Autoregressiver integrierter gleitender Durchschnitt (ARIMA)

Der autoregressive integrierte gleitende Durchschnitt (ARIMA) ist eine statistische Methode, die ein Autoregressionsmodell für Zeitreihendaten verwendet, um auf der Grundlage historischer Daten zukünftige Prognosen abzuleiten. ARIMA kombiniert zwei Hauptmethoden: ein autoregressives Modell und ein gleitendes Durchschnittsmodell. Die Autoregressionsmethode lernt eine lineare Gleichung zwischen aktuellen und vergangenen Werten und verwendet sie, um zukünftige Prognosen der Variablen zu erstellen. Ein autoregressives Modell der Ordnung p AR(p) ist gegeben durch:

$$y_t = C + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \ldots + \phi_p y_{t-p} + \varepsilon_t$$

Dabei ist ε_t das weiße Rauschen, ϕ die Koeffizienten der linearen Gleichung. P ist die Reihenfolge einer Autoregression und stellt die Anzahl der vorhergehenden Werte der Variablen dar, die zur Erstellung zukünftiger Prognosen verwendet werden.

Andererseits verwendet ein gleitendes Durchschnittsmodell eine lineare Kombination von Prognosefehlern für seine Vorhersagen. Die folgende Gleichung erklärt ein gleitendes Durchschnittsmodell der Ordnung p:

$$y_t = C + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \theta_2 \varepsilon_{t-2} + \ldots + \theta_q \varepsilon_{t-q}$$

Dabei ist θ die Koeffizienten der linearen Gleichung.

Wenn die Differenzierung zwischen aufeinanderfolgenden Beobachtungen zusammen mit der Autoregression und einem gleitenden Durchschnittsmodell erfolgt, erhalten wir ein nicht-saisonales ARIMA-Modell. Das ARIMA-Modell kann mit der folgenden Gleichung dargestellt werden:

$$y'_t = C + \phi_1 y'_{t-1} + \ldots + \phi_p y'_{t-p} + \theta_1 \varepsilon_{t-1} + \ldots + \theta q \varepsilon_{t-q} + \varepsilon_t$$

wobei y'_t die differenzierte Reihe ist. Dieses Modell wird als ARIMA-Modell (p, d, q) bezeichnet, wobei p die Ordnung der autoregressiven Komponente, d die Ordnung der beteiligten Differenzierungen und q die Ordnung des gleitenden Durchschnitts sind. Die Differenzierung ist eine Transformationsmethode, mit der die zeitliche Abhängigkeit in der Zeitreihe beseitigt wird. Dazu gehört die Eliminierung von Komponenten wie Trends und Saisonalität, um Zeitreihen stationär zu machen. Die Häufigkeit, mit der die Differenzierung durchgeführt wird, wird als Differenzierungsreihenfolge bezeichnet. Die Differenzierung erster Ordnung ist also die Differenz zwischen den aktuellen und den vorherigen Zeitdatenpunkten ($y_i - y_{i-1}$), während die Differenzierung zweiter Ordnung wie folgt beschrieben wird: $(y_i - y_{i-1})-(y_{i-1} - y_{i-2})$. Obwohl die Differenzierung ein wichtiger Schritt bei der Vorbereitung von Daten für die Verwendung in einem ARIMA-Modell ist, muss sie sorgfältig durchgeführt werden, um eine übermäßige Differenzierung der Reihen zu vermeiden. Die richtige Differenzierungsreihenfolge ist die Mindestdifferenzierung, die erforderlich ist, um eine nahezu stationäre Reihe zu erhalten. Wenn die Serie beispielsweise für viele Verzögerungen (10 oder mehr) immer noch positive Autokorrelationen aufweist, muss die Serie weiter differenziert werden. Wenn andererseits die Autokorrelation bei Verzögerung 1 zu klein ist, müssen wir möglicherweise die Differenzierungsordnung verringern.

Das SARIMA-Modell (Seasonal ARIMA) ist eine Erweiterung des ARIMA-Modells. Dies wird durch die Addition einer linearen Kombination aus saisonalen Vergangenheitswerten und Prognosefehlern erreicht.

ARIMA Übung

Für unsere Modelle werden wir hier tägliche Zeitreihen zur Stromproduktion verwenden. Zuerst müssen wir die Serie importieren und einen kurzen Blick darauf werfen:

In [1]:
```
#  lags: the number of lags to show the autocorrelation of data points at;
# the autocorrelation of values which are N/lags periods apart is.
# plot_acf() : statsmodels function to inspect which lags have a strong correlation to the forecasted value,
# this plot shows how much each previous lag influences the future lag.
# dataset : electric production

import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.graphics.tsaplots import plot_acf
import numpy as np
df = pd.read_csv('Electric_Production.csv')
df['DATE']= pd.to_datetime(df['DATE'])
df
```

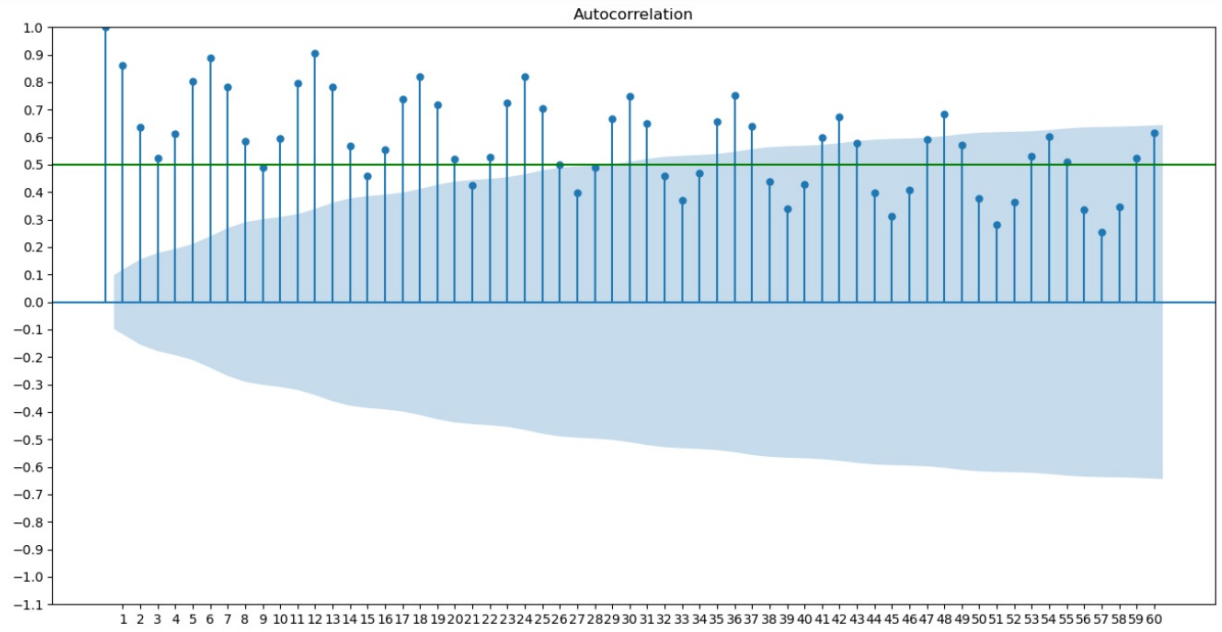
Out[1]:

	DATE	Value
0	1985-01-01	72.5052
1	1985-02-01	70.6720
2	1985-03-01	62.4502
3	1985-04-01	57.4714
4	1985-05-01	55.3151
...	...	...
392	2017-09-01	98.6154
393	2017-10-01	93.6137
394	2017-11-01	97.3359

Jetzt überprüfen wir die Autokorrelation in der Zeitreihe mit der Funktion plot_acf, die die Autokorrelationen bei n Verzögerungen darstellt. Hier setzen wir „Lags" auf 60. Dieser Schritt ist wichtig, um die Anzahl der Lags zu bestimmen, die in das Modell einbezogen werden sollen.

Die Werte über dem Signifikanzniveau (0,5) weisen auf positive Autokorrelationen in der Zeitreihe mit der entsprechenden Verzögerung hin. Wir werden offensichtlich signifikante Autokorrelationen bis zur Verzögerung 60 bemerken:

In [4]:
```
fig, ax = plt.subplots(figsize=(16,8))
# nlags= 10 *log10(N) ≈ 60, N: dataset length; show autocorrelations at nlags
plot_acf(df['Value'], lags=60, ax=ax)
plt.ylim([0,1])
plt.yticks(np.arange(-1.1, 1.1, 0.1))
plt.xticks(np.arange(1, 61, 1))
plt.axhline(y=0.5, color="green")
plt.show()

# high positive autocorelation (above the threshold :0.5 ) of data points till  lag 60
```



Nachdem wir die Daten in Trainings- und Testsätze aufgeteilt haben, werden wir ein autoregressives Modell trainieren und bewerten. Die hier verwendete Bewertungsmetrik ist der mittlere durchschnittliche prozentuale Fehler (MAPE):

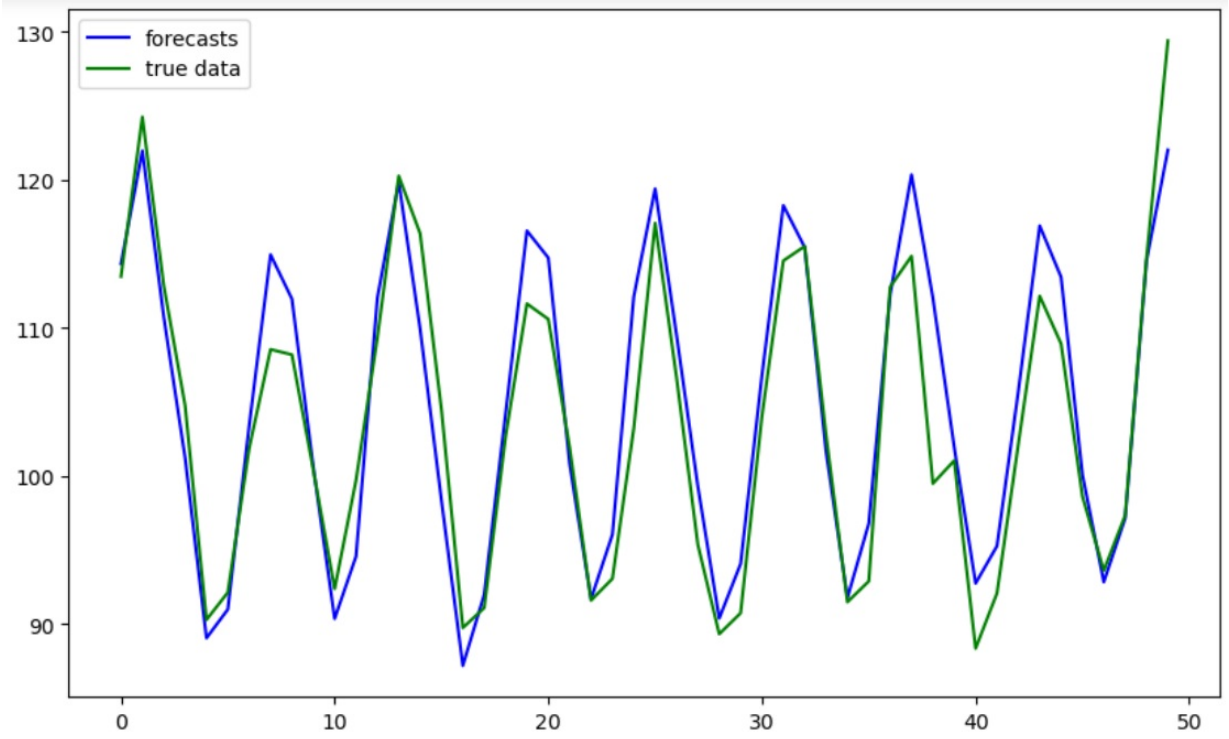
In [5]:
```
# splitting dataset into train and test
train_data = df['Value'].iloc[:-50]
test_data = df['Value'].iloc[-50:]
len(train_data)
from statsmodels.tsa.ar_model import AutoReg
# The number of lags to include in the model
model = AutoReg(train_data, lags=60).fit()
```

In [6]:
```
# evaluating autoregressive model
forecasts = model.forecast(50).tolist()
test_values = test_data.tolist()

# evaluate the model using MAPE metric
from sklearn.metrics import mean_absolute_percentage_error
print(mean_absolute_percentage_error(test_values, forecasts))
# obviously, the error is about 3%
```

```
0.028449641966619997
```

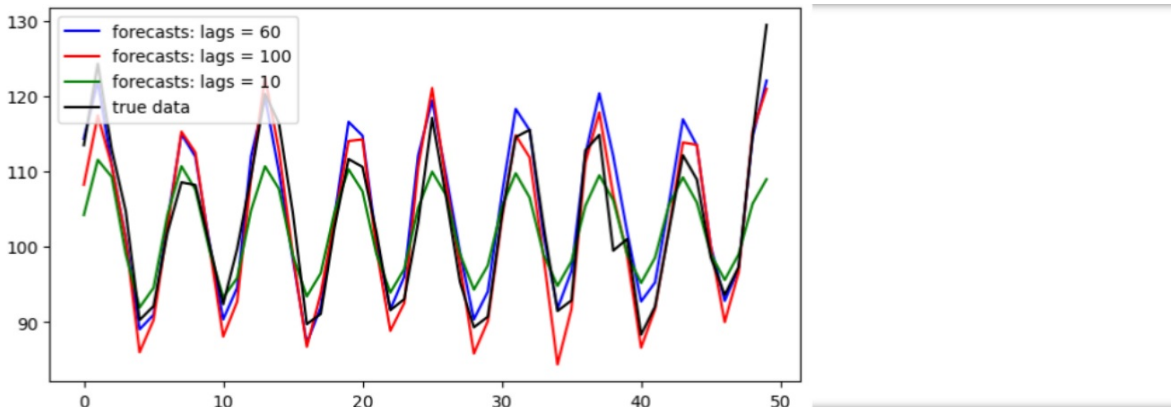
Um eine visuelle Vorstellung davon zu bekommen, wie genau die Prognosen im Vergleich zu echten Daten sind, können wir sowohl die Prognosen als auch die Testdaten grafisch darstellen, um zu sehen, wie nahe die Kurven erscheinen:

In [7]:
```
#To get a visual idea of how accurate the forecasts against true data are,
# we can plot both the forecasts and the test data to see how close the curves seem
fig = plt.subplots(figsize=(10,6))
plt.plot(forecasts, color="blue", label ='forecasts')
plt.plot(test_values,color="green", label ='true data')
plt.legend(loc="upper left")
plt.show()
# both curves seem to in agreement with each other
```



Jetzt werden wir das AR-Modell mit einer anderen Anzahl von Verzögerungen neu erstellen und die resultierenden Prognosen mit echten Daten vergleichen:

In [63]:
```
# re-build model with different lags values:100, 10
model1 = AutoReg(train_data, lags=100).fit()
model2=  AutoReg(train_data, lags=10).fit()
# evaluating autoregressive model
forecasts1 = model1.forecast(50).tolist()
forecasts2 = model2.forecast(50).tolist()
test_values = test_data.tolist()

# plot the curves against each other

fig = plt.subplots(figsize=(12,8))
plt.plot(forecasts, color="blue", label ='forecasts: lags = 60')
plt.plot(forecasts1, color="red", label ='forecasts: lags = 100')
plt.plot(forecasts2, color="green", label ='forecasts: lags = 10')
plt.plot(test_values,color="black", label ='true data')
plt.legend(loc="upper left")
plt.show()
```



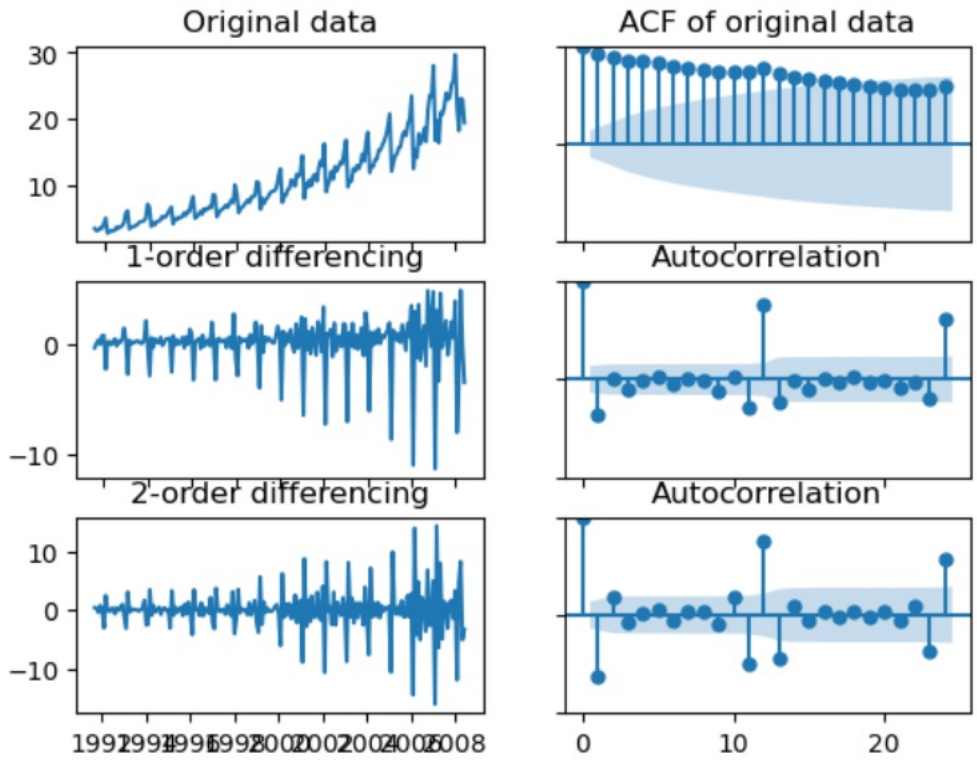
Jetzt erstellen wir ein ARIMA-Modell unter Verwendung des Zeitreihendatensatzes hier .
Zuerst prüfen wir mithilfe des Augmented Dickey Fuller-Tests (adfuller()) aus dem Statsmodels-Paket, ob die Daten stationär sind. Die Nullhypothese des ADF-Tests besagt, dass die Zeitreihe instationär ist. Wenn also der p-Wert des Tests unter dem Signifikanzniveau (0,05) liegt, lehnen Sie die Nullhypothese ab und folgern, dass die Zeitreihe tatsächlich stationär ist und umgekehrt:

In [28]:
```
# import time series data
df2 = pd.read_csv('https://raw.githubusercontent.com/selva86/datasets/master/a10.csv', parse_dates=['date'])

df2.head(5)
#df2['date'] = pd.to_datetime(df2['date'])
df2 = pd.DataFrame(df2).set_index('date')
df2.index = pd.DatetimeIndex(df2.index).to_period('M')
# The null hypothesis of the ADF test is that the time series is non-stationary.
#So, if the p-value of the test is less than the significance level (0.05) then you reject the null hypothesis and
#infer that the time series is indeed stationary.

result = adfuller(df2.value)
print('ADF Statistic: %f' % result[0])
print('p-value: %f' % result[1])
# p-value is higher than significae level 0.05 > there is no reason to reject null_hypothesis and thus the data is non stationary
# with non stationary data, ARIMA can not be applied;use differencing
```

```
ADF Statistic: 3.145186
p-value: 1.000000
```

Für ARIMA müssen drei Hauptargumente ermittelt werden: die Reihenfolge des AR-Terms (p), die Reihenfolge des MA-Terms (q) und die Reihenfolge der Differenzierung (d). Da die Zeitreihe nicht stationär ist, müssen wir Differenzierungen anwenden. Die Differenzierungsreihenfolge wird bestimmt, indem die Abnahme der Autokorrelation in den Daten nach jedem Differenzierungsschritt überprüft wird. Lassen Sie uns eine Differenzierung 1. und 2. Ordnung durchführen und die Ergebnisse überprüfen:

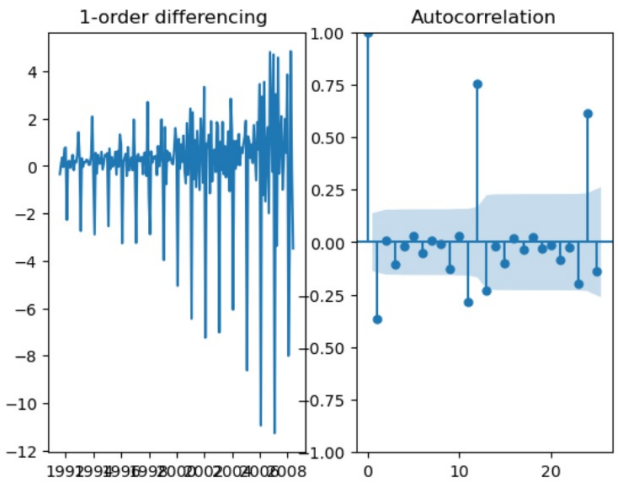
In [33]:
```
# tuning p, d, q
# p is the order of the AR term
# q is the order of the MA term
# d is the order of differencing required to make the time series stationary
# time Series original acf plot

df1 = df2.to_timestamp()
figure, axis = plt.subplots(3, 2)
axis[0, 0].plot(df1)
axis[0, 0].set_title('Original data')
# default number of lags are considered
plot_acf(df2, ax=axis[0, 1])
axis[0, 1].set_title('ACF of original data')
axis[1, 0].plot(df1.diff())
axis[1, 0].set_title('1-order differencing ')
plot_acf(df2.diff().dropna(), ax=axis[1, 1])

axis[2, 0].plot(df1.diff().diff())
axis[2, 0].set_title('2-order differencing')
plot_acf(df2.diff().diff().dropna(), ax=axis[2, 1])
# Hide x labels and tick labels for top plots and y ticks for right plots.
for ax in axis.flat:
    ax.label_outer()
# we will stop at first differencing since autocorrelation doesn't decrease afterthat.
```



ACF-Diagramme zeigen an, dass die Differenzierung 2. Ordnung keinen signifikanten Rückgang der Autokorrelation mit sich bringt, daher wird d auf 1 gesetzt. ACF-Diagramme geben auch den q-Wert an, den ARIMA annehmen sollte. Wenn wir uns das ACF-Diagramm der Differenzierung 1. Ordnung ansehen, können wir offensichtlich eine signifikante Autokorrelation bei Verzögerung 24 erkennen, daher ist q auf 24 gesetzt:

In [314]:
```
fig, (ax1, ax2) = plt.subplots(1, 2)
ax1.plot(df1.diff().dropna())
ax1.set_title('1-order differencing ')
plot_acf(df2.diff().dropna(),lags=25, ax=ax2)
```

Out[314]:

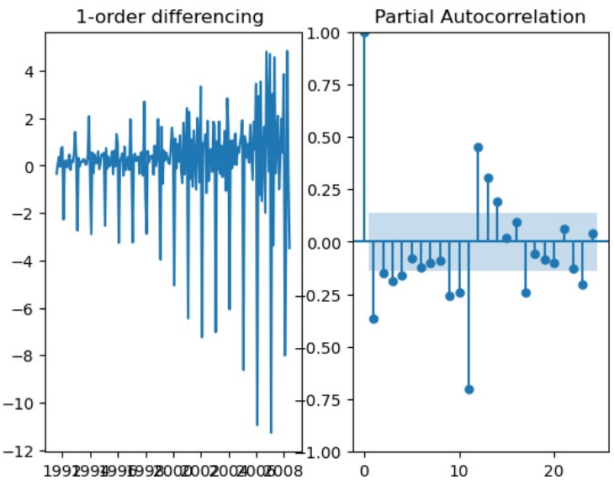


etzt müssen wir uns für den p-Wert entscheiden, der den AR-Term darstellt. Wir können die erforderliche Anzahl von AR-Termen herausfinden, indem wir das Diagramm der partiellen Autokorrelation (PACF) untersuchen. Partielle Autokorrelation kann man sich als die Korrelation zwischen der Reihe und ihrer Verzögerung vorstellen, nach Ausschluss der Beiträge aus den Zwischenverzögerungen. Lassen Sie uns die partielle Autokorrelation der differenzierten Daten erster Ordnung untersuchen:

In [315]:
```
# AR term: p, You can find out the required number of AR terms
# by inspecting the Partial Autocorrelation (PACF) plot.
# Partial autocorrelation can be imagined as the correlation between the series and its lag,
# after excluding the contributions from the intermediate lags.

fig, (ax1, ax2) = plt.subplots(1, 2)
ax1.plot(df1.diff().dropna())
ax1.set_title('1-order differencing ')
plot_pacf(df2.diff().dropna(), ax=ax2)
# pacf starts from lag 1, so we should look at the first lag in plot
# the blue region is the significance threshold
# there is cutoff at lag 2, so we choose p=1
```

```
C:\Users\la2022\anaconda3\lib\site-packages\statsmodels\graphics\tsaplots.py:348: FutureWarning: The default method 'yw' can pr
oduce PACF values outside of the [-1,1] interval. After 0.13, the default will change tounadjusted Yule-Walker ('ywm'). You can
use this method now by setting method='ywm'.
  warnings.warn(
```

Out[315]:

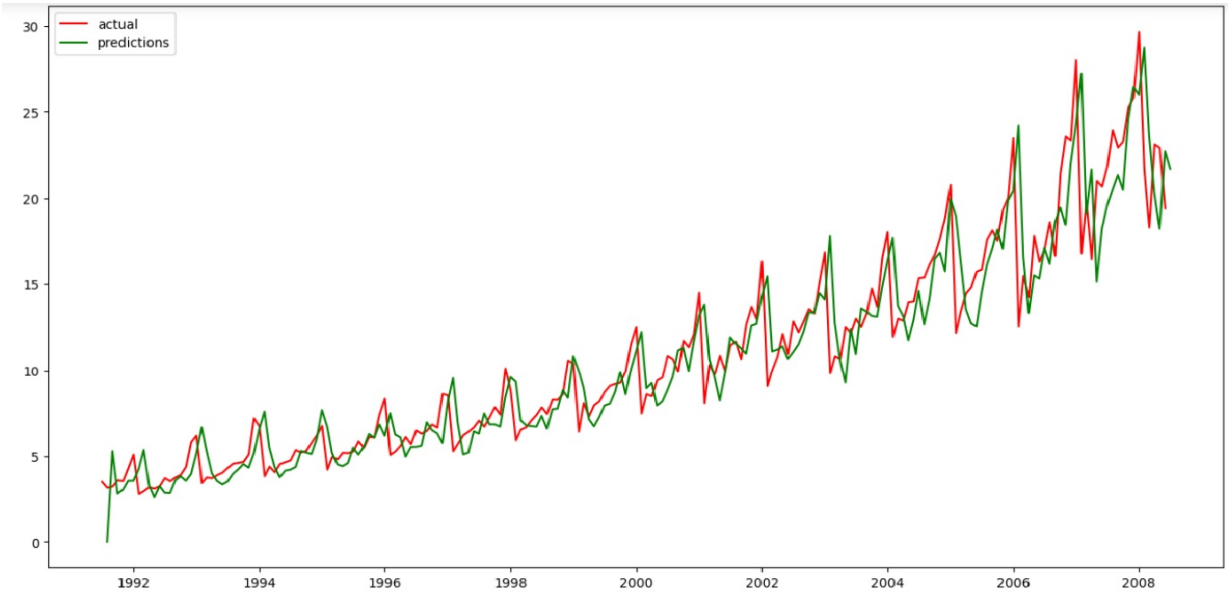


Jetzt erstellen wir ein ARIMA-Modell und verwenden es, um Vorhersagen innerhalb der Stichprobe zu treffen. Lassen Sie uns Vorhersagen anhand wahrer Daten darstellen und den resultierenden Fehlerwert überprüfen:

In [42]:
```
# arima: order(p,d,q),  p is the order (number of time lags) of the autoregressive model
from statsmodels.tsa.arima.model import ARIMA

# 1,1,24 ARIMA Model
model = ARIMA(df2, order=(1,1,24))
model_fit = model.fit()
print(model_fit.summary())
```

```
C:\Users\la2022\anaconda3\lib\site-packages\statsmodels\tsa\statespace\sarimax.py:978: UserWarning: Non-invertible starting MA
parameters found. Using zeros as starting parameters.
  warn('Non-invertible starting MA parameters found.'
```

