Time Series Analysis

* **Seasonal Plot of a Time Series**

# Prepare data

df['year'] = [d.year for d in df.date]

df['month'] = [d.strftime('%b') for d in df.date]

years = df['year'].unique()

# Prep Colors

np.random.seed(100)

mycolors = np.random.choice(list(mpl.colors.XKCD\_COLORS.keys()), len(years), replace=False)

# Draw Plot

plt.figure(figsize=(16,12), dpi= 80)

for i, y in enumerate(years):

if i > 0:

plt.plot('month', 'value', data=df.loc[df.year==y, :], color=mycolors[i], label=y)

plt.text(df.loc[df.year==y, :].shape[0]-.9, df.loc[df.year==y, 'value'][-1:].values[0], y,

fontsize=12, color=mycolors[i])

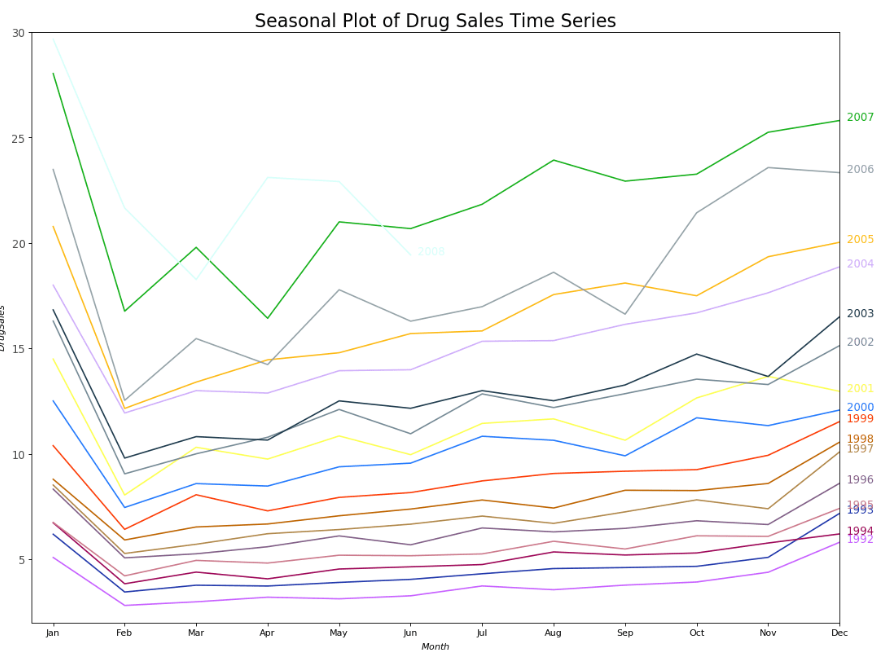
# Decoration

plt.gca().set(xlim=(-0.3, 11), ylim=(2, 30), ylabel='$Drug Sales$', xlabel='$Month$')

plt.yticks(fontsize=12, alpha=.7)

plt.title("Seasonal Plot of Drug Sales Time Series", fontsize=20)

plt.show()



* Decomposition of Time Series

from statsmodels.tsa.seasonal import seasonal\_decompose

from dateutil.parser import parse

# Multiplicative Decomposition

result\_mul = seasonal\_decompose(df['value'], model='multiplicative', extrapolate\_trend='freq')

# Additive Decomposition

result\_add = seasonal\_decompose(df['value'], model='additive', extrapolate\_trend='freq')

