```
# -*- coding: utf-8 -*-
"""
Created on Fri Apr  6 03:30:21 2018

@author: Tawanda Vera
"""

# Import the libraries and read the Gold price data
import sys
import warnings
import itertools
import calendar
import pandas as pd
import quandl
import numpy as np
from scipy import stats
import statsmodels.api as sm
from statsmodels.tsa.stattools import adfuller
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')  # A matplotlib style of fivethirtyeight

##############################################################################
# Model 1. Forecasting Gold Using the OLS Model
###############################################################################

# Step 1: import the gold price data
# since years prior to 1979 did not have complete daily data, the series only
# starts from 1979.
gold = quandl.get("WGC/GOLD_DAILY_USD",
                  authtoken="nPxEmsTFZUATpB25wB_-", start_date="1979-01-01")


print('gold.tail(5):', gold.tail(5))

#logarithm of gold price
gold2 = np.log(gold['Value']).values

print('Log of Gold price:', gold2)
# plot the gold price evolution
plt.plot(gold)
plt.grid()
plt.title("Gold price evolution from 1979 until 2018")
plt.xlabel("Period")
plt.ylabel("USD per troy ounce")

# Forecast Gold using the current and lagged Gold prices
lag = gold2[:-1]  # gold price evolution in the previous period
gold3 = gold2[1:]  # gold price evolution in the current period


beta, alpha, r_value, p_value, std_err = stats.linregress(gold3, lag)
print('beta, alpha:', beta, alpha)  # the estimated parameters


print("R-squared=", r_value**2)  # the coefficient of determination

print("p-value =", p_value)
```

```
# 4 days forecast
forecast_gold1 = np.exp(alpha + beta*gold3[(np.size(gold3)-1)])
forecast_gold2 = np.exp(alpha + beta*np.log(forecast_gold1))
forecast_gold3 = np.exp(alpha + beta*np.log(forecast_gold2))
forecast_gold4 = np.exp(alpha + beta*np.log(forecast_gold3))
# the 4 step forecast is obtained using the following relation
# beta*gold price from the current period + alpha forecast_gold
print('Forecast_gold 4 steps:', forecast_gold4)
# The forecasted gold price for 5 April 2018  1321.67
# The Gold Price from the World Gold Council data in GOLDAMGBD228NLBM.csv
# was for 5 April 2018 1327.05

##############################################################################
# Model 2: The ARIMA Time Series Model
##############################################################################

# Visualize Gold Price data as time series
#
dates = gold.index
#
gold
ar_gold = gold
ar_gold['Month'] = dates.month
ar_gold['Month'] = gold['Month'].apply(lambda x: calendar.month_abbr[x])
ar_gold['Year'] = dates.year

# extract out the time-series
gold_ts = ar_gold['Value']
#
plt.figure(figsize=(10, 5))
plt.plot(gold_ts)
plt.title("Gold price evolution from 1979 until 2018")
plt.xlabel('Years')
plt.ylabel('Daily Gold Prices')
#

# split into train and test series into two equal parts
train = ar_gold.loc[:'1998-08-16']
test = ar_gold.loc['1998-08-17':]

# extract out the time-series
train_ts = train['Value']
test_ts = test['Value']

# check size of datasets
np.size(train_ts)
np.size(test_ts)
print('Train:', train.tail(5))
print('Test:', test.head(5))


