Time Series Analysis and Forecasting in Python

Importing Libraries for time series forecasting

In [1027]:

```
import warnings
import itertools
import numpy as np
import matplotlib.pyplot as plt
warnings.filterwarnings("ignore")
plt.style.use('fivethirtyeight')

import pandas as pd
pd.set_option('display.expand_frame_repr', False)
pd.set_option('display.max_columns', 500)
pd.set_option('display.width', 1000)

import statsmodels.api as sm
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.graphics.tsaplots import plot_pacf
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.ar_model import AR
from statsmodels.tsa.arima_model import ARMA, ARIMA
from pyramid.arima import auto_arima
from statsmodels.tsa.statespace.sarimax import SARIMAX
from fbprophet import Prophet

from math import sqrt

import matplotlib
matplotlib.rcParams['axes.labelsize'] = 14
matplotlib.rcParams['xtick.labelsize'] = 12
matplotlib.rcParams['ytick.labelsize'] = 12
matplotlib.rcParams['text.color'] = 'k'
import seaborn as sns

from random import random

from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error,
median_absolute_error, mean_squared_log_error
```

Importing data

- Dataset: International airline passengers
- Unit: Thousands

In [1028]:
```
df = pd.read_csv('international-airline-passengers.csv',header=None)
```

In [1029]:
```
df.columns = ['year','passengers']
```

Out[1030]:

In [1030]:
```
df.head(3)
```

	year	passengers
0	1949-01	112
1	1949-02	118
2	1949-03	132

Out[1031]:

In [1031]:
```
df.describe()
```

	passengers
count	144.000000
mean	280.298611
std	119.966317
min	104.000000
25%	180.000000
50%	265.500000
75%	360.500000
max	622.000000

Out[1032]:

In [1032]:
```
df.describe(include='O')
```

	year
count	144
unique	144
top	1959-07
freq	1

In [1033]:
```
print('Time period start: {}\nTime period end: {}'.format(df.year.min(),df.year.max()))
```

```
Time period start: 1949-01
Time period end: 1960-12
```

In [1034]:
```
df.columns
```

Out[1034]: Index(['year', 'passengers'], dtype='object')

In [1035]:

```
df.shape
```

Out[1035]: (144, 2)

Data Preprocessing and Visualization

Converting to datetime format:

In [1036]:
```
df['year'] = pd.to_datetime(df['year'], format='%Y-%m')
```

Setting index as the datetime column for easier manipulations:

In

In [1037]:
```
y = df.set_index('year')
```

[1038]:
```
y.index
```

Out[1038]:
```
DatetimeIndex(['1949-01-01', '1949-02-01', '1949-03-01', '1949-04-01', '1949-05
-01', '1949-06-01', '1949-07-01', '1949-08-01', '1949-09-01', '1949-10-01',
               ...
               '1960-03-01', '1960-04-01', '1960-05-01', '1960-06-01', '1960-07
-01', '1960-08-01', '1960-09-01', '1960-10-01', '1960-11-01', '1960-12-01'], dt
ype='datetime64[ns]', name='year', length=144, freq=None)
```

In [1039]:
```
y.isnull().sum()
```

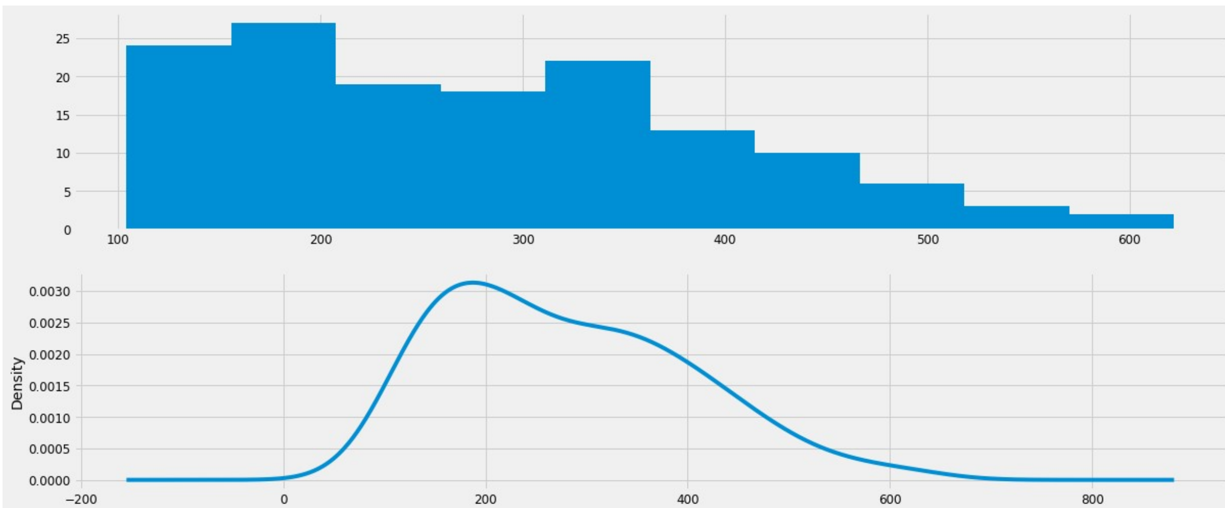
Out[1039]:
```
passengers    0
dtype: int64
```

In [1040]:
```
y.plot(figsize=(15, 6))
plt.show()
```

Reviewing plots of the density of observations can provide further insight into the structure of the data:

- The distribution is not perfectly Gaussian (normal distribution).
- The distribution is left shifted.
- Transformations might be useful prior to modelling.

In [1041]:
```
from pandas import Series
from matplotlib import pyplot
pyplot.figure(1)
pyplot.subplot(211)
y.passengers.hist()
pyplot.subplot(212)
y.passengers.plot(kind='kde')
pyplot.show()
```

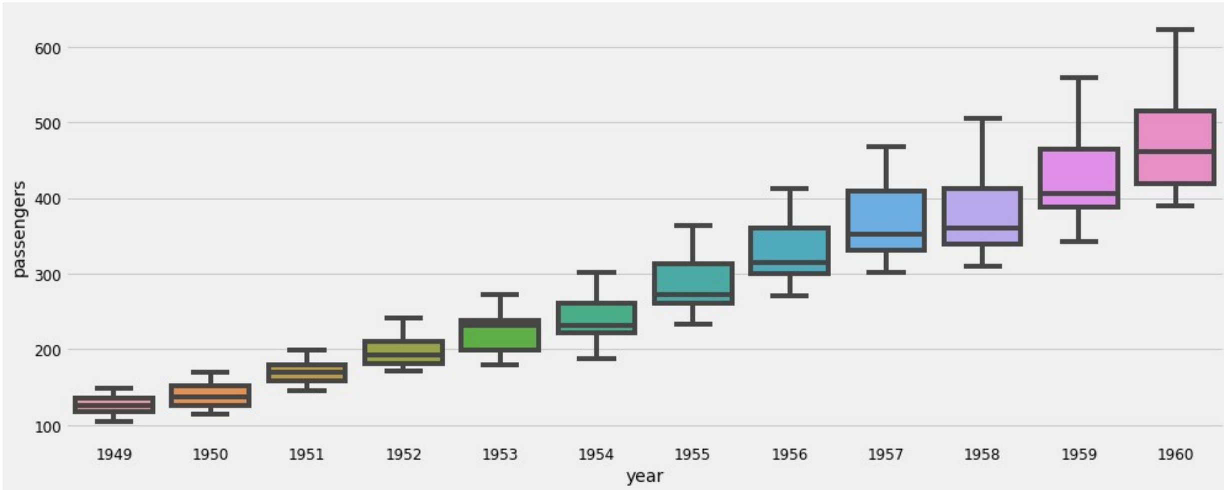


Box and Whisker Plots:

- Median values across years confirms an upwards trend
- Steady increase in the spread, or middle 50% of the data (boxes) over time
- A model considering seasonality might work well

In [1042]:
```
fig, ax = plt.subplots(figsize=(15,6))
sns.boxplot(y.passengers.index.year, y.passengers, ax=ax)
```

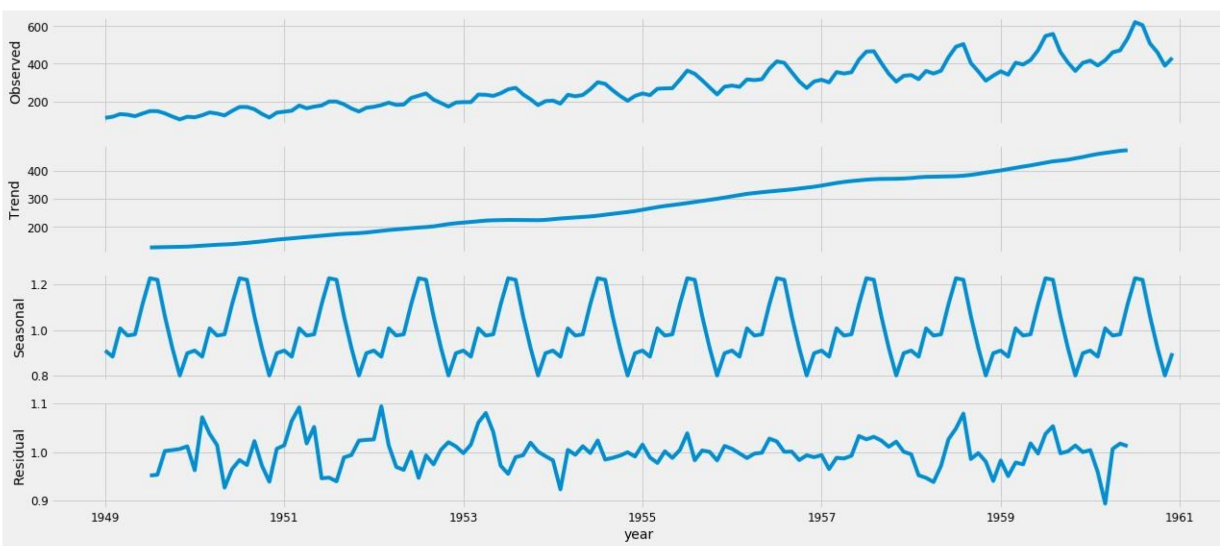
Out[1042]: <matplotlib.axes._subplots.AxesSubplot at 0x27a066ba9b0>



Decomposing using statsmodel:

- We can use statsmodels to perform a decomposition of this time series.
- The decomposition of time series is a statistical task that deconstructs a time series into several components, each representing one of the underlying categories of patterns. With
- statsmodels we will be able to see the trend, seasonal, and residual components of our data.

In [1043]:
```
from pylab import rcParams
rcParams['figure.figsize'] = 18, 8
decomposition = sm.tsa.seasonal_decompose(y, model='multiplicative')
fig = decomposition.plot()
plt.show()
```



Stationarity

- A Time Series is said to be stationary if its statistical properties such as mean, variance remain constant over time.
-

Most of the Time Series models work on the assumption that the TS is stationary. Major reason for this is that there are many ways in which a series can be non-stationary, but only one way for stationarity.

- Intuitively, we can say that if a Time Series has a particular behaviour over time, there is a very high probability that it will follow the same in the future.
- Also, the theories related to stationary series are more mature and easier to implement as compared to non-stationary series.

In [1044]:
```
plt.plot(y)
```

Out[1044]: [<matplotlib.lines.Line2D at 0x27a03eb5438>]

We can check stationarity using the following:

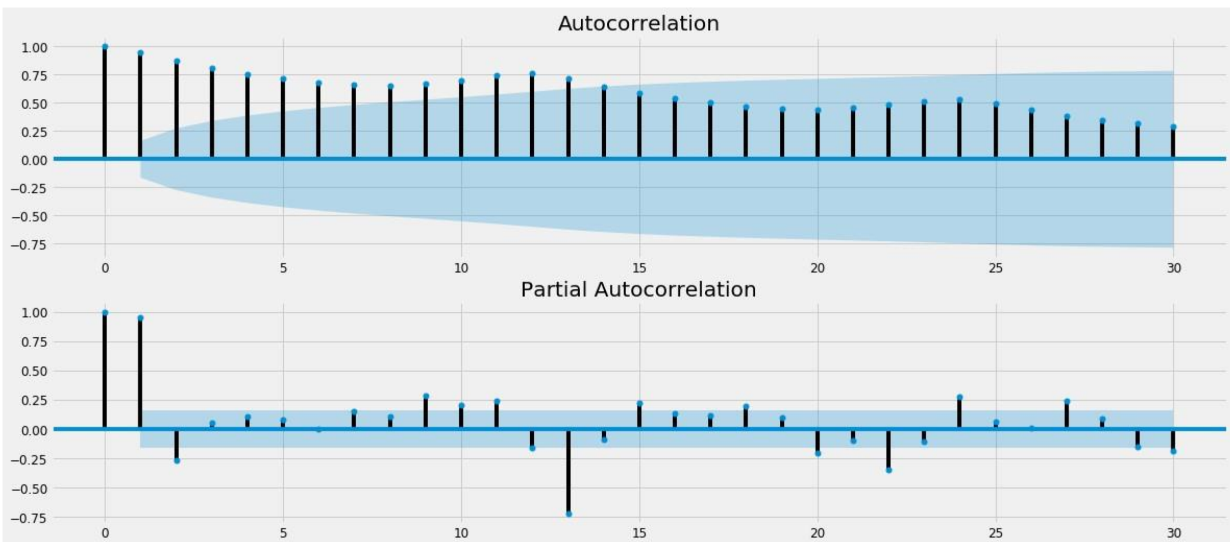
- **ACF and PACF plots**: If the time series is stationary, the ACF/PACF plots will show a **quick drop-off in correlation** after a small amount of lag between points.
- **Plotting Rolling Statistics**: We can plot the moving average or moving variance and see if it varies with time. Moving average/variance is for any instant ‘t’, the average/variance of the last year, i.e. last 12 months.
- **Augmented Dickey-Fuller Test:** This is one of the statistical tests for checking stationarity. Here the null hypothesis is that the TS is non-stationary. The test results comprise of a Test Statistic and some Critical Values for difference confidence levels. If the ‘Test Statistic’ is less than the ‘Critical Value’, we can reject the null hypothesis and say that the series is stationary. Refer this article for details.

ACF and PACF plots

- Let's review the Autocorrelation Function (ACF) and Partial Autocorrelation Function (PACF) plots
- If the time series is stationary, the ACF/PACF plots will show a **quick drop-off in correlation** after a small amount of lag between points.

- This data is non-stationary as a high number of previous observations are correlated with future values.
- Confidence intervals are drawn as a cone.
- By default, this is set to a 95% confidence interval, suggesting that correlation values outside of this code are very likely a correlation and not a statistical fluke.
- The partial autocorrelation at lag k is the correlation that results after removing the effect of any correlations due to the terms at shorter lags.