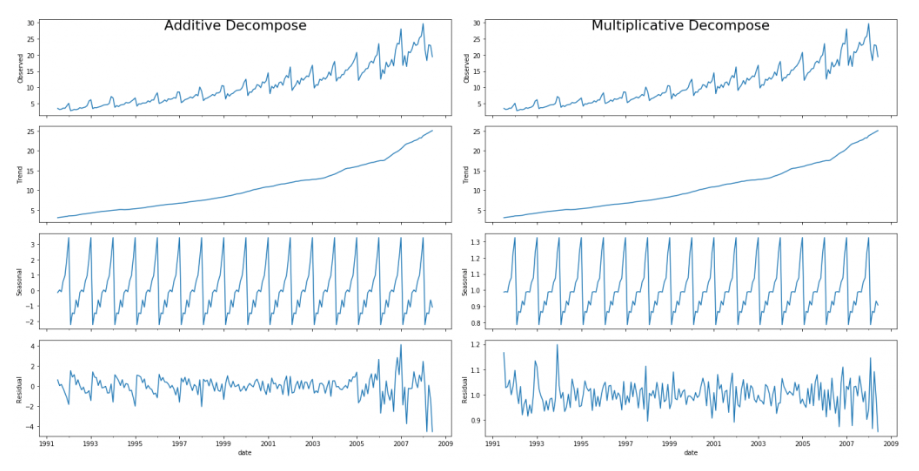
# Plot

plt.rcParams.update({'figure.figsize': (10,10)})

result\_mul.plot().suptitle('Multiplicative Decompose', fontsize=22)

result\_add.plot().suptitle('Additive Decompose', fontsize=22)

plt.show()



Note:- Setting ***extrapolate\_trend='freq'*** takes care of any missing values in the trend and residuals at the beginning of the series.

# Store the results in a DataFrame

*(Actual Values = Seasonal \* Trend \* Residual)*

df\_reconstructed = pd.concat([result\_mul.seasonal, result\_mul.trend, result\_mul.resid, result\_mul.observed], axis=1)

df\_reconstructed.columns = ['seas', 'trend', 'resid', 'actual\_values']

df\_reconstructed.head()

* Stationarity Test

from statsmodels.tsa.stattools import adfuller, kpss

# ADF Test

result = adfuller(df.value.values, autolag='AIC')

print(f'ADF Statistic: {result[0]}')

print(f'p-value: {result[1]}')

for key, value in result[4].items():

print('Critial Values:')

print(f' {key}, {value}')

# KPSS Test

result = kpss(df.value.values, regression='c')

print('\nKPSS Statistic: %f' % result[0])

print('p-value: %f' % result[1])

for key, value in result[3].items():

print('Critial Values:')

print(f' {key}, {value}')

* Detrend a Time Series

1. Subtract the line of best fit from the time series. The line of best fit may be obtained from a linear regression model with the time steps as the predictor. For more complex trends, you may want to use quadratic terms (x^2) in the model.
2. Subtract the trend component obtained from time series decomposition we saw earlier.
3. Subtract the mean
4. Apply a filter like Baxter-King filter(statsmodels.tsa.filters.bkfilter) or the Hodrick-Prescott Filter (statsmodels.tsa.filters.hpfilter) to remove the moving average trend lines or the cyclical components.

# Using scipy: Subtract the line of best fit

from scipy import signal

detrended = signal.detrend(df.value.values)

plt.plot(detrended)

plt.title('Drug Sales detrended by subtracting the least squares fit', fontsize=16)

