Lobster Land Marketing Analysis

Team Members: Jiaxun Wang, Yantong Li, Qian Liu, Cong Duan

In [1]:
```
%cd /Users/rihiko/Desktop/AD654/Project
```

```
/Users/rihiko/Desktop/AD654/Project
```

In [2]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

%matplotlib inline
```

In [3]:
```
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
```

In [4]:
```
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.graphics.tsaplots import plot_pacf
from matplotlib import pyplot
import statsmodels
import statsmodels.api as sm
from statsmodels.tsa.api import ExponentialSmoothing, SimpleExpSmoothing
from statsmodels.tsa.seasonal import seasonal_decompose
```

In [5]:
```
from sklearn.metrics import mean_squared_error
from math import sqrt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
import random
```

In [6]:
```
from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn.metrics import roc_curve, auc
import matplotlib as mpl
import matplotlib.pyplot as plt
```

In [7]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
import xgboost as xgb
```

In [8]:
```
from scipy import stats
```

Summery Statistics

In [9]:
```
nycconsumers=pd.read_csv("nycconsumers.csv")
```

In [10]:
```
nycconsumers.head()
```

Out[10]:

	householdID	state	county	householdpax	AGI	conusleisure	children	leisureavg
0	1	New York	Nassau	4	179883.69	7115.75	1	9193.40
1	2	New York	Nassau	2	169985.69	4967.27	1	7914.61
2	3	New York	Nassau	3	174330.05	3088.74	2	7885.29
3	4	New York	Nassau	5	192924.29	4841.22	2	5472.83
4	5	New York	Nassau	2	153443.55	5745.79	1	7859.36

In [11]:
```
nycconsumers.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6000 entries, 0 to 5999
Data columns (total 8 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   householdID    6000 non-null   int64  
 1   state          6000 non-null   object 
 2   county         6000 non-null   object 
 3   householdpax   6000 non-null   int64  
 4   AGI            6000 non-null   float64
 5   conusleisure   6000 non-null   float64
 6   children       6000 non-null   int64  
 7   leisureavg     6000 non-null   float64
dtypes: float64(3), int64(3), object(2)
memory usage: 375.1+ KB
```

In [12]:
```
nycconsumers.isnull().sum()
```

Out[12]:
```
householdID     0
state           0
county          0
householdpax    0
AGI             0
conusleisure    0
children        0
leisureavg      0
dtype: int64
```

There is no null value in this dataset.

In [13]:
```
nycconsumers.describe()
```

Out[13]:

	householdID	householdpax	AGI	conusleisure	children	leisureavg
count	6000.000000	6000.000000	6000.000000	6000.000000	6000.000000	6000.000000
mean	3000.500000	3.079333	180436.704478	4815.972537	1.018000	7472.143352
std	1732.195139	1.463351	29905.597508	1386.009162	0.778103	1204.675429
min	1.000000	1.000000	74761.520000	-585.350000	0.000000	2570.000000
25%	1500.750000	2.000000	160028.737500	3890.097500	0.000000	6668.377500
50%	3000.500000	3.000000	180718.865000	4864.675000	1.000000	7452.885000
75%	4500.250000	4.000000	200375.475000	5793.805000	2.000000	8298.950000
max	6000.000000	9.000000	285725.010000	9450.490000	4.000000	11531.300000

In [14]:

```
nycconsumers.groupby('county').describe()
```

Out[14]:

	householdID								householdpax		...	children		leisureavg		
	count	mean	std	min	25%	50%	75%	max	count	mean	...	75%	max	count	mean	std
county																
Bergen	500.0	4250.5	144.481833	4001.0	4125.75	4250.5	4375.25	4500.0	500.0	4.078	...	2.0	3.0	500.0	7496.66408	1206.78
Bronx	500.0	3750.5	144.481833	3501.0	3625.75	3750.5	3875.25	4000.0	500.0	3.116	...	2.0	3.0	500.0	7426.38236	1173.11
Fairfield	500.0	5750.5	144.481833	5501.0	5625.75	5750.5	5875.25	6000.0	500.0	3.070	...	2.0	3.0	500.0	7537.49430	1167.45
Hudson	250.0	4625.5	72.312977	4501.0	4563.25	4625.5	4687.75	4750.0	250.0	3.176	...	2.0	3.0	250.0	7441.16172	1216.07
Kings	500.0	1750.5	144.481833	1501.0	1625.75	1750.5	1875.25	2000.0	500.0	2.984	...	2.0	3.0	500.0	7474.82102	1238.73
Morris	500.0	5250.5	144.481833	5001.0	5125.75	5250.5	5375.25	5500.0	500.0	3.112	...	2.0	3.0	500.0	7447.50262	1208.56
Nassau	500.0	250.5	144.481833	1.0	125.75	250.5	375.25	500.0	500.0	2.968	...	2.0	4.0	500.0	7526.01608	1159.00
New York	500.0	3250.5	144.481833	3001.0	3125.75	3250.5	3375.25	3500.0	500.0	2.316	...	1.0	3.0	500.0	7390.21844	1214.01
Passaic	250.0	4875.5	72.312977	4751.0	4813.25	4875.5	4937.75	5000.0	250.0	3.012	...	1.0	3.0	250.0	7538.15932	1247.29
Queens	500.0	1250.5	144.481833	1001.0	1125.75	1250.5	1375.25	1500.0	500.0	3.136	...	1.0	3.0	500.0	7463.39362	1213.60
Suffolk	500.0	750.5	144.481833	501.0	625.75	750.5	875.25	1000.0	500.0	3.042	...	2.0	4.0	500.0	7505.84534	1222.56
Westchester	1000.0	2500.5	288.819436	2001.0	2250.75	2500.5	2750.25	3000.0	1000.0	3.018	...	2.0	3.0	1000.0	7453.86092	1210.94

12 rows × 48 columns

In [15]:

```
nycconsumers.groupby('children').describe()
```

Out[15]:

	householdID								householdpax		...	conusleisure		leisureavg	
	count	mean	std	min	25%	50%	75%	max	count	mean	...	75%	max	count	mean
children															
0	1631.0	2944.078479	1697.360952	7.0	1493.50	3028.0	4300.00	5999.0	1631.0	1.892091	...	5739.5600	9173.90	1631.0	7480.21
1	2760.0	3030.597826	1730.111785	1.0	1532.50	3021.0	4543.00	6000.0	2760.0	3.232609	...	5810.5000	8810.28	2760.0	7449.14
2	1481.0	3016.802161	1774.873358	3.0	1463.00	2947.0	4563.00	5998.0	1481.0	3.966239	...	5810.0900	9450.49	1481.0	7502.29
3	126.0	2919.817460	1700.550779	48.0	1659.25	2774.5	4417.50	5986.0	126.0	4.634921	...	5782.8625	8769.78	126.0	7516.86
4	2.0	488.500000	406.586399	201.0	344.75	488.5	632.25	776.0	2.0	5.000000	...	4155.0900	4508.33	2.0	7482.65

5 rows × 40 columns

Summery Statistics

There are totally 6000 rows (groups of data) in this dataset.The average estimatimation of the household's Adjusted Gross Income for the most recent calendar year is 180436.70.

The average annual estimated leisure spending by that household for each of the past three years is 7472.14 In all counties mentioned in this dataframe, people in Fairfield, Nassau, Passaic and Suffolk seem to spend more on leisure than people in other county. According to this, we think the manager of the park can use different business strategy. For those counties where people tend to pay a lot on their leisure activities, the park can set up stamp collection activities, for example when people have 10 stamps, they can redeem their prizes. For those counties where people tend to pay little on their leisure activities, the primary task of managers is to find ways to attract them to the park. They can lower the price of tickets, such as selling discounted family packages.

We also group the data according to the number of children. However, we are surperised to find that no matter how many children in a household, the time spent on leisure is not much different. So we think that most of the entertainment facilities in Lobsterland are aimed at all ages, not just for children, and adults like to play here.

In [ ]:

```
```

Segmenting and Targeting

In [16]:
```
newconsumers=nycconsumers.drop(["householdID"],axis=1)
newconsumers1=newconsumers.drop(["state"],axis=1)
newconsumers2=newconsumers1.drop(["county"],axis=1)
```

In [17]:
```
newconsumers2
```

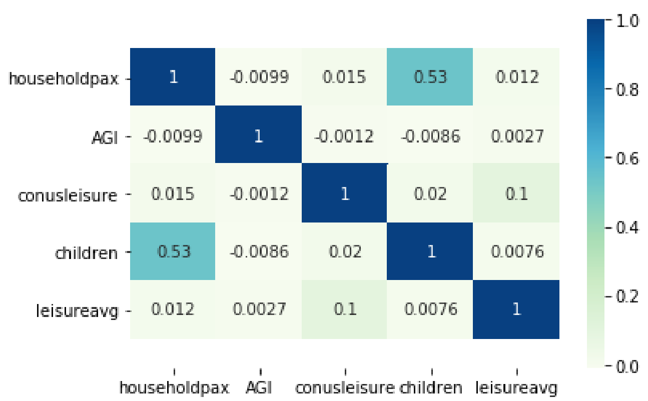
Out[17]:

	householdpax	AGI	conusleisure	children	leisureavg
0	4	179883.69	7115.75	1	9193.40
1	2	169985.69	4967.27	1	7914.61
2	3	174330.05	3088.74	2	7885.29
3	5	192924.29	4841.22	2	5472.83
4	2	153443.55	5745.79	1	7859.36
...	...	...	...	...	...
5995	3	213659.08	4992.11	1	7520.56
5996	5	153097.12	5765.21	1	8001.09
5997	5	217697.66	4476.71	2	8300.09
5998	4	160159.22	5833.82	0	9206.90
5999	2	248009.17	3707.26	1	7803.31

6000 rows × 5 columns

In [18]:
```
df_corr = newconsumers2.corr()
ax = sns.heatmap(df_corr, annot=True, cmap="GnBu")
bottom, top = ax.get_ylim()
ax.set_ylim(bottom + 0.5, top - 0.5)
```

Out[18]: (5.5, -0.5)

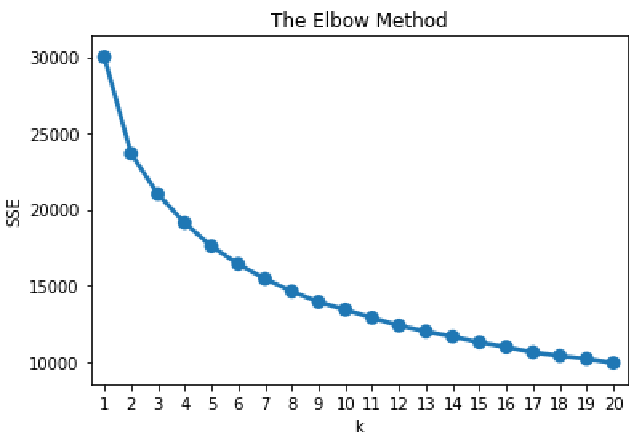


In [19]:
```
scaler = StandardScaler()
scaler.fit(newconsumers2)
newconsumers2_normalized = scaler.transform(newconsumers2)
newconsumers2_normalized=pd.DataFrame(data=newconsumers2_normalized, index=newconsumers2.index, columns=n
ewconsumers2.columns)
print(newconsumers2_normalized.describe().round(2))
```

```
       householdpax      AGI  conusleisure  children  leisureavg
count       6000.00  6000.00       6000.00   6000.00     6000.00
mean          -0.00     0.00          0.00      0.00        0.00
std            1.00     1.00          1.00      1.00        1.00
min           -1.42    -3.53         -3.90     -1.31       -4.07
25%           -0.74    -0.68         -0.67     -1.31       -0.67
50%           -0.05     0.01          0.04     -0.02       -0.02
75%            0.63     0.67          0.71      1.26        0.69
max            4.05     3.52          3.34      3.83        3.37
```

In [20]:
```python
sse = {}
for k in range(1, 21):
# Initialize KMeans with k clusters
    kmeans = KMeans(n_clusters=k, random_state=654)
# Fit KMeans on the normalized dataset
    kmeans.fit(newconsumers2_normalized)
    sse[k] = kmeans.inertia_
# Add the plot title "The Elbow Method"
plt.title('The Elbow Method')
# Add X-axis label "k"
plt.xlabel('k')
# Add Y-axis label "SSE"
plt.ylabel('SSE')
sns.pointplot(x=list(sse.keys()), y=list(sse.values()))
```

Out[20]: <matplotlib.axes._subplots.AxesSubplot at 0x1a27891210>



In [21]:
```python
kmeans = KMeans(n_clusters=4, random_state=1000)
kmeans.fit(newconsumers2_normalized)
cluster_labels = kmeans.labels_
newconsumers2_k4 = newconsumers2.assign(Cluster = cluster_labels)
newconsumers2_k4.groupby(['Cluster']).agg({'householdpax':'mean',
                                    'AGI':'mean',
                                    'conusleisure':'mean',
                                    'children':'mean',
                                    'leisureavg':['mean','count']
}).round(2)
```

Out[21]:

	householdpax	AGI	conusleisure	children	leisureavg	
	mean	mean	mean	mean	mean	count
Cluster						
0	4.04	166354.49	4146.29	1.50	6612.85	1516
1	2.00	162591.85	5547.13	0.47	7764.43	1533
2	4.12	191937.38	5482.04	1.55	8272.74	1579
3	2.01	202699.99	3972.42	0.49	7173.65	1372

Segmentation

Clustering

Cluster 0: High-paid class who like to travel abroad

The AGI in this cluster is really high, so we guess people from this cluster may do some high-paid work. What's more, they spent a lot of money on leisure over the past three years, but there were not too much that took place within the Continental U.S. So we consider they prefer to travel abroad.

Cluster 1: Family with heavy learning tasks

In this cluster, people spend the least amount of time playing, no matter within the Continental U.S or not. We also find the mean of children number in this cluster is 1.49, which is a relatively large number compared to other clusters. So we think children in those household may have heavy study tasks and their parient may have to spend their spare time to give their children training and guidence.

Cluster 2: Couple in the playground

The average number of people living in the household in this cluster is 1.97, which means the main members of these families may be a young couple. And for those young couple, they may like to date in the playground.

Cluster 3: Big family happy hour

The average number of people living in the household in this cluster is 4.13, which is the biggest one among those 4 clusters. Moreover, the average children number in each household is 1.56. We can infer that they are huge families. Besides, the average annual estimated leisure spending by that household for each of the past three years is also the biggest one, which shows that they are highly willing to spend time in the playground with their family.

Targeting

Cluster 0: Family income in this cluster is relatively high and they like to travel abroad, which indicate that they are not stingy about spending money on leisure. I think for these people, the manager can sell some higher-priced packages with high-quality survices. When visitors pay for this ticket, they can skip the line, enjoy some free food and drink, and have priority to use the front seats when watching a show. Based on these high-quality services, people in this cluster may be willing to pay more.

Cluster 1: Most of the households in this group may have children. These children have to go to school so they may have little time to play in the park. What's more, parents of these children need to work and take care of children usually, so they may also not have too much free time. For people in cluster 1, we can set up a holiday discount. For example, family packages are sold during the summer vacation, and the prices of family packages are lower than those of ordinary single tickets. Or we can add some performances belonging to children in the park during holiday, which can also attract them to play.

Cluster 2: We suspect that most people in this cluster may be young couples, so the park can set up some special promotions for couples or small challenges with rewards. For example, when you buy two ice creams at the same time, you can enjoy the second half-price discount, or make a face at the top of the roller coaster at the same time to win free drinks.

Cluster 3: This group may be very entertaining families. They are happy to enjoy family time in the park, so for them we can sell annual or season tickets with some discounts. Or when they come to the park more than 20 times a year, they can enjoy a 20% discount on the annual ticket price for the second year.

The process that you used for arriving at the number of clusters for your model

- Firstly, we should standardize our data and variables. Because we need to adjust the mean of each group of data to zero and the standard deviation to 1. Standardization can eliminate the problem of excessive numerical differences between the various feature variables.
- Then building an elbow chart. The point where the largest elbow is formed in an elbow plot gives a good starting point for determining the optimal number of clusters. For the elbow chart we get, point 4 seems to be the largest elbow which is also the number of clusters.

In [ ]:

Forecasting Total Spending

In [22]:
```
data = pd.read_csv("nyc_to_ne.csv",index_col='year', parse_dates=True)
```

Data Exploration

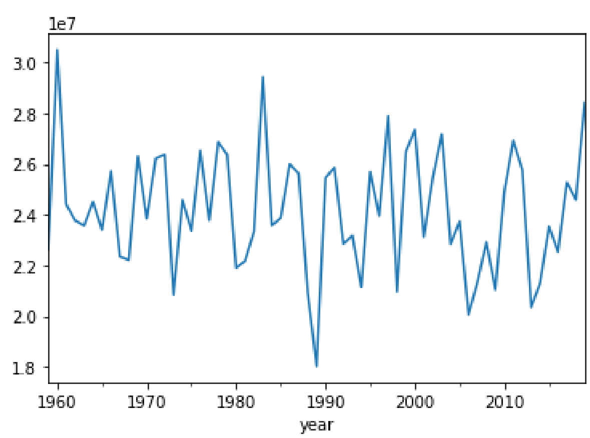
In [23]:
```
data.head()
```
Out[23]:

	summerspend
year	
1959-01-01	2.262733e+07
1960-01-01	3.050980e+07
1961-01-01	2.440828e+07
1962-01-01	2.377221e+07
1963-01-01	2.357277e+07

In [24]:
```
data.index
```
Out[24]:
```
DatetimeIndex(['1959-01-01', '1960-01-01', '1961-01-01', '1962-01-01',
               '1963-01-01', '1964-01-01', '1965-01-01', '1966-01-01',
               '1967-01-01', '1968-01-01', '1969-01-01', '1970-01-01',
               '1971-01-01', '1972-01-01', '1973-01-01', '1974-01-01',
               '1975-01-01', '1976-01-01', '1977-01-01', '1978-01-01',
               '1979-01-01', '1980-01-01', '1981-01-01', '1982-01-01',
               '1983-01-01', '1984-01-01', '1985-01-01', '1986-01-01',
               '1987-01-01', '1988-01-01', '1989-01-01', '1990-01-01',
               '1991-01-01', '1992-01-01', '1993-01-01', '1994-01-01',
               '1995-01-01', '1996-01-01', '1997-01-01', '1998-01-01',
               '1999-01-01', '2000-01-01', '2001-01-01', '2002-01-01',
               '2003-01-01', '2004-01-01', '2005-01-01', '2006-01-01',
               '2007-01-01', '2008-01-01', '2009-01-01', '2010-01-01',
               '2011-01-01', '2012-01-01', '2013-01-01', '2014-01-01',
               '2015-01-01', '2016-01-01', '2017-01-01', '2018-01-01',
               '2019-01-01'],
              dtype='datetime64[ns]', name='year', freq=None)
```

Given 61 years of data(1959-2019) at yearly level with the number of commuters travelling, we need to predict the total spending by greater NYC visitors to New England parks for each of the next five years.