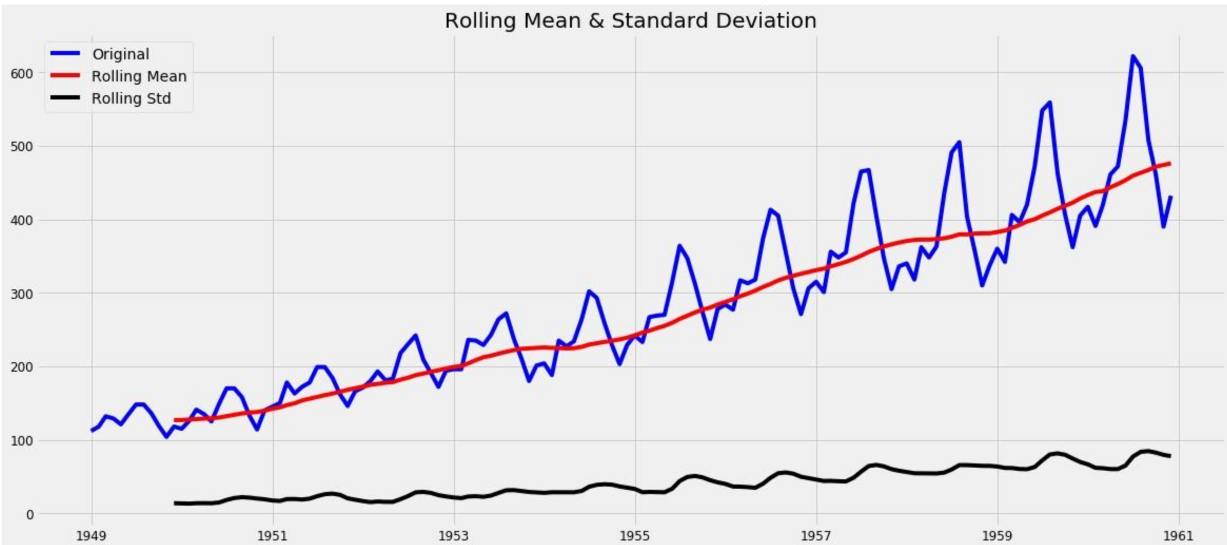
In [1045]:
```
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.graphics.tsaplots import plot_pacf

pyplot.figure()
pyplot.subplot(211)
plot_acf(y.passengers, ax=pyplot.gca(), lags = 30)
pyplot.subplot(212)
plot_pacf(y.passengers, ax=pyplot.gca(), lags = 30)
pyplot.show()
```



Plotting Rolling Statistics

- We observe that the rolling mean and Standard deviation are not constant with respect to time (increasing trend)
- The time series is hence not stationary

In [1046]:
```
#Determing rolling statistics
rolmean = pd.rolling_mean(y, window=12)
rolstd = pd.rolling_std(y, window=12)

#Plot rolling statistics:
orig = plt.plot(y, color='blue',label='Original')
mean = plt.plot(rolmean, color='red', label='Rolling Mean')
std = plt.plot(rolstd, color='black', label = 'Rolling Std')
plt.legend(loc='best')
plt.title('Rolling Mean & Standard Deviation')
plt.show(block=False)
```



Augmented Dickey-Fuller Test

- The intuition behind the test is that if the series is integrated then the lagged level of the series y(t-1) will provide no relevant information in predicting the change in y(t).
- Null hypothesis: The time series is not stationary
- Rejecting the null hypothesis (i.e. a very low p-value) will indicate staionarity

In [1047]:
```
from statsmodels.tsa.stattools import adfuller
```

```
[1048]: #Perform Dickey-Fuller test: print ('Results
 of Dickey-Fuller Test:') dftest =
 adfuller(y.passengers, autolag='AIC')
 dfoutput = pd.Series(dftest[0:4], index=['Test Statistic','p-value','#Lags
 Used','Number of Observations Used']) for key,value in dftest[4].items():
 dfoutput['Critical Value (%s)'%key] = value print (dfoutput)
```

```
Results of Dickey-Fuller Test:
Test Statistic                    0.815369 p-
value                           0.991880 #Lags
Used                       13.000000
Number of Observations Used    130.000000
Critical Value (1%)             -3.481682
Critical Value (5%)             -2.884042
Critical Value (10%)            -2.578770
dtype: float64
```

```
In [1049]: def test_stationarity(timeseries):
    #Determing rolling statistics
    rolmean = pd.rolling_mean(timeseries, window=12)
rolstd = pd.rolling_std(timeseries, window=12)

    #Plot rolling statistics:
    orig = plt.plot(timeseries, color='blue',label='Original')
mean = plt.plot(rolmean, color='red', label='Rolling Mean')
std = plt.plot(rolstd, color='black', label = 'Rolling Std')
plt.legend(loc='best')
    plt.title('Rolling Mean & Standard Deviation')
plt.show(block=False)

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

Making Time Series Stationary

There are 2 major reasons behind non-stationaruty of a TS:

1. **Trend** – varying mean over time. For eg, in this case we saw that on average, the number of passengers was growing over time.
2. **Seasonality** – variations at specific time-frames. eg people might have a tendency to buy cars in a particular month because of pay increment or festivals.

Transformations

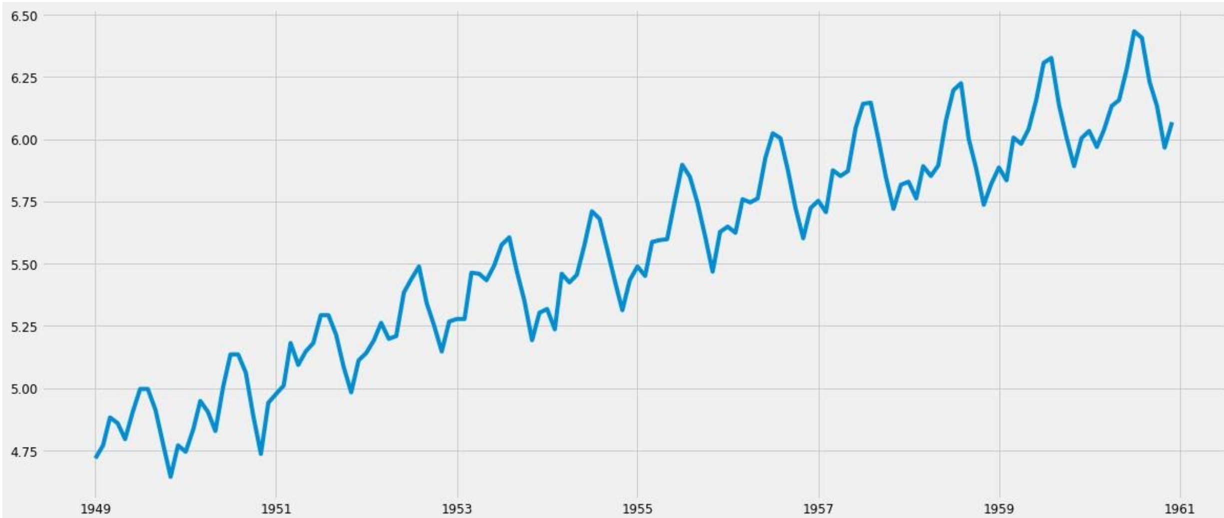
- We can apply transformation which penalize higher values more than smaller values. These can be taking a log, square root, cube root, etc. Lets take a log transform here for simplicity:

In

Log Scale Transformation

In [1050]:
```
ts_log = np.log(y)
plt.plot(ts_log)
```

Out[1050]: [<matplotlib.lines.Line2D at 0x27a06cfde10>]



Other possible transformations:

- Exponential tranformation
- Box Cox transformation
- Square root transformation

Techniques to remove Trend - Smoothing

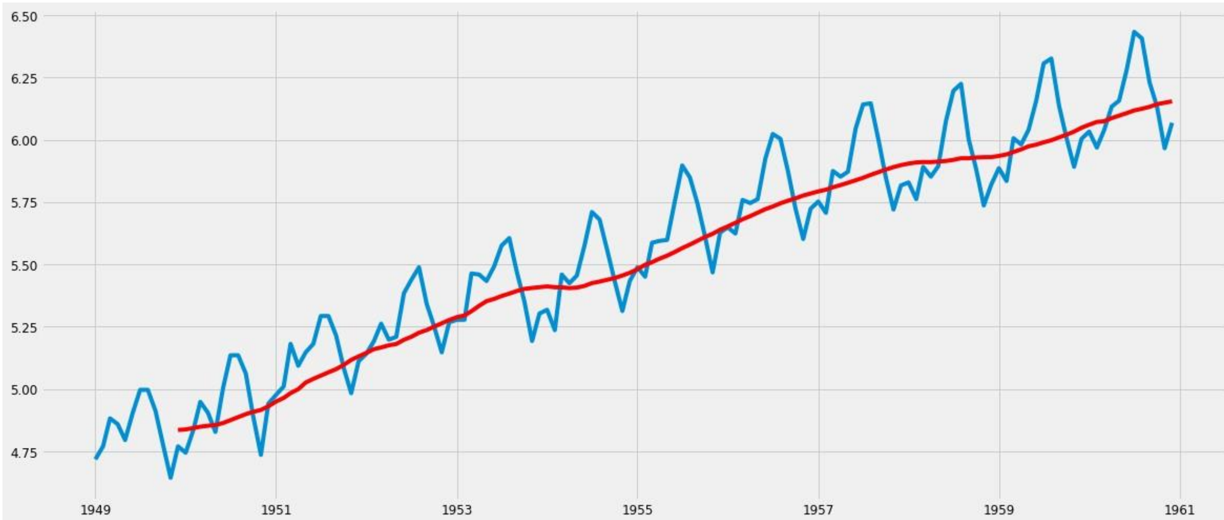
- Smoothing is taking rolling averages over windows of time

Moving Average

- We take average of ‘k’ consecutive values depending on the frequency of time series.
- Here we can take the average over the past 1 year, i.e. last 12 values.
- A drawback in this particular approach is that the time-period has to be strictly defined.

[1051]:
```
moving_avg = pd.rolling_mean(ts_log,12)
plt.plot(ts_log)
plt.plot(moving_avg, color='red')
```

Out[1051]: [<matplotlib.lines.Line2D at 0x27a097d86d8>]

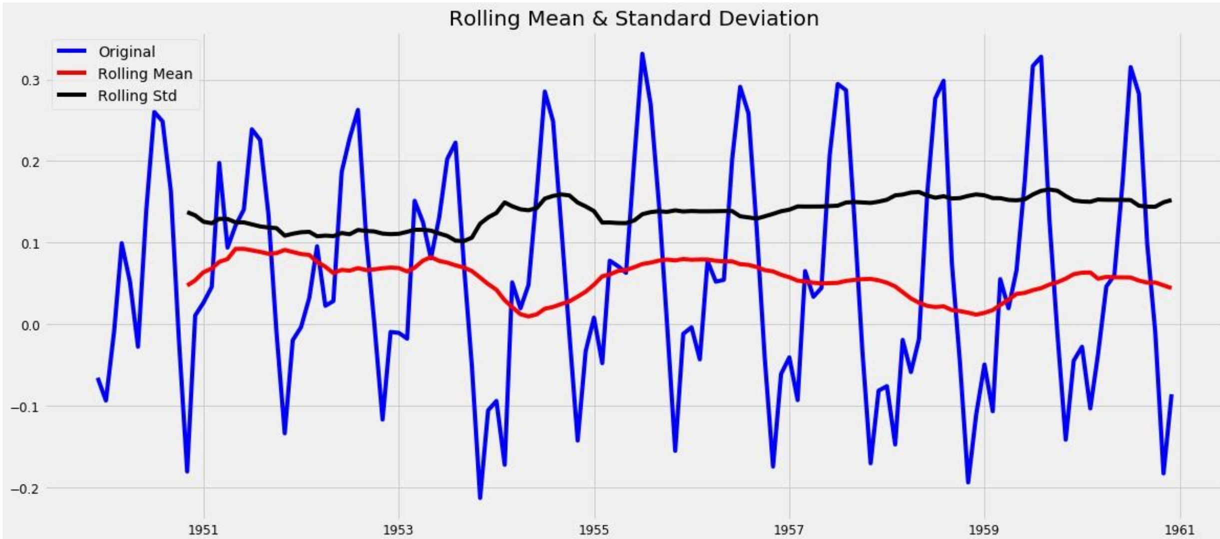



```
In [1052]: ts_log_moving_avg_diff = ts_log.passengers - moving_avg.passengers
           ts_log_moving_avg_diff.head(12)
```

```
Out[1052]: year
           1949-01-01         NaN
           1949-02-01         NaN
           1949-03-01         NaN
           1949-04-01         NaN
           1949-05-01         NaN
           1949-06-01         NaN
           1949-07-01         NaN
           1949-08-01         NaN
           1949-09-01         NaN
           1949-10-01         NaN
           1949-11-01         NaN
           1949-12-01   -0.065494
           Name: passengers, dtype: float64
```

In

In [1053]:
```
ts_log_moving_avg_diff.dropna(inplace=True)
test_stationarity(ts_log_moving_avg_diff)
```



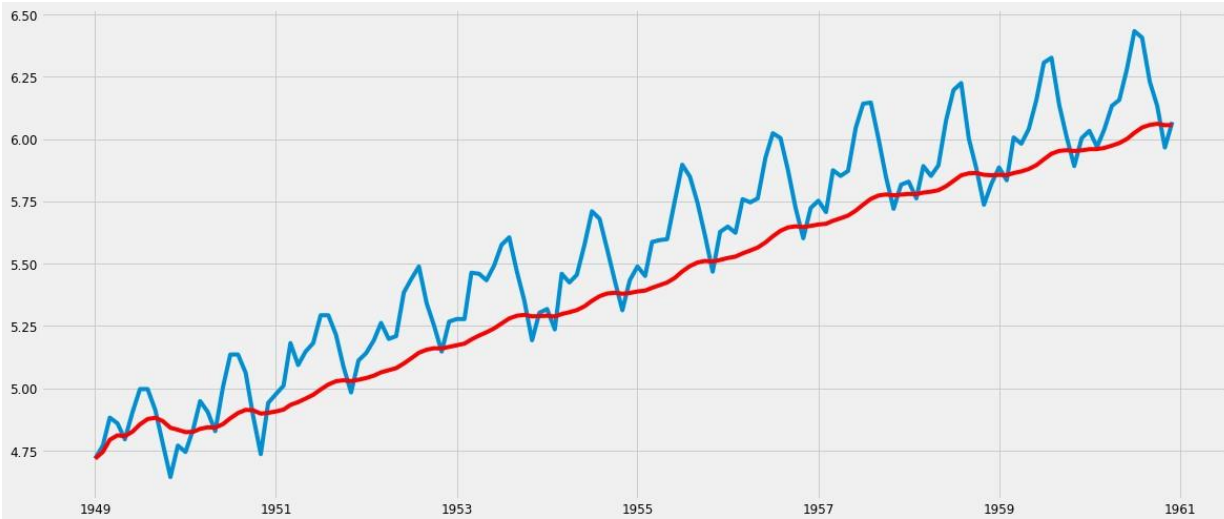

