Technical Assessment Data Scientist Report

May 20, 2022

1 ArkonData - Technical Assesment

- Ángel Andrés Moreno Sánchez

1.1 Abstract

According to Market Research Report, the global bike and scooter rental market is projected to grow from USD 2.1 billion in 2019 to 10.1 billion by 2027 following a rising demand for micro-mobility and emission-free vehicles as well as an increasing demand for an economical mode of transportation. In this scenario, availability and affordable plans must be the top priorities for any compelling company. In this regard, the following analysis shows meaningful insights about a rental bike company located in Los Angeles, it is shown how the availability behaves among its docking stations as well as how their pass-subscription service will evolve on a year basis. A cross-validated machine-learning classification model for passholder prediction with a 68.7% accuracy is presented.

1.2 Exploratory Data Analysis

[1]:
```
# Import required libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import cufflinks as cf
import chart_studio.plotly as py
import plotly.express as px
from plotly.offline import download_plotlyjs, init_notebook_mode, plot, iplot
init_notebook_mode(connected = True)
cf.go_offline()
import seaborn as sns
```

[2]:
```
# Import the dataset
data = pd.read_csv('train_set.csv')
```

```
C:\Users\Andrew S\AppData\Local\Temp\ipykernel_2816\2980385459.py:2:
DtypeWarning:

Columns (8) have mixed types. Specify dtype option on import or set
low_memory=False.
```

[3]:
```
# Id columns dropped as they provide no meaningful info
data = data.drop(columns=['trip_id','bike_id'])
data.head(5)
```

[3]:

	duration	start_time	end_time	start_lat	start_lon	end_lat	end_lon	plan_duration	trip_route_category	passholder_type	start_station	end_station
0	35	2018-08-07 11:20:00	2018-08-07 11:55:00	33.748920	-118.275192	33.748920	-118.275192	1.0	Round Trip	Walk-up	4127	4127
1	32	9/17/2017 17:51	9/17/2017 18:23	34.035679	-118.270813	34.047749	-118.243172	0.0	One Way	Walk-up	3057	3062
2	6	2019-04-22 09:22:00	2019-04-22 09:28:00	34.046070	-118.233093	34.047749	-118.243172	30.0	One Way	Monthly Pass	3022	3062
3	138	9/22/2019 11:27	9/22/2019 13:45	34.062580	-118.290092	34.059689	-118.294662	1.0	One Way	One Day Pass	4304	4311
4	14	1/31/2020 17:11	1/31/2020 17:25	34.026291	-118.277687	34.021660	-118.278687	30.0	One Way	Monthly Pass	4266	4443

Data Cleaning 1.- Check for null values

[4]:
```
data.isnull().sum()/len(data)*100
```

[4]:
```
duration              0.000000
start_time            0.000000
end_time              0.000000
start_lat             0.794714
start_lon             0.794714
end_lat               2.653429
end_lon               2.653429
plan_duration         0.029714
trip_route_category   0.000000
passholder_type       0.368000
start_station         0.000000
end_station           0.000000
dtype: float64
```

We see that there's a considerable >2% of null data on the `end_lat/lon` variables so those observation will be dropped. This info could be better used with the station info database. However, `passholder_type` class is essential so any of those observations will be droped

[5]:
```
data.dropna(subset=['passholder_type'],inplace=True)
```

[6]:
```
data.
↪dropna(subset=['start_lat','start_lon','end_lat','end_lon','plan_duration'],inplace=True)
```

[7]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 675626 entries, 0 to 699999
Data columns (total 12 columns):
 #   Column               Non-Null Count   Dtype
---  ------               --------------   -----
 0   duration             675626 non-null  int64
 1   start_time           675626 non-null  object
 2   end_time             675626 non-null  object
 3   start_lat            675626 non-null  float64
 4   start_lon            675626 non-null  float64
 5   end_lat              675626 non-null  float64
 6   end_lon              675626 non-null  float64
 7   plan_duration        675626 non-null  float64
 8   trip_route_category  675626 non-null  object
 9   passholder_type      675626 non-null  object
 10  start_station        675626 non-null  int64
 11  end_station          675626 non-null  int64
dtypes: float64(5), int64(3), object(4)
memory usage: 67.0+ MB
```

`duration` and `plan duration` variables could be changed to a less consuming memory Dtype `uint16`

[8]:
```
data['duration'] = data['duration'].astype('uint16')
data['plan_duration'] = data['plan_duration'].astype('uint16')
```

[9]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 675626 entries, 0 to 699999
Data columns (total 12 columns):
 #   Column               Non-Null Count   Dtype
---  ------               --------------   -----
 0   duration             675626 non-null  uint16
 1   start_time           675626 non-null  object
 2   end_time             675626 non-null  object
 3   start_lat            675626 non-null  float64
 4   start_lon            675626 non-null  float64
 5   end_lat              675626 non-null  float64
 6   end_lon              675626 non-null  float64
 7   plan_duration        675626 non-null  uint16
 8   trip_route_category  675626 non-null  object
```

```
 9   passholder_type      675626 non-null  object
 10  start_station        675626 non-null  int64
 11  end_station          675626 non-null  int64
dtypes: float64(4), int64(2), object(4), uint16(2)
memory usage: 59.3+ MB
```

1.2.1 Visualizing the target variable

[10]:
```
fig = px.histogram(data, x='passholder_type')
fig.show()
```

We are dealing with a multiclass problem that is also highly imbalanced. SMOTE and undersampling techniques will be used.

1.3 Data Transformation

In order to understand better the passholder types, it's a question of interest to know the time of the day (morning, afternoon, night) in which each kind of customer uses the service.

[11]:
```
# Data transformation from string to datetime object
data_t = data.copy()
data_t['start_time'] = pd.to_datetime(data_t['start_time'])
data_t['end_time'] = pd.to_datetime(data_t['end_time'])
```

We create a new variable according to the different times of the day where according to the `start_time` variable we can classify if the trip was taken during the morning, afternoon, evening or night.

[12]:
```
import datetime
def day_time_cls(x):
    dawn = datetime.time(6,0,0)
    noon = datetime.time(12,0,0)
    dusk = datetime.time(18,0,0)
    midnight = datetime.time(0,0,0)
    if noon >=x.time() >= dawn:
        daytime = 'morning'
    elif dusk >= x.time() >= noon:
        daytime = 'afternoon'
    elif dawn >= x.time() >= midnight:
        daytime = 'night'
    else:
        daytime = 'evening'
    return daytime
```

[13]:
```
data_t['day_time'] = data_t['start_time'].apply(lambda x: day_time_cls(x))
```

1.3.1 Getting insights on the dataset

Since we are dealing with a high amount of observations, a sample will be used for the data visualization.

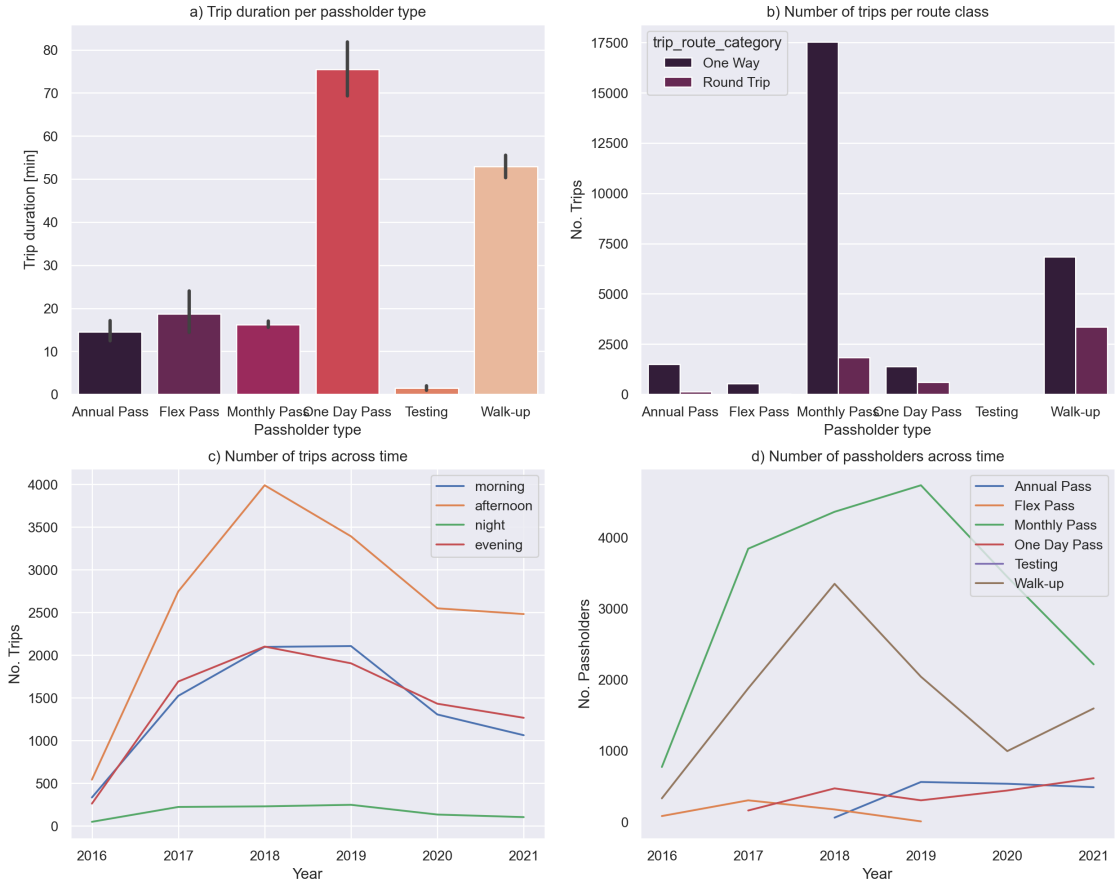
First, let's get info about the `passholder_type` variable.

