Machine Learning Engineer Nanodegree

Capstone Project

Francisco Ramos
July 14tht, 2019

I. Definition

Project Overview

With the boom of AI the financial industry is experiencing interesting changes and improvements. Many companies are starting to integrate Machine Learning in their businesses. Fraud prevention, risk management, portfolio management, investment prediction, process automation, are some of the applications of ML in finance. The Stock Market and Algorithmic Trading are other areas benefiting from it. Traders are embracing some of the new techniques and algorithms to improve their performance.

Time series forecasting has become a hot topic with the advances in Machine Learning. Time series analysis and predictions is one of the oldest and most applied data science techniques in finance, but new ones are emerging, more sophisticated and powerful, providing with more accuracy, using the power of Deep Learning... But before we start rubbing our hands in glee and greediness thinking we can all use these algorithms to easily get rich, I need to warn you: Stock market prices are highly unpredictable and volatile. There are no consistent patterns in the data that would allow you to model stock prices over time. Princeton University economist Burton Malkiel wrote in his 1973 book, “A Random Walk Down Wall Street”, that “A blindfolded monkey throwing darts at a newspaper's financial pages could select a portfolio that would do just as well as one carefully selected by experts.”

Problem Statement

There is a lot of literature on internet about Stock Market prices forecasting using different Machine Learning algorithms, most recently using complex LSTM Networks, or combination of LSTM-CNN. They all show incredible performance, predicting almost the same behaviour as the price. But these are mostly optical illusions due to the fact that most of them are simply predicting one day ahead. Simple methods such as Simple Moving Average or a bit more sophisticated Exponential Moving Average could also achieve similar or even better

performance, and this is because prices normally don't change drastically overnight. Predicting more than one day is a whole different story. I intend to predict one week of prices, that is, 5 trading days.

As mentioned above, Stock Market prediction is a difficult task. There are many variables that affect how the share prices change over time. Combination of different factors make them extremely volatile and therefore quite unpredictable, or at least hard to predict with some level of accuracy.

According to financial theory, the stock market prices evolve following the so called random walk, but there is evidence that dismisses this theory, and they could be somehow modeled based on historical information.

Stock Market analysis is divided into two parts: Fundamental Analysis and Technical Analysis. Fundamental Analysis involves analyzing the company's future profitability on the basis of its current business environment and financial performance. Technical Analysis, on the other hand, includes reading the charts and using statistical figures to identify the trends in the stock market. In this project I'm gonna focus on the technical analysis part.

There are many solutions out there for time series forecasting such as ARIMA models, Support Vector Machines, or state-of-the-art LSTM models, CNN model, or a combination of both. Unfortunately there is no much written – only found a few articles but no examples – about using Generative Adversarial Networks for such problems. GANs are mostly used to generate very realistic images, regardless of the type of image. In this project we're gonna explore these amazing networks and see how we can use them to predict future prices.

Metrics

Time series prediction performance measures provide a summary of the skills and capability of the forecast model that made the predictions. We're gonna take a look at the following ones:

- Mean Forecast Error (or Forecast Bias). Forecast errors can be positive and negative. This means that when the average of these values is calculated, an ideal mean forecast error would be zero. A mean forecast error value other than zero suggests a tendency of the model to over forecast (negative bias) or under forecast (positive bias).

Calculating Forecast Error:

$$e(t) = y(t) - \hat{y}(t|t-1)$$

where,

$y(t)$ = observation

$\hat{y}(t|t-1)$ = denote the forecast of $y(t)$ based on all previous observations

We calculate the bias by averaging these errors.

- Mean Absolute Error (MAE). It's calculated as the average of the forecast error values, where all of the forecast values are forced to be positive.

$$\mathrm{MAE} = \frac{\sum_{i=1}^{n} |y_i - x_i|}{n} = \frac{\sum_{i=1}^{n} |e_i|}{n}$$

- Mean Squared Error (MSE). It's calculated as the average of the squared forecast error values. Squaring the forecast error values forces them to be positive; it also has the effect of putting more weight on large errors. Very large or outlier forecast errors are squared, which in turn has the effect of dragging the mean of the squared forecast errors out resulting in a larger mean squared error score. In effect, the score gives worse performance to those models that make large wrong forecasts.

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

- Root Mean Square Deviation (RMSD), or Root Mean Square Error (RMSE). MSE is in the squared units of the predictions. It can be transformed back into the original units of the predictions by taking the square root of the mean squared error score.

$$\mathrm{RMSD} = \sqrt{\frac{\sum_{t=1}^{T}(\hat{y}_t - y_t)^2}{T}}$$

- Mean Absolute Percentage Error (MAPE). It's another measure of prediction accuracy of a forecasting method worth to take into consideration.

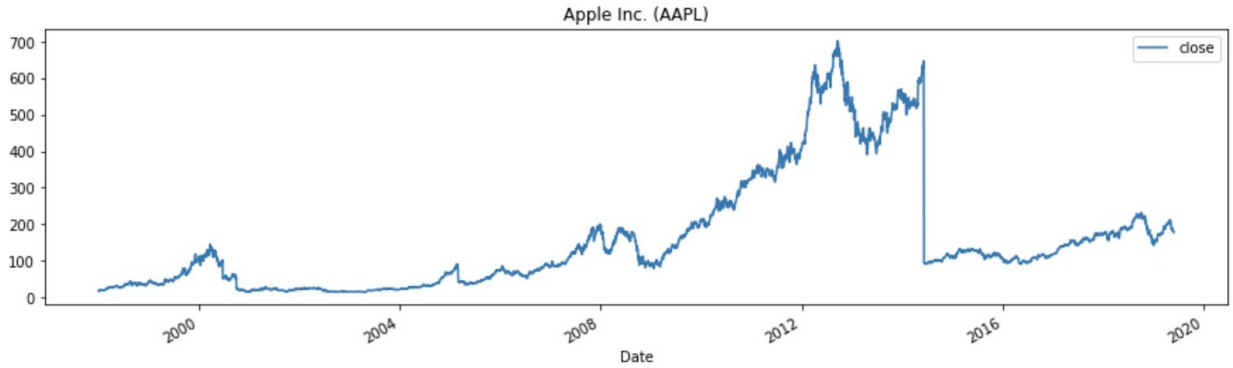
$$\mathrm{M} = \frac{100\%}{n}\sum_{t=1}^{n}\left|\frac{A_t - F_t}{A_t}\right|$$

where A_t is the actual value and F_t is the forecast value.

II. Analysis

Data Exploration

For this project I'm gonna make use of historical stock market data that one can easily obtain for free from Kaggle, Yahoo! Finance or Alpha Vantage (API key is required). I'm gonna focus on Apple stock prices (AAPL) just because, well, it's a well known company and I'm curious to know what I can get out of it.



The dataset we have on our hands is the typical OHLC + Volume dataset providing with the four major data points over a period, Open price, High price, Low price, with the Closing price being considered the most important by many traders. This is what it looks like:

Date	Open	High	Low	Close	Volume
1980-12-12	0.513393	0.515625	0.513393	0.513393	117258400.0
1980-12-15	0.488839	0.488839	0.486607	0.486607	43971200.0
1980-12-16	0.453125	0.453125	0.450893	0.450893	26432000.0
1980-12-17	0.462054	0.464286	0.462054	0.462054	21610400.0
1980-12-18	0.475446	0.477679	0.475446	0.475446	18362400.0

Note: there was another column, Adjusted Close price (Adj Close), which I decided to drop for the sake of simplicity.

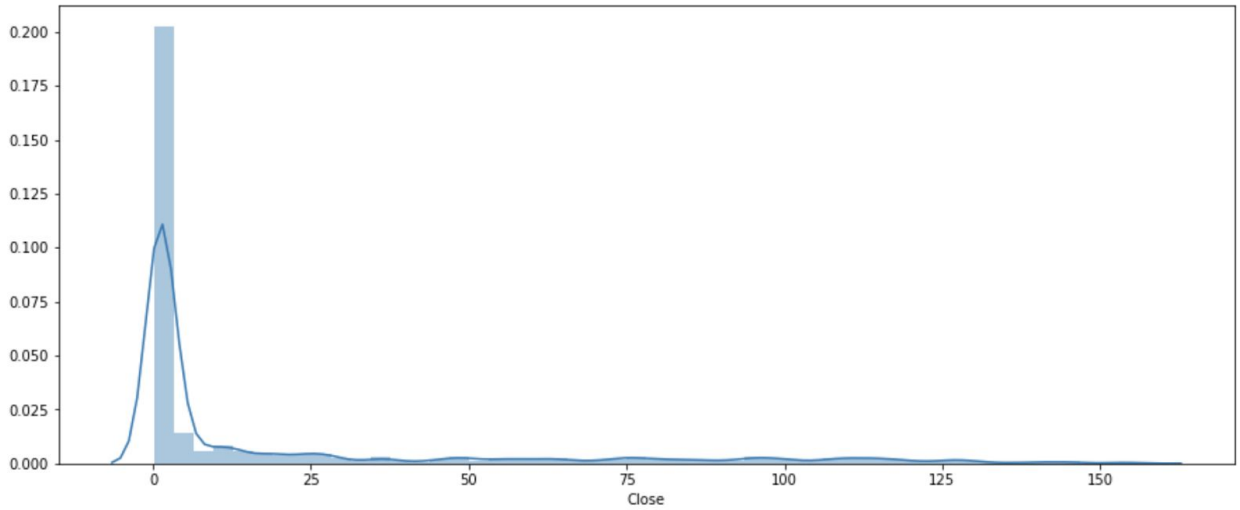
The first thing to notice here is the sudden drop around 2014, precisely June 9th, 2014. What happened here was a Stock Split. This happened when the company divided its existing shares into multiple ones, in this case a 7 to 1 split, to boost the liquidity of the shares. Refer to the section Data Preprocessing for the solution.

To finish this section, let's show some descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution:

	Open	High	Low	Close	Volume
count	9699.000000	9699.000000	9699.000000	9699.000000	9.699000e+03
mean	27.943288	28.212760	27.659935	27.943220	8.696928e+07
std	49.277582	49.707736	48.844054	49.288333	8.650217e+07
min	0.198661	0.198661	0.196429	0.196429	3.472000e+05
25%	1.057143	1.077857	1.035714	1.058036	3.368495e+07
50%	1.678571	1.718750	1.651786	1.683036	5.880980e+07
75%	27.407142	27.742143	27.095714	27.362857	1.084856e+08
max	230.779999	233.470001	229.779999	232.070007	1.855410e+09

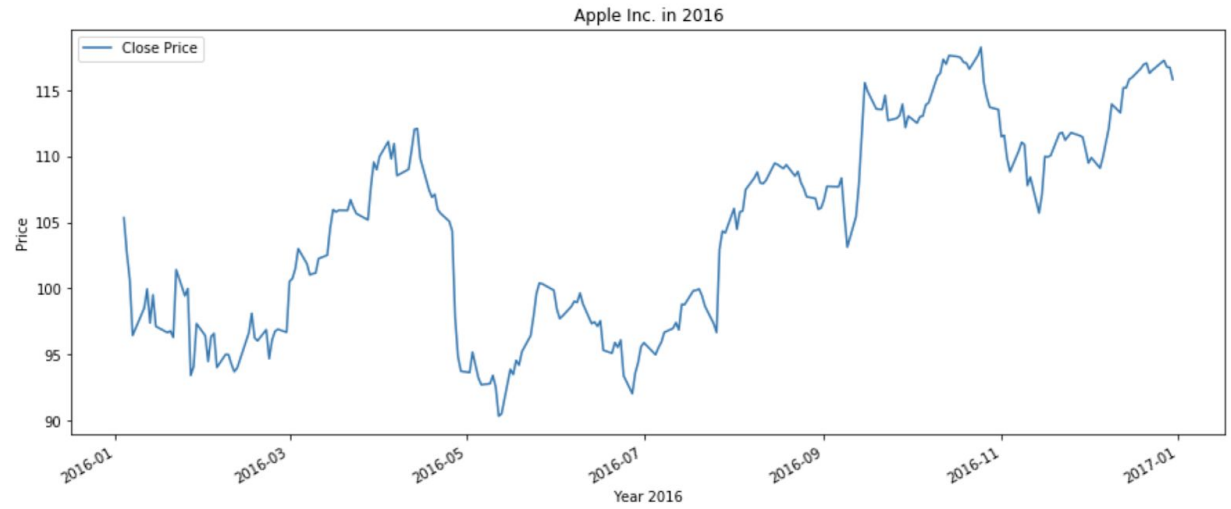
Exploratory Visualization

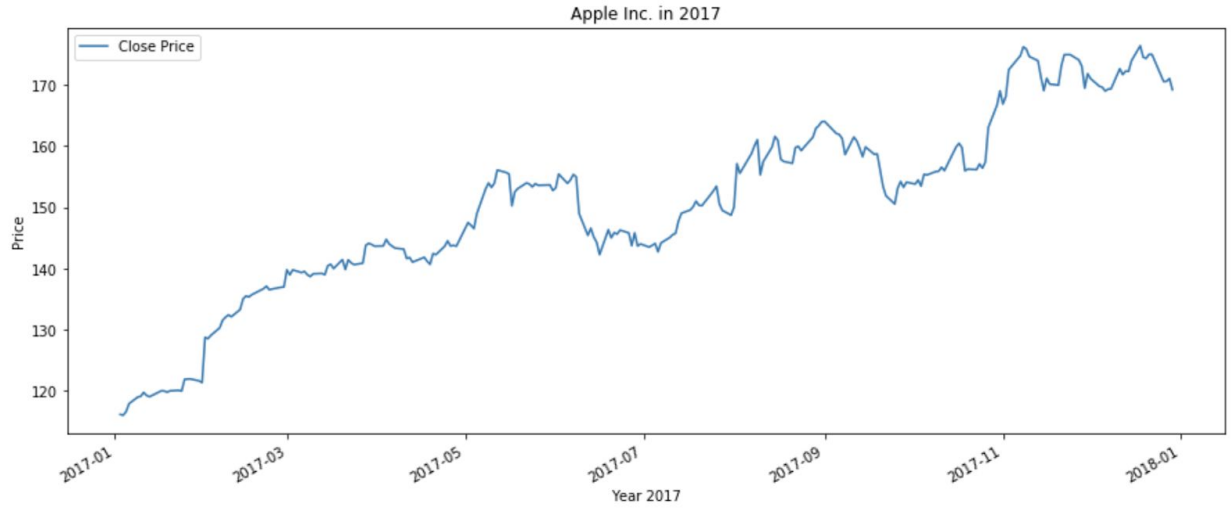
We're trying to predict Close price. Let's have a look at the distribution of this column. Let's not forget that there is no temporal ordering when plotting the price distribution:

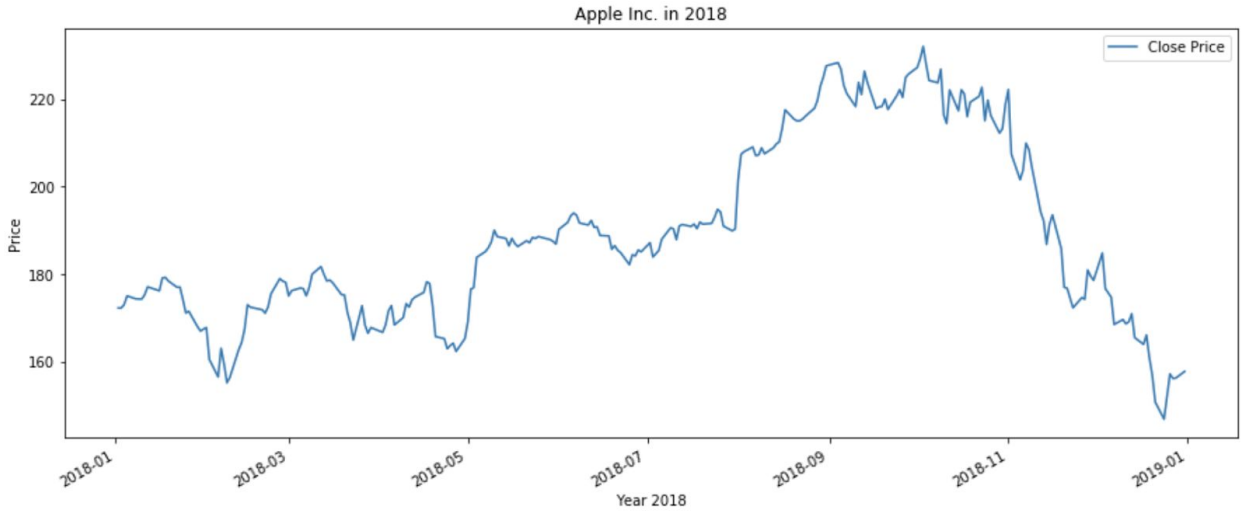


As we can see it's right skewed. By applying a power transformation we could make it more-normal (Gaussian), removing a change in variance over time. Data transforms are also intended to remove noise and improve the signal in time series forecasting.