# Using statmodels: Subtracting the Trend Component.

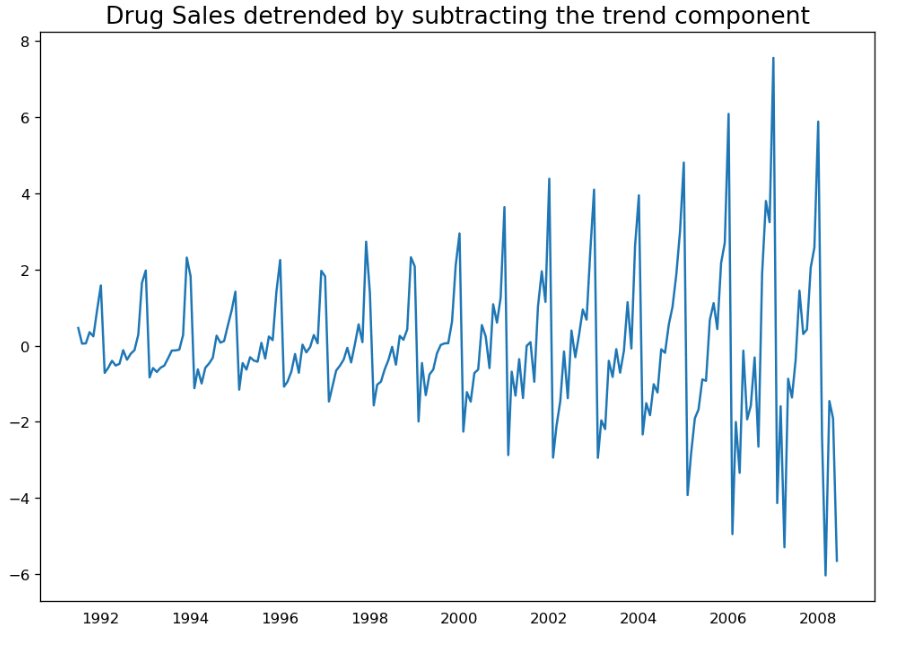
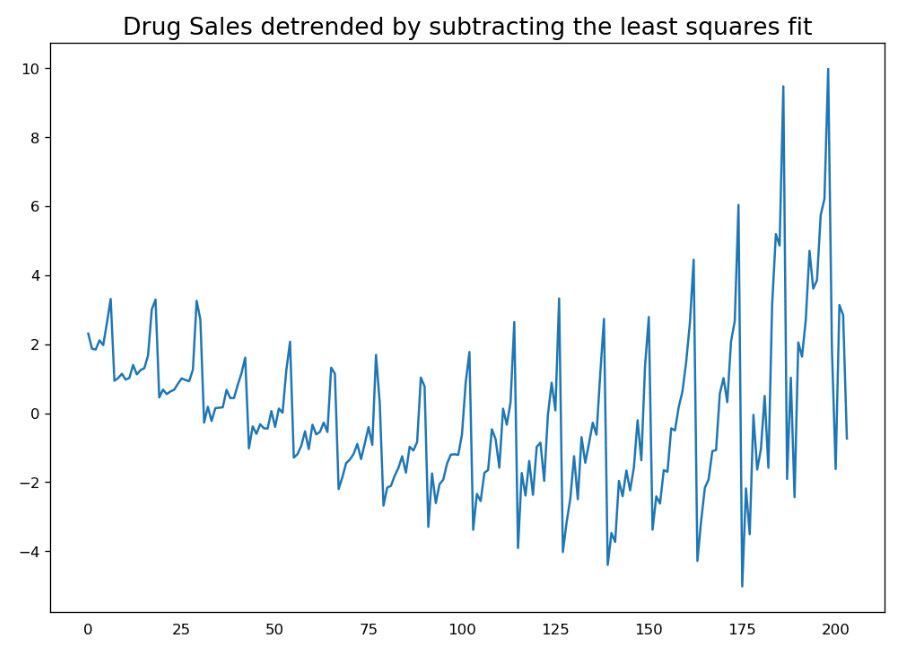
from statsmodels.tsa.seasonal import seasonal\_decompose

result\_mul = seasonal\_decompose(df['value'], model='multiplicative', extrapolate\_trend='freq')

detrended = df.value.values - result\_mul.trend

plt.plot(detrended)

plt.title('Drug Sales detrended by subtracting the trend component', fontsize=16)



* Deseasonalise Time Series

1. Take a moving average with length as the seasonal window. This will smoothen in series in the process.

2. Seasonal difference the series (subtract the value of previous season from the current value)

3. Divide the series by the seasonal index obtained from STL decomposition

# Time Series Decomposition

result\_mul = seasonal\_decompose(df['value'], model='multiplicative', extrapolate\_trend='freq')

