NYC TLC Project Part 5

November 5, 2024

1 NYC TLC Project Part 5

Build a machine learning model to predict if a customer will not leave a tip. This can further aid in using the model in an app that will alert taxi drivers to customers who are unlikely to tip, since drivers depend on tips.

2 Build a machine learning model

In this project, we will use 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

Part 2: Feature engineering

Part 3: Modeling

Ideally, we'd have behavioral history for each customer, so we could know how much they tipped on previous taxi rides. We'd also want times, dates, and locations of both pickups and dropoffs, estimated fares, and payment method.

The target variable would be a binary variable (1 or 0) that indicates whether or not the customer is expected to tip 20%.

This is a supervised learning, classification task. We could use accuracy, precision, recall, F-score, area under the ROC curve, or a number of other metrics. However, we don't have enough information at this time to know which are most appropriate. We need to know the class balance of the target variable.

2.0.1 Task 1. Imports and data loading

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

[1]:
```python
# Import packages and libraries

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split, PredefinedSplit,␣
 ↪GridSearchCV
from sklearn.metrics import roc_auc_score, roc_curve, RocCurveDisplay
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
 ↪f1_score, confusion_matrix, ConfusionMatrixDisplay

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)
```

[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(10)
```

[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
5    23345809         2   03/25/2017 8:34:11 PM   03/25/2017 8:42:11 PM
6    37660487         2   05/03/2017 7:04:09 PM   05/03/2017 8:03:47 PM
7    69059411         2   08/15/2017 5:41:06 PM   08/15/2017 6:03:05 PM
8     8433159         2   02/04/2017 4:17:07 PM   02/04/2017 4:29:14 PM
9    95294817         1   11/10/2017 3:20:29 PM   11/10/2017 3:40:55 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
5                6        2.30           1                  N
6                1       12.83           1                  N
7                1        2.98           1                  N
8                1        1.20           1                  N
9                1        1.60           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
5           161           236             1          9.0    0.5      0.5
6            79           241             1         47.5    1.0      0.5
7           237           114             1         16.0    1.0      0.5
8           234           249             2          9.0    0.0      0.5
9           239           237             1         13.0    0.0      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        2.06           0.0                    0.3         12.36
6        9.86           0.0                    0.3         59.16
7        1.78           0.0                    0.3         19.58
8        0.00           0.0                    0.3          9.80
9        2.75           0.0                    0.3         16.55
```

Inspect the first few rows of `nyc_preds_means`.

[5]:
```
# Inspect the first few rows of `nyc_preds_means`

nyc_preds_means.head(10)
```

[5]:
```
   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
5      11.855376       2.052258       10.441351
6      59.633333      12.830000       45.374542
7      26.437500       4.022500       18.555128
```

```
8       7.873457       1.019259       7.151511
9      10.541111       1.580000       9.122755
```

Join the two dataframes Join the two dataframes using any method.

[6]:
```python
# Merge datasets
df0 = df0.merge(nyc_preds_means, left_index=True, right_index=True)
df0.head()
```

[6]:
```
   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
```

2.0.2 Task 2. Feature engineering

[7]:
```
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
```

We know from our EDA that customers who pay cash generally have a tip amount of $0. To meet the modeling objective, we'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.

[8]:
```
# Subset the data to isolate only customers who paid by credit card
df1 = df0[df0['payment_type']==1]
```

Target Notice that there isn't a column that indicates tip percent, which is what we need to create the target variable. We'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.

[10]:
```python
# Create tip % col

df1['tip_percent'] = round(df1['tip_amount']/(df1['total_amount']-df1['tip_amount']),3)
```

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).

[11]:
```python
# Create 'generous' col (target)
df1['generous'] = df1['tip_percent']
df1['generous'] = (df1['generous']>=0.2)
df1['generous'] = df1['generous'].astype(int)
```

Create day column Convert the `tpep_pickup_datetime` and `tpep_dropoff_datetime` columns to datetime.

[12]:
```python
# Convert pickup and dropoff cols to datetime

df1['tpep_pickup_datetime'] = pd.to_datetime(df1['tpep_pickup_datetime'], format='%m/%d/%Y %I:%M:%S %p')
df1['tpep_dropoff_datetime'] = pd.to_datetime(df1['tpep_dropoff_datetime'], format='%m/%d/%Y %I:%M:%S %p')
```

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

[13]:
```python
# Create a 'day' col

df1['day'] = df1['tpep_pickup_datetime'].dt.day_name().str.lower()
```

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.

[14]:
```
# 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
```

[51]:
```
# Define 'am_rush()' conversion function [06:00-10:00)

def am_rush(hour):
    if 6 <= hour['am_rush'] < 10:
        val = 1
    else:
        val = 0
    return val
```

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.

