Platinum_Palladium_Price_Forecasting

June 22, 2020

1 Imports

[1]:
```
from itertools import permutations
from functools import reduce
import itertools
import pandas as pd
import numpy as np
import pickle

# Data source
import quandl

# Facebook's Prophet
from fbprophet import Prophet

# Stats
import statsmodels.api as sm
import statsmodels.graphics.tsaplots as sgt
import statsmodels.tsa.stattools as sts
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.statespace.sarimax import SARIMAX
from statsmodels.tsa.seasonal import STL

# Plotting
from pylab import rcParams
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec
plt.style.use('fivethirtyeight')

import warnings
warnings.filterwarnings("ignore")
warnings.simplefilter('once', category=UserWarning)
```

2 Functions

[2]:
```
def dickey_fuller_test(df = None):
    """
    Performs the Dickey-Fuller

    Parameters
    ----------
    df : dataframe
        A series of values

    Returns
    -------
    dataframe
        Returns a dataframe containing the Dickey-Fuller statistics

    """
    if df is None:
        raise RuntimeError('The dataframe and columns must both be specified')

    indicies = ['Test Statistic',
                'p-value',
                'Number of lags used',
                'Number of observations',
                'Critical Value (1%)',
                'Critical Value (5%)',
                'Critical Value (10%)',
                'Maximized information critiera',
               ]

    # Perform Dickey-Fuller Test
    df_res = list(sts.adfuller(df))

    # Process output for dataframe
    stats = df_res[:4] + [x for x in df_res[4].values()] + df_res[5:]

    return pd.DataFrame({'indicies': indicies, 'Statistic': stats}).
 ↪set_index('indicies')
```

[3]:
```
def plot_acf_pacf(df, column, title, nlags=40):
    """
    Plots the observed values and their acf and pacf plots

    Parameters
    ----------
    df : dataframe
        A series of values
```

```python
    column: string
        a string representing the column name desired from the dataframe
    nlags: int
        number of lags to consider

    """
    top_row = plt.subplot2grid((2,2), (0,0), rowspan=1, colspan=2)
    bot_left = plt.subplot2grid((2, 2), (1, 0), rowspan=1, colspan=1)
    bot_right = plt.subplot2grid((2, 2), (1, 1), rowspan=1, colspan=1)

    df[column].plot(ax=top_row)
    top_row.set_title(title)

    sgt.plot_pacf(df[column], lags=nlags, zero=True, ax=bot_left)
    bot_left.set_title('PACF')

    sgt.plot_acf(df[column], lags=nlags, zero=True, ax=bot_right)
    bot_right.set_title('ACF')

    plt.show()
```

[4]:
```python
def sarima_model_selection(df, orders, method = 'lbfgs'):
    """
    Performs a grid search to find the model which minimizes the AICc score

    Parameters
    ----------
    df : dataframe
        A series of values
    orders: dict
        a dictionary of lists for keys p,d,q, P, D, Q and an integer for s.
    method: string
        indicates the optimizer for the model selection

    Returns
    -------
    mod: SARIMAX model
        Returns the SARIMAX which minimizes the AICc score among all the order␣
↪combinations

    """

    p, d, q = orders['p'], orders['d'], orders['q']
    P, D, Q = orders['P'], orders['D'], orders['Q']
    s = orders['s']

    min_aicc = float('inf')
```

```
    best_order = None
    best_model = None

    iterations = reduce(lambda a,b: a * b, [len(x) for x in [p,q,d,P,Q,D]])
    print('There are {} combination to evaluate...'.format(iterations))

    pdq = list(itertools.product(p, d, q))
    seasonal_pdq = [(x[0], x[1], x[2], s) for x in list(itertools.product(P, D,␣
↪Q))]

    for param in pdq:
        for param_seasonal in seasonal_pdq:
            try:
                mod = sm.tsa.statespace.SARIMAX(df,
                                                order=param,
                                                seasonal_order=param_seasonal,
                                                enforce_stationarity=False,
                                                enforce_invertibility=False)
                results = mod.fit(max_iter=500, disp=0)
                if results.aicc < min_aicc:
                    min_aicc = results.aicc
                    best_model = mod
                    best_params = '{} x {}{}'.format(param, param_seasonal, s)
            except:
                continue
    print('Best Model:{}\nLowest Score:{}'.format(best_params, min_aicc))

    return mod
```

[5]:

```
def plot_STL_decomposition(df, method='additive'):
    """
    Plots the observed values and their trend, seasonality, and residuals.␣
↪Additionally, outputs
    the trend strength and seasonality strength.

    Parameters
    ----------
    df : dataframe
        A series of values
    method: string
        a string indicating whether to use a additive or multiplicative␣
↪decomposition method

    """
    observed_ax = plt.subplot2grid((4,1), (0,0), rowspan=1, colspan=1)
    trend_ax = plt.subplot2grid((4, 1), (1, 0), rowspan=1, colspan=1)
    season_ax = plt.subplot2grid((4, 1), (2, 0), rowspan=1, colspan=1)
```

```
    resid_ax = plt.subplot2grid((4, 1), (3, 0), rowspan=1, colspan=1)

    trend_strength, seasonality_strength = 0, 0

    observed_ax.set_title(df.name + ' STL '+ method + ' decomposition')
    observed_ax.set_ylabel('Observed')
    trend_ax.set_ylabel('Trend')
    season_ax.set_ylabel('Season')
    resid_ax.set_ylabel('Resid')
    if method == 'additive':
        stl = STL(df).fit()
        df.plot(ax=observed_ax)
        stl.trend.plot(ax=trend_ax)
        stl.seasonal.plot(ax=season_ax)
        stl.resid.plot(ax=resid_ax)
        trend_strength = max(0, 1 - np.var( stl.resid) / np.var( stl.trend + ␣
↪stl.resid))
        seasonality_strength = max(0, 1 - np.var( stl.resid) / np.var( stl.
↪seasonal +  stl.resid))
    elif method == 'multiplicative':
        df = np.log(df)
        stl = STL(df).fit()
        np.exp(df).plot(ax=observed_ax)
        np.exp(stl.trend).plot(ax=trend_ax)
        np.exp(stl.seasonal).plot(ax=season_ax)
        np.exp(stl.resid).plot(ax=resid_ax)
        trend_strength = max(0, 1 - np.var(stl.resid) / np.var( stl.trend + ␣
↪stl.resid))
        seasonality_strength = max(0, 1 - np.var( stl.resid) / np.var( stl.
↪seasonal +  stl.resid))
    else:
        raise RuntimeError("Method must either 'additive' or 'multiplicative'.")

    print('Trend Strength: {}\nSeasonality Strength: {}'.format(trend_strength,␣
↪seasonality_strength))
```

3 Loading the Data

[6]:
```
quandl.ApiConfig.api_key = 'k2csLJUpyjYPGKKeCSBN'
platinum_data = quandl.get('LPPM/PLAT', column_index='1')
palladium_prices = quandl.get('LPPM/PALL', column_index='1')
```

[7]:
```
df = platinum_data.rename(columns={'USD AM': 'plat_prices'})
df['pall_prices'] = palladium_prices['USD AM']
```

[8]:
```
df.index.min(), df.index.max()
```

[8]:
```
(Timestamp('1990-04-02 00:00:00'), Timestamp('2020-06-19 00:00:00'))
```

There is about 20 years worth of data

4 Pre-process Data

[9]:
```
df.head()
```

[9]:
```
            plat_prices  pall_prices
Date
1990-04-02       471.00       128.00
1990-04-03       475.80       128.35
1990-04-04       475.70       128.35
1990-04-05       481.75       128.40
1990-04-06       481.00       128.75
```