```
Results of Dickey-Fuller Test:
Test Statistic                 -3.162908 p-
value                         0.022235 #Lags
Used                     13.000000
Number of Observations Used    119.000000
Critical Value (1%)            -3.486535
Critical Value (5%)            -2.886151
Critical Value (10%)           -2.579896
dtype: float64
```

Exponentially weighted moving average:

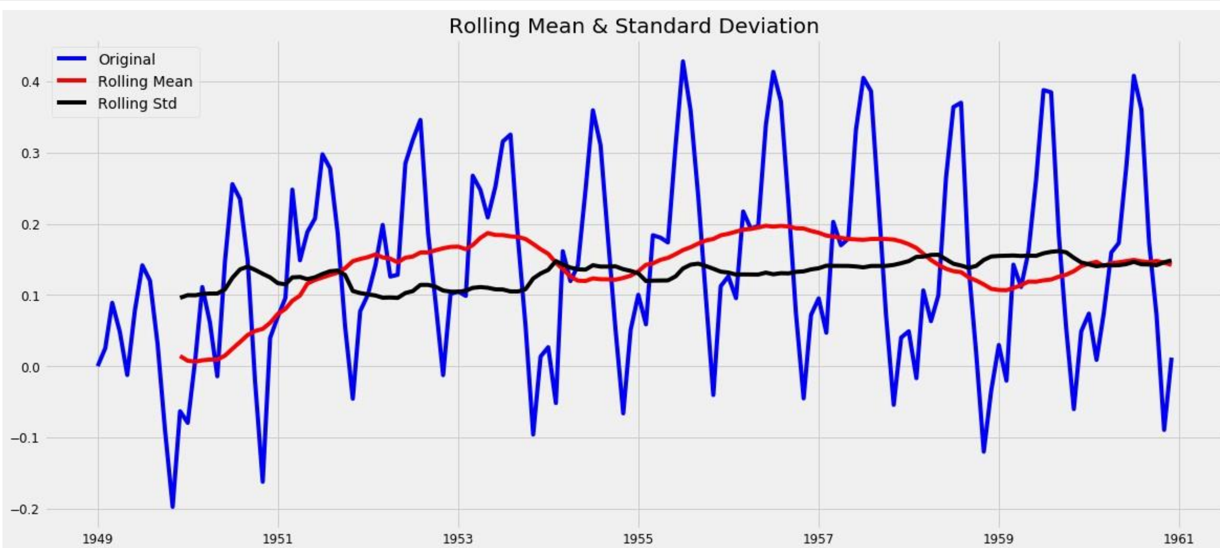
- To overcome the problem of choosing a defined window in moving average, we can use exponential weighted moving average
- We take a ‘weighted moving average’ where more recent values are given a higher weight.
- There can be many technique for assigning weights. A popular one is exponentially weighted moving average where weights are assigned to all the previous values with a decay factor.

[1054]:
```
expwighted_avg = pd.ewma(ts_log, halflife=12)
plt.plot(ts_log)
plt.plot(expwighted_avg, color='red')
```

Out[1054]: [<matplotlib.lines.Line2D at 0x27a75923748>]



In [1055]:
```
ts_log_ewma_diff = ts_log.passengers - expwighted_avg.passengers
test_stationarity(ts_log_ewma_diff)
```




```
Results of Dickey-Fuller Test:
Test Statistic                  -3.601262
p-value                          0.005737
#Lags Used                      13.000000
Number of Observations Used    130.000000
Critical Value (1%)             -3.481682
Critical Value (5%)             -2.884042
Critical Value (10%)            -2.578770
dtype: float64
```

In

Further Techniques to remove Seasonality and Trend

- The simple trend reduction techniques discussed before don't work in all cases, particularly the ones with high seasonality.

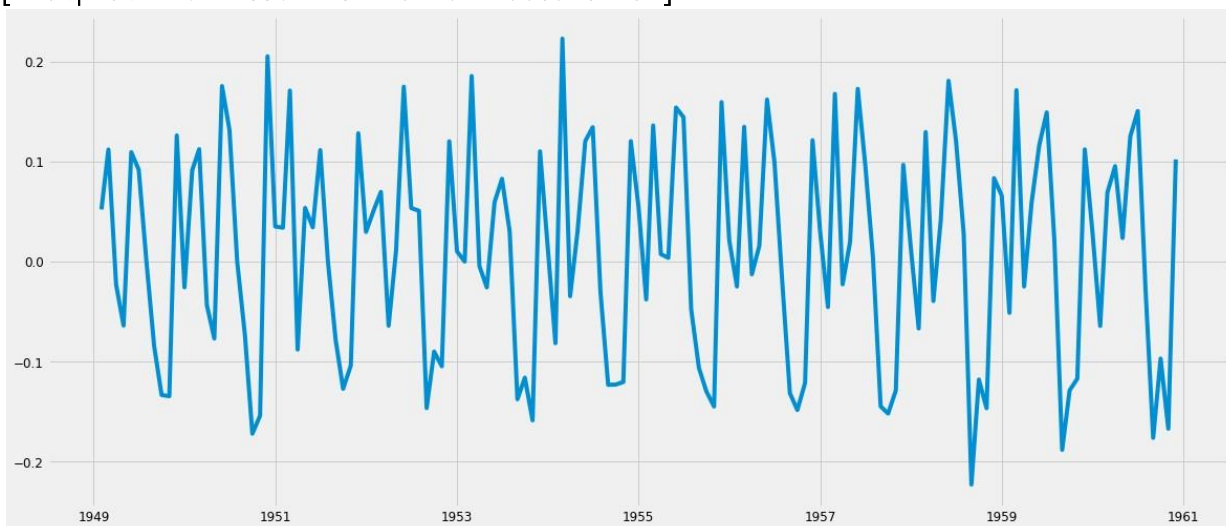
Differencing

- In this technique, we take the difference of the observation at a particular instant with that at the previous instant.
- First order differencing in Pandas

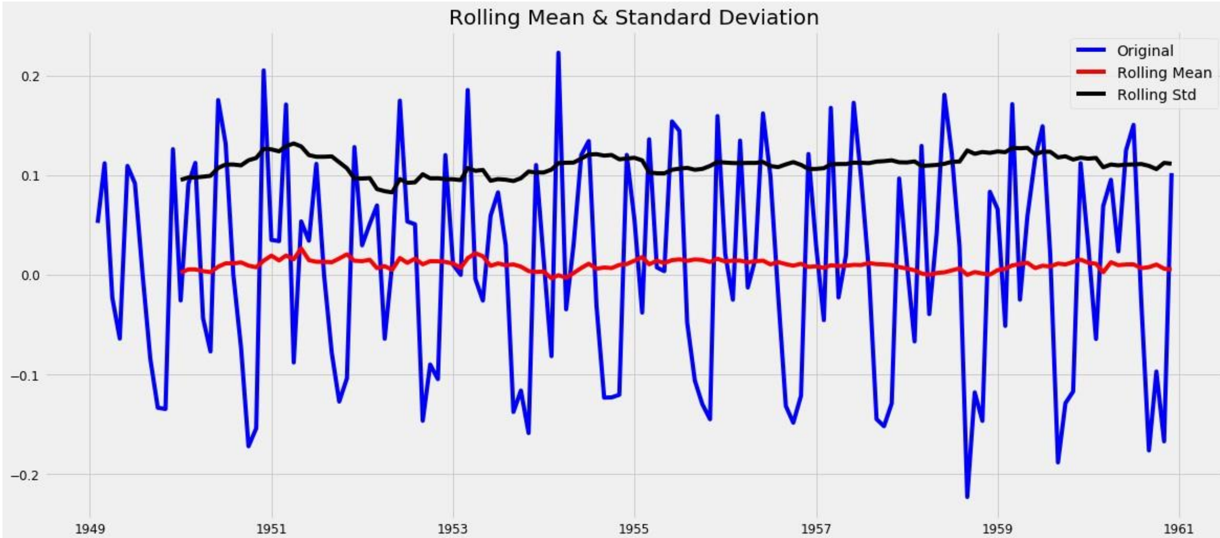
In [1056]:

```
ts_log_diff = ts_log.passengers - ts_log.passengers.shift()
plt.plot(ts_log_diff)
```

Out[1056]: [<matplotlib.lines.Line2D at 0x27a06d26978>]



In [1057]:
```
ts_log_diff.dropna(inplace=True)
test_stationarity(ts_log_diff)
```



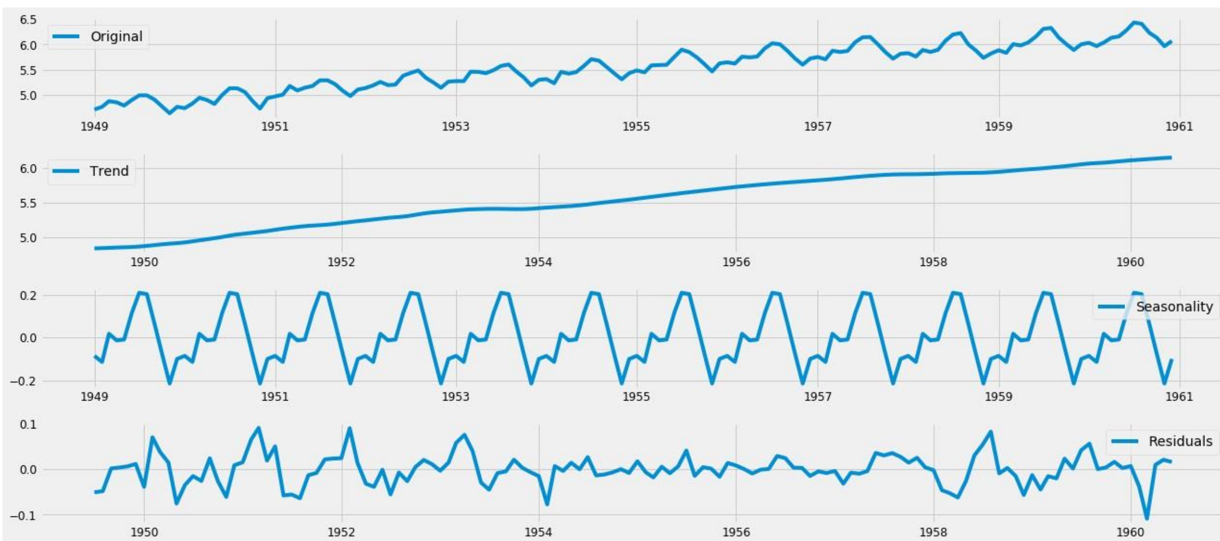

```
Results of Dickey-Fuller Test:
Test Statistic                  -2.717131 p-
value                          0.071121 #Lags
Used                      14.000000
Number of Observations Used    128.000000
Critical Value (1%)             -3.482501
Critical Value (5%)             -2.884398
Critical Value (10%)            -2.578960
dtype: float64
```

Decomposition

- In this approach, both trend and seasonality are modeled separately and the remaining part of the series is returned.

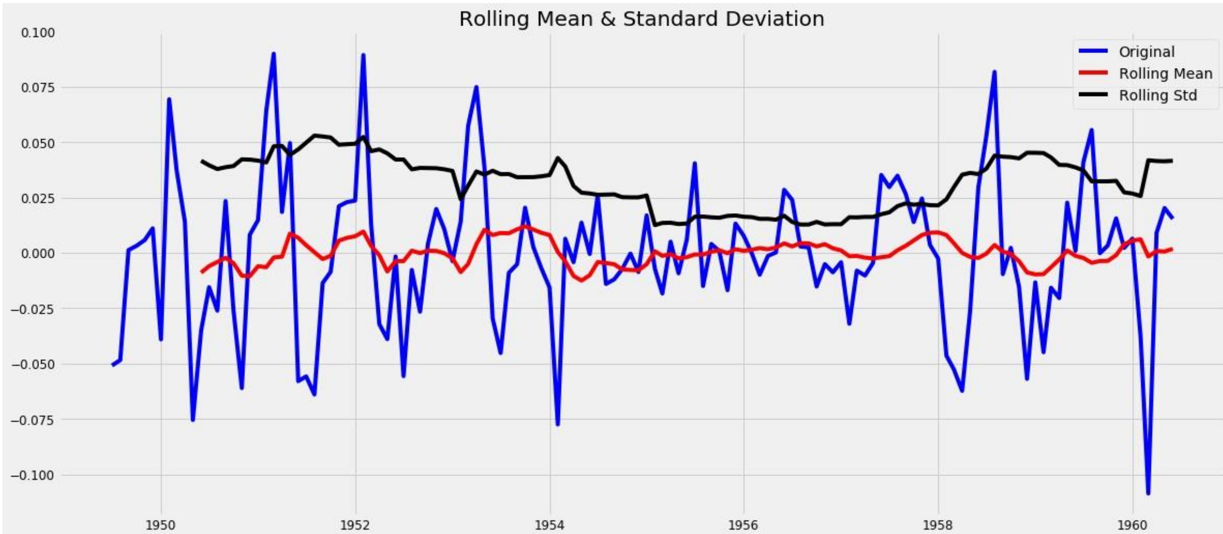
In [1058]:

```
from statsmodels.tsa.seasonal import seasonal_decompose
decomposition = seasonal_decompose(ts_log)

trend = decomposition.trend
seasonal = decomposition.seasonal
residual = decomposition.resid

plt.subplot(411)
plt.plot(ts_log, label='Original')
plt.legend(loc='best')
plt.subplot(412)
plt.plot(trend, label='Trend')
plt.legend(loc='best')
plt.subplot(413)
plt.plot(seasonal,label='Seasonality')
plt.legend(loc='best')
plt.subplot(414)
plt.plot(residual, label='Residuals')
plt.legend(loc='best')
plt.tight_layout()
```



In [1059]:

```
ts_log_decompose = residual.passengers
ts_log_decompose.dropna(inplace=True)
test_stationarity(ts_log_decompose)
```




```
Results of Dickey-Fuller Test:
Test Statistic                  -6.332387e+00 p-
value                          2.885059e-08 #Lags
Used                       9.000000e+00
Number of Observations Used     1.220000e+02
Critical Value (1%)            -3.485122e+00
Critical Value (5%)            -2.885538e+00
Critical Value (10%)           -2.579569e+00
dtype: float64
```

Time Series forecasting

[Statsmodel example notebooks (https://github.com/statsmodels/statsmodels/tree/master/examples/notebooks)](https://github.com/statsmodels/statsmodels/tree/master/examples/notebooks)

Autoregression (AR)

- The autoregression (AR) method models the next step in the sequence as a linear function of the observations at prior time steps.
-

In [1060]:

```
from statsmodels.tsa.ar_model import AR
from random import random
```

In [1061]:

```
# fit model
model = AR(ts_log_diff)
model_fit = model.fit()
```

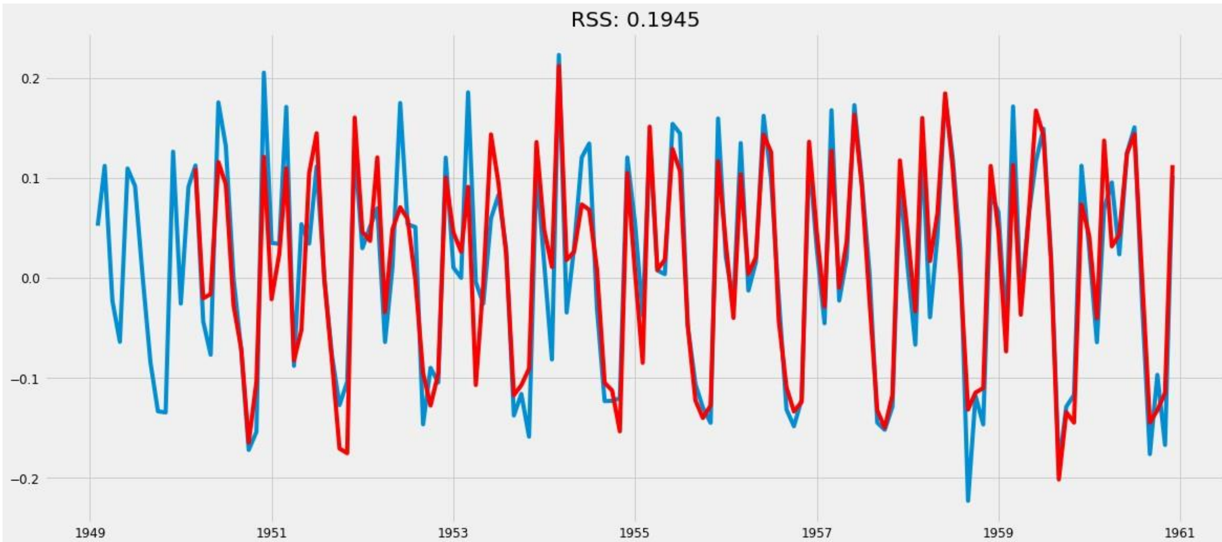
In [1060]:

Number of AR (Auto-Regressive) terms (p): p is the parameter associated with the autoregressive aspect of the model, which incorporates past values i.e lags of dependent variable.

For instance if p is 5, the predictors for x(t) will be x(t-1)….x(t-5).

In [1062]:

```
plt.plot(ts_log_diff)
plt.plot(model_fit.fittedvalues, color='red')
plt.title('RSS: %.4f'% np.nansum((model_fit.fittedvalues-ts_log_diff)**2))
plt.show()
```



Reversing the transformations

Fitted or predicted values:

In [1063]:

```
predictions_ARIMA_diff = pd.Series(model_fit.fittedvalues, copy=True)
print (predictions_ARIMA_diff.head())
```

```
year
1950-03-01    0.109713
1950-04-01   -0.020423
1950-05-01   -0.016243
1950-06-01    0.115842 1950-
07-01    0.093564 dtype:
float64
```

Cumulative Sum to reverse differencing:

In [1064]:

```
predictions_ARIMA_diff_cumsum = predictions_ARIMA_diff.cumsum()
print (predictions_ARIMA_diff_cumsum.head())
```

```
year
1950-03-01    0.109713
1950-04-01    0.089291
1950-05-01    0.073048
1950-06-01    0.188891 1950-
07-01    0.282455 dtype:
float64
```

In

Adding 1st month value which was previously removed while differencing:

[1065]:
```
predictions_ARIMA_log = pd.Series(ts_log.passengers.iloc[0], index=ts_log.index)
predictions_ARIMA_log =
predictions_ARIMA_log.add(predictions_ARIMA_diff_cumsum,fill_value=0)
predictions_ARIMA_log.head()
```

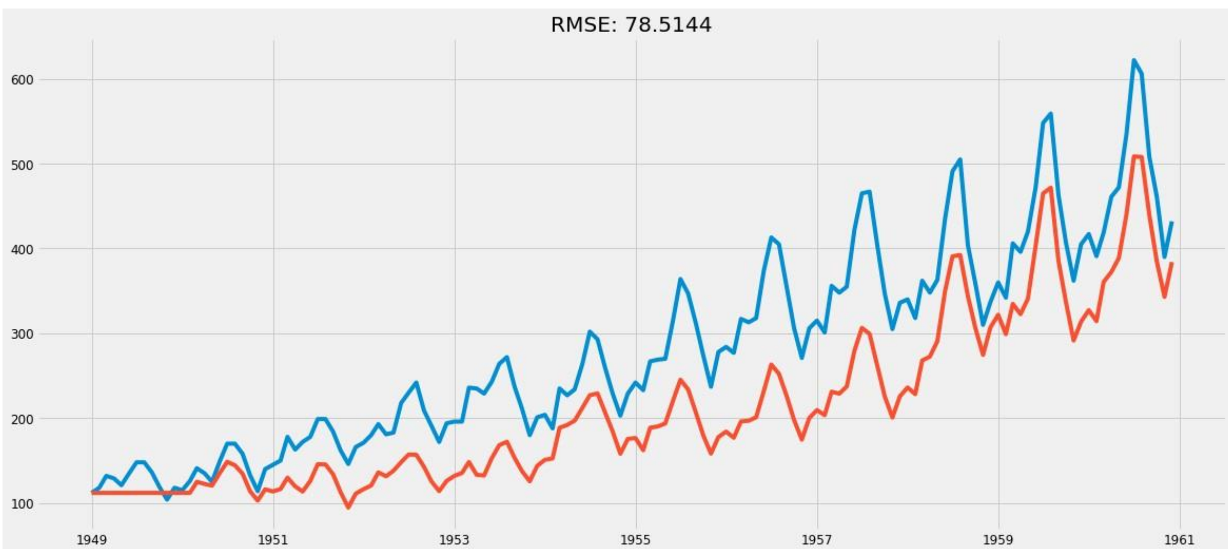
Out[1065]:
```
year
1949-01-01    4.718499
1949-02-01    4.718499
1949-03-01    4.718499
1949-04-01    4.718499 1949-
05-01    4.718499 dtype:
float64
```

Taking Exponent to reverse Log Transform:

In [1066]:
```
predictions_ARIMA = np.exp(predictions_ARIMA_log)
```

In [1067]:
```
plt.plot(y.passengers)
plt.plot(predictions_ARIMA)
plt.title('RMSE: %.4f'% np.sqrt(np.nansum((predictions_ARIMA-
y.passengers)**2)/len(y.passengers)))
```

Out[1067]: `Text(0.5,1,'RMSE: 78.5144')`



Forecast quality scoring metrics

- **R squared**
- **Mean Absolute Error**
- **Median Absolute Error**
- **Mean Squared Error**
- **Mean Squared Logarithmic Error**
- **Mean Absolute Percentage Error**

In [1068]:
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error,
median_absolute_error, mean_squared_log_error
```

R squared, coefficient of determination (it can be interpreted as a percentage of variance explained by the model), (-inf, 1]

- sklearn.metrics.r2_score

In [1069]:
```
r2_score(y.passengers, predictions_ARIMA)
```

Out[1069]:
```
0.5686734896130763
```

Mean Absolute Error, it is an interpretable metric because it has the same unit of measurement as the initial series, [0, +inf)

- sklearn.metrics.mean_absolute_error

In [1070]:
```
mean_absolute_error(y.passengers, predictions_ARIMA)
```

Out[1070]:
```
69.4286283887273
```

Median Absolute Error, again an interpretable metric, particularly interesting because it is robust to outliers, [0, +inf)

- sklearn.metrics.median_absolute_error

In [1071]:
```
median_absolute_error(y.passengers, predictions_ARIMA)
```

Out[1071]:
```
69.36695435384745
```

Mean Squared Error, most commonly used, gives higher penalty to big mistakes and vise versa, [0, +inf)

- sklearn.metrics.mean_squared_error

In [1072]:
```
mean_squared_error(y.passengers, predictions_ARIMA)
```

Out[1072]:
```
6164.506983577602
```

Mean Squared Logarithmic Error, practically the same as MSE but we initially take logarithm of the series, as a result we give attention to small mistakes as well, usually is used when data has exponential trends, [0, +inf) •sklearn.metrics.mean_squared_log_error

[1073]:
```
mean_squared_log_error(y.passengers, predictions_ARIMA)
```

Out[1073]:
```
0.09945599448249715
```

Mean Absolute Percentage Error, same as MAE but percentage,—very convenient when you want to explain the quality of the model to your management, [0, +inf),

In

- not implemented in sklearn

In [1074]:
```
def mean_absolute_percentage_error(y_true, y_pred):
    return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
```

In [1075]:
```
mean_absolute_percentage_error(y.passengers, predictions_ARIMA)
```

Out[1075]:
```
24.47240542986229
```

Function to evaluate forecast using above metrics:

In [1076]:
```
def evaluate_forecast(y,pred):     results = 
pd.DataFrame({'r2_score':r2_score(y, pred),
                            }, index=[0])
    results['mean_absolute_error'] = mean_absolute_error(y, pred)
results['median_absolute_error'] = median_absolute_error(y, pred)
results['mse'] = mean_squared_error(y, pred)     results['msle'] = 
mean_squared_log_error(y, pred)     results['mape'] = 
mean_absolute_percentage_error(y, pred)     results['rmse'] = 
np.sqrt(results['mse'])     return results
```

In [1077]:
```
evaluate_forecast(y.passengers, predictions_ARIMA)
```

Out[1077]:

	r2_score	mean_absolute_error	median_absolute_error	mse	msle	mape	rm
0	0.568673	69.428628	69.366954	6164.506984	0.099456	24.472405	78.514

- RMSE has the benefit of penalizing large errors more so can be more appropriate in some cases, for example, if being off by 10 is more than twice as bad as being off by 5. But if being off by 10 is just twice as bad as being off by 5, then MAE is more appropriate.
- From an interpretation standpoint, MAE is clearly the winner. RMSE does not describe average error alone and has other implications that are more difficult to tease out and
- understand. On the other hand, one distinct advantage of RMSE over MAE is that RMSE avoids the use of taking the absolute value, which is undesirable in many mathematical calculations

Moving Average (MA)

- **Number of MA (Moving Average) terms (q):** q is size of the moving average part window of the model i.e. lagged forecast errors in prediction equation. For instance if q is 5, the predictors for x(t) will be e(t-1)….e(t-5) where e(i) is the difference between the moving average at ith instant and actual value.

In [1078]:
```
# MA example
from statsmodels.tsa.arima_model import ARMA
from random import random

# fit model
model = ARMA(ts_log_diff, order=(0, 1))
model_fit = model.fit(disp=False)
```

In [1079]:
```
model_fit.summary()
```

Out[1079]:

ARMA Model Results

Dep. Variable:	passengers	**No. Observations:**	143
Model:	ARMA(0, 1)	**Log Likelihood**	121.754
Method:	css-mle	**S.D. of innovations**	0.103
Date:	Tue, 11 Dec 2018	**AIC**	-237.507
Time:	14:19:21	**BIC**	-228.619
Sample:	02-01-1949 - 12-01-1960	**HQIC**	-233.895

	coef	std err	z	P>|z|	[0.025	0.975]
const	0.0097	0.011	0.887	0.377	-0.012	0.031
ma.L1.passengers	0.2722	0.095	2.873	0.005	0.086	0.458

Roots

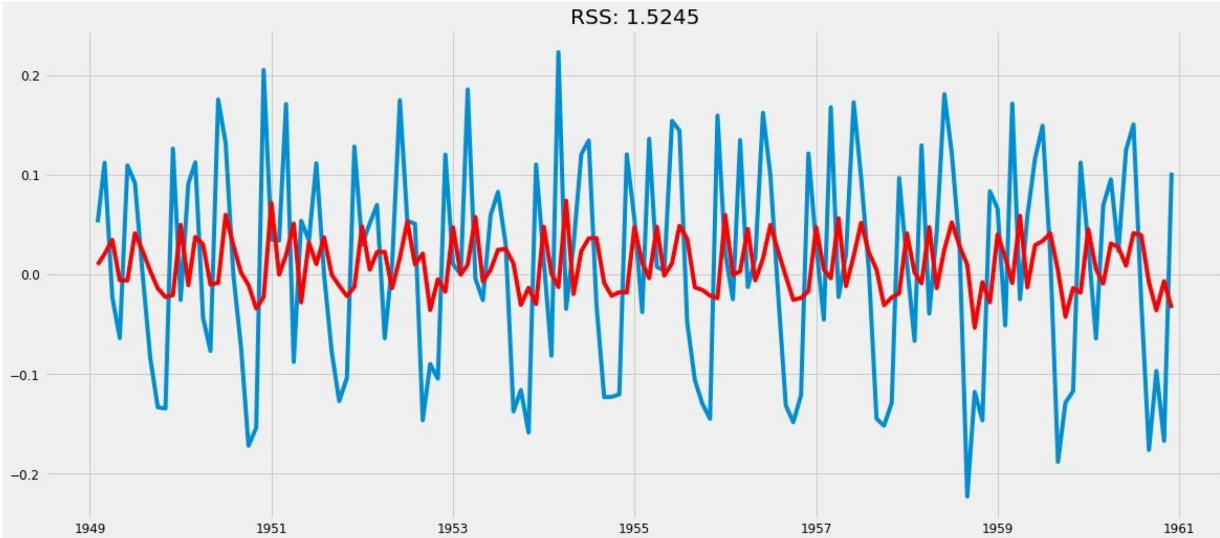
	Real	Imaginary	Modulus	Frequency
MA.1	-3.6744	+0.0000j	3.6744	0.5000

[1080]:
```
plt.plot(ts_log_diff)    plt.plot(model_fit.fittedvalues,    color='red')
plt.title('RSS: %.4f'% np.nansum((model_fit.fittedvalues-ts_log_diff)**2))
```

Out[1080]: Text(0.5,1,'RSS: 1.5245')

In [1081]:
```
# ARMA example
from statsmodels.tsa.arima_model import ARMA
from random import random

# fit model
model = ARMA(ts_log_diff, order=(2, 1))
model_fit = model.fit(disp=False)
```

In



Autoregressive Moving Average (ARMA)

- **Number of AR (Auto-Regressive) terms (p):** p is the parameter associated with the autoregressive aspect of the model, which incorporates past values i.e lags of dependent variable.
  For instance if p is 5, the predictors for x(t) will be x(t-1)….x(t-5).
- **Number of MA (Moving Average) terms (q):** q is size of the moving average part window of the model i.e. lagged forecast errors in prediction equation. For instance if q is 5, the predictors for x(t) will be e(t-1)….e(t-5) where e(i) is the difference between the moving average at ith instant and actual value.

[1082]:
```
model_fit.summary()
```

Out[1082]:

ARMA Model Results

Dep. Variable:	passengers	**No. Observations:**	143
Model:	ARMA(2, 1)	**Log Likelihood**	140.076
Method:	css-mle	**S.D. of innovations**	0.090
Date:	Tue, 11 Dec 2018	**AIC**	-270.151
Time:	14:19:22	**BIC**	-255.337

In

Sample:	02-01-1949		**HQIC**	-264.131	
	- 12-01-1960				

	coef	std err	z	P>|z|	[0.025	0.975]
const	0.0101	0.000	23.509	0.000	0.009	0.011
ar.L1.passengers	0.9982	0.076	13.162	0.000	0.850	1.147
ar.L2.passengers	-0.4134	0.077	-5.384	0.000	-0.564	-0.263
ma.L1.passengers	-1.0000	0.028	-35.273	0.000	-1.056	-0.944

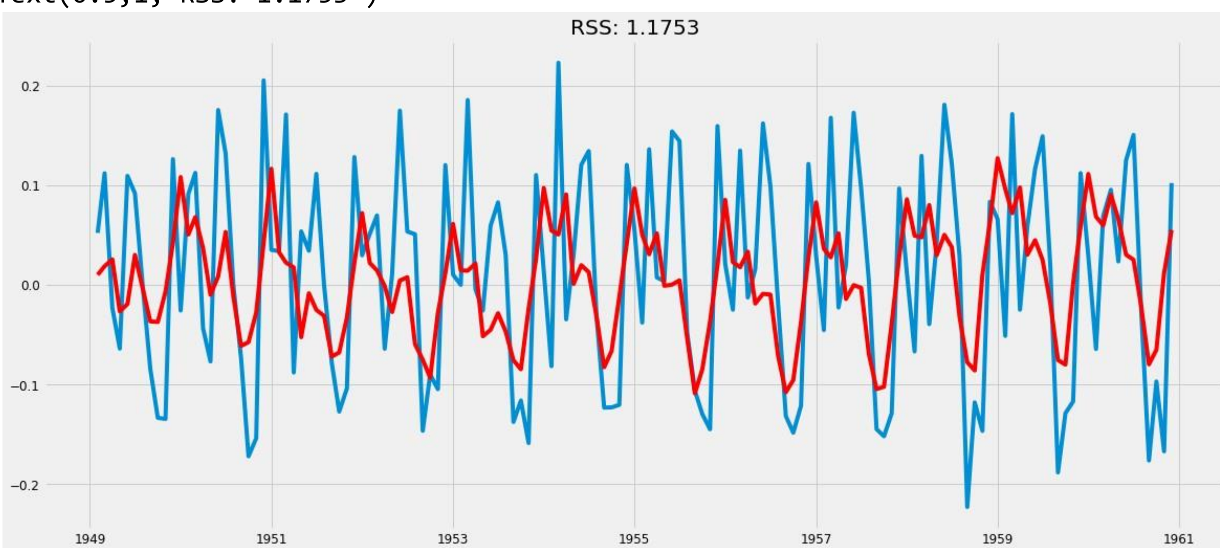
Roots

	Real	Imaginary	Modulus	Frequency
AR.1	1.2073	-0.9805j	1.5553	-0.1086
AR.2	1.2073	+0.9805j	1.5553	0.1086
MA.1	1.0000	+0.0000j	1.0000	0.0000

In [1083]:

```
plt.plot(ts_log_diff)   plt.plot(model_fit.fittedvalues,   color='red')
plt.title('RSS: %.4f'% np.nansum((model_fit.fittedvalues-ts_log_diff)**2))
```

Out[1083]: Text(0.5,1,'RSS: 1.1753')



Autoregressive Integrated Moving Average (ARIMA)

In an ARIMA model there are 3 parameters that are used to help model the major aspects of a times series: seasonality, trend, and noise. These parameters are labeled p,d,and q.