[52]:
```
# Apply 'am_rush' function to the 'am_rush' series

df1['am_rush'] = df1.apply(am_rush, axis=1)
df1['am_rush'].head()
```

[52]:
```
0    1
1    0
2    1
3    0
5    0
Name: am_rush, dtype: int64
```

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

[53]:
```python
# Define 'daytime()' conversion function [10:00-16:00)

def daytime(hour):
    if 10 <= hour['daytime'] < 16:
        val = 1
    else:
        val = 0
    return val
```

[54]:
```python
# Apply 'daytime()' function to the 'daytime' series

df1['daytime'] = df1.apply(daytime, axis=1)
df1['daytime'].head()
```

[54]:
```
0    0
1    1
2    0
3    1
5    0
Name: daytime, dtype: int64
```

[55]:
```python
# Define 'pm_rush()' conversion function [16:00-20:00)

def pm_rush(hour):
    if 16 <= hour['pm_rush'] < 20:
        val = 1
    else:
        val = 0
    return val
```

[58]:
```python
# Apply 'pm_rush()' function to the 'pm_rush' series

df1['pm_rush'] = df1.apply(pm_rush, axis=1)
df1['pm_rush'].head()
```

[58]:
```
0    0
1    0
2    0
3    0
5    0
Name: pm_rush, dtype: int64
```

[59]:
```python
# Define 'nighttime()' conversion function [20:00-06:00)

def nighttime(hour):
    if 20 <= hour['nighttime'] < 24:
        val = 1
```

```
    elif 0 <= hour['nighttime'] < 6:
        val = 1
    else:
        val = 0
    return val
```

[60]:
```
# Apply 'nighttime' function to the 'nighttime' series

df1['nighttime'] = df1.apply(nighttime, axis=1)
df1['nighttime'].head()
```

[60]:
```
0    0
1    0
2    0
3    0
5    1
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.

[62]:
```
# Create 'month' col

df1['month'] = df1['tpep_pickup_datetime'].dt.strftime('%b').str.lower()
```

Examine the first five rows of your dataframe.

[63]:
```
#==> ENTER YOUR CODE HERE
df1.head()
```

[63]:
```
   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
5    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
5                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
5          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
5        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
5      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
5  saturday        0        0        0          1   mar
```

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.

[64]:
```python
# Drop columns

drop_cols = ['Unnamed: 0', 'tpep_pickup_datetime', 'tpep_dropoff_datetime',
             'payment_type', '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(drop_cols, axis=1)
df1.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 15265 entries, 0 to 22698
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: 1.9+ 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, we'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.

[65]:
```
# 1. Define list of cols to convert to string

cols_to_str = ['RatecodeID', 'PULocationID', 'DOLocationID', 'VendorID']

# 2. Convert each column to string

for col in cols_to_str:
    df1[col] = df1[col].astype('str')
```

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`.

[66]:
```
# Convert categoricals to binary

df2 = pd.get_dummies(df1, drop_first=True)
df2.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 15265 entries, 0 to 22698
Columns: 347 entries, passenger_count to month_sep
```

```
dtypes: float64(3), int64(6), uint8(338)
memory usage: 6.1 MB
```

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

1. Examine the class balance of our target variable.

[69]:
```python
# Get class balance of 'generous' col

df2['generous'].value_counts(normalize=True)
```

[69]:
```
1    0.526368
0    0.473632
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. We want to help taxi drivers make more money, but we don't want this to anger customers. Our metric should weigh both precision and recall equally. So we use F1 score.

2.0.3 Task 3. Modeling

Split the data 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.

[70]:
```python
# Isolate target variable (y)

y = df2["generous"]

# Isolate the features (X)

x = df2.drop('generous', axis=1)
```

```
# 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)
```

Random forest

[73]:
```
# 1. Instantiate the random forest classifier

rf = RandomForestClassifier(random_state=42)

# 2. Create a dictionary of hyperparameters to tune

cv_params = {
    'max_depth': [5, 10, 15],
    'max_features': ['sqrt'],
    'max_samples': [0.8],
    'min_samples_leaf': [1, 3],
    'min_samples_split': [2, 3, 5],
    'n_estimators': [200, 300]
}

