Estimating Monthly Power Prices Using Time Series Models

Vladas Alesius

IBM Machine Learning Professional Certificate

Specialized Models: Time Series And Survival Analysis

Coursera

2022-01-03

Main Objective

The objective of this analysis is to find out which time series prediction model does the best job predicting monthly energy prices for 2020 using the prices from 2000 to 2019 as the training dataset. Our focus is to select the model with the best *accuracy*, which will be assessed with Root Mean Squared Error (RMSE) as a metric. The models will be Exponential Smoothing, ARIMA and Simple Recurrent Neural Network (Simple RNN).

For Exponential Smoothing and ARIMA, 2000-2019 prices will be used as the training dataset. However, the prices from 2000 to 2018 will be used to train the Simple RNN model.

In this study, Exponential Smoothing and ARIMA models will include seasonality assumptions, in particular - that monthly prices have a 12-month seasonality. However, as we will see below, there is no clear trend pattern, so the seasonality pattern might be more complex than we assume.

Data Description

According to Wikipedia, "Nord Pool AS is a European power exchange owned by Euronext and the continental Nordic and Baltic countries' Transmission system operators (TSOs). Nord Pool delivers power trading across Europe. Nord Pool offers day-ahead and intraday trading, clearing and settlement, data and compliance, as well as consultancy services. More than 360 customers trade on Nord Pool today." Nord Pool website will be used as a source for historic System (SYS) monthly energy prices (euros per megawatt hour - EUR/MWh) that will be used to forecast monthly prices for 2021.

We will start with importing necessary modules and functions:

```
In [2]:
import pandas as pd
import seaborn as sns
from datetime import datetime
from scipy.stats import normaltest, norm, uniform, boxcox
from pmdarima.preprocessing import BoxCoxEndogTransformer
import statsmodels.api as sm
from statsmodels.tsa.statespace.sarimax import SARIMAX
import pmdarima as pm
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf, month_plot, quarter_plot
from statsmodels.tsa.seasonal import seasonal_decompose
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.api import ExponentialSmoothing
from sklearn.metrics import mean_squared_error
from math import sqrt
import keras
from keras.models import Sequential
from keras.layers import Dense, SimpleRNN, LSTM, Activation, Dropout
```

Historic monthly prices from 2000 to 2020 will be extracted from Nord Pool website using the links below:

```
In [4]:
price_links = [
    "https://www.nordpoolgroup.com/4abfef/globalassets/marketdata-excel-files/elspot-prices_2000_monthly_eur.xl
    "https://www.nordpoolgroup.com/48e48a/globalassets/marketdata-excel-files/elspot-prices_2001_monthly_eur.xl
    "https://www.nordpoolgroup.com/498947/globalassets/marketdata-excel-files/elspot-prices_2002_monthly_eur.xl
    "https://www.nordpoolgroup.com/48df7f/globalassets/marketdata-excel-files/elspot-prices_2003_monthly_eur.xl
    "https://www.nordpoolgroup.com/48c924/globalassets/marketdata-excel-files/elspot-prices_2004_monthly_eur.xl
    "https://www.nordpoolgroup.com/48c912/globalassets/marketdata-excel-files/elspot-prices_2005_monthly_eur.xl
    "https://www.nordpoolgroup.com/48c90b/globalassets/marketdata-excel-files/elspot-prices_2006_monthly_eur.xl
    "https://www.nordpoolgroup.com/48c908/globalassets/marketdata-excel-files/elspot-prices_2007_monthly_eur.xl
    "https://www.nordpoolgroup.com/48c904/globalassets/marketdata-excel-files/elspot-prices_2008_monthly_eur.xl
    "https://www.nordpoolgroup.com/48c900/globalassets/marketdata-excel-files/elspot-prices_2009_monthly_eur.xl
    "https://www.nordpoolgroup.com/48c8fc/globalassets/marketdata-excel-files/elspot-prices_2010_monthly_eur.xl
    "https://www.nordpoolgroup.com/48c8f8/globalassets/marketdata-excel-files/elspot-prices_2011_monthly_eur.xl
    "https://www.nordpoolgroup.com/48c8f3/globalassets/marketdata-excel-files/elspot-prices_2012_monthly_eur.xl
    "https://www.nordpoolgroup.com/48c8ef/globalassets/marketdata-excel-files/elspot-prices_2013_monthly_eur.xl
    "https://www.nordpoolgroup.com/48c8eb/globalassets/marketdata-excel-files/elspot-prices_2014_monthly_eur.xl
    "https://www.nordpoolgroup.com/48c8e8/globalassets/marketdata-excel-files/elspot-prices_2015_monthly_eur.xl
    "https://www.nordpoolgroup.com/48c8e3/globalassets/marketdata-excel-files/elspot-prices_2016_monthly_eur.xl
    "https://www.nordpoolgroup.com/48c904/globalassets/marketdata-excel-files/elspot-prices_2017_monthly_eur.xl
    "https://www.nordpoolgroup.com/48c900/globalassets/marketdata-excel-files/elspot-prices_2018_monthly_eur.xl
    "https://www.nordpoolgroup.com/48c8fc/globalassets/marketdata-excel-files/elspot-prices_2019_monthly_eur.xl
    "https://www.nordpoolgroup.com/48c8f8/globalassets/marketdata-excel-files/elspot-prices_2020_monthly_eur.xl
]
```

