TIME SERIES FORECASTING

# Forecasting Time Series Data - Stock Price Analysis

Focused on forecasting the Time-series data using different smoothing methods and ARIMA in Python.


Prathamesh Thakar · Follow

Published in Towards Data Science · 8 min read · Jul 21, 2020


Photo by Io Io on Unsplash

In this article, we will be going through the stock prices of a certain company. **However, this article does not encourage anyone to trade ONLY based on this forecast.** Stock prices are dependent on various factors like supply and demand, company performance, the sentiment of the investors, etc.

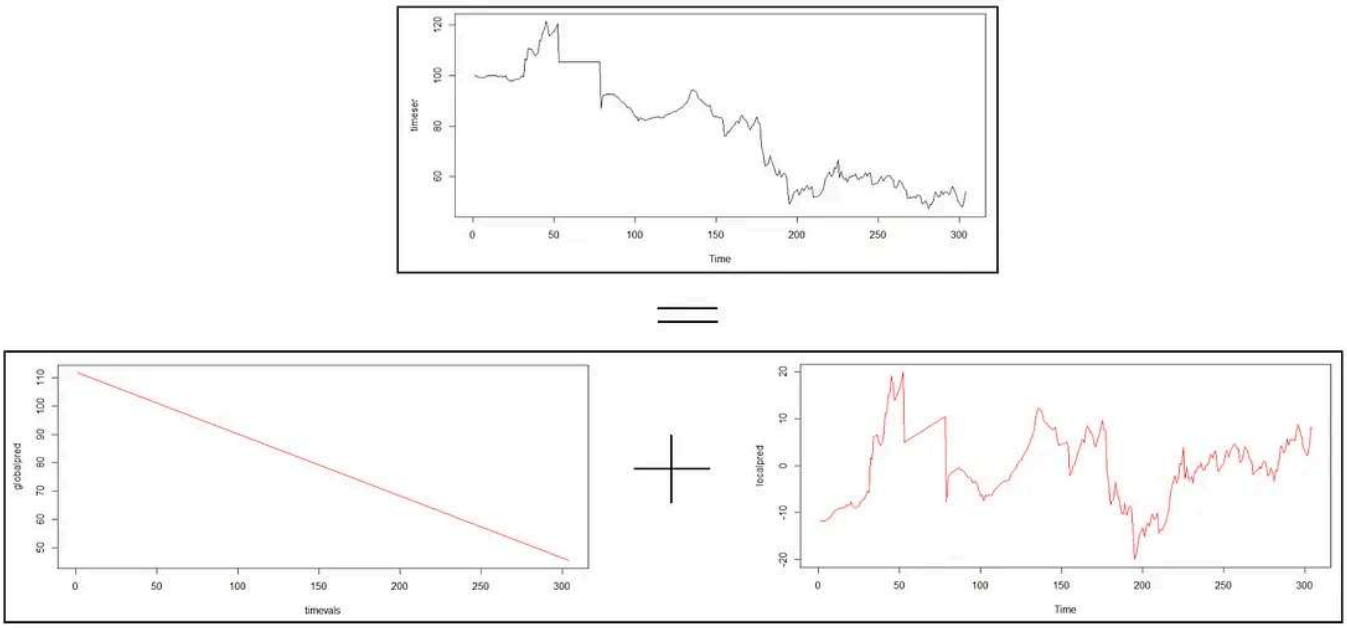
### What is Time-Series?

Time Series comprises of observations that are captured at regular intervals. Time Series datasets have a strong temporal dependence. It can be used to forecast future observations based on previous ones.

### Decomposing the Time Series:

The Time-Series can be divided into several parts as follows:

*Trend:* The increase or decrease in the value of the data. This can further be divided into the global trend and local trend.




Actual Observations = Global Trend + Local Trend (Source: UpGrad)

*Seasonality:* It is the repetitive pattern that is visible in the series. This rise or fall in the values of the data is of fixed frequency. For example, the sales of Christmas trees are always greater during December and less for the rest.

*Cyclicity:* It is the rise and fall in the value of the data which is not of a fixed frequency. It can be seen as an outcome of economic conditions or other external factors

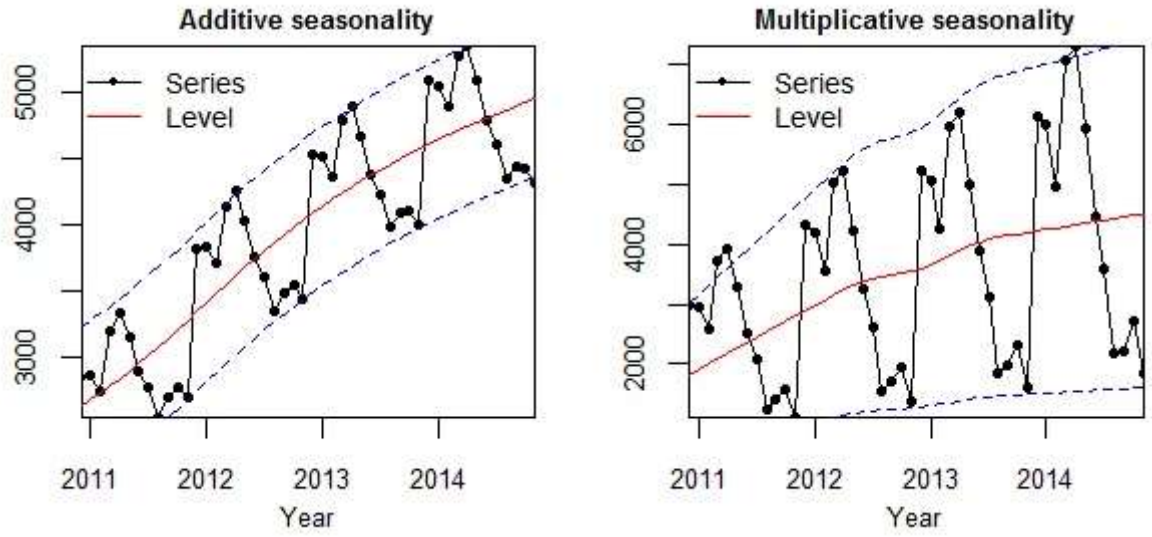
*Noise:* It is just some random data that we obtain after extracting the trend and the seasonal components from the time series.

