adease-time-series-2

May 21, 2024

About

Ad Ease is an ads and marketing based company helping businesses elicit maximum clicks @ minimum cost. AdEase is an ad infrastructure to help businesses promote themselves easily, effectively, and economically. The interplay of 3 AI modules - Design, Dispense, and Decipher, come together to make it this an end-to-end 3 step process digital advertising solution for all.

[ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from datetime import datetime
import seaborn as sns
import seaborn.objects as so
import re
from itertools import product


from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.statespace.sarimax import SARIMAX
from prophet import Prophet


sns.set(style = 'darkgrid')
pd.set_option('display.max_columns', None)
pd.options.display.max_colwidth = 100
plt.rcParams["figure.figsize"] = (15,7)
import warnings # supress warnings
warnings.filterwarnings('ignore')
```

Importing the dataset and performing exploratory analysis : * Checking the structure * Characteristics of the dataset

[ ]:
```
data = pd.read_csv('train_1.csv')
```

[ ]:
```
exog = pd.read_csv('Exog_Campaign_eng')
```

[ ]:
```
raw_data = data.copy(deep=True)
```

[ ]:
```
data.head()
```

[ ]:
```
data.shape
```

[ ]:
```
(113650, 551)
```

Checking for Missing values

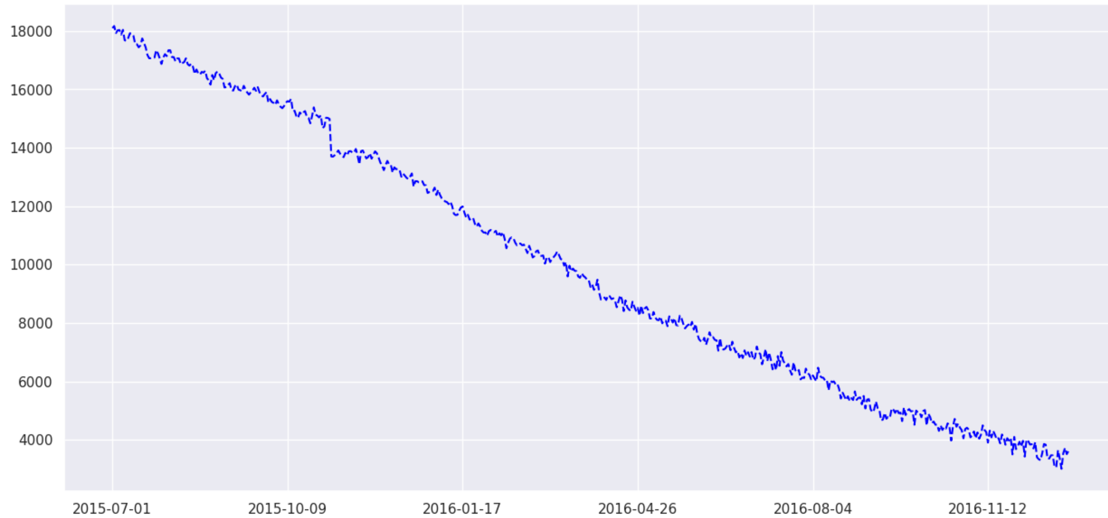
[ ]:
```
data.isnull().sum()
```

[ ]:
```
Page              0
2015-07-01    18097
2015-07-02    18169
2015-07-03    17926
2015-07-04    18022
              …
2016-12-27     3509
2016-12-28     3618
2016-12-29     3637
2016-12-30     3446
2016-12-31     3293
Length: 551, dtype: int64
```

[ ]:
```
data.loc[data['Page']=='52_Hz_I_Love_You_zh.wikipedia.org_all-access_spider']
d1 = datetime.strptime('2015-07-01', "%Y-%m-%d")
print('Start date:', d1)

d2 = datetime.strptime('2016-04-16', "%Y-%m-%d")
print('End time:',d2)

# get difference
delta = d2 - d1

# time difference in seconds
print(f"Days difference is {delta} seconds")
```

```
Start date: 2015-07-01 00:00:00
End time: 2016-04-16 00:00:00
Days difference is 290 days, 0:00:00 seconds
```

[ ]:
```
data.iloc[:, 1:-3 ].isnull().sum().plot(color='blue', linestyle='dashed')
plt.show()
```



- The chart above illustrates a decreasing trend in NaN/Null values over time. Recent dates exhibit fewer Null Values compared to earlier dates.
- This phenomenon is plausible because pages created or hosted at later dates naturally lack data for previous dates (dates preceding their creation/hosting).
- To address this, we plan to eliminate rows containing more than 300 Null Values and substitute the remaining Null Values with 0.

[ ]:
```
data.dropna(thresh = 300, inplace = True)
print(f'Shape of Data : {data.shape}')
data.fillna(0, inplace = True)
```

```
Shape of Data : (103564, 551)
```

[ ]:
```
data.fillna(0, inplace = True)
```

Feature Engineering

[ ]:
```
def get_language(name):
    if len(re.findall(r'_(.{2}).wikipedia.org_', name)) == 1 :
        return re.findall(r'_(.{2}).wikipedia.org_', name)[0]
    else: return 'Unknown_language'

data['language'] = data['Page'].apply(get_language)


language_dict ={'de':'German',
                'en':'English',
                'es': 'Spanish',
                'fr': 'French',
```

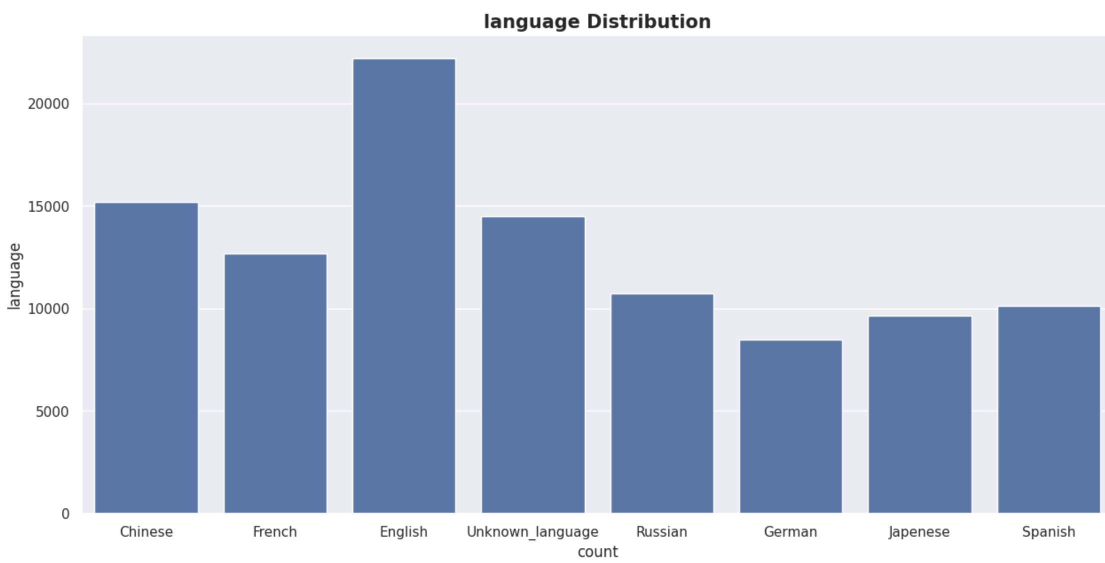
```python
                'ja': 'Japenese' ,
                'ru': 'Russian',
                'zh': 'Chinese',
                'Unknown_language': 'Unknown_language'}

data['language'] = data['language'].map(language_dict)
```

[ ]:
```python
def get_access_type(name):
    if len(re.findall(r'all-access|mobile-web|desktop', name)) == 1 :
        return re.findall(r'all-access|mobile-web|desktop', name)[0]
    else: return 'No Access_type'

data['access_type'] = data['Page'].apply(get_access_type)
```

[ ]:
```python
def get_access_origin(name):
    if len(re.findall(r'[ai].org_(.*)_(.*)$', name)) == 1 :
        return re.findall(r'[ai].org_(.*)_(.*)$', name)[0][1]
    else: return 'No Access_origin'