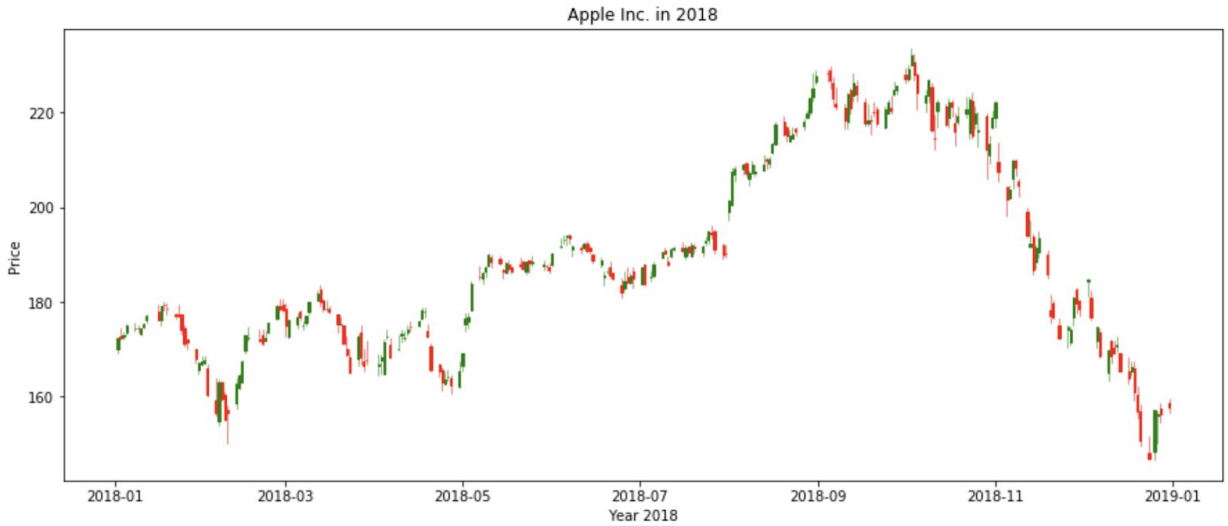
Let's plot some latest years in order to see if we can spot patterns:



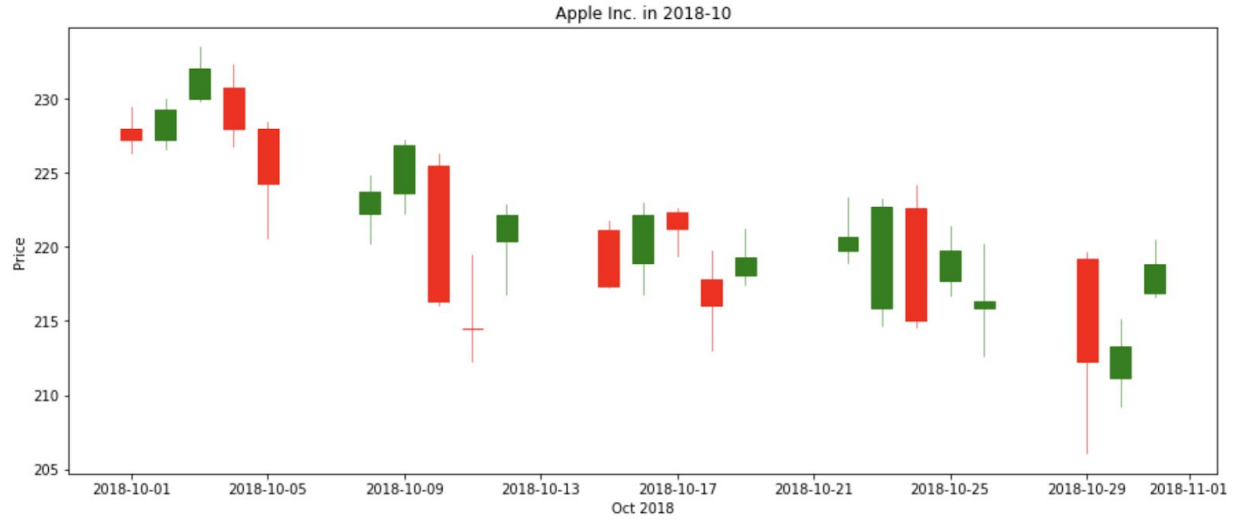


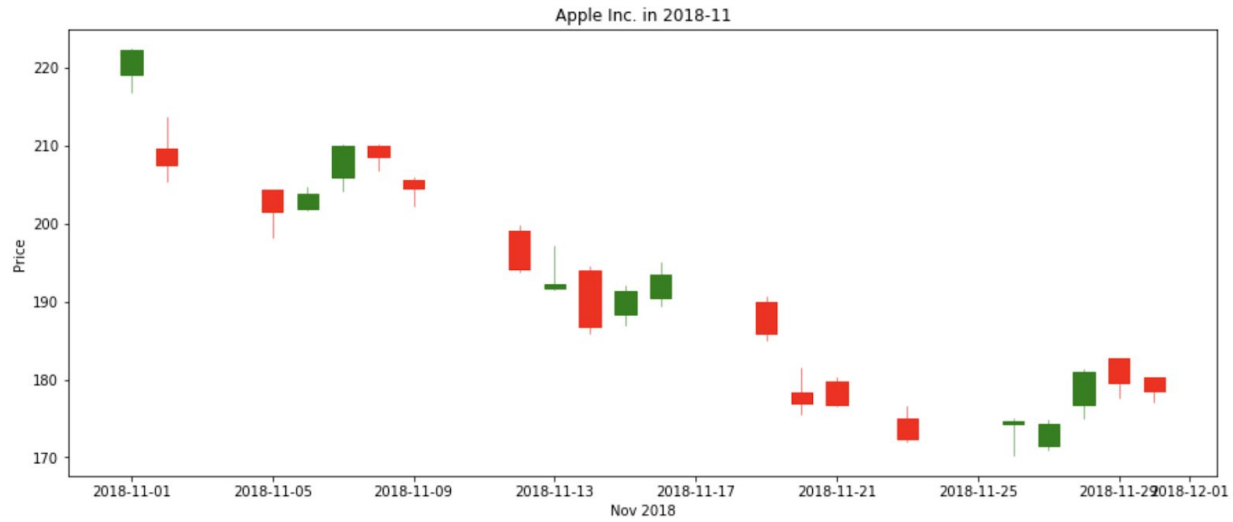


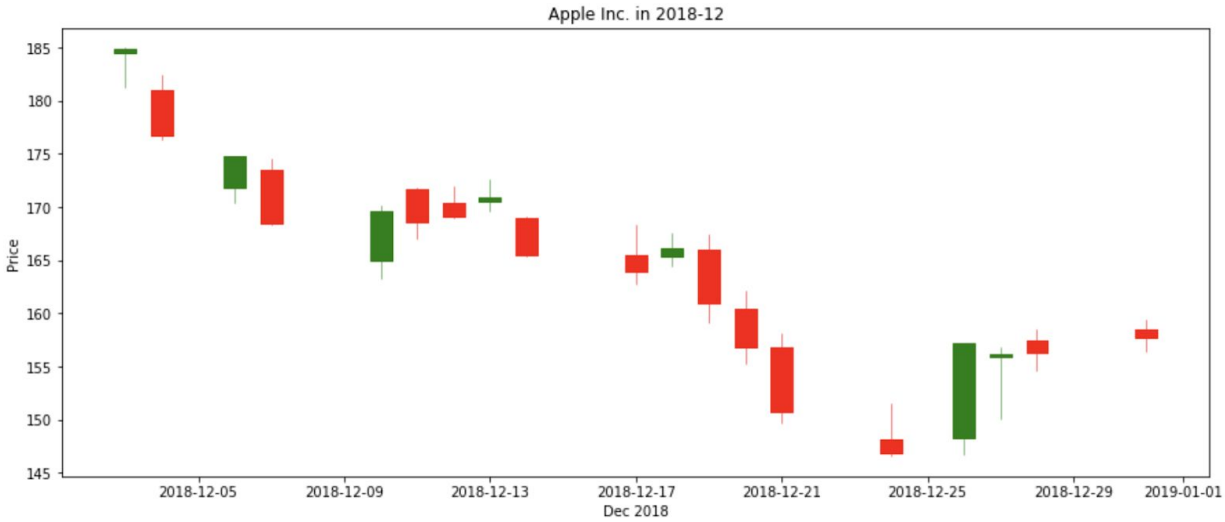
As we can see there isn't a clear pattern here. Instead it all looks quite random. Following, we have a look at this last year 2018 using candlestick plots where we have a clearer image of the Open, High, Low and Close columns:



Let's zoom in a bit more. Oct, Nov and Dec 2018:







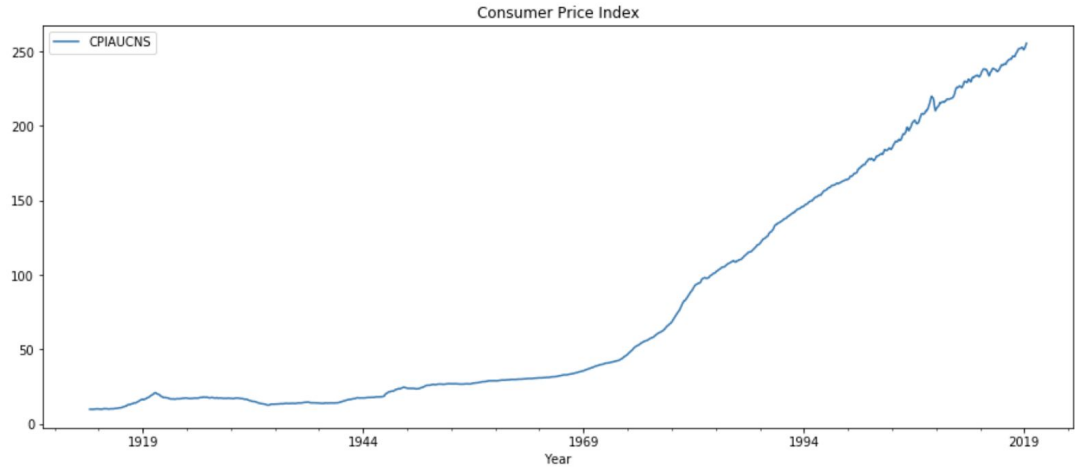
It would be interesting to see what the time series looks like after adjusting inflation. Inflation-adjustment gives you a much better understanding of real prices. We need to first load the [Consumer Price Index for All Urban Consumers: All Items]

```
cpi.head()
```

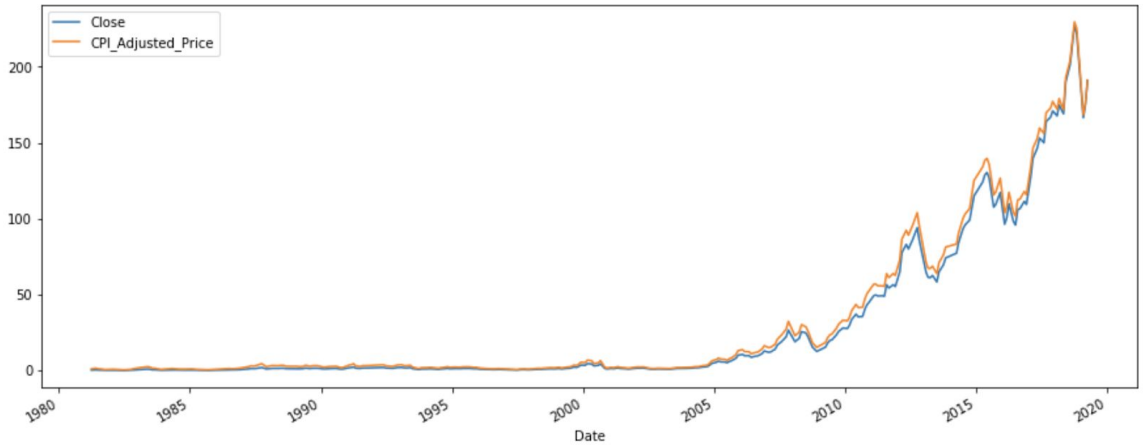
Date	CPIAUCNS
1913-01-01	9.8
1913-02-01	9.8
1913-03-01	9.8
1913-04-01	9.8
1913-05-01	9.7

```
print('Consumer Price Index from {} to {}'.format(cpi.index[0], cpi.index[-1]))
```

```
Consumer Price Index from 1913-01-01 00:00:00 to 2019-04-01 00:00:00
```



After adjusting the price, I was expecting a flatter line. To my surprise there wasn't an obvious change, which means Apple shares rocketed around 2005:

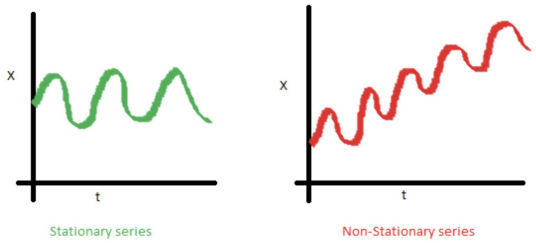


Note: see Jupyter notebook to see how the price was adjusted.

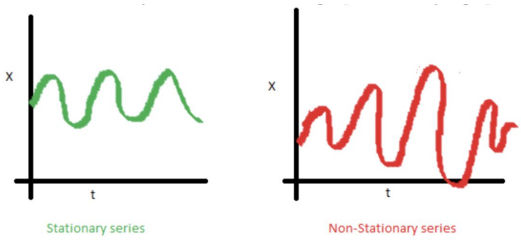
We're handling here a time series. As such, it's very important the concept of Stationarity. Much of the analysis carried out on financial time series data involves identifying if the series we want to predict is stationary, and if it is not, finding ways to transform it such that it is stationary. A time series is said to be stationary if its mean and variance don't change over time, this means, it doesn't show trends or seasonal effects.

Mean of a time series x_t *is* $E(x_t) = \mu(t)$

Variance of a time series x_t *is* $\sigma^2(t) = E[(x_t - \mu(t))^2]$



The image above shows a clear trend (red line), where the mean is increasing over time.



Images by courtesy of seanabu.com

In this one we can clearly see that the variance is a function of time. Notice in the red graph the varying spread of data over time.

Why is this important? When running a linear regression the assumption is that the observations are all independent of each other. In a time series, however, we know that observations are time dependent. It turns out that a lot of nice results that hold for independent random variables (law of large numbers and central limit theorem to name a couple) hold for stationary random variables. So by making the data stationary, we can actually apply regression techniques to this time dependent variable.

There are a few ways to test for stationarity. I decided to go for Dickey–Fuller test. StatsModel library provides with this test.

The Dickey Fuller test is one of the most popular statistical tests. It can be used to determine the presence of unit root in the series, and hence help us understand if the series is stationary or not. The null and alternate hypothesis of this test are:

- Null Hypothesis: The series has a unit root (value of a =1)
- Alternate Hypothesis: The series has no unit root.

If we fail to reject the null hypothesis, we can say that the series is non-stationary. This means that the series can be linear or difference stationary.

This is the result of testing Close price:

```
adf_test(apple_stock['Close'])
```

```
Results of Dickey-Fuller Test:
Test Statistic                    1.059025
p-value                           0.994851
Lags Used                        37.000000
Number of Observations Used    9661.000000
Critical Value (1%)              -3.431027
Critical Value (5%)              -2.861839
Critical Value (10%)             -2.566929
dtype: float64
```

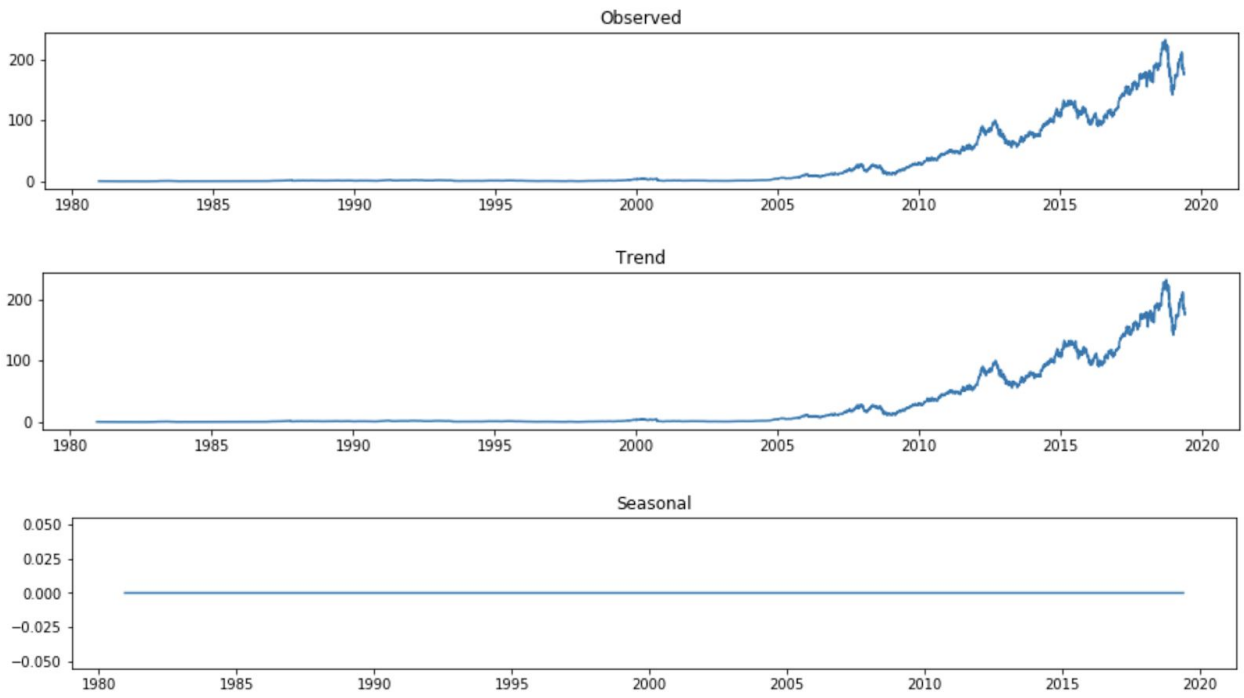
If the Test Statistic is less than the Critical Value, we can reject the null hypothesis, then the series is stationary. When the Test Statistic is greater than the Critical Value, we fail to reject the null hypothesis, which means the series is non-stationary. This is what's happening with Apple Close price, Test Statistic = 1.06 > Critical Value (1%) = -3.43, therefore it's not stationary.