# 3. Define a set of scoring metrics to capture

scoring = {'accuracy', 'precision', 'recall', 'f1'}

# 4. Instantiate the GridSearchCV object

rf1 = GridSearchCV(rf, cv_params, scoring=scoring, cv=5, refit='f1')
```

[74]:
```
%%time

rf1.fit(x_train, y_train)
```

```
CPU times: user 6min 26s, sys: 666 ms, total: 6min 27s
Wall time: 6min 27s
```

[74]:
```
GridSearchCV(cv=5, 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=42,
                                                  verbose=0, warm_start=False),
             iid='deprecated', n_jobs=None,
             param_grid={'max_depth': [5, 10, 15], 'max_features': ['sqrt'],
                         'max_samples': [0.8], 'min_samples_leaf': [1, 3],
                         'min_samples_split': [2, 3, 5],
                         'n_estimators': [200, 300]},
             pre_dispatch='2*n_jobs', refit='f1', return_train_score=False,
             scoring={'precision', 'recall', 'f1', 'accuracy'}, verbose=0)
```

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

[76]:
```
import pickle

# Define a path to the folder where you want to save the model
path = 'C:/Users/disis/Downloads'
```

[77]:
```
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)
```

[78]:
```
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
```

Examine the best average score across all the validation folds.

[79]:
```
# Examine best score

rf1.best_score_
```

[79]:
```
0.7498766982595237
```

Examine the best combination of hyperparameters.

[80]:
```
rf1.best_params_
```

[80]:
```
{'max_depth': 5,
 'max_features': 'sqrt',
 'max_samples': 0.8,
 'min_samples_leaf': 1,
 'min_samples_split': 5,
 'n_estimators': 200}
```

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

[81]:
```
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.

[84]:
```
results = make_results('RF CV', rf1, 'f1')
results
```

[84]:

	model	precision	recall	F1	accuracy
0	RF CV	0.691642	0.818917	0.749877	0.712413

Our 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 our use case.

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

[86]:
```
# Get scores on test data

rf_preds = rf1.best_estimator_.predict(x_test)
```

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

[87]:
```
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

[88]:
```
# Get scores on test data
rf_test_scores = get_test_scores('RF test', rf_preds, y_test)
results = pd.concat([results, rf_test_scores], axis=0)
results
```

[88]:

	model	precision	recall	F1	accuracy
0	RF CV	0.691642	0.818917	0.749877	0.712413
0	RF test	0.679876	0.822029	0.744225	0.702588

All scores increased by at most ~0.02.

XGBoost Try to improve the 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'`)

[89]:
```
# 1. Instantiate the XGBoost classifier

xgb = XGBClassifier(objective='binary:logistic', random_state=0)

# 2. Create a dictionary of hyperparameters to tune

cv_params = {'learning_rate': [0.1, 0.3],
             'max_depth': [8, 10],
             'min_child_weight': [2, 5, 10],
             'n_estimators': [500]
             }

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

# 4. Instantiate the GridSearchCV object
xgb1 = GridSearchCV(xgb, cv_params, scoring=scoring, cv=4, refit='f1')
```

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

[ ]:
```
%%time
xgb1.fit(x_train, y_train)
```

Get the best score from this model.

[44]:
```
# Examine best score
xgb1.best_score_
```

And the best parameters.

[45]:
```
# Examine best parameters
xgb1.best_params_
```

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

[46]:
```
# Call 'make_results()' on the GridSearch object
xgb1_cv_results = make_results('XGB CV', xgb1, 'f1')
results = pd.concat([results, xgb1_cv_results], axis=0)
results
```

[47]:
```
# Get scores on test data
xgb_preds = xgb1.best_estimator_.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.

[48]:
```
# Get scores on test data
xgb_test_scores = get_test_scores('XGB test', xgb_preds, y_test)
results = pd.concat([results, xgb_test_scores], axis=0)
results
```

The F1 score is ~0.01 lower than the random forest model. Both models are acceptable, but the random forest model is the champion.

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

[49]:
```
# 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='');
```

The model is almost twice as likely to predict a false positive than it is to predict a false negative. Therefore, type I errors are more common. This is less desirable, because it's better for a driver to be pleasantly surprised by a generous tip when they weren't expecting one than to be disappointed by a low tip when they were expecting a generous one. However, the overall performance of this model is satisfactory.

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

[50]:
```
importances = rf1.best_estimator_.feature_importances_
rf_importances = pd.Series(importances, index=X_test.columns)
rf_importances = rf_importances.sort_values(ascending=False)[:15]

fig, ax = plt.subplots(figsize=(8,5))
rf_importances.plot.bar(ax=ax)
ax.set_title('Feature importances')
ax.set_ylabel('Mean decrease in impurity')
fig.tight_layout();
```

2.0.4 Task 4. Conclusion

F1 score of this model was 0.7235 and it had an overall accuracy of 0.6865. It correctly identified ~78% of the actual responders in the test set, which is 48% better than a random guess. It may be worthwhile to test the model with a select group of taxi drivers to get feedback.

Unfortunately, random forest is not the most transparent machine learning algorithm. We know that VendorID, predicted_fare, mean_duration, and mean_distance are the most important features, but we don't know how they influence tipping. This would require further exploration.