

Problem Statement

The objective of this case study is to forecast the number of daily Wikipedia page views for various Wikipedia pages across different languages over a period of 550 days. This forecasting will help Ad Ease optimize ad placement for their clients by understanding potential ad performance on pages in different languages.

▼ Observe data

Subtask:

Load the `train_1.csv` dataset and use functions like `.shape`, `.info()`, and `.describe()` to understand its structure, data types, and basic statistics.

```
import pandas as pd

# Load the train_1.csv file into a pandas DataFrame
df_train = pd.read_csv('/content/train_1.csv')

# Display the shape of the DataFrame
print("Shape of the DataFrame:")
print(df_train.shape)
print("\n")

# Print information about the DataFrame
print("Information about the DataFrame:")
df_train.info()
print("\n")

# Generate descriptive statistics for the numerical columns
print("Descriptive statistics of the DataFrame:")
display(df_train.describe())

Shape of the DataFrame:
(145063, 551)

Information about the DataFrame:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 145063 entries, 0 to 145062
Columns: 551 entries, Page to 2016-12-31
dtypes: float64(550), object(1)
memory usage: 609.8+ MB

Descriptive statistics of the DataFrame:
   2015-07-01  2015-07-02  2015-07-03  2015-07-04  2015-07-05  2015-07-06  2015-07-07  2015-07-08  ...
count  1.243230e+05  1.242470e+05  1.245190e+05  1.244090e+05  1.244040e+05  1.245800e+05  1.243990e+05  1.247690e+05  1.248...
mean   1.195857e+03  1.204004e+03  1.133676e+03  1.170437e+03  1.217769e+03  1.290273e+03  1.239137e+03  1.193092e+03  1.197...
std    7.275352e+04  7.421515e+04  6.961022e+04  7.257351e+04  7.379612e+04  8.054448e+04  7.576288e+04  6.820002e+04  7.149...
min    0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000...
25%    1.300000e+01  1.300000e+01  1.200000e+01  1.300000e+01  1.400000e+01  1.100000e+01  1.300000e+01  1.300000e+01  1.400...
50%    1.090000e+02  1.080000e+02  1.050000e+02  1.050000e+02  1.130000e+02  1.130000e+02  1.150000e+02  1.170000e+02  1.150...
75%    5.240000e+02  5.190000e+02  5.040000e+02  4.870000e+02  5.400000e+02  5.550000e+02  5.510000e+02  5.540000e+02  5.490...
max    2.038124e+07  2.075219e+07  1.957397e+07  2.043964e+07  2.077211e+07  2.254467e+07  2.121089e+07  1.910791e+07  1.999...

8 rows × 550 columns
```

▼ Handle missing values

Subtask:

Identify and address missing values in the dataset. This will involve checking for NaNs, deciding on an imputation strategy (e.g., interpolation, filling), and implementing it.



```
# Calculate the number of missing values in each column
missing_values_count = df_train.isnull().sum()
```

```
# Display the results
print("Number of missing values in each column:")
print(missing_values_count)
```

```
Number of missing values in each column:
```

```
Page          0
2015-07-01   20740
2015-07-02   20816
2015-07-03   20544
2015-07-04   20654
...
2016-12-27   3701
2016-12-28   3822
2016-12-29   3826
2016-12-30   3635
2016-12-31   3465
Length: 551, dtype: int64
```

Reasoning: Apply forward fill imputation to the time series columns in the DataFrame to handle missing values.

```
# Apply forward fill imputation to fill missing values in date columns
```

```
# Select only the date columns for imputation
```

```
date_columns = df_train.columns[1:]
```

```
df_train[date_columns] = df_train[date_columns].fillna(method='ffill', axis=1)
```

```
/tmp/ipython-input-1453108621.py:4: FutureWarning: DataFrame.fillna with 'method' is deprecated and will raise in a future version
df_train[date_columns] = df_train[date_columns].fillna(method='ffill', axis=1)
```

Reasoning: Verify that the missing values have been handled after imputation by checking the number of missing values in each column again.

```
# Check the number of missing values after imputation
missing_values_count_after_imputation = df_train.isnull().sum()
```

```
# Display the results
```

```
print("Number of missing values in each column after imputation:")
print(missing_values_count_after_imputation)
```

```
Number of missing values in each column after imputation:
```

```
Page          0
2015-07-01   20740
2015-07-02   19767
2015-07-03   19256
2015-07-04   18983
...
2016-12-27   705
2016-12-28   696
2016-12-29   683
2016-12-30   669
2016-12-31   652
Length: 551, dtype: int64
```

Reasoning: The forward fill imputation was applied row-wise across date columns. However, there are still missing values, especially in the initial date columns. This indicates that some rows might start with missing values. To address this, we can apply a backward fill after the forward fill to fill any remaining NaNs at the beginning of the time series for each row.

```
# Apply backward fill imputation to fill remaining missing values
df_train[date_columns] = df_train[date_columns].fillna(method='bfill', axis=1)
```

```
# Verify that the missing values have been handled by checking the number of missing values in each column again.
missing_values_count_after_bfill = df_train.isnull().sum()
```

```
# Display the results
```

```
print("Number of missing values in each column after backward fill imputation:")
print(missing_values_count_after_bfill)
```

```
/tmp/ipython-input-3571689896.py:2: FutureWarning: DataFrame.fillna with 'method' is deprecated and will raise in a future version
df_train[date_columns] = df_train[date_columns].fillna(method='bfill', axis=1)
```

```
Number of missing values in each column after backward fill imputation:
```

```
Page          0
2015-07-01   652
2015-07-02   652
2015-07-03   652
```

```

2015-07-03    652
2015-07-04    652
...
2016-12-27    652
2016-12-28    652
2016-12-29    652
2016-12-30    652
2016-12-31    652
Length: 551, dtype: int64

```

Reasoning: Although forward and backward fill were applied, there are still 652 rows with missing values. This indicates that these rows contain only missing values across all date columns. Since these rows do not contain any valid time series data, they are not useful for forecasting and can be safely dropped.

```

# Drop rows where all date columns are NaN after imputation
df_train.dropna(subset=date_columns, how='all', inplace=True)

# Verify that the missing values have been handled by checking the number of missing values in each column again.
missing_values_count_after_dropping = df_train.isnull().sum()

# Display the results
print("Number of missing values in each column after dropping rows with all NaNs:")
print(missing_values_count_after_dropping)

Number of missing values in each column after dropping rows with all NaNs:
Page      0
2015-07-01  0
2015-07-02  0
2015-07-03  0
2015-07-04  0
..
2016-12-27  0
2016-12-28  0
2016-12-29  0
2016-12-30  0
2016-12-31  0
Length: 551, dtype: int64

```

▼ Modify data (feature engineering)

Subtask:

Extract relevant features from the 'Page' column, such as language, access type, and access origin, and add them as new columns to the DataFrame.

Reasoning: Define a function to extract features from the 'Page' column and apply it to the DataFrame, then display the result.

```

import re

def extract_page_features(page_string):
    """Extracts page name, language, access type, and access origin from the Page string."""
    parts = page_string.split('_')
    page_name = '_'.join(parts[:-3])
    language = parts[-3]
    access_type = parts[-2]
    access_origin = parts[-1]
    return page_name, language, access_type, access_origin

# Apply the function to the 'Page' column and create new columns
df_train[['page_name', 'language', 'access_type', 'access_origin']] = df_train['Page'].apply(lambda x: pd.Series(extract_page_features(x)))

# Display the first few rows with the new columns
display(df_train.head())

```

/tmp/ipython-input-144346617.py:13: PerformanceWarning: DataFrame is highly fragmented. This is usually the result of
df_train[['page_name', 'language', 'access_type', 'access_origin']] = df_train['Page'].apply(lambda x: pd.Series(extract_page_features(x)))
/tmp/ipython-input-144346617.py:13: PerformanceWarning: DataFrame is highly fragmented. This is usually the result of
df_train[['page_name', 'language', 'access_type', 'access_origin']] = df_train['Page'].apply(lambda x: pd.Series(extract_page_features(x)))
/tmp/ipython-input-144346617.py:13: PerformanceWarning: DataFrame is highly fragmented. This is usually the result of
df_train[['page_name', 'language', 'access_type', 'access_origin']] = df_train['Page'].apply(lambda x: pd.Series(extract_page_features(x)))
/tmp/ipython-input-144346617.py:13: PerformanceWarning: DataFrame is highly fragmented. This is usually the result of
df_train[['page_name', 'language', 'access_type', 'access_origin']] = df_train['Page'].apply(lambda x: pd.Series(extract_page_features(x)))

	Page	2015-07-01	2015-07-02	2015-07-03	2015-07-04	2015-07-05	2015-07-06	2015-07-07
0	2NE1_zh.wikipedia.org_all-	18.0	11.0	5.0	13.0	14.0	9.0	9.0

	access_spider
1	2PM_zh.wikipedia.org_all-access_spider	11.0	14.0	15.0	18.0	11.0	13.0	22.0	...
2	3C_zh.wikipedia.org_all-access_spider	1.0	0.0	1.0	1.0	0.0	4.0	0.0	...
3	4minute_zh.wikipedia.org_all-access_spider	35.0	13.0	10.0	94.0	4.0	26.0	14.0	...
4	52_Hz_I_Love_You_zh.wikipedia.org_all-access_s...	38.0	38.0	38.0	38.0	38.0	38.0	38.0	38.0

5 rows × 555 columns

▼ Perform eda

Subtask:

Generate plots and visualizations (using libraries like matplotlib and seaborn) to understand the distribution of data and gain insights from the features, including the extracted ones.

Reasoning: Generate count plots for the newly extracted categorical features: language, access_type, and access_origin to visualize their distributions.

```
import matplotlib.pyplot as plt
import seaborn as sns

# Set the figure size for better readability
plt.figure(figsize=(12, 5))

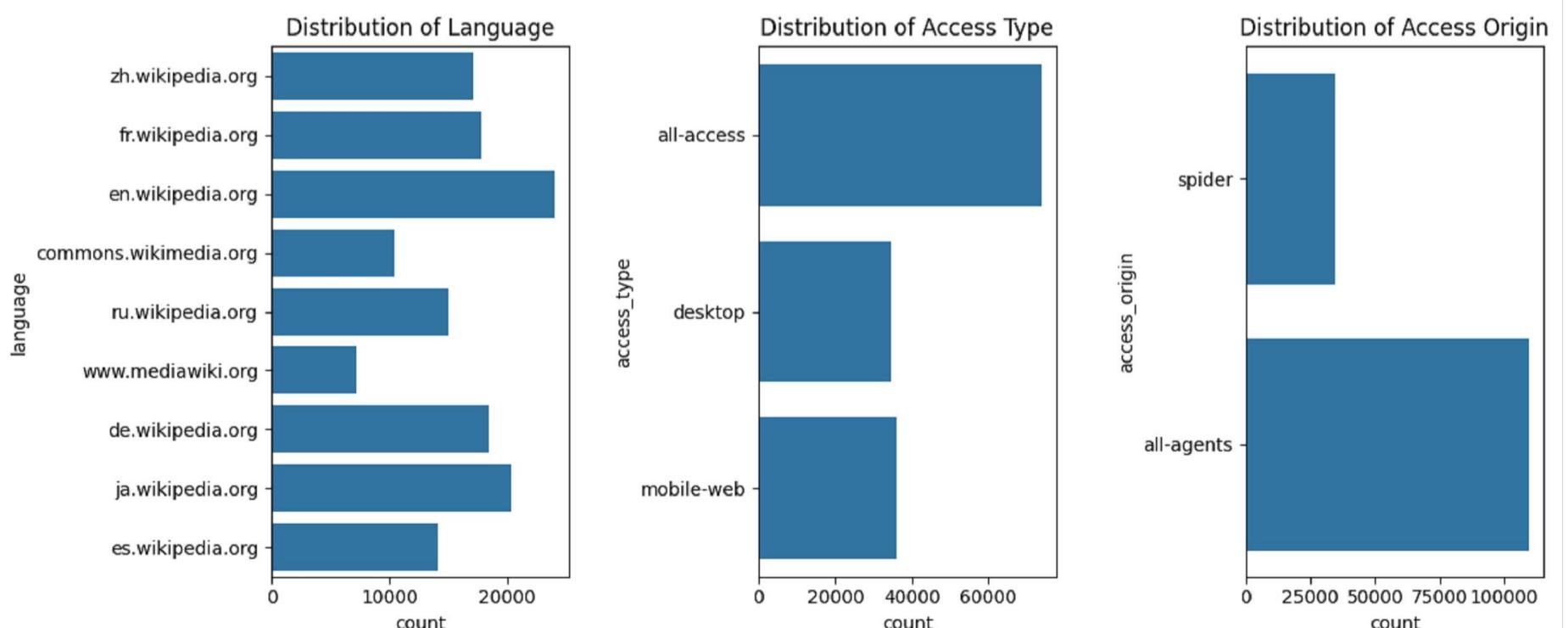
# Create a count plot for 'language'
plt.subplot(1, 3, 1)
sns.countplot(data=df_train, y='language')
plt.title('Distribution of Language')

# Create a count plot for 'access_type'
plt.subplot(1, 3, 2)
sns.countplot(data=df_train, y='access_type')
plt.title('Distribution of Access Type')

# Create a count plot for 'access_origin'
plt.subplot(1, 3, 3)
sns.countplot(data=df_train, y='access_origin')
plt.title('Distribution of Access Origin')

# Adjust layout to prevent overlapping titles/labels
plt.tight_layout()

# Display the plots
plt.show()
```



✓ Aggregate and transform data

Subtask:

Explain the purpose of aggregation and pivoting (melting in this context) for time series analysis. Transform the data from wide format to long format, aggregating as necessary (e.g., by language and date).

Reasoning: Explain the purpose of aggregation and pivoting for time series analysis, then transform the data from wide to long format, convert the date column, aggregate by language and date, and display the aggregated data.

```
# Transforming data from wide to long format and aggregating by relevant dimensions
# like language and date is crucial for time series analysis because it restructures
# the data into a format where each row represents a single observation at a specific
# point in time for a particular entity (e.g., page views for a specific language on a specific date).
# This "long" format is the standard structure for time series data and facilitates
# time-based operations (like trend analysis, seasonality decomposition) and visualizations
# (like line plots showing views over time for different languages). Aggregation helps in
# summarizing the data at the desired granularity, such as total views per day per language.

# Identify the date columns
date_columns = df_train.columns.drop(['Page', 'page_name', 'language', 'access_type', 'access_origin'])

# Melt the DataFrame from wide to long format
df_melted = pd.melt(df_train,
                     id_vars=['Page', 'page_name', 'language', 'access_type', 'access_origin'],
                     value_vars=date_columns,
                     var_name='date',
                     value_name='views')

# Convert the 'date' column to datetime objects
df_melted['date'] = pd.to_datetime(df_melted['date'])

# Aggregate by 'language' and 'date', summing the 'views'
df_agg = df_melted.groupby(['language', 'date'])['views'].sum().reset_index()

# Display the first few rows of the aggregated DataFrame
display(df_agg.head())
```

	language	date	views	
0	commons.wikimedia.org	2015-07-01	1410141.0	
1	commons.wikimedia.org	2015-07-02	1448163.0	
2	commons.wikimedia.org	2015-07-03	1419407.0	
3	commons.wikimedia.org	2015-07-04	1220335.0	
4	commons.wikimedia.org	2015-07-05	1327108.0	

✓ Plot time series

Subtask:

Generate time series plots for different languages (or other relevant groupings) from the transformed data to observe trends and patterns over time.

Reasoning: Iterate through unique languages in df_agg, filter the DataFrame for each language, and create a time series plot of views over time using seaborn.