data['access_origin'] = data['Page'].apply(get_access_origin)
```

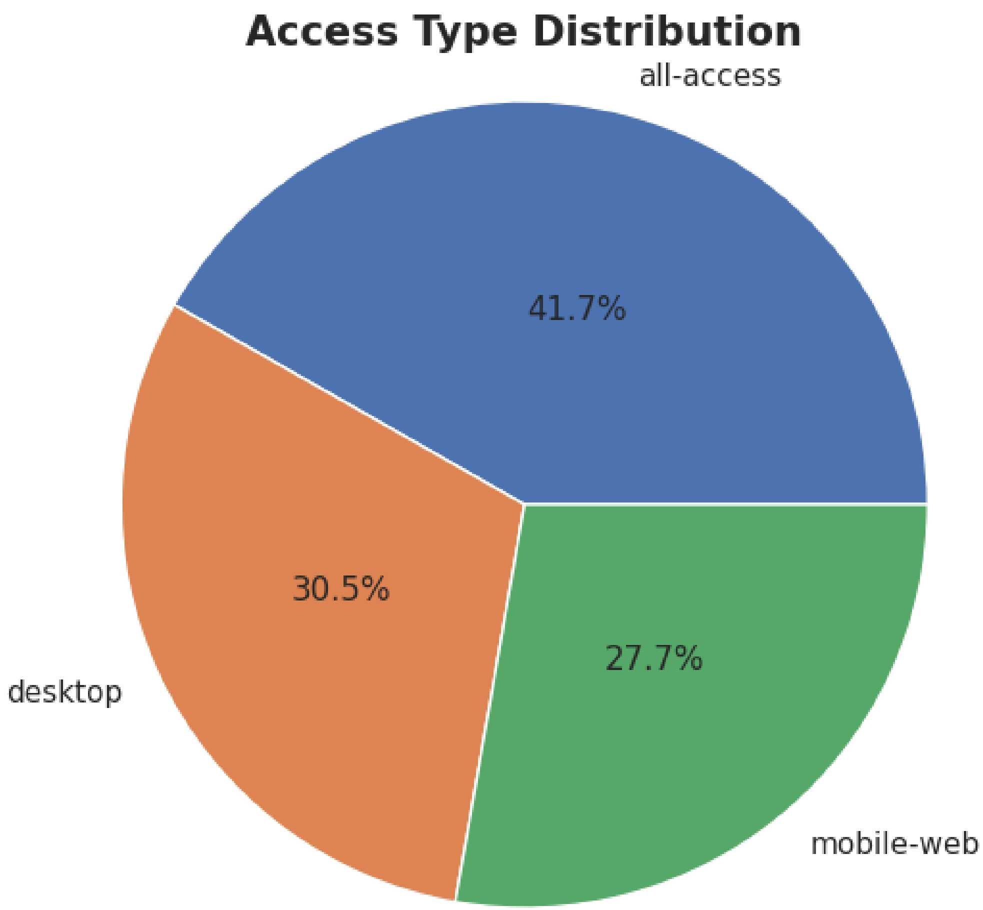
Number of langauges

[ ]:
```python
sns.countplot(x='language' , data=data)
plt.title('language Distribution')
plt.xlabel('count')
plt.ylabel('language')
plt.title('language Distribution', fontsize = 15, fontweight = 'bold')
plt.show()
```



Access type

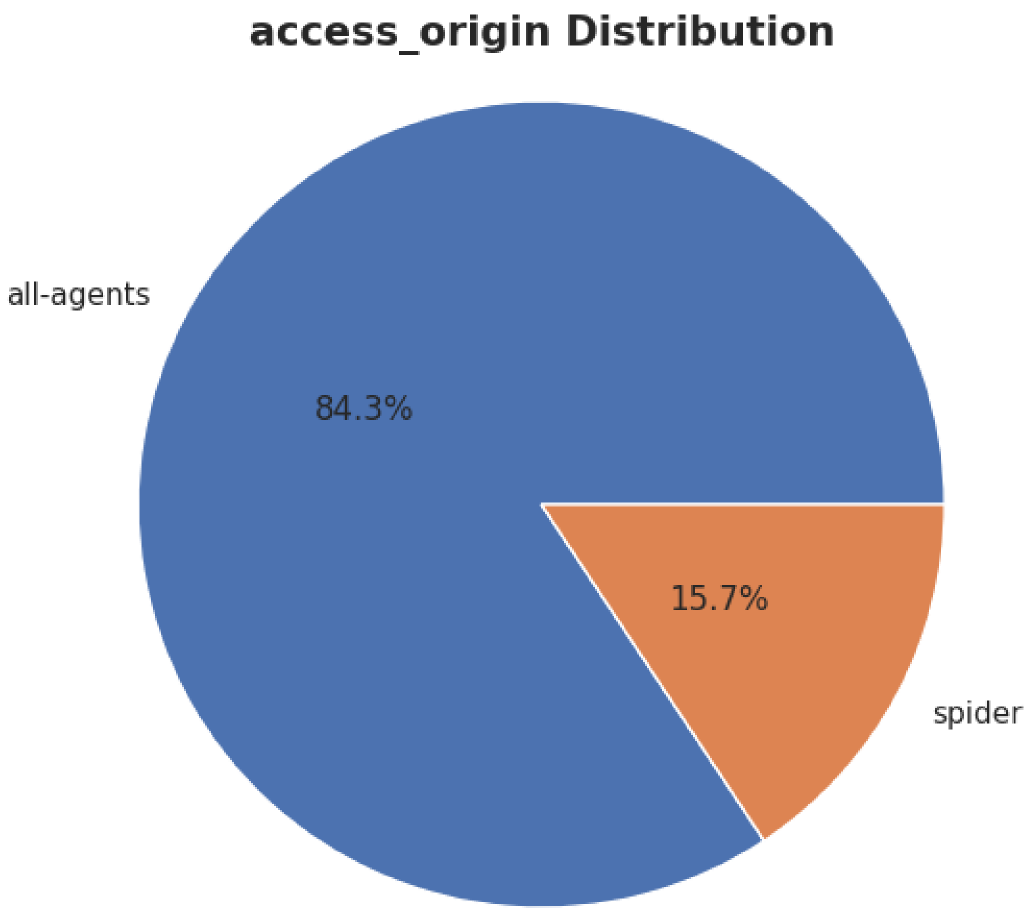
```
[ ]: x = data['access_type'].value_counts().values
     y = data['access_type'].value_counts().index

     plt.figure(figsize=(7, 6))
     plt.pie(x, labels = y, center=(0, 0), radius=1.5,  autopct='%1.1f%%',␣
      ↪pctdistance=0.5)
     plt.title('Access Type Distribution', fontsize = 15, fontweight = 'bold')
     plt.axis('equal')
     plt.show()
```



Access orgin spread

```
[ ]: var = 'access_origin'
     x = data[var].value_counts().values
```

```
y = data[var].value_counts().index

plt.figure(figsize=(7, 6))
plt.pie(x, labels = y, center=(0, 0), radius=1.5,  autopct='%1.1f%%',␣
 ↪pctdistance=0.5)
plt.title(f'{var} Distribution', fontsize = 15, fontweight = 'bold')
plt.axis('equal')
plt.show()
```



[ ]:
```
reshaped = data.melt(id_vars =␣
 ↪['Page','language','access_type','access_origin'])
```

[ ]:
```
reshaped.head()
```

[ ]:
```
                                         Page language access_type  \
0     2NE1_zh.wikipedia.org_all-access_spider  Chinese  all-access
1      2PM_zh.wikipedia.org_all-access_spider  Chinese  all-access
```

```
2       3C_zh.wikipedia.org_all-access_spider  Chinese  all-access
3  4minute_zh.wikipedia.org_all-access_spider  Chinese  all-access
4     5566_zh.wikipedia.org_all-access_spider  Chinese  all-access

  access_origin    variable  value
0        spider  2015-07-01   18.0
1        spider  2015-07-01   11.0
2        spider  2015-07-01    1.0
3        spider  2015-07-01   35.0
4        spider  2015-07-01   12.0
```

[ ]:
```
reshaped.columns = ['Page','language','access_type', 'access_origin','Date',␣
↪'Visits']
```

[ ]:
```
reshaped.Date = pd.to_datetime(reshaped.Date, format ='%Y-%m-%d')
```

[ ]:
```
lang_data = reshaped.groupby(['language', 'Date'],as_index=False)['Visits'].
↪sum()
```

[ ]:
```
lang_data.shape
```

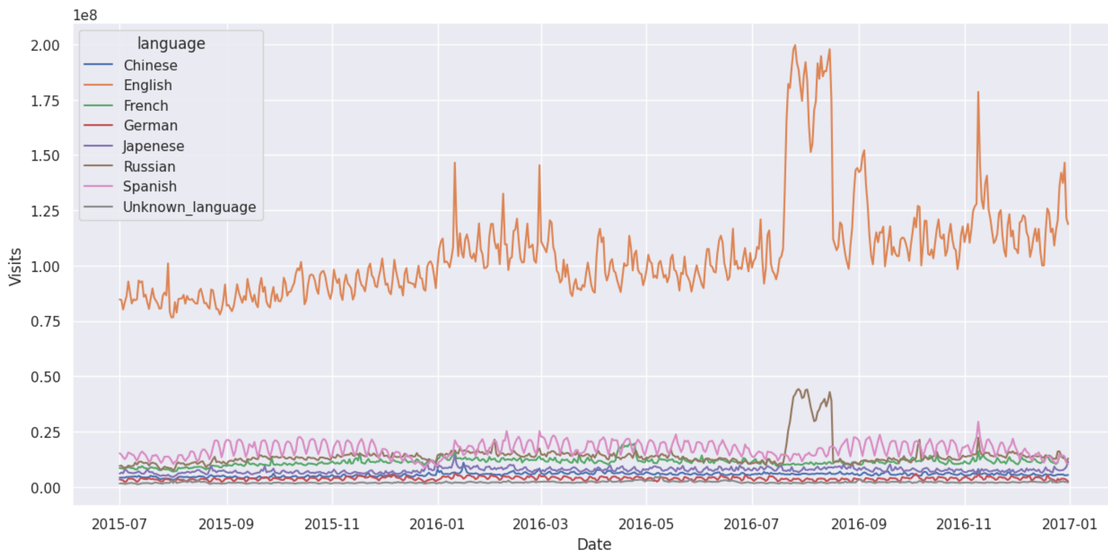
[ ]:
```
(4400, 3)
```

[ ]:
```
lang_data.head()
```

[ ]:
```
  language       Date     Visits
0  Chinese 2015-07-01  4144975.0
1  Chinese 2015-07-02  4151185.0
2  Chinese 2015-07-03  4123659.0
3  Chinese 2015-07-04  4163439.0
4  Chinese 2015-07-05  4441273.0
```

[ ]:
```
sns.lineplot(data=lang_data, y ='Visits',x='Date', hue='language')
```

[ ]:
```
<Axes: xlabel='Date', ylabel='Visits'>
```



[ ]:
```
lang_data.head()
```

[ ]:

	language	Date	Visits
0	Chinese	2015-07-01	4144975.0
1	Chinese	2015-07-02	4151185.0
2	Chinese	2015-07-03	4123659.0
3	Chinese	2015-07-04	4163439.0
4	Chinese	2015-07-05	4441273.0

Checking Stationarity using ADF

[ ]:
```
def adf_test(timeseries):
    print ('Results of Dickey-Fuller Test:')
    dftest = adfuller(timeseries, autolag='AIC')
    df_output = pd.Series(dftest[0:4], index=['Test Statistic','p-value','#Lags␣
 ↪Used','Number of Observations Used'])
    for key, value in dftest[4].items():
       df_output['Critical Value (%s)' %key] = round(value,2)
    print (df_output)
```

[ ]:
```
adf_test(lang_data[lang_data['language'] == 'English']['Visits'])
```

```
Results of Dickey-Fuller Test:
Test Statistic                  -2.373583
p-value                          0.149332
#Lags Used                      14.000000
Number of Observations Used    535.000000
Critical Value (1%)             -3.440000
Critical Value (5%)             -2.870000
```