- **Number of AR (Auto-Regressive) terms (p):** p is the parameter associated with the autoregressive aspect of the model, which incorporates past values i.e lags of dependent variable.
  For instance if p is 5, the predictors for x(t) will be x(t-1)….x(t-5).
- **Number of Differences (d):** d is the parameter associated with the integrated part of the model, which effects the amount of differencing to apply to a time series.
- **Number of MA (Moving Average) terms (q):** q is size of the moving average part window of the model i.e. lagged forecast errors in prediction equation. For instance if q is 5, the predictors for x(t) will be e(t-1)….e(t-5) where e(i) is the difference between the moving average at ith instant and actual value.

Observations from EDA on the time series:

- Non stationarity implies at least one level of differencing (d) is required in ARIMA
- [The next step is to select the lag values for the Autoregression (AR) and Moving Average (MA) parameters, p and q respectively, using PACF, ACF plots (https://people.duke.edu/~rnau/411arim3.htm)](https://people.duke.edu/~rnau/411arim3.htm)

[Tuning ARIMA parameters (https://machinelearningmastery.com/tune-arima-parameters-python/)](https://machinelearningmastery.com/tune-arima-parameters-python/)

Note: A problem with ARIMA is that it does not support seasonal data. That is a time series with a repeating cycle. ARIMA expects data that is either not seasonal or has the seasonal component removed, e.g. seasonally adjusted via methods such as seasonal differencing.

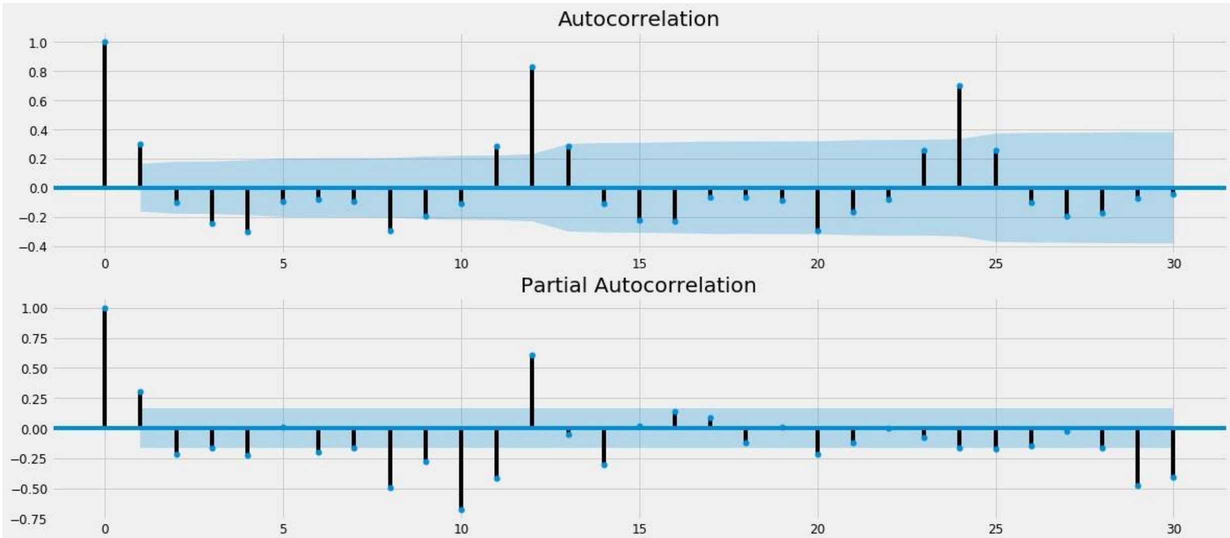
In [1084]:
```
ts = y.passengers - y.passengers.shift()
ts.dropna(inplace=True)
```

ACF and PACF plots after differencing:

- Confidence intervals are drawn as a cone.
- By default, this is set to a 95% confidence interval, suggesting that correlation values outside of this code are very likely a correlation and not a statistical fluke. AR(1) process -- has ACF
- tailing out and PACF cutting off at lag=1
- AR(2) process -- has ACF tailing out and PACF cutting off at lag=2
- MA(1) process -- has ACF cut off at lag=1
- MA(2) process -- has ACF cut off at lag=2

In [1085]:

```
pyplot.figure()
pyplot.subplot(211)
plot_acf(ts, ax=pyplot.gca(),lags=30)
pyplot.subplot(212)
plot_pacf(ts, ax=pyplot.gca(),lags=30)
pyplot.show()
```

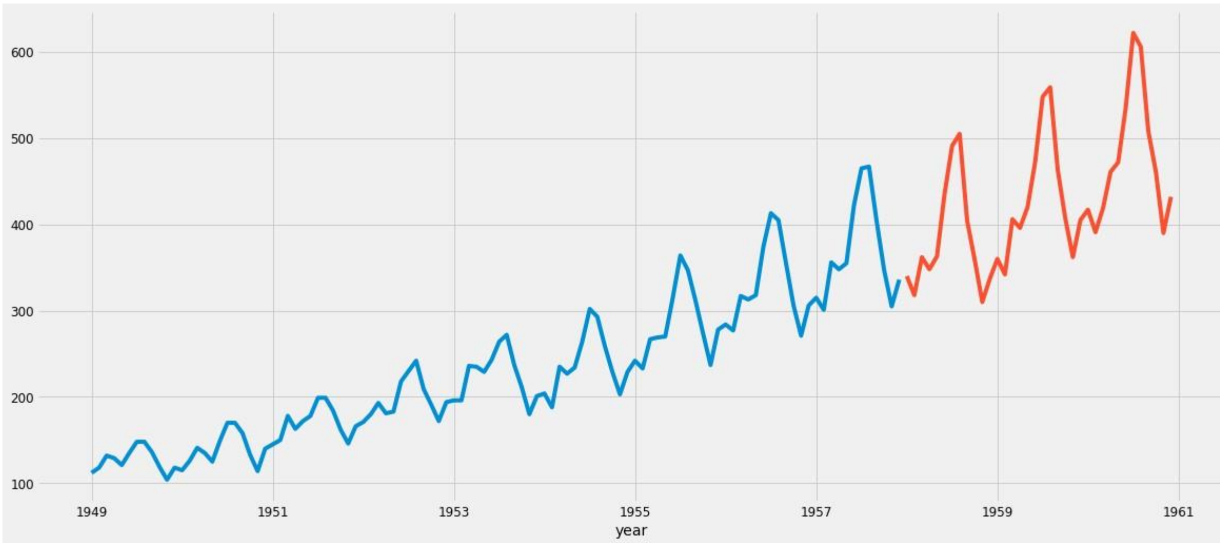


Interpreting ACF plots

ACF Shape	Indicated Model
Exponential, decaying to zero	Autoregressive model. Use the partial autocorrelation plot to identify the order of the autoregressive model
Alternating positive and negative, decaying to zero Autoregressive model.	Use the partial autocorrelation plot to help identify the order.
One or more spikes, rest are essentially zero	Moving average model, order identified by where plot becomes zero.
Decay, starting after a few lags	Mixed autoregressive and moving average (ARMA) model.
All zero or close to zero	Data are essentially random.
High values at fixed intervals	Include seasonal autoregressive term.
No decay to zero	Series is not stationary

In [1086]:
```python
#divide into train and validation set
train = y[:int(0.75*(len(y)))]
valid = y[int(0.75*(len(y))):]

#plotting the data
train['passengers'].plot()
valid['passengers'].plot()
```

Out[1086]: <matplotlib.axes._subplots.AxesSubplot at 0x27a06734f98>



In [1087]:
```python
# ARIMA example
from statsmodels.tsa.arima_model import ARIMA
from sklearn.metrics import mean_squared_error
from math import sqrt

# fit model
model = ARIMA(train, order=(1, 1, 1))
model_fit = model.fit(disp=1)
```

[1088]:
```python
model_fit.summary()
```

Out[1088]:

ARIMA Model Results

Dep. Variable:	D.passengers	**No. Observations:**	107
Model:	ARIMA(1, 1, 1)	**Log Likelihood**	-493.230
Method:	css-mle	**S.D. of innovations**	23.986
Date:	Tue, 11 Dec 2018	**AIC**	994.461
Time:	14:19:22	**BIC**	1005.152

In

Sample:	02-01-1949	**HQIC**	998.795
	- 12-01-1957		

	coef	std err	z	P>|z|	[0.025	0.975]
const	2.4356	0.265	9.186	0.000	1.916	2.955
ar.L1.D.passengers	0.7409	0.067	10.991	0.000	0.609	0.873
ma.L1.D.passengers	-1.0000	0.025	-39.435	0.000	-1.050	-0.950

Roots

	Real	Imaginary	Modulus	Frequency
AR.1	1.3496	+0.0000j	1.3496	0.0000
MA.1	1.0000	+0.0000j	1.0000	0.0000

In [1089]:

```
start_index = valid.index.min()
end_index = valid.index.max()

#Predictions
predictions = model_fit.predict(start=start_index, end=end_index)
```

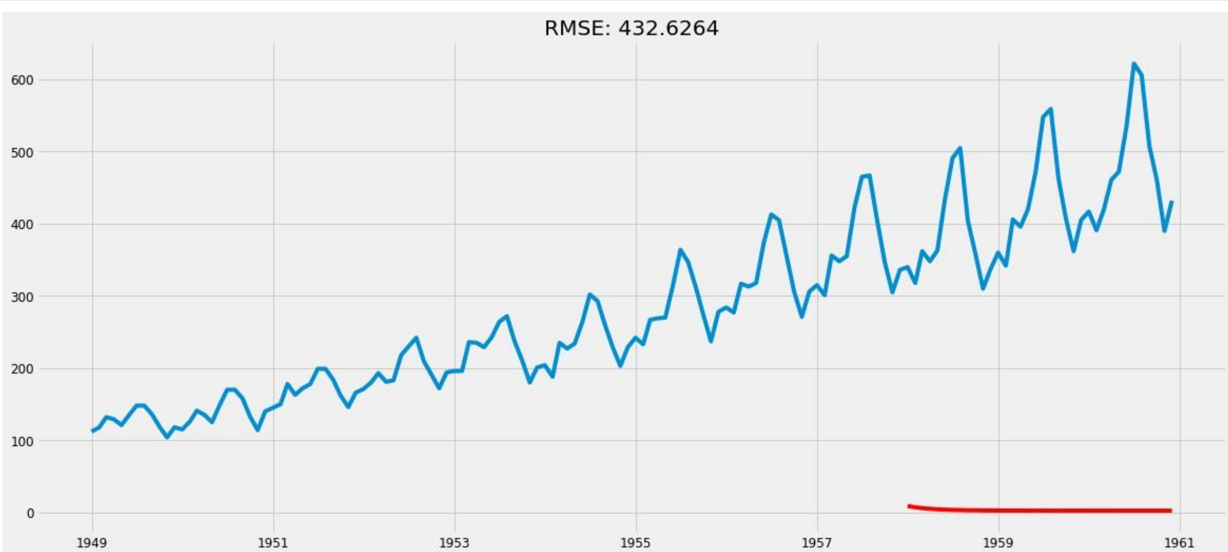
In [1090]:

```
# report performance
mse = mean_squared_error(y[start_index:end_index], predictions)
rmse = sqrt(mse)
print('RMSE: {}, MSE:{}'.format(rmse,mse))
```

```
RMSE: 432.62638025739915, MSE:187165.5848946197
```

[1091]:

```
plt.plot(y.passengers)
plt.plot(predictions, color='red')
plt.title('RMSE: %.4f'% rmse)
plt.show()
```



In

Fitted or predicted values:

In [1092]:
```
predictions_ARIMA_diff = pd.Series(predictions, copy=True)
print (predictions_ARIMA_diff.head())
```

```
1958-01-01    8.743424
1958-02-01    7.109319
1958-03-01    5.898543
1958-04-01    5.001428
1958-05-01    4.336718
Freq: MS, dtype: float64
```

Cumulative Sum to reverse differencing:

[1093]:
```
predictions_ARIMA_diff_cumsum = predictions_ARIMA_diff.cumsum()
print (predictions_ARIMA_diff_cumsum.head())
```

```
1958-01-01     8.743424
1958-02-01    15.852743
1958-03-01    21.751286
1958-04-01    26.752714
1958-05-01    31.089432
Freq: MS, dtype: float64
```

Adding 1st month value which was previously removed while differencing:

In [1094]:
```
predictions_ARIMA_log = pd.Series(valid.passengers.iloc[0], index=valid.index)
predictions_ARIMA_log =
predictions_ARIMA_log.add(predictions_ARIMA_diff_cumsum,fill_value=0)
predictions_ARIMA_log.head()
```

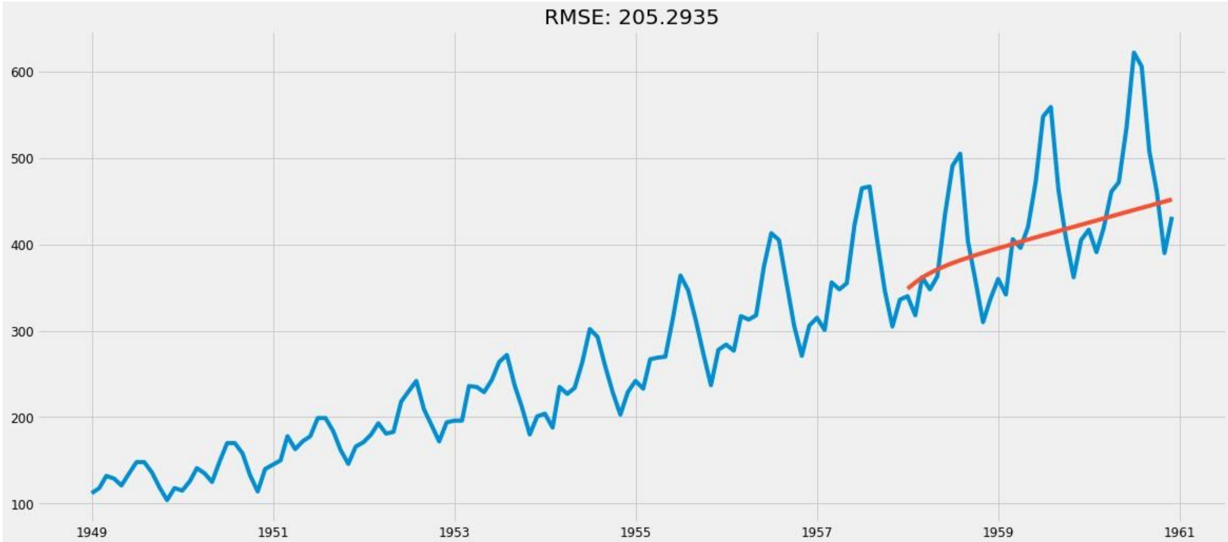
Out[1094]:
```
year
1958-01-01    348.743424
1958-02-01    355.852743
1958-03-01    361.751286
1958-04-01    366.752714 1958-
05-01    371.089432 dtype:
float64
```

