daytime"] = df1.apply(daytime, axis=1)
df1["daytime"].head(5)
```

[23]:
```
0    0
1    1
2    0
3    1
4    0
Name: daytime, dtype: int64
```

[24]:
```python
# Define 'pm_rush()' conversion function [16:00-20:00)
def pm_rush(hour):
    if 16 < hour["pm_rush"] < 20:
        result = 1
    else:
        result = 0
    return result
```

[25]:
```python
# Apply 'pm_rush()' function to the 'pm_rush' series
df1["pm_rush"] = df1.apply(pm_rush, axis= 1)
df1["pm_rush"].head(5)
```

[25]:
```
0    0
1    0
2    0
3    0
4    0
Name: pm_rush, dtype: int64
```

[26]:
```python
# Define 'nighttime()' conversion function [20:00-06:00)
def nighttime(hour):
```

```
    if 20 < hour["nighttime"] <= 24:
        result = 1
    elif 0 <= hour["nighttime"] < 6:
        result = 1
    else:
        result = 0
    return result
```

[27]:
```
# Apply 'nighttime' function to the 'nighttime' series
df1["nighttime"] = df1.apply(nighttime, axis= 1)
df1["nighttime"].head(5)
```

[27]:
```
0    0
1    0
2    0
3    0
4    0
Name: nighttime, dtype: int64
```

Create month column Now, create a `month` column that contains only the abbreviated name of the month when each passenger was picked up, then convert the result to lowercase.

HINT

Refer to the strftime cheatsheet for help.

[28]:
```
# Create 'month' col
df1["month"] = df1["tpep_pickup_datetime"].dt.strftime("%b").str.lower()
```

Examine the first five rows of your dataframe.

[29]:
```
df1.head(5)
```

[29]:
```
   Unnamed: 0  VendorID tpep_pickup_datetime tpep_dropoff_datetime  \
0    24870114         2  2017-03-25 08:55:43   2017-03-25 09:09:47
1    35634249         1  2017-04-11 14:53:28   2017-04-11 15:19:58
2   106203690         1  2017-12-15 07:26:56   2017-12-15 07:34:08
3    38942136         2  2017-05-07 13:17:59   2017-05-07 13:48:14
4    23345809         2  2017-03-25 20:34:11   2017-03-25 20:42:11

   passenger_count  trip_distance  RatecodeID store_and_fwd_flag  \
0                6           3.34           1                  N
1                1           1.80           1                  N
2                1           1.00           1                  N
3                1           3.70           1                  N
4                6           2.30           1                  N

   PULocationID  DOLocationID  payment_type  fare_amount  extra  mta_tax  \
```

```
0          100          231               1         13.0    0.0      0.5
1          186           43               1         16.0    0.0      0.5
2          262          236               1          6.5    0.0      0.5
3          188           97               1         20.5    0.0      0.5
4          161          236               1          9.0    0.5      0.5

   tip_amount  tolls_amount  improvement_surcharge  total_amount  \
0        2.76           0.0                    0.3         16.56
1        4.00           0.0                    0.3         20.80
2        1.45           0.0                    0.3          8.75
3        6.39           0.0                    0.3         27.69
4        2.06           0.0                    0.3         12.36

   mean_duration  mean_distance  predicted_fare  tip_percent  generous  \
0      22.847222       3.521667       16.434245        0.200         1
1      24.470370       3.108889       16.052218        0.238         1
2       7.250000       0.881429        7.053706        0.199         0
3      30.250000       3.700000       18.731650        0.300         1
4      11.855376       2.052258       10.441351        0.200         1

        day  am_rush  daytime  pm_rush  nighttime month
0  saturday        1        0        0          0   mar
1   tuesday        0        1        0          0   apr
2    friday        1        0        0          0   dec
3    sunday        0        1        0          0   may
4  saturday        0        0        0          0   mar
```

[30]:
```
df1.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 15265 entries, 0 to 15264
Data columns (total 29 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   Unnamed: 0             15265 non-null  int64
 1   VendorID               15265 non-null  int64
 2   tpep_pickup_datetime   15265 non-null  datetime64[ns]
 3   tpep_dropoff_datetime  15265 non-null  datetime64[ns]
 4   passenger_count        15265 non-null  int64
 5   trip_distance          15265 non-null  float64
 6   RatecodeID             15265 non-null  int64
 7   store_and_fwd_flag     15265 non-null  object
 8   PULocationID           15265 non-null  int64
 9   DOLocationID           15265 non-null  int64
 10  payment_type           15265 non-null  int64
 11  fare_amount            15265 non-null  float64
 12  extra                  15265 non-null  float64
```