# Deseasonalize

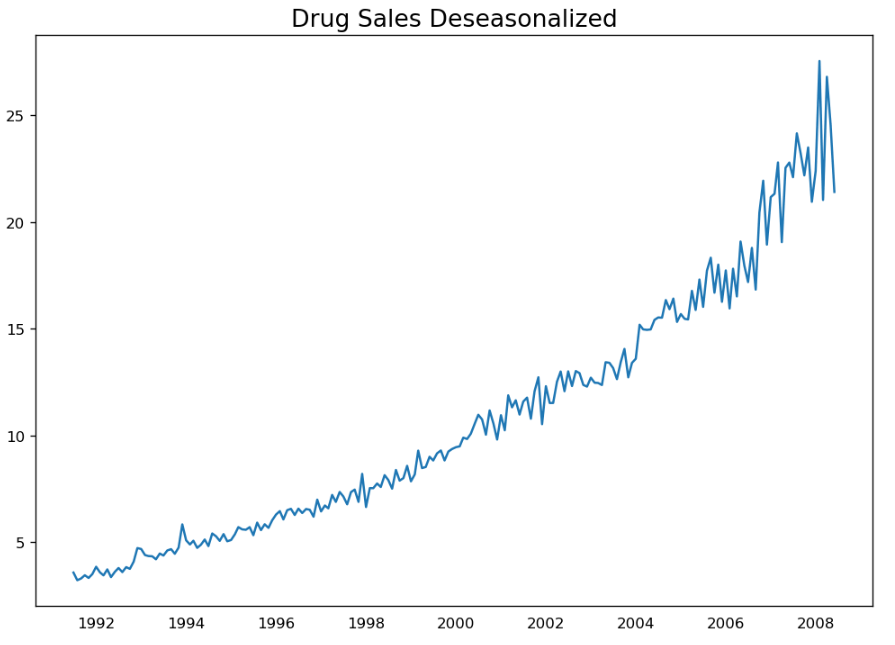
deseasonalized = df.value.values / result\_mul.seasonal

# Plot

plt.plot(deseasonalized)

plt.title('Drug Sales Deseasonalized', fontsize=16)

plt.plot()



* Treating missing values in a time series

Some effective alternatives to imputation are:

1. Backward Fill
2. Linear Interpolation
3. Quadratic interpolation
4. Mean of nearest neighbors
5. Mean of seasonal couterparts

from scipy.interpolate import interp1d

from sklearn.metrics import mean\_squared\_error

## 1. Actual -------------------------------

df\_orig.plot(title='Actual', ax=axes[0], label='Actual', color='red', style=".-")

df.plot(title='Actual', ax=axes[0], label='Actual', color='green', style=".-")

axes[0].legend(["Missing Data", "Available Data"])

## 2. Forward Fill --------------------------

df\_ffill = df.ffill()

error = np.round(mean\_squared\_error(df\_orig['value'], df\_ffill['value']), 2)

df\_ffill['value'].plot(title='Forward Fill (MSE: ' + str(error) +")", ax=axes[1], label='Forward Fill', style=".-")

## 3. Backward Fill -------------------------

df\_bfill = df.bfill()

error = np.round(mean\_squared\_error(df\_orig['value'], df\_bfill['value']), 2)

df\_bfill['value'].plot(title="Backward Fill (MSE: " + str(error) +")", ax=axes[2], label='Back Fill', color='firebrick', style=".-")

## 4. Linear Interpolation ------------------

df['rownum'] = np.arange(df.shape[0])

df\_nona = df.dropna(subset = ['value'])

f = interp1d(df\_nona['rownum'], df\_nona['value'])

df['linear\_fill'] = f(df['rownum'])

error = np.round(mean\_squared\_error(df\_orig['value'], df['linear\_fill']), 2)

df['linear\_fill'].plot(title="Linear Fill (MSE: " + str(error) +")", ax=axes[3], label='Cubic Fill', color='brown', style=".-")

## 5. Cubic Interpolation --------------------

f2 = interp1d(df\_nona['rownum'], df\_nona['value'], kind='cubic')

df['cubic\_fill'] = f2(df['rownum'])

error = np.round(mean\_squared\_error(df\_orig['value'], df['cubic\_fill']), 2)

df['cubic\_fill'].plot(title="Cubic Fill (MSE: " + str(error) +")", ax=axes[4], label='Cubic Fill', color='red', style=".-")