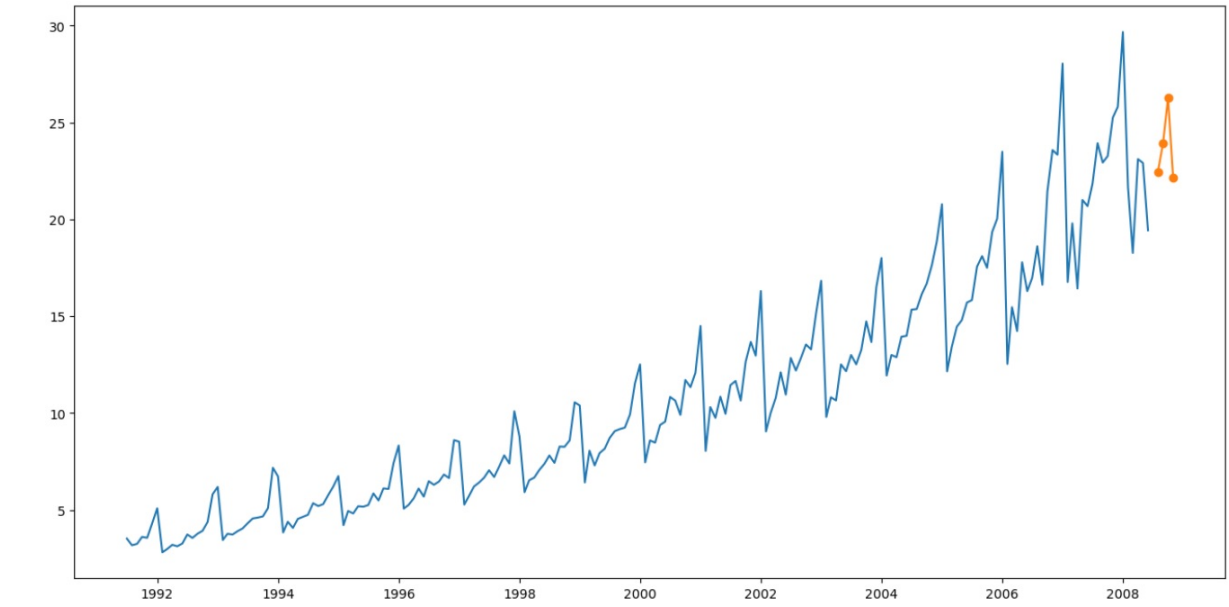
```
                               SARIMAX Results
==============================================================================
Dep. Variable:                  value   No. Observations:                  204
Model:                ARIMA(1, 1, 24)   Log Likelihood                -310.453
Date:                Mon, 19 Jun 2023   AIC                            672.907
Time:                        16:13:16   BIC                            759.050
Sample:                    07-31-1991   HQIC                           707.757
                         - 06-30-2008
Covariance Type:                  opg
```




```
In [44]: mape = mean_absolute_percentage_error(df2, pred)
         mape
```

```
Out[44]: 0.08448952981466776
```

