Some literature review from: https://medium.com/thedeephub/mastering-time-series-forecasting-f49f45855c83

Time Series Forecasting Project

In this project, I will show a time series forecasting challenge first with the ARIMA statistical model using two datasets from a gaming company, and then with machine learning. The data has been collected at different time intervals, and is comprised of the amount of game currency and the total number of ad views. The purpose of this challenge to show figure out if I can make a model that will make accurate predictions about the amount of game currency spent based on information about the advertisements run. Some of the methods of time series forecasting I will show include ARIMA (Autoregressive Integrated Moving Average) model, plotting to check stationarity, using seasonal decomposition methods to identify seasonality, trend, and residuals All of these methods can be used to predict future values based on historical data patterns and trends, considering seasonality and other factors depending on the chosen method. The advantage of using time series analysis is that all the methods are statistical, and we know their accuracy, the theorems are mathematically proven. Within the limits of our assumptions about the characteristics of the distributions involved, we can have statistical accuracy and certainty, we can have confidence intervals, and accurate odds. Machine learning methods are roughly based on statistical ideas, but do not depend on any assumptions about the form of distributions. Their form of accuracy is basically finding non-linear equations “that work” to predict known outcomes based on known inputs, and they then get as accurate as the training data allows them to get.

import os

notebook_path = os.getcwd() print(notebook_path)

In [4]:
```
#Loading Libraries
#!pip install nbconvert
#!pip install pandoc
#!pip install matplotlib
#!pip install --upgrade jupyter
#!pip install --upgrade nbconvert
#!pip install chromium
#!pip install -U notebook-as-pdf
#!pip install statsmodels
#!pip install lazypredict
#!pip install seaborn
#!pip install reportlab
#!python -m ensurepip --upgrade

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import os
import seaborn as sns
import pandas as pd
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from sklearn.model_selection import TimeSeriesSplit
```

In [5]:
```
#setting file path
ads_path = 'C:\\Users\\Rbrig\\Downloads\\Final Proj CSV\\ads.csv'
ads = pd.read_csv(ads_path, parse_dates=['Time'])
print(ads.shape)

#display first 5 rows of ads
ads.head()
```

```
(216, 2)
```

Out[5]:

	Time	Ads
0	2017-09-13 00:00:00	80115
1	2017-09-13 01:00:00	79885
2	2017-09-13 02:00:00	89325
3	2017-09-13 03:00:00	101930
4	2017-09-13 04:00:00	121630

In [6]:
```
#define path for currency csv
currency_path = 'C:\\Users\\Rbrig\\Downloads\\Final Proj CSV\\currency.csv'

#load csv
currency = pd.read_csv(currency_path)

#convert the 'time' column to a date/time format
currency['Time'] = pd.to_datetime(currency['Time'], format = '%m/%d/%y')

print(currency.shape)

currency.head()
```

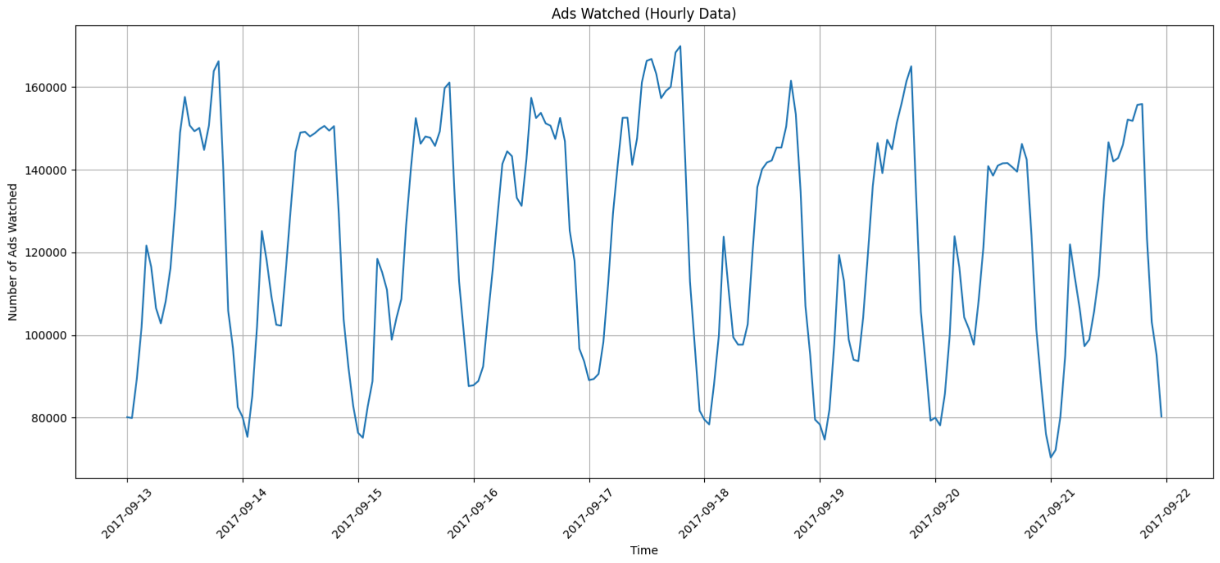
```
(300, 2)
```

Out[6]:

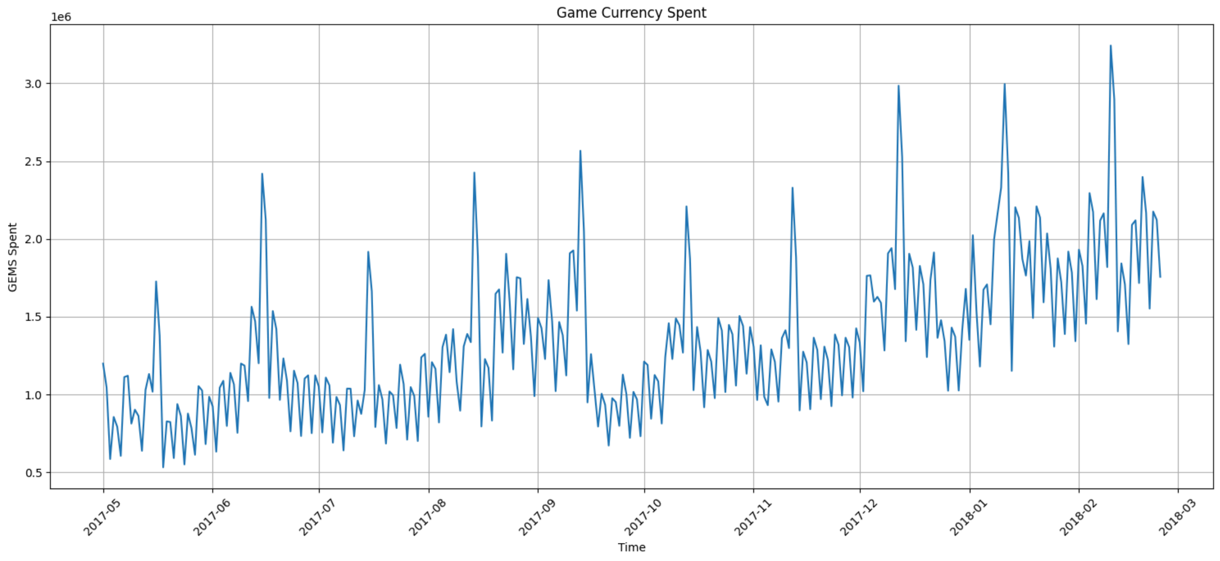
	Time	GEMS_GEMS_SPENT
0	2017-05-01	1199436
1	2017-05-02	1045515
2	2017-05-03	586111
3	2017-05-04	856601
4	2017-05-05	793775

In [7]:
```
#creating plot of ads wathced vs date
fig, ax = plt.subplots(figsize=(15, 7))
sns.lineplot(x=ads['Time'], y=ads['Ads'], data=ads)
plt.grid(True)
plt.title("Ads Watched (Hourly Data)")
plt.xlabel("Time")
plt.ylabel("Number of Ads Watched")
```

```
#rotate x-axis labels for better readability
plt.xticks(rotation=45)

#changed layout to prevent clipping of tick-labels
plt.tight_layout()
plt.show()
```



