Advanced Time Series Modeling and Forecasting

Neba Nfonsang

University of Denver

Introduction

- We will examine different advanced approaches to time series forecasting including:
  - Autoregressive (AR)
  - Moving Average (MA)
  - Mixed Autoregressive-Moving Average (ARMA)
  - Autoregressive Integrated Moving Average (ARIMA)

Packages Used

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import pydataset
import pandas_datareader as pdr
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.tsa.stattools import acf
from pandas.plotting import autocorrelation_plot
from statsmodels.formula.api import ols
from statsmodels.tsa.ar_model import AR
from statsmodels.tsa.seasonal import seasonal_decompose
from sklearn.metrics import r2_score
from sklearn.metrics import mean_squared_error
from statsmodels.tsa.arima_model import ARMA
from statsmodels.tsa.arima_model import ARIMA
import statsmodels.tsa
```

Stochastic Process

- A stochastic process is a models that describe the probability distribution of a sequence of observation.
- A stochastic process is a random process, that cannot be predicted or determined
- A process that can be determined or predicted is said to be deterministic.
  - A regression model, we can predict the values of y given x.
- Note that, time series data can be continuous or discrete.

Reinsel, Jenkins, and Box (2008)

Stochastic Processes

- Time series is said to be **deterministic** if future values can be **exactly** determined by some mathematical function such as $y_t = \cos(2\pi f t)$
- A time series is **statistical** if the future values are determined only by a probability distribution.

Reinsel, Jenkins, and Box (2008)

- Trends and seasonal patterns in a time series are stochastic because they cannot be predicted in the long-run.

Stochastic Processes

- The mean of a stochastic process is the same as the sample mean of time series data.
- The variance is the same as the sample variance for the time series data.

- Note that, **Gaussian process** is the probability distribution generating a sequence of time series observations that follows a multivariate normal distribution.

Stationary Stochastic Processes

- **A stationary process** is a special class of the stochastic process where the process is assumed to be in state of statistical equilibrium.
- Time series generated by a stationary process fluctuates around a fixed mean and has a constant variance.
- That is, stationary time series data has a fairly stable mean and variance over time.
- Such time series data does not have a trend or seasonality.

Stationary Stochastic Process

- From the time series plot of Air Passengers, it can be noticed that the time series is not stationary because the mean is increasing with time (not stable) and the variance is increasing with time (not stable).

```
import pydataset
```

```
air_passenger = pydataset.data("AirPassengers")
air_passenger.head()
```

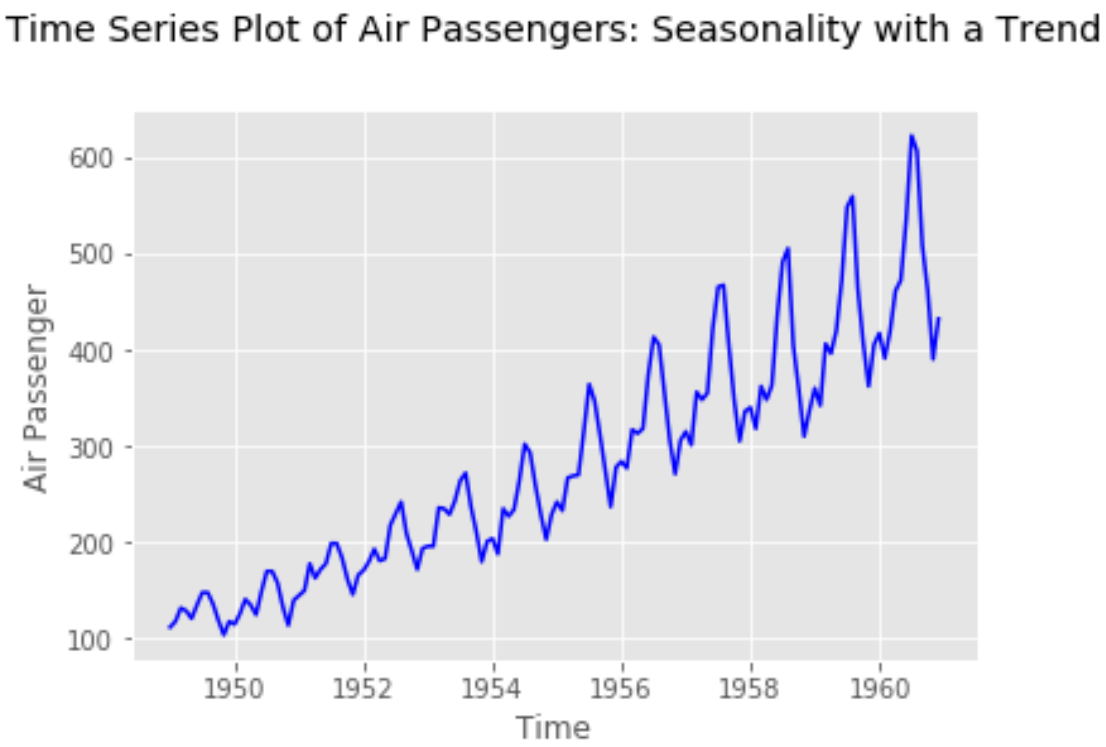
	time	AirPassengers
1	1949.000000	112
2	1949.083333	118
3	1949.166667	132
4	1949.250000	129
5	1949.333333	121

Data Source:

Box, G. E. P., Jenkins, G. M. and Reinsel, G. C. (1976). Time Series Analysis, Forecasting and Control. Third Edition. Holden-Day. Series G.

Stationary Stochastic Process

```
plt.style.use("ggplot")
#plt.figure(figsize=(15, 6))
plt.title("Time Series Plot of Air Passengers: \
Seasonality with a Trend", y=1.1)
plt.xlabel("Time")
plt.ylabel("Air Passenger")
plt.plot(air_passenger.time,
         air_passenger.AirPassengers, c="b");
```



Data Source:

Box, G. E. P., Jenkins, G. M. and Reinsel, G. C. (1976). Time Series Analysis, Forecasting and Control. Third Edition. Holden-Day. Series G.

Stationary Stochastic Process

- Some time series models require time series data to be stationary. Such models include:
  - Autoregressive (AR)
  - Moving Average (MA)
  - Mixed Autoregressive-Moving Average (ARMA) models, etc.
- Most time series data such as stock data is not stationary, and can be model through a non-stationary models called:
  - Autoregressive Integrated Moving Average (ARIMA)
- These models are the workhorse of time series analysis.

Strictly and Weakly Stationary

- For time series data to be **strictly stationary**, the joint probability of the data in one period t is the same as the joint probability of the data in another period t + k where k is an integer.

- A stochastic process is weakly stationary if:
  - It has a constant mean at all times points
  - It has a constant variance at all time points
  - The covariance between time t and its lag t-k is constant for all times. So, the covariance depends only on k.

Why Stationarity Matters

- Stationarity matters because a large number of models assume stationarity.
- A model used for a time series data will have an accuracy that varies with time since the time series metrics (for example mean and variance) vary over time.
- **How can you make a non-stationary non-time series to be stationary?**
  - Use the log transformation or square root transformation to address the changing variance.
  - Use differencing to address the trend issues.

(Nielson, 2019)

Another Assumption of Forecasting Models

- Statistical time series models such as AR, MA, ARMA and ARIMA assume that the data for the input and predicted variable is normally distributed.
- The Box Cox transformation can be used to transform non-normal data into normal data.
- The scipy.stats module in Python has the box-cox transformation functionality.

(Nielson, 2019)

Why Not use Linear Regression?

- Statistical time series models are used instead of linear regression in time series analysis because statistical models take into account the autocorrelation between past and future values.
- To apply OLS regression, all the assumptions must hold.

- **Linear regression are not very suitable for forecasting time series data** because linear regression assumes that the data is independently identically distributed (iid) but this is not the case for time series data which has autocorrelation.

(Nielson, 2019)

Why Not use Regression?

- **To use OLS regression for time series, the assumptions of OLS must hold.**
  - That means, the errors must be independent (uncorrelated)
  - The errors must have a mean of zero at each point in time
  - The variance of the errors must be constant across different time periods.
- If these assumptions are satisfied, then OLS regression can be used for time series forecasting.

Exploratory Concepts in Time Series

- According to Nielsen (2019), three exploratory concepts in time series analysis are:
  - Stationarity: check if the data is stationary
  - Self-correlation (autocorrelation and partial autocorrelation): check to what extend previous or past values predict future values.
  - Spurious correlations:
- Preliminary time series also involves using exploratory graphs such as:
  - Autocorrelation function
  - Window functions

Window Functions

- Window functions allow us to compute aggregates or summary statistics over time in a flexible way.
- The aggregates compress or smoothen the time series.
- Practically used for moving average smoothing (simple moving average calculations).
- Statistics computed with window functions include descriptive statistics such as mean, sum, count, standard deviation, skewness, etc.
- Types of windows include rolling and expanding windows.

Window Functions

Rolling Window

- A rolling window has a fixed window size that includes previous data up to the current data (most recent k datapoints) to generate the average for the current period.
- Rolling windows are used to compute simple moving averages for smoothing.
- Note that moving average smoothing is different from moving average model.

Window Functions

Expanding window

- Expanding window is useful for estimating stable statistics that do no evolve over time.
- The expanding window is meaningful when the statistic being computed is stable over time.
- The size of the expanding window grows with time.
- Expanding window uses all past data and the current data to compute an aggregate.
- Expanding window is used to generated cumulative statistics over time.

Moving Average Smoothing

- **Moving Average Smoothing can be used for:**
  - Data preparation: involves smoothening a time series by removing the variations or fluctuations so as estimate the cycle-trend. As such, the underlying process for the time series is clearer as the noise is removed.
  - Predictions or forecasting: Using moving average smoothing for forecasting assumes that the trend and seasonality has been removed, so this is a naïve method. This is suitable when there is a horizontal pattern in the time series.

Types of Moving Averages

Centered Moving Average

- A centered Moving Average is generally used to remove seasonality and trend.
- A centered moving average at time t is calculated using time points before and after the time t.

$$Centered\_MA(t) = mean(x_{t+1}, x_t, x_{t-2})$$

$$for\ a\ window\ size\ of\ 3$$

Types of Moving Averages

Trailing Moving Average

- The trailing moving average at time t is calculated using the time series at time t and values before t.
- The transformed series is usually lagged compared to original time series data.

Trailing_$MA(t) = mean(x_t, x_{t-1}, x_{t-2})$

$for\ a\ window\ size\ of\ 3$

Moving Average Smoothing for Data Preparation

- As a data preparation technique, moving average smoothing is used to remove the variations from the time series.
- The rolling() method of the pandas Series can be used to compute moving averages.
- By default, the trailing moving average is computed.
- A centered moving average can be computed by turning the center parameter to “True”.
- Moving average smoothing is used for data transformation.

Moving Average Smoothing with Rolling Window

```
# to apply rolling window,
# first set the time as index
air_pass = air_passenger.set_index("time")
air_pass.head()
```

time	AirPassengers
1949.000000	112
1949.083333	118
1949.166667	132
1949.250000	129
1949.333333	121

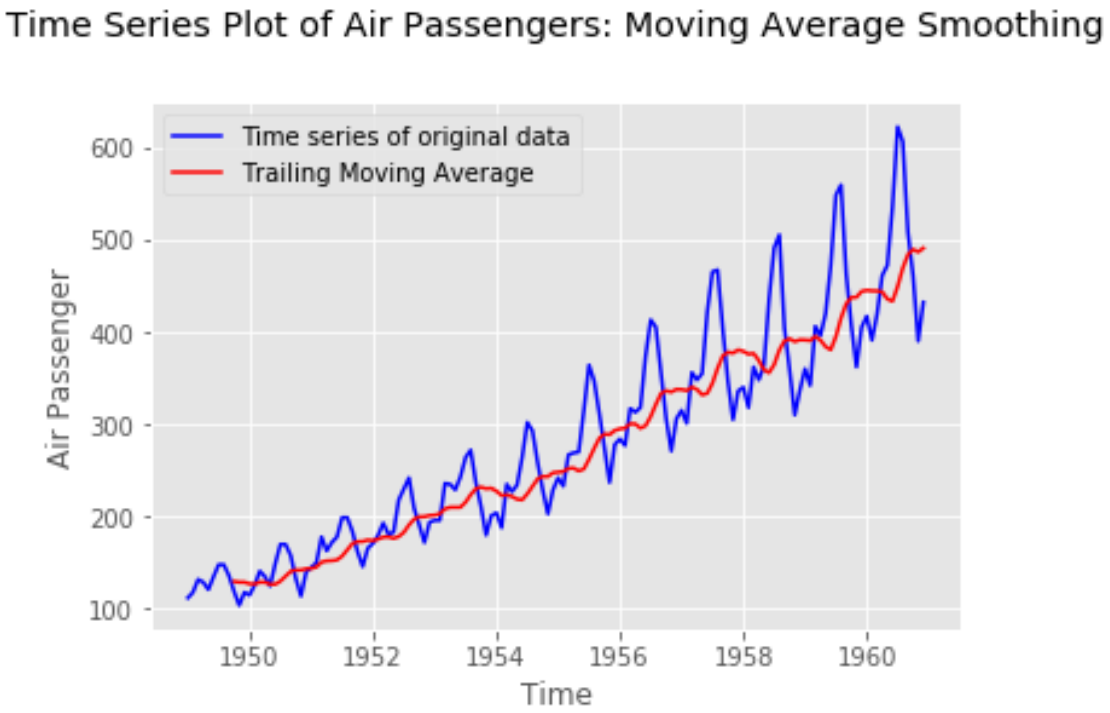
The rolling window is used for computing simple moving averages

```
rolling = air_pass.rolling(10).mean()
rolling.head(12)
```

time	AirPassengers
1949.000000	NaN
1949.083333	NaN
1949.166667	NaN
1949.250000	NaN
1949.333333	NaN
1949.416667	NaN
1949.500000	NaN
1949.583333	NaN
1949.666667	NaN
1949.750000	129.8
1949.833333	129.0
1949.916667	129.0

Moving Average Smoothing with Rolling Window

```
plt.style.use("ggplot")
#plt.figure(figsize=(15, 6))
plt.title("Time Series Plot of Air Passengers: \
Moving Average Smoothing", y=1.1)
plt.xlabel("Time")
plt.ylabel("Air Passenger")
plt.plot(air_passenger.time,
         air_passenger.AirPassengers, c="b")
plt.plot(rolling.index,
         rolling.AirPassengers, c="r")
plt.legend(["Time series of original data",
            "Trailing Moving Average"]);
```



Moving Average Smoothing with Rolling Window

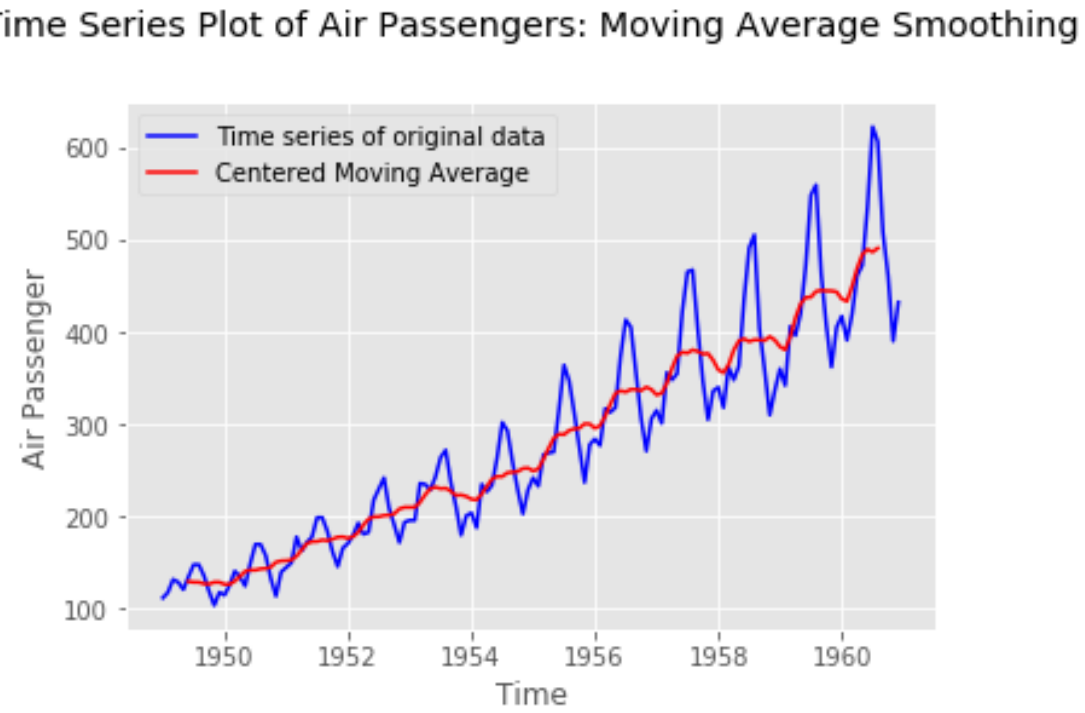
- Centered moving averages are computed by setting the centered parameter of the rolling() method to “True”.
- Compared to trailing moving averages, the transformed series is shifted backward for the centered moving averages.