Below, the data is split into training and testing subsets. For Exponential Smoothing and ARIMA models, the training data will contain values from 2000 to 2019, and the testing data - prices for 2020.

```
In [5]:
price_train = pd.DataFrame()
price_test = pd.DataFrame()
price_total = pd.DataFrame()
for i in range(len(price_links)):
    prices = pd.read_html(price_links[i], header=2, decimal=',', thousands='.')[0]
    prices["Month"] = '20' + prices["Unnamed: 0"].str[:2] + '-' + prices["Unnamed: 0"].str[-3:]
    prices.set_index(["Month"], inplace=True)
    prices.index = pd.to_datetime(prices.index)
    prices = prices[["SYS"]]
    price_total=pd.concat([price_total, prices], axis=0)
    if i<len(price_links)-1:
        price_train=pd.concat([price_train, prices], axis=0)
    else:
        price_test=pd.concat([price_test, prices], axis=0)
```

The final training dataset consists of a month index and SYS prices from 2000 to 2019:

```
In [4]:
price_train
```

Out[4]:

	SYS
Month	
2000-01-01	16.22
2000-02-01	12.89
2000-03-01	11.78
2000-04-01	12.80
2000-05-01	9.51
...	...
2019-08-01	36.11
2019-09-01	32.92
2019-10-01	37.10
2019-11-01	42.15
2019-12-01	36.79

240 rows × 1 columns

Data Exploration

As we can see from the plot and summary tables below, monthly energy prices fluctuate a lot, from less than 10 EUR/MWh in 2000 to more than 80 EUR/MWh in 2010. The average monthly price is about 34 EUR/MWh, and the standard deviation is about 13 EUR/MWh:

```
In [5]:
price_train.describe()
```

Out[5]:

	SYS
count	240.000000
mean	33.679708
std	12.877149
min	6.350000
25%	25.317500
50%	31.690000
75%	40.550000
max	81.650000

```
In [6]:
print("Minimum Price")
price_train.loc[price_train.idxmin(), :]
```

```
Minimum Price
```

Out[6]:

	SYS
Month	
2000-07-01	6.35

```
In [7]:
print("Maximum Price")
price_train.loc[price_train.idxmax(), :]
```

```
Maximum Price
```

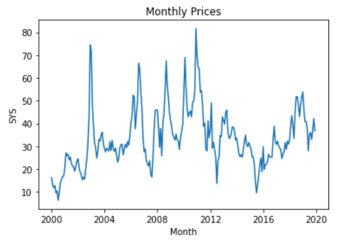
Out[7]:

	SYS
Month	
2010-12-01	81.65

The plot below shows sharp price swings throughout the years:

```
In [22]:
sns.lineplot(data=price_train, y='SYS', x=price_train.index).set_title("Monthly Prices")
```

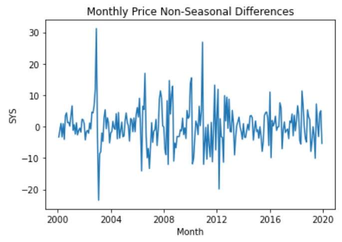
```
Out[22]:
Text(0.5, 1.0, 'Monthly Prices')
```



Also, average price levels change throughout the years, which might indicate a need to difference data to make it stationary. The plot below shows that average price levels are more constant for the differenced data, although extremely high and low values for some months still remain:

```
In [23]:
sns.lineplot(data=price_train.diff(), y='SYS', x=price_train.index).set_title("Monthly Price Non-Seasonal Diffe
```

```
Out[23]:
Text(0.5, 1.0, 'Monthly Price Non-Seasonal Differences')
```



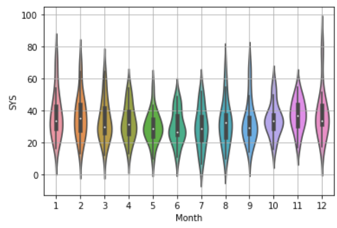
Monthly price differences fluctuate around 0, but the highest absolute differences are for increases:

```
In [10]:
price_train.diff().mean()
```

Out[10]:
```
SYS    0.086067
dtype: float64
```

The violin plot below demonstrates that although the highest price values are for winter months, typical price values are not much different throughout the year. So our initial assumption of 12-month seasonality might be put into question.

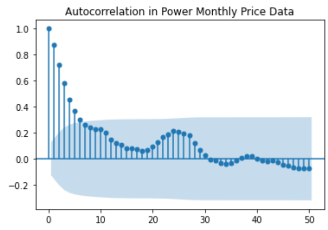
In [12]:
```
# violinplot of months to determine variance and range
sns.violinplot(x=price_train.index.month, y=price_train.SYS)
plt.grid(b=True);
```

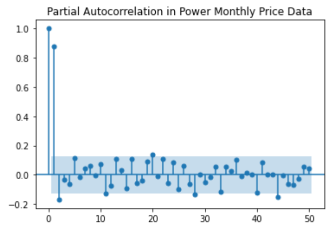


Below, autocorrelation and partial autocorrelation functions are plotted for the training dataset. Autocorrelation increases for lags 20-30 again confirm that the initial data is not stationary.