In [8]:
```
#plot of game currency spent vs time
fig, ax = plt.subplots(figsize=(15, 7))
sns.lineplot(x=currency['Time'], y=currency['GEMS_GEMS_SPENT'], data=currency)
plt.grid(True)
plt.title("Game Currency Spent")
plt.xlabel("Time")
plt.ylabel("GEMS Spent")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



In [9]:
```
#set time as index
ads=ads.set_index('Time')
```

```
#display first 5 rows
ads.head()
```

Out[9]:

Time	Ads
2017-09-13 00:00:00	80115
2017-09-13 01:00:00	79885
2017-09-13 02:00:00	89325
2017-09-13 03:00:00	101930
2017-09-13 04:00:00	121630

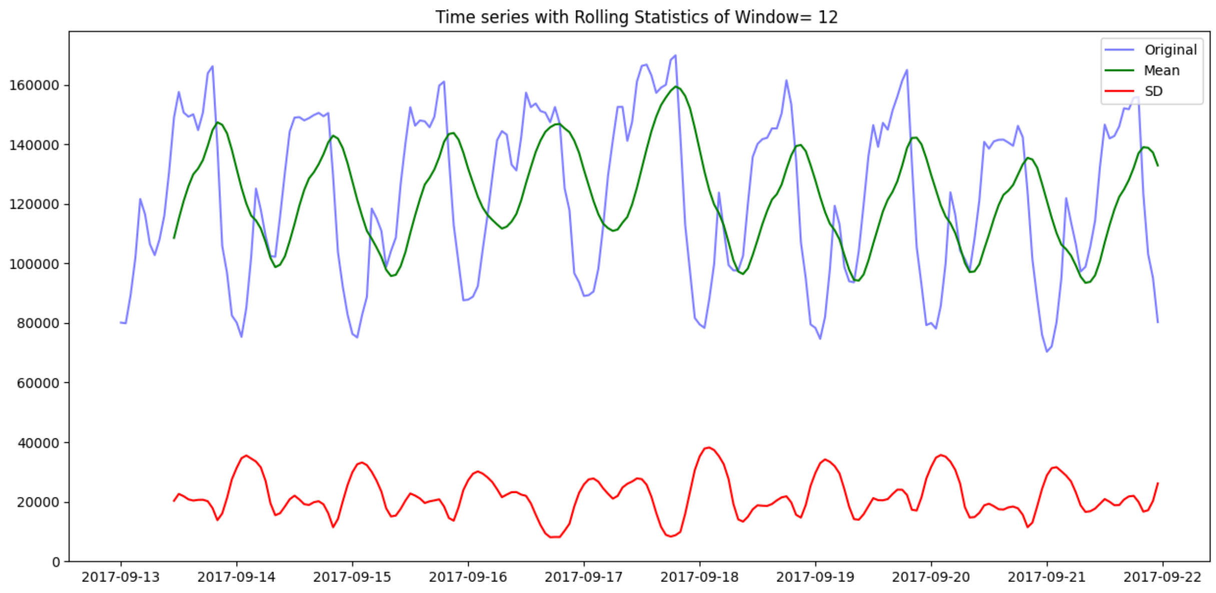
Plot Rolling Window Statistics to check Stationarity

I wrote a functon to plot the standard deviation and mean of a set number of time periods. A Dickey-Fuller Test is also performed to check for stationarity, meaning:

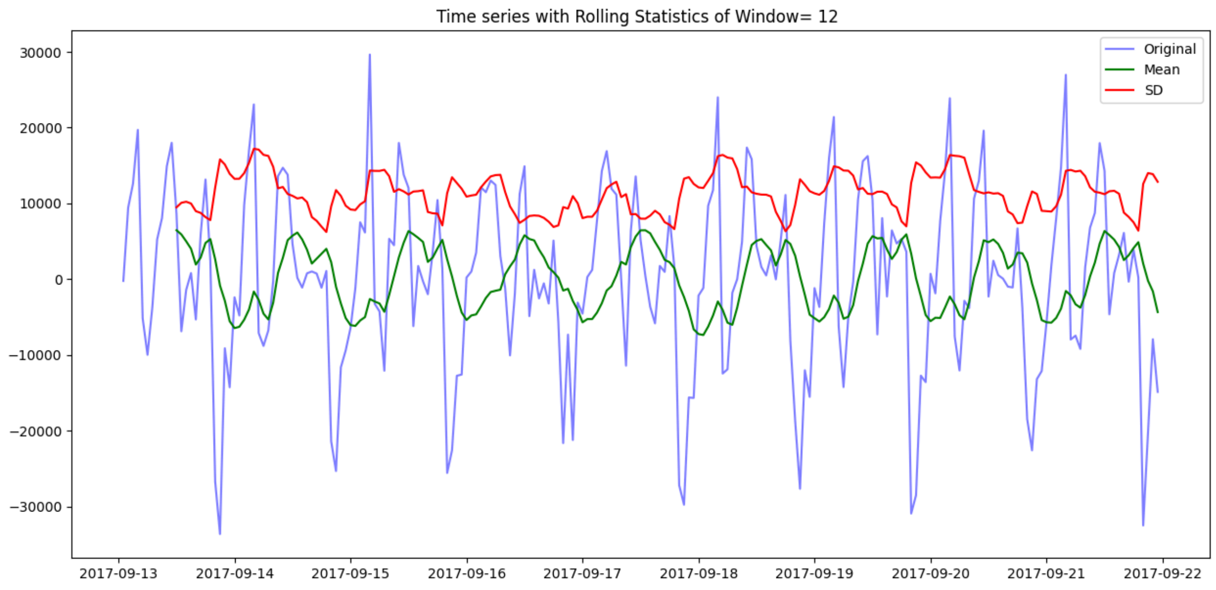
- A flat looking series, without trend, with constant variance over time, and a constant autocorrelation structure over time and no periodic fluctuations.
- If we see seasonality, it is likely to be non-stationary, meaning the test could indicate a unit root

In [10]:
```
def plot_time_series(ts,window):

    from statsmodels.tsa.stattools import adfuller

    #drop any missing values
    ts=ts.dropna()

    #mean and standard deviation
    rolling_mean = ts['Ads'].rolling(window).mean()
    rolling_std  = ts['Ads'].rolling(window).std()

    #plot
    fig, ax= plt.subplots(figsize=(15,7))
    ax.plot(ts['Ads'],color='blue',alpha=0.5)
    ax.plot(rolling_mean,color='green')
    ax.plot(rolling_std,color='red')
    plt.legend(['Original','Mean','SD'],loc='upper right')
    plt.title("Time series with Rolling Statistics of Window= {} ".format(window))
    plt.show()

    #Dickey-Fuller test
    print('\n----------------Dickey-Fuller Test Results:----------------\n')
    dftest = adfuller(ts, autolag='AIC')
    dfoutput = pd.Series(dftest[0:4], index=['Test Statistic','p-value','usedlag','
    for key,value in dftest[4].items():
        dfoutput['Critical Value (%s)'%key] = value
    print(dfoutput)
```

In [11]:
```
plot_time_series(ads,window=12)

#plot ime series w/ first differencing
plot_time_series(ads.diff(),window=12)
```



```
----------------Dickey-Fuller Test Results:----------------

Test Statistic                -7.089634e+00
p-value                        4.444804e-10
usedlag                        9.000000e+00
Number of Observations         2.060000e+02
Critical Value (1%)           -3.462499e+00
Critical Value (5%)           -2.875675e+00
Critical Value (10%)          -2.574304e+00
dtype: float64
```



