Activity_Course 6 Automatidata project lab

May 13, 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, and the ability to filter out people who don't tip would help increase driver revenue.

Recall that you have a helpful tool at your disposal! Refer to the PACE Strategy Document to apply your learnings, apply new problem-solving skills, and guide your approach to this project.

2 PACE stages

- `[Plan](#scrollTo=psz51YkZVwtN&line=3&uniqifier=1)`
- `[Analyze](#scrollTo=mA7Mz_SnI8km&line=4&uniqifier=1)`
- `[Construct](#scrollTo=Lca9c8XON8lc&line=2&uniqifier=1)`
- `[Execute](#scrollTo=401PgchTPr4E&line=2&uniqifier=1)`

2.1 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?** Predict whether the customer will leave a tip higher than 20% of the trip's value.
2. **What are the ethical implications of the model? What are the consequences of your model making errors?**

`False Positive (when the model predicts that the customer will leave a tip):`

Drivers would distrust with the app because they do not receive tip when the app says so.

`False Negative:(when the model predicts that the customer won't leave a tip):`

The customer's credibility would become negatively affected and they would have trouble finding new trips, which limits the accesibility of taxi service to people who pay tips.

3. **Do the benefits of such a model outweigh the potential problems?** Due to the ethical conflict generated by the division of customers it would be problematic for the company, and people would distrust the app.
4. **Would you proceed with the request to build this model? Why or why not?** It depends on the approach that would be taken with this information.
5. **Can the objective be modified to make it less problematic?** The model can be focused on presenting the results as a characteristic of the customer, such as a feature that displays that the person is a generous customer.

Complete the following steps to begin:

2.1.1 Task 1. Imports and data loading

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

[278]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

pd.set_option('display.max_columns', None)

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

from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier

from xgboost import plot_importance

import pickle
```

`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.

[279]:
```
df0 = pd.read_csv('2017_Yellow_Taxi_Trip_Data.csv')
```

2.2 PACE: Analyze

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

2.2.1 Task 2. Feature engineering

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

[280]:
```
df0.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 22699 entries, 0 to 22698
Data columns (total 18 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
dtypes: float64(8), int64(7), object(3)
memory usage: 3.1+ MB
```

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 `df`. Then, use a Boolean mask to filter `df1` so it contains only customers who paid with credit card.

[281]:
```
df1 = df0.copy()
df1 = df1[df1['payment_type']==1]
```

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:

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

[282]:
```
df1['tip_percent'] = df1['tip_amount'] / (df1['total_amount'] - df1['tip_amount'])
```

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).

[283]:
```
df1['generous'] = df1['tip_percent']
df1['generous'] = (df1['generous'] >= 0.2)
df1['generous'] = df1['generous'].astype(int)
```

Features Which columns are obviously unpredictive of tip percentage? Refer to the data dictionary.

Drop `Unnamed: 0` and `store_and_fwd_flag` columns. Assign the result back to `df1`.

[284]:
```
drop_cols = ['Unnamed: 0', 'store_and_fwd_flag']
df1 = df1.drop(drop_cols, axis=1)
```

Next, you're going to be working with the pickup and dropoff columns. To do this, you'll need to import the `datetime` module. Import this module as `dt`.

Then, convert the `tpep_pickup_datetime` and `tpep_dropoff_datetime` columns to the datetime class.

[285]:
```
import datetime as dt
df1[['tpep_pickup_datetime', 'tpep_dropoff_datetime']] = df1[['tpep_pickup_datetime', 'tpep_dropoff_datetime']].apply(pd.to_datetime)
```

Create a new column called `duration`, which captures the time elapsed from pickup to dropoff.

1. Subtract `tpep_pickup_datetime` from `tpep_dropoff_datetime` and assign the result to a new column called `duration`.
2. Convert the `duration` column to seconds.

[286]:
```
df1['duration'] = df1['tpep_dropoff_datetime'] - df1['tpep_pickup_datetime']
df1['duration'] = df1['duration'].dt.total_seconds()
```

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

[287]:
```
df1['day'] = df1['day'] = df1['tpep_pickup_datetime'].dt.day_name().str.lower()
```

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 contain the same information: the hour (only) from the `tpep_pickup_datetime` column.

[288]:
```
df1['am_rush'] = df1['tpep_pickup_datetime'].dt.hour

df1['daytime'] = df1['tpep_pickup_datetime'].dt.hour

df1['pm_rush'] = df1['tpep_pickup_datetime'].dt.hour

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.

[289]:
```
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.

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

[290]:
```
df1['am_rush'] = df1.apply(am_rush, axis=1)
df1['am_rush'].head()
```

[290]:
```
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.

