Universidad Zaragoza

Facultad de Ciencias Universidad Zaragoza

APPENDIX of
Predictive methods in time series and machine learning of macroeconomic data

Author:

Raúl Almuzara Diarte

Supervisors:

Juan Luis Fernández Martínez

Lucas Fernández Brillet

Contents

A Comments on deep learning models

The project has focused on the study and implementation of the Holt-Winters, Prophet and SARIMA models. These models fit within the definition of machine learning models, but in a way, they also use techniques that could be considered as more typical of classical statistics. The advantage of these models is that they are specifically designed for time series processing. Therefore, they can take good advantage of the intrinsic structure of the series to model the data and make well-founded and, to some extent, explainable predictions. However, it is well known that there are other more complex techniques within the subset of machine learning that we call deep learning. In particular, techniques based on neural networks stand out widely for their applications. These are computationally intensive models that have proven to be very useful in many areas such as image recognition or natural language processing. Of course, they have also been successfully applied to time series modeling. However, it is very important to keep in mind that a more complex model will not necessarily give better results than a classical model in all types of problems. Neural network models are a very powerful tool for approximating complex functions. However, it should be noted that these architectures are also used in other fields of deep learning and are not specifically designed for the decomposition and extraction of time series signals, unlike previous statistical methods that are focused on time series processing. In many cases, we may find that classical models have properties that are beneficial when working with data with a specific structure as opposed to some modern models that lack explainability and could *miss out on a lot of structure that would help in forecasting.*[1]
Even so, the following is a summary of one of the most popular models that has been explored during the course of this work and which has been of crucial importance in the last decade due to its applications.

A.1 LSTM networks

A Long Short-Term Memory (LSTM) model[2] is a type of artificial neural network from the family of Recurrent Neural Networks (RNN), which are designed to deal with sequential data. For this reason, there has been a great interest in its use in natural language processing and time series problems in recent years. LSTMs networks were formally introduced in 1997[3], years after some problems with classic RNNs, which will be described below, were analyzed.
A recurrent neural network is not like a standard feedforward neural network. The main difference is that connections in an RNN can form cycles. The loops in RNNs connect

[1] Kolassa, S. (2015). Answer in *Why time series analysis is not considered a machine learning algorithm.* Version of 2017-04-13.
`https://stats.stackexchange.com/q/160417` (accessed in August 2023)

[2] Olah, C. (2015). *Understanding LSTM Networks.*
`http://colah.github.io/posts/2015-08-Understanding-LSTMs/` (accessed in August 2023)

[3] Hochreiter, S., & Schmidhuber, J. (1997). *Long short-term memory.* Neural computation, 9(8), 1735-1780.

the output responses to the input layer and the output responses act as additional input variables. Unlike classic neural networks, the RNNs can deal with sequences of inputs of arbitrary length and they have a notion of the directionality of the inputs, which is fundamental in a time series problem where the order of the data is very relevant.

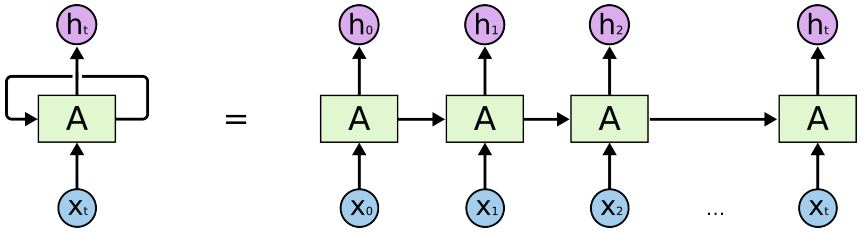


Figure 1: Rolled and unrolled representation of an RNN which takes an input x_t, produces an output h_t and the information is passed through a hidden layer in different steps.