The components of the time series can be an additive or multiplicative version. The multiplicative model is preferred when the magnitude of the seasonal pattern increases or decreases with the increase or decrease in the

data values. The additive model is preferred when the magnitude of the seasonal pattern does not correlate with the data values.

$$\text{Additive Model} - TS = T_t + S_t + X_t + Z_t$$

$$\text{Multiplicative Model} - TS = T_t * S_t * X_t * Z_t$$




Additive and Multiplicative Seasonality — Source: kourentzes

In the additive version, the magnitude of the seasonal pattern remains to be uncorrelated with the rising trend whereas it is increasing with the increase in the trend of the time series in the multiplicative version.

## How to obtain stock price data?

We can obtain stock prices by creating an API request using the requests package. We can also use the quandl package which is designed to provide financial and economic data.

We will extract the prices of US stocks from alphavantage website by creating an **API call.** Using the code snippet below, we have extracted the daily stock

prices of IBM. You can refer to this link to alter your request needs.

```
# Creating an API request to get the data for a stock.
req = requests.get('https://www.alphavantage.co/query?function=TIME_SERIES_DAILY_ADJUSTED&symbol=

# Transforming the content of the request to a dataframe.
request_content = req.content
data = list(map(lambda x: x.split(','),request_content.decode('utf-8').split("\r\n")))
dataframe = pd.DataFrame(data[1:], columns=data[0])
dataframe = dataframe.set_index('timestamp')
dataframe.index = pd.to_datetime(dataframe.index)
dataframe.head()
```

api_request.py hosted with ❤ by GitHub view raw

For using quandl package to import data into python, you might need to install quandl package. Enter the command **!pip install quandl** in your Jupyter notebook and you're good to go. We have imported the prices of Infosys (BOM500209) and will use these for our further analysis. More documentation on quandl and how to get the best out of it can be found here.

```
# Using quandl package to obtain data regarding a stock.
Stock_data = quandl.get("BSE/BOM500209",api_key="<your api key>")
```

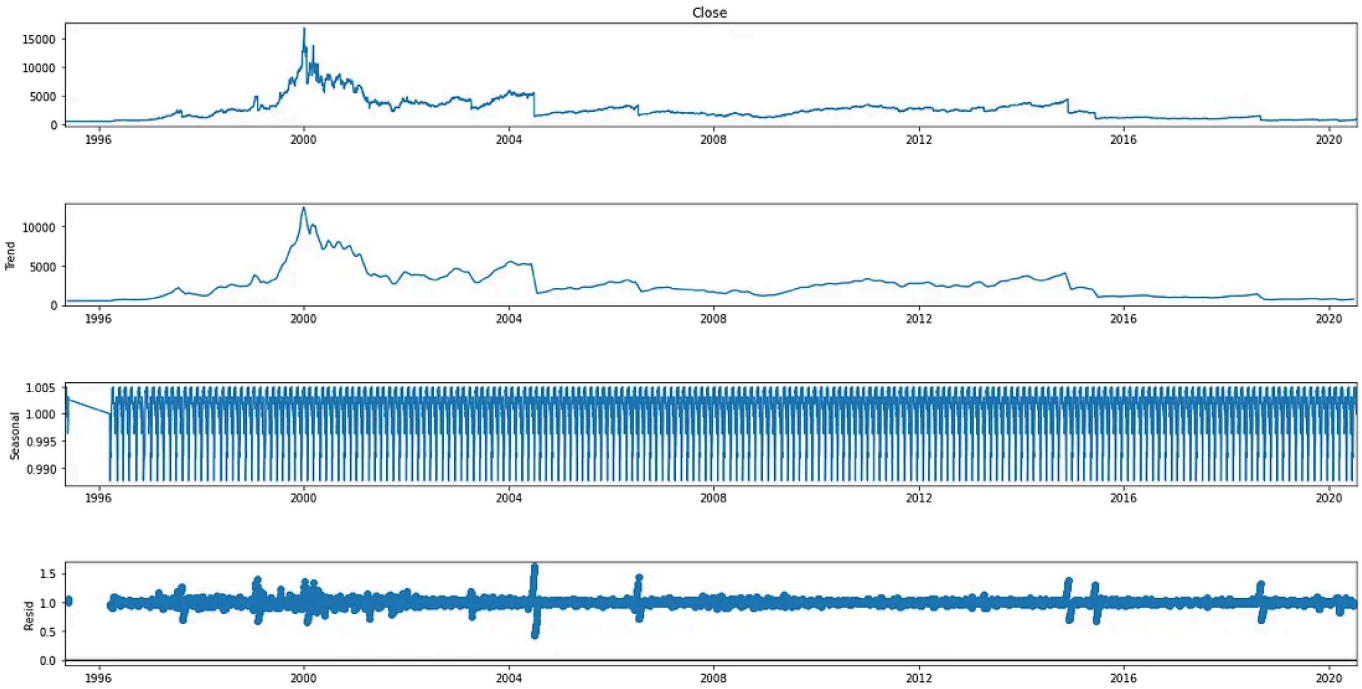
quandl_package.py hosted with ❤ by GitHub view raw

For both the methods, you will need to create an API key which you will use to obtain the data. You can easily get your API by clicking these links: alphavantage and quandl.

Now that we have got the data, we need to plot it to get an overview of how the trend looks like.

```
# Checking decomposition of trend, seasonality and residue of the original time seires.
decomposition = seasonal_decompose(Stock_data['Close'], model='multiplicative', period=30)
fig = plt.figure()
fig = decomposition.plot()
fig.set_size_inches(20, 10)
```

stock_overview.py hosted with ❤ by GitHub view raw




The trend experiences some ups and downs as a stock generally does. It is not seasonal as the seasonal component does not give any clearer picture. The residuals' variance seems to remain the same except for a few observations.

## Checking for Stationarity

For ARIMA, time series has to be made **stationary** for further analysis. For a time series to be stationary, its statistical properties(mean, variance, etc) will be the same throughout the series, irrespective of the time at which you observe them. A stationary time series will have no long-term predictable

patterns such as trends or seasonality. Time plots will show the series to roughly have a horizontal trend with the constant variance.

### Rolling Statistics

We can plot the rolling mean and standard deviation to check if the statistics show an upward or downward trend. If these statistics vary over time, then the time series is highly likely to be non-stationary.