[291]:
```
# Define 'daytime()' conversion function [10:00-16:00)
def daytime(hour):
```

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

[292]:
```
df1['daytime'] = df1.apply(daytime, axis=1)
df1['daytime'].head()
```

[292]:
```
0    0
1    1
2    0
3    1
5    0
Name: daytime, dtype: int64
```

[293]:
```
def pm_rush(hour):
    if 16 <= hour['pm_rush'] < 20:
        val = 1
    else:
        val = 0
    return val
```

[294]:
```
df1['pm_rush'] = df1.apply(pm_rush, axis=1)
df1['pm_rush'].head()
```

[294]:
```
0    0
1    0
2    0
3    0
5    0
Name: pm_rush, dtype: int64
```

[295]:
```
# 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
```

[296]:
```
df1['nighttime'] = df1.apply(nighttime, axis=1)
df1['nighttime'].head()
```

[296]:
```
0    0
1    0
2    0
3    0
5    1
Name: nighttime, dtype: int64
```

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.

[297]:
```python
df1['month'] = df1['tpep_pickup_datetime'].dt.strftime('%b').str.lower()
```

Because you have encoded much of the information contained in the pickup and dropoff columns into new columns, you can drop them for modeling.

1. Drop the `tpep_pickup_datetime` and `tpep_dropoff_datetime` columns and reassign the result back to `df1`.

[298]:
```python
df1 = df1.drop(['tpep_pickup_datetime','tpep_dropoff_datetime'], axis=1)
```

Examine the first five rows of your dataframe.

[299]:
```python
df1.head()
```

[299]:
```
   VendorID  passenger_count  trip_distance  RatecodeID  PULocationID  \
0         2                6           3.34           1           100
1         1                1           1.80           1           186
2         1                1           1.00           1           262
3         2                1           3.70           1           188
5         2                6           2.30           1           161

   DOLocationID  payment_type  fare_amount  extra  mta_tax  tip_amount  \
0           231             1         13.0    0.0      0.5        2.76
1            43             1         16.0    0.0      0.5        4.00
2           236             1          6.5    0.0      0.5        1.45
3            97             1         20.5    0.0      0.5        6.39
5           236             1          9.0    0.5      0.5        2.06

   tolls_amount  improvement_surcharge  total_amount  tip_percent  generous  \
0           0.0                    0.3         16.56     0.200000         1
1           0.0                    0.3         20.80     0.238095         1
2           0.0                    0.3          8.75     0.198630         0
3           0.0                    0.3         27.69     0.300000         1
5           0.0                    0.3         12.36     0.200000         1

   duration       day  am_rush  daytime  pm_rush  nighttime month
0     844.0  saturday        1        0        0          0   mar
1    1590.0   tuesday        0        1        0          0   apr
2     432.0    friday        1        0        0          0   dec
```

```
3    1815.0    sunday          0          1          0            0    may
5     480.0  saturday          0          0          0            1    mar
```

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_string` to string.

[300]:
```
cols_to_str = ['RatecodeID','PULocationID','DOLocationID']

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

HINT

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

The `VendorID` column is also a numerical column that contains categorical information (which taxi cab company picked up the passenger). The values are all 1 or 2.

1. Convert this to binary by subtracting 1 from every value in the column.

[301]:
```
df1['VendorID'] = df1['VendorID'] - 1
```

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`. Don't use the `drop_first` parameter.

[302]:
```
df2 = pd.get_dummies(df1)
```

Finally, drop the columns that are constant or that contain information that would be a proxy for our target variable. For example, `total_amount` contains tip amount, and therefore tip percentage, if used with `fare_amount`. And `mta_tax` is $0.50 99.6% of the time, so it's not adding any predictive signal to the model.

1. Drop the following features: `payment_type`, `mta_tax`, `tip_amount`, `total_amount`, and `tip_percent`. Assign the results to a new dataframe called `df3`.

[303]:
```
df3 = df2.drop(['payment_type', 'mta_tax', 'tip_amount', 'total_amount',␣
↪'tip_percent'], axis=1)
```

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

1. Examine the class balance of your target variable.

[304]:
```
df3['generous'].value_counts()
```

[304]:
```
0    9944
1    5321
Name: generous, dtype: int64
```

Approximately 1/3 of the customers in this dataset were "generous" (tipped 20%). The dataset is imbalanced, but not extremely so.

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 receiving one.

False negatives are worse for customers, because a cab driver would likely pick up a different customer who was predicted to tip more.

2.3 PACE: Construct

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

2.3.1 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.

[305]:
```
y = df3['generous']

X = df3.drop('generous', axis=1)