In [25]:
```
data['summerspend'].plot();
```



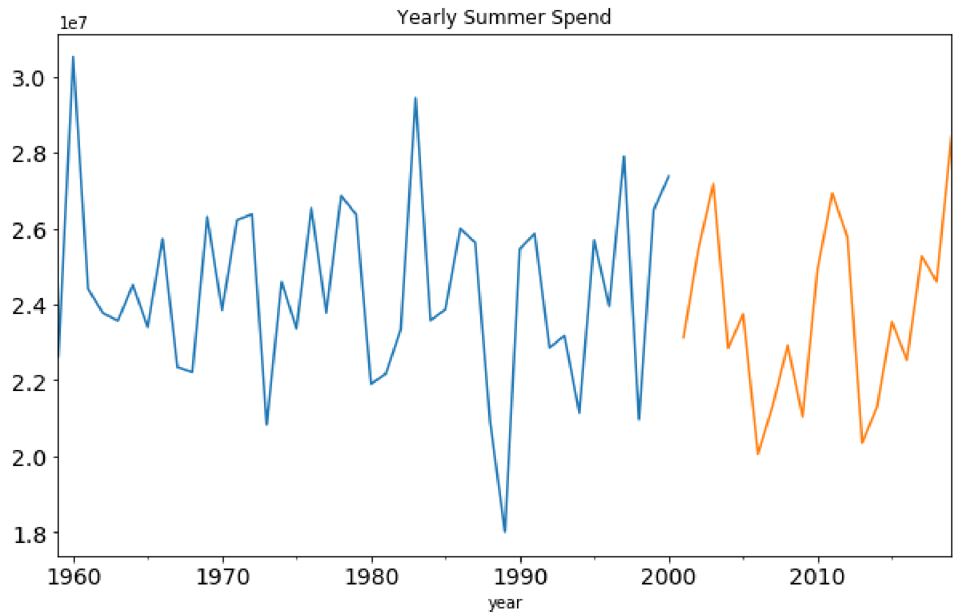
Creating train and test datasets for modeling

Because we need to capture the time factor in time series data, I devided total data as training data and test data by time. The first 70% older data is training data, and 30% newer data is test data.

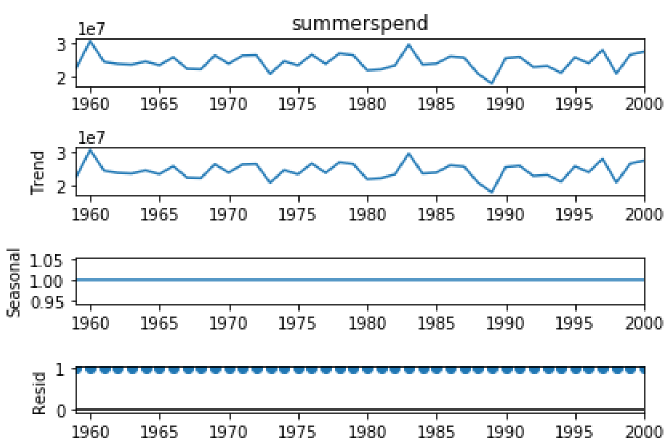
In [26]:
```
#Index 42 marks 2001-01-01
train=data[0:42]
test=data[42:]
```

Let’s visualize the data (train and test together) to know how it varies over a time period.

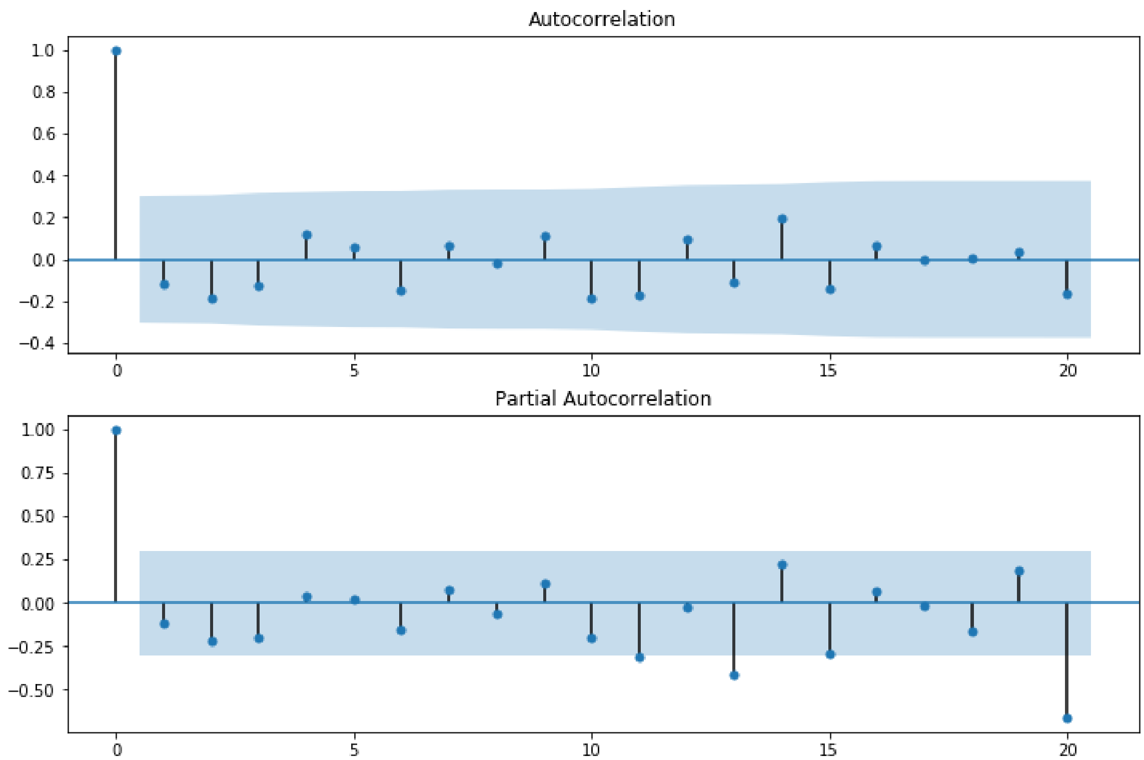
In [27]:
```
#Plotting data
train.summerspend.plot(figsize=(10,6), title= 'Yearly Summer Spend', fontsize=14)
test.summerspend.plot(figsize=(10,6), title= 'Yearly Summer Spend', fontsize=14)
plt.show()
```



In [28]:
```
result = seasonal_decompose(train['summerspend'], model='multiplicative')  # model='mul' also works
result.plot();
```



In [29]:
```
fig = plt.figure(figsize=(12,8))
ax1 = fig.add_subplot(211)
fig = sm.graphics.tsa.plot_acf(train,lags=20,ax=ax1)
ax2 = fig.add_subplot(212)
fig = sm.graphics.tsa.plot_pacf(train,lags=20,ax=ax2)
```



- According to the ETS decomposition, there is no significant seasonal and trend. Therefore, we will not consider the seasonal model, such as SARIMAX, Holt-Winters, Holt's Linear Trend.
- What's more, this time series seems to be stationary. Both ACF and PACF fall into confidence interval abruptly, cutting off at q = 0 and p = 0,respectively. But for more precisely prediction, we will try AR(1), MA(1), ARMA(1,1) in the following modeling process.

Modeling

We use training dataset for modeling, and test dataset to measure the performance of models. The performance indicator mainly is RMSE. But we also use AIC and BIC to measure the performances of AR(1), MA(1), ARMA(1,1) , to select the best model in ARMA.

In [30]:
```
RMSE = [] # collect rmse of all the models
```

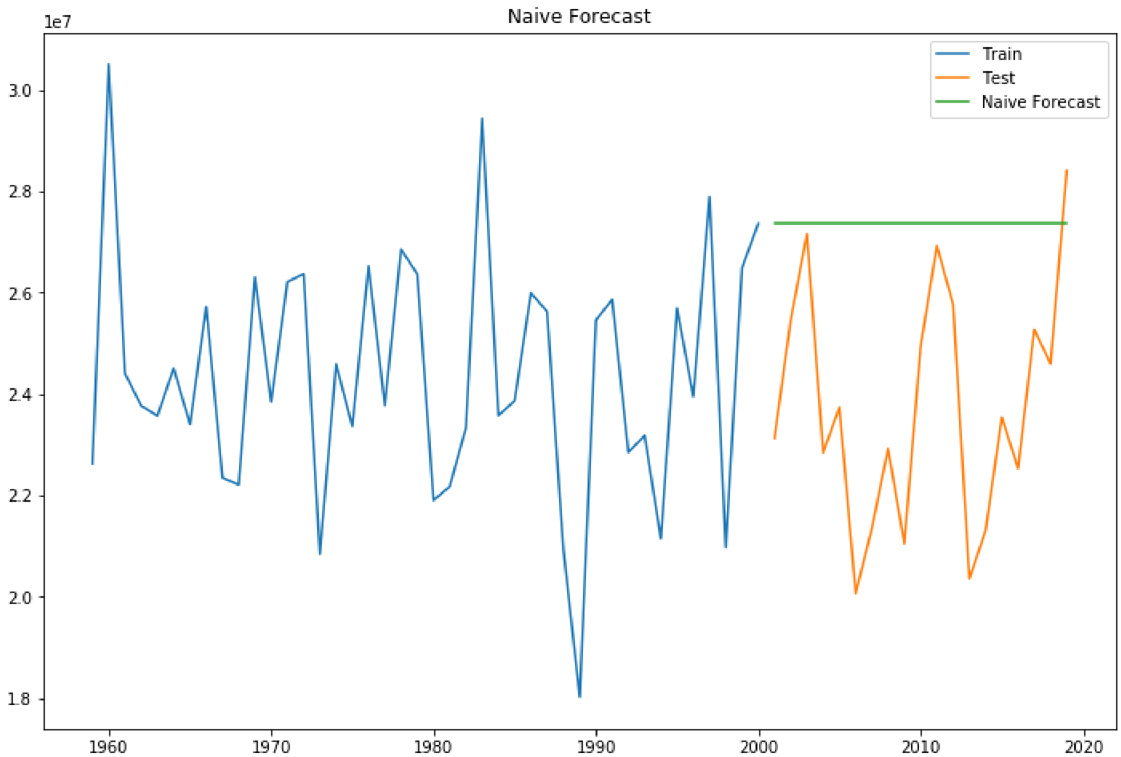
Method 1: Naive Approach

In [31]:
```
# Consider the process is no trend and no seasonal factor, we apply the Naive Approach at first
```

$$\hat{y}_{t+1} = y_t$$

In [ ]:
```

```

In [32]:
```
# Now we will implement the Naive method to forecast the prices for test data.

train_summerspend = np.asarray(train.summerspend)
y_hat = test.copy()
y_hat['naive'] = train_summerspend[len(train_summerspend)-1]
plt.figure(figsize=(12,8))
plt.plot(train.index, train['summerspend'], label='Train')
plt.plot(test.index,test['summerspend'], label='Test')
plt.plot(y_hat.index,y_hat['naive'], label='Naive Forecast')
plt.legend(loc='best')
plt.title("Naive Forecast")
plt.show()
```



In [33]:
```
# We will now calculate RMSE to check to accuracy of our model on test data set.


rms = sqrt(mean_squared_error(test.summerspend, y_hat.naive))
RMSE.append(rms)
print(rms)
# RMSE = 4292286.145806624
```

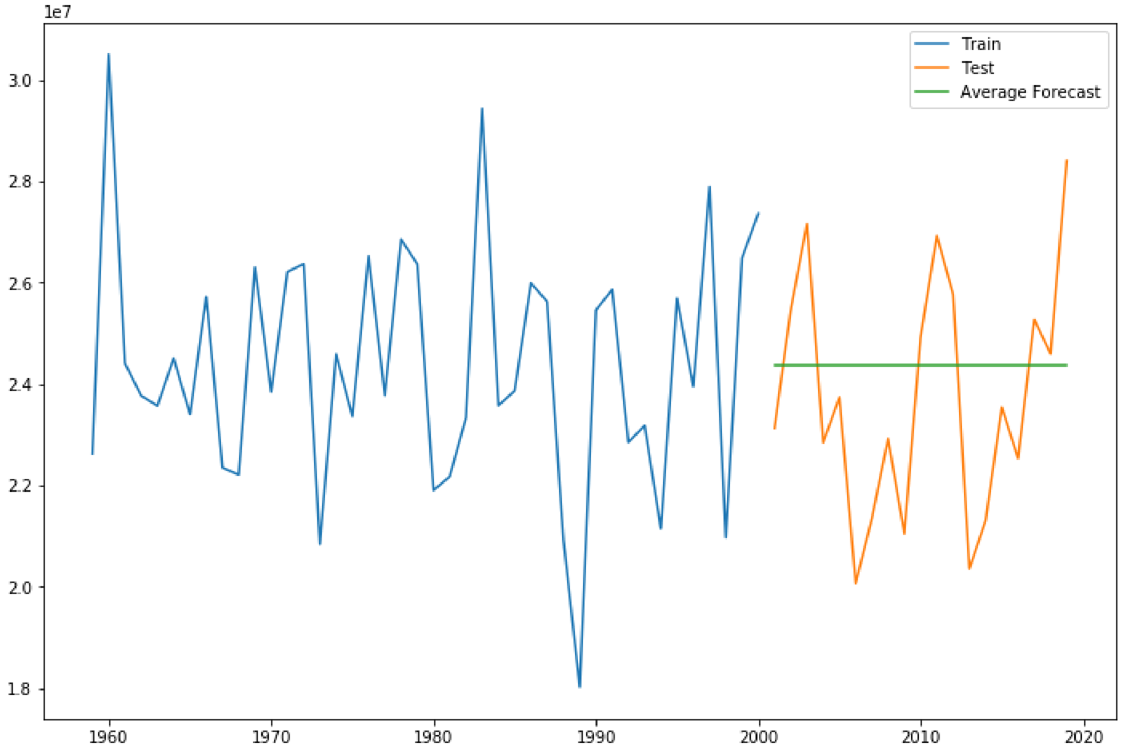
```
4292286.145806624
```

We can infer from the RMSE value and the graph above, that Naive method isn't suited for datasets with high variability. It is best suited for stable datasets. We can still improve our score by adopting different techniques.

Method 2: Simple Average

$$\hat{y}_{t+1} = \frac{1}{x}\sum_{i=1}^{x} y_i$$

In [34]:
```
y_hat_avg = test.copy()
y_hat_avg['avg_forecast'] = train['summerspend'].mean()
plt.figure(figsize=(12,8))
plt.plot(train['summerspend'], label='Train')
plt.plot(test['summerspend'], label='Test')
plt.plot(y_hat_avg['avg_forecast'], label='Average Forecast')
plt.legend(loc='best')
plt.show()
```



In [35]:
```
rms = sqrt(mean_squared_error(test.summerspend, y_hat_avg.avg_forecast))
RMSE.append(rms)
print(rms)

# RMSE = 2404469.0779477805
```

```
2404469.0779477805
```

We can see the simple average can improve the score. The reason might be the time series is no trend.

