In [1]:
```python
import os
import time
import random
import numpy as np
import pandas as pd
import seaborn as sns
from itertools import product
import matplotlib.pyplot as plt

from pylab import rcParams
from IPython.display import display

from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.graphics.tsaplots import plot_pacf
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.arima_model import ARIMA
from statsmodels.tsa.statespace.sarimax import SARIMAX
from sklearn.model_selection import TimeSeriesSplit

from scipy import stats
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, mean_absolute_error
import statsmodels.api as sm

import warnings
warnings.filterwarnings("ignore")

plt.style.use('seaborn')
```

Seeding everything first

This will allow us to have reproducible experiments

In [2]:
```python
SEED = 42
random.seed(SEED)
np.random.seed(SEED)
os.environ['PYTHONHASHSEED'] = str(SEED)
```

Loading the Time Series

- For this project will as time series data the price of Bitcoin
- The dataset can be found here: https://www.kaggle.com/mczielinski/bitcoin-historical-data

In [3]:
```python
data = pd.read_csv('data/btc.csv')
display(data)
```

	Timestamp	Open	High	Low	Close	Volume_(BTC)	Volume_(Currency)	Weighted_Price
0	1325317920	4.39	4.39	4.39	4.39	0.455581	2.000000	4.390000
1	1325317980	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	1325318040	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	1325318100	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	1325318160	NaN	NaN	NaN	NaN	NaN	NaN	NaN
...	...	...	...	...	...	...	...	...
4857372	1617148560	58714.31	58714.31	58686.00	58686.00	1.384487	81259.372187	58692.753339
4857373	1617148620	58683.97	58693.43	58683.97	58685.81	7.294848	428158.146640	58693.226508
4857374	1617148680	58693.43	58723.84	58693.43	58723.84	1.705682	100117.070370	58696.198496
4857375	1617148740	58742.18	58770.38	58742.18	58760.59	0.720415	42332.958633	58761.866202
4857376	1617148800	58767.75	58778.18	58755.97	58778.18	2.712831	159417.751000	58764.349363

4857377 rows × 8 columns

Exploratory Data Analysis

- Before we start, we can observe that our data it's collected minute by minute and multivariate
- For the first problem we will resample the data by month using the mean value
- And will use as single feature only the 'Weighted_Price' as the time series

Resampling data by month

In [4]:
```python
data.Timestamp = pd.to_datetime(data.Timestamp, unit='s')
data.index = data.Timestamp
data = data.resample('M').mean()
series = data[['Weighted_Price']]
display(data)
```

Timestamp	Open	High	Low	Close	Volume_(BTC)	Volume_(Currency)	Weighted_Price
2011-12-31	4.465000	4.482500	4.465000	4.482500	23.829470	106.330084	4.471603
2012-01-31	6.345389	6.348982	6.341218	6.346148	4.031777	25.168238	6.345955
2012-02-29	5.230208	5.231646	5.227036	5.228510	8.313993	42.239422	5.228443
2012-03-31	4.985481	4.986695	4.982580	4.983828	15.197791	76.509751	4.984397
2012-04-30	4.995171	4.996447	4.993763	4.995079	21.683913	108.218094	4.995091
...	...	...	...	...	...	...	...
2020-11-30	16535.778528	16545.663704	16525.571002	16536.023486	6.695166	111021.991229	16535.990325
2020-12-31	21811.751812	21826.119052	21796.889787	21812.155606	5.742400	129237.684380	21811.782847
2021-01-31	34554.125793	34594.169353	34512.497779	34554.252479	10.253061	352510.183906	34552.337249
2021-02-28	46077.343214	46117.833367	46036.025287	46077.905341	5.965070	274360.971284	46075.783298
2021-03-31	54500.773165	54536.467642	54465.302911	54501.997804	3.555766	193360.583713	54499.282182

112 rows × 7 columns

In [5]:
```python
display(series.head(n = 10))
```

Timestamp	Weighted_Price
2011-12-31	4.471603
2012-01-31	6.345955
2012-02-29	5.228443
2012-03-31	4.984397
2012-04-30	4.995091
2012-05-31	5.046848
2012-06-30	6.047198
2012-07-31	7.907613
2012-08-31	10.984670
2012-09-30	11.435692

In [6]:
```python
display(series.tail(n = 10))
```

Timestamp	Weighted_Price
2020-06-30	9459.783762
2020-07-31	9558.816690
2020-08-31	11637.963222
2020-09-30	10656.147579
2020-10-31	11844.141987
2020-11-30	16535.990325
2020-12-31	21811.782847
2021-01-31	34552.337249
2021-02-28	46075.783298
2021-03-31	54499.282182

Obviously, we can observe a "slightly" increase in price over time

Statistical description of the time series

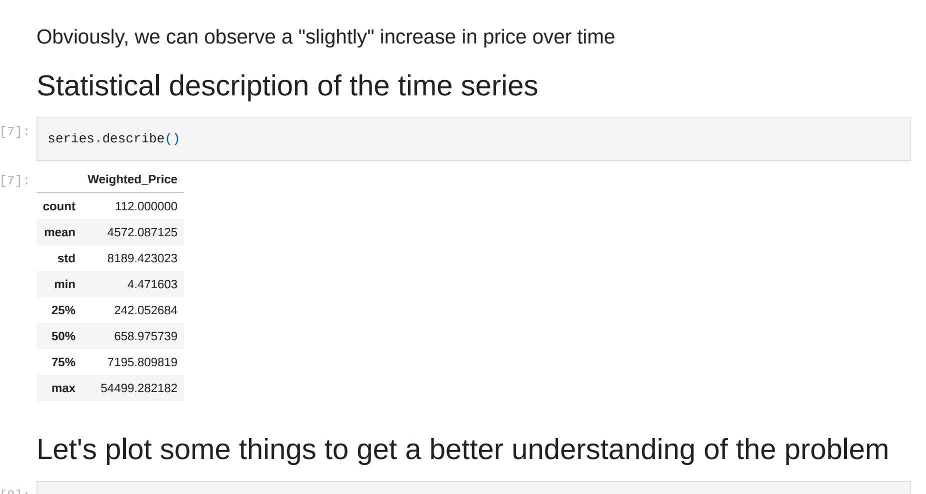
In [7]:
```python
series.describe()
```

Out[7]:

