Time Series Analisys on Geoespatial Data with Python

Author: João Otavio Nascimento Firigato

email: joaootavionf007@gmail.com

LinkedIn: https://www.linkedin.com/in/jo%C3%A3o-otavio-firigato-4876b3aa/

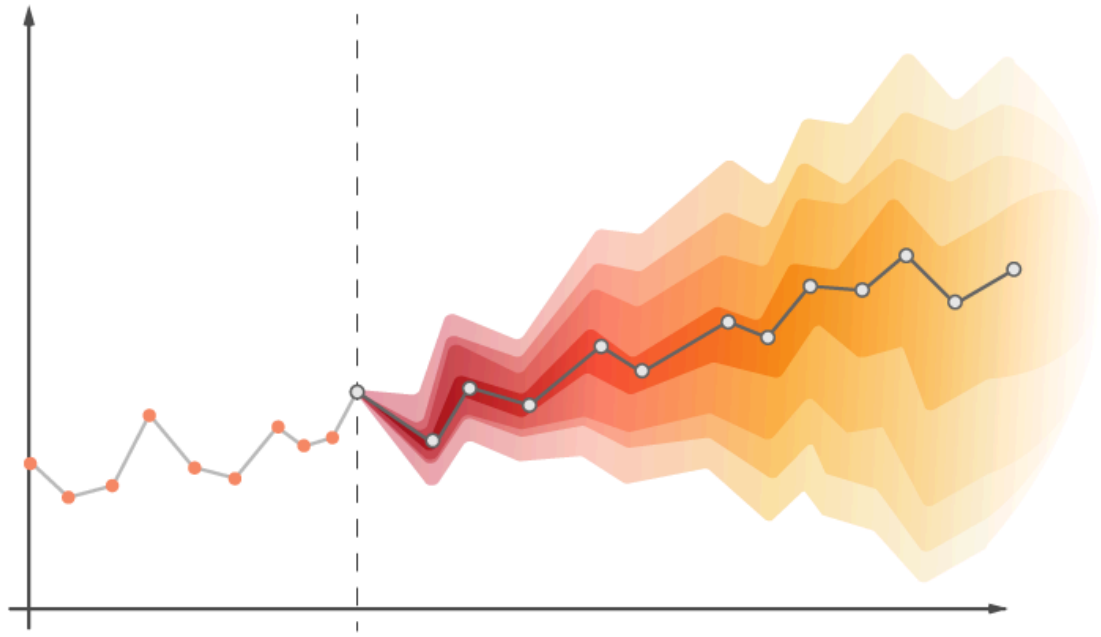
First instructions:

✅ Access the link to join our private WhatsApp community for students: https://chat.whatsapp.com/EPn27ZgR07lF3e1vnj8FiI

❗ It is important to access the Whatsapp Group to get the Colab Notebooks, as the PDF files are protected from text copying.

Chapter 8 - Time Series Forecast - Part 1

Time series forecasting involves analyzing data that evolves over a period of time and then using statistical models to make predictions about future patterns and trends. It takes into account the sequential nature of data, where each observation depends on previous observations.



It is not an exact prediction, nor is it possible to predict with 100% accuracy, especially when dealing with variables that change frequently and some beyond our control variables.

However, forecasting can provide insights into the likelihood of certain outcomes. Generally, a larger data set leads to a more accurate forecast.

Time series analysis is often combined with time series forecasting. In time series analysis, models are developed to understand the underlying causes of the data. By analyzing the results, you can understand “why” they occur. As a result, forecasting takes the next step of extrapolating the future from knowledge derived from the past.

Time Series Model for Forecasting

Time series models are statistical tools that experts use to study and predict data that changes over time. These models help us discover patterns, trends, and relationships in data, which in turn allows us to make informed predictions about what might happen in the future.

Below is a brief overview of some commonly used time series models:

Moving Average (MA) Model: This model averages past observations in order to predict future values. It is useful for capturing short-term fluctuations and random variations in data.

- Assumptions: Observations are a linear combination of past error terms, and there is no autocorrelation between the error terms.
- Parameters: The order of the model (q) determines the number of lagged error terms to include.
- Strengths: MA models are effective in capturing short-term dependencies and smoothing out random fluctuations in data.

Autoregressive (AR) model: This model predicts future values based on a linear combination of past observations.

- Assumptions: It assumes that future values depend on past values, capturing long-term trends and dependencies.
- Parameters: The order of the model (p) determines the number of lagged observations to include.
- Strengths: AR models are useful for capturing long-term trends and dependencies in data.

Autoregressive Moving Average (ARMA) Model: The ARMA model combines the AR and MA models to capture short-term and long-term patterns in data. It is effective for analyzing stationary time series data.

- Assumptions: Observations are a linear combination of past observations and past error terms, and there is no autocorrelation between the error terms.
- Parameters: The orders of the AR and MA components (p and q) determine the number of lagged observations and error terms to be included.

- Strengths: ARMA models combine the strengths of the AR and MA models by capturing both short-term and long-term dependencies in the data.

Autoregressive Integrated Moving Average (ARIMA) Model: This model extends the ARMA model by incorporating differencing to handle non-stationary data. It is suitable for data with trends or seasonality.

- Assumptions: The data is stationary after differencing, meaning that the differences between consecutive observations are stationary.
- Parameters: The orders of the AR, I, and MA components (p, d, and q) determine the number of lagged observations, differencing, and lagged error terms to include.
- Strengths: ARIMA models can handle non-stationary data by incorporating differencing, making them suitable for time series with trends or seasonality.

Seasonal ARIMA (SARIMA) model: This model is an extension of the ARIMA model and includes seasonal components. It is useful for analyzing and forecasting data with recurring seasonal patterns.

- Assumptions: The data exhibits seasonal patterns as well as trends and dependencies.
- Parameters: The orders of the seasonal AR, I, and MA components (P, D, and Q) determine the number of lagged seasonal observations, seasonal differencing, and lagged seasonal error terms to include.
- Strengths: SARIMA models are effective for analyzing and forecasting time series data with seasonal patterns.

Difference between a Time Series and a Regression problem

Here you might think that since the target variable is numeric, it can be predicted using regression techniques, but a time series problem is different from a regression problem in the following ways:

- The main difference is that a time series is time-dependent. Therefore, the basic assumption of a linear regression model that observations are independent does not apply in this case.
- Along with an increasing or decreasing trend, most time series have some form of seasonality trends, i.e., variations specific to a given time period. Therefore, predicting a time series using regression techniques is not a good approach.

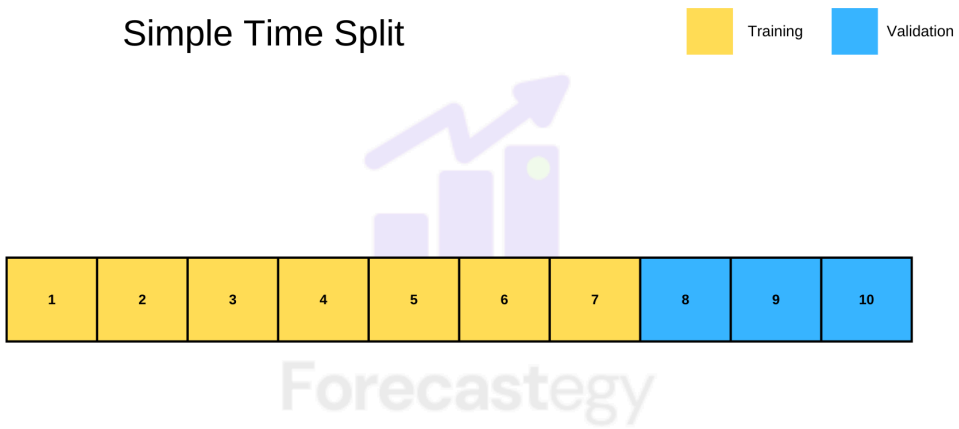
Time series analysis comprises methods for analyzing time series data in order to extract meaningful statistics and other features from the data. Time series forecasting is the use of a model to predict future values based on previously observed values.

Time series model validation techniques

Since we have validated all our hypotheses, let's move ahead and build models for Time Series Forecasting. But before we do that, we will need a dataset (validation) to check the performance and generalization ability of our model. Below are some of the properties of the dataset required for the purpose.

- The dataset should have the true values of the dependent variable against which the predictions can be checked. Hence, the test dataset cannot be used for the purpose.
- The model should not be trained on the validation dataset. Hence, we cannot train the model on the training dataset and validate on it as well.

So, for the above two reasons, we generally divide the training dataset into two parts. One part is used to train the model and the other part is used as the validation dataset. Now, there are various ways to divide the training dataset like Random Split etc.



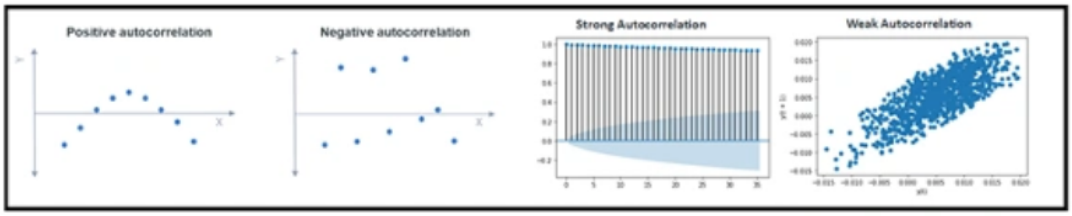
Autocorrelation Function (ACF)

ACF indicates how similar a value is within a given time series and the previous value. (OR) It measures the degree of similarity between a given time series and the lagged version of that time series at the various intervals we observe.

Partial Autocorrelation (PACF)

PACF is similar to Autocorrelation Function and is a bit challenging to understand. It always shows the correlation of the sequence with itself with some number of time units in sequence order in which only the direct effect has been shown, and all other intermediate effects are removed from the given time series.