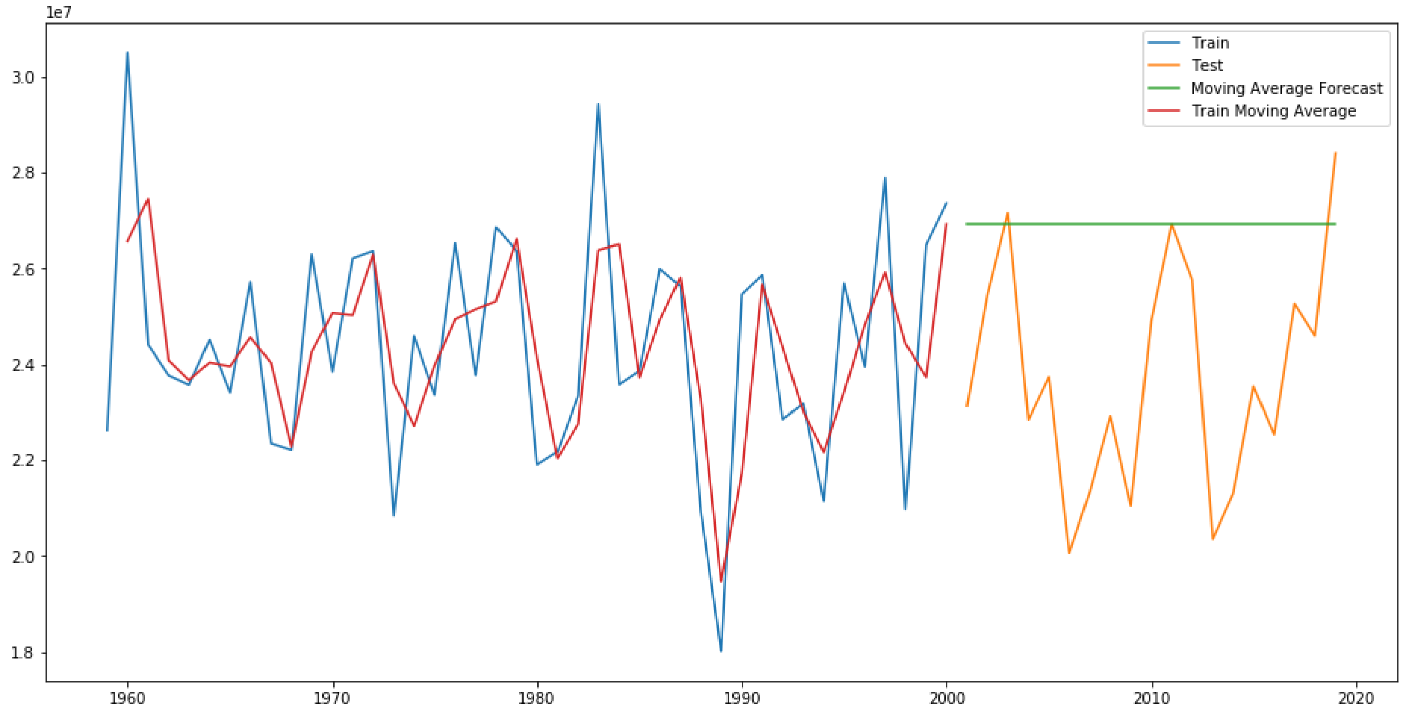
Method 3 Moving Average

$$\hat{y}_i = \frac{1}{p} \sum_{t=i-p}^{i-1} y_t$$

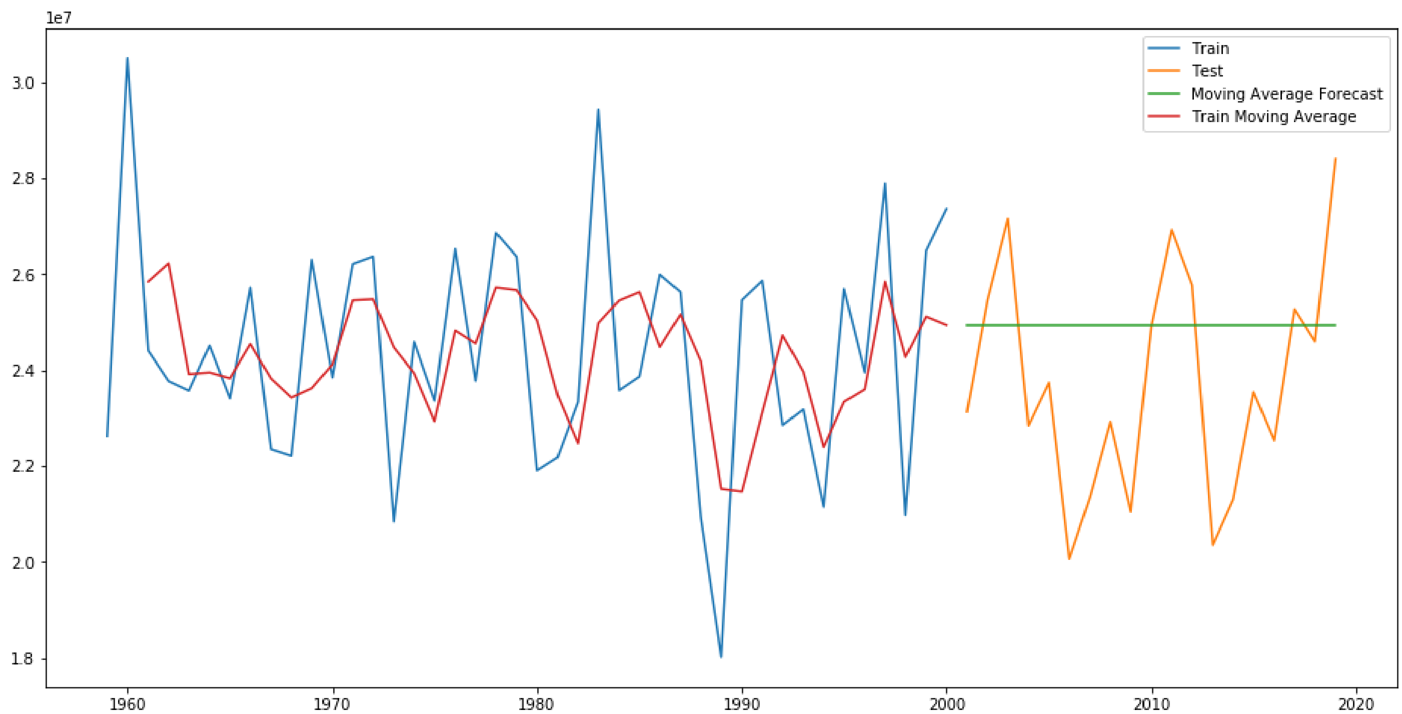
where p is the timewindow

In [36]:

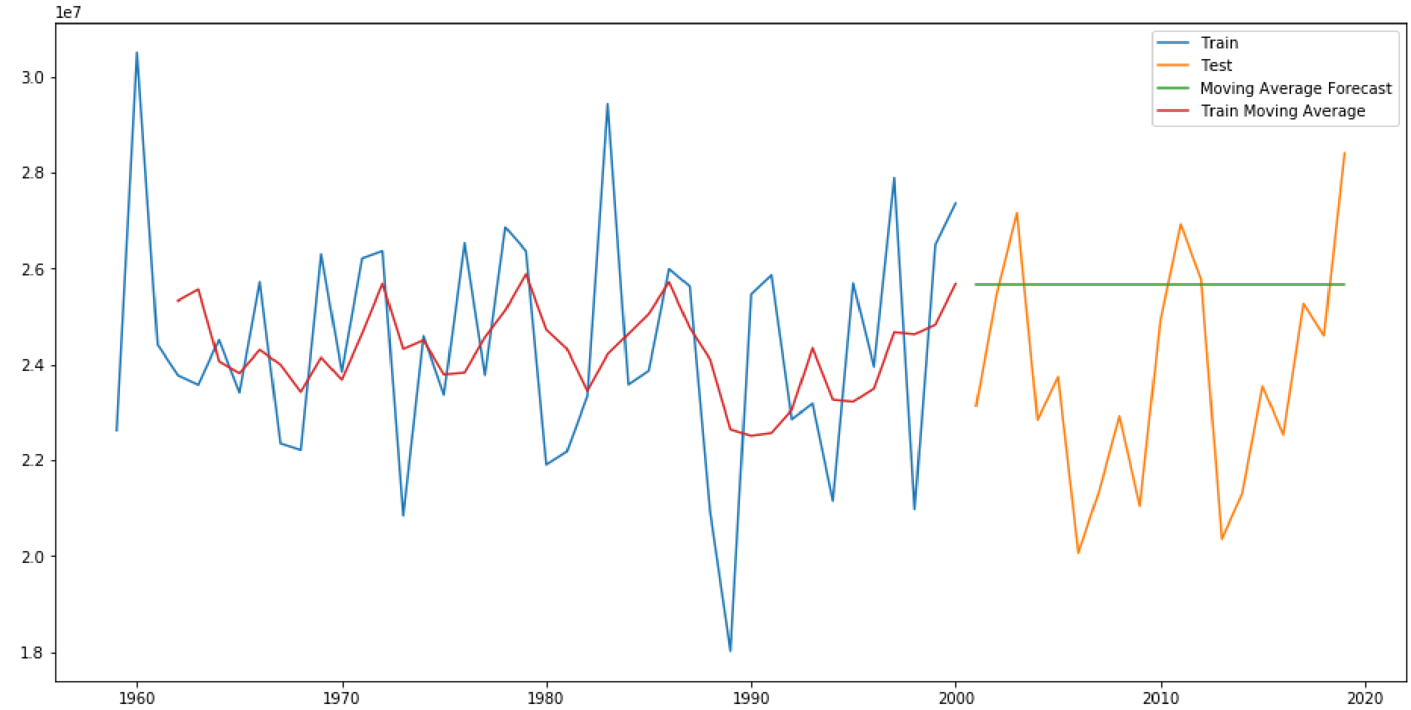
```
p_number = []
rmse_value = []
for i in range(2,11):
        y_hat_avg = test.copy()
        y_hat_avg['moving_avg_forecast'] = train['summerspend'].rolling(i).mean().iloc[-1]
        plt.figure(figsize=(16,8))
        plt.plot(train['summerspend'], label='Train')
        plt.plot(test['summerspend'], label='Test')
        plt.plot(y_hat_avg['moving_avg_forecast'], label='Moving Average Forecast')
        plt.plot(train['summerspend'].rolling(i).mean(), label='Train Moving Average')
        plt.legend(loc='best')
        plt.show()
        rms = sqrt(mean_squared_error(test.summerspend, y_hat_avg.moving_avg_forecast))
        p_number.append(i)
        rmse_value.append(rms)
        print("RMSe of Moving Average Model when p =",i,"is",rms)
```



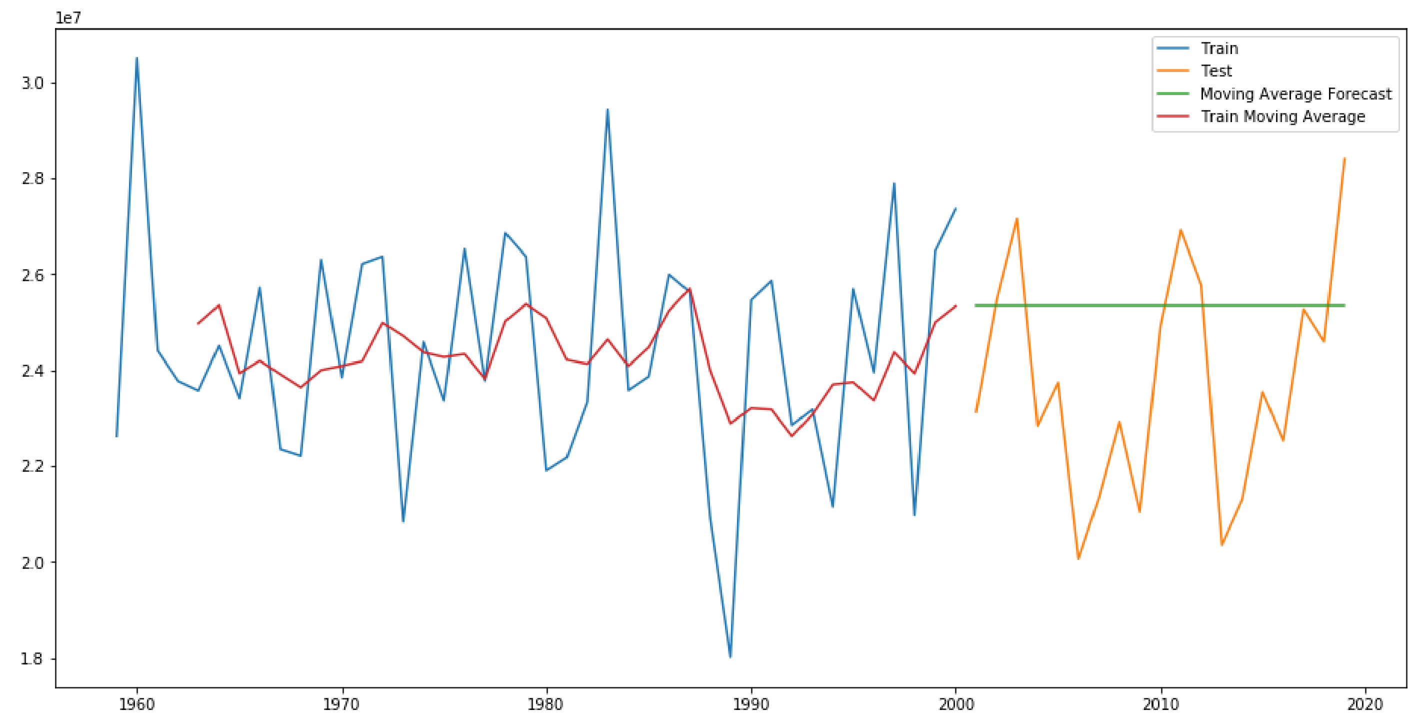
RMSe of Moving Average Model when p = 2 is 3931155.951810835



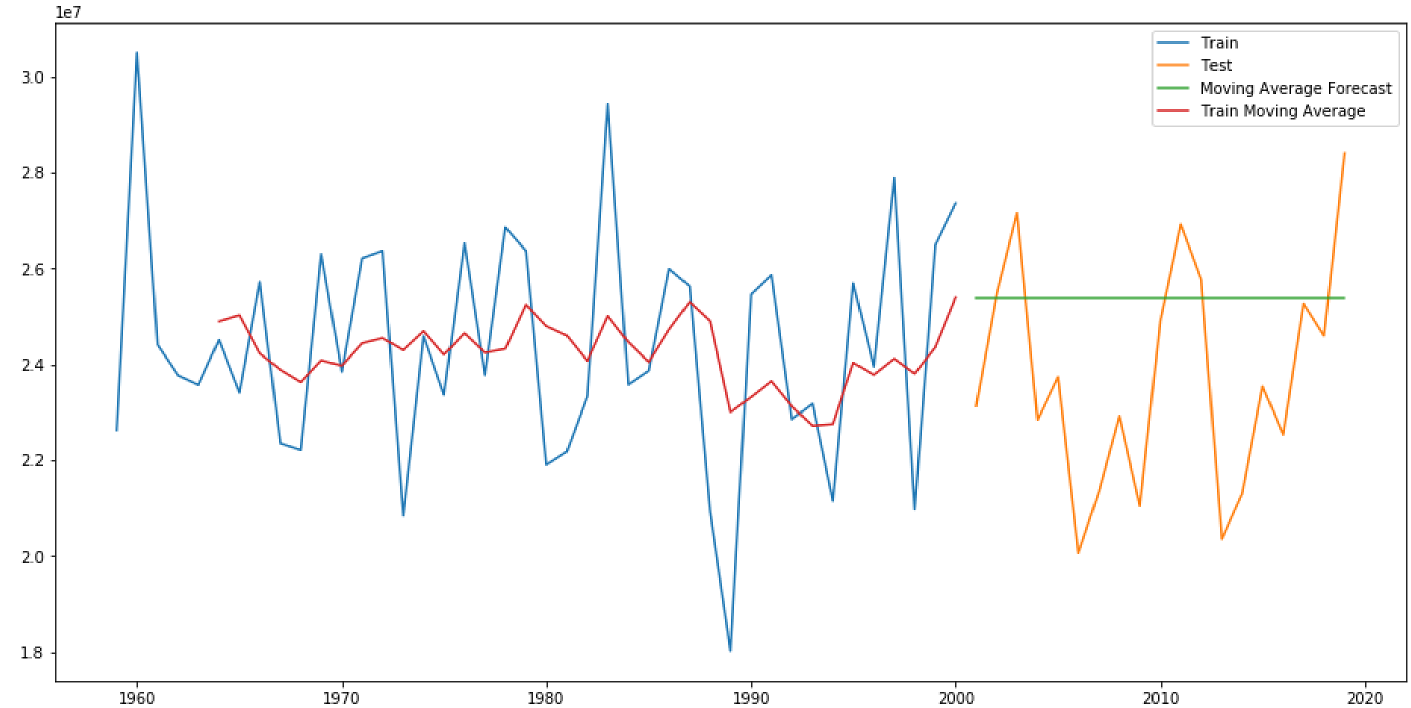
RMSe of Moving Average Model when p = 3 is 2608152.4815611592



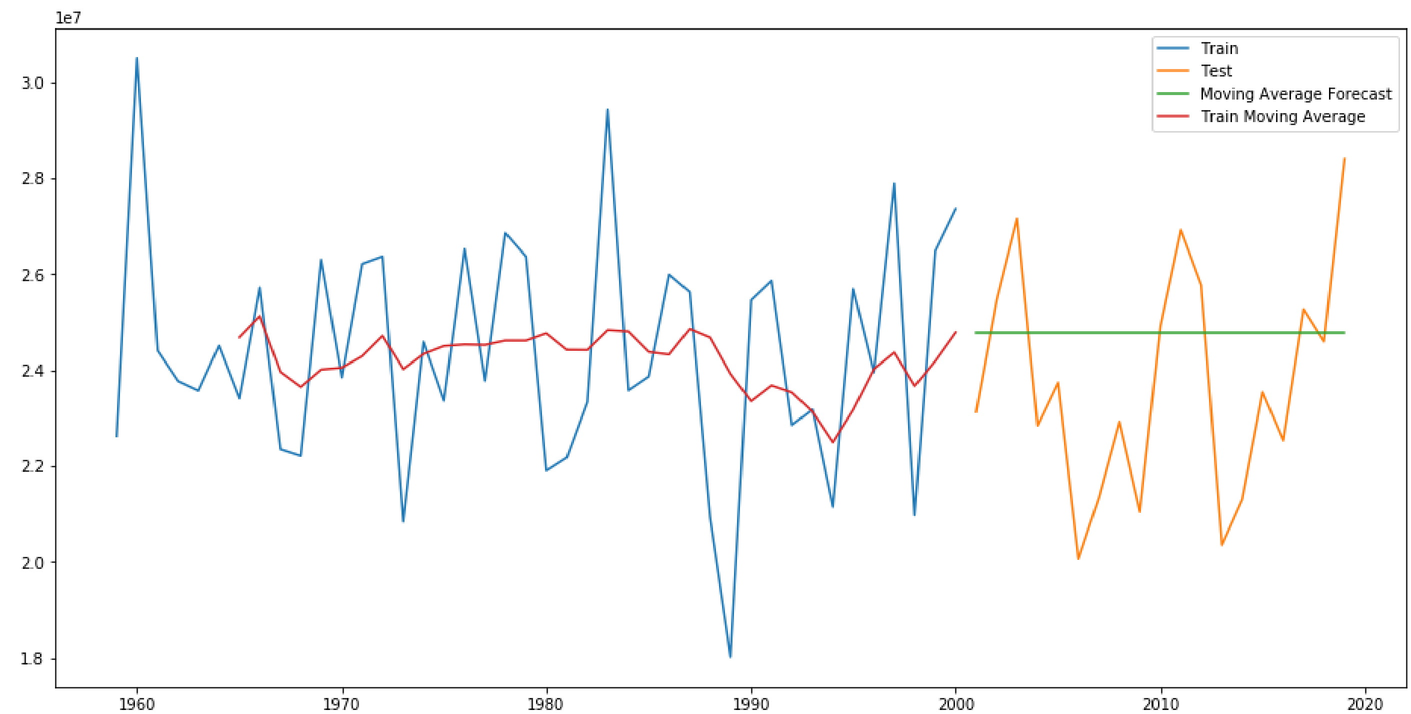

```
RMSe of Moving Average Model when p = 4 is 3016208.3383096172
```