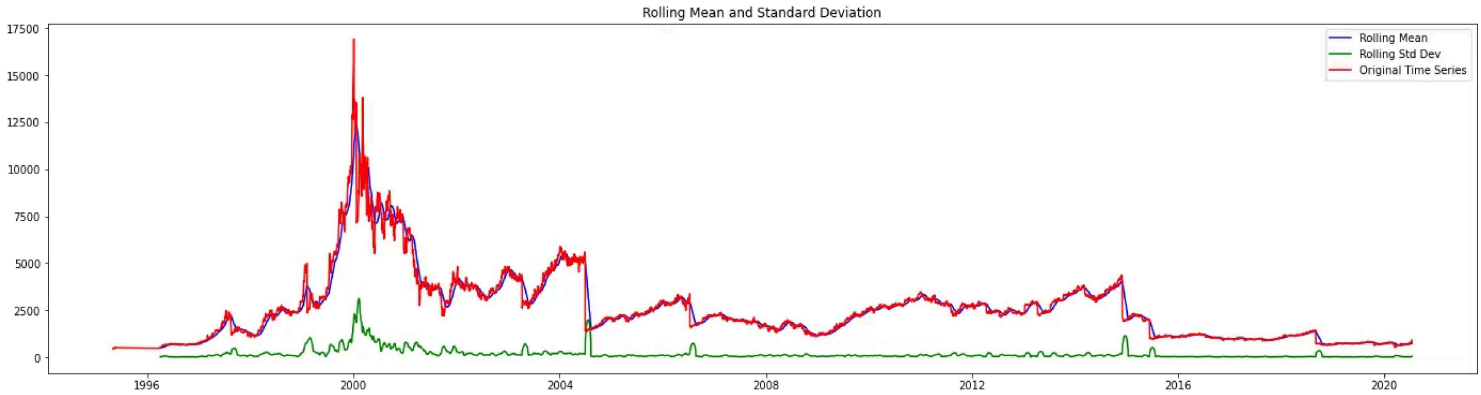
### ADF and KPSS Test

To check the stationarity of the time series, we will also use the ADF **(Augmented Dickey-Fuller)** test and KPSS **(Kwiatkowski–Phillips–Schmidt–Shintests)** test. The null hypothesis of the ADF test is that the time series is not stationary whereas that for the KPSS is that it is stationary.

```python
#Testing for stationarity using ADF and KPSS Tests.

def stationarity_test(stock_close_price):
    # Calculating rolling mean and rolling standard deviation:
    rolling_mean = stock_close_price.rolling(30).mean()
    rolling_std_dev = stock_close_price.rolling(30).std()

    # Plotting the statistics:
    plt.figure(figsize=(24,6))
    plt.plot(rolling_mean, color='blue', label='Rolling Mean')
    plt.plot(rolling_std_dev, color='green', label = 'Rolling Std Dev')
    plt.plot(stock_close_price, color='red',label='Original Time Series')
    plt.legend(loc='best')
    plt.title('Rolling Mean and Standard Deviation')

    print("ADF Test:")
    adf_test = adfuller(stock_close_price,autolag='AIC')
    print('Null Hypothesis: Not Stationary')
    print('ADF Statistic: %f' % adf_test[0])
    print('p-value: %f' % adf_test[1])
    print('Critical Values:')
    for key, value in adf_test[4].items():
        print('\t%s: %.3f' % (key, value))

    print("KPSS Test:")
    kpss_test = kpss(stock_close_price, regression='c', nlags=None, store=False)
    print('Null Hypothesis: Stationary')
    print('KPSS Statistic: %f' % kpss_test[0])
    print('p-value: %f' % kpss_test[1])
    print('Critical Values:')
    for key, value in kpss_test[3].items():
        print('\t%s: %.3f' % (key, value))

stationarity_test(Stock_data['Close'])
```



Stationarity check of time series




```
ADF Test:
Null Hypothesis: Not Stationary
ADF Statistic: -2.356262
p-value: 0.154464
Critical Values:
        1%: -3.431
        5%: -2.862
        10%: -2.567
```

```
KPSS Test:
Null Hypothesis: Stationary
KPSS Statistic: 4.169242
p-value: 0.010000
Critical Values:
        10%: 0.347
        5%: 0.463
        2.5%: 0.574
        1%: 0.739
```

We can see the rolling mean trending up and down over time as the price of the stock increases and decreases respectively. The p-value of the ADF test turns out to be greater than 0.05 which means we cannot rule out null hypothesis. The p-value of the KPSS test is below 0.05 which means we reject the null hypothesis. All these tests conclude that the time series is not stationary.

## How to de-trend the time series?

**Differencing:** A new series is constructed by calculating the value at the current time by differencing the value of actual observation of current time and its previous time.

> *value(t) = actual_observation(t) - actual_observation(t-1)*

**Transformation:** Transforming the values using power, square root, log, etc can help to linearize the data. For example, taking a log of the values can help in obtaining a linear trend to the series with an exponential trend.
**log(exp(x))=x**

**Seasonal Differencing:** The values of the time series are calculated by differencing between one observation and its previous Nth observation. This can help in removing the trend

> *value(t) = actual_observation(t) - actual_observation(t-N)*

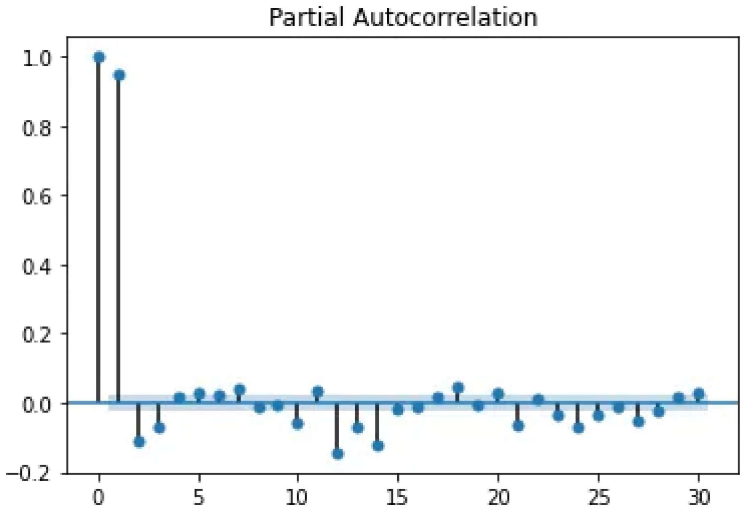
**Fitting a model:** We can fit a linear regression model to the time series. It will fit a linear trend on the time series. The values for the de-trended time series can be calculated by subtracting the actual observations with the values predicted by the model.