```
# centered moving average
rolling_centered = air_pass.rolling(10, center=True).mean()
rolling_centered.head(12)
```

	AirPassengers
time	
1949.000000	NaN
1949.083333	NaN
1949.166667	NaN
1949.250000	NaN
1949.333333	NaN
1949.416667	129.8
1949.500000	129.0
1949.583333	129.0
1949.666667	127.3
1949.750000	127.0
1949.833333	129.0
1949.916667	129.0

Moving Average Smoothing with Rolling Window

```
plt.style.use("ggplot")
#plt.figure(figsize=(15, 6))
plt.title("Time Series Plot of Air Passengers: \
Moving Average Smoothing", y=1.1)
plt.xlabel("Time")
plt.ylabel("Air Passenger")
plt.plot(air_passenger.time,
         air_passenger.AirPassengers, c="b")
plt.plot(rolling_centered.index,
         rolling_centered.AirPassengers, c="r")
plt.legend(["Time series of original data",
            "Centered Moving Average"]);
```



Moving Average Smoothing with Expanding Window

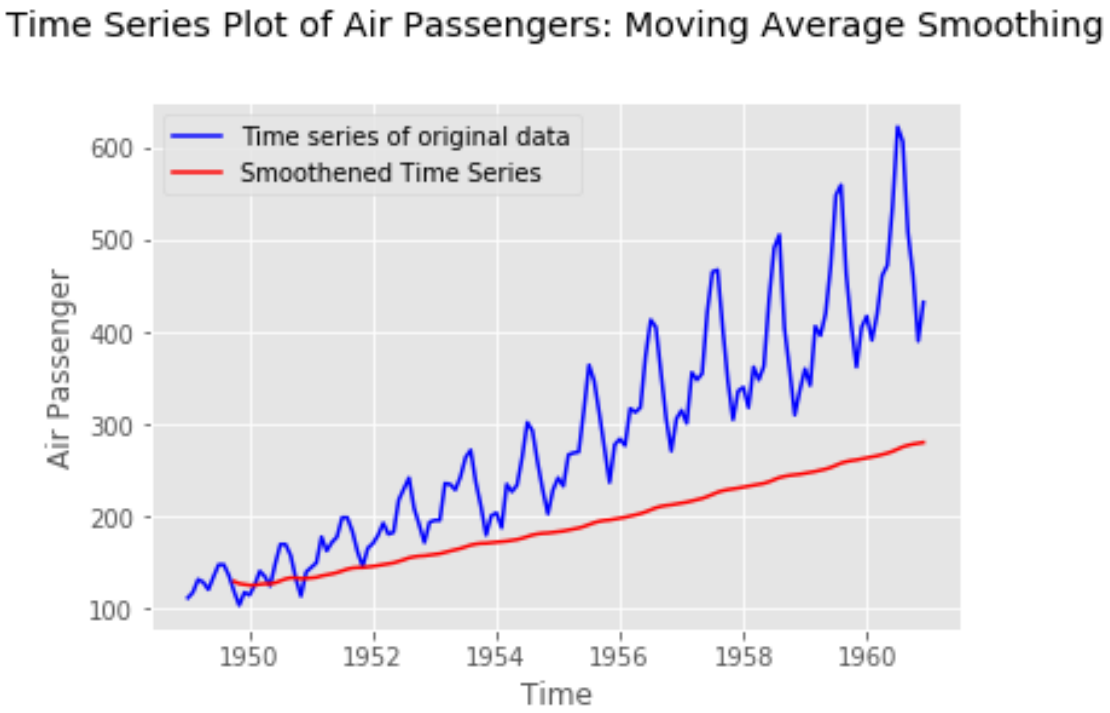
Expanding window is used to compute cumulative moving averages. Expanding window does not do a good job in capturing the trend when the time series mean at each point in time changes.

```
expanding = air_pass.expanding(10).mean()
expanding.head(12)
```

	AirPassengers
time	
1949.000000	NaN
1949.083333	NaN
1949.166667	NaN
1949.250000	NaN
1949.333333	NaN
1949.416667	NaN
1949.500000	NaN
1949.583333	NaN
1949.666667	NaN
1949.750000	129.800000
1949.833333	127.454545
1949.916667	126.666667

Moving Average Smoothing with Expanding Window

```
plt.style.use("ggplot")
#plt.figure(figsize=(15, 6))
plt.title("Time Series Plot of Air Passengers: \
Moving Average Smoothing", y=1.1)
plt.xlabel("Time")
plt.ylabel("Air Passenger")
plt.plot(air_passenger.time,
         air_passenger.AirPassengers, c="b")
plt.plot(expanding.index,
         expanding.AirPassengers, c="r")
plt.legend(["Time series of original data",
            "Smoothened Time Series"]);
```



Lag Time

- If two time series values are separated by k interval of time, then the lag between those values is said to be k.
- Lag is the time interval that separates two time series values.
- If the time lag is 1, it means one value at time t and the other value is at t-1. That is, a value and the previous value.
- Two points in a time series separated by lag k are k intervals apart, where k is an integer.

Autocorrelation

- Autocorrelation is when data points at time t always correlate with points at another time such as t-1 or t-2 through out the entire time series data.
- Autocorrelation at lag k is correlation between a time series data point and it’s kth lag.
- Autocorrelation is also called serial correlation because values of the same time series are correlated.

Autocorrelation

- **Stationary time series data can be described by its:**
  - Constant mean
  - Constant variance
  - Constant Autocorrelation function

Reinsel, Jenkins, and Box (2008)

- ***Mean:*** $\mu = E(y_t) = E(y_{t+k})$ *since mean is constant for all time periods.*
- $\boldsymbol{Variance}$: $\sigma^2 = var(y_t) = var(y_{t+k} = E[(y_t - \mu)^2$
- *Since variance is constant across all points*

Autocorrelation

- First, autocovariance between values $y_t \; and \; y_{t+1}$ at lag k is given by:
- $\gamma_k = \text{cov}(y_t, y_{t+k})$

$\gamma_k = E[(y_t - \mu)(y_{t+k} - \mu)]$

Where $\mu = E(y_t) = E(y_{t+k})$ *since mean is constant for all time periods.*

- Autocorrelation between values $y_t \; and \; y_{t+1}$ at lag k is given by:
- $\rho_k = \text{corr}(y_t, y_{t+k})$
- $\rho_k = \frac{cov(y_t, y_{t+k})}{\sqrt{var(y_t) * var(y_{t+k})}} = \frac{\gamma_k}{\sigma^2} = \frac{\gamma_k}{\gamma_0}$

$var(y_t) = var(y_{t+k}) = \sigma^2 = \gamma_0$

Since variance is constant across all time periods

Autocorrelation Function

- A plot of autocorrelation coefficients ρ_k versus the corresponding lag k gives the **autocorrelation function**.
- For a stationary time series, the autocorrelation function is constant.
- For a time series with a trend, smaller lags tend to have larger and positive autocorrelations.
- For a time series with seasonality, there are larger auto regressions at the seasonal lags (multiples of seasonal frequency).

Hyndman & Athanasopoulos (2018)

Create a Lag of 1

- The pandas Series Method, .shift() can be used to create a lag by shifting a time series.
- The lag number is passed into the shift() method. A positive number shifts the series forward while a negative number shifts the series backward.

```
air = air_pass.copy()
air["lag1"] = air_pass["AirPassengers"].shift(1)
air.head()
```

time	AirPassengers	lag1
1949.000000	112	NaN
1949.083333	118	112.0
1949.166667	132	118.0
1949.250000	129	132.0
1949.333333	121	129.0

Create Multiple Lags

```
for i in range(1, 21):
    air[f"lag{i}"] = air_pass["AirPassengers"].shift(i)
air.head(10)
```

time	AirPassengers	lag1	lag2	lag3	lag4	lag5	lag6	lag7	lag8	lag9	...
1949.000000	112	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...
1949.083333	118	112.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...
1949.166667	132	118.0	112.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...
1949.250000	129	132.0	118.0	112.0	NaN	NaN	NaN	NaN	NaN	NaN	...
1949.333333	121	129.0	132.0	118.0	112.0	NaN	NaN	NaN	NaN	NaN	...
1949.416667	135	121.0	129.0	132.0	118.0	112.0	NaN	NaN	NaN	NaN	...
1949.500000	148	135.0	121.0	129.0	132.0	118.0	112.0	NaN	NaN	NaN	...
1949.583333	148	148.0	135.0	121.0	129.0	132.0	118.0	112.0	NaN	NaN	...
1949.666667	136	148.0	148.0	135.0	121.0	129.0	132.0	118.0	112.0	NaN	...
1949.750000	119	136.0	148.0	148.0	135.0	121.0	129.0	132.0	118.0	112.0	...

10 rows × 21 columns

Compute Autocovariances

```
# compute autocovariance for a single lag
air.AirPassengers.cov(air.lag1)
```

```
13741.562198365013
```

```
# compute autocovariance for multiple lags
cov_list = []
lags = range(1,21)
for i in lags:
    covariance = air.AirPassengers.cov(air[f"lag{i}"])
    cov_list.append(covariance)
print(cov_list)
```

```
[13741.562198365013, 12784.257167116171, 11869.192958459978, 11165.1
1798561151, 10685.684130956106, 10307.269438273564, 10122.6901030485
17, 10111.148583877997, 10450.690768380318, 11051.489339019188, 1180
0.03964456596, 12196.593627110802, 11574.180152671757, 10641.4422182
46868, 9794.949794089149, 9149.189468503937, 8674.30889888764, 8309.
871999999996, 8128.054451612902, 8114.742460005246]
```

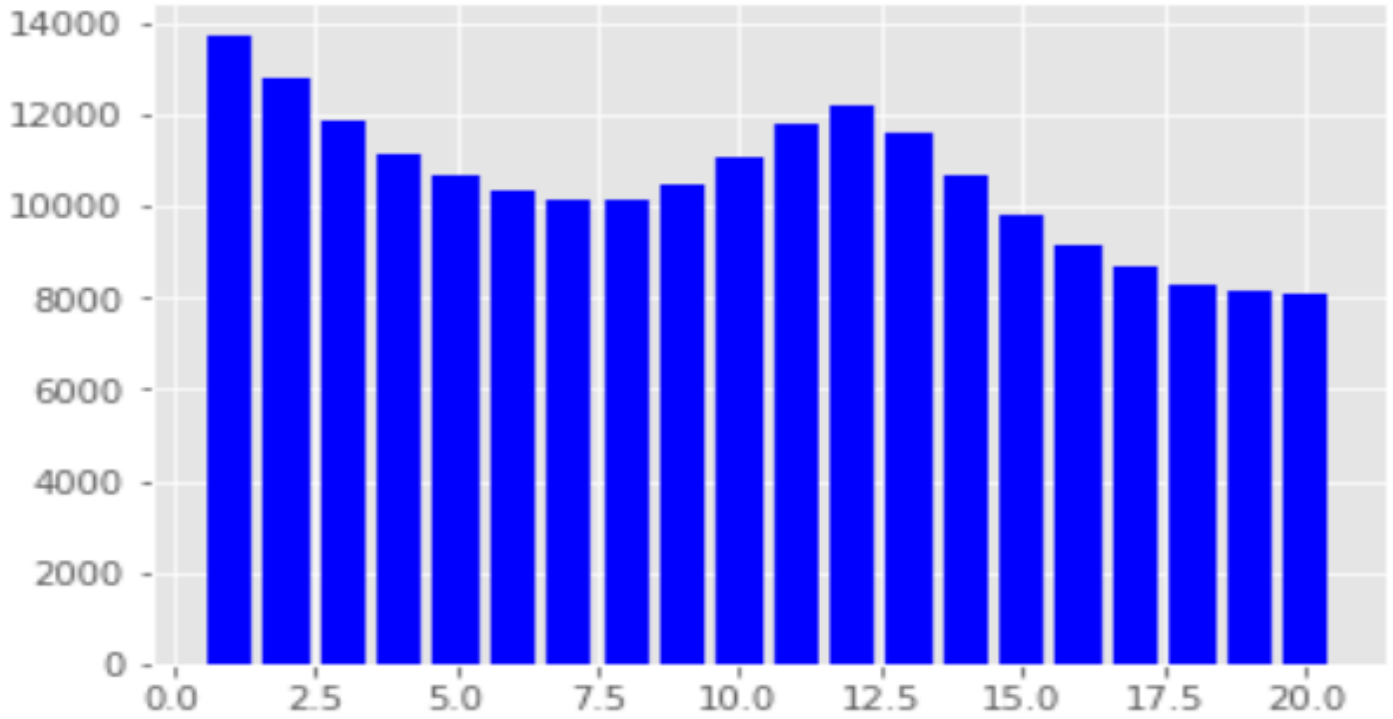
Compute Autocorrelation

```
corr_list = []
lags = range(1,21)
for i in lags:
    correlation = air.AirPassengers.corr(air[f"lag{i}"])
    corr_list.append(correlation)
print(corr_list)
```

```
[0.9601946480498522, 0.8956753113926392, 0.837394765081794, 0.797734
6989350624, 0.7859431491184304, 0.7839187959206183, 0.7845921291388301, 0.792215047259574, 0.8278519011167602, 0.8827127951607842, 0.94
97020331006317, 0.9905273692085446, 0.9481066160592017, 0.8754477915
539791, 0.8114659384543108, 0.7694487920842656, 0.7558191230371455,
0.7487523142605247, 0.7455000168641182, 0.7517886585378154]
```

Plot of Autocovariance Function

```
# autocovariance function (correlogram)
plt.bar(lags, cov_list, color="b");
```



Compute Autocorrelations Using statsmodels

```
# create the autocorrelation function from
from statsmodels.tsa.stattools import acf
autocorr = acf(air.AirPassengers, nlags=40, fft=False)
autocorr
```

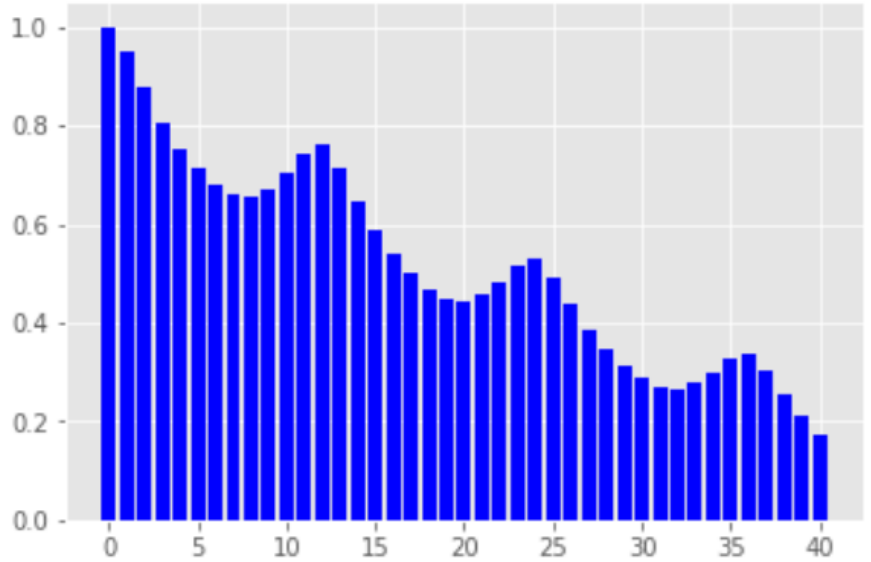
```
array([1.        , 0.94804734, 0.87557484, 0.80668116, 0.75262542,
       0.71376997, 0.6817336 , 0.66290439, 0.65561048, 0.67094833,
       0.70271992, 0.74324019, 0.76039504, 0.71266087, 0.64634228,
       0.58592342, 0.53795519, 0.49974753, 0.46873401, 0.44987066,
       0.4416288 , 0.45722376, 0.48248203, 0.51712699, 0.53218983,
       0.49397569, 0.43772134, 0.3876029 , 0.34802503, 0.31498388,
       0.28849682, 0.27080187, 0.26429011, 0.27679934, 0.2985215 ,
       0.32558712, 0.3370236 , 0.30333486, 0.25397708, 0.21065534,
       0.17217092])
```

Note: All time series have an autocorrelation of 1 at lag 0 (Nielsen, 2019)

A Plot of Autocorrelation Function

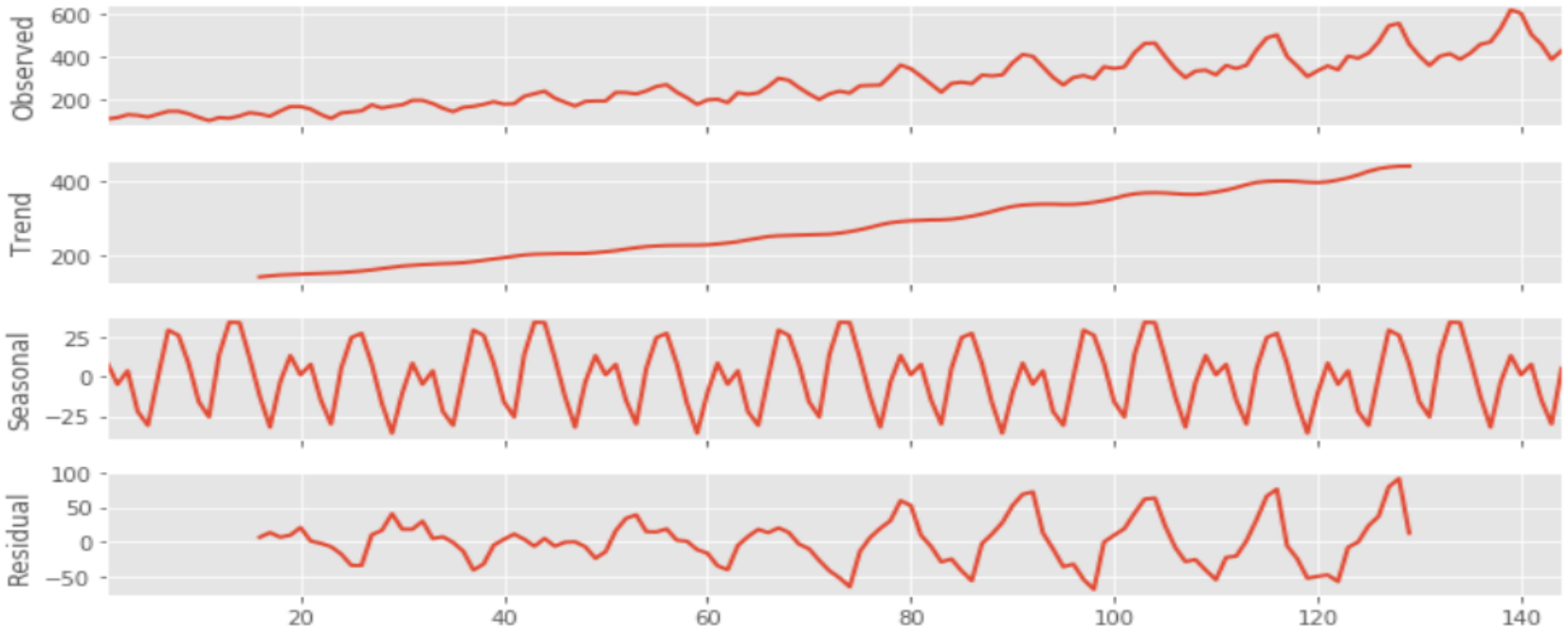
- It can be seen that autocorrelations are higher at seasonal lags showing the presence of seasonality.
- Autocorrelations are higher for smaller lags indicating the presence of a trend in the time series data