The vanilla RNN serves as the basis for a whole class of more complex recurrent network models with several variations that have been introduced over the years. However, there is a major problem that motivates the development of some of these variants. As in many neural networks, the parameters of the network are learned by gradient-based techniques. RNN models perform backpropagation through time to update weights. Unfortunately, when computing a gradient of an output with respect to an input both with very far apart time indices, the gradient may virtually vanish due to the repeated multiplication of elements with norm less than 1. This is known as the *vanishing gradient problem* (analogously, the *exploding gradient problem* deals with the repeated multiplication of elements with norm greater than 1 that causes gradients to explode, effectively hampering the learning process). In other words, RNNs may have difficulty retaining crucial information over time and they struggle with longer sequences of data when trying to capture long-term dependencies. LSTM models aim to solve this problem and they are better suited to remember information for longer periods of time. Unlike standard RNNs, which usually have one simple layer in each repeating module, LSTM networks have four connected layers.

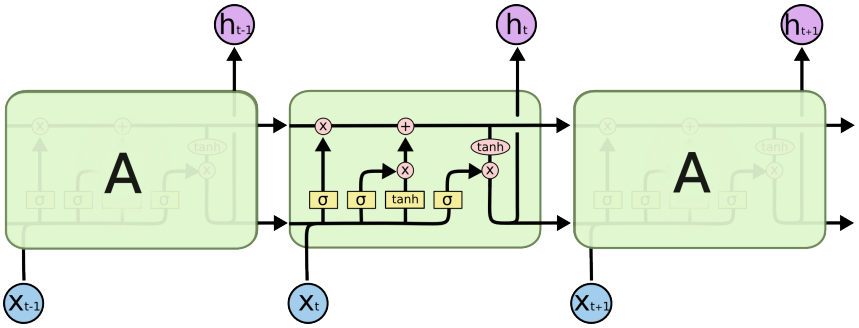


Figure 2: Representation of an LSTM network. The central module schematizes the transformations that a cell state C_{t-1}, an input x_t and a previous output h_{t-1} undergo. σ is the sigmoid activation function and tanh is the hyperbolic tangent activation function. The operations are applied in a pointwise sense.

An important feature of the LSTM network is the presence of a vector called the *cell state*. Its purpose is to remember long-term information if it is useful in the future.

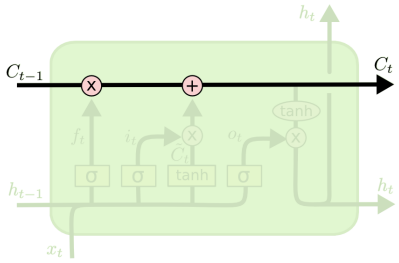
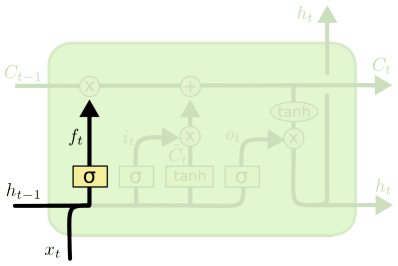


Figure 3: The cell state C_{t-1} flows along the horizontal line at the top and some operations will be applied before leaving for the next module as C_t.

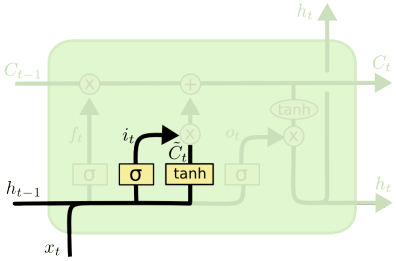
This cell state is more resistant to gradient problems and will be modified according to the amount of information we want to add or remove through the *gates*.
First, we consider a *forget gate* which helps eliminate unimportant information of the previous cell state C_t that we do not want to continue propagating. We perform a linear combination of h_{t-1} and x_t. After applying a sigmoid activation function, we obtain the forget vector f_t. The concatenation of h_{t-1} and x_t will be denoted as $[h_{t-1}, x_t]$ to simplify the notation.



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Figure 4: h_{t-1} and x_t are combined linearly with a matrix of weights W_f and a bias vector b_f. Then, a nonlinear sigmoid is applied to build f_t.