## 6. Mean of 'n' Nearest Past Neighbors ------

def knn\_mean(ts, n):

out = np.copy(ts)

for i, val in enumerate(ts):

if np.isnan(val):

n\_by\_2 = np.ceil(n/2)

lower = np.max([0, int(i-n\_by\_2)])

upper = np.min([len(ts)+1, int(i+n\_by\_2)])

ts\_near = np.concatenate([ts[lower:i], ts[i:upper]])

out[i] = np.nanmean(ts\_near)

return out

df['knn\_mean'] = knn\_mean(df.value.values, 8)

error = np.round(mean\_squared\_error(df\_orig['value'], df['knn\_mean']), 2)

df['knn\_mean'].plot(title="KNN Mean (MSE: " + str(error) +")", ax=axes[5], label='KNN Mean', color='tomato', alpha=0.5, style=".-")

## 7. Seasonal Mean ----------------------------

def seasonal\_mean(ts, n, lr=0.7):

out = np.copy(ts)

for i, val in enumerate(ts):

if np.isnan(val):

ts\_seas = ts[i-1::-n] # previous seasons only

if np.isnan(np.nanmean(ts\_seas)):

ts\_seas = np.concatenate([ts[i-1::-n], ts[i::n]]) # previous and forward

out[i] = np.nanmean(ts\_seas) \* lr

return out

df['seasonal\_mean'] = seasonal\_mean(df.value, n=12, lr=1.25)

error = np.round(mean\_squared\_error(df\_orig['value'], df['seasonal\_mean']), 2)

df['seasonal\_mean'].plot(title="Seasonal Mean (MSE: " + str(error) +")", ax=axes[6], label='Seasonal Mean', color='blue', alpha=0.5, style=".-")

* Autocorrelation and Partial Autocorrelation functions

from statsmodels.tsa.stattools import acf, pacf

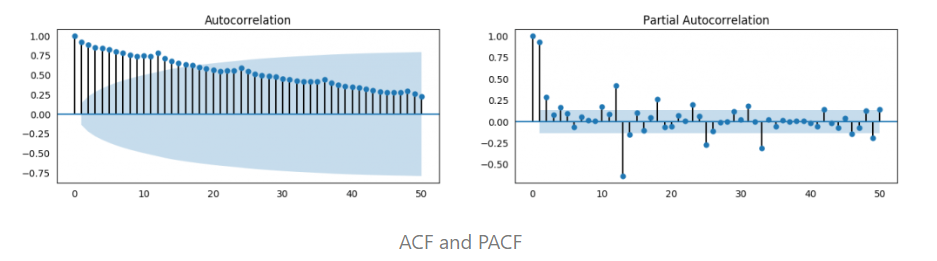
from statsmodels.graphics.tsaplots import plot\_acf, plot\_pacf

# Draw Plot

fig, axes = plt.subplots(1,2,figsize=(16,3), dpi= 100)

plot\_acf(df.value.tolist(), lags=50, ax=axes[0])

plot\_pacf(df.value.tolist(), lags=50, ax=axes[1])



* Lag Plots

A Lag plot is a scatter plot of a time series against a lag of itself. It is normally used to check for autocorrelation. If there is any pattern existing in the series like the one you see below, the series is autocorrelated. If there is no such pattern, the series is likely to be random white noise.

from pandas.plotting import lag\_plot

plt.rcParams.update({'ytick.left' : False, 'axes.titlepad':10})

# Plot

fig, axes = plt.subplots(1, 4, figsize=(10,3), sharex=True, sharey=True, dpi=100)

for i, ax in enumerate(axes.flatten()[:4]):

lag\_plot(df.value, lag=i+1, ax=ax, c='firebrick')

ax.set\_title('Lag ' + str(i+1))

fig.suptitle('title1',y=1.05)

plt.show()