	Weighted_Price
count	112.000000
mean	4572.087125
std	8189.423023
min	4.471603
25%	242.052684
50%	658.975739
75%	7195.809819
max	54499.282182

Let's plot some things to get a better understanding of the problem

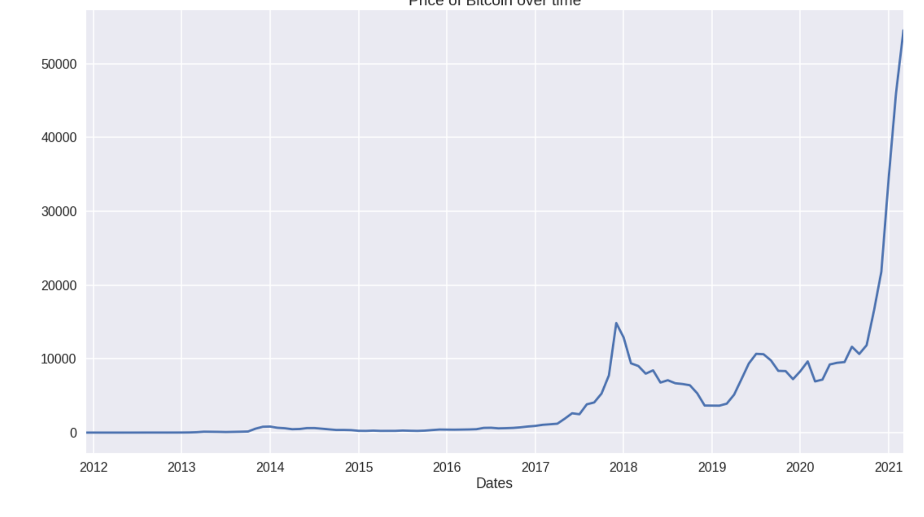
In [8]:
```python
plt.style.use('seaborn-poster')
plt.figure(figsize = (18, 10))
plt.title("Price of Bitcoin over time")
series['Weighted_Price'].plot()
plt.xlabel("Dates")
plt.show()
```



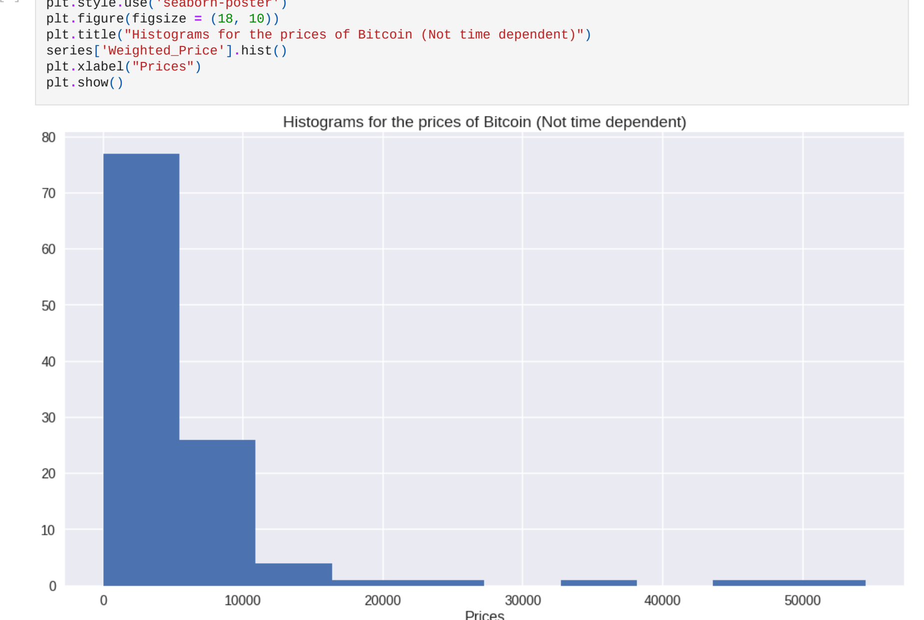
It seems that we have big increase in the price of Bitcoin starting by the end of 2020

Removing the time component from the problem, at least for the moment

In [9]:
```python
plt.style.use('seaborn-poster')
plt.figure(figsize = (18, 10))
plt.title("Histograms for the prices of Bitcoin (Not time dependent)")
series['Weighted_Price'].hist()
plt.xlabel("Prices")
plt.show()
```



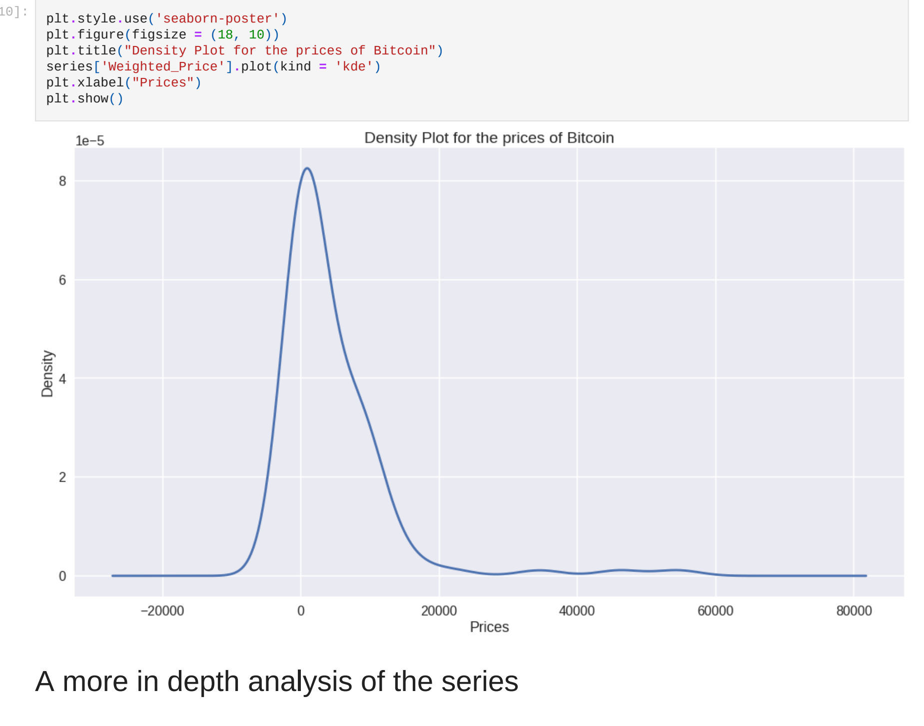
In [10]:
```python
plt.style.use('seaborn-poster')
plt.figure(figsize = (18, 10))
plt.title("Density Plot for the prices of Bitcoin")
series['Weighted_Price'].plot(kind = 'kde')
plt.xlabel("Prices")
plt.show()
```