[10]:
```
df.isna().sum()
```

[10]:
```
plat_prices    0
pall_prices    3
dtype: int64
```

4.0.1 Fill missing data and resample into monthly frequency

[11]:
```
df['pall_prices'] = df['pall_prices'].fillna(method='ffill')
df = df[['plat_prices','pall_prices']].resample('MS').mean()
df = df.apply(lambda x: round(x, 2))
```

5 Platinum

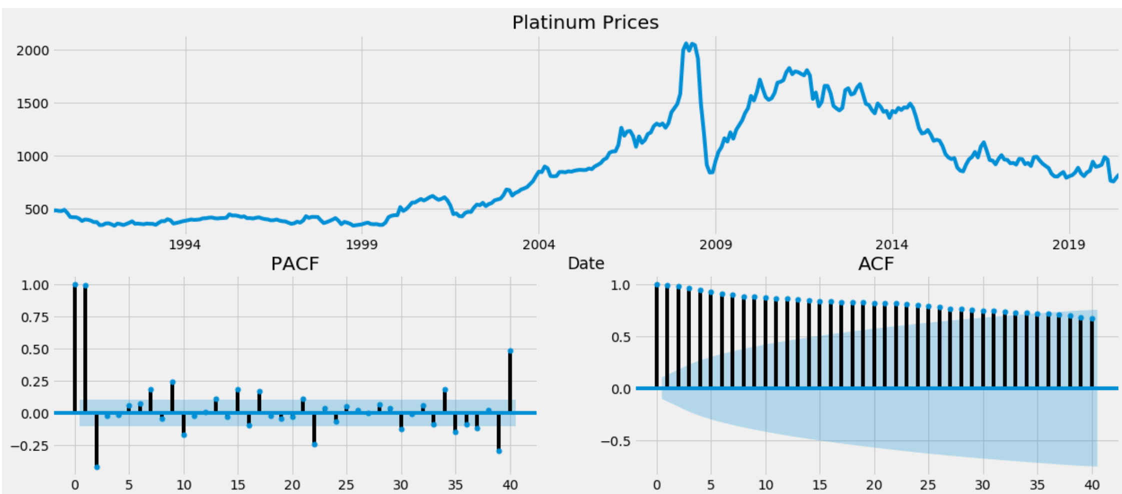
5.1 Time Plot of Platinum

[12]:
```
rcParams['figure.figsize'] = 18, 8
df.plat_prices.plot()
```

[12]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f82022d2050>
```

The time-series has a seasonally pattern, although it is hard to discern, where the prices has a series a consecutive increases followed by consecutive decreases. The trend seems to gradually until 2002 where the price spikes and experiences great flucation, likely due to the stock market crash, and then gradually decreases.

[13]:
```
plot_acf_pacf(df, 'plat_prices', 'Platinum Prices')
```



While classical decomposition is widely used, it's not recommended. Instead, I will opt to use STL decomposition which is not only versatile and robust. STL has many advantages over over the classical, SEATS and X11 decomposition methods.

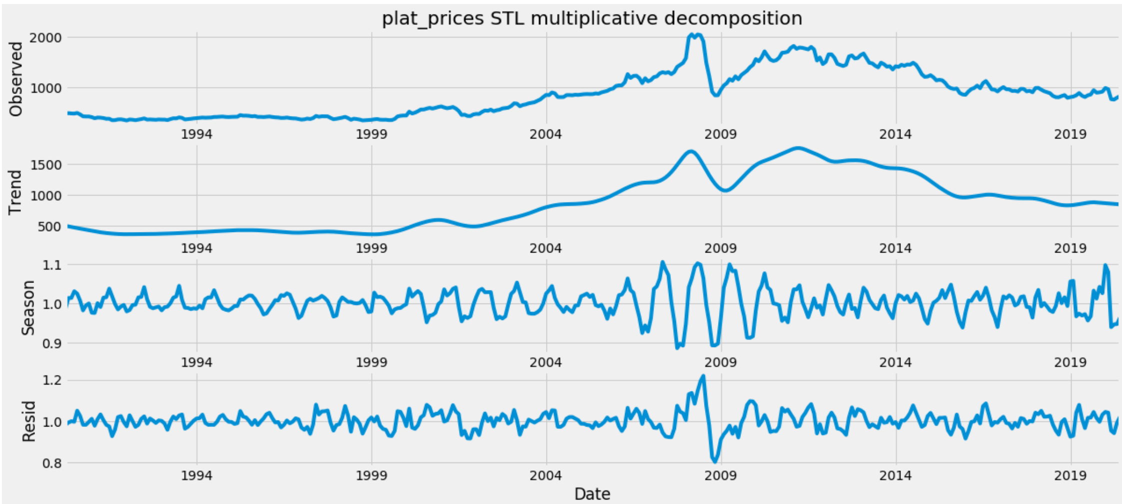
The variation in the seasonal pattern appears proportional to the level of the time series, therefore we will opt for a multiplicative decomposition. The caveat of STL is that it only works for additive models. However, it is possible to obtain a multiplicative decomposition by takign the logs of the data and then back-transforming the componenets. This is possible because $y_t = S_t$ x T_t x $R_t \equiv$

$log(y_t)$ $log(S_t)$ x $log(T_t)$ x $log(R_t)$

https://otexts.com/fpp3/stl.html

[14]:
```
plot_STL_decomposition(df.plat_prices, method='multiplicative')
```

```
Trend Strength: 0.9932297077382147
Seasonality Strength: 0.4369190761188051
```



The plot indicates a trend that gradually increases until 2008 and then gradually decreases. Moreover, the plot definitiely indicates seasonality

[15]:
```
dickey_fuller_test(df.plat_prices)
```

[15]:

indicies	Statistic
Test Statistic	-1.316759
p-value	0.621471
Number of lags used	8.000000
Number of observations	354.000000
Critical Value (1%)	-3.448958
Critical Value (5%)	-2.869739
Critical Value (10%)	-2.571138
Maximized information critiera	3759.805843

Examining the Dickey-Fuller test, the p-value suggests 62% of not rejecting the null hypothesis. Additionally, the test statistic is larger than the critical values at 1%, 5%, and 10%. All this suggests we can't confirm stationarity.

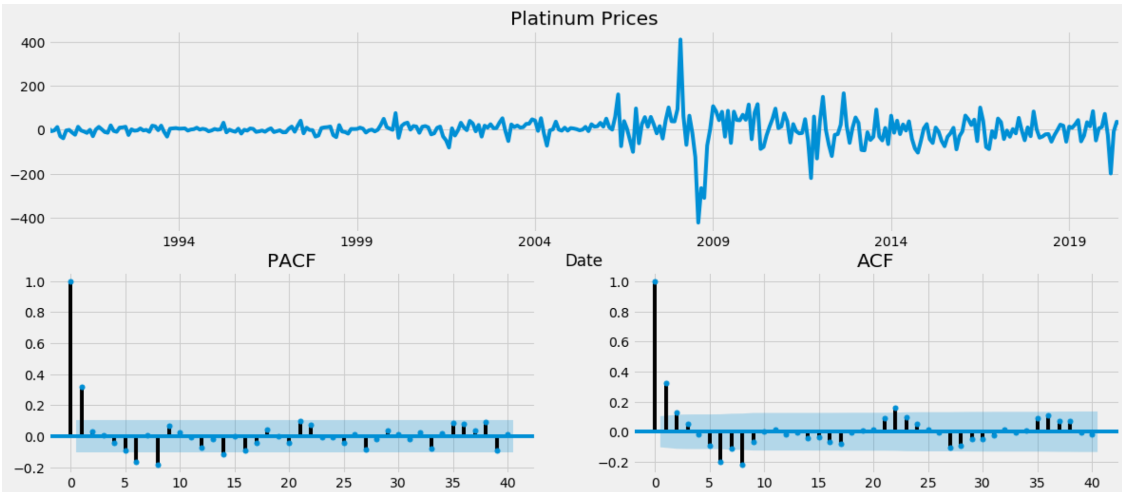
ARIMA models require stationarity, so we must perform differencing

[16]:
```
dickey_fuller_test(df.plat_prices.diff()[1:])
```

[16]:
```
                                   Statistic
indicies
Test Statistic                -8.882680e+00
p-value                        1.305076e-14
Number of lags used            7.000000e+00
Number of observations         3.540000e+02
Critical Value (1%)           -3.448958e+00
Critical Value (5%)           -2.869739e+00
Critical Value (10%)          -2.571138e+00
Maximized information critiera  3.749959e+03
```

Perfect, the test statistic is much smaller than all the critical values and the p-value is also significant indicating stationarity.

[17]:
```
plot_acf_pacf(df.diff()[1:], 'plat_prices', 'Platinum Prices')
```



For the AR component p, we examine the significant lags in the PACF plot For the seasonal AR component, we look at lags of multiples of 12. We see the 24th lag is significant so we can examine seasonality when P = 2.

For the MA component q, we examine the significant lags in the ACF plot. For the seasonal MA component, we look at the lags of multiples of 12. There doesn't seem to be any significant lags.

[18]:
```
orders = {'p': [1,2,6,8],
          'd': [1],
          'q': [1,2,3,6,8],
          'P': [0,1,2],
          'D': [1],
          'Q': [0,1],
          's': 12
}
```