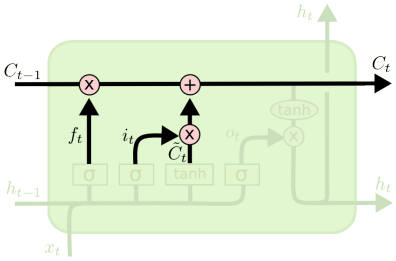
Analogously, we have an *input gate* and we perform a similar linear combination of h_{t-1} and x_t and apply the sigmoid function to obtain i_t. The purpose of this vector will be to decide how to keep the information of a new candidate cell state $\tilde{C}_t$. This new vector is also built as a linear combination of h_{t-1} and x_t with associated weights and biases, although the nonlinear activation function will be the hyperbolic tangent.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Figure 5: For i_t, we combine h_{t-1} and x_t linearly with a matrix of weights W_i and a bias vector b_i. Then, a sigmoid is applied. For $\tilde{C}_t$, we combine h_{t-1} and x_t linearly with a matrix of weights W_C and a bias vector b_C. Then, a tanh is applied.

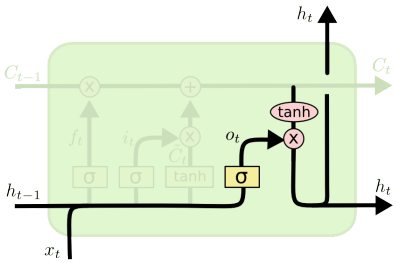
The new forget and input gate vectors and the candidate cell state vector are poured into the cell state and combined via pointwise operations to build the new cell state vector C_t.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Figure 6: From h_{t-1} and x_t, we built f_t, i_t and $\tilde{C}_t$ and now we update the cell state vector C_{t-1} to C_t. The sum $+$ and the multiplication $*$ act pointwisely.

Finally, we have an *output gate* built with an analogous combination of weights and biases from h_{t-1} and x_t and a nonlinear sigmoid to obtain o_t. The output h_t for the next module is dependent on o_t and C_t.



$$o_t = \sigma\left(W_o \; [\, h_{t-1}, x_t] \; + \; b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Figure 7: h_{t-1} and x_t are combined linearly with a matrix of weights W_o and a bias vector b_o. Then, a nonlinear sigmoid is applied to build o_t. The final h_t is a pointwise multiplication of o_t and $\tanh(C_t)$.

The Holt-Winters and Prophet models follow certain smoothing approaches, so the price series can be passed directly to them and the smoothing techniques will take care of learning the necessary information about the components of the series. The ARIMA models are also prepared to work with non-stationary series as long as the application of

as many differences as indicated by the I part converts them into stationary series to be processed by the ARMA part. However, neural networks work best when the input data are properly normalized over a bounded range and on a common scale. We must also divide the data into windows respecting the order of the data in the time series. That is, divide the data into overlapping (sometimes, non-overlapping to save time) windows and assign them the next value of the time series as a target.
The implementation can be done with the `Keras` model builder. With relatively similar preprocessing, we can build very different networks by directly specifying the layers. Among some of the most popular options, we find the standard LSTM layer, but also others such as the Bidirectional layer or the TimeDistributed layer.

B Code snippets

This project has been carried out in Python. In the next pages, we show some of the fundamental lines.

Relevant libraries:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import os
import datetime as dt
import warnings
import csv
from sklearn.metrics import confusion_matrix

import yfinance as yf

from statsmodels.tsa.exponential_smoothing.ets import ETSModel
from prophet import Prophet
from pmdarima.arima import auto_arima
```

Loading data:

```
def fetch_financial_data(initial_train, end_train, initial_test, end_test, tickers, granularity, features):

    full_dataset = yf.download(tickers,
                               start=initial_train,
                               end=end_test,
                               interval=granularity,
                               progress=False)[features]

    train_dataset = yf.download(tickers,
                               start=initial_train,
                               end=end_train,
                               interval=granularity,
                               progress=False)[features]

    test_dataset = yf.download(tickers,
                               start=initial_test,
                               end=end_test,
                               interval=granularity,
                               progress=False)[features]

    return full_dataset, train_dataset, test_dataset

initial_train = dt.date(2015, 1, 1)
end_train = dt.date(2022, 1, 1)
initial_test = dt.date(2022, 1, 1)
end_test = dt.date(2023, 1, 1)
#end_test = dt.datetime.today()
tickers = ['AAPL','SPY']

daily_data = fetch_financial_data(initial_train,
                                  end_train,
                                  initial_test,
                                  end_test,
                                  tickers,
                                  '1d',
                                  ['Close','Open','Volume'])

data_full_D = daily_data[0]
data_train_D = daily_data[1]
data_test_D = daily_data[2]
```

Holt-Winters rolling forecasts:

```
def rolling_holt_winters(time_window_train, seasonal_periods, trend_type, seasonal_type, ahead, data,
     split_date):

    data_train_D = data.iloc[data.index < split_date]
    data_test_D = data.iloc[data.index >= split_date]

    n_prediction_rounds = int(len(data_test_D)/ahead)

    df_forecasts = pd.DataFrame()
    df_residuals = pd.DataFrame()
    parameters = []

    for i in range(n_prediction_rounds):

        if ahead*i < time_window_train:
            train = pd.concat([data_train_D.iloc[-(time_window_train-ahead*i):],
                               data_test_D.iloc[:(ahead*i)]])
        else:
            train = data_test_D.iloc[-(time_window_train-ahead*i):(ahead*i)]

        exp_smooth_model = ETSModel(endog=train,
                                    trend=trend_type,
                                    seasonal=seasonal_type,
                                    damped_trend=True,
                                    seasonal_periods=seasonal_periods)

        fitted_exp_model = exp_smooth_model.fit()

        df_residuals = pd.concat([df_residuals,pd.DataFrame(fitted_exp_model.resid)], axis=1, ignore_index=
     False)

        parameters.append(fitted_exp_model.params)

        forecasts = fitted_exp_model.forecast(ahead)

        ci = fitted_exp_model.get_prediction(start=forecasts.index[0], end=forecasts.index[-1])

        lower_ci = ci.pred_int(alpha=0.05).iloc[:,0]
        upper_ci = ci.pred_int(alpha=0.05).iloc[:,1]

        new_data = pd.DataFrame({'Date':data_test_D.iloc[ahead*i:(ahead*(i+1))].index,
                                 'Forecast':forecasts,
                                 'Lower bound':lower_ci,
                                 'Upper bound':upper_ci}).set_index('Date')

        df_forecasts = pd.concat([df_forecasts,new_data], ignore_index=False)

    return df_forecasts, df_residuals, parameters
```

Prophet rolling forecasts:

```
def rolling_prophet(time_window_train, cps, sps, sm, ahead, data, split_date):

    data_train_D = data.iloc[data.index < split_date]
    data_test_D = data.iloc[data.index >= split_date]

    n_prediction_rounds = int(len(data_test_D)/ahead)

    df_forecasts = pd.DataFrame()
    df_model_data = pd.DataFrame()

    df_residuals = pd.DataFrame()

    for i in range(n_prediction_rounds):

        if ahead*i < time_window_train:
            train = pd.concat([pd.DataFrame({'ds':data_train_D.iloc[-(time_window_train-ahead*i):].index,
                                             'y':data_train_D.iloc[-(time_window_train-ahead*i):].values.
     reshape(-1)}),
                               pd.DataFrame({'ds':data_test_D.iloc[:(ahead*i)].index,
                                             'y':data_test_D.iloc[:(ahead*i)].values.reshape(-1)})])
        else:
            train = pd.DataFrame({'ds':data_test_D.iloc[-(time_window_train-ahead*i):(ahead*i)].index,
                                  'y':data_test_D.iloc[-(time_window_train-ahead*i):(ahead*i)].values.reshape
     (-1)})

        prophet_model = Prophet(weekly_seasonality=True,
                                yearly_seasonality=False,
                                changepoint_prior_scale=cps,
                                seasonality_prior_scale=sps,
                                seasonality_mode=sm,
                                interval_width=0.95)
        prophet_model.add_seasonality(name='monthly', period=21, fourier_order=5)
        prophet_model.fit(train)