```
plt.bar(range(len(autocorr)), autocorr, color="b");
```



A Plot of Autocorrelation Function

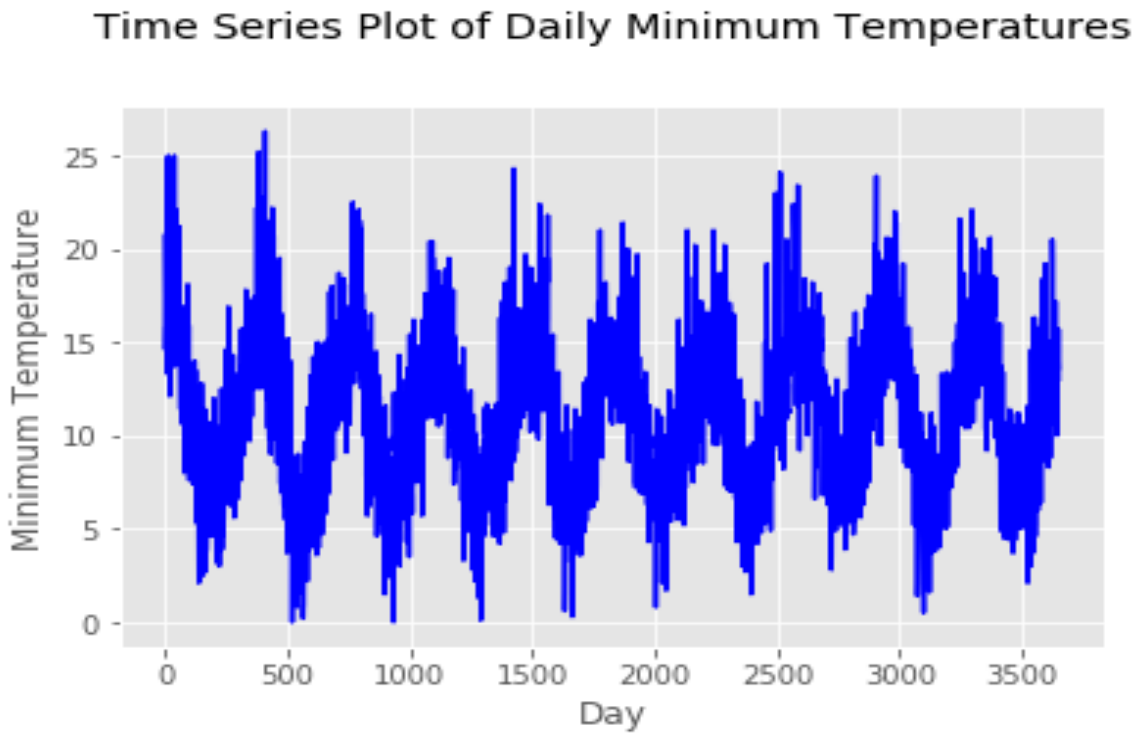
```
# we can also decompose() method in statsmodel
# to check for seasonality and trend
#statsmodels.tsa.seasonal.seasonal_decompose

from statsmodels.tsa.seasonal import seasonal_decompose
decomp = seasonal_decompose(air_passenger["AirPassengers"],
                            model='additive', freq=30)
decomp.plot();
```



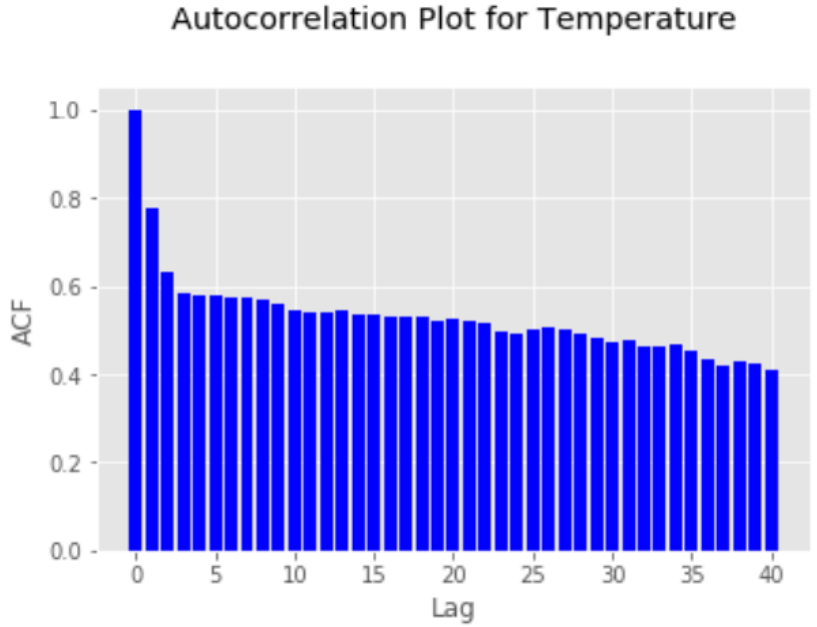
Seasonal decomposition using moving averages.

A Plot of Autocorrelation Function

```
url = r"https://raw.githubusercontent.com/jbrownlee/Datasets/master/daily-min-temperatures.csv"
temperature = pd.read_csv(url)
temperature.set_index("Date")
plt.title("Time Series Plot of Daily Minimum Temperatures", y=1.1)
plt.xlabel("Day")
plt.ylabel("Minimum Temperature")
plt.plot(temperature.index, temperature.Temp, color="b");
```




```
acf_temp = acf(temperature.Temp,  nlags=40, fft=False)
plt.bar(range(len(acf_temp)), acf_temp, color="b")
plt.title("Autocorrelation Plot for Temperature", y=1.1)
plt.xlabel("Lag")
plt.ylabel("ACF");
```



Statistical Models for Time Series

Time Series Statistical Models

- We will take a look at the following time series statistical models:
  - Autoregressive (AR)
  - Moving Average (MA)
  - Mixed Autoregressive-Moving Average (ARMA)
  - Autoregressive Integrated Moving Average (ARIMA)
- The first three models are used when the data is stationary
- The last model is used when the time series data is non-stationary.
- These models are used for univariate time series.

Autoregressive (AR) Model

- An autoregressive (AR) model is a model where the p past values predict the future values.
- The forecast at time t is a linear function of a specified number of time series values at previous time periods.

- An AR model can be formulated as:
- $y_t = f\left(y_{t-1}, y_{t-2}, \cdots, y_{t-p}, \epsilon_t\right)$
- An AR(p) Model is generally written as:

$$y_t = \beta_0 + \beta_1 * y_{t-1} + \beta_2 * y_{t-2} + \ldots + \beta_p * y_{t-p} + \epsilon_t$$

AR Model: Lag Variables

- An AR(p) model takes p parameters which corresponds to the number of lag observation included in the model.
- The number of parameters is also called lag order.

```
# AR(3): AR model of order 3
temp_AR = temperature.copy()
for i in range(1, 4):
    temp_AR[f"temp_{i}"] = temp_AR["Temp"].shift(i)
temp_AR.head(10)
```

	Date	Temp	temp_1	temp_2	temp_3
0	1981-01-01	20.7	NaN	NaN	NaN
1	1981-01-02	17.9	20.7	NaN	NaN
2	1981-01-03	18.8	17.9	20.7	NaN
3	1981-01-04	14.6	18.8	17.9	20.7
4	1981-01-05	15.8	14.6	18.8	17.9
5	1981-01-06	15.8	15.8	14.6	18.8
6	1981-01-07	15.8	15.8	15.8	14.6
7	1981-01-08	17.4	15.8	15.8	15.8
8	1981-01-09	21.8	17.4	15.8	15.8
9	1981-01-10	20.0	21.8	17.4	15.8

We want to fit an AR(3) model manually to show how AR Models work.
We start by creating the lag variables, but later on, the packages will do this for us behind the hood and choose the optimal number of lag variables for us.

AR Model: How it Works

```
from statsmodels.formula.api import ols

formula = "Temp ~ temp_1 + temp_2 + temp_3"
model = ols(formula, data=temp_AR).fit()
model.summary2()
```

Model:	OLS	Adj. R-squared:	0.616
Dependent Variable:	Temp	AIC:	17093.1448
Date:	2020-02-08 22:19	BIC:	17117.9514
No. Observations:	3647	Log-Likelihood:	-8542.6
Df Model:	3	F-statistic:	1953.
Df Residuals:	3643	Prob (F-statistic):	0.00
R-squared:	0.617	Scale:	6.3467

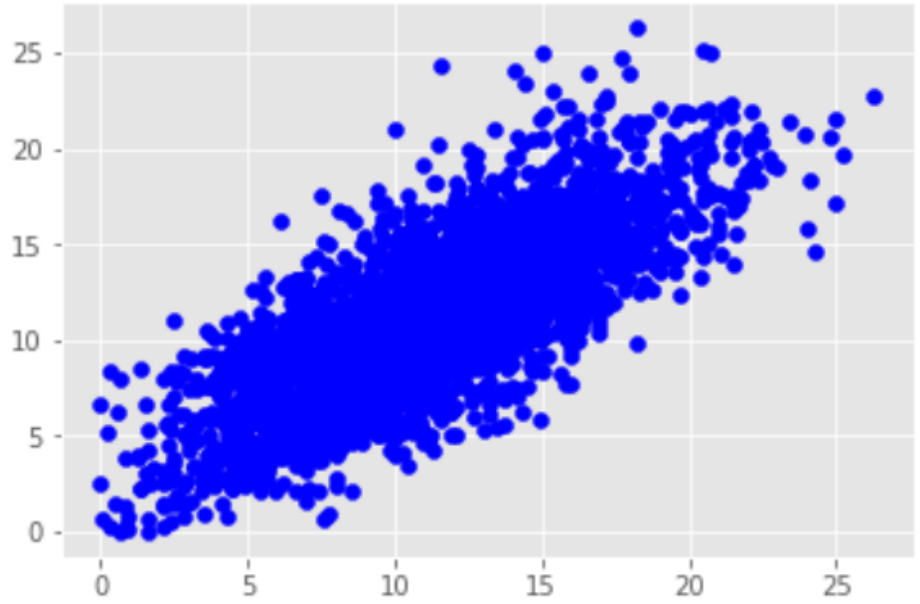
	Coef.	Std.Err.	t	P>|t|	[0.025	0.975]
Intercept	1.8882	0.1340	14.0938	0.0000	1.6255	2.1508
temp_1	0.7000	0.0163	43.0404	0.0000	0.6681	0.7319
temp_2	-0.0594	0.0200	-2.9766	0.0029	-0.0985	-0.0203
temp_3	0.1902	0.0163	11.6995	0.0000	0.1583	0.2221

Omnibus:	8.194	Durbin-Watson:	2.056
Prob(Omnibus):	0.017	Jarque-Bera (JB):	9.630
Skew:	0.032	Prob(JB):	0.008
Kurtosis:	3.243	Condition No.:	66

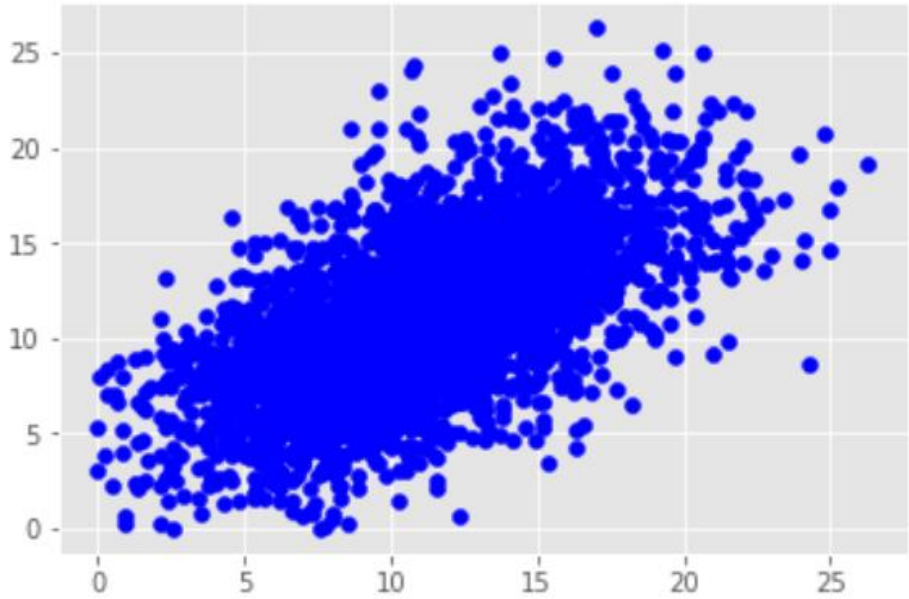
The last three time series values significantly predict the future values.

AR Model: Scatter Plot

```
# check autocorrelation
plt.scatter(temp_AR.Temp, temp_AR.temp_1, c="b");
```



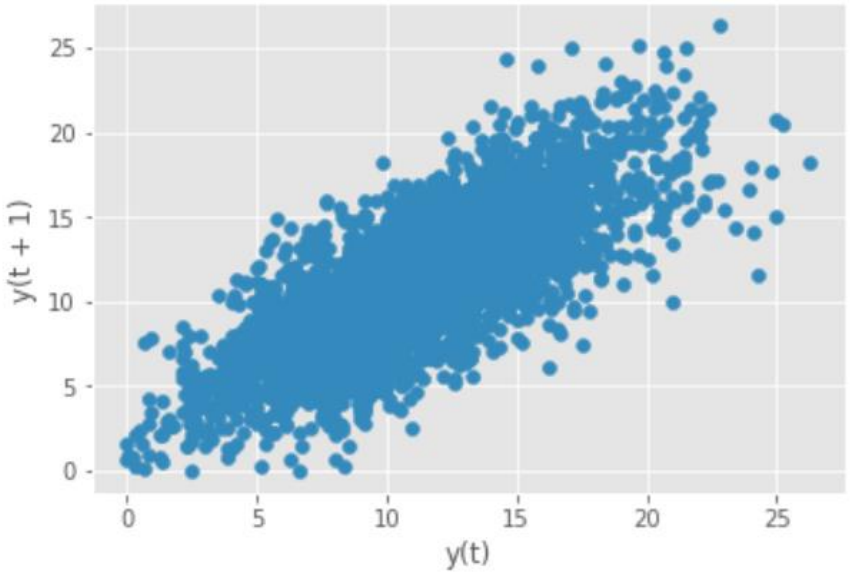