Taking Exponent to reverse Log Transform:

In [1095]:
```
plt.plot(y.passengers)
plt.plot(predictions_ARIMA_log)
plt.title('RMSE: %.4f'% np.sqrt(np.nansum((predictions_ARIMA_log-
ts)**2)/len(ts)))
```

Out[1095]:
```
Text(0.5,1,'RMSE: 205.2935')
```

In



Out[1096]:

[1096]:

```
evaluate_forecast(y[start_index:end_index], predictions_ARIMA_log)
```

	r2_score	mean_absolute_error	median_absolute_error	mse	msle	mape	rmse
0	0.179865	52.106956	36.843691	5017.837103	0.023691	NaN	70.836693

Auto ARIMA

In [1097]:

```
#building the model
from pyramid.arima import auto_arima
model = auto_arima(train, trace=True, error_action='ignore',
suppress_warnings=True)
model.fit(train)
```

```
Fit ARIMA: order=(2, 1, 2) seasonal_order=(0, 0, 0, 1); AIC=959.218, BIC=975.25
5, Fit time=0.325 seconds
Fit ARIMA: order=(0, 1, 0) seasonal_order=(0, 0, 0, 1); AIC=1002.826, BIC=1008.
172, Fit time=0.009 seconds
Fit ARIMA: order=(1, 1, 0) seasonal_order=(0, 0, 0, 1); AIC=996.373, BIC=1004.3
92, Fit time=0.044 seconds
Fit ARIMA: order=(0, 1, 1) seasonal_order=(0, 0, 0, 1); AIC=991.646, BIC=999.66
4, Fit time=0.084 seconds
Fit ARIMA: order=(1, 1, 2) seasonal_order=(0, 0, 0, 1); AIC=971.486, BIC=984.85
0, Fit time=0.286 seconds
Fit ARIMA: order=(3, 1, 2) seasonal_order=(0, 0, 0, 1); AIC=966.590, BIC=985.30
0, Fit time=0.393 seconds
Fit ARIMA: order=(2, 1, 1) seasonal_order=(0, 0, 0, 1); AIC=969.040, BIC=982.40
5, Fit time=0.282 seconds
Fit ARIMA: order=(2, 1, 3) seasonal_order=(0, 0, 0, 1); AIC=nan, BIC=nan, Fit t
ime=nan seconds
Fit ARIMA: order=(1, 1, 1) seasonal_order=(0, 0, 0, 1); AIC=988.670, BIC=999.36
1, Fit time=0.108 seconds
```

In

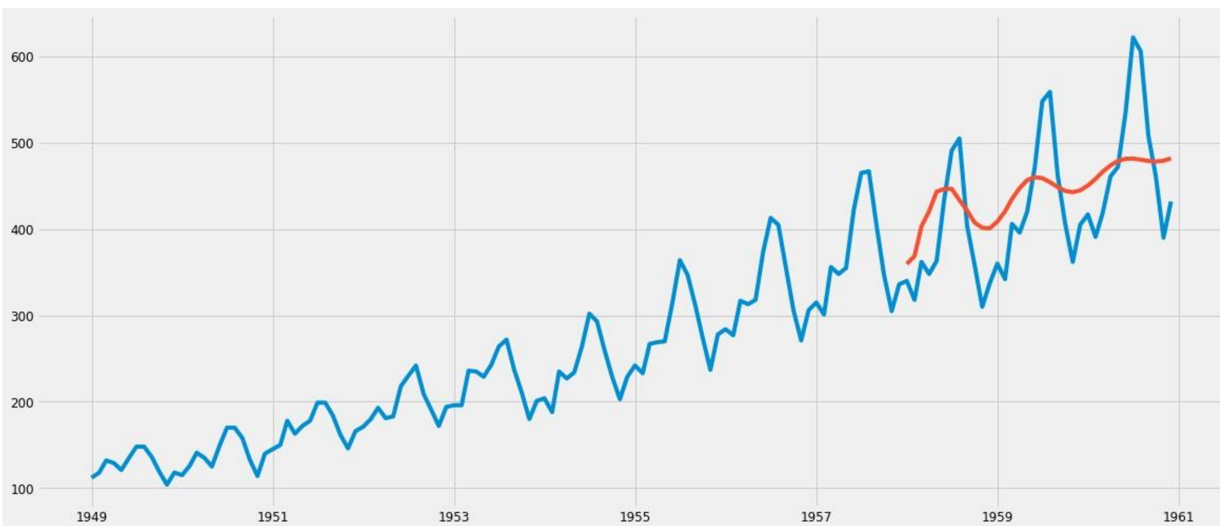
```
Fit ARIMA: order=(3, 1, 3) seasonal_order=(0, 0, 0, 1); AIC=953.638, BIC=975.02
1, Fit time=0.372 seconds
Fit ARIMA: order=(4, 1, 3) seasonal_order=(0, 0, 0, 1); AIC=964.936, BIC=988.99
2, Fit time=0.534 seconds
Fit ARIMA: order=(3, 1, 4) seasonal_order=(0, 0, 0, 1); AIC=nan, BIC=nan, Fit t
ime=nan seconds
Fit ARIMA: order=(4, 1, 4) seasonal_order=(0, 0, 0, 1); AIC=nan, BIC=nan, Fit t
ime=nan seconds
Total fit time: 2.448 seconds
```

Out[1097]:

```
ARIMA(callback=None, disp=0, maxiter=50, method=None, order=(3, 1, 3),
 out_of_sample_size=0, scoring='mse', scoring_args={},   seasonal_order=(0,
 0, 0, 1), solver='lbfgs', start_params=None,   suppress_warnings=True,
 transparams=True, trend='c')
```

[1098]:

```
forecast = model.predict(n_periods=len(valid))
forecast = pd.DataFrame(forecast,index = valid.index,columns=['Prediction'])

#plot the predictions for validation set
plt.plot(y.passengers, label='Train')
#plt.plot(valid, label='Valid')
plt.plot(forecast, label='Prediction')
plt.show()
```



Out[1099]:

In [1099]:

```
evaluate_forecast(valid, forecast)
```

	r2_score	mean_absolute_error	median_absolute_error	mse	msle	mape	rmse
0	0.369673	53.009948	48.670199	3856.531403	0.019694	NaN	62.100977

In

Seasonal Autoregressive Integrated Moving-Average (SARIMA)

Seasonal Autoregressive Integrated Moving Average, SARIMA or Seasonal ARIMA, is an extension of ARIMA that explicitly supports univariate time series data with a seasonal component.

It adds three new hyperparameters to specify the autoregression (AR), differencing (I) and moving average (MA) for the seasonal component of the series, as well as an additional parameter for the period of the seasonality.

Trend Elements:

There are three trend elements that require configuration. They are the same as the ARIMA model, specifically:

- p: Trend autoregression order.

- d: Trend difference order. q:
- Trend moving average order.

Seasonal Elements:

There are four seasonal elements that are not part of ARIMA that must be configured; they are:

- P: Seasonal autoregressive order.
- D: Seasonal difference order. Q:
- Seasonal moving average order.
- m: The number of time steps for a single seasonal period. For example, an S of 12 for monthly data suggests a yearly seasonal cycle.

SARIMA notation: SARIMA(p,d,q)(P,D,Q,m)

In [1100]:
```
# SARIMA example
from statsmodels.tsa.statespace.sarimax import SARIMAX

# fit model
model = SARIMAX(train, order=(3, 1, 3), seasonal_order=(1, 1, 1, 1))
model_fit = model.fit(disp=False)12
```

In [1101]:
```
start_index = valid.index.min()
end_index = valid.index.max()

#Predictions
predictions = model_fit.predict(start=start_index, end=end_index)
```

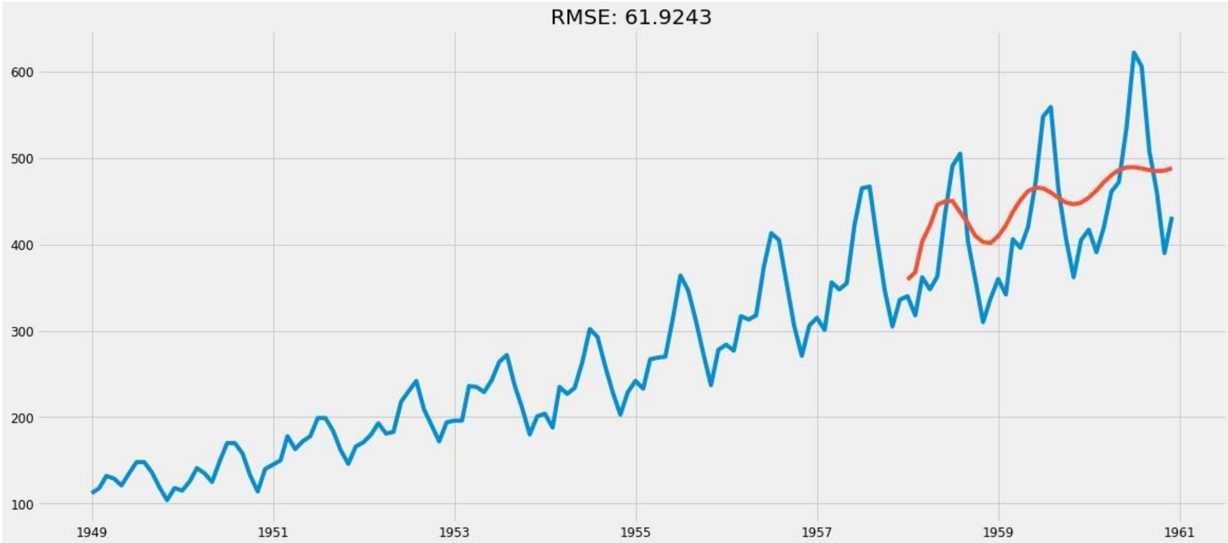
In [1102]:
```
# report performance
mse = mean_squared_error(y[start_index:end_index], predictions)
rmse = sqrt(mse)
print('RMSE: {}, MSE:{}'.format(rmse,mse))
```

```
RMSE: 61.92429713668956, MSE:3834.6185758730185
```

[1103]:
```
plt.plot(y)
plt.plot(predictions)
plt.title('RMSE: %.4f'% rmse)
```

Out[1103]:
```
Text(0.5,1,'RMSE: 61.9243')
```

In



Out[1104]:

In [1104]:

```
evaluate_forecast(y[start_index:end_index], predictions)
```

	r2_score	mean_absolute_error	median_absolute_error	mse	msle	mape	rmse
0	0.373255	53.669067	49.62041	3834.618576	0.01983	NaN	61.924297

Auto - SARIMA

[auto_arima documentation for selecting best model (https://www.alkalineml.com/pmdarima/tips_and_tricks.html)](https://www.alkalineml.com/pmdarima/tips_and_tricks.html)

[1105]:

```
#building the model from
pyramid.arima import auto_arima
model = auto_arima(train, trace=True, error_action='ignore',
suppress_warnings=True, seasonal=True, m=6, stepwise=True)
model.fit(train)
```

```
Fit ARIMA: order=(2, 1, 2) seasonal_order=(1, 0, 1, 6); AIC=938.797, BIC=960.18
0, Fit time=0.576 seconds
Fit ARIMA: order=(0, 1, 0) seasonal_order=(0, 0, 0, 6); AIC=1002.826, BIC=1008.
172, Fit time=0.013 seconds
Fit ARIMA: order=(1, 1, 0) seasonal_order=(1, 0, 0, 6); AIC=997.161, BIC=1007.8
52, Fit time=0.117 seconds
Fit ARIMA: order=(0, 1, 1) seasonal_order=(0, 0, 1, 6); AIC=993.576, BIC=1004.2
68, Fit time=0.118 seconds
Fit ARIMA: order=(2, 1, 2) seasonal_order=(0, 0, 1, 6); AIC=956.753, BIC=975.46
3, Fit time=0.466 seconds
Fit ARIMA: order=(2, 1, 2) seasonal_order=(2, 0, 1, 6); AIC=nan, BIC=nan, Fit t
ime=nan seconds
Fit ARIMA: order=(2, 1, 2) seasonal_order=(1, 0, 0, 6); AIC=974.546, BIC=993.25
6, Fit time=0.461 seconds
Fit ARIMA: order=(2, 1, 2) seasonal_order=(1, 0, 2, 6); AIC=898.251, BIC=922.30
6, Fit time=0.643 seconds
Fit ARIMA: order=(1, 1, 2) seasonal_order=(1, 0, 2, 6); AIC=899.319, BIC=920.70
2, Fit time=0.544 seconds
```

In

```
Fit ARIMA: order=(3, 1, 2) seasonal_order=(1, 0, 2, 6); AIC=897.206, BIC=923.93
4, Fit time=0.833 seconds
Fit ARIMA: order=(3, 1, 1) seasonal_order=(1, 0, 2, 6); AIC=903.528, BIC=927.58
4, Fit time=0.702 seconds
Fit ARIMA: order=(3, 1, 3) seasonal_order=(1, 0, 2, 6); AIC=893.757, BIC=923.15
8, Fit time=0.859 seconds
Fit ARIMA: order=(3, 1, 3) seasonal_order=(0, 0, 2, 6); AIC=897.446, BIC=924.17
4, Fit time=0.758 seconds
Fit ARIMA: order=(3, 1, 3) seasonal_order=(2, 0, 2, 6); AIC=nan, BIC=nan, Fit t
ime=nan seconds
Fit ARIMA: order=(3, 1, 3) seasonal_order=(1, 0, 1, 6); AIC=932.212, BIC=958.94
0, Fit time=0.667 seconds
Fit ARIMA: order=(3, 1, 3) seasonal_order=(0, 0, 1, 6); AIC=951.201, BIC=975.25
6, Fit time=0.552 seconds
Fit ARIMA: order=(2, 1, 3) seasonal_order=(1, 0, 2, 6); AIC=nan, BIC=nan, Fit t
ime=nan seconds
Fit ARIMA: order=(4, 1, 3) seasonal_order=(1, 0, 2, 6); AIC=897.087, BIC=929.16
1, Fit time=0.957 seconds
Fit ARIMA: order=(3, 1, 4) seasonal_order=(1, 0, 2, 6); AIC=nan, BIC=nan, Fit t
ime=nan seconds
Total fit time: 8.283 seconds
```