In [34]:
```
print('Monthly data Autocorrelation Plots')
# Autocorrelation and Partial Autocorrelation Functions for Monthly Data
acf_plot_2 = plot_acf(price_train['SYS'], lags=50, title='Autocorrelation in Power Monthly Price Data')
pacf_plot_2 = plot_pacf(price_train['SYS'], lags=50, title='Partial Autocorrelation in Power Monthly Price Data
```

```
Monthly data Autocorrelation Plots
```

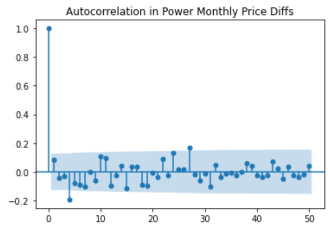

Autocorrelation in Power Monthly Price Data

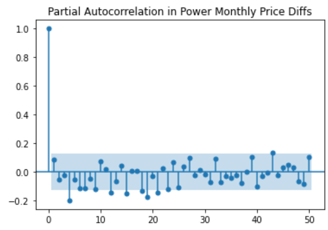

Partial Autocorrelation in Power Monthly Price Data

The plots below show that for non-seasonally differenced data, both ACF and PACF values are small and patternless. This is a sign of not over-differencing. However, the plots below do not help to select the best number of either AR or MA components.

In [25]:
```
print('Monthly data diff Autocorrelation Plots')
# Autocorrelation and Partial Autocorrelation Functions for Monthly Data Diffs
acf_plot_2 = plot_acf(price_train['SYS'].diff().dropna(), lags=50, title='Autocorrelation in Power Monthly Pric
pacf_plot_2 = plot_pacf(price_train['SYS'].diff().dropna(), lags=50, title='Partial Autocorrelation in Power Mo
```

```
Monthly data diff Autocorrelation Plots
```


Autocorrelation in Power Monthly Price Diffs


Partial Autocorrelation in Power Monthly Price Diffs

Augmented Dickey-Fuller test confirms that differenced monthly prices are stationary:

In [13]:
```
adf, pvalue, usedlag, nobs, critical_values, icbest = adfuller(price_train['SYS'].diff().dropna())
```

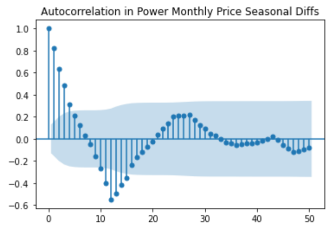
In [14]:
```
print(pvalue)
```

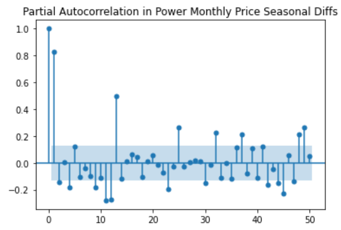
```
2.869552662687494e-12
```

Seasonal differencing seems to be unnecessary, since it produces strongly negative autocorrelation values (which indicates overdifferencing):

In [26]:
```
print('Monthly data seasonal diff Autocorrelation Plots')
# Autocorrelation and Partial Autocorrelation Functions for Monthly Data Seasonal Diffs
acf_plot_2 = plot_acf(price_train['SYS'].diff(periods=12).dropna(), lags=50, title='Autocorrelation in Power Mo
pacf_plot_2 = plot_pacf(price_train['SYS'].diff(periods=12).dropna(), lags=50, title='Partial Autocorrelation i
```

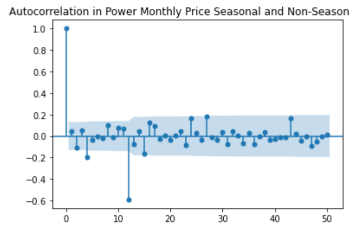
```
Monthly data seasonal diff Autocorrelation Plots
```

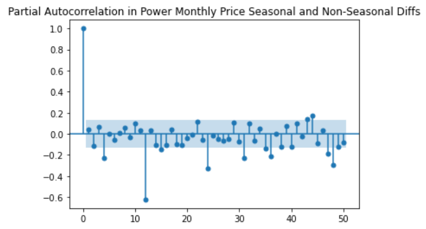

Autocorrelation in Power Monthly Price Seasonal Diffs


Partial Autocorrelation in Power Monthly Price Seasonal Diffs

In [27]:
```
print('Monthly data Autocorrelation Plots')
# Autocorrelation and Partial Autocorrelation Functions for Monthly Data Seasonal and Non-Seasonal Diffs
acf_plot_2 = plot_acf(price_train['SYS'].diff().diff(periods=12).dropna(), lags=50, title='Autocorrelation in P
pacf_plot_2 = plot_pacf(price_train['SYS'].diff().diff(periods=12).dropna(), lags=50, title='Partial Autocorrel
```

```
Monthly data Autocorrelation Plots
```


Autocorrelation in Power Monthly Price Seasonal and Non-Seasonal Diffs


Partial Autocorrelation in Power Monthly Price Seasonal and Non-Seasonal Diffs

The distribution of prices is approximately bell shaped, although somewhat skewed:

In [34]:
```
price_train['SYS'].hist().set_title("Distribution Of Monthly Prices")
```

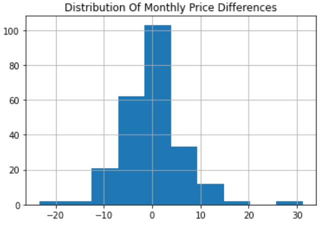
Out[34]:
```
Text(0.5, 1.0, 'Distribution Of Monthly Prices')
```