Types of AutoCorrelation



Interpret ACF and PACF graphs

ACF	PACF	Perfect ML -Model
Plot declines gradually	Plot drops instantly	Auto Regressive model.
Plot drops instantly	Plot declines gradually	Moving Average model
Plot decline gradually	Plot Decline gradually	ARMA
Plot drop instantly	Plot drop instantly	You wouldn't perform any model

Moving Averge Model

Let's start with the moving average model. First we will get our dataset from Google Drive:

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

In [ ]:
```
import pandas as pd
import matplotlib.pyplot as plt
```

Download link:
https://drive.google.com/file/d/1Xx044h8ZwsbuY7kWAc1bcQy2XKz0BIYP/view?usp=sharing

In [ ]:
```
path = '/content/drive/MyDrive/Datasets_TS/precipitacionsbcndesde1786_2024.csv'
```

In [ ]:
```
df_precipitacion = pd.read_csv(path)
```

In [ ]:
```
df_precipitacion
```

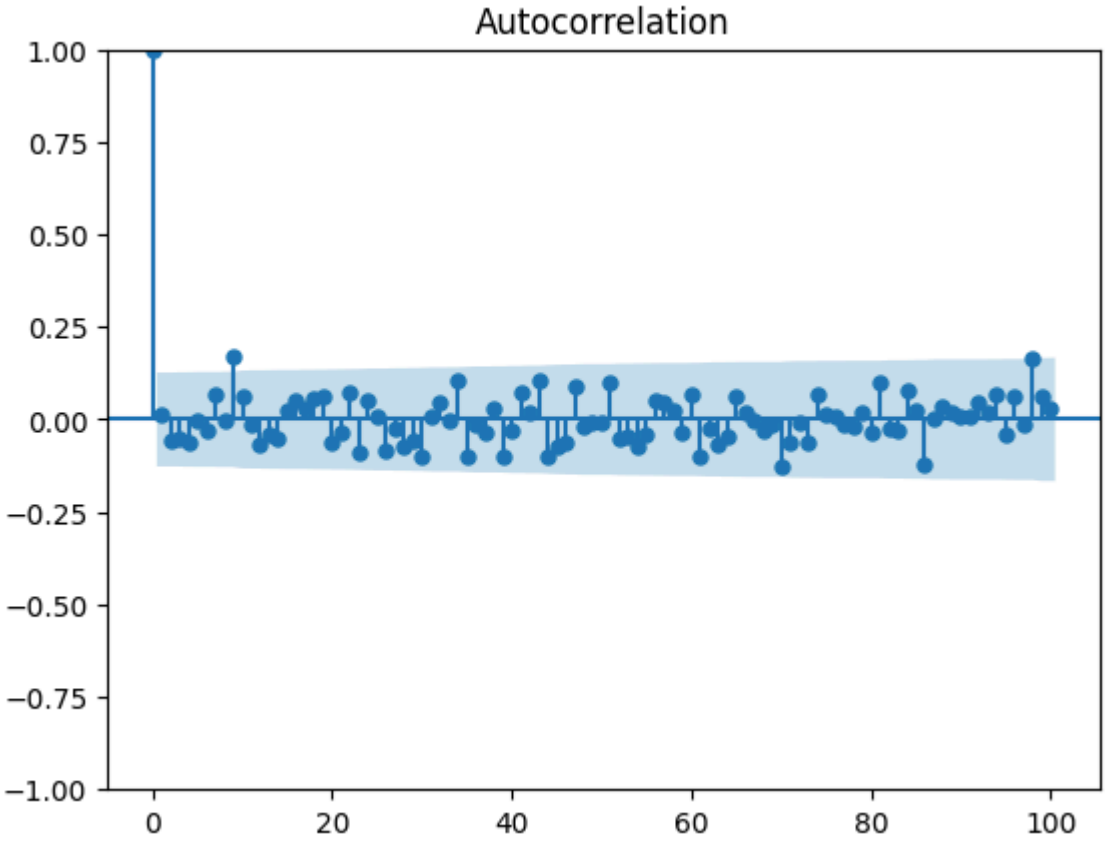
Out[ ]:

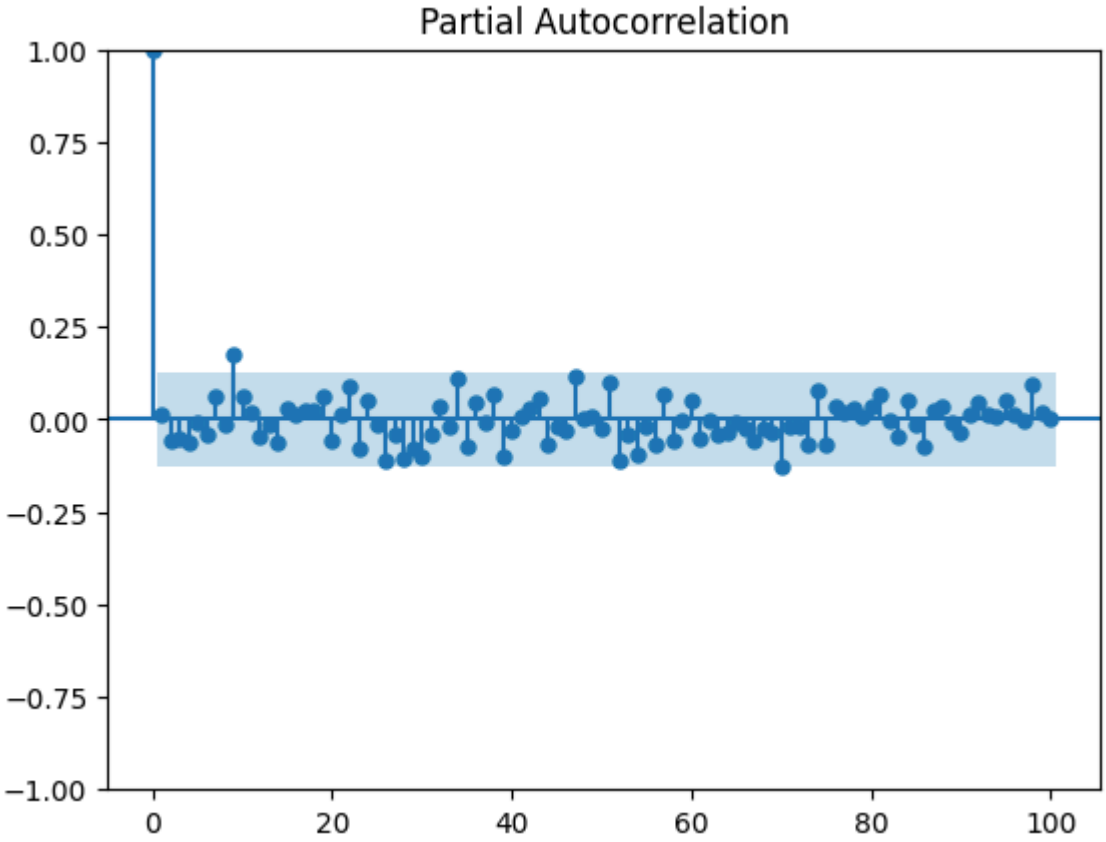
	Any	Precip_Acum_Gener	Precip_Acum_Febrer	Precip_Acum_Marc	Precip_Acum_Al
0	1786	32.8	28.4	84.4	4
1	1787	136.4	27.4	44.6	7
2	1788	9.9	14.9	32.2	1
3	1789	12.4	12.4	23.6	
4	1790	44.6	1.2	188.5	7
...	...	...	...	...	
234	2020	89.3	2.8	61.6	25
235	2021	26.2	26.5	7.7	6
236	2022	12.3	1.9	89.5	3
237	2023	8.7	40.7	1.0	3
238	2024	26.8	27.8	87.9	9

239 rows × 13 columns

We will select the column: Precip_Acum_Gener. We will create the ACF and PACF graphs

In [ ]:
```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf


# Plot ACF
plot_acf(df_precipitacion['Precip_Acum_Gener'], lags=100) # Adjust lags as neede
plt.show()

# Plot PACF
plot_pacf(df_precipitacion['Precip_Acum_Gener'], lags=100) # Adjust lags as nea
plt.show()
```





Let's group the data by year and divide it into training data and testing data;

In [ ]:
```
df_precipitacion['Any'] = pd.to_datetime(df_precipitacion['Any'], format='%Y')
yearly_precipitation = df_precipitacion.groupby('Any')['Precip_Acum_Gener'].sum(
yearly_precipitation = yearly_precipitation.reset_index()
```

In [ ]:
```
idx = 219
train = yearly_precipitation[0:idx]
valid = yearly_precipitation[idx:]
```

In [ ]:
```
train
```

Out[ ]:

	Any	Precip_Acum_Gener
0	1786-01-01	32.8
1	1787-01-01	136.4
2	1788-01-01	9.9
3	1789-01-01	12.4
4	1790-01-01	44.6
...	...	...
214	2000-01-01	26.6
215	2001-01-01	84.2
216	2002-01-01	46.3
217	2003-01-01	24.6
218	2004-01-01	2.6