```
Critical Value (10%)            -2.570000
dtype: float64
```

The test statistic > critical value / p_value > 5%. This implies that the series is not stationary.

Decomposing Time Series

Time Series Decomposition

Time series decomposition is a statistical technique used to break down a time series into its constituent components in order to understand its underlying structure, trends, seasonality, and irregular fluctuations. The decomposition typically involves separating the time series data into three main components:

- Trend ((T_t)): The long-term movement or pattern in the data, representing the overall direction in which the time series is moving.
- Seasonality ((S_t)): The repeating patterns or fluctuations that occur at regular intervals within the time series data.
- Residuals ((R_t)): The remaining variation in the data after removing the trend and seasonality components.

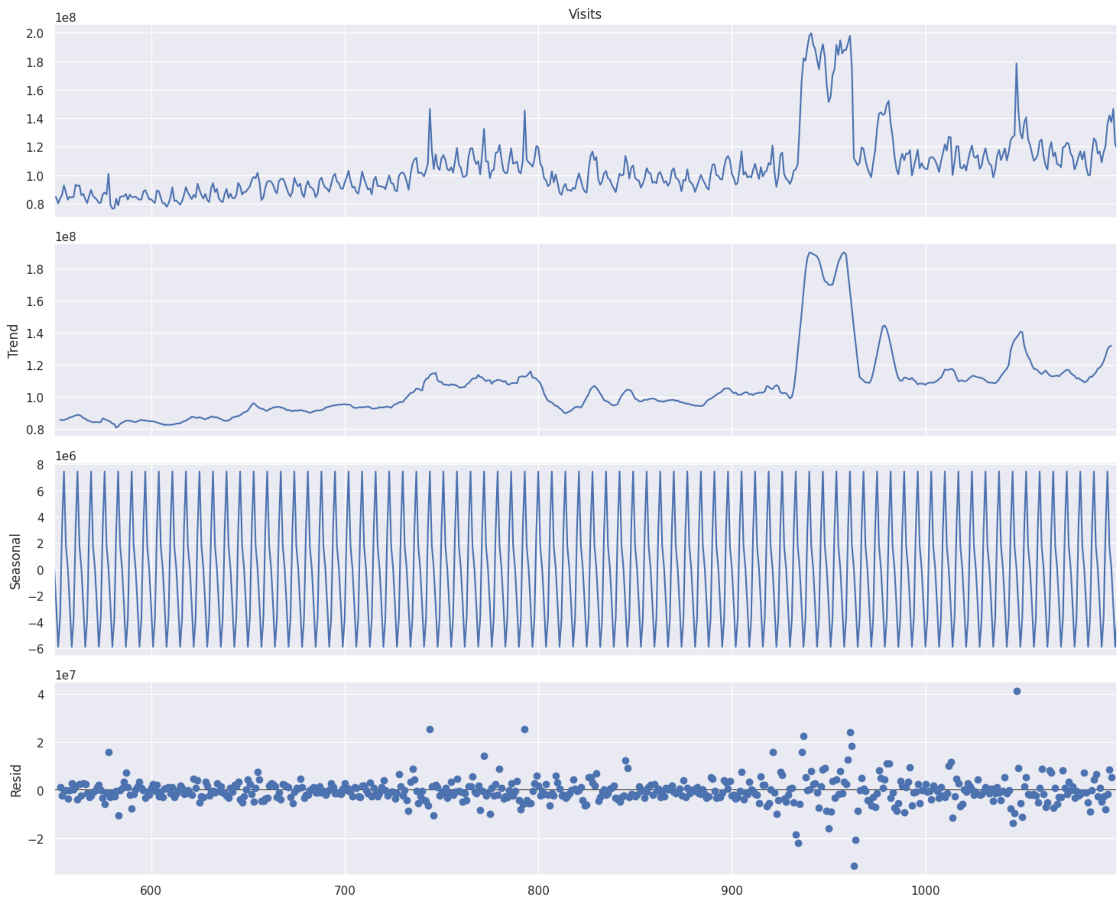
The time series (y_t) can be decomposed into its components as follows:

- Additive Decomposition: [y_t = T_t + S_t + R_t]
- Multiplicative Decomposition: [y_t = T_t $\times$S_t $\times$R_t]

Various techniques such as moving averages, exponential smoothing, or mathematical models can be used to estimate the trend and seasonal components, leaving the residual component as the leftover variation in the data.

[ ]:
```
ts_english = lang_data[lang_data['language'] == 'English']['Visits']
```

[ ]:
```
decomposition = seasonal_decompose(ts_english, model='additive', period=7)

fig = decomposition.plot()
fig.set_size_inches((15, 12))
fig.tight_layout()
plt.show()
```




```
[ ]: residual = pd.DataFrame(decomposition.resid.fillna(0).values)
     adf_test(residual)
```

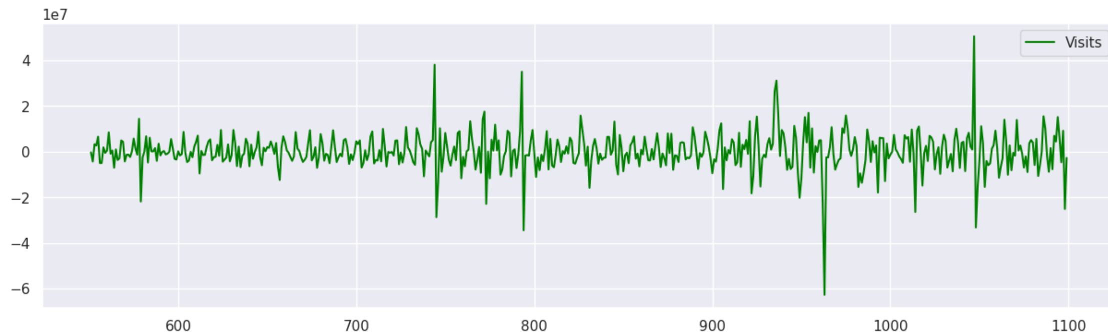
```
Results of Dickey-Fuller Test:
Test Statistic                -1.152195e+01
p-value                        4.020129e-21
#Lags Used                     1.700000e+01
Number of Observations Used    5.320000e+02
Critical Value (1%)           -3.440000e+00
Critical Value (5%)           -2.870000e+00
Critical Value (10%)          -2.570000e+00
dtype: float64
```

Residuals from time-series decomposition are now Stationary

Estimating (p,q,d) & Interpreting ACF and PACF plots

```
[ ]: ts_diff = pd.DataFrame(ts_english).diff(1)
     ts_diff.dropna(inplace = True)
```

[ ]:
```
ts_diff.plot(color = 'green', figsize=(15, 4))
plt.show()
```

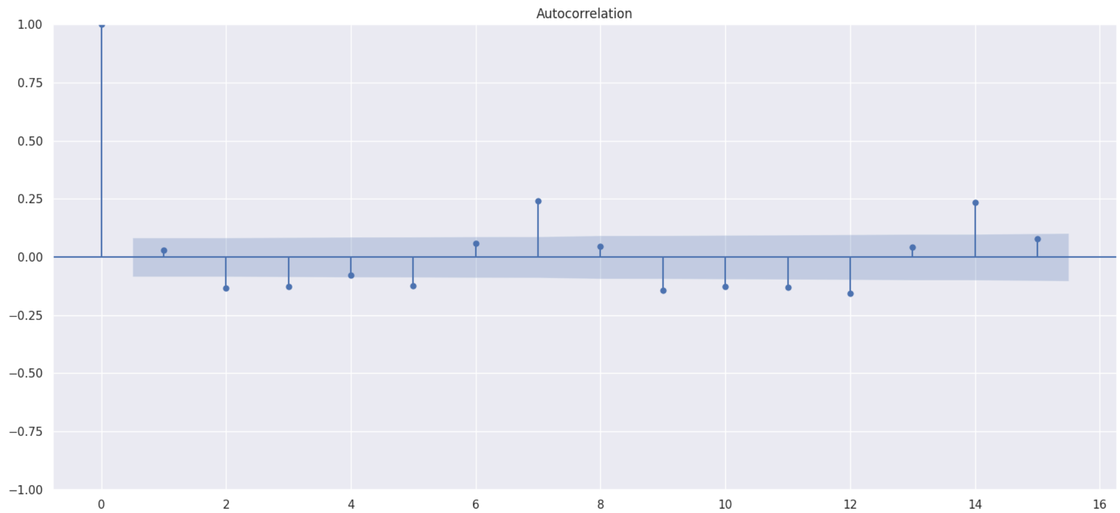


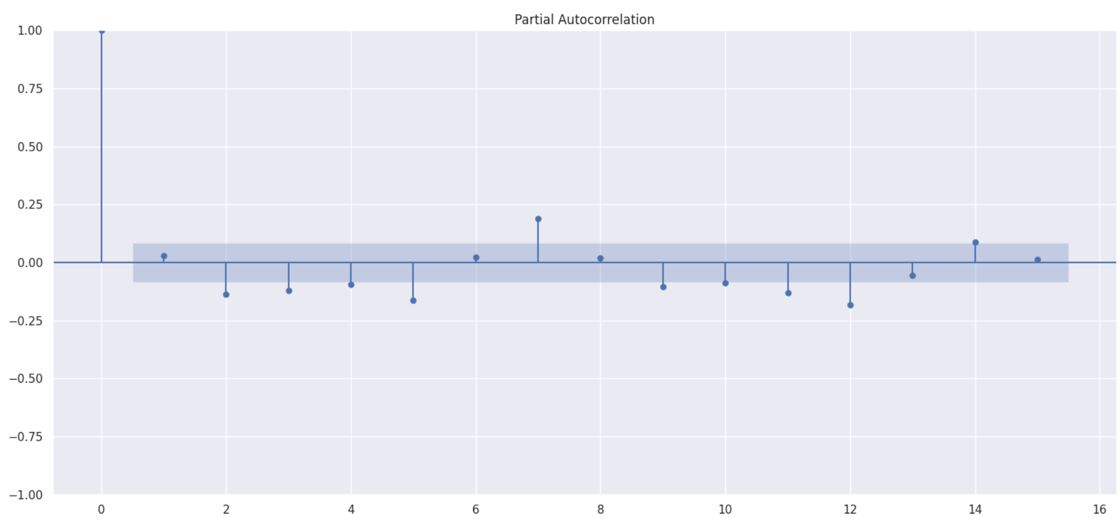
[ ]:
```
adf_test(ts_diff)
```

```
Results of Dickey-Fuller Test:
Test Statistic                -8.273643e+00
p-value                        4.719811e-13
#Lags Used                     1.300000e+01
Number of Observations Used    5.350000e+02
Critical Value (1%)           -3.440000e+00
Critical Value (5%)           -2.870000e+00
Critical Value (10%)          -2.570000e+00
dtype: float64
```

We are getting a stationary time series after a differentiation of 1. d can therefore be 1.

[ ]:
```
acf = plot_acf(ts_diff, lags= 15)
acf.tight_layout()
pacf = plot_pacf(ts_diff, lags= 15)
pacf.tight_layout()
```





###ACF###

- If the ACF shows a sharp cutoff after lag ‘k’, it suggests that an AR(k) model may be appropriate.
- If the ACF decreases gradually, it suggests a non-stationary series, and differencing (d) may be needed.
- If the ACF has a sinusoidal pattern or fluctuates around zero, it suggests a seasonal component.
- The ACF shows a sharp cutoff after lag 0, it suggests that an AR(0) model may be appropriate.

###PACF###

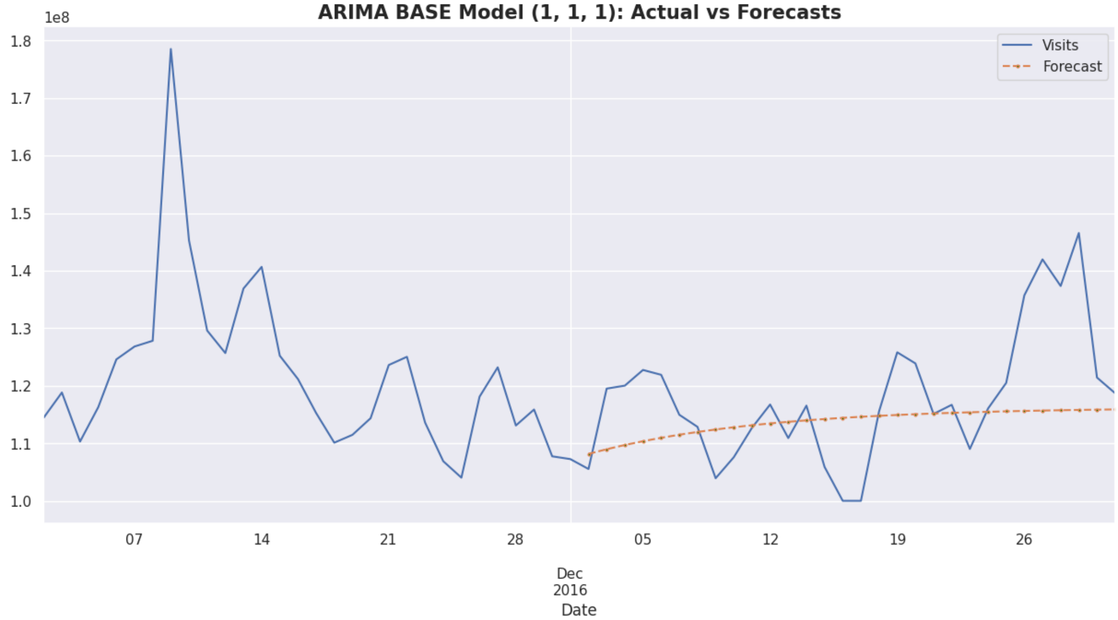
- If the PACF has a sharp cutoff after lag ‘k’, it suggests an MA(k) model may be appropriate.
- If the PACF gradually decreases, it suggests an AR component.

- If there are significant spikes at seasonal lags, it suggests a seasonal AR or MA component.
- **The PACF has a sharp cutoff after lag 0, it suggests an MA(0) model may be appropriate.**

[ ]:
```
ts_english = lang_data[lang_data.language == 'English'][['Date', 'Visits']]
ts_english.set_index('Date', drop=True, inplace=True)
```

[ ]:
```
def arima_model(n, order, time_series):
    model = ARIMA(time_series[:-n], order=order)
    model_fit = model.fit()
    forecast = model_fit.forecast(steps=n, alpha=0.05)
    time_series.index = pd.to_datetime(time_series.index)
    forecast.index = pd.to_datetime(forecast.index)
    time_series[-60:].plot(label='Actual')
    forecast.plot(label='Forecast', linestyle='dashed', marker='o',␣
 ↪markerfacecolor='green', markersize=2)
    plt.legend(loc="upper right")
    plt.title(f'ARIMA BASE Model {order}: Actual vs Forecasts', fontsize=15,␣
 ↪fontweight='bold')
    plt.show()