[14]:
```
# Create the stratified sampling from the dataset
data_vis = data_t.groupby('passholder_type', group_keys=False).apply(lambda x: x.
 ↪sample(frac=0.05))
```

[15]:
```
# Set a seaborn environment for the bar and countplots
sns.set(rc={'figure.figsize':(15,12),'figure.dpi':200})
c = sns.color_palette("rocket")
fig,ax_ = plt.subplots(2,2)
# Trip duration barplot
sns.barplot(data=data_vis, x='passholder_type',␣
 ↪y='duration',ax=ax_[0,0],palette=c)
ax_[0,0].title.set_text('a) Trip duration per passholder type')
ax_[0,0].set_xlabel('Passholder type')
ax_[0,0].set_ylabel('Trip duration [min]')
# Kind of trip proportions
sns.countplot(data=data_vis,␣
 ↪x='passholder_type',hue='trip_route_category',ax=ax_[0,1],palette=c)
ax_[0,1].title.set_text('b) Number of trips per route class')
ax_[0,1].set_xlabel('Passholder type')
ax_[0,1].set_ylabel('No. Trips')
# Create a year column for the visualization df in order to group by the
data_vis['year'] = data_vis['start_time'].apply(lambda x: x.year)
# No. trips per year
for dt in data_vis['day_time'].unique():
    data_vis.groupby(data_vis[data_vis['day_time']==dt]['year']).
 ↪count()['day_time'].plot(ax=ax_[1,0],label=dt,kind='line')
ax_[1,0].title.set_text('c) Number of trips across time')
ax_[1,0].set_xlabel('Year')
ax_[1,0].set_ylabel('No. Trips')
ax_[1,0].legend()
# sns.countplot(data=data_vis, x='year',hue='day_time',ax=ax_[0])

# No. passholders per year
for p in data_vis['passholder_type'].unique():
    data_vis.groupby(data_vis[data_vis['passholder_type']==p]['year']).
 ↪count()['passholder_type'].plot(ax=ax_[1,1],label=p)
ax_[1,1].title.set_text('d) Number of passholders across time')
ax_[1,1].set_xlabel('Year')
ax_[1,1].set_ylabel('No. Passholders')
ax_[1,1].legend()
ax_[1,1].grid()
```

```
plt.show()
```



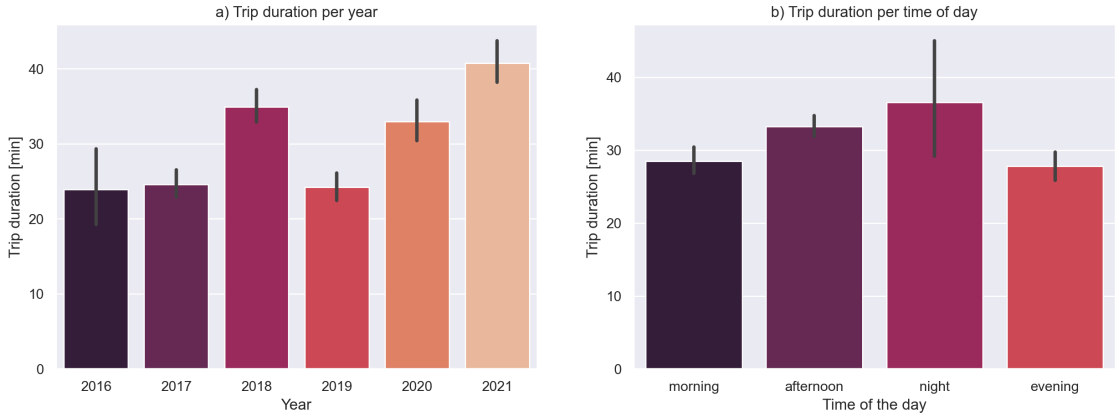
The last figure gives us 3 insights about the differences between the passholders.

1. From figure a) we get to see that the ***One day*** **passholders are the ones with the most mean trip** duration, followed up by the *Walk-up* passholders. On the other hand, *Monthly* and *Annual* passholders are the customers with the shortest mean trip duration.
2. From figure b) we get to see that it is also the ***Monthly*** **passholders customers with the highest number of trips**, followed up by the *Walk-up* passholders, as expected according to the `passholder` distribution. However, we notice that the **One Way trips are the most communs** among all the pass types.
3. From figures c) and d) we notice that there's has been **a reduction on the number of trips and passholders within the last 2 years**. We can also notice that the *afternoon* afternoon and *monthly* passes are the most popular combination of user decision across time.

Additionally we wish to know how do trips behave in the day across the years.

[16]:
```
# Set a seaborn environment for the bar and countplots
sns.set(rc={'figure.figsize':(15,5),'figure.dpi':200})
c = sns.color_palette("rocket")
```

```
fig,ax_ = plt.subplots(1,2)
# Year basis
sns.barplot(data=data_vis, x='year', y='duration',ax=ax_[0],palette=c)
ax_[0].title.set_text('a) Trip duration per year')
ax_[0].set_xlabel('Year')
ax_[0].set_ylabel('Trip duration [min]')
# Time of day basis
sns.barplot(data=data_vis, x='day_time', y='duration',ax=ax_[1],palette=c)
ax_[1].title.set_text('b) Trip duration per time of day')
ax_[1].set_xlabel('Time of the day')
ax_[1].set_ylabel('Trip duration [min]')
plt.show()
```



The last figures give us 2 insights about the trip durations across the years and the time of day.

1. From figure a) We get to see that **trip duration for the last 3 years has remained within the same range around 40** minutes long, since the mean duration per year hasn't changed more than one standard deviation meassure from the last year.
2. From figure b) we notice that all kind of trips are less than 40 minutes long and that **the afternoon trips are the longest kind of trips**.

Now let's get insights about the `stations` variable

1.3.2 Geopandas dataframe

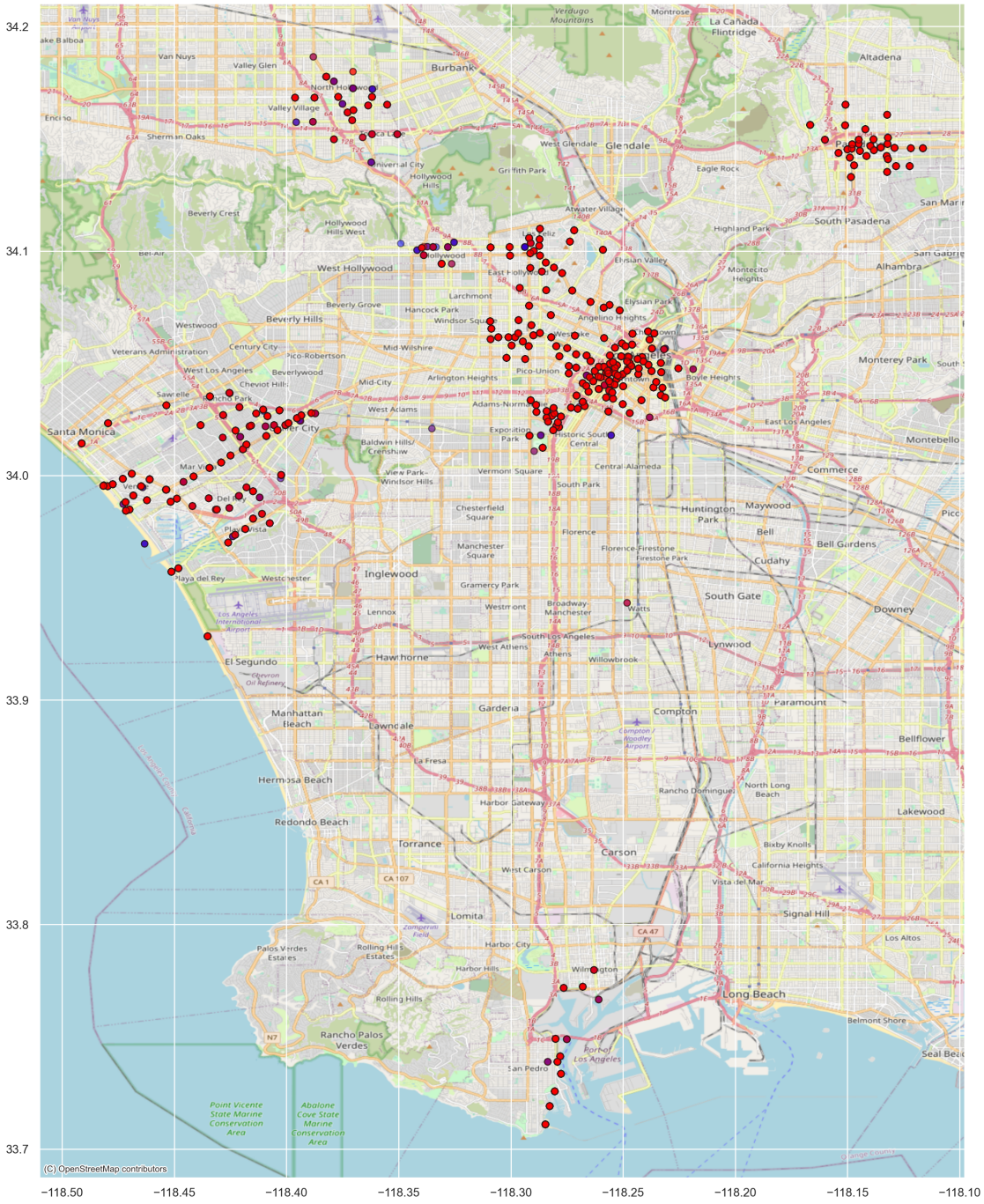
Since we want to get geospatial information the geopandas library will be used to create a geopandas dataframe according to the stations coordinates.

[26]:
```
# Import the useful libraries
import geopandas as gpd
import geoplot as gplt
import contextily as cx
```

1.3.3 Initial and Final station distribution accross LA

[27]:
```
fig,ax = plt.subplots(figsize=(20, 20))    # Figure created to store both plot␣
↪layers
data_vis_geo = data_vis[(data_vis.start_lon<-110)]    # Outlier coordinates␣
↪removal
geopdf = gpd.GeoDataFrame(data_vis_geo, geometry = gpd.
↪points_from_xy(data_vis_geo.start_lon, data_vis_geo.start_lat))
geopdf = geopdf.set_crs(epsg=4326)
ax_g = geopdf.plot(figsize=(20, 20), alpha=0.5, edgecolor='k', color='blue', ax=␣
↪ax)

data_vis_geo = data_vis[(data_vis.end_lat<36)]    # Outlier coordinates removal
geopdf = gpd.GeoDataFrame(data_vis_geo, geometry = gpd.
↪points_from_xy(data_vis_geo.end_lon, data_vis_geo.end_lat))
geopdf = geopdf.set_crs(epsg=4326)
ax_g = geopdf.plot(figsize=(15, 15), alpha=0.2, edgecolor='k', color='red',ax=ax)
cx.add_basemap(ax_g, crs =geopdf.crs, zoom=12, source=cx.providers.OpenStreetMap.
↪Mapnik)
```