# Step 1.1 Trend - Time Series Decomposition
# Time series decomposition of this data is the first step in understanding the
# underlying patterns such as trend, seasonality, cycle and irregular
# remainder for Gold Prices.
# Determing rolling statistics
rolmean_up = train_ts.rolling(window=12).mean()
rolmean_down = test_ts.rolling(window=12).mean()

#Plot rolling statistics:
up = plt.plot(rolmean_up, label='Train Period')
down = plt.plot(rolmean_down, label='Test Period')
plt.legend(loc='best')
plt.title('Time Series Decomposition')
plt.show(block=False)



# Perform Dickey-Fuller test for the difference time periods:
print('Results of Dickey-Fuller Test for complete gold data:')
dftest = adfuller(gold_ts, autolag='AIC')
dfoutput = pd.Series(dftest[0:4], index=['Test Statistic', 'p-value',
                     '#lags Used', 'Number of Observations Used'])
for key, value in dftest[4].items():
    dfoutput['Critical Value (%s)'%key] = value
print(dfoutput)

# Perform Dickey-Fuller test for Train Data (prior to 1998-08-17):
print('Results of Dickey-Fuller Test for Train Data:')
dftest = adfuller(train_ts, autolag='AIC')
print(dfoutput)


# Perform Dickey-Fuller test for Test Data from 1998-08-17:
print('Results of Dickey-Fuller Test for Test Data:')
dftest = adfuller(test_ts, autolag='AIC')
dfoutput = pd.Series(dftest[0:4], index=['Test Statistic', 'p-value', '#lags Used', 'Number of Obse
for key, value in dftest[4].items():
    dfoutput['Critical Value (%s)'%key] = value

print(dfoutput)

# Comments
#
print('\n If the ADF p-values are higher or less negative compared to the \
      critical values of(1%)= -3.43,(5%) = -2.86 and (10%) = -2.57.\
      Then, there is strong evidence to suggest that we cannot reject the null\
      hypothesis (unit root).The results show that the Train period data is\
      stationary, while the Test Data is nonstationary')
#
# Step 1.2: Seasonality - Time Series Decomposition

# Yearly Seasonality

yearly_gold_data = pd.pivot_table(ar_gold, values = "Value", columns = "Month",
                              index = "Year")
yearly_gold_data = yearly_gold_data[['Jan','Feb','Mar', 'Apr', 'May', 'Jun',
                                 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']]
yearly_gold_data
print('Seasonality of the Test data (1998-2018):', yearly_gold_data.boxplot())

# Gold prices perform poorly between April to June with two peaks, one in
```

```python
# February and another in September


# Step 1.3: Irregular Remainder – Time Series Decomposition
""" To decipher underlying patterns in the gold price, we build a \
multiplicative time series decomposition model for the Test Data with the\
 following equation:\
Yt=Trendt × Seasonalityt × Remaindert \

Instead of multiplicative model we would choose additive model for the train\
data. In that case the equation would been:
Yt= Trendt + Seasonalityt + Remaindert\

However, it would have made very little difference in terms of conclusion\
we will draw from this time series decomposition exercise.
"""
#
# Time Series Decomposition
#
print('Decomposition of the Train data (1998-2018):')
decomposition = sm.tsa.seasonal_decompose(train_ts, model='additive')
fig = decomposition.plot()
fig.set_figwidth(12)
fig.set_figheight(8)
fig.suptitle('Decomposition of Additive time series for Train Data')
plt.show()

print('Decomposition of the Test data (1998-2018):')
decomposition = sm.tsa.seasonal_decompose(test_ts, model='multiplicative')
fig = decomposition.plot()
fig.set_figwidth(12)
fig.set_figheight(8)
fig.suptitle('Decomposition of multiplicative time series for Test Data')
plt.show()

# Comment
#Train data is close to zero, during this period Gold was a zero beta asset,
# while in the Test data period, the gold price has residuals white noise and
# a linear upward trend.
# Moreover, the Test Data tail is increasingly mimicking the head of the
# Train Data. Therefore, we can that the Test Data is tracking the Train data
# As such, we might be able to forecast the Test data with a Train data, MA(q)
# model or an ARIMA class to implement the model.

# Step 2: ARIMA Modelling
"""  ARIMA is a model that can be fitted to time series data in order to better
 understand or predict future points in the series. There are three distinct
 integers (p, d, q) that are used to parametrize ARIMA models. Together these
 three parameters account for seasonality, trend, and noise in datasets:
•  p is the auto-regressive part of the model.
•  d is the integrated part of the model.
•  q is the moving average part of the model.