A more in depth analysis of the series

Seasonal Decomposition

In [11]:
```python
decomposition = seasonal_decompose(series['Weighted_Price'], model = 'additive')
rcParams['figure.figsize'] = 18, 15
decomposition.plot()
plt.show()
```



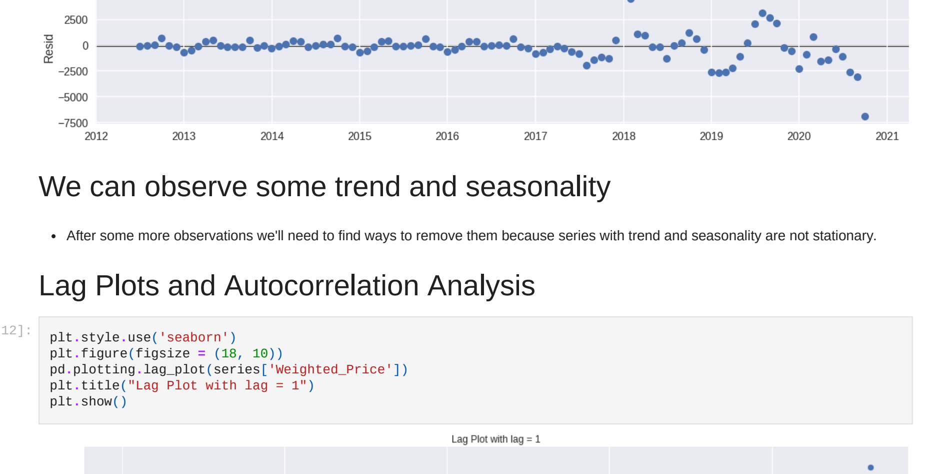
We can observe some trend and seasonality

- After some more observations we'll need to find ways to remove them because series with trend and seasonality are not stationary.

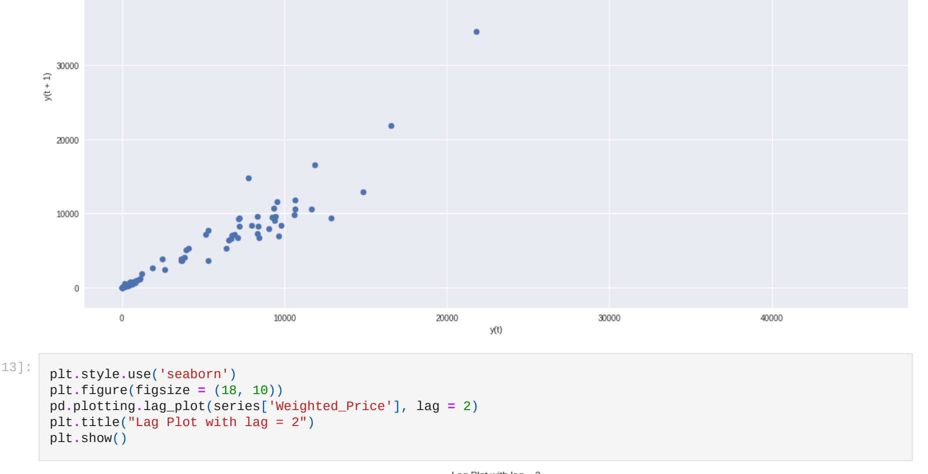
Lag Plots and Autocorrelation Analysis

In [12]:
```python
plt.style.use('seaborn')
plt.figure(figsize = (18, 10))
pd.plotting.lag_plot(series['Weighted_Price'])
plt.title("Lag Plot with lag = 1")
plt.show()
```

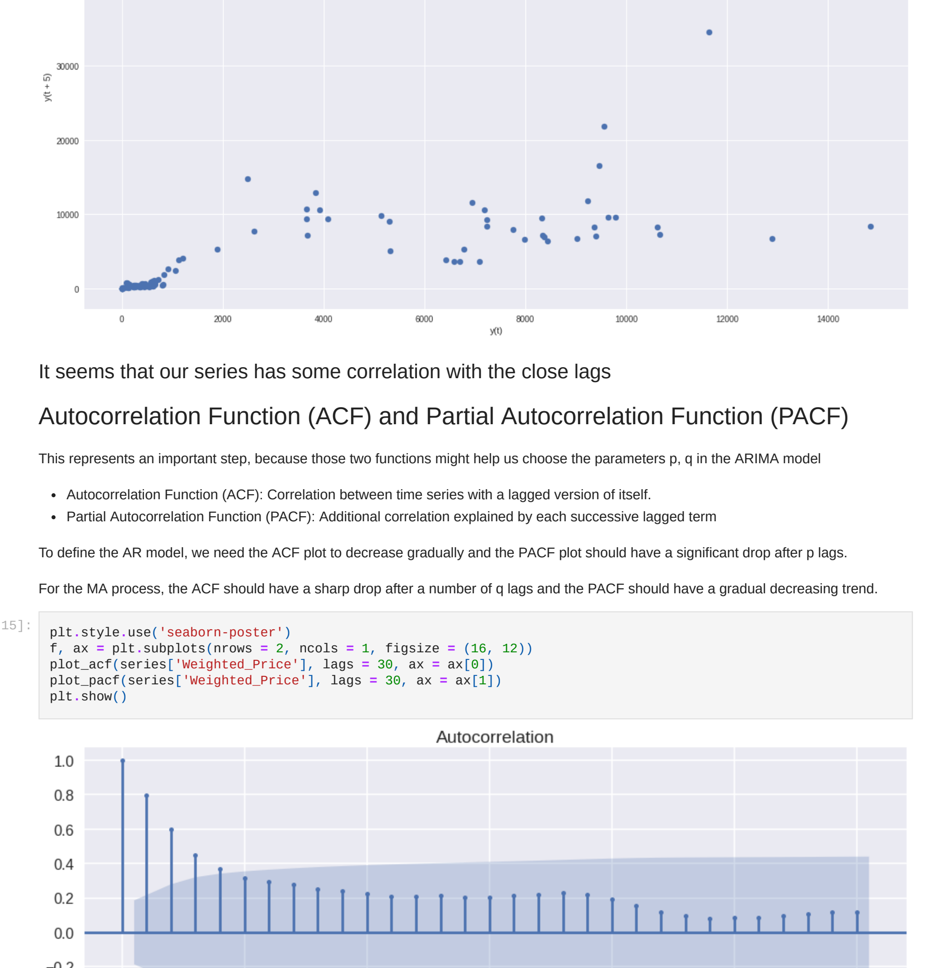
In [13]:
```python
plt.style.use('seaborn')
plt.figure(figsize = (18, 10))
pd.plotting.lag_plot(series['Weighted_Price'], lag = 2)
plt.title("Lag Plot with lag = 2")
plt.show()
```