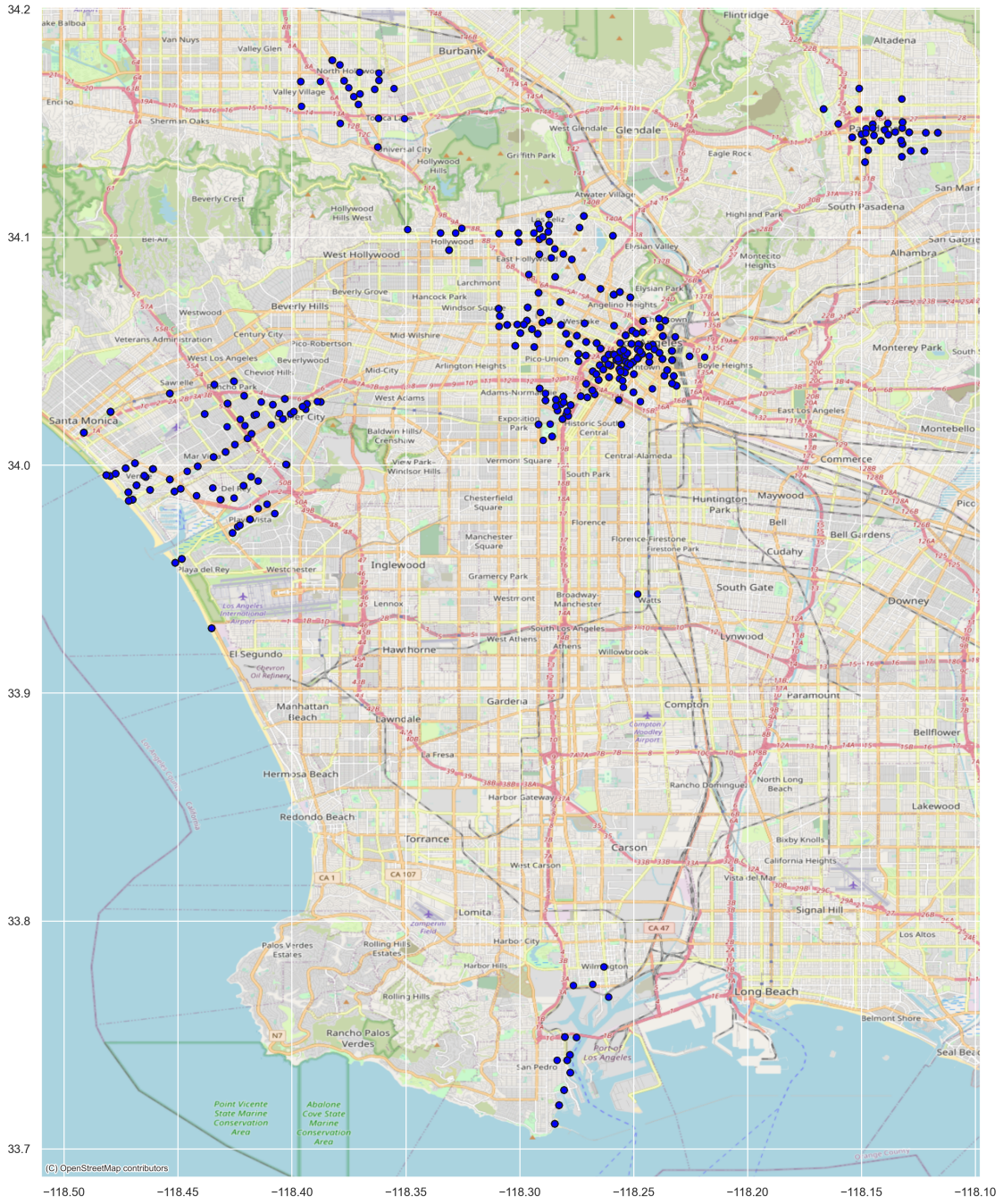
We can observe that, according to the map, **the customers tend to stay within the same area of transit they started their trip.**

1.3.4 Bicicle demand

The service demand will be analyzed according to the density of trips across the day time.

Morning daytime

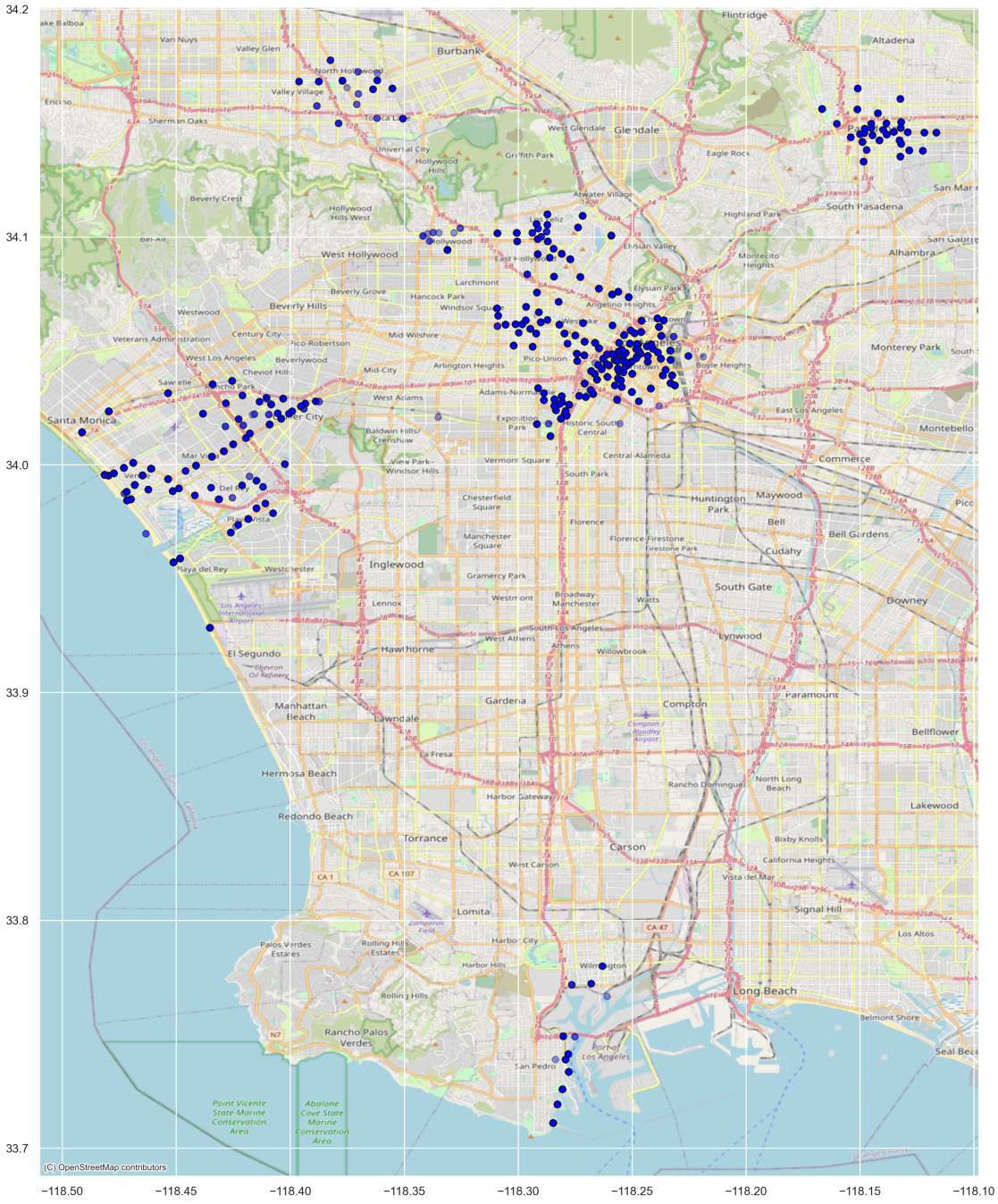
[28]:
```
data_vis_geo = data_vis[(data_vis.start_lon<-110)]    # Outlier coordinates␣
↪removal
data_vis_geo = data_vis_geo[data_vis_geo.day_time == 'morning']
geopdf = gpd.GeoDataFrame(data_vis_geo, geometry = gpd.
↪points_from_xy(data_vis_geo.start_lon, data_vis_geo.start_lat))
geopdf = geopdf.set_crs(epsg=4326)
ax_g = geopdf.plot(figsize=(20, 20), alpha=1, edgecolor='k', color='blue')
cx.add_basemap(ax_g, crs =geopdf.crs, zoom=12, source=cx.providers.OpenStreetMap.
↪Mapnik)
```



Afternoon daytime

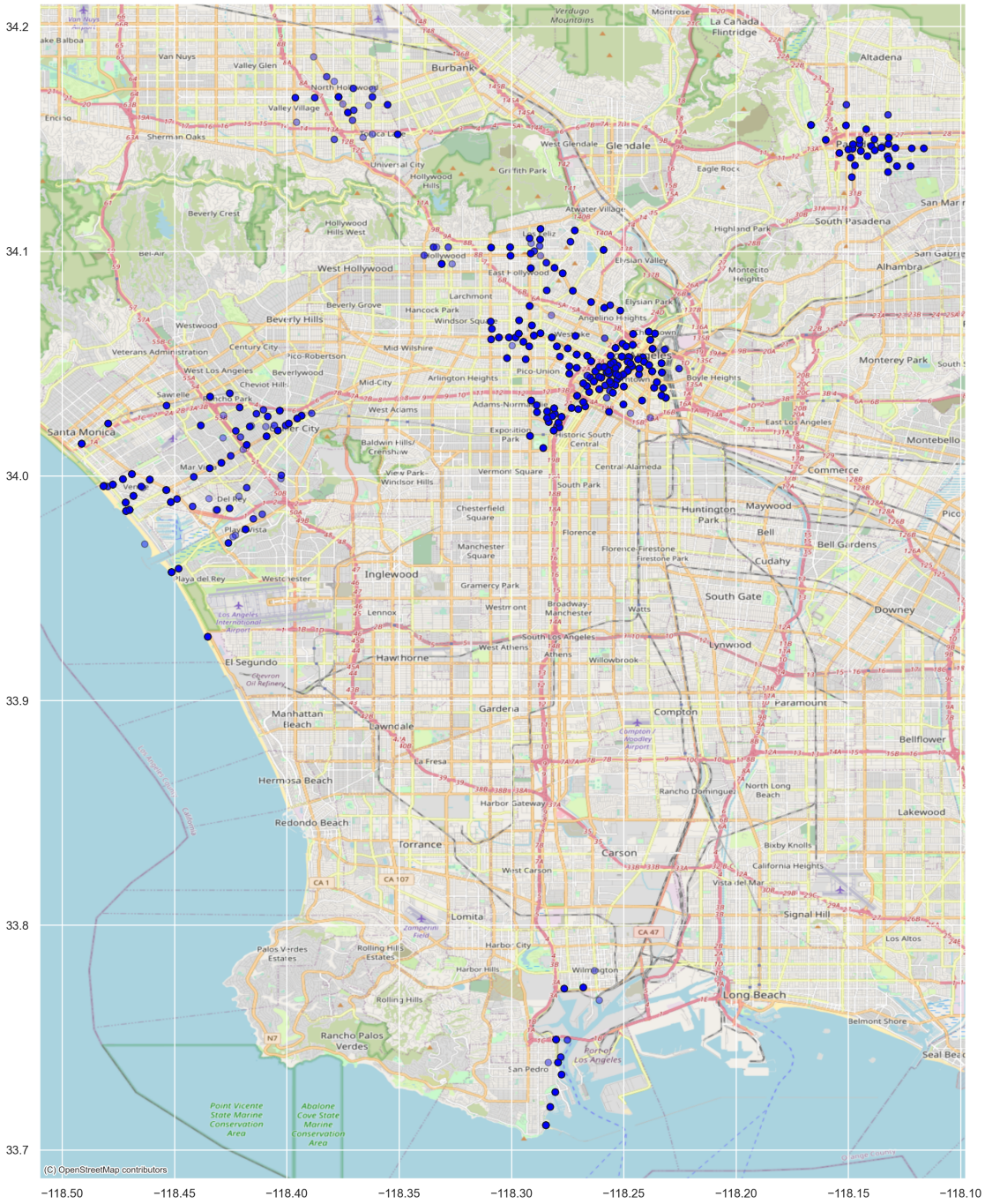
[29]:
```
data_vis_geo = data_vis[(data_vis.start_lon<-110)]
data_vis_geo = data_vis_geo[data_vis_geo.day_time == 'afternoon']
geopdf = gpd.GeoDataFrame(data_vis_geo, geometry = gpd.
 ↪points_from_xy(data_vis_geo.start_lon, data_vis_geo.start_lat))
geopdf = geopdf.set_crs(epsg=4326)
```

```
ax_g = geopdf.plot(figsize=(20, 20), alpha=0.4, edgecolor='k', color='blue')
cx.add_basemap(ax_g, crs =geopdf.crs, zoom=12, source=cx.providers.OpenStreetMap.
↪Mapnik)
```