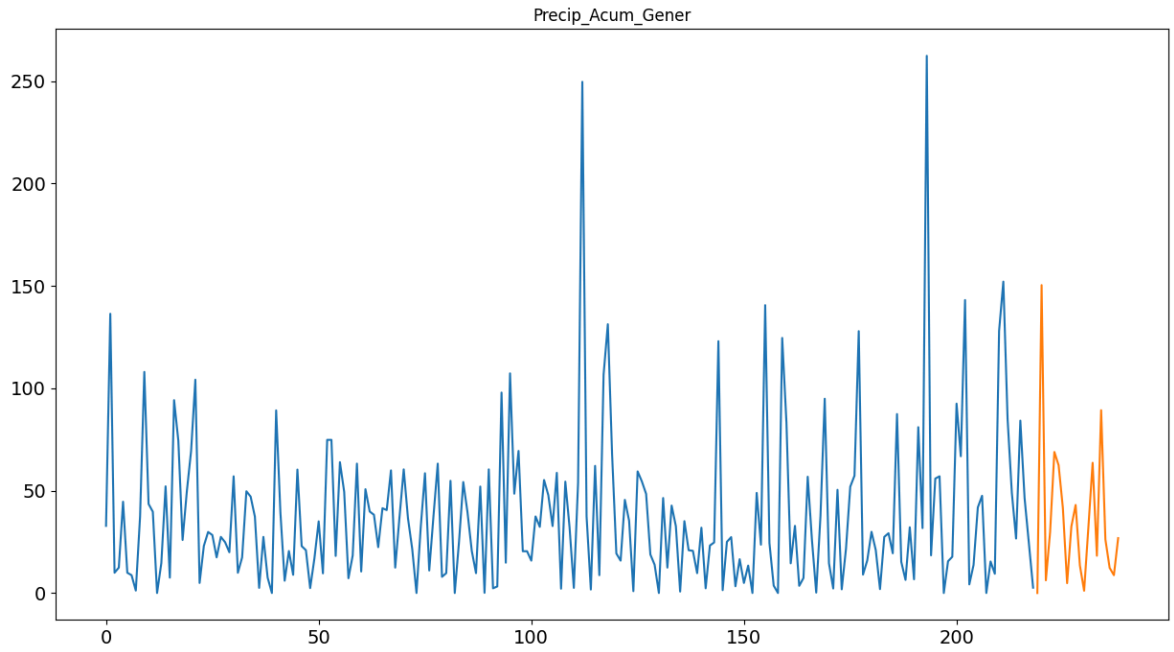
219 rows × 2 columns

In [ ]:
```
valid
```

Out[ ]:

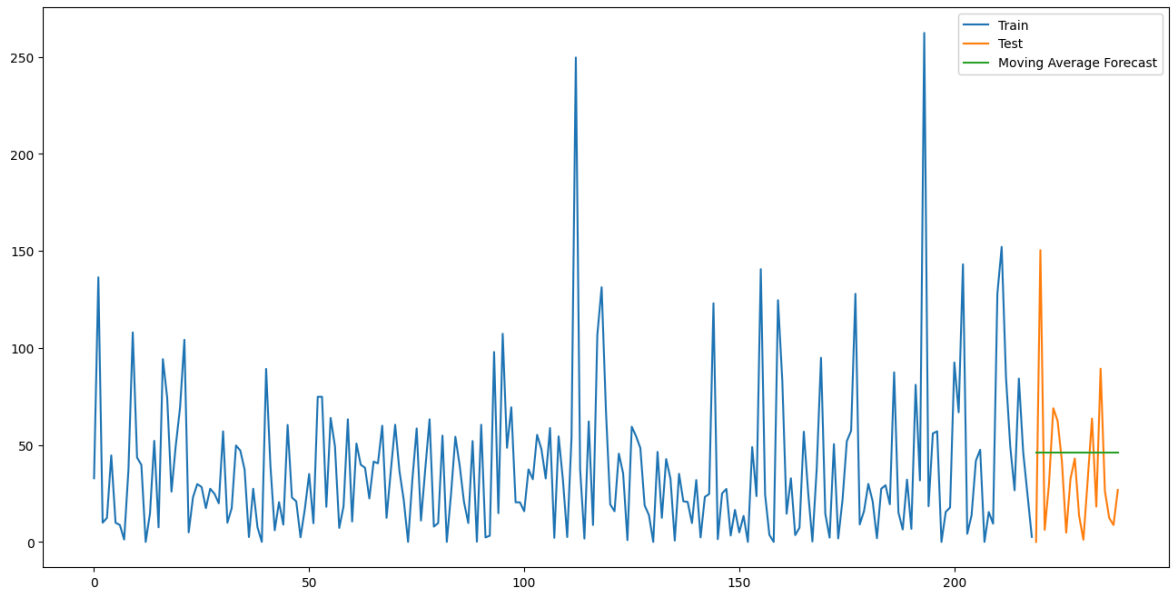
	Any	Precip_Acum_Gener
219	2005-01-01	0.0
220	2006-01-01	150.4
221	2007-01-01	6.2
222	2008-01-01	29.8
223	2009-01-01	68.9
224	2010-01-01	62.4
225	2011-01-01	41.6
226	2012-01-01	4.8
227	2013-01-01	32.5
228	2014-01-01	43.0
229	2015-01-01	13.5
230	2016-01-01	1.1
231	2017-01-01	32.0
232	2018-01-01	63.6
233	2019-01-01	18.2
234	2020-01-01	89.3
235	2021-01-01	26.2
236	2022-01-01	12.3
237	2023-01-01	8.7
238	2024-01-01	26.8

In [ ]:
```
train['Precip_Acum_Gener'].plot(figsize=(15,8), title= 'Precip_Acum_Gener', font
valid['Precip_Acum_Gener'].plot(figsize=(15,8), title= 'Precip_Acum_Gener', font
plt.show()
```



So we can apply the moving average considering a range of 50 previous records:

In [ ]:
```
y_hat_avg = valid.copy()
y_hat_avg['Precip_Acum_Gener'] = train['Precip_Acum_Gener'].rolling(50).mean().i
plt.figure(figsize=(16,8))
plt.plot(train['Precip_Acum_Gener'], label='Train')
plt.plot(valid['Precip_Acum_Gener'], label='Test')
plt.plot(y_hat_avg['Precip_Acum_Gener'], label='Moving Average Forecast')
plt.legend(loc='best')
plt.show()
```



In [ ]:
```
from sklearn.metrics import mean_squared_error
from math import sqrt
rms = sqrt(mean_squared_error(valid['Precip_Acum_Gener'], y_hat_avg['Precip_Acum
print(rms)
```

```
36.77607978020497
```

Auto-Regressive Models

Autoregressive models operate under the premise that past values have an effect on current values, which makes them a popular statistical technique for analyzing nature, economics, and other processes that vary over time. Multiple regression models predict a variable using a linear combination of predictors, while autoregressive models use a combination of past values of the variable.

An AR(1) autoregressive process is one in which the current value is based on the immediately preceding value, while an AR(2) process is one in which the current value is based on the two preceding values. An AR(0) process is used for white noise and has no dependence between terms. In addition to these variations, there are also many different ways to calculate the coefficients used in these calculations, such as the least squares method.

These concepts and techniques are used by technical analysts to predict security prices. However, because autoregressive models base their predictions solely on past information, they implicitly assume that the fundamental forces that influenced past prices will not change over time. This can lead to surprising and inaccurate predictions if the underlying forces in question are in fact changing, such as if an industry is undergoing a rapid and unprecedented technological transformation.

However, traders continue to refine the use of autoregressive models for forecasting purposes. A great example is the Autoregressive Integrated Moving Average (ARIMA), a sophisticated autoregressive model that can take into account trends, cycles, seasonality, errors, and other types of non-static data when making predictions.

Benefits of the autoregressive model

The advantage of this model is that you can tell if there is a lack of randomness using the autocorrelation function.

In addition, it is able to predict recurring patterns in data.

It is also possible to predict outcomes with less information using autovariate series.

Limitations of the autoregressive model

There are several limitations associated with this method:

The autocorrelation coefficient should be at least 0.5 in this case for it to be appropriate. This means that if it is less than 0.5, the forecast result will be inaccurate.

It is usually used when predicting things associated with the economy based on historical data. Something that is significantly affected by social factors. It is highly recommended to use the vector autoregressive model. The reason is that a single model can be used to predict multiple time series variables at the same time.

Auto Regressive (AR) Model

Autoregressive models belong to the family of time series models. These models capture the relationship between an observation and several lagged observations (previous time steps). The central idea is that the current value of a time series can be expressed as a linear combination of its past values, with some random noise.

Let's work with a new dataset:

Download link:
https://drive.google.com/drive/folders/1ogwMbHEzylWsT7QlMrQolnnJBMt1hcrT?usp=sharing

In [ ]:
```
path = '/content/drive/MyDrive/Datasets_TS/hourly_data/temperature.csv'
```

In [ ]:
```
df = pd.read_csv(path)
```

In [ ]:
```
df
```

Out[ ]:

	datetime	Vancouver	Portland	San Francisco	Seattle	Los Angeles	San Dieg
0	2012-10-01 12:00:00	NaN	NaN	NaN	NaN	NaN	Na
1	2012-10-01 13:00:00	284.630000	282.080000	289.480000	281.800000	291.870000	291.53000
2	2012-10-01 14:00:00	284.629041	282.083252	289.474993	281.797217	291.868186	291.53350
3	2012-10-01 15:00:00	284.626998	282.091866	289.460618	281.789833	291.862844	291.54335
4	2012-10-01 16:00:00	284.624955	282.100481	289.446243	281.782449	291.857503	291.55320
...	...	...	...	...	...	...	
45248	2017-11-29 20:00:00	NaN	282.000000	NaN	280.820000	293.550000	292.15000
45249	2017-11-29 21:00:00	NaN	282.890000	NaN	281.650000	295.680000	292.74000
45250	2017-11-29 22:00:00	NaN	283.390000	NaN	282.750000	295.960000	292.58000
45251	2017-11-29 23:00:00	NaN	283.020000	NaN	282.960000	295.650000	292.61000
45252	2017-11-30 00:00:00	NaN	282.280000	NaN	283.040000	294.930000	291.40000

45253 rows × 37 columns

We will obtain the Los Angeles temperature time series:

In [ ]:
```
df_LA =df.loc[:,['datetime','Los Angeles']]
```

In [ ]:
```
df_LA
```

Out[ ]:

	datetime	Los Angeles
0	2012-10-01 12:00:00	NaN
1	2012-10-01 13:00:00	291.870000
2	2012-10-01 14:00:00	291.868186
3	2012-10-01 15:00:00	291.862844
4	2012-10-01 16:00:00	291.857503
...	...	...
45248	2017-11-29 20:00:00	293.550000
45249	2017-11-29 21:00:00	295.680000
45250	2017-11-29 22:00:00	295.960000
45251	2017-11-29 23:00:00	295.650000
45252	2017-11-30 00:00:00	294.930000

45253 rows × 2 columns

We will get the data in certain ranges and convert it to Celsius degrees.