```
plat_best_model = sarima_model_selection(df.plat_prices, orders)
```

```
There are 120 combination to evaluate…

/opt/anaconda3/envs/mlenv/lib/python3.7/site-
packages/statsmodels/base/model.py:568: ConvergenceWarning: Maximum Likelihood
optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)

Best Model:(8, 1, 1) x (2, 1, 1, 12)12
Lowest Score:3501.4581092542307
```

[19]:
```
plat_results = plat_best_model.fit()
print(plat_results.summary().tables[1])
```

```
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1         -1.3628      0.104    -13.094      0.000      -1.567      -1.159
ar.L2         -1.1127      0.183     -6.088      0.000      -1.471      -0.754
ar.L3         -1.0501      0.204     -5.142      0.000      -1.450      -0.650
ar.L4         -0.7624      0.227     -3.354      0.001      -1.208      -0.317
ar.L5         -0.7762      0.220     -3.536      0.000      -1.206      -0.346
ar.L6         -0.8390      0.181     -4.625      0.000      -1.195      -0.483
ar.L7         -0.9734      0.144     -6.754      0.000      -1.256      -0.691
ar.L8         -0.6362      0.084     -7.599      0.000      -0.800      -0.472
ma.L1          1.6726      0.109     15.317      0.000       1.459       1.887
ma.L2          1.6350      0.205      7.975      0.000       1.233       2.037
ma.L3          1.6614      0.255      6.511      0.000       1.161       2.162
ma.L4          1.4228      0.282      5.046      0.000       0.870       1.975
ma.L5          1.3915      0.275      5.061      0.000       0.853       1.930
ma.L6          1.2932      0.233      5.540      0.000       0.836       1.751
ma.L7          1.3362      0.164      8.144      0.000       1.015       1.658
ma.L8          0.8389      0.082     10.229      0.000       0.678       1.000
ar.S.L12      -0.2094      0.101     -2.077      0.038      -0.407      -0.012
ar.S.L24      -0.0390      0.094     -0.414      0.679      -0.224       0.146
ma.S.L12      -0.8854      0.073    -12.075      0.000      -1.029      -0.742
sigma2      4087.9684    419.815      9.738      0.000    3265.146    4910.791
==============================================================================
```

Many of the p-values are significant indicating the lags are significantly different from 0.

5.1.1 Ljung-Box test

In addition to looking at the ACF plot, we can do a more formal test for autocorrelation by considering a whole set residuals instead of treating each one separately. Therefore, we test whether the first autocorrelations are significantly different from what would be expected from a white noise process.

It is suggested to use lags = 10 for non-seasonal data and lags = 2m for seasonal data where m is the period of seasonality. However, if

However, the test is not good when the number of lags is too high so if the the number of lags is larger than T/5 where T is the number of observations, then use lags = T/5.

source: https://otexts.com/fpp3/diagnostics.html

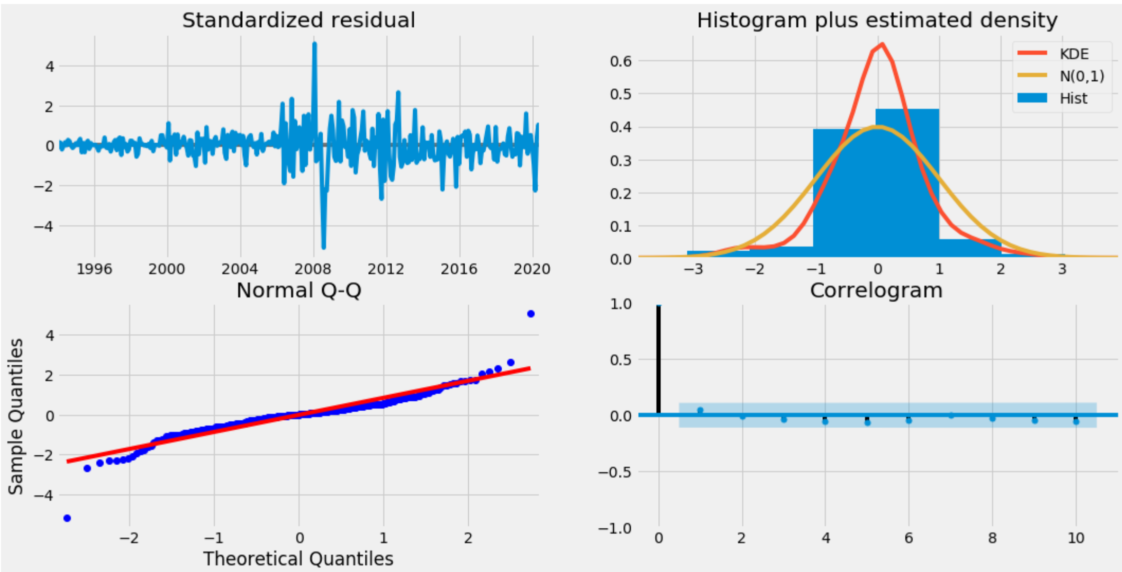
[20]:
```
sm.stats.acorr_ljungbox(plat_results.resid, lags=[24], return_df=True)
```

[20]:

	lb_stat	lb_pvalue
24	57.705438	0.000134

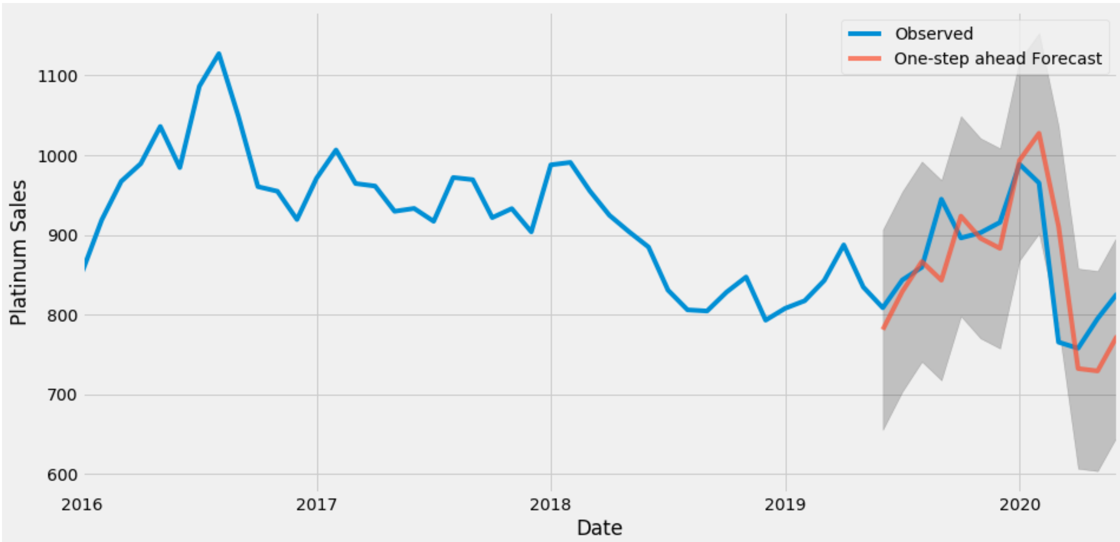
Our p-value indicates the first autocorelations are significantly different from what would be expected from a white noise process.

[21]:
```
plat_results.plot_diagnostics(figsize=(16, 8))
plt.show()
```



[44]:
```
pred = plat_results.get_prediction(start=pd.to_datetime('2019-06-01'),␣
 ↪dynamic=False)
pred_ci = pred.conf_int()
ax = df.plat_prices['2016':].plot(label='Observed')
pred.predicted_mean.plot(ax=ax, label='One-step ahead 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('Platinum Sales')
```