StatsModel library has also an interesting tool to decompose a time series into trend and seasonal components. It could help us to know what we're dealing with and try to remove those components making it stationary:

Decomposing the time series

```
from statsmodels.tsa.seasonal import seasonal_decompose
```

```
result = seasonal_decompose(apple_stock['Close'], freq=1)
```



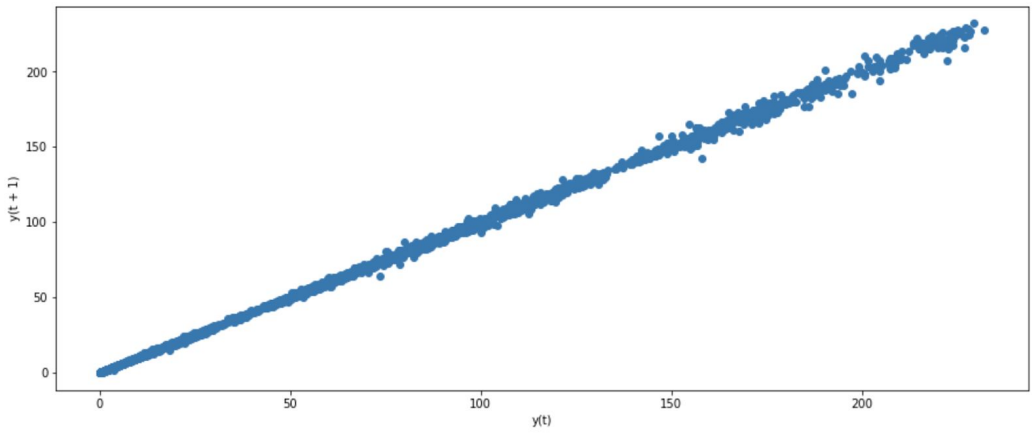
We can confirm that this time series is seasonal stationary.

Time series modeling assumes a relationship between an observation and the previous observation. Previous observations in a time series are called lags, with the observation at the previous time step called lag=1, the observation at two time steps ago lag=2, and so on. A useful type of plot to explore the relationship between each observation and a lag of that observation is called the scatter plot. Pandas has a built-in function for exactly this called the lag plot. It plots the observation at time t on the x-axis and the observation at the next time step (t+1) on the y-axis.

```
from pandas.plotting import lag_plot, autocorrelation_plot
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
```

```
fig, ax = plt.subplots(figsize=(15, 6))
lag_plot(apple_stock['Close'], lag=1, ax=ax)
```

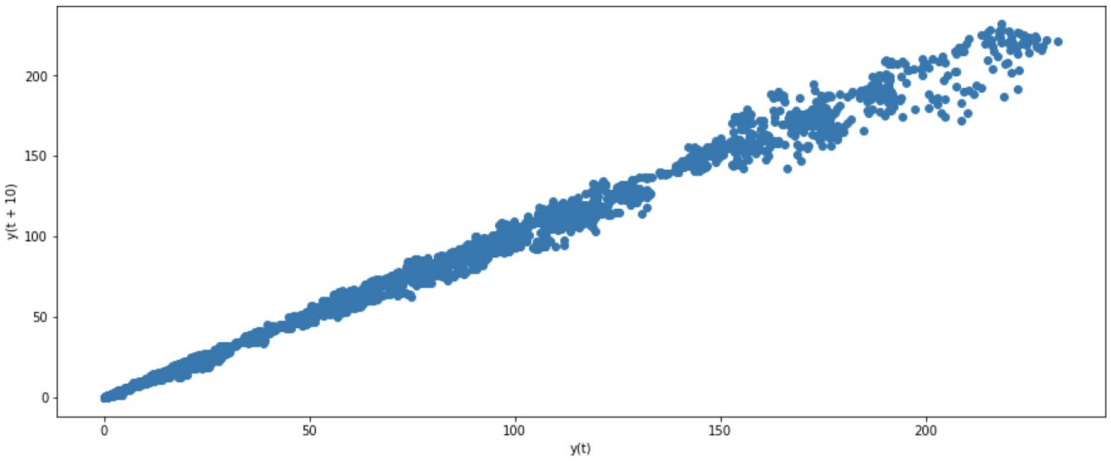
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fb39ad45ef0>
```



We can see a linear pattern that indicates the data is not random, with a positive and strong correlation (autocorrelation is present) at lag=1. It'll be less strong as my increase the lag:

```
fig, ax = plt.subplots(figsize=(15, 6))
lag_plot(apple_stock['Close'], lag=10, ax=ax)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fb382b105c0>
```

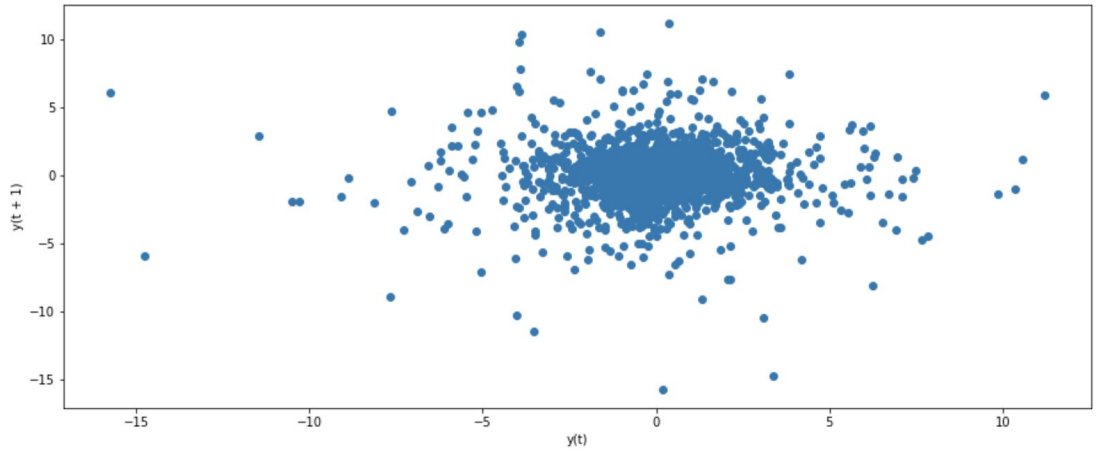


Let's remove trend and confirm we get white noise:

```
# Difference transform
diffed_series = np.diff(apple_stock['Close'], n=1)
```

```
fig, ax = plt.subplots(figsize=(15, 6))
lag_plot(pd.Series(diffed_series), ax=ax)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fb3830f06d8>
```



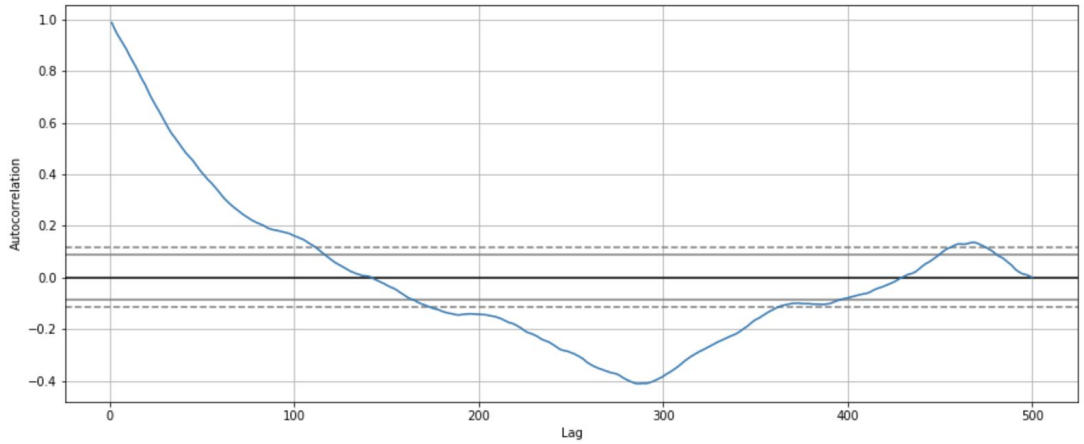
Differencing removes completely the correlation.

Other tools worth mentioning are provided by StatsModel library, such as plot_acf, to plot autocorrelation, and plot_pacf to plot partial autocorrelation. A partial autocorrelation is a summary of the relationship between an observation in a time series with observations at prior time steps with the relationships of intervening observations removed.
The partial autocorrelation at lag k is the correlation that results after removing the effect of any correlations due to the terms at shorter lags.

Let's explore these two in our Apple Close price:

```
fig, ax = plt.subplots(figsize=(15, 6))
autocorrelation_plot(apple_stock['Close'][:500], ax=ax)
```

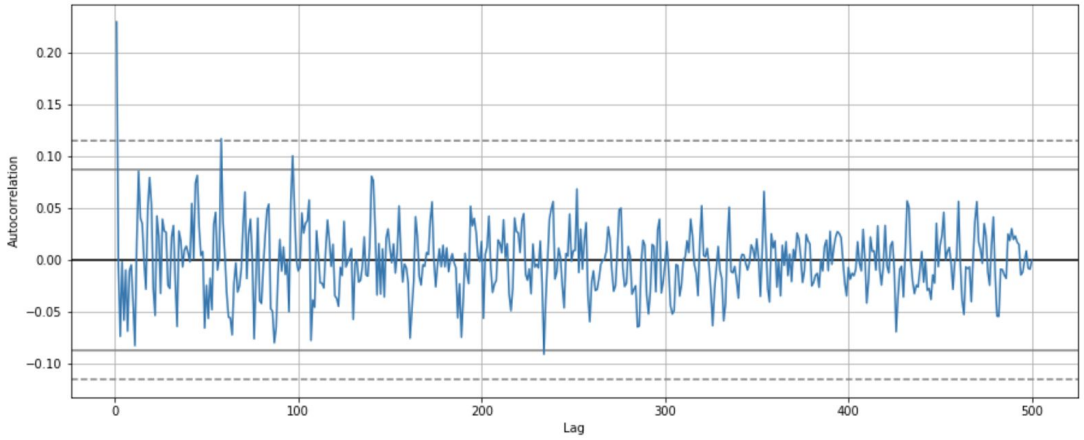
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fb382211748>
```



The interpretation of this chart is that within the first 500 points there is a positive correlation upto approximately 100 points. After that, we can see the line goes between the grey bands, means no correlation, according to Pearson's correlation coefficient. Around lag 200 upto 350 approximately there is negative correlation. If we try to plot differenced series, we no longer see autocorrelation:

```
fig, ax = plt.subplots(figsize=(15, 6))
autocorrelation_plot(pd.Series(diffed_series[:500]), ax=ax)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fb38258d438>
```



Algorithms and Techniques

Before starting implementing our models, we need to first re-frame our problem as Supervised Learning. This re-framing of our time series will allow us to access to the suite of standard linear and nonlinear machine learning algorithms on our problem. Let's remember that in Supervised Learning we have inputs variables (x) and target label (y) and we use an algorithm to learn the mapping function *f(x) = y*. We can convert our time series in a way that based on the last 5 days

{x_1, x_2, x_3, x_4, x_5} – these are our input variables – we try to predict the next 5 days {y_1, y_2, y_3, y_4, y_5} – these are our target labels. The following algorithm splits our sequence in such way:

```
def split_sequence(seq, look_back, n_outputs=1):
    '''
    split a sequence into samples.
    Example:
        seq = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
        look_back = 3
        n_outputs = 2

            X          y
        ------------------
        [1, 2, 3]   [4,  5]
        [2, 3, 4]   [5,  6]
        [3, 4, 5]   [6,  7]
        [4, 5, 6]   [7,  8]
        [6, 7, 8]   [9, 10]

    '''
    X, y = list(), list()
    seq_len = len(seq)

    for i in range(seq_len):
        # find the end of this pattern
        target_i = i + look_back

        # check if we are beyond the sequence
        if target_i + n_outputs > seq_len: break

        # gather input and output parts of the pattern
        seq_x, seq_y = seq[i:target_i], seq[target_i:target_i+n_outputs]

        X.append(seq_x)
        y.append(seq_y)

    return np.array(X), np.array(y)
```

In order to validate the performance of these models, we're gonna run a [walk-fordward validation] on the test set. We'll make the next 5 days prediction and walk forward as new prices arrive, that means, after one week of new prices, training the model with them, giving it the best opportunity to make good forecasts after 5 time step. We can evaluate our machine learning models under this assumption. This is how the model would be used in a production environment. The following algorithm will help us with this validation:

For ARIMA model:

```
def ARIMA_walk_forward_validation(train, test, order, size=1, steps=1, debug=True):
    '''
    Performs a walk-forward validation on ARIMA model
    params:
        train: Series - train set
        test: Series - test set
        order: Tuple - order parameter of ARIMA model
        size: Integer - amount of days we're gonna walk
        steps: Integer - how many days we're gonna forecast
        debug: Bool - prints debug prediction vs expected prices
    '''

    history = [x for x in train]
    pred = list()
    limit_range = len(test[:size])

    for t in range(0, limit_range, steps):
        model = ARIMA(history, order=order)
        model_fit = model.fit(disp=0) # trains the model with the new history
        output = model_fit.forecast(steps=steps) # make predictions
        yhat = output[0]
        pred = pred + yhat.tolist()
        obs = test[t:t+steps]
        history = history + obs.values.tolist()
        history = history[len(obs.values):] # shift to forget the oldest prices

        if debug == True:
            print('predicted={}, expected={}'.format(yhat, obs.values))

    return pred[:limit_range]
```

For NN models, it’s a bit different since the model is already trained, and needs inputs/outputs reshaping:

```
def NN_walk_forward_validation(model,
                               train, test,
                               size=1, look_back=1, n_outputs=1):
    '''
    Performs a walk-forward validation on a NN model
    params:
        model: NN model
        train: Series - train set
        test: Series - test set
        size: Integer - amount of days we're gonna walk
        look_back: Integer - amount of past days to forecast future ones
        n_outputs: Integer - amount of days predicted (output of predictor)
    '''

    past = train.reshape(-1,).copy()
    future = test.reshape(-1,)[:size]

    predictions = list()
    limit_range = len(future)

    for t in range(0, limit_range, n_outputs):
        x_input = past[-look_back:] # grab the last look_back days from the past
        x_input = x_input.reshape(1, look_back, 1)

        # predict the next n_outputs days
        y_hat = model.predict(x_input)
        predictions.append(y_hat.reshape(n_outputs,))

        # add the next real days to the past
        past = np.concatenate((past, future[t:t+n_outputs]))

        if len(future[t:t+n_outputs]) == n_outputs:
            X_batch = x_input
            y_batch = future[t:t+n_outputs].reshape(-1, n_outputs)

            # Time to re-train the model with the new non-seen days
            model.train_on_batch(X_batch, y_batch)

    return np.array(predictions).reshape(-1,)[:limit_range]
```

In order to solve this problem – predicting one week of prices – I'm gonna explore a few solutions: the widely used statistical model ARIMA, which we could considered as benchmark, to compare it with more recent state-of-the-art Neural Network solutions, such as Multilayer Perceptron (MLP), Convolutional Neural Networks (CNN), Long Short Term Memory (LSTM) and finally the most recent Generative Adversarial Networks (GAN).

The main purpose of this project is to shed light as to how to build a GAN architecture for time series prediction and see how it performs compared to the other solutions. As mentioned earlier on, there is a lot of literature about LSTM models for time series forecasting. There is little mention about using MLP, CNN, and even less using GAN. The curiosity arises as to how to build them for time series forecasting and how they perform when trying to predict the "unpredictable" stock market prices, and if they can beat the naive predictors (hopefully yes!).

Following some details about this algorithms:

ARIMA model

An ARIMA model is a class of statistical models for analyzing and forecasting time series data. It explicitly caters to a suite of standard structures in time series data, and as such provides a simple yet powerful method for making skillful time series forecasts. ARIMA is an acronym that stands for AutoRegressive Integrated Moving Average. It is a generalization of the simpler AutoRegressive Moving Average and adds the notion of integration. This acronym is descriptive, capturing the key aspects of the model itself. Briefly, they are:

- AR: Autoregression. A model that uses the dependent relationship between an observation and some number of lagged observations.
- I: Integrated. The use of differencing of raw observations (e.g. subtracting an observation from an observation at the previous time step) in order to make the time series stationary.
- MA: Moving Average. A model that uses the dependency between an observation and a residual error from a moving average model applied to lagged observations.

Note: refer to Implementation section for its architecture/configuration.

Multilayer Perceptron

It's the most basic type of Neural Network, but as I'll show, it can also be used for time series forecasting. It comprises a stack of Dense layers.

Note: refer to Implementation section for its architecture/implementation.

Convolutional Neural Network

Convolutional Neural Network models were developed for image classification, in which the model accepts a two-dimensional input representing an image's pixels and color channels, in a process called feature learning.

This same process can be applied to one-dimensional sequences of data. The model extracts features from sequences data and maps the internal features of the sequence. A 1D CNN is very effective for deriving features from a fixed-length segment of the overall dataset, where it is not so important where the feature is located in the segment.

1D Convolutional Neural Networks work well for:

- Analysis of a time series of sensor data.
- Analysis of signal data over a fixed-length period, for example, an audio recording.
- Natural Language Processing (NLP), although Recurrent Neural Networks which leverage Long Short Term Memory (LSTM) cells are more promising than CNN as they take into account the proximity of words to create trainable patterns.

Note: refer to Implementation section for its architecture/implementation.

Long Short Term Memory

Long Short Term Memory networks – usually just called "LSTMs" – are a special kind of RNN, capable of learning long-term dependencies. They were introduced by Hochreiter & Schmidhuber (1997), and were refined and popularized by many people. They work tremendously well on a large variety of problems, including also time series forecast, and are now widely used. There is an excellent article explaining LSTM Networks.