Jetzt erstellen wir ein ARIMA-Modell und verwenden es, um Vorhersagen innerhalb der Stichprobe zu treffen. Lassen Sie uns Vorhersagen anhand wahrer Daten darstellen und den resultierenden Fehlerwert überprüfen:




```
{
 "cells": [
  {
   "cell_type": "code",
   "execution_count": 1,
   "id": "ac764d5d",
   "metadata": {},
   "outputs": [
    {
     "data": {
      "text/html": [
       "<div>\n",
       "<style scoped>\n",
       "    .dataframe tbody tr th:only-of-type {\n",
       "        vertical-align: middle;\n",
       "    }\n",
       "\n",
       "    .dataframe tbody tr th {\n",
       "        vertical-align: top;\n",
       "    }\n",
       "\n",
       "    .dataframe thead th {\n",
       "        text-align: right;\n",
       "    }\n",
       "</style>\n",
       "<table border=\"1\" class=\"dataframe\">\n",
       "  <thead>\n",
       "    <tr style=\"text-align: right;\">\n",
       "      <th></th>\n",
       "      <th>DATE</th>\n",
       "      <th>Value</th>\n",
       "```
```

```
        "    </tr>\n",
        "  </thead>\n",
        "  <tbody>\n",
        "    <tr>\n",
        "      <th>0</th>\n",
        "      <td>1985-01-01</td>\n",
        "      <td>72.5052</td>\n",
        "    </tr>\n",
        "    <tr>\n",
        "      <th>1</th>\n",
        "      <td>1985-02-01</td>\n",
        "      <td>70.6720</td>\n",
        "    </tr>\n",
        "    <tr>\n",
        "      <th>2</th>\n",
        "      <td>1985-03-01</td>\n",
        "      <td>62.4502</td>\n",
        "    </tr>\n",
        "    <tr>\n",
        "      <th>3</th>\n",
        "      <td>1985-04-01</td>\n",
        "      <td>57.4714</td>\n",
        "    </tr>\n",
        "    <tr>\n",
        "      <th>4</th>\n",
        "      <td>1985-05-01</td>\n",
        "      <td>55.3151</td>\n",
        "    </tr>\n",
        "    <tr>\n",
        "      <th>...</th>\n",
        "      <td>...</td>\n",
        "      <td>...</td>\n",
        "    </tr>\n",
        "    <tr>\n",
        "      <th>392</th>\n",
        "      <td>2017-09-01</td>\n",
        "      <td>98.6154</td>\n",
        "    </tr>\n",
        "    <tr>\n",
        "      <th>393</th>\n",
        "      <td>2017-10-01</td>\n",
        "      <td>93.6137</td>\n",
        "    </tr>\n",
        "    <tr>\n",
        "      <th>394</th>\n",
        "      <td>2017-11-01</td>\n",
        "      <td>97.3359</td>\n",
        "    </tr>\n",
        "    <tr>\n",
        "      <th>395</th>\n",
        "      <td>2017-12-01</td>\n",
        "      <td>114.7212</td>\n",
        "    </tr>\n",
        "    <tr>\n",
        "      <th>396</th>\n",
        "      <td>2018-01-01</td>\n",
        "      <td>129.4048</td>\n",
        "    </tr>\n",
        "  </tbody>\n",
        "</table>\n",
        "<p>397 rows × 2 columns</p>\n",
        "</div>"
       ],
       "text/plain": [
        "          DATE     Value\n",
        "0   1985-01-01   72.5052\n",
        "1   1985-02-01   70.6720\n",
        "2   1985-03-01   62.4502\n",
        "3   1985-04-01   57.4714\n",
        "4   1985-05-01   55.3151\n",
        "..         ...       ...\n",
        "392 2017-09-01   98.6154\n",
        "393 2017-10-01   93.6137\n",
        "394 2017-11-01   97.3359\n",
        "395 2017-12-01  114.7212\n",
        "396 2018-01-01  129.4048\n",
        "\n",
        "[397 rows x 2 columns]"
       ]
      },
      "execution_count": 1,
      "metadata": {},
      "output_type": "execute_result"
     }
    ],
    "source": [
     "#  lags: the number of lags to show the autocorrelation of data points at;\n",
     "# the autocorrelation of values which are N/lags periods apart is.\n",
     "# plot_acf() : statsmodels function to inspect which lags have a strong correlation to the forecasted value,\n",
     "# this plot shows how much each previous lag influences the future lag.\n",
     "# dataset : electric production \n",
     "\n",
     "import pandas as pd\n",
     "import matplotlib.pyplot as plt\n",
     "from statsmodels.graphics.tsaplots import plot_acf\n",
     "import numpy as np\n",
     "df = pd.read_csv('Electric_Production.csv')\n",
     "df['DATE']= pd.to_datetime(df['DATE'])\n",
     "df"
    ]
   },
   {
    "cell_type": "code",
    "execution_count": 2,
```

```
   "id": "a30dc660",
   "metadata": {},
   "outputs": [
    {
     "data": {
      "image/png": "iVBORw0KGgoAAAANSUhEUgAABRcAAAKoCAYAAADtfshrAAAAOXRFWHRTb2Z0d2FyZQBNYXRwbG90bGliIHZlcnNpb24zLjcuMSwgaHR0
      "text/plain": [
       "<Figure size 1600x800 with 1 Axes>"
      ]
     },
     "metadata": {},
     "output_type": "display_data"
    }
   ],
   "source": [
    "fig, ax = plt.subplots(figsize=(16,8))\n",
    "# nlags= 10 *log10(N) ? 60, N: dataset length; show autocorrelations at nlags\n",
    "plot_acf(df['Value'], lags=60, ax=ax)\n",
    "plt.ylim([0,1])\n",
    "plt.yticks(np.arange(-1.1, 1.1, 0.1))\n",
    "plt.xticks(np.arange(1, 61, 1))\n",
    "plt.axhline(y=0.5, color=\"green\")\n",
    "plt.show()\n",
    "\n",
    "# high positive autocorelation (above the threshold :0.5 ) of data points till  lag 60"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": 3,
   "id": "04756836",
   "metadata": {},
   "outputs": [],
   "source": [
    "# splitting dataset into train and test\n",
    "train_data = df['Value'].iloc[:-50]\n",
    "test_data = df['Value'].iloc[-50:]\n",
    "len(train_data)\n",
    "from statsmodels.tsa.ar_model import AutoReg\n",
    "# The number of lags to include in the model \n",
    "model = AutoReg(train_data, lags=60).fit()"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": 4,
   "id": "ba2f555e",
   "metadata": {},
   "outputs": [
    {
     "name": "stdout",
     "output_type": "stream",
     "text": [
      "0.028449641966619997\n"
     ]
    }
   ],
   "source": [
    "# evaluating autoregressive model \n",
    "forecasts = model.forecast(50).tolist()\n",
    "test_values = test_data.tolist()\n",
    "\n",
    "# evaluate the model using MAPE metric\n",
    "from sklearn.metrics import mean_absolute_percentage_error\n",
    "print(mean_absolute_percentage_error(test_values, forecasts))\n",
    "# obviously, the error is about 3%"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "af933ddd",
   "metadata": {},
   "outputs": [],
   "source": []
  },
  {
   "cell_type": "code",
   "execution_count": 5,
   "id": "00c67b6e",
   "metadata": {},
   "outputs": [
    {
     "data": {
      "image/png": "iVBORw0KGgoAAAANSUhEUgAAAAz8AAAH5CAYAAACve4DDAAAAOXRFWHRTb2Z0d2FyZQBNYXRwbG90bGliIHZlcnNpb24zLjcuMSwgaHR0
      "text/plain": [
       "<Figure size 1000x600 with 1 Axes>"
      ]
     },
     "metadata": {},
     "output_type": "display_data"
    }
   ],
   "source": [
    "#To get a visual idea of how accurate the forecasts against true data are,\n",
    "# we can plot both the forecasts and the test data to see how close the curves seem\n",
    "fig = plt.subplots(figsize=(10,6))\n",
    "plt.plot(forecasts, color=\"blue\", label ='forecasts')\n",
    "plt.plot(test_values,color=\"green\", label ='true data')\n",
    "plt.legend(loc=\"upper left\")\n",
    "plt.show()\n",
    "# both curves seem to in agreement with each other"
   ]
  },
  {
```

```
{
 "cell_type": "code",
 "execution_count": 6,
 "id": "01fdbc29",
 "metadata": {},
 "outputs": [],
 "source": [
  "# re-build model with different lags values:100, 10\n",
  "model1 = AutoReg(train_data, lags=100).fit()\n",
  "model2=  AutoReg(train_data, lags=10).fit()\n",
  "# evaluating autoregressive model \n",
  "forecasts1 = model1.forecast(50).tolist()\n",
  "forecasts2 = model2.forecast(50).tolist()\n",
  "test_values = test_data.tolist()"
 ]
},
{
 "cell_type": "code",
 "execution_count": 7,
 "id": "fb3bc50b",
 "metadata": {
  "scrolled": true
 },
 "outputs": [
  {
   "data": {
    "image/png": "iVBORw0KGgoAAAANSUhEUgAAAqQAAAFfCAYAAACY+97uAAAAOXRFWHRTb2Z0d2FyZQBNYXRwbG90bGliIHZlcnNpb24zLjcuMSwgaHR0
    "text/plain": [
     "<Figure size 800x400 with 1 Axes>"
    ]
   },
   "metadata": {},
   "output_type": "display_data"
  }
 ],
 "source": [
  "# plot the curves against each other \n",
  "\n",
  "fig = plt.subplots(figsize=(8,4))\n",
  "plt.plot(forecasts, color=\"blue\", label ='forecasts: lags = 60')\n",
  "plt.plot(forecasts1, color=\"red\", label ='forecasts: lags = 100')\n",
  "plt.plot(forecasts2, color=\"green\", label ='forecasts: lags = 10')\n",
  "plt.plot(test_values,color=\"black\", label ='true data')\n",
  "plt.legend(loc=\"upper left\")\n",
  "plt.show()\n",
  "   "
 ]
},
{
 "cell_type": "code",
 "execution_count": 8,
 "id": "fb54019c",
 "metadata": {},
 "outputs": [
  {
   "name": "stdout",
   "output_type": "stream",
   "text": [
    "ADF Statistic: -1.773219\n",
    "p-value: 0.39383653785330297\n"
   ]
  },
  {
   "name": "stderr",
   "output_type": "stream",
   "text": [
    "C:\\Users\\la2022\\AppData\\Local\\Temp\\ipykernel_17244\\4186456414.py:26: FutureWarning: The default value of numer
    "  df2 = df2.groupby(level=0).sum()\n"
   ]
  },
  {
   "data": {
    "text/html": [
     "<div>\n",
     "<style scoped>\n",
     "    .dataframe tbody tr th:only-of-type {\n",
     "        vertical-align: middle;\n",
     "    }\n",
     "\n",
     "    .dataframe tbody tr th {\n",
     "        vertical-align: top;\n",
     "    }\n",
     "\n",
     "    .dataframe thead th {\n",
     "        text-align: right;\n",
     "    }\n",
     "</style>\n",
     "<table border=\"1\" class=\"dataframe\">\n",
     "  <thead>\n",
     "    <tr style=\"text-align: right;\">\n",
     "      <th></th>\n",
     "      <th>sales</th>\n",
     "    </tr>\n",
     "    <tr>\n",
     "      <th>date</th>\n",
     "      <th></th>\n",
     "    </tr>\n",
     "  </thead>\n",
     "  <tbody>\n",
     "    <tr>\n",
     "      <th>2013-01-01</th>\n",
     "      <td>2.0</td>\n",
     "    </tr>\n",
     "    <tr>\n",
```

```
      "      <th>2013-01-02</th>\n",
      "      <td>207.0</td>\n",
      "    </tr>\n",
      "    <tr>\n",
      "      <th>2013-01-03</th>\n",
      "      <td>125.0</td>\n",
      "    </tr>\n",
      "    <tr>\n",
      "      <th>2013-01-04</th>\n",
      "      <td>133.0</td>\n",
      "    </tr>\n",
      "    <tr>\n",
      "      <th>2013-01-05</th>\n",
      "      <td>191.0</td>\n",
      "    </tr>\n",
      "    <tr>\n",
      "      <th>...</th>\n",
      "      <td>...</td>\n",
      "    </tr>\n",
      "    <tr>\n",
      "      <th>2017-08-11</th>\n",
      "      <td>351.0</td>\n",
      "    </tr>\n",
      "    <tr>\n",
      "      <th>2017-08-12</th>\n",
      "      <td>369.0</td>\n",
      "    </tr>\n",
      "    <tr>\n",
      "      <th>2017-08-13</th>\n",
      "      <td>433.0</td>\n",
      "    </tr>\n",
      "    <tr>\n",
      "      <th>2017-08-14</th>\n",
      "      <td>337.0</td>\n",
      "    </tr>\n",
      "    <tr>\n",
      "      <th>2017-08-15</th>\n",
      "      <td>339.0</td>\n",
      "    </tr>\n",
      "  </tbody>\n",
      "</table>\n",
      "<p>1684 rows × 1 columns</p>\n",
      "</div>"
     ],
     "text/plain": [
      "            sales\n",
      "date             \n",
      "2013-01-01    2.0\n",
      "2013-01-02  207.0\n",
      "2013-01-03  125.0\n",
      "2013-01-04  133.0\n",
      "2013-01-05  191.0\n",
      "...           ...\n",
      "2017-08-11  351.0\n",
      "2017-08-12  369.0\n",
      "2017-08-13  433.0\n",
      "2017-08-14  337.0\n",
      "2017-08-15  339.0\n",
      "\n",
      "[1684 rows x 1 columns]"
     ]
    },
    "execution_count": 8,
    "metadata": {},
    "output_type": "execute_result"
   }
  ],
  "source": [
   "# AutoRegressive Integrated Moving Average(ARIMA)  is a time series forecasting model that incorporates autocorrelation\n",
   "# to model temporal structures within the time series data to predict future values.\n",
   "# The autoregression part of the model measures the dependency of a particular sample with a few past observations. \n"
   "# These differences are measured and integrated to make the data patterns stationary or\n",
   "# minimize the obvious correlation with past data (collinearity or linear dependence)\n",
   "# (since linear independence and no collinearity is one of the fundamental assumptions of the linear regression model).
   "# After this, a moving average helps condense and bring out significant features from the data.\n",
   "# stationary time series is Consequently, parameters such as mean and variance also do not change over time.\n",
   "# Thus, time series with trends, or with seasonality, are not stationary,\n",
   "# because trenda and seasonality affect the time series at different time points.\n",
   "# ARIMA equation :y?t = c + ?1y?t?1 + ? + ?py?t?p + ?1?t?1 + ? + ?q?t?q + ?t\n",
   "\n",
   "# we will be using Store_sale time series dataset: \n",
   "# https://www.kaggle.com/competitions/store-sales-time-series-forecasting/data\n",
   "# first we check if the data is stationary using the Augmented Dickey Fuller test (adfuller()),from the statsmodels pac
   "from statsmodels.tsa.stattools import adfuller\n",
   "from numpy import log\n",
   "import pandas as pd\n",
   "df2 = pd.read_csv('Store_sales_train.csv',parse_dates=['date'])\n",
   "\n",
   "#df2['date'] = pd.to_datetime(df2['date'])\n",
   "df2 = pd.DataFrame(df2, columns=['date','sales', 'family']).set_index('date')\n",
   "df2.index = pd.DatetimeIndex(df2.index).to_period('D')\n",
   "df2 = df2[df2.sales != 0]\n",
   "df2 = df2[df2.family == 'BEAUTY']\n",
   "df2 = df2.groupby(level=0).sum()\n",
   "## remove Family column\n",
   "df = pd.DataFrame(df2, columns=['sales'])\n",
   "\n",
   "result = adfuller(df)\n",
   "print('ADF Statistic: %f' % result[0])\n",
   "print('p-value: %s' % result[1])\n",
   "df\n"
  ]
 },
 {
```

```
   "cell_type": "code",
   "execution_count": 9,
   "id": "199c290b",
   "metadata": {},
   "outputs": [
    {
     "name": "stdout",
     "output_type": "stream",
     "text": [
      "ADF Statistic: 3.145186\n",
      "p-value: 1.000000\n"
     ]
    }
   ],
   "source": [
    "# import time series data\n",
    "df2 = pd.read_csv('https://raw.githubusercontent.com/selva86/datasets/master/a10.csv', parse_dates=['date'])\n",
    "\n",
    "df2.head(5)\n",
    "#df2['date'] = pd.to_datetime(df2['date'])\n",
    "df2 = pd.DataFrame(df2).set_index('date')\n",
    "df2.index = pd.DatetimeIndex(df2.index).to_period('M')\n",
    "# The null hypothesis of the ADF test is that the time series is non-stationary.\n",
    "#So, if the p-value of the test is less than the significance level (0.05) then you reject the null hypothesis and\n",
    "#infer that the time series is indeed stationary.\n",
    "\n",
    "result = adfuller(df2.value)\n",
    "print('ADF Statistic: %f' % result[0])\n",
    "print('p-value: %f' % result[1])\n",
    "# p-value is higher than significae level 0.05 > there is no reason to reject null_hypothesis and thus the data is non
    "# with non stationary data, ARIMA can not be applied;use differencing\n"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": 17,
   "id": "748cf318",
   "metadata": {},
   "outputs": [
    {
     "data": {
      "image/png": "iVBORw0KGgoAAAANSUhEUgAABRcAAAKoCAYAAAADtfshrAAAAOXRFWHRTb2Z0d2FyZQBNYXRwbG90bGliIHZlcnNpb24zLjcuMSwgaHR0
      "text/plain": [
       "<Figure size 1600x800 with 1 Axes>"
      ]
     },
     "metadata": {},
     "output_type": "display_data"
    }
   ],
   "source": [
    "import matplotlib.pyplot as plt\n",
    "from statsmodels.graphics.tsaplots import plot_acf, plot_pacf\n",
    "import numpy as np\n",
    "fig, ax = plt.subplots(figsize=(16,8))\n",
    "# nlags= 10 *log10(N) ? 32, N: dataset length; show autocorrelations at nlags\n",
    "plot_acf(df2,lags=32,  ax=ax)\n",
    "plt.ylim([0,1])\n",
    "plt.yticks(np.arange(-1.1, 1.1, 0.1))\n",
    "plt.xticks(np.arange(1, 33, 1))\n",
    "plt.axhline(y=0.5, color=\"green\")\n",
    "plt.show()\n",
    "# we always ignore the long spike at lag 0, we look from lag 1 on .."
   ]
  },
  {
   "cell_type": "code",
   "execution_count": 14,
   "id": "181debe8",
   "metadata": {},
   "outputs": [
    {
     "data": {
      "image/png": "iVBORw0KGgoAAAANSUhEUgAAAisAAAGxCAYAAACju/aQAAAAOXRFWHRTb2Z0d2FyZQBNYXRwbG90bGliIHZlcnNpb24zLjcuMSwgaHR0
      "text/plain": [
       "<Figure size 640x480 with 6 Axes>"
      ]
     },
     "metadata": {},
     "output_type": "display_data"
    }
   ],
   "source": [
    "# tuning p, d, q \n",
    "# p is the order of the AR term\n",
    "# q is the order of the MA term\n",
    "# d is the order of differencing required to make the time series stationary\n",
    "# time Series original acf plot\n",
    "\n",
    "df1 = df2.to_timestamp()\n",
    "figure, axis = plt.subplots(3, 2)\n",
    "axis[0, 0].plot(df1)\n",
    "axis[0, 0].set_title('Original data')\n",
    "# default number of lags are considered\n",
    "plot_acf(df2, ax=axis[0, 1])\n",
    "axis[0, 1].set_title('ACF of original data')\n",
    "axis[1, 0].plot(df1.diff())\n",
    "axis[1, 0].set_title('1-order differencing ')\n",
    "plot_acf(df2.diff().dropna(), ax=axis[1, 1])\n",
    "\n",
    "axis[2, 0].plot(df1.diff().diff())\n",
    "axis[2, 0].set_title('2-order differencing')\n",
    "plot_acf(df2.diff().diff().dropna(), ax=axis[2, 1])\n",
    "# Hide x labels and tick labels for top plots and y ticks for right plots.\n",
```

```
    "for ax in axis.flat:\n",
    "    ax.label_outer()\n",
    "# we will stop at first differencing since autocorrelation doesn't decrease afterthat.     "
   ]
  },
  {
   "cell_type": "code",
   "execution_count": 20,
   "id": "506f6ea0",
   "metadata": {},
   "outputs": [
    {
     "data": {
      "image/png": "iVBORw0KGgoAAAANSUhEUgAAAiIAAAGxCAYAAABfrt1aAAAAOXRFWHRTb2Z0d2FyZQBNYXRwbG90bGliIHZlcnNpb24zLjcuMSwgaHR0
      "text/plain": [
       "<Figure size 640x480 with 4 Axes>"
      ]
     },
     "metadata": {},
     "output_type": "display_data"
    }
   ],
   "source": [
    "figure, (ax1, ax2, ax3, ax4) = plt.subplots(4)\n",
    "plot_acf(df2, ax=ax1)\n",
    "#add semicolon to suppress plotting returned output twice \n",
    "plot_acf(df2.diff().dropna(), ax=ax2);\n",
    "plot_acf(df2.diff().diff().dropna(), ax=ax3);\n",
    "plot_acf(df2.diff().diff().diff().dropna(), ax=ax4);"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": 21,
   "id": "c5bb284e",
   "metadata": {},
   "outputs": [
    {
     "data": {
      "image/png": "iVBORw0KGgoAAAANSUhEUgAAAiIAAAGxCAYAAABfrt1aAAAAOXRFWHRTb2Z0d2FyZQBNYXRwbG90bGliIHZlcnNpb24zLjcuMSwgaHR0
      "text/plain": [
       "<Figure size 640x480 with 2 Axes>"
      ]
     },
     "metadata": {},
     "output_type": "display_data"
    }
   ],
   "source": [
    "fig, (ax1, ax2) = plt.subplots(1, 2)\n",
    "ax1.plot(df1.diff().dropna())\n",
    "ax1.set_title('1-order differencing ')\n",
    "plot_acf(df2.diff().dropna(),lags=25, ax=ax2);"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": 23,
   "id": "df86a764",
   "metadata": {},
   "outputs": [
    {
     "name": "stderr",
     "output_type": "stream",
     "text": [
      "C:\\Users\\la2022\\anaconda3\\lib\\site-packages\\statsmodels\\graphics\\tsaplots.py:348: FutureWarning: The default m
      "  warnings.warn(\n"
     ]
    },
    {
     "data": {
      "image/png": "iVBORw0KGgoAAAANSUhEUgAAiIAAAGxCAYAAABfrt1aAAAAOXRFWHRTb2Z0d2FyZQBNYXRwbG90bGliIHZlcnNpb24zLjcuMSwgaHR0
      "text/plain": [
       "<Figure size 640x480 with 2 Axes>"
      ]
     },
     "metadata": {},
     "output_type": "display_data"
    }
   ],
   "source": [
    "# AR term: p, You can find out the required number of AR terms\n",
    "# by inspecting the Partial Autocorrelation (PACF) plot.\n",
    "# Partial autocorrelation can be imagined as the correlation between the series and its lag,\n",
    "# after excluding the contributions from the intermediate lags.\n",
    "\n",
    "fig, (ax1, ax2) = plt.subplots(1, 2)\n",
    "ax1.plot(df1.diff().dropna());\n",
    "ax1.set_title('1-order differencing ')\n",
    "plot_pacf(df2.diff().dropna(), ax=ax2);\n",
    "# pacf starts from lag 1, so we should look at the first lag in plot\n",
    "# the blue region is the significance threshold\n",
    "# there is cutoff at lag 2, so we choose p=1"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": 24,
   "id": "2d3eae28",
   "metadata": {
    "scrolled": true
   },
   "outputs": [
    {
```

```
    "data": {
     "image/png": "iVBORw0KGgoAAAANSUhEUgAABSAAAAKoCAYAAABnUVEqAAAAOXRFWHRTb2Z0d2FyZQBNYXRwbG90bGliIHZlcnNpb24zLjcuMSwgaHR0
     "text/plain": [
      "<Figure size 1600x800 with 1 Axes>"
     ]
    },
    "metadata": {},
    "output_type": "display_data"
   }
  ],
  "source": [
   "# MA term q :  from the ACF plot we decide the value of q.\n",
   "# An MA term is technically, the error of the lagged forecast.\n",
   "# The ACF tells how many MA terms (lags) are required to remove any autocorrelation in the stationarized series.\n",
   "\n",
   "fig, ax = plt.subplots(figsize=(16,8))\n",
   "plot_acf(df2.diff().dropna(),lags=30, ax=ax);\n",
   "# this plot indicates that after lag=24 the autocorrelation gets obviously insignificant"
  ]
 },
 {
  "cell_type": "code",
  "execution_count": 25,
  "id": "18e2d9df",
  "metadata": {},
  "outputs": [
   {
    "name": "stderr",
    "output_type": "stream",
    "text": [
     "C:\\Users\\la2022\\anaconda3\\lib\\site-packages\\statsmodels\\tsa\\statespace\\sarimax.py:978: UserWarning: Non-inve
     "  warn('Non-invertible starting MA parameters found.'\n"
    ]
   },
   {
    "name": "stdout",
    "output_type": "stream",
    "text": [
     "                               SARIMAX Results                                \n",
     "==============================================================================\n",
     "Dep. Variable:                  value   No. Observations:                  204\n",
     "Model:                 ARIMA(1, 1, 24)   Log Likelihood                -310.453\n",
     "Date:                Wed, 18 Oct 2023   AIC                            672.907\n",
     "Time:                        17:15:51   BIC                            759.050\n",
     "Sample:                  07-31-1991   HQIC                           707.757\n",
     "                       - 06-30-2008                                         \n",
     "Covariance Type:                opg                                         \n",
     "==============================================================================\n",
     "                 coef    std err          z      P>|z|      [0.025      0.975]\n",
     "------------------------------------------------------------------------------\n",
     "ar.L1         -0.5384      0.138     -3.893      0.000      -0.809      -0.267\n",
     "ma.L1         -0.1004      0.672     -0.149      0.881      -1.417       1.216\n",
     "ma.L2         -0.3508      0.414     -0.847      0.397      -1.162       0.461\n",
     "ma.L3         -0.0110      0.709     -0.016      0.988      -1.402       1.380\n",
     "ma.L4         -0.1149      0.706     -0.163      0.871      -1.499       1.269\n",
     "ma.L5         -0.0072      0.591     -0.012      0.990      -1.166       1.152\n",
     "ma.L6         -0.3124      0.283     -1.103      0.270      -0.867       0.242\n",
     "ma.L7          0.0010      0.736      0.001      0.999      -1.441       1.443\n",
     "ma.L8          0.2245      0.712      0.315      0.752      -1.171       1.620\n",
     "ma.L9         -0.2578      0.782     -0.330      0.742      -1.790       1.275\n",
     "ma.L10        -0.2377      0.544     -0.437      0.662      -1.304       0.829\n",
     "ma.L11         0.0050      0.788      0.006      0.995      -1.540       1.550\n",
     "ma.L12         1.2720      0.700      1.817      0.069      -0.100       2.644\n",
     "ma.L13        -0.0417      0.690     -0.060      0.952      -1.395       1.311\n",
     "ma.L14        -0.4580      0.621     -0.737      0.461      -1.676       0.760\n",
     "ma.L15         0.0620      0.697      0.089      0.929      -1.304       1.428\n",
     "ma.L16         0.0705      0.659      0.107      0.915      -1.221       1.362\n",
     "ma.L17        -0.0183      0.571     -0.032      0.974      -1.137       1.100\n",
     "ma.L18        -0.2395      0.416     -0.575      0.565      -1.056       0.577\n",
     "ma.L19        -0.1094      0.608     -0.180      0.857      -1.302       1.083\n",
     "ma.L20         0.0460      0.654      0.070      0.944      -1.235       1.327\n",
     "ma.L21        -0.2108      0.656     -0.321      0.748      -1.496       1.075\n",
     "ma.L22        -0.3021      0.518     -0.583      0.560      -1.317       0.713\n",
     "ma.L23         0.1558      0.527      0.295      0.768      -0.878       1.189\n",
     "ma.L24         0.7804      0.666      1.172      0.241      -0.525       2.086\n",
     "sigma2         0.9935      0.788      1.261      0.207      -0.550       2.537\n",
     "==============================================================================\n",
     "Ljung-Box (L1) (Q):                   0.49   Jarque-Bera (JB):                30.32\n",
     "Prob(Q):                              0.48   Prob(JB):                         0.00\n",
     "Heteroskedasticity (H):               9.09   Skew:                            -0.28\n",
     "Prob(H) (two-sided):                  0.00   Kurtosis:                         4.81\n",
     "==============================================================================\n",
     "\n",
     "Warnings:\n",
     "[1] Covariance matrix calculated using the outer product of gradients (complex-step).\n"
    ]
   },
   {
    "name": "stderr",
    "output_type": "stream",
    "text": [
     "C:\\Users\\la2022\\anaconda3\\lib\\site-packages\\statsmodels\\base\\model.py:604: ConvergenceWarning: Maximum Likeli
     "  warnings.warn(\"Maximum Likelihood optimization failed to \"\n"
    ]
   }
  ],
  "source": [
   "# arima: order(p,d,q),  p is the order (number of time lags) of the autoregressive model\n",
   "from statsmodels.tsa.arima.model import ARIMA\n",
   "\n",
   "# 1,1,24 ARIMA Model\n",
   "model = ARIMA(df2, order=(1,1,24))\n",
   "model_fit = model.fit()\n",
   "print(model_fit.summary())"
```

```
   ]
  },
  {
   "cell_type": "code",
   "execution_count": 26,
   "id": "a6b2adf6",
   "metadata": {},
   "outputs": [
    {
     "data": {
      "image/png": "iVBORw0KGgoAAAANSUhEUgAABQcAAAKTCAYAAACkbb7tAAAAOXRFWHRTb2Z0d2FyZQBNYXRwbG90bGliIHZlcnNpb24zLjcuMSwgaHR0
      "text/plain": [
       "<Figure size 1600x800 with 1 Axes>"
      ]
     },
     "metadata": {},
     "output_type": "display_data"
    }
   ],
   "source": [
    "# Actual vs Fitted\n",
    "from sklearn.metrics import mean_absolute_percentage_error\n",
    "fig, ax = plt.subplots(figsize=(16,8))\n",
    "#pred= model_fit.forecast(steps= 100, dynamic=False)\n",
    "# in sample predictions: predict() After a model has been fit predict returns the fitted values.\n",
    "pred= model_fit.predict( dynamic=False)\n",
    "\n",
    "#df1 = df.to_timestamp()\n",
    "plt.plot(df1, label ='actual',color='r')\n",
    "plt.plot(pred,color='g',label= 'predictions')\n",
    "plt.legend(loc=\"upper left\")\n",
    "plt.show()"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "48b76cbb",
   "metadata": {},
   "outputs": [],
   "source": [
    "mape = mean_absolute_percentage_error(df2, pred)\n",
    "mape"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": 27,
   "id": "61bc7451",
   "metadata": {},
   "outputs": [
    {
     "name": "stderr",
     "output_type": "stream",
     "text": [
      "C:\\Users\\la2022\\AppData\\Local\\Temp\\ipykernel_17244\\3493184615.py:3: FutureWarning: The pandas.datetime class i
      "  from pandas import datetime\n"
     ]
    },
    {
     "data": {
      "image/png": "iVBORw0KGgoAAAANSUhEUgAABQcAAAKTCAYAAACkbb7tAAAAOXRFWHRTb2Z0d2FyZQBNYXRwbG90bGliIHZlcnNpb24zLjcuMSwgaHR0
      "text/plain": [
       "<Figure size 1600x800 with 1 Axes>"
      ]
     },
     "metadata": {},
     "output_type": "display_data"
    }
   ],
   "source": [
    "#out of samples forecasts\n",
    "\n",
    "from pandas import datetime\n",
    "#start_index = datetime(2008, 7)\n",
    "#end_index = datetime(2008, 12)\n",
    "forecasts = model_fit.forecast(steps=4)\n",
    "fig, ax = plt.subplots(figsize=(16,8))\n",
    "# convert from period into timestamp\n",
    "\n",
    "plt.plot(df1, label ='true data')\n",
    "plt.plot(forecasts, label='forecasts',  marker='o')\n",
    "plt.show()\n"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "153762cb",
   "metadata": {},
   "outputs": [],
   "source": []
  }
 ],
 "metadata": {
  "kernelspec": {
   "display_name": "Python 3 (ipykernel)",
   "language": "python",
   "name": "python3"
  },
  "language_info": {
   "codemirror_mode": {
    "name": "ipython",
    "version": 3
```

```
   },
   "file_extension": ".py",
   "mimetype": "text/x-python",
   "name": "python",
   "nbconvert_exporter": "python",
   "pygments_lexer": "ipython3",
   "version": "3.9.13"
  }
 },
 "nbformat": 4,
 "nbformat_minor": 5
}
```