```
plt.legend()
plt.show()
```

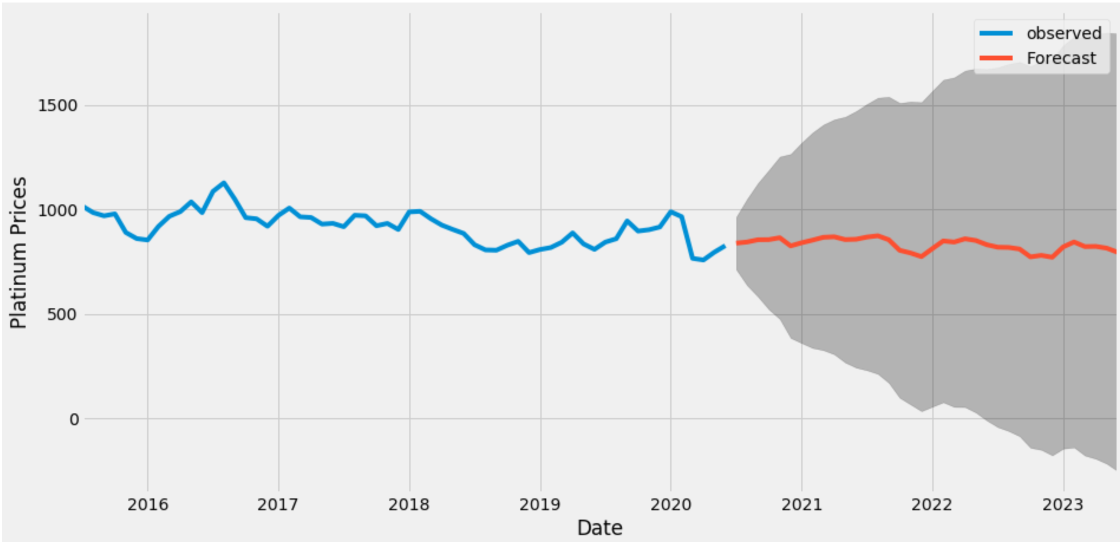


The forecasts seem to align well with the obsereved values.

[23]:
```
y_forecasted = pred.predicted_mean
y_truth = df.plat_prices['2008-01-01':]
mse = ((y_forecasted - y_truth) ** 2).mean()
print('The Mean Squared Error of our forecasts is {}'.format(round(mse, 2)))
print('The Root Mean Squared Error of our forecasts is {}'.format(round(np.
 ↪sqrt(mse), 2)))
```

```
The Mean Squared Error of our forecasts is 3536.01
The Root Mean Squared Error of our forecasts is 59.46
```

[43]:
```
pred_uc = plat_results.get_forecast(steps=36)
pred_ci = pred_uc.conf_int()
ax = df.plat_prices['2015-07-01':].plot(label='observed', figsize=(14, 7))
pred_uc.predicted_mean.plot(ax=ax, label='Forecast')
ax.fill_between(pred_ci.index,
                pred_ci.iloc[:, 0],
                pred_ci.iloc[:, 1], color='k', alpha=.25)
ax.set_xlabel('Date')
ax.set_ylabel('Platinum Prices')
plt.legend()
plt.show()
```



6 Palladium

6.1 Time Plot of Palladium

[25]:
```
rcParams['figure.figsize'] = 18, 8
df.pall_prices.plot()
```

[25]: <matplotlib.axes._subplots.AxesSubplot at 0x7f81d81cf6d0>

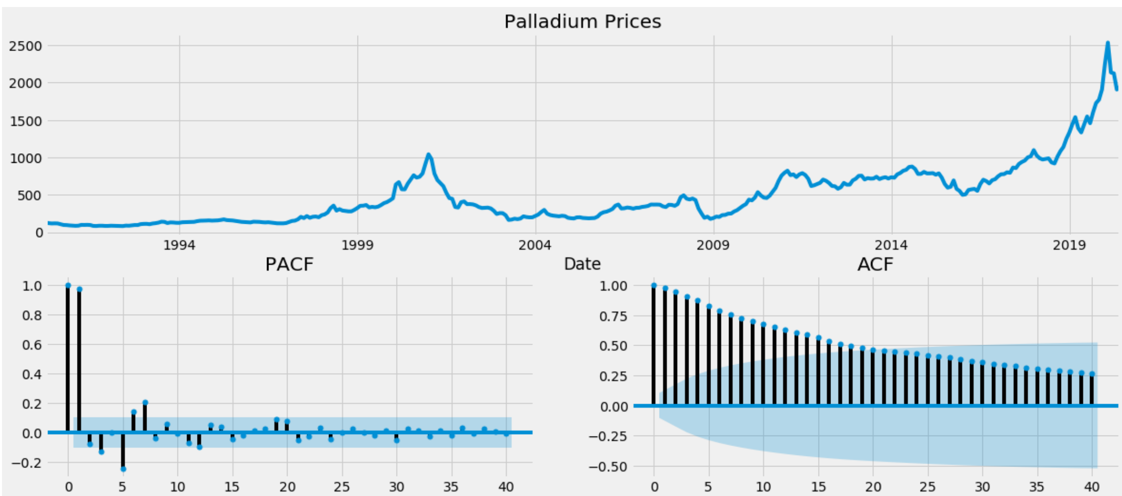
The time-series has a seasonally pattern with occasional spikes such as in 2001-2002. The trend increases through the entire timeseries with notable expoential growth after 2016.

[26]:
```
dickey_fuller_test(df.pall_prices)
```

[26]:
```
                                 Statistic
indicies
Test Statistic                    0.812363
p-value                           0.991833
Number of lags used              17.000000
Number of observations          345.000000
Critical Value (1%)              -3.449447
Critical Value (5%)              -2.869954
Critical Value (10%)             -2.571253
Maximized information critiera  3691.410626
```

The p-value suggests 99.2% of not rejecting the null hypothesis. Additionally, the test statistic is larger than the critical values at 1%, 5%, and 10%. All this suggests we can't confirm stationarity

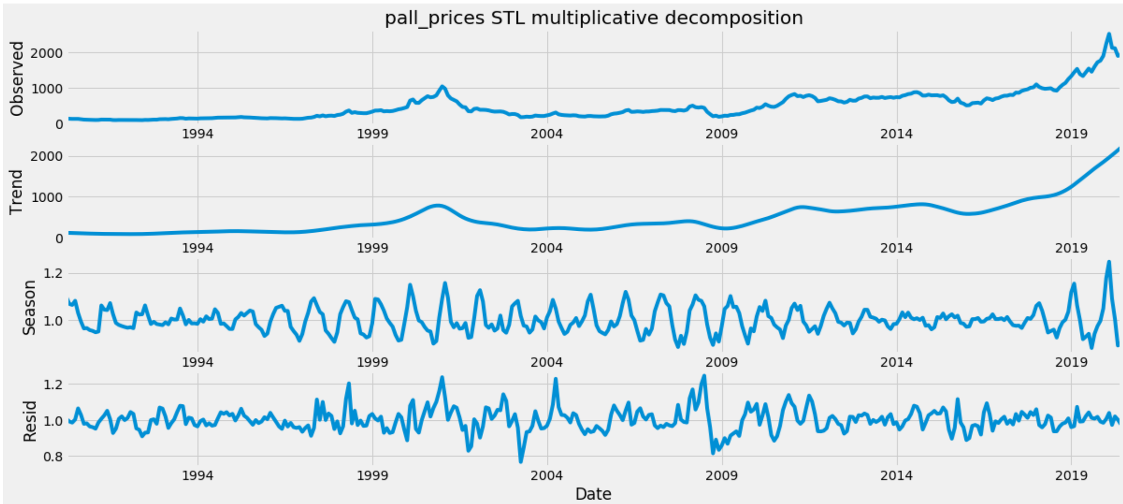
[27]:
```
plot_acf_pacf(df, 'pall_prices', 'Palladium Prices')
```



As above, we will use multiplicative STL decomposition

[28]:
```
stl = plot_STL_decomposition(df.pall_prices, method='multiplicative')
```