"""
# Step 2.1: Difference log transform the Test data & Train Data to make data
# stationary on both mean and variance.The train data is already stationary
```

```
#Log-Difference of Complete Gold data
gold_ts_log = np.log10(gold_ts)
gold_ts_log.dropna(inplace=True)

# same as ts_log_diff = ts_log - ts_log.shift(periods=1)
gold_ts_log_diff = 1000 * gold_ts_log.diff(periods=1)
gold_ts_log_diff.dropna(inplace=True)
#

#Log-Difference of Train
train_ts_log = np.log10(train_ts)
train_ts_log.dropna(inplace=True)

# same as ts_log_diff = ts_log - ts_log.shift(periods=1)
train_ts_log_diff = 1000 * train_ts_log.diff(periods=1)
train_ts_log_diff.dropna(inplace=True)
#
# Log-difference of Test Data
test_ts_log = np.log10(test_ts)
test_ts_log.dropna(inplace=True)

# same as ts_log_diff = ts_log - ts_log.shift(periods=1)
test_ts_log_diff = 1000 * test_ts_log.diff(periods=1)
test_ts_log_diff.dropna(inplace=True)

# Step 2.2: Plot ACF and PACF to identify potential AR and MA model

# Train data plots
print('Identify potential AR and MA model of the Train data (1978-1998):')
fig, axes = plt.subplots(1, 2, sharey=False, sharex=False)
fig.set_figwidth(12)
fig.set_figheight(4)
sm.tsa.graphics.plot_acf(train_ts_log_diff, lags=25, ax=axes[0], alpha=0.5)
sm.tsa.graphics.plot_pacf(train_ts_log_diff, lags=25, ax=axes[1], alpha=0.5)
plt.tight_layout()
#

#
# Comment
#
# Since, there are enough spikes in the plots outside the insignificant zone
# (dotted horizontal lines) we can conclude that the residuals are not random.
# This implies that there is information available in residuals to be
# extracted by AR and MA models. Also, there is a seasonal component
# in the residuals at the lag 11 (represented by spikes at lag 11).
# This makes sense since we are analyzing daily data, the seasonality
# corresponds to the trading days in a month.

# Step 2.3: Identification of best fit ARMA model
""" When looking to fit time series data with an ARMA model,
the first goal is to find the values of ARIMA(p,d,q) that optimize
a metric of interest. A "grid search" function is used to iteratively explore
different combinations of parameters. Each combination of parameters is fit
to a fit a new seasonal ARIMA model with the SARIMAX() function from the
statsmodels module by assessing the AIC or BIC scores. The model with the
best score wins and the parmeters for that model are the optimal parmeters.
"""
```

```python
print('Auto.ARIMA:')
# Define the p, d and q parameters to take any value between 0 and 2
p = d = q = range(0, 2)

# Generate all different combinations of p, d and q triplets
pdq = list(itertools.product(p, d, q))

# Generate all different combinations of seasonal p, q and q triplets
seasonal_pdq = [(x[0], x[1], x[2], 11) for x in list(itertools.product(p, d,
                q))]

warnings.filterwarnings("ignore") # specify to ignore warning messages

best_aic = np.inf
best_pdq = None
best_seasonal_pdq = None
temp_model = None

for param in pdq:
    for param_seasonal in seasonal_pdq:

        try:
            temp_model = sm.tsa.statespace.SARIMAX(train_ts_log,
                                                 order = param,
                                                 seasonal_order = param_seasonal,
                                                 enforce_stationarity=True,
                                                 enforce_invertibility=True)
            results = temp_model.fit()

            # print("SARIMAX{}x{}12 - AIC:{}".format(param, param_seasonal, results.aic))
            if results.aic < best_aic:
                best_aic = results.aic
                best_pdq = param
                best_seasonal_pdq = param_seasonal
        except:
            #print("Unexpected error:", sys.exc_info()[0])
            continue
print("Best SARIMAX{}x{}12 model - AIC:{}".format(best_pdq, best_seasonal_pdq,
      best_aic))

# Comment
# Using tsa.ARMA got ARIMA(1,1,0)+c = regression of Y_DIFF1 on Y_DIFF1_LAG1
# (1st-order AR model applied to first difference of Y). The TSA was faster
# than the SARIMAX class but needed the seasonality component, thus SARIMAX
# returned the Best consideration:
#
# SARIMAX(0, 1, 0)x(1, 0, 0, 11)12 model - AIC:-40309.13881864493

#
# Step 3 – Predict Gold Price on in-sample date using the best fit ARIMA
# Use log of the complete data of Gold price from 1979 to 2018:
#
mod = sm.tsa.statespace.SARIMAX(gold_ts_log,
                                order=(0, 1, 0),
                                seasonal_order=(1, 0, 0, 11),
                                enforce_stationarity=False,
                                enforce_invertibility=False)
```

```
results = mod.fit()


print('The model fit results:', results.summary())

# Plot diagnostics to investigate for any unusual behavior:
results.plot_diagnostics(figsize=(15, 12))
plt.show()