    actuals = time_series.values[-n:]
    errors = time_series.values[-n:] - forecast.values

    mape = np.mean(np.abs(errors) / np.abs(actuals))
    rmse = np.sqrt(np.mean(errors**2))

    # Print MAPE & RMSE
    print('-' * 80)
    print(f'MAPE of Model: {np.round(mape, 5)}')
    print('-' * 80)
    print(f'RMSE of Model: {np.round(rmse, 3)}')
    print('-' * 80)
```

[ ]:
```
arima_model(30, (1,1,1), ts_english)
```




```
--------------------------------------------------------------------------------
MAPE of Model: 0.0723
--------------------------------------------------------------------------------
RMSE of Model: 12072210.287
--------------------------------------------------------------------------------
```

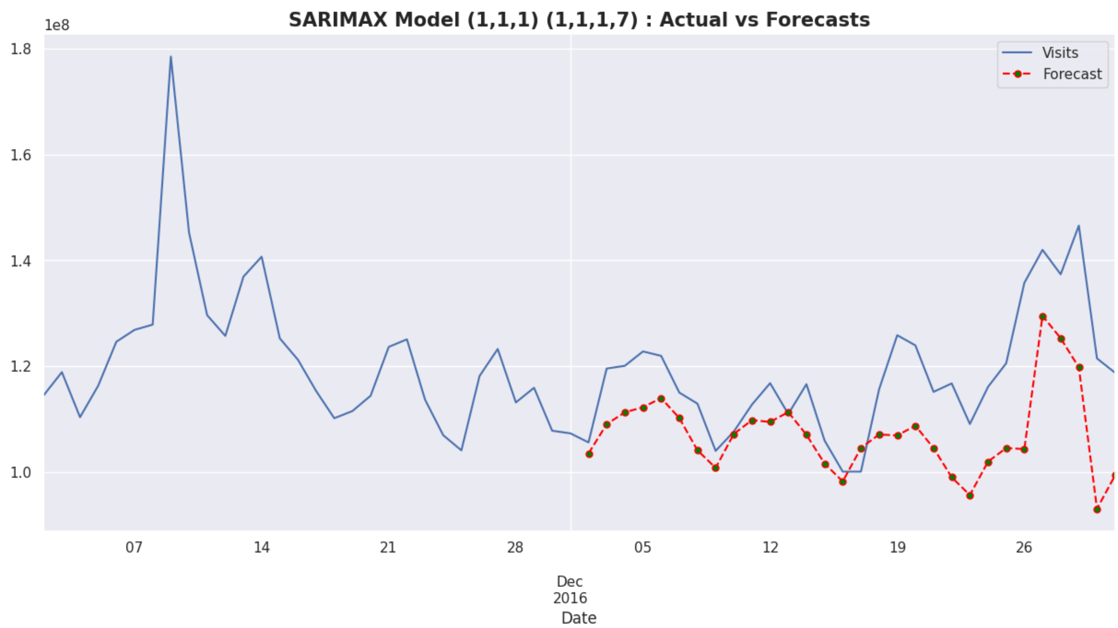
[ ]:
```
def sarimax_model(time_series, n, p=0, d=0, q=0, P=0, D=0, Q=0, s=0, exog = []):

    #Creating SARIMAX Model with order(p,d,q) & seasonal_order=(P, D, Q, s)
    model = SARIMAX(time_series[:-n],
                    order=(p, d, q),
                    seasonal_order=(P, D, Q, s),
                    exog=exog[:-n],
                    initialization='approximate_diffuse')
    model_fit = model.fit()

    # Forecasting last n-values
    model_forecast = model_fit.forecast(n, dynamic=True, exog=pd.
 ↪DataFrame(exog[-n:]))

    # Plotting Actual & Forecasted values
    plt.figure(figsize=(20, 8))
    time_series[-60:].plot(label='Actual')
    model_forecast[-60:].plot(label='Forecast', color='red',
                              linestyle='dashed', marker='o',␣
 ↪markerfacecolor='green', markersize=5)
```

```
    plt.legend(loc="upper right")
    plt.title(f'SARIMAX Model ({p},{d},{q}) ({P},{D},{Q},{s}) : Actual vs␣
↪Forecasts', fontsize=15, fontweight='bold')
    plt.show()

    # Calculating MAPE & RMSE
    actuals = time_series.values[-n:]
    errors = time_series.values[-n:] - model_forecast.values

    mape = np.mean(np.abs(errors) / np.abs(actuals))
    rmse = np.sqrt(np.mean(errors ** 2))

    # Printing metrics
    print('-' * 80)
    print(f'MAPE of Model : {np.round(mape, 5)}')
    print('-' * 80)
    print(f'RMSE of Model : {np.round(rmse, 3)}')
    print('-' * 80)
```

[ ]:
```
exog = exog['Exog'].to_numpy()
```

[ ]:
```
time_series = ts_english
test_size= 0.1
p,d,q, P,D,Q,s = 1,1,1,1,1,1,7
n = 30
sarimax_model(time_series, n, p = p, d=d,q=q,  P=P, D=D, Q=Q, s=s, exog = exog)
```

```
<Figure size 2000x800 with 0 Axes>
```



```
--------------------------------------------------------------------------------
MAPE of Model : 0.11207
--------------------------------------------------------------------------------
RMSE of Model : 17324743.861
--------------------------------------------------------------------------------
```

Sarimax algorithm is giving us less than 12 % MAPE.

###Grid Search###

```
[ ]: def sarimax_grid_search(time_series, n, param, d_param, s_param, exog=[]):
         # Creating df for storing results summary
         param_df = pd.DataFrame(columns=['serial', 'pdq', 'PDQs', 'mape', 'rmse'])

         # Generate all parameter combinations
         param_combinations = product(param, d_param, param, param, d_param, param,␣
      ↪s_param)

         # Counter for keeping track of iterations
         counter = 0

         for p, d, q, P, D, Q, s in param_combinations:
             model = SARIMAX(time_series[:-n],
                             order=(p, d, q),
                             seasonal_order=(P, D, Q, s),
                             exog=exog[:-n],
                             initialization='approximate_diffuse')
             model_fit = model.fit()

             model_forecast = model_fit.forecast(n, dynamic=True, exog=pd.
      ↪DataFrame(exog[-n:]))

             actuals = time_series.values[-n:]
             errors = time_series.values[-n:] - model_forecast.values

             mape = np.mean(np.abs(errors) / np.abs(actuals))
             rmse = np.sqrt(np.mean(errors**2))
             mape = np.round(mape, 5)
             rmse = np.round(rmse, 3)

             counter += 1
             list_row = [counter, (p, d, q), (P, D, Q, s), mape, rmse]
             param_df.loc[len(param_df)] = list_row

             # Print statement to check progress of Loop
```

```
        print(f'Possible Combination: {counter} out of {len(param)**4 *␣
↪len(s_param) * len(d_param)**2} calculated')

    return param_df
```

[ ]:
```
time_series = ts_english
n = 30
param = [0,1,2]
d_param = [0,1]
s_param = [7]