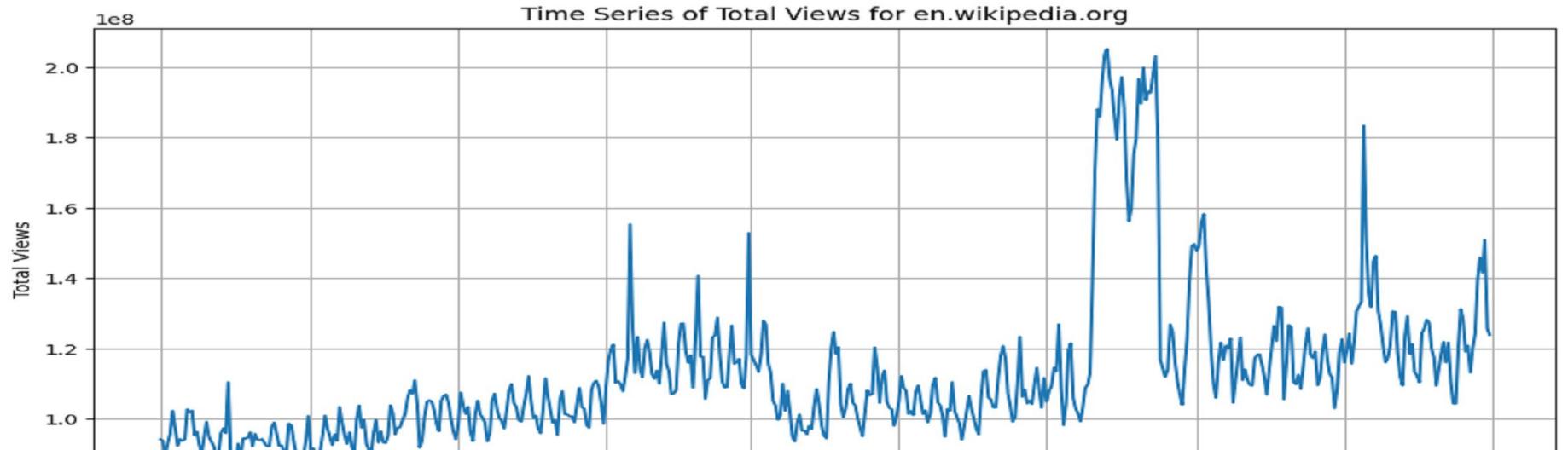
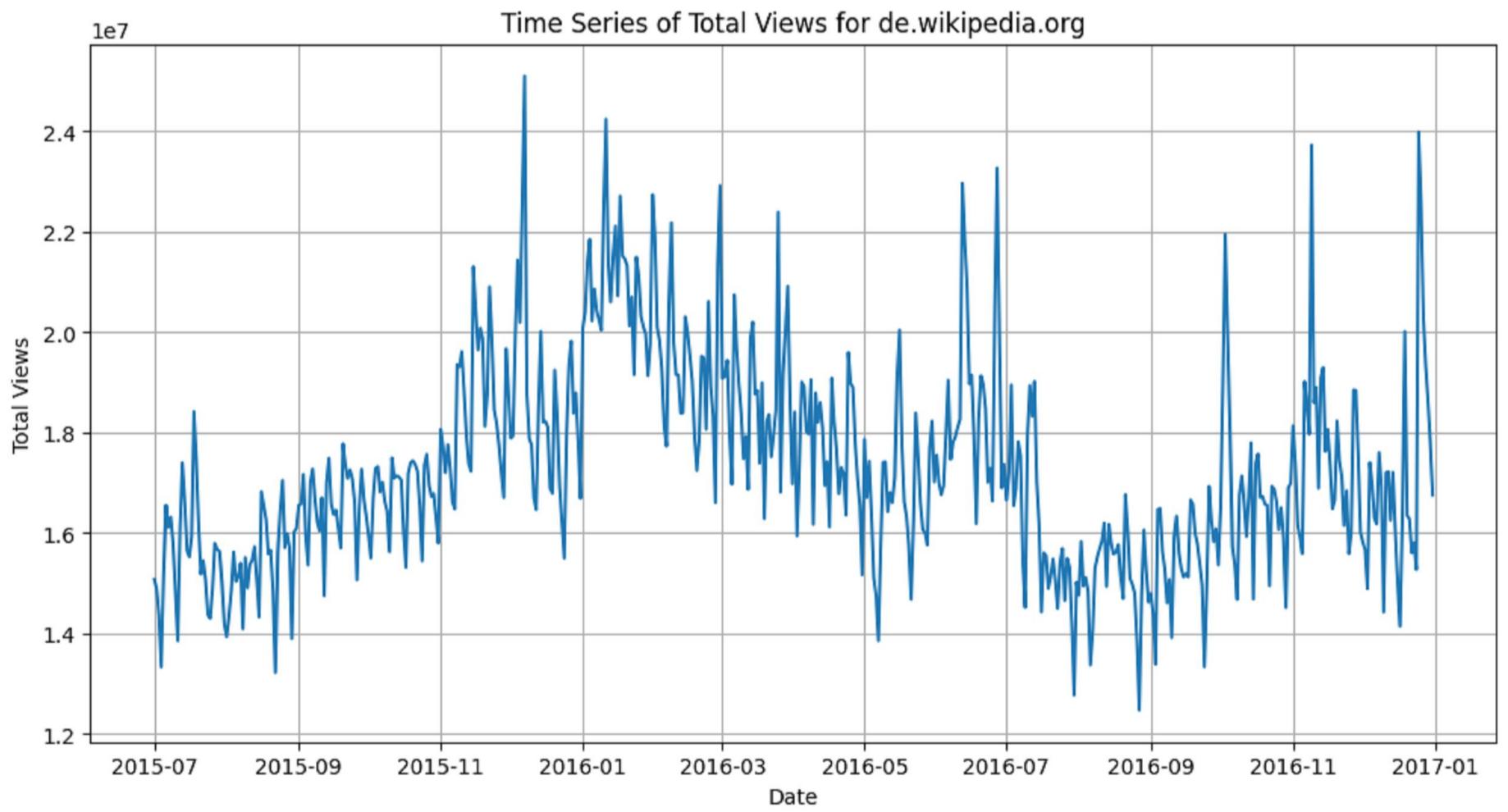
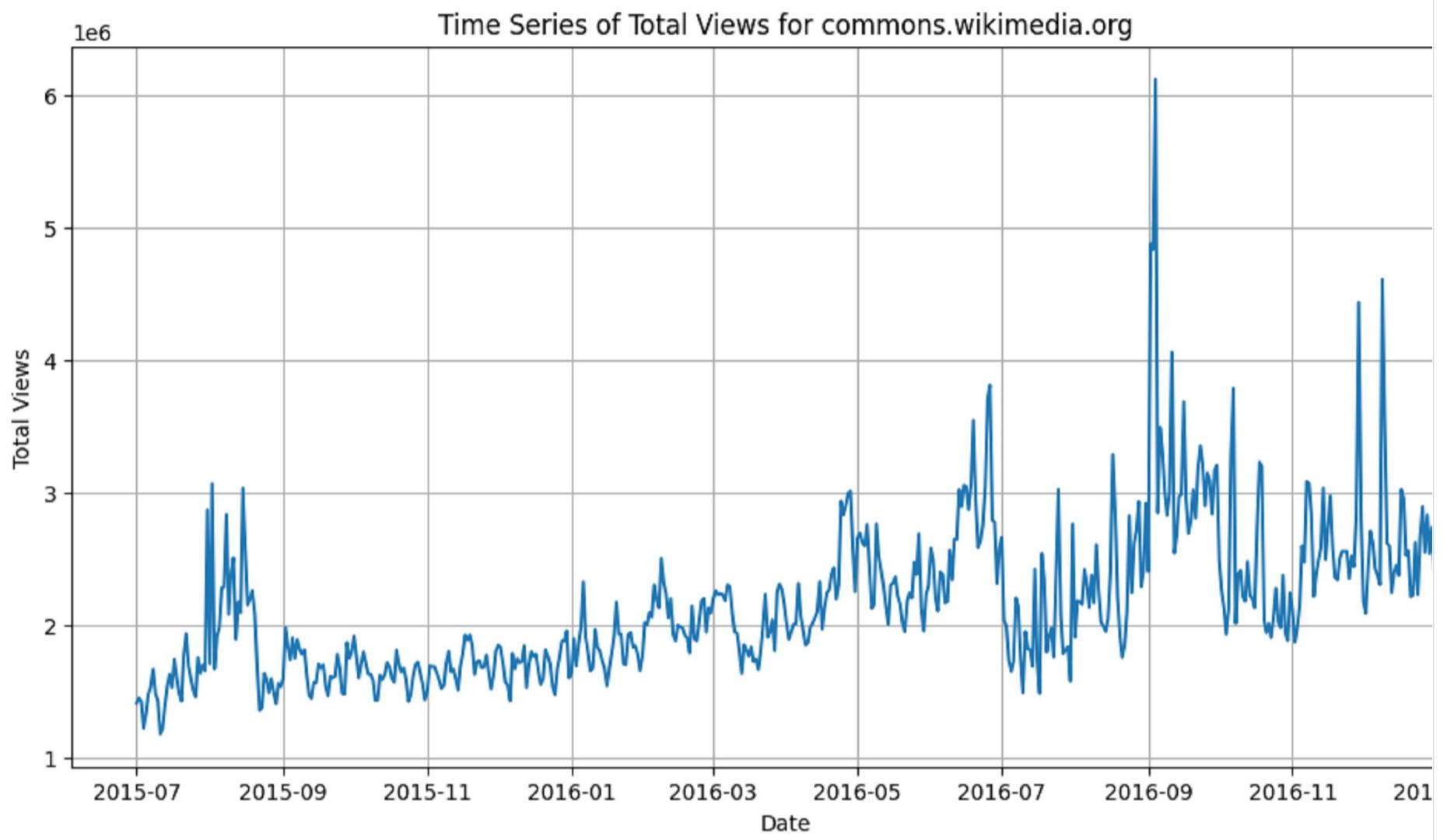
```
# Get unique languages
unique_languages = df_agg['language'].unique()

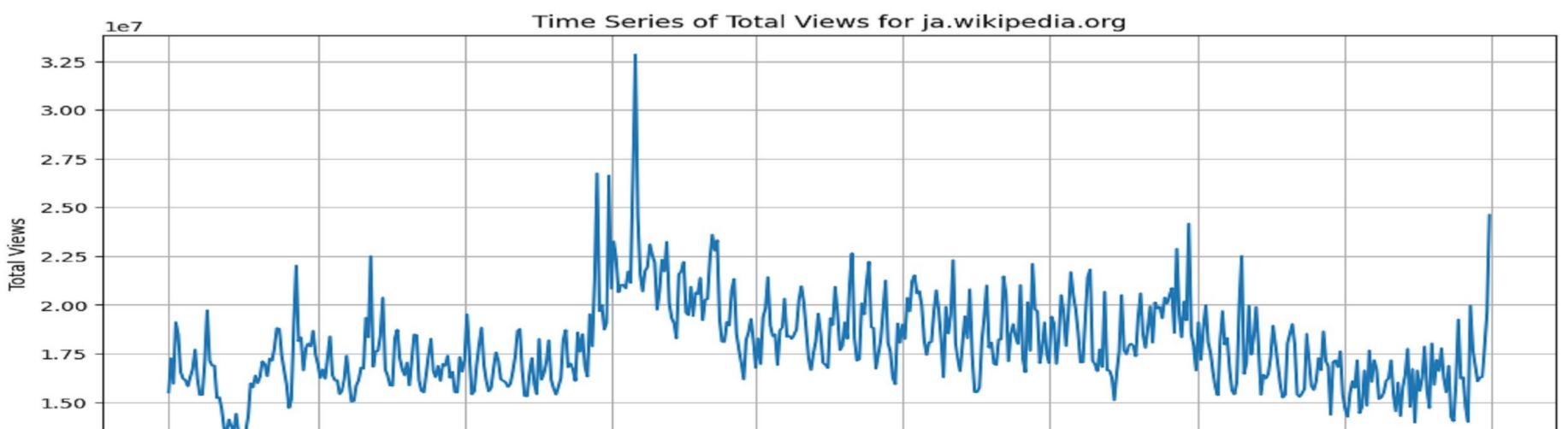
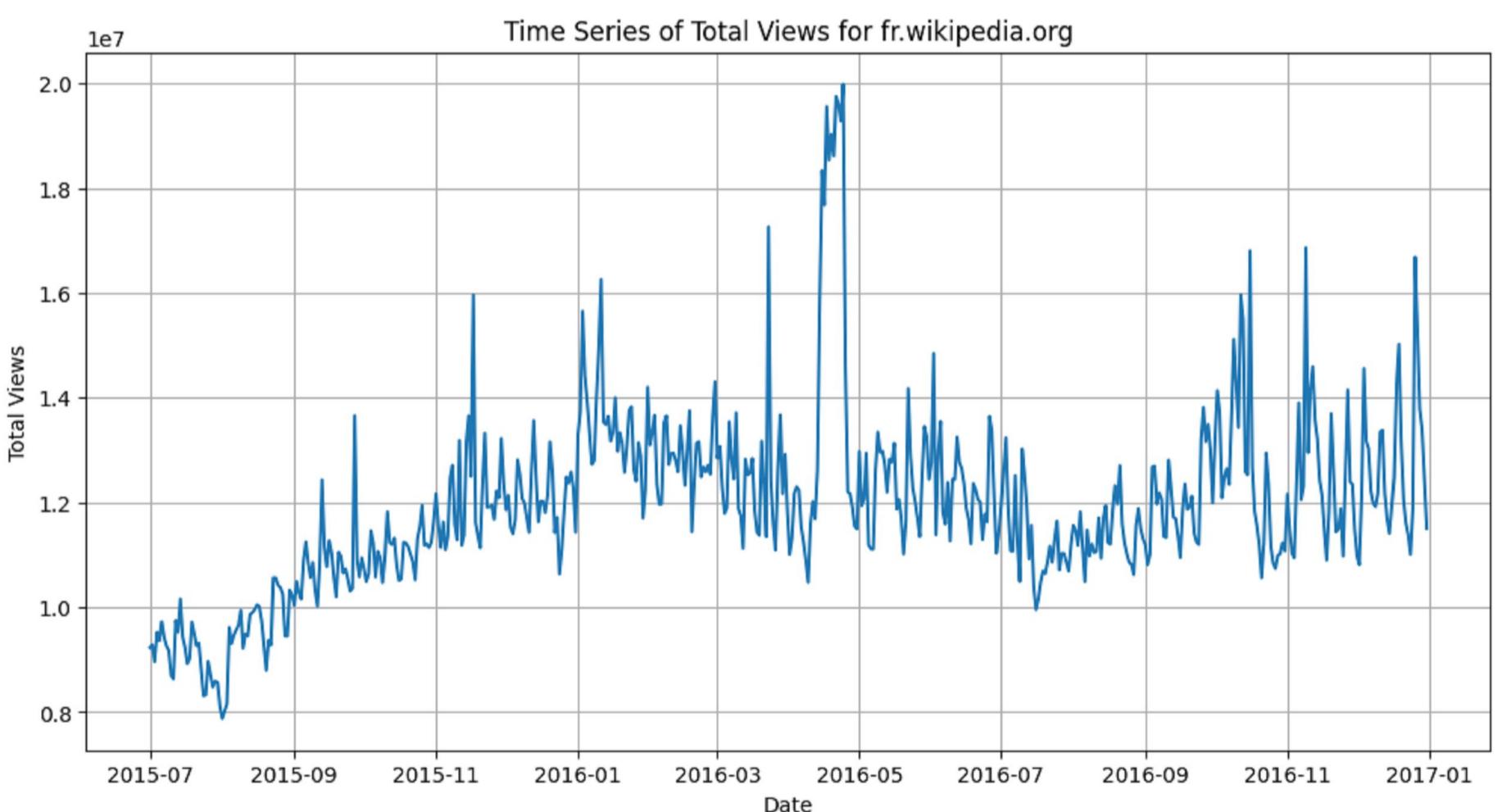
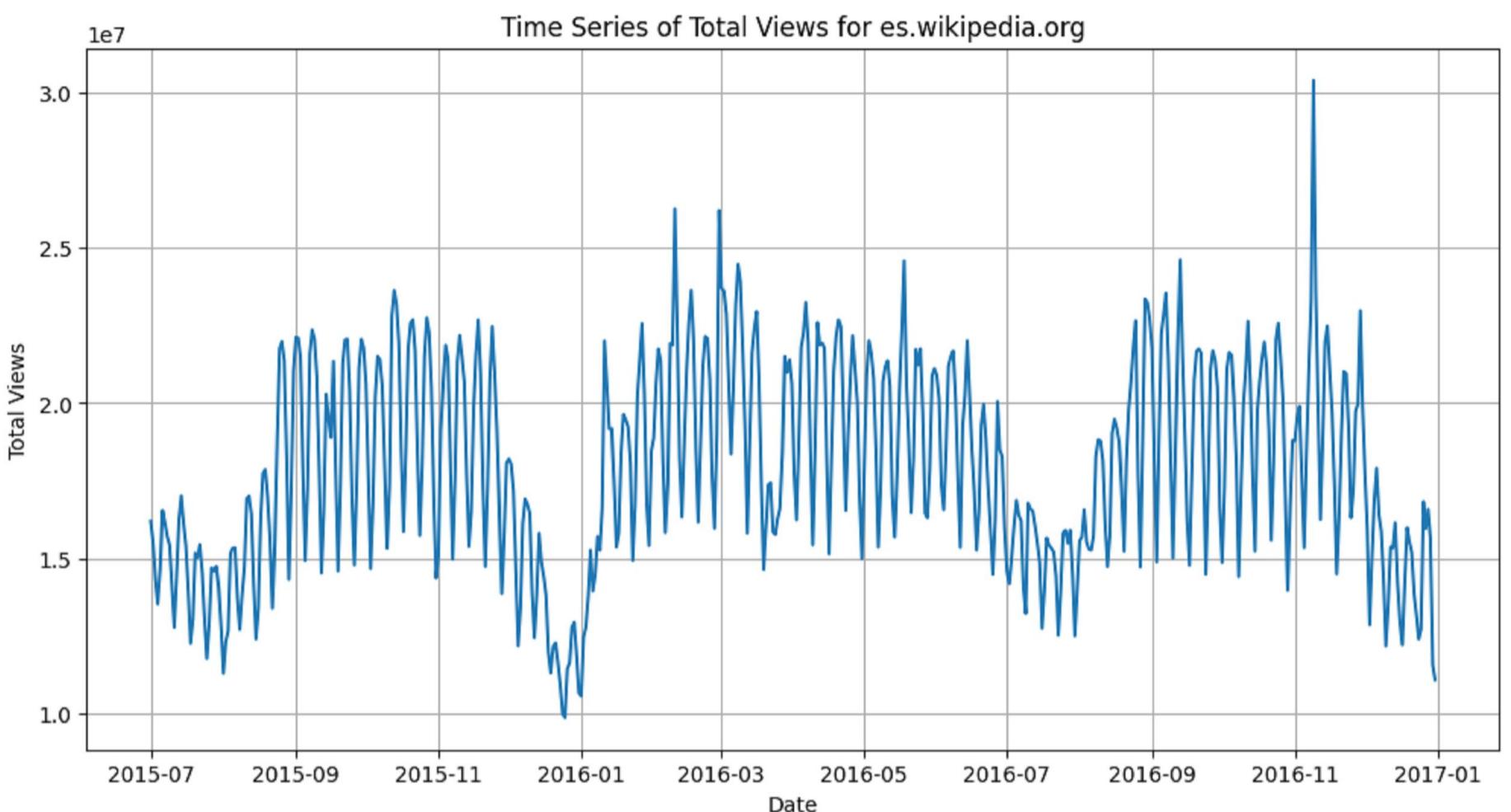
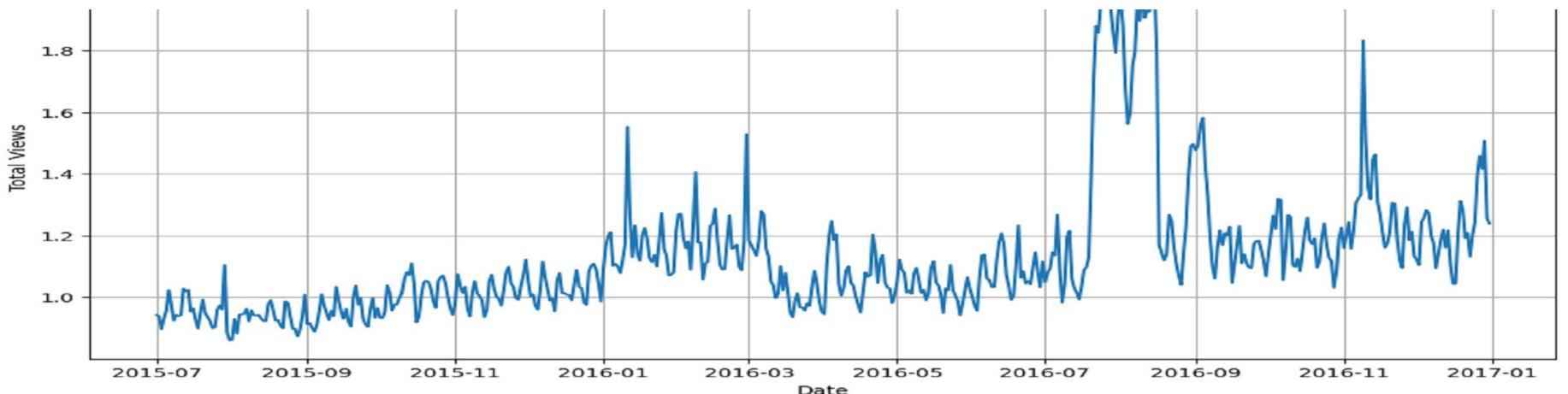
# Iterate through each language and create a time series plot
for language in unique_languages:
    # Filter data for the current language
    df_language = df_agg[df_agg['language'] == language]
```