```
 13   mta_tax                 15265 non-null   float64
 14   tip_amount              15265 non-null   float64
 15   tolls_amount            15265 non-null   float64
 16   improvement_surcharge   15265 non-null   float64
 17   total_amount            15265 non-null   float64
 18   mean_duration           15265 non-null   float64
 19   mean_distance           15265 non-null   float64
 20   predicted_fare          15265 non-null   float64
 21   tip_percent             15262 non-null   float64
 22   generous                15265 non-null   int64
 23   day                     15265 non-null   object
 24   am_rush                 15265 non-null   int64
 25   daytime                 15265 non-null   int64
 26   pm_rush                 15265 non-null   int64
 27   nighttime               15265 non-null   int64
 28   month                   15265 non-null   object
dtypes: datetime64[ns](2), float64(12), int64(12), object(3)
memory usage: 3.4+ MB
```

Drop columns Drop redundant and irrelevant columns as well as those that would not be available when the model is deployed. This includes information like payment type, trip distance, tip amount, tip percentage, total amount, toll amount, etc. The target variable (`generous`) must remain in the data because it will get isolated as the y data for modeling.

[31]:
```
# Drop columns
dropped = ["Unnamed: 0", "tpep_pickup_datetime", "tpep_dropoff_datetime",␣
 ↪"trip_distance", "store_and_fwd_flag", "payment_type", "fare_amount",␣
 ↪"extra", "mta_tax", "tip_amount", "tolls_amount", "improvement_surcharge",␣
 ↪"total_amount", "tip_percent"]

df1 = df1.drop(dropped, axis= 1)
df1.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 15265 entries, 0 to 15264
Data columns (total 15 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   VendorID         15265 non-null  int64
 1   passenger_count  15265 non-null  int64
 2   RatecodeID       15265 non-null  int64
 3   PULocationID     15265 non-null  int64
 4   DOLocationID     15265 non-null  int64
 5   mean_duration    15265 non-null  float64
 6   mean_distance    15265 non-null  float64
 7   predicted_fare   15265 non-null  float64
 8   generous         15265 non-null  int64
```

```
 9   day              15265 non-null  object
 10  am_rush          15265 non-null  int64
 11  daytime          15265 non-null  int64
 12  pm_rush          15265 non-null  int64
 13  nighttime        15265 non-null  int64
 14  month            15265 non-null  object
dtypes: float64(3), int64(10), object(2)
memory usage: 1.7+ MB
```

Variable encoding Many of the columns are categorical and will need to be dummied (converted to binary). Some of these columns are numeric, but they actually encode categorical information, such as `RatecodeID` and the pickup and dropoff locations. To make these columns recognizable to the `get_dummies()` function as categorical variables, you'll first need to convert them to `type(str)`.

1. Define a variable called `cols_to_str`, which is a list of the numeric columns that contain categorical information and must be converted to string: `RatecodeID`, `PULocationID`, `DOLocationID`.
2. Write a for loop that converts each column in `cols_to_str` to string.

[32]:
```
# 1. Define list of cols to convert to string
cols_to_str = ["RatecodeID", "PULocationID", "DOLocationID"]

# 2. Convert each column to string
for col in cols_to_str:
    df1[col] = df1[col].astype("object")
```

[33]:
```
df1.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 15265 entries, 0 to 15264
Data columns (total 15 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   VendorID         15265 non-null  int64
 1   passenger_count  15265 non-null  int64
 2   RatecodeID       15265 non-null  object
 3   PULocationID     15265 non-null  object
 4   DOLocationID     15265 non-null  object
 5   mean_duration    15265 non-null  float64
 6   mean_distance    15265 non-null  float64
 7   predicted_fare   15265 non-null  float64
 8   generous         15265 non-null  int64
 9   day              15265 non-null  object
 10  am_rush          15265 non-null  int64
 11  daytime          15265 non-null  int64
 12  pm_rush          15265 non-null  int64
 13  nighttime        15265 non-null  int64
 14  month            15265 non-null  object
```