```
----------------Dickey-Fuller Test Results:----------------

Test Statistic                -6.614309e+00
p-value                        6.262978e-09
usedlag                        1.500000e+01
Number of Observations         1.990000e+02
Critical Value (1%)           -3.463645e+00
Critical Value (5%)           -2.876176e+00
Critical Value (10%)          -2.574572e+00
dtype: float64
```

Dickey-Fuller Test Result

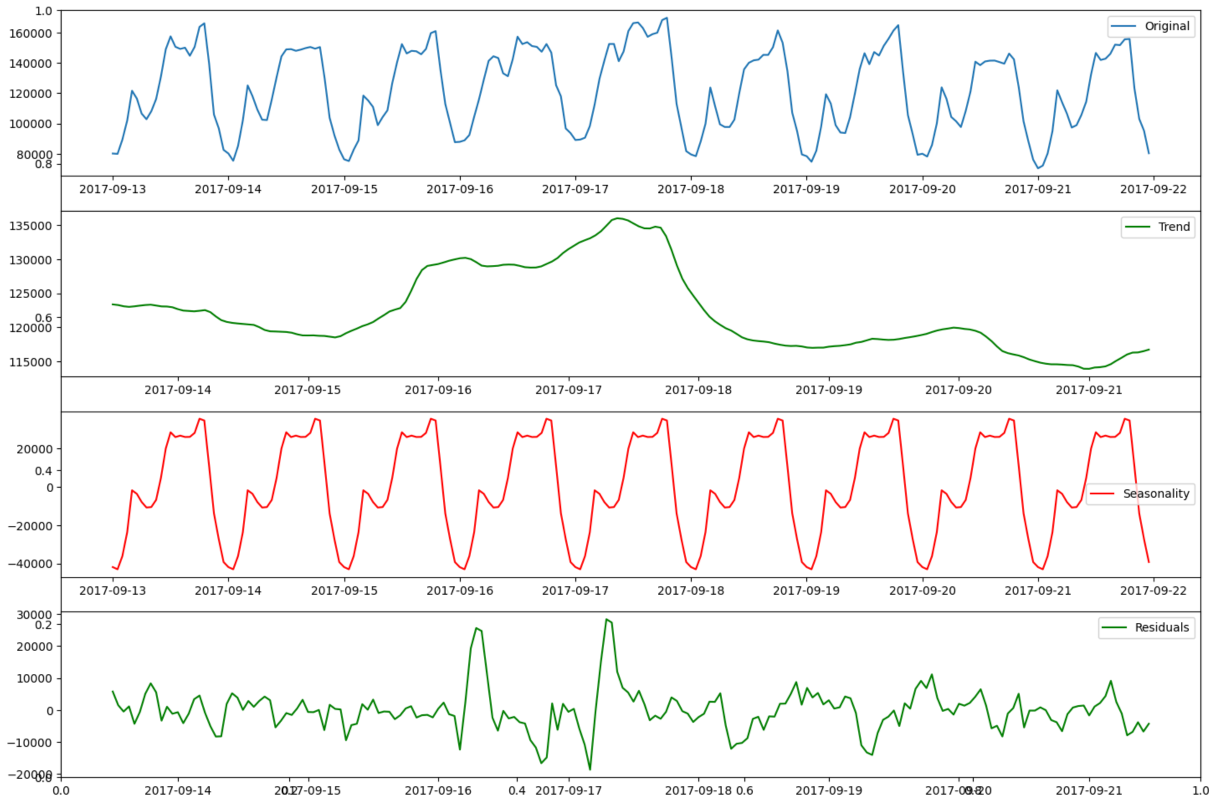
The results of the DF Test show:

- test statistics: more negative statistics means time series is non stationary.
- p-value: This is probability of data in null hypothesis, means how likely the data is non stationary.
- lags used: The number of lags used in the test.
- number of observations: the number of observations used in the test.
- critical values: values used to compare with the ADF statistic to determine the stationarity of the time series at different confidence levels

Decompose Time Series into Seasonality, Trend, and Residuals

I decompose to highlight the seasonality (recurring patterns that repeat at regular intervals), in this case monthly fluctuations, the trend, which refers to the overall long-term direction of the data (increasing, decreasing, or stable), and residuals (the remaining random fluctuations left after removing the trend and seasonality components from the data), which represent noise or unexplained variations.

In [12]:
```
#decompose time series
decomposition = seasonal_decompose(ads)

#extract components
seasonal = decomposition.seasonal
trend = decomposition.trend
residual = decomposition.resid