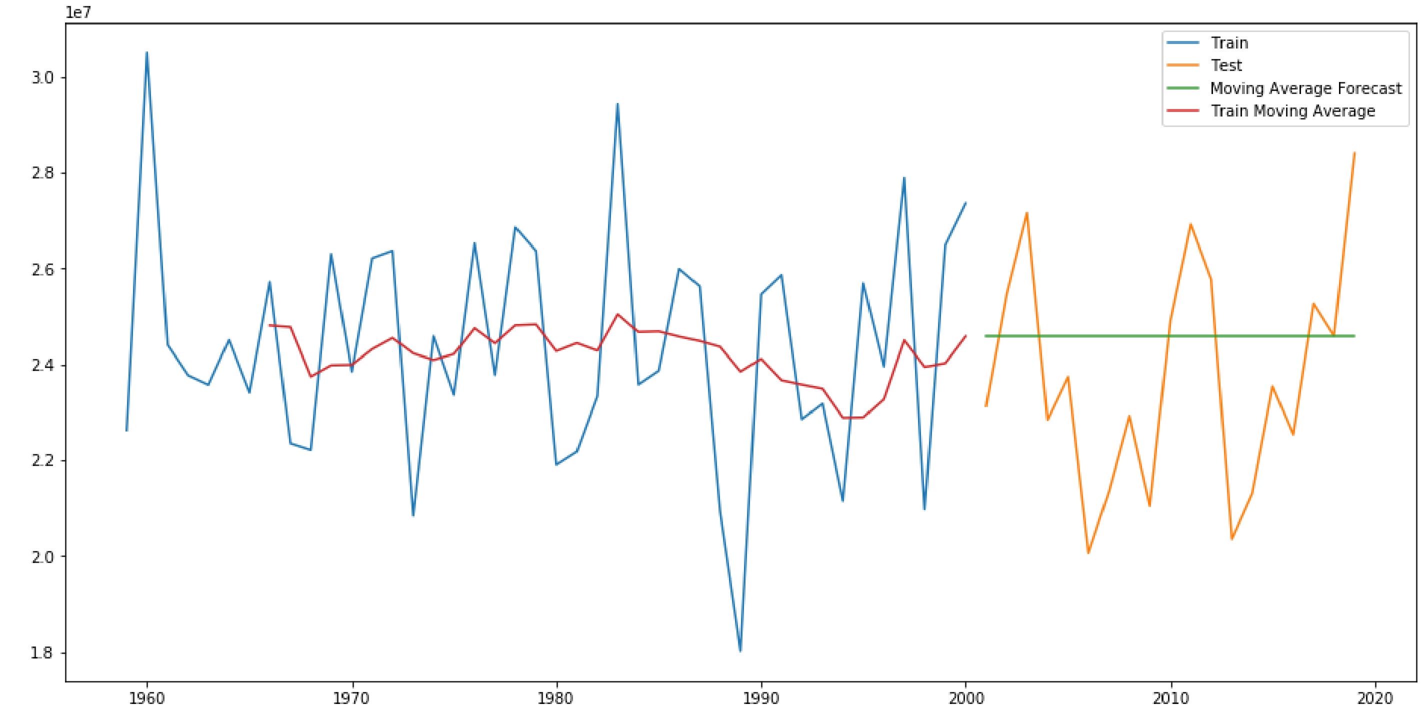

```
RMSe of Moving Average Model when p = 5 is 2808586.1769593935
```



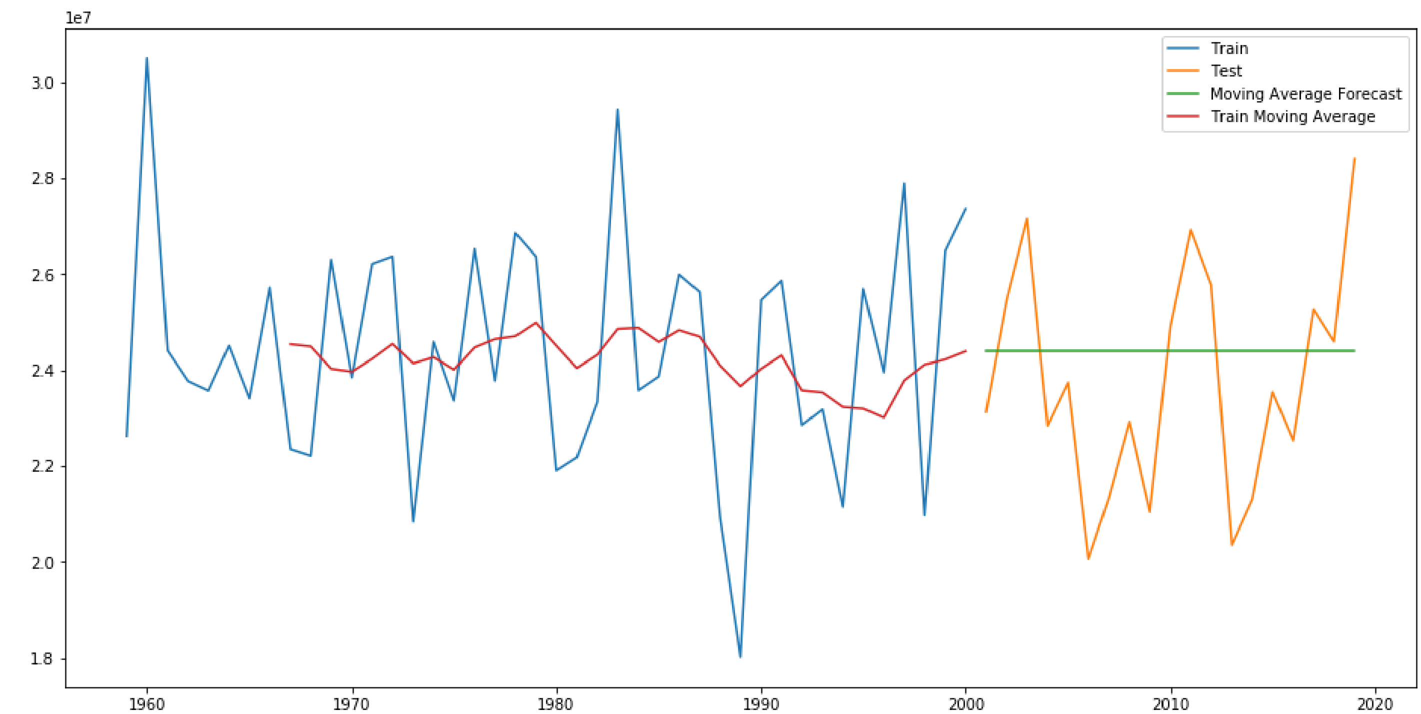
RMSe of Moving Average Model when p = 6 is 2842544.9815152325



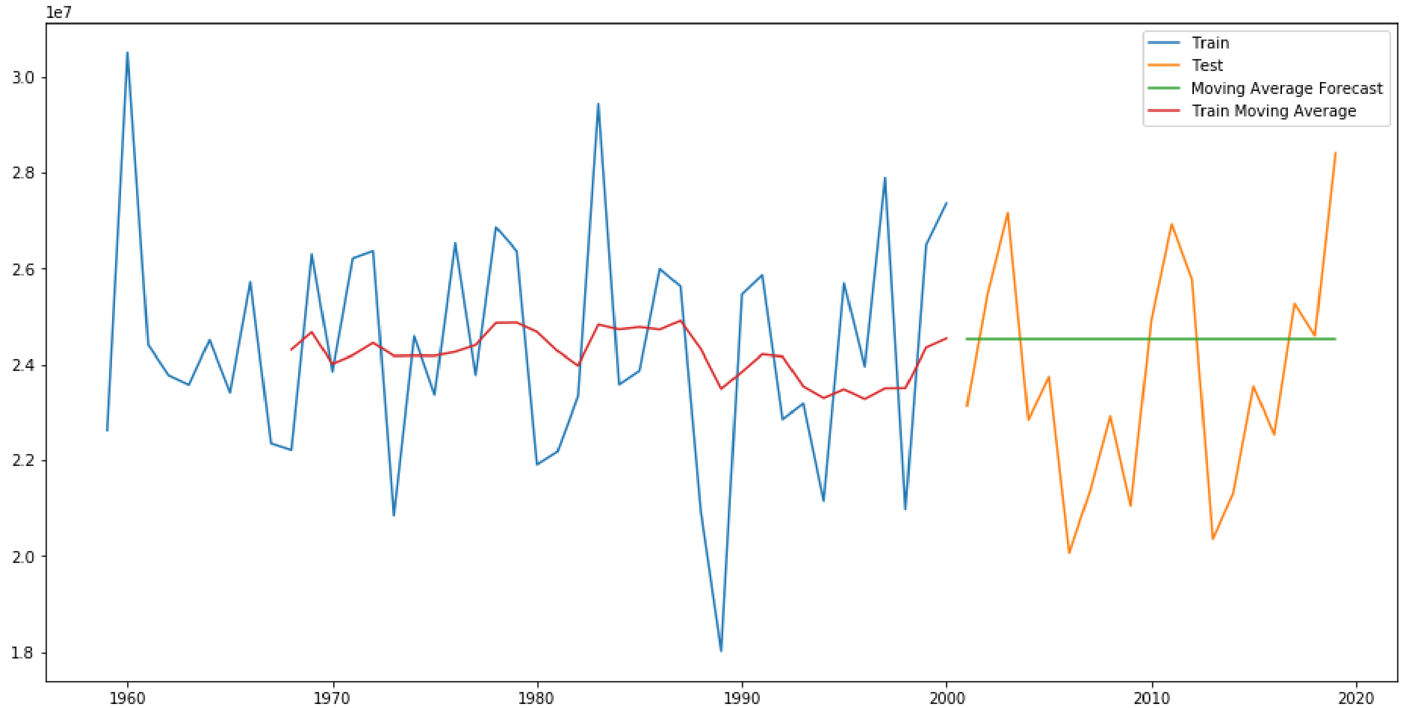
RMSe of Moving Average Model when p = 7 is 2541544.0900346516




```
RMSe of Moving Average Model when p = 8 is 2466775.2099965443
```




```
RMSe of Moving Average Model when p = 9 is 2408699.0105826296
```




```
RMSe of Moving Average Model when p = 10 is 2451783.8706656103
```

In [37]:
```
RMSE_MA = pd.DataFrame({"p_number" : p_number,"rmse_value" : rmse_value})
RMSE_MA
```

Out[37]:

	p_number	rmse_value
0	2	3.931156e+06
1	3	2.608152e+06
2	4	3.016208e+06
3	5	2.808586e+06
4	6	2.842545e+06
5	7	2.541544e+06
6	8	2.466775e+06
7	9	2.408699e+06
8	10	2.451784e+06

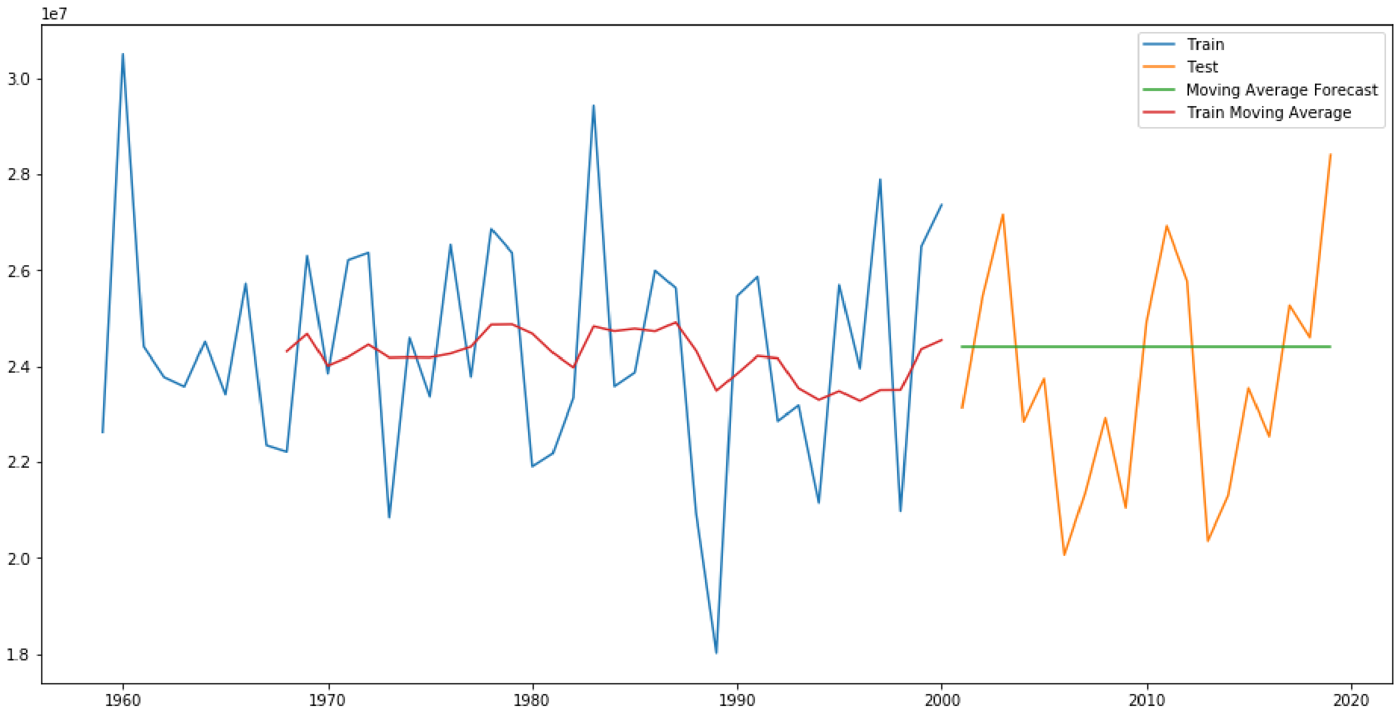
In [38]:
```
RMSE_MA[RMSE_MA.rmse_value == RMSE_MA.rmse_value.min()]
```

Out[38]:

	p_number	rmse_value
7	9	2.408699e+06

In Moving Average model, we use p = 9.

In [39]:
```
y_hat_avg = test.copy()
y_hat_avg['moving_avg_forecast'] = train['summerspend'].rolling(9).mean().iloc[-1]
plt.figure(figsize=(16,8))
plt.plot(train['summerspend'], label='Train')
plt.plot(test['summerspend'], label='Test')
plt.plot(y_hat_avg['moving_avg_forecast'], label='Moving Average Forecast')
plt.plot(train['summerspend'].rolling(i).mean(), label='Train Moving Average')
plt.legend(loc='best')
plt.show()
```



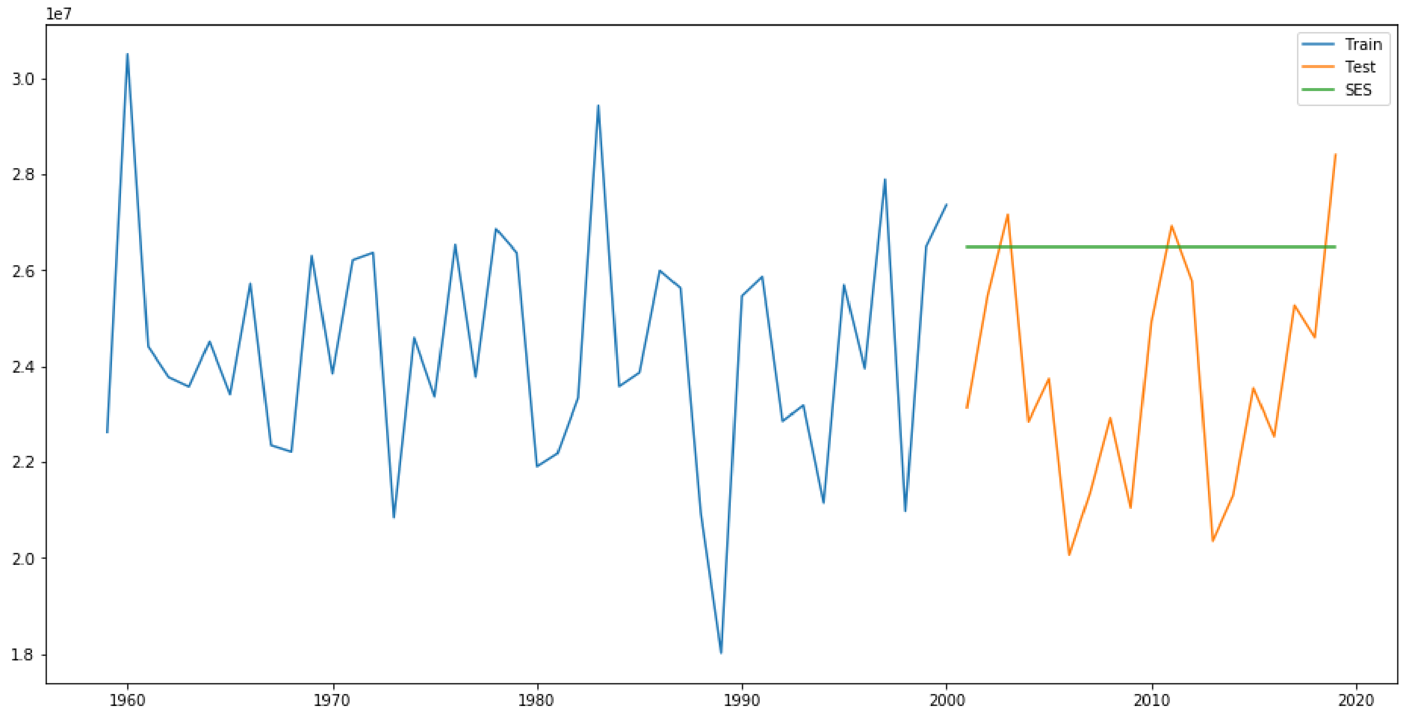
In [40]:
```
rms = sqrt(mean_squared_error(test.summerspend, y_hat_avg.moving_avg_forecast))
RMSE.append(rms)
rms
```

Out[40]: 2408699.0105826296

Method 4 Simple Exponential Smoothing

$$\hat{y}_{t+1|t} = \alpha y_t + (1-\alpha)\hat{y}_{t|t-1}$$

In [41]:
```
y_hat_avg = test.copy()
fit2 = SimpleExpSmoothing(np.asarray(train['summerspend'])).fit(smoothing_level=0.6,optimized=False)
y_hat_avg['SES'] = fit2.forecast(len(test))
plt.figure(figsize=(16,8))
plt.plot(train['summerspend'], label='Train')
plt.plot(test['summerspend'], label='Test')
plt.plot(y_hat_avg['SES'], label='SES')
plt.legend(loc='best')
plt.show()
```