```
plt.scatter(temp_AR.Temp, temp_AR.temp_2, c="b");
```



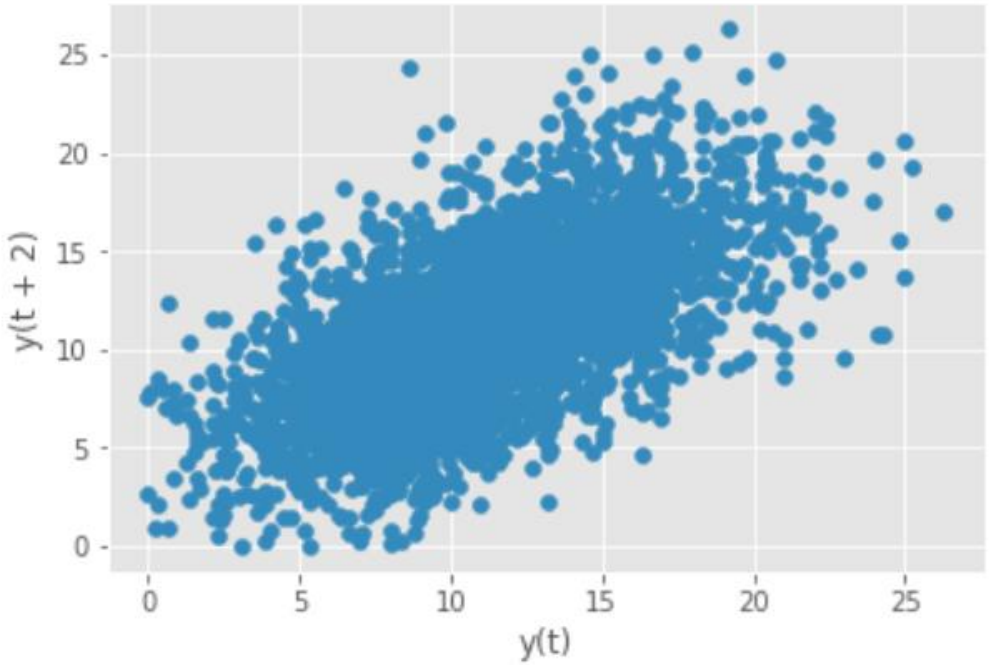
Check for autocorrelation using scatter plots of data vs lags

AR Model: Scatter Plot

```
# visualize correlation between data and lag variables
# using pandas plotting features
pd.plotting.lag_plot(temp_AR.Temp, lag=1);
```




```
pd.plotting.lag_plot(temp_AR.Temp, lag=2);
```



Check for autocorrelation between data vs lags using pandas plotting features

AR Model: Correlation

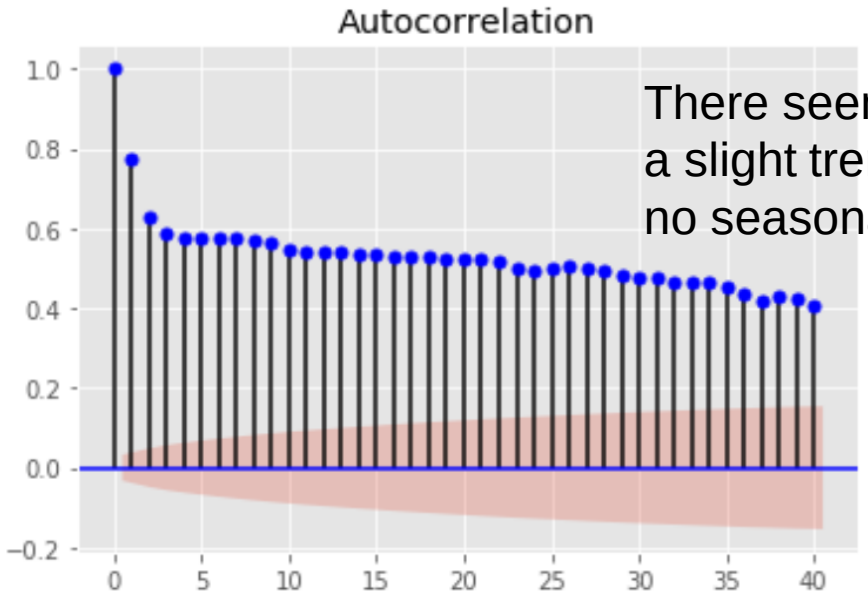
- Use numerical method to check for autocorrelation between data and lag variables.
- This can be tedious, so a better way is to use the autocorrelation plot

```
# check correlation among time series and lags
temp_AR.corr()
```

	Temp	temp_1	temp_2	temp_3
Temp	1.000000	0.774870	0.631119	0.586375
temp_1	0.774870	1.000000	0.774886	0.631095
temp_2	0.631119	0.774886	1.000000	0.774878
temp_3	0.586375	0.631095	0.774878	1.000000

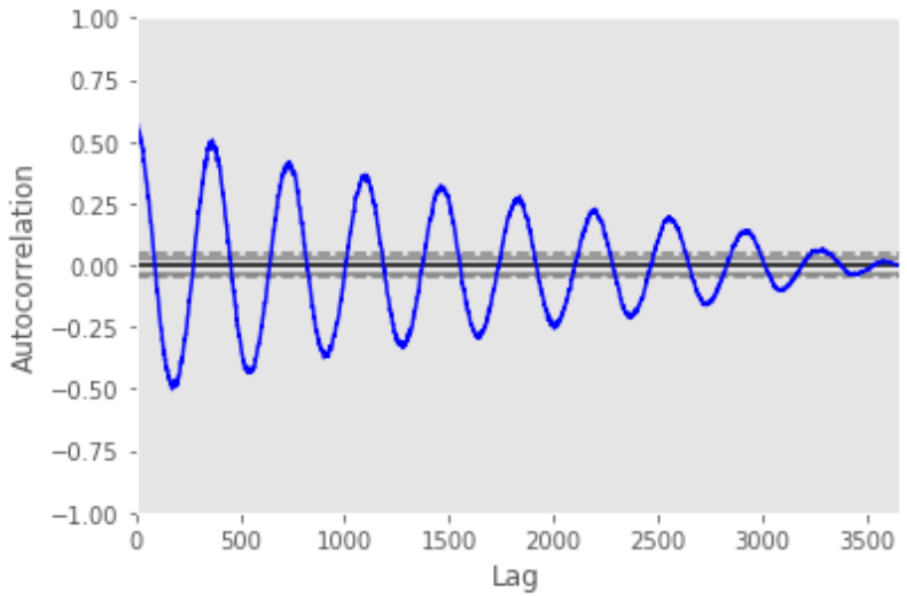
AR Model: AFC

```
# plot an autocorrelation function using statsmodels
from statsmodels.graphics.tsaplots import plot_acf

plot_acf(temp_AR.Temp, lags=40, c="b");
```



There seems to be a slight trend and no seasonality.

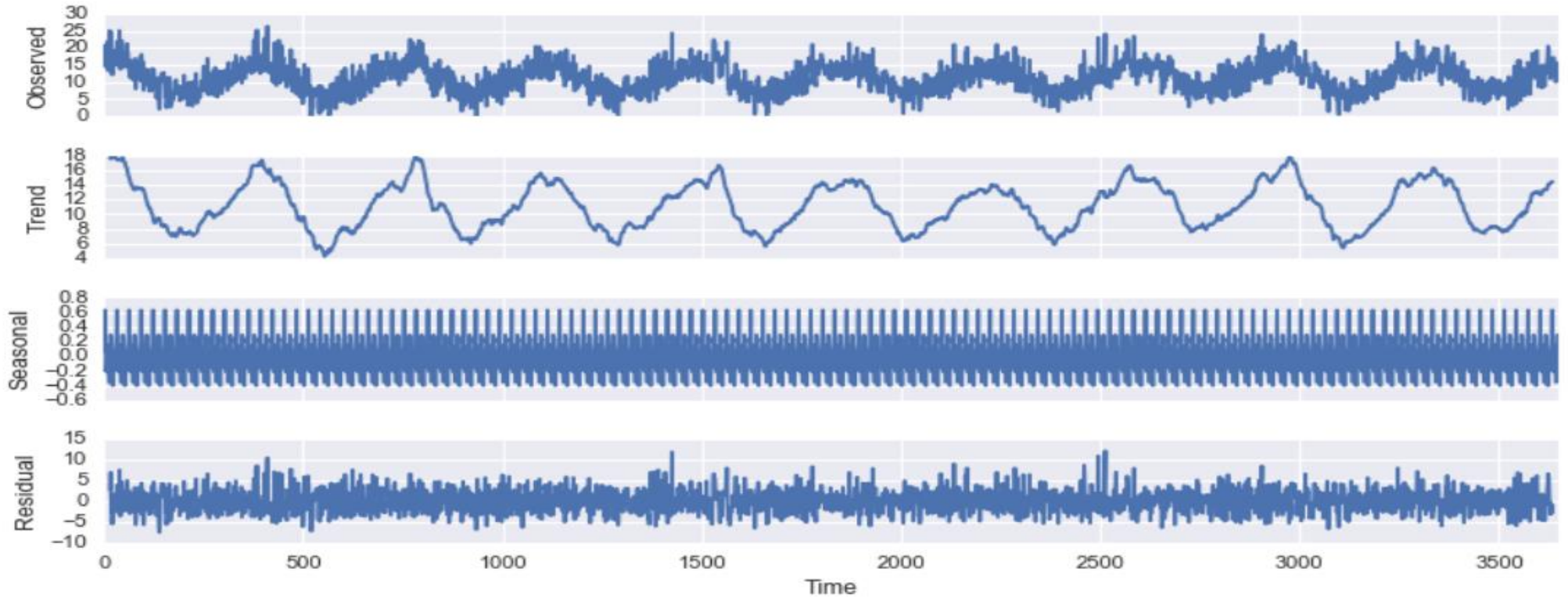
```
# plot an autocorrelation function using pandas
from pandas.plotting import autocorrelation_plot

autocorrelation_plot(temp_AR.Temp, c="b");
```



pandas and statsmodels have different versions of the autocorrelation plots

AR Model: AFC

```
# use statsmodels.tsa.seasonal.seasonal_decompose
# to decompose visual plots
plt.rcParams['figure.figsize']=(10,5)
decomp = seasonal_decompose(temp_AR["Temp"].values,
                            model='additive', freq=30)
decomp.plot();
```



AR Model: Stationarity Test

- After checking the trend and seasonality through the autocorrelation function, a statistical test can be conducted to test for stationary.

```
# perform the Dickey-Fuller test to statistically test for stationarity
from statsmodels.tsa.stattools import adfuller

test = adfuller(temp_AR.Temp, autolag="AIC")
pd.DataFrame(test[0:4], index=["teststat", "p-value",
                               "#lags used", "#observations used"])
```

	0
teststat	-4.444805
p-value	0.000247
#lags used	20.000000
#observations used	3629.000000

AR Model: Differencing

- If there was a serious trend in the data, we could remove or reduce the trend through differencing to make the data stationary before we fit the model.
- Differencing involves transforming the data by finding the difference between consecutive observations.

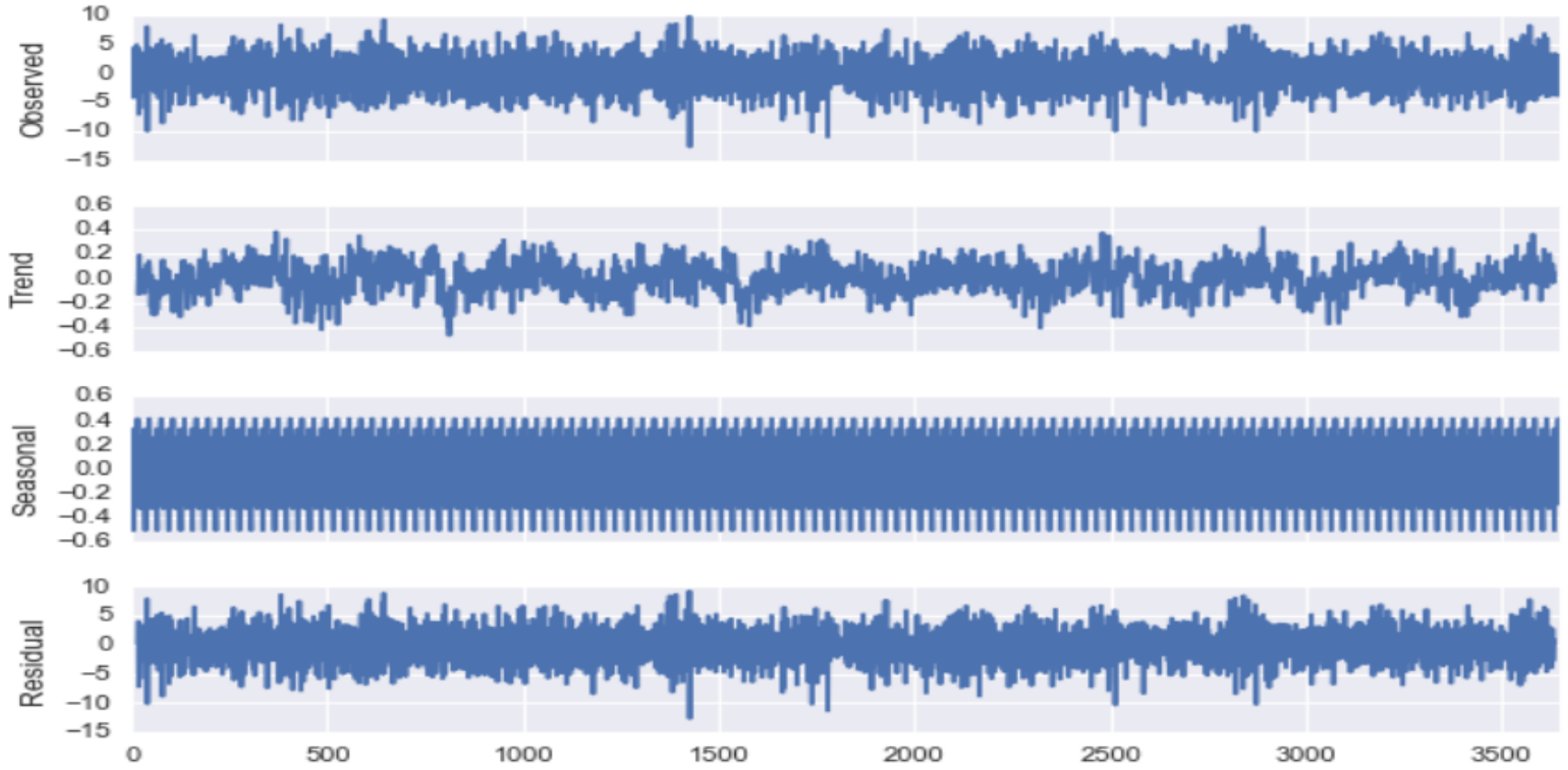
```
# use pandas diff() method to find the
# difference between observations
temp_AR["stationary"] = temp_AR["Temp"].diff()
temp_AR[["Temp", "stationary"]].head()
```

	Temp	stationary
0	20.7	NaN
1	17.9	-2.8
2	18.8	0.9
3	14.6	-4.2
4	15.8	1.2

The differencing we do here is for illustrative purpose, you should actually differencing if the stationary test fails. Then use the transformed results to train your model.

AR Model: Decomposed ACF

```
# check stationarity again with the transformed data
decomp = seasonal_decompose(temp_AR["stationary"].dropna(),
                            model='additive', freq=30)
decomp.plot();
```



Visualize the ACF of the differenced data. How does this compare to the ACF of the original data?

AR Model: Model Fitting

```
# split data into train and test set
X = temp_AR['Temp'].dropna()
X = X.values # extract only values without indexes
n = len(X)
n
```

```
3650
```

```
# prediction will be made for the last 7 days
train_data = X[1:n-7]
test_data = X[n-7:]
```