```
dtypes: float64(3), int64(7), object(5)
memory usage: 1.7+ MB
```

HINT

To convert to string, use `astype(str)` on the column.

Now convert all the categorical columns to binary.

1. Call `get_dummies()` on the dataframe and assign the results back to a new dataframe called `df2`.

[34]:
```python
# Convert categoricals to binary
df2 = pd.get_dummies(df1)
df2.shape
```

[34]:
```
(15265, 352)
```

Evaluation metric Before modeling, you must decide on an evaluation metric.

1. Examine the class balance of your target variable.

[35]:
```python
# Get class balance of 'generous' col
round(df1["generous"].value_counts(normalize= True) * 100, 2)
```

[35]:
```
1    52.64
0    47.36
Name: generous, dtype: float64
```

A little over half of the customers in this dataset were "generous" (tipped 20%). The dataset is very nearly balanced.

To determine a metric, consider the cost of both kinds of model error: * False positives (the model predicts a tip 20%, but the customer does not give one) * False negatives (the model predicts a tip < 20%, but the customer gives more)

False positives are worse for cab drivers, because they would pick up a customer expecting a good tip and then not receive one, frustrating the driver.

False negatives are worse for customers, because a cab driver would likely pick up a different customer who was predicted to tip more—even when the original customer would have tipped generously.

The stakes are relatively even. You want to help taxi drivers make more money, but you don't want this to anger customers. Your metric should weigh both precision and recall equally. Which metric is this?

This metric is f1 score, which is the harmonic mean between precision and recall.

4.3 PACE: Construct

Consider the questions in your PACE Strategy Document to reflect on the Construct stage.

4.3.1 Task 3. Modeling

Split the data Now you're ready to model. The only remaining step is to split the data into features/target variable and training/testing data.

1. Define a variable `y` that isolates the target variable (`generous`).
2. Define a variable `X` that isolates the features.
3. Split the data into training and testing sets. Put 20% of the samples into the test set, stratify the data, and set the random state.

[36]:
```
df2.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 15265 entries, 0 to 15264
Columns: 352 entries, VendorID to month_sep
dtypes: float64(3), int64(7), uint8(342)
memory usage: 6.1 MB
```

[37]:
```
df2.select_dtypes("int64").columns
```

[37]:
```
Index(['VendorID', 'passenger_count', 'generous', 'am_rush', 'daytime',
       'pm_rush', 'nighttime'],
      dtype='object')
```

[38]:
```
# Isolate target variable (y)
y = df2["generous"]

# Isolate the features (X)
X = df2.drop("generous", axis= 1)
```

[39]:
```
# Split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify= y,␣
↪test_size= 0.2, random_state= 42)
```

[40]:
```
print(X_train.shape)
print()
print(y_train.shape)
```

```
(12212, 351)

(12212,)
```

Random forest Begin with using `GridSearchCV` to tune a random forest model.

1. Instantiate the random forest classifier `rf` and set the random state.
2. Create a dictionary `cv_params` of any of the following hyperparameters and their corresponding values to tune. The more you tune, the better your model will fit the data, but the longer it will take.

- `max_depth`
- `max_features`
- `max_samples`
- `min_samples_leaf`
- `min_samples_split`
- `n_estimators`

3. Define a set `scoring` of scoring metrics for GridSearch to capture (precision, recall, F1 score, and accuracy).

4. Instantiate the `GridSearchCV` object `rf1`. Pass to it as arguments:

- estimator=`rf`
- param_grid=`cv_params`
- scoring=`scoring`
- cv: define the number of you cross-validation folds you want (`cv=_`)
- refit: indicate which evaluation metric you want to use to select the model (`refit=_`)

Note: `refit` should be set to `'f1'`.

[41]:
```
# 1. Instantiate the random forest classifier
rf = RandomForestClassifier(random_state= 17)

# 2. Create a dictionary of hyperparameters to tune
cv_params = {"max_depth": [5, 6, 7, 8, 9], "max_features": [0.8, 1],␣
↪"max_samples": [0.8, 1],\
             "min_samples_leaf": [3, 5, 7, 9, None], "n_estimators": [200]}