Trigonometric seasonality, Box-Cox transformation, ARMA errors, Trend and Seasonal components (TBATS)

Während die meisten gängigen Modelle wie ARIMA und exponentielle Glättung nur einzelnes Saisonalitätsmuster lernen können, wurde die TBATS-Prognosemethode entwickelt, um Zeitreihen zu modellieren, die komplexe und mehrere saisonale Muster umfassen, indem exponentielle Glättung verwendet wird, beispielsweise tägliche Daten, die ein wöchentliches und ein jährliches Muster aufweisen können.

TBATS ist die Abkürzung für:

- Trigonometrische Saisonalität
- Box-Cox-Transformation
- ARMA-Fehler
- Trend
- Saisonale Komponenten

In TBATS wird ein Box-Cox-Transformator verwendet, um Zeitreihen so zu transformieren, dass die Daten eine Normalverteilung aufweisen. Dies ist nützlich, da viele statistische Methoden eine Normalverteilung der Modellfehler annehmen. Anschließend lernt TBATS eine lineare Kombination aus einem exponentiell geglätteten Trend, einer saisonalen Komponente und ARMA-Fehlern. Die saisonale Komponente wird durch trigonometrische Funktionen berechnet. Dadurch ist TBATS in der Lage, genauere Langzeitprognosen zu erstellen und andere Prognosemodelle zu übertreffen. Ein Nachteil dieses Modells ist seine langsame Vorhersage- und Lernleistung bei der Modellierung langer Zeitreihen.

TBATS Übung

Hier werden wir Zeitreihen für Ladenverkäufe hier mithilfe der TBATS-Methode modellieren. Dieser Datensatz enthält tägliche Daten zu Verkäufen verschiedener Artikel. Importieren wir zunächst die Daten:

In [1]:
```
import pandas as pd
# https://www.kaggle.com/competitions/store-sales-time-series-forecasting/data
data = pd.read_csv('Store_sales_train.csv')
data
```

Out[1]:

	id	date	store_nbr	family	sales	onpromotion
0	0	2013-01-01	1	AUTOMOTIVE	0.000	0
1	1	2013-01-01	1	BABY CARE	0.000	0
2	2	2013-01-01	1	BEAUTY	0.000	0
3	3	2013-01-01	1	BEVERAGES	0.000	0
4	4	2013-01-01	1	BOOKS	0.000	0
...	...	...	...	...	...	...
3000883	3000883	2017-08-15	9	POULTRY	438.133	0
3000884	3000884	2017-08-15	9	PREPARED FOODS	154.553	1
3000885	3000885	2017-08-15	9	PRODUCE	2419.729	148
3000886	3000886	2017-08-15	9	SCHOOL AND OFFICE SUPPLIES	121.000	8
3000887	3000887	2017-08-15	9	SEAFOOD	16.000	0

3000888 rows × 6 columns

Als nächstes extrahieren wir Zeitreihen für einen Artikel, die für das Modell verwendet werden sollen: Wir benötigen nur die Spalte „Verkäufe", die mithilfe der Spalte „Datum" indiziert wird. Extrahieren wir Zeitreihen von Geflügelverkäufen und sehen wir, wie die Saisonalität aussieht:

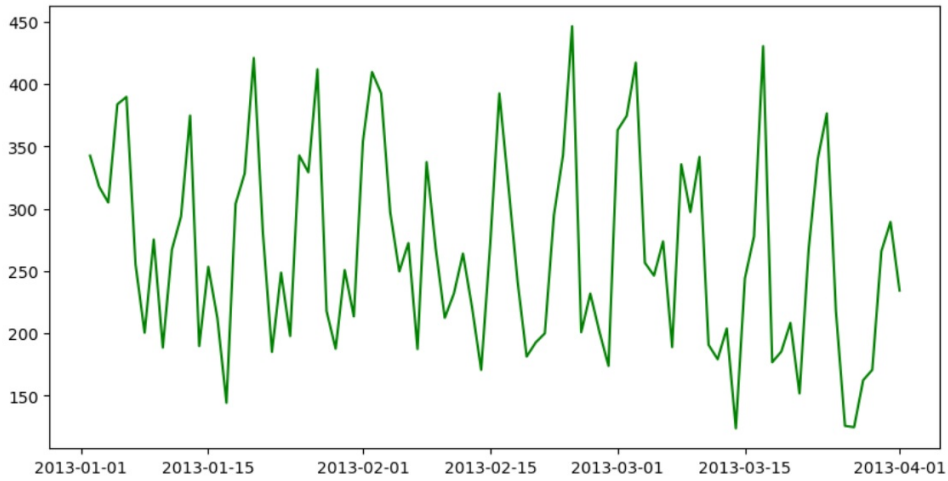
In [8]:
```
# modelling poultry sales
#df = data[(data['store_nbr'] == 2) & (data['family'] == 'BEVERAGES') & (data['sales'] > 0) ]
df = data[(data['store_nbr'] == 2) & (data['family'] == 'POULTRY') & (data['sales'] > 0) ]
df = df.set_index('date')
train_data = df['sales']
train_data2 = train_data[:-100]
type(train_data.index)
print(len(train_data))
```

```
1679
```

In [9]:
```
#checking seasonality in the data using plots
import matplotlib.pyplot as plt
fig, ax = plt.subplots(figsize = (12,8))
data_to_plot = train_data2
data_to_plot.index = pd.DatetimeIndex(data_to_plot.index).to_period('d')
data2 = data_to_plot.to_timestamp(freq='d')

plt.plot(data2[:90], label = 'actual data', color="green" )
```

Out[9]:
```
[<matplotlib.lines.Line2D at 0x1624034f6d0>]
```

Out[11]: [<matplotlib.lines.Line2D at 0x162403bbc10>]



Wenn wir uns das Datendiagramm ansehen, sehen wir eine Art wöchentliche Saisonalität.
Jetzt trainieren wir zwei TBATS-Modelle: mit und ohne COX_box-Transformator. Anschließend werden wir die Leistung beider Modelle anhand von Testdaten vergleichen. TBATS wird wöchentliche und monatliche Saisonalitäten verwenden, um das Modell zu erstellen. Importieren wir zunächst die erforderlichen Python-Pakete:

In [21]:
```
# !pip install tbats
#!pip install sktime
from sktime.forecasting.tbats import TBATS
from sktime.forecasting.model_selection import temporal_train_test_split
from sktime.forecasting.base import ForecastingHorizon


y_train, y_test = temporal_train_test_split(data2, test_size = 0.1)# 10% of the data for testing
y_train.index= pd.DatetimeIndex(y_train.index).to_period('d')
y_test.index= pd.DatetimeIndex(y_test.index).to_period('d')
fh = ForecastingHorizon(y_test.index, is_relative=False)

#box-cox transformation:a statistical technique that transforms your target variable
#so that your data closely resembles a normal distribution. This help making the residuals of the model as normal as possible
# and thus leads to better forecasting performance
# n_jobs: How many jobs to run in parallel when fitting TBATS model.
# monthly and weekly seasonalities
forecastor = TBATS(sp=[ 7, 30], use_box_cox=True, n_jobs=5)# with box_cox transformation
forecastor2 = TBATS(sp=[ 7, 30], use_box_cox=False, n_jobs=5)# box_cox transformation is disabled
fitted_model = forecastor.fit(y_train)
fitted_model2 = forecastor2.fit(y_train)
print(forecastor, forecastor2)
```

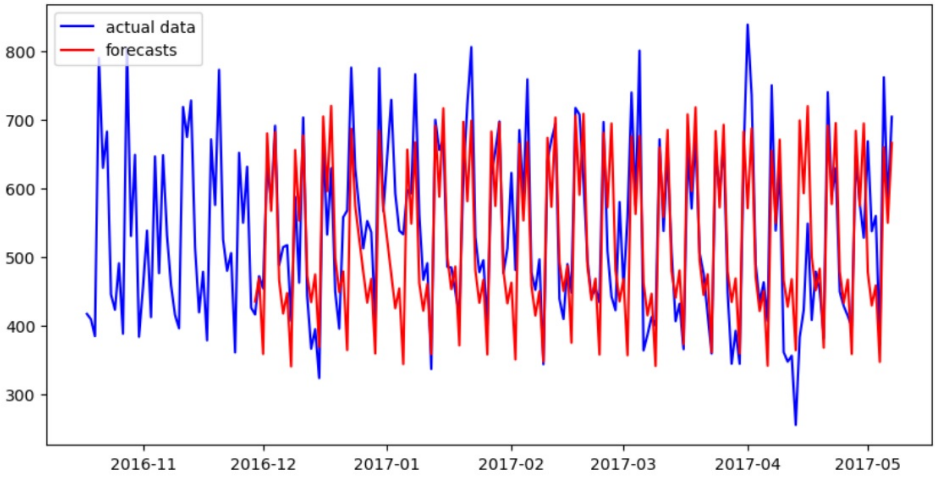
```
TBATS(n_jobs=5, sp=[7, 30], use_box_cox=True) TBATS(n_jobs=5, sp=[7, 30], use_box_cox=False)
```