```
from statsmodels.tsa.ar_model import AR

#train the autoregression model
model = AR(train_data).fit()
```

```
# lag value selected
model.k_ar
```

```
29
```

```
# model parameters
model.params
```

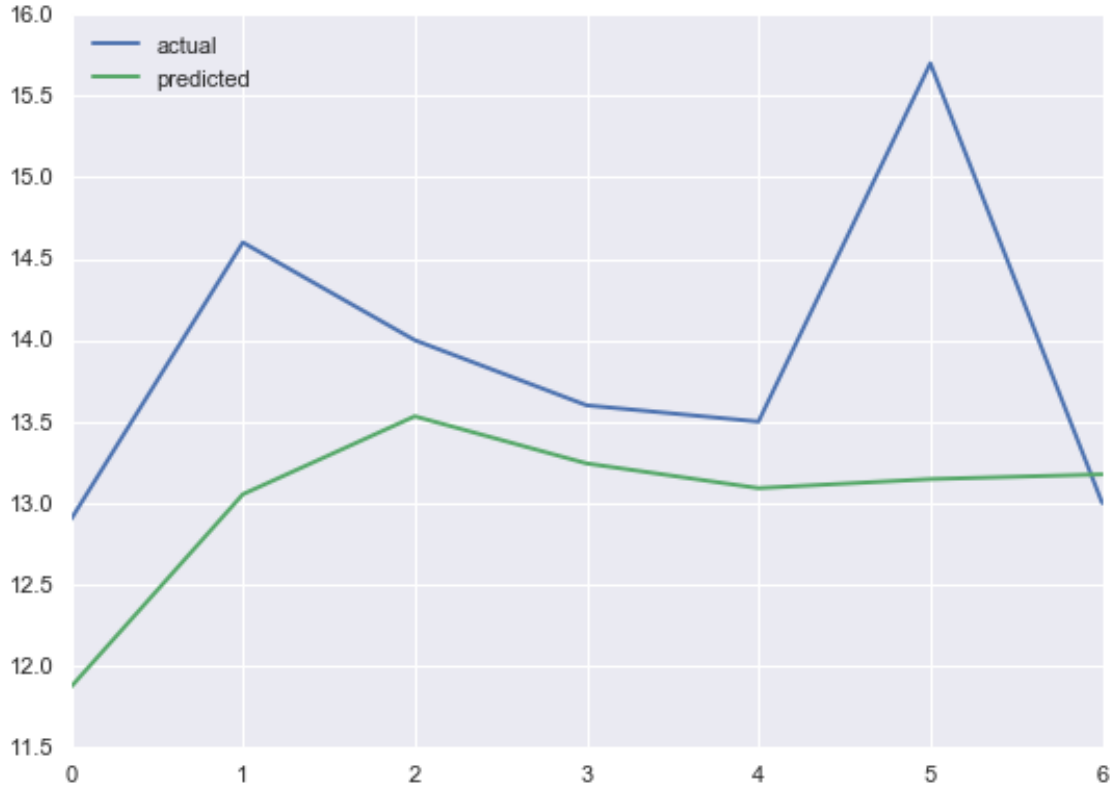
```
array([ 5.57543506e-01,  5.88595221e-01, -9.08257090e-02,  4.82615092e-02,
        4.00650265e-02,  3.93020055e-02,  2.59463738e-02,  4.46675960e-02,
        1.27681498e-02,  3.74362239e-02, -8.11700276e-04,  4.79081949e-03,
        1.84731397e-02,  2.68908418e-02,  5.75906178e-04,  2.48096415e-02,
        7.40316579e-03,  9.91622149e-03,  3.41599123e-02, -9.11961877e-03,
        2.42127561e-02,  1.87870751e-02,  1.21841870e-02, -1.85534575e-02,
       -1.77162867e-03,  1.67319894e-02,  1.97615668e-02,  9.83245087e-03,
        6.22710723e-03, -1.37732255e-03])
```

AR Model: Assessment

```
# make predictions

predictions = model.predict(start=len(train_data),
                            end=len(train_data) + len(test_data)-1,
                            dynamic=False)

test_pred = pd.concat([pd.DataFrame(test_data),
                       pd.DataFrame(predictions)], axis="columns")

test_pred.columns = ["actual", "predicted"]
test_pred.plot();
```

```
# overall prediction error
from sklearn.metrics import mean_squared_error
mean_squared_error(test_data, predictions)
```

```
1.5015252310069296
```



Residual Diagnostics

- The model residuals should satisfy the following:
  - The residuals should have no autocorrelation (should be uncorrelated or have autocorrelation close to zero).
  - Residuals should be normally distributed, with a mean of zero and constant variance.
- Residual ACF can be used to check autocorrelation, zero mean, and constant variance.
- A histogram can also be used to check for normality.

Residual Diagnosis Using ACF

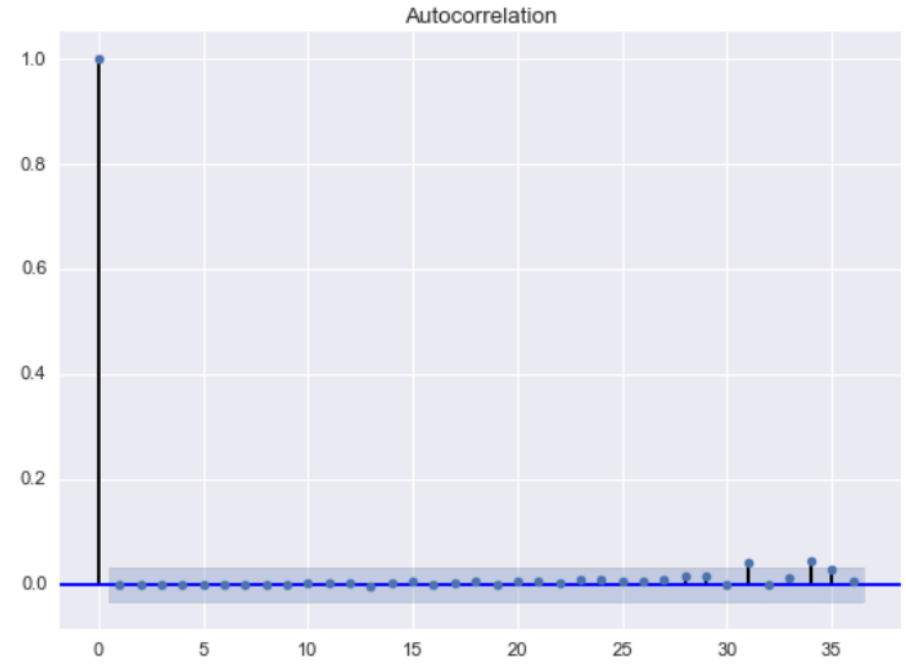
- The plot indicates that the error are fairly within the 95% confidence interval, fairly constant with a mean of zero.
- Residuals are the leftovers after fitting the model. Correlation in the residuals indicates the model did not adequately capture the information in the model.

(Hyndman & Athanasopoulos, 2020)

```
# check if residuals are normally distributed
residuals = model.resid
residuals
```

```
array([-0.68105821, -0.69572786,  2.74223153, ..., -0.0726287 ,
        0.15670338, -4.17767517])
```

```
plot_acf(residuals);
```

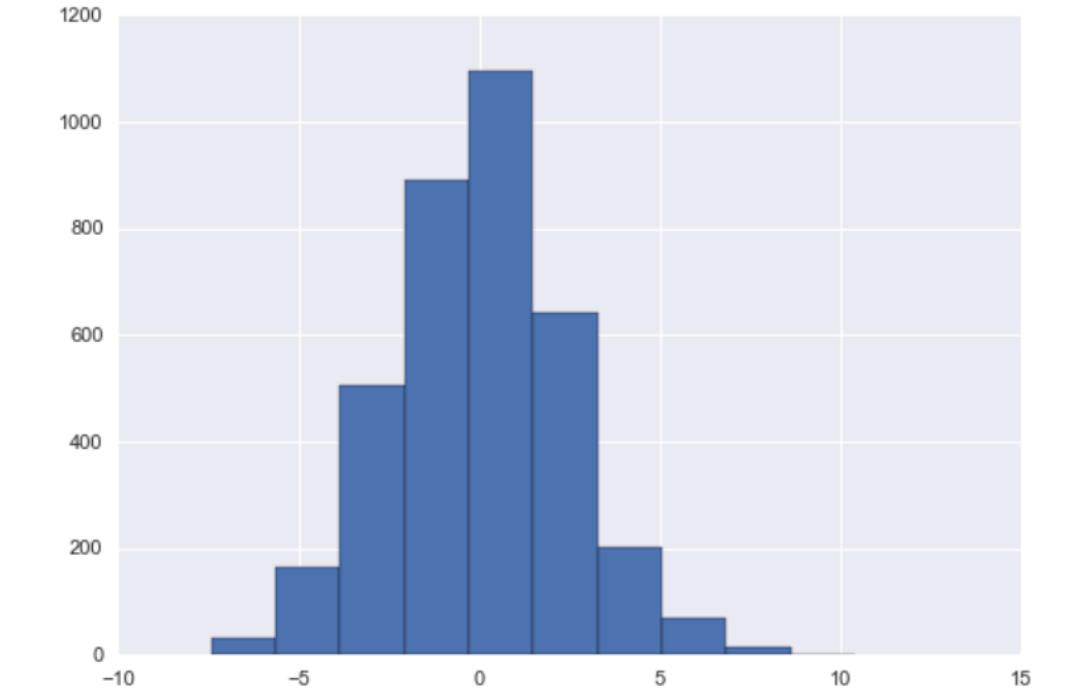


Residual Diagnostics Using a Histogram

- The histogram indicates that the residuals are approximately normally distributed with a mean around zero.
- Mean of residual errors need to be zero else predictions will be systematically biased.

(Hyndman & Athanasopoulos, 2020)

```
plt.hist(residuals);
```



Residual Diagnostics

- The variance also need to be constant so that prediction intervals can be easily produced.
- All these assumptions are those of a linear regression and apply to time series statistical models.
- Again, OLS regression assumptions are that:
  - the residuals need to be independent (no autocorrelation)
  - normally distributed,
  - with a mean of zero and
  - with a constant variance.

Moving Average (MA) Model

- A moving average (MA) model is a model where the q most recent error terms are used to predict the future time series values.
- The forecast at time t is a linear function of error terms in the present and past.
- The error terms are assumed to be white noise process, independent from each other, with a mean of zero and a constant variance σ^2.
- Note that moving average smoothing is not the same as moving average model.

MA Model

- The MA model can be expressed as a function of the q most recent errors:
- $y_t = f(\varepsilon_t, \varepsilon_{t-1}, \varepsilon_{t-2}, \dots, \varepsilon_{t-q})$
- The parameter of MA(q) is q which is the number of past error terms. It is the MA window size also called the order of the MA.

- The MA model can be represented as:

$y_t = \mu + \varepsilon_t + \emptyset_1 * \varepsilon_{t-1} + \emptyset_2 * \varepsilon_{t-2} + \dots + \emptyset_q * \varepsilon_{t-q}$

- The error terms are iid, with mean of zero and constant variance σ^2.

$E(y_t) = \mu$

ARMA Model

- Autoregressive Moving Average (ARMA) Model is a combination of AR and MA model.
- That means, future values are both a function of p past values and the q most recent error term values.
- The stationary of ARMA is comes down to the same way as the stationarity of AR.
- ARMA(p, q) model has parameters p, and q referring to p autoregressive terms and q moving average terms.

ARMA Model

- The ARMA model can be written as:
- $y_t = \beta_0 + \beta_1 * y_{t-1} + \beta_2 * y_{t-2} + \ldots + \beta_p * y_{t-p} + \epsilon_t$

$$\emptyset_1 * \varepsilon_{t-1} + \emptyset_2 * \varepsilon_{t-2} + \ldots + \emptyset_q * \varepsilon_{t-q}$$

P is the number of past observation and q is number of past error terms (MA window size) included in the model.

$$y_t = \beta_0 + \varepsilon_t + \sum_{i=1}^{p} \beta_0 * y_{t-i} + \sum_{i=1}^{q} \emptyset_q * \varepsilon_{t-i}$$

https://en.wikipedia.org/wiki/Autoregressive%E2%80%93moving-average_model

Autoregressive Integrated Moving Average (ARIMA) Model

- An Autoregressive Integrated Moving Average (ARIMA) model is similar to ARMA but has an integrated part which involves the number of times a time series must be differenced to produce stationarity.
- ARIMA(p, d, q) has parameters p, d and q. p and q are as previously defined, d is the order of differencing.

ARIMA Model

- The model is mathematically the same as that for ARMA:

$$y_t = \beta_0 + \beta_1 * y_{t-1} + \beta_2 * y_{t-2} + \ldots + \beta_p * y_{t-p} + \epsilon_t$$

$$\emptyset_1 * \varepsilon_{t-1} + \emptyset_2 * \varepsilon_{t-2} + \ldots + \emptyset_q * \varepsilon_{t-q}$$

The difference is that, the data needs to be first differenced for the ARIMA model to make the time series stationary.

This model is used for non-stationary time series.

ARIMA Model

- ARIMA filters the noise from the signal, and the signal is then used for forecasting.
- The following models are special cases of ARIMA
  - Random walk model
  - Autoregressive model
  - Exponential smoothing

https://people.duke.edu/~rnau/411arim.htm

Special Cases of ARIMA Model

- ARIMA(1,0,0) = First-order autoregressive model
  - $\widehat{y_t} = \beta_0 + \beta_1 * y_{t-1}$
- ARIMA(0,1,0) = random walk with a drift.
  - $\hat{Y}_t = \beta_0 + Y_{t-1}$
- ARIMA(0,1,0) = a simple random walk: $\hat{Y}_t = Y_{t-1}$
- ARIMA(1, 1, 0) = differenced first autoregressive model
  - $\widehat{y_t} = \beta_0 + y_{t-1} + \beta_1(y_{t-1} - y_{t-2})$
- ARIMA(0,1,1) = Simple exponential smoothing
  - $\widehat{y_t} = \hat{y}_{t-1} + \varepsilon_{t-1}$
  - This is the “error” correction form of exponential smoothing

Implement the ARMA Model

- We can use the ARIMA() model to implement the MA model by setting the AR order to 0 as well as setting the differencing order to 0. The MA order must be specified in the model.
- You can train on all the data and use it to predict the next period as shown.

```
# use all data for training
# make prediction for the next period
mod = ARMA(X, order=(0,1)).fit(disp=False)
mod.predict(start=len(X), end=len(X), dynamic=False)
```

```
array([10.91151172])
```

Implementing MA Model in Code

```
# split data into train and test set
X = temp_AR['Temp'].dropna()
X = X.values # extract only values without indexes
n = len(X)

# prediction will be made for the last 7 days
train_data = X[1:n-7]
test_data = X[n-7:]
```

```
# fit model
model_MA = ARMA(train_data, order=(0,1)).fit(disp=False)

# make predictions
predict_MA = model_MA.predict(start=len(train_data),
                              end=len(train_data) + len(test_data)-1,
                              dynamic=False)
```

Note that ARMA(0, 1) is equivalent to ARIMA(0, 0, 1)

Evaluate the MA Model

```
# MSE
mean_squared_error(test_data, predict_MA)
```

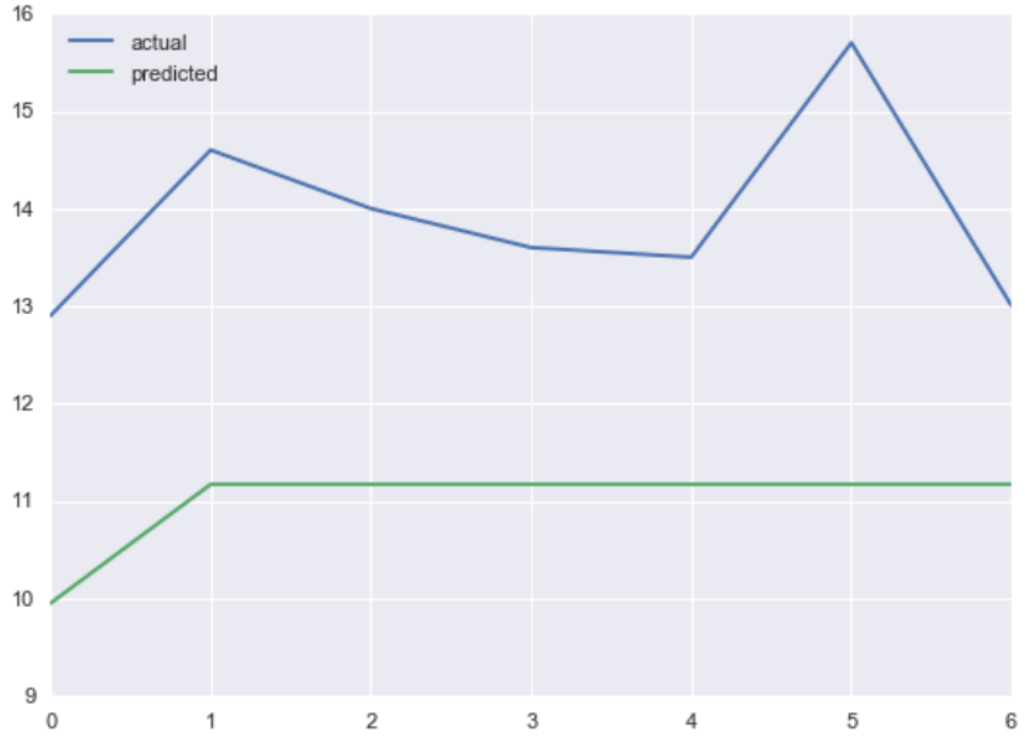
```
9.096985026186768
```