> *value(t) = actual_observation(t) - predicted(t)*

```python
#Testing for stationarity of de-trended time series using ADF and KPSS Tests.

# De-trending the time series
Stock_data['Close_Detrend'] = (Stock_data['Close'] - Stock_data['Close'].shift(30))

def stationarity_test(stock_close_price):
    # Calculating rolling mean and rolling standard deviation:
    rolling_mean = stock_close_price.rolling(30).mean()
    rolling_std_dev = stock_close_price.rolling(30).std()

    # Plotting the statistics:
    plt.figure(figsize=(24,6))
    plt.plot(rolling_mean, label='Rolling Mean',linewidth=2.0)
    plt.plot(rolling_std_dev, label = 'Rolling Std Dev',linewidth=2.0)
    plt.plot(stock_close_price,label='De-Trended Time Series')
    plt.legend(loc='best')
    plt.title('Rolling Mean and Standard Deviation')
    plt.tight_layout()

    print("ADF Test:")
    adf_test = adfuller(stock_close_price,autolag='AIC')
    print('Null Hypothesis: Not Stationary')
    print('ADF Statistic: %f' % adf_test[0])
    print('p-value: %f' % adf_test[1])
    print('Critical Values:')
    for key, value in adf_test[4].items():
        print('\t%s: %.3f' % (key, value))

    print("KPSS Test:")
    kpss_test = kpss(stock_close_price, regression='c', nlags=None, store=False)
    print('Null Hypothesis: Stationary')
    print('KPSS Statistic: %f' % kpss_test[0])
    print('p-value: %f' % kpss_test[1])
    print('Critical Values:')
    for key, value in kpss_test[3].items():
        print('\t%s: %.3f' % (key, value))

stationarity_test(Stock_data['Close_Detrend'].dropna())

# PACF Plot
from statsmodels.graphics.tsaplots import plot_pacf
pacf = plot_pacf(Stock_data['Close_Detrend'].dropna(), lags=30)
```


```
ADF Test:
Null Hypothesis: Not Stationary
ADF Statistic: -8.600989
p-value: 0.000000
Critical Values:
        1%: -3.431
        5%: -2.862
        10%: -2.567
```

```
KPSS Test:
Null Hypothesis: Stationary
KPSS Statistic: 0.120653
p-value: 0.100000
Critical Values:
        10%: 0.347
        5%: 0.463
        2.5%: 0.574
        1%: 0.739
```



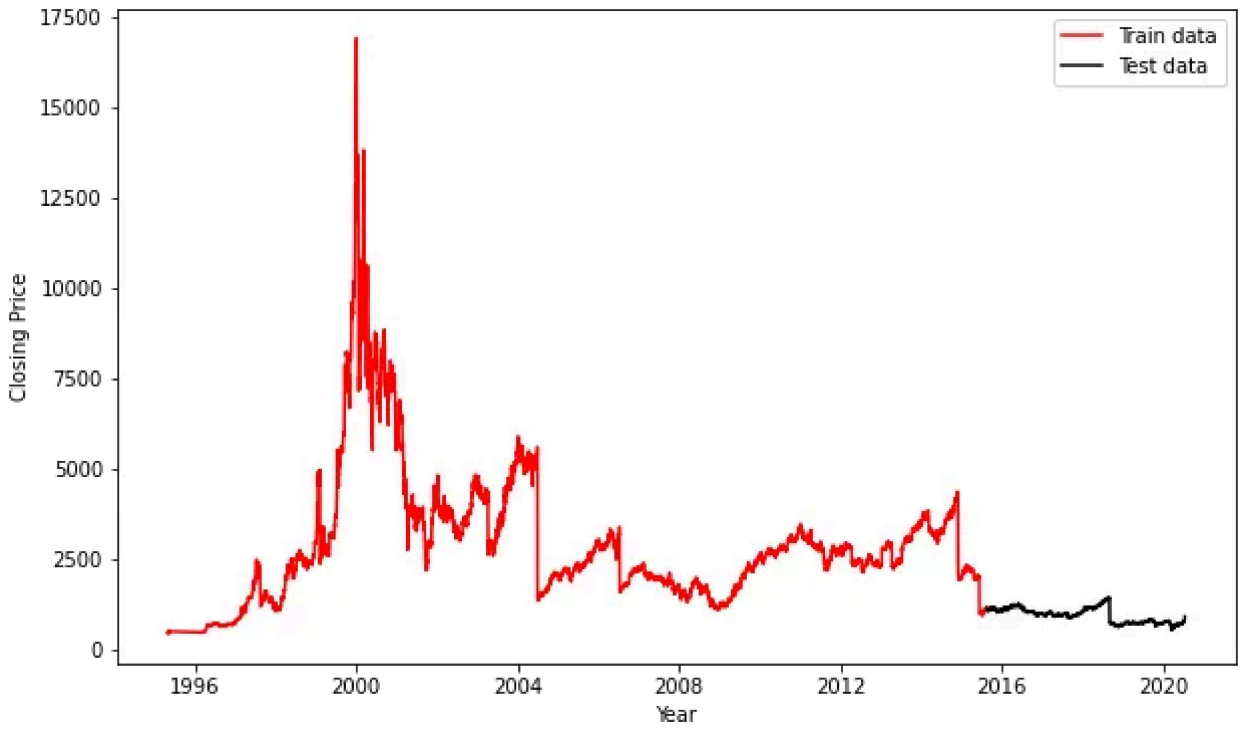

After de-trending the time series, ADF and KPSS tests indicate that the time-series is stationary. Partial AutoCorrelation Function (PACF) Plot suggests

that correlation exists at certain lags.

## Splitting the Dataset

```
#split data into train and training set
Stock_data_arima = Stock_data['Close']
train_test_split_ratio = int(len(Stock_data_arima)*0.8)
train_data, test_data = Stock_data_arima[:train_test_split_ratio], Stock_data_arima[train_test_sp


# Plotting the train and test sets.
plt.figure(figsize=(10,6))
plt.xlabel('Year')
plt.ylabel('Closing Price')
plt.plot(train_data, 'red', label='Train data')
plt.plot(test_data, 'black', label='Test data')
plt.legend()
```

dataset_split.py hosted with ❤ by GitHub view raw




Train - Test Split

## Model Building

To forecast the prices, we can use smoothing methods and ARIMA methods. Smoothing methods can be used for non-stationary data whereas ARIMA requires the time series to be stationary. We can also make use of *auto_arima,* which makes the series stationary and determines the optimal order for the ARIMA model.