In [14]:
```python
plt.style.use('seaborn')
plt.figure(figsize = (18, 10))
pd.plotting.lag_plot(series['Weighted_Price'], lag = 5)
plt.title("Lag Plot with lag = 5")
plt.show()
```



[illegible]



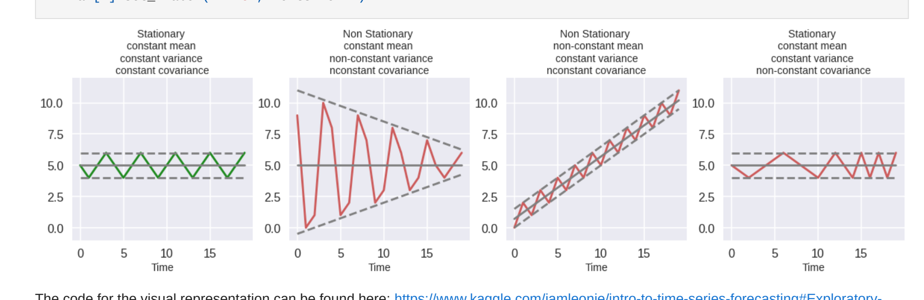
[illegible]

```python
[illegible]
ax[1].lines[2].set_linestyle("--")
ax[1].lines[3].set_linestyle("--")
ax[1].set_title(f'Non Stationary \nconstant mean \n non-constant variance\nnconstant covariance', fontsize=14)

nonstationary2 = [0, 2, 1, 3, 2, 4, 3, 5, 4, 6, 5, 7, 6, 8, 7, 9, 8, 10, 9, 11,]
sns.lineplot(x=t, y=nonstationary2, ax=ax[2], color='indianred' )
sns.lineplot(x=t, y=t*0.5+0.7, ax=ax[2], color='grey')
sns.lineplot(x=t, y=t*0.5, ax=ax[2], color='grey')
sns.lineplot(x=t, y=t*0.5+1.5, ax=ax[2], color='grey')
ax[2].lines[2].set_linestyle("--")
ax[2].lines[3].set_linestyle("--")
ax[2].set_title(f'Non Stationary \n non-constant mean\nconstant variance\nnconstant covariance', fontsize=14)

nonstationary3 = [5, 4.5, 4, 4.5, 5, 5.5, 6, 5.5, 5, 4.5, 4, 5, 6, 5, 4, 6, 4, 6, 4, 6,]
sns.lineplot(x=t, y=nonstationary3, ax=ax[3], color='indianred')
sns.lineplot(x=t, y=5, ax=ax[3], color='grey')
sns.lineplot(x=t, y=6, ax=ax[3], color='grey')
sns.lineplot(x=t, y=4, ax=ax[3], color='grey')
ax[3].lines[2].set_linestyle("--")
ax[3].lines[3].set_linestyle("--")
ax[3].set_title(f'Stationary \nconstant mean \nconstant variance \nnon-constant covariance', fontsize=14)

for i in range(4):
    ax[i].set_ylim([-1, 12])
    ax[i].set_xlabel('Time', fontsize=14)
```



The code for the visual representation can be found here: https://www.kaggle.com/iamleonie/intro-to-time-series-forecasting#Exploratory-Data-Analysis

Testing for Stationarity

Usually, testing for stationarity can be done in three main ways:

- visual approach: plot the series and check for any trend or seasonality
- statistics over rolling windows: iterate through series with a rolling window and compare mean, variance and covariance
- hypothesis testing: Augmented Dickey Fuller Test

In this project we will use for stationarity testing only Augmented Dickey Fuller Test

Augmented Dickey Fuller (ADF) Test

Is a statistical test called a unit root test. Unit roots being a possible cause for non-stationarity.

-**Null Hypothesis (H0):** Time series has a unit root. (The series is **not stationary**)

-**Alternate Hypothesis (H1):** Time series has no unit root. (The series is **stationary**)

If we succeed in rejecting the null hypothesis, we can confirm that our time series is stationary.

Ways To Reject the null hypothesis in ADF Test:

The null hypothesis can be rejected if the test statistic is less than the critical value.

ADF statistic > critical value ⟹ Fail to reject the null hypothesis, the series is non-stationary.

ADF statistic <= critical value ⟹ Reject the null hypothesis, the series is stationary.

The null hypothesis can be rejected if the p-value is below a set significance level.

p-value > significance level (default: 0.05) : Fail to reject the null hypothesis, the series is non-stationary.

p-value < significance level (default: 0.05) : Reject the null hypothesis, the series is stationary.

The idea for presenting this section came from: https://www.kaggle.com/iamleonie/intro-to-time-series-forecasting#Exploratory-Data-Analysis

In [17]:
```python
def stationarity_test(series):
    test = adfuller(series, autolag = 'AIC')
    results = pd.Series(test[0 : 4], \
              index = ['ADF Statistic', 'p-value','#Lags Used','Number of Observations Used'])

    for key,value in test[4].items():
        results['Critical Value (%s)'%key] = value

    print("ADF Results")
    display(results)
```

In [18]:
```python
stationarity_test(series['Weighted_Price'])
```

```
ADF Results
ADF Statistic                    2.143886
p-value                          0.998833
#Lags Used                       1.000000
Number of Observations Used    110.000000
Critical Value (1%)             -3.491245
Critical Value (5%)             -2.888195
Critical Value (10%)            -2.580988
dtype: float64
```

As we assumed, our time series is not stationary

To solve this problem we can:

- Differencing: e.g. removing trend, seasonality
- Transformations: e.g. box-cox transform, log, square root, from that we can stabilize non-constant variance

Box Cox Transformation and Removing Seasonality

Why Would We Want to Transform Our Data?