#plot
fig, ax= plt.subplots(figsize=(15,10))
plt.subplot(411)
plt.plot(ads, label='Original')
plt.legend(loc='best')
plt.subplot(412)
plt.plot(trend, label='Trend',color='green')
plt.legend(loc='best')
plt.subplot(413)
plt.plot(seasonal,label='Seasonality',color='red')
plt.legend(loc='best')
plt.subplot(414)
plt.plot(residual, label='Residuals',color='green')
plt.legend(loc='best')
plt.tight_layout()
```

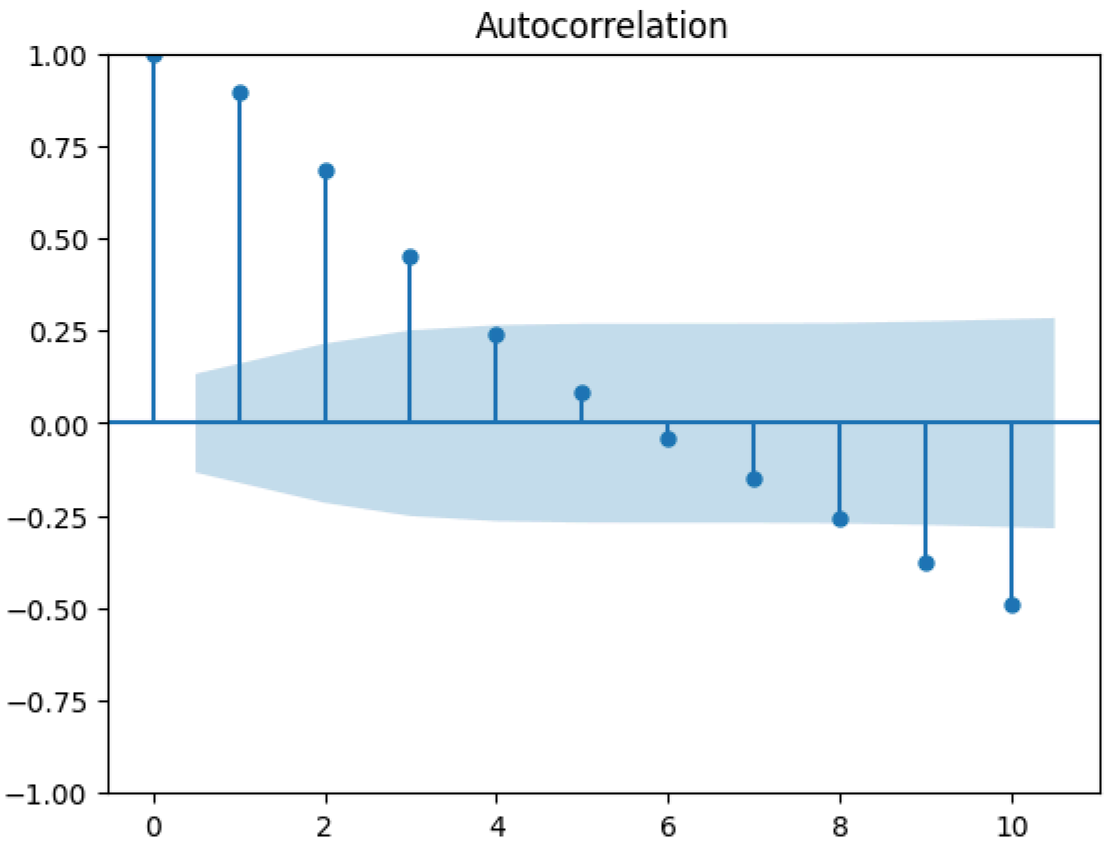


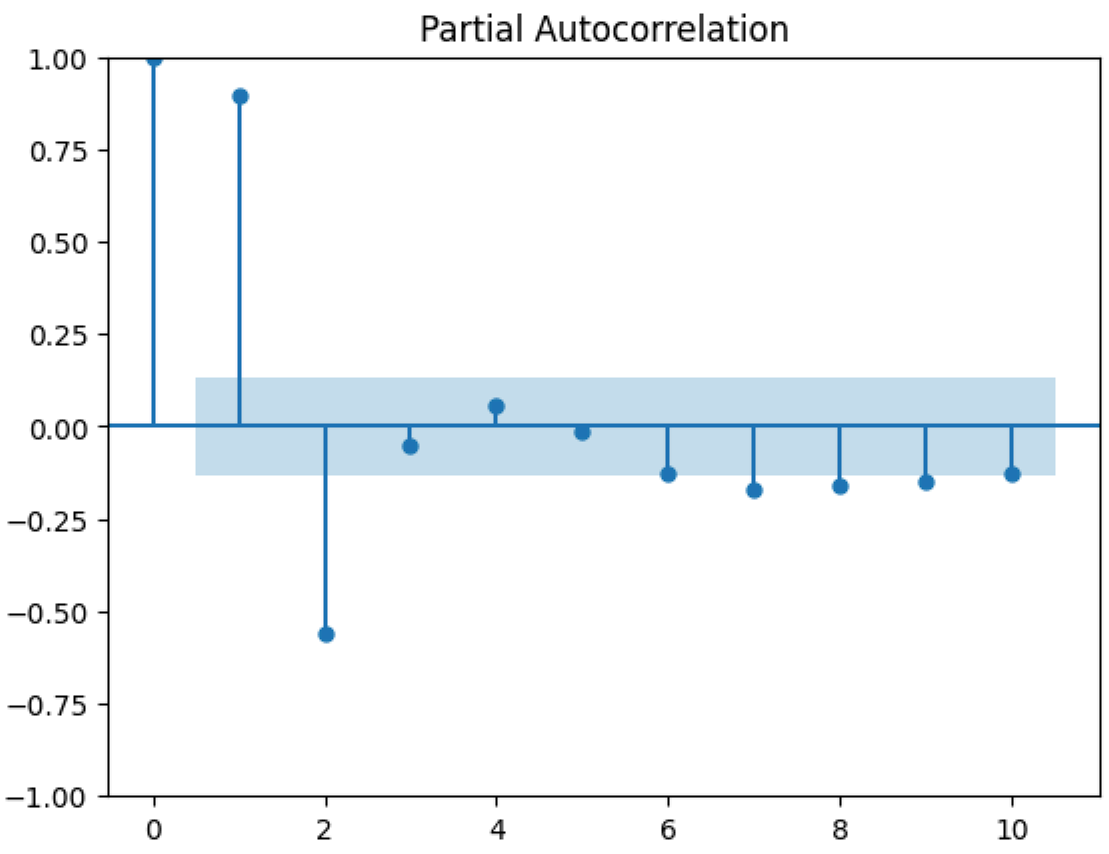
FORECASTING WITH A TIME SERIES MODEL

Plotting the autocorrelation function and partial autocorrelation function helps us to understand the temporal dependencies in the data, which can help in selecting appropriate models for time series forecasting, such as ARIMA.

- The ACF is used to help show the correlation between observations in a series and their past values at different time lags. It provides a measure of the linear relationship between the time series and a lagged version of itself.
- The partial autocorrelation function is used to understand the direct relationship between an observation at a specific time and its lagged values, while removing the influence of shorter lags. This means PACF(x) gives the correlation between the series and itself lagged by ‘x’ time periods, after accounting for the linear dependence on the series at shorter lags, isolating the effect of a specific lag.

In [13]:
```python
#plot autocorrelation function/partial-autocorrelation function
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
plot_acf(ads, lags=10)
plot_pacf(ads, lags=10)
plt.show()
```





When analyzing correlation plots, we look for several key features, inlcuding:

- Positive and Negative Correlation: Positive correlations (which above the horizontal line) indicate a positive autocorrelation at certain lags, which suggest that past values have a similar effect on future values. Negative correlations (below the horizontal line) show a negative autocorrelation, suggesting that past values have an opposite effect on future values.
- Decay in Correlation: A common observation is the decay in correlations as lags increase, indicating that past observations have a diminishing impact on current values. This pattern suggests that the influence of past observations on future values decreases over time.
- Significant Peaks: Noticing significant peaks crossing the horizontal line at certain lags can mean seasonality or periodicity within the data.

Select an ARIMA model and train it

An ARIMA model works with stationary data, meaning our mean and standard deviation remain relatively constant over time.

In [14]:
```
ads_train = ads[:217]
ads_test = ads[217:]
```

In [15]:
```
arima_mod20 = ARIMA(ads_train, order=(2, 0, 2)).fit()
```

```
C:\Users\Rbrig\AppData\Roaming\Python\Python313\site-packages\statsmodels\tsa\base\t
sa_model.py:473: ValueWarning: No frequency information was provided, so inferred fr
equency h will be used.
  self._init_dates(dates, freq)
C:\Users\Rbrig\AppData\Roaming\Python\Python313\site-packages\statsmodels\tsa\base\t
sa_model.py:473: ValueWarning: No frequency information was provided, so inferred fr
equency h will be used.
  self._init_dates(dates, freq)
C:\Users\Rbrig\AppData\Roaming\Python\Python313\site-packages\statsmodels\tsa\base\t
sa_model.py:473: ValueWarning: No frequency information was provided, so inferred fr
equency h will be used.
  self._init_dates(dates, freq)
```

In [ ]:
```
from statsmodels.tsa.arima.model import ARIMA

#split data into train and test sets
ads_train = ads[:217]
ads_test = ads[217:]

#ensure ads_train is a 1D array
ads_train_values = ads_train.values.flatten()  # Ensure it's 1D

#fit ARIMA model
model = ARIMA(ads_train_values, order=(2, 0, 2))
results_ARIMA = model.fit()

#generate forecasts for the test set
forecast_steps = len(ads_test)
# Use the `start` and `end` parameters correctly
forecast = results_ARIMA.forecast(steps=forecast_steps)

#function for Mean Absolute Percentage Error
def mean_absolute_percentage_error(y_true, y_pred):
    y_true, y_pred = np.array(y_true), np.array(y_pred)

    #check for zeros in y_true to avoid division by zero
    if np.any(y_true == 0):
        raise ValueError("True values contain zero, MAPE calculation is not possibl
    return np.mean(np.abs((y_true - y_pred) / y_true)) * 100

#calculate and print MAPE if no zeros in ads_test
try:
    mape_value = mean_absolute_percentage_error(ads_test.values.flatten(), forecast
    print('MAPE% = ', mape_value)
except ValueError as e:
    print(e)
```

In [ ]:
```
model = ARIMA(ads_train.iloc[:, 1].values, order=(2, 0, 0))

#model = ARIMA(ads_train.values, order=(2,0,0))
results_ARIMA = model.fit(disp=-1)
y_pred_ar =  results_ARIMA.fittedvalues
```

```
def mean_absolute_percentage_error(y_true, y_pred):
    y_true, y_pred = np.array(y_true), np.array(y_pred)
    return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
print('MAPE% = ',mean_absolute_percentage_error(ads_test,y_pred_ar))
```

Fit ARIMA model with p=2/d=0/q=0

- P is our autoregressive order. This is the number of past observations considered for making future predictions.
- Q is our moving average order. It accounts for a specific number of previous residuals when making future predictions.
- D is our integration order. It determines the number of differences needed to make the time series stationary.

In [ ]:
```
model = ARIMA(ads_train.values, order=(2,0,0))
results_ARIMA = model.fit(disp=-1)
y_pred_ar =  results_ARIMA.fittedvalues