Distribution Of Monthly Prices

Differences are distributed more symmetrically, however:

In [35]:
```
price_train['SYS'].diff().hist().set_title("Distribution Of Monthly Price Differences")
```

Out[35]:
```
Text(0.5, 1.0, 'Distribution Of Monthly Price Differences')
```



Model Training

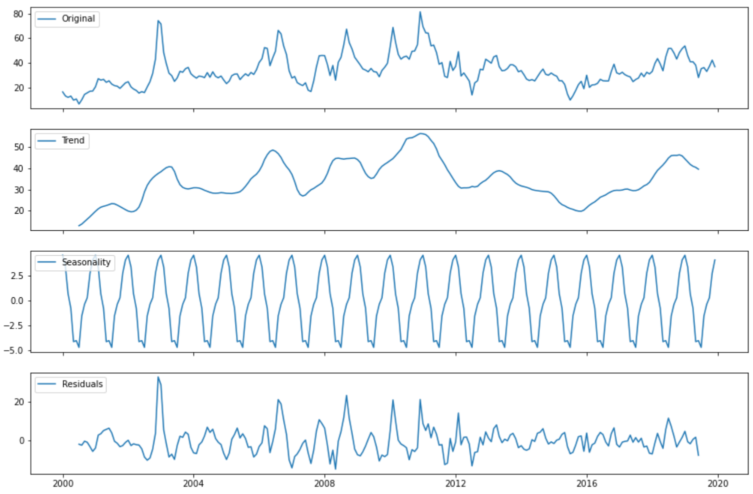
As described above, three different models will be examined - Exponential Smoothing, ARIMA and Simple Recurrent Neural Network. Each of them will be used to predict monthly power prices for 2020, and their efficiency will be assessed using Root Mean Squared Error (RMSE) as a metric.

Exponential Smoothing

For Exponential Smoothing model, we will begin with a naive data decomposition (using moving averages) into three parts: trend, seasonality, and residual. We assume an additive model and 12-month seasonality. The components are plotted below.

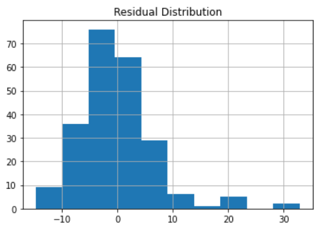
In [37]:
```
ss_decomposition = seasonal_decompose(x=price_train['SYS'], model='additive', period=12)
estimated_trend = ss_decomposition.trend.dropna()
estimated_seasonal = ss_decomposition.seasonal.dropna()
estimated_residual = ss_decomposition.resid.dropna()
```

In [38]:
```
fig, axes = plt.subplots(4, 1, sharex=True, sharey=False)
fig.set_figheight(10)
fig.set_figwidth(15)

axes[0].plot(price_train['SYS'], label='Original')
axes[0].legend(loc='upper left');

axes[1].plot(estimated_trend, label='Trend')
axes[1].legend(loc='upper left');

axes[2].plot(estimated_seasonal, label='Seasonality')
axes[2].legend(loc='upper left');

axes[3].plot(estimated_residual, label='Residuals')
axes[3].legend(loc='upper left');
```



The residuals tend to be more asymmetric to the positive side - there are some values above 20. This might be a reason to rethink our assumptions, but we will stick to them for now.

In [39]:
```
pd.Series(estimated_residual).hist().set_title("Residual Distribution");
```



The average residual value is still almost 0:

In [40]:
```
pd.Series(estimated_residual).mean()
```

Out[40]:
```
0.007050438596493289
```

The model is fit below - we assume an additive model for both trend and seasonal components, and 12-month seasonality:

In [10]:
```
triple = ExponentialSmoothing(price_train["SYS"],
                              trend="additive",
                              seasonal="additive",
                              seasonal_periods=12, freq="MS").fit(optimized=True)
triple_preds = triple.forecast(len(price_test))
triple_preds_df = pd.DataFrame(triple_preds, columns=["SYS"], index=price_test.index)
```

```
C:\Users\Lenovo\anaconda3\lib\site-packages\statsmodels\tsa\holtwinters\model.py:427: FutureWarning: After 0.13
initialization must be handled at model creation
  warnings.warn(
```

Model summary is printed below. Coefficient values show that the forecasts mainly depend on seasonal components, and the trend has only a minor impact. Seasonal component values show that middle year prices are expected to be lower:

In [11]:
```
triple.summary()
```

Out[11]:

ExponentialSmoothing Model Results

Dep. Variable:	SYS	**No. Observations:**	240
Model:	ExponentialSmoothing	**SSE**	8937.826
Optimized:	True	**AIC**	900.178
Trend:	Additive	**BIC**	955.868
Seasonal:	Additive	**AICC**	903.273
Seasonal Periods:	12	**Date:**	Sat, 01 Jan 2022
Box-Cox:	False	**Time:**	17:14:12
Box-Cox Coeff.:	None		

	coeff	**code**	**optimized**
smoothing_level	0.9999389	alpha	True
smoothing_trend	2.6875e-07	beta	True
smoothing_seasonal	8.9648e-16	gamma	True
initial_level	36.573169	l.0	True
initial_trend	0.0873898	b.0	True
initial_seasons.0	-20.524345	s.0	True
initial_seasons.1	-21.791752	s.1	True
initial_seasons.2	-24.371459	s.2	True
initial_seasons.3	-25.720214	s.3	True
initial_seasons.4	-29.048464	s.4	True
initial_seasons.5	-28.900650	s.5	True
initial_seasons.6	-29.590462	s.6	True
initial_seasons.7	-26.487169	s.7	True
initial_seasons.8	-25.499835	s.8	True
initial_seasons.9	-24.634499	s.9	True
initial_seasons.10	-22.008141	s.10	True
initial_seasons.11	-21.009110	s.11	True

