Modeling

September 20, 2020

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import gc
import sklearn
import os
import warnings
warnings.filterwarnings('ignore')
import math
import shutil
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, mean_absolute_error, max_error
from sklearn.preprocessing import OneHotEncoder
from sklearn.ensemble import RandomForestRegressor
import shap
import os
```

0.1 Data transformations, feature creation

[ ]:
```
#Download the data if needed
#os.chdir('./data')
#!wget 'https://s3.amazonaws.com/nyc-tlc/trip+data/yellow_tripdata_2017-03.csv'
#!wget 'https://s3.amazonaws.com/nyc-tlc/trip+data/yellow_tripdata_2017-06.csv'
#!wget 'https://s3.amazonaws.com/nyc-tlc/trip+data/yellow_tripdata_2017-11.csv'
#os.chdir('..')
```

[2]:
```
#Select credit card payments only, calculate the total cost of the trip without␣
↪the tip.
def totals(df):
    df = df[df['payment_type']==1]
    df.drop(['payment_type'],axis=1, inplace=True)
    df['total_cost'] = df[['fare_amount', 'extra', 'mta_tax', 'tolls_amount',␣
↪'improvement_surcharge']].sum(axis=1)
```

```
    df.drop('total_amount', axis=1, inplace=True)
    return df
```

[3]:
```
#Convert tpep_* columns to datetime format. Create features based on these␣
↪datetimes.
def process_tpep(df):
    df['tpep_dropoff_datetime'] = pd.to_datetime(df['tpep_dropoff_datetime'])
    df['tpep_pickup_datetime'] = pd.to_datetime(df['tpep_pickup_datetime'])
    df['month'] = df['tpep_pickup_datetime'].dt.month
    df['day'] = df['tpep_pickup_datetime'].dt.day
    df['dayofweek'] = df['tpep_pickup_datetime'].dt.dayofweek
    df['hour_PU'] = df['tpep_pickup_datetime'].dt.hour
    df['hour_DO'] = df['tpep_dropoff_datetime'].dt.hour
    df.drop(['tpep_pickup_datetime', 'tpep_dropoff_datetime'], axis=1,␣
↪inplace=True)
    return df
```

[4]:
```
#Add the taxi zone data to the DataFrame. Drop Zones because they are already␣
↪included in the PULocationID
#and DOLocationID. Fill missing data with 'Unknown'.
def process_zones(df):
    taxi_zones = pd.read_csv('./data/taxi+_zone_lookup.csv')
    df = df.merge(taxi_zones, how='left', left_on='PULocationID',␣
↪right_on='LocationID')
    df = df.merge(taxi_zones, how='left', left_on='DOLocationID',␣
↪right_on='LocationID', suffixes=(None, '_DO'))
    del taxi_zones
    gc.collect()
    df.drop(['LocationID', 'LocationID_DO', 'Zone', 'Zone_DO'], axis=1,␣
↪inplace=True)
    df = df.rename(columns={'Borough': 'Borough_PU', 'service_zone':␣
↪'service_zone_PU'})
    gc.collect()
    df[['Borough_PU', 'Borough_DO', 'service_zone_PU', 'service_zone_DO']] =␣
↪df[['Borough_PU', 'Borough_DO', 'service_zone_PU', 'service_zone_DO']].
↪fillna('Unknown')
    return df
```

[5]:
```
#Process each months' data, split it into train and test sets and save them in␣
↪the tmp folder.
if os.path.isdir("./tmp"):
    shutil.rmtree('/tmp')
os.mkdir('tmp')

#For march, june, november
for i,month in enumerate(['03', '06', '11']):
```

```
    df = pd.read_csv('./data/yellow_tripdata_2017-{0}.csv'.format(month))
    df = totals(df)
    df = process_tpep(df)
    df = process_zones(df)
    train, test = train_test_split(df, test_size = 0.2, random_state=42)
    if i == 0:
        train.to_csv('./tmp/train.csv', index=False, header=True)
        test.to_csv('./tmp/test.csv', index=False, header=True)
    else:
        train.to_csv('./tmp/train.csv', index=False, header=False, mode='a')
        test.to_csv('./tmp/test.csv', index=False, header=False, mode='a')
    del train, test
    gc.collect()
```

0.2 Baseline model

Let's create a quick baseline model to have something to compare to.

When deciding how much to tip, most people looks at the total to pay and quickly calculates a certain percentage of that amount for tipping. So my baseline model will be a simple linear regression using the total cost of trip.

[6]:
```
df = pd.read_csv('./tmp/train.csv')
```

[7]:
```
#Let's split the data to train and validation sets.
X = df[['fare_amount', 'extra', 'mta_tax', 'tolls_amount',␣
 ↪'improvement_surcharge']]
y = df['tip_amount']
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)
del df, X, y
gc.collect()
```

[7]:
```
0
```

[8]:
```
print(X_train.shape, X_val.shape)
```