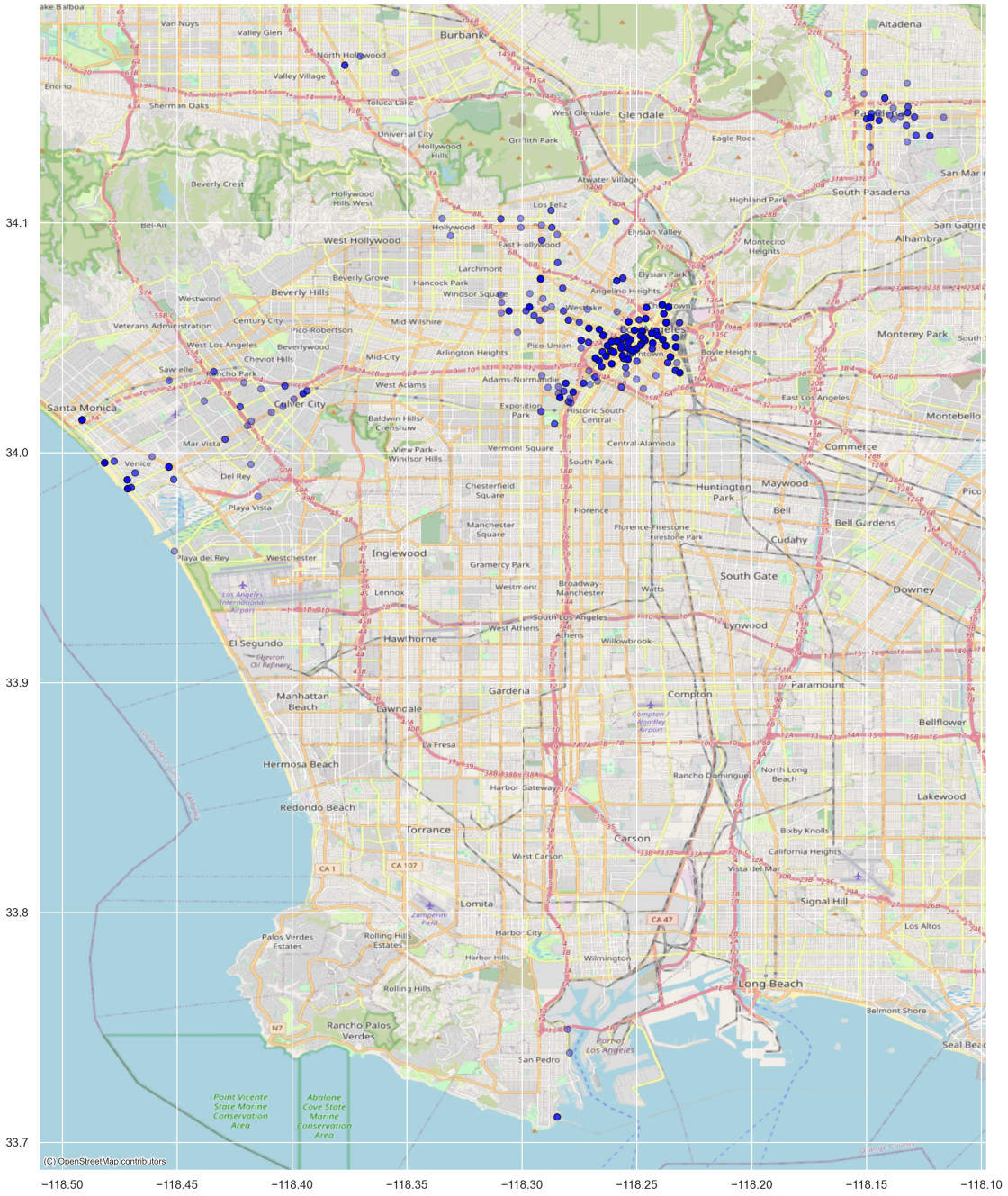
Evening daytime

[30]:
```
data_vis_geo = data_vis[(data_vis.start_lon<-110)]
data_vis_geo = data_vis[data_vis.day_time == 'evening']
geopdf = gpd.GeoDataFrame(data_vis_geo, geometry = gpd.
 ↪points_from_xy(data_vis_geo.start_lon, data_vis_geo.start_lat))
geopdf = geopdf.set_crs(epsg=4326)
ax_g = geopdf.plot(figsize=(20, 20), alpha=0.4, edgecolor='k', color='blue')
cx.add_basemap(ax_g, crs =geopdf.crs, zoom=12, source=cx.providers.OpenStreetMap.
 ↪Mapnik)
```



Night daytime

[31]:
```
data_vis_geo = data_vis[(data_vis.start_lon<-110)]
data_vis_geo = data_vis[data_vis.day_time == 'night']
geopdf = gpd.GeoDataFrame(data_vis_geo, geometry = gpd.
 ↪points_from_xy(data_vis_geo.start_lon, data_vis_geo.start_lat))
geopdf = geopdf.set_crs(epsg=4326)
ax_g = geopdf.plot(figsize=(20, 20), alpha=0.4, edgecolor='k', color='blue')
cx.add_basemap(ax_g, crs =geopdf.crs, zoom=12, source=cx.providers.OpenStreetMap.
 ↪Mapnik)
```



According to the last figures we can get the following insights: 1. There are **5 principal areas of demand**: Center LA (1), Santa Monica (2), San Pedro (3), North Hollywood(4) and Pasadena (5). 2. The **most requested area is (1) Center LA**, followed by (2) Santa Monica, both across al different day times. 3. During the evening, the most requested day time, bicicle requests on North Hollywood and San Pedro falls in comparison to the afternoon.

1.4 EDA Conclusions

Talking about plans growth, this client company had a considerable growth between 2016 and 2018, when it reached its highest number of trips and pass members. However, within the last two years the number of trips has been declining, having its largest fall after 2019. Historically, the company's customer basis has a monthly pass subscription and tend to travel in the afternoon. On the other hand, during the last two years the annual subscriptions has retained its growth despite the global decaying trend.

Finally, talking about availability, we conclude that the most requested stations are the ones near Center LA and Santa Monica while the most requested daytime is during the evening across all stations. Availability can also be tuned, knowing the fact that trips don't take longer than 40 minutes on average and they would be mostly one-way trips.

1.5 Machine Learning Model

The goal is to develop a classification model capable of predicting the kind of pass according to the trip features.

1.5.1 Data Preprocessing

Before creating a ML pipeline, we will define the features that the model will use for the classification, in this regard, a baseline model without further feature engineering is proposed. In this model the feature variables will be transformed using PCA and the number of components will be another parameter for the model tuning. If needed a second model will be proposed using additional info about the stations and/or applying further feature engineering.

1.5.2 Additional dataset cleaning and model boundaries

IMPORTANT NOTICE: The model task is to classify trips' passholder type, so in this regard, the company is only interested on plans that are currently available since after being place in production the model will never have to classify a discontinued type of pass. Keeping data about a class that in practice will never be part of the classification will only generate noise for the model, so *in a real world scenario* it would be suggested to drop the *Flex Pass*. However, due to the fact that the test_set dataset contains trips (observations) from different years, including the ones where the *Flex Pass* was still available, we can't drop this class in this academic scenario.

The first variable that will be droped is `plan_duration` since it's redundant side by side the `passholder_type`. This feature is only another way to describe the target variable so it needs to be removed from the feature set.

[17]:
```
data_t.drop(columns='plan_duration',inplace=True)
```

Similarlly as we defined the `day_time` feature using the `start_time` variable, we can use the `end_time` variable to define a new feature `end_day_time`. This new defined variables will be used to encode the time values for the trip. Additionally, the seasonality of the trip can also be encoded acording to the day of the week, month and year.

[18]:
```
# Split the dataset
X = data_t.drop(columns='passholder_type')
y = data_t.passholder_type
```

[19]:
```
X['end_day_time'] = X.end_time.apply(lambda x: day_time_cls(x))    # Classify␣
↪the end time
```

[20]:
```
# Extract seasonality of the trip
X['year'] = X.start_time.apply(lambda x: x.year)
X['month'] = X.start_time.apply(lambda x: x.month)
X['weekday'] = X.start_time.apply(lambda x: x.weekday())
X.drop(columns=['start_time','end_time'],inplace=True)
```

Now we must encode the categorical predictor variables

[21]:
```
X['trip_route_category'] = X.trip_route_category.astype('category')    # Set the␣
↪dtype to category
X['trip_route_category'] = X.trip_route_category.cat.codes
daytime_enc_dic = {'morning':0, 'afternoon':1, 'evening': 2, 'night':3}
X['day_time_ord'] = X.day_time.map(daytime_enc_dic)
X['end_day_time_ord'] = X.end_day_time.map(daytime_enc_dic)
X['year'] = X.year.apply(lambda x: x-2017)
X.drop(columns=['day_time','end_day_time'],inplace=True)
```

[22]:
```
X['start_station'] = X.start_station.astype('category')
X['start_station'] = X.start_station.cat.codes
X['end_station'] = X.end_station.astype('category')
X['end_station'] = X.end_station.cat.codes
X
```