In [42]:
```
rms = sqrt(mean_squared_error(test.summerspend, y_hat_avg.SES))
RMSE.append(rms)
print(rms)

#RMSE = 3577811.749127613
```

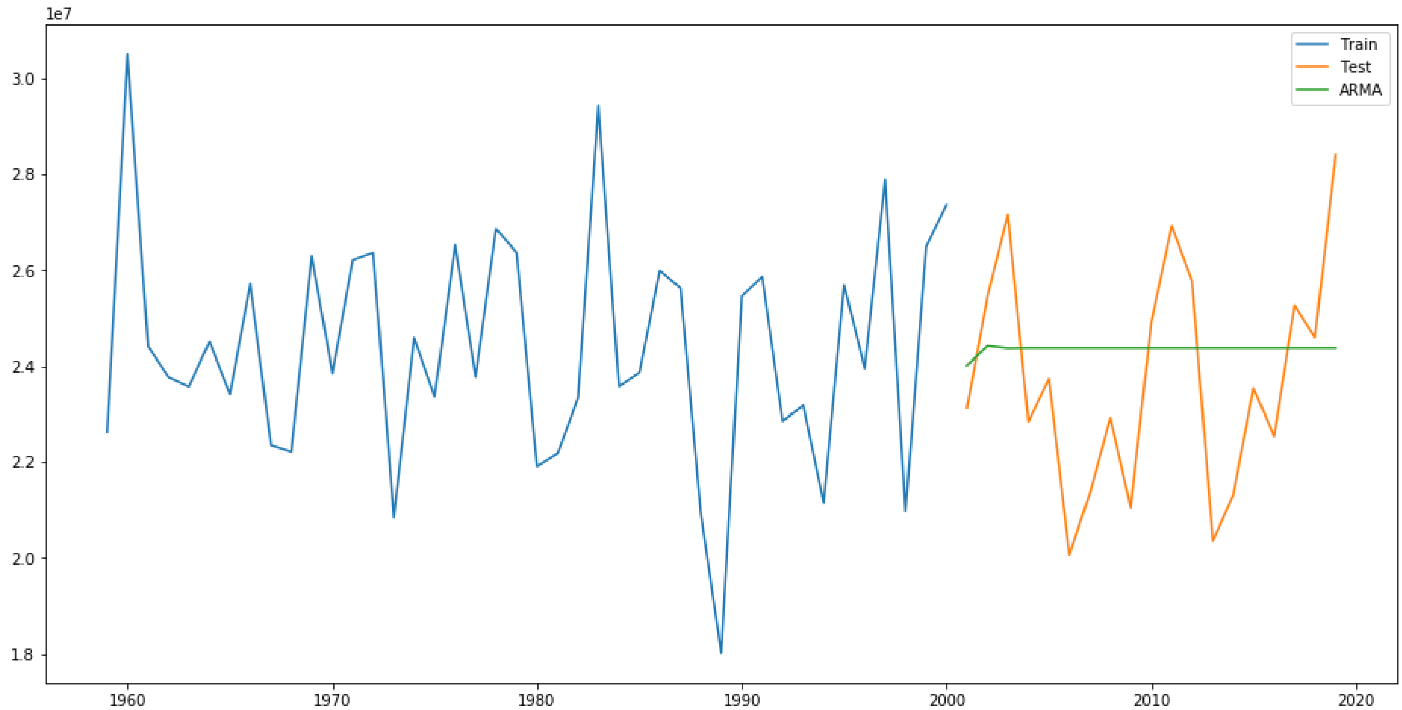
```
3577811.749127613
```

Method 5 ARMA

In [43]:
```
ARMA_name = ["AR(1)","ARMA(1,1)"]
AIC = []
BIC = []
```

In [44]:
```
#AR(1)
y_hat_avg = test.copy()
fit1 = sm.tsa.ARMA(train.summerspend, order=(1,0)).fit()
y_hat_avg['ARMA'] = fit1.predict(start="2001-1-1", end="2019-1-1", dynamic=True)
plt.figure(figsize=(16,8))
plt.plot( train['summerspend'], label='Train')
plt.plot(test['summerspend'], label='Test')
plt.plot(y_hat_avg['ARMA'], label='ARMA')
plt.legend(loc='best')
plt.show()
```

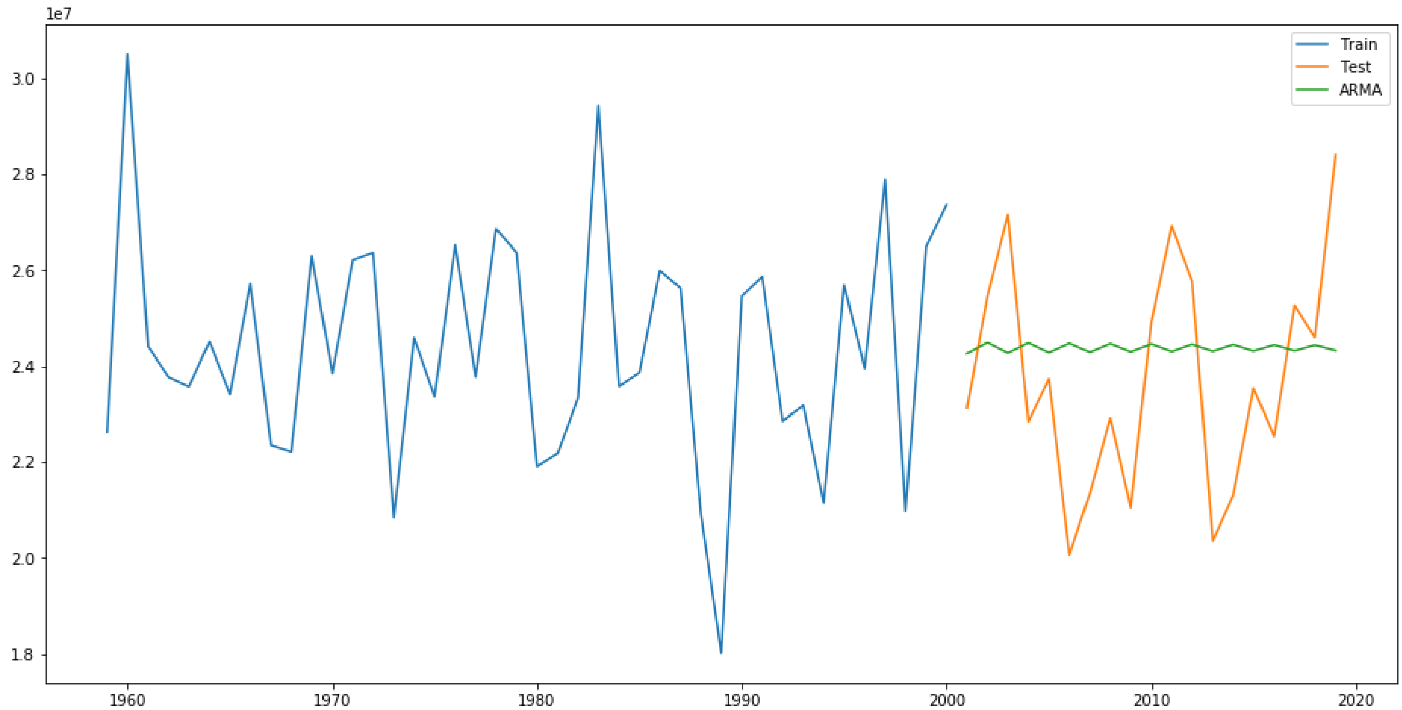
```
/Users/rihiko/opt/anaconda3/lib/python3.7/site-packages/statsmodels/tsa/base/tsa_model.py:162: ValueWarn
ing: No frequency information was provided, so inferred frequency AS-JAN will be used.
  % freq, ValueWarning)
```



In [45]:
```
AIC.append(fit1.aic)
BIC.append(fit1.bic)
```

In [46]:
```
#ARMA(1,1)
y_hat_avg = test.copy()
fit3 = sm.tsa.ARMA(train.summerspend, order=(1,1)).fit()
y_hat_avg['ARMA'] = fit3.predict(start="2001-1-1", end="2019-1-1", dynamic=True)
plt.figure(figsize=(16,8))
plt.plot( train['summerspend'], label='Train')
plt.plot(test['summerspend'], label='Test')
plt.plot(y_hat_avg['ARMA'], label='ARMA')
plt.legend(loc='best')
plt.show()
```

```
/Users/rihiko/opt/anaconda3/lib/python3.7/site-packages/statsmodels/tsa/base/tsa_model.py:162: ValueWarn
ing: No frequency information was provided, so inferred frequency AS-JAN will be used.
  % freq, ValueWarning)
```



In [47]:
```
AIC.append(fit3.aic)
BIC.append(fit3.bic)
```

In [48]:
```
ARMA_performance = pd.DataFrame({"ARMA_name" : ARMA_name,"AIC":AIC,"BIC":BIC})
```

In [49]:
```
ARMA_performance
```

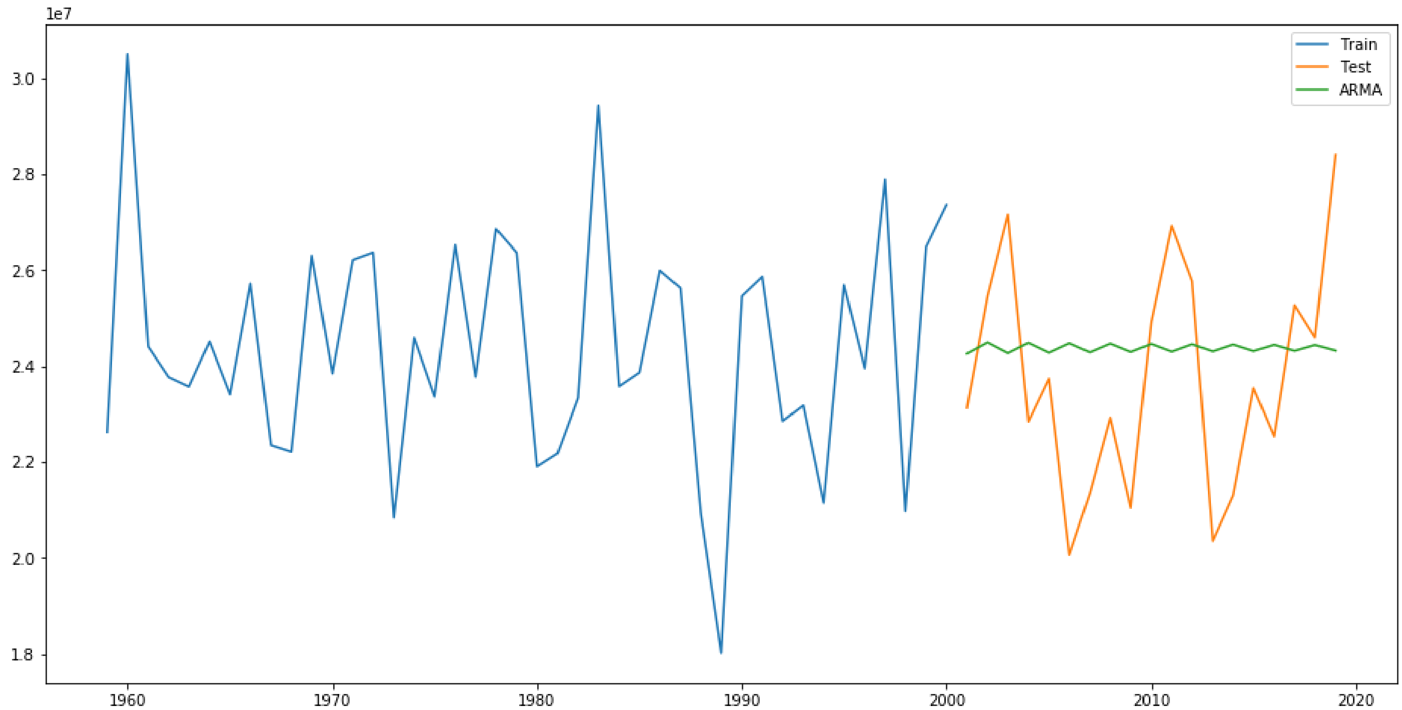
Out[49]:

	ARMA_name	AIC	BIC
0	AR(1)	1359.187356	1364.400365
1	ARMA(1,1)	1361.446995	1368.397673

We select ARMA(1,1) with the largest AIC value and BIC value

In [50]:
```
#ARMA(1,1)
y_hat_avg = test.copy()
fit3 = sm.tsa.ARMA(train.summerspend, order=(1,1)).fit()
y_hat_avg['ARMA'] = fit3.predict(start="2001-1-1", end="2019-1-1", dynamic=True)
plt.figure(figsize=(16,8))
plt.plot( train['summerspend'], label='Train')
plt.plot(test['summerspend'], label='Test')
plt.plot(y_hat_avg['ARMA'], label='ARMA')
plt.legend(loc='best')
plt.show()
```

```
/Users/rihiko/opt/anaconda3/lib/python3.7/site-packages/statsmodels/tsa/base/tsa_model.py:162: ValueWarn
ing: No frequency information was provided, so inferred frequency AS-JAN will be used.
  % freq, ValueWarning)
```



In [51]:
```
rms = sqrt(mean_squared_error(test.summerspend, y_hat_avg.ARMA))
RMSE.append(rms)
print(rms)
#RMSE = 2415096.1564545105
```

```
2415096.1564545105
```

In [52]:
```
Model = ["Naive Model","Simple Average","Moving Average","Simple Exponential Smoothing","ARMA(1,1)"]
```

In [53]:
```
TS_performance = pd.DataFrame({"Model":Model,"RMSE":RMSE[:5]})
TS_performance
TS_performance.RMSE[1]
TS_performance[TS_performance.RMSE == TS_performance.RMSE.min()]
```

Out[53]:

	Model	RMSE
1	Simple Average	2.404469e+06

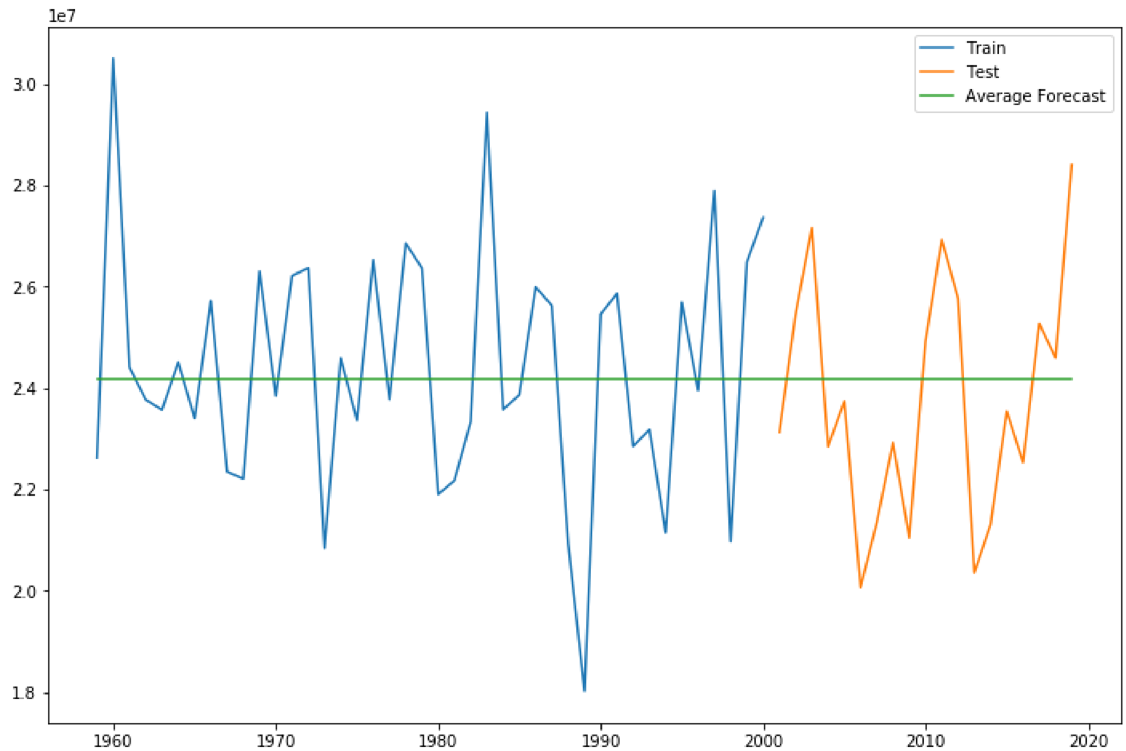
To minimize RMSE, I select Simple Average model. For more precisely prediction, I use all the dataset to build the model.

In [54]:
```
# Build Simple Average Model with all data


pred_dates = pd.date_range('2020-01-01', periods=5, freq='AS')
pred = pd.Series(data['summerspend'].mean(),index=pred_dates)
pred = pd.DataFrame(pred)
pred.columns = ["Forecast"]
pred
```

Out[54]:

	Forecast
2020-01-01	2.418581e+07
2021-01-01	2.418581e+07
2022-01-01	2.418581e+07
2023-01-01	2.418581e+07
2024-01-01	2.418581e+07

In [55]:
```
y_hat_avg = data.copy()
y_hat_avg['avg_forecast'] = data['summerspend'].mean()
plt.figure(figsize=(12,8))
plt.plot(train['summerspend'], label='Train')
plt.plot(test['summerspend'], label='Test')
plt.plot(y_hat_avg['avg_forecast'], label='Average Forecast')
plt.legend(loc='best')
plt.show()
```



According to Moving Average model, we predict the total spending in summer by greater NYC visitors to New England parks for each of the next five years are 24185805.88 dollars.

Forecasting

The time series is the total spending by greater NYC visitors to New England parks during 1959 to 2019. Our goal is to build a predictive model to predict the future spending for the next five years. There are mainly three steps: data exploration, building models, and selecting the best model for predicting.

Process

1. Data Spliting Trick

At first, we split data into training dataset and test dataset. Because of the time factor, we divided the total data as training data and test data by time. The first 70% older data is training data, and 30% newer data is test data.