The Box-Cox transformation transforms our data so that it closely resembles a normal distribution.

"In many statistical techniques, we assume that the errors are normally distributed. This assumption allows us to construct confidence intervals and conduct hypothesis tests. By transforming your target variable, we can (hopefully) normalize our errors (if they are not already normal).

Additionally, transforming our variables can improve the predictive power of our models because transformations can cut away white noise."

https://towardsdatascience.com/box-cox-transformation-explained-51d745e34203

In [19]:
```python
series['Weighted_Price_Box_Transform'], lmbda = stats.boxcox(series['Weighted_Price'])
```

In [20]:
```python
stationarity_test(series['Weighted_Price_Box_Transform'])
```

```
ADF Results
ADF Statistic                   -0.293579
p-value                          0.926409
#Lags Used                       1.000000
Number of Observations Used    110.000000
Critical Value (1%)             -3.491245
Critical Value (5%)             -2.888195
Critical Value (10%)            -2.580988
dtype: float64
```

Our data it's still not stationary

Let's apply a 12 shift differencing to remove possible annual seasonality

In [21]:
```python
series['Weighted_Price_Box_Seasonal_Diff'] = series.Weighted_Price_Box_Transform - series.Weighted_Price_Box_Tr
```

In [22]:
```python
display(series)
```

Timestamp	Weighted_Price	Weighted_Price_Box_Transform	Weighted_Price_Box_Seasonal_Diff
2011-12-31	4.471603	1.633738	NaN
2012-01-31	6.345955	2.057681	NaN
2012-02-29	5.228443	1.821004	NaN
2012-03-31	4.984397	1.763401	NaN
2012-04-30	4.995091	1.765977	NaN
...	...	...	...
2020-11-30	16535.990325	17.816633	2.001607
2020-12-31	21811.782847	18.671514	3.249221
2021-01-31	34552.337249	20.152969	4.341819
2021-02-28	46075.783298	21.120280	4.891492
2021-03-31	54499.282182	21.699465	6.395992

112 rows × 3 columns

In [23]:
```python
stationarity_test(series['Weighted_Price_Box_Seasonal_Diff'][12:])
```

```
ADF Results
ADF Statistic                  -1.678562
p-value                         0.442190
#Lags Used                     12.000000
Number of Observations Used    87.000000
Critical Value (1%)            -3.507853
Critical Value (5%)            -2.895382
Critical Value (10%)           -2.584824
dtype: float64
```

As we can observe from the p-value the series it's still not stationary

Will try a first order differencing to remove the trend of the series

In [24]:
```python
series['Weighted_Price_Box_Diff'] = series.Weighted_Price_Box_Seasonal_Diff - series.Weighted_Price_Box_Seasona
```

In [25]:
```python
display(series)
```

Timestamp	Weighted_Price	Weighted_Price_Box_Transform	Weighted_Price_Box_Seasonal_Diff	Weighted_Price_Box_Diff
2011-12-31	4.471603	1.633738	NaN	NaN
2012-01-31	6.345955	2.057681	NaN	NaN
2012-02-29	5.228443	1.821004	NaN	NaN
2012-03-31	4.984397	1.763401	NaN	NaN
2012-04-30	4.995091	1.765977	NaN	NaN
...	...	...	...	...
2020-11-30	16535.990325	17.816633	2.001607	1.005741
2020-12-31	21811.782847	18.671514	3.249221	1.247615
2021-01-31	34552.337249	20.152969	4.341819	1.092598
2021-02-28	46075.783298	21.120280	4.891492	0.549673
2021-03-31	54499.282182	21.699465	6.395992	1.504500

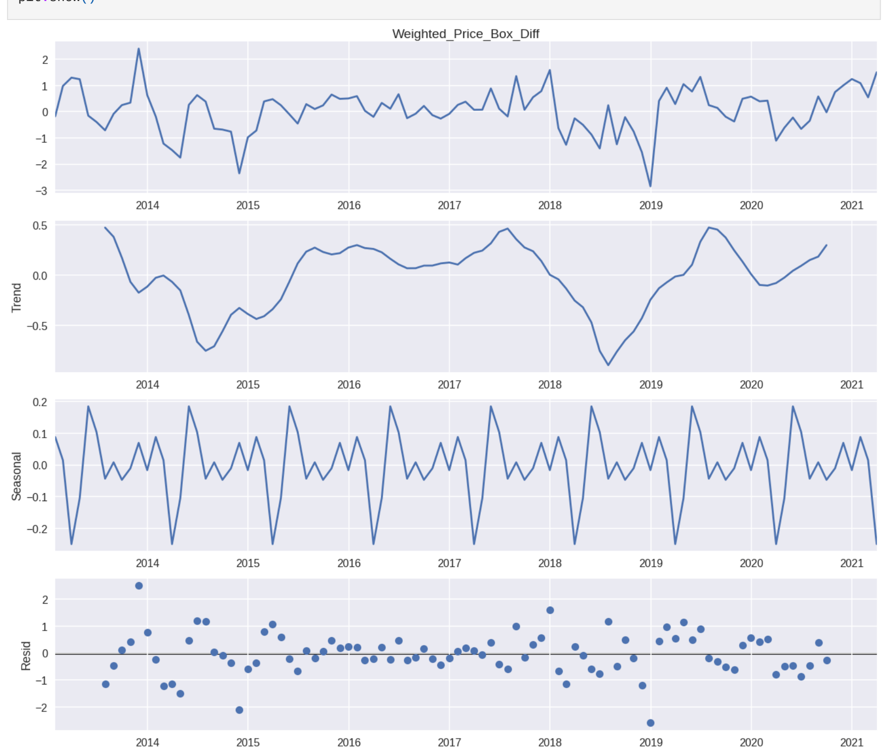
112 rows × 4 columns

In [26]:
```python
stationarity_test(series['Weighted_Price_Box_Diff'][13:])
```

```
ADF Results
ADF Statistic                  -4.972855
p-value                         0.000025
#Lags Used                     11.000000
Number of Observations Used    87.000000
Critical Value (1%)            -3.507853
Critical Value (5%)            -2.895382
Critical Value (10%)           -2.584824
dtype: float64
```