        backward_prediction = prophet_model.predict()
        df = pd.merge(train, backward_prediction, on='ds')
        residuals_data = df.set_index('ds')['y'] - df.set_index('ds')['yhat']
        df_residuals = pd.concat([df_residuals,residuals_data], axis=1, ignore_index=False)

        forecasting_periods = pd.DataFrame({'ds': data_test_D.iloc[ahead*i:(ahead*(i+1))].index})

        forecast_prophet = prophet_model.predict(forecasting_periods)

        df_model_data = pd.concat([df_model_data, forecast_prophet], ignore_index = False)

    df_model_data = df_model_data.set_index('ds')

    df_forecasts = df_model_data[['yhat','yhat_lower','yhat_upper']]
    df_model_components = df_model_data.drop(['yhat','yhat_lower','yhat_upper'], axis=1)

    return df_forecasts, df_model_components, df_residuals
```

SARIMA rolling forecasts without covariates:

```
def rolling_sarima(time_window_train, seasonal_periods, ahead, data, split_date):

    data_train_D = data.iloc[data.index < split_date]
    data_test_D = data.iloc[data.index >= split_date]

    n_prediction_rounds = int(len(data_test_D)/ahead)

    df_forecasts = pd.DataFrame()
    df_residuals = pd.DataFrame()

    orders = []
    seasonal_orders = []
    parameters = []

    for i in range(n_prediction_rounds):

        if ahead*i < time_window_train:
            train = pd.concat([data_train_D.iloc[-(time_window_train-ahead*i):],
                               data_test_D.iloc[:(ahead*i)]])
        else:
            train = data_test_D.iloc[-(time_window_train-ahead*i):(ahead*i)]

        arima_model = auto_arima(train,
                                 start_p=0, start_q=0,
                                 test='adf',
                                 max_p=5, max_q=5, d=1, D=1,
                                 m=seasonal_periods,
                                 seasonal=True,
                                 trace=False,
                                 error_action='ignore',
                                 suppress_warnings=True,
                                 stepwise=True)

        model_data = arima_model.to_dict()
        df_residuals = pd.concat([df_residuals,pd.DataFrame(model_data['resid'])], axis=1, ignore_index=False)

        orders.append(model_data['order'])
        seasonal_orders.append(model_data['seasonal_order'])
        parameters.append(model_data['params'])

        forecast = arima_model.predict(ahead, return_conf_int=True, alpha=0.05)
        forecasts_auto_arima = pd.DataFrame({'Date':data_test_D.iloc[ahead*i:(ahead*(i+1))].index,
                                             'Forecast':forecast[0],
                                             'Lower bound':[a[0] for a in forecast[1]],
                                             'Upper bound':[a[1] for a in forecast[1]]}).set_index('Date')

        df_forecasts = pd.concat([df_forecasts, forecasts_auto_arima], ignore_index = False)

    return df_forecasts, df_residuals, orders, seasonal_orders, parameters
```

SARIMA rolling forecasts with covariates (version 1):

```
def rolling_sarima_covariates_1(time_window_train, seasonal_periods, ahead, data, covariates, split_date):

    data_train_D = data.iloc[data.index < split_date]
    data_test_D = data.iloc[data.index >= split_date]

    covariates_train_D = covariates.iloc[covariates.index < split_date]
    covariates_test_D = covariates.iloc[covariates.index >= split_date]

    n_prediction_rounds = int(len(data_test_D)/ahead)

    df_forecasts = pd.DataFrame()
    df_residuals = pd.DataFrame()

    orders = []
    seasonal_orders = []
    parameters = []

    for i in range(n_prediction_rounds):