For each of the methods, we will perform multiple fits for the optimization of the hyperparameters and use the optimal values for the final model.

### Simple Exponential Smoothing

Simple Exponential Smoothing or SES is used when the data does not contain any trend or seasonality. Smoothing Factor for level (α) provides weightage to the influence of the observations. Larger values of α mean that more attention is given to the most recent past observation whereas smaller values indicate that more past observations are being considered for forecasting.

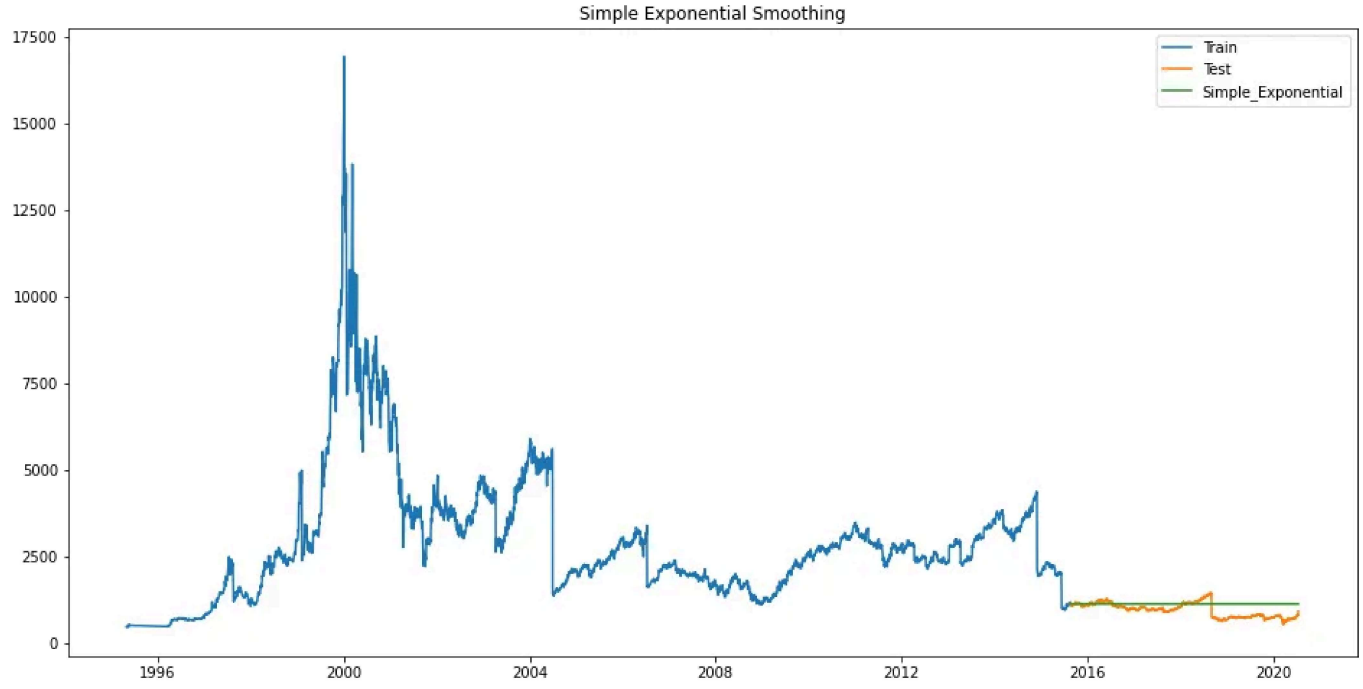
$$F_{t+1} = \alpha d_t + (1 - \alpha) F_t$$

Simple Exponential Smoothing

```
# Simple Exponential Smoothing Method
pred_values = test_data.copy()
pred_values = pd.DataFrame(pred_values)

Simple_Exponential_df = pd.DataFrame(columns = ['RMS','Smoothing Level'])

from itertools import permutations
perm = permutations(list(np.linspace(0.05,1,num=20)), 1)
for i in list(perm):
  fit_sim_exp = SimpleExpSmoothing(np.asarray(train_data)).fit(smoothing_level = i[0])
  pred_values['Simple_Exponential'] = fit_sim_exp.forecast(len(test_data))

  rms = round(sqrt(mean_squared_error(test_data.values, pred_values.Simple_Exponential)),3)
  Simple_Exponential_df = Simple_Exponential_df.append(other = {'RMS' : rms , 'Smoothing Level' :

opt_values = Simple_Exponential_df.loc[Simple_Exponential_df['RMS'] == min(Simple_Exponential_df[


# Using optimised values from the lists.
fit_sim_exp = SimpleExpSmoothing(np.asarray(train_data)).fit(smoothing_level = opt_values[0][0])
pred_values['Simple_Exponential'] = fit_sim_exp.forecast(len(test_data))

plt.figure(figsize=(16,8))
plt.plot(train_data, label='Train')
plt.plot(test_data, label='Test')
plt.plot(pred_values['Simple_Exponential'], label='Simple_Exponential')
plt.legend(loc='best')
plt.show()

rms_sim_exp = sqrt(mean_squared_error(test_data.values, pred_values.Simple_Exponential))
print("Simple Exponential Smoothing RMS :- " + str(round(rms_sim_exp,3)) + " & Smoothing Level :-
```

simple_exponential.py hosted with ❤ by GitHub view raw

Simple Exponential Smoothing Snippet




Simple Exponential Smoothing Forecast

## Holt's Exponential Smoothing

Holt's Exponential Smoothing takes the trend into account for forecasting the time series. It is used when there is a trend in the data and no seasonality. It calculates the Smoothing value (the first equation), which is the same used in SES for forecasting. Trend Coefficient (β) provides weightage to the difference in the consequent smoothing values and the previous trend estimate. The forecasting is a combination of the smoothing value and the trend estimate.