#Mean Absolute Percentage Error
def mean_absolute_percentage_error(y_true, y_pred):
    y_true, y_pred = np.array(y_true), np.array(y_pred)
    return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
print('MAPE% = ',mean_absolute_percentage_error(ads_test,y_pred_ar))
```

FORECASTING TIME SERIES WITH MACHINE LEARNING

In order to enhance the predictive power of the model, I created 3 types of features from the dataset

- Lag Features These are past values of the target variable that are used as inputs for predicting future values.
- Date-Time Features These features extract useful information from the date-time index of the time series, like the day of the week, month, year, or if a date falls on a weekend. These features help capture patterns and trends in the data
- Rolling Window Summary Statistics This involves calculating statistics (such as mean, sum, or standard deviation) over a rolling window of past observations. For example, a rolling average of the past seven days can smooth out short-term fluctuations and highlight longer-term trends.

In [17]:
```
ads.diff().head()
```

Out[17]:

	Ads
Time	
2017-09-13 00:00:00	NaN
2017-09-13 01:00:00	-230.0
2017-09-13 02:00:00	9440.0
2017-09-13 03:00:00	12605.0
2017-09-13 04:00:00	19700.0

In [18]:
```
ads_ts = ads
ads_ts['hour_of_day'] = ads_ts.index.hour
ads_ts.head()
```

Out[18]:

	Ads	hour_of_day
Time		
2017-09-13 00:00:00	80115	0
2017-09-13 01:00:00	79885	1
2017-09-13 02:00:00	89325	2
2017-09-13 03:00:00	101930	3
2017-09-13 04:00:00	121630	4

In [19]:
```
def create_lagged_features(ads_ts,lag):
    for i in range(1,lag+1):
        feature_name = "Ads_lag{}".format(i)
        ads_ts[feature_name]=ads_ts['Ads'].shift(i);
    ads_ts.dropna()
```

In [20]:
```
create_lagged_features(ads_ts,lag=6)
ads_ts = ads_ts.dropna()
ads_ts.head()
```

Out[20]:

Time	Ads	hour_of_day	Ads_lag1	Ads_lag2	Ads_lag3	Ads_lag4	Ads_lag5	Ads_lag
2017-09-13 06:00:00	106495	6	116475.0	121630.0	101930.0	89325.0	79885.0	80115.
2017-09-13 07:00:00	102795	7	106495.0	116475.0	121630.0	101930.0	89325.0	79885.
2017-09-13 08:00:00	108055	8	102795.0	106495.0	116475.0	121630.0	101930.0	89325.
2017-09-13 09:00:00	116125	9	108055.0	102795.0	106495.0	116475.0	121630.0	101930.
2017-09-13 10:00:00	131030	10	116125.0	108055.0	102795.0	106495.0	116475.0	121630.

Create Window Features

In [21]:
```
def create_rolling_window_features(ads_ts,window=1):
    ads_ts['rolling_mean_{}'.format(window)]= ads_ts['Ads'].shift(window-1).rolling
    ads_ts['rolling_std_{}'.format(window)]= ads_ts['Ads'].shift(window-1).rolling(
    ads_ts['rolling_max_{}'.format(window)]= ads_ts['Ads'].shift(window-1).rolling(
    ads_ts['rolling_min_{}'.format(window)]= ads_ts['Ads'].shift(window-1).rolling(
```

In [22]:
```
def create_rolling_window_features(ads_ts, window=3):

    #dtaFrame
    ads_ts = ads_ts.copy()

    #get rolling features
    ads_ts.loc[:, 'rolling_mean_{}'.format(window)] = ads_ts['Ads'].shift(window-1)
    ads_ts.loc[:, 'rolling_std_{}'.format(window)] = ads_ts['Ads'].shift(window-1).
    ads_ts.loc[:, 'rolling_max_{}'.format(window)] = ads_ts['Ads'].shift(window-1).
    ads_ts.loc[:, 'rolling_min_{}'.format(window)] = ads_ts['Ads'].shift(window-1).

    #drop rows with nan values
    ads_ts = ads_ts.dropna()