X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,␣
↪test_size=0.2, random_state=42)
```

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 dictionary `scoring` of scoring metrics for GridSearch to capture (precision, recall, F1 score, and accuracy).

4. Instantiate the `GridSearchCV` object `rf_cv1`. 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 `'precision'`.

[306]:
```
rf = RandomForestClassifier(random_state=0)

cv_params = {'max_depth': [3,5, None],
             'max_features': [1.0],
             'max_samples': [0.7, 1.0],
             'min_samples_leaf': [1,2,3],
             'min_samples_split': [2,3,4],
             'n_estimators': [300, 500],
             }

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

rf_cv1 = GridSearchCV(rf, cv_params, scoring=scoring, cv=4, refit='precision')
```

Now fit the model to the training data.

Note: *The following operation may take over an hour to complete.* Therefore, the cell has been commented out along with code cell #33 (where we pickle the model). To save time, you can skip these cells and continue to execute the cells in order.

[307]:
```
%%time

rf_cv1.fit(X_train, y_train)
```

```
CPU times: user 1h 12min 29s, sys: 1.45 s, total: 1h 12min 30s
Wall time: 1h 12min 30s
```

[307]:
```
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=0,
                                              verbose=0, warm_start=False),
             iid='deprecated', n_jobs=None,
             param_grid={'max_depth': [3, 5, None], 'max_features': [1.0],
                         'max_samples': [0.7, 1.0],
                         'min_samples_leaf': [1, 2, 3],
                         'min_samples_split': [2, 3, 4],
                         'n_estimators': [300, 500]},
             pre_dispatch='2*n_jobs', refit='precision',
             return_train_score=False,
             scoring={'f1', 'accuracy', 'precision', 'recall'}, verbose=0)
```

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.

[308]:
```python
import pickle

path = '/home/jovyan/work/'
```

[309]:
```python
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)
```

[310]:
```python
write_pickle(path, rf_cv1, 'taxi_rf_cv1')
```

[311]:
```python
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
```

[312]:
```
rf_cv1 = read_pickle(path, 'taxi_rf_cv1')
```

Examine the best average score across all the validation folds.

[313]:
```
rf_cv1.best_score_
```

[313]:
```
0.6759301437424484
```

Examine the best combination of hyperparameters.

[314]:
```
rf_cv1.best_params_
```

[314]:
```
{'max_depth': 5,
 'max_features': 1.0,
 'max_samples': 0.7,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'n_estimators': 300}
```

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

[315]:
```
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()
    table = table.append({'Model': model_name,
                          'Precision': precision,
                          'Recall': recall,
                          'F1': f1,
                          'Accuracy': accuracy,
                          },
                          ignore_index=True
                         )

    return table
```

Call `make_results()` on the GridSearch object.

[316]:
```
results = make_results('random forest 1: precision', rf_cv1, 'precision')
results
```

The precision seems satisfactory, but not great. The other scores are very bad.

A model with such low F1 and recall scores is not good enough. Try retuning the model to select based on F1 score instead. Consider adjusting the hyperparameters that you try based on the results of the above model.

[317]:
```
rf = RandomForestClassifier(random_state=0)

cv_params = {'max_depth': [5, 6, 7],
             'max_features': [1.0],
             'max_samples': [0.7],
             'min_samples_leaf': [1,2],
             'min_samples_split': [2,3],
             'n_estimators': [300],
             }

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

rf_cv2 = GridSearchCV(rf, cv_params, scoring=scoring, cv=4, refit='f1')
```

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

[318]:
```
%%time
rf_cv2.fit(X_train, y_train)
```

Get the best score from this model.

[319]:
```
rf_cv2.best_score_
```

And the best parameters.

[320]:
```
rf_cv2.best_params_
```

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

[321]:
```
results = make_results('random forest: f1', rf_cv2, 'f1')
results
```

There was a modest improvement in both F1 and recall scores, but these results still are not good enough to deploy the model.

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

[322]:
```
preds = rf_cv2.best_estimator_.predict(X_test)
```

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

[323]:
```
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 = round(accuracy_score(y_test_data, preds), 3)
    precision = round(precision_score(y_test_data, preds), 3)
    recall = round(recall_score(y_test_data, preds), 3)
    f1 = round(f1_score(y_test_data, preds), 3)

    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_cv2_test_scores`.
2. Call `rf_cv2_test_scores` to output the results.

[324]:
```
rf_cv2_test_scores = get_test_scores('random forest: f1', preds, y_test)
rf_cv2_test_scores
```

How do your test results compare to your validation results?

All scores increased by 0.02.