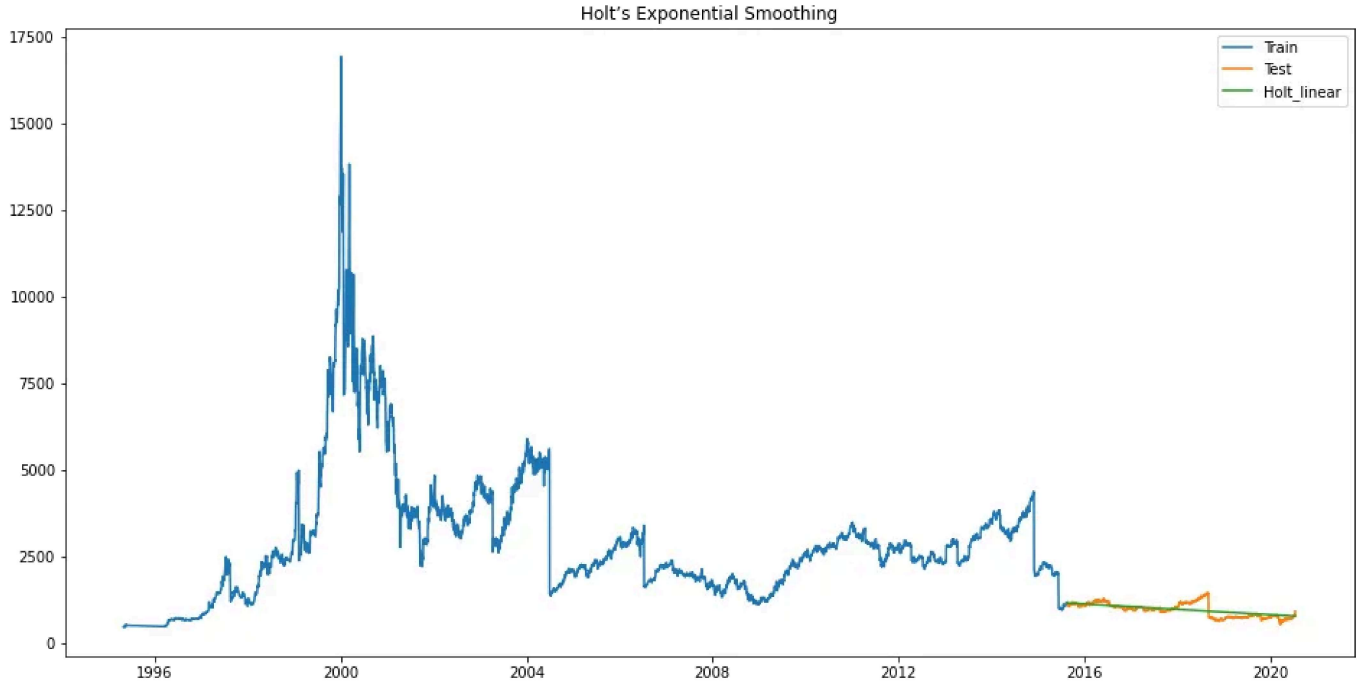
$$S_t = \alpha d_t + (1-\alpha)F_t$$
$$b_t = \beta(S_t - S_{t-1}) + (1-\beta)b_{t-1}$$
$$F_{t+1} = S_t + b_t$$

Holt's Exponential Smoothing

```
# Holt's Exponential Smoothing Method
holt_linear_df = pd.DataFrame(columns = ['RMS','Smoothing Level','Smoothing Slope'])

from itertools import permutations
perm = permutations(list(np.linspace(0.05,1,num=20)), 2)
for i in list(perm):
  fit_holt = Holt(np.asarray(train_data)).fit(smoothing_level = i[0],smoothing_slope=i[1])
  pred_values['Holt_linear'] = fit_holt.forecast(len(test_data))

  rms = round(sqrt(mean_squared_error(test_data.values, pred_values.Holt_linear)),3)
  holt_linear_df = holt_linear_df.append(other = {'RMS' : rms , 'Smoothing Level' : i[0], 'Smooth

opt_values = holt_linear_df.loc[holt_linear_df['RMS'] == min(holt_linear_df['RMS']),['Smoothing L


# Using optimised values from the lists.
fit_holt = Holt(np.asarray(train_data)).fit(smoothing_level = opt_values[0][0],smoothing_slope=op
pred_values['Holt_linear'] = fit_holt.forecast(len(test_data))

plt.figure(figsize=(16,8))
plt.plot(train_data, label='Train')
plt.plot(test_data, label='Test')
plt.plot(pred_values['Holt_linear'], label='Holt_linear')
plt.legend(loc='best')
plt.title('Holt Exponential Smoothing')
plt.show()

rms_holt_exp = sqrt(mean_squared_error(test_data.values, pred_values.Holt_linear))
print("Holt's Exponential Smoothing RMS :- " + str(round(rms_holt_exp,3)) + " & Smoothing Level :
```

holt_exponential.py hosted with ❤ by GitHub view raw

Holt's Exponential Smoothing Snippet




Holt's Exponential Smoothing Forecast

## Holt-Winters Exponential Smoothing

Holt-Winters Exponential Smoothing takes trend as well as seasonality into account for forecasting the time series. It forecasts the values using equations for calculating the level component, trend component, and the seasonal component in the time series. According to the seasonal variations in the data, either additive or the multiplicative version is used.

$$\hat{y}_{t+h|t} = \ell_t + hb_t + s_{t+h-m(k+1)}$$
$$\ell_t = \alpha(y_t - s_{t-m}) + (1-\alpha)(\ell_{t-1} + b_{t-1})$$
$$b_t = \beta^*(\ell_t - \ell_{t-1}) + (1-\beta^*)b_{t-1}$$
$$s_t = \gamma(y_t - \ell_{t-1} - b_{t-1}) + (1-\gamma)s_{t-m},$$

Additive Method (Source: otexts)

$$\hat{y}_{t+h|t} = (\ell_t + hb_t)s_{t+h-m(k+1)}$$
$$\ell_t = \alpha\frac{y_t}{s_{t-m}} + (1-\alpha)(\ell_{t-1} + b_{t-1})$$
$$b_t = \beta^*(\ell_t - \ell_{t-1}) + (1-\beta^*)b_{t-1}$$
$$s_t = \gamma\frac{y_t}{(\ell_{t-1} + b_{t-1})} + (1-\gamma)s_{t-m}$$

Multiplicative Method (Source: otexts)

### Auto-Regressive Integrated Moving Average (ARIMA)

ARIMA model is a combination of Auto-Regressive model and Moving Average model along with the Integration of differencing. Auto-Regressive model determines the relationship between an observation and a certain number of lagged observations. The Integrated part is the differencing of the actual observations to make the time series stationary. Moving Average determines the relationship between an observation and residual error obtained by using a moving average model on the lagged observations.