```
# AIC
model_MA.aic
```

```
18511.488006977022
```

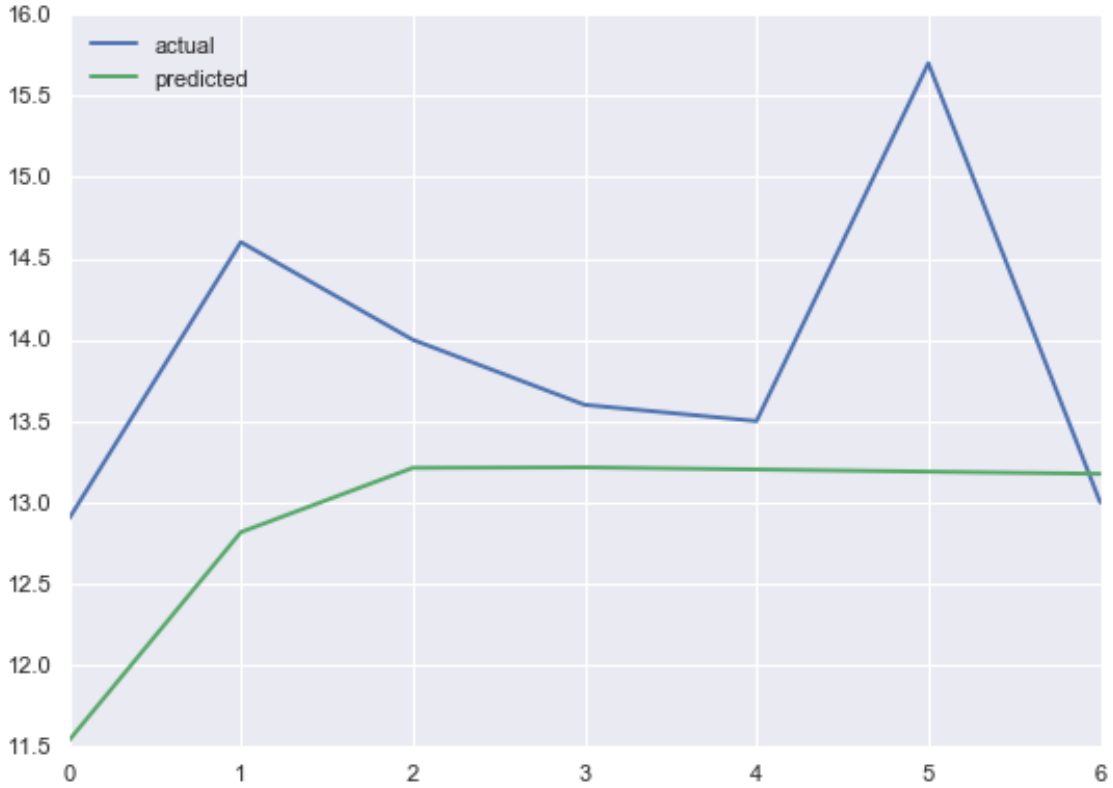
MSE or AIC can be used for model comparison and selection

```
test_pred_MA = pd.concat([pd.DataFrame(test_data),
                          pd.DataFrame(predict_MA)], axis="columns")

test_pred_MA.columns = ["actual", "predicted"]
test_pred_MA.plot();
```



Implement ARMA Model

```
# fit ARMA model
model_ARMA = ARMA(train_data, order=(2,3)).fit(disp=False)

# make predictions
predict_ARMA = model_ARMA.predict(start=len(train_data),
                                  end=len(train_data) + len(test_data)-1,
                                  dynamic=False)

test_pred_ARMA = pd.concat([pd.DataFrame(test_data),
                            pd.DataFrame(predict_ARMA)], axis="columns")

test_pred_ARMA.columns = ["actual", "predicted"]
test_pred_ARMA.plot();
```




```
# MSE
mean_squared_error(test_data, predict_ARMA)
```

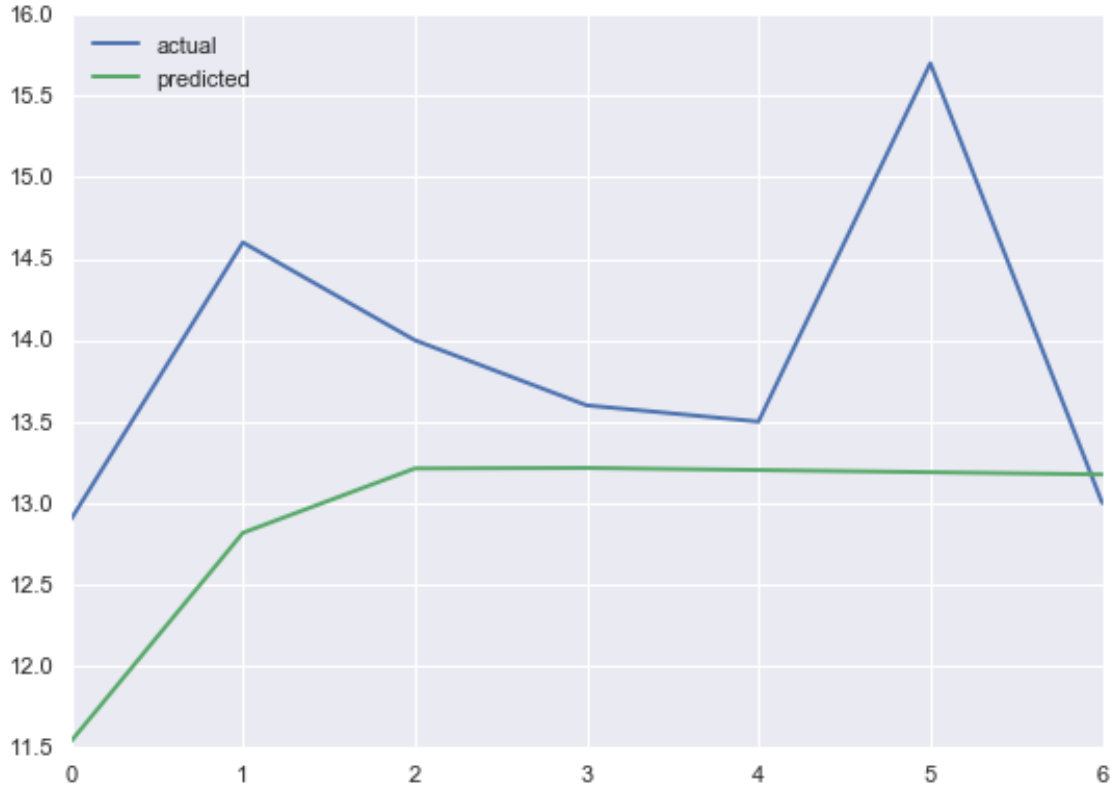
```
1.7470674403217132
```

```
# AIC
model_ARMA.aic
```

```
16748.03006032794
```

Implement ARIMA Models

```
# fit ARIMA model
model_ARIMA = ARIMA(train_data, order=(2, 0, 3)).fit(disp=False)

# make predictions
predict_ARIMA = model_ARIMA.predict(start=len(train_data),
                                    end=len(train_data) + len(test_data)-1,
                                    dynamic=False)

test_pred_ARIMA = pd.concat([pd.DataFrame(test_data),
                             pd.DataFrame(predict_ARIMA)], axis="columns")

test_pred_ARIMA.columns = ["actual", "predicted"]
test_pred_ARIMA.plot();
```




```
# MSE
mean_squared_error(test_data, predict_ARMA)
```

```
1.7470674403217132
```

```
# AIC
model_ARIMA.aic
```

```
16748.03006032794
```

ARIMA(2, 0, 3) = ARMA(2, 3)

Implement an ARIMA Model

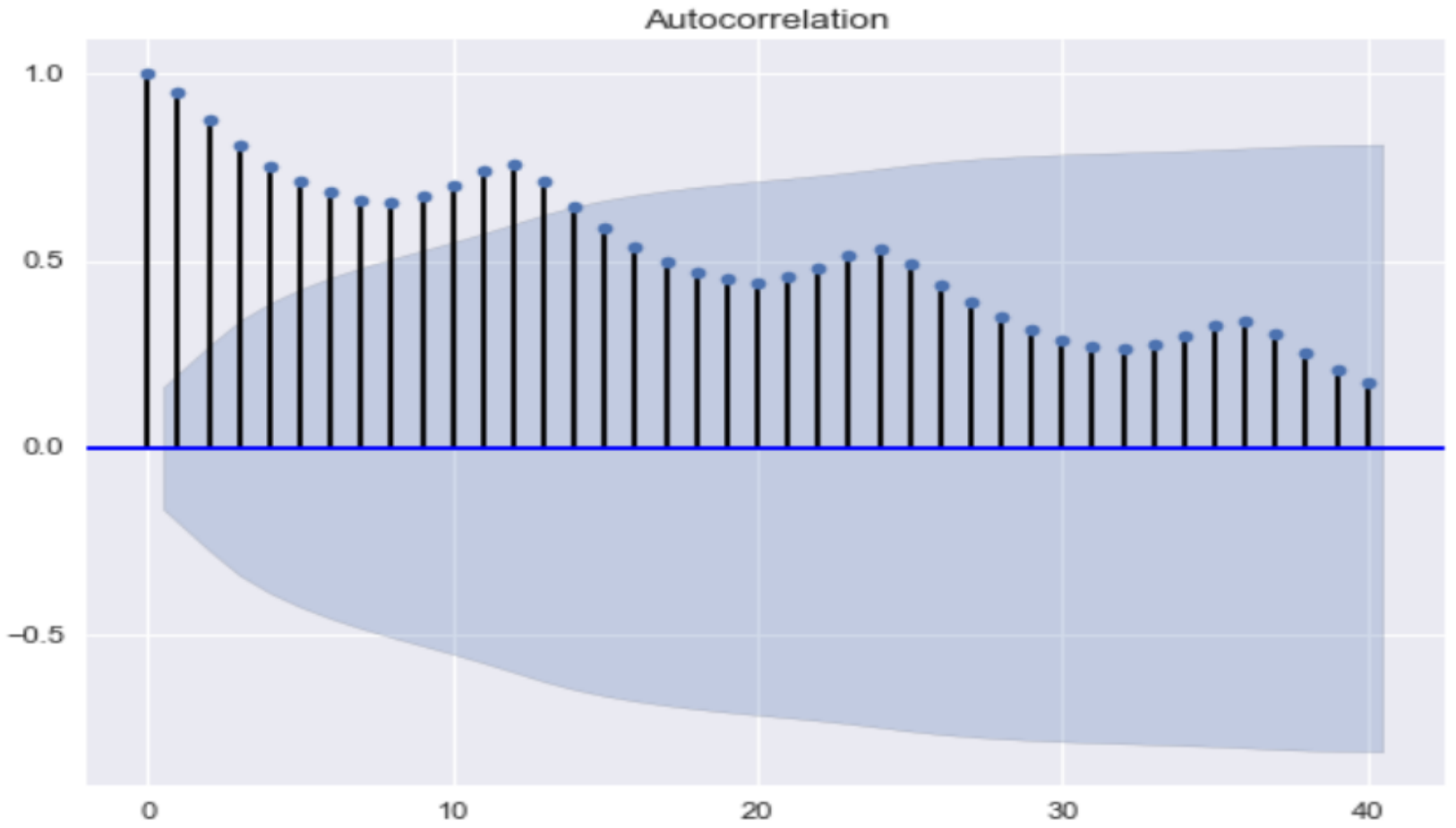
- ARIMA models are more suitable for non-stationary data.
- So, let’s apply an ARIMA model on non stationary data.
- Let’s use the Air Passenger data.

```
air_pass.head()
```

time	AirPassengers
1949.000000	112
1949.083333	118
1949.166667	132
1949.250000	129
1949.333333	121

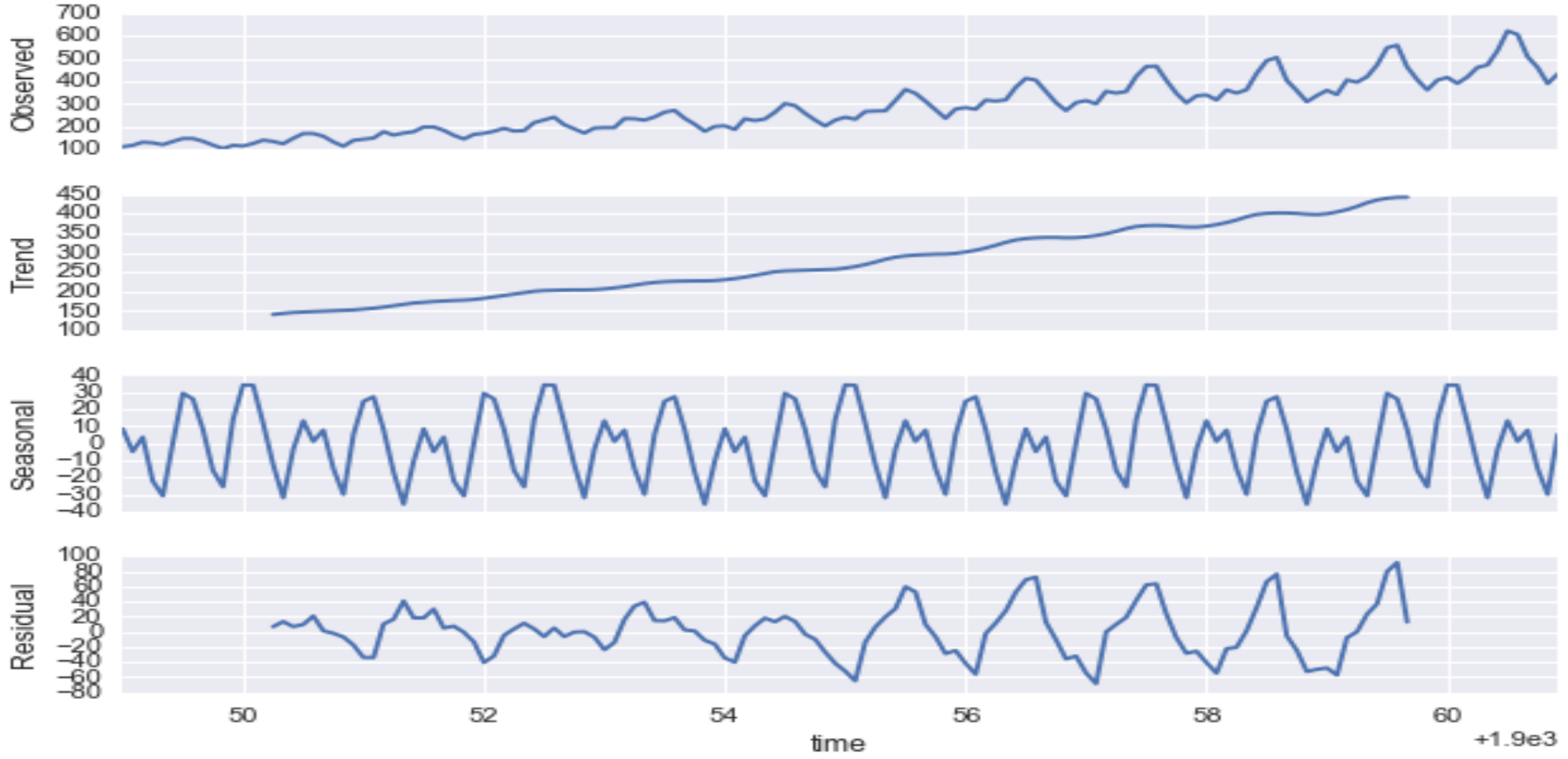
ARIMA Model: ACF

```
# Autocorrelation Function
plot_acf(air_pass.AirPassengers, lags=40);
```



This ACF indicates seasonality because of the periodic pattern. There is a trend because of the gradual decrease in ACF as lags increase.

ARIMA Model: Seasonal Decomposition



ARIMA Model Fitting

```
# split data into train and test set
X_air = air_pass['AirPassengers'].dropna()
X_air = X_air.values # extract only values without indexes
n = len(X_air)

# prediction will be made for the last 11 days
# since a new season seems to begins every 12th day
train = X_air[1:n-11]
test = X_air[n-11:]
```

```
# fit ARIMA model
model_ARIMA2 = ARIMA(train, order=(1, 1, 2)).fit(disp=False)

# make predictions
predict_ARIMA2 = model_ARIMA2.predict(start=len(train),
                                      end=len(train) + len(test)-1,
                                      dynamic=False, typ="levels")
```

ARIMA Model Fitting Results