        if ahead*i < time_window_train:
            train = pd.concat([data_train_D.iloc[-(time_window_train-ahead*i):],
                               data_test_D.iloc[:(ahead*i)]])
            exogenous = pd.concat([pd.concat([pd.DataFrame(covariates_train_D['Open AAPL']),
                                              pd.DataFrame(covariates_train_D['Open SPY']),
                                              pd.DataFrame(covariates_train_D['Volume(mil) AAPL'])],axis=1).
     iloc[-(time_window_train-ahead*i):],
                                   pd.concat([pd.DataFrame(covariates_test_D['Open AAPL']),
                                              pd.DataFrame(covariates_test_D['Open SPY']),
                                              pd.DataFrame(covariates_test_D['Volume(mil) AAPL'])],axis=1).iloc
     [:(ahead*i)]])

        else:
            train = data_test_D.iloc[-(time_window_train-ahead*i):(ahead*i)]
            exogenous = pd.concat([
                pd.concat([pd.DataFrame(covariates_test_D['Open AAPL']),
                           pd.DataFrame(covariates_test_D['Open SPY']),
                           pd.DataFrame(covariates_test_D['Volume(mil) AAPL'])],axis=1).iloc[-(
     time_window_train-ahead*i):(ahead*i)]])

        arima_model = auto_arima(train,
                                 X=exogenous,
                                 start_p=0, start_q=0,
                                 test='adf',
                                 max_p=5, max_q=5, d=1, D=1,
                                 m=seasonal_periods,
                                 seasonal=True,
                                 trace=False,
                                 error_action='ignore',
                                 suppress_warnings=True,
                                 stepwise=True)

        model_data = arima_model.to_dict()
        df_residuals = pd.concat([df_residuals,pd.DataFrame(model_data['resid'])], axis=1, ignore_index=False)

        orders.append(model_data['order'])
        seasonal_orders.append(model_data['seasonal_order'])
        parameters.append(model_data['params'])

        if i==0:
            contemporary_covariates = pd.concat([pd.DataFrame(covariates_test_D['Open AAPL']).iloc[ahead*i:(
     ahead*(i+1))],
                                                 pd.DataFrame(covariates_test_D['Open SPY']).iloc[ahead*i:(
     ahead*(i+1))],
                                                 pd.DataFrame(np.repeat(covariates_train_D['Volume(mil) AAPL'].
     loc[train.index[-1]],ahead)).set_index(covariates_test_D.index[ahead*i:(ahead*(i+1))])],axis=1)

        else:
            contemporary_covariates = pd.concat([pd.DataFrame(covariates_test_D['Open AAPL']).iloc[ahead*i:(
     ahead*(i+1))],
                                                 pd.DataFrame(covariates_test_D['Open SPY']).iloc[ahead*i:(
     ahead*(i+1))],
                                                 pd.DataFrame(np.repeat(covariates_test_D['Volume(mil) AAPL'].
     loc[train.index[-1]],ahead)).set_index(covariates_test_D.index[ahead*i:(ahead*(i+1))])],axis=1)

        forecast = arima_model.predict(ahead,
                                       return_conf_int=True,
```

```
                                    alpha=0.05,
                                    X=contemporary_covariates)

        forecasts_auto_arima = pd.DataFrame({'Date':data_test_D.iloc[ahead*i:(ahead*(i+1))].index,
                                             'Forecast':forecast[0],
                                             'Lower bound':[a[0] for a in forecast[1]],
                                             'Upper bound':[a[1] for a in forecast[1]]}).set_index('Date')

        df_forecasts = pd.concat([df_forecasts, forecasts_auto_arima], ignore_index = False)

    return df_forecasts, df_residuals, orders, seasonal_orders, parameters
```

SARIMA rolling forecasts with covariates (version 2):

```
def rolling_sarima_covariates_2(time_window_train, seasonal_periods, ahead, data, covariates, split_date):

    data_train_D = data.iloc[data.index < split_date]
    data_test_D = data.iloc[data.index >= split_date]

    covariates_train_D = covariates.iloc[covariates.index < split_date]
    covariates_test_D = covariates.iloc[covariates.index >= split_date]

    n_prediction_rounds = int(len(data_test_D)/ahead)

    df_forecasts = pd.DataFrame()
    df_residuals = pd.DataFrame()

    orders = []
    seasonal_orders = []
    parameters = []

    for i in range(n_prediction_rounds):