*Auto-Regressive (p) -> Number of lag observations in the model. Also called as the lag order.*

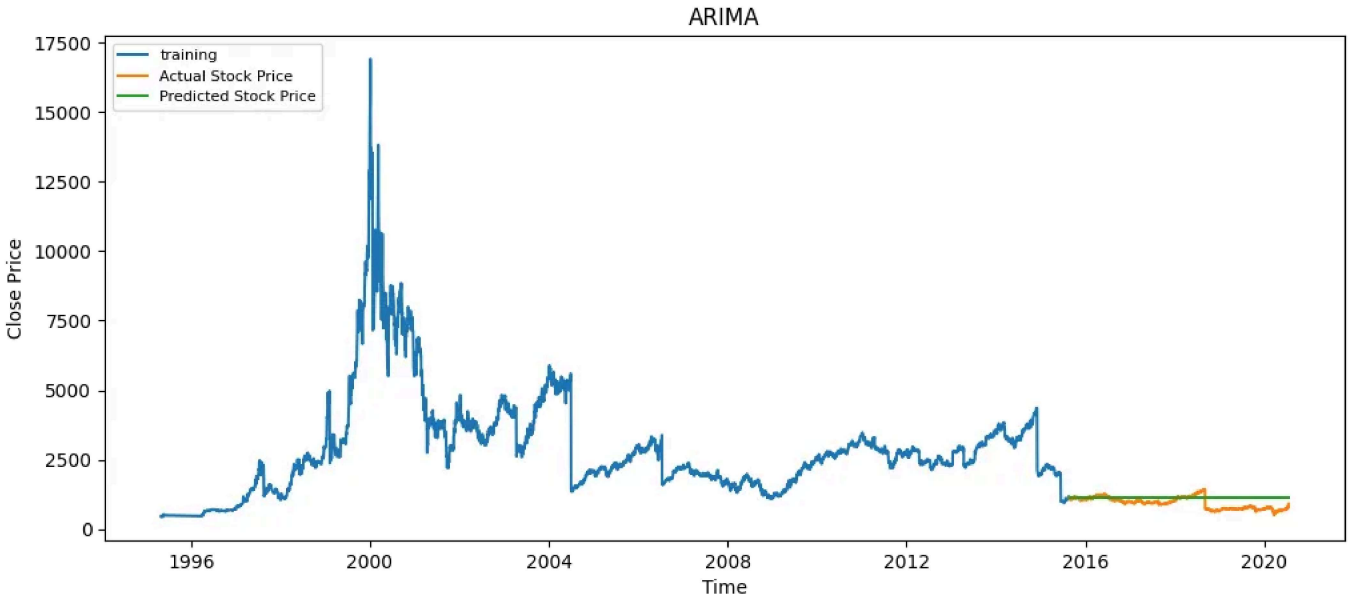
*Integrated (d) -> The number of times the actual observations are differenced for stationarity. Also called as the degree of differencing.*

*Moving Average (q) -> Size of the moving average window. Also called as the order of moving average.*

```python
# Auto ARIMA Method
arima_model = auto_arima(train_data,
                         start_p=1, start_q=1,
                         max_p=5, max_q=5,
                         test='adf',
                         trace=True,
                         alpha=0.05,
                         scoring='mse',
                         suppress_warnings=True,
                         seasonal = False
                         )

# Fitting the final model with the order
fitted_model = arima_model.fit(train_data)
print(fitted_model.summary())

# Forecasting the values.
forecast_values = fitted_model.predict(len(test_data), alpha=0.05)
fcv_series = pd.Series(forecast_values[0], index=test_data.index)

#Plotting the predicted stock price and original price.
plt.figure(figsize=(12,5), dpi=100)
plt.plot(train_data, label='training')
plt.plot(test_data, label='Actual Stock Price')
plt.plot(fcv_series,label='Predicted Stock Price')
plt.title('Stock Price Prediction')
plt.xlabel('Time')
plt.ylabel('Close Price')
plt.legend(loc='upper left', fontsize=8)
plt.show()

# Evaluating the model by calculating RMSE.
rms_auto_arima = sqrt(mean_squared_error(test_data.values, fcv_series))
print("Auto-Arima RMSE :- " + str(round(rms_auto_arima,3)))
```



ARIMA Snippet




ARIMA Forecast

## Summary

To evaluate the performance of the model, we will use Root Mean Squared Error (RMSE) and compare which model performed better than the rest.

```
RMSE of all the methods
Auto-Arima:- 278.264
Simple Exponential Smoothing:- 264.387
Holt's Exponential Smoothing:- 148.269
```

RMSE Summary

Out of the three models, the better performing model was Holt's Exponential Smoothing method which obtained least RMSE.

In this article, we saw about time series and its components, fetching data using API call and packages, checking stationarity of the time series, detrending the time series, different types to model the time series, and finally its evaluation.

You can find the notebook for your reference here.

Towards Data Science | Timeseries Forecasting | Machine Learning | Analytics | Time Series Analysis

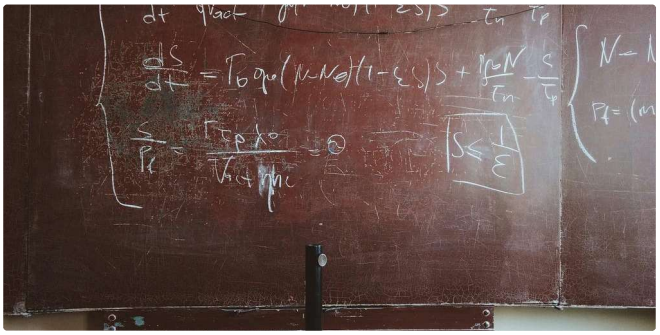
## Written by Prathamesh Thakar

Follow

34 Followers · Writer for Towards Data Science

Interested in learning and sharing about data science concepts.

### More from Prathamesh Thakar and Towards Data Science



Prathamesh Thakar in Towards Data Science

Shaw Talebi in Towards Data Science

## The math behind Machine Learning Algorithms

How do different machine learning algorithms learn from the data and predict on the unsee...