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 dictionary `scoring` of scoring metrics for grid search to capture (precision, recall, F1 score, and accuracy).

4. Instantiate the `GridSearchCV` object `xgb_cv1`. Pass to it as arguments:

- estimator=`xgb`
- 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='f1'`)

[334]:
```
xgb = XGBClassifier(objective='binary:logistic', random_state=0)

cv_params = {'max_depth': [4,8,12],
             'min_child_weight': [3, 5],
             'learning_rate': [0.01, 0.1],
             'n_estimators': [300, 500]
             }

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

```
xgb_cv1 = GridSearchCV(xgb, cv_params, scoring=scoring, cv=4, refit='f1')
```

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

Note: *The following operation may take over an hour to complete.* Therefore, the cell has been commented out along with code cell #50 (where we pickle the model). To save time, you can skip these cells and continue to execute the cells in order.

[335]:
```
%%time

xgb_cv1.fit(X_train, y_train)
```

```
    ␣
↪---------------------------------------------------------------------------

    KeyboardInterrupt                         Traceback (most recent call␣
↪last)

    <timed eval> in <module>


    /opt/conda/lib/python3.7/site-packages/sklearn/model_selection/_search.
↪py in fit(self, X, y, groups, **fit_params)
    708                     return results
    709
--> 710                 self._run_search(evaluate_candidates)
    711
    712             # For multi-metric evaluation, store the best_index_,␣
↪best_params_ and


    /opt/conda/lib/python3.7/site-packages/sklearn/model_selection/_search.
↪py in _run_search(self, evaluate_candidates)
   1149     def _run_search(self, evaluate_candidates):
   1150         """Search all candidates in param_grid"""
-> 1151         evaluate_candidates(ParameterGrid(self.param_grid))
   1152
   1153


    /opt/conda/lib/python3.7/site-packages/sklearn/model_selection/_search.
↪py in evaluate_candidates(candidate_params)
    687                                for parameters, (train, test)
    688                                in product(candidate_params,
--> 689                                           cv.split(X, y, groups)))
    690
```

```
    691                 if len(out) < 1:


    /opt/conda/lib/python3.7/site-packages/joblib/parallel.py in␣
↪__call__(self, iterable)
   1005                 self._iterating = self._original_iterator is not None
   1006
-> 1007             while self.dispatch_one_batch(iterator):
   1008                 pass
   1009


    /opt/conda/lib/python3.7/site-packages/joblib/parallel.py in␣
↪dispatch_one_batch(self, iterator)
    833                 return False
    834             else:
--> 835                 self._dispatch(tasks)
    836                 return True
    837


    /opt/conda/lib/python3.7/site-packages/joblib/parallel.py in␣
↪_dispatch(self, batch)
    752         with self._lock:
    753             job_idx = len(self._jobs)
--> 754             job = self._backend.apply_async(batch, callback=cb)
    755             # A job can complete so quickly than its callback is
    756             # called before we get here, causing self._jobs to


    /opt/conda/lib/python3.7/site-packages/joblib/_parallel_backends.py in␣
↪apply_async(self, func, callback)
    207     def apply_async(self, func, callback=None):
    208         """Schedule a func to be run"""
--> 209         result = ImmediateResult(func)
    210         if callback:
    211             callback(result)


    /opt/conda/lib/python3.7/site-packages/joblib/_parallel_backends.py in␣
↪__init__(self, batch)
    588         # Don't delay the application, to avoid keeping the input
    589         # arguments in memory
--> 590         self.results = batch()
    591
    592     def get(self):
```

```
    /opt/conda/lib/python3.7/site-packages/joblib/parallel.py in␣
↪__call__(self)
    254         with parallel_backend(self._backend, n_jobs=self._n_jobs):
    255             return [func(*args, **kwargs)
--> 256                     for func, args, kwargs in self.items]
    257
    258     def __len__(self):


    /opt/conda/lib/python3.7/site-packages/joblib/parallel.py in <listcomp>(.
↪0)
    254         with parallel_backend(self._backend, n_jobs=self._n_jobs):
    255             return [func(*args, **kwargs)
--> 256                     for func, args, kwargs in self.items]
    257
    258     def __len__(self):


    /opt/conda/lib/python3.7/site-packages/sklearn/model_selection/
↪_validation.py in _fit_and_score(estimator, X, y, scorer, train, test,␣
↪verbose, parameters, fit_params, return_train_score, return_parameters,␣
↪return_n_test_samples, return_times, return_estimator, error_score)
    513             estimator.fit(X_train, **fit_params)
    514         else:
--> 515             estimator.fit(X_train, y_train, **fit_params)
    516
    517     except Exception as e:


    /opt/conda/lib/python3.7/site-packages/xgboost/core.py in inner_f(*args,␣
↪**kwargs)
    573         for k, arg in zip(sig.parameters, args):
    574             kwargs[k] = arg
--> 575         return f(**kwargs)
    576
    577     return inner_f


    /opt/conda/lib/python3.7/site-packages/xgboost/sklearn.py in fit(self,␣
↪X, y, sample_weight, base_margin, eval_set, eval_metric,␣
↪early_stopping_rounds, verbose, xgb_model, sample_weight_eval_set,␣
↪base_margin_eval_set, feature_weights, callbacks)
   1409             verbose_eval=verbose,
   1410             xgb_model=model,
-> 1411             callbacks=callbacks,
   1412         )
```

```
   1413

    /opt/conda/lib/python3.7/site-packages/xgboost/core.py in inner_f(*args,␣
↪**kwargs)
    573         for k, arg in zip(sig.parameters, args):
    574             kwargs[k] = arg
--> 575         return f(**kwargs)
    576
    577     return inner_f

    /opt/conda/lib/python3.7/site-packages/xgboost/training.py in␣
↪train(params, dtrain, num_boost_round, evals, obj, feval, maximize,␣
↪early_stopping_rounds, evals_result, verbose_eval, xgb_model, callbacks,␣
↪custom_metric)
    179         if cb_container.before_iteration(bst, i, dtrain, evals):
    180             break
--> 181         bst.update(dtrain, i, obj)
    182         if cb_container.after_iteration(bst, i, dtrain, evals):
    183             break

    /opt/conda/lib/python3.7/site-packages/xgboost/core.py in update(self,␣
↪dtrain, iteration, fobj)
   1778             _check_call(_LIB.XGBoosterUpdateOneIter(self.handle,
   1779                                                     ctypes.
↪c_int(iteration),
-> 1780                                                     dtrain.handle))
   1781         else:
   1782             pred = self.predict(dtrain, output_margin=True,␣
↪training=True)

    KeyboardInterrupt:
```

[327]:
```
write_pickle(path, xgb_cv1, 'taxi_xgb_cv1')
```

[328]:
```
xgb_cv1 = read_pickle(path, 'taxi_xgb_cv1')
```

Get the best score from this model.

[ ]:
```
xgb_cv1.best_score_
```

And the best parameters.

[ ]:
```
xgb_cv1.best_params_
```

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

```
[329]: results = make_results('XGBoost 1: f1', xgb_cv1, 'f1')
       results
```

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

```
[330]: preds = xgb_cv1.best_estimator_.predict(X_test)
```

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

```
[331]: xgb_cv_test_scores = get_test_scores('XGBoost 1: f1', preds, y_test)
       xgb_cv_test_scores
```

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

The precision is 0.02 lower than the random forest model, but recall is over 40% better and F1 is 24% better. Even accuracy improved. XGBoost is the better model.

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

```
[336]: cm = confusion_matrix(y_test, preds, labels=xgb_cv1.classes_)

       disp = ConfusionMatrixDisplay(confusion_matrix=cm,
                                     display_labels=xgb_cv1.classes_)
       disp.plot();
```

```
                                                                                ␣
↪---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call␣
↪last)

        <ipython-input-336-99b2a46fad22> in <module>
    ----> 1 cm = confusion_matrix(y_test, preds, labels=xgb_cv1.classes_)
          2
          3 disp = ConfusionMatrixDisplay(confusion_matrix=cm,
          4                               display_labels=xgb_cv1.classes_)
          5 disp.plot();


        NameError: name 'preds' is not defined
```

What type of errors are more common for your model?

Type II errors, because it is as twice possible to have a false negative than a false positive.

Feature importance Use the `plot_importance` function to inspect the top 10 most important features of your final model.

[333]:
```
plot_importance(xgb_cv1.best_estimator_, max_num_features=10);
```

2.4 PACE: Execute

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

2.4.1 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?**
   I would not recommend this model because it is not as precise as it can be expected. However, if the results are taken into account I would strongly ecourage to display the precision of the model, so the drivers would understand its limits. Additionally, further analysis in the response of the drivers is recommended to evaluate the performance of the model among users.
2. **What was your model doing? Can you explain how it was making predictions?**
   Because XGBoost was used for the model, there is no clear explanation about how variables affect the model. However, we understand that the duration and distance of the trip have a high influence.
3. **What features would you want to have that would likely improve the performance of your model?**
   It would be usefull to analyze the history of the client and the exactly amount that was used for a tip, there can be some patterns such as a maximum tip per trip regardless of the distance and duration of the trip, which means that there could be a roof in the curve. 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.