* Smoothening a time series

Different Smoothening techniques that may be useful:

1. Moving Average
2. LoESS Smoothing
3. Simple Exponential Smoothing
4. Holt Exponential Smoothing
5. HoltWinter Exponential Smoothing

from statsmodels.nonparametric.smoothers\_lowess import lowess

from statsmodels.tsa.holtwinters import SimpleExpSmoothing

from statsmodels.tsa.holtwinters import ExponentialSmoothing

plt.rcParams.update({'xtick.bottom' : False, 'axes.titlepad':5})

# 1. Moving Average

df\_ma = df\_orig.value.rolling(3, center=True, closed='both').mean()

# 2. LoESS Smoothing (5% and 15%)

df\_loess\_5 = pd.DataFrame(lowess(df\_orig.value, np.arange(len(df\_orig.value)), frac=0.05)[:, 1], index=df\_orig.index, columns=['value'])

df\_loess\_15 = pd.DataFrame(lowess(df\_orig.value, np.arange(len(df\_orig.value)), frac=0.15)[:, 1], index=df\_orig.index, columns=['value'])

# 3. Simple Exponential Smoothing

model = SimpleExpSmoothing(df.value)

model\_fit = model.fit(smoothing\_level=0.2)

df\_simpleexp[‘value’] = pd.DataFrame(model\_fit.forecast(24)]

# 4. Holt Exponential Smoothing

model = ExponentialSmoothing(df.value, seasonal\_period = 12, trend = ‘multiplicative’)

model\_fit = model.fit(smoothing\_level = 0.2, smoothing\_slope = 0.04)

df\_holt[‘value’] = pd.DataFrame(model\_fit.forecast(24))

# 5. HoltWinter Exponential Smoothing

model = ExponentialSmoothing(df.value, seasonal\_period = 12, trend = ‘multiplicative’, seasonal = ‘additive’)

model\_fit = model.fit(smoothing\_level = 0.2, smoothing\_slope = 0.04)

df\_holtwinter[‘value’] = pd.DataFrame(model\_fit.forecast(24))

# Plot

fig, axes = plt.subplots(7,1, figsize=(7, 7), sharex=True, dpi=120)

df\_orig['value'].plot(ax=axes[0], color='k', title='Original Series')

df\_loess\_5['value'].plot(ax=axes[1], title='Loess Smoothed 5%')

df\_loess\_15['value'].plot(ax=axes[2], title='Loess Smoothed 15%')

df\_ma.plot(ax=axes[3], title='Moving Average (3)')

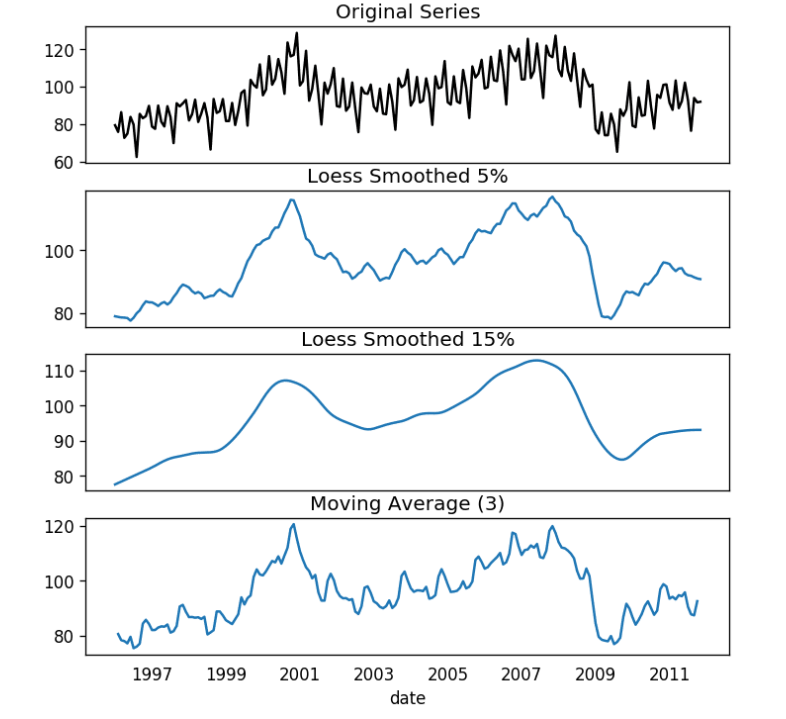
df\_simpleexp [‘value’] .plot(ax=axes[4], title=’Simple Exponential Smoothing')