Jul 31, 2020 300 1

## 5 AI Projects You Can Build This Weekend (with Python)

From beginner-friendly to advanced

Oct 9 3K 49

Mauro Di Pietro in Towards Data Science

## GenAI with Python: Build Agents from Scratch (Complete Tutorial)

with Ollama, LangChain, LangGraph (No GPU, No APIKEY)

Sep 29 1.6K 23

Thuwarakesh Murallie in Towards Data Science

## The Most Valuable LLM Dev Skill is Easy to Learn, But Costly to...

Here's how not to waste your budget on evaluating models and systems.

Oct 9 897 24

See all from Prathamesh Thakar

See all from Towards Data Science

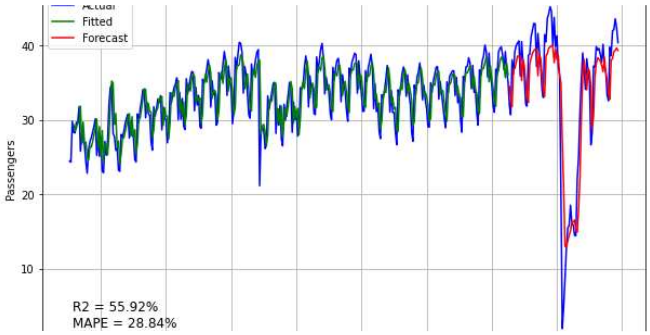
## Recommended from Medium

Jonas Dieckmann in Towards Data Science

## Getting Started Predicting Time Series Data with Facebook Prophet

This article aims to take away the entry barriers to get started with time series...

Jan 30 347 3




Seyed Mousavi

## How to fix a common mistake in LSTM time series forecasting

May 3 481 4

### Lists


**Predictive Modeling w/ Python**

20 stories · 1601 saves

**Practical Guides to Machine Learning**

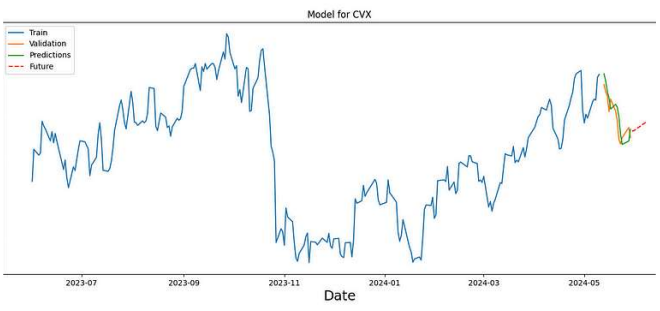
10 stories · 1949 saves

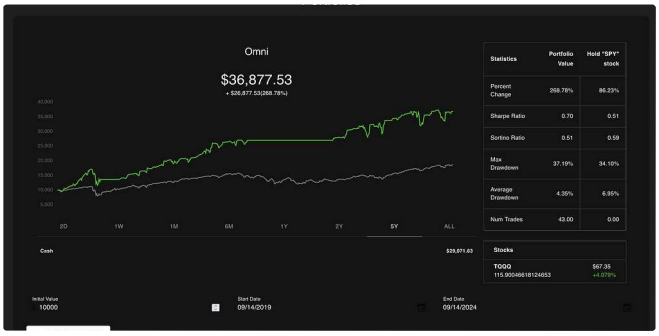
**Natural Language Processing**

1760 stories · 1359 saves

**The New Chatbots: ChatGPT, Bard, and Beyond**

12 stories · 485 saves



Sercan Bugra Gultekin

## (1) Stock Price Prediction with ML in Python: LSTM (Long short-ter...

In this series, we will discuss how we can make predictions about stock prices with...

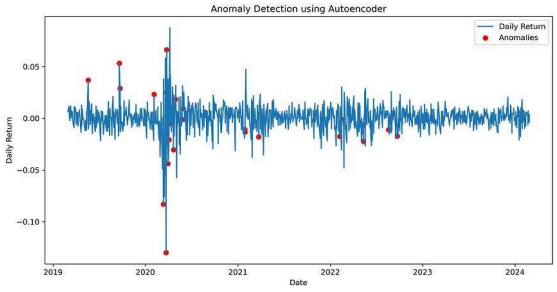
May 30 14

Austin Starks in DataDrivenInvestor

## I used OpenAI's o1 model to develop a trading strategy. It is...

It literally took one try. I was shocked.

Sep 16 4.1K 115

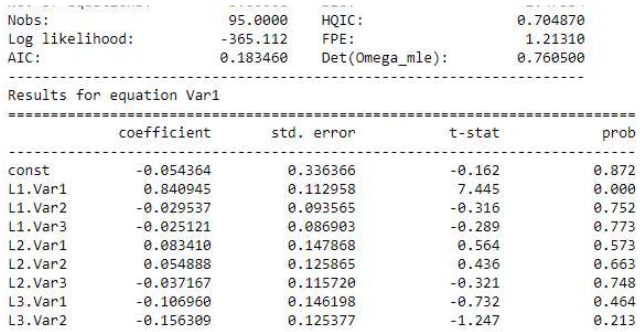



Abhay Dodiya in GoPenAI

## Anomaly Detection in Time Series using Autoencoder.

Anomaly Detection in Time Series with the help of Autoencoder will help us to decode...

Oct 3 76



| | | | |
|---|---|---|---|
| Nobs: | 95.0000 | HQIC: | 0.704870 |
| Log likelihood: | -365.112 | FPE: | 1.21310 |
| AIC: | 0.183460 | Det(Omega_mle): | 0.760500 |

Results for equation Var1

| | coefficient | std. error | t-stat | prob |
|---|---|---|---|---|
| const | -0.054364 | 0.336366 | -0.162 | 0.872 |
| L1.Var1 | 0.840945 | 0.112958 | 7.445 | 0.000 |
| L1.Var2 | -0.029537 | 0.093565 | -0.316 | 0.752 |
| L1.Var3 | -0.025121 | 0.086903 | -0.289 | 0.773 |
| L2.Var1 | 0.083410 | 0.147868 | 0.564 | 0.573 |
| L2.Var2 | 0.054888 | 0.125865 | 0.436 | 0.663 |
| L2.Var3 | -0.037167 | 0.115720 | -0.321 | 0.748 |
| L3.Var1 | -0.106960 | 0.146198 | -0.732 | 0.464 |
| L3.Var2 | -0.156309 | 0.125377 | -1.247 | 0.213 |

Chris Yan

## Understanding the Vector AutoRegressive (VAR) Model Wh...

The Vector AutoRegressive (VAR) model is a powerful tool in time series analysis,...

Jul 31 1

See more recommendations