Note: refer to Implementation section for its architecture/implementation.

Generative Adversarial Network

A Generative Adversarial Network consists of two models, a Generator (G) and Discriminator (D). G uses random data – or not, as we'll see here – to generate data indistinguishable of, or extremely close to, the real data. Its purpose is to learn the distribution of the real data. Randomly, real or generated data, is fitted into D, which acts as a classifier and tries to understand whether the data is coming from G or is the real data. D estimates the probabilities of the incoming sample to the real dataset. Then, the losses from G and D are combined and propagated back through G. G's loss depends on both G and D. This is the step that helps G learn about the real data distribution. If G doesn't do a good job at generating realistic data (having the same distribution), it'll be very easy for D to distinguish generated from real data. Therefore D's loss will be very small and G's loss will be big. The process goes on until D is no longer able to distinguish generated from real data, since G has learned very well the distribution behind the real data and it's able to produce very realistic one.

Note: refer to Implementation section for its architecture/implementation.

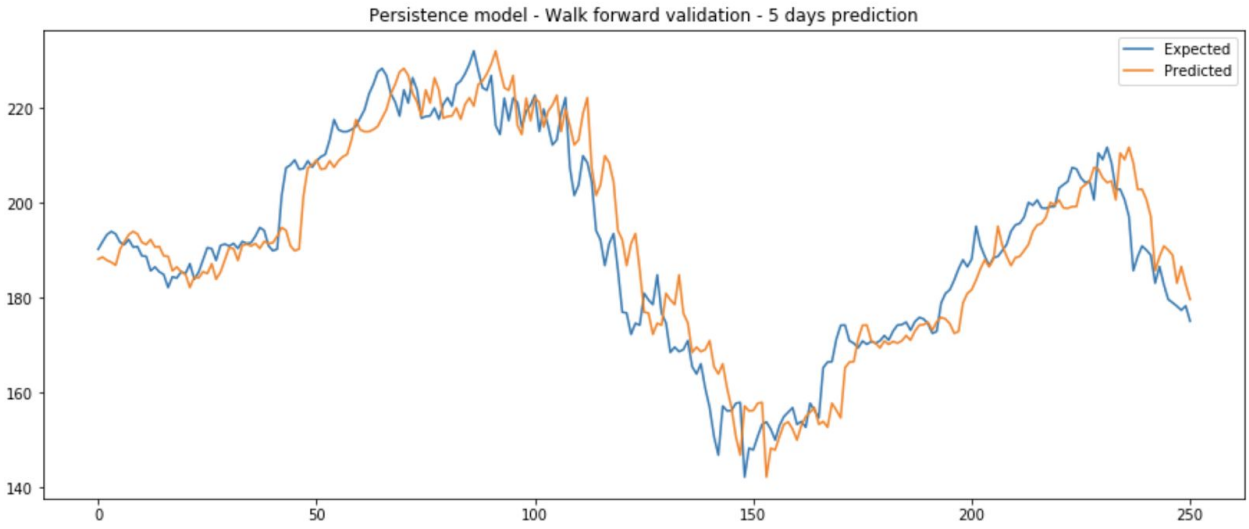
Benchmark

Establishing a baseline is essential on any time series forecasting problem. A baseline in performance gives you an idea of how well all other models will actually perform on your problem, a point of comparison. If a model achieves performance at or below the baseline, the technique should be fixed or abandoned. The technique used to generate a forecast to calculate the baseline performance must be easy to implement and naive of problem-specific details.

The most common baseline method for supervised machine learning is the Zero Rule algorithm. This algorithm predicts the majority class in the case of classification, or the average outcome in the case of regression. This could be used for time series, but does not respect the serial correlation structure in time series datasets. The equivalent technique for use with time series dataset is the persistence algorithm. This algorithm uses the value at the current time step (t) to predict the expected outcome at the next time step (t+1). I implemented this algorithm as followed:

```python
# The prediction is the last n_outputs of the window (naive prediction)
def persistence_model(x_input, n_outputs):
    return x_input[-n_outputs:]
```

Being *n_outputs* the days to predict. As expected, with this very naive predictor we just have the last days shifted:

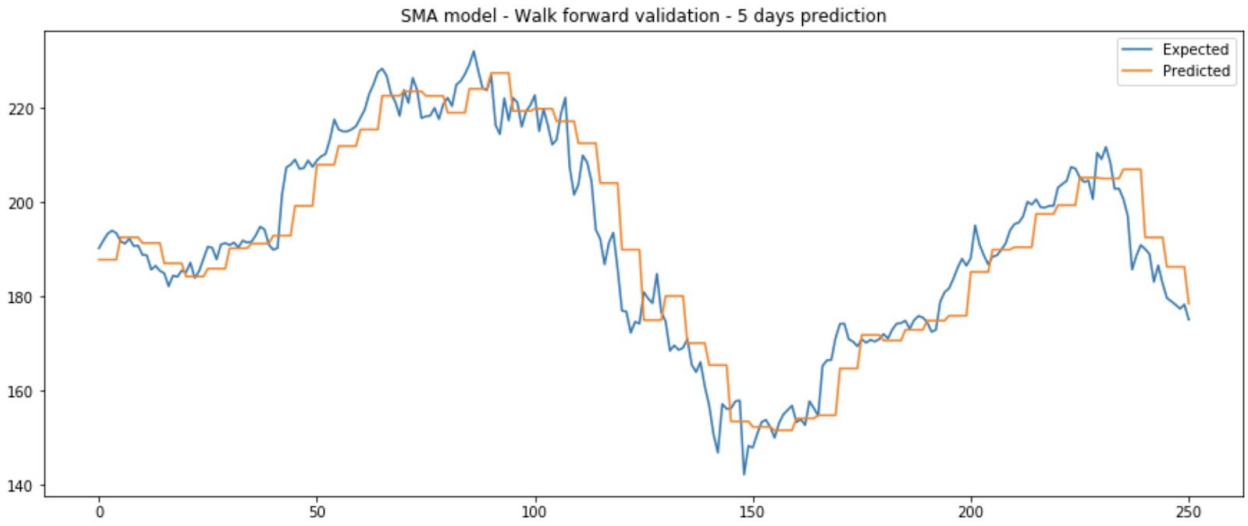



```
Persistence model[252 days, 5 days forecast]:

Forecast Bias: -0.203
MAE: 5.729
MSE: 54.819
RMSE: 7.404
MAPE: 3.050
```

The next benchmark I'm gonna use is a Simple Moving Average. A moving average (MA) is a widely used indicator in technical analysis that helps smooth out price action by filtering out the "noise" from random short-term price fluctuations. It is a trend-following, or lagging, indicator because it is based on past prices.

SMA is calculated by adding recent closing prices and then dividing that by the number of time periods in the calculation average. I'll be simply repeating the prediction *n_output* times

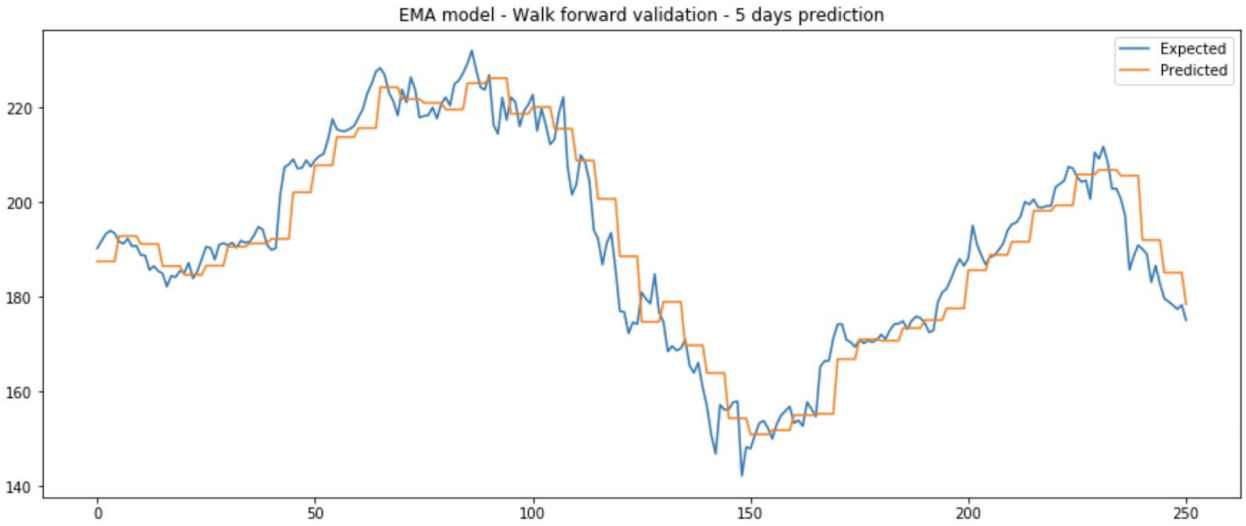
```python
# The prediction is the simple moving avarage repeated n_outputs times
def SMA_model(x_input, n_outputs):
    return np.repeat(np.mean(x_input), n_outputs)
```




```
SMA model[252 days, 5 days forecast]:

Forecast Bias: -0.166
MAE: 5.286
MSE: 47.474
RMSE: 6.890
MAPE: 2.816
```

The last benchmark will be the Exponential Moving Average. EMA is a type of moving average (MA) that places a greater weight and significance on the most recent data points. Same as before, I'll repeat the prediction *n_output* times

```python
# see https://sciencing.com/calculate-exponential-moving-averages-8221813.html
def EMA_model(x_input, look_back, n_outputs, prev_ema, prev_day=None):
    if prev_day is None:
        prev_ema = np.mean(x_input) # EMA = SMA
    else:
        prev_ema = prev_day[-1] # EMA = previous EMA

    multiplier = 2/(look_back + 1)
    x_input = (x_input[-1] - prev_ema) * multiplier + prev_ema
    return np.repeat(x_input, n_outputs)
```




```
EMA model[252 days, 5 days forecast]:

Forecast Bias: -0.074
MAE: 4.694
MSE: 37.944
RMSE: 6.160
MAPE: 2.504
```

Scores for these three benchmarks:

Benchmark	Bias	MAE	MSE	RMSE	MAPE
Persistence	-0.203	5.729	54.819	7.404	3.050
SMA	-0.166	5.286	47.474	6.890	2.816
EMA	-0.074	4.694	37.944	6.160	2.504

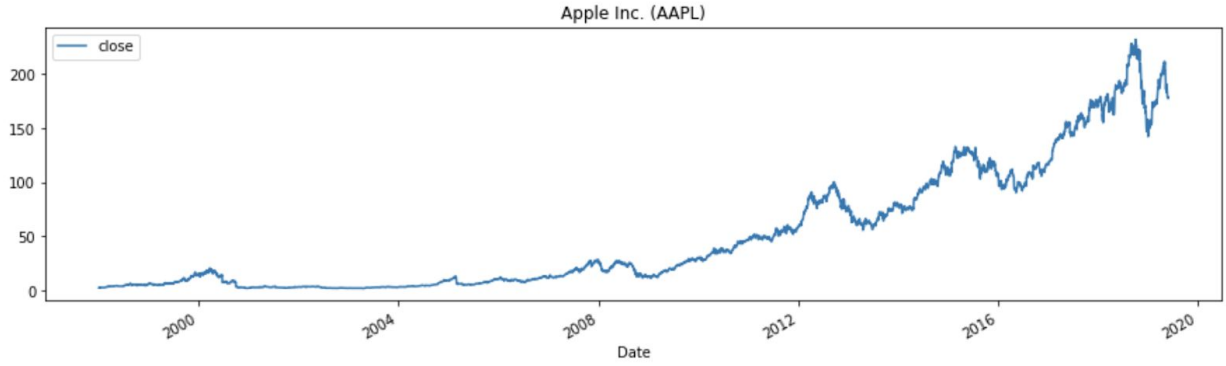
III. Methodology

Data Preprocessing

As seeing in the dataset presentation, there is a sudden drop on June 9th, 2014 because of a Stock Split. The solution to this is to divide by 7 all the prices before the split:

```
1 before_split = apple_price[apple_price.index < pd.to_datetime('2014-06-09')]
2 before_split['close'] = before_split['close'] / 7
3
4 after_split = apple_price[apple_price.index >= pd.to_datetime('2014-06-09')]
5 apple_price_corrected = pd.concat([before_split, after_split])
```

The result:



I obtained this dataset from Alpha Vantage. Later on I decided to go for the one provided by Yahoo! Finance, which has more prices back to 1980:

```
print('Total data points: {}'.format(len(apple_stock)))
print('From {} to {}'.format(apple_stock.index[0], apple_stock.index[-1]))
```

```
Total data points: 9699
From 1980-12-12 00:00:00 to 2019-05-31 00:00:00
```

I ran into some issues when I was trying to calculate some technical indicators. I was getting some null values. I found out the problem:

Do we have null values? and how many?

```
apple_stock.isnull().any()
```

```
Open      True
High      True
Low       True
Close     True
Volume    True
dtype: bool
```

```
apple_stock.isnull().sum()
```

```
Open      1
High      1
Low       1
Close     1
Volume    1
dtype: int64
```

```
apple_stock[apple_stock['Close'].isnull()]
```

Date	Open	High	Low	Close	Volume
1981-08-10	NaN	NaN	NaN	NaN	NaN

In order to fix it I decided to populate this missing prices with the average of the previous and next day:

```
from utils import get_range
```

```
get_range('1981-08-07', '1981-08-11', apple_stock)
```

Date	Open	High	Low	Close	Volume
1981-08-07	0.450893	0.453125	0.450893	0.450893	2301600.0
1981-08-10	NaN	NaN	NaN	NaN	NaN
1981-08-11	0.441964	0.441964	0.437500	0.437500	17864000.0

Let's populate this value with mean of the other two

```
apple_stock.loc['1981-08-10'] = \
    (apple_stock.loc['1981-08-07'] + apple_stock.loc['1981-08-11']) / 2
```

```
get_range('1981-08-07', '1981-08-11', apple_stock)
```

Date	Open	High	Low	Close	Volume
1981-08-07	0.450893	0.453125	0.450893	0.450893	2301600.0
1981-08-10	0.446429	0.447545	0.444196	0.444196	10082800.0
1981-08-11	0.441964	0.441964	0.437500	0.437500	17864000.0

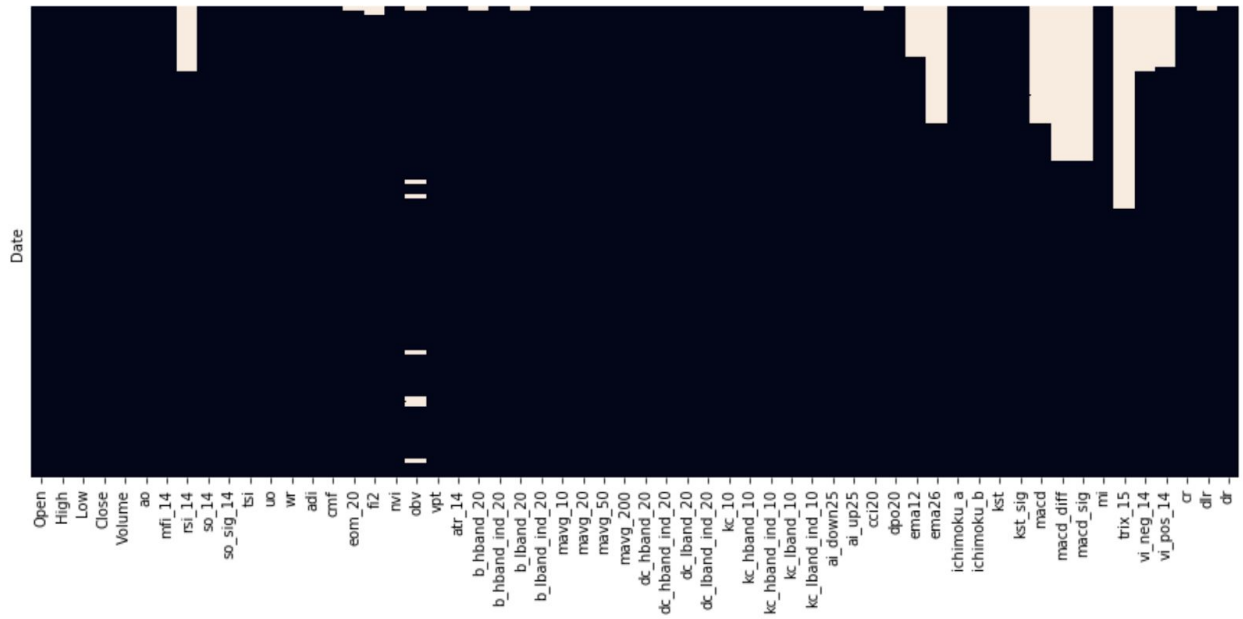
The next thing I'm gonna do is to add more features to our dataset. Some technical indicators that could help the model to predict prices. A lot of investors follow technical indicators to try to predict market movements. Since I'm not an expert in stock market, I'm gonna be using this library, ta, that will help me to add some good indicators.

Note: refer to Jupyter notebooks to see how these indicators were added.