```
Trend Strength: 0.9940630024022675
Seasonality Strength: 0.4592015311752917
```



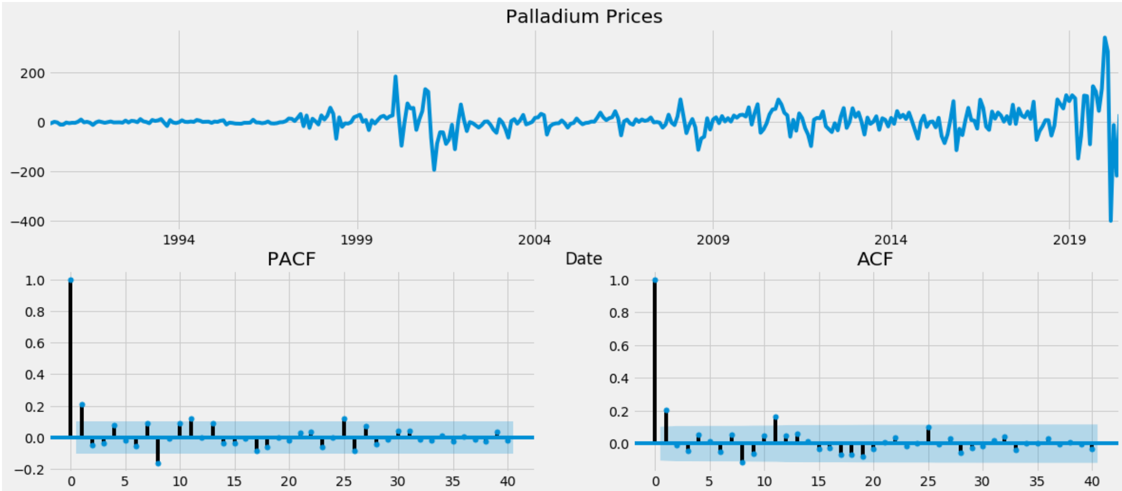
Let's examine the stationary after differencing

[29]:
```
dickey_fuller_test(df.pall_prices.diff()[1:])
```

[29]:
```
                                    Statistic
indicies
Test Statistic                      -3.484995
p-value                              0.008381
Number of lags used                 12.000000
Number of observations             349.000000
Critical Value (1%)                 -3.449227
Critical Value (5%)                 -2.869857
Critical Value (10%)                -2.571201
Maximized information critiera    3679.906205
```

Perfect, the test statistic is much smaller than all the critical values and the p-value is also significant indicating stationarity.

[30]:
```
plot_acf_pacf(df.diff()[1:], 'pall_prices', 'Palladium Prices')
```



[31]:
```
orders = {'p': [1,2, 9], #8
          'd': [1],
          'q': [1,2, 9],
          'P': [0,1],
          'D': [1],
          'Q': [0,1],
          's': 12
}

pall_best_model = sarima_model_selection(df.pall_prices, orders)
```

```
There are 36 combination to evaluate…

/opt/anaconda3/envs/mlenv/lib/python3.7/site-
packages/statsmodels/base/model.py:568: ConvergenceWarning: Maximum Likelihood
optimization failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)

Best Model:(2, 1, 9) x (1, 1, 1, 12)12
Lowest Score:3546.042914254437
```

Best Model:(1, 1, 9) x (0, 1, 1, 12)12 Lowest Score:3549.1285862915197

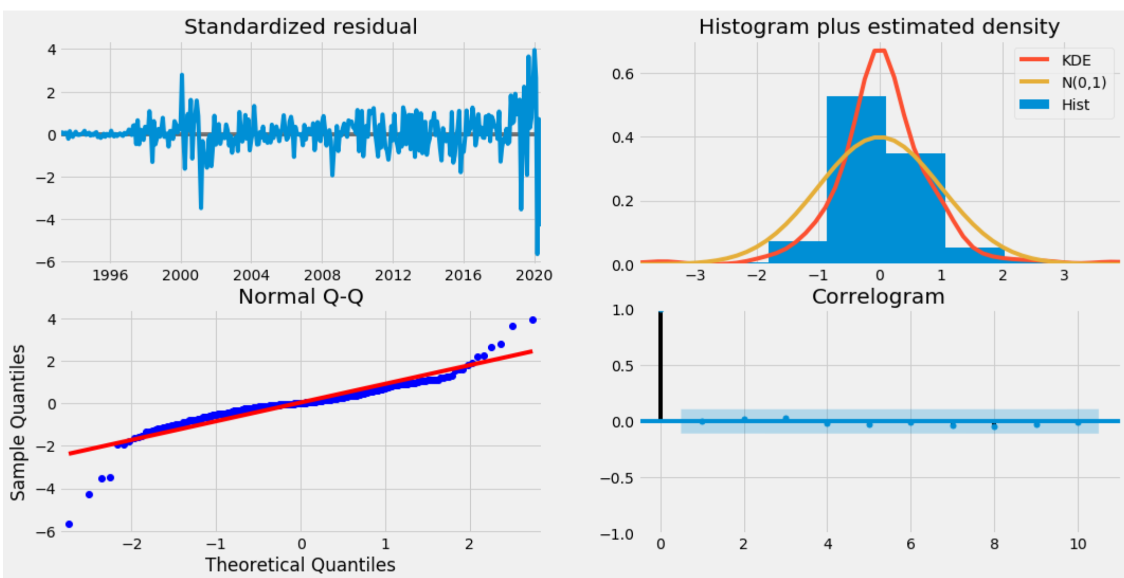
[32]:
```
pall_results = pall_best_model.fit()
print(pall_results.summary().tables[1])
```

```
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          0.1705      0.659      0.259      0.796      -1.122       1.463
ar.L2         -0.3624      0.565     -0.641      0.522      -1.471       0.746
ar.L3         -0.7616      0.600     -1.269      0.204      -1.938       0.415
```