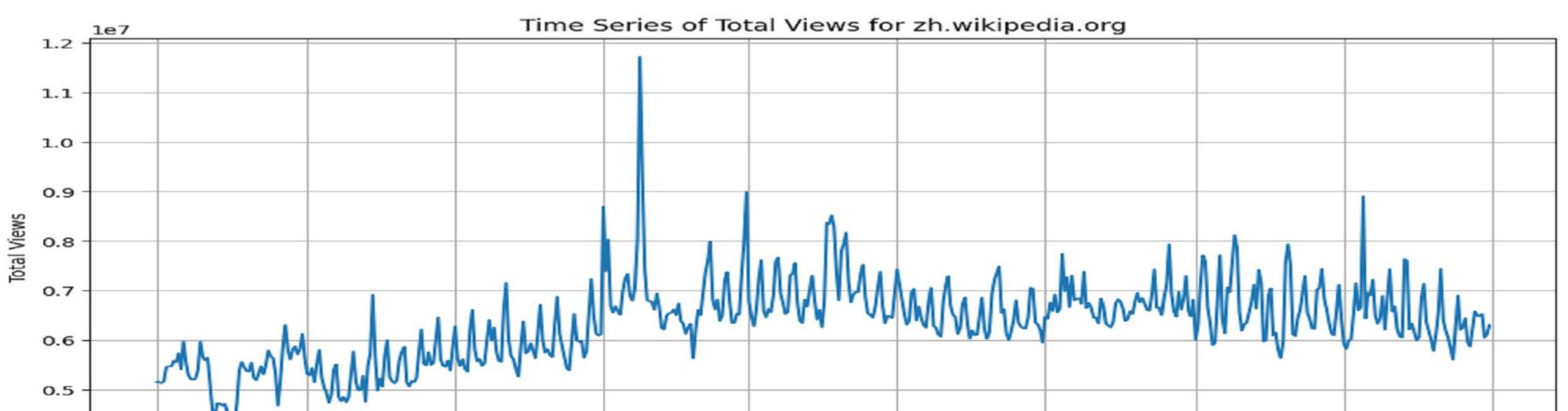
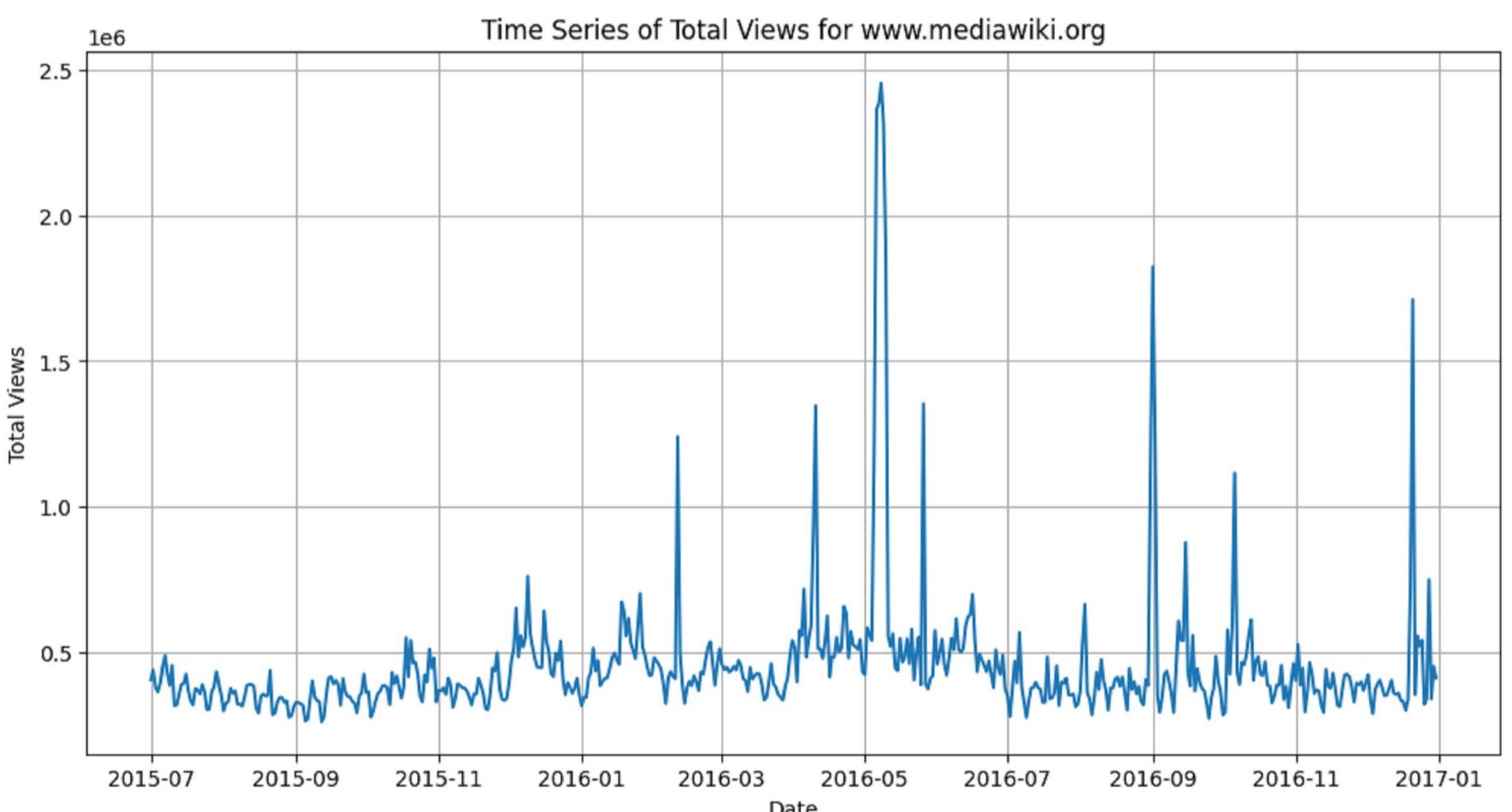
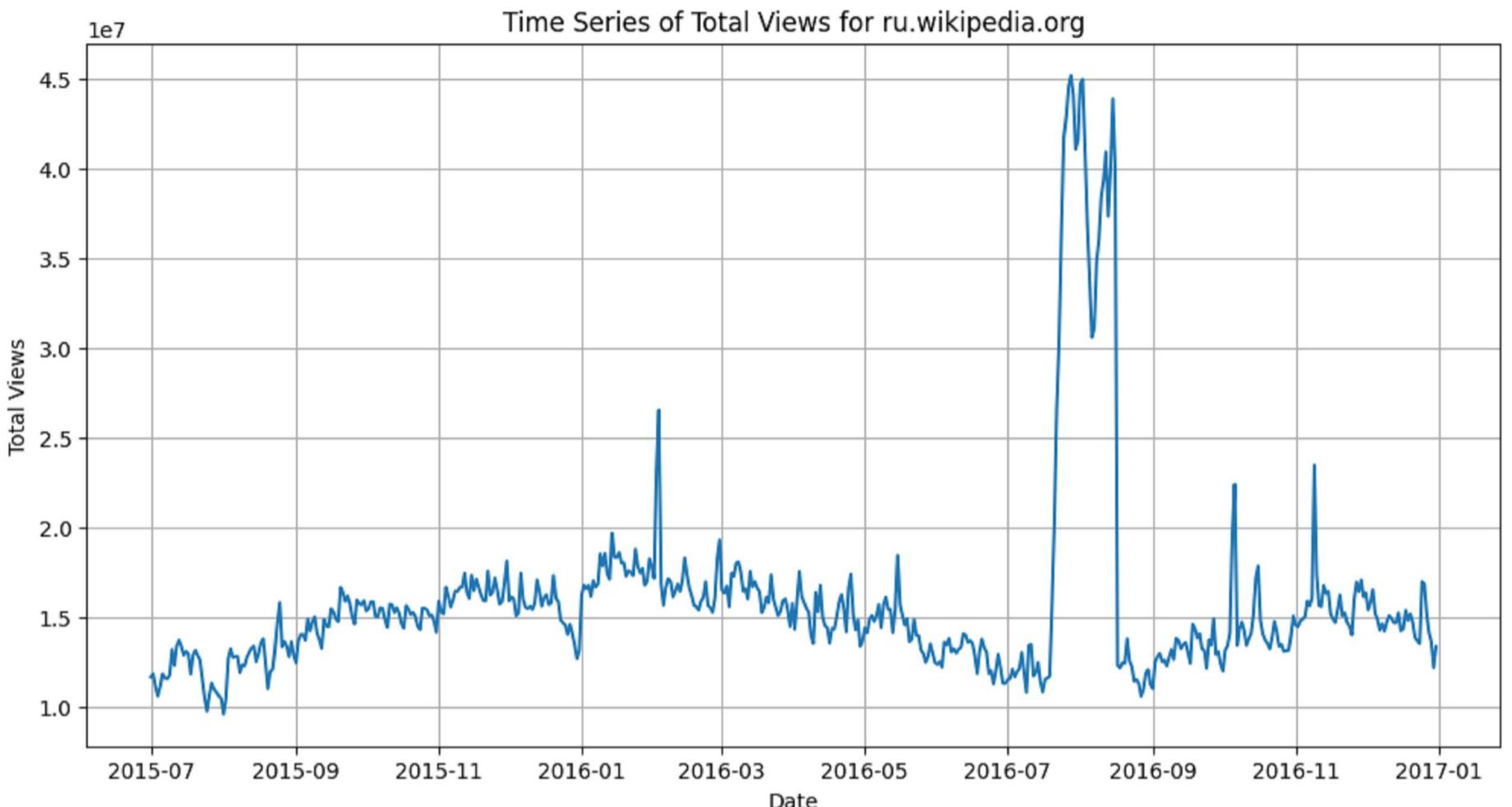
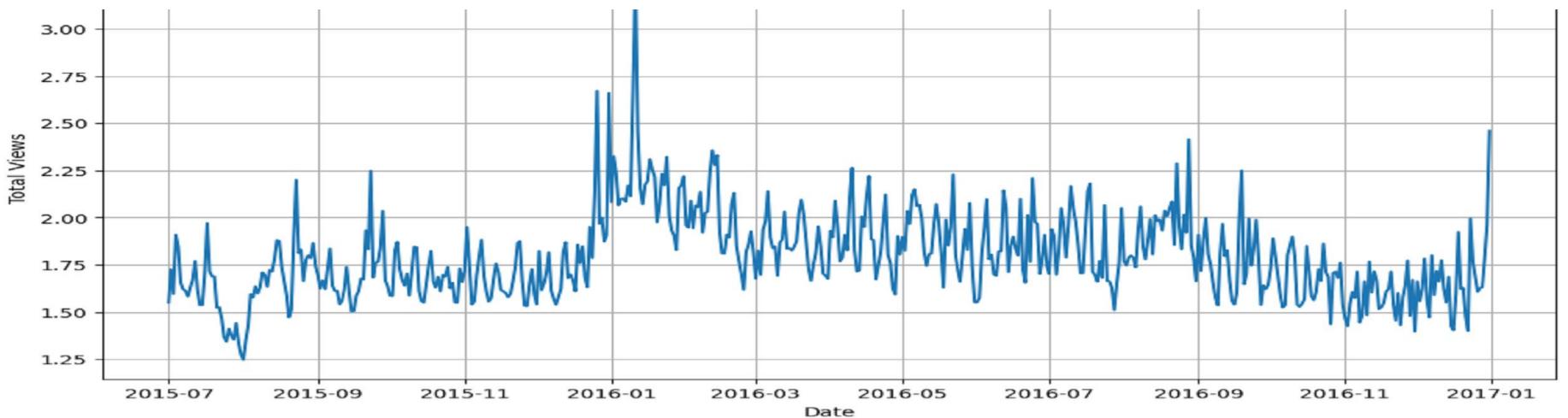
```

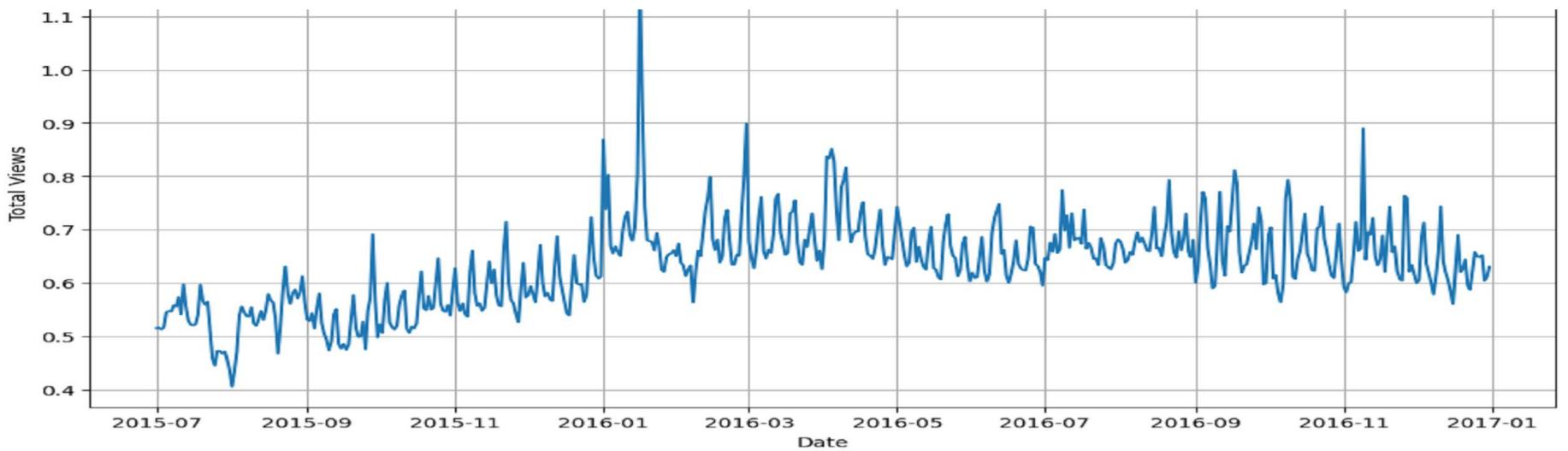
# Create the time series plot
plt.figure(figsize=(12, 6))
sns.lineplot(data=df_language, x='date', y='views')
plt.title(f'Time Series of Total Views for {language}')
plt.xlabel('Date')
plt.ylabel('Total Views')
plt.grid(True)
plt.show()