After adding these features, I noticed some null values:

```
fig, ax = plt.subplots(figsize=(15, 6))
sns.heatmap(featured_series.isnull().head(100), cbar=False, yticklabels=False)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7feaeb86ec50>
```

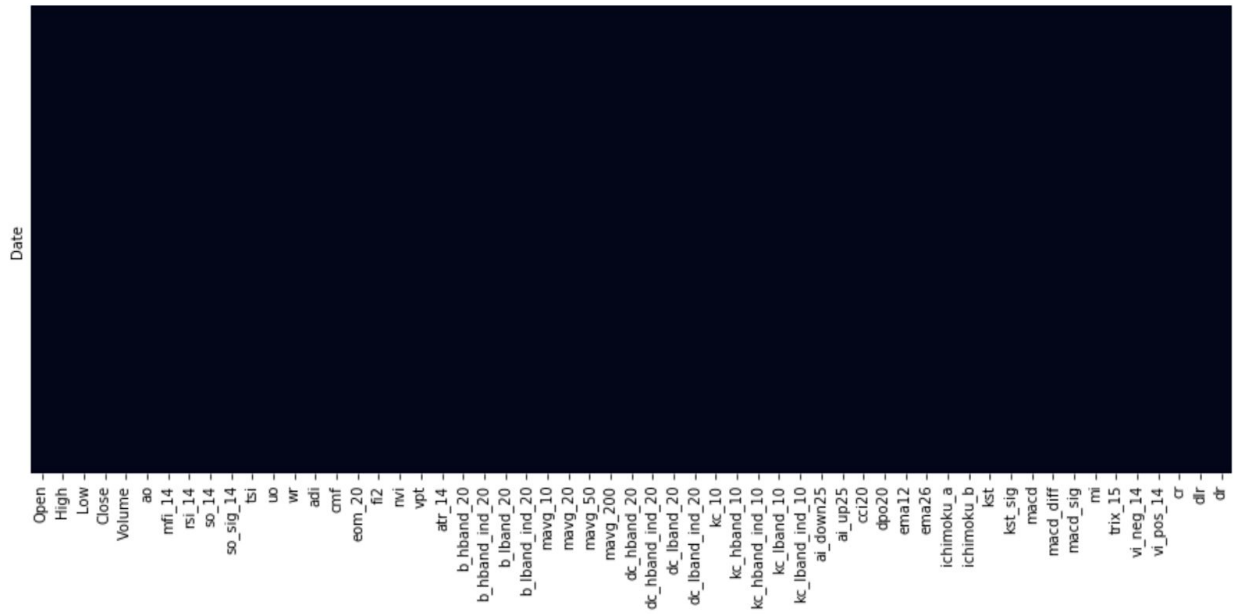


obv (On-balance volume) had lots of random null values across the dataset. I decided to drop this indicator. For the other null values, we can drop the rows without discontinuing the series because these are all at the beginning. They are used to calculate next values in the series. They have null values because they need previous ones that simply don't exist.

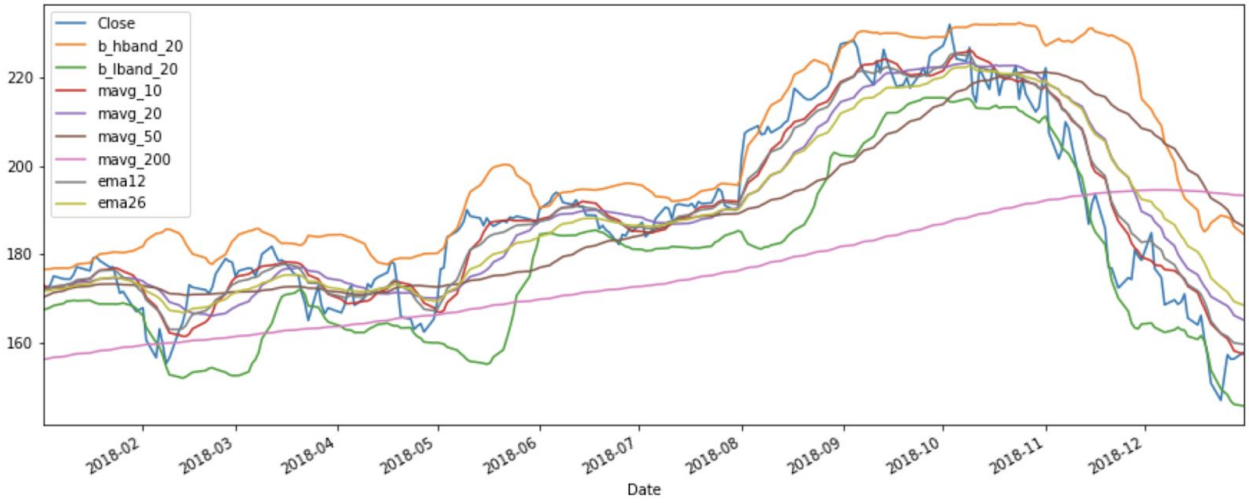
```
featured_series.dropna(inplace=True)
```

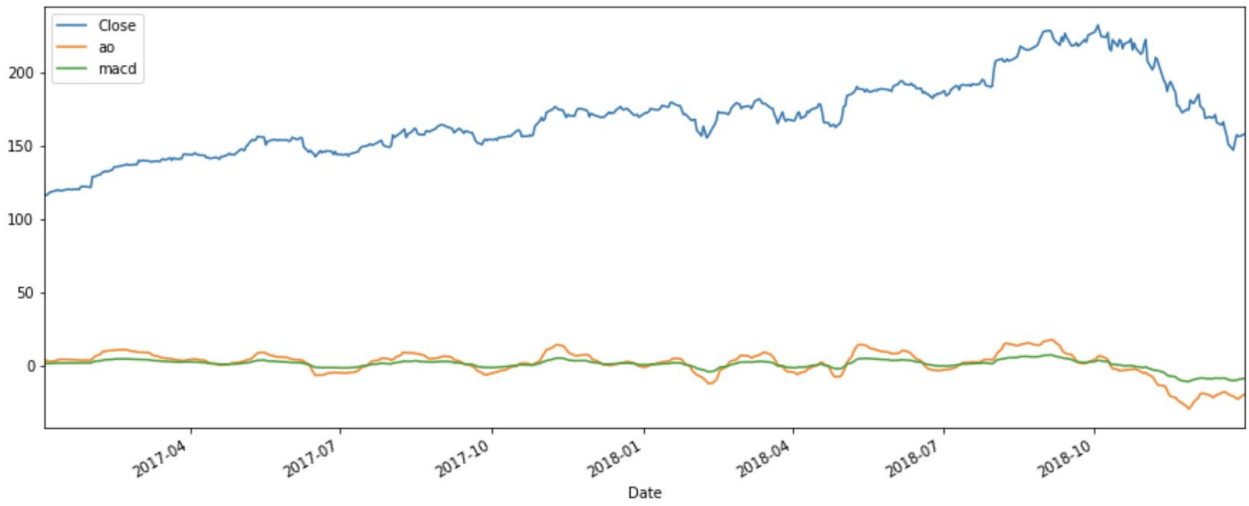
```
fig, ax = plt.subplots(figsize=(15, 6))
sns.heatmap(featured_series.isnull(), cbar=False, yticklabels=False)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7feaeb86e5c0>
```

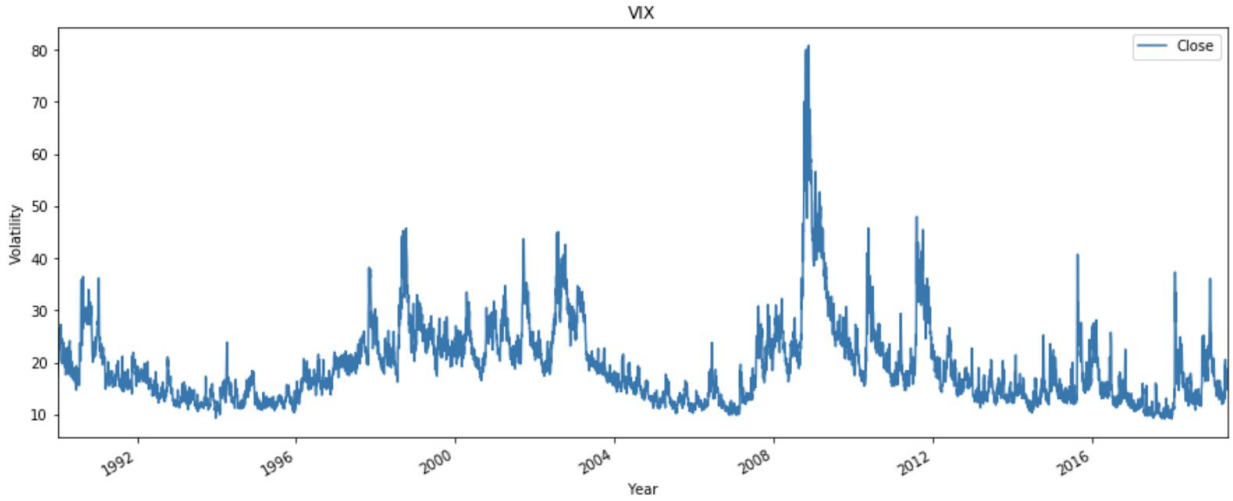


We got rid of all the nulls. Let's have a look at some of the most used technical indicators along with the Close price:

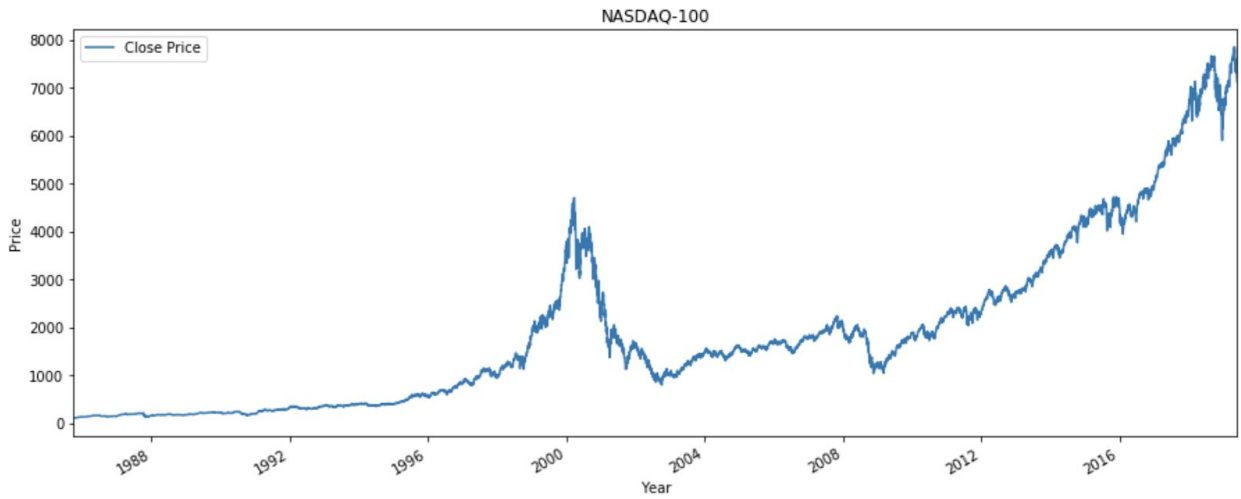




Other features I added were, [CBOE Volatility Index] and [Nasdaq 100 Index]. VIX is a popular measure of the stock market's expectation of 30-day forward-looking volatility derived from the price inputs of the S&P 500 index options, it provides a measure of market risk and investors' sentiments:



Nasdaq 100 is a basket of the 100 largest, most actively traded U.S companies listed on the Nasdaq stock exchange. The index includes companies from various industries except for the financial industry, like commercial and investment banks:



I finally ended up with 60 features in our dataset. At this point it's important to consider whether they're all good indicatives of price movements. Let's find out how important these features are to predict prices. We're gonna make use of ensembles of decision tree methods like gradient boosting, XGBoost. XGBoost provides a score that indicates how useful or valuable each feature was in the construction of the boosted decision trees within the model. The more an attribute is used to make key decisions with decision trees, the higher its relative importance. Following is the code I wrote to train this regressor in order to get the importance score:

```
def train_xgbregressor(series, look_ahead=5):
    '''
    XGBRegressor has some limitations that prevents us from using
    multiple dimensions inputs and outputs. But we can still use it
    to predict the price x days ahead based on one day of features
    '''
    X = series[:-look_ahead]
    y = series['Close'][look_ahead:]

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, shuffle=False)

    regressor = XGBRegressor()
    regressor.fit(X_train, y_train, eval_set=[(X_test, y_test)], verbose=False)

    return regressor
```

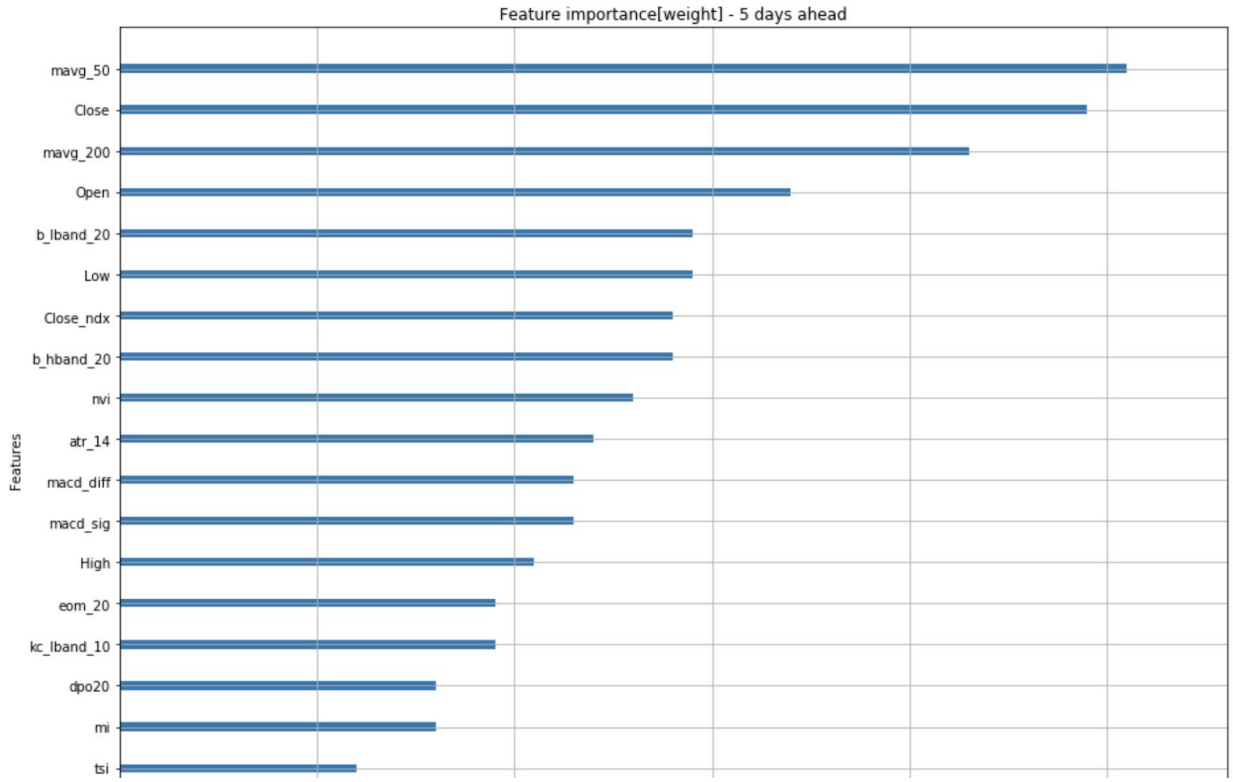
Training the regressor gets us the most important features:

```
# let's see the results using `gain` as type of importance
plot_feature_importance(regressor,
                        title='Feature importance[gain] - 5 days ahead',
                        importance_type='gain')
```

Something to detail here is the type of importance. There different types, but I'm using only two, Gain and Weight:

- "Gain" implies the relative contribution of the corresponding feature to the model calculated by taking each feature's contribution for each tree in the model. A higher value of this metric when compared to another feature implies it is more important for generating a prediction.
- "Weight" is the percentage representing the relative number of times a particular feature occurs in the trees of the model.

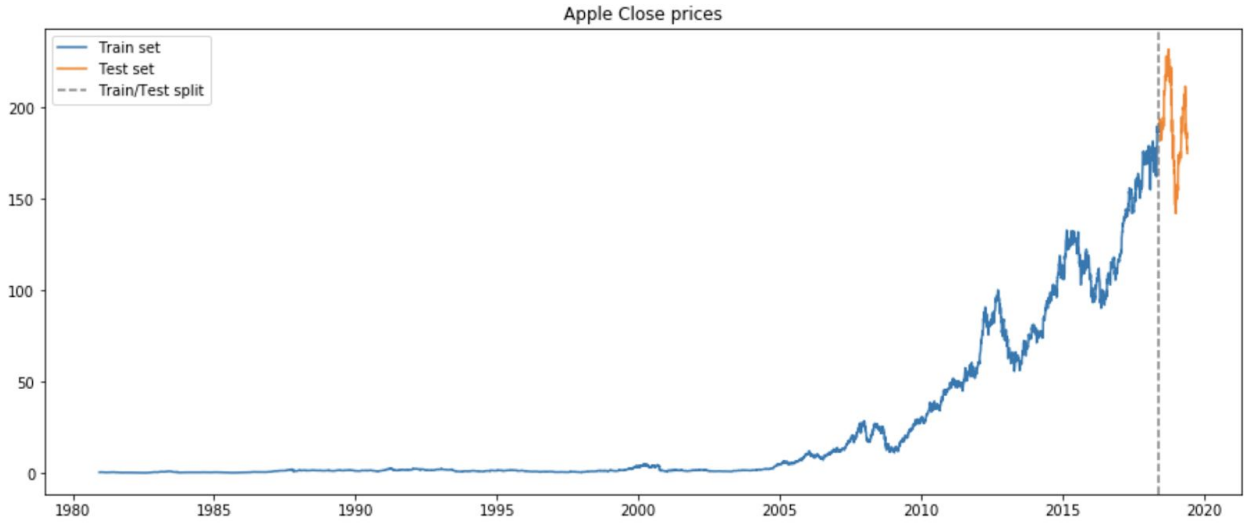
```
# let's see the results using `weight` as type of importance
plot_feature_importance(regressor, title='Feature importance[weight] - 5 days ahead', importance_type='weight
```



As expected, Open, High, Low and Close have all high score since they are quite similar to Close price and they don't change much in 5 days.