```
ar.L4           0.1224      0.905      0.135      0.892      -1.652       1.897
ar.L5          -0.1744      0.788     -0.221      0.825      -1.719       1.370
ar.L6           0.2941      0.573      0.514      0.607      -0.828       1.416
ar.L7           0.0770      0.466      0.165      0.869      -0.837       0.991
ar.L8          -0.0114      0.384     -0.030      0.976      -0.764       0.742
ar.L9           0.5283      0.298      1.776      0.076      -0.055       1.111
ma.L1           0.0686      0.664      0.103      0.918      -1.233       1.371
ma.L2           0.3112      0.672      0.463      0.643      -1.006       1.629
ma.L3           0.7651      0.680      1.125      0.260      -0.568       2.098
ma.L4           0.1757      0.990      0.177      0.859      -1.764       2.116
ma.L5           0.2270      1.028      0.221      0.825      -1.788       2.242
ma.L6          -0.4673      0.860     -0.543      0.587      -2.153       1.219
ma.L7           0.2207      0.551      0.400      0.689      -0.860       1.301
ma.L8          -0.1454      0.606     -0.240      0.810      -1.332       1.041
ma.L9          -0.5643      0.447     -1.262      0.207      -1.441       0.312
ar.S.L12        0.0133      0.158      0.084      0.933      -0.296       0.323
ma.S.L12       -0.8855      0.145     -6.126      0.000      -1.169      -0.602
sigma2       3264.7086    248.572     13.134      0.000    2777.517    3751.900
==============================================================================
```

[33]:

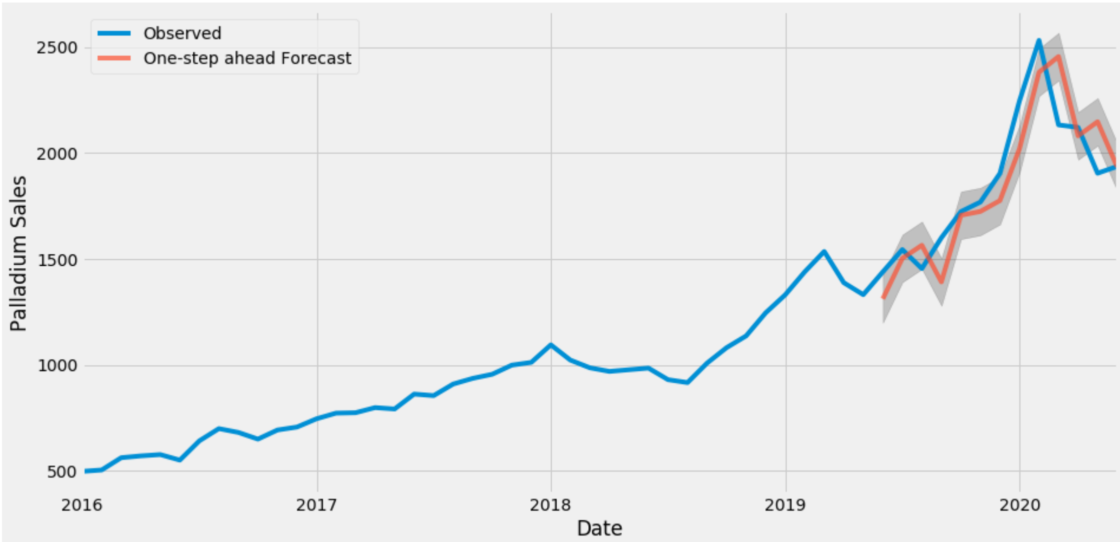
```python
pall_results.plot_diagnostics(figsize=(16, 8))
plt.show()
```



[34]:

```python
pred = pall_results.get_prediction(start=pd.to_datetime('2019-06-01'),␣
 ↪dynamic=False)
pred_ci = pred.conf_int()
ax = df.pall_prices['2016':].plot(label='Observed')
```

```
pred.predicted_mean.plot(ax=ax, label='One-step ahead 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('Palladium Sales')
plt.legend()
plt.show()
```



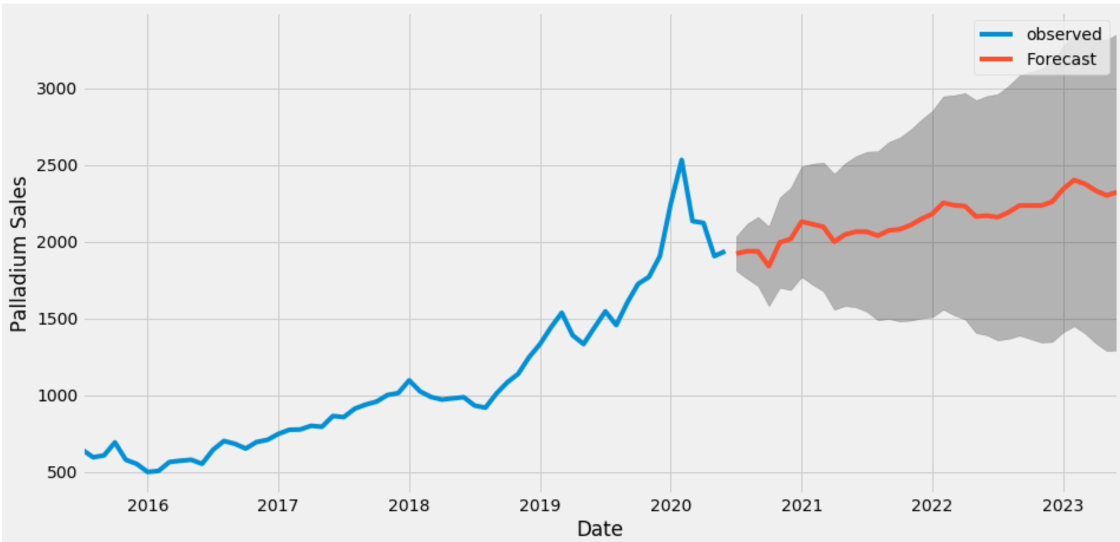
The forecasts seem to align well with the obsereved values

[35]:
```
y_forecasted = pred.predicted_mean
y_truth = df.pall_prices['2008-01-01':]
mse = ((y_forecasted - y_truth) ** 2).mean()
print('The Mean Squared Error of our forecasts is {}'.format(round(mse, 2)))
print('The Root Mean Squared Error of our forecasts is {}'.format(round(np.
↪sqrt(mse), 2)))
```