1. Time Series Decomposition

Then we decompose the time series into trend, seasonal variation, and notice that there is no seasonality and no trend. Time series is fluctuating around the mean value.

1. ACF & PACF

To check the stationary property, we plotted ACF and PACF. Both fall into a confidence interval when lag = 0.

1. Five types of models

Considering these properties, we decided to model Naive model, Simple Average model, Moving Average model, Simple Exponential Smoothing model and ARMA model. In the Moving Average model, we try different window sizes from 1 to 10, and select the moving average model with the minimized RMSE, whose window size is 9. For the ARMA model, we only try AR(1) and ARMA(1,1) due to ACF and PACF plots cutting off after lag = 1, selecting the one with the largest AIC and BIC.

Result

Comparing all the models, the simple average model minimized RMSE. For more precise prediction, we used all the dataset to build a new simple average model. This model forecasts the expected value equal to the average of all previously observed points. The predictions for five years are the same value, which is 24185805.88 dollars. The constant level is due to its algorithm. Unlike the ARMA model, it won't capture the autoregressive factor. It takes the average of all the values previously known as the next value. Of course it won't be exact, but somewhat close.

The reason why it is the best one would be its stationary property. This data is no upward or downward trend, no cycle fluctuation and no seasonality. Just like white noise, it is hard to predict the exact value. Sometimes, the simple one is the best one.

In [ ]:

In [ ]:

Classification

In [56]:
```
nyc_historical =  pd.read_csv("nyc_historical.csv")
```

In [57]:
```
nyc_historical.head()
```

Out[57]:

	householdID	visits	avgrides_perperson	avgmerch_perperson	avggoldzone_perperson	avgfood_perperson	goldzone_playersclub	own_car
0	44	20	9.8	32.4	27.2	70.7	0	1
1	57	20	11.7	71.8	40.8	1.6	0	1
2	63	20	9.8	27.4	25.7	74.9	0	1
3	159	17	2.2	1.5	91.1	28.9	1	1
4	162	19	3.4	5.0	12.0	9.2	0	1

Data Exploration and Preparation

In [58]:
```
nyc_historical.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3200 entries, 0 to 3199
Data columns (total 11 columns):
 #   Column                  Non-Null Count  Dtype  
---  ------                  --------------  -----  
 0   householdID             3200 non-null   int64  
 1   visits                  3200 non-null   int64  
 2   avgrides_perperson      3200 non-null   float64
 3   avgmerch_perperson      3200 non-null   float64
 4   avggoldzone_perperson   3200 non-null   float64
 5   avgfood_perperson       3200 non-null   float64
 6   goldzone_playersclub    3200 non-null   int64  
 7   own_car                 3200 non-null   int64  
 8   homestate               3200 non-null   object 
 9   FB_Like                 3200 non-null   int64  
 10  renew                   3200 non-null   int64  
dtypes: float64(4), int64(6), object(1)
memory usage: 275.1+ KB
```

Target Variable

In [59]:
```
nyc_historical.renew.value_counts()
```

Out[59]:
```
1    2126
0    1074
Name: renew, dtype: int64
```

This dataset is imbalanced, nearly 66% householders renewing the pass card. But fortunately, it is an extreme case. Dealing with slight imbalanced data, we will measure model performance with recall, F1 score, ROC curve, and its AUC

In [ ]:

In [60]:
```
nyc_historical.homestate.value_counts()
```

Out[60]:
```
NJ    1076
NY    1065
CT    1059
Name: homestate, dtype: int64
```

In [61]:
```
renew_data = nyc_historical
```

In [62]:
```
homestate_dummy = pd.get_dummies(nyc_historical.homestate)
homestate_dummy = homestate_dummy.drop(["CT"], axis=1)
homestate_dummy.rename(columns = {'NJ':'homestate_NJ'}, inplace = True)
homestate_dummy.rename(columns = {'NY':'homestate_NY'}, inplace = True)

renew_data = pd.concat([nyc_historical,homestate_dummy],axis = 1)
renew_data = renew_data.drop(['homestate'],axis = 1)
```

In [63]:
```
renew_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3200 entries, 0 to 3199
Data columns (total 12 columns):
 #   Column                 Non-Null Count  Dtype  
---  ------                 --------------  -----  
 0   householdID            3200 non-null   int64  
 1   visits                 3200 non-null   int64  
 2   avgrides_perperson     3200 non-null   float64
 3   avgmerch_perperson     3200 non-null   float64
 4   avggoldzone_perperson  3200 non-null   float64
 5   avgfood_perperson      3200 non-null   float64
 6   goldzone_playersclub   3200 non-null   int64  
 7   own_car                3200 non-null   int64  
 8   FB_Like                3200 non-null   int64  
 9   renew                  3200 non-null   int64  
 10  homestate_NJ           3200 non-null   uint8  
 11  homestate_NY           3200 non-null   uint8  
dtypes: float64(4), int64(6), uint8(2)
memory usage: 256.4 KB
```

In [64]:
```
renew_data.head()
```

Out[64]:

	householdID	visits	avgrides_perperson	avgmerch_perperson	avggoldzone_perperson	avgfood_perperson	goldzone_playersclub	own_car
0	44	20	9.8	32.4	27.2	70.7	0	1
1	57	20	11.7	71.8	40.8	1.6	0	1
2	63	20	9.8	27.4	25.7	74.9	0	1
3	159	17	2.2	1.5	91.1	28.9	1	1
4	162	19	3.4	5.0	12.0	9.2	0	1

In [65]:
```
# household ID is useless, so we drop it
renew_data = renew_data.drop(["householdID"],axis = 1)
```

In [66]:
```
X = renew_data[["visits","avgrides_perperson","avgmerch_perperson","avggoldzone_perperson","avgfood_perpe
rson","goldzone_playersclub",
"own_car","FB_Like","homestate_NJ","homestate_NY"]]
y = renew_data["renew"]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=21)
```

In [ ]:

Forecasting Process The time series is the total spending by greater NYC visitors to New England parks during 1959 to 2019. Our goal is to build a predictive model to predict the future spending for the next five years. There are mainly three steps: data exploration, building models, and selecting the best model for predicting. At first, we split data into training dataset and test dataset. Because of the time factor, we divided the total data as training data and test data by time. The first 70% older data is training data, and 30% newer data is test data. Then we decompose the time series into trend, seasonal variation, and notice that there is no seasonality and no trend. Time series is fluctuating around the mean value. To check the stationary property, we plotted ACF and PACF. Both fall into a confidence interval when lag = 0. Considering these properties, we decided to model Naive model, Simple Average model, Moving Average model, Simple Exponential Smoothing model and ARMA model. In the Moving Average model, we try different window sizes from 1 to 10, and select the moving average model with the minimized RMSE, whose window size is 9. For the ARMA model, we only try AR(1) and ARMA(1,1) due to ACF and PACF plots cutting off after lag = 1, selecting the one with the largest AIC and BIC. Comparing all the models, the simple average model minimized RMSE. For more precise prediction, we used all the dataset to build a new simple average model. This model forecasts the expected value equal to the average of all previously observed points. The predictions for five years are the same value, which is 24185805.88 dollars. The constant level is due to its algorithm. Unlike the ARMA model, it won't capture the autoregressive factor. It takes the average of all the values previously known as the next value. Of course it won’t be exact, but somewhat close. The reason why it is the best one would be its stationary property. This data is no upward or downward trend, no cycle fluctuation and no seasonality. Just like white noise, it is hard to predict the exact value. Sometimes, the simple one is the best one.

In [ ]:

Feature Selection

1. Compute the primary value ratio of each variable.

If the ratio is larger than 85%, just delete it. We only keep the variables which can identify the target variable.

In [67]:
```
def primaryvalue_ratio(data, ratiolimit = 1):
    recordcount = data.shape[0]  #number of row
    x = []
    for col in data.columns:
        primaryvalue = data[col].value_counts().index[0]
        ratio = float(data[col].value_counts().iloc[0])/recordcount
        x.append([ratio,primaryvalue])
    feature_primaryvalue_ratio = pd.DataFrame(x,index = data.columns)
    feature_primaryvalue_ratio.columns = ['primaryvalue_ratio','primaryvalue']

    needcol = feature_primaryvalue_ratio[feature_primaryvalue_ratio['primaryvalue_ratio']<ratiolimit]
    needcol = needcol.reset_index()
    select_data = data[list(needcol['index'])]
    return select_data
```

In [68]:
```
def primaryvalue_ratio(data):
    recordcount = data.shape[0]  #number of row
    x = []
    for col in data.columns:
        primaryvalue = data[col].value_counts().index[0]
        ratio = float(data[col].value_counts().iloc[0])/recordcount
        x.append([primaryvalue,ratio])
    feature_primaryvalue_ratio = pd.DataFrame(x,index = data.columns)
    feature_primaryvalue_ratio.columns = ["primaryvalue","primaryvalue_ratio"]
    return feature_primaryvalue_ratio
```

In [69]:
```
d = primaryvalue_ratio(X_train)
```

In [70]:
```
d.sort_values(['primaryvalue_ratio'], ascending=[0])
```

Out[70]:

	primaryvalue	primaryvalue_ratio
goldzone_playersclub	0.0	0.821429
own_car	1.0	0.750446
homestate_NJ	0.0	0.663393
homestate_NY	0.0	0.662054
FB_Like	0.0	0.537500
visits	2.0	0.144643
avgrides_perperson	10.0	0.022321
avgfood_perperson	18.4	0.005804
avgmerch_perperson	23.1	0.004464
avggoldzone_perperson	85.8	0.003571

As we can see, there is no variable being demonated by its primary value. Therefore, we keep all the variables. (we set the criticle point as 85%)

2. Check the missing value

In [71]:
```
na_table = X_train.isnull().sum(axis = 0)/X_train.shape[0]
```

In [72]:
```
na_table # there is no missing value
```

Out[72]:
```
visits                      0.0
avgrides_perperson          0.0
avgmerch_perperson          0.0
avggoldzone_perperson       0.0
avgfood_perperson           0.0
goldzone_playersclub        0.0
own_car                     0.0
FB_Like                     0.0
homestate_NJ                0.0
homestate_NY                0.0
dtype: float64
```

There is no missing value.

Modeling

In [73]:
```
Accuracy_train = []
Recall_train = []
F1_Score_train = []
Precision_train = []
Accuracy_test = []
Recall_test = []
F1_Score_test = []
Precision_test = []
AUC = []
```

Part I: Logistic Regression Model:

Correlation Analysis

Logistic regression is a linear model, therefore, we should do the correlation analysis to remove the multicorreltion

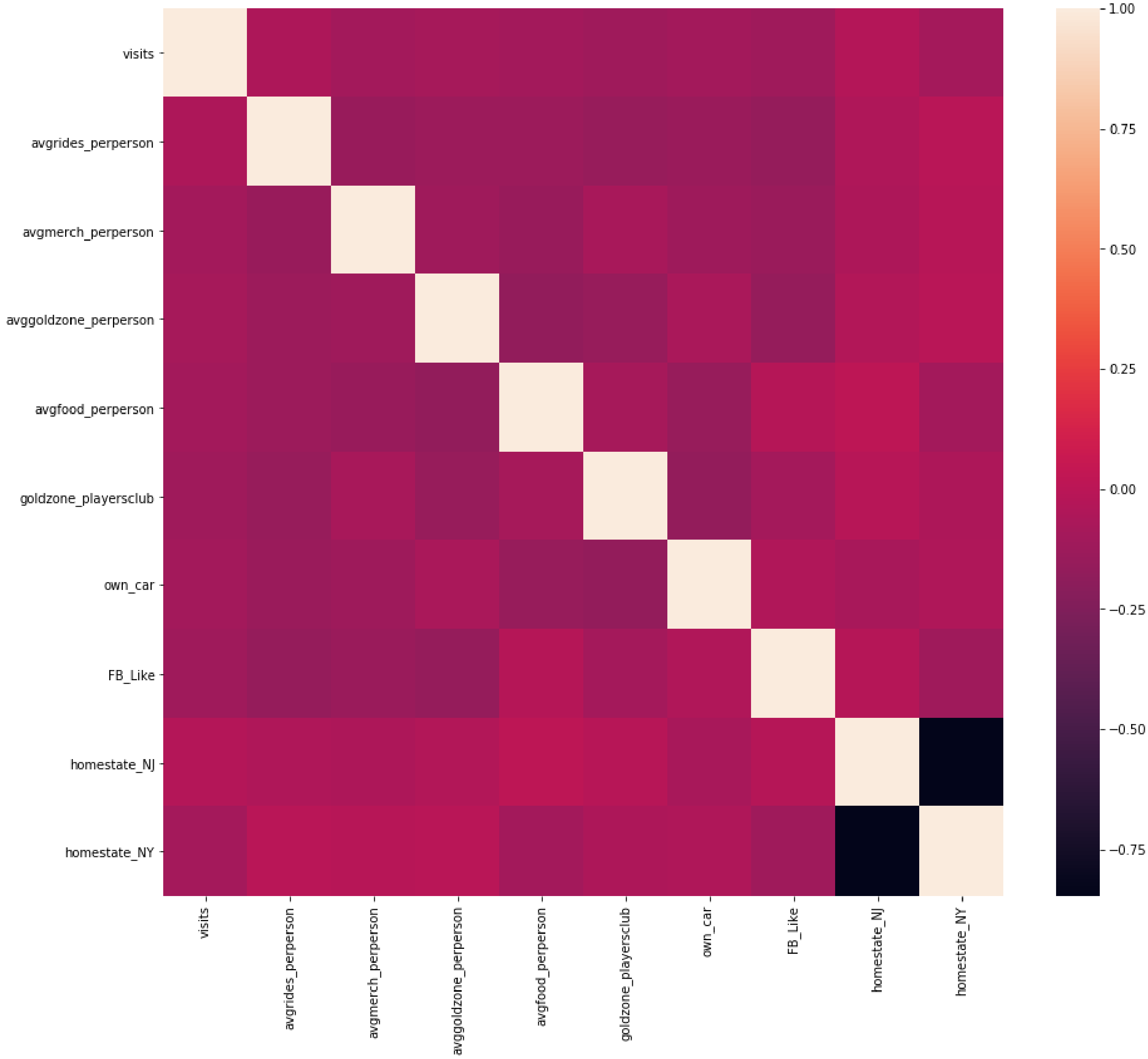
In [74]:
```
cor_table = X_train.corr()
cor_table
```

Out[74]:

	visits	avgrides_perperson	avgmerch_perperson	avggoldzone_perperson	avgfood_perperson	goldzone_playersclub
visits	1.000000	0.025168	0.006725	0.013401	0.005420	-0.002148
avgrides_perperson	0.025168	1.000000	-0.020056	-0.013009	-0.014635	-0.022243
avgmerch_perperson	0.006725	-0.020056	1.000000	-0.003353	-0.018686	0.011748
avggoldzone_perperson	0.013401	-0.013009	-0.003353	1.000000	-0.031227	-0.020628
avgfood_perperson	0.005420	-0.014635	-0.018686	-0.031227	1.000000	0.005289
goldzone_playersclub	-0.002148	-0.022243	0.011748	-0.020628	0.005289	1.000000
own_car	0.005085	-0.019313	-0.009070	0.012132	-0.025119	-0.030110
FB_Like	0.001360	-0.025086	-0.014066	-0.022254	0.037274	0.007014
homestate_NJ	0.008360	0.005241	0.003229	0.017494	0.028503	0.018148
homestate_NY	-0.021332	0.018100	0.016020	0.025145	-0.023999	-0.002904

In [75]:
```
f, ax = plt.subplots(figsize=(15, 13))
sns.heatmap(cor_table.corr())
```

Out[75]: <matplotlib.axes._subplots.AxesSubplot at 0x1a288ed0d0>



As we can see, there is no high correlations in our dataset

In [76]:
```
logmodel = LogisticRegression()
logmodel.fit(X_train,y_train)
```

```
/Users/rihiko/opt/anaconda3/lib/python3.7/site-packages/sklearn/linear_model/_logistic.py:940: Convergen
ceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)
```

Out[76]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False)
```

In [77]:
```
logmodel.intercept_
logmodel.coef_
```

Out[77]:
```
array([[ 0.11772918,  0.03811318,  0.00485627,  0.00292365,  0.00243818,
         0.52549573,  0.86123004, -0.10407854, -0.41440561, -0.27656558]])
```

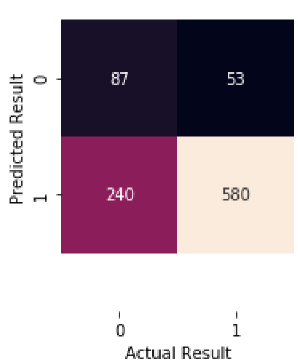
In [78]:
```
logmodel.coef_
```

Out[78]:
```
array([[ 0.11772918,  0.03811318,  0.00485627,  0.00292365,  0.00243818,
         0.52549573,  0.86123004, -0.10407854, -0.41440561, -0.27656558]])
```

In [79]:
```
X_train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 2240 entries, 2946 to 3017
Data columns (total 10 columns):
 #   Column                 Non-Null Count  Dtype  
---  ------                 --------------  -----  
 0   visits                 2240 non-null   int64  
 1   avgrides_perperson     2240 non-null   float64
 2   avgmerch_perperson     2240 non-null   float64
 3   avggoldzone_perperson  2240 non-null   float64
 4   avgfood_perperson      2240 non-null   float64
 5   goldzone_playersclub   2240 non-null   int64  
 6   own_car                2240 non-null   int64  
 7   FB_Like                2240 non-null   int64  
 8   homestate_NJ           2240 non-null   uint8  
 9   homestate_NY           2240 non-null   uint8  
dtypes: float64(4), int64(4), uint8(2)
memory usage: 161.9 KB
```

In [80]:
```
# prediction
predictions = logmodel.predict(X_test)
# confusion matrix
mat = confusion_matrix(predictions, y_test)
sns.heatmap(mat, square=True, annot=True, cbar=False,fmt='.20g')
plt.xlabel("Actual Result")
plt.ylabel("Predicted Result")
a, b = plt.ylim()
a += 0.5
b -= 0.5
plt.ylim(a, b)
plt.show()
```



In [81]:
```
pred_train = logmodel.predict(X_train)
print(classification_report(y_train, pred_train,digits=4))
```