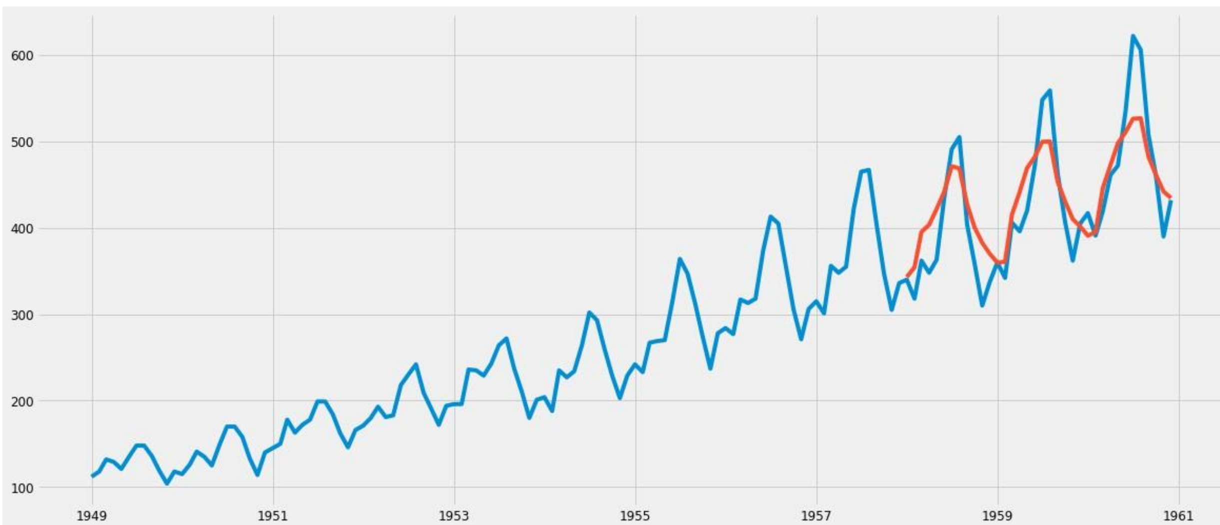
Out[1105]:

```
ARIMA(callback=None, disp=0, maxiter=50, method=None, order=(3, 1, 3),
 out_of_sample_size=0, scoring='mse', scoring_args={},   seasonal_order=(1,
 0, 2, 6), solver='lbfgs', start_params=None,   suppress_warnings=True,
 transparams=True, trend='c')
```

In [1106]:
```
start_index = valid.index.min()
end_index = valid.index.max()

#Predictions
pred = model.predict()
```

In [1107]:
```
pred = model.predict(n_periods=len(valid))
pred = pd.DataFrame(pred,index = valid.index,columns=['Prediction'])
```

In [1108]:
```
forecast = model.predict(n_periods=len(valid))
forecast = pd.DataFrame(forecast,index = valid.index,columns=['Prediction'])

#plot the predictions for validation set
plt.plot(y.passengers, label='Train')
#plt.plot(valid, label='Valid')
plt.plot(forecast, label='Prediction')
plt.show()
```



Out[1109]:

In [1109]:
```
evaluate_forecast(y[start_index:end_index], forecast)
```

	r2_score	mean_absolute_error	median_absolute_error	mse	msle	mape	rmse
0	0.750988	31.117199	26.022291	1523.531223	0.007953	NaN	39.032438

Tuned SARIMA

[1110]:
```
p = d = q = range(0, 2) pdq =
list(itertools.product(p, d, q))
seasonal_pdq = [(x[0], x[1], x[2], 6) for x in list(itertools.product(p, d, q))]
print('Examples of parameter combinations for Seasonal ARIMA...')
print('SARIMAX: {} x {}'.format(pdq[1], seasonal_pdq[1])) print('SARIMAX: {} x
{}'.format(pdq[1], seasonal_pdq[2])) print('SARIMAX: {} x {}'.format(pdq[2],
seasonal_pdq[3])) print('SARIMAX: {} x {}'.format(pdq[2], seasonal_pdq[4]))
```

```
Examples of parameter combinations for Seasonal ARIMA...
```

In

```
SARIMAX: (0, 0, 1) x (0, 0, 1, 6)
SARIMAX: (0, 0, 1) x (0, 1, 0, 6)
SARIMAX: (0, 1, 0) x (0, 1, 1, 6)
SARIMAX: (0, 1, 0) x (1, 0, 0, 6)
```

In [1111]:

```
min_aic = 999999999 for param in pdq:      for
 param_seasonal in seasonal_pdq:          try:
 mod = sm.tsa.statespace.SARIMAX(train,
 order=param,
                                            seasonal_order=param_seasonal,
 enforce_stationarity=False,
 enforce_invertibility=False)

             results = mod.fit()
             print('ARIMA{}x{}12 - AIC:{}'.format(param, param_seasonal,
 results.aic))

             #Check for best model with Lowest AIC
 if results.aic < min_aic:                    min_aic
 = results.aic                  min_aic_model =
 results          except:              continue
```

```
ARIMA(0, 0, 0)x(0, 0, 0, 6)12 - AIC:1484.6772209911371 ARIMA(0,
0, 0)x(0, 0, 1, 6)12 - AIC:1332.70817170708
ARIMA(0, 0, 0)x(0, 1, 0, 6)12 - AIC:1106.9983169558561
ARIMA(0, 0, 0)x(0, 1, 1, 6)12 - AIC:1015.2677070067782
ARIMA(0, 0, 0)x(1, 0, 0, 6)12 - AIC:1115.9461051704866
ARIMA(0, 0, 0)x(1, 0, 1, 6)12 - AIC:1001.4755946445601
ARIMA(0, 0, 0)x(1, 1, 0, 6)12 - AIC:951.0958895418044
ARIMA(0, 0, 0)x(1, 1, 1, 6)12 - AIC:860.255589360235
ARIMA(0, 0, 1)x(0, 0, 0, 6)12 - AIC:1334.2309362006272
ARIMA(0, 0, 1)x(0, 0, 1, 6)12 - AIC:1194.12573571134
ARIMA(0, 0, 1)x(0, 1, 0, 6)12 - AIC:998.4912121256906
ARIMA(0, 0, 1)x(0, 1, 1, 6)12 - AIC:912.878068945886
ARIMA(0, 0, 1)x(1, 0, 0, 6)12 - AIC:1018.9733569352445
ARIMA(0, 0, 1)x(1, 0, 1, 6)12 - AIC:914.9884746085561
ARIMA(0, 0, 1)x(1, 1, 0, 6)12 - AIC:866.3727396781308
ARIMA(0, 0, 1)x(1, 1, 1, 6)12 - AIC:792.5520247091739
ARIMA(0, 1, 0)x(0, 0, 0, 6)12 - AIC:993.1312724630138
ARIMA(0, 1, 0)x(0, 0, 1, 6)12 - AIC:943.9245123025381
ARIMA(0, 1, 0)x(0, 1, 0, 6)12 - AIC:999.3755478796342
ARIMA(0, 1, 0)x(0, 1, 1, 6)12 - AIC:853.8944017377265
ARIMA(0, 1, 0)x(1, 0, 0, 6)12 - AIC:952.2256377506902
ARIMA(0, 1, 0)x(1, 0, 1, 6)12 - AIC:904.221415093926
ARIMA(0, 1, 0)x(1, 1, 0, 6)12 - AIC:706.818028396739
ARIMA(0, 1, 0)x(1, 1, 1, 6)12 - AIC:700.9697603933553
ARIMA(0, 1, 1)x(0, 0, 0, 6)12 - AIC:973.2055693625808
ARIMA(0, 1, 1)x(0, 0, 1, 6)12 - AIC:924.7899877818263
ARIMA(0, 1, 1)x(0, 1, 0, 6)12 - AIC:979.0951064213158
ARIMA(0, 1, 1)x(0, 1, 1, 6)12 - AIC:837.3373300218982
ARIMA(0, 1, 1)x(1, 0, 0, 6)12 - AIC:941.4721477228281
ARIMA(0, 1, 1)x(1, 0, 1, 6)12 - AIC:886.8511031802221
ARIMA(0, 1, 1)x(1, 1, 0, 6)12 - AIC:698.3539185545628
ARIMA(0, 1, 1)x(1, 1, 1, 6)12 - AIC:682.4714093404433
ARIMA(1, 0, 0)x(0, 0, 0, 6)12 - AIC:1003.4820392779112 ARIMA(1,
0, 0)x(0, 0, 1, 6)12 - AIC:954.390876653235
ARIMA(1, 0, 0)x(0, 1, 0, 6)12 - AIC:1001.1995622331247
```

In

```
ARIMA(1, 0, 0)x(0, 1, 1, 6)12 - AIC:862.1556791545082
ARIMA(1, 0, 0)x(1, 0, 0, 6)12 - AIC:954.2184484398757
ARIMA(1, 0, 0)x(1, 0, 1, 6)12 - AIC:905.0278354938664
ARIMA(1, 0, 0)x(1, 1, 0, 6)12 - AIC:707.3904075303261
ARIMA(1, 0, 0)x(1, 1, 1, 6)12 - AIC:707.4979558719868
ARIMA(1, 0, 1)x(0, 0, 0, 6)12 - AIC:983.8745153729881
ARIMA(1, 0, 1)x(0, 0, 1, 6)12 - AIC:935.4856179230443
ARIMA(1, 0, 1)x(0, 1, 0, 6)12 - AIC:976.5242009160917
ARIMA(1, 0, 1)x(0, 1, 1, 6)12 - AIC:844.833173577821
ARIMA(1, 0, 1)x(1, 0, 0, 6)12 - AIC:943.501939269728
ARIMA(1, 0, 1)x(1, 0, 1, 6)12 - AIC:884.0817229227741
ARIMA(1, 0, 1)x(1, 1, 0, 6)12 - AIC:700.1612840320383
ARIMA(1, 0, 1)x(1, 1, 1, 6)12 - AIC:690.5229323222985
ARIMA(1, 1, 0)x(0, 0, 0, 6)12 - AIC:986.4207435070009
ARIMA(1, 1, 0)x(0, 0, 1, 6)12 - AIC:937.2676970229218
ARIMA(1, 1, 0)x(0, 1, 0, 6)12 - AIC:985.3821728074678 ARIMA(1, 1, 0)x(0, 1,
1, 6)12 - AIC:844.059025440482
ARIMA(1, 1, 0)x(1, 0, 0, 6)12 - AIC:936.3388643486377
ARIMA(1, 1, 0)x(1, 0, 1, 6)12 - AIC:893.6268860673151 ARIMA(1, 1, 0)x(1, 1,
0, 6)12 - AIC:692.553752220819
ARIMA(1, 1, 0)x(1, 1, 1, 6)12 - AIC:690.3400925924694
ARIMA(1, 1, 1)x(0, 0, 0, 6)12 - AIC:970.8380287106427
ARIMA(1, 1, 1)x(0, 0, 1, 6)12 - AIC:922.0887630236411
ARIMA(1, 1, 1)x(0, 1, 0, 6)12 - AIC:977.8777245169276
ARIMA(1, 1, 1)x(0, 1, 1, 6)12 - AIC:840.6216450674011 ARIMA(1, 1, 1)x(1, 0,
0, 6)12 - AIC:930.637813366519
ARIMA(1, 1, 1)x(1, 0, 1, 6)12 - AIC:886.2198394847763
ARIMA(1, 1, 1)x(1, 1, 0, 6)12 - AIC:693.0569052511208
ARIMA(1, 1, 1)x(1, 1, 1, 6)12 - AIC:684.3818335634074
```

[1112]:

```
min_aic_model.summary()
```

Out[1112]:

Statespace Model Results

Dep. Variable:	passengers	**No. Observations:**	108
Model:	SARIMAX(0, 1, 1)x(1, 1, 1, 6)	**Log Likelihood**	-337.236
Date:	Tue, 11 Dec 2018	**AIC**	682.471
Time:	14:19:43	**BIC**	692.602
Sample:	01-01-1949	**HQIC**	686.562
	- 12-01-1957		
Covariance Type:	opg		

	coef	**std err**	**z**	**P>|z|**	**[0.025**	**0.975]**
ma.L1	-0.4552	0.082	-5.543	0.000	-0.616	-0.294
ar.S.L6	-1.0894	0.017	-65.219	0.000	-1.122	-1.057
ma.S.L6	2.2735	0.600	3.792	0.000	1.098	3.449
sigma2	15.8272	7.949	1.991	0.046	0.248	31.406

Ljung-Box (Q):	45.44	**Jarque-Bera (JB):**	2.17

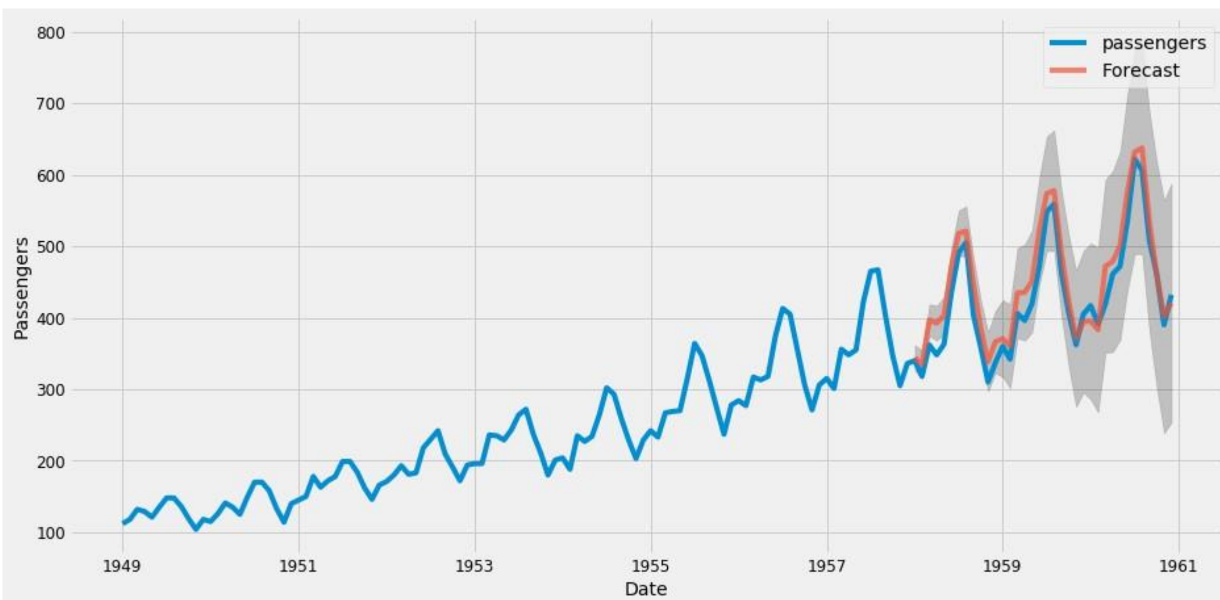
Prob(Q):	0.26	**Prob(JB):**	0.34
Heteroskedasticity (H):	0.61	**Skew:**	0.37
Prob(H) (two-sided):	0.17	**Kurtosis:**	2.99

Warnings:

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

In [1113]:
```
start_index = valid.index.min()
end_index = valid.index.max()

#Predictions
pred = min_aic_model.get_prediction(start=start_index,end=end_index,
dynamic=False)
```

In [1114]:
```
pred_ci = pred.conf_int()
ax = y['1949':].plot(label='observed')
pred.predicted_mean.plot(ax=ax, label='Forecast', alpha=.7, figsize=(14, 7))
ax.fill_between(pred_ci.index,
                pred_ci.iloc[:, 0],
                pred_ci.iloc[:, 1], color='k', alpha=.2)
ax.set_xlabel('Date')
ax.set_ylabel('Passengers')
plt.legend()
plt.show()
```



In

Model diagnostics:

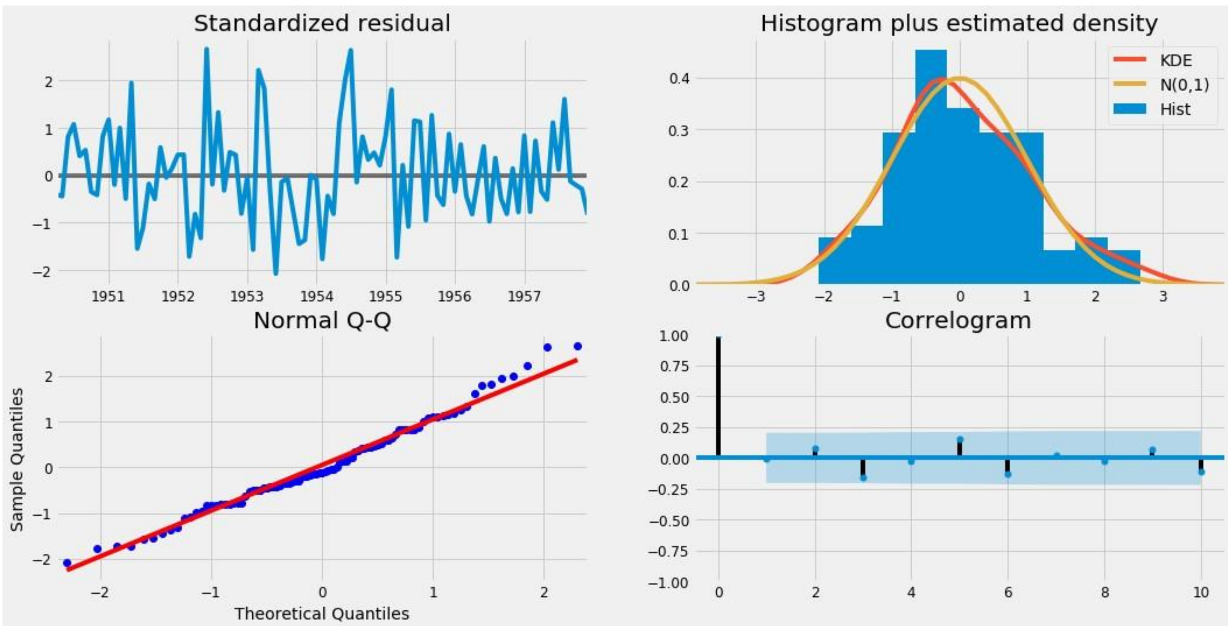
- Our primary concern is to ensure that the residuals of our model are uncorrelated and normally distributed with zero-mean.
- If the seasonal ARIMA model does not satisfy these properties, it is a good indication that it can be further improved.