```
The Mean Squared Error of our forecasts is 25437.19
The Root Mean Squared Error of our forecasts is 159.49
```

[36]:
```
pred_uc = pall_results.get_forecast(steps=36)
pred_ci = pred_uc.conf_int()
ax = df.pall_prices['2015-07-01':].plot(label='observed', figsize=(14, 7))
pred_uc.predicted_mean.plot(ax=ax, label='Forecast')
ax.fill_between(pred_ci.index,
                pred_ci.iloc[:, 0],
                pred_ci.iloc[:, 1], color='k', alpha=.25)
```

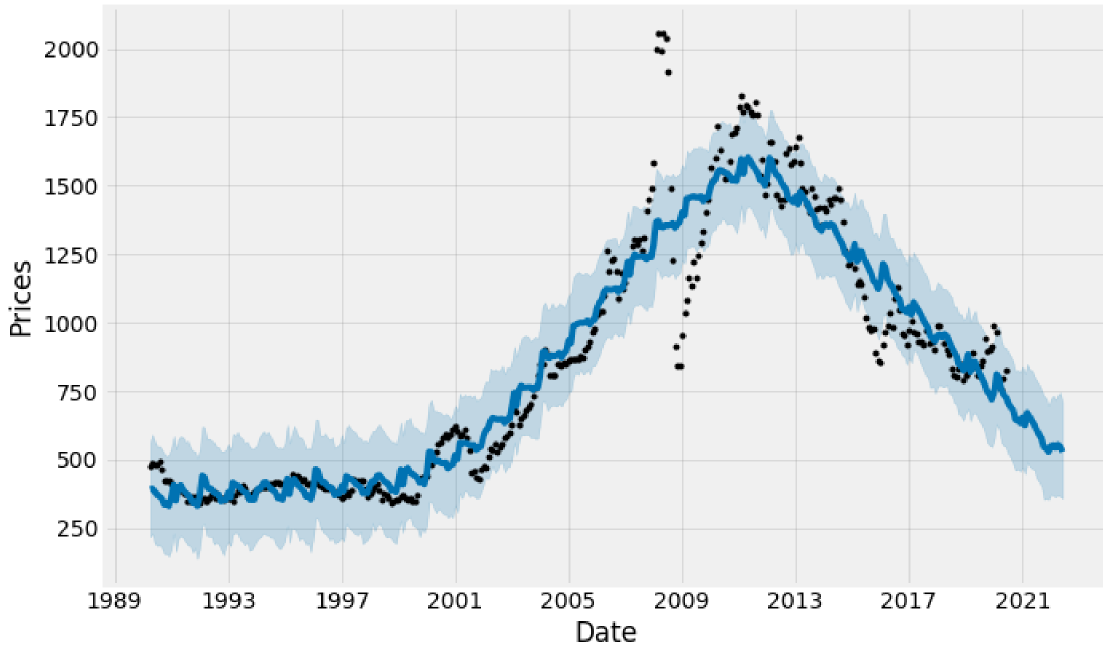
```
ax.set_xlabel('Date')
ax.set_ylabel('Palladium Sales')
plt.legend()
plt.show()
```



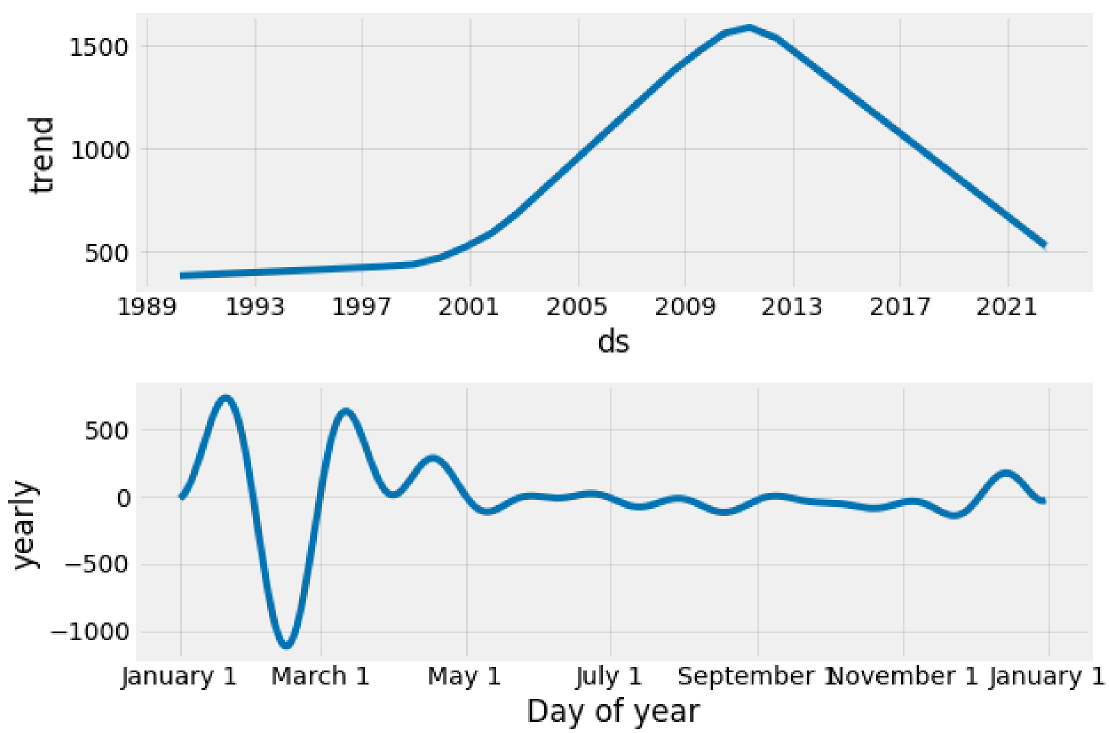
7 Prophet

[37]:
```
fbp_plat_model = Prophet()
fbp_plat_model.fit(pd.DataFrame({'ds': df.index,'y': df.plat_prices}))
fbp_plat_future = fbp_plat_model.make_future_dataframe(periods=24, freq='MS')
fbp_plat_forecast = fbp_plat_model.predict(fbp_plat_future)
fbp_plat_model.plot(fbp_plat_forecast, xlabel = 'Date', ylabel = 'Prices')
plt.show()
```

```
INFO:fbprophet:Disabling weekly seasonality. Run prophet with
weekly_seasonality=True to override this.
INFO:fbprophet:Disabling daily seasonality. Run prophet with
daily_seasonality=True to override this.
```

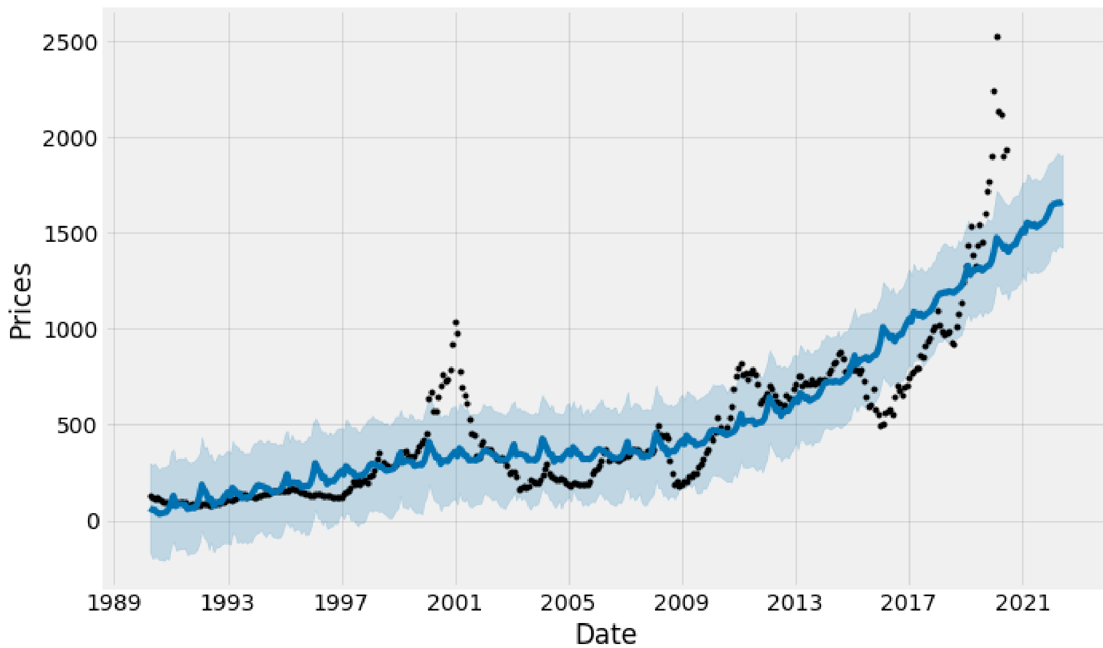


[38]:
```
fbp_plat_model.plot_components(fbp_plat_forecast);
```

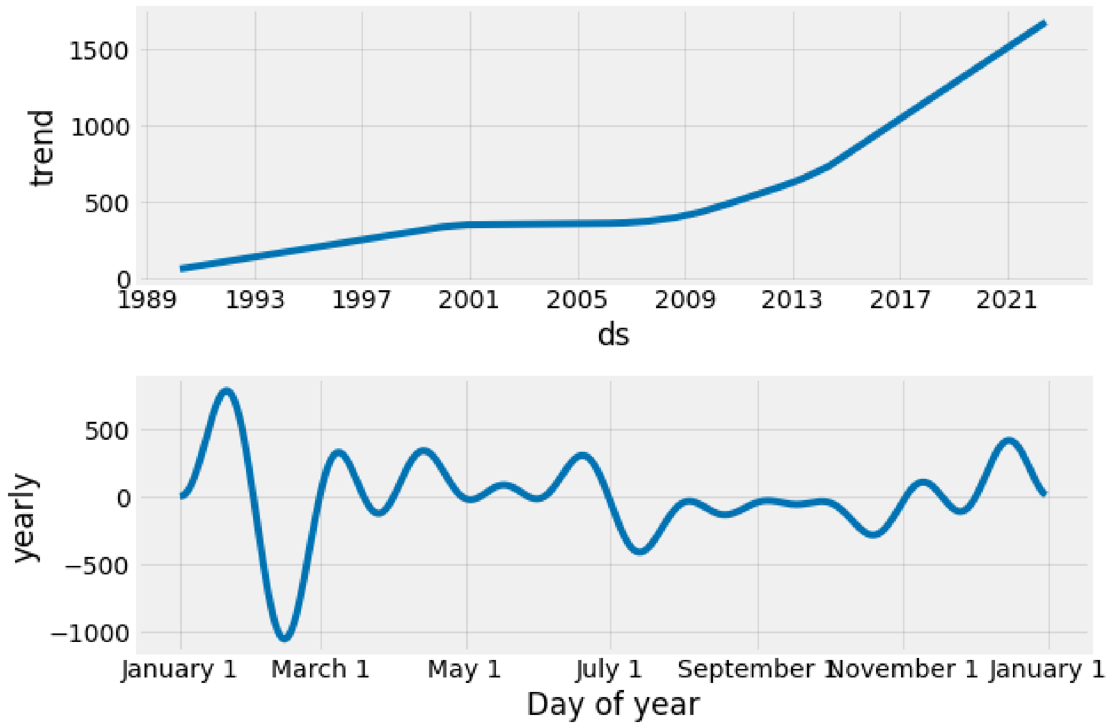


[39]:
```
fbp_pall_model = Prophet()
fbp_pall_model.fit(pd.DataFrame({'ds': df.index,'y': df.pall_prices}))
fbp_pall_future = fbp_pall_model.make_future_dataframe(periods=24, freq='MS')
fbp_pall_forecast = fbp_pall_model.predict(fbp_plat_future)
fbp_pall_model.plot(fbp_pall_forecast, xlabel = 'Date', ylabel = 'Prices')
plt.show()
```