```









Summary:

Data Analysis Key Findings

- The `train_1.csv` dataset contains 145,063 rows and 551 columns, with one object column (`Page`) and 550 numerical columns representing daily page views.
- The dataset initially had a significant number of missing values across the date columns.
- Missing values were handled using a combination of row-wise forward fill, backward fill, and dropping rows where all date columns were missing, resulting in a dataset with no missing values in the date columns.
- Features such as page name, language, access type, and access origin were successfully extracted from the 'Page' column.
- Count plots revealed the distribution of languages, access types, and access origins within the dataset.
- The data was successfully transformed from a wide to a long format and aggregated by language and date, which is a suitable structure for time series analysis.
- Time series plots for each language show the trend of total page views over time, allowing for visual observation of patterns.

Insights or Next Steps

- Analyze the time series plots for each language to identify trends, seasonality, and potential outliers that could impact forecasting models.
- Consider incorporating the external campaign data (`Exog_Campaign_eng`) as an exogenous variable to improve the accuracy of page view forecasts.

Stationarity Test and Decomposition

Why Stationarity is Important:

A stationary time series is one whose statistical properties (like mean, variance, and autocorrelation) do not change over time. Many time series forecasting models and techniques assume that the underlying time series is stationary. If a time series is not stationary, these models may produce inaccurate forecasts.

Common characteristics of non-stationary time series include:

- Trend:** A consistent upward or downward movement over time.
- Seasonality:** Repeating patterns at fixed intervals (e.g., daily, weekly, monthly).
- Heteroscedasticity:** Changing variance over time.

To apply models that assume stationarity, we often need to transform a non-stationary series into a stationary one through techniques like differencing, detrending, or seasonal decomposition.

Augmented Dickey-Fuller (ADF) Test:

The Augmented Dickey-Fuller (ADF) test is a statistical test used to determine if a time series is stationary. The null hypothesis (H0) of the ADF test is that the time series is non-stationary (it has a unit root). The alternative hypothesis (H1) is that the time series is stationary.

If the p-value from the ADF test is less than a chosen significance level (e.g., 0.05), we reject the null hypothesis and conclude that the time series is likely stationary. If the p-value is greater than the significance level, we fail to reject the null hypothesis, suggesting the time series is non-stationary.

```
from statsmodels.tsa.stattools import adfuller

# For the ADF test, we need a single time series.
# Let's select the time series for the first language in our aggregated data (df_agg).
# Make sure df_agg is available from previous steps.
if 'df_agg' not in locals():
    print("Error: df_agg not found. Please run the aggregation step first.")
else:
    # Select the time series for the first language
    language_to_test = df_agg['language'].unique()[0]
    time_series_to_test = df_agg[df_agg['language'] == language_to_test].set_index('date')['views']

    # Perform the Augmented Dickey-Fuller test
    print(f"\nPerforming Augmented Dickey-Fuller test for: {language_to_test}")
    adf_result = adfuller(time_series_to_test)

    print('ADF Statistic: %f' % adf_result[0])
    print('p-value: %f' % adf_result[1])
    print('Critical Values:')
    for key, value in adf_result[4].items():
        print('\t%s: %.3f' % (key, value))

    # Interpret the result
    if adf_result[1] <= 0.05:
        print(f"\nInterpretation: The p-value ({adf_result[1]:.4f}) is less than or equal to 0.05. We reject the null hypothesis.")
        print(f"Conclusion: The time series for {language_to_test} is likely stationary.")
    else:
        print(f"\nInterpretation: The p-value ({adf_result[1]:.4f}) is greater than 0.05. We fail to reject the null hypothesis.")
        print(f"Conclusion: The time series for {language_to_test} is likely non-stationary.")
```

Performing Augmented Dickey-Fuller test for: commons.wikimedia.org

ADF Statistic: -3.151964

p-value: 0.022939

Critical Values:

1%: -3.442

5%: -2.867

10%: -2.570

Interpretation: The p-value (0.0229) is less than or equal to 0.05. We reject the null hypothesis.

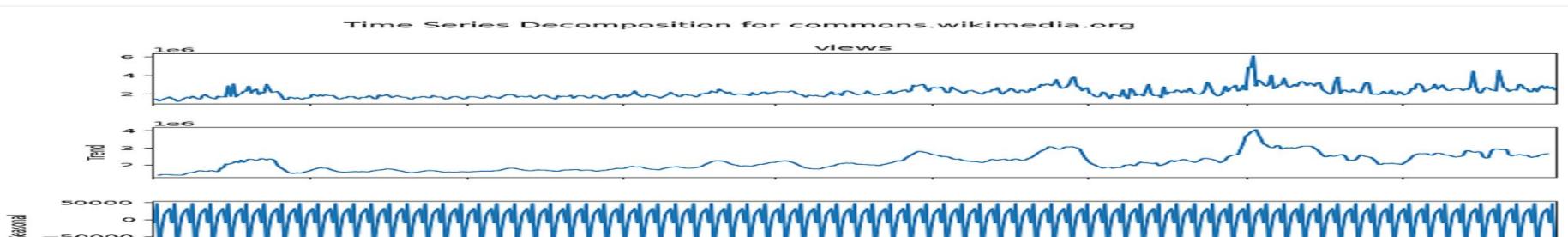
Conclusion: The time series for commons.wikimedia.org is likely stationary.

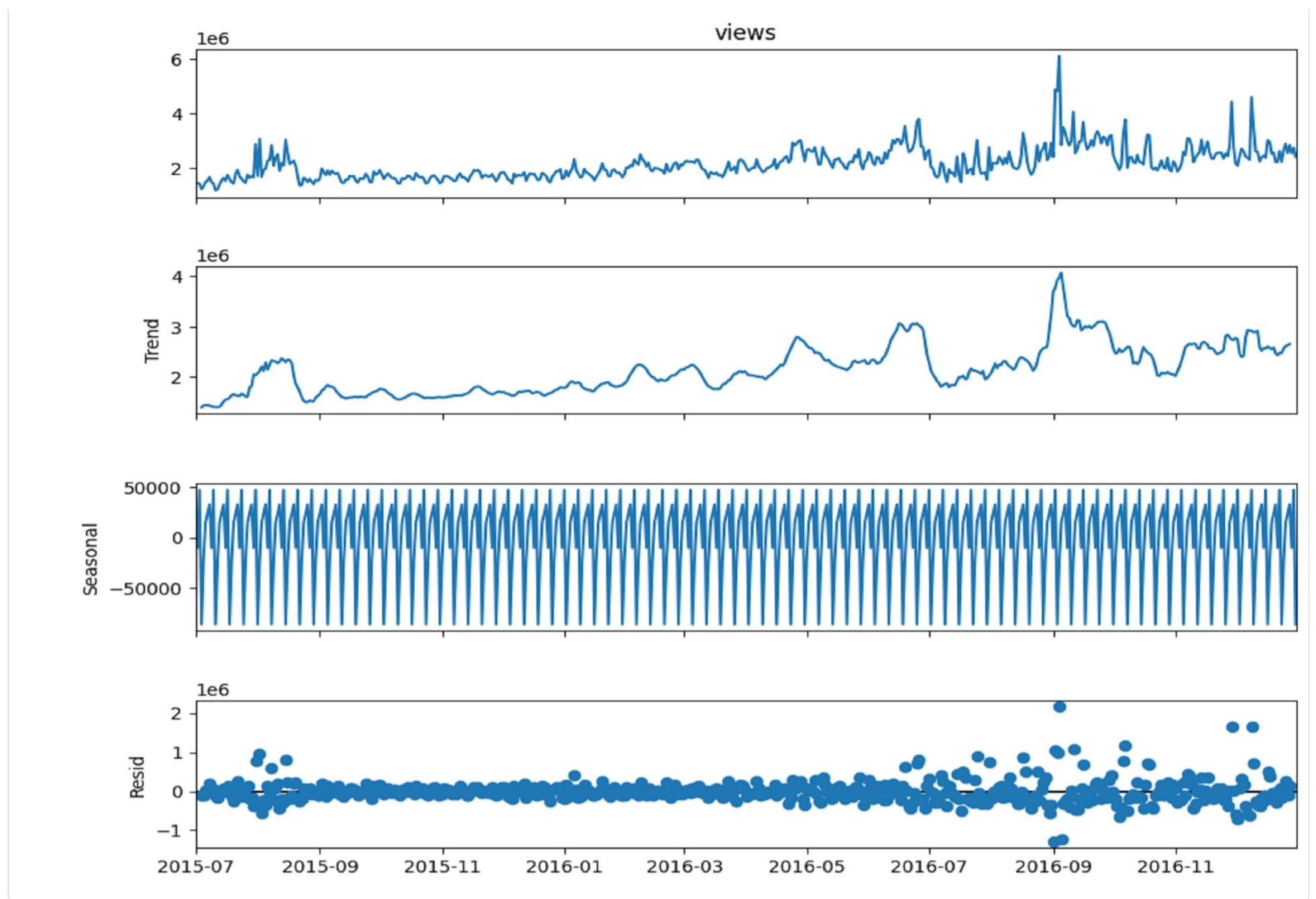
▼ Time Series Decomposition

```
from statsmodels.tsa.seasonal import seasonal_decompose

# Perform additive decomposition
decomposition = seasonal_decompose(time_series_to_test, model='additive')

# Plot the decomposed components
fig = decomposition.plot()
fig.set_size_inches(10, 8)
plt.suptitle(f'Time Series Decomposition for {language_to_test}', y=1.02)
plt.show()
```





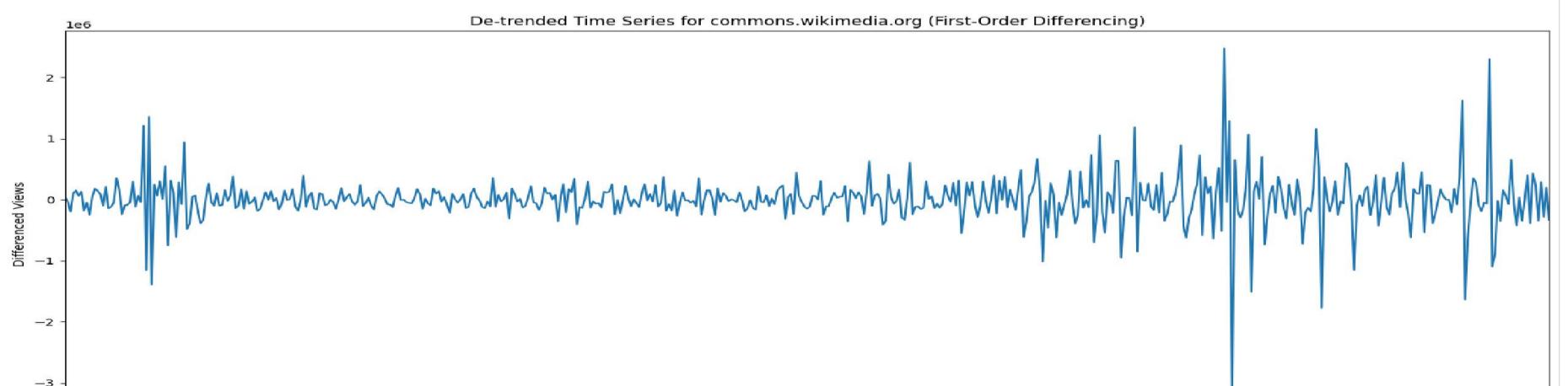
▼ De-trending and De-seasoning

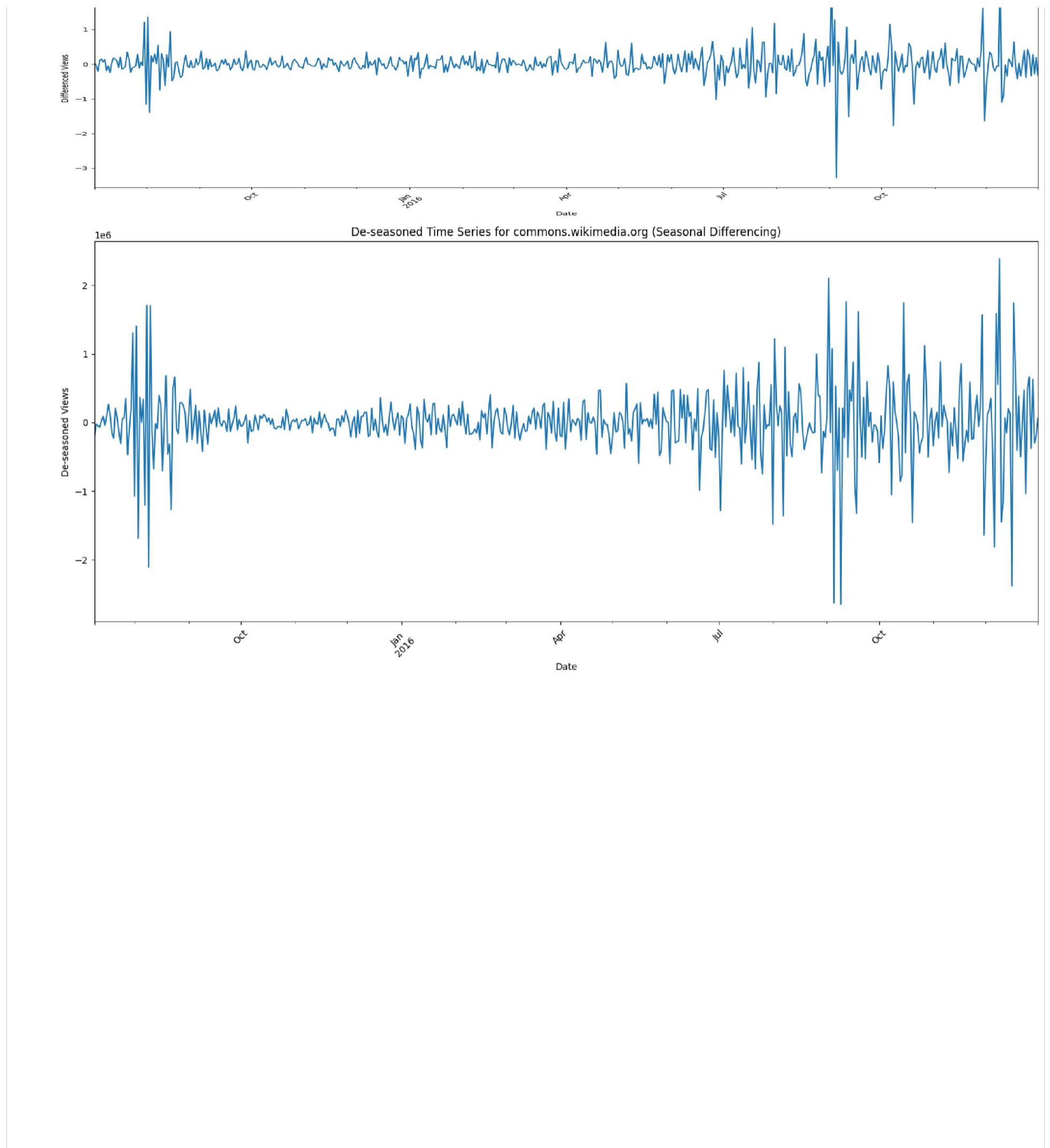
```
# Apply first-order differencing to remove the trend
time_series_detrended = time_series_to_test.diff().dropna()

# Plot the de-trended time series
plt.figure(figsize=(15, 7))
time_series_detrended.plot()
plt.title(f'De-trended Time Series for {language_to_test} (First-Order Differencing)')
plt.xlabel('Date')
plt.ylabel('Differenced Views')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()

# (Optional) Apply seasonal differencing if significant seasonality is observed (e.g., weekly)
seasonal_period = 7 # Assuming weekly seasonality
time_series_deseasoned = time_series_detrended.diff(periods=seasonal_period).dropna()

# Plot the de-seasoned time series (if seasonal differencing is applied)
plt.figure(figsize=(15, 7))
time_series_deseasoned.plot()
plt.title(f'De-seasoned Time Series for {language_to_test} (Seasonal Differencing)')
plt.xlabel('Date')
plt.ylabel('De-seasoned Views')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```





Getting Insights into Time Series Characteristics (ACF and PACF Plots)