df\_holt [‘value’] .plot(ax=axes[5], title=’Holt Exponential Smoothing')

df\_holtwinter [‘value’] .plot(ax=axes[6], title=’HoltWinter Exponential Smoothing')

fig.suptitle('Smoothening a Time Series', y=0.95, fontsize=14)

plt.show()



* ARIMA Model

from statsmodels.tsa.arima\_model import ARIMA

model = ARIMA(df.value, order=(1,1,2)) # 1,1,2 ARIMA(p, d, q) Model

model\_fit = model.fit(disp=0)

print(model\_fit.summary())

# Plot residual errors

residuals = pd.DataFrame(model\_fit.resid)

fig, ax = plt.subplots(1,2)

residuals.plot(title="Residuals", ax=ax[0])

residuals.plot(kind='kde', title='Density', ax=ax[1])

plt.show()

# Actual vs Fitted

model\_fit.plot\_predict(dynamic=False)

plt.show()

* Optimal ARIMA model using Out-of-Time Cross validation

# Create Training and Test

train = df.value[:85]

test = df.value[85:]

# Build Model

model = ARIMA(train, order=(1, 1, 1))

fitted = model.fit(disp=-1)

# Forecast

fc, se, conf = fitted.forecast(15, alpha=0.05) # 95% conf

# Make as pandas series

fc\_series = pd.Series(fc, index=test.index)

lower\_series = pd.Series(conf[:, 0], index=test.index)

upper\_series = pd.Series(conf[:, 1], index=test.index)

# Plot

plt.figure(figsize=(12,5), dpi=100)

plt.plot(train, label='training')

plt.plot(test, label='actual')

plt.plot(fc\_series, label='forecast')

plt.fill\_between(lower\_series.index, lower\_series, upper\_series,

color='k', alpha=.15)

plt.title('Forecast vs Actuals')

plt.legend(loc='upper left', fontsize=8)

plt.show()

* Accuracy Metrics for Time Series Forecast

The commonly used accuracy metrics to judge forecasts are:

1. Mean Absolute Percentage Error (MAPE)
2. Mean Error (ME)
3. Mean Absolute Error (MAE)
4. Mean Percentage Error (MPE)
5. Root Mean Squared Error (RMSE)
6. Lag 1 Autocorrelation of Error (ACF1)
7. Correlation between the Actual and the Forecast (corr)
8. Min-Max Error (minmax)

Typically, if you are comparing forecasts of two different series, the MAPE, Correlation and Min-Max Error can be used.

# Accuracy metrics

from statsmodels.tsa.stattools import acf

def forecast\_accuracy(forecast, actual):

mape = np.mean(np.abs(forecast - actual)/np.abs(actual)) # MAPE

me = np.mean(forecast - actual) # ME

mae = np.mean(np.abs(forecast - actual)) # MAE

mpe = np.mean((forecast - actual)/actual) # MPE

rmse = np.mean((forecast - actual)\*\*2)\*\*.5 # RMSE

corr = np.corrcoef(forecast, actual)[0,1] # corr

mins = np.amin(np.hstack([forecast[:,None], actual[:,None]]), axis=1)

maxs = np.amax(np.hstack([forecast[:,None], actual[:,None]]), axis=1)