[22]:

	duration	start_lat	start_lon	end_lat	end_lon	\
0	35	33.748920	-118.275192	33.748920	-118.275192	
1	32	34.035679	-118.270813	34.047749	-118.243172	
2	6	34.046070	-118.233093	34.047749	-118.243172	
3	138	34.062580	-118.290092	34.059689	-118.294662	
4	14	34.026291	-118.277687	34.021660	-118.278687	
...	...	...	...	...	...	
699995	17	34.049889	-118.255882	34.074829	-118.258728	
699996	9	34.039188	-118.232529	34.056610	-118.237213	
699997	6	34.049889	-118.255882	34.050480	-118.254593	
699998	3	34.045181	-118.250237	34.045540	-118.256668	
699999	59	33.984341	-118.471550	34.023392	-118.479637	

	trip_route_category	start_station	end_station	year	month	weekday	\
0	1	69	71	1	8	1	
1	0	44	49	0	9	6	
2	0	14	49	2	4	0	
3	0	147	151	2	9	6	
4	0	137	247	3	1	4	
...	...	...	...	...	...	...	
699995	0	24	128	1	4	2	

```
699996                    0             28            9      2     11        1
699997                    0             24            3      1      1        5
699998                    0             54            2      2      2        0
699999                    0            118          126      4      8        6

        day_time_ord  end_day_time_ord
0                  0                 0
1                  1                 2
2                  0                 0
3                  0                 1
4                  1                 1
...              ...               ...
699995             2                 2
699996             1                 1
699997             1                 1
699998             1                 1
699999             2                 2

[675626 rows x 13 columns]
```

[23]:
```
from sklearn.preprocessing import LabelEncoder
LE = LabelEncoder()
LE.fit(y)
```

[23]:
```
LabelEncoder()
```

[24]:
```
LE.classes_
```

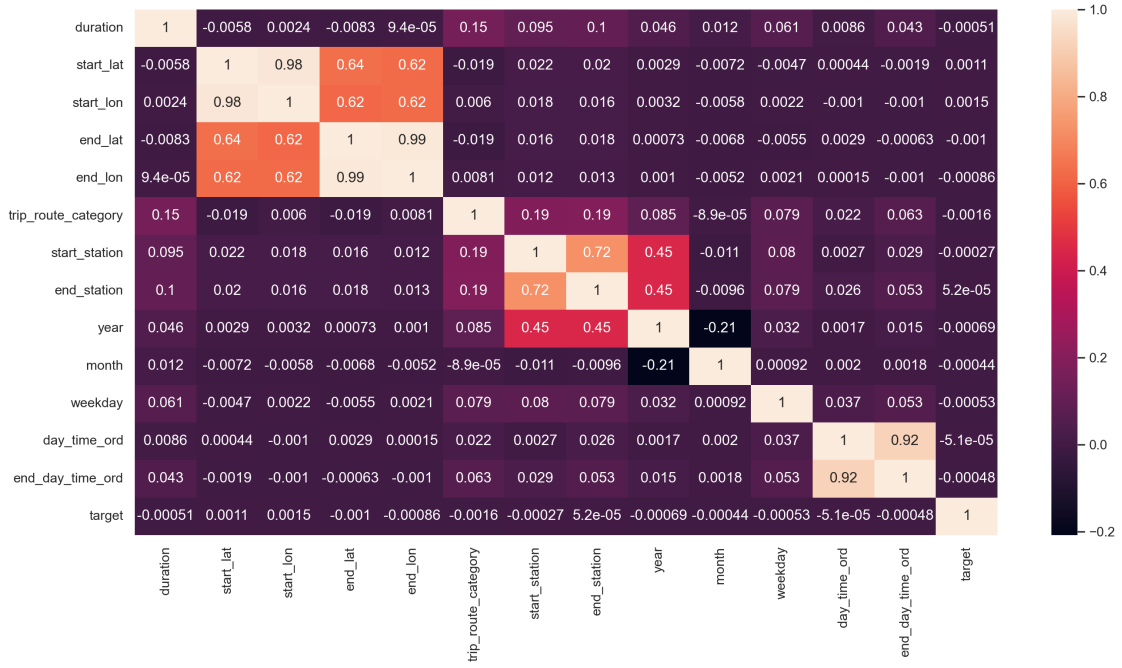
[24]:
```
array(['Annual Pass', 'Flex Pass', 'Monthly Pass', 'One Day Pass',
       'Testing', 'Walk-up'], dtype=object)
```

[25]:
```
y = LE.transform(y)
```

[26]:
```
y_df = pd.Series(y,name='target')
```

[27]:
```
df_cat = X.join(y_df)
corr_df = df_cat.corr(method='pearson')

plt.figure(figsize=(16, 8))
sns.heatmap(corr_df, annot=True)
plt.show()
```



[28]:
```
input_val = X.copy()
```

[29]:
```
output_val = y_df.copy()
```

[30]:
```
input_val.head()
```

[30]:

	duration	start_lat	start_lon	end_lat	end_lon	trip_route_category	start_station	end_station	year	month	weekday	day_time_ord	end_day_time_ord
0	35	33.748920	-118.275192	33.748920	-118.275192	1	69	71	1	8	1	0	0
1	32	34.035679	-118.270813	34.047749	-118.243172	0	44	49	0	9	6	1	2
2	6	34.046070	-118.233093	34.047749	-118.243172	0	14	49	2	4	0	0	0
3	138	34.062580	-118.290092	34.059689	-118.294662	0	147	151	2	9	6	0	1
4	14	34.026291	-118.277687	34.021660	-118.278687	0	137	247	3	1	4	0	

```
4              1                  1
```

[31]:
```
output_val.head()
```

[31]:
```
0    5
1    5
2    2
3    3
4    2
Name: target, dtype: int32
```

1.5.3 SMOTE Oversampling & Random Undersampling

Due to class imbalance on the target variable, SMOTE (Synthetic minority oversampling technique) and Random Undersampling (RU)technique will be used to balance de dataset. The approach to reduce data leackeage will be apply RU and SMOTE to the complete dataset and then do the exclusion method.

[32]:
```
output_val.value_counts()
```

[32]:
```
2    387435
5    203867
3     39888
0     32973
1     11420
4        43
Name: target, dtype: int64
```

[33]:
```
from imblearn.under_sampling import RandomUnderSampler
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import train_test_split

# Undersampling
n = int(len(X)/5)    # Number of final samples per class
sampling_dict = {0:32973, 1:11420, 2:n, 3:39888, 4:43,5:n}   # Number of points␣
 ↪for each class
rus = RandomUnderSampler(random_state=42, sampling_strategy=sampling_dict)
X_rus, y_rus = rus.fit_resample(input_val,output_val)
# SMOTE
sampling_dict = {0:n, 1:n, 2:n, 3:n, 4:n, 5:n}    # Balanced classes
sms = SMOTE(random_state=42, sampling_strategy=sampling_dict)
X_res, y_res = sms.fit_resample(X_rus,y_rus)
```

[34]:
```
X_train_sm,X_test_sm,y_train_sm,y_test_sm = train_test_split(X_res,y_res,␣
 ↪test_size=0.2,
                                                              ␣
 ↪stratify=y_res,random_state=42)
```

```
print("X_train Shape : ", X_train_sm.shape)
print("X_test Shape  : ", X_test_sm.shape)
print("y_train Shape : ", y_train_sm.shape)
print("y_test Shape  : ", y_test_sm.shape)
```

```
X_train Shape :  (648600, 13)
X_test Shape  :  (162150, 13)
y_train Shape :  (648600,)
y_test Shape  :  (162150,)
```

[35]:
```
X_train_sm.describe()
```

[35]:

	duration	start_lat	start_lon	end_lat	\
count	648600.000000	648600.000000	648600.000000	648600.000000	
mean	34.146050	34.041553	-118.265566	34.039174	
std	106.405846	0.392294	2.820856	0.324412	
min	1.000000	33.710979	-118.495422	33.710979	
25%	6.000000	34.014309	-118.418409	34.014309	
50%	12.000000	34.045181	-118.260948	34.044701	
75%	25.000000	34.050880	-118.248770	34.049980	
max	1440.000000	55.705528	118.238258	55.705528	

	end_lon	trip_route_category	start_station	end_station	\
count	648600.000000	648600.000000	648600.000000	648600.000000	
mean	-118.282560	0.231721	101.418523	101.657598	
std	2.313953	0.421932	88.782563	89.118662	
min	-118.495422	0.000000	0.000000	0.000000	
25%	-118.419702	0.000000	27.000000	27.000000	
50%	-118.260948	0.000000	59.000000	61.000000	
75%	-118.248352	0.000000	171.000000	174.000000	
max	37.606541	1.000000	361.000000	365.000000	

	year	month	weekday	day_time_ord	\
count	648600.000000	648600.000000	648600.000000	648600.000000	
mean	1.766980	6.561560	2.836930	0.860487	
std	1.333715	3.220986	1.886476	0.743431	
min	-1.000000	1.000000	0.000000	0.000000	
25%	1.000000	4.000000	1.000000	0.000000	
50%	2.000000	7.000000	3.000000	1.000000	
75%	3.000000	9.000000	4.000000	1.000000	
max	4.000000	12.000000	6.000000	3.000000	

	end_day_time_ord
count	648600.000000
mean	0.932863
std	0.748583
min	0.000000

```
25%            0.000000
50%            1.000000
75%            1.000000
max            3.000000
```

1.5.4 Modelling

We will compare 5 different algorithms and their according bussines metrics.

a. Logistic Regression
b. Decission Trees
c. Support Vector Machine
d. Random Forest
e. GVCBossting

For this classification problem a good bussines metric is the accuracy score, as we are dealing with a multi-class classification problem. A high accuracy scored modell will be the one with a balanced correct prediction among all classes.

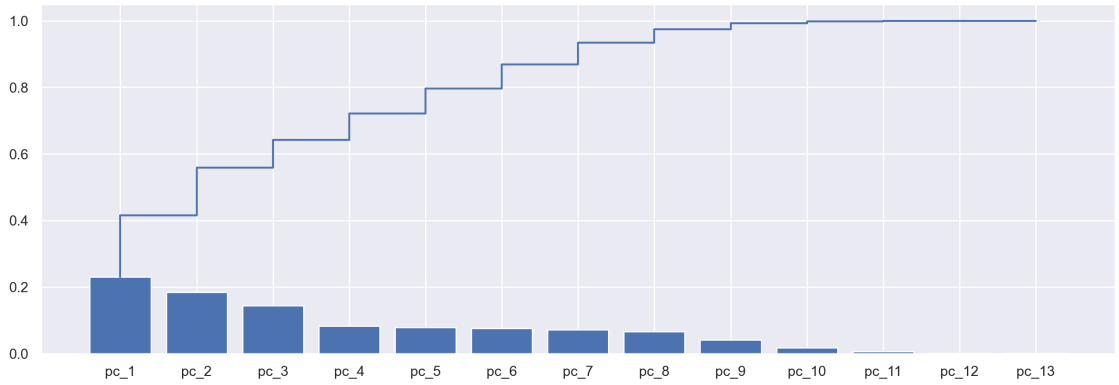
[38]:
```
from sklearn.model_selection import GridSearchCV, cross_val_score
from sklearn.metrics import f1_score, recall_score, precision_score,␣
 ↪confusion_matrix, classification_report,accuracy_score
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
```

[39]:
```
pca = PCA()
scaler = StandardScaler()
X_pca = scaler.fit_transform(X_train_sm)
pca.fit(X_pca)

var_ratio= pca.explained_variance_ratio_
cumu_sum= np.cumsum(var_ratio)
cumu_sum
labels= ["pc_{}".format(n+1) for n in range(len(var_ratio))]

plt.bar(labels, var_ratio)
plt.step(labels, cumu_sum, where= "pre")
plt.plot
```

[39]:
```
<function matplotlib.pyplot.plot(*args, scalex=True, scaley=True, data=None,
**kwargs)>
```



Using PCA can help reduce the feature dimensionality, but it can also help the classification and regression problems by preventing overfitting and reducing noise. It will be added to the pipeline, in order to adjust the number of components and prevent data leakage.