```
INFO:fbprophet:Disabling weekly seasonality. Run prophet with
weekly_seasonality=True to override this.
INFO:fbprophet:Disabling daily seasonality. Run prophet with
daily_seasonality=True to override this.
```

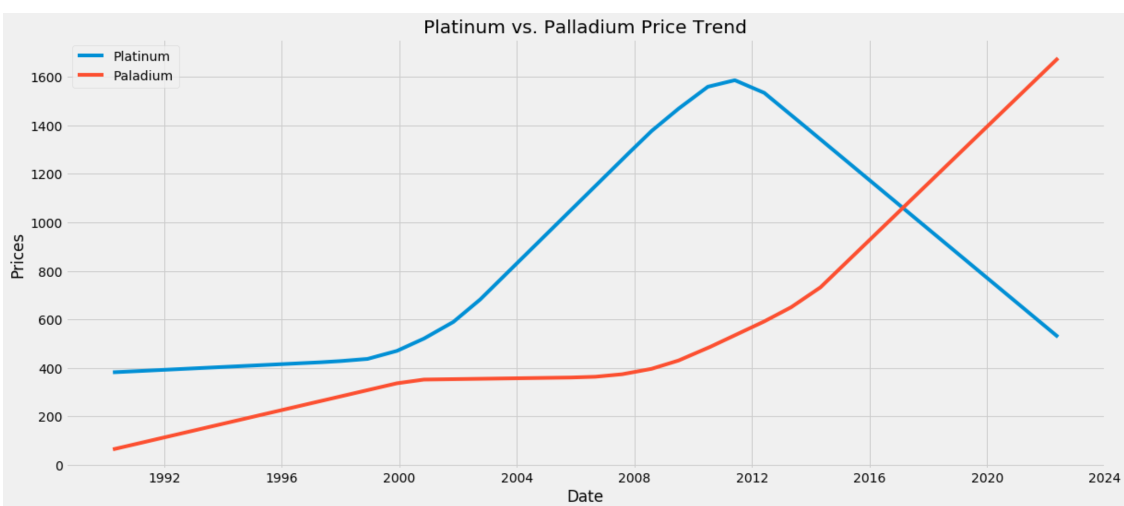


[40]:
```
fbp_pall_model.plot_components(fbp_pall_forecast);
```

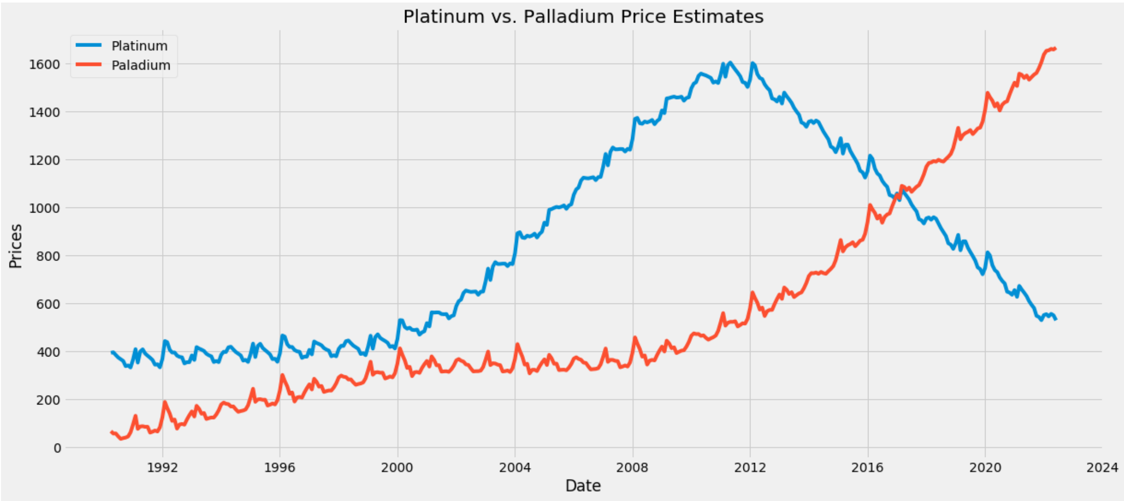


[41]:

```
plt.plot(fbp_plat_forecast['ds'], fbp_plat_forecast['trend'], label='Platinum')
plt.plot(fbp_pall_forecast['ds'], fbp_pall_forecast['trend'], label='Paladium')
plt.xlabel('Date')
plt.ylabel('Prices')
plt.title('Platinum vs. Palladium Price Trend');
plt.legend()
plt.show()
```



[42]:
```
plt.plot(fbp_plat_forecast['ds'], fbp_plat_forecast['yhat'], label='Platinum')
plt.plot(fbp_pall_forecast['ds'], fbp_pall_forecast['yhat'], label='Paladium')
plt.xlabel('Date')
plt.ylabel('Prices')
plt.title('Platinum vs. Palladium Price Estimates');
plt.legend()
plt.show()
```



7.1 Pickle Forecasting Models

[43]:
```
pickle.dump( plat_results, open( "platinum_forecasting_model.p", "wb" ) )
pickle.dump( pall_results, open( "palladium_forecasting_model.p", "wb" ) )
pickle.dump( fbp_plat_model, open( "fbprophet_platinum_forecasting_model.p",␣
↪"wb" ) )
pickle.dump( fbp_pall_model, open( "fbprophet_palladium_forecasting_model.p",␣
↪"wb" ) )
```