As wee can observe the p-value close to 0, we can be certain that the time series it's stationary

In [27]:
```python
decomposition = seasonal_decompose(series['Weighted_Price_Box_Diff'][13:], model = 'additive')
rcParams['figure.figsize'] = 18, 15
decomposition.plot()
plt.show()
```

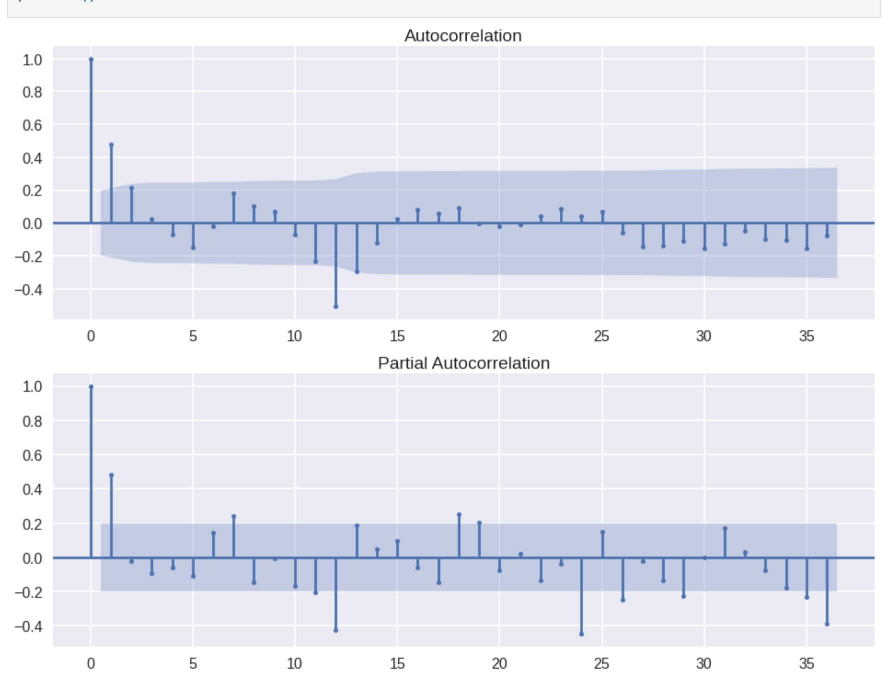


Our series still has a strong seasonality

- This will be the main reason for us to use a Seasonal ARIMA model instead of a standard ARIMA model

Let's check again the autocorrelation function and the partial autocorrelation function

In [28]:
```python
plt.style.use('seaborn-poster')
f, ax = plt.subplots(nrows = 2, ncols = 1, figsize = (16, 12))
plot_acf(series['Weighted_Price_Box_Diff'][13:], lags = 36, ax = ax[0])
plot_pacf(series['Weighted_Price_Box_Diff'][13:], lags = 36, ax = ax[1])
plt.show()
```



Time Series Modelling

Based on the above plots we can select the p and q parameters for the Seasonal ARIMA model

For the AR(p) Model

- Based on the PACF plot we can observe that there are strong correlation with the lags: 1, 7, 12.

There are more lags that can be considered for the p parameter, those might be caused by some trace of seasonality, but for the moment we will consider only the values selected above

For the MA(q) Model

- Based on the ACF plot we can observe that there are strong correlation with the lags: 1, 2, 12.

Because we are using a Seasonal ARIMA Model we can remove the value 12 from our hyperparameters because this will be used as S = the order of seasonality

For the I(d) Model

- From our analysis, we saw that to achieve stationarity we need to apply a single difference by one shift.
- This concludes that our orders of differencing (d, D) should be equal to 1.

For testing "stand alone" AR(p) and MA(q) will add 0 for possible values of p and q

Model Validation

- Based on the extracted parameters from PACF and ACF plots we can try to select the best model from all possible combinations.
- During this step will try to optimize as a performance metric the mean of the AIC estimator and the MSE errors of the predictions of the validation folds.
- The reason for that it's because the AIC estimator it's used to find the balance between the model complexity (the number of parameters) and the how well the model fits our data, but this does not neccesary represent how good our model is able to adapt to future forecasts. This will be the main reason on why we need a validation set.
- To get a more robust perspective of the the parameters we will use the AIC estimator averaged over k folds of time series split (more details here: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.TimeSeriesSplit.html)
- Some of those combinations might not be valid due to a convergence problem of the model, in that case we will simply pass that combination

In [29]:
```python
def inverse_transform(y, lmbda):
    if lmbda == 0:
        return(np.exp(y))
    else:
        return(np.exp(np.log(lmbda * y + 1) / lmbda))
```