# 3. Define a set of scoring metrics to capture
scoring = {"precision", "recall", "f1", "accuracy"}

# 4. Instantiate the GridSearchCV object
rf1 = GridSearchCV(estimator= rf, param_grid= cv_params, scoring= scoring, cv=␣
↪4, refit= "f1", n_jobs= -1, verbose= 1)
```

Now fit the model to the training data. Note that, depending on how many options you include in your search grid and the number of cross-validation folds you select, this could take a very long time—even hours. If you use 4-fold validation and include only one possible value for each hyperparameter and grow 300 trees to full depth, it should take about 5 minutes. If you add another value for GridSearch to check for, say, `min_samples_split` (so all hyperparameters now have 1 value except for `min_samples_split`, which has 2 possibilities), it would double the time to ~10 minutes. Each additional parameter would approximately double the time.

[42]:
```
%%time

rf1.fit(X_train, y_train)
```

```
Fitting 4 folds for each of 100 candidates, totalling 400 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 2 concurrent workers.
[Parallel(n_jobs=-1)]: Done  46 tasks      | elapsed:   51.9s
[Parallel(n_jobs=-1)]: Done 196 tasks      | elapsed:  3.1min
[Parallel(n_jobs=-1)]: Done 400 out of 400 | elapsed:  5.9min finished

CPU times: user 9.01 s, sys: 335 ms, total: 9.34 s
Wall time: 5min 59s
```

[42]:
```
GridSearchCV(cv=4, error_score=nan,
             estimator=RandomForestClassifier(bootstrap=True, ccp_alpha=0.0,
                                              class_weight=None,
                                              criterion='gini', max_depth=None,
                                              max_features='auto',
                                              max_leaf_nodes=None,
                                              max_samples=None,
                                              min_impurity_decrease=0.0,
                                              min_impurity_split=None,
                                              min_samples_leaf=1,
                                              min_samples_split=2,
                                              min_weight_fraction_leaf=0.0,
                                              n_estimators=100, n_jobs=None,
                                              oob_score=False, random_state=17,
                                              verbose=0, warm_start=False),
             iid='deprecated', n_jobs=-1,
             param_grid={'max_depth': [5, 6, 7, 8, 9], 'max_features': [0.8, 1],
                         'max_samples': [0.8, 1],
                         'min_samples_leaf': [3, 5, 7, 9, None],
                         'n_estimators': [200]},
             pre_dispatch='2*n_jobs', refit='f1', return_train_score=False,
             scoring={'precision', 'accuracy', 'recall', 'f1'}, verbose=1)
```

HINT

If you get a warning that a metric is 0 due to no predicted samples, think about how many features you're sampling with `max_features`. How many features are in the dataset? How many are likely predictive enough to give good predictions within the number of splits you've allowed (determined by the `max_depth` hyperparameter)? Consider increasing `max_features`.

If you want, use `pickle` to save your models and read them back in. This can be particularly helpful when performing a search over many possible hyperparameter values.

[43]:
```python
import pickle

# Define a path to the folder where you want to save the model
path = '/home/jovyan/work/'
```

```
[44]: def write_pickle(path, model_object, save_name:str):
          '''
          save_name is a string.
          '''
          with open(path + save_name + '.pickle', 'wb') as to_write:
              pickle.dump(model_object, to_write)
```

```
[45]: write_pickle(path, rf1, "random_forest_1")
```

```
[46]: def read_pickle(path, saved_model_name:str):
          '''
          saved_model_name is a string.
          '''
          with open(path + saved_model_name + '.pickle', 'rb') as to_read:
              model = pickle.load(to_read)

              return model
```

```
[47]: read_pickle(path, "random_forest_1")
```

```
[47]: GridSearchCV(cv=4, error_score=nan,
                   estimator=RandomForestClassifier(bootstrap=True, ccp_alpha=0.0,
                                                    class_weight=None,
                                                    criterion='gini', max_depth=None,
                                                    max_features='auto',
                                                    max_leaf_nodes=None,
                                                    max_samples=None,
                                                    min_impurity_decrease=0.0,
                                                    min_impurity_split=None,
                                                    min_samples_leaf=1,
                                                    min_samples_split=2,
                                                    min_weight_fraction_leaf=0.0,
                                                    n_estimators=100, n_jobs=None,
                                                    oob_score=False, random_state=17,
                                                    verbose=0, warm_start=False),
                   iid='deprecated', n_jobs=-1,
                   param_grid={'max_depth': [5, 6, 7, 8, 9], 'max_features': [0.8, 1],
                               'max_samples': [0.8, 1],
                               'min_samples_leaf': [3, 5, 7, 9, None],
                               'n_estimators': [200]},
                   pre_dispatch='2*n_jobs', refit='f1', return_train_score=False,
                   scoring={'precision', 'accuracy', 'recall', 'f1'}, verbose=1)
```

Examine the best average score across all the validation folds.