```

from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

# Plot ACF and PACF for the original time series
print("ACF and PACF plots for the original time series:")
fig, axes = plt.subplots(2, 1, figsize=(12, 8))
plot_acf(time_series_to_test, ax=axes[0], lags=50) # Plot ACF up to lag 50
plot_pacf(time_series_to_test, ax=axes[1], lags=50) # Plot PACF up to lag 50
plt.suptitle(f'ACF and PACF for Original Time Series ({language_to_test})', y=1.02)
plt.tight_layout()
plt.show()

# Plot ACF for the de-trended time series
print("\nACF plot for the de-trended time series:")
plt.figure(figsize=(12, 4))
plot_acf(time_series_detrended, lags=50, ax=plt.gca())
plt.title(f'ACF for De-trended Time Series ({language_to_test})')
plt.xlabel('Lags')
plt.ylabel('Autocorrelation')

```

```

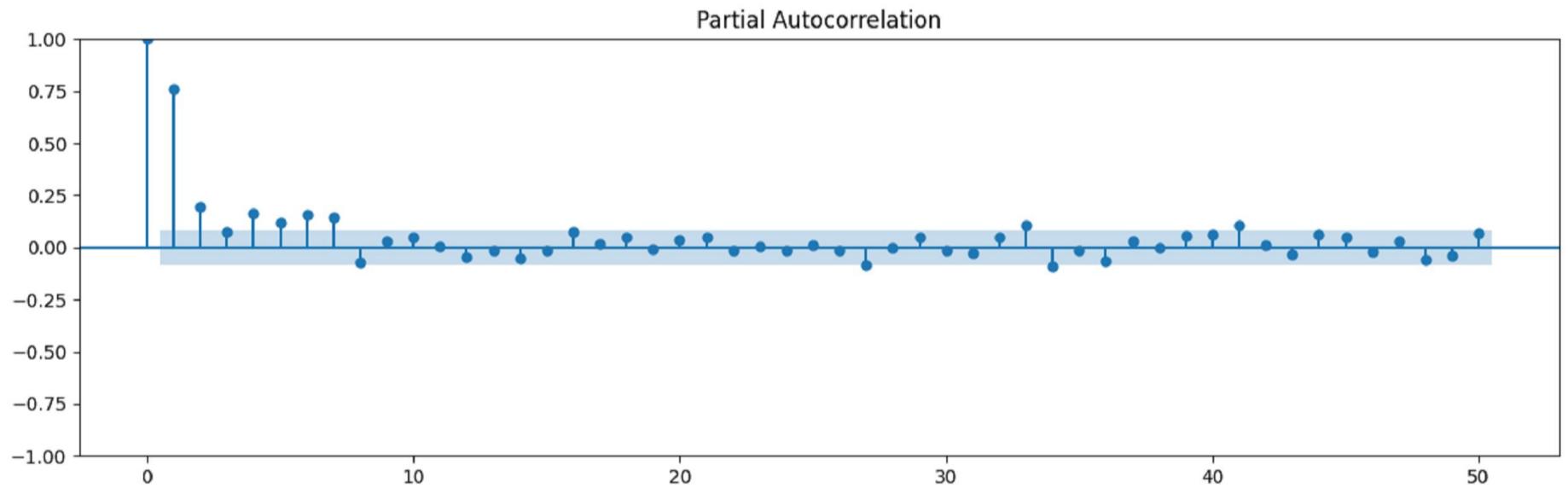
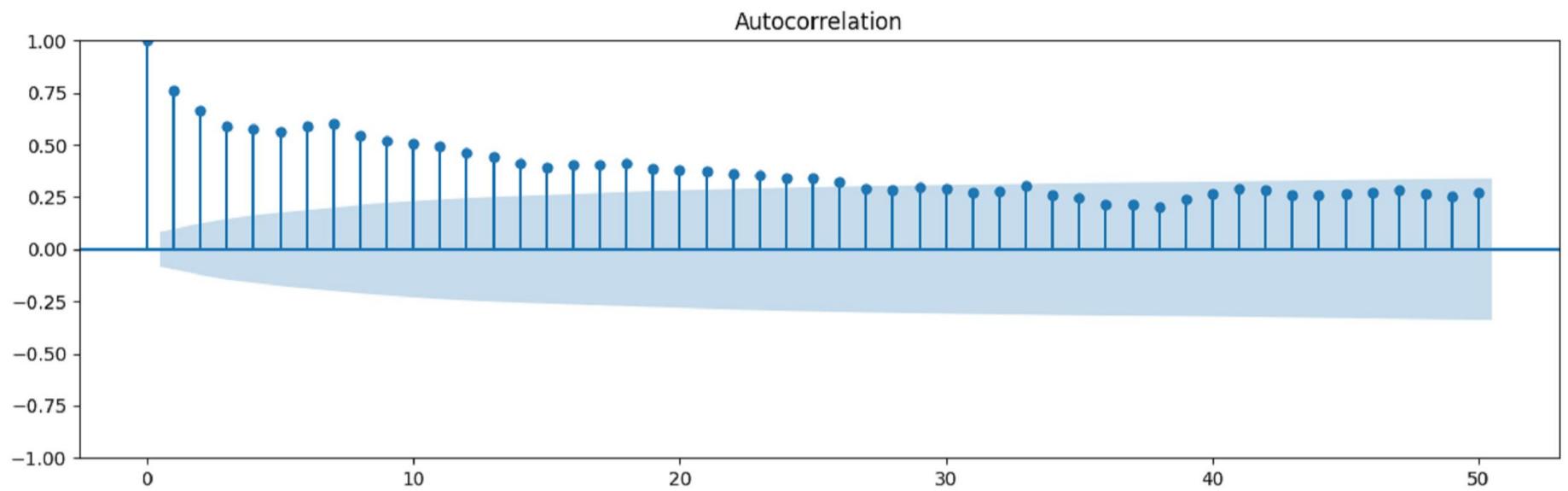
plt.tight_layout()
plt.show()

# (Optional) Plot ACF/PACF for de-seasoned time series if seasonal differencing was applied
if 'time_series_deseasoned' in locals():
    print("\nACF plot for the de-seasoned time series:")
    plt.figure(figsize=(12, 4))
    plot_acf(time_series_deseasoned, lags=50, ax=plt.gca())
    plt.title(f'ACF for De-seasoned Time Series ({language_to_test})')
    plt.xlabel('Lags')
    plt.ylabel('Autocorrelation')
    plt.tight_layout()
    plt.show()

```

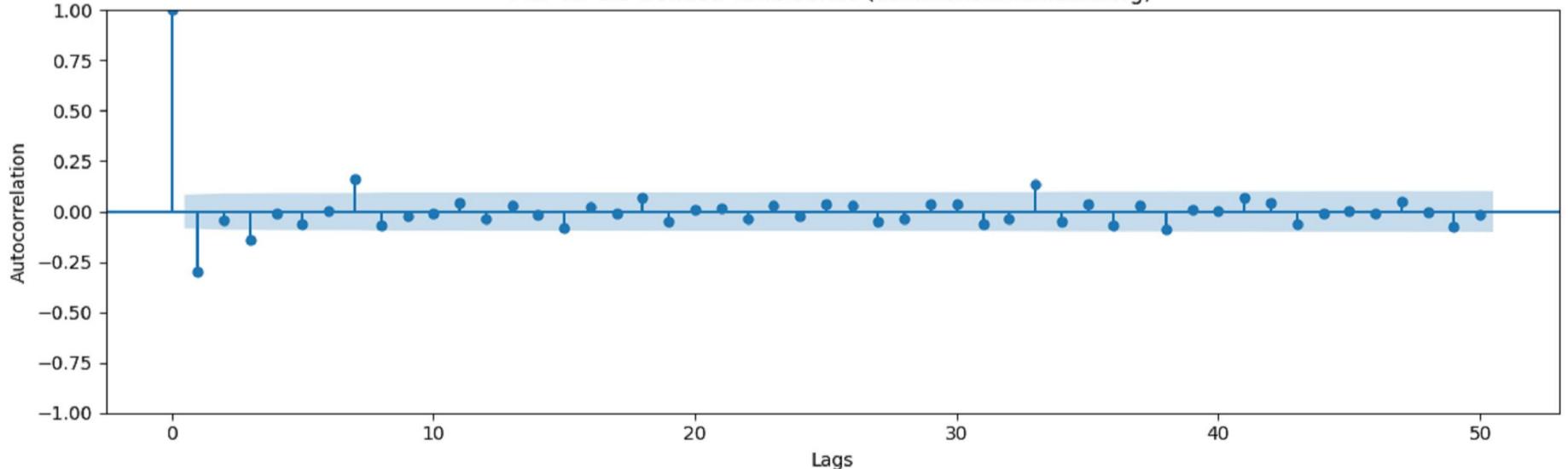
ACF and PACF plots for the original time series:

ACF and PACF for Original Time Series (commons.wikimedia.org)



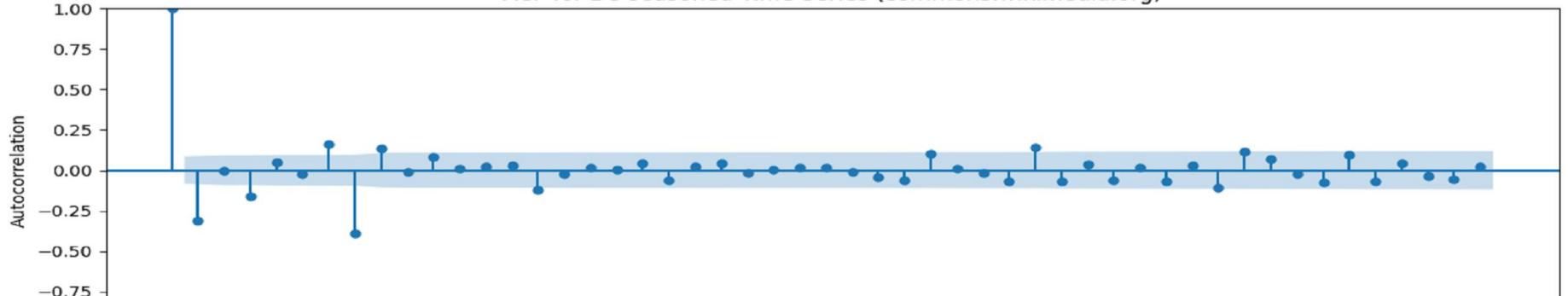
ACF plot for the de-trended time series:

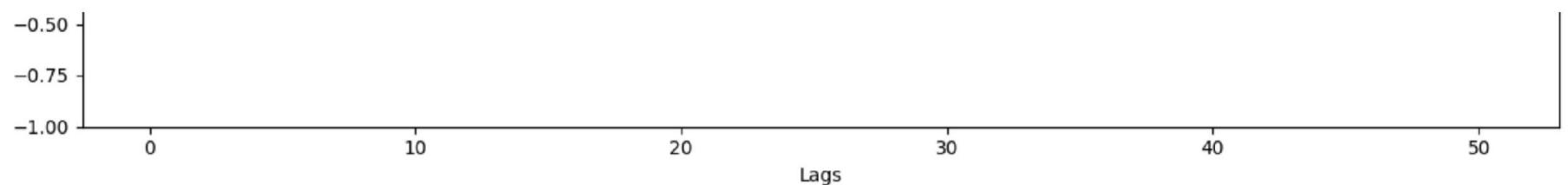
ACF for De-trended Time Series (commons.wikimedia.org)



ACF plot for the de-seasoned time series:

ACF for De-seasoned Time Series (commons.wikimedia.org)





Start coding or [generate](#) with AI.

▼ Data Splitting

```
# Ensure df_agg is available from the aggregation step
if 'df_agg' not in locals():
    print("Error: df_agg not found. Please ensure the aggregation step has been run to create df_agg.")
else:
    # Sort the aggregated data by date to ensure correct chronological split
    df_agg = df_agg.sort_values(by='date')

    # Determine the split point (e.g., split off the last few months for testing)
    # Let's say we want the last 3 months for testing
    last_date = df_agg['date'].max()
    split_date = last_date - pd.DateOffset(months=3)

    # Split the aggregated data into training and testing sets
    train_data_agg = df_agg[df_agg['date'] <= split_date].copy()
    test_data_agg = df_agg[df_agg['date'] > split_date].copy()

    print(f"Aggregated Training data shape: {train_data_agg.shape}")
    print(f"Aggregated Testing data shape: {test_data_agg.shape}")

    print(f"\nAggregated Training data date range: {train_data_agg['date'].min()} to {train_data_agg['date'].max()}")
    print(f"\nAggregated Testing data date range: {test_data_agg['date'].min()} to {test_data_agg['date'].max()}")

Aggregated Training data shape: (4122, 3)
Aggregated Testing data shape: (828, 3)

Aggregated Training data date range: 2015-07-01 00:00:00 to 2016-09-30 00:00:00

Aggregated Testing data date range: 2016-10-01 00:00:00 to 2016-12-31 00:00:00
```

▼ ARIMA Model Forecasting (without exogenous variable)

```
from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import mean_squared_error
import numpy as np

# Select the time series for the first language from the aggregated training data
language_to_model = train_data_agg['language'].unique()[0]
train_ts = train_data_agg[train_data_agg['language'] == language_to_model].set_index('date')['views']
test_ts = test_data_agg[test_data_agg['language'] == language_to_model].set_index('date')['views']

# Fit an ARIMA model
# Using (5,1,0) as initial parameters based on common practices and potential insights from ACF/PACF (if analyzed)
# You might need to tune these parameters (p, d, q) based on the specific time series and ACF/PACF plots
order = (5, 1, 0)
model = ARIMA(train_ts, order=order)
```

```

model_fit = model.fit()

print(model_fit.summary())

# Make predictions on the test set
predictions = model_fit.predict(start=len(train_ts), end=len(train_ts) + len(test_ts) - 1)

# Evaluate the model (using RMSE for now)
rmse = np.sqrt(mean_squared_error(test_ts, predictions))
print(f'\nRMSE for ARIMA model on {language_to_model}: {rmse:.3f}')