english_params  = sarimax_grid_search(time_series, n, param,␣
↪d_param,s_param,exog)
```

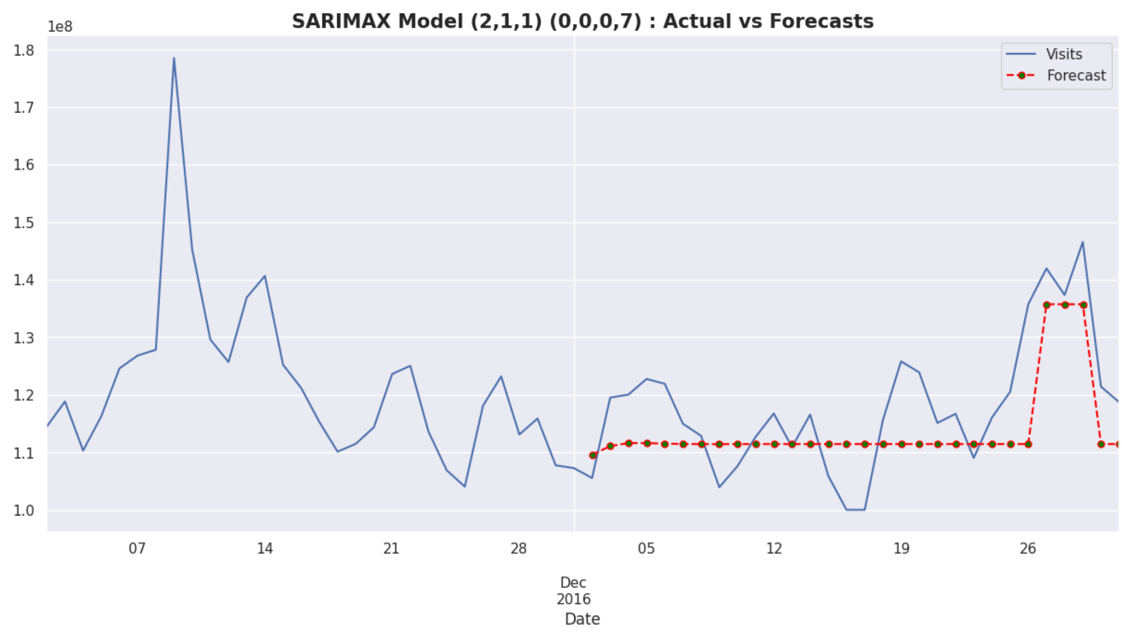
[ ]:
```
english_params.sort_values(['mape', 'rmse']).head()
```

[ ]:

	serial	pdq	PDQs	mape	rmse
288	289	(2, 1, 1)	(0, 0, 0, 7)	0.08738	1.390990e+07
289	290	(2, 1, 1)	(0, 0, 1, 7)	0.08903	1.411112e+07
290	291	(2, 1, 1)	(0, 0, 2, 7)	0.08988	1.421023e+07
294	295	(2, 1, 1)	(1, 0, 0, 7)	0.09013	1.423613e+07
300	301	(2, 1, 1)	(2, 0, 0, 7)	0.09273	1.456684e+07

[ ]:
```
time_series = ts_english
p,d,q, P,D,Q,s = 2,1,1, 0,0,0,7
n = 30
sarimax_model(time_series, n, p=p, d=d, q=q, P=P, D=D, Q=Q, s=s, exog = exog)
```

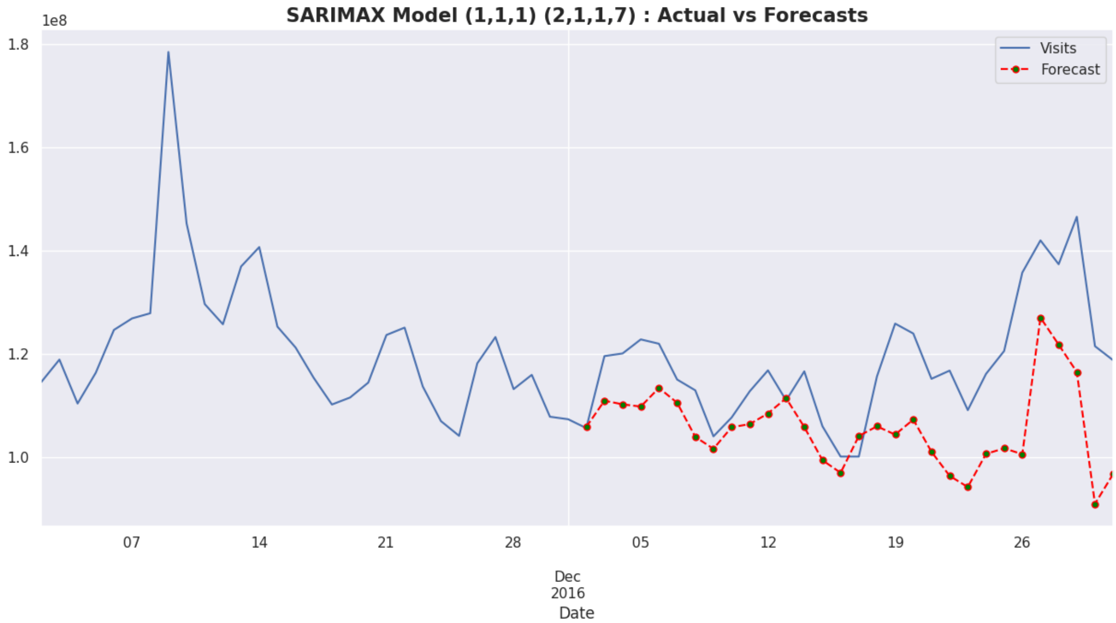
<Figure size 2000x800 with 0 Axes>



```
--------------------------------------------------------------------------------
MAPE of Model : 0.08738
--------------------------------------------------------------------------------
RMSE of Model : 13909895.954
--------------------------------------------------------------------------------
```

[ ]:
```
time_series = ts_english
p,d,q, P,D,Q,s = 1,1,1, 2,1,1,7
n = 30
sarimax_model(time_series, n, p=p, d=d, q=q, P=P, D=D, Q=Q, s=s, exog = exog)
```

```
<Figure size 2000x800 with 0 Axes>
```




```
--------------------------------------------------------------------------------
MAPE of Model : 0.1187
--------------------------------------------------------------------------------
RMSE of Model : 18266240.798
--------------------------------------------------------------------------------
```

[ ]:
```
def pipeline_sarimax_grid_search_without_exog(languages, data_language, n,␣
↪param, d_param, s_param):

    best_param_df = pd.DataFrame(columns=['language', 'p', 'd', 'q', 'P', 'D',␣
↪'Q', 's', 'mape'])
```

```python
for lang in languages:
    print(f'--------------------------------------------------------------')
    print(f'          Finding best parameters for {lang}                  ')
    print(f'--------------------------------------------------------------')

    time_series = data_language[data_language['language'] == lang][['Date', 'Visits']]
    time_series.set_index('Date', drop=True, inplace=True)
    best_mape = 100

    counter = 0
    param_combinations = product(param, d_param, param, param, d_param, param, s_param)

    for p, d, q, P, D, Q, s in param_combinations:
        model = SARIMAX(time_series[:-n],
                        order=(p, d, q),
                        seasonal_order=(P, D, Q, s),
                        initialization='approximate_diffuse')
        model_fit = model.fit()
        model_forecast = model_fit.forecast(n, dynamic=True)

        actuals = time_series.values[-n:]
        errors = time_series.values[-n:] - model_forecast.values
        mape = np.mean(np.abs(errors) / np.abs(actuals))

        counter += 1
        if mape < best_mape:
            best_mape = mape
            best_p, best_d, best_q = p, d, q
            best_P, best_D, best_Q = P, D, Q
            best_s = s

        print(f'Possible Combination: {counter} out of {(len(param)**4)*len(s_param)*(len(d_param)**2)} calculated')

    best_mape = np.round(best_mape, 5)
    print(f'--------------------------------------------------------------')
    print(f'Minimum MAPE for {lang} = {best_mape}')
    print(f'Corresponding Best Parameters are {best_p, best_d, best_q, best_P, best_D, best_Q, best_s}')
    print(f'--------------------------------------------------------------')

    best_param_row = [lang, best_p, best_d, best_q, best_P, best_D, best_Q, best_s, best_mape]
    best_param_df.loc[len(best_param_df)] = best_param_row
```

```
    return best_param_df
```

[ ]:
```
languages = ['Chinese', 'French', 'German', 'Japenese', 'Russian', 'Spanish']
n = 30
param = [0,1,2]
d_param = [0,1]
s_param = [7]


best_param_df = pipeline_sarimax_grid_search_without_exog(languages, lang_data,␣
↪n, param, d_param, s_param)
```

[ ]:
```
best_param_df.sort_values(['mape'], inplace = True)
best_param_df
```

[ ]:

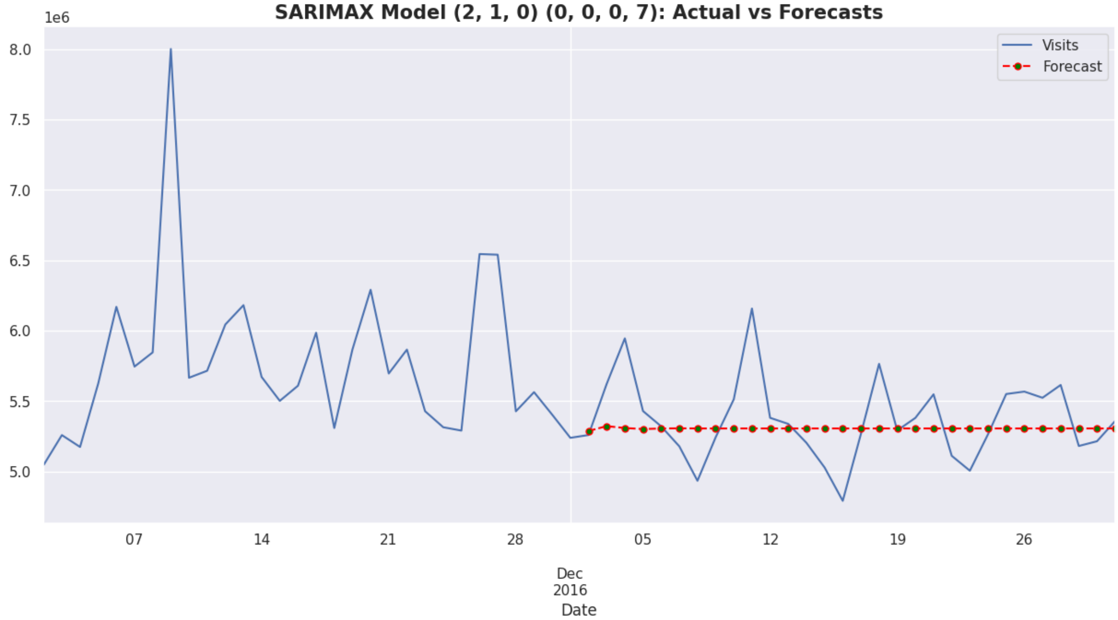
	language	p	d	q	P	D	Q	s	mape
0	Chinese	2	1	0	0	0	0	7	0.03932
4	Russian	0	0	2	2	0	2	7	0.06417
1	French	2	1	2	0	0	0	7	0.08205
3	Japenese	2	0	2	0	1	2	7	0.09184
5	Spanish	2	0	0	0	0	0	7	0.15102
2	German	0	1	2	0	0	0	7	0.16889

[ ]:
```
def plot_best_SARIMAX_model(languages, data_language, n, best_param_df):
    for lang in languages:
        # Fetching respective best parameters for that language
        params_lang = best_param_df[best_param_df['language'] == lang].iloc[0]
        p, d, q, P, D, Q, s = params_lang[['p', 'd', 'q', 'P', 'D', 'Q', 's']]

        # Creating language time-series
        time_series = data_language[data_language['language'] == lang][['Date',␣
↪'Visits']]
        time_series.set_index('Date', drop=True, inplace=True)

        # Creating SARIMAX Model
        model = SARIMAX(time_series[:-n], order=(p, d, q),
                        seasonal_order=(P, D, Q, s),␣
↪initialization='approximate_diffuse')
        model_fit = model.fit()