Baseline Models The baseline models will be the more simple ones: Logistic Regression, Decision Tree and SVM. PCA transform will also use all the components

Logistic Regression

[42]:
```
from sklearn.linear_model import LogisticRegression
# Creating the sklearn pipeline
logi = LogisticRegression(max_iter=10000, tol=0.1)
scaler = StandardScaler()
pca = PCA()
logi_pipe = Pipeline(steps=[('scaler',scaler),('pca',pca),('clf',logi)])
logi_pipe.fit(X_train_sm,y_train_sm)
```

[42]:
```
Pipeline(steps=[('scaler', StandardScaler()), ('pca', PCA()),
                ('clf', LogisticRegression(max_iter=10000, tol=0.1))])
```

[43]:
```
# Test set f1 score
logi_pipe.score(X_test_sm,y_test_sm)
```

[43]:
```
0.5304101140918902
```

Decision Tree

[46]:
```
from sklearn.tree import DecisionTreeClassifier
# sklearn pipeline
tree = DecisionTreeClassifier(random_state=42)
scaler = StandardScaler()
pca = PCA()
tree_pipe = Pipeline(steps=[('scaler',scaler),('pca',pca),('clf',tree)])
tree_pipe.fit(X_train_sm,y_train_sm)
```

[46]:
```
Pipeline(steps=[('scaler', StandardScaler()), ('pca', PCA()),
                ('clf', DecisionTreeClassifier(random_state=42))])
```

[47]:
```
tree_pipe.score(X_test_sm,y_test_sm)
```

[47]:
```
0.7251310514955288
```

Support Vector Classifier

[48]:
```
from sklearn.svm import SVC

svc = SVC(random_state=42,max_iter=50, verbose=True)
scaler = StandardScaler()
pca = PCA()
svc_pipe = Pipeline(steps=[('scaler',scaler),('pca',pca),('clf',svc)])
svc_pipe.fit(X_train_sm,y_train_sm)
```

```
[LibSVM]
C:\Users\Andrew S\anaconda3\lib\site-packages\sklearn\svm\_base.py:284:
ConvergenceWarning:

Solver terminated early (max_iter=50).  Consider pre-processing your data with
StandardScaler or MinMaxScaler.
```

[48]:
```
Pipeline(steps=[('scaler', StandardScaler()), ('pca', PCA()),
                ('clf', SVC(max_iter=50, random_state=42, verbose=True))])
```

[49]:
```
svc_pipe.score(X_test_sm,y_test_sm)
```

[49]:
```
0.2324699352451434
```

1.5.5 Complex models

Here ML ensembles will be tested: RandomForest and XGBoostClassifier algorithms.

Random Forest

[50]:
```
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier(random_state=42, n_jobs=-1)
scaler = StandardScaler()
pca = PCA()
rf_pipe = Pipeline(steps=[('scaler',scaler),('pca',pca),('clf',rf)])
rf_pipe.fit(X_train_sm,y_train_sm)
```

[50]:
```
Pipeline(steps=[('scaler', StandardScaler()), ('pca', PCA()),
                ('clf', RandomForestClassifier(n_jobs=-1, random_state=42))])
```

[51]:
```
rf_pipe.score(X_test_sm,y_test_sm)
```

[51]:
```
0.7841258094357076
```

XGBoost Classifier

[95]:
```
from sklearn.ensemble import GradientBoostingClassifier

gb = GradientBoostingClassifier(random_state=42,verbose=1,n_estimators=80,tol=0.
↪001)
scaler = StandardScaler()
pca = PCA()
gb_pipe = Pipeline(steps=[('scaler',scaler),('pca',pca),('clf',gb)])
gb_pipe.fit(X_train_sm,y_train_sm)
```

```
      Iter       Train Loss   Remaining Time
         1           1.6630           27.60m
         2           1.5697           26.86m
         3           1.4964           26.35m
         4           1.4373           26.07m
         5           1.3866           25.47m
         6           1.3439           24.97m
         7           1.3075           24.83m
         8           1.2760           24.47m
         9           1.2488           24.13m
        10           1.2253           23.78m
        20           1.0935           20.22m
        30           1.0380           16.83m
        40           1.0067           13.50m
        50           0.9879           10.12m
        60           0.9741            6.74m
        70           0.9623            3.35m
        80           0.9516            0.00s
```

[95]:
```
Pipeline(steps=[('scaler', StandardScaler()), ('pca', PCA()),
                ('clf',
                 GradientBoostingClassifier(n_estimators=80, random_state=42,
                                            tol=0.001, verbose=1))])
```

[97]:
```
gb_pipe.score(X_test_sm,y_test_sm)
```

[97]:
```
0.6123280912735122
```

1.5.6 Modelling: CrossValidation and Parameter Tuning

Once we have detected the potential candidates for the clasificaction, CrossValidation and/or GridSearchCV will be implemented

[68]:
```
from sklearn.model_selection import GridSearchCV
```

GSCV-Logistic Regression

[52]:
```
# Creating the sklearn pipeline for CV
logi = LogisticRegression(max_iter=10000, tol=0.1,verbose=1)
scaler = StandardScaler()
pca = PCA()
logi_pipe = Pipeline(steps=[('scaler',scaler),('pca',pca),('clf',logi)])
logi_param_grid = {'pca__n_components':[5,7,9],'clf__solver': ['lbfgs'],
                   'clf__C': [0.001,0.01,0.1,1,10,100]}
metrics = ['accuracy','f1_macro']
logi_search = GridSearchCV(logi_pipe, logi_param_grid,cv=3,
                           verbose=2,scoring=metrics,
                           return_train_score=True,n_jobs=-1, refit='accuracy')
logi_search.fit(X_train_sm,y_train_sm)
```

```
Fitting 3 folds for each of 18 candidates, totalling 54 fits
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:   25.1s finished
```

[52]:
```
GridSearchCV(cv=3,
             estimator=Pipeline(steps=[('scaler', StandardScaler()),
                                       ('pca', PCA()),
                                       ('clf',
                                        LogisticRegression(max_iter=10000,
                                                           tol=0.1,
                                                           verbose=1))]),
             n_jobs=-1,
             param_grid={'clf__C': [0.001, 0.01, 0.1, 1, 10, 100],
                         'clf__solver': ['lbfgs'],
                         'pca__n_components': [5, 7, 9]},
             refit='accuracy', return_train_score=True,
             scoring=['accuracy', 'f1_macro'], verbose=2)
```

[53]:
```
def print_best(search):
    # score - test set
    print('Score:',search.score(X_test_sm,y_test_sm))
    # Best parameters
    print('Best params',search.best_params_)
    return
def result_df(search):
    df = pd.DataFrame(search.cv_results_)
    return df.T
```

[54]:
```
print_best(logi_search)
```

```
Score: 0.5226765340733889
Best params {'clf__C': 0.1, 'clf__solver': 'lbfgs', 'pca__n_components': 9}
```