I also ran some feature correlation, but most of the were highly correlated (they are calculated based on Open, High, Low, Close price), and decided to keep the ones that traders use the most. Please, refer to Jupyter notebook for feature correlation.

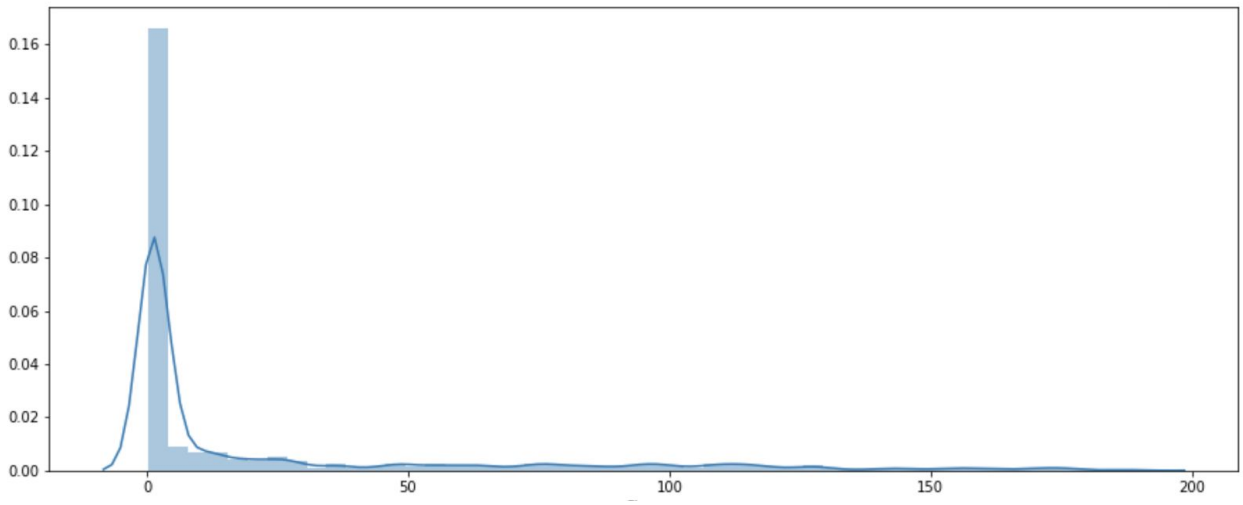
As a common task in many Machine Learning process, we need to split our dataset into train (, validation) and test set, in order to train our models and test them. For this time series I'm gonna make the split on May 31st, 2018, so our train set will be until this date, and our test set will have exactly one year of prices, until May 31st, 2019:



It’s important to explore different transformations to make the learning easier for our Machine Learning algorithms. Transformations are used to stabilize the non-constant variance of a series. Common transformation methods include power transform, square root, and log transform. What kind of distribution do we have here?

```
train.describe()
```

```
count    9448.000000
mean       23.597855
std        41.856155
min         0.196429
25%         1.040179
50%         1.643214
75%        23.588572
max       190.039993
Name: Close, dtype: float64
```



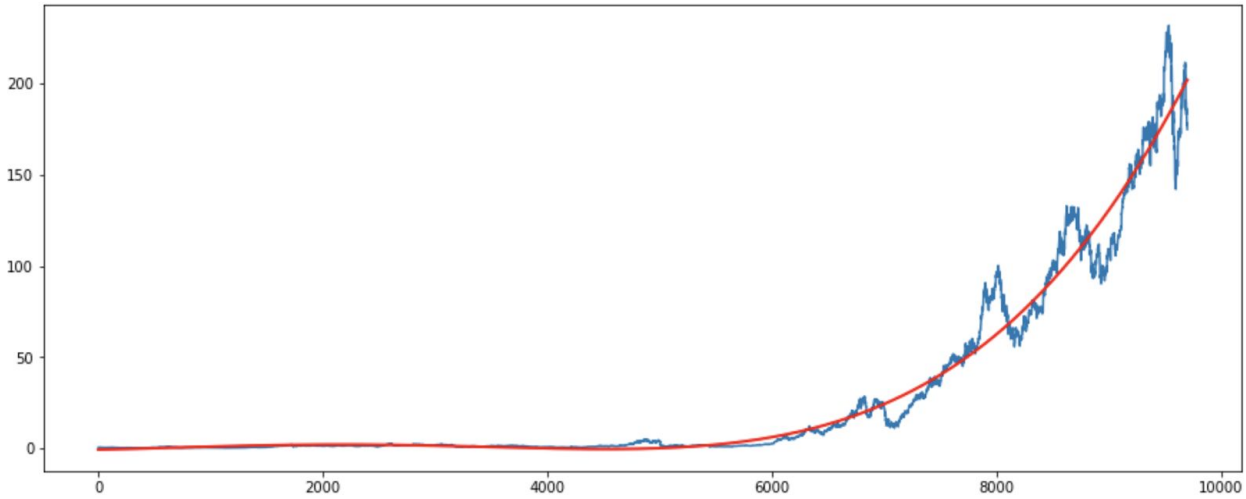
As seen before, the distribution of our feature is highly right skewed. By looking at the time series we can spot a polynomial trend. After some experiments I figured out the best polynomial degree that fits perfectly the trend:

```
from numpy import polyfit
```

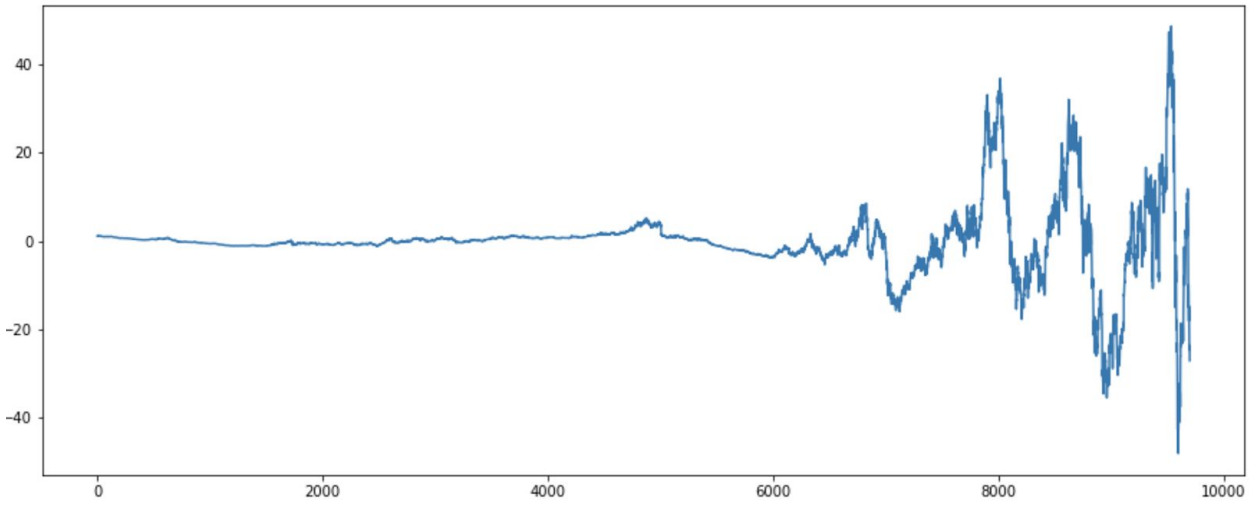
```
degree = 4
X1d = X.flatten()
y1d = y.flatten()
coef = polyfit(X1d, y1d, degree)
print('Coefficients: %s' % coef)
```

```
Coefficients: [ 7.54975843e-14 -6.34433691e-10  1.09347483e-06  1.11302440e-03
 -6.38839677e-01]
```

```
# create curve
curve = list()
for i in range(len(X)):
    value = coef[-1]
    for d in range(degree):
        value += X[i]**(degree-d) * coef[d]
    curve.append(value)

# plot curve over original data
plt.subplots(figsize=(15, 6))
plt.plot(y)
plt.plot(curve, color='red', linewidth=2)
```



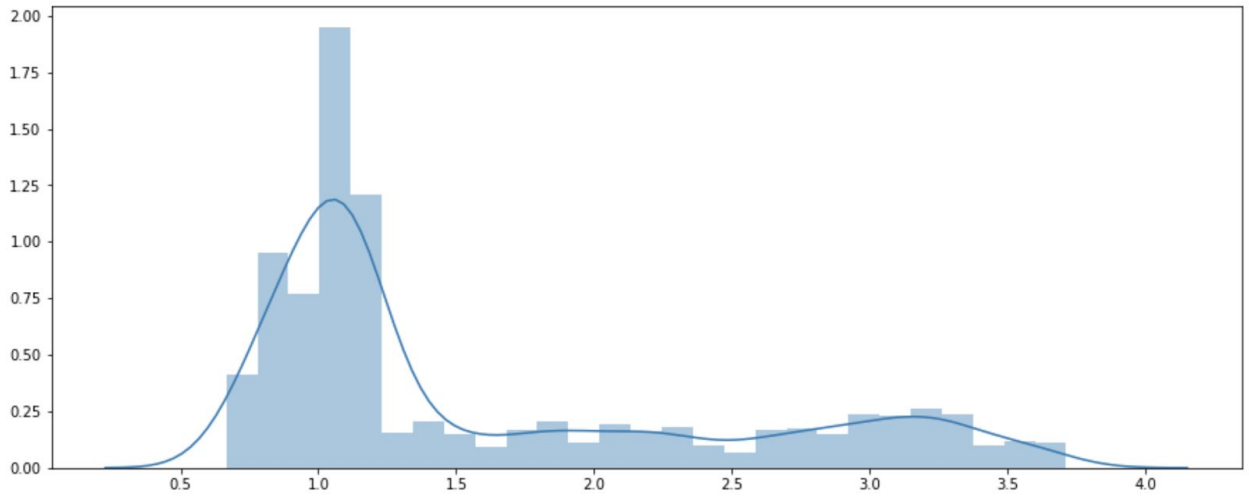
Removing it could help to make it more gaussian-like:




```
power_1_4_series = np.power(train, 1/4)
```

```
power_1_4_series.describe()
```

```
count    9448.000000
mean        1.602472
std         0.861220
min         0.665735
25%         1.009897
50%         1.132201
75%         2.203816
max         3.712883
Name: Close, dtype: float64
```

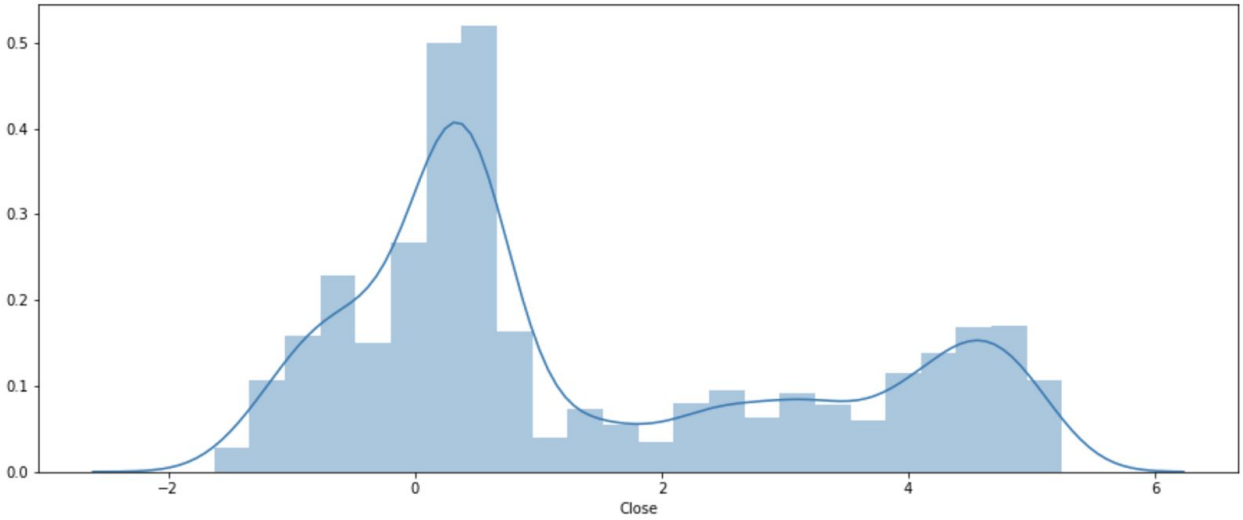


Not perfect. Let’s try other power transformations:

```
log_series = np.log(train)
```

```
log_series.describe()
```

```
count    9448.000000
mean        1.385558
std         1.929891
min        -1.627454
25%         0.039393
50%         0.496654
75%         3.160762
max         5.247235
Name: Close, dtype: float64
```



Almost there. Let's make use of Box-Cox test to find the best transformation.

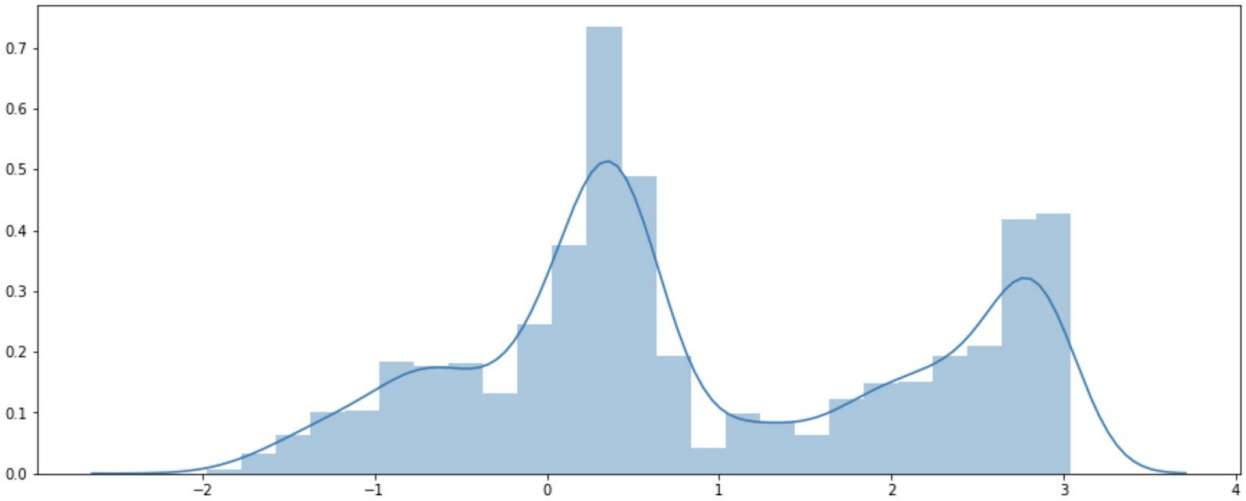
```
from scipy.stats import boxcox
```

```
boxcox_series, lmbda = boxcox(train.values)
print('Best lmbda for Box-Cox test:', lmbda)
```

```
Best lmbda for Box-Cox test: -0.23066752434999835
```

```
pd.Series(boxcox_series).describe()
```

```
count    9448.000000
mean        0.893943
std         1.306610
min        -1.975028
25%         0.039214
50%         0.469261
75%         2.244139
max         3.042960
dtype: float64
```



Excellent. Looks quite good. Almost ready to be fed into our models. But before that, let's make the last transformation, feature scaling or normalization. The goal of normalization is to change the values of numeric columns in the dataset to a common scale, without distorting differences in the ranges of values. This will help our model to be more efficient:

```
from sklearn.preprocessing import MinMaxScaler
```

```
scaler = MinMaxScaler()
```

```
scaled_train = scaler.fit_transform(transformed_train.reshape(-1, 1))
```

```
scaled_test = scaler.transform(transformed_test.reshape(-1, 1))
```

```
pd.Series(scaled_train.reshape(-1,)).describe()
```

```
count    9448.000000
mean        0.571737
std         0.260385
min         0.000000
25%         0.401404
50%         0.487105
75%         0.840809
max         1.000000
dtype: float64
```

```
pd.Series(scaled_test.reshape(-1,)).describe()
```

```
count    251.000000
mean       0.999986
std        0.006823
min        0.982179
25%        0.995028
50%        1.000271
75%        1.005553
max        1.011600
dtype: float64
```

Implementation

As mentioned in Algorithms and Techniques, we're gonna be using a statistical model, ARIMA, and different NN architectures to solve this problem and see which one perform the best. These are the global parameters for all the models, including benchmarks:

```
# Model parameters
look_back = 5 # days window look back
n_features = 1 # our only feature will be Close price
n_outputs = 5 # days forecast
batch_size = 32 # for NN, batch size before updating weights
n_epochs = 100 # for NN, number of training epochs
```

The idea is to predict one week of prices, *n_outputs = 5*, based on the previous week, *look_back = 5*. I'll be using, initially, only the Close price as the only feature, *n_features = 1*. Following are their implementation, basic architectures and configurations:

ARIMA model

There was no need to implement this model since it's already provided by StatsModel library

```
order = best_order
ARIMAmodel = ARIMA(train, order=order)
```

```
ARIMAmodel_fit = ARIMAmodel.fit(disp=0)
```

```
print(ARIMAmodel_fit.summary())
```

```
                             ARIMA Model Results
==============================================================================
Dep. Variable:                D.Close   No. Observations:                 3879
Model:                 ARIMA(5, 1, 2)   Log Likelihood               -5933.330
Method:                       css-mle   S.D. of innovations              1.117
Date:                Thu, 11 Jul 2019   AIC                          11884.660
Time:                        09:23:46   BIC                          11941.030
Sample:                             1   HQIC                         11904.673

==================================================================================
                     coef    std err          z      P>|z|      [0.025      0.975]
----------------------------------------------------------------------------------
const              0.0479      0.018      2.621      0.009       0.012       0.084
ar.L1.D.Close     -0.6192      0.050    -12.347      0.000      -0.717      -0.521
ar.L2.D.Close     -0.8686      0.043    -20.010      0.000      -0.954      -0.783
ar.L3.D.Close      0.0179      0.024      0.762      0.446      -0.028       0.064
ar.L4.D.Close      0.0034      0.021      0.168      0.867      -0.037       0.044
ar.L5.D.Close     -0.0264      0.018     -1.481      0.139      -0.061       0.009
ma.L1.D.Close      0.6605      0.048     13.850      0.000       0.567       0.754
ma.L2.D.Close      0.8782      0.040     22.114      0.000       0.800       0.956
                                    Roots
=============================================================================
                  Real          Imaginary           Modulus         Frequency
-----------------------------------------------------------------------------
AR.1           -0.3568           -0.9826j            1.0454           -0.3054
AR.2           -0.3568           +0.9826j            1.0454            0.3054
AR.3           -3.1104           -0.0000j            3.1104           -0.5000
AR.4            1.9772           -2.6872j            3.3363           -0.1490
AR.5            1.9772           +2.6872j            3.3363            0.1490
MA.1           -0.3761           -0.9986j            1.0671           -0.3073
MA.2           -0.3761           +0.9986j            1.0671            0.3073
-----------------------------------------------------------------------------
```

In order to obtain the order parameter (p, i, d) for ARIMA model, I used a search grid, fixing *p*, lagged observations, to *look_back = 5*:

```
best_aic = np.inf
best_order = None

p = look_back # we fix the autoregressive p param (lags) to look back window
d_range = range(2) # [0, 1]
q_range = range(3) # [0, 1, 2]

for d in d_range:
    for q in q_range:
        try:
            order = (p, d, q)
            tmp_model = ARIMA(train, order=order).fit(disp=0)
            tmp_aic = tmp_model.aic
            if tmp_aic < best_aic:
                best_aic = tmp_aic
                best_order = order
                best_model = tmp_model
                print('best aic: {:6.2f} | order: {}'.format(best_aic, best_order))
        except: continue


print('Final aic: {:6.2f} | order: {}'.format(best_aic, best_order))
```

```
best aic: 11921.62 | order: (5, 0, 0)
best aic: 11902.81 | order: (5, 1, 0)
best aic: 11902.09 | order: (5, 1, 1)
best aic: 11884.66 | order: (5, 1, 2)
Final aic: 11884.66 | order: (5, 1, 2)
```

Multilayer Perceptron

```
def build_MLP(look_back, n_features, n_outputs, optimizer='adam'):
    model = Sequential()

    model.add(Dense(64,
                    activation='relu',
                    input_shape=(look_back, n_features)))
    model.add(Flatten())
    model.add(Dense(n_outputs))

    model.compile(optimizer=optimizer, loss='mean_squared_error')

    return model
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_26 (Dense)             (None, 5, 64)             128
_________________________________________________________________
flatten_12 (Flatten)         (None, 320)               0
_________________________________________________________________
dense_27 (Dense)             (None, 5)                 1605
=================================================================
Total params: 1,733
Trainable params: 1,733
Non-trainable params: 0
_________________________________________________________________
```

Convolutional Neural Network

```
def build_CNN(look_back, n_features, n_outputs, optimizer='adam'):
    model = Sequential()

    model.add(Conv1D(64,
                     kernel_size=4,
                     activation='relu',
                     input_shape=(look_back, n_features)))
    model.add(Flatten())
    model.add(Dense(n_outputs))

    model.compile(optimizer=optimizer, loss='mean_squared_error')

    return model
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv1d_2 (Conv1D)            (None, 2, 64)             320
_________________________________________________________________
flatten_2 (Flatten)          (None, 128)               0
_________________________________________________________________
dense_2 (Dense)              (None, 5)                 645
=================================================================
Total params: 965
Trainable params: 965
Non-trainable params: 0
_________________________________________________________________
```

Something to keep in mind here is that *padding='valid'*, or no padding, plus *kernel_size=4*, which is the reason why output shape of the first Conv1D layer is not *(None, 5, 64)* but *(None, 2, 64)*.

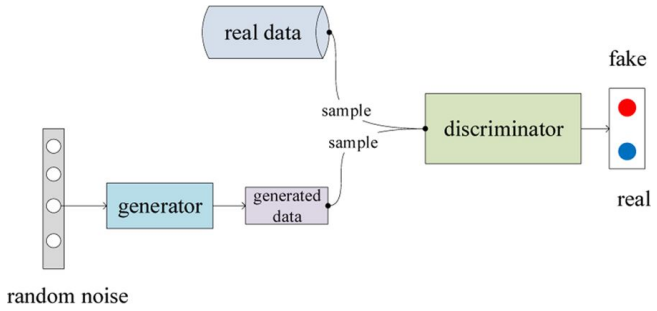
Long Short Term Memory

```
def build_LSTM(look_back, n_features, n_outputs, optimizer='adam'):
    model = Sequential()

    model.add(LSTM(50,
                   activation='relu',
                   input_shape=(look_back, n_features)))
    model.add(Dense(n_outputs))

    model.compile(optimizer=optimizer, loss='mean_squared_error')

    return model
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_4 (LSTM)                (None, 50)                10400
_________________________________________________________________
dense_4 (Dense)              (None, 5)                 255
=================================================================
Total params: 10,655
Trainable params: 10,655
Non-trainable params: 0
_________________________________________________________________
```

Generative Adversarial Network

The GAN model architecture involves two sub-models: a generator model for generating new examples and a discriminator model for classifying whether generated examples are real (from the domain) or fake (generated by the generator model):



Architecture of a GAN

Let’s have a look at the basic implementation of the different models I’m gonna be using:

```
def build_generator(look_back, n_features=1, n_outputs=1):
    model = Sequential()

    model.add(LSTM(50, input_shape=(look_back, n_features)))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dense(n_outputs))

    print('Generator summary:')
    model.summary()

    return model
```

```
Generator summary:
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_3 (LSTM)                (None, 50)                10400
_________________________________________________________________
leaky_re_lu_6 (LeakyReLU)    (None, 50)                0
_________________________________________________________________
dense_6 (Dense)              (None, 5)                 255
=================================================================
Total params: 10,655
Trainable params: 10,655
Non-trainable params: 0
_________________________________________________________________
```

```python
def build_discriminator(look_back, n_features=1, n_outputs=1, optimizer=Adam()):
    features_input = Input((look_back, n_features))
    target_input = Input((n_outputs, 1))

    x = Conv1D(64,
               kernel_size=5,
               padding='same')(features_input)
    x = LeakyReLU(alpha=0.2)(x)
    x = Flatten()(x)

    y = Conv1D(64,
               kernel_size=5,
               padding='same')(target_input)
    y = LeakyReLU(alpha=0.2)(y)
    y = Flatten()(y)

    xy = concatenate([x, y])
    xy = Dense(32)(xy)
    xy = LeakyReLU(alpha=0.2)(xy)

    valid = Dense(1, activation='sigmoid')(xy)

    model = Model([features_input, target_input], valid);

    model.compile(loss='binary_crossentropy', optimizer=optimizer)

    print('Discriminator summary:')
    model.summary()

    return model
```

```
Discriminator summary:
_________________________________________________________________________________________
Layer (type)                 Output Shape              Param #     Connected to
=========================================================================================
input_10 (InputLayer)        (None, 5, 1)              0
_________________________________________________________________________________________
input_11 (InputLayer)        (None, 5, 1)              0
_________________________________________________________________________________________
conv1d_7 (Conv1D)            (None, 5, 64)             384         input_10[0][0]
_________________________________________________________________________________________
conv1d_8 (Conv1D)            (None, 5, 64)             384         input_11[0][0]
_________________________________________________________________________________________
leaky_re_lu_13 (LeakyReLU)   (None, 5, 64)             0           conv1d_7[0][0]
_________________________________________________________________________________________
leaky_re_lu_14 (LeakyReLU)   (None, 5, 64)             0           conv1d_8[0][0]
_________________________________________________________________________________________
flatten_7 (Flatten)          (None, 320)               0           leaky_re_lu_13[0][0]
_________________________________________________________________________________________
flatten_8 (Flatten)          (None, 320)               0           leaky_re_lu_14[0][0]
_________________________________________________________________________________________
concatenate_4 (Concatenate)  (None, 640)               0           flatten_7[0][0]
                                                                   flatten_8[0][0]
_________________________________________________________________________________________
dense_10 (Dense)             (None, 32)                20512       concatenate_4[0][0]
_________________________________________________________________________________________
leaky_re_lu_15 (LeakyReLU)   (None, 32)                0           dense_10[0][0]
_________________________________________________________________________________________
dense_11 (Dense)             (None, 1)                 33          leaky_re_lu_15[0][0]
=========================================================================================
Total params: 21,313
Trainable params: 21,313
Non-trainable params: 0
_________________________________________________________________________________________
```

Note: *LeakyReLU* is recommended here, but at this point I haven't yet tried to optimize anything.

We stop here for a moment to understand the discriminator architecture. It has two inputs. First one is the inputs features. These are the look back days (1st dimension) and the list of features (2nd dimension). Initially I'm gonna be using only the Close price, but later on I'll add the

technical indicators. The second input is the target input, the prediction from the generator. It has *shape=(n_outputs, 1)*. First dimension is the amount of days predicted. The discriminator will take these two inputs, do some convolutions to extract information, and then concatenate the result to later on be passed into a Dense layer and do binary classification of real or fake using sigmoid activation function.

```
def build_adversarial(look_back,
                      n_features=1, n_outputs=1,
                      dis_optimizer=Adam(), adv_optimizer=Adam()):

    discriminator = build_discriminator(look_back, n_features, n_outputs, optimizer=dis_optimizer)
    generator = build_generator(look_back, n_features, n_outputs)

    seq = Sequential()
    seq.add(generator)

    gen_input = Input((look_back, n_features))
    gen_output = seq(gen_input)
    gen_output = Reshape((n_outputs, 1))(gen_output)

    valid = discriminator([gen_input, gen_output])

    model = Model(gen_input, valid)

    discriminator.trainable = False # We need to freeze the discriminator's weights
    model.compile(loss='binary_crossentropy', optimizer=adv_optimizer)

    print('Adversarial summary:')
    model.summary()

    return model, discriminator, generator
```

```
Adversarial summary:
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_12 (InputLayer)        (None, 5, 1)              0
_________________________________________________________________
sequential_8 (Sequential)    (None, 5)                 10655
_________________________________________________________________
reshape_4 (Reshape)          (None, 5, 1)              0
_________________________________________________________________
model_7 (Model)              (None, 1)                 21313
=================================================================
Total params: 31,968
Trainable params: 10,655
Non-trainable params: 21,313
_________________________________________________________________
```

The adversarial will put together the two other models to compete against each other. It's important to freeze the weights of the discriminator before compiling the adversarial model to make sure we're not changing its weights when updating the weights of the adversarial model. This instead will just update the weights of the generator, which is the one we're interested in.

A GAN is an architecture for training the generative model, which is the one we are interested in. The discriminator model is used to assist in the training of the generator. We need to do this training manually, as followed:

```python
def get_batch(X, y, batch_idx, batch_size):
    '''
    Returns batches sequentially based on
    the current batch_idx in the loop
    '''

    X_batch = X[batch_idx:batch_idx+batch_size]
    y_batch = y[batch_idx:batch_idx+batch_size]

    return X_batch, y_batch
```

```python
def train_models(adv, dis, X_batch, y_batch, y_pred, n_outputs, n_features):
    '''
    Trains discriminator and adversarial (generator) models
    It's recommended to train the discriminator first with a
    batch of real data and then train with a batch of fake data
    '''

    X_real = y_batch.reshape((y_batch.shape[0], n_outputs, n_features))
    X_fake = y_pred.reshape((y_pred.shape[0], n_outputs, n_features))

    y_real = np.ones((y_batch.shape[0], 1))
    y_fake = np.zeros((y_pred.shape[0], 1))

    # train discriminator:
    d_loss_real = dis.train_on_batch(X_real, y_real)
    d_loss_fake = dis.train_on_batch(X_fake, y_fake)
    d_loss = np.add(d_loss_real, d_loss_fake) / 2

    # train generator
    g_loss = adv.train_on_batch(X_batch, y_real)

    return d_loss_real, d_loss_fake, d_loss, g_loss
```

train_model function is the most important one. Here it's recommended by most of the papers that we train the discriminator with a batch of real ones and then with a batch of fake ones produced by the generator. After this training we train the whole adversarial to update the generator's weights. We return the losses to keep track of how the training is going.

```
def train_GAN(X, y,
              adv, dis, gen,
              n_epochs=100, batch_size=100,
              look_back=3, n_features=1, n_outputs=1):

    data_len = len(X)
    hist_d_loss_real = []; hist_d_loss_fake = []
    hist_d_loss = []; hist_g_loss = []

    for epoch in range(n_epochs):
        start = time.time()
        for batch_idx in range(0, data_len, batch_size):

            X_batch, y_batch = get_batch(X, y, batch_idx, batch_size)

            # laten space from a Gaussian distribution
            # noise = np.random.normal(0, 1, X_batch.shape)

            y_pred = gen.predict(X_batch)

            d_loss_real, d_loss_fake, d_loss, g_loss = train_models(adv, dis,
                                                                    X_batch, y_batch, y_pred,
                                                                    n_outputs, n_features)
        end = time.time()

        print ("Epoch %d/%d [D loss: %f] [G loss: %f] | %ds" % (epoch+1, n_epochs, d_loss, g_loss, end-start))

        hist_d_loss_real.append(d_loss_real); hist_d_loss_fake.append(d_loss_fake)
        hist_d_loss.append(d_loss); hist_g_loss.append(g_loss)

    return hist_d_loss_real, hist_d_loss_fake, hist_d_loss, hist_g_loss
```

The implementation of this architecture has been the most difficult one. It's based on what's written on different papers and articles, but all the implementations out there are for some kind image creation. No clear examples exist about time series prediction. I built tons of different architectures, and change hundreds of different parameters. All of them giving me different performances. None of them a good one, worse that the benchmarks. Unfortunately all my trainings seemed to be falling into what's called Mode Collapse. Mode collapse is when the generator generates a limited diversity of samples, or even the same sample, regardless of the input. Mode collapse is one of the hardest problems to solve in GAN.

Refinement

The main goal of this project was to build a GAN architecture for time series forecasting, applied to Stock Market prices, and able to predict one week of prices. Even though I got really bad results using GANs, I persisted in this feat. I managed to improve the performance by reducing the size of the kernel of the two *Conv1D* in the discriminator, lowering the learning rate in both discriminator and adversarial training down to *1e-4* and *1e-5* respectively. Unfortunately, and because of the low computer power, increasing the epochs wasn't a good option for me. Training was taking quite a long time. Still, according to the metrics, performance was quite poor on the test set compared to the benchmarks. I realized that when predicting 5 consecutive days the generator was able to get the first price quite right, so I decided to change the strategy: predict only one day based on the previous 5 days, and perform a Recursive Multi-step strategy to predict the following 4 days. The recursive strategy involves using a one-step model multiple

times where the prediction for the prior time step is used as an input for making a prediction on the following time step. The main problem using this strategy is using predictions in place of observations. This allows prediction errors to accumulate such that performance can quickly degrade as the prediction time horizon increases. I thought this wouldn't be a big issue because I'm trying to predict 5 days, so I wasn't expecting much error to be accumulated. Following is the algorithm I implemented for this strategy:

```python
def GAN_walk_forward_validation(gen,
                                train, test,
                                size=1,
                                look_back=3,
                                n_features=1,
                                steps=1):
    '''
    Walk Forward validation. Recursive Multi-step Forecast strategy
    See https://machinelearningmastery.com/multi-step-time-series-forecasting
    params:
        gen: Generator model
        train: Series - train set
        test: Series - test set
        size: Integer - amount of days we're gonna walk
        look_back: Integer - amount of past days to forecast future ones
        n_features: Integer - amount of features (only Close price)
        steps: Integer - how many steps to make in the prediction
    '''

    past = train.reshape(-1,).copy()
    future = test.reshape(-1,)[:size]

    predictions = list()
    limit_range = len(future)

    # For re-training
    gen.compile(optimizer=RMSprop(lr=0.0002), loss='mean_squared_error')

    for t in range(0, limit_range, steps):
        x_input = past[-look_back:] # grab the last look_back days from the past

        preds_seq = []

        # Multi-step forecast loop
        for p in range(steps):
            y_output = gen.predict(x_input.reshape(1, look_back, n_features))

            y_output = y_output.reshape(-1,)

            # save the prediction in the sequence
            preds_seq.append(y_output)

            # get rid of the first input (first day look back)
            x_input = x_input[1:]

            # appends the new predicted one
            x_input = np.concatenate((x_input, y_output), axis=0)

        predicted = np.array(preds_seq).reshape(steps,)
        predictions.append(predicted)

        new_days = future[t:t+steps]

        # add the new days (real ones) to the past
        past = np.concatenate((past, new_days))

        if len(future[t:t+steps]) == steps:
            seq = past[-(look_back+steps):]
            X, y = split_sequence(seq, look_back, 1)
            X = X.reshape(-1, look_back, n_features)

            # Time to re-train the model with the new non-seen days
            gen.train_on_batch(X, y)

    return np.array(predictions).reshape(-1,)[:limit_range]
```

For the other models, MLP, CNN and LSTM, I simply decreased the learning rates from the default value *1e-3* down to *1e-4*, use *LeakyReLU* with *alpha=0.2* and increase the epochs from 100 up to 1000. They all improved the performance beating the benchmarks according to the metrics used.