    return ads_ts
```

In [23]:
```
ads_ts.shape
```

Out[23]:
```
(210, 8)
```

```
In [24]: ads_ts.tail(48)
```

Out[24]:

	Ads	hour_of_day	Ads_lag1	Ads_lag2	Ads_lag3	Ads_lag4	Ads_lag5	Ads_lag
Time								
2017-09-20 00:00:00	79980	0	79270.0	92855.0	105585.0	134090.0	165010.0	161385.
2017-09-20 01:00:00	78110	1	79980.0	79270.0	92855.0	105585.0	134090.0	165010.
2017-09-20 02:00:00	85785	2	78110.0	79980.0	79270.0	92855.0	105585.0	134090.
2017-09-20 03:00:00	100010	3	85785.0	78110.0	79980.0	79270.0	92855.0	105585.
2017-09-20 04:00:00	123880	4	100010.0	85785.0	78110.0	79980.0	79270.0	92855.
2017-09-20 05:00:00	116335	5	123880.0	100010.0	85785.0	78110.0	79980.0	79270.
2017-09-20 06:00:00	104290	6	116335.0	123880.0	100010.0	85785.0	78110.0	79980.
2017-09-20 07:00:00	101440	7	104290.0	116335.0	123880.0	100010.0	85785.0	78110.
2017-09-20 08:00:00	97635	8	101440.0	104290.0	116335.0	123880.0	100010.0	85785.
2017-09-20 09:00:00	108265	9	97635.0	101440.0	104290.0	116335.0	123880.0	100010.
2017-09-20 10:00:00	121250	10	108265.0	97635.0	101440.0	104290.0	116335.0	123880.
2017-09-20 11:00:00	140850	11	121250.0	108265.0	97635.0	101440.0	104290.0	116335.
2017-09-20 12:00:00	138555	12	140850.0	121250.0	108265.0	97635.0	101440.0	104290.
2017-09-20 13:00:00	140990	13	138555.0	140850.0	121250.0	108265.0	97635.0	101440.

	Ads	hour_of_day	Ads_lag1	Ads_lag2	Ads_lag3	Ads_lag4	Ads_lag5	Ads_lag
Time								
2017-09-20 14:00:00	141525	14	140990.0	138555.0	140850.0	121250.0	108265.0	97635.
2017-09-20 15:00:00	141590	15	141525.0	140990.0	138555.0	140850.0	121250.0	108265.
2017-09-20 16:00:00	140610	16	141590.0	141525.0	140990.0	138555.0	140850.0	121250.
2017-09-20 17:00:00	139515	17	140610.0	141590.0	141525.0	140990.0	138555.0	140850.
2017-09-20 18:00:00	146215	18	139515.0	140610.0	141590.0	141525.0	140990.0	138555.
2017-09-20 19:00:00	142425	19	146215.0	139515.0	140610.0	141590.0	141525.0	140990.
2017-09-20 20:00:00	123945	20	142425.0	146215.0	139515.0	140610.0	141590.0	141525.
2017-09-20 21:00:00	101360	21	123945.0	142425.0	146215.0	139515.0	140610.0	141590.
2017-09-20 22:00:00	88170	22	101360.0	123945.0	142425.0	146215.0	139515.0	140610.
2017-09-20 23:00:00	76050	23	88170.0	101360.0	123945.0	142425.0	146215.0	139515.
2017-09-21 00:00:00	70335	0	76050.0	88170.0	101360.0	123945.0	142425.0	146215.
2017-09-21 01:00:00	72150	1	70335.0	76050.0	88170.0	101360.0	123945.0	142425.
2017-09-21 02:00:00	80195	2	72150.0	70335.0	76050.0	88170.0	101360.0	123945.
2017-09-21 03:00:00	94945	3	80195.0	72150.0	70335.0	76050.0	88170.0	101360.

	Ads	hour_of_day	Ads_lag1	Ads_lag2	Ads_lag3	Ads_lag4	Ads_lag5	Ads_lag
Time								
2017-09-21 04:00:00	121910	4	94945.0	80195.0	72150.0	70335.0	76050.0	88170.
2017-09-21 05:00:00	113950	5	121910.0	94945.0	80195.0	72150.0	70335.0	76050.
2017-09-21 06:00:00	106495	6	113950.0	121910.0	94945.0	80195.0	72150.0	70335.
2017-09-21 07:00:00	97290	7	106495.0	113950.0	121910.0	94945.0	80195.0	72150.
2017-09-21 08:00:00	98860	8	97290.0	106495.0	113950.0	121910.0	94945.0	80195.
2017-09-21 09:00:00	105635	9	98860.0	97290.0	106495.0	113950.0	121910.0	94945.
2017-09-21 10:00:00	114380	10	105635.0	98860.0	97290.0	106495.0	113950.0	121910.
2017-09-21 11:00:00	132335	11	114380.0	105635.0	98860.0	97290.0	106495.0	113950.
2017-09-21 12:00:00	146630	12	132335.0	114380.0	105635.0	98860.0	97290.0	106495.
2017-09-21 13:00:00	141995	13	146630.0	132335.0	114380.0	105635.0	98860.0	97290.
2017-09-21 14:00:00	142815	14	141995.0	146630.0	132335.0	114380.0	105635.0	98860.
2017-09-21 15:00:00	146020	15	142815.0	141995.0	146630.0	132335.0	114380.0	105635.
2017-09-21 16:00:00	152120	16	146020.0	142815.0	141995.0	146630.0	132335.0	114380.
2017-09-21 17:00:00	151790	17	152120.0	146020.0	142815.0	141995.0	146630.0	132335.

	Ads	hour_of_day	Ads_lag1	Ads_lag2	Ads_lag3	Ads_lag4	Ads_lag5	Ads_lag
Time								
2017-09-21 18:00:00	155665	18	151790.0	152120.0	146020.0	142815.0	141995.0	146630.
2017-09-21 19:00:00	155890	19	155665.0	151790.0	152120.0	146020.0	142815.0	141995.
2017-09-21 20:00:00	123395	20	155890.0	155665.0	151790.0	152120.0	146020.0	142815.
2017-09-21 21:00:00	103080	21	123395.0	155890.0	155665.0	151790.0	152120.0	146020.
2017-09-21 22:00:00	95155	22	103080.0	123395.0	155890.0	155665.0	151790.0	152120.
2017-09-21 23:00:00	80285	23	95155.0	103080.0	123395.0	155890.0	155665.0	151790.

Train/Test data

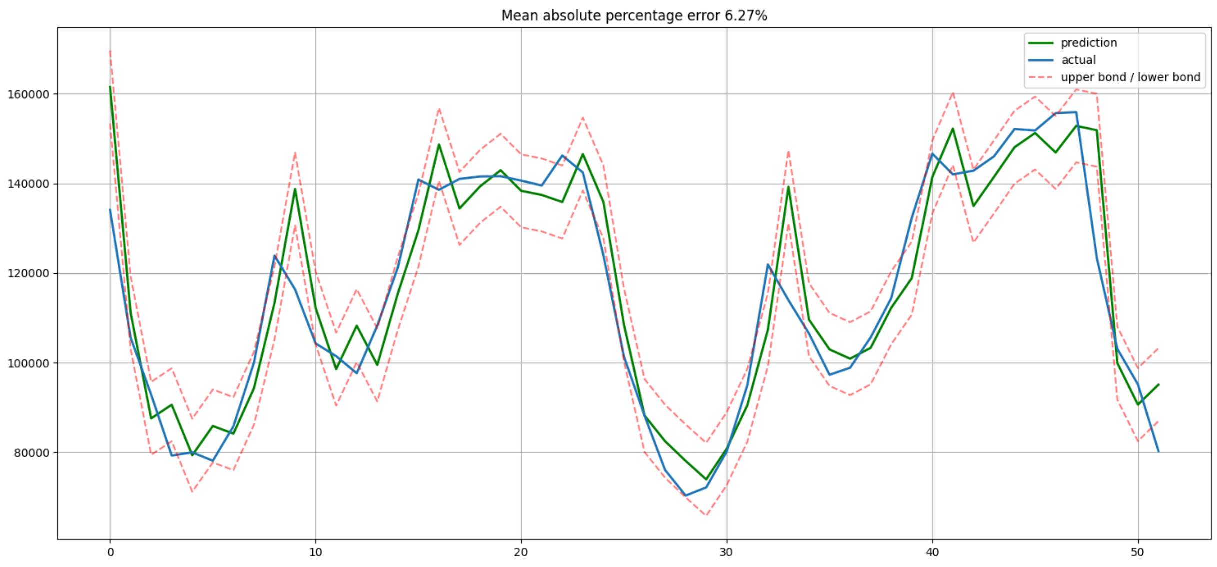
In [25]:
```
from sklearn.linear_model import LinearRegression
y = ads_ts['Ads']
X = ads_ts.drop(['Ads'],axis=1)
X_train = X.iloc[:158,]
y_train = y[:158]
X_test = X.iloc[158:,]
y_test = y[158:]
```

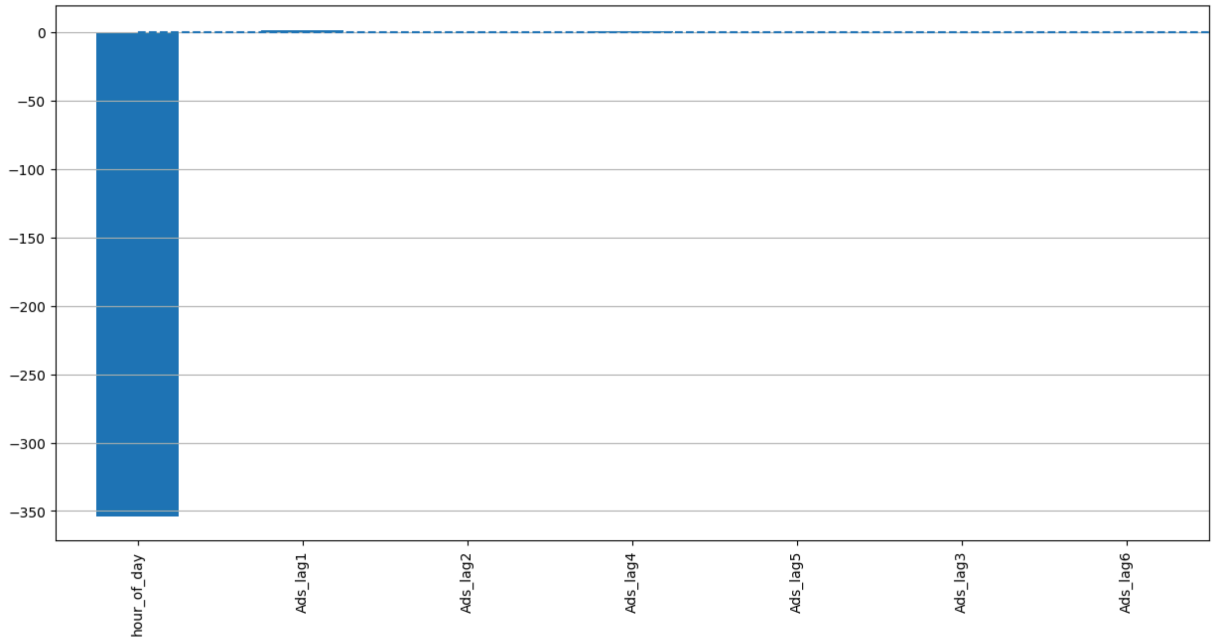
In [26]:
```
lr = LinearRegression()
lr.fit(X_train, y_train)
```

Out[26]:
```
▾ LinearRegression
LinearRegression()
```

In [28]:
```
def plotModelResults(model, X_train=X_train, X_test=X_test, plot_intervals=False, p
    """
        Plots modelled vs fact values, prediction intervals and anomalies

    """
    from sklearn.model_selection import cross_val_score
    from sklearn.model_selection import TimeSeriesSplit
    from sklearn.metrics import r2_score, median_absolute_error, mean_absolute_erro
    from sklearn.metrics import median_absolute_error, mean_squared_error, mean_squ

    #function to calculate the MAPE
    def mean_absolute_percentage_error(y_true, y_pred):
        return np.mean(np.abs((y_true - y_pred) / y_true)) * 100

    #time series cross validation
    tscv = TimeSeriesSplit(n_splits=5)