        # Creating forecast for last n-values
        model_forecast = model_fit.forecast(n, dynamic=True)

        # Calculating MAPE & RMSE
        actuals = time_series.values[-n:]
        errors = time_series.values[-n:] - model_forecast.values
```

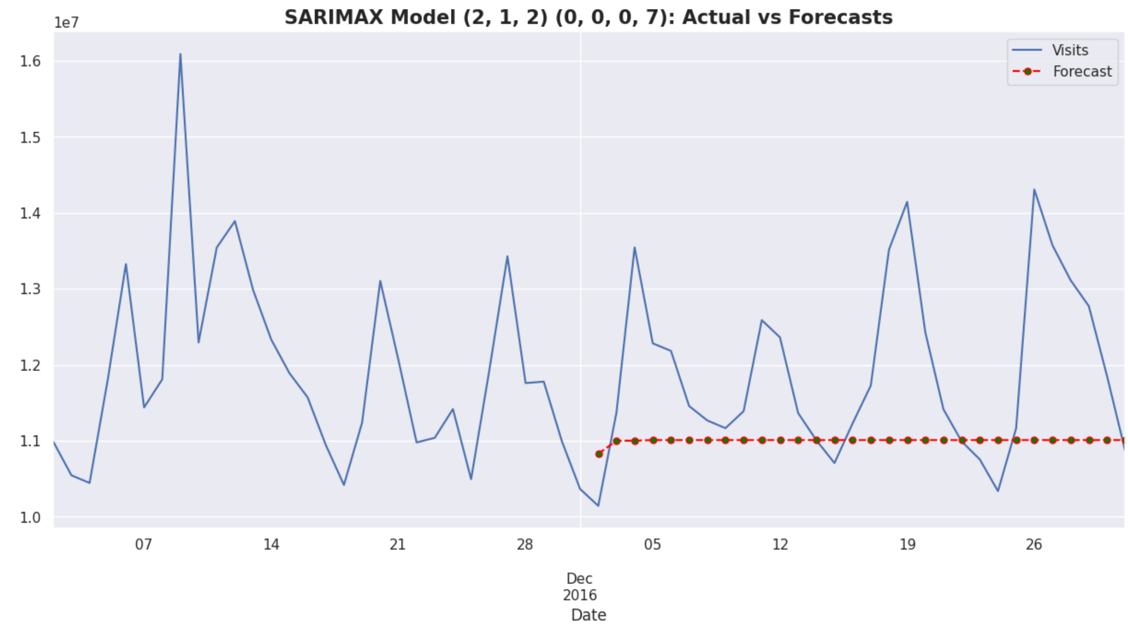
```
        mape = np.mean(np.abs(errors) / np.abs(actuals))
        rmse = np.sqrt(np.mean(errors**2))

        # Printing model statistics
        print(f'\n{"-" * 90}')
        print(f'SARIMAX model for {lang} Time Series')
        print(f'Parameters of Model: ({p}, {d}, {q}) ({P}, {D}, {Q}, {s})')
        print(f'MAPE of Model: {np.round(mape, 5)}')
        print(f'RMSE of Model: {np.round(rmse, 3)}')
        print(f'{"-" * 90}')

        # Plotting Actual & Forecasted values
        time_series.index = time_series.index.astype('datetime64[ns]')
        model_forecast.index = model_forecast.index.astype('datetime64[ns]')
        plt.figure(figsize=(20, 8))
        time_series[-60:].plot(label='Actual')
        model_forecast[-60:].plot(label='Forecast', color='red',
                                  linestyle='dashed', marker='o',␣
↪markerfacecolor='green', markersize=5)
        plt.legend(loc="upper right")
        plt.title(f'SARIMAX Model ({p}, {d}, {q}) ({P}, {D}, {Q}, {s}): Actual␣
↪vs Forecasts',
                  fontsize=15, fontweight='bold')
        plt.show()

    return 0
```

[ ]:
```
languages = ['Chinese', 'French', 'German', 'Japenese', 'Russian', 'Spanish']
n = 30
plot_best_SARIMAX_model(languages, lang_data, n, best_param_df)
```

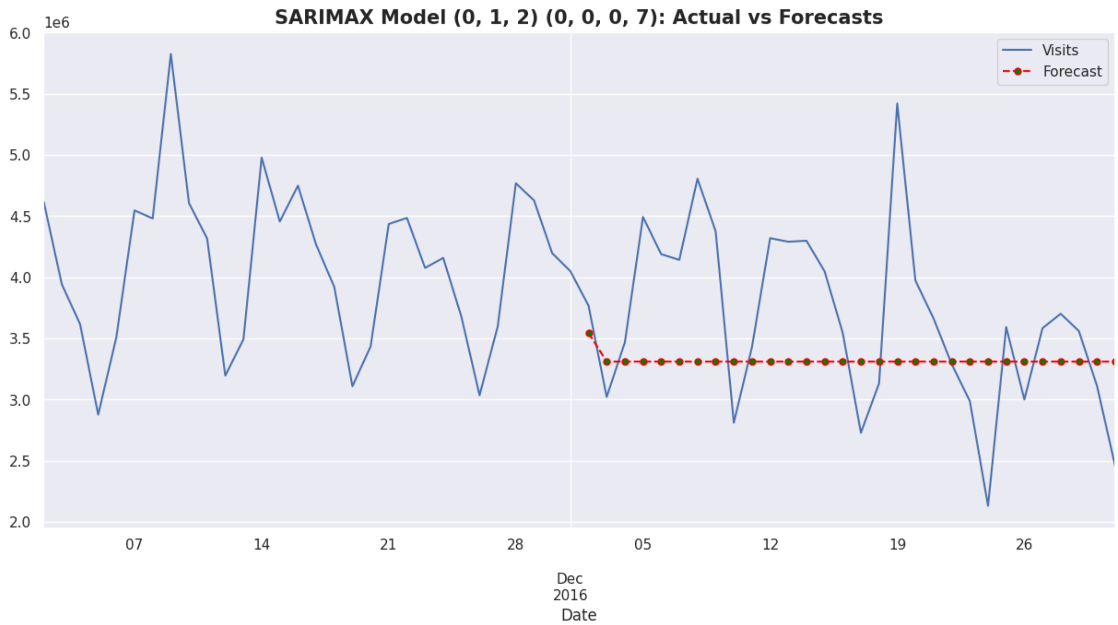
```
--------------------------------------------------------------------------------
----------
SARIMAX model for Chinese Time Series
Parameters of Model: (2, 1, 0) (0, 0, 0, 7)
MAPE of Model: 0.03932
RMSE of Model: 289943.436
--------------------------------------------------------------------------------
----------

<Figure size 2000x800 with 0 Axes>
```



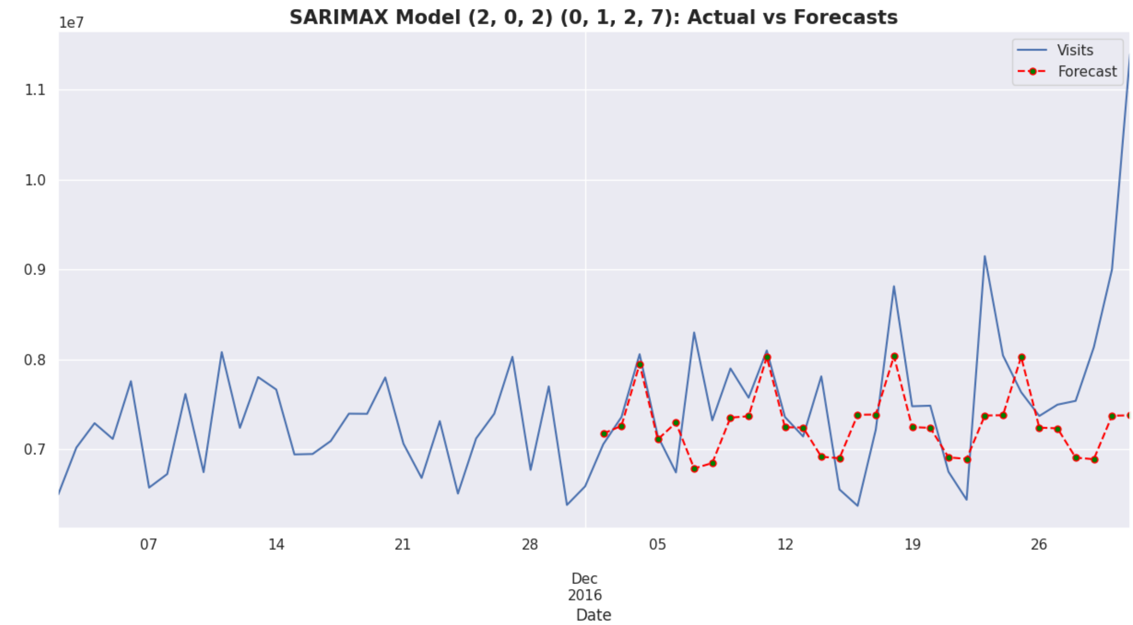

```
--------------------------------------------------------------------------------
----------
SARIMAX model for French Time Series
Parameters of Model: (2, 1, 2) (0, 0, 0, 7)
MAPE of Model: 0.08205
RMSE of Model: 1422711.185
--------------------------------------------------------------------------------
----------

<Figure size 2000x800 with 0 Axes>
```



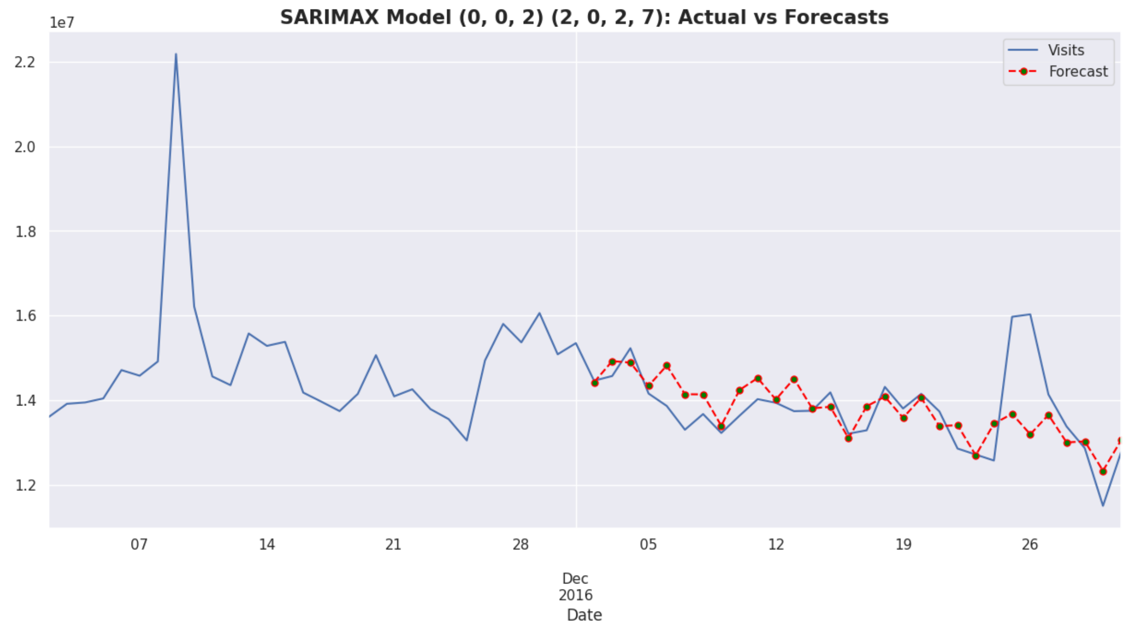