In [ ]:
```
df_LA=df_LA[(df_LA['datetime']>='2017-11-15 00:00:00') & (df_LA['datetime']<'201
```

In [ ]:
```
df_LA
```

Out[ ]:

	datetime	Los Angeles
44892	2017-11-15 00:00:00	297.22
44893	2017-11-15 01:00:00	296.04
44894	2017-11-15 02:00:00	293.87
44895	2017-11-15 03:00:00	292.62
44896	2017-11-15 04:00:00	291.28
...	...	...
45151	2017-11-25 19:00:00	296.90
45152	2017-11-25 20:00:00	298.96
45153	2017-11-25 21:00:00	299.92
45154	2017-11-25 22:00:00	299.75
45155	2017-11-25 23:00:00	299.49

264 rows × 2 columns

In [ ]:
```
print(df_LA.isnull().sum())
```

```
datetime       0
Los Angeles    0
dtype: int64
```

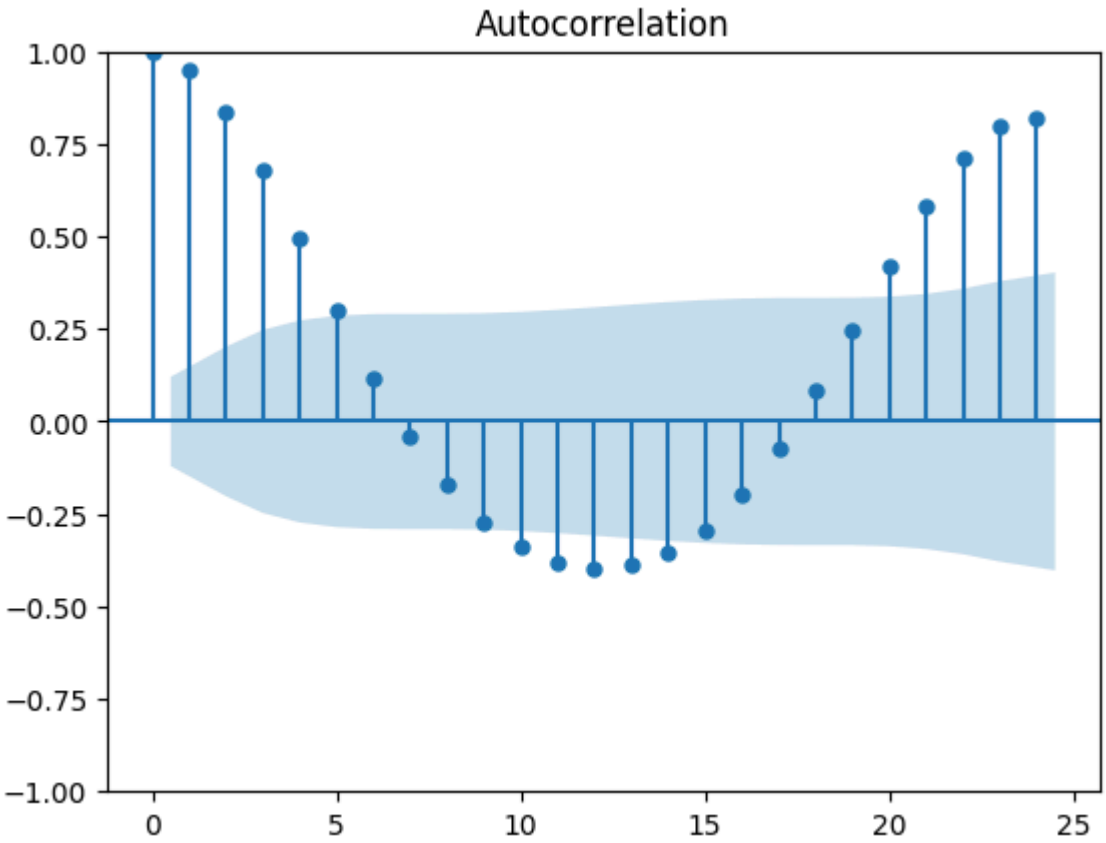
In [ ]:
```
df_LA.iloc[:,1]=df_LA.iloc[:,1].apply(round)-273.15
```

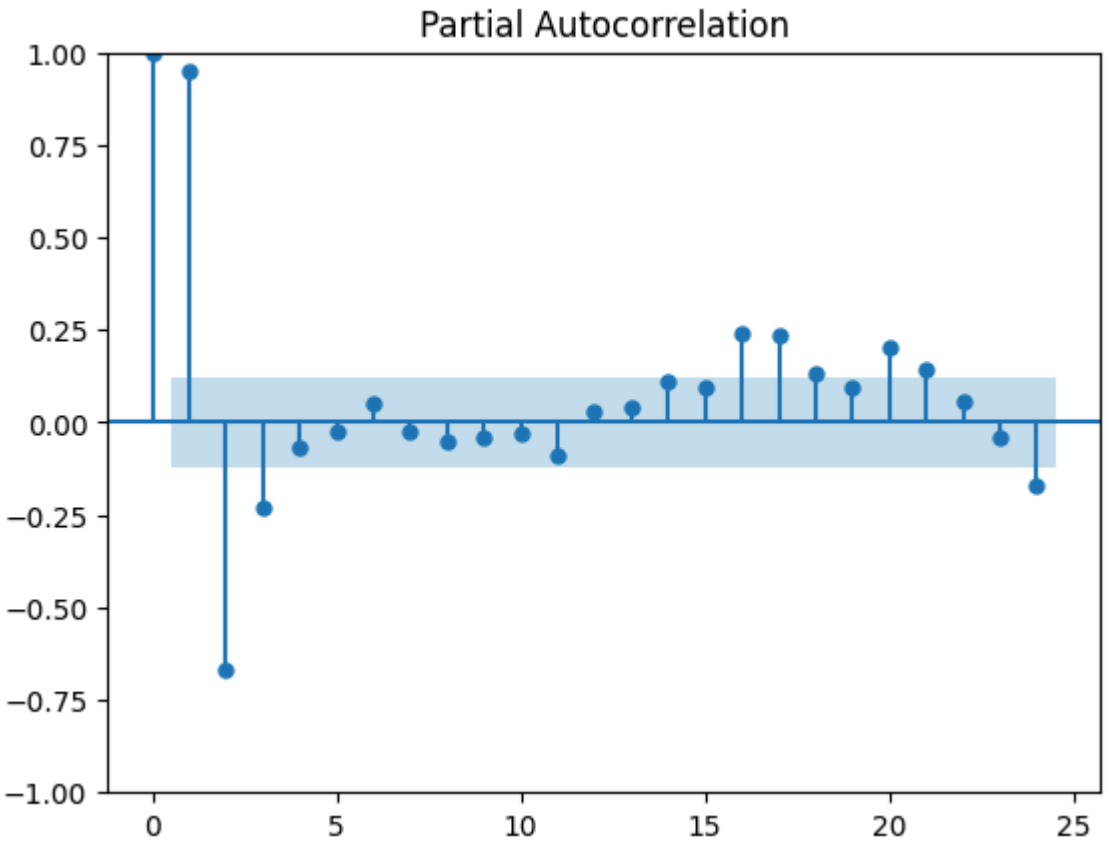
In [ ]:
```
df_LA['datetime']=pd.to_datetime(df_LA['datetime'])
```

```
<ipython-input-23-20268c8ad12f>:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df_LA['datetime']=pd.to_datetime(df_LA['datetime'])
```

Let's present the ACF and PACF graphs:

In [ ]:
```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

# Plot ACF
plot_acf(df_LA['Los Angeles'], lags = 24)
plt.show()

# Plot PACF
plot_pacf(df_LA['Los Angeles'], lags = 24)
plt.show()
```





In [ ]:
```
df_LA['Los Angeles'].corr(df_LA['Los Angeles'].shift(1))
```

Out[ ]:
```
np.float64(0.9540576367233533)
```

In [ ]:
```
df_LA.set_index('datetime', inplace=True)
```

Now we can split the data into training and testing:

In [ ]:
```
idx = 200
train = df_LA[0:idx]
valid = df_LA[idx:]
```

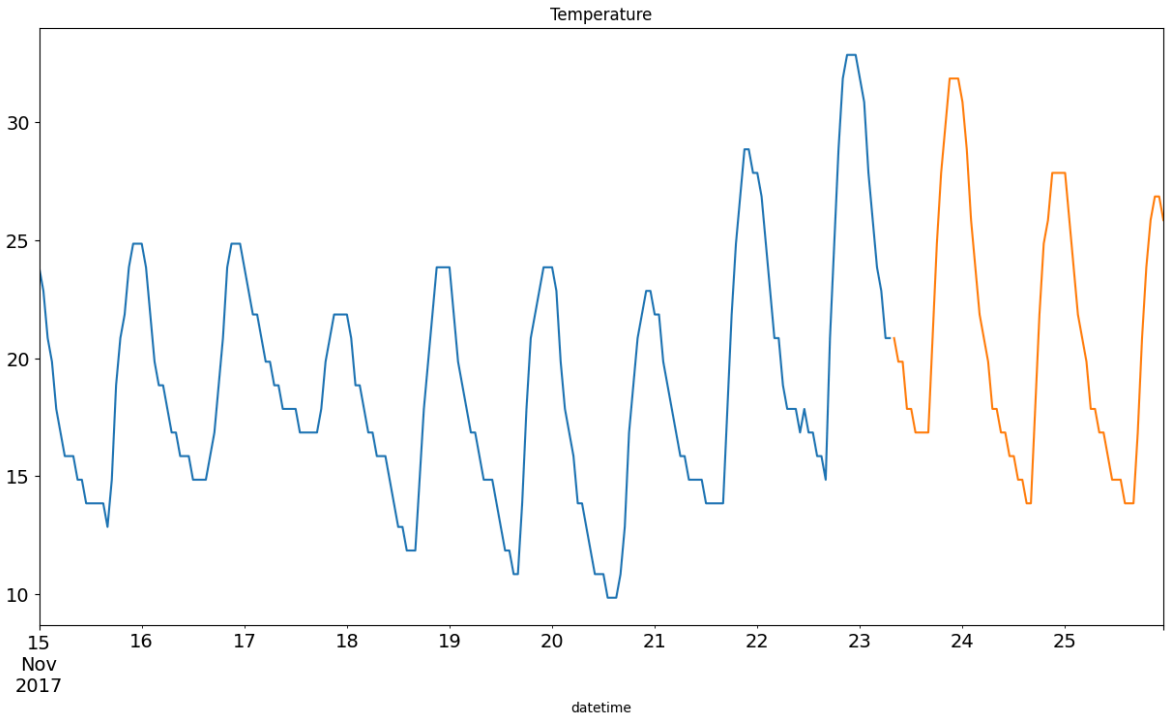
In [ ]:
```
train
```

Out[ ]:

	Los Angeles
datetime	
2017-11-15 00:00:00	23.85
2017-11-15 01:00:00	22.85
2017-11-15 02:00:00	20.85
2017-11-15 03:00:00	19.85
2017-11-15 04:00:00	17.85
...	...
2017-11-23 03:00:00	25.85
2017-11-23 04:00:00	23.85
2017-11-23 05:00:00	22.85
2017-11-23 06:00:00	20.85
2017-11-23 07:00:00	20.85

200 rows × 1 columns

In [ ]:
```
train['Los Angeles'].plot(figsize=(15,8), title= 'Temperature', fontsize=14)
valid['Los Angeles'].plot(figsize=(15,8), title= 'Temperature', fontsize=14)
plt.show()
```



We can then apply the autoregressive model:

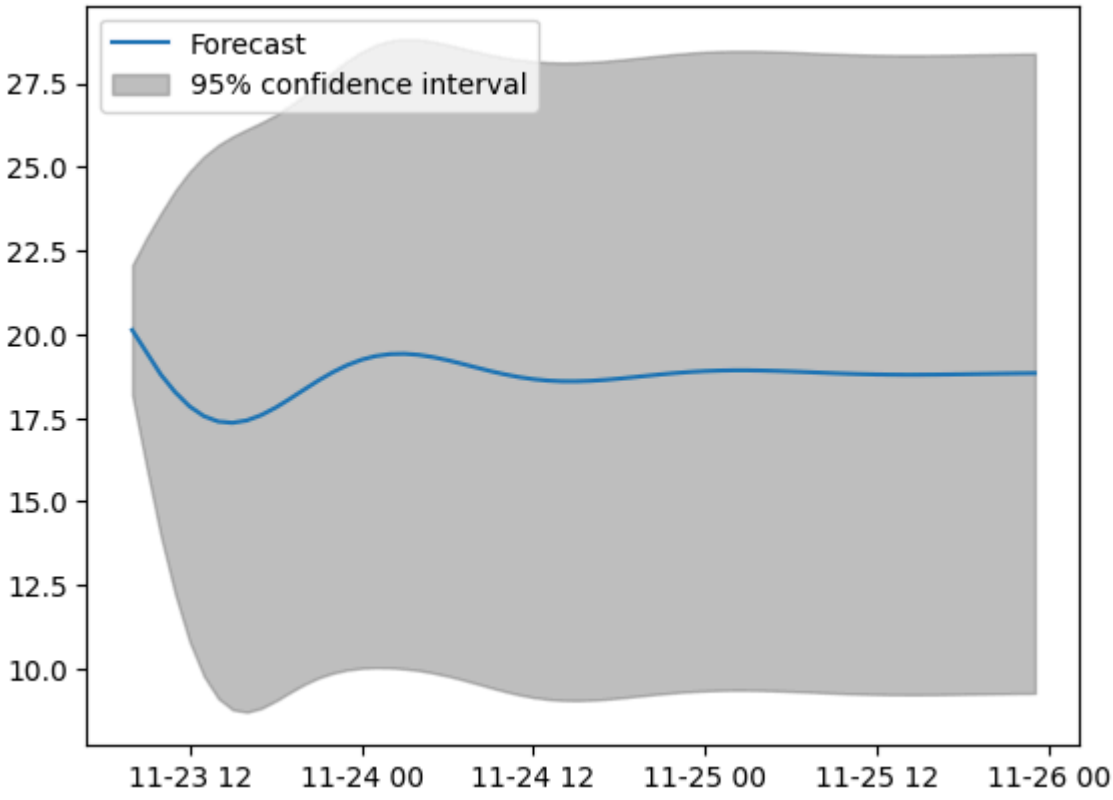
In [ ]:
```
from statsmodels.tsa.ar_model import AutoReg
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.tsa.api import AutoReg
from sklearn.metrics import mean_absolute_error, mean_squared_error
```

```
import numpy as np

# Create and train the autoregressive model
lag_order = 5 # Adjust this based on the ACF plot
ar_model = AutoReg(train['Los Angeles'], lags=lag_order)
ar_results = ar_model.fit()
```

```
/usr/local/lib/python3.11/dist-packages/statsmodels/tsa/base/tsa_model.py:473: Va
lueWarning: No frequency information was provided, so inferred frequency h will b
e used.
  self._init_dates(dates, freq)
```

In [ ]:
```
ar_results.plot_predict(start=len(train['Los Angeles']), end=len(train['Los Ange
```

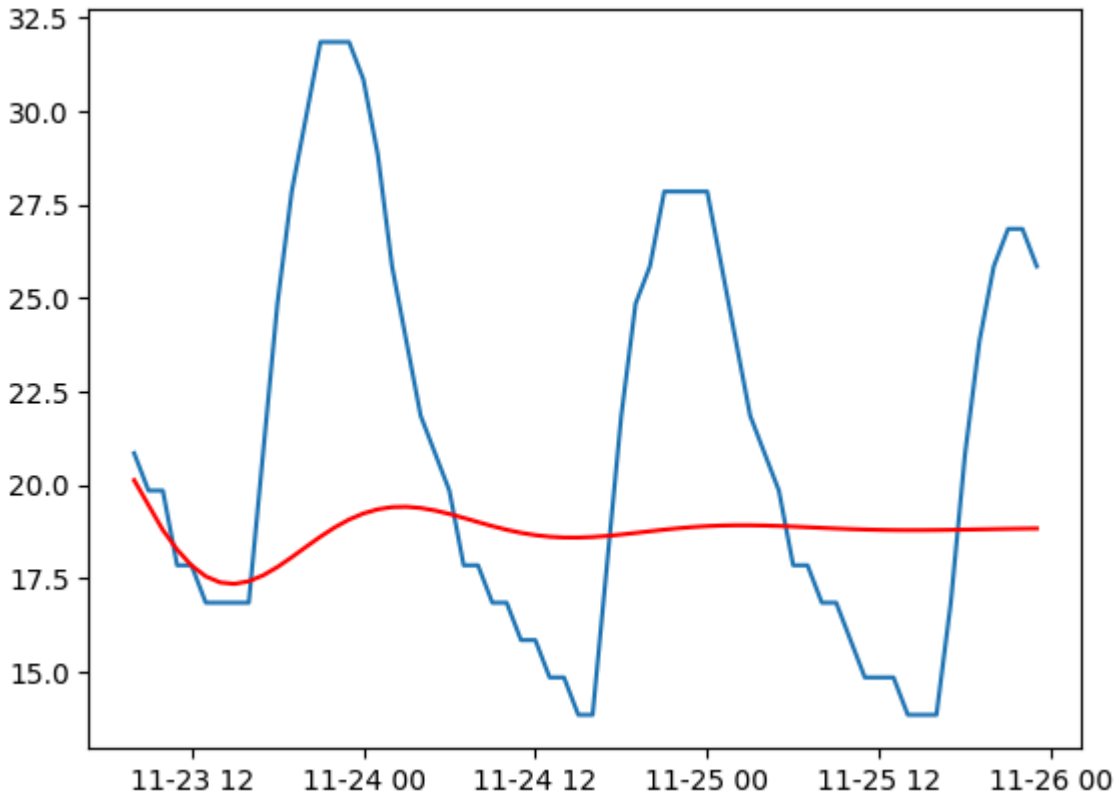
```
/usr/local/lib/python3.11/dist-packages/statsmodels/tsa/deterministic.py:308: Use
rWarning: Only PeriodIndexes, DatetimeIndexes with a frequency set, RangesIndexe
s, and Index with a unit increment support extending. The index is set will conta
in the position relative to the data length.
  fcast_index = self._extend_index(index, steps, forecast_index)
```



In [ ]:
```
predictions = ar_results.predict(start=len(train['Los Angeles']), end=len(train[

mae = mean_absolute_error(valid['Los Angeles'], predictions)
rmse = np.sqrt(mean_squared_error(valid['Los Angeles'], predictions))
print(f'Mean Absolute Error: {mae:.2f}')
print(f'Root Mean Squared Error: {rmse:.2f}')


plt.plot(valid['Los Angeles'])
plt.plot(predictions, color='red')
plt.show()
```

```
Mean Absolute Error: 4.48
Root Mean Squared Error: 5.73
```

```
/usr/local/lib/python3.11/dist-packages/statsmodels/tsa/deterministic.py:308: Use
rWarning: Only PeriodIndexes, DatetimeIndexes with a frequency set, RangesIndexe
s, and Index with a unit increment support extending. The index is set will conta
in the position relative to the data length.
  fcast_index = self._extend_index(index, steps, forecast_index)
```