    #generate predictions
    prediction = model.predict(X_test)
    plt.figure(figsize=(15, 7))
    plt.plot(prediction, "g", label="prediction", linewidth=2.0)
    plt.plot(y_test.values, label="actual", linewidth=2.0)


    #perform cross validation if predictions should be plotted
    if plot_intervals:
        cv = cross_val_score(model, X_train, y_train,
                                    cv=tscv,
                                    scoring="neg_mean_absolute_error")
        mae = cv.mean() * (-1)
        deviation = cv.std()


        #prediction intervals
        scale = 1.96
        lower = prediction - (mae + scale * deviation)
        upper = prediction + (mae + scale * deviation)

        plt.plot(lower, "r--", label="upper bond / lower bond", alpha=0.5)
        plt.plot(upper, "r--", alpha=0.5)

        if plot_anomalies:
            anomalies = np.array([np.nan]*len(y_test))
            anomalies[y_test<lower] = y_test[y_test<lower]
            anomalies[y_test>upper] = y_test[y_test>upper]
            plt.plot(anomalies, "o", markersize=10, label = "Anomalies")

    #get/display the MAPE
    error = mean_absolute_percentage_error(prediction, y_test)
    plt.title("Mean absolute percentage error {0:.2f}%".format(error))
    plt.legend(loc="best")
    plt.tight_layout()
    plt.grid(True);

def plotCoefficients(model):
    """
        Plots sorted coefficient values of the model
    """

    #aataFrame for model coefficients and absolute values
    coefs = pd.DataFrame(model.coef_, X_train.columns)
    coefs.columns = ["coef"]
    coefs["abs"] = coefs.coef.apply(np.abs)
    coefs = coefs.sort_values(by="abs", ascending=False).drop(["abs"], axis=1)

    plt.figure(figsize=(15, 7))
    coefs.coef.plot(kind='bar')
    plt.grid(True, axis='y')
    plt.hlines(y=0, xmin=0, xmax=len(coefs), linestyles='dashed');
```

In [29]:
```
plotModelResults(lr, plot_intervals=True)
plotCoefficients(lr)
```

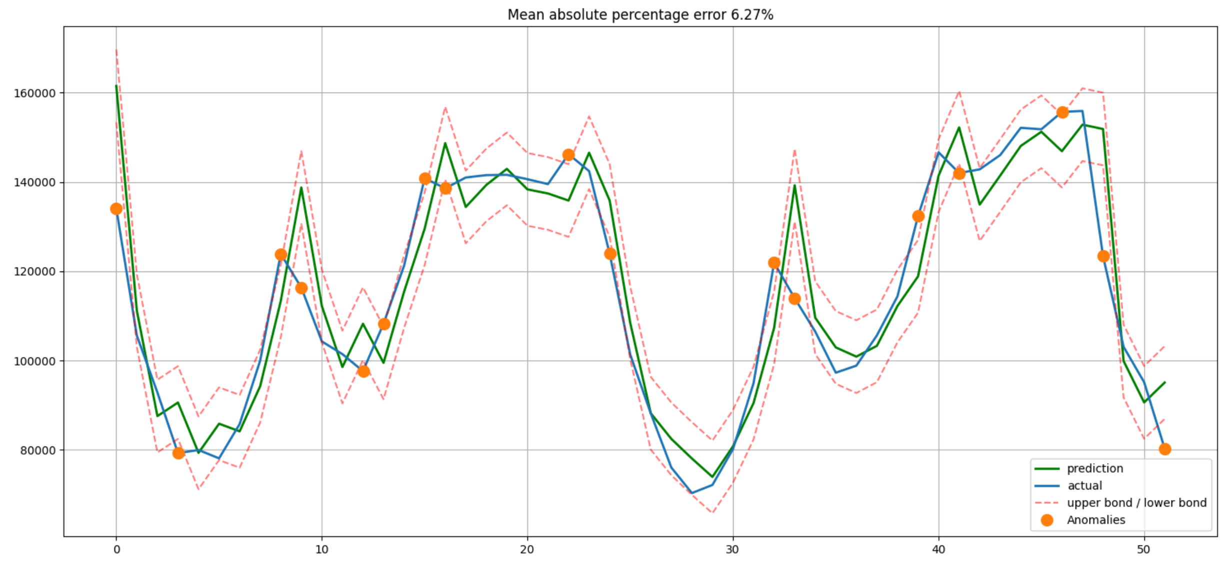


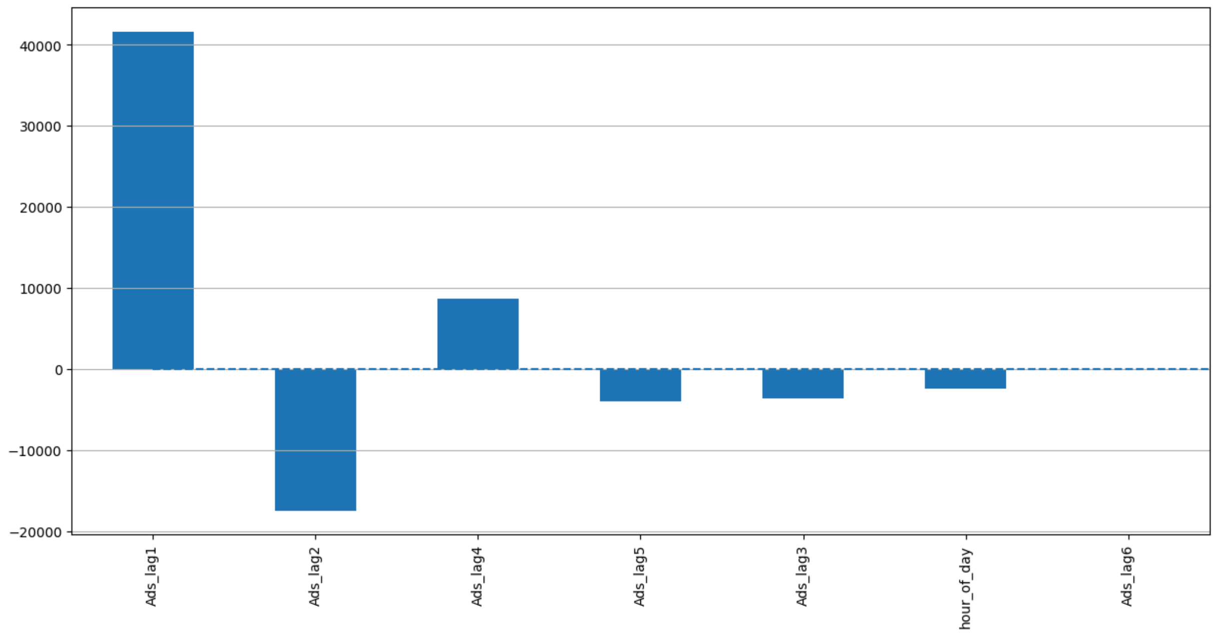


Linear Regression on Scaled Dataset

I preform linear regression on the scaled dataset, now we can analyze the results of this based on several preformance metrics. The mean absolute percentage error is calculated to assess the accuracy of the model's predictions relative to the actual values. A lower MAPE indicates better predictive performance.

In [30]:
```
from sklearn.preprocessing import StandardScaler

#initialize standard scaler to standardize features by removing mean and scaling to

scaler = StandardScaler()

#fit scaler on training data and transform training and test datasets
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

#initialize linear regression
lr = LinearRegression()

#fit the linear regression to scaled data
lr.fit(X_train_scaled, y_train)

plotModelResults(lr, X_train=X_train_scaled, X_test=X_test_scaled, plot_intervals=T

#plot sorted coefficients
plotCoefficients(lr)
```





```
In [32]: from lazypredict.Supervised import LazyRegressor

         #convert data to save memory
         X = X.astype(np.float32)

         #split into training/testing sets
         offset = 158
         X_train, y_train = X[:offset], y[:offset]
         X_test, y_test = X[offset:], y[offset:]