```
[48]: # Examine best score
      print(round(rf1.best_score_, 3))
```

0.749

Examine the best combination of hyperparameters.

[49]:
```python
print(rf1.best_params_)
```

```
{'max_depth': 5, 'max_features': 0.8, 'max_samples': 0.8, 'min_samples_leaf': 7,
'n_estimators': 200}
```

Use the `make_results()` function to output all of the scores of your model. Note that it accepts three arguments.

HINT

To learn more about how this function accesses the cross-validation results, refer to the `GridSearchCV` scikit-learn documentation for the `cv_results_` attribute.

[50]:
```python
def make_results(model_name:str, model_object, metric:str):
    '''
    Arguments:
    model_name (string): what you want the model to be called in the output␣
↪table
    model_object: a fit GridSearchCV object
    metric (string): precision, recall, f1, or accuracy

    Returns a pandas df with the F1, recall, precision, and accuracy scores
    for the model with the best mean 'metric' score across all validation folds.
    '''

    # Create dictionary that maps input metric to actual metric name in␣
↪GridSearchCV
    metric_dict = {'precision': 'mean_test_precision',
                   'recall': 'mean_test_recall',
                   'f1': 'mean_test_f1',
                   'accuracy': 'mean_test_accuracy',
                   }

    # Get all the results from the CV and put them in a df
    cv_results = pd.DataFrame(model_object.cv_results_)

    # Isolate the row of the df with the max(metric) score
    best_estimator_results = cv_results.iloc[cv_results[metric_dict[metric]].
↪idxmax(), :]

    # Extract Accuracy, precision, recall, and f1 score from that row
    f1 = best_estimator_results.mean_test_f1
    recall = best_estimator_results.mean_test_recall
    precision = best_estimator_results.mean_test_precision
    accuracy = best_estimator_results.mean_test_accuracy
```

```
    # Create table of results
    table = pd.DataFrame({'model': [model_name],
                          'precision': [precision],
                          'recall': [recall],
                          'F1': [f1],
                          'accuracy': [accuracy]})

    return table
```

Call `make_results()` on the GridSearch object.

[51]:
```
rf_validation_Scores = make_results("Random_Forest_1", rf1, "f1")
rf_validation_Scores
```

[51]:

	model	precision	recall	F1	accuracy
0	Random_Forest_1	0.693976	0.812539	0.748538	0.71266

Your results should produce an acceptable model across the board. Typically scores of 0.65 or better are considered acceptable, but this is always dependent on your use case. Optional: try to improve the scores. It's worth trying, especially to practice searching over different hyperparameters.

HINT

For example, if the available values for `min_samples_split` were [2, 3, 4] and GridSearch identified the best value as 4, consider trying [4, 5, 6] this time.

Use your model to predict on the test data. Assign the results to a variable called `rf_preds`.

HINT

You cannot call `predict()` on the GridSearchCV object directly. You must call it on the `best_estimator_`.

For this project, you will use several models to predict on the test data. Remember that this decision comes with a trade-off. What is the benefit of this? What is the drawback?

The benefit of comparing different models is to get the best predictive approach.

[52]:
```
# Get scores on test data
rf_preds = rf1.predict(X_test)
```

Use the below `get_test_scores()` function you will use to output the scores of the model on the test data.

[53]:
```
def get_test_scores(model_name:str, preds, y_test_data):
    '''
    Generate a table of test scores.