```
--------------------------------------------------------------------------------
----------
SARIMAX model for German Time Series
Parameters of Model: (0, 1, 2) (0, 0, 0, 7)
MAPE of Model: 0.16889
RMSE of Model: 782375.607
--------------------------------------------------------------------------------
----------

<Figure size 2000x800 with 0 Axes>
```



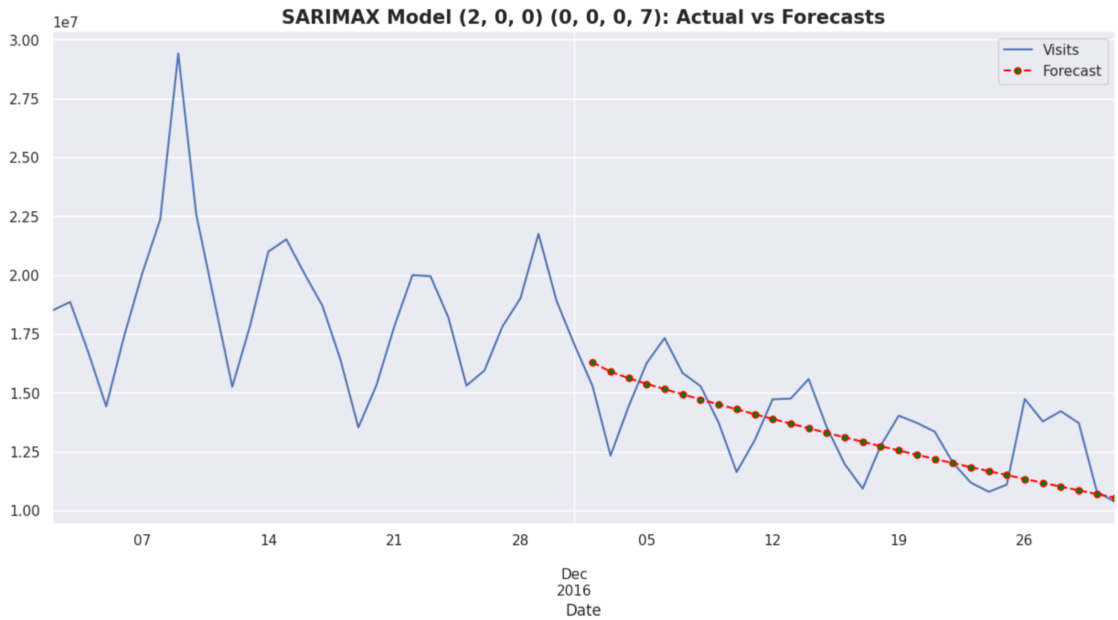

```
--------------------------------------------------------------------------------
----------
SARIMAX model for Japenese Time Series
Parameters of Model: (2, 0, 2) (0, 1, 2, 7)
MAPE of Model: 0.09184
RMSE of Model: 1107200.185
--------------------------------------------------------------------------------
----------

<Figure size 2000x800 with 0 Axes>
```




```
--------------------------------------------------------------------------------
----------
SARIMAX model for Russian Time Series
Parameters of Model: (0, 0, 2) (2, 0, 2, 7)
MAPE of Model: 0.06417
RMSE of Model: 1130800.511
--------------------------------------------------------------------------------
----------

<Figure size 2000x800 with 0 Axes>
```




```
--------------------------------------------------------------------------------
----------
SARIMAX model for Spanish Time Series
Parameters of Model: (2, 0, 0) (0, 0, 0, 7)
MAPE of Model: 0.15102
RMSE of Model: 2470108.819
--------------------------------------------------------------------------------
----------

<Figure size 2000x800 with 0 Axes>
```



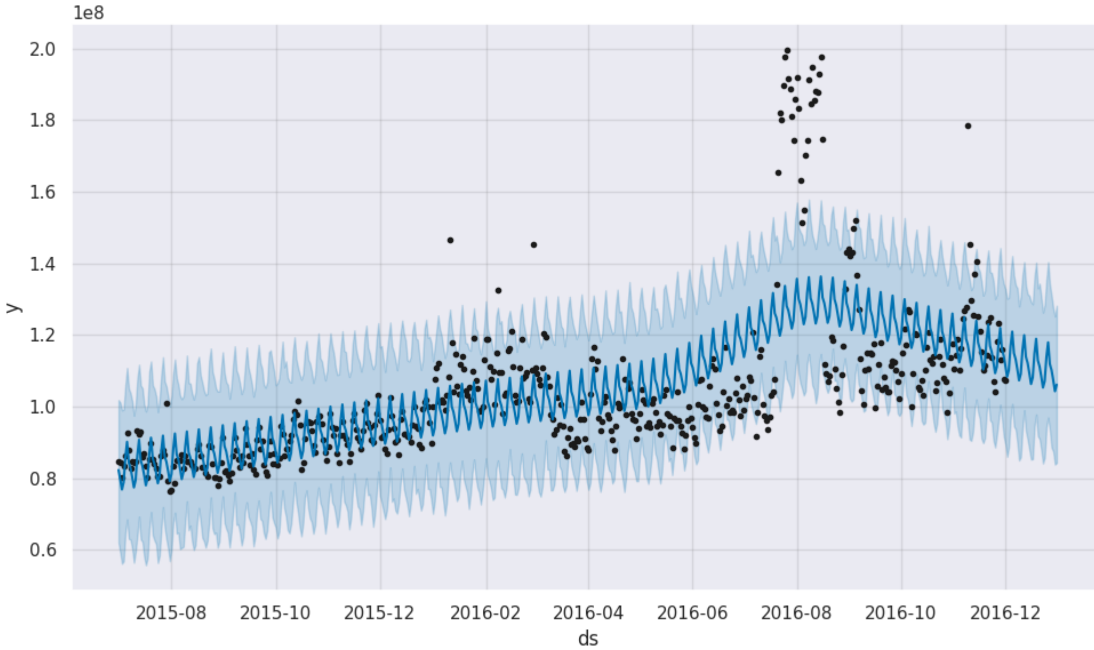
[ ]: 0

[ ]:
```
time_series = lang_data[lang_data['language'] == 'English'][['Date', 'Visits']]
# time_series.set_index('Date', drop=True, inplace=True)
time_series.columns = ['ds', 'y']
time_series['exog'] = exog
```

[ ]:
```
prophet1 = Prophet(weekly_seasonality=True)
prophet1.fit(time_series[['ds', 'y']][:-30])
future = prophet1.make_future_dataframe(periods=30, freq= 'D')
forecast = prophet1.predict(future)
fig1 = prophet1.plot(forecast)
```

```
INFO:prophet:Disabling yearly seasonality. Run prophet with
yearly_seasonality=True to override this.
INFO:prophet:Disabling daily seasonality. Run prophet with
daily_seasonality=True to override this.
DEBUG:cmdstanpy:input tempfile: /tmp/tmpsjub09sx/yxptykkz.json
DEBUG:cmdstanpy:input tempfile: /tmp/tmpsjub09sx/nzi25y8h.json
DEBUG:cmdstanpy:idx 0
DEBUG:cmdstanpy:running CmdStan, num_threads: None
DEBUG:cmdstanpy:CmdStan args: ['/usr/local/lib/python3.10/dist-
packages/prophet/stan_model/prophet_model.bin', 'random', 'seed=17945', 'data',
'file=/tmp/tmpsjub09sx/yxptykkz.json', 'init=/tmp/tmpsjub09sx/nzi25y8h.json',
'output',
'file=/tmp/tmpsjub09sx/prophet_model5v8mklue/prophet_model-20240521122150.csv',
'method=optimize', 'algorithm=lbfgs', 'iter=10000']
```

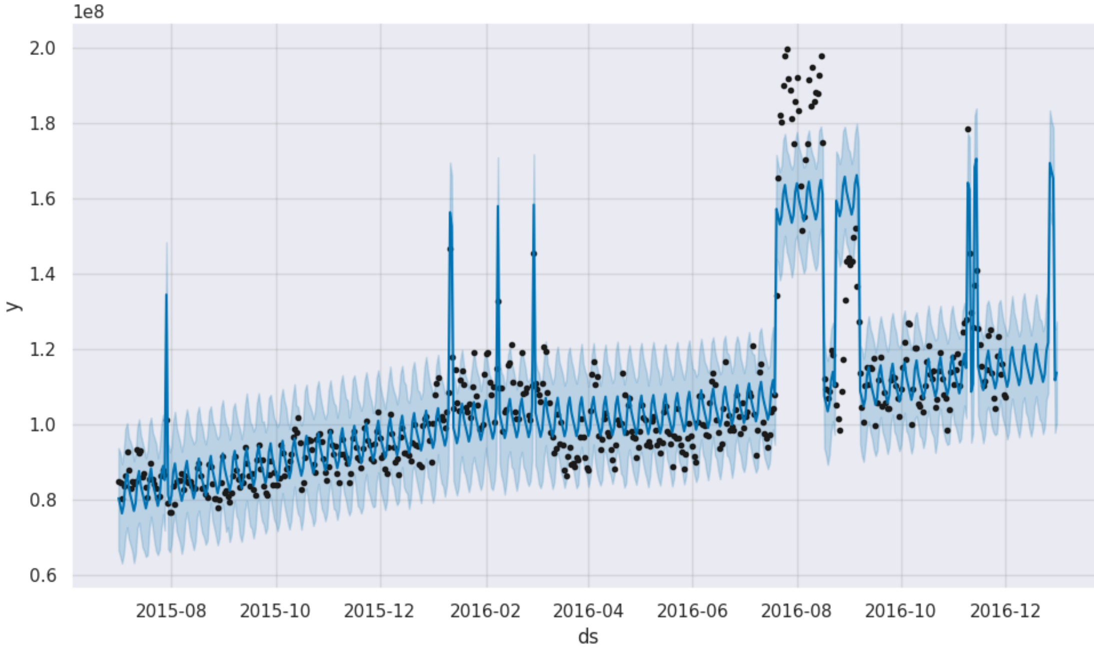
```
12:21:50 - cmdstanpy - INFO - Chain [1] start processing
INFO:cmdstanpy:Chain [1] start processing
12:21:50 - cmdstanpy - INFO - Chain [1] done processing
INFO:cmdstanpy:Chain [1] done processing
```



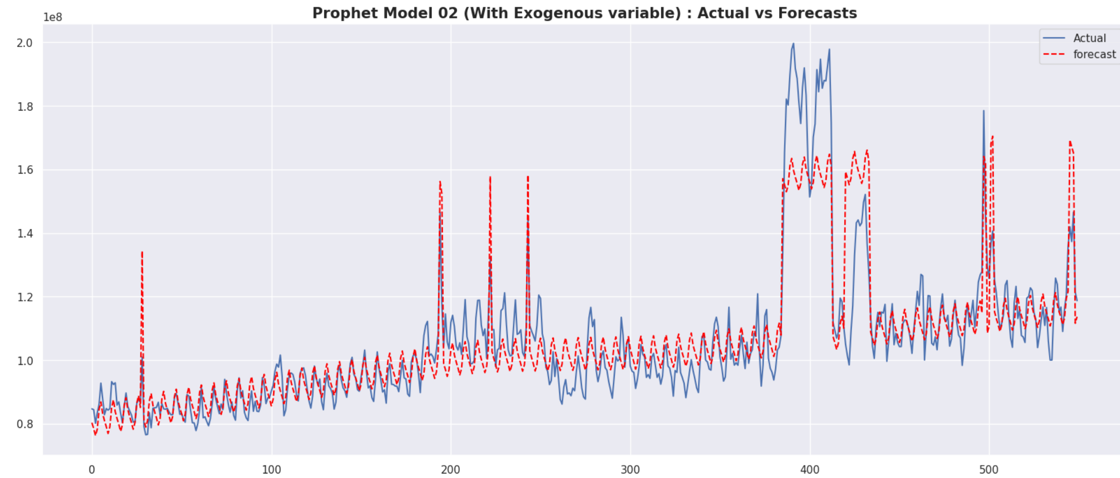
[ ]:
```
prophet2 = Prophet(weekly_seasonality=True)
prophet2.add_regressor('exog')
prophet2.fit(time_series[:-30])
#future2 = prophet2.make_future_dataframe(periods=30, freq= 'D')
forecast2 = prophet2.predict(time_series)
fig2 = prophet2.plot(forecast2)
```