The model diagnostic suggests that the model residual is normally distributed based on the following:

- In the top right plot, the red KDE line follows closely with the N(0,1) line. Where, N(0,1) is the standard notation for a normal distribution with mean 0 and standard deviation of 1. This is a good indication that the residuals are normally distributed.

- The qq-plot on the bottom left shows that the ordered distribution of residuals (blue dots) follows the linear trend of the samples taken from a standard normal distribution. Again, this is a strong indication that the residuals are normally distributed.
- The residuals over time (top left plot) don't display any obvious seasonality and appear to be white noise.
- This is confirmed by the autocorrelation (i.e. correlogram) plot on the bottom right, which shows that the time series residuals have low correlation with lagged versions of itself.

In [1115]:
```
results.plot_diagnostics(figsize=(16, 8))
plt.show()
```



In [1116]:
```
y_forecasted = pred.predicted_mean.values y_truth =
y[start_index:end_index].passengers.values mse = ((y_forecasted - y_truth)
** 2).mean() print('The Mean Squared Error of our forecasts is
{}'.format(round(mse, 2)))
```

```
The Mean Squared Error of our forecasts is 781.5
```

In [1117]:
```
print('The Root Mean Squared Error of our forecasts is
{}'.format(round(np.sqrt(mse), 2)))
```

```
The Root Mean Squared Error of our forecasts is 27.96
```

Out[1118]:

In [1118]:
```
evaluate_forecast(y_truth, y_forecasted)
```

	r2_score	mean_absolute_error	median_absolute_error	mse	msle	mape	rmse
0	0.872268	24.733402	26.185186	781.50462	0.004216	5.932291	27.955404

In

SARIMAX

- The implementation is called SARIMAX instead of SARIMA because the “X” addition to the method name means that the implementation also supports exogenous variables.
- Exogenous variables are optional can be specified via the “exog” argument.
  - model = SARIMAX(data, exog=other_data, ...)
- Examples of exogenous variables: Population, holidays, number of airline companies, major events

Prophet

- [Prophet (https://facebook.github.io/prophet/)](https://facebook.github.io/prophet/) is open source software released by Facebook's Core Data Science team.
- Prophet is a procedure for forecasting time series data based on an additive/multiplicative model where non-linear trends are fit with yearly, weekly, and daily seasonality, plus holiday effects.
- It works best with time series that have strong seasonal effects and several seasons of historical data.
- Prophet is robust to missing data and shifts in the trend, and typically handles outliers well.
- The Prophet package provides intuitive parameters which are easy to tune.

[Prophet example notebooks (https://github.com/facebook/prophet/tree/master/notebooks)](https://github.com/facebook/prophet/tree/master/notebooks)

Trend parameters

- growth: 'linear' or 'logistic' to specify a linear or logistic trend changepoints: List of dates at
- which to include potential changepoints (automatic if not specified) n_changepoints: If changepoints in not supplied, you may provide the number of changepoints to be
- automatically included changepoint_prior_scale: Parameter for changing flexibility of automatic changepoint selection
-

Seasonality and Holiday Parameters

- yearly_seasonality: Fit yearly seasonality weekly_seasonality: Fit weekly
- seasonality daily_seasonality: Fit daily seasonality holidays: Feed dataframe
- containing holiday name and date seasonality_prior_scale: Parameter for
- changing strength of seasonality model holiday_prior_scale: Parameter for
- changing strength of holiday model
-

Prophet requires the variable names in the time series to be:

- y – Target ds
- – Datetime

[1119]:
```
train.head()
```

Out[1119]:

	passengers
year	
1949-01-01	112
1949-02-01	118
1949-03-01	132
1949-04-01	129
1949-05-01	121

In [1120]:

```
train_prophet = pd.DataFrame()
train_prophet['ds'] = train.index
train_prophet['y'] = train.passengers.values
```

In [1121]:

```
train_prophet.head()
```

Out[1121]:

	ds	y
0	1949-01-01	112
1	1949-02-01	118
2	1949-03-01	132
3	1949-04-01	129
4	1949-05-01	121

In [1122]:

```
from fbprophet import Prophet

#instantiate Prophet with only yearly seasonality as our data is monthly model
= Prophet( yearly_seasonality=True, seasonality_mode = 'multiplicative')
model.fit(train_prophet) #fit the model with your dataframe
```

```
INFO:fbprophet.forecaster:Disabling weekly seasonality. Run prophet with weekly
_seasonality=True to override this.
INFO:fbprophet.forecaster:Disabling daily seasonality. Run prophet with daily_s
easonality=True to override this.
```

Out[1122]: <fbprophet.forecaster.Prophet at 0x27a09311c18>

In [1123]:

```python
# predict for five months in the furure and MS - month start is the frequency
future = model.make_future_dataframe(periods = 36, freq = 'MS')
future.tail()
```

Out[1123]:

	ds
139	1960-08-01
140	1960-09-01
141	1960-10-01
142	1960-11-01
143	1960-12-01

In [1124]:

```python
forecast.columns
```

Out[1124]:

```
Index(['Prediction'], dtype='object')
```

In [1125]:

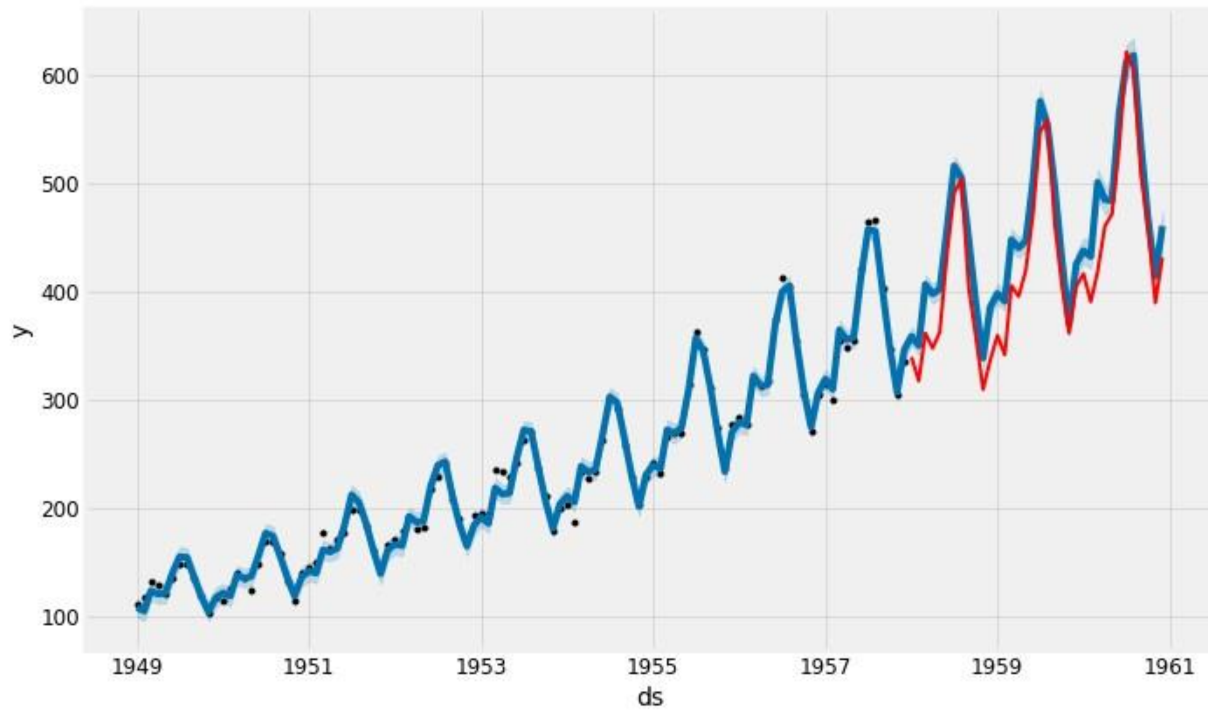
```python
# now lets make the forecasts
forecast = model.predict(future)
forecast[['ds', 'yhat', 'yhat_lower', 'yhat_upper']].tail()
```

Out[1125]:

	ds	yhat	yhat_lower	yhat_upper
139	1960-08-01	618.747937	602.074096	634.827297
140	1960-09-01	535.714501	519.552416	550.998934
141	1960-10-01	467.354249	452.303994	482.693898
142	1960-11-01	414.836968	400.967561	428.114647
143	1960-12-01	461.106174	446.181466	475.608465

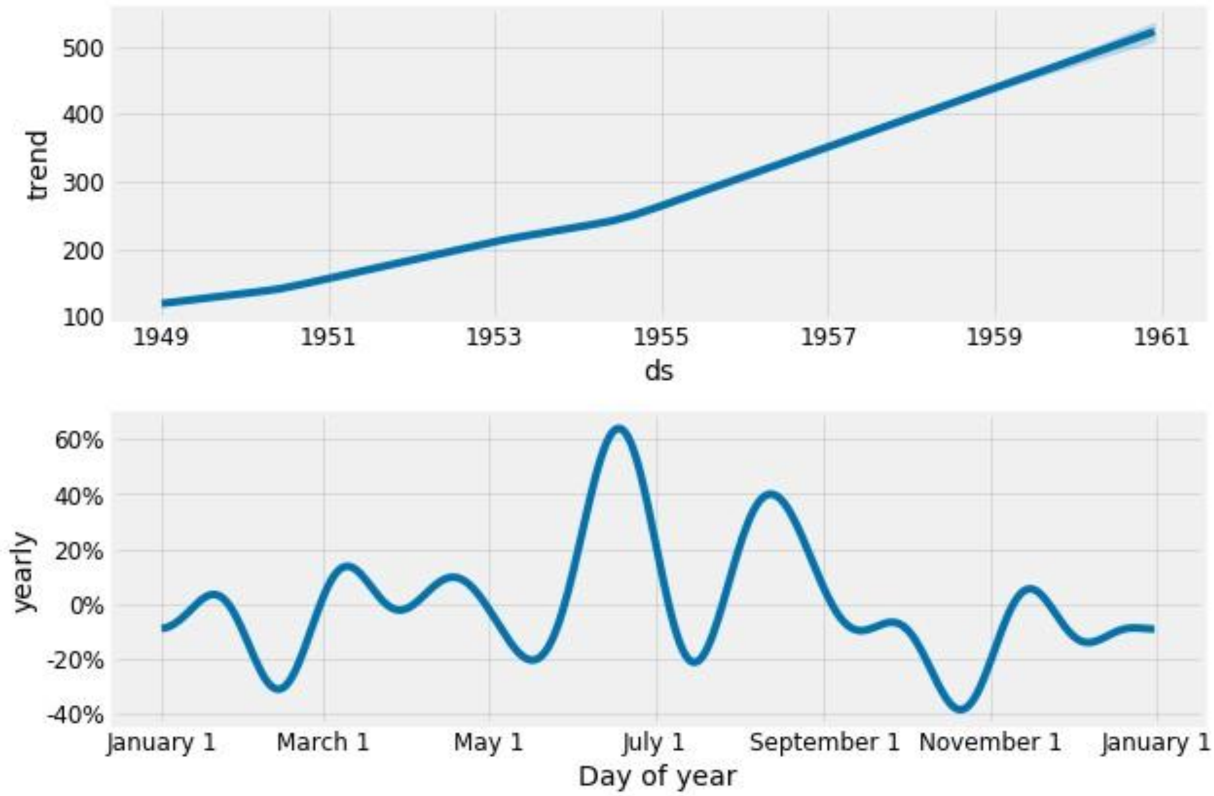
In [1126]:

```
fig = model.plot(forecast)
#plot the predictions for validation set

plt.plot(valid, label='Valid', color = 'red', linewidth = 2)

plt.show()
```



In

In [1127]:
```
model.plot_components(forecast);
```



Out[1130]:

[1128]:
```
y_prophet = pd.DataFrame()
y_prophet['ds'] = y.index
y_prophet['y'] = y.passengers.values
```

In [1129]:
```
y_prophet = y_prophet.set_index('ds')
forecast_prophet = forecast.set_index('ds')
```

In [1130]:
```
evaluate_forecast(y_prophet.y[start_index:end_index],
forecast_prophet.yhat[start_index:end_index])
```

	r2_score	mean_absolute_error	median_absolute_error	mse	msle	mape	rms
0	0.814637	29.69586	28.692342	1134.108291	0.006653	7.440315	33.67652

Improving Time Series Forecast models

1. Hyperparamter Optimization: Finding the optimal parameters of ARIMA/Prophet models.
2. Exogenous variables (SARIMAX): Including external variables like campaigns, holidays, events, natural calamities etc.
3. [Combining models for advanced time series predictions]()

In

(https://www.kdnuggets.com/2016/11/combining-different-methods-create-advanced-timeseries-prediction.html)

4. Long Short Term Memory Network (LSTM) (https://machinelearningmastery.com/time-seriesprediction-lstm-recurrent-neural-networks-python-keras/)

Solve a problem!

Store Item Demand Forecasting Challenge: https://www.kaggle.com/c/demand-forecastingkernels-only (https://www.kaggle.com/c/demand-forecasting-kernels-only)

- This competition is provided as a way to explore different time series techniques on a relatively simple and clean dataset.
- You are given 5 years of store-item sales data, and asked to predict 3 months of sales for 50 different items at 10 different stores.
- What's the best way to deal with seasonality? Should stores be modeled separately, or can you pool them together? Does deep learning work better than ARIMA? Can either beat xgboost?