    In:
    model_name (string): Your choice: how the model will be named in the output␣
 ↪table
```

```
    preds: numpy array of test predictions
    y_test_data: numpy array of y_test data

    Out:
    table: a pandas df of precision, recall, f1, and accuracy scores for your␣
↪model
    '''
    accuracy = accuracy_score(y_test_data, preds)
    precision = precision_score(y_test_data, preds)
    recall = recall_score(y_test_data, preds)
    f1 = f1_score(y_test_data, preds)

    table = pd.DataFrame({'model': [model_name],
                          'precision': [precision],
                          'recall': [recall],
                          'F1': [f1],
                          'accuracy': [accuracy]})

    return table
```

1. Use the `get_test_scores()` function to generate the scores on the test data. Assign the results to `rf_test_scores`.
2. Call `rf_test_scores` to output the results.

RF test results

[54]:
```
# Get scores on test data
rf_test_scores = get_test_scores("Random_Forest_1", rf_preds, y_test)
rf_test_scores
```

[54]:

	model	precision	recall	F1	accuracy
0	Random_Forest_1	0.684567	0.81705	0.744965	0.705536

[55]:
```
rf_validation_Scores
```

[55]:

	model	precision	recall	F1	accuracy
0	Random_Forest_1	0.693976	0.812539	0.748538	0.71266

Question: How do your test results compare to your validation results?

Both of the two scores are relatively equal, which is a good indicator that our model generalizes well on unseen data.

XGBoost Try to improve your scores using an XGBoost model.

1. Instantiate the XGBoost classifier `xgb` and set `objective='binary:logistic'`. Also set the random state.

2. Create a dictionary `cv_params` of the following hyperparameters and their corresponding values to tune:

- `max_depth`
- `min_child_weight`
- `learning_rate`
- `n_estimators`

3. Define a set `scoring` of scoring metrics for grid search to capture (precision, recall, F1 score, and accuracy).

4. Instantiate the `GridSearchCV` object `xgb1`. Pass to it as arguments:

- estimator=`xgb`
- param_grid=`cv_params`
- scoring=`scoring`
- cv: define the number of cross-validation folds you want (`cv=_`)
- refit: indicate which evaluation metric you want to use to select the model (`refit='f1'`)

[57]:
```
# 1. Instantiate the XGBoost classifier
xgb = XGBClassifier(objective= "binary:logistic", random_state= 17)

# 2. Create a dictionary of hyperparameters to tune
cv_params = {"max_depth": [5, 7, 9], "min_child_weight": [3, 5, 7],␣
↪"learning_rate": [0.05, 0.1, 0.2], "n_estimators": [300]}

# 3. Define a set of scoring metrics to capture
scoring = {"precision", "recall", "f1", "accuracy"}

# 4. Instantiate the GridSearchCV object
xgb1 = GridSearchCV(estimator= xgb, param_grid= cv_params, scoring= scoring,␣
↪cv= 4, refit= "f1", n_jobs= -1, verbose= 1)
```

Now fit the model to the `X_train` and `y_train` data.

[58]:
```
%%time

xgb1.fit(X_train, y_train)
```

```
Fitting 4 folds for each of 27 candidates, totalling 108 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 2 concurrent workers.
[Parallel(n_jobs=-1)]: Done  46 tasks      | elapsed: 13.2min
[Parallel(n_jobs=-1)]: Done 108 out of 108 | elapsed: 33.3min finished

CPU times: user 41.2 s, sys: 176 ms, total: 41.4 s
Wall time: 33min 37s
```

[58]:
```
GridSearchCV(cv=4, error_score=nan,
             estimator=XGBClassifier(base_score=None, booster=None,
                                     callbacks=None, colsample_bylevel=None,
```

```
                                    colsample_bynode=None,
                                    colsample_bytree=None,
                                    early_stopping_rounds=None,
                                    enable_categorical=False, eval_metric=None,
                                    gamma=None, gpu_id=None, grow_policy=None,
                                    importance_type=None,
                                    interaction_constraints=None,
                                    learning_rate=None, max…
                                    num_parallel_tree=None,
                                    objective='binary:logistic',
                                    predictor=None, random_state=17,
                                    reg_alpha=None, …),
             iid='deprecated', n_jobs=-1,
             param_grid={'learning_rate': [0.05, 0.1, 0.2],
                         'max_depth': [5, 7, 9], 'min_child_weight': [3, 5, 7],
                         'n_estimators': [300]},
             pre_dispatch='2*n_jobs', refit='f1', return_train_score=False,
             scoring={'precision', 'accuracy', 'recall', 'f1'}, verbose=1)
```

Get the best score from this model.

[59]:
```python
# Examine best score
print(xgb1.best_score_)
```