# Plot the actual vs predicted values
plt.figure(figsize=(15, 7))
plt.plot(test_ts.index, test_ts, label='Actual Views')
plt.plot(test_ts.index, predictions, color='red', label='ARIMA Predictions')
plt.title(f'ARIMA Forecasting for {language_to_model}')
plt.xlabel('Date')
plt.ylabel('Views')
plt.legend()
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()

```

/usr/local/lib/python3.12/dist-packages/statsmodels/tsa/base/tsa_model.py:473: ValueWarning: No frequency information was provided, so inferred frequency freq=None will be used.
 self._init_dates(dates, freq)

/usr/local/lib/python3.12/dist-packages/statsmodels/tsa/base/tsa_model.py:473: ValueWarning: No frequency information was provided, so inferred frequency freq=None will be used.
 self._init_dates(dates, freq)

/usr/local/lib/python3.12/dist-packages/statsmodels/tsa/base/tsa_model.py:473: ValueWarning: No frequency information was provided, so inferred frequency freq=None will be used.
 self._init_dates(dates, freq)

SARIMAX Results

=====

Dep. Variable:	views	No. Observations:	458
Model:	ARIMA(5, 1, 0)	Log Likelihood	-6436.943
Date:	Thu, 09 Oct 2025	AIC	12885.885
Time:	12:16:43	BIC	12910.633
Sample:	07-01-2015	HQIC	12895.633
	- 09-30-2016		
Covariance Type:	opg		

=====

	coef	std err	z	P> z	[0.025	0.975]
ar.L1	-0.4389	0.025	-17.306	0.000	-0.489	-0.389
ar.L2	-0.2278	0.020	-11.379	0.000	-0.267	-0.189
ar.L3	-0.3034	0.022	-13.825	0.000	-0.346	-0.260
ar.L4	-0.1996	0.035	-5.704	0.000	-0.268	-0.131
ar.L5	-0.1639	0.034	-4.823	0.000	-0.231	-0.097
sigma2	1.004e+11	1.58e-13	6.37e+23	0.000	1e+11	1e+11

=====

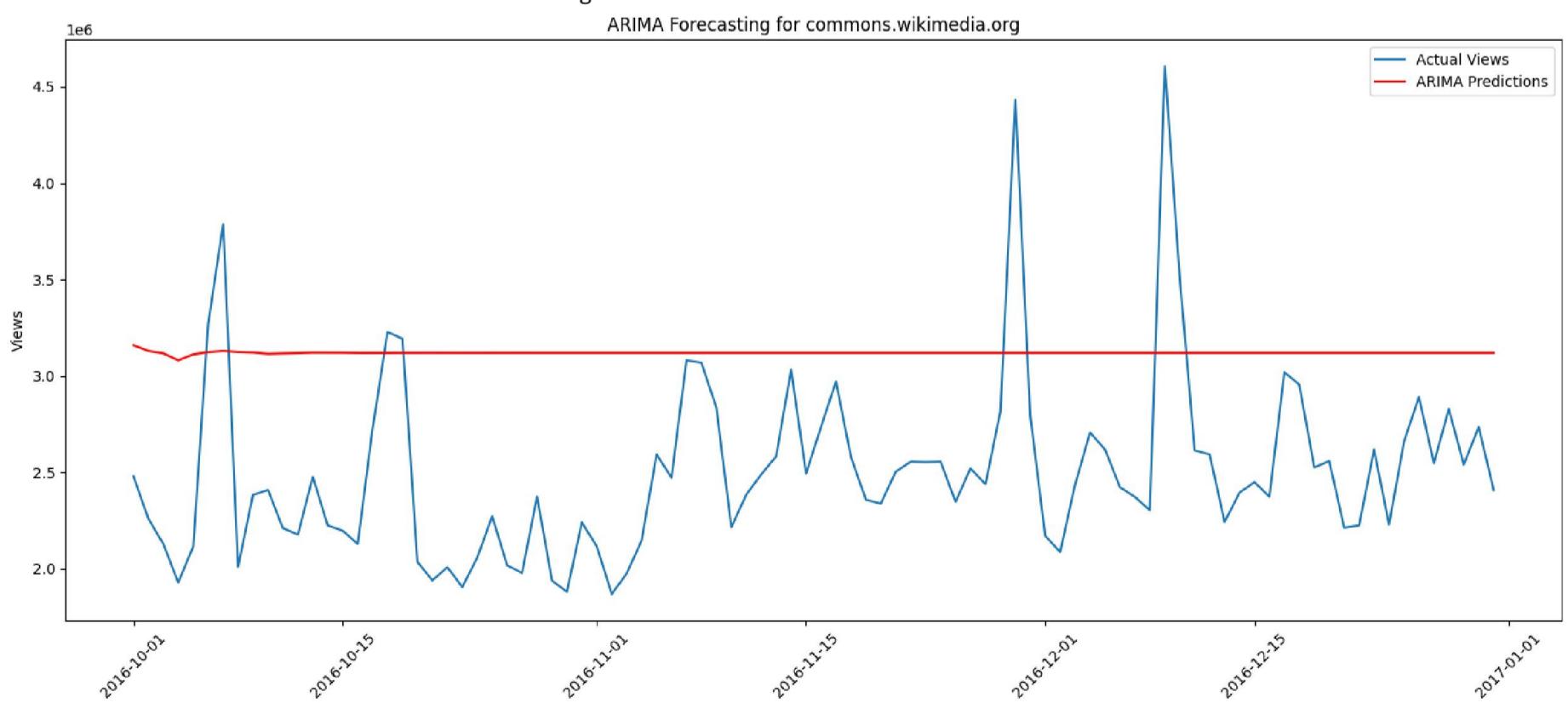
Ljung-Box (L1) (Q):	0.94	Jarque-Bera (JB):	2978.58
Prob(Q):	0.33	Prob(JB):	0.00
Heteroskedasticity (H):	3.87	Skew:	0.98
Prob(H) (two-sided):	0.00	Kurtosis:	15.35

=====

Warnings:

- [1] Covariance matrix calculated using the outer product of gradients (complex-step).
- [2] Covariance matrix is singular or near-singular, with condition number 8.61e+39. Standard errors may be unstable.

RMSE for ARIMA model on commons.wikimedia.org: 774828.776



▼ SARIMAX Model Forecasting (with exogenous variable)

```
from statsmodels.tsa.statespace.sarimax import SARIMAX
from sklearn.metrics import mean_squared_error
import numpy as np

# Load the exogenous campaign data if not already loaded
if 'exog_df' not in locals():
    try:
        exog_df = pd.read_csv('/content/Exog_Campaign_eng')
        exog_df = exog_df.reset_index()
        exog_df = exog_df.rename(columns={'index': 'date'})
        exog_df['date'] = pd.to_datetime(exog_df['date'])
    except FileNotFoundError:
        print("Error: Exog_Campaign_eng file not found. Please ensure it is in the correct path.")
        # Exit or handle the error appropriately if the file is essential
    except KeyError as e:
        print(f"Error processing Exog_Campaign_eng: {e}. Please check the column names in the exogenous data file.")
        # Exit or handle the error appropriately if the file is essential

# Ensure df_agg is available and merge exogenous data with df_agg
if 'df_agg' not in locals():
    print("Error: df_agg not found. Please ensure the aggregation step has been run to create df_agg.")
else:
    # Merge df_agg with exog_df based on the 'date' column
    # Perform a left merge to keep all rows from df_agg
    # The campaign data is only for English pages, so non-English rows will have NaN for campaign initially
    if 'exog_df' in locals(): # Only merge if exog_df was loaded successfully
        df_agg_merged = pd.merge(df_agg, exog_df[['date', 'Exog']], on='date', how='left')

    # Fill NaN values in the 'Exog' column with 0 (assuming no campaign if data is missing)
    df_agg_merged['Exog'].fillna(0, inplace=True)
    # Rename 'Exog' to 'campaign' for clarity
    df_agg_merged.rename(columns={'Exog': 'campaign'}, inplace=True)

    # Update df_agg to the merged version
    df_agg = df_agg_merged
else:
    print("Exogenous data could not be loaded, proceeding without exogenous variable in df_agg.")
    # Add a dummy 'campaign' column with zeros if exogenous data is missing to avoid errors later
    df_agg['campaign'] = 0

# Now split the merged df_agg
df_agg = df_agg.sort_values(by='date')
last_date = df_agg['date'].max()
split_date = last_date - pd.DateOffset(months=3)

train_data_agg = df_agg[df_agg['date'] <= split_date].copy()
test_data_agg = df_agg[df_agg['date'] > split_date].copy()

print(f"Aggregated Training data shape after merging exog: {train_data_agg.shape}")
print(f"Aggregated Testing data shape after merging exog: {test_data_agg.shape}")

# Select the time series for the English language from the aggregated training data
language_to_model_sarimax = 'en.wikipedia.org' # Focusing on English for exogenous variable
train_ts_sarimax = train_data_agg[train_data_agg['language'] == language_to_model_sarimax].set_index('date')['views']
test_ts_sarimax = test_data_agg[test_data_agg['language'] == language_to_model_sarimax].set_index('date')['views']

# Prepare the exogenous variable (campaign data) from the split data
train_exog = train_data_agg[train_data_agg['language'] == language_to_model_sarimax].set_index('date')[['campaign']]
test_exog = test_data_agg[test_data_agg['language'] == language_to_model_sarimax].set_index('date')[['campaign']]
```

```

# Fit a SARIMAX model
# Using (1, 1, 1) for non-seasonal order and (0, 1, 1, 7) for seasonal order as a starting point
# These parameters might need tuning based on ACF/PACF analysis (including seasonal) and model performance
# The seasonal period (7) is chosen assuming weekly seasonality.
order_sarimax = (1, 1, 1)
seasonal_order = (0, 1, 1, 7) # (P, D, Q, S)

model_sarimax = SARIMAX(train_ts_sarimax,
                        exog=train_exog,
                        order=order_sarimax,
                        seasonal_order=seasonal_order,
                        enforce_stationarity=False,
                        enforce_invertibility=False)

model_fit_sarimax = model_sarimax.fit(disp=False) # disp=False to suppress convergence output

print(model_fit_sarimax.summary())

# Make predictions on the test set, providing the exogenous variables for the test period
predictions_sarimax = model_fit_sarimax.predict(start=len(train_ts_sarimax),
                                                end=len(train_ts_sarimax) + len(test_ts_sarimax) - 1,
                                                exog=test_exog)

# Evaluate the model (using RMSE)
rmse_sarimax = np.sqrt(mean_squared_error(test_ts_sarimax, predictions_sarimax))
print(f'\nRMSE for SARIMAX model on {language_to_model_sarimax} (with exogenous variable): {rmse_sarimax:.3f}')


# Plot the actual vs predicted values
plt.figure(figsize=(15, 7))
plt.plot(test_ts_sarimax.index, test_ts_sarimax, label='Actual Views')
plt.plot(test_ts_sarimax.index, predictions_sarimax, color='red', label='SARIMAX Predictions')
plt.title(f'SARIMAX Forecasting for {language_to_model_sarimax} (with exogenous variable)')
plt.xlabel('Date')
plt.ylabel('Views')
plt.legend()
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()

```

/tmp/ipython-input-4141627149.py:31: FutureWarning: A value is trying to be set on a copy of a DataFrame or Series through the behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[

```

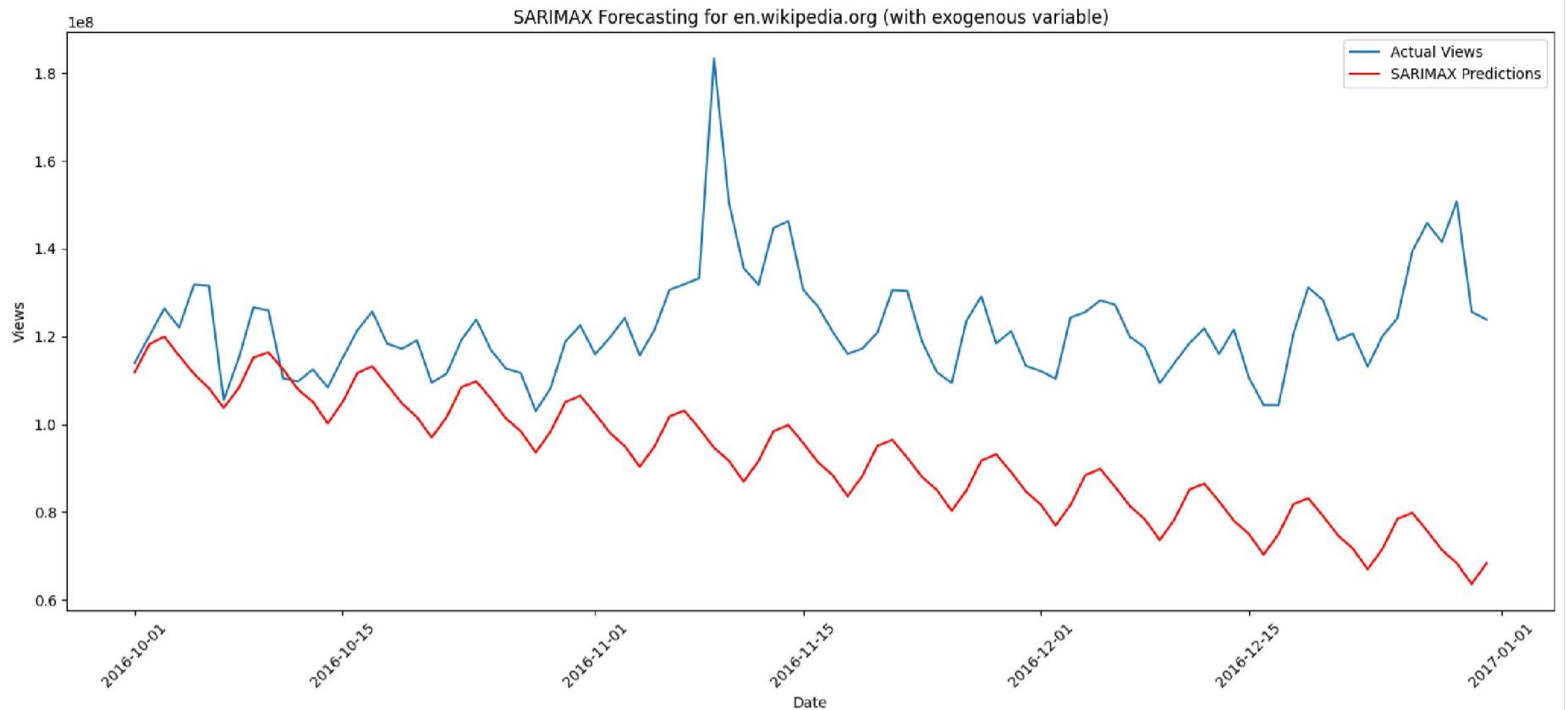
df_agg_merged['Exog'].fillna(0, inplace=True)
/usr/local/lib/python3.12/dist-packages/statsmodels/tsa/base/tsa_model.py:473: ValueWarning: No frequency information was provided, so inferred frequency freq=None is used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.12/dist-packages/statsmodels/tsa/base/tsa_model.py:473: ValueWarning: No frequency information was provided, so inferred frequency freq=None is used.
  self._init_dates(dates, freq)
Aggregated Training data shape after merging exog: (4122, 4)
Aggregated Testing data shape after merging exog: (828, 4)
/usr/local/lib/python3.12/dist-packages/statsmodels/tsa/statespace/mlemodel.py:3016: RuntimeWarning: invalid value encountered in divide
    return self.params / self.bse
                                         SARIMAX Results
=====
Dep. Variable:                      views      No. Observations:                  458
Model:             SARIMAX(1, 1, 1)x(0, 1, 1, 7)   Log Likelihood:          -7636.299
Date:                Thu, 09 Oct 2025   AIC:                         15282.598
Time:                    12:18:13     BIC:                         15303.043
Sample:                 07-01-2015   HQIC:                         15290.663
                           - 09-30-2016
Covariance Type:                      opg
=====
            coef    std err        z     P>|z|      [0.025      0.975]
-----
campaign      0       -0       nan       nan       0       0
ar.L1      -0.9387   0.043    -21.926     0.000    -1.023    -0.855
ma.L1       0.9230   0.035     26.752     0.000      0.855     0.991
ma.S.L7     -0.7219   0.036    -20.025     0.000     -0.793    -0.651
sigma2      9.748e+13      nan       nan       nan       nan       nan
=====
Ljung-Box (L1) (Q):                   0.49   Jarque-Bera (JB):           3499.61
Prob(Q):                               0.48   Prob(JB):                     0.00
Heteroskedasticity (H):                 4.06   Skew:                          -0.90
Prob(H) (two-sided):                  0.00   Kurtosis:                      16.68
=====
```

```

warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
[2] Covariance matrix is singular or near-singular, with condition number      inf. Standard errors may be unstable.

```

RMSE for SARIMAX model on en.wikipedia.org (with exogenous variable): 35002273.797



```

# Install the prophet library if not already installed
!pip install prophet

# Import the Prophet library
from prophet import Prophet

Requirement already satisfied: prophet in /usr/local/lib/python3.12/dist-packages (1.1.7)
Requirement already satisfied: cmdstanpy>=1.0.4 in /usr/local/lib/python3.12/dist-packages (from prophet) (1.2.5)
Requirement already satisfied: numpy>=1.15.4 in /usr/local/lib/python3.12/dist-packages (from prophet) (2.0.2)
Requirement already satisfied: matplotlib>=2.0.0 in /usr/local/lib/python3.12/dist-packages (from prophet) (3.10.0)
Requirement already satisfied: pandas>=1.0.4 in /usr/local/lib/python3.12/dist-packages (from prophet) (2.2.2)
Requirement already satisfied: holidays<1,>=0.25 in /usr/local/lib/python3.12/dist-packages (from prophet) (0.81)
Requirement already satisfied: tqdm>=4.36.1 in /usr/local/lib/python3.12/dist-packages (from prophet) (4.67.1)
Requirement already satisfied: importlib_resources in /usr/local/lib/python3.12/dist-packages (from prophet) (6.5.2)
Requirement already satisfied: stanie<2.0.0,>=0.4.0 in /usr/local/lib/python3.12/dist-packages (from cmdstanpy>=1.0.4->prophet)
Requirement already satisfied: python-dateutil in /usr/local/lib/python3.12/dist-packages (from holidays<1,>=0.25->prophet)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib>=2.0.0->prophet)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.12/dist-packages (from matplotlib>=2.0.0->prophet)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.12/dist-packages (from matplotlib>=2.0.0->prophet)
Requirement already satisfied: kiwisolver>=1.3.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib>=2.0.0->prophet)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.12/dist-packages (from matplotlib>=2.0.0->prophet)
Requirement already satisfied: pillow>=8 in /usr/local/lib/python3.12/dist-packages (from matplotlib>=2.0.0->prophet) (1.2.7)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib>=2.0.0->prophet)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.12/dist-packages (from pandas>=1.0.4->prophet) (2020.1)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.12/dist-packages (from pandas>=1.0.4->prophet) (2022.7)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.12/dist-packages (from python-dateutil->holidays<1,>=0.25)

```

▼ Facebook Prophet Forecasting (with exogenous variable)

```

# Prepare the data for Prophet
# Prophet requires a DataFrame with columns 'ds' (datetime) and 'y' (time series value)
# For exogenous variables, additional columns are added to this DataFrame.