In [262]:
```
# prediction
y_pred = forecastor.predict(fh)
y_pred2 = forecastor2.predict(fh)

y_pred[:31]
```

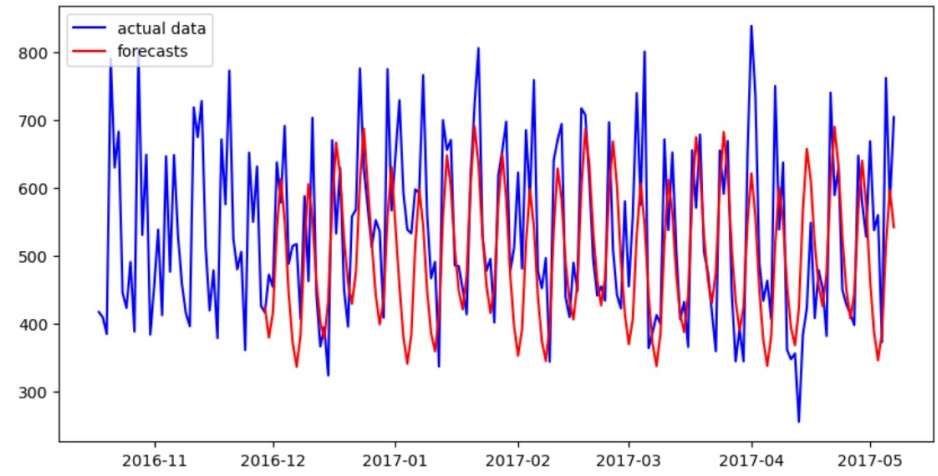
Out[262]:
```
date
2016-11-29    435.293772
2016-11-30    469.053532
2016-12-01    358.737733
2016-12-02    680.352287
2016-12-03    567.444919
2016-12-04    682.392106
2016-12-05    466.123288
2016-12-06    417.723508
2016-12-07    447.631427
2016-12-08    340.625047
2016-12-09    656.131723
2016-12-10    552.965208
2016-12-11    677.404144
2016-12-12    472.873148
```

Nach der Anpassung der TBATS-Modelle werden wir sie an 10 % der Zeitreihen testen. Lassen Sie uns Diagramme der tatsächlichen Werte mit den Prognosen beider Modelle vergleichen:

In [263]:
```
df_forecast=pd.DataFrame(data= y_pred, index=y_test.index)

df_forecast = df_forecast.to_timestamp(freq='d')
fig, ax = plt.subplots(figsize = (10,5))
train_data2.index = pd.DatetimeIndex(train_data2.index).to_period('d')
train = train_data2.to_timestamp(freq='d')
plt.plot(train[-200:], label = 'actual data', color="blue" )
plt.plot(df_forecast, label= 'forecasts', color = 'red')
plt.legend(loc="upper left")
plt.show()
```



In [264]:
```
df_forecast2=pd.DataFrame(data= y_pred2, index=y_test.index)

df_forecast2 = df_forecast2.to_timestamp(freq='d')
fig, ax = plt.subplots(figsize = (10,5))

plt.plot(train[-200:], label = 'actual data', color="blue" )
plt.plot(df_forecast2, label= 'forecasts', color = 'red')
plt.legend(loc="upper left")
plt.show()
```



Wenn wir abschließend die resultierenden Fehlerwerte beider Modelle überprüfen und vergleichen, sehen wir offensichtlich, dass die Einbeziehung des Cox-Box-Transformators zu einer besseren Leistung führt:

In [265]:
```
# TBATS forecasting model evaluation:
from sklearn.metrics import mean_absolute_percentage_error
print('mean absolute percentage error on test data using box-cox transformer is:', mean_absolute_percentage_error(y_pred, y_test))
print('mean absolute percentage error on test data without using box-cox transformer is:', mean_absolute_percentage_error(y_pred2
```

```
mean absolute percentage error on test data using box-cox transformer is: 0.12534766422768567
mean absolute percentage error on test data without using box-cox transformer is: 0.1808882858413106
```


```
{
 "cells": [
  {
   "cell_type": "code",
   "execution_count": 8,
   "id": "89de8ca2",
   "metadata": {},
   "outputs": [
    {
     "data": {
      "text/html": [
       "<div>\n",
       "<style scoped>\n",
       "    .dataframe tbody tr th:only-of-type {\n",
       "        vertical-align: middle;\n",
       "    }\n",
       "\n",
       "    .dataframe tbody tr th {\n",
       "        vertical-align: top;\n",
       "    }\n",
       "\n",
       "    .dataframe thead th {\n",
       "        text-align: right;\n",
       "    }\n",
       "</style>\n",
       "<table border=\"1\" class=\"dataframe\">\n",
```

```
"  <thead>\n",
"    <tr style=\"text-align: right;\">\n",
"      <th></th>\n",
"      <th>id</th>\n",
"      <th>date</th>\n",
"      <th>store_nbr</th>\n",
"      <th>family</th>\n",
"      <th>sales</th>\n",
"      <th>onpromotion</th>\n",
"    </tr>\n",
"  </thead>\n",
"  <tbody>\n",
"    <tr>\n",
"      <th>0</th>\n",
"      <td>0</td>\n",
"      <td>2013-01-01</td>\n",
"      <td>1</td>\n",
"      <td>AUTOMOTIVE</td>\n",
"      <td>0.000</td>\n",
"      <td>0</td>\n",
"    </tr>\n",
"    <tr>\n",
"      <th>1</th>\n",
"      <td>1</td>\n",
"      <td>2013-01-01</td>\n",
"      <td>1</td>\n",
"      <td>BABY CARE</td>\n",
"      <td>0.000</td>\n",
"      <td>0</td>\n",
"    </tr>\n",
"    <tr>\n",
"      <th>2</th>\n",
"      <td>2</td>\n",
"      <td>2013-01-01</td>\n",
"      <td>1</td>\n",
"      <td>BEAUTY</td>\n",
"      <td>0.000</td>\n",
"      <td>0</td>\n",
"    </tr>\n",
"    <tr>\n",
"      <th>3</th>\n",
"      <td>3</td>\n",
"      <td>2013-01-01</td>\n",
"      <td>1</td>\n",
"      <td>BEVERAGES</td>\n",
"      <td>0.000</td>\n",
"      <td>0</td>\n",
"    </tr>\n",
"    <tr>\n",
"      <th>4</th>\n",
"      <td>4</td>\n",
"      <td>2013-01-01</td>\n",
"      <td>1</td>\n",
"      <td>BOOKS</td>\n",
"      <td>0.000</td>\n",
"      <td>0</td>\n",
"    </tr>\n",
"    <tr>\n",
"      <th>...</th>\n",
"      <td>...</td>\n",
"      <td>...</td>\n",
"      <td>...</td>\n",
"      <td>...</td>\n",
"      <td>...</td>\n",
"      <td>...</td>\n",
"    </tr>\n",
"    <tr>\n",
"      <th>3000883</th>\n",
"      <td>3000883</td>\n",
"      <td>2017-08-15</td>\n",
"      <td>9</td>\n",
"      <td>POULTRY</td>\n",
"      <td>438.133</td>\n",
"      <td>0</td>\n",
"    </tr>\n",
"    <tr>\n",
"      <th>3000884</th>\n",
"      <td>3000884</td>\n",
"      <td>2017-08-15</td>\n",
"      <td>9</td>\n",
"      <td>PREPARED FOODS</td>\n",
"      <td>154.553</td>\n",
"      <td>1</td>\n",
"    </tr>\n",
"    <tr>\n",
"      <th>3000885</th>\n",
"      <td>3000885</td>\n",
"      <td>2017-08-15</td>\n",
"      <td>9</td>\n",
"      <td>PRODUCE</td>\n",
"      <td>2419.729</td>\n",
"      <td>148</td>\n",
"    </tr>\n",
"    <tr>\n",
"      <th>3000886</th>\n",
"      <td>3000886</td>\n",
"      <td>2017-08-15</td>\n",
"      <td>9</td>\n",
"      <td>SCHOOL AND OFFICE SUPPLIES</td>\n",
"      <td>121.000</td>\n",
"      <td>8</td>\n",
"    </tr>\n",
"    <tr>\n",
"      <th>3000887</th>\n",
```

```
       "      <td>3000887</td>\n",
       "      <td>2017-08-15</td>\n",
       "      <td>9</td>\n",
       "      <td>SEAFOOD</td>\n",
       "      <td>16.000</td>\n",
       "      <td>0</td>\n",
       "    </tr>\n",
       "  </tbody>\n",
       "</table>\n",
       "<p>3000888 rows × 6 columns</p>\n",
       "</div>"
      ],
      "text/plain": [
       "              id        date  store_nbr                    family     sales  \\\n",
       "0              0  2013-01-01          1                AUTOMOTIVE     0.000   \n",
       "1              1  2013-01-01          1                 BABY CARE     0.000   \n",
       "2              2  2013-01-01          1                    BEAUTY     0.000   \n",
       "3              3  2013-01-01          1                 BEVERAGES     0.000   \n",
       "4              4  2013-01-01          1                     BOOKS     0.000   \n",
       "...          ...         ...        ...                       ...       ...   \n",
       "3000883  3000883  2017-08-15          9                   POULTRY   438.133   \n",
       "3000884  3000884  2017-08-15          9            PREPARED FOODS   154.553   \n",
       "3000885  3000885  2017-08-15          9                   PRODUCE  2419.729   \n",
       "3000886  3000886  2017-08-15          9  SCHOOL AND OFFICE SUPPLIES   121.000   \n",
       "3000887  3000887  2017-08-15          9                   SEAFOOD    16.000   \n",
       "\n",
       "         onpromotion  \n",
       "0                  0  \n",
       "1                  0  \n",
       "2                  0  \n",
       "3                  0  \n",
       "4                  0  \n",
       "...              ...  \n",
       "3000883            0  \n",
       "3000884            1  \n",
       "3000885          148  \n",
       "3000886            8  \n",
       "3000887            0  \n",
       "\n",
       "[3000888 rows x 6 columns]"
      ]
     },
     "execution_count": 8,
     "metadata": {},
     "output_type": "execute_result"
    }
   ],
   "source": [
    "import pandas as pd\n",
    "# https://www.kaggle.com/competitions/store-sales-time-series-forecasting/data\n",
    "data = pd.read_csv('Store_sales_train.csv')\n",
    "data"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": 18,
   "id": "26e30043",
   "metadata": {},
   "outputs": [
    {
     "name": "stdout",
     "output_type": "stream",
     "text": [
      "1679\n"
     ]
    }
   ],
   "source": [
    "# modelling poultry sales\n",
    "#df = data[(data['store_nbr'] == 2) & (data['family'] == 'BEVERAGES') & (data['sales'] > 0) ] \n",
    "df = data[(data['store_nbr'] == 2) & (data['family'] == 'POULTRY') & (data['sales'] > 0) ] \n",
    "df = df.set_index('date')\n",
    "train_data = df['sales']\n",
    "train_data2 = train_data[:-100]\n",
    "type(train_data.index)\n",
    "print(len(train_data))"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": 10,
   "id": "4b1ae127",
   "metadata": {},
   "outputs": [
    {
     "data": {
      "text/plain": [
       "[<matplotlib.lines.Line2D at 0x2086fc84220>]"
      ]
     },
     "execution_count": 10,
     "metadata": {},
     "output_type": "execute_result"
    },
    {
     "data": {
      "image/png": "iVBORw0KGgoAAAANSUhEUgAAA0QAAAGsCAYAAAD9kfoAAAAAOXRFWHRTb2Z0d2FyZQBNYXRwbG90bGliIHZlcnNpb24zLjcuMSwgaHR0
      "text/plain": [
       "<Figure size 1000x500 with 1 Axes>"
      ]
     },
     "metadata": {},
     "output_type": "display_data"
```

```
   }
  ],
  "source": [
   "#checking seasonality in the data using plots\n",
   "import matplotlib.pyplot as plt\n",
   "fig, ax = plt.subplots(figsize = (10,5))\n",
   "data_to_plot = train_data2\n",
   "data_to_plot.index = pd.DatetimeIndex(data_to_plot.index).to_period('d')\n",
   "data2 = data_to_plot.to_timestamp(freq='d')\n",
   "\n",
   "plt.plot(data2[:90], label = 'actual data', color=\"green\" )"
  ]
 },
 {
  "cell_type": "code",
  "execution_count": 21,
  "id": "05e90646",
  "metadata": {},
  "outputs": [
   {
    "name": "stdout",
    "output_type": "stream",
    "text": [
     "TBATS(n_jobs=5, sp=[7, 30], use_box_cox=True) TBATS(n_jobs=5, sp=[7, 30], use_box_cox=False)\n"
    ]
   }
  ],
  "source": [
   "# !pip install tbats\n",
   "#!pip install sktime\n",
   "from sktime.forecasting.tbats import TBATS\n",
   "from sktime.forecasting.model_selection import temporal_train_test_split\n",
   "from sktime.forecasting.base import ForecastingHorizon\n",
   "\n",
   "\n",
   "y_train, y_test = temporal_train_test_split(data2, test_size = 0.1)# 10% of the data for testing \n",
   "y_train.index= pd.DatetimeIndex(y_train.index).to_period('d')\n",
   "y_test.index= pd.DatetimeIndex(y_test.index).to_period('d')\n",
   "fh = ForecastingHorizon(y_test.index, is_relative=False)\n",
   "\n",
   "#box-cox transformation:a statistical technique that transforms your target variable\n",
   "#so that your data closely resembles a normal distribution. This help making the residuals of the model as normal as po
   "# and thus leads to better forecasting performance\n",
   "# n_jobs: How many jobs to run in parallel when fitting TBATS model.\n",
   "# monthly and weekly seasonalities\n",
   "forecastor = TBATS(sp=[ 7, 30], use_box_cox=True, n_jobs=5)# with box_cox transformation \n",
   "forecastor2 = TBATS(sp=[ 7, 30], use_box_cox=False, n_jobs=5)# box_cox transformation is disabled\n",
   "fitted_model = forecastor.fit(y_train)\n",
   "fitted_model2 = forecastor2.fit(y_train)\n",
   "print(forecastor, forecastor2)"
  ]
 },
 {
  "cell_type": "code",
  "execution_count": 22,
  "id": "bdaa6a22",
  "metadata": {},
  "outputs": [
   {
    "data": {
     "text/plain": [
      "date\n",
      "2016-11-29    435.293771\n",
      "2016-11-30    469.053531\n",
      "2016-12-01    358.737733\n",
      "2016-12-02    680.352288\n",
      "2016-12-03    567.444920\n",
      "2016-12-04    682.392107\n",
      "2016-12-05    466.123289\n",
      "2016-12-06    417.723508\n",
      "2016-12-07    447.631427\n",
      "2016-12-08    340.625047\n",
      "2016-12-09    656.131724\n",
      "2016-12-10    552.965209\n",
      "2016-12-11    677.404144\n",
      "2016-12-12    472.873147\n",
      "2016-12-13    434.279893\n",
      "2016-12-14    474.687027\n",
      "2016-12-15    369.224240\n",
      "2016-12-16    704.953463\n",
      "2016-12-17    595.895730\n",
      "2016-12-18    720.441934\n",
      "2016-12-19    498.540929\n",
      "2016-12-20    448.743349\n",
      "2016-12-21    479.056565\n",
      "2016-12-22    364.409213\n",
      "2016-12-23    687.148571\n",
      "2016-12-24    574.165854\n",
      "2016-12-26    478.701776\n",
      "2016-12-27    433.658950\n",
      "2016-12-28    468.119896\n",
      "2016-12-29    359.386117\n",
      "2016-12-30    684.294173\n",
      "Freq: D, Name: sales, dtype: float64"
     ]
    },
    "execution_count": 22,
    "metadata": {},
    "output_type": "execute_result"
   }
  ],
  "source": [
   "# forecasting using the first and second models: with and without Cox_Box transformation\n",
   "y_pred = forecastor.predict(fh)\n",
```

```
    "y_pred2 = forecastor2.predict(fh)\n",
    "\n",
    "y_pred[:31]"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": 23,
   "id": "7e3c45a2",
   "metadata": {},
   "outputs": [
    {
     "data": {
      "image/png": "iVBORw0KGgoAAAANSUhEUgAAAz8AAAGsCAYAAAADzOBmHAAAAOXRFWHRTb2Z0d2FyZQBNYXRwbG90bGliIHZlcnNpb24zLjcuMSwgaHR0
      "text/plain": [
       "<Figure size 1000x500 with 1 Axes>"
      ]
     },
     "metadata": {},
     "output_type": "display_data"
    }
   ],
   "source": [
    "# plotting true against forecasts using the first model.\n",
    "df_forecast=pd.DataFrame(data= y_pred, index=y_test.index)\n",
    "\n",
    "df_forecast = df_forecast.to_timestamp(freq='d')\n",
    "fig, ax = plt.subplots(figsize = (10,5))\n",
    "train_data2.index = pd.DatetimeIndex(train_data2.index).to_period('d')\n",
    "train = train_data2.to_timestamp(freq='d')\n",
    "plt.plot(train[-200:], label = 'actual data', color=\"blue\" )\n",
    "plt.plot(df_forecast, label= 'forecasts', color = 'red')\n",
    "plt.legend(loc=\"upper left\")\n",
    "plt.show()"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": 24,
   "id": "7a3dae9b",
   "metadata": {},
   "outputs": [
    {
     "data": {
      "image/png": "iVBORw0KGgoAAAANSUhEUgAAAz8AAAGsCAYAAAADzOBmHAAAAOXRFWHRTb2Z0d2FyZQBNYXRwbG90bGliIHZlcnNpb24zLjcuMSwgaHR0
      "text/plain": [
       "<Figure size 1000x500 with 1 Axes>"
      ]
     },
     "metadata": {},
     "output_type": "display_data"
    }
   ],
   "source": [
    "# plotting true against forecasts using the second model.\n",
    "df_forecast2=pd.DataFrame(data= y_pred2, index=y_test.index)\n",
    "\n",
    "df_forecast2 = df_forecast2.to_timestamp(freq='d')\n",
    "fig, ax = plt.subplots(figsize = (10,5))\n",
    "\n",
    "plt.plot(train[-200:], label = 'actual data', color=\"blue\" )\n",
    "plt.plot(df_forecast2, label= 'forecasts', color = 'red')\n",
    "plt.legend(loc=\"upper left\")\n",
    "plt.show()"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": 25,
   "id": "fbabbd7a",
   "metadata": {},
   "outputs": [
    {
     "name": "stdout",
     "output_type": "stream",
     "text": [
      "mean absolute percentage error on test data using box-cox transformer is: 0.12534766373642992\n",
      "mean absolute percentage error on test data without using box-cox transformer is: 0.18088828584131061\n"
     ]
    }
   ],
   "source": [
    "# TBATS forecasting model evaluation: comparison\n",
    "from sklearn.metrics import mean_absolute_percentage_error\n",
    "print('mean absolute percentage error on test data using box-cox transformer is:', mean_absolute_percentage_error(y_pre
    "print('mean absolute percentage error on test data without using box-cox transformer is:', mean_absolute_percentage_err
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "b4f011e8",
   "metadata": {},
   "outputs": [],
   "source": []
  }
 ],
 "metadata": {
  "kernelspec": {
   "display_name": "Python 3 (ipykernel)",
   "language": "python",
   "name": "python3"
  },
  "language_info": {
```

```
   "codemirror_mode": {
    "name": "ipython",
    "version": 3
   },
   "file_extension": ".py",
   "mimetype": "text/x-python",
   "name": "python",
   "nbconvert_exporter": "python",
   "pygments_lexer": "ipython3",
   "version": "3.9.13"
  }
 },
 "nbformat": 4,
 "nbformat_minor": 5
}
```