Price predictions for 2020 are printed below:

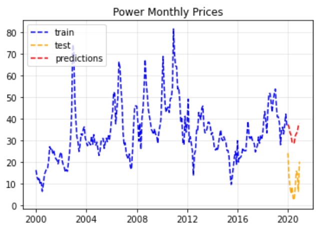
In [12]:
```
triple_preds_df
```

Out[12]:

	SYS
Month	
2020-01-01	37.362548
2020-02-01	36.182531
2020-03-01	33.690214
2020-04-01	32.428849
2020-05-01	29.187989
2020-06-01	29.423193
2020-07-01	28.820771
2020-08-01	32.011453
2020-09-01	33.086177
2020-10-01	34.038903
2020-11-01	36.752651
2020-12-01	37.839072

The plot below compares testing subset prices and price predictions - the latter are much higher, which shows that the model has done not a very good job in predicting:

In [33]:
```
plt.plot(price_train.index, price_train['SYS'], 'b--', label="train")
plt.plot(price_test.index, price_test, color='orange', linestyle="--", label="test")
plt.plot(price_test.index, triple_preds_df['SYS'], 'r--', label="predictions")
plt.legend(loc='upper left')
plt.title("Power Monthly Prices")
plt.grid(alpha=0.3);

plt.show()
```



Root Mean Squared Error (RMSE) for Exponential Smoothing is almost 23, which is higher than most test subset price values. This again shows that model predictive power is not high:

```
In [34]: sqrt(mean_squared_error(price_test, triple_preds_df['SYS']))
```

```
Out[34]: 22.987055200850136
```

ARIMA

The next model is AutoRegressive Integrated Moving Average (ARIMA). It has three parameters: p, d and q. p is the order (number of time lags) of the autoregressive model, d is the degree of differencing, and q is the order of the moving-average model. Seasonal ARIMA models are usually denoted ARIMA(p,d,q)(P,D,Q)m, where m refers to the number of periods in each season, and the uppercase P, D, Q refer to the autoregressive, differencing, and moving average terms for the seasonal part of the ARIMA model.

As we saw in Data Exploration part, neither ACF not PACF plot give clear hints on how many AR and MA components are the best. So this time, we will rely on automatic parameter search.

For this purpose, we will use auto_arima function from pmdarima package. We will assume 12-month seasonality as previously and will use the last 12 month values from the training dataset to assess each parameter combination. Akaike Information Criterion value for the best model is printed as well:

```
In [35]: stepwise_model = pm.auto_arima(price_train["SYS"],
                                        m=12,
                                        max_order=None, seasonal=True,
                                        trace=False,
                                        error_action='ignore',
                                        suppress_warnings=True,
                                        stepwise=False, out_of_sample=12)
         print(stepwise_model.aic())
```

```
1552.7962449122297
```

Please see the best model parameters below:

```
In [36]: stepwise_model
```

```
Out[36]: ARIMA(order=(2, 1, 1), scoring_args={}, seasonal_order=(0, 0, 2, 12),
               suppress_warnings=True)
```

As predicted above, one non-seasonal difference is necessary to make the series stationary, and no seasonal differences are needed.

In order to perform detailed model diagnostics, this model will be refit using statsmodels package. As we can see from the summary below, the coefficient value for 12-month seasonal lags is not statistically significant, which shows that our assumption about yearly seasonality might be dubious:

```
In [6]: sar = sm.tsa.statespace.SARIMAX(price_train["SYS"],
                                         order=(2,1,1),
                                         seasonal_order=(0,0,2,12),
                                         trend='c', freq='MS').fit()
        sar.summary()
```

```
C:\Users\Lenovo\anaconda3\lib\site-packages\statsmodels\tsa\base\tsa_model.py:524: ValueWarning: No frequency i
nformation was provided, so inferred frequency MS will be used.
  warnings.warn('No frequency information was'
```

Out[6]:

SARIMAX Results

Dep. Variable:	SYS	**No. Observations:**	240
Model:	SARIMAX(2, 1, 1)x(0, 0, [1, 2], 12)	**Log Likelihood**	-769.398
Date:	Mon, 03 Jan 2022	**AIC**	1552.796
Time:	12:19:13	**BIC**	1577.131
Sample:	01-01-2000	**HQIC**	1562.603
	- 12-01-2019		
Covariance Type:	opg		

	coef	**std err**	**z**	**P>|z|**	**[0.025**	**0.975]**
intercept	0.0174	0.022	0.786	0.432	-0.026	0.061
ar.L1	0.9900	0.067	14.863	0.000	0.859	1.121
ar.L2	-0.1892	0.065	-2.914	0.004	-0.316	-0.062
ma.L1	-0.9564	0.036	-26.403	0.000	-1.027	-0.885
ma.S.L12	-0.0820	0.084	-0.976	0.329	-0.247	0.083
ma.S.L24	0.1522	0.070	2.166	0.030	0.014	0.290
sigma2	36.4156	2.022	18.008	0.000	32.452	40.379