```
(12700498, 5) (3175125, 5)
```

[9]:
```
#Fit the model
lr = LinearRegression().fit(X_train, y_train)
#Let's look at the score
print(f'Training score: {lr.score(X_train, y_train):.4f}')
#Predict the labels for the validation set
y_pred = lr.predict(X_val)
```

```
Training score: 0.5797
```

```
[10]: #Predict the labels for the validation set
y_pred = lr.predict(X_val)
#Let's look at the rmse and mae metrics of the results
print(f'Validation score: {lr.score(X_val, y_val):.4f}')
print(f'RMSE: {math.sqrt(mean_squared_error(y_val, y_pred)):.4f}')
print(f'MAE: {mean_absolute_error(y_val, y_pred):.4f}')
```

```
Validation score: 0.5938
RMSE: 1.7587
MAE: 0.7737
```

```
[11]: feature_importance = abs(lr.coef_[0])
feature_importance = 100.0 * (feature_importance / feature_importance.max())
feature_importances = pd.DataFrame(feature_importance, index = X_train.columns,
                                   columns=['importance']).
 ↪sort_values('importance', ascending=False)
feature_importances.head(10)
```

[11]:

	importance
fare_amount	100.0
extra	100.0
mta_tax	100.0
tolls_amount	100.0
improvement_surcharge	100.0

0.3 Random Forest

With so many categorical variables tree-based algorithm are usually prove to be a good choice. Here I will use first the random forest algorithm to see how much improvement I get compared to the baseline.

I will subsample the data to make it run faster.

```
[12]: df = pd.read_csv('./tmp/train.csv')
df = df.sample(1000000)
```

```
[13]: # Since the fare_amount and the total_cost columns are highly correlated, I␣
 ↪decided to drop the total_cost
# and keep the fare_amount.
X = df.drop(['tip_amount', 'total_cost'], axis=1)
y = df['tip_amount']
del df
gc.collect()
```

```
[13]: 0
```

[14]:
```
categorical = ['VendorID', 'RatecodeID', 'PULocationID', 'DOLocationID',␣
↪'month', 'day', 'dayofweek', 'hour_PU', 'hour_DO', 'store_and_fwd_flag',␣
↪'Borough_PU', 'service_zone_PU', 'Borough_DO', 'service_zone_DO']
numerical = ['passenger_count', 'trip_distance', 'fare_amount', 'extra',␣
↪'mta_tax', 'tolls_amount', 'improvement_surcharge', 'total']
```

[15]:
```
#Split to train and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)
del X, y
gc.collect()
X_train.reset_index(drop=True, inplace=True)
y_train.reset_index(drop=True, inplace=True)
X_val.reset_index(drop=True, inplace=True)
y_val.reset_index(drop=True, inplace=True)
```

[16]:
```
print(X_train.shape, X_val.shape)
```

```
(800000, 21) (200000, 21)
```

[17]:
```
# When categorical columns with numerical encoding are not ordinal, it works␣
↪better to use one-hot encoding.
# The train set is used to fit the encoder, which is then saved to transform␣
↪the validation and test data.

encoders = {}
for col in categorical:
    print(col)
    ohe = OneHotEncoder(sparse=False, handle_unknown='ignore' )
    a = ohe.fit_transform(np.asarray(X_train[col]).reshape(-1, 1))
    a = pd.DataFrame(a)
    a.columns = ['_'.join([col, str(c)]) for c in ohe.categories_[0]]
    X_train = pd.concat((X_train, a), axis=1)
    del a
    encoders[col] = ohe
    X_train.drop(col, axis=1, inplace=True)
    gc.collect()

#Let's transform the validation set, too
for col in categorical:
    print(col)
    ohe = encoders[col]
    a = ohe.transform(np.asarray(X_val[col]).reshape(-1, 1))
    a = pd.DataFrame(a)
    a.columns = [col + '_' + str(c) for c in ohe.categories_[0]]
    X_val = pd.concat((X_val, a), axis=1)
    del a
```

```
    X_val.drop(col, axis=1, inplace=True)
    gc.collect()

assert X_train.shape[1] == X_val.shape[1]
```

```
VendorID
RatecodeID
PULocationID
DOLocationID
month
day
dayofweek
hour_PU
hour_DO
store_and_fwd_flag
Borough_PU
service_zone_PU
Borough_DO
service_zone_DO
VendorID
RatecodeID
PULocationID
DOLocationID
month
day
dayofweek
hour_PU
hour_DO
store_and_fwd_flag
Borough_PU
service_zone_PU
Borough_DO
service_zone_DO
```

[18]:
```
%%time