```
0.7412087145777077
```

And the best parameters.

[60]:
```python
# Examine best parameters
print(xgb1.best_params_)
```

```
{'learning_rate': 0.05, 'max_depth': 5, 'min_child_weight': 7, 'n_estimators':
300}
```

XGB CV Results Use the `make_results()` function to output all of the scores of your model. Note that it accepts three arguments.

[61]:
```python
# Call 'make_results()' on the GridSearch object
xgb_valuation_scores = make_results("XGBoost_1", xgb1, "f1")
xgb_valuation_scores
```

[61]:

	model	precision	recall	F1	accuracy
0	XGBoost_1	0.692954	0.796826	0.741209	0.707173

Use your model to predict on the test data. Assign the results to a variable called `xgb_preds`.

HINT

You cannot call `predict()` on the GridSearchCV object directly. You must call it on the `best_estimator_`.

[62]:
```
# Get scores on test data
xgb_preds = xgb1.predict(X_test)
```

XGB test results

1. Use the `get_test_scores()` function to generate the scores on the test data. Assign the results to `xgb_test_scores`.
2. Call `xgb_test_scores` to output the results.

[65]:
```
# Get scores on test data
xgb_test_scores = get_test_scores("XGBoost_1", xgb_preds, y_test)
xgb_test_scores
```

[65]:

	model	precision	recall	F1	accuracy
0	XGBoost_1	0.687068	0.80336	0.740677	0.703898

[66]:
```
rf_test_scores
```

[66]:

	model	precision	recall	F1	accuracy
0	Random_Forest_1	0.684567	0.81705	0.744965	0.705536

Question: Compare these scores to the random forest test scores. What do you notice? Which model would you choose?

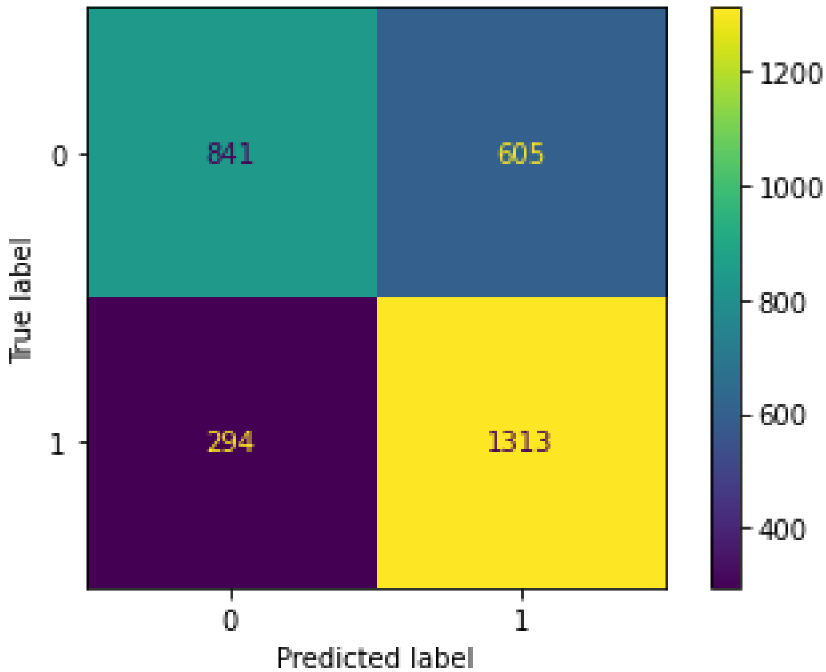
The test scores for the two models are approximately equal to each other, however the scores for the random forest model are slightly higher than those for the gradient boost model.

Plot a confusion matrix of the model's predictions on the test data.

[67]:
```
# Generate array of values for confusion matrix
cm = confusion_matrix(y_test, rf_preds, labels= rf1.classes_)

# Plot confusion matrix
disp = ConfusionMatrixDisplay(confusion_matrix= cm, display_labels= rf1.
 ↪classes_)
disp.plot(values_format='')
```

[67]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at
0x7fc69f8acbd0>
```