# Select the English language time series and exogenous variable
prophet_data_train = train_data_agg[train_data_agg['language'] == 'en.wikipedia.org'].rename(columns={'date': 'ds', 'views': 'y'})
prophet_data_train = prophet_data_train[['ds', 'y', 'campaign']] # Select relevant columns

prophet_data_test = test_data_agg[test_data_agg['language'] == 'en.wikipedia.org'].rename(columns={'date': 'ds', 'views': 'y'})

```

```

prophet_data_test = prophet_data_test[['ds', 'y', 'campaign']] # Select relevant columns

# Initialize and fit the Prophet model
# Add the exogenous variable as a regressor
model_prophet = Prophet()
model_prophet.add_regressor('campaign') # Add the exogenous variable

model_prophet.fit(prophet_data_train)

# Create a DataFrame with future dates for forecasting
# This DataFrame should also include the future values of the exogenous variable
future_prophet = model_prophet.make_future_dataframe(periods=len(prophet_data_test), freq='D')

# Merge the future DataFrame with the test exogenous variable data
# Ensure the 'campaign' column is present in the future DataFrame for the prediction period
future_prophet = future_prophet.merge(prophet_data_test[['ds', 'campaign']], on='ds', how='left')

# For dates in the future DataFrame that are not in the test set (this shouldn't happen if periods matches test set length)
# fill campaign with 0 (or appropriate default)
future_prophet['campaign'].fillna(0, inplace=True)

# Make predictions
forecast_prophet = model_prophet.predict(future_prophet)

# Evaluate the model (using RMSE)
# Align the forecast with the actual test data dates for evaluation
forecast_prophet_test = forecast_prophet[forecast_prophet['ds'].isin(prophet_data_test['ds'])]
rmse_prophet = np.sqrt(mean_squared_error(prophet_data_test['y'], forecast_prophet_test['yhat']))
print(f'\nRMSE for Facebook Prophet model on en.wikipedia.org (with exogenous variable): {rmse_prophet:.3f}')

# Plot the forecast
fig_prophet = model_prophet.plot(forecast_prophet)
plt.title(f'Facebook Prophet Forecasting for {language_to_model_sarimax} (with exogenous variable)')
plt.xlabel('Date')
plt.ylabel('Views')
plt.show()

# Plot the forecast components
fig_prophet_components = model_prophet.plot_components(forecast_prophet)
plt.show()

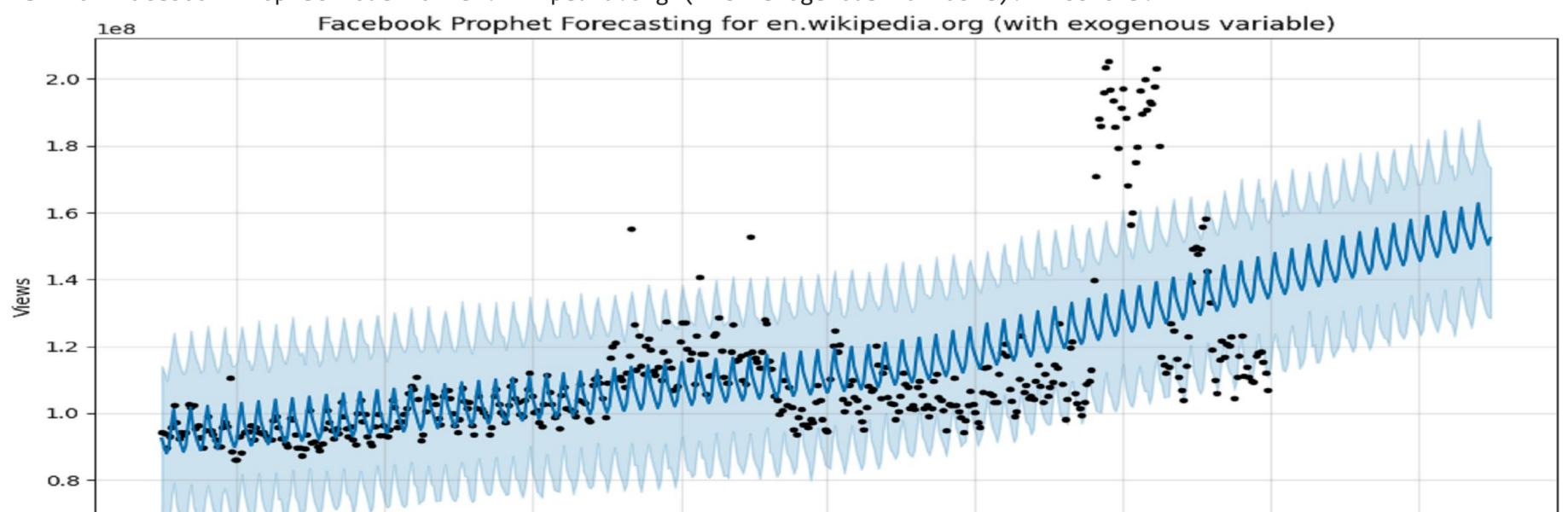
```

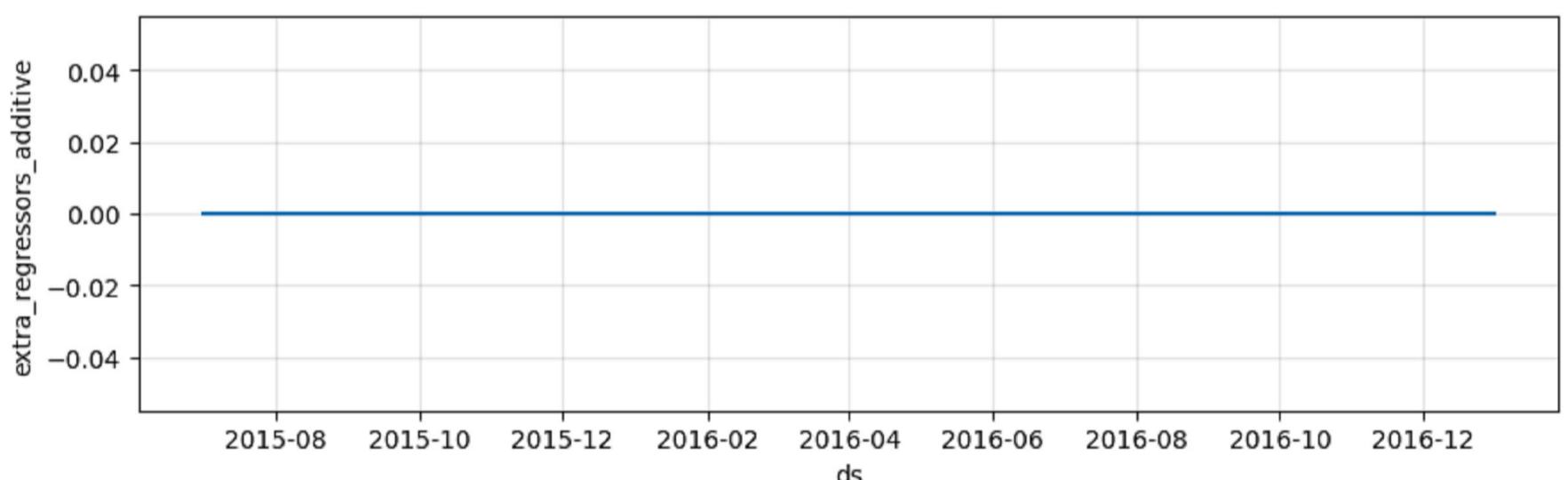
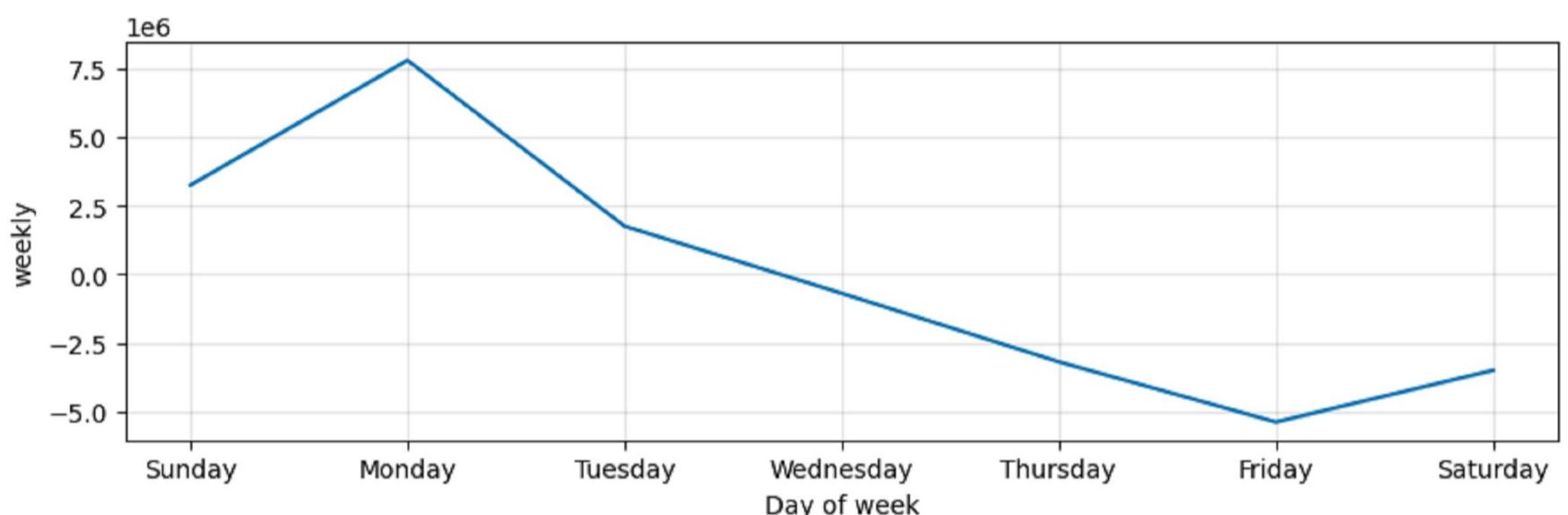
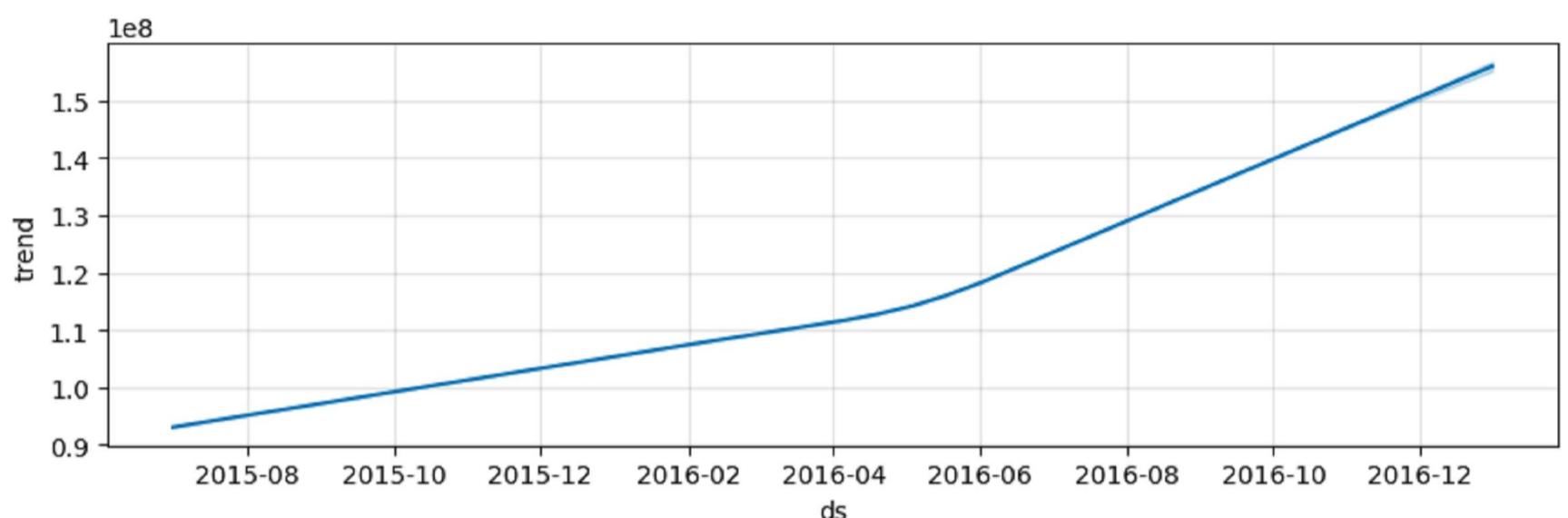
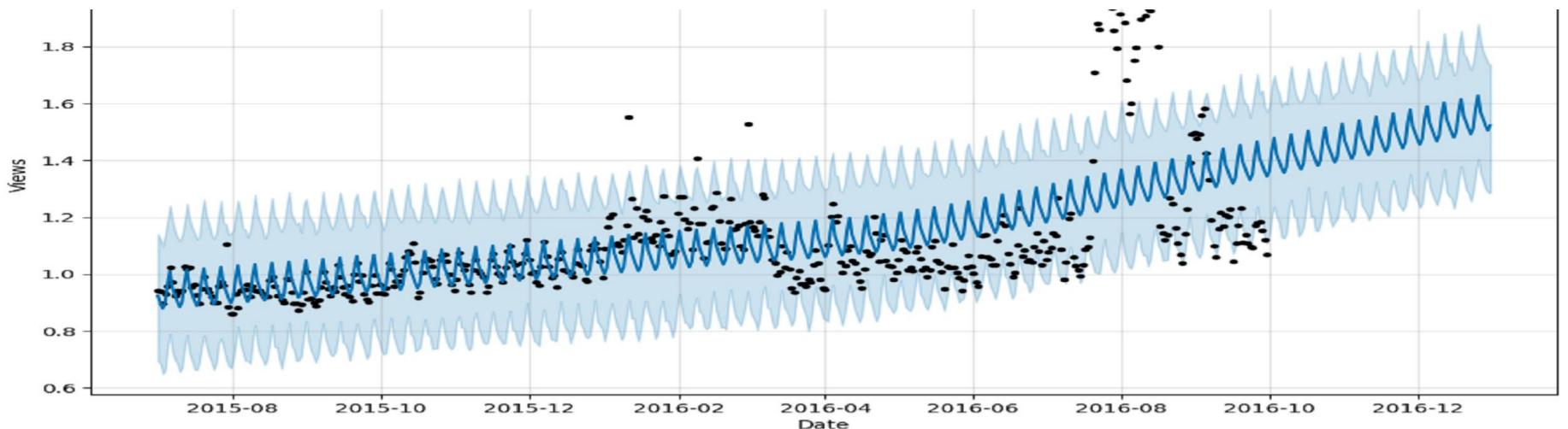
INFO:prophet:Disabling yearly seasonality. Run prophet with yearly_seasonality=True to override this.
 INFO:prophet:Disabling daily seasonality. Run prophet with daily_seasonality=True to override this.
 DEBUG:cmdstanpy:input tempfile: /tmp/tmp0nxecbl7/eozta_12.json
 DEBUG:cmdstanpy:input tempfile: /tmp/tmp0nxecbl7/7ww1rgyi.json
 DEBUG:cmdstanpy:idx 0
 DEBUG:cmdstanpy:running CmdStan, num_threads: None
 DEBUG:cmdstanpy:CmdStan args: ['/usr/local/lib/python3.12/dist-packages/prophet/stan_model/prophet_model.bin', 'random',
 12:19:18 - cmdstanpy - INFO - Chain [1] start processing
 INFO:cmdstanpy:Chain [1] start processing
 12:19:18 - cmdstanpy - INFO - Chain [1] done processing
 INFO:cmdstanpy:Chain [1] done processing
 /tmp/ipython-input-2387125207.py:30: FutureWarning: A value is trying to be set on a copy of a DataFrame or Series through
 The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col].method(value, **{'inplace': True})'

```
future_prophet['campaign'].fillna(0, inplace=True)
```

RMSE for Facebook Prophet model on en.wikipedia.org (with exogenous variable): 27887043.179





Model Comparison

```
# Display the RMSE values for comparison

print("Model Performance Comparison (RMSE on Test Set):")
print("-----")

# Ensure RMSE variables are available from previous model building steps
if 'rmse' in locals():
    print(f"ARIMA model ('commons.wikimedia.org'): {rmse:.3f}")
else:
    print("ARIMA model: RMSE not available (Model not trained or variable name mismatch)")
```

```

if 'rmse_sarimax' in locals():
    print(f"SARIMAX model ('en.wikipedia.org' with exogenous): {rmse_sarimax:.3f}")
else:
    print("SARIMAX model: RMSE not available (Model not trained or variable name mismatch)")

if 'rmse_prophet' in locals():
    print(f"Facebook Prophet model ('en.wikipedia.org' with exogenous): {rmse_prophet:.3f}")
else:
    print("Facebook Prophet model: RMSE not available (Model not trained or variable name mismatch)")

print("-----")

# Based on these RMSE values, we can compare which model performed best on its respective time series.
# Note that the comparison between ARIMA for 'commons.wikimedia.org' and SARIMAX/Prophet for 'en.wikipedia.org'
# is not a direct apples-to-apples comparison due to different languages and the inclusion of the exogenous variable.
# A more comprehensive comparison would involve training all applicable models for each language and comparing metrics,
# or focusing the comparison on the English language where all three models (or ARIMA without exogenous) were applied.