```
INFO:prophet:Disabling yearly seasonality. Run prophet with
yearly_seasonality=True to override this.
INFO:prophet:Disabling daily seasonality. Run prophet with
daily_seasonality=True to override this.
DEBUG:cmdstanpy:input tempfile: /tmp/tmpsjub09sx/9lo6s_dg.json
DEBUG:cmdstanpy:input tempfile: /tmp/tmpsjub09sx/cf5srown.json
DEBUG:cmdstanpy:idx 0
DEBUG:cmdstanpy:running CmdStan, num_threads: None
DEBUG:cmdstanpy:CmdStan args: ['/usr/local/lib/python3.10/dist-
packages/prophet/stan_model/prophet_model.bin', 'random', 'seed=8198', 'data',
'file=/tmp/tmpsjub09sx/9lo6s_dg.json', 'init=/tmp/tmpsjub09sx/cf5srown.json',
'output',
'file=/tmp/tmpsjub09sx/prophet_modelkd6qv4qa/prophet_model-20240521122151.csv',
'method=optimize', 'algorithm=lbfgs', 'iter=10000']
```

```
12:21:51 - cmdstanpy - INFO - Chain [1] start processing
INFO:cmdstanpy:Chain [1] start processing
12:21:51 - cmdstanpy - INFO - Chain [1] done processing
INFO:cmdstanpy:Chain [1] done processing
```



[ ]:
```
actual = time_series['y'].values
forecast = forecast2['yhat'].values

plt.figure(figsize = (20,8))
plt.plot(actual, label = 'Actual')
plt.plot(forecast, label = 'forecast', color = 'red', linestyle='dashed')
plt.legend(loc="upper right")
plt.title(f'Prophet Model 02 (With Exogenous variable) : Actual vs Forecasts',
  fontsize = 15, fontweight = 'bold')
plt.show()
```




```
[ ]: errors = abs(actual - forecast)
mape = np.mean(errors/abs(actual))
mape
```

```
[ ]: 0.05955846978627715
```

FB Prophet Model is able to capture peaks because of exogenous variable and is giving a MAPE of 6%

###Recommendations###

- Prioritize English-language pages due to their low MAPE and high mean visits, making them optimal for advertising efforts to maximize reach and effectiveness.
- Avoid advertising on Chinese-language pages unless there's a specific marketing strategy tailored for Chinese populations, as they have the lowest number of visits.
- Russian-language pages present a promising opportunity for high conversion rates with their decent number of visits and low MAPE, if utilized effectively.
- Despite having the second-highest number of visits, Spanish-language pages exhibit the highest MAPE, suggesting that advertisements on these pages may not effectively reach the intended audience.
- French, German, and Japanese-language pages show moderate levels of visits and MAPE. Depending on the target customers, consider advertising campaigns on these pages to capitalize on their potential reach and conversion rates.

###Questionnaire###

Defining the problem statements and where can this and modifications of this be used?

- The Data Science team at Ad ease aims to analyze per page view reports for various Wikipedia pages spanning 550 days.
- The objective includes forecasting page views to enhance ad placement optimization for clients.

- Dataset encompasses 145k Wikipedia pages with daily view counts.
- Client base extends across diverse regions, necessitating insights into ad performance across different languages.

Importance of forecasting model:

Identification of the problem and its applications:

- Implementing a robust forecasting model is pivotal in predicting fluctuations in page visits.
- This model aids the business team in optimizing marketing expenditure.
- Precise prediction of high-traffic days enables strategic ad placement, maximizing audience reach while optimizing spending.

Write 3 inferences you made from the data visualizations.

- **Linguistic Diversity:**
- The data reveals the presence of 7 languages, with English dominating, followed by Japanese, German, and French.
- Access Type Distribution: Three access types are identified—All-access, mobile-web, and desktop—comprising 51.4%, 24.9%, and 23.6% respectively.
- Access-Origin Insights: The dataset illustrates two access origins—'all-agents' and 'spider'—with 'all-agents' constituting 75.8% and 'spider' 24.2% of the data.
- **Advertising Strategies:**
- **Inferences from Data Visualizations:**
- English Language Dominance: English emerges as the most prominent language, suggesting prioritized advertisement placement due to its low Mean Absolute Percentage Error (MAPE) and high mean visit count.
- Chinese Language Considerations: Pages in Chinese exhibit the lowest visit counts, signaling caution in advertisement allocation unless specifically targeting Chinese demographics.
- Russian Language Potential: Russian language pages demonstrate a favorable balance between visit count and MAPE, indicating potential for maximum conversion if utilized effectively.
- Spanish Language Challenges: Despite being the second-highest in visit count, Spanish pages exhibit the highest MAPE, suggesting potential challenges in advertisement efficacy.
- Moderate Performers: French, German, and Japanese languages present medium-level visit counts and MAPE levels, prompting tailored advertisement strategies based on target customer demographics.

###Time Series Decomposition###

- What does the decomposition of series do? Time series decomposition is a statistical technique used to break down a time series into its constituent components in order to understand its underlying structure, trends, seasonality, and irregular fluctuations. The decomposition typically involves separating the time series data into three main components:
- Trend ((T_t)): The long-term movement or pattern in the data, representing the overall direction in which the time series is moving.

- Seasonality ((S_t)): The repeating patterns or fluctuations that occur at regular intervals within the time series data.
- Residuals ((R_t)): The remaining variation in the data after removing the trend and seasonality components.

The time series (y_t) can be decomposed into its components as follows:

- Additive Decomposition: [y_t = T_t + S_t + R_t]
- Multiplicative Decomposition: [y_t = T_t ×S_t ×R_t]

Various techniques such as moving averages, exponential smoothing, or mathematical models can be used to estimate the trend and seasonal components, leaving the residual component as the leftover variation in the data.

What level of differencing gave you a stationary series?

- First order differencing

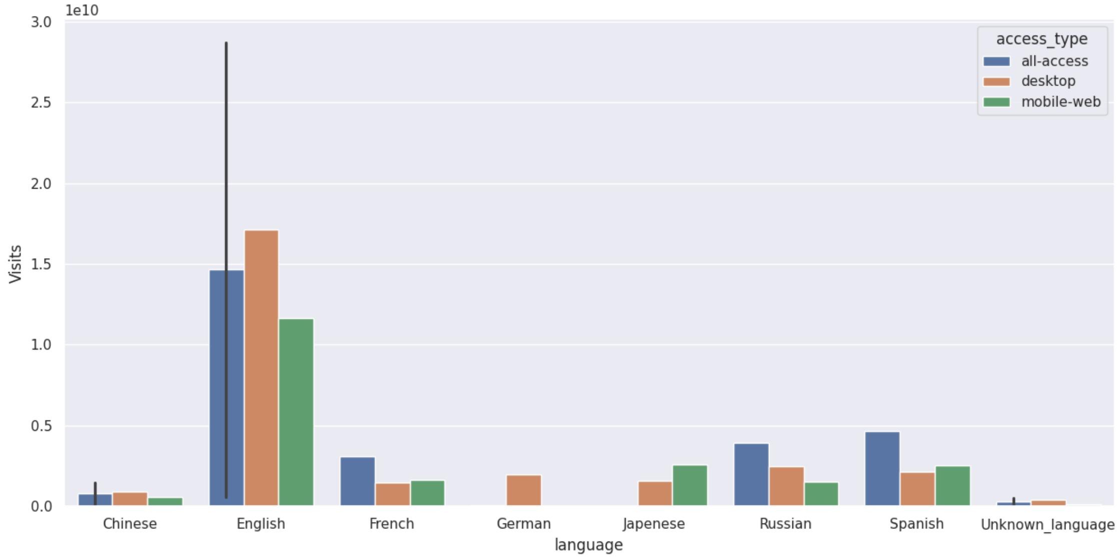
Difference between arima, sarima & sarimax.

- ARIMA is a time series forecasting model that combines autoregression (AR), differencing (I), and moving average (MA) components.
- It's suitable for univariate time series data without exogenous variables.
- ARIMA(p,d,q) where p represents the autoregressive order, d represents the differencing order, and q represents the moving average order. SARIMA is an extension of ARIMA that incorporates seasonal components in addition to the non-seasonal ones.
- It's suitable for time series data with seasonal patterns. SARIMA(p,d,q)(P,D,Q)m where P, D, and Q represent the seasonal autoregressive, differencing, and moving average orders respectively, and 'm' represents the seasonal period.
- SARIMAX (Seasonal Autoregressive Integrated Moving Average with Exogenous Variables):
- SARIMA (Seasonal Autoregressive Integrated Moving Average):
- ARIMA (Autoregressive Integrated Moving Average):
- SARIMAX extends SARIMA by allowing the inclusion of exogenous variables, which are external factors that can influence the time series.
- It's suitable for time series data with both seasonal patterns and external variables.
- SARIMAX(p,d,q)(P,D,Q)m with exogenous variables.
- These models are commonly used in time series analysis and forecasting tasks, each offering different capabilities to handle various types of data and patterns.

Compare the number of views in different languages

[ ]:
```
grouped = reshaped.groupby(['language','access_type','access_origin'],␣
↪as_index=False)['Visits'].sum()
sns.barplot(grouped, x="language", y="Visits", hue="access_type")
```

[ ]: <Axes: xlabel='language', ylabel='Visits'>



What other methods other than grid search would be suitable to get the model for all languages?

- We can use packages like hyperopt, optuna and sci-kit-optimize
- We can try and use different models like tsmixer and deep learning models