In [30]:
```python
tscv = TimeSeriesSplit(n_splits = 3)

Ps = [0, 1, 7]
ps = [0, 1, 7]
Qs = [0, 1, 2]
qs = [0, 1, 2]

D = 1
d = 1
S = 12
parameters = list(product(ps, qs, Ps, Qs))

tic = time.time()
results = []
best_metric = np.inf
best_combination, best_model = None, None
for combination in parameters:
    p, q, P, Q = combination[0], combination[1], combination[2], combination[3]
    print("Combination: p = {}, d = {}, q = {}, P = {}, D = {}, Q = {}".format(p, d, q, P, D, Q))
    order = (p, d, q)
    seasonal_order = (P, D, Q, S)

    try:
        model_aics = []
        mean_squared_errors = []
        for step, (train_idx, valid_idx) in enumerate(tscv.split(series['Weighted_Price_Box_Transform'])):
            train_series = series['Weighted_Price_Box_Transform'][train_idx]
            valid_series = series['Weighted_Price'][valid_idx]

            model = SARIMAX(train_series, order = order, seasonal_order = seasonal_order)
            model_fit = model.fit()

            forecast = model_fit.predict(start = valid_series.index[0], end = valid_series.index[-1])
            predictions = inverse_transform(forecast, lmbda)
            error = np.sqrt(mean_squared_error(predictions, valid_series))

            model_aics.append(model_fit.aic)
            mean_squared_errors.append(error)

        model_aic   = np.mean(model_aics)
        model_error = np.mean(mean_squared_errors)
        average     = 0.5 * model_aic + 0.5 * model_error

        metric = average

        if metric < best_metric:
            best_metric = metric
            best_combination = combination

        results.append([combination, model_aic, model_error, average])

    except:
        pass

toc = time.time()
print("Time for the hyperparameters search: {}'s".format(toc - tic))
```

```
Combination: p = 0, d = 1, q = 0, P = 0, D = 1, Q = 0
Combination: p = 0, d = 1, q = 0, P = 0, D = 1, Q = 1
Combination: p = 0, d = 1, q = 0, P = 0, D = 1, Q = 2
Combination: p = 0, d = 1, q = 0, P = 1, D = 1, Q = 0
Combination: p = 0, d = 1, q = 0, P = 1, D = 1, Q = 1
Combination: p = 0, d = 1, q = 0, P = 1, D = 1, Q = 2
Combination: p = 0, d = 1, q = 0, P = 7, D = 1, Q = 0
Combination: p = 0, d = 1, q = 0, P = 7, D = 1, Q = 1
Combination: p = 0, d = 1, q = 0, P = 7, D = 1, Q = 2
Combination: p = 0, d = 1, q = 1, P = 0, D = 1, Q = 0
Combination: p = 0, d = 1, q = 1, P = 0, D = 1, Q = 1
Combination: p = 0, d = 1, q = 1, P = 0, D = 1, Q = 2
Combination: p = 0, d = 1, q = 1, P = 1, D = 1, Q = 0
Combination: p = 0, d = 1, q = 1, P = 1, D = 1, Q = 1
Combination: p = 0, d = 1, q = 1, P = 1, D = 1, Q = 2
Combination: p = 0, d = 1, q = 1, P = 7, D = 1, Q = 0
Combination: p = 0, d = 1, q = 1, P = 7, D = 1, Q = 1
Combination: p = 0, d = 1, q = 1, P = 7, D = 1, Q = 2
Combination: p = 0, d = 1, q = 2, P = 0, D = 1, Q = 0
Combination: p = 0, d = 1, q = 2, P = 0, D = 1, Q = 1
Combination: p = 0, d = 1, q = 2, P = 0, D = 1, Q = 2
Combination: p = 0, d = 1, q = 2, P = 1, D = 1, Q = 0
Combination: p = 0, d = 1, q = 2, P = 1, D = 1, Q = 1
Combination: p = 0, d = 1, q = 2, P = 1, D = 1, Q = 2
Combination: p = 0, d = 1, q = 2, P = 7, D = 1, Q = 0
Combination: p = 0, d = 1, q = 2, P = 7, D = 1, Q = 1
Combination: p = 0, d = 1, q = 2, P = 7, D = 1, Q = 2
Combination: p = 1, d = 1, q = 0, P = 0, D = 1, Q = 0
Combination: p = 1, d = 1, q = 0, P = 0, D = 1, Q = 1
Combination: p = 1, d = 1, q = 0, P = 0, D = 1, Q = 2
Combination: p = 1, d = 1, q = 0, P = 1, D = 1, Q = 0
Combination: p = 1, d = 1, q = 0, P = 1, D = 1, Q = 1
Combination: p = 1, d = 1, q = 0, P = 1, D = 1, Q = 2
Combination: p = 1, d = 1, q = 0, P = 7, D = 1, Q = 0
Combination: p = 1, d = 1, q = 0, P = 7, D = 1, Q = 1
Combination: p = 1, d = 1, q = 0, P = 7, D = 1, Q = 2
Combination: p = 1, d = 1, q = 1, P = 0, D = 1, Q = 0
Combination: p = 1, d = 1, q = 1, P = 0, D = 1, Q = 1
Combination: p = 1, d = 1, q = 1, P = 0, D = 1, Q = 2
Combination: p = 1, d = 1, q = 1, P = 1, D = 1, Q = 0
Combination: p = 1, d = 1, q = 1, P = 1, D = 1, Q = 1
Combination: p = 1, d = 1, q = 1, P = 1, D = 1, Q = 2
Combination: p = 1, d = 1, q = 1, P = 7, D = 1, Q = 0
Combination: p = 1, d = 1, q = 1, P = 7, D = 1, Q = 1
Combination: p = 1, d = 1, q = 1, P = 7, D = 1, Q = 2
Combination: p = 1, d = 1, q = 2, P = 0, D = 1, Q = 0
Combination: p = 1, d = 1, q = 2, P = 0, D = 1, Q = 1
Combination: p = 1, d = 1, q = 2, P = 0, D = 1, Q = 2
Combination: p = 1, d = 1, q = 2, P = 1, D = 1, Q = 0
Combination: p = 1, d = 1, q = 2, P = 1, D = 1, Q = 1
Combination: p = 1, d = 1, q = 2, P = 1, D = 1, Q = 2
Combination: p = 1, d = 1, q = 2, P = 7, D = 1, Q = 0
Combination: p = 1, d = 1, q = 2, P = 7, D = 1, Q = 1
Combination: p = 1, d = 1, q = 2, P = 7, D = 1, Q = 2
Combination: p = 7, d = 1, q = 0, P = 0, D = 1, Q = 0
Combination: p = 7, d = 1, q = 0, P = 0, D = 1, Q = 1
Combination: p = 7, d = 1, q = 0, P = 0, D = 1, Q = 2
Combination: p = 7, d = 1, q = 0, P = 1, D = 1, Q = 0
Combination: p = 7, d = 1, q = 0, P = 1, D = 1, Q = 1
Combination: p = 7, d = 1, q = 0, P = 1, D = 1, Q = 2
Combination: p = 7, d = 1, q = 0, P = 7, D = 1, Q = 0
Combination: p = 7, d = 1, q = 0, P = 7, D = 1, Q = 1
Combination: p = 7, d = 1, q = 0, P = 7, D = 1, Q = 2
Combination: p = 7, d = 1, q = 1, P = 0, D = 1, Q = 0
Combination: p = 7, d = 1, q = 1, P = 0, D = 1, Q = 1
Combination: p = 7, d = 1, q = 1, P = 0, D = 1, Q = 2
Combination: p = 7, d = 1, q = 1, P = 1, D = 1, Q = 0
Combination: p = 7, d = 1, q = 1, P = 1, D = 1, Q = 1
Combination: p = 7, d = 1, q = 1, P = 1, D = 1, Q = 2
Combination: p = 7, d = 1, q = 1, P = 7, D = 1, Q = 0
Combination: p = 7, d = 1, q = 1, P = 7, D = 1, Q = 1
Combination: p = 7, d = 1, q = 1, P = 7, D = 1, Q = 2
Combination: p = 7, d = 1, q = 2, P = 0, D = 1, Q = 0
Combination: p = 7, d = 1, q = 2, P = 0, D = 1, Q = 1
Combination: p = 7, d = 1, q = 2, P = 0, D = 1, Q = 2
Combination: p = 7, d = 1, q = 2, P = 1, D = 1, Q = 0
Combination: p = 7, d = 1, q = 2, P = 1, D = 1, Q = 1
Combination: p = 7, d = 1, q = 2, P = 1, D = 1, Q = 2
Combination: p = 7, d = 1, q = 2, P = 7, D = 1, Q = 0
Combination: p = 7, d = 1, q = 2, P = 7, D = 1, Q = 1
Combination: p = 7, d = 1, q = 2, P = 7, D = 1, Q = 2
Time for the hyperparameters search: 1516.154671907425's
```