When we apply our autoregressive model to future data the result looks like this above. However we can apply the autoregressive model in real time, that is, for each prediction, we use all the previous data as training data:

In [ ]:
```
window = 24
model = AutoReg(train['Los Angeles'], lags=24)
model_fit = model.fit()
coef = model_fit.params
# walk forward over time steps in test
history = train['Los Angeles'][len(train['Los Angeles'])-window:]
history = [history[i] for i in range(len(history))]
predictions = list()
for t in range(len(valid['Los Angeles'])):
        length = len(history)
        lag = [history[i] for i in range(length-window,length)]
        yhat = coef[0]
        for d in range(window):
                yhat += coef[d+1] * lag[window-d-1]
        obs = valid['Los Angeles'][t]
        predictions.append(yhat)
        history.append(obs)
        print('predicted=%f, expected=%f' % (yhat, obs))
rmse = sqrt(mean_squared_error(valid['Los Angeles'], predictions))
print('Test RMSE: %.3f' % rmse)
```

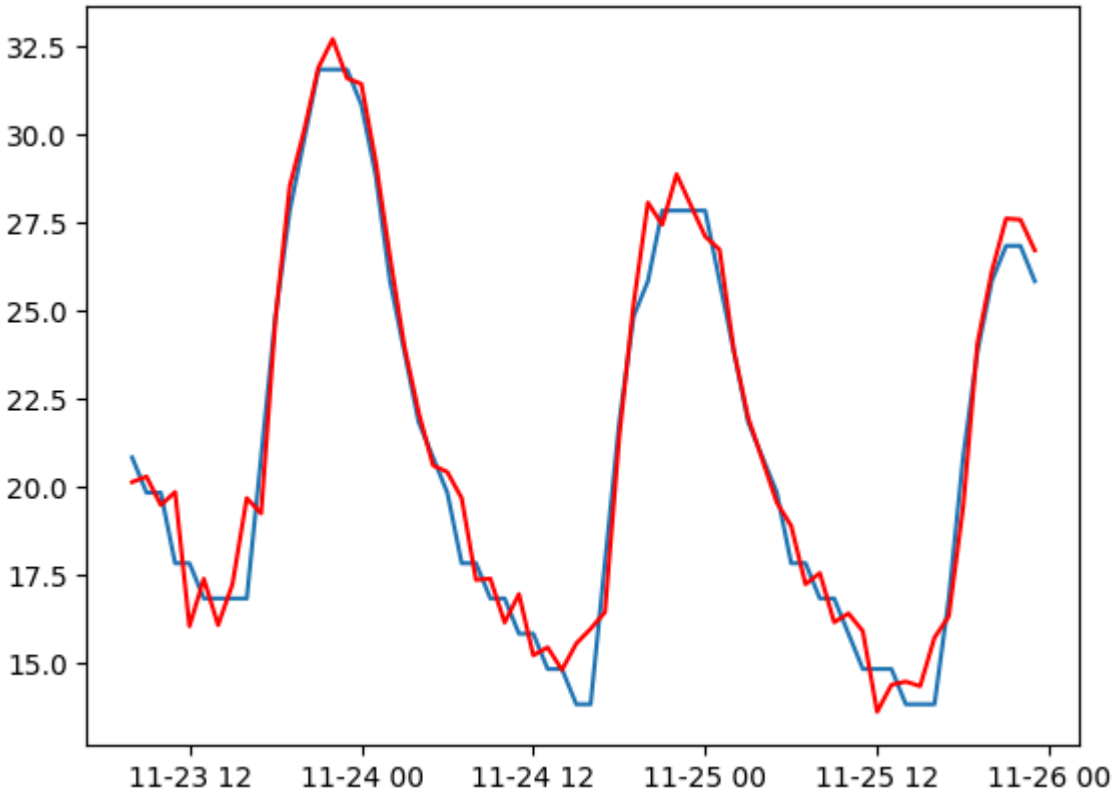
```
predicted=20.150098, expected=20.850000
predicted=20.306258, expected=19.850000
predicted=19.497652, expected=19.850000
predicted=19.868464, expected=17.850000
predicted=16.061633, expected=17.850000
predicted=17.407378, expected=16.850000
predicted=16.094686, expected=16.850000
predicted=17.251489, expected=16.850000
predicted=19.695412, expected=16.850000
predicted=19.262843, expected=20.850000
predicted=24.757140, expected=24.850000
predicted=28.545299, expected=27.850000
predicted=30.116002, expected=29.850000
predicted=31.904223, expected=31.850000
predicted=32.715341, expected=31.850000
predicted=31.603303, expected=31.850000
predicted=31.440098, expected=30.850000
predicted=29.248135, expected=28.850000
predicted=26.532841, expected=25.850000
predicted=24.005504, expected=23.850000
predicted=22.123792, expected=21.850000
predicted=20.626481, expected=20.850000
predicted=20.433954, expected=19.850000
predicted=19.694693, expected=17.850000
predicted=17.377854, expected=17.850000
predicted=17.411340, expected=16.850000
predicted=16.166369, expected=16.850000
predicted=16.973852, expected=15.850000
predicted=15.236640, expected=15.850000
predicted=15.462112, expected=14.850000
predicted=14.824573, expected=14.850000
predicted=15.572400, expected=13.850000
predicted=15.988088, expected=13.850000
predicted=16.461134, expected=17.850000
predicted=21.429875, expected=21.850000
predicted=25.199990, expected=24.850000
predicted=28.076162, expected=25.850000
predicted=27.449359, expected=27.850000
predicted=28.880215, expected=27.850000
predicted=27.994910, expected=27.850000
predicted=27.116784, expected=27.850000
predicted=26.741848, expected=25.850000
predicted=23.878045, expected=23.850000
predicted=21.972677, expected=21.850000
predicted=20.770774, expected=20.850000
predicted=19.566555, expected=19.850000
predicted=18.906938, expected=17.850000
predicted=17.253209, expected=17.850000
predicted=17.570457, expected=16.850000
predicted=16.174185, expected=16.850000
predicted=16.425458, expected=15.850000
predicted=15.922224, expected=14.850000
predicted=13.635297, expected=14.850000
predicted=14.401663, expected=14.850000
predicted=14.493323, expected=13.850000
predicted=14.361676, expected=13.850000
predicted=15.737800, expected=13.850000
predicted=16.347131, expected=16.850000
predicted=19.440061, expected=20.850000
predicted=24.078788, expected=23.850000
```

```
predicted=26.119428, expected=25.850000
predicted=27.623183, expected=26.850000
predicted=27.592955, expected=26.850000
predicted=26.723722, expected=25.850000
Test RMSE: 0.973
```

```
/usr/local/lib/python3.11/dist-packages/statsmodels/tsa/base/tsa_model.py:473: Va
lueWarning: No frequency information was provided, so inferred frequency h will b
e used.
  self._init_dates(dates, freq)
<ipython-input-33-d33fabac8d4e>:7: FutureWarning: Series.__getitem__ treating key
s as positions is deprecated. In a future version, integer keys will always be tr
eated as labels (consistent with DataFrame behavior). To access a value by positi
on, use `ser.iloc[pos]`
  history = [history[i] for i in range(len(history))]
<ipython-input-33-d33fabac8d4e>:12: FutureWarning: Series.__getitem__ treating ke
ys as positions is deprecated. In a future version, integer keys will always be t
reated as labels (consistent with DataFrame behavior). To access a value by posit
ion, use `ser.iloc[pos]`
  yhat = coef[0]
<ipython-input-33-d33fabac8d4e>:14: FutureWarning: Series.__getitem__ treating ke
ys as positions is deprecated. In a future version, integer keys will always be t
reated as labels (consistent with DataFrame behavior). To access a value by posit
ion, use `ser.iloc[pos]`
  yhat += coef[d+1] * lag[window-d-1]
<ipython-input-33-d33fabac8d4e>:15: FutureWarning: Series.__getitem__ treating ke
ys as positions is deprecated. In a future version, integer keys will always be t
reated as labels (consistent with DataFrame behavior). To access a value by posit
ion, use `ser.iloc[pos]`
  obs = valid['Los Angeles'][t]
```

In [ ]:
```
valid['predictions'] = predictions
plt.plot(valid['Los Angeles'])
plt.plot(valid['predictions'], color='red')
plt.show()
```

```
<ipython-input-34-a53de82ca58c>:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stabl
e/user_guide/indexing.html#returning-a-view-versus-a-copy
  valid['predictions'] = predictions
```



Autoregressive Moving Average (ARMA) Model

The ARMA model is a combination of two simpler models: the autoregressive (AR) model and the moving average (MA) model. The ARMA model is used to describe time series data that are stationary, meaning that their statistical properties do not change over time.

- Autoregressive (AR) model: This model uses the dependence between an observation and a number of lagged observations (previous time points). It is denoted as AR(p), where p is the number of lagged observations included.
- Moving average (MA) model: This model uses the dependence between an observation and a residual error from a moving average model applied to lagged observations. It is denoted as MA(q), where ?q is the number of lagged forecast errors included.

How to determine the p and q orders in the ARMA model?

Determining the appropriate values for p and q is crucial to building an effective ARMA model. This can be done using the following methods:

Partial Autocorrelation Function (PACF):

The PACF is used to determine the p-order of the AR model. It measures the correlation between observations at different lags, excluding the influence of intermediate lags.

The p-order is determined by the lag at which the PACF plot is cut.

Autocorrelation Function (ACF):

The ACF is used to determine the q-order of the MA model. It measures the correlation between observations at different lags.

The q-order is determined by the lag at which the ACF plot is cut.