# Fit the random forest regressor
rf = RandomForestRegressor(max_depth=10, random_state=0, n_jobs=-1)
rf.fit(X_train, y_train)
print(f'Training score: {rf.score(X_train, y_train):.4f}')
```

```
Training score: 0.7104
CPU times: user 1h 49min 49s, sys: 2.53 s, total: 1h 49min 52s
Wall time: 7min 13s
```

[19]:
```
print(rf.score(X_train, y_train))
print(rf.score(X_val, y_val))
```

```
0.7103899836689429
0.5356208056404663
```

[20]:
```
y_pred = rf.predict(X_val)
print(f'Validation score: {rf.score(X_val, y_val):.4f}')
print(f'RMSE: {math.sqrt(mean_squared_error(y_val, y_pred)):.4f}')
print(f'MAE: {mean_absolute_error(y_val, y_pred):.4f}')
```

```
Validation score: 0.5356
RMSE: 2.0256
MAE: 0.7569
```

[21]:
```
#Examine the most important features
feature_importances = pd.DataFrame(rf.feature_importances_, index = X_train.
 ↪columns,
                                    columns=['importance']).
 ↪sort_values('importance', ascending=False)
feature_importances.head(10)
```

[21]:

	importance
fare_amount	0.810982
PULocationID_114	0.024208
trip_distance	0.019568
DOLocationID_265	0.017951
tolls_amount	0.017890
DOLocationID_21	0.007538
PULocationID_265	0.004906
day_17	0.003448
hour_PU_21	0.002851
day_29	0.002764

The random forest algorithm gave practically the same result as the linear regression. As we can see from the feature importances, the most important feature here is also the price of the trip, all the other features have significantly less importance.

It's possible that using all the data instead of the subsample, and applying hyperparameter tuning we could achieve a bit better results. However, it's clear that by far the most important feature is the price of the trip, and we would be wasting time and resources for only a little improvement.

Therefore I would propose to use the more simple algorithm for predicting tips. Linear regression is simple, very fast to run, and easy to build an API with it.

0.4 Linear regression

[22]:
```
df = pd.read_csv('./tmp/train.csv')
X=df[['fare_amount', 'extra', 'mta_tax', 'tolls_amount',␣
 ↪'improvement_surcharge']]
```

```
y = df['tip_amount']
del df
gc.collect()
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)
del X, y
gc.collect()
```

[22]: 0

[23]:
```
lr = LinearRegression().fit(X_train, y_train)
print(lr.coef_, lr.intercept_)
```

```
[0.15462518 0.20830072 0.64231625 0.27978208 6.09069465] -1.6595347908575366
```

[24]:
```
y_pred = lr.predict(X_val).round(2)
print(f'RMSE: {math.sqrt(mean_squared_error(y_val, y_pred))}')
print(f'MAE: {mean_absolute_error(y_val, y_pred)}')
print(f'max error: {max_error(y_val, y_pred)}')
```

```
RMSE: 1.7586851479746746
MAE: 0.7735659698437072
max error: 288.78
```

People often like to calculate their tips in a way that the total amount is an integer number. We can also very simply suggest such a tip, and it only slight increases the error:

[26]:
```
def rounded_tip(X, y_pred):
    return (X.sum(axis=1) + y_pred).apply(np.rint) - X.sum(axis=1)

y_pred_rounded = rounded_tip(X_val, y_pred)
print(f'RMSE: {math.sqrt(mean_squared_error(y_val, y_pred_rounded))}')
print(f'MAE: {mean_absolute_error(y_val, y_pred_rounded)}')
print(f'max error: {max_error(y_val, y_pred_rounded)}')
```