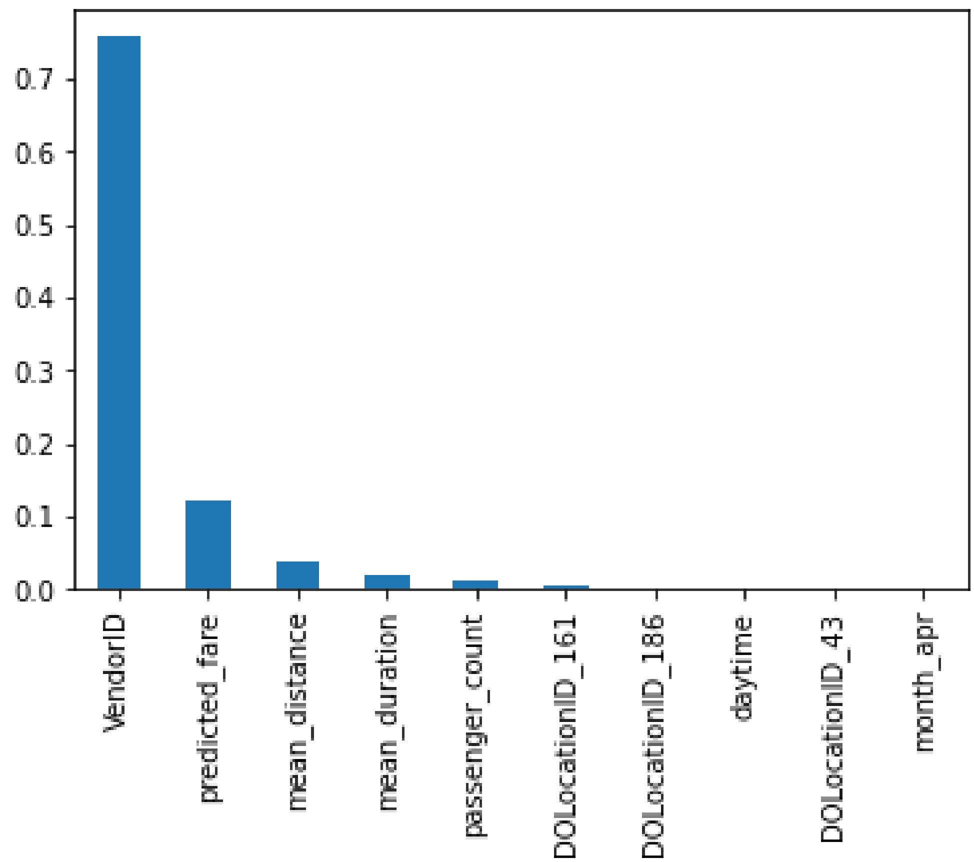
Question: What type of errors are more common for your model?

The model is producing more false positives than the false negatives, which means that the model would predict that some customers are generous, while in fact they are not.

Feature importance Use the `feature_importances_` attribute of the best estimator object to inspect the features of your final model. You can then sort them and plot the most important ones.

[70]:
```
importances = rf1.best_estimator_.feature_importances_
rf_importances = pd.Series(importances, index= X_test.columns)
rf_importances = rf_importances.sort_values(ascending=False)[:10]

rf_importances.plot.bar()
ax.set_title('Feature importances')
ax.set_ylabel('Mean decrease in impurity')
fig.tight_layout()
```



4.4 PACE: Execute

Consider the questions in your PACE Strategy Document to reflect on the Execute stage.

4.4.1 Task 4. Conclusion

In this step, use the results of the models above to formulate a conclusion. Consider the following questions:

1. **Would you recommend using this model? Why or why not?**
2. **What was your model doing? Can you explain how it was making predictions?**
3. **Are there new features that you can engineer that might improve model performance?**
4. **What features would you want to have that would likely improve the performance of your model?**

Remember, sometimes your data simply will not be predictive of your chosen target. This is common. Machine learning is a powerful tool, but it is not magic. If your data does not contain predictive signal, even the most complex algorithm will not be able to deliver consistent and accurate predictions. Do not be afraid to draw this conclusion. Even if you cannot use the model to make strong predictions, was the work done in vain? Consider any insights that you could report back to stakeholders.

Congratulations! You've completed this lab. However, you may not notice a green check mark next to this item on Coursera's platform. Please continue your progress regardless of the check mark. Just click on the "save" icon at the top of this notebook to ensure your work has been logged.