minmax = 1 - np.mean(mins/maxs) # minmax

acf1 = acf(fc-test)[1] # ACF1

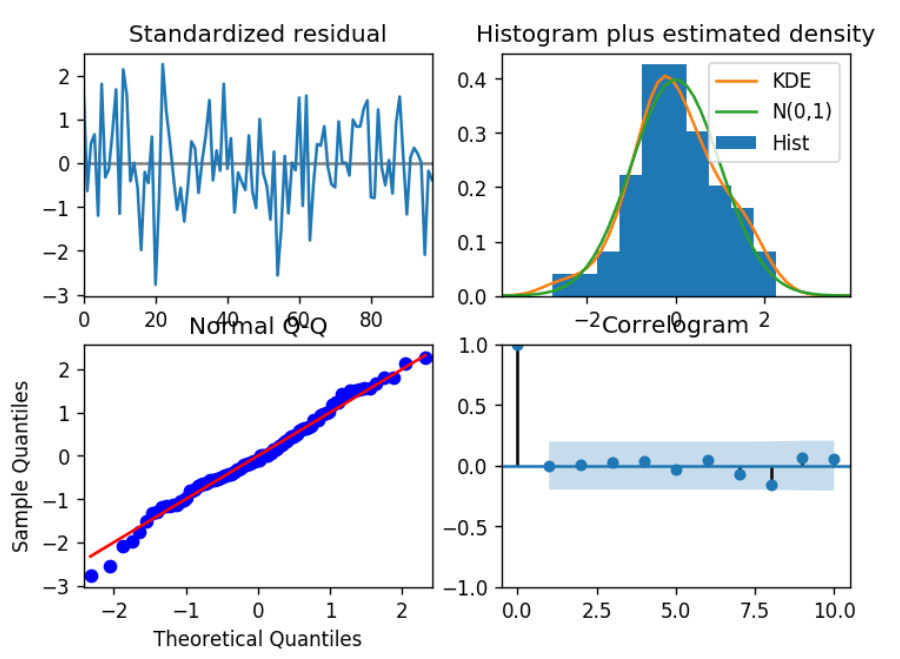
return({'mape':mape, 'me':me, 'mae': mae, 'mpe': mpe, 'rmse':rmse, 'acf1':acf1, 'corr':corr,

'minmax':minmax})

* Residual Plots in ARIMA model

model.plot\_diagnostics(figsize=(7,5))

plt.show()



* Top left: The residual errors seem to fluctuate around a mean of zero and have a uniform variance.
* Top Right: The density plot suggest normal distribution with mean zero.
* Bottom left: All the dots should fall perfectly in line with the red line. Any significant deviations would imply the distribution is skewed.
* Bottom Right: The Correlogram, aka, ACF plot shows the residual errors are not autocorrelated. Any autocorrelation would imply that there is some pattern in the residual errors which are not explained in the model. So you will need to look for more X’s (predictors) to the model.
* Forecasting

# Forecast

n\_periods = 24

fc, confint = model.predict(n\_periods=n\_periods, return\_conf\_int=True)

index\_of\_fc = np.arange(len(df.value), len(df.value)+n\_periods)

# make series for plotting purpose

fc\_series = pd.Series(fc, index=index\_of\_fc)

lower\_series = pd.Series(confint[:, 0], index=index\_of\_fc)

upper\_series = pd.Series(confint[:, 1], index=index\_of\_fc)

# Plot

plt.plot(df.value)

plt.plot(fc\_series, color='darkgreen')

plt.fill\_between(lower\_series.index,

lower\_series,

upper\_series,

color='k', alpha=.15)

plt.title("Final Forecast of WWW Usage")

plt.show()

* SARIMA Model

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.

Together, the notation for an SARIMA model is specified as: SARIMA(p,d,q)(P,D,Q)m

from statsmodels.tsa.statespace.sarimax import SARIMAX

model=SARIMAX(df['Close'],order=(2, 1, 1),seasonal\_order=(2,1,1,30))

results=model.fit()

df['forecast']=results.predict(dynamic=True)

df[['Value','forecast']].plot(figsize=(12,8))