```
              precision    recall  f1-score   support

           0     0.6476    0.3025    0.4124       747
           1     0.7245    0.9176    0.8097      1493

    accuracy                         0.7125      2240
   macro avg     0.6860    0.6101    0.6111      2240
weighted avg     0.6988    0.7125    0.6772      2240
```

In [82]:
```
print(classification_report(y_test, predictions,digits=4))
```

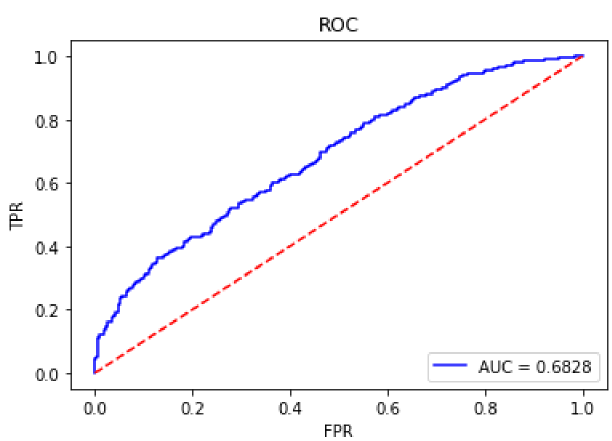
```
              precision    recall  f1-score   support

           0     0.6214    0.2661    0.3726       327
           1     0.7073    0.9163    0.7983       633

    accuracy                         0.6948       960
   macro avg     0.6644    0.5912    0.5855       960
weighted avg     0.6781    0.6948    0.6533       960
```

In [83]:
```
predslog_lr = logmodel.predict_proba(X_test)[:,1]
metrics.roc_auc_score(y_test,predslog_lr, average='macro', sample_weight=None)
```

Out[83]: 0.6828171273147141

In [84]:
```
def plot_roc(labels, predict_prob):
    false_positive_rate,true_positive_rate,thresholds=roc_curve(labels, predict_prob)
    roc_auc=auc(false_positive_rate, true_positive_rate)
    plt.title('ROC')
    plt.plot(false_positive_rate, true_positive_rate,'b',label='AUC = %0.4f'% roc_auc)
    plt.legend(loc='lower right')
    plt.plot([0,1],[0,1],'r--')
    plt.ylabel('TPR')
    plt.xlabel('FPR')
```

In [85]:
```
plot_roc(y_test,predslog_lr)
```



In [86]:
```
Accuracy_train.append(0.7125)
Recall_train.append(0.9176)
F1_Score_train.append(0.8097)
Precision_train.append(0.7245)
Accuracy_test.append(0.6948)
Recall_test.append(0.9163)
F1_Score_test.append(0.7983)
Precision_test.append(0.7073)
AUC.append(0.6828171273147141)
```

Part II: Random Forest Model

In [87]:
```
clf_rf=RandomForestClassifier(random_state = 654)
clf_rf.fit(X_train,y_train)
```

Out[87]:
```
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=654,
                       verbose=0, warm_start=False)
```

In [88]:
```
param_grid = {
    'n_estimators': [200],    # large n_estimators can predict more precise. Therefore, I only consider a
 large number, 200 trees.
    'max_depth': [2, 4, 6, 8],
    'max_features': [2, 3, 4],   # In general, max_features should be set as sqrt of n_feature. sqrt(10) =
3 or 4
    'min_samples_leaf': [6, 8, 10, 12],   # smaller number of leaf would tend to capture the noise of data
set. Therefore, I set []
}
```

In [ ]:
```

```

In [ ]:
```

```

In [89]:
```
CV_rfc = GridSearchCV(estimator=clf_rf, param_grid=param_grid, cv= 5)
CV_rfc.fit(X_train, y_train)
print(CV_rfc.best_params_)
```

```
{'max_depth': 8, 'max_features': 4, 'min_samples_leaf': 8, 'n_estimators': 200}
```

In [90]:
```
clf_rf=RandomForestClassifier(n_estimators=200, max_depth=8, max_features=4, min_samples_leaf=8, random_s
tate=654)
clf_rf.fit(X_train,y_train)
```

Out[90]:
```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=8, max_features=4,
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=8, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=200,
                       n_jobs=None, oob_score=False, random_state=654,
                       verbose=0, warm_start=False)
```

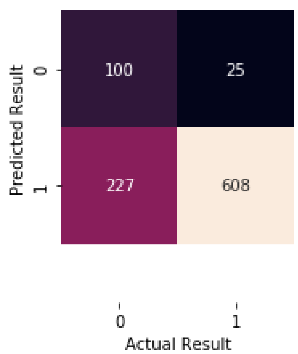
In [92]:
```
feature_imp_df = pd.DataFrame(list(zip(clf_rf.feature_importances_, X_train)))
feature_imp_df.columns = ['feature importance', 'feature']
feature_imp_df = feature_imp_df.sort_values(by='feature importance', ascending=False)
feature_imp_df
```

Out[92]:

	feature importance	feature
0	0.276660	visits
1	0.115035	avgrides_perperson
3	0.114043	avggoldzone_perperson
8	0.111199	homestate_NJ
2	0.109533	avgmerch_perperson
6	0.106009	own_car
4	0.090761	avgfood_perperson
9	0.034174	homestate_NY
5	0.029154	goldzone_playersclub
7	0.013431	FB_Like

- According to feature importance value, we can figure out the top 5 important features, which are avggoldzone_perperson, avgmerch_perperson, visits, avgfood_perperson and avgrides_perperson. These features have important value larger than 0.16, which means they are strongly predictable.
- In general, when we have large size of features, we keep those have an importance of more than 0.15. However, our dataset is only have 10 variables. It doesn't matter to keep all the variables in random forest and the following XGBoost because they are not linear model. And more variables will keep more information. Therefore, we keep all the feature.

In [103]:
```
# prediction
predictions = clf_rf.predict(X_test)
# confusion matrix
mat = confusion_matrix(predictions, y_test)
sns.heatmap(mat, square=True, annot=True, cbar=False,fmt='.20g')
plt.xlabel("Actual Result")
plt.ylabel("Predicted Result")
a, b = plt.ylim()
a += 0.5
b -= 0.5
plt.ylim(a, b)
plt.show()
```



In [104]:
```
predictions = clf_rf.predict(X_train)
print(classification_report(y_train, predictions,digits=4))
```

```
              precision    recall  f1-score   support

           0     0.9233    0.4029    0.5610       747
           1     0.7670    0.9833    0.8618      1493

    accuracy                         0.7897      2240
   macro avg     0.8451    0.6931    0.7114      2240
weighted avg     0.8191    0.7897    0.7615      2240
```

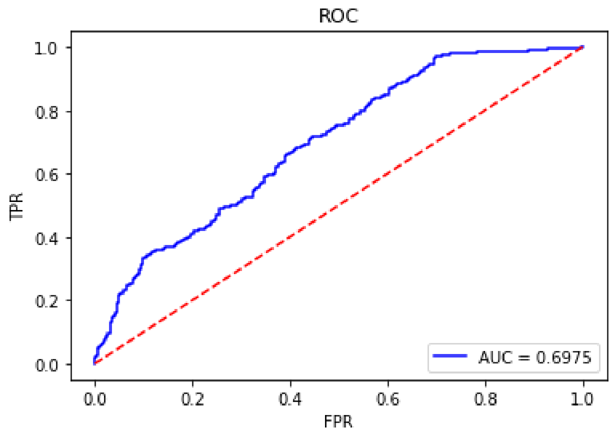
In [105]:
```
predictions = clf_rf.predict(X_test)
print(classification_report(y_test, predictions,digits=4))
```

```
              precision    recall  f1-score   support

           0     0.8000    0.3058    0.4425       327
           1     0.7281    0.9605    0.8283       633

    accuracy                         0.7375       960
   macro avg     0.7641    0.6332    0.6354       960
weighted avg     0.7526    0.7375    0.6969       960
```

In [106]:
```
predslog_rf = clf_rf.predict_proba(X_test)[:,1]
metrics.roc_auc_score(y_test,predslog_rf, average='macro', sample_weight=None)
```

Out[106]: 0.6974892628181901

In [107]:
```
plot_roc(y_test,predslog_rf)
```



In [108]:
```
Accuracy_train.append(0.7897)
Recall_train.append(0.9833)
F1_Score_train.append(0.8618)
Precision_train.append(0.7670)
Accuracy_test.append(0.7375)
Recall_test.append(0.9605)
F1_Score_test.append(0.8283)
Precision_test.append(0.7281)
AUC.append(0.6974892628181901)
```

In [ ]:

In [ ]:

Part III : XGBoost Model

Build initial model

In [109]:
```
clf_xgb=xgb.XGBClassifier(random_state=654)
clf_xgb.fit(X_train,y_train)
```

Out[109]:
```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, gamma=0,
              learning_rate=0.1, max_delta_step=0, max_depth=3,
              min_child_weight=1, missing=None, n_estimators=100, n_jobs=1,
              nthread=None, objective='binary:logistic', random_state=654,
              reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
              silent=None, subsample=1, verbosity=1)
```

In XGBoost, I tuned hyperparameters in 6 steps.

Hyoeroarameters tuning

step 1: n_estimators

In [110]:
```
cv_params = {'n_estimators': [200, 300, 400]}
other_params = {'learning_rate': 0.1, 'n_estimators': 500, 'max_depth': 5, 'min_child_weight': 1, 'seed':
0,
                    'subsample': 0.8, 'colsample_bytree': 0.8, 'gamma': 0, 'reg_alpha': 0, 'reg_lambda':
1,'objective': 'binary:logistic'}
```

In [111]:
```
clf_xgb = xgb.XGBRegressor(**other_params)
optimized_GBM = GridSearchCV(estimator=clf_xgb, param_grid=cv_params, scoring='r2', cv=5, verbose=1, n_jo
bs=4)
optimized_GBM.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 3 candidates, totalling 15 fits

[Parallel(n_jobs=4)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=4)]: Done  15 out of  15 | elapsed:    4.5s finished
```

Out[111]:
```
GridSearchCV(cv=5, error_score=nan,
             estimator=XGBRegressor(base_score=0.5, booster='gbtree',
                                    colsample_bylevel=1, colsample_bynode=1,
                                    colsample_bytree=0.8, gamma=0,
                                    importance_type='gain', learning_rate=0.1,
                                    max_delta_step=0, max_depth=5,
                                    min_child_weight=1, missing=None,
                                    n_estimators=500, n_jobs=1, nthread=None,
                                    objective='binary:logistic', random_state=0,
                                    reg_alpha=0, reg_lambda=1,
                                    scale_pos_weight=1, seed=0, silent=None,
                                    subsample=0.8, verbosity=1),
             iid='deprecated', n_jobs=4,
             param_grid={'n_estimators': [200, 300, 400]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring='r2', verbose=1)
```

In [112]:
```
optimized_GBM.best_params_
```

Out[112]:
```
{'n_estimators': 200}
```

Step 2: max_depth & min_child_weight

In [113]:
```
cv_params = {'max_depth': list(range(1,5,1)), 'min_child_weight': [1, 2, 3, 4, 5, 6]}
other_params = {'learning_rate': 0.1, 'n_estimators': 200, 'max_depth': 5, 'min_child_weight': 1, 'seed':
0,
                    'subsample': 0.8, 'colsample_bytree': 0.8, 'gamma': 0, 'reg_alpha': 0, 'reg_lambda':
1}

clf_xgb = xgb.XGBRegressor(**other_params)
optimized_GBM = GridSearchCV(estimator=clf_xgb, param_grid=cv_params, scoring='r2', cv=5, verbose=1, n_jo
bs=4)
optimized_GBM.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 24 candidates, totalling 120 fits

[Parallel(n_jobs=4)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=4)]: Done  76 tasks      | elapsed:    4.7s
[Parallel(n_jobs=4)]: Done 120 out of 120 | elapsed:    9.2s finished

[18:11:00] WARNING: src/objective/regression_obj.cu:152: reg:linear is now deprecated in favor of reg:sq
uarederror.
```

Out[113]:
```
GridSearchCV(cv=5, error_score=nan,
             estimator=XGBRegressor(base_score=0.5, booster='gbtree',
                                    colsample_bylevel=1, colsample_bynode=1,
                                    colsample_bytree=0.8, gamma=0,
                                    importance_type='gain', learning_rate=0.1,
                                    max_delta_step=0, max_depth=5,
                                    min_child_weight=1, missing=None,
                                    n_estimators=200, n_jobs=1, nthread=None,
                                    objective='reg:linear', random_state=0,
                                    reg_alpha=0, reg_lambda=1,
                                    scale_pos_weight=1, seed=0, silent=None,
                                    subsample=0.8, verbosity=1),
             iid='deprecated', n_jobs=4,
             param_grid={'max_depth': [1, 2, 3, 4],
                         'min_child_weight': [1, 2, 3, 4, 5, 6]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring='r2', verbose=1)
```

In [114]:
```
optimized_GBM.best_params_
```

Out[114]:
```
{'max_depth': 3, 'min_child_weight': 5}
```

Step 3: gamma

In [115]:
```
cv_params = {'gamma': [0.1, 0.2, 0.3, 0.4, 0.5, 0.6]}
other_params = {'learning_rate': 0.1, 'n_estimators': 200, 'max_depth': 3, 'min_child_weight': 5, 'seed': 0,
                    'subsample': 0.8, 'colsample_bytree': 0.8, 'gamma': 0, 'reg_alpha': 0, 'reg_lambda': 1}

clf_xgb = xgb.XGBRegressor(**other_params)
optimized_GBM = GridSearchCV(estimator=clf_xgb, param_grid=cv_params, scoring='r2', cv=5, verbose=1, n_jobs=4)
optimized_GBM.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 6 candidates, totalling 30 fits

[Parallel(n_jobs=4)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=4)]: Done  30 out of  30 | elapsed:    2.3s finished

[18:11:02] WARNING: src/objective/regression_obj.cu:152: reg:linear is now deprecated in favor of reg:squarederror.
```

Out[115]:
```
GridSearchCV(cv=5, error_score=nan,
             estimator=XGBRegressor(base_score=0.5, booster='gbtree',
                                    colsample_bylevel=1, colsample_bynode=1,
                                    colsample_bytree=0.8, gamma=0,
                                    importance_type='gain', learning_rate=0.1,
                                    max_delta_step=0, max_depth=3,
                                    min_child_weight=5, missing=None,
                                    n_estimators=200, n_jobs=1, nthread=None,
                                    objective='reg:linear', random_state=0,
                                    reg_alpha=0, reg_lambda=1,
                                    scale_pos_weight=1, seed=0, silent=None,
                                    subsample=0.8, verbosity=1),
             iid='deprecated', n_jobs=4,
             param_grid={'gamma': [0.1, 0.2, 0.3, 0.4, 0.5, 0.6]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring='r2', verbose=1)
```

In [116]:
```
optimized_GBM.best_params_
```

Out[116]:
```
{'gamma': 0.6}
```

step 4: subsample, colsample_bytree：

In [117]:
```
cv_params = {'subsample': [0.6, 0.7, 0.8, 0.9], 'colsample_bytree': [0.6, 0.7, 0.8, 0.9]}
other_params = {'learning_rate': 0.1, 'n_estimators': 200, 'max_depth': 3, 'min_child_weight': 5, 'seed': 0,
                    'subsample': 0.8, 'colsample_bytree': 0.8, 'gamma': 0.6, 'reg_alpha': 0, 'reg_lambda': 1}
```

In [118]:
```
clf_xgb = xgb.XGBRegressor(**other_params)
optimized_GBM = GridSearchCV(estimator=clf_xgb, param_grid=cv_params, scoring='r2', cv=5, verbose=1, n_jobs=4)
optimized_GBM.fit(X_train, y_train)
optimized_GBM.best_params_
```

```
Fitting 5 folds for each of 16 candidates, totalling 80 fits

[Parallel(n_jobs=4)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=4)]: Done  42 tasks      | elapsed:    3.1s
[Parallel(n_jobs=4)]: Done  80 out of  80 | elapsed:    5.7s finished

[18:11:08] WARNING: src/objective/regression_obj.cu:152: reg:linear is now deprecated in favor of reg:squarederror.
```

Out[118]:
```
{'colsample_bytree': 0.9, 'subsample': 0.9}
```

step 5: regalpha & reglambda

In [119]:
```
cv_params = {'reg_alpha': [0.05, 0.1, 1, 2, 3], 'reg_lambda': [0.05, 0.1, 1, 2, 3]}
other_params = {'learning_rate': 0.1, 'n_estimators': 200, 'max_depth': 3, 'min_child_weight': 5, 'seed':
0,
                    'subsample': 0.9, 'colsample_bytree': 0.9, 'gamma': 0.6, 'reg_alpha': 0, 'reg_lambda'
: 1}
```

In [120]:
```
clf_xgb = xgb.XGBRegressor(**other_params)
optimized_GBM = GridSearchCV(estimator=clf_xgb, param_grid=cv_params, scoring='r2', cv=5, verbose=1, n_jo
bs=4)
optimized_GBM.fit(X_train, y_train)
optimized_GBM.best_params_
```

```
Fitting 5 folds for each of 25 candidates, totalling 125 fits

[Parallel(n_jobs=4)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=4)]: Done  42 tasks      | elapsed:    3.9s
[Parallel(n_jobs=4)]: Done 125 out of 125 | elapsed:   10.3s finished

[18:11:19] WARNING: src/objective/regression_obj.cu:152: reg:linear is now deprecated in favor of reg:sq
uarederror.
```

Out[120]:
```
{'reg_alpha': 3, 'reg_lambda': 0.05}
```

step 6: learning rate

In [121]:
```
cv_params = {'learning_rate': [0.01, 0.05, 0.07, 0.1, 0.2]}
other_params = {'learning_rate': 0.1, 'n_estimators': 200, 'max_depth': 3, 'min_child_weight': 5, 'seed':
0,
                    'subsample': 0.9, 'colsample_bytree': 0.9, 'gamma': 0.6, 'reg_alpha': 3, 'reg_lambda'
: 0.05}
```

In [122]:
```
clf_xgb = xgb.XGBRegressor(**other_params)
optimized_GBM = GridSearchCV(estimator=clf_xgb, param_grid=cv_params, scoring='r2', cv=5, verbose=1, n_jo
bs=4)
optimized_GBM.fit(X_train, y_train)
optimized_GBM.best_params_
```