Multivariate Zeitreihenvorhersage

Eine multivariate Zeitreihe besteht aus mehreren voneinander abhängigen Variablen, wobei jede Variable nicht nur eine Zeitabhängigkeit von ihren historischen Daten, sondern auch von den vergangenen Werten anderer Variablen aufweist. Multivariate Zeitreihenvorhersagen finden Anwendung in vielen Bereichen wie Energie, Aerologie, Meteorologie, Finanzen, Transportwesen usw. Eine der am häufigsten verwendeten Methoden für multivariate Zeitreihenvorhersagen ist die Vektorautoregression (VAR). In einem VAR-Algorithmus ist jeder Variablenwert eine lineare Beziehung der vergangenen Werte beider: der Variablen selbst und aller anderen Variablen, die als Vektoren dargestellt werden: Gewichtsvektor, Fehlervektor usw. Die VAR-Gleichung wird als beschrieben :

$$y_t = a + w_1 * y_{t-1} + ... + w_p * y_{t-p} + e$$

wobei a, w, y, e alle Vektoren sind und daher die vorherige Gleichung wie folgt ausgedrückt wird:

$$\begin{bmatrix} y1 \\ \vdots \\ yn \end{bmatrix} = \begin{bmatrix} a1 \\ \vdots \\ an \end{bmatrix} + \begin{bmatrix} w11 & \dots & w1n \\ \vdots & \vdots & \vdots \\ wn1 & \dots & wnn \end{bmatrix} \begin{bmatrix} y1(t-1) \\ \vdots \\ yn(t-1) \end{bmatrix} + \dots\dots + \begin{bmatrix} w'11 & \dots & w'1n \\ \vdots & \vdots & \vdots \\ w'n1 & \dots & w'nn \end{bmatrix} \begin{bmatrix} y1(t-p) \\ \vdots \\ yn(t-p) \end{bmatrix} + \begin{bmatrix} \varepsilon 1 \\ \vdots \\ \varepsilon n \end{bmatrix}$$

Dabei ist P die Anzahl der vergangenen Verzögerungen, die zur Berechnung der Prognosen verwendet wurden, n die Anzahl der voneinander abhängigen Variablen und ε der Vektor des weißen Rauschens.

Multivariate Zeitreihenvorhersage Übung

Wir beginnen mit dem Import der Zeitreihen. Wir werden zwei Datensätze, Geld und Ausgaben, zu einer Zeitreihe mit täglichen Aufzeichnungen zusammenführen. Anschließend werden wir anhand dieser Zeitreihen eine Autoregression und ein multivariates LSTM-Zeitreihenvorhersagemodell erstellen. Lasst uns die Datensätze importieren und einen Blick darauf werfen

In [3]:
```
# downloading dataset
import pandas as pd
money = pd.read_csv('M2SLMoneyStock.csv')
spending = pd.read_csv('PCEPersonalSpending.csv')
df =pd.merge(money, spending, on='Date', how='outer')
# set index to Date
df = df.set_index('Date')
# convert index into datetime format
df.index = pd.DatetimeIndex(df.index).to_period('m')
df
```

Out[3]:

Date	Money	Spending
1995-01	3492.4	4851.2
1995-02	3489.9	4850.8
1995-03	3491.1	4885.4
1995-04	3499.2	4890.2
1995-05	3524.2	4933.1
...	...	...
2015-08	12096.8	12394.0
2015-09	12153.8	12392.8
2015-10	12187.7	12416.1
2015-11	12277.4	12450.1

Wir werden 90 % der Zeitreihen zum Training der Modelle verwenden. Wir werden ein AR-Modell unter Verwendung der letzten 5 Werte jeder Variablen erstellen. Lassen Sie uns die Zeitreihe aufteilen, die erforderlichen Python-Pakete importieren, das Modell an den Trainingssatz anpassen und schließlich das angepasste Modell testen.

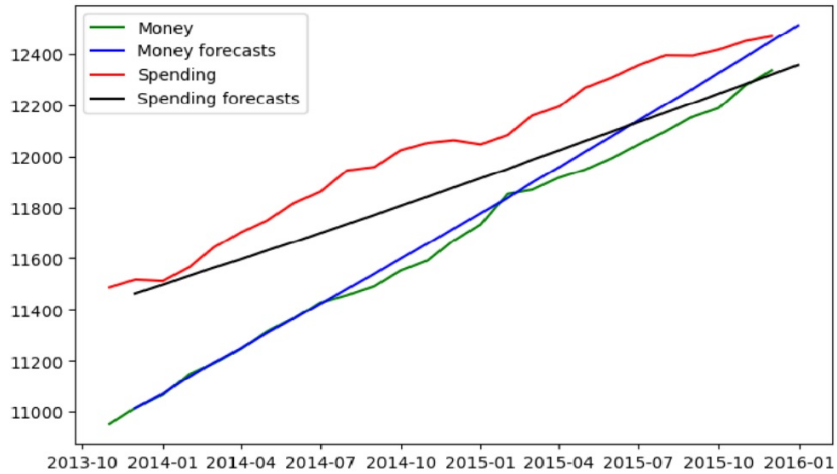
In [5]:
```
# splitting the data into train and validation sets: 90% for training, 10% for validation
data= df.dropna()
train = data[:int(0.9*(len(df)))]
test = data[int(0.9*(len(df))):]
```

In [6]:
```
# using vector autogregression model
from statsmodels.tsa.vector_ar.var_model import VAR
# a pth-order VAR refers to a VAR model which includes lags for the last p time periods
# fit(maxlags): Maximum number of lags to check for order selection,
# defaults to 12 * (nobs/100.)**(1./4)
# we set order = 5
model = VAR(endog=train, freq='M')
fitted_var = model.fit(5)# fitted_var.summary(): to access model output


# make predictions and evaluate the model on validation set
# VARResults.forecast: Produce linear minimum MSE forecasts for desired number of steps ahead, using prior values y
lagged_Vals = train.values[-10:]# The VAR .forecast() requires passing in a lag order number of previous observations.
predictions = fitted_var.forecast(y=lagged_Vals, steps=len(test))
```

Jetzt werten wir das angepasste Autoregressionsmodell am Testsatz aus und zeichnen die tatsächlichen Werte im Vergleich zu den Vorhersagen für beide Variablen auf: „Money" und „Spending".

In [7]:
```
#the predictions are made as an array, where each list represents the predictions of a row in the dataset.
# we need first to convert the array into  a dataframe
from sklearn.metrics import mean_absolute_percentage_error
# locate the predictions in a new dataframe with the same col names: without 'CO(GT)'
cols =df.columns
idx = test.index
pred = pd.DataFrame(index=idx,columns=[cols])
for j in range(0,2):
    for i in range(0, len(predictions)):
        pred.iloc[i][j] = predictions[i][j]

#calculate MAPE for all cols
for col in cols:
    print('MAPE value for', col, 'is : ', mean_absolute_percentage_error(pred[col], test[col]))
```

```
MAPE value for Money is :  0.008557826316923147
MAPE value for Spending is :  0.009869428903370645
```



JJetzt verwenden wir den LSTM-Algorithmus für denselben Datensatz, um ein multivariates Zeitreihenmodell zu erstellen. Zuerst müssen wir die Zeitreihe mit dem StandardScaler-Transformator in eine Standardverteilung umwandeln.Wir werden die Variable „Geld" als Zielvariable des LSTM-Modells verwenden:

In [15]:
```
# using LSTM;
# first we need to scale input and output of LSTM using standard scaler
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
x_train_scaled =scaler.fit_transform(train)
x_train_scaled.shape
```

Out[15]: (226, 2)

In [319]:
```
# to predict money variable
scaler2= StandardScaler()
y_train_scaled =scaler2.fit_transform(train[['Money']])
y_train_scaled.shape
```

Out[319]: (226, 1)

Ein wichtiger Schritt besteht darin, die Eingabe- und Ausgabesequenzen für LSTM vorzubereiten. Wir werden die gleiche Anzahl von Verzögerungen wie für AR verwenden, also 5 Verzögerungen:

In [19]:
```
# for lag= 5 we need to create the sequences of the input array manually;reshaping input data
import numpy as np
x_train=[]
y_train = []
for i in range(5, 226):
    x_train.append(x_train_scaled[i-5:i])
    y_train.append(y_train_scaled[i][0])
x_train= np.array(x_train)
y_train = np.array(y_train)
print(x_train.shape)
```

```
(221, 5, 2)
```

Jetzt erstellen wir unser LSTM-Modell, das aus einer Eingabeschicht, einer verborgenen Schicht, zwei Dropout-Schichten und einer Ausgabeschicht besteht. Lassen Sie uns das Modell kompilieren und an Trainingsdaten anpassen:

In [21]:
```
# build LSTM model
from tensorflow import keras
from tensorflow.keras.layers import  Dropout, Dense, LSTM
from tensorflow.keras.models import Sequential

model = keras.Sequential()
model.add(LSTM(units=50, return_sequences=True, input_shape=(5,2)))
model.add(Dropout(0.2))
model.add(LSTM(units=50))
model.add(Dropout(0.2))
model.add(Dense(1))
```