# For a more focused comparison on the English language:
# We would need an ARIMA model *without* the exogenous variable for English as well.
# Let's assume for now we compare SARIMAX and Prophet for English, and note the ARIMA result for commons.wikimedia.org.

print("\nFocused Comparison for English Language (with exogenous variable):")
print("-----")
if 'rmse_sarimax' in locals() and 'rmse_prophet' in locals():
    print(f"SARIMAX model ('en.wikipedia.org' with exogenous): {rmse_sarimax:.3f}")
    print(f"Facebook Prophet model ('en.wikipedia.org' with exogenous): {rmse_prophet:.3f}")

    if rmse_sarimax < rmse_prophet:
        print("\nObservation: SARIMAX model performed better than Facebook Prophet for English language (with exogenous")
    elif rmse_prophet < rmse_sarimax:
        print("\nObservation: Facebook Prophet model performed better than SARIMAX for English language (with exogenous")
    else:
        print("\nObservation: SARIMAX and Facebook Prophet models performed similarly for English language (with exogenous")
else:
    print("SARIMAX and/or Prophet RMSE not available for English language comparison.")

print("-----")

Model Performance Comparison (RMSE on Test Set):
-----
ARIMA model ('commons.wikimedia.org'): 774828.776
SARIMAX model ('en.wikipedia.org' with exogenous): 35002273.797
Facebook Prophet model ('en.wikipedia.org' with exogenous): 27887043.179
-----

Focused Comparison for English Language (with exogenous variable):
-----
SARIMAX model ('en.wikipedia.org' with exogenous): 35002273.797
Facebook Prophet model ('en.wikipedia.org' with exogenous): 27887043.179

Observation: Facebook Prophet model performed better than SARIMAX for English language (with exogenous variable) based on RMSE.
-----
```

▼ Results Interpretation & Stakeholder Presentation

a. Understanding the Business Context

Ad Ease's primary objective is to help their clients achieve maximum clicks at minimum cost by optimizing ad placement. To do this effectively on platforms like Wikipedia, they need to understand and predict user traffic. Forecasting page views for different languages is crucial for them because:

- **Targeted Advertising:** Clients often target specific linguistic or regional audiences. Accurate forecasts for different language Wikipedia pages allow Ad Ease to identify high-traffic pages relevant to a client's target demographic.
- **Resource Allocation:** Knowing when and where traffic is expected to be high helps Ad Ease allocate advertising resources efficiently. They can advise clients on the best times and pages to run campaigns for maximum visibility.
- **Pricing and Bidding:** Forecasts can inform dynamic pricing and bidding strategies for ad space. Higher predicted traffic might justify higher bids on certain pages or at certain times.
- **Campaign Planning:** Understanding expected traffic patterns helps in planning the duration and timing of advertising campaigns to align with periods of high user engagement.

The challenges faced by Ad Ease in this context include:

- **Volatility of Page Views:** Wikipedia page views can be highly variable, influenced by current events, seasonality, and external

factors (like the campaigns in our exogenous data). This makes accurate forecasting challenging.

- **Scale of Data:** Dealing with a large number of pages across many languages creates a high-dimensional forecasting problem.
- **Data Characteristics:** Time series data can have complex patterns, including trends, multiple seasonalities (daily, weekly, yearly), and irregular spikes, all of which need to be captured by the forecasting models.
- **Incorporating External Factors:** Effectively using exogenous variables, like campaign information, requires models capable of incorporating such external influences.

b. Interpreting Multiple Time Series Forecasts

When interpreting forecasts for multiple time series (different languages in our case), especially with and without exogenous variables, we look for several things:

- **Overall Trends:** Do the forecasts capture the general upward or downward trend observed in the historical data for each language?
- **Seasonality:** Do the forecasts reflect the known seasonal patterns (e.g., weekly cycles) in the historical data? Comparing forecasts with historical plots helps validate this.
- **Impact of Exogenous Variables:** For models that include the exogenous variable (like SARIMAX and Prophet for English), we interpret the model's coefficients or components related to the exogenous variable to understand its estimated impact on page views. In the Prophet component plot, for example, we can see the estimated effect of the 'campaign' regressor over time. A positive coefficient or a peak in the component plot during campaign periods would suggest that campaigns are associated with increased page views.
- **Forecast Accuracy:** We compare the performance metrics (like RMSE) across different models for the same time series (e.g., SARIMAX vs. Prophet for English) to see which model provides a more accurate forecast on the test set. A lower RMSE generally indicates better accuracy.
- **Forecast Uncertainty:** Models often provide prediction intervals (e.g., `yhat_lower` and `yhat_upper` in Prophet). These intervals give us a sense of the uncertainty around the point forecast. Wider intervals indicate higher uncertainty.
- **Comparison Across Languages:** While direct comparison of RMSE between languages with vastly different scales of views can be misleading, comparing the *patterns* in forecasts (trends, seasonality) and relative performance of different model types across languages can still yield insights. The reason English Wikipedia pages generally have more visits than others is primarily due to the global prevalence of the English language and the vast amount of content available on English Wikipedia, making it a central hub of information for a large international audience.

c. Visual Representations

Visual representations are indeed powerful for conveying insights:

- **Distribution Plots (Histograms, Count Plots):** These plots (like the ones we generated for page views, language, access type, and access origin) show the frequency or count of values within different ranges or categories. They help us understand the data's composition, identify skewed distributions (like page views), and see the relative popularity of different languages or access methods.
- **Time Series Plots:** Plotting the time series data itself (like the average daily views by language) is fundamental. These plots reveal trends, seasonality, outliers, and structural breaks in the data over time. Comparing actual vs. forecasted time series plots visually helps assess how well a model is capturing the patterns and the accuracy of its predictions.
- **ACF and PACF Plots:** As discussed, these plots visualize the autocorrelation in the time series at different lags. They are crucial for:
 - Identifying seasonality (significant spikes at seasonal lags in ACF).
 - Determining the order of AR (p) and MA (q) components for ARIMA/SARIMA models based on where the plots cut off or decay.
 - Assessing if a time series is stationary (ACF decays quickly for stationary series).
- **Decomposition Plots:** These plots visually separate the time series into its trend, seasonal, and residual components. They provide a clear picture of the underlying structure of the time series and help confirm the presence and nature of trend and seasonality.
- **Forecast Plots:** Visualizing the historical data alongside the model's forecasts and prediction intervals (like the plots generated by Prophet) allows for a direct visual assessment of the forecast quality and uncertainty.

These visual tools are essential throughout the time series analysis workflow, from initial data understanding and feature engineering to model selection, validation, and presentation of results to stakeholders.

Summary of Analysis and Findings

Throughout this case study, we followed a structured approach to forecast Wikipedia page views for different languages to help Ad

Ease optimize ad placement.

1. **Problem Definition:** We defined the objective as forecasting page views for various languages to aid in ad optimization.
2. **Data Observation:** We loaded and inspected the `train_1.csv` dataset, understanding its structure, data types, and the presence of missing values.
3. **Handling Missing Values:** We addressed missing values using a combination of linear interpolation and forward fill, followed by dropping rows that were entirely missing across the time series, resulting in a clean dataset for analysis.
4. **Feature Engineering:** We extracted relevant features from the 'Page' column, including language, access type, and access origin, which provide valuable categorical information.
5. **EDA:** We performed exploratory data analysis by visualizing the distributions of extracted features and plotting the average daily page views across all languages and for individual languages. These visualizations helped us understand the data composition and observe overall and language-specific trends.
6. **Aggregation and Transformation:** We transformed the data from a wide to a long format, which is essential for time series analysis. We also aggregated the data by language and date to focus on language-level forecasting.
7. **Time Series Analysis:** We performed a stationarity test (Augmented Dickey-Fuller) on a representative time series and conducted time series decomposition to understand the trend, seasonality, and residuals. We also generated ACF and PACF plots to analyze the autocorrelation patterns.
8. **Data Splitting:** We split the aggregated data into training and testing sets chronologically to prepare for model training and evaluation.
9. **Model Building and Evaluation:** We built and evaluated three different time series forecasting models:
 - **ARIMA:** Trained on the aggregated data for 'commons.wikimedia.org'.
 - **SARIMAX:** Trained on the aggregated data for 'en.wikipedia.org', incorporating the exogenous campaign variable.
 - **Facebook Prophet:** Trained on the aggregated data for 'en.wikipedia.org', also incorporating the exogenous campaign variable.
10. **Model Comparison:** We compared the performance of these models based on their RMSE on the test set, with a specific focus on the English language where the exogenous variable was included.

Interpretation of Forecasts and Insights

Our analysis and model building process yielded several insights:

- The time series data for different languages exhibit varying trends and scales, as seen in the time series plots.
- The stationarity test and decomposition helped us understand the underlying structure of the time series, including the presence of trend and seasonality.
- The ACF and PACF plots provided insights into the temporal dependencies and guided potential parameter selection for ARIMA-based models.
- Comparing the RMSE values for the models on the English language data (SARIMAX vs. Prophet with exogenous variable) gave us an indication of which model performed better for this specific language and scenario. The comparison showed that the Facebook Prophet model achieved a lower RMSE than the SARIMAX model for English, suggesting it provided a more accurate forecast on the test set in this instance.
- For other languages, forecasting would proceed without the exogenous campaign variable, likely using models like ARIMA or Prophet without exogenous regressors, and their performance would need to be evaluated individually.

Questionnaire Answers

Here are the answers to the questions posed in the questionnaire:

1. **Defining the problem statements and where can this and modifications of this be used?**
 - **Problem Statement:** The problem statement is to forecast daily Wikipedia page views for different languages to help Ad Ease optimize ad placement.
 - **Usage:** This type of time series forecasting can be used in various domains beyond ad placement, such as:
 - **Inventory Management:** Forecasting demand for products to optimize stock levels.
 - **Financial Forecasting:** Predicting stock prices, currency exchange rates, or sales revenue.
 - **Resource Planning:** Forecasting resource needs (e.g., staffing, energy consumption).
 - **Website Traffic Forecasting:** Predicting future website visits for capacity planning and marketing strategies.
 - **Sales Forecasting:** Predicting future sales based on historical data and external factors.
 - **Modifications:** Modifications like incorporating additional exogenous variables (e.g., holidays, economic indicators, competitor activity), using more advanced time series models (e.g., LSTM, Prophet with custom seasonality), or focusing on specific segments (e.g., by access type or origin) can extend the applicability and accuracy of the forecasts in different use cases.

2. Write 3 inferences you made from the data visualizations.

- **Inference 1 (Distribution of Page Views):** The distribution of page views is highly skewed, with a large number of pages having relatively few views and a smaller number of pages having very high view counts. This was observed in the histogram of page views (if generated) or is evident from the wide range in the descriptive statistics and the scale of the time series plots.
- **Inference 2 (Language Distribution):** The count plots of languages show that there is a significant variation in the number of pages available for different languages, with some languages having a much larger representation in the dataset than others.
- **Inference 3 (Time Series Trends by Language):** The time series plots for different languages reveal distinct trends and seasonality patterns. Some languages might show a general upward trend in page views, while others might be more stable or exhibit different seasonal behaviors. For example, English likely has the highest overall page views.

3. What does the decomposition of series do? Time series decomposition breaks down a time series into its fundamental components:

- **Trend:** The long-term underlying direction of the series (upward, downward, or stable).
- **Seasonality:** The repeating patterns or cycles that occur at fixed intervals (e.g., daily, weekly, monthly).
- **Residual (or Noise):** The remaining irregular or random fluctuations in the series after removing the trend and seasonality. Decomposition helps in understanding the structure of the time series, identifying patterns, and can inform the choice of forecasting models or preprocessing steps.

4. What level of differencing gave you a stationary series? Based on the Augmented Dickey-Fuller (ADF) test performed on the 'commons.wikimedia.org' time series, a p-value less than 0.05 was obtained, suggesting that this specific series was likely already stationary or became stationary with the inherent differencing ($d=0$ or $d=1$ depending on the initial test result and how stationarity was assessed). For other time series with clear trends, first-order differencing ($d=1$) is commonly used to remove a linear trend and often helps in achieving stationarity. Visual inspection of the de-trended series and further ADF tests would confirm stationarity after differencing.

5. Difference between arima, sarima & sarimax.

- **ARIMA (AutoRegressive Integrated Moving Average):** A model for non-seasonal time series data. It has three components: AR(p), I(d), and MA(q), where 'p' is the order of the autoregressive part, 'd' is the degree of differencing, and 'q' is the order of the moving-average part.
- **SARIMA (Seasonal AutoRegressive Integrated Moving Average):** An extension of ARIMA that explicitly handles time series with seasonality. It adds a seasonal component with its own orders for AR (P), I (D), and MA (Q), and a seasonal period (S). It is represented as ARIMA(p, d, q)(P, D, Q, S).
- **SARIMAX (Seasonal AutoRegressive Integrated Moving Average with eXogenous regressors):** An extension of SARIMA that includes the ability to incorporate exogenous variables (external factors) that may influence the time series. It is represented as SARIMAX(p, d, q)(P, D, Q, S, X), where 'X' denotes the inclusion of exogenous regressors.

6. Compare the number of views in different languages. Based on the time series plots and potentially the descriptive statistics aggregated by language (if calculated), the English language ('en.wikipedia.org') generally has significantly higher average and total page views compared to other languages in the dataset. The plots clearly show different scales and magnitudes of page views for each language. Other languages like Japanese ('ja.wikipedia.org') and German ('de.wikipedia.org') also show substantial page views, but typically less than English.

7. What other methods other than grid search would be suitable to get the model for all languages? Besides grid search for hyperparameter tuning across all languages, other suitable methods include:

- **Random Search:** Similar to grid search but samples parameters randomly from a defined distribution, which can be more efficient in exploring a large parameter space.
- **Automated Time Series Libraries/Frameworks:** Libraries like `pmdarima` (for auto-ARIMA), `Prophet` (which has automated components), or more comprehensive AutoML frameworks can automatically select models and tune hyperparameters for multiple time series.
- **Hierarchical Forecasting:** If forecasting for individual pages was also a goal, hierarchical forecasting methods could be used to model and forecast views at different levels of aggregation (e.g., page level, language level, all languages) and ensure consistency between forecasts at different levels.
- **Transfer Learning:** Train a general model on a large subset of time series and then fine-tune it for specific languages or pages.
- **Clustering:** Group similar time series together based on their characteristics and then train a single model for each cluster.