         #initialize model
         reg = LazyRegressor(verbose=0, ignore_warnings=False, custom_metric=None)

         #fit the model
         models, predictions = reg.fit(X_train, X_test, y_train, y_test)
```

```
100%|████████████████████████████████████████
███| 42/42 [00:01<00:00, 28.31it/s]
```

```
XGBRegressor model failed to execute
'super' object has no attribute '__sklearn_tags__'
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of testing wa
s 0.000048 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 349
[LightGBM] [Info] Number of data points in the train set: 158, number of used featur
es: 7
[LightGBM] [Info] Start training from score 125083.132911
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
```

```
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
```

In [33]:
```
print(models)
```

```
                               Adjusted R-Squared  R-Squared      RMSE  \
Model                                                                     
ExtraTreesRegressor                          0.94       0.95   5783.20   
GradientBoostingRegressor                    0.93       0.94   6251.40   
RandomForestRegressor                        0.93       0.94   6297.64   
LGBMRegressor                                0.92       0.93   6833.44   
HistGradientBoostingRegressor                0.91       0.92   6988.10   
BaggingRegressor                             0.91       0.92   7137.68   
AdaBoostRegressor                            0.89       0.91   7669.28   
DecisionTreeRegressor                        0.89       0.91   7790.57   
KNeighborsRegressor                          0.89       0.90   7908.56   
GaussianProcessRegressor                     0.84       0.86   9424.31   
OrthogonalMatchingPursuitCV                  0.83       0.85   9742.23   
Ridge                                        0.82       0.85   9873.09   
LassoCV                                      0.82       0.85   9873.91   
BayesianRidge                                0.82       0.85   9905.35   
RidgeCV                                      0.82       0.85   9926.79   
LassoLarsCV                                  0.82       0.85   9929.07   
LassoLars                                    0.82       0.85   9941.40   
Lasso                                        0.82       0.85   9941.43   
LassoLarsIC                                  0.82       0.85   9944.51   
TransformedTargetRegressor                   0.82       0.85   9944.53   
LinearRegression                             0.82       0.85   9944.53   
SGDRegressor                                 0.82       0.85   9961.91   
RANSACRegressor                              0.81       0.84  10183.82   
HuberRegressor                               0.81       0.84  10265.33   
PoissonRegressor                             0.80       0.83  10585.90   
ExtraTreeRegressor                           0.76       0.79  11482.86   
LarsCV                                       0.75       0.79  11677.09   
PassiveAggressiveRegressor                   0.73       0.76  12280.19   
OrthogonalMatchingPursuit                    0.71       0.75  12753.45   
ElasticNet                                   0.60       0.65  14867.97   
GammaRegressor                               0.50       0.57  16622.54   
TweedieRegressor                             0.47       0.55  17045.96   
Lars                                        -0.02       0.12  23698.93   
ElasticNetCV                                -0.19      -0.03  25663.25   
NuSVR                                       -0.31      -0.13  26921.29   
DummyRegressor                              -0.33      -0.15  27144.57   
QuantileRegressor                           -0.52      -0.31  29020.58   
SVR                                         -0.55      -0.34  29282.19   
LinearSVR                                  -24.14     -20.69 117858.52   
MLPRegressor                               -24.20     -20.74 117995.60   
KernelRidge                                -26.70     -22.90 123717.47   

                               Time Taken  
Model                                      
ExtraTreesRegressor                  0.14  
GradientBoostingRegressor            0.11  
RandomForestRegressor                0.21  
LGBMRegressor                        0.03  
HistGradientBoostingRegressor        0.08  
BaggingRegressor                     0.04  
AdaBoostRegressor                    0.11  
DecisionTreeRegressor                0.01  
KNeighborsRegressor                  0.01  
GaussianProcessRegressor             0.04  
```

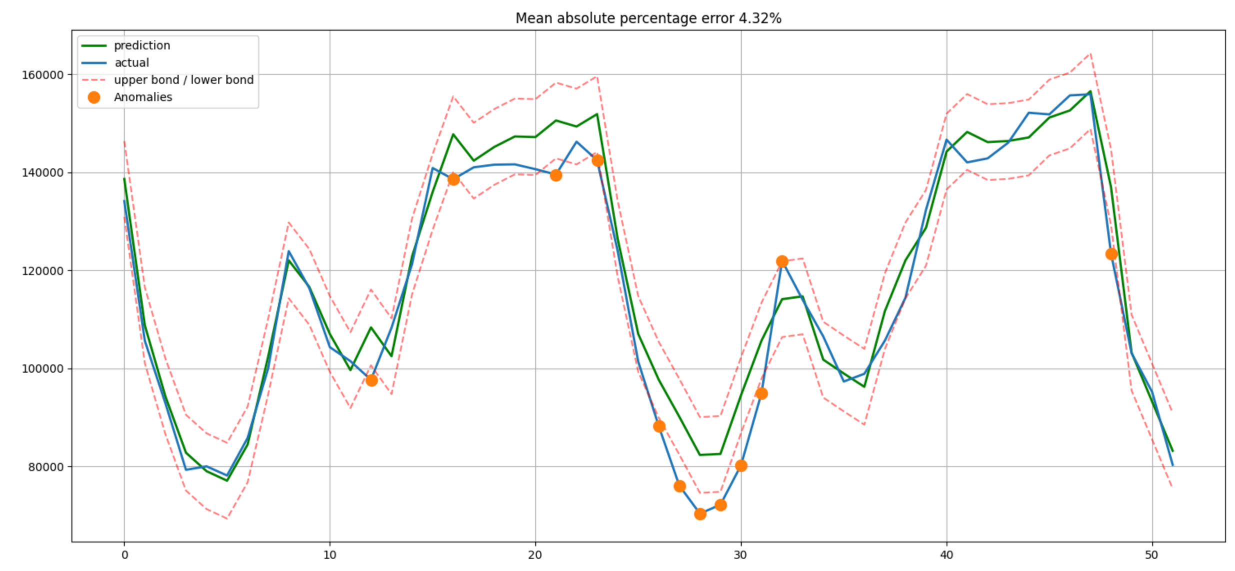
```
OrthogonalMatchingPursuitCV          0.01
Ridge                                0.01
LassoCV                              0.07
BayesianRidge                        0.01
RidgeCV                              0.01
LassoLarsCV                          0.02
LassoLars                            0.01
Lasso                                0.01
LassoLarsIC                          0.01
TransformedTargetRegressor           0.01
LinearRegression                     0.01
SGDRegressor                         0.01
RANSACRegressor                      0.02
HuberRegressor                       0.02
PoissonRegressor                     0.01
ExtraTreeRegressor                   0.01
LarsCV                               0.02
PassiveAggressiveRegressor           0.02
OrthogonalMatchingPursuit            0.01
ElasticNet                           0.01
GammaRegressor                       0.01
TweedieRegressor                     0.01
Lars                                 0.01
ElasticNetCV                         0.06
NuSVR                                0.01
DummyRegressor                       0.01
QuantileRegressor                    0.02
SVR                                  0.01
LinearSVR                            0.01
MLPRegressor                         0.12
KernelRidge                          0.01
```

I use the GradientBoostingRegressor model to improve prediction accuracy.

In [34]:
```
from sklearn.ensemble import GradientBoostingRegressor
gb = GradientBoostingRegressor()

#train boosting model using the scaled data
gb.fit(X_train_scaled, y_train)
```

Out[34]:
```
▾ GradientBoostingRegressor
GradientBoostingRegressor()
```

In [35]:
```
plotModelResults(gb,
                 X_train=X_train_scaled,
                 X_test=X_test_scaled,
                 plot_intervals=True,
                 plot_anomalies=True)
```



After implementing the GradientBoostingRegressor model, our MAPE went down from 6.27% to about 4.33%. This means the model is much more accurate in its predictions.

In conclusion, with the low mean absolute percentage error of approximately 4.33%, the model is preforming well, making predictions that are very close to actual values. This means that with the given information about advertisements run, we can use it to make fairly accurate assessments on the amount of game currency that will be spent.