In [22]:
```
model.compile(loss='mean_squared_error',optimizer='adam')
history = model.fit(x_train, y_train, epochs=200, batch_size= 32, shuffle=False, validation_split=0.2)
```

```
6/6 [==============================] - 0s 10ms/step - loss: 0.0078 - val_loss: 0.0078
Epoch 168/200
6/6 [==============================] - 0s 10ms/step - loss: 0.0064 - val_loss: 0.0371
Epoch 169/200
6/6 [==============================] - 0s 9ms/step - loss: 0.0053 - val_loss: 0.0170
Epoch 170/200
6/6 [==============================] - 0s 9ms/step - loss: 0.0054 - val_loss: 0.0192
Epoch 171/200
6/6 [==============================] - 0s 11ms/step - loss: 0.0058 - val_loss: 0.0295
Epoch 172/200
6/6 [==============================] - 0s 11ms/step - loss: 0.0053 - val_loss: 0.0368
Epoch 173/200
6/6 [==============================] - 0s 11ms/step - loss: 0.0066 - val_loss: 0.0092
Epoch 174/200
6/6 [==============================] - 0s 9ms/step - loss: 0.0061 - val_loss: 0.0344
Epoch 175/200
6/6 [==============================] - 0s 9ms/step - loss: 0.0074 - val_loss: 0.0133
Epoch 176/200
6/6 [==============================] - 0s 9ms/step - loss: 0.0041 - val_loss: 0.0094
```

Um das Modell am Testsatz auszuwerten, müssen wir die letzten 5 Werte des Trainingssatzes anhängen, da das Modell Vorhersagen mit 5 Verzögerungen erzeugt. Die Testsequenz sollte skaliert und vorbereitet werden, bevor Vorhersagen getroffen werden, die wiederum ebenfalls umgekehrt skaliert werden sollten. model.predict() wird verwendet, um Vorhersagen von Testzeitreihen zu berechnen:

In [32]:
```
# appending the last 10 of train set to the test set
train_last_10= train[-5:]
extended_test= pd.concat((train_last_10, test), axis=0)
# rescaling test data
scaled_test= scaler.fit_transform(extended_test)
scaled_test.shape

# rescaling y
y_test_scaled = scaler2.fit_transform(extended_test[['Money']])
y_test_scaled.shape
```

Out[32]:
```
(31, 1)
```

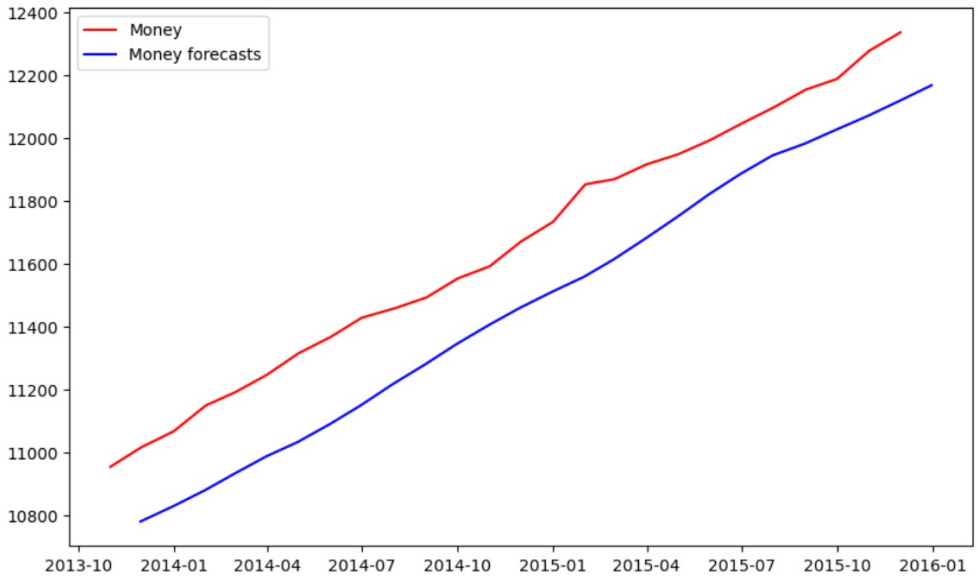
In [33]:
```
# reshaping input data for testing the model
x_test=[]
for i in range(5, 31):
    x_test.append(scaled_test[i-5:i])
x_test= np.array(x_test)
print(x_test.shape)
```

```
(26, 5, 2)
```

In [34]:
```
# make predictions
y_test=model.predict(x_test)
# inverse scaling of money forecasts
y= scaler2.inverse_transform(y_test)
y
```

Überprüfen wir die Darstellung der Vorhersagen anhand der tatsächlichen Geldwerte:

In [35]:
```
# plotting predictions against actual money values
fig = plt.subplots(figsize=(12,8))
test2 = test.to_timestamp(freq='d')
df_forecast=pd.DataFrame(data=y, index=idx)
plt.plot(test2['Money'],color="red", label ='Money')
plt.plot(df_forecast, color="blue", label ='Money forecasts')
plt.legend(loc="upper left")
plt.show()
```



Lassen Sie uns den Vorhersagefehlerwert für die Variable „money" überprüfen:

In [36]:
```
# MAPE between forecasts and actual money
print('MAPE value for Money is:', mean_absolute_percentage_error(y, test['Money']))
```

```
MAPE value for Money is: 0.014847821070793958
```


```
{
 "cells": [
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "a833112b",
   "metadata": {},
   "outputs": [],
   "source": [
    "# downloading dataset\n",
    "import pandas as pd\n",
    "money = pd.read_csv('M2SLMoneyStock.csv')\n",
    "spending = pd.read_csv('PCEPersonalSpending.csv')\n",
    "df =pd.merge(money, spending, on='Date', how='outer')\n",
    "#df2 = pd.merge(passengers, restaurant, on='Date', how='outer')\n",
    "# set index to Date\n",
    "df = df.set_index('Date')\n",
    "# convert index into datetime format\n",
    "df.index = pd.DatetimeIndex(df.index).to_period('m')\n",
    "df"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "85e39b7d",
   "metadata": {},
   "outputs": [],
   "source": [
    "#checking stationarity, \n",
    "# Cointegration is a statistical method used to test the correlation between two or more non-stationary time series \n"
    "# in the long run or for a specified period\n",
    "from statsmodels.tsa.vector_ar.vecm import coint_johansen\n",
    "#since the test works for only 12 variables, I have randomly dropped\n",
    "#in the next iteration, I would drop another and check the eigenvalues\n",
    "\n",
    "#coint_johansen(endog, det_order, k_ar_diff); k_ar_diff Number of lagged differences in the model.\n",
    "# det_order: determintation order; -1 no-determinstic terms: the order of time polynomial in the null-hypothesis\n",
    "data= df.dropna()\n",
    "# Eigenvalues of  coefficient matrix\n",
    "result=coint_johansen(data,-1,1).eig\n",
    "print(result)"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "28c1f9fb",
   "metadata": {},
   "outputs": [],
   "source": [
    "df"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "647d01ba",
   "metadata": {},
   "outputs": [],
   "source": [
    "# splitting the data into train and validation sets: 90% for training, 10% for validation\n",
```

```
   "data= df.dropna()\n",
   "train = data[:int(0.9*(len(df)))]\n",
   "test = data[int(0.9*(len(df))):]"
  ]
 },
 {
  "cell_type": "code",
  "execution_count": null,
  "id": "ed8eb81e",
  "metadata": {},
  "outputs": [],
  "source": [
   "# using vector autogregression model \n",
   "from statsmodels.tsa.vector_ar.var_model import VAR\n",
   "# a pth-order VAR refers to a VAR model which includes lags for the last p time periods\n",
   "# fit(maxlags): Maximum number of lags to check for order selection,\n",
   "# defaults to 12 * (nobs/100.)**(1./4)\n",
   "# we set order = 5 \n",
   "model = VAR(endog=train, freq='M')\n",
   "fitted_var = model.fit(5)# fitted_var.summary(): to access model output\n",
   "\n",
   "\n",
   "# make predictions and evaluate the model on validation set\n",
   "# VARResults.forecast: Produce linear minimum MSE forecasts for desired number of steps ahead, using prior values y\n",
   "lagged_Vals = train.values[-10:]# The VAR .forecast() requires passing in a lag order number of previous observations.\
   "predictions = fitted_var.forecast(y=lagged_Vals, steps=len(test))"
  ]
 },
 {
  "cell_type": "code",
  "execution_count": null,
  "id": "b81c5df3",
  "metadata": {},
  "outputs": [],
  "source": [
   "#the predictions are made as an array, where each list represents the predictions of a row in the dataset.\n",
   "# we need first to convert the array into  a dataframe\n",
   "from sklearn.metrics import mean_absolute_percentage_error\n",
   "# locate the predictions in a new dataframe with the same col names: without 'CO(GT)'\n",
   "cols =df.columns\n",
   "idx = test.index\n",
   "pred = pd.DataFrame(index=idx,columns=[cols])\n",
   "for j in range(0,2):\n",
   "    for i in range(0, len(predictions)):\n",
   "        pred.iloc[i][j] = predictions[i][j]\n",
   "\n",
   "#calculate MAPE for all cols\n",
   "for col in cols:\n",
   "    print('MAPE value for', col, 'is : ', mean_absolute_percentage_error(pred[col], test[col]))"
  ]
 },
 {
  "cell_type": "code",
  "execution_count": null,
  "id": "26c20cfb",
  "metadata": {},
  "outputs": [],
  "source": [
   "#plot cols against predictions\n",
   "import matplotlib.pyplot as plt\n",
   "fig = plt.subplots(figsize=(8,5))\n",
   "test2 = test.to_timestamp(freq='d')\n",
   "\n",
   "df_forecast=pd.DataFrame(data=pred, index=idx, columns=cols)\n",
   "plt.plot(test2['Money'],color=\"green\", label ='Money')\n",
   "plt.plot(pred['Money'], color=\"blue\", label ='Money forecasts')\n",
   "plt.plot(test2['Spending'],color=\"red\", label ='Spending')\n",
   "plt.plot(pred['Spending'], color=\"black\", label ='Spending forecasts')\n",
   "plt.legend(loc=\"upper left\")\n",
   "plt.show()\n",
   "# since each variable is a linear function of past lags of itself and past lags of the other variables, we get fitted l
  ]
 },
 {
  "cell_type": "code",
  "execution_count": null,
  "id": "eedc9c9f",
  "metadata": {},
  "outputs": [],
  "source": [
   "# using LSTM;\n",
   "# first we need to scale input and output of LSTM using standard scaler\n",
   "from sklearn.preprocessing import StandardScaler\n",
   "scaler = StandardScaler()\n",
   "x_train_scaled =scaler.fit_transform(train)\n",
   "x_train_scaled.shape "
  ]
 },
 {
  "cell_type": "code",
  "execution_count": null,
  "id": "514e95e2",
  "metadata": {},
  "outputs": [],
  "source": [
   "#using money variable as input time series\n",
   "scaler2= StandardScaler()\n",
   "y_train_scaled =scaler2.fit_transform(train[['Money']])\n",
   "y_train_scaled.shape"
  ]
 },
 {
  "cell_type": "code",
  "execution_count": null,
```

```
   "id": "bb46576b",
   "metadata": {},
   "outputs": [],
   "source": [
    "# for lag= 5 we need to create the sequences of the input array manually;reshaping input data\n",
    "import numpy as np\n",
    "x_train=[]\n",
    "y_train = []\n",
    "for i in range(5, 226):\n",
    "    x_train.append(x_train_scaled[i-5:i])\n",
    "    y_train.append(y_train_scaled[i][0])\n",
    "x_train= np.array(x_train)    \n",
    "y_train = np.array(y_train) \n",
    "print(x_train.shape)"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "dca9df45",
   "metadata": {},
   "outputs": [],
   "source": [
    "# build LSTM model \n",
    "from tensorflow import keras\n",
    "from tensorflow.keras.layers import  Dropout, Dense, LSTM\n",
    "from tensorflow.keras.models import Sequential\n",
    "\n",
    "model = keras.Sequential()\n",
    "model.add(LSTM(units=50, return_sequences=True, input_shape=(5,2)))\n",
    "model.add(Dropout(0.2))\n",
    "model.add(LSTM(units=50))\n",
    "model.add(Dropout(0.2))\n",
    "model.add(Dense(1))"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "16544270",
   "metadata": {},
   "outputs": [],
   "source": [
    "model.compile(loss='mean_squared_error',optimizer='adam')\n",
    "history = model.fit(x_train, y_train, epochs=200, batch_size= 32, shuffle=False, validation_split=0.2)"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "43f649a7",
   "metadata": {},
   "outputs": [],
   "source": [
    "# appending the last 5 of train set to the test set: to be used for forecasting, since we use 5 lags \n",
    "train_last_5= train[-5:]\n",
    "extended_test= pd.concat((train_last_5, test), axis=0)\n",
    "# rescaling test data\n",
    "scaled_test= scaler.fit_transform(extended_test)\n",
    "scaled_test.shape\n",
    "\n",
    "# rescaling y \n",
    "y_test_scaled = scaler2.fit_transform(extended_test[['Money']])\n",
    "y_test_scaled.shape"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "2a5c8162",
   "metadata": {},
   "outputs": [],
   "source": [
    "# reshaping input data for testing the model\n",
    "x_test=[]\n",
    "for i in range(5, 31):\n",
    "    x_test.append(scaled_test[i-5:i])    \n",
    "x_test= np.array(x_test)    \n",
    "print(x_test.shape)"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "ba2490c1",
   "metadata": {},
   "outputs": [],
   "source": [
    "# forecasting on test data\n",
    "y_test=model.predict(x_test)\n",
    "# inverse scaling of money forecasts\n",
    "y= scaler2.inverse_transform(y_test)\n",
    "y"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "bfd59e08",
   "metadata": {},
   "outputs": [],
   "source": [
    "# plotting predictions against actual money values\n",
    "fig = plt.subplots(figsize=(10,6))\n",
```

```
    "test2 = test.to_timestamp(freq='d')\n",
    "df_forecast=pd.DataFrame(data=y, index=idx)\n",
    "plt.plot(test2['Money'],color=\"red\", label ='Money')\n",
    "plt.plot(df_forecast, color=\"blue\", label ='Money forecasts')\n",
    "plt.legend(loc=\"upper left\")\n",
    "plt.show()"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "1e7d73eb",
   "metadata": {},
   "outputs": [],
   "source": [
    "# MAPE between forecasts and actual money\n",
    "print('MAPE value for Money is:', mean_absolute_percentage_error(y, test['Money']))\n"
   ]
  }
 ],
 "metadata": {
  "kernelspec": {
   "display_name": "Python 3 (ipykernel)",
   "language": "python",
   "name": "python3"
  },
  "language_info": {
   "codemirror_mode": {
    "name": "ipython",
    "version": 3
   },
   "file_extension": ".py",
   "mimetype": "text/x-python",
   "name": "python",
   "nbconvert_exporter": "python",
   "pygments_lexer": "ipython3",
   "version": "3.9.13"
  }
 },
 "nbformat": 4,
 "nbformat_minor": 5
}
```


Die resultierenden Fehlerwerte beider Modelle weisen auf eine höhere Leistung des Autoregressionsvektormodells als des LSTM-Modells hin.