Ljung-Box (L1) (Q):	0.02	**Jarque-Bera (JB):**	319.79
Prob(Q):	0.88	**Prob(JB):**	0.00
Heteroskedasticity (H):	0.57	**Skew:**	1.00
Prob(H) (two-sided):	0.01	**Kurtosis:**	8.30

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

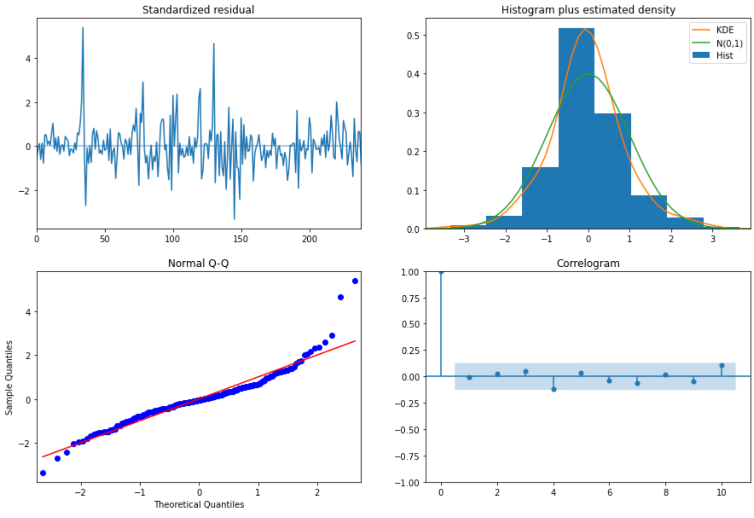
Below some statistical tests are run on this model. Test p-value show that standardized residuals are not normally distributed, and their standard deviations depend on their values. These issues show that the model should be improved. However, there is no serial correlation between the standardized residuals, which is good.

```
In [40]: # create and run statistical tests on model
         norm_val, norm_p, skew, kurtosis = sar.test_normality('jarquebera')[0]
         lb_val, lb_p = sar.test_serial_correlation(method='ljungbox',)[0]
         het_val, het_p = sar.test_heteroskedasticity('breakvar')[0]


         # we want to look at largest lag for Ljung-Box, so take largest number in series
         # there's intelligence in the method to determine how many lags back to calculate this stat
         lb_val = lb_val[-1]
         lb_p = lb_p[-1]
         durbin_watson = sm.stats.stattools.durbin_watson(
             sar.filter_results.standardized_forecasts_error[0, sar.loglikelihood_burn:])

         print('Normality: val={:.3f}, p={:.3f}'.format(norm_val, norm_p));
         print('Ljung-Box: val={:.3f}, p={:.3f}'.format(lb_val, lb_p));
         print('Heteroskedasticity: val={:.3f}, p={:.3f}'.format(het_val, het_p));
         print('Durbin-Watson: d={:.2f}'.format(durbin_watson))
```

```
Normality: val=319.789, p=0.000
Ljung-Box: val=26.374, p=0.334
Heteroskedasticity: val=0.574, p=0.014
Durbin-Watson: d=2.01
C:\Users\Lenovo\anaconda3\lib\site-packages\statsmodels\tsa\statespace\mlemodel.py:3220: FutureWarning: The def
ault value of lags is changing.  After 0.12, this value will become min(10, nobs//5) for non-seasonal time seri
es and min (2*m, nobs//5) for seasonal time series. Directly set lags to silence this warning.
  warnings.warn(
```

The diagnostic plots below confirm the previous insights about the standardized model residuals. Firstly, they are not symmetric - there are larger positive spikes than the negative ones. Also, they deviate from standard normal distribution, especially on the positive side. However, they are uncorrelated (which is good).

```
In [41]: stepwise_model.plot_diagnostics(figsize=(15,10));
```



ARIMA model predictions for 2020 are printed below:

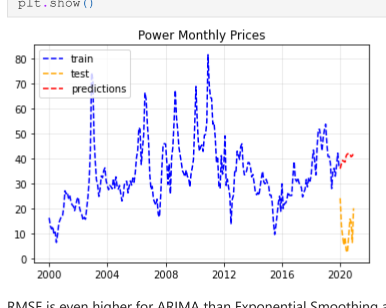
```python
auto_arima_fc = stepwise_model.predict(len(price_test))
auto_arima_fc_df = pd.DataFrame(auto_arima_fc, index=price_test.index, columns=["SYS"])
```

```python
auto_arima_fc_df
```

Month	SYS
2020-01-01	35.933251
2020-02-01	37.984997
2020-03-01	39.250499
2020-04-01	38.907914
2020-05-01	38.572125
2020-06-01	41.158508
2020-07-01	41.870328
2020-08-01	41.977700
2020-09-01	41.919301
2020-10-01	40.717496
2020-11-01	41.137459
2020-12-01	42.286845