GSCV-DecisionTree

[102]:
```
tree = DecisionTreeClassifier(random_state=42)
scaler = StandardScaler()
pca = PCA()
tree_pipe = Pipeline(steps=[('scaler',scaler),('pca',pca),('clf',tree)])
tree_param_grid = {'clf__criterion': ['gini'],
                   'clf__max_depth': [20,50,70,90],
                   'clf__min_samples_leaf': [10,20,30],
                  'clf__splitter':['random'],'pca__n_components':[9]}
metrics = ['accuracy','f1_macro']
tree_search = GridSearchCV(tree_pipe, tree_param_grid,cv=3,
                           verbose=1,scoring=metrics,
                           return_train_score=True,n_jobs=-1, refit='accuracy')
tree_search.fit(X_train_sm,y_train_sm)
```

```
Fitting 3 folds for each of 12 candidates, totalling 36 fits
```

```
---------------------------------------------------------------------------
_RemoteTraceback                          Traceback (most recent call last)
_RemoteTraceback:
"""
Traceback (most recent call last):
  File "C:\Users\Andrew␣
↪S\anaconda3\lib\site-packages\joblib\externals\loky\process_executor.py", line
↪436, in _process_worker
    r = call_item()
  File "C:\Users\Andrew␣
↪S\anaconda3\lib\site-packages\joblib\externals\loky\process_executor.py", line
↪288, in __call__
    return self.fn(*self.args, **self.kwargs)
  File "C:\Users\Andrew S\anaconda3\lib\site-packages\joblib\_parallel_backends.
↪py", line 595, in __call__
    return self.func(*args, **kwargs)
  File "C:\Users\Andrew S\anaconda3\lib\site-packages\joblib\parallel.py", line␣
↪262, in __call__
    return [func(*args, **kwargs)
  File "C:\Users\Andrew S\anaconda3\lib\site-packages\joblib\parallel.py", line␣
↪262, in <listcomp>
    return [func(*args, **kwargs)
  File "C:\Users\Andrew S\anaconda3\lib\site-packages\sklearn\utils\fixes.py",␣
↪line 216, in __call__
    return self.function(*args, **kwargs)
  File "C:\Users\Andrew␣
↪S\anaconda3\lib\site-packages\sklearn\model_selection\_validation.py", line␣
↪668, in _fit_and_score
```

```
    estimator = estimator.set_params(**cloned_parameters)
  File "C:\Users\Andrew S\anaconda3\lib\site-packages\sklearn\pipeline.py", line␣
 ↪188, in set_params
    self._set_params("steps", **kwargs)
  File "C:\Users\Andrew S\anaconda3\lib\site-packages\sklearn\utils\metaestimator
 ↪py", line 54, in _set_params
    super().set_params(**params)
  File "C:\Users\Andrew S\anaconda3\lib\site-packages\sklearn\base.py", line 258␣
 ↪in set_params
    valid_params[key].set_params(**sub_params)
  File "C:\Users\Andrew S\anaconda3\lib\site-packages\sklearn\base.py", line 245␣
 ↪in set_params
    raise ValueError(
ValueError: Invalid parameter n_estimators for estimator␣
 ↪DecisionTreeClassifier(max_depth=1, random_state=42). Check the list of␣
 ↪available parameters with `estimator.get_params().keys()`.
"""

The above exception was the direct cause of the following exception:

ValueError                                Traceback (most recent call last)
Input In [102], in <cell line: 12>()
      8 metrics = ['accuracy','f1_macro']
      9 tree_search = GridSearchCV(tree_pipe, tree_param_grid,cv=3,
     10                             verbose=1,scoring=metrics,
     11                             return_train_score=True,n_jobs=-1,␣
 ↪refit='accuracy')
---> 12 tree_search.fit(X_train_sm,y_train_sm)

File ~\anaconda3\lib\site-packages\sklearn\model_selection\_search.py:891, in␣
 ↪BaseSearchCV.fit(self, X, y, groups, **fit_params)
    885     results = self._format_results(
    886         all_candidate_params, n_splits, all_out, all_more_results
    887     )
    889     return results
--> 891 self._run_search(evaluate_candidates)
    893 # multimetric is determined here because in the case of a callable
    894 # self.scoring the return type is only known after calling
    895 first_test_score = all_out[0]["test_scores"]

File ~\anaconda3\lib\site-packages\sklearn\model_selection\_search.py:1392, in␣
 ↪GridSearchCV._run_search(self, evaluate_candidates)
   1390 def _run_search(self, evaluate_candidates):
   1391     """Search all candidates in param_grid"""
-> 1392     evaluate_candidates(ParameterGrid(self.param_grid))
```

```
File ~\anaconda3\lib\site-packages\sklearn\model_selection\_search.py:838, in␣
 ↪BaseSearchCV.fit.<locals>.evaluate_candidates(candidate_params, cv,␣
 ↪more_results)
    830 if self.verbose > 0:
    831     print(
    832         "Fitting {0} folds for each of {1} candidates,"
    833         " totalling {2} fits".format(
    834             n_splits, n_candidates, n_candidates * n_splits
    835         )
    836     )
--> 838 out = parallel(
    839     delayed(_fit_and_score)(
    840         clone(base_estimator),
    841         X,
    842         y,
    843         train=train,
    844         test=test,
    845         parameters=parameters,
    846         split_progress=(split_idx, n_splits),
    847         candidate_progress=(cand_idx, n_candidates),
    848         **fit_and_score_kwargs,
    849     )
    850     for (cand_idx, parameters), (split_idx, (train, test)) in product(
    851         enumerate(candidate_params), enumerate(cv.split(X, y, groups))
    852     )
    853 )
    855 if len(out) < 1:
    856     raise ValueError(
    857         "No fits were performed. "
    858         "Was the CV iterator empty? "
    859         "Were there no candidates?"
    860     )

File ~\anaconda3\lib\site-packages\joblib\parallel.py:1056, in Parallel.
 ↪__call__(self, iterable)
   1053     self._iterating = False
   1055 with self._backend.retrieval_context():
-> 1056     self.retrieve()
   1057 # Make sure that we get a last message telling us we are done
   1058 elapsed_time = time.time() - self._start_time

File ~\anaconda3\lib\site-packages\joblib\parallel.py:935, in Parallel.
 ↪retrieve(self)
    933 try:
    934     if getattr(self._backend, 'supports_timeout', False):
--> 935         self._output.extend(job.get(timeout=self.timeout))
    936     else:
    937         self._output.extend(job.get())
```

```
File ~\anaconda3\lib\site-packages\joblib\_parallel_backends.py:542, in␣
 ↪LokyBackend.wrap_future_result(future, timeout)
    539 """Wrapper for Future.result to implement the same behaviour as
    540 AsyncResults.get from multiprocessing."""
    541 try:
--> 542     return future.result(timeout=timeout)
    543 except CfTimeoutError as e:
    544     raise TimeoutError from e

File ~\anaconda3\lib\concurrent\futures\_base.py:439, in Future.result(self,␣
 ↪timeout)
    437     raise CancelledError()
    438 elif self._state == FINISHED:
--> 439     return self.__get_result()
    440 else:
    441     raise TimeoutError()

File ~\anaconda3\lib\concurrent\futures\_base.py:388, in Future.__get_result(self)
    386 def __get_result(self):
    387     if self._exception:
--> 388         raise self._exception
    389     else:
    390         return self._result

ValueError: Invalid parameter n_estimators for estimator␣
 ↪DecisionTreeClassifier(max_depth=1, random_state=42). Check the list of␣
 ↪available parameters with `estimator.get_params().keys()`.
```

[99]:
```
print_best(tree_search)
```

```
Score: 0.6605303731113167
Best params {'clf__criterion': 'gini', 'clf__max_depth': 50,
'clf__min_samples_leaf': 10, 'clf__splitter': 'random', 'pca__n_components': 9}
```

[59]:
```
tree_search.best_params_
```

[59]:
```
{'clf__criterion': 'gini',
 'clf__max_depth': 50,
 'clf__max_features': 'log2',
 'clf__min_samples_leaf': 10,
 'clf__splitter': 'random',
 'pca__n_components': 9}
```

GSCV-RandomForest

[110]:
```
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier(random_state=42,verbose=1,n_jobs=-1,n_estimators=50)
scaler = StandardScaler()
pca = PCA()
rf_pipe = Pipeline(steps=[('scaler',scaler),('pca',pca),('clf',rf)])
rf_param_grid = {'clf__criterion': ['gini'], 'clf__max_depth': [75,90],
                 'clf__max_leaf_nodes': [1000,5000],'pca__n_components':[5,7,9]}
metrics = ['accuracy','f1_macro']
rf_search = GridSearchCV(rf_pipe, rf_param_grid,cv=4,
                         verbose=1,scoring=metrics,
                         return_train_score=True,n_jobs=11, refit='accuracy')
rf_search.fit(X_train_sm,y_train_sm)
```

```
Fitting 4 folds for each of 12 candidates, totalling 48 fits

[Parallel(n_jobs=-1)]: Using backend ThreadingBackend with 12 concurrent
workers.
[Parallel(n_jobs=-1)]: Done  26 tasks      | elapsed:   16.8s
[Parallel(n_jobs=-1)]: Done  50 out of  50 | elapsed:   26.0s finished
```

[110]:
```
GridSearchCV(cv=4,
             estimator=Pipeline(steps=[('scaler', StandardScaler()),
                                       ('pca', PCA()),
                                       ('clf',
                                        RandomForestClassifier(n_estimators=50,
                                                               n_jobs=-1,
                                                               random_state=42,
                                                               verbose=1))]),
             n_jobs=11,
             param_grid={'clf__criterion': ['gini'], 'clf__max_depth': [75, 90],
                         'clf__max_leaf_nodes': [1000, 5000],
                         'pca__n_components': [5, 7, 9]},
             refit='accuracy', return_train_score=True,
             scoring=['accuracy', 'f1_macro'], verbose=1)
```

[111]:
```
print_best(rf_search)
```

```
[Parallel(n_jobs=12)]: Using backend ThreadingBackend with 12 concurrent
workers.

Score: 0.6873573851372187
Best params {'clf__criterion': 'gini', 'clf__max_depth': 75,
'clf__max_leaf_nodes': 5000, 'pca__n_components': 9}

[Parallel(n_jobs=12)]: Done  26 tasks      | elapsed:    0.1s
[Parallel(n_jobs=12)]: Done  50 out of  50 | elapsed:    0.2s finished
```

[113]:
```
rf_search.best_params_
```

[113]:
```
{'clf__criterion': 'gini',
 'clf__max_depth': 75,
 'clf__max_leaf_nodes': 5000,
 'pca__n_components': 9}
```

1.6 Results

1.6.1 Modelling: Selection

It's now time to evaluate the best models for each algorithm, Accuracy score will be the metric to be considered also taking account that the model has a good F1 score overall. The reason behind choosing accuracy as the main target meassure is because we are dealing with a multiclass classification problem.

[85]:
```
from sklearn.metrics import recall_score
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
```

Logistic Regression Summary

[65]:
```
# Creating the sklearn pipeline for CV
logi = LogisticRegression(max_iter=10000, tol=0.1, solver='lbfgs', C=0.1)
pca = PCA(n_components=9)
scaler = StandardScaler()
logi_pipe = Pipeline(steps=[('scaler',scaler),('pca',pca),('clf',logi)])
logi_pipe.fit(X_train_sm,y_train_sm)
y_pred_logi = logi_pipe.predict(X_test_sm)
print(confusion_matrix(y_test_sm,y_pred_logi))
print("\n")
print(classification_report(y_test_sm, y_pred_logi,digits=6))
```

```
[[16443     3  6063  2253  1518   745]
 [   37 22847  2615    55   330  1141]
 [ 7612  7526  6747  1601  1048  2491]
 [ 5213  1944  3089  7505  4580  4694]
 [  569     0    47  2034 23465   910]
 [ 3162  4190  4380  3916  3631  7746]]
```

	precision	recall	f1-score	support
0	0.497730	0.608437	0.547543	27025
1	0.625774	0.845402	0.719194	27025
2	0.294102	0.249658	0.270064	27025
3	0.432216	0.277706	0.338147	27025
4	0.678728	0.868270	0.761888	27025
5	0.436961	0.286623	0.346174	27025

```
    accuracy                           0.522683    162150
   macro avg   0.494252  0.522683  0.497168    162150
weighted avg   0.494252  0.522683  0.497168    162150
```

Decision Tree Summary

[66]:
```
tree = DecisionTreeClassifier(random_state=42, criterion='gini',
                              max_depth=50, max_features='log2',␣
 ↪min_samples_leaf=10, splitter='random' )
scaler = StandardScaler()
pca = PCA(n_components=9)
tree_pipe = Pipeline(steps=[('scaler',scaler),('clf',tree)])
tree_pipe.fit(X_train_sm,y_train_sm)
y_pred_tree = tree_pipe.predict(X_test_sm)
print(confusion_matrix(y_test_sm,y_pred_tree))
print("\n")
print(classification_report(y_test_sm, y_pred_tree,digits=6,
                            target_names=['Annual Pass', 'Flex Pass', 'Monthly␣
 ↪Pass', 'One Day Pass', 'Testing',
       'Walk-up']))
```

```
[[21346    63  2349  2051   161  1055]
 [  196 23672  1512   624     0  1021]
 [ 7626  6486  7440  2492    95  2886]
 [ 4537  1727  1961 14286   775  3739]
 [    8     0     7    47 26956     7]
 [ 3769  3708  3205  5804   381 10158]]


              precision    recall  f1-score   support

 Annual Pass   0.569500  0.789861  0.661820     27025
   Flex Pass   0.663899  0.875930  0.755317     27025
Monthly Pass   0.451621  0.275301  0.342077     27025
One Day Pass   0.564575  0.528622  0.546007     27025
     Testing   0.950226  0.997447  0.973264     27025
     Walk-up   0.538429  0.375874  0.442701     27025

    accuracy                       0.640506    162150
   macro avg   0.623042  0.640506  0.620198    162150
weighted avg   0.623042  0.640506  0.620198    162150
```