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call dri
ve.mount("/content/drive", force_remount=True).
```

In [ ]:
```
import pandas as pd
import matplotlib.pyplot as plt
```

Here we will use a new dataset:

Link to Dataset:
https://drive.google.com/file/d/10omy6A3pjEAbQzISjglvSkpiCpBpx3A_/view?usp=sharing

In [ ]:
```
path = '/content/drive/MyDrive/Datasets_TS/Global Temperature.csv'
```

In [ ]:
```
df = pd.read_csv(path)
```

In [ ]:
```
df.head()
```

Out[ ]:

	Year	Month	Monthly Anomaly	Monthly Unc.	Annual Anomaly	Annual Unc	Five-Year Anomaly	Five-Year Unc.	Ten-Year Anomaly	Ten Yea Unc
0	1850	1	-0.801	0.482	NaN	NaN	NaN	NaN	NaN	Nal
1	1850	2	-0.102	0.592	NaN	NaN	NaN	NaN	NaN	Nal
2	1850	3	-0.119	0.819	NaN	NaN	NaN	NaN	NaN	Nal
3	1850	4	-0.485	0.575	NaN	NaN	NaN	NaN	NaN	Nal
4	1850	5	-0.351	0.549	NaN	NaN	NaN	NaN	NaN	Nal

We will process the data and select from 1900:

In [ ]:
```
df.rename(columns={' Month': 'Month'}, inplace=True)
```

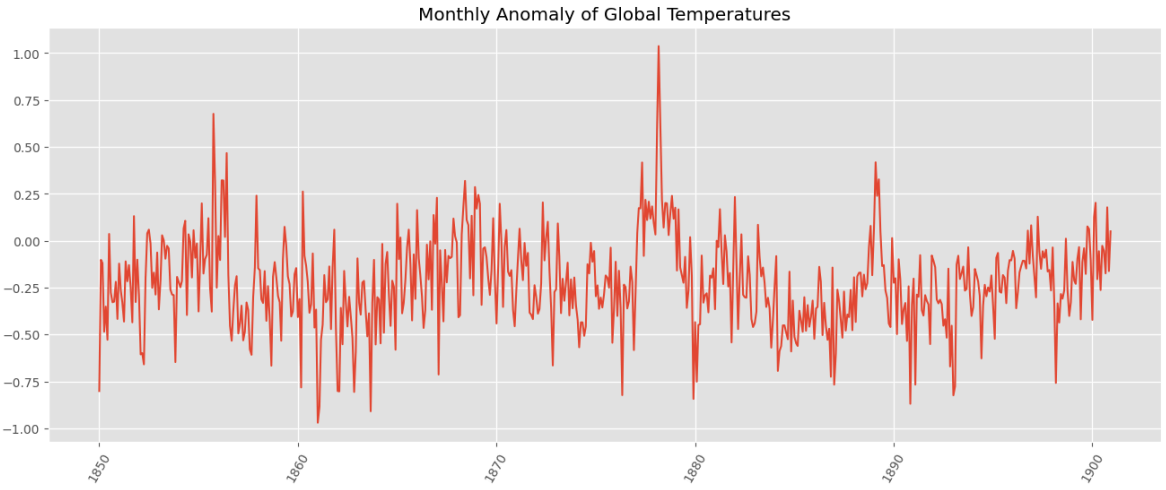
In [ ]:
```
df['Date'] = pd.to_datetime(df.Year.astype(str) + '/' + df.Month.astype(str) + '
```

In [ ]:
```
df.set_index('Date', inplace=True)
```

In [ ]:
```
df_1900 = df['1900'].copy()
```

In [ ]:
```
plt.figure(figsize=(16,6))
plt.style.use("ggplot")
plt.plot(df_1900.index, df_1900['Monthly Anomaly'])
```

```
plt.title('Monthly Anomaly of Global Temperatures')
plt.xticks(rotation=60)
plt.show()
```



In [ ]:
```
df_1900['Monthly Anomaly']
```

Out[ ]:

Date	Monthly Anomaly
1850-01-01	-0.801
1850-02-01	-0.102
1850-03-01	-0.119
1850-04-01	-0.485
1850-05-01	-0.351
...	...
1900-08-01	-0.054
1900-09-01	-0.174
1900-10-01	0.178
1900-11-01	-0.162
1900-12-01	0.050

612 rows × 1 columns

dtype: float64

To fit an ARMA model, the time series data must be stationary. First, we check for stationarity and, if necessary, difference the data to make it stationary. We use the Augmented Dickey-Fuller (ADF) test to check for stationarity. The ADF test provides a statistic and a p-value. If the p-value is less than 0.05, the series is considered stationary.

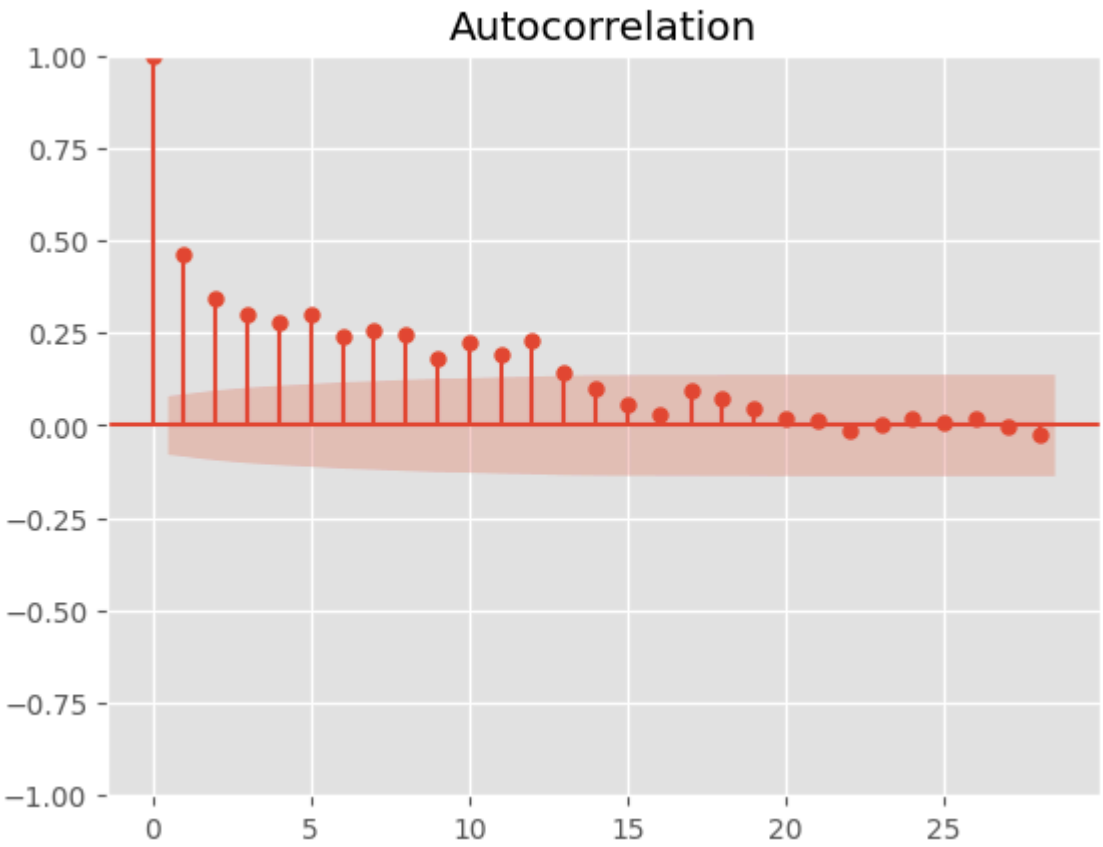
In [ ]:
```
from statsmodels.tsa.stattools import adfuller

# Check for stationarity
```

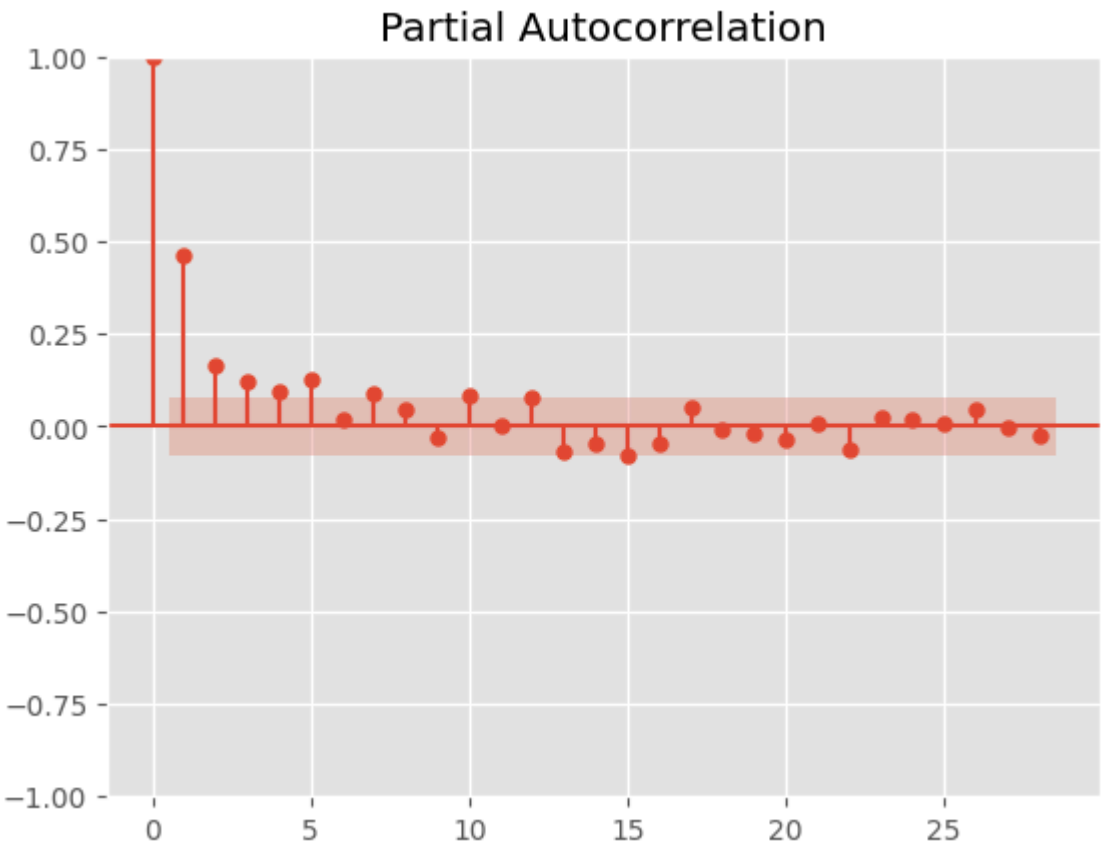
```
result = adfuller(df_1900['Monthly Anomaly'])
print('ADF Statistic:', result[0])
print('p-value:', result[1])
```