Test subset price values and ARIMA predictions are compared below. As for Exponential Smoothing, predictions are much higher than actual 2020 prices. However, Exponential Smoothing predicted lower mid-year values (the same price fall was actually observed), while ARIMA predictions tend to moderately increase throughout the year:

```python
plt.plot(price_train.index, price_train['SYS'], 'b--', label="train")
plt.plot(price_test.index, price_test, color='orange', linestyle="--", label="test")
plt.plot(price_test.index, auto_arima_fc_df['SYS'], 'r--', label="predictions")
plt.legend(loc='upper left')
plt.title("Power Monthly Prices")
plt.grid(alpha=0.3);

plt.show()
```



RMSE is even higher for ARIMA than Exponential Smoothing and exceeds 30. This shows that selected ARIMA model predictive power is even lower than for Exponential Smoothing:

```python
sqrt(mean_squared_error(price_test, auto_arima_fc_df['SYS']))
```

```
30.138921149172827
```

Deep Learning

The final model is a Simple Recurrent Neural Network (RNN). Its purpose is to automatically learn how to model series components like seasonality and autocorrelation. Python library **keras** will be used for it.

The following functions are taken from Deep Learning Demo notebook (and slightly adjusted). Firstly, we define a function to convert a series to a format suitable for Keras:

```python
def get_keras_format_series(series):
    """
    Convert a series to a numpy array of shape
    [n_samples, time_steps, features]
    """

    series = np.array(series)
    return series.reshape(series.shape[0], series.shape[1], 1)
```

The below function splits the total dataset into training and testing subsets. For the training subset, many training samples will be generated and added to train_X list, each sample of a selected length (in our case, they will be of length 12), and the following price value will be added to train_y list. A sample gap is used to make the samples not follow each other. For the testing subset, the same number of first values (12 in our case) will be used as X-s and used for the model based prediction, and the rest - as y-s.

```python
def get_train_test_data(df, series_name, input_months,
                        test_months, sample_gap=3):
    """
    Utility processing function that splits a monthly time series into
    train and test with keras-friendly format, according to user-specified
    choice of shape.

    arguments
    ---------
    df (dataframe): dataframe with time series columns
    series_name (string): column name in df
    input_months (int): length of sequence input to network
    test_months (int): length of held-out terminal sequence
    sample_gap (int): step size between start of train sequences; default 3

    returns
    ---------
    tuple: train_X, test_X_init, train_y, test_y
    """

    train = df[series_name][:-test_months] # training data is remaining months until amount of test_months
    test = df[series_name][-test_months:] # test data is the remaining test_months

    train_X, train_y = [], []

    # range 0 through # of train samples - input_months by sample_gap.
    for i in range(0, train.shape[0]-input_months, sample_gap):
        train_X.append(train[i:i+input_months]) # each training sample is of length input months
        train_y.append(train[i+input_months]) # each y is just the next step after training sample

    train_X = get_keras_format_series(train_X) # format our new training set to keras format
    train_y = np.array(train_y) # make sure y is an array to work properly with keras

    # The set that we had held out for testing (must be same length as original train input)
    test_X_init = test[:input_months]
    test_y = test[input_months:] # test_y is remaining values from test set

    return train_X, test_X_init, train_y, test_y
```

As already mentioned, training sequences will be of 12 month data each, and the testing subset will consist of the last two years. The testing subset will again be split into two parts, 12 months each, the first of which will be used as input:

```python
input_months = 12
test_months = 24

train_X, test_X_init, train_y, test_y = \
    (get_train_test_data(price_total, 'SYS', input_months, test_months))
```

A Simple RNN model with one layer is defined and fit below:

```python
def fit_SimpleRNN(train_X, train_y, cell_units, epochs):
    """
    Fit Simple RNN to data train_X, train_y

    arguments
    ---------
    train_X (array): input sequence samples for training
    train_y (list): next step in sequence targets
    cell_units (int): number of hidden units for RNN cells
    epochs (int): number of training epochs
    """

    # initialize model
    model = Sequential()

    # construct an RNN layer with specified number of hidden units
    # per cell and desired sequence input format
    model.add(SimpleRNN(cell_units, input_shape=(train_X.shape[1],1)))

    # add an output layer to make final predictions
    model.add(Dense(1))

    # define the loss function / optimization strategy, and fit
    # the model with the desired number of passes over the data (epochs)
    model.compile(loss='mean_squared_error', optimizer='adam')
    model.fit(train_X, train_y, epochs=epochs, verbose=0)

    return model
```

```python
model = fit_SimpleRNN(train_X, train_y, cell_units=50, epochs=50000)
```

The model above was trained to predict only one future time step. For multi-step forecasting, we will iteratively generate one prediction, append it to the end of the input sequence (and shift that sequence forward by one step), then feed the new sequence back to the model. We stop once we have generated all the time step predictions we need. In our case, the first 12 values of testing subset will be used as the initial set, and predictions will be generated for the rest 12 months one by one, shifting the input sequence.