```
Fitting 5 folds for each of 5 candidates, totalling 25 fits

[Parallel(n_jobs=4)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=4)]: Done  25 out of  25 | elapsed:    2.2s finished

[18:11:21] WARNING: src/objective/regression_obj.cu:152: reg:linear is now deprecated in favor of reg:sq
uarederror.
```

Out[122]:
```
{'learning_rate': 0.1}
```

Build XGBoost Model

In [123]:
```
clf_xgb = xgb.XGBClassifier(n_estimators=200, max_depth=3,
                             learning_rate=0.1, subsample=0.9, colsample_bytree=0.9,scale_pos_weight=3.0,
                              silent=True, nthread=-1, seed=0, missing=None,objective='binary:logistic',
                              reg_alpha=3, reg_lambda=0.05,
                              gamma=0.6, min_child_weight=5,
                              max_delta_step=0,base_score=0.5)

clf_xgb.fit(X_train, y_train)
```

Out[123]:
```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=0.9, gamma=0.6,
              learning_rate=0.1, max_delta_step=0, max_depth=3,
              min_child_weight=5, missing=None, n_estimators=200, n_jobs=1,
              nthread=-1, objective='binary:logistic', random_state=0,
              reg_alpha=3, reg_lambda=0.05, scale_pos_weight=3.0, seed=0,
              silent=True, subsample=0.9, verbosity=1)
```

Feature importance

In [124]:
```
# feature importance
feature_imp_df = pd.DataFrame(list(zip(clf_xgb.feature_importances_, X_train)))
feature_imp_df.columns = ['feature importance', 'feature']
feature_imp_df = feature_imp_df.sort_values(by='feature importance', ascending=False)
feature_imp_df
```

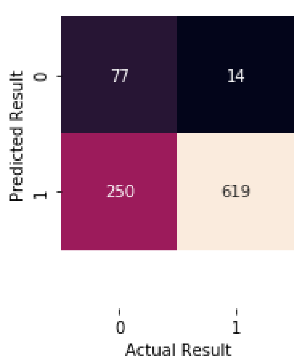
Out[124]:

	feature importance	feature
8	0.321522	homestate_NJ
6	0.151555	own_car
0	0.135880	visits
9	0.095966	homestate_NY
5	0.075694	goldzone_playersclub
1	0.061954	avgrides_perperson
3	0.043305	avggoldzone_perperson
4	0.041291	avgfood_perperson
2	0.038462	avgmerch_perperson
7	0.034371	FB_Like

- According to feature importance value, it is noticable that homestate_NJ is the most important feature. Next is own_car, visits. These variables are strongly predictable.

Measure performance

In [125]:
```
# prediction
predictions = clf_xgb.predict(X_test)
# confusion matrix
mat = confusion_matrix(predictions, y_test)
sns.heatmap(mat, square=True, annot=True, cbar=False,fmt='.20g')
plt.xlabel("Actual Result")
plt.ylabel("Predicted Result")
a, b = plt.ylim()
a += 0.5
b -= 0.5
plt.ylim(a, b)
plt.show()
```



In [126]:
```
predictions = clf_xgb.predict(X_train)
print(classification_report(y_train, predictions,digits=4))
```

```
              precision    recall  f1-score   support

           0     0.9598    0.2878    0.4428       747
           1     0.7361    0.9940    0.8458      1493

    accuracy                         0.7585      2240
   macro avg     0.8480    0.6409    0.6443      2240
weighted avg     0.8107    0.7585    0.7114      2240
```

In [127]:
```
predictions = clf_xgb.predict(X_train)
print(classification_report(y_train, predictions,digits=4))
```

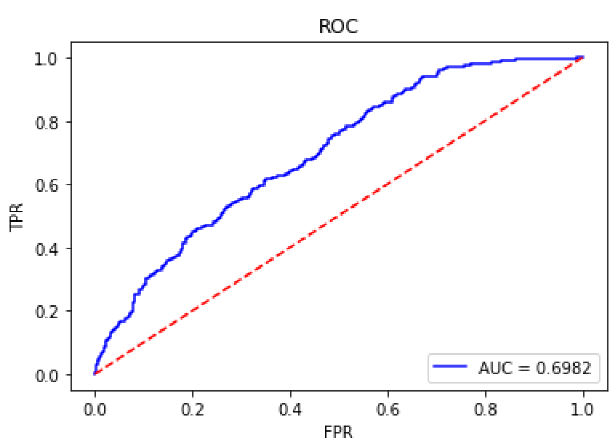
```
              precision    recall  f1-score   support

           0     0.9598    0.2878    0.4428       747
           1     0.7361    0.9940    0.8458      1493

    accuracy                         0.7585      2240
   macro avg     0.8480    0.6409    0.6443      2240
weighted avg     0.8107    0.7585    0.7114      2240
```

In [128]:
```
predslog_rf = clf_xgb.predict_proba(X_test)[:,1]
metrics.roc_auc_score(y_test,predslog_rf, average='macro', sample_weight=None)
```

Out[128]: 0.6981704518553946

In [129]:
```
Accuracy_train.append(0.7585)
Recall_train.append(0.9940)
F1_Score_train.append(0.8458)
Precision_train.append(0.7361)
Accuracy_test.append(0.7585)
Recall_test.append(0.9940)
F1_Score_test.append(0.8458)
Precision_test.append(0.7361)
AUC.append(0.6981704518553946)
```

In [130]:
```
plot_roc(y_test,predslog_rf)
```



Model Selection

In [131]:
```
name = ["Logistic Regression","Random Forest", "XGBoost"]
```

In [132]:
```
clf_performance = pd.DataFrame({"Accuracy_test" :Accuracy_test,"Recall_test": Recall_test,"F1_Score_test"
: F1_Score_test,
                                "Precision_test": Precision_test,"AUC":AUC,
                                "Accuracy_train" : Accuracy_train,"Recall_train":Recall_train,"F1_Score_t
rain" :F1_Score_train,
                                "Precision_train": Precision_train},index = name)
```

In [133]:
```
clf_performance
```

Out[133]:

	Accuracy_test	Recall_test	F1_Score_test	Precision_test	AUC	Accuracy_train	Recall_train	F1_Score_train	Precision_train
Logistic Regression	0.6948	0.9163	0.7983	0.7073	0.682817	0.7125	0.9176	0.8097	0.7245
Random Forest	0.7375	0.9605	0.8283	0.7281	0.697489	0.7897	0.9833	0.8618	0.7670
XGBoost	0.7585	0.9940	0.8458	0.7361	0.698170	0.7585	0.9940	0.8458	0.7361

As shown in the performance matrix, XGBoost is the best model with greatest accuracy, recall, f1 score, precision, AUC

Because this data is imbalanced, so we focus on recall, f1 score, precision and AUC.

In [ ]:

Classification

This dataset records the spending and personal information of Greater NYC households that held Lobster Land family season passes. The goal is to predict whether an unknown customer renews the pass card or not. Understanding what consumers want and need is an ongoing imperative for marketing strategy. Machine Learning can make this job much easier and efficient.

1. Predict Whether Customers Renew Passes or Not

- To uncover the potential factors toward renewing a fast card, we built three classification models using all the variables. These three models are Logistic Regression, Random Forest and XGBoost. We selected XGBoost as our final model for better performance. Based on the criteria, the classification models can divide individuals as two groups.
- When we know an unknown customer information such as owning a car or not, the number of times that the pass was used during the season, we can predict whether the member will renew their card for next season or not.

- However, the model seems to be useless because we already know whether these customers renew or not. Even though these classification models might be useful for prediction in next season, these are more seasonal factors influencing on renewing. Situations are quite distinct in different seasons. For example, the ratio for spring customers to renew summer passes must be higher than that for the winter to renew spring passes. Winter and summer are two holiday for most students.
- Therefore, there are more crucial things than predictions. We should be mining what important factors leading to customers renew their passes. And use these factors to promote the potential customers.

2. Apply Feature Importance to Target Potential Customers

Each model reveals the importances of features. That is, these relative scores can highlight which features may be most relevant to the target, and the converse, which features are the least relevant.

In Logistic Regression, these coefficients can provide the basis for a crude feature importance score. The higher the coefficient, the higher the "importance" of a feature. As the result shown, **goldzone_playersclub**, **homestate_NJ**(most negative), **own_car** these three features have higher coefficients. In Random Forest and XGBoost, the results are more obvious. These importance values are calculated by how to split the tree. We can obtain them by models' outputs. In random forest, most top 5 valuable features are **avggoldzone_perperson**, **avgmerch_perperson**, **visits**, **avgfood_perperson**, **avgrides_perperson**. In XGBoost, **homestate_NJ** , **own_car** and **visits** are the most three important features. To summarize, **homestate_NJ**, **own_car**, **visits**, **avggoldzone_perperson**, **avgmerch_perperson** might be most relevant for customers to renew their season passes.

1. **homestate_NJ**: here is a note that both homestate_NJ and homestate_NY have negative coefficients in Logistic Regression. In addition, the rest variable homestate_CT has been droped to avoid the multi correlation, so the coefficient is larger than homestate_NJ and homestate_NY. According to these statistics, we can infer that customers in New York and New Jersey would be busy with their works. Busy lives made them have fewer visits during last season. The second conjecture is that fewer family members in New Jersey and New York families. There are many different reasons to make them tend to believe it doesn't worth spending money on season pass over a long period. Even though they have time on vacation, amusement parks are not usually their first leisure places. Therefore, the coefficients are negative. In order to attract more target people, we should focus on the customers living in Connecticut. These customers are more likely to renew their season passes than other two states.
2. **own_car**: this variable is the essential to classify the customers. In addition, the coefficient is positive in Logistic Regression. Obviously, families owning cars are more willing to spend time in amusement parks. Big families are willing to use this type of seasonal pass. They seem to be more beneficial if more family members use them. Owning a car is a common property of such a large family. According to this finding, we can make more promotions to attract families with cars. If they become our seasonal passes members for the first time, they would be more likely to renew the cards for the next seasons.
3. **visits**: coefficient of visit is positive in Logistic Regression, and we can indicate that families with more visits are more likely to renew seasonal passes. It is reasonable that amusement park lovers are more likely to renew season passes. Lobster Land should promote the seasonal passes to those customers who have already spent a lot of time but are still not the seasonal pass members.

In conclusion, the ability to identify our target customer group and get it right, is gure gold for Lobster Land. With the help of a well-trained classification model, marketers can rely on assumptions and guesswork and more on data-driven insights to find target customers precisely. According to the feature importance values, we can do more promotions for the players living in Connecticut, owning cars, or being willing to spend time in Lobsterland.

AB Testing

In [134]:
```
adcampaign = pd.read_csv('online_merch.csv')
```

In [135]:
```
adcampaign.head(10)
```

Out[135]:

	customerID	adcampaign	periodspend
0	1	Gibson	181.14
1	2	Gibson	194.11
2	3	Gibson	217.88
3	4	Gibson	205.37
4	5	Merlino	218.79
5	6	Gibson	187.07
6	7	Merlino	194.39
7	8	Gibson	191.78
8	9	Gibson	186.22
9	10	Gibson	168.19

In [136]:
```
adcampaign.describe()
```

Out[136]:

	customerID	periodspend
count	1200.000000	1200.000000
mean	600.500000	209.965108
std	346.554469	16.887549
min	1.000000	138.160000
25%	300.750000	199.155000
50%	600.500000	211.255000
75%	900.250000	221.572500
max	1200.000000	252.020000

In [137]:
```
adcampaign.columns
```

Out[137]:
```
Index(['customerID', 'adcampaign', 'periodspend'], dtype='object')
```

In [138]:
```
adcampaign.groupby('adcampaign').describe()['periodspend']    #group size similar
```

Out[138]:

	count	mean	std	min	25%	50%	75%	max
adcampaign								
Gibson	619.0	202.406898	17.462896	138.16	191.61	202.45	213.60	252.02
Merlino	581.0	218.017659	11.777079	184.68	210.22	218.40	225.95	251.05

In [139]:
```
Gibson = adcampaign[adcampaign.adcampaign == 'Gibson']  #subset: to filter out the adcampaign == Gibson
Gibson.head()
```

Out[139]:

	customerID	adcampaign	periodspend
0	1	Gibson	181.14
1	2	Gibson	194.11
2	3	Gibson	217.88
3	4	Gibson	205.37
5	6	Gibson	187.07

In [140]:
```
Merlino = adcampaign[adcampaign.adcampaign == 'Merlino']  #subset: to filter out the adcampaign == Merlin
o
```

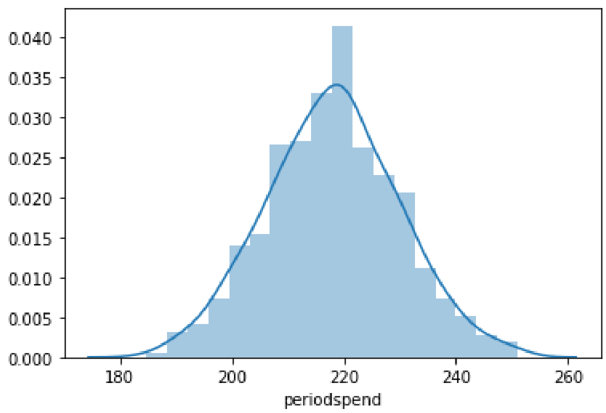
In [141]:
```
#mean of sample
G_mean = round(Gibson['periodspend'].mean(),4)
M_mean = round(Merlino['periodspend'].mean(),4)
#sd of sample
G_std = round(Gibson['periodspend'].std(),4)
M_std = round(Merlino['periodspend'].std(),4)

print('The mean spending and sd of Gibson campaign are ', G_mean,',', G_std,
      '\nThe mean spending and sd of Merlino campaign are', M_mean,',',  M_std)
```

```
The mean spending and sd of Gibson campaign are  202.4069 , 17.4629
The mean spending and sd of Merlino campaign are 218.0177 , 11.7771
```

1. H0: no difference ; Ha: G_mean != M_mean

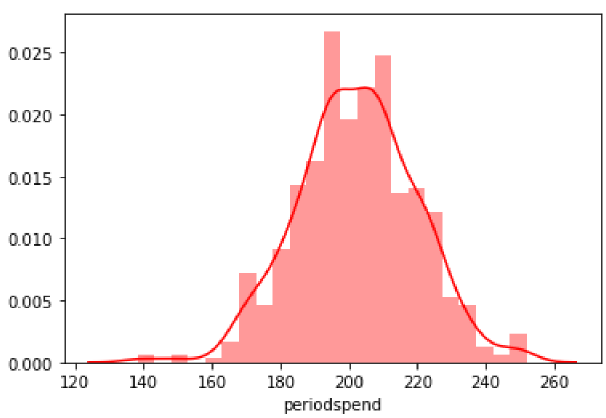
In [142]:
```
sns.distplot(Merlino['periodspend'])
```

Out[142]: <matplotlib.axes._subplots.AxesSubplot at 0x1a27783350>



In [143]:
```
sns.distplot(Gibson['periodspend'], color = 'red')
```

Out[143]: <matplotlib.axes._subplots.AxesSubplot at 0x1a28a040d0>

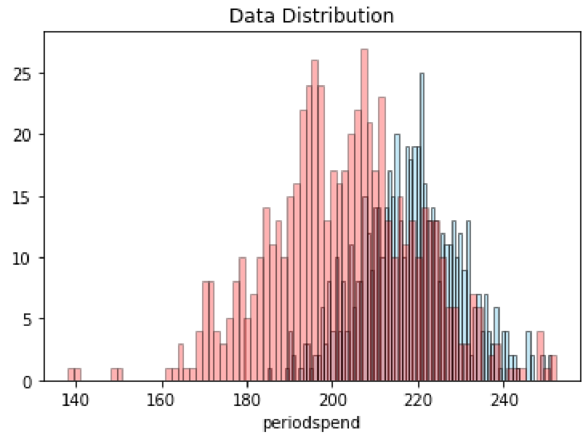


In [144]:

```
#2.compare the two groups----distribution
plt.hist(Merlino['periodspend'], 80,color = "skyblue",ec='black',alpha = 0.5)
plt.xlabel('periodspend')
plt.title('Data Distribution')

plt.hist(Gibson['periodspend'], 80,color = "red",ec='black',alpha = 0.3)
plt.xlabel('periodspend')
```

Out[144]:

```
Text(0.5, 0, 'periodspend')
```



In [145]:

```
t, p = stats.ttest_ind(adcampaign.loc[adcampaign['adcampaign'] == 'Gibson', 'periodspend'].values,
                       adcampaign.loc[adcampaign['adcampaign'] == 'Merlino', 'periodspend'].values, equal
_var=False)
print('t =', t, 'p = ', p)
```

```
t = -18.2537748476896 p =  3.7015932584398592e-65
```

In [146]:

```
alpha = 0.05
if(p < alpha/2):
    print('Reject the null hypothesis, statistically significant, accept alternative hypotheses')
    print('Alternative hypothesis: Ads G and Ads M are different')
else:
    print('Accept the null hypothesis, no statistically significant')
    print('Null assumption: No difference between Ads G and Ads M')
```

```
Reject the null hypothesis, statistically significant, accept alternative hypotheses
Alternative hypothesis: Ads G and Ads M are different
```

A/B Testing

In this step, we want to use AB-testing to analysis whether there is a meaningful difference between Two advertisement campaigns -- Gibson & Merlino. Before we start the analysis, we have accumulated 1200 regular online shoppers, and randomly assigned them to each of campaign.

Firstly, we used groupby() function to directly compare the two groups. From the results, we can see there are 619 records in Gibson group and 581 records in Merlino. Sample size of the two groups is similar. We also compared the means and standard deviation of the two groups, we found the mean spending of Gibson is less than Merlino. But is there statistical significance between the Gibson and the Merlino? We need to go through the following hypothesis test.

Ho(Null hypothesis): no statistically significance in two versions; Ha(Alternative hypothesis): Ads G and Ads M are different

Then, we plot the data distribution of the two versions, and assumed that they follow Normal Distribution. Under the assumption of normal distribution and random sampling, we performed the T-test, and got the following results:

The p-value in our t-test is extremely low. At a significance level far greater than 95 percent, we can state that the variation in spendings between members of the two groups is not the result of random chance. We will reject the null hypothesis that there is no meaningful variation among the groups, and accept the alternative hypothesis that there is a meaningful difference between Gibson & Merlino.