```
model_ARIMA2.summary()
```

ARIMA Model Results

Dep. Variable:	D.y	**No. Observations:**	131
Model:	ARIMA(1, 1, 2)	**Log Likelihood**	-615.366
Method:	css-mle	**S.D. of innovations**	26.160
Date:	Sun, 09 Feb 2020	**AIC**	1240.732
Time:	21:22:47	**BIC**	1255.108
Sample:	1	**HQIC**	1246.574

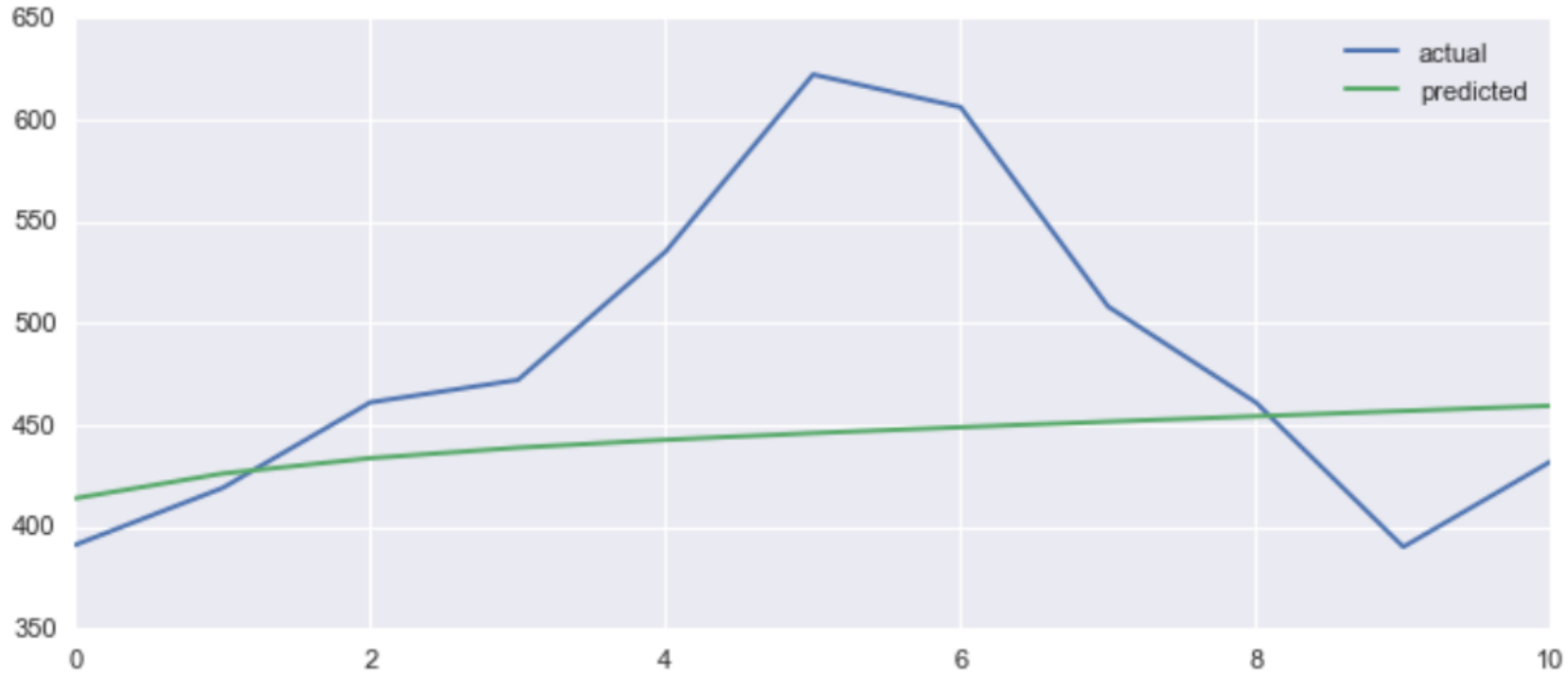
	coef	std err	z	P>|z|	[0.025	0.975]
const	2.5425	0.188	13.519	0.000	2.174	2.911
ar.L1.D.y	0.5205	0.103	5.039	0.000	0.318	0.723
ma.L1.D.y	-0.4416	0.127	-3.479	0.001	-0.690	-0.193
ma.L2.D.y	-0.5583	0.126	-4.442	0.000	-0.805	-0.312

Roots

	Real	Imaginary	Modulus	Frequency
AR.1	1.9211	+0.0000j	1.9211	0.0000
MA.1	1.0000	+0.0000j	1.0000	0.0000
MA.2	-1.7911	+0.0000j	1.7911	0.5000

ARIMA Model Evaluation

```
test_pred_ARIMA2 = pd.concat([pd.DataFrame(test),
                              pd.DataFrame(predict_ARIMA2)], axis="columns")

test_pred_ARIMA2.columns = ["actual", "predicted"]
test_pred_ARIMA2.plot(figsize=(10, 4));
```



Model Evaluation

```
# MSE
mean_squared_error(test, predict_ARIMA2)
```

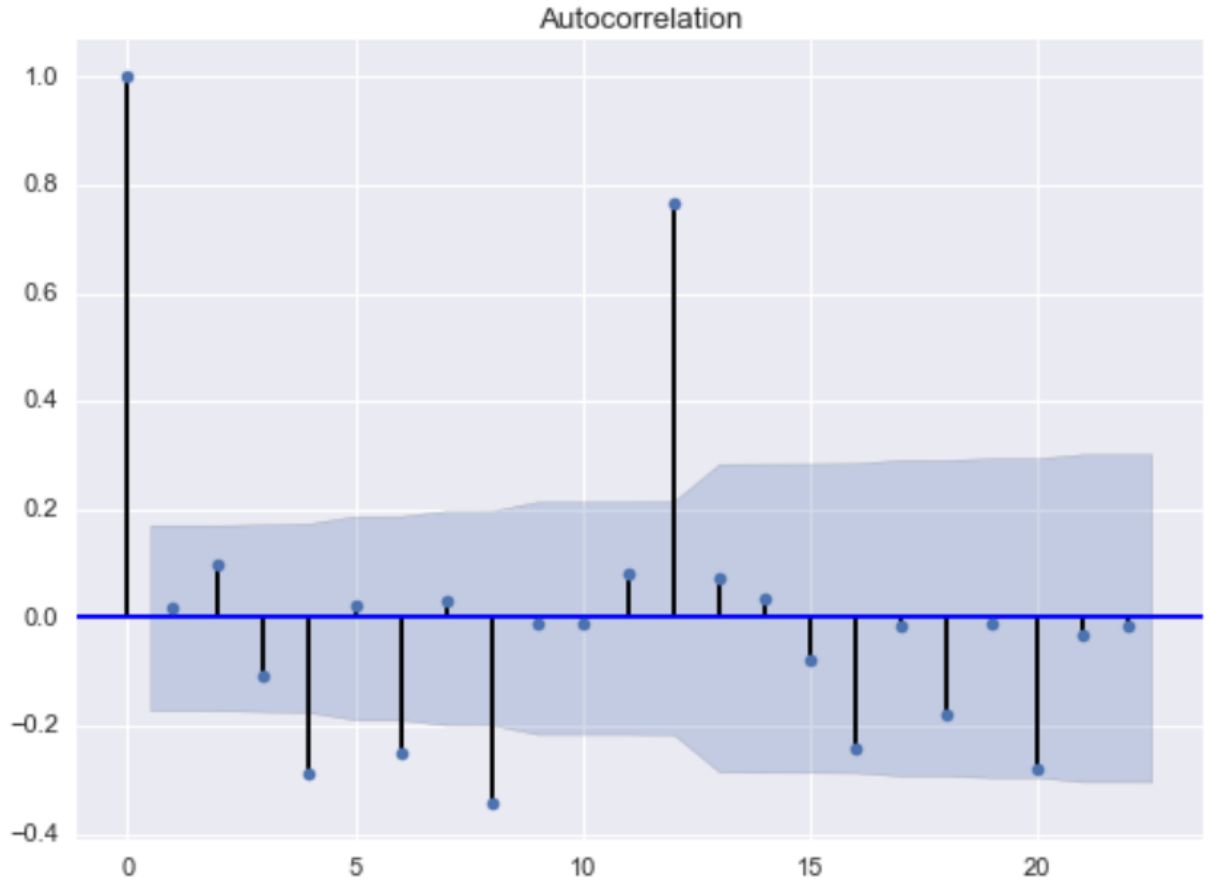
```
6830.152050985579
```

```
# AIC
model_ARIMA2.aic
```

```
1240.732331226167
```

Model Diagnostics: Residual ACF

```
# residual ACF
plot_acf(model_ARIMA2.resid);
```

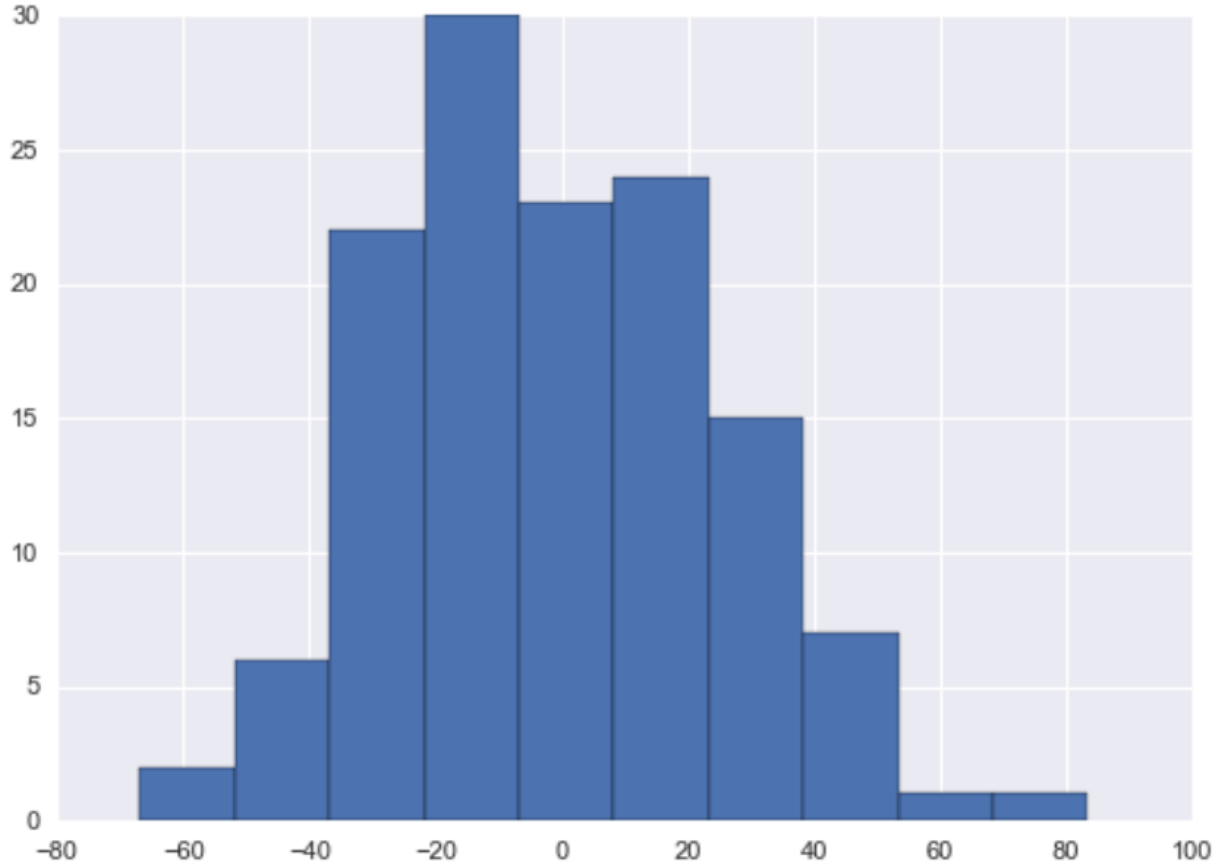


Residual ACF indicates some errors are out of the 95% confidence. These errors out of the 95% are correlated to some other errors. This suggest for model improvement.

The errors also seem not to have a constant variance.

Model Diagnostics: Residual Histogram

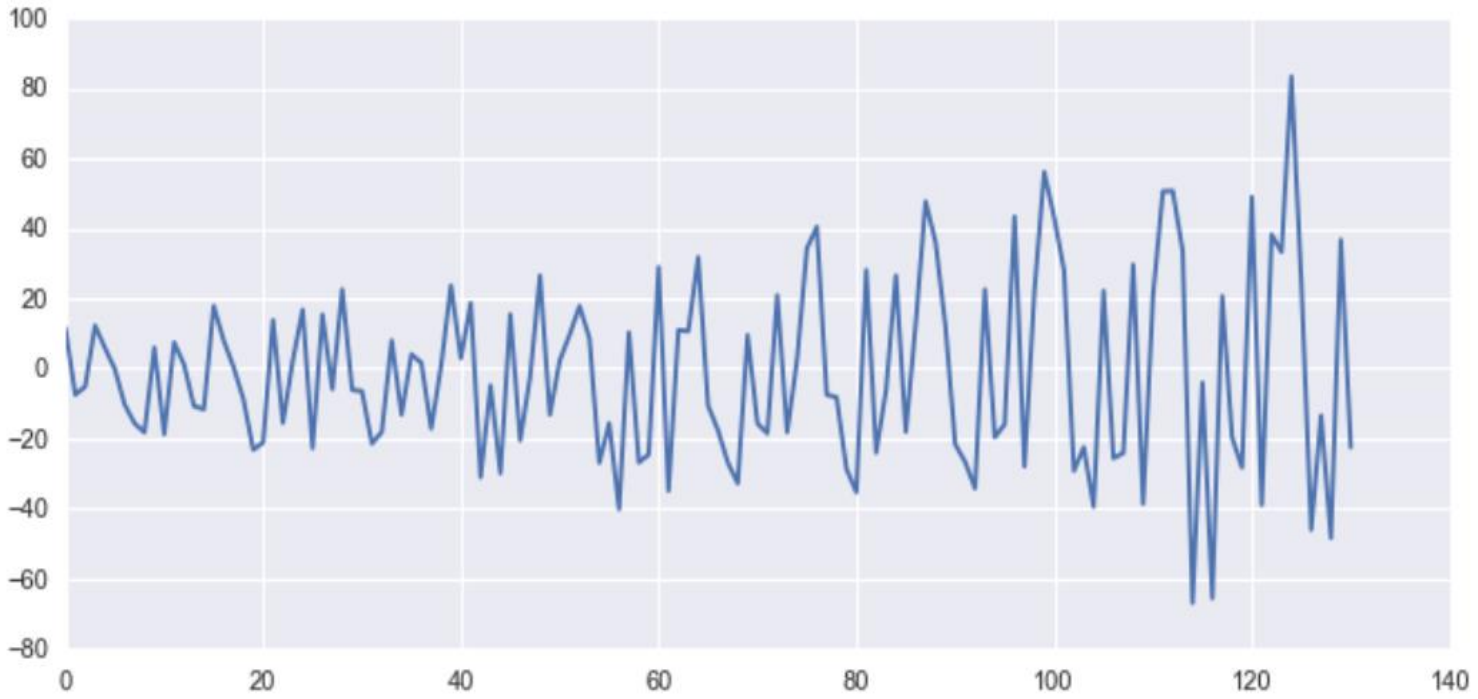
```
plt.hist(model_ARIMA2.resid);
```



Errors seem not follow a normal distribution though mean is around 0

Model Diagnostics: A Line Plot of Residuals

```
# line plot of residuals
plt.figure(figsize=(10, 4))
plt.plot(model_ARIMA2.resid);
```



A line plot of residuals indicates that the error variance increases with time.

ARIMA Model with Log Transformation

- The MSE seems to be very high, and seems to indicate poor predication accuracy.
- Inspection of residual/ACF plots also indicates linear regression assumptions are violated. There now non-normality, non constant variance and autocorrelation.
- Let’s apply a log transformation to the data before fitting it and compare the new model with the non-transformed Model.

ARIMA Model with Log Transformation

```
# split data into train and test set
X_air= air_pass['AirPassengers'].dropna()
X_air =X_air.values # extract only values without indexes
n = len(X_air)

# prediction will be made for the last 11 days
# since a new season seems to begins every 12th day
train_log = np.log(X_air[1:n-11])
test_log = np.log(X_air[n-11:])
```

```
# fit ARIMA model
model_ARIMA3 = ARIMA(train, order=(1, 1, 2)).fit(disp=False)

# make predictions
predict_ARIMA3 = model_ARIMA3.predict(start=len(train_log),
                                      end=len(train_log) + len(test_log)-1,
                                      dynamic=False, typ="levels")
# specify typ="levels" to keep predictions to same scale as original
# data in a case where differencing was performed.
```

ARIMA Model with Log Transformation