        if ahead*i < time_window_train:
            train = pd.concat([data_train_D.iloc[-(time_window_train-ahead*i):],
                               data_test_D.iloc[:(ahead*i)]])
            exogenous = pd.concat([pd.concat([pd.DataFrame(covariates_train_D['Open AAPL']),
                                              pd.DataFrame(covariates_train_D['Open SPY']),
                                              pd.DataFrame(covariates_train_D['Volume(mil) AAPL'])],axis=1).
     iloc[-(time_window_train-ahead*i):],
                                   pd.concat([pd.DataFrame(covariates_test_D['Open AAPL']),
                                              pd.DataFrame(covariates_test_D['Open SPY']),
                                              pd.DataFrame(covariates_test_D['Volume(mil) AAPL'])],axis=1).iloc
     [:(ahead*i)]])

        else:
            train = data_test_D.iloc[-(time_window_train-ahead*i):(ahead*i)]
            exogenous = pd.concat([
                pd.concat([pd.DataFrame(covariates_test_D['Open AAPL']),
                           pd.DataFrame(covariates_test_D['Open SPY']),
                           pd.DataFrame(covariates_test_D['Volume(mil) AAPL'])],axis=1).iloc[-(
     time_window_train-ahead*i):(ahead*i)]])

        arima_model = auto_arima(train,
                                 X=exogenous,
                                 start_p=0, start_q=0,
                                 test='adf',
                                 max_p=5, max_q=5, d=1, D=1,
                                 m=seasonal_periods,
                                 seasonal=True,
                                 trace=False,
                                 error_action='ignore',
                                 suppress_warnings=True,
                                 stepwise=True)

        model_data = arima_model.to_dict()
        df_residuals = pd.concat([df_residuals,pd.DataFrame(model_data['resid'])], axis=1, ignore_index=False)

        orders.append(model_data['order'])
        seasonal_orders.append(model_data['seasonal_order'])
        parameters.append(model_data['params'])

        if i==0:
            contemporary_covariates = pd.concat([pd.DataFrame(np.repeat(covariates_train_D['Close AAPL'].loc[
     train.index[-1]],ahead)).set_index(covariates_test_D.index[ahead*i:(ahead*(i+1))]),
                                                 pd.DataFrame(np.repeat(covariates_train_D['Close SPY'].loc[
     train.index[-1]],ahead)).set_index(covariates_test_D.index[ahead*i:(ahead*(i+1))]),
                                                 pd.DataFrame(np.repeat(covariates_train_D['Volume(mil) AAPL'].
     loc[train.index[-1]],ahead)).set_index(covariates_test_D.index[ahead*i:(ahead*(i+1))])],axis=1)

        else:
            contemporary_covariates = pd.concat([pd.DataFrame(np.repeat(covariates_test_D['Close AAPL'].loc[
     train.index[-1]],ahead)).set_index(covariates_test_D.index[ahead*i:(ahead*(i+1))]),
                                                 pd.DataFrame(np.repeat(covariates_test_D['Close SPY'].loc[
     train.index[-1]],ahead)).set_index(covariates_test_D.index[ahead*i:(ahead*(i+1))]),
                                                 pd.DataFrame(np.repeat(covariates_test_D['Volume(mil) AAPL'].
     loc[train.index[-1]],ahead)).set_index(covariates_test_D.index[ahead*i:(ahead*(i+1))])],axis=1)

        forecast = arima_model.predict(ahead,
                                       return_conf_int=True,
```

```
                                    alpha=0.05,
                                    X=contemporary_covariates)

        forecasts_auto_arima = pd.DataFrame({'Date':data_test_D.iloc[ahead*i:(ahead*(i+1))].index,
                                             'Forecast':forecast[0],
                                             'Lower bound':[a[0] for a in forecast[1]],
                                             'Upper bound':[a[1] for a in forecast[1]]}).set_index('Date')

        df_forecasts = pd.concat([df_forecasts, forecasts_auto_arima], ignore_index = False)

    return df_forecasts, df_residuals, orders, seasonal_orders, parameters
```