```
RMSE: 1.7836382508240742
MAE: 0.8431203810873585
max error: 288.56
```

[27]:
```
#Here we can compare the predicted tips with the real values (y_val)
out = X_val.copy()
out['y_pred'] = y_pred.flatten()
out['y_pred_rounded'] = y_pred_rounded
out['y_val'] = y_val
out.head(20)
```

[27]:
```
         fare_amount  extra  mta_tax  tolls_amount  improvement_surcharge  \
2987345          7.5    0.0      0.5           0.0                    0.3
```

3702940	11.5	1.0	0.5	0.0	0.3
13285301	9.0	0.0	0.5	0.0	0.3
14005756	2.5	0.0	0.5	0.0	0.3
5479312	9.0	0.0	0.5	0.0	0.3
10079834	7.5	0.0	0.5	0.0	0.3
15647979	14.0	0.0	0.5	0.0	0.3
4244662	14.5	0.0	0.5	0.0	0.3
8370752	5.0	0.0	0.5	0.0	0.3
15073795	4.5	0.5	0.5	0.0	0.3
7125365	4.5	0.0	0.5	0.0	0.3
13076896	14.0	0.5	0.5	0.0	0.3
4256254	19.5	0.0	0.5	0.0	0.3
6473016	24.0	0.5	0.5	0.0	0.3
9849691	8.0	0.0	0.5	0.0	0.3
14656001	7.0	0.0	0.5	0.0	0.3
4905843	5.0	1.0	0.5	0.0	0.3
598104	3.5	0.5	0.5	0.0	0.3
9046476	7.5	0.0	0.5	0.0	0.3
262505	11.0	0.0	0.5	0.0	0.3

	y_pred	y_pred_rounded	y_val
2987345	1.65	1.7	1.66
3702940	2.48	2.7	2.08
13285301	1.88	2.2	1.95
14005756	0.88	0.7	4.00
5479312	1.88	2.2	2.00
10079834	1.65	1.7	1.66
15647979	2.65	2.2	3.70
4244662	2.73	2.7	3.05
8370752	1.26	1.2	1.70
15073795	1.29	1.2	1.74
7125365	1.18	0.7	1.55
13076896	2.76	2.7	4.59
4256254	3.50	3.7	3.25
6473016	4.30	4.7	5.00
9849691	1.73	2.2	2.20
14656001	1.57	1.2	1.95
4905843	1.47	1.2	1.35
598104	1.13	1.2	1.20
9046476	1.65	1.7	1.70
262505	2.19	2.2	2.95

The max error is quite high. The reason for that seem to be that there are entries where the amount of the tip is significantly higher than the fare of the trip. We might consider removing these entries and repeat the fitting, but these are just a few entries, so this time I will not do that.

[32]:
```
out[abs(out['y_val']-out['y_pred'])>200]
```

[32]:
```
          fare_amount  extra  mta_tax  tolls_amount  improvement_surcharge  \
11366086         12.5    0.0      0.5          0.00                    0.3
13042138         10.5    0.0      0.5          0.00                    0.3
6186775         133.0    0.0      0.0         16.50                    0.3
9152465          59.0    0.0      0.5          5.76                    0.3
737201           13.5    0.0      0.5          0.00                    0.3
7425135          52.0    0.0      0.5         12.50                    0.3
15400213         52.0    0.0      0.5          0.00                    0.3
2354259          22.5    0.0      0.5          0.00                    0.3
12208899         52.0    0.0      0.5          0.00                    0.3

          y_pred  y_pred_rounded   y_val
11366086    2.42            2.70  222.22
13042138    2.11            1.70  257.72
6186775    25.35           25.20  250.00
9152465    11.22           11.44  300.00
737201      2.58            2.70  250.00
7425135    12.03           11.70  226.76
15400213    8.53            8.20  222.00
2354259     3.97            3.70  255.00
12208899    8.53            8.20  265.00
```

[33]:
```python
# Let's use our test data we saced in the beginning of the notebook and check␣
↪our model:
df_test = pd.read_csv('./tmp/test.csv')
X_test = df_test[['fare_amount', 'extra', 'mta_tax', 'tolls_amount',␣
↪'improvement_surcharge']]
y_test = df_test['tip_amount']
del df_test
gc.collect()
```

[33]: 0

[34]:
```python
# Predicted tip
y_pred_test = lr.predict(X_test).round(2)
print(f'RMSE: {math.sqrt(mean_squared_error(y_test, y_pred_test))}')
print(f'MAE: {mean_absolute_error(y_test, y_pred_test)}')
print(f'max error: {max_error(y_test, y_pred_test)}')
```

```
RMSE: 1.8915373163866176
MAE: 0.7754614039932386
max error: 447.94
```

[35]:
```python
#Predicted tip with rounding
pred_rounded = rounded_tip(X_test, y_pred_test)
print(f'RMSE: {math.sqrt(mean_squared_error(y_test, pred_rounded))}')
print(f'MAE: {mean_absolute_error(y_test, pred_rounded)}')
```

```
print(f'max error: {max_error(y_test, pred_rounded)}')
```

```
RMSE: 1.914827147285085
MAE: 0.8449968253232376
max error: 447.8
```

Summary In this notebook first I carried out a few data processing steps to create new features and deal with missing values. Then I created a baseline model using linear regression. The next step was creating a random forest model, using a subsample of the data to make it faster. This model however did not improve our results. Therefore I decided that the simpler model is suitable for our purposes.

0.5 API for the model

Since the model is quite light weight, and there's no need for costly preprocessing, a very simple app can be created to serve a tip recommendation for each trip.

For example we could create a simple Flask app. The details of the trip arrive with the request. Next, a function can calculate the total cost of the trip (if not included in the input data), another function can do the prediction and then calculate the predicted tip amount, with can be served as the response of the API.

The Flask app can be deployed for example using Google App Engine.