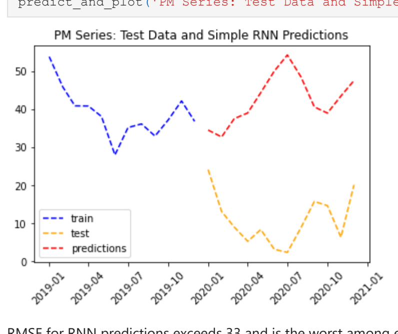
```python
def predict(X_init, n_steps, model):
    """
    Given an input series matching the model's expected format,
    generates model's predictions for next n_steps in the series
    """

    X_init = np.array(X_init).copy().reshape(1,-1,1)
    preds = []

    # iteratively take current input sequence, generate next step pred,
    # and shift input sequence forward by a step (to end with latest pred).
    # collect preds as we go.
    for _ in range(n_steps):
        pred = model.predict(X_init)
        preds.append(pred)
        X_init[:,:-1,:] = X_init[:,1:,:] # replace first 11 values with 2nd through 12th
        X_init[:,-1,:] = pred # replace 12th value with prediction

    preds = np.array(preds).reshape(-1,1)

    return preds
```

```python
y_preds_rnn = predict(test_X_init, len(test_y), model)
```

Finally, a comparison of RNN predictions and actual data is plotted below. True values are much lower than the predictions. Furthermore, RNN prediced a price increase for the mid-year when the price actually fell:

```python
def predict_and_plot(title):
    """
    Plots model's predictions against the ground truth

    arguments
    ---------
    title (string): plot title
    """

    #using our ranges we plot X_init
    plt.plot(test_X_init.index, test_X_init, color='blue', linestyle="--")
    #and test and actual preds
    plt.plot(test_y.index, test_y, color='orange', linestyle="--")
    plt.plot(test_y.index, y_preds_rnn, color='red', linestyle='--')

    plt.title(title)
    plt.legend(['train','test','predictions'])
    plt.xticks(rotation=45)
```

```python
predict_and_plot('PM Series: Test Data and Simple RNN Predictions')
```



RMSE for RNN predictions exceeds 33 and is the worst among examined models:

```python
sqrt(mean_squared_error(test_y, y_preds_rnn))
```

```
33.59413182852007
```

The model summary below shows that it had more than 2000 parameters, so still not very complex:

```python
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn (SimpleRNN)       (None, 50)                2600
_________________________________________________________________
dense (Dense)                (None, 1)                 51
=================================================================
Total params: 2,651
Trainable params: 2,651
Non-trainable params: 0
_________________________________________________________________
```

Final Recommendation

In this analysis, we tried running three time series prediction models for monthly energy price forecast: Exponential Smoothing, ARIMA and Recurrent Neural Network (RNN). Exponential Smoothing forecasts are the most in line with the actual values, and the model took the least time to train. RNN took the longest to model and provided the worst results in terms of RMSE, so it is the worst model of the three. ARIMA is in between in both regards.

So the best model in our case is **Exponential Smoothing**.

Summary Key Findings and Insights

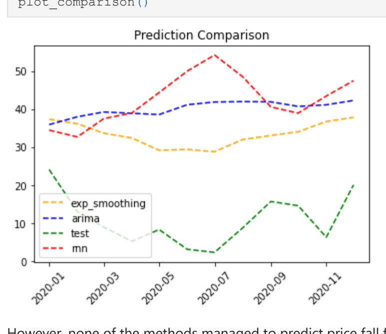
As we discussed above, Exponential Smoothing forecast was the best for monthly energy price forecating. Its RMSE is the lowest - almost 23. Also, it took the shortest to train - Auto ARIMA tries many models to select the best one, and RNN takes some time depending on numbers of cells and epochs.

The plot below compares the forecasts obtained with all three prediction methods and actual 2020 monthly price values. As we can see, actual data is much lower than for predictions. However, Exponential Smoothing values tend to be the lowest (and thus the closest to the actual data), and they go down in the middle of year - the same as for the actual data. Other forecasting methods failed to predict this. So this is one more argument for Exponential Smoothing usage.

ARIMA forecast is pretty flat throughout 2020, and RNN forecast *increases* in the middle of year - the opposite direction than it should be.

```python
def plot_comparison():
    plt.plot(test_y.index, triple_preds_df['SYS'], color='orange', linestyle="--")
    plt.plot(test_y.index, auto_arima_fc_df['SYS'], color='blue', linestyle="--")
    plt.plot(test_y.index, test_y, color='green', linestyle="--")
    plt.plot(test_y.index, y_preds_rnn, color='red', linestyle='--')

    plt.title("Prediction Comparison")
    plt.legend(['exp_smoothing','arima','test','rnn'])
    plt.xticks(rotation=45)
```

```python
plot_comparison()
```



However, none of the methods managed to predict price fall for 2020, compared to previous years. This might be due to a multi-year cyclical pattern that forecasting did not include, or simply because of prices changing randomly.

Next Steps

Some possible model improvement steps are described below.

We made an assumption about yearly seasonality, but this did not help to forecast a price downturn for 2020. Adding a cyclical component can be useful for long-term forecasting.

Exponential Smoothing model was trained based on certain assumptions, such as additive trend and seasonality. Experimenting with multiplicative trend and seasonality might be worthy as well.

ARIMA model was trained using auto_arima. It took some time, since many different models were tried to find the best one. However, some human judgement might also be useful to determine the final model. So relying on experimentation might help to come up with a different (simpler, or maybe even more efficient) model.

Simple RNN model could be modified changing a number of cell units and using more training epochs. Furthermore, since RNN model weights are initialized randomly, they depend on the selected random seed in our case. Trying a different random seed might yield different results, and probably a more effective model. Apart from that, we used default parameters (the activation function, initializers etc.) - trying to change these might be useful as well.

Also, instead of a Simple RNN model, an LSTM model can be used - experience shows that they are better at predicting more complex data patterns. However, it has more parameters, so takes more time to train.