# In the top right plot, we see that the red KDE line follows closely with
# the N(0,1) line (where N(0,1)) is the standard notation for a normal
# distribution with mean 0 and standard deviation of 1). This is a good
# indication that the residuals are normally distributed.
#
#The residuals over time (top left plot) don't display any obvious seasonality
# and appear to be white noise. This is confirmed by the autocorrelation
#(i.e. correlogram) plot on the bottom right, which shows that the time series
# residuals have low correlation with lagged versions of itself.
#
# Those observations lead us to conclude that our model produces a
# satisfactory fit that could help us understand our time series data and
# forecast future values.

# Step 3.1: Validating Forecasts: Static model
# The get_prediction() and conf_int() attributes allow us to obtain the values
# and associated confidence intervals for forecasts of the time series.
#
# Predict the values from the Test Period using a static model
pred_test = results.get_prediction(start=pd.to_datetime('1982-01-01'),
                              dynamic=False)
#
# The dynamic=False argument ensures that we produce one-step ahead forecasts,
# meaning that forecasts at each point are generated using the full history up
# to that point.

# Confidence interval
pred_ci = pred_test.conf_int()
#

# Plot the One step ahead forecast prediction
ax = gold_ts_log.plot(label='observed')

pred_test.predicted_mean.plot(ax=ax, label='One-step ahead Forecast', alpha=.7)

ax.fill_between(pred_ci.index,
                pred_ci.iloc[:, 0],
                pred_ci.iloc[:, 1], color='k', alpha=.2)

ax.set_xlabel('Date')
ax.set_ylabel('Gold Prices')
plt.legend()

plt.show()

# Overall, the forecasts align with the true values very well, showing an
# overall increase trend. It is also useful to quantify the accuracy of our
# forecasts through the MSE (Mean Squared Error), which summarizes the
# average error of the forecasts.
# For each predicted value, the distance to the true value is computed and
# square the result, so that positive/negative differences,
# do not cancel each other out when we compute the overall mean.

y_forecasted = pred_test.predicted_mean # forecast from 1982-01-01
y_truth = test_ts_log['1982-01-01':]  # data from 1982-01-01

# Compute the mean square error
mse = ((y_forecasted - y_truth) ** 2).mean()
# 1000 was used to make the small errors more visible, so we multiply by 0.001
# to get the actual value:
print('The Mean Squared Error of our forecasts is {}'.format(round(mse, 2)))
# MSE = 0.01


# Forecast 11 steps in future using dynamic model
print ('x takes the value of the forecasted gold price for any day between\
       1 and 11')
# Dynamic model for a forecast of 10 days results from the SARIMAX model
# steps
n_steps = 11

# alpha=0.01 signifies 99% confidence interval
pred_uc_99 = results.get_forecast(steps=11, alpha=0.01)

# Forecast 11 steps in the future
forecasted_11_steps = pred_uc_99.predicted_mean

# Data frame converted to values
forecasted_log = forecasted_11_steps.values

# forecast gold price at step i = np.exp(x[i])

def forecasted_price_after_steps(x):
    x = forecasted_log[x]
    return 10**x

print(forecasted_price_after_steps(4))
# The gold price for the fourth step is the price for 2018-04-05 1323.68
# The World Gold Council price for the same day was  1327.05
# The Linear regression forecast was 1321.67.
# The results show that the ARIMA model tracked the daily price better than
# the OLS regression forecast.

###############################################################################
```