```
ADF Statistic: -4.532529896227389
p-value: 0.0001716719931140968
```

In [ ]:

```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

# Plot ACF
plot_acf(df_1900['Monthly Anomaly'])
plt.show()

# Plot PACF
plot_pacf(df_1900['Monthly Anomaly'])
plt.show()
```





Let's apply the ARMA model:

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

model = ARIMA(df_1900['Monthly Anomaly'], order=(9, 0, 21))
model_fit = model.fit()
# Print the model summary
print(model_fit.summary())
```

```
/usr/local/lib/python3.11/dist-packages/statsmodels/tsa/base/tsa_model.py:473: Va
lueWarning: No frequency information was provided, so inferred frequency MS will
be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.11/dist-packages/statsmodels/tsa/base/tsa_model.py:473: Va
lueWarning: No frequency information was provided, so inferred frequency MS will
be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.11/dist-packages/statsmodels/tsa/base/tsa_model.py:473: Va
lueWarning: No frequency information was provided, so inferred frequency MS will
be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.11/dist-packages/statsmodels/base/model.py:607: Convergenc
eWarning: Maximum Likelihood optimization failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
```

SARIMAX Results

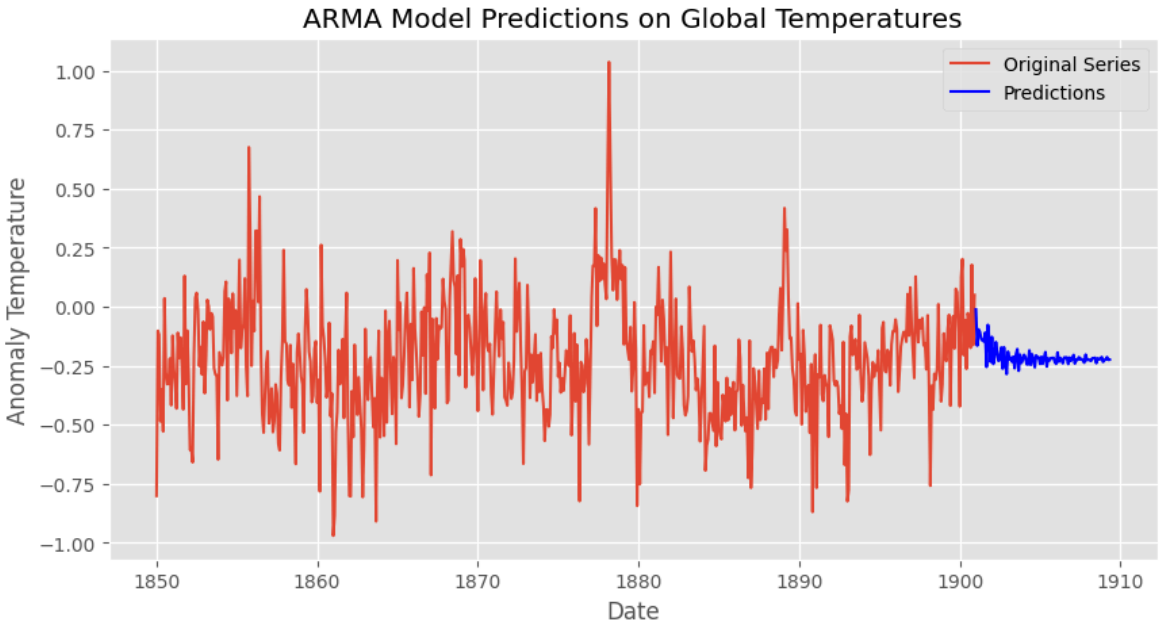
Dep. Variable:	Monthly Anomaly	No. Observations:	612
Model:	ARIMA(9, 0, 21)	Log Likelihood	112.804
Date:	Fri, 23 May 2025	AIC	-161.608
Time:	00:03:28	BIC	-20.272
Sample:	01-01-1850 - 12-01-1900	HQIC	-106.637
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
const	-0.2227	0.030	-7.525	0.000	-0.281	-0.165
ar.L1	-0.0409	0.397	-0.103	0.918	-0.818	0.736
ar.L2	-0.1461	0.221	-0.662	0.508	-0.578	0.286
ar.L3	-0.0282	0.217	-0.130	0.896	-0.453	0.396
ar.L4	-0.3072	0.222	-1.382	0.167	-0.743	0.129
ar.L5	0.2638	0.147	1.792	0.073	-0.025	0.552
ar.L6	0.0714	0.201	0.356	0.722	-0.322	0.465
ar.L7	0.0049	0.184	0.026	0.979	-0.355	0.365
ar.L8	-0.5838	0.189	-3.087	0.002	-0.954	-0.213
ar.L9	0.3214	0.333	0.964	0.335	-0.332	0.975
ma.L1	0.3873	0.397	0.975	0.330	-0.392	1.166
ma.L2	0.3652	0.295	1.237	0.216	-0.213	0.944
ma.L3	0.2648	0.297	0.890	0.373	-0.318	0.848
ma.L4	0.4999	0.295	1.696	0.090	-0.078	1.078
ma.L5	0.0664	0.252	0.263	0.792	-0.428	0.561
ma.L6	0.0354	0.255	0.139	0.890	-0.465	0.535
ma.L7	0.1427	0.241	0.593	0.553	-0.329	0.614
ma.L8	0.8217	0.227	3.627	0.000	0.378	1.266
ma.L9	-0.0214	0.390	-0.055	0.956	-0.785	0.743
ma.L10	0.1961	0.142	1.380	0.168	-0.082	0.475
ma.L11	0.1791	0.130	1.382	0.167	-0.075	0.433
ma.L12	0.3106	0.110	2.827	0.005	0.095	0.526
ma.L13	0.1873	0.150	1.249	0.212	-0.107	0.481
ma.L14	0.1540	0.106	1.449	0.147	-0.054	0.362
ma.L15	0.1233	0.102	1.208	0.227	-0.077	0.323
ma.L16	0.1340	0.093	1.446	0.148	-0.048	0.316
ma.L17	0.0150	0.099	0.151	0.880	-0.180	0.210
ma.L18	0.1568	0.083	1.891	0.059	-0.006	0.319
ma.L19	0.0467	0.087	0.540	0.589	-0.123	0.216
ma.L20	0.1919	0.071	2.709	0.007	0.053	0.331
ma.L21	0.0534	0.090	0.594	0.552	-0.123	0.230
sigma2	0.0398	0.002	18.316	0.000	0.036	0.044

Ljung-Box (L1) (Q):	0.00	Jarque-Bera (JB):	15.11
Prob(Q):	0.97	Prob(JB):	0.00
Heteroskedasticity (H):	0.63	Skew:	0.03
Prob(H) (two-sided):	0.00	Kurtosis:	3.77

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).

In [ ]:
```
start = len(df_1900['Monthly Anomaly'])
end = start + 20
predictions = model_fit.predict(start=start, end=end)

# Plot the results
plt.figure(figsize=(10, 5))
plt.plot(df_1900['Monthly Anomaly'], label='Original Series')
plt.plot(predictions, label='Predictions', color='blue')
plt.legend()
plt.title('ARMA Model Predictions on Global Temperatures')
plt.xlabel('Date')
plt.ylabel('Anomaly Temperature')
plt.show()
```



Thank you! See you in the next Chapter!

References:

https://towardsdatascience.com/time-series-forecasting-with-arima-sarima-and-sarimax-ee61099e78f6

https://forecastegy.com/posts/time-series-cross-validation-python/

https://medium.com/@wainaina.pierre/the-complete-guide-to-time-series-forecasting-models-ef9c8cd40037

https://www.investopedia.com/terms/a/autoregressive.asp

https://www.turing.com/kb/guide-to-autoregressive-models

https://machinelearningmastery.com/time-series-forecasting-methods-in-python-cheat-sheet/

https://towardsdatascience.com/an-end-to-end-project-on-time-series-analysis-and-forecasting-with-python-4835e6bf050b

https://www.analyticsvidhya.com/blog/2021/10/a-comprehensive-guide-to-time-series-analysis/

https://www.geeksforgeeks.org/time-series-analysis-and-forecasting/

https://www.geeksforgeeks.org/autoregressive-ar-model-for-time-series-forecasting/

https://www.analyticsvidhya.com/blog/2018/02/time-series-forecasting-methods/

https://machinelearningmastery.com/autoregression-models-time-series-forecasting-python/

https://www.geeksforgeeks.org/arma-time-series-model/

https://medium.com/@rajukumardalimss/how-to-run-an-arma-model-in-python-a-step-by-step-guide-7acc5dd68713

https://github.com/Apress/advanced-forecasting-python/blob/main/Chapter%2003%20-%20The%20AR%20Model.ipynb