```
model_ARIMA3.summary()
```

ARIMA Model Results

Dep. Variable:	D.y	**No. Observations:**	131
Model:	ARIMA(1, 1, 2)	**Log Likelihood**	-615.366
Method:	css-mle	**S.D. of innovations**	26.160
Date:	Sun, 09 Feb 2020	**AIC**	1240.732
Time:	21:23:02	**BIC**	1255.108
Sample:	1	**HQIC**	1246.574

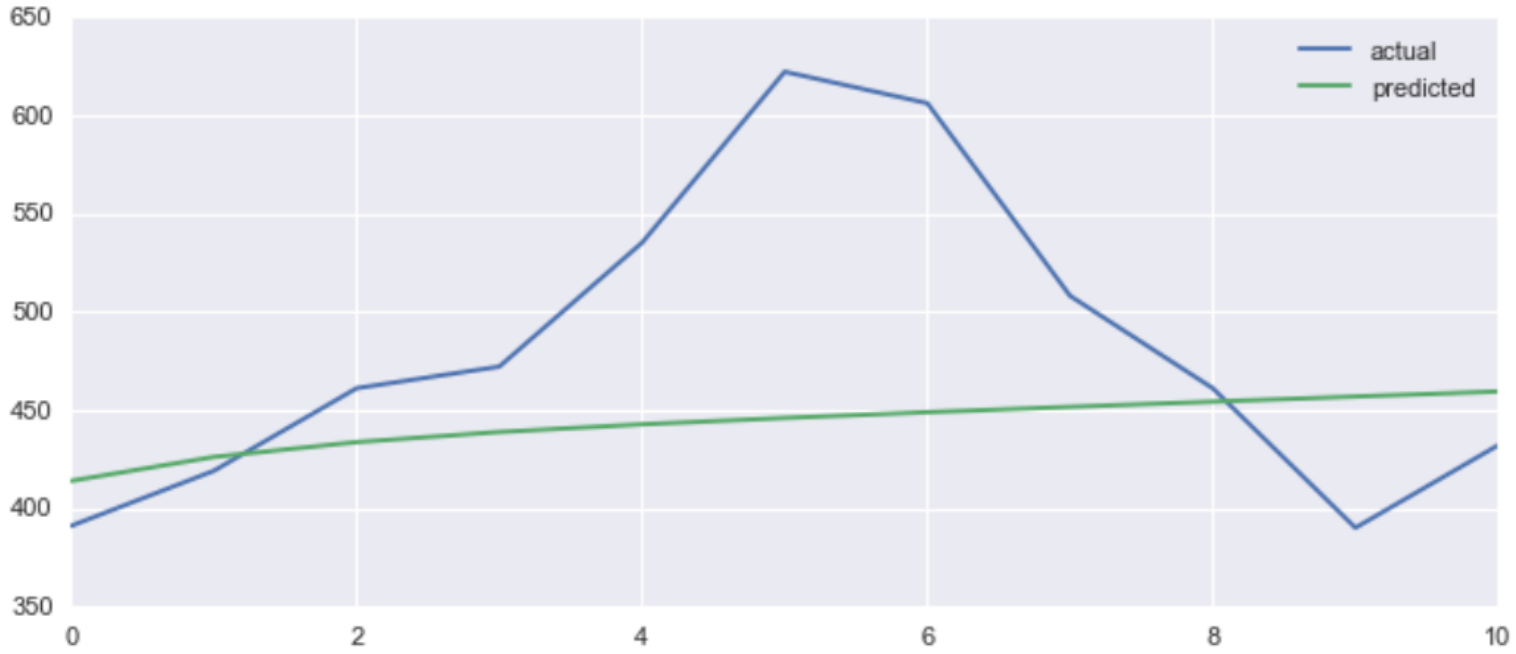
	coef	std err	z	P>|z|	[0.025	0.975]
const	2.5425	0.188	13.519	0.000	2.174	2.911
ar.L1.D.y	0.5205	0.103	5.039	0.000	0.318	0.723
ma.L1.D.y	-0.4416	0.127	-3.479	0.001	-0.690	-0.193
ma.L2.D.y	-0.5583	0.126	-4.442	0.000	-0.805	-0.312

Roots

	Real	Imaginary	Modulus	Frequency
AR.1	1.9211	+0.0000j	1.9211	0.0000
MA.1	1.0000	+0.0000j	1.0000	0.0000
MA.2	-1.7911	+0.0000j	1.7911	0.5000

ARIMA Model with Log Transformation

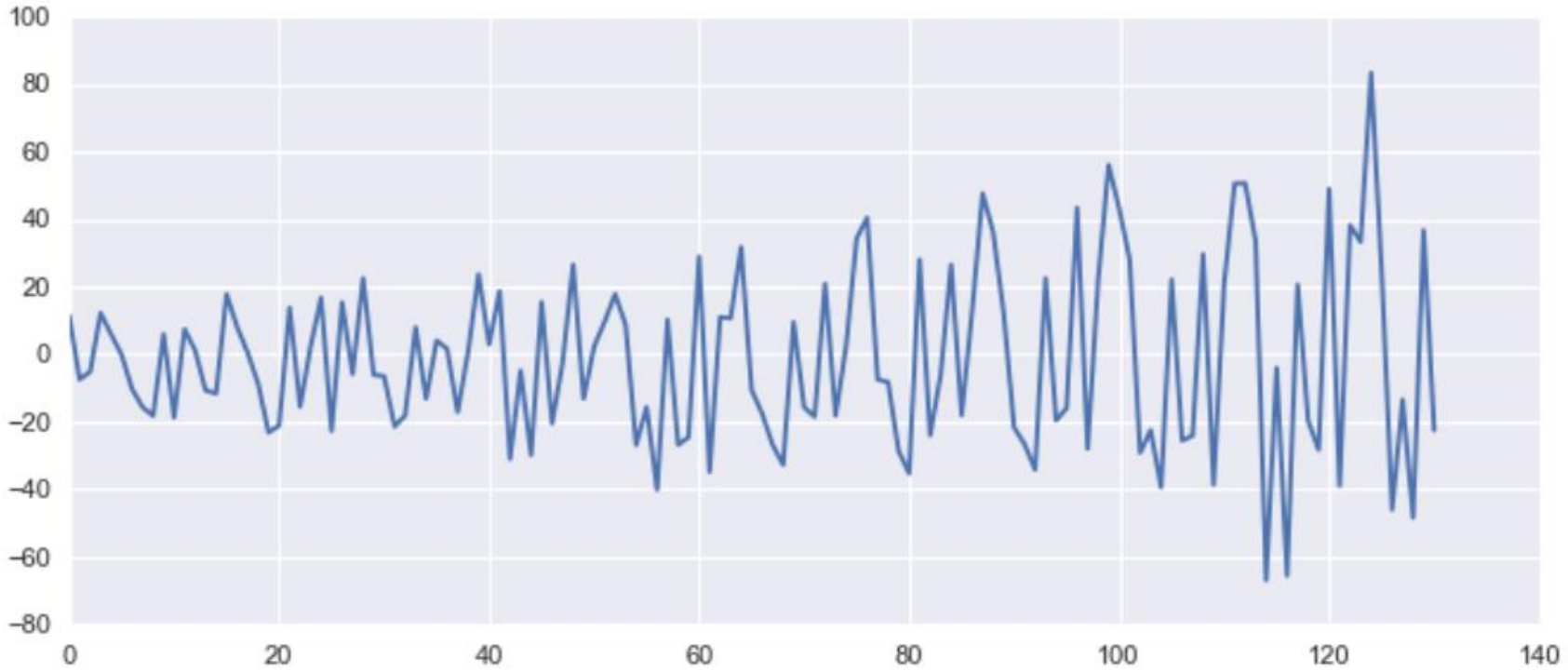
```
test_pred_ARIMA3 = pd.concat([pd.DataFrame(X_air[n-11:]),# use original scale
                        pd.DataFrame(predict_ARIMA3)], axis="columns")

test_pred_ARIMA3.columns = ["actual", "predicted"]
test_pred_ARIMA3.plot(figsize=(10, 4));
```



Model Evaluation

ARIMA Model with Log Transformation

```
# line plot of residuals
plt.figure(figsize=(10, 4))
plt.plot(model_ARIMA3.resid);
```



Model Diagnostics

A line plot of the residuals shows that the residuals do not have a constant variance. This would affect the prediction error interval.

ARIMA Model with Log Transformation

Model with Log Transformation

```
# MSE
mean_squared_error(test_log, predict_ARIMA3)
```

```
190872.11608371304
```

```
model_ARIMA3.aic
```

```
1240.732331226167
```

The overall predicted error or accuracy of the model with the original data is better than that of the model with the transformed data. If models are nested, compare them with AIC. For example, a model with moving average of order 2 is nested in a model with moving average of order 5.

Model with original Data

```
# MSE
mean_squared_error(test, predict_ARIMA2)
```

```
6830.152050985579
```

```
# AIC
model_ARIMA2.aic
```

```
1240.732331226167
```

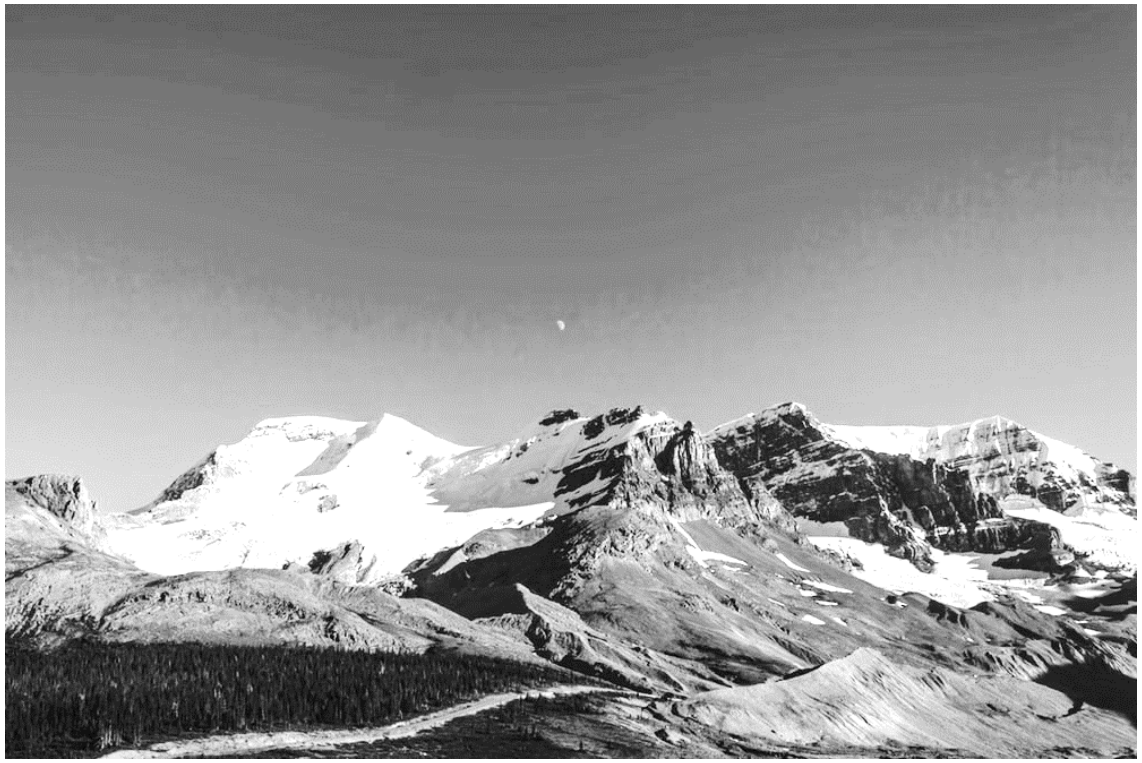
Model Comparison

How to Decide on which Model to Run

- Autocorrelation plots can help with the decision of which model is suitable.
- A time series having an autocorrelation plot with a slowly decaying ACF is suitable for AR(p) Model.
- When there is a sharp cutoff in ACF for values greater than q, the order of the process, then MA is suitable for modeling this time series.
- ARIMA should be used when the ACF of a time series has no sharp cutoff.

How to Decide on which Model to Run

- Note that AR models can be run using ARIMA by setting the d and q to 0.
- MA can also be run by setting d and p to 0.



Model Comparison

- When comparing two models, the one with less Akaike information criterion (AIC) is preferred:
  - $AIC = 2k - 2lnL$, where k is the number of parameters of the model and L is the maximum likelihood value for that function.
- We want less parameters and greater maximum likelihood/goodness of fit so we will favor models with less AIC.
- AIC penalizes the addition of parameters: Great for comparing nested models.

Statistical Time Series Modeling Framework

1. Data Preparation:

- Assess stationarity (constant mean, variance and autocorrelation using ACF)
- Data transformation to reduce or remove variations that change with time, through log or square root.
- Differencing to remove trends and seasonality so as to obtain a stationary time series.

2. Model Selection (Identification)

- Use graphs such as AFC based on the transformed and differenced data to identify potential ARIMA processes which might fit the data.
- Several suitable models could be used, then, compared using AIC, after parameter estimation.

Statistical Time Series Modeling Framework

3. **Parameter Estimation:**
   - This involves finding the values of the model coefficients which provide best fit for the data.

4. Model Checking:
   - Test assumptions of model to identifies areas where model is inadequate. If model is inadequate, go to step 2 and select another better model.

5. Forecasting:
   - Use the selected model to compute forecast if it has been estimated and checked, and it provides a good fit for the data.

Sources of Time Series Data: Internet

```
import pandas_datareader as pdr

plt.figure(figsize=(10, 4))
google = pdr.get_data_yahoo("AAPL", "2019-01-30")
plt.plot(google.index, google.Close);
```

Get Stock Data Using pandas_datareader()

```
# pull data for more than one company
pdr.get_data_yahoo(["AMZN", "GOOG"], "2008-02-01", "2009-02-01").head()
```

Attributes	Adj Close		Close		High		Low		Open		Volume	
Symbols	AMZN	GOOG	AMZN	GOOG	AMZN	GOOG	AMZN	GOOG	AMZN	GOOG	AMZN	GOOG
Date												
2008-02-01	74.629997	256.986755	74.629997	256.986755	79.400002	267.332977	73.370003	254.047760	79.019997	263.347900	16361000	35332900
2008-02-04	73.949997	246.789978	73.949997	246.789978	76.660004	255.432571	73.900002	245.355347	74.500000	253.584503	9155200	26412800
2008-02-05	72.089996	252.453735	72.089996	252.453735	74.209999	253.549637	72.000000	243.347870	72.800003	243.801178	9633900	22490500
2008-02-06	68.489998	249.918243	68.489998	249.918243	72.430000	254.630585	68.169998	248.035309	72.300003	254.615631	12399500	15329900
2008-02-07	70.910004	251.532196	70.910004	251.532196	72.709999	256.134949	67.220001	246.456223	67.370003	247.502304	14501700	15917100

Get Stock Data Using pandas_datareader()

```
# another way of pulling stock data from the internet using
# the data.DataReader() in pandas_datareader
pdr.data.DataReader("AMZN", data_source="yahoo",
                    start="2019-01-01", end="2020-01-01").head()
```

Date	High	Low	Open	Close	Volume	Adj Close
2019-01-02	1553.359985	1460.930054	1465.199951	1539.130005	7983100	1539.130005
2019-01-03	1538.000000	1497.109985	1520.010010	1500.280029	6975600	1500.280029
2019-01-04	1594.000000	1518.310059	1530.000000	1575.390015	9182600	1575.390015
2019-01-07	1634.560059	1589.189941	1602.310059	1629.510010	7993200	1629.510010
2019-01-08	1676.609985	1616.609985	1664.689941	1656.579956	8881400	1656.579956

Get Stock Data Using pandas_datareader()

```
# pull data for more than one company
pdr.get_data_yahoo(["AMZN", "GOOG"], "2008-02-01", "2009-02-01").head()
```

Attributes	Adj Close		Close		High		Low		Open		Volume	
Symbols	AMZN	GOOG	AMZN	GOOG	AMZN	GOOG	AMZN	GOOG	AMZN	GOOG	AMZN	GOOG
Date												
2008-02-01	74.629997	256.986755	74.629997	256.986755	79.400002	267.332977	73.370003	254.047760	79.019997	263.347900	16361000	35332900
2008-02-04	73.949997	246.789978	73.949997	246.789978	76.660004	255.432571	73.900002	245.355347	74.500000	253.584503	9155200	26412800
2008-02-05	72.089996	252.453735	72.089996	252.453735	74.209999	253.549637	72.000000	243.347870	72.800003	243.801178	9633900	22490500
2008-02-06	68.489998	249.918243	68.489998	249.918243	72.430000	254.630585	68.169998	248.035309	72.300003	254.615631	12399500	15329900
2008-02-07	70.910004	251.532196	70.910004	251.532196	72.709999	256.134949	67.220001	246.456223	67.370003	247.502304	14501700	15917100

References

- Box, & Box, George E. P. (2016). *Time series analysis : Forecasting and control* (5th ed., Wiley series in probability and statistics).
- Nielsen, A., & Safari, an O'Reilly Media Company. (2019). *Practical Time Series Analysis (1st.edition).*
- Hyndman, R.J., & Athanasopoulos, G. (2018) Forecasting: principles and practice, 2nd edition, OTexts: Melbourne, Australia. OTexts.com/fpp2. Accessed on 2/8/2020.
- Hyndman, R.J. (2001). The Box-Jenkins approach to modelling ARIMA. https://robjhyndman.com/papers/BoxJenkins.pdf.