IV. Results

Model Evaluation and Validation

For model evaluation I used a test set from June 1st 2018 to May 31st 2019. All unseen days by the models. For the validation of their performance I used the already mentioned in section Algorithms and Techniques, Walk-Forward validation, to predict 5 days ahead, re-training the model with the new data, real data, next 5 days Close price, once they arrive. This validation is how the model would be used realistically.

Even though I managed to improve the NN models to beat all the benchmarks, unfortunately I could not beat ARIMA, which was actually one of the goals of this project. It was a bit disappointing also the results of my GAN, where I put lots of hopes to get great results.

Something important to mention here is that I did try to use regularization (see Jupyter notebooks) methods such as Dropout layers, or L2 regularizer with a *decay_weight=1e-4*, since all the models seems to be overfitting, but the results were always much worse. I guess the randomness of the time series is the reason for this.

Justification

Following are the results of the best models I built, after hyperparameter tuning, against the benchmarks using the metrics mentioned in Metrics section:

Model	**Bias**	**MAE**	**MSE**	**RMSE**	**MAPE**
ARIMA	-0.040	4.039	28.784	5.365	2.147
MLP	-0.361	4.656	37.020	6.084	2.496
CNN	-0.422	4.607	36.686	6.057	2.473
LSTM	-0.408	4.448	36.005	6.000	2.380
GAN	-0.057	4.781	38.020	6.166	2.535

Benchmark	Bias	MAE	MSE	RMSE	MAPE
Persistence	-0.203	5.729	54.819	7.404	3.050
SMA	-0.166	5.286	47.474	6.890	2.816
EMA	-0.074	4.694	37.944	6.160	2.504

Based on these results, I'd say along with ARIMA, LSTM models are a good option to be used by traders for stock market prediction, achieving good enough performance in predicting one week of prices.

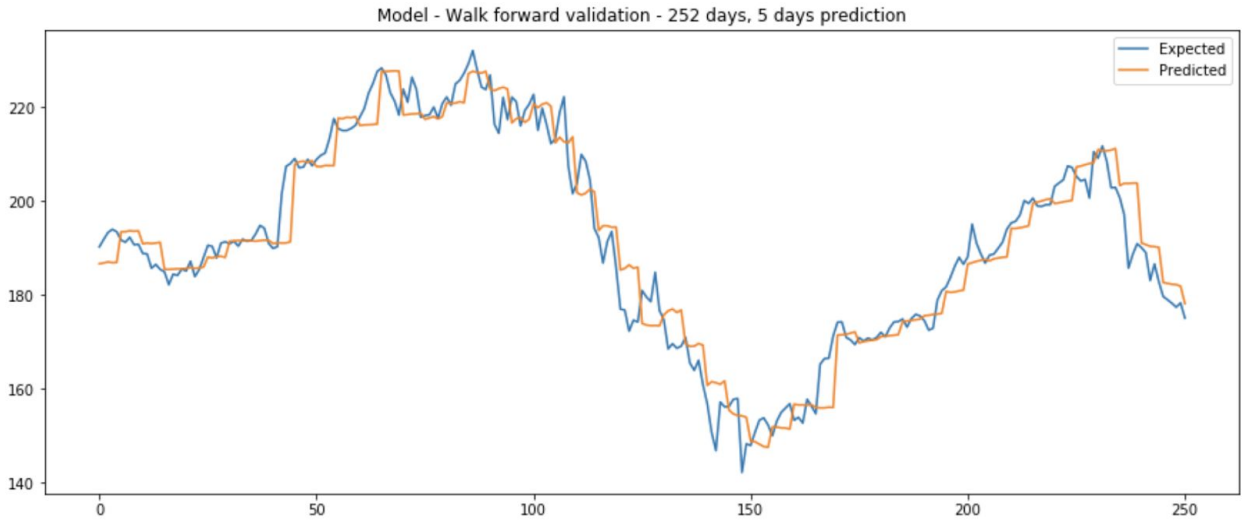
V. Conclusion

(approx. 1-2 pages)

Free-Form Visualization

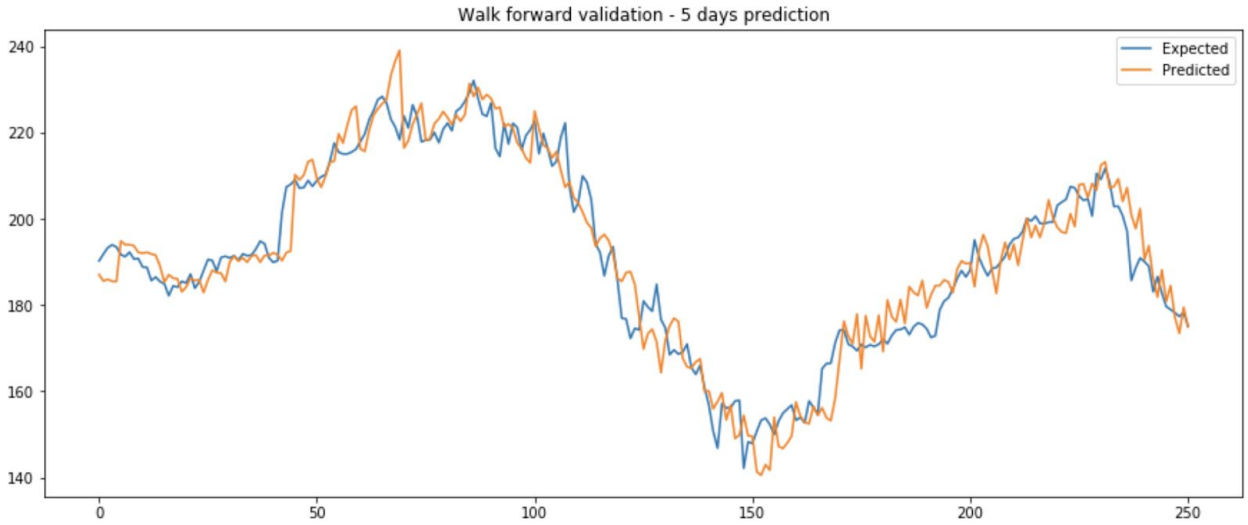
We can see by plotting the prediction against actual prices that all the models are reacting quite fast to changes, which could make them interesting indicators to be used by traders. Let's compare the prediction of all the models against the real prices. Remember these are 5 days prediction:

ARIMA model




```
ARIMA[252 days, 5 days forecast]:

Forecast Bias: -0.040
MAE: 4.039
MSE: 28.784
RMSE: 5.365
MAPE: 2.147
```

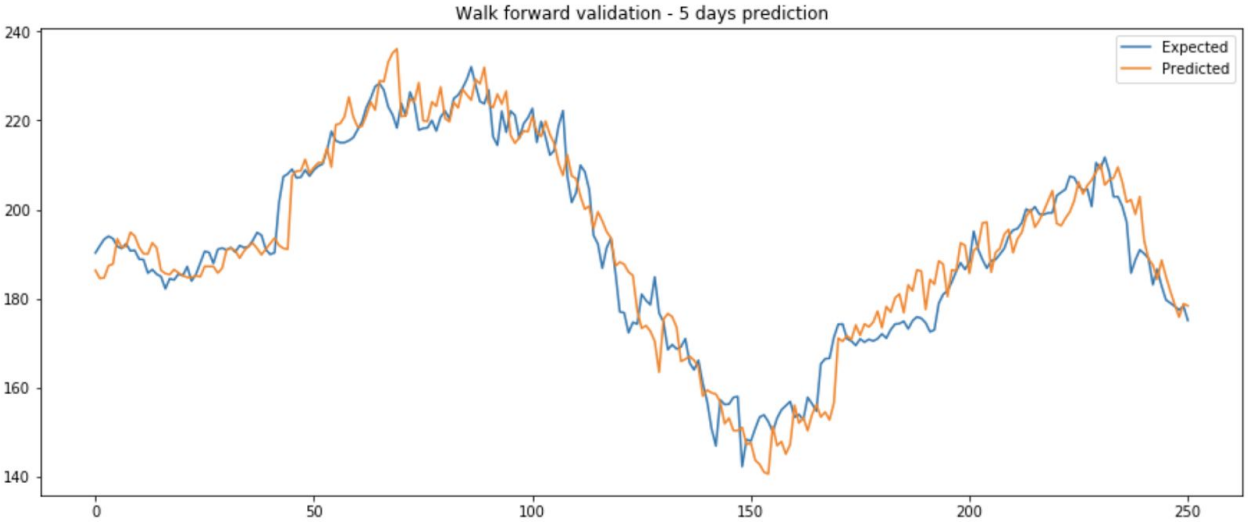
MLP model



MLP[252 days, 5 days forecast]:

Forecast Bias: -0.361
MAE: 4.656
MSE: 37.020
RMSE: 6.084
MAPE: 2.496

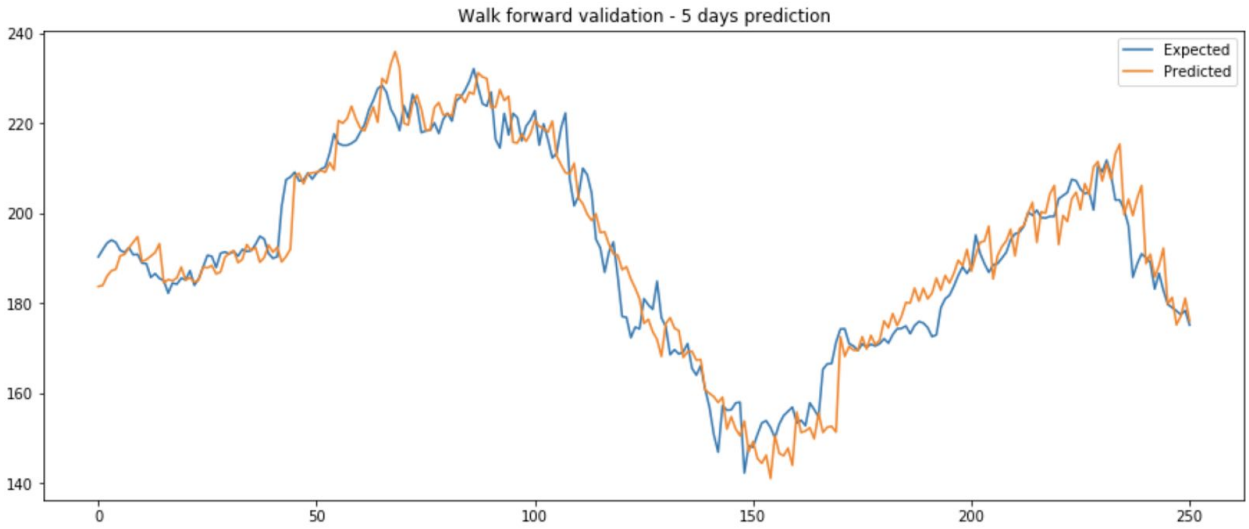
CNN model



CNN[252 days, 5 days forecast]:

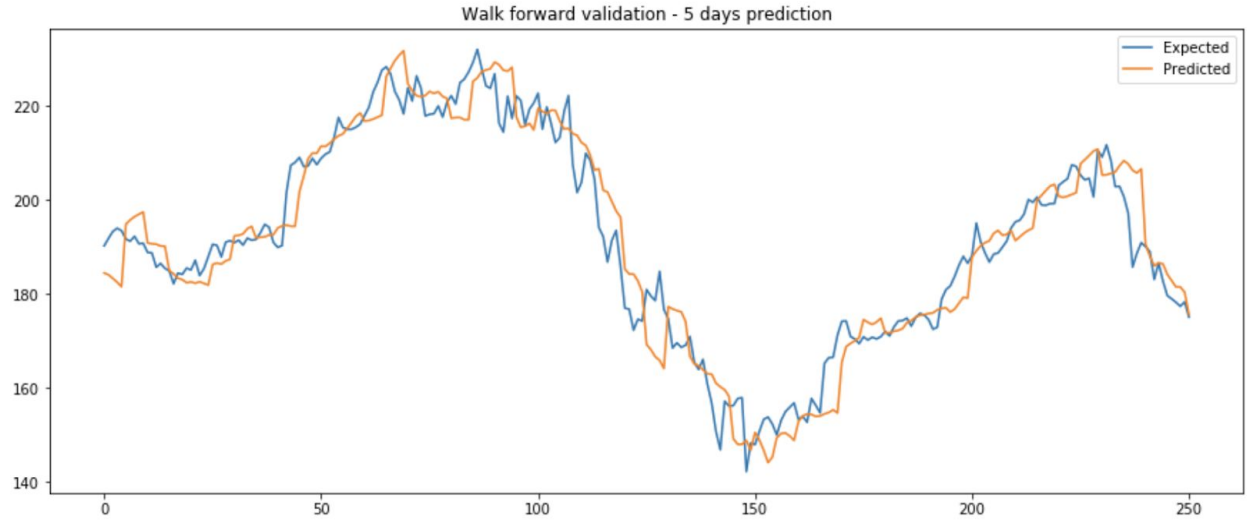
Forecast Bias: -0.422
MAE: 4.607
MSE: 36.686
RMSE: 6.057
MAPE: 2.473

LSTM model




```
LSTM[252 days, 5 days forecast]:

Forecast Bias: -0.408
MAE: 4.448
MSE: 36.005
RMSE: 6.000
MAPE: 2.380
```

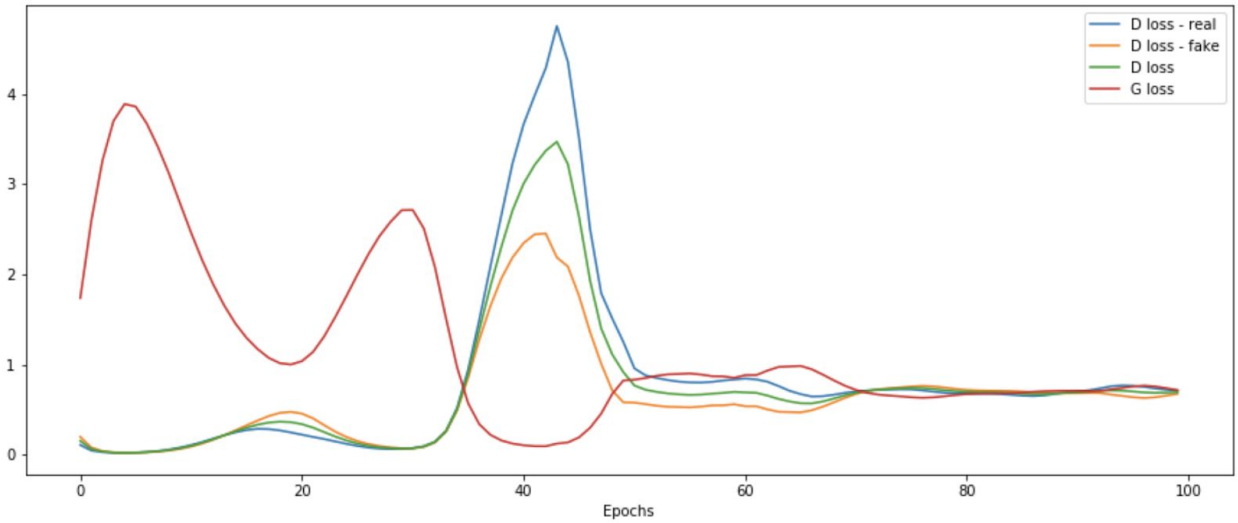
GAN model




```
GAN[252 days, 1 days forecast]:

Forecast Bias: -0.057
MAE: 4.781
MSE: 38.020
RMSE: 6.166
MAPE: 2.535
```

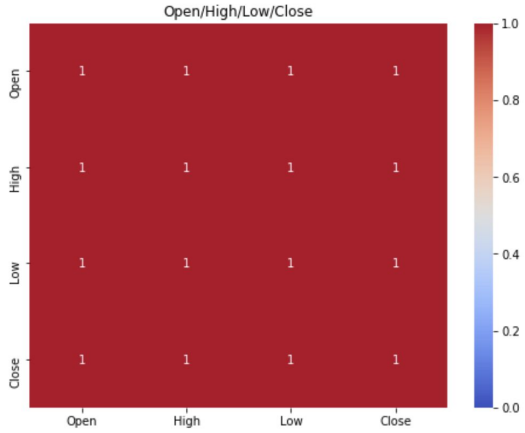
Something worth to notice is that ARIMA is more “conservative” in the way is predicting, kind of shifting prices and assuming the next 5 days will just slightly go up or down. On the other hand, the NN models are taking more risk by trying to predict the movement based on what they have learnt. This is the reason why they fail more, because what they have learnt is quite random, without all the inputs that really change the stock market, and they’re trying to reproduce some kind of pattern they’ve seen in the historical data, which of course, it’s just not enough.

Also very interesting to look at the GAN model’s loss, where one can easily see the fierce fight between generator and discriminator, reaching a point of equilibrium, Nash Equilibrium. GAN is based on the zero-sum non-cooperative game. In short, if one wins the other loses. A zero-sum game is also called minimax. Your opponent wants to maximize its actions and your actions are to minimize them. In game theory, the GAN model converges when the discriminator and the generator reach a Nash equilibrium.