In [31]:
```python
results = pd.DataFrame(results, columns = ['Parameters', 'AIC', "MSE Errors", "0.5 * AIC + 0.5 * MSE"])
display(results.sort_values(by = 'AIC', ascending=True).head(n = 10))
display(results.sort_values(by = 'MSE Errors', ascending=True).head(n = 10))
[illegible]
```

[illegible]

	Parameters	AIC	MSE Errors	0.5 * AIC + 0.5 * MSE
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
29	(1, 0, 0, 2)	80.543263	6288.375644	3184.459454
10	(0, 1, 0, 1)	82.359510	6329.867285	3206.113397
1	(0, 0, 0, 1)	88.266813	6363.465340	3225.866076
38	(1, 1, 0, 2)	82.331046	6425.699217	3254.015131
20	(0, 2, 0, 2)	81.700908	6507.385532	3294.543220
11	(0, 1, 0, 2)	81.817365	6514.629496	3298.223431
2	(0, 0, 0, 2)	87.086850	6564.569973	3325.828411

Model Analysis

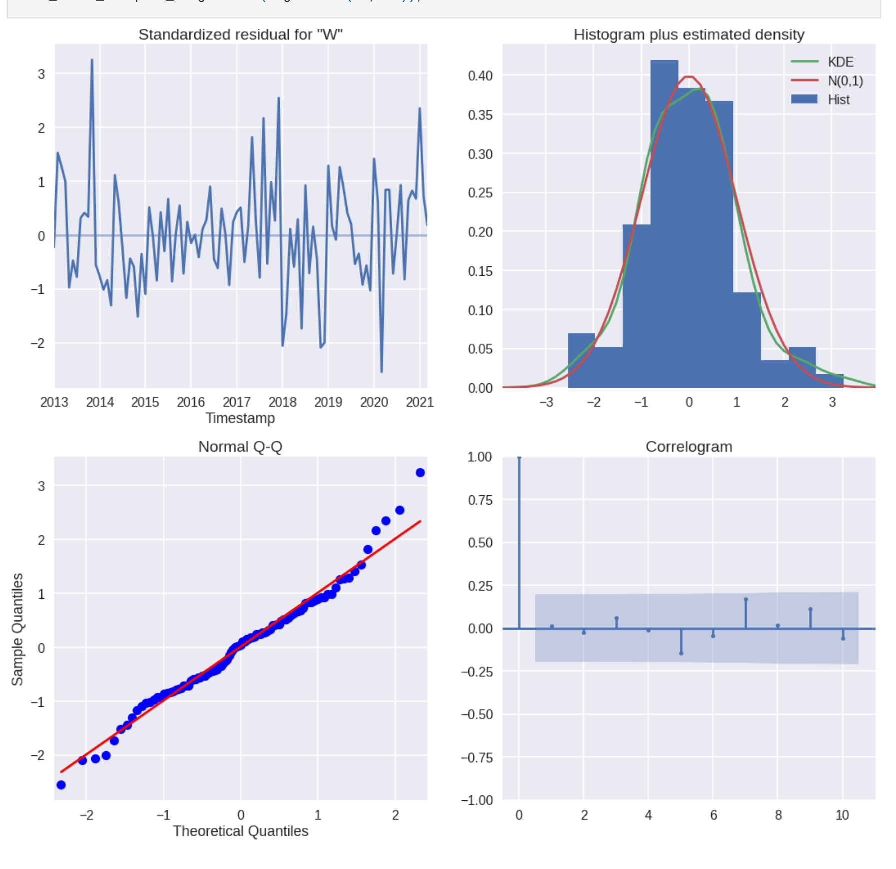
- We will analyze the standardized residuals of our predictions based on the Autocorrelation plots, checking the Density Function, Normal Q-Q Plots

In [32]:
```python
p, q, P, Q = best_combination[0], best_combination[1], best_combination[2], best_combination[3]

order = (p, 1, q)
seasonal_order = (P, 1, Q, 12)

best_model = SARIMAX(series['Weighted_Price_Box_Transform'], order = order, seasonal_order = seasonal_order)
best_model_fit = best_model.fit()
```

In [33]:
```python
plt.style.use('seaborn-poster')
best_model_fit.plot_diagnostics(figsize = (18, 17));
```



From our residuals we can conclude that our model is well fitted

- From the autocorrelation plot of the standardized residuals we can observe that we have no significant lags
- We can see that our residuals are normally distributed with 0 mean
- And again from the Quantile-Quantile Plot we can observe how our residuals point fit well the quantiles of a normal distribution

Predicting the Future

In [34]:
```python
timestamps = pd.date_range(start = series.index[-1], periods = 30, freq = 'M')
original   = series[['Weighted_Price']]
future     = pd.DataFrame(index = timestamps, columns = original.columns)

predicted = pd.concat([original, future])
predicted['Forecast'] = inverse_transform(best_model_fit.predict(start = predicted.index[0], end = predicted.ir

plt.figure(figsize=(18, 10))
predicted['Weighted_Price'].plot(color = 'blue', label = 'Original Series')
predicted['Forecast'].plot(color='r', ls='--', label = 'Predicted Series')
plt.title('Bitcoin Price by Month')
plt.ylabel('USD Dollars')
plt.legend(loc = 'best')
plt.show()
```