RandomForest Summary

[126]:
```
rf = RandomForestClassifier(random_state=42,n_estimators=50, criterion='gini',␣
 ↪max_depth=75, max_leaf_nodes=5000,n_jobs=-1)
```

```
scaler = StandardScaler()
pca = PCA(n_components=9)
rf_pipe = Pipeline(steps=[('scaler',scaler),('clf',rf)])
rf_pipe.fit(X_train_sm,y_train_sm)
y_pred_rf = rf_pipe.predict(X_test_sm)
print(confusion_matrix(y_test_sm,y_pred_rf))
print("\n")
print(classification_report(y_test_sm, y_pred_rf,digits=6,
                            target_names=['Annual Pass', 'Flex Pass', 'Monthly␣
 ↪Pass', 'One Day Pass', 'Testing',
       'Walk-up']))
```

```
[[23670     7  1338  1529    82   399]
 [   41 25128   693   480     0   683]
 [ 5040  4742 12602  2225    55  2361]
 [ 2346   919  1787 18877   166  2930]
 [    2     0     1     0 27022     0]
 [ 2463  2322  3600  5747   117 12776]]


              precision    recall  f1-score   support

 Annual Pass   0.705262  0.875856  0.781356     27025
   Flex Pass   0.758741  0.929806  0.835608     27025
Monthly Pass   0.629439  0.466309  0.535731     27025
One Day Pass   0.654134  0.698501  0.675590     27025
     Testing   0.984695  0.999889  0.992234     27025
     Walk-up   0.667189  0.472747  0.553385     27025

    accuracy                       0.740518    162150
   macro avg   0.733243  0.740518  0.728984    162150
weighted avg   0.733243  0.740518  0.728984    162150
```

1.6.2 Discussion of Results

After comparing baseline models and applying Cross-Validation and parameter tuning, the best model overall is RandomForest Classifier with the following parameters:

1. Criterion: Gini
2. Max depth: 75
3. Max leaf nodes: 5000
4. PCA - 9 components

Parameter tuning was needed since the baseline models presented overfitting because the train score was significantly higher than the test score. With the selected model we have lost, raw accuracy and f1 score but we have won confidence in our model capability of generalization. In real practice, a balanced model is preferred over a high variance model with a higher score. Our selected model was verified following the industry-standard cross-validation.

[127]:
```
import itertools

def plot_confusion_matrix(cm, classes,normalize=True,title='Confusion␣
 ↪matrix',cmap=plt.cm.Blues):

    if normalize:
        cm = np.round(cm.astype('float') / cm.sum(axis=1)[:, np.newaxis],4)
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=0)
    plt.yticks(tick_marks, classes)
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, cm[i, j],
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

[128]:
```
import matplotlib as mpl
mpl.rc('font',family='Times New Roman')
```

[129]:
```
class_names = ['Annual Pass', 'Flex Pass', 'Monthly Pass', 'One Day Pass',␣
 ↪'Testing','Walk-up']
plt.figure(figsize=(8.5,8.5), dpi=100)
plot_confusion_matrix(confusion_matrix(y_test_sm,y_pred_rf)
                      , classes=class_names)
plt.grid(False)
plt.show()
```

```
C:\Users\Andrew S\AppData\Local\Temp\ipykernel_2816\1036016505.py:9:
MatplotlibDeprecationWarning:

Auto-removal of grids by pcolor() and pcolormesh() is deprecated since 3.5 and
will be removed two minor releases later; please call grid(False) first.
```

The confusion matrix of the selected model is presented, and with it we can see which classes are the more difficult for the model to classify. We note that the Walk-up and Monthly passholders type have the least amount of correct predictions, while the Test and Flex passholders has the largest percentage of correct predictions.

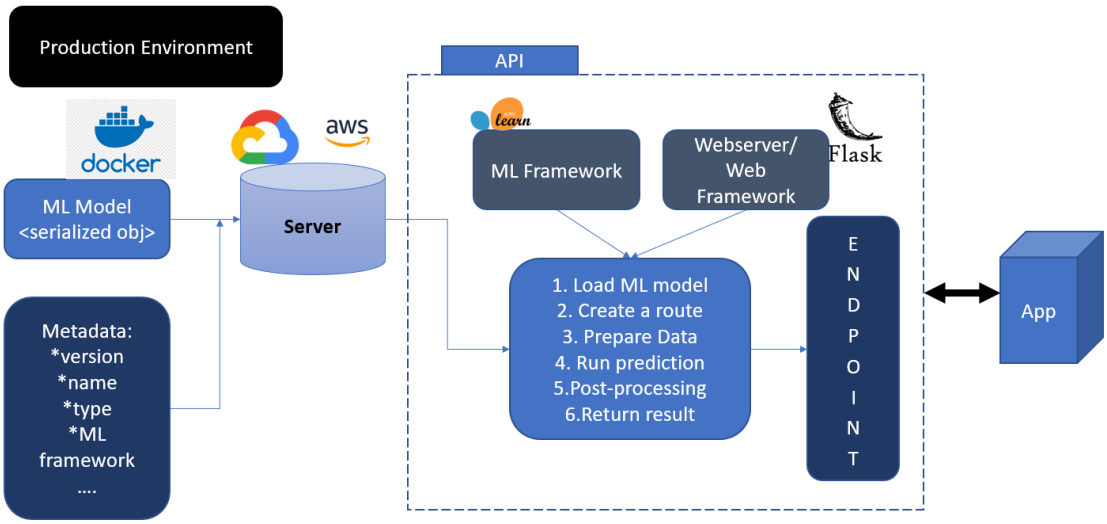
1.6.3 Conclussions & Further work

A RandomForest Classifier algorithm with 0.687 +/- 0.001 accuracy for passholder type prediction is presented. The model was selected among other classification algorithms such as LogisticRegression, DecisionTree and GradientBoostClassifier. The selection was in accordance of the cross-validation and parameter-tuning results selecting the most balanced model overall, avoiding overfitting. The models were trained using a preprocesed feature-engineered dataset. Categorical variables were encoded and the features were transformed using PCA. Nevertheless, further work is highly suggested

in order to increase the model scores, to achieve this further feature engineering and variables selection could be made as well as a more robust comparison between different models and a deeper parameter tuning.

1.6.4 Model deployment

The results and conclusions found in this notebook could be used to the model deployment since the preprocess methodology, classification algorithm and parameters have been found and defined. In order to achieve the deployment, the model must be exported as a serialized object to be later sent to a production environment. The flow diagram is presented next:



Finally and as a first step in production, the presented model will be packed in a dockerfile image called model-docker

1.6.5 Kaggle Validation

The obtained model will be used to predict and tag a set of observations.

[130]:
```
validation = pd.read_csv('test_set.csv')
val_id = validation.trip_id
validation = validation.drop(columns=['trip_id','bike_id'])
validation.start_lat.fillna(value=data.start_lat.mean(),inplace=True)
validation.start_lon.fillna(value=data.start_lon.mean(),inplace=True)
validation.end_lon.fillna(value=data.end_lat.mean(),inplace=True)
validation.end_lat.fillna(value=data.end_lat.mean(),inplace=True)
```

```
C:\Users\Andrew S\AppData\Local\Temp\ipykernel_2816\4164149916.py:1:
DtypeWarning:

Columns (8) have mixed types. Specify dtype option on import or set
low_memory=False.
```

[131]:
```
val_t = validation.copy()
val_t['start_time'] = pd.to_datetime(val_t['start_time'])
val_t['end_time'] = pd.to_datetime(val_t['end_time'])
val_t['day_time'] = val_t['start_time'].apply(lambda x: day_time_cls(x))
X_val = val_t.copy()
X_val['end_day_time'] = X_val.end_time.apply(lambda x: day_time_cls(x))    #␣
 ↪Classify the end time
# Extract seasonality of the trip
X_val['year'] = X_val.start_time.apply(lambda x: x.year)
X_val['month'] = X_val.start_time.apply(lambda x: x.month)
X_val['weekday'] = X_val.start_time.apply(lambda x: x.weekday())
X_val.drop(columns=['start_time','end_time'],inplace=True)
```

[132]:
```
X_val['trip_route_category'] = X_val.trip_route_category.astype('category')    #␣
 ↪Set the dtype to category
X_val['trip_route_category'] = X_val.trip_route_category.cat.codes
daytime_enc_dic = {'morning':0, 'afternoon':1, 'evening': 2, 'night':3}
X_val['day_time_ord'] = X_val.day_time.map(daytime_enc_dic)
X_val['end_day_time_ord'] = X_val.end_day_time.map(daytime_enc_dic)
X_val['year'] = X_val.year.apply(lambda x: x-2017)
X_val.drop(columns=['day_time','end_day_time'],inplace=True)

X_val['start_station'] = X_val.start_station.astype('category')
X_val['start_station'] = X_val.start_station.cat.codes
X_val['end_station'] = X_val.end_station.astype('category')
X_val['end_station'] = X_val.end_station.cat.codes
X_val
```

[132]:

	duration	start_lat	start_lon	end_lat	end_lon	trip_route_category	start_station	end_station	year	month	weekday
0	12	34.058319	-118.246094	34.058319	-118.246094	1	20	20	0	1	6
1	17	34.049980	-118.247162	34.043732	-118.260139	0	19	11	0	1	6
2	20	34.063389	-118.236160	34.044159	-118.251579	0	53	42	0	1	6
3	12	34.048851	-118.246422	34.050140	-118.233238	0	21	62	0	1	6
4	48	34.049198	-118.252831	34.049198	-118.252831	1	50	50	0	1	6
...	...	...	...	...	...	...	...	...	...	...	...
569881	19	34.040989	-118.255798	34.041130	-118.267982	0	19	19	0	1	6
569882	8	34.044701	-118.252441	34.051941	-118.243530	0	53	11	0	1	6
569883	43	34.044701	-118.252441	34.044701	-118.252441	0	53	53	0	1	6
569884	42	34.044701	-118.252441	34.044701	-118.252441	0	53	53	0	1	6
569885	1440	34.051941	-118.243530	34.047440	-118.247940	0	11	19	0	1	6

```
...                    ...             ...           ...   ...     ...       ...
569881                   0              25             7     4      12         4
569882                   0              23            22     4      12         4
569883                   1              23            23     4      12         4
569884                   1              23            23     4      12         4
569885                   0              22           287     4      12         4

        day_time_ord  end_day_time_ord
0                  3                 3
1                  3                 3
2                  3                 3
3                  3                 3
4                  3                 3
...              ...               ...
569881             2                 2
569882             2                 2
569883             2                 3
569884             2                 3
569885             2                 1

[569886 rows x 13 columns]
```

[133]:
```
y_predictions = rf_pipe.predict(X_val)
y_predictions
```

[133]:
```
array([5, 5, 5, ..., 5, 5, 3])
```

[134]:
```
results = pd.DataFrame(y_predictions, columns=['passholder_type'])
results = results.join(val_id, how='left')
results.passholder_type = LE.inverse_transform(results.passholder_type)
# Export predictions
results.to_csv(f'test_set-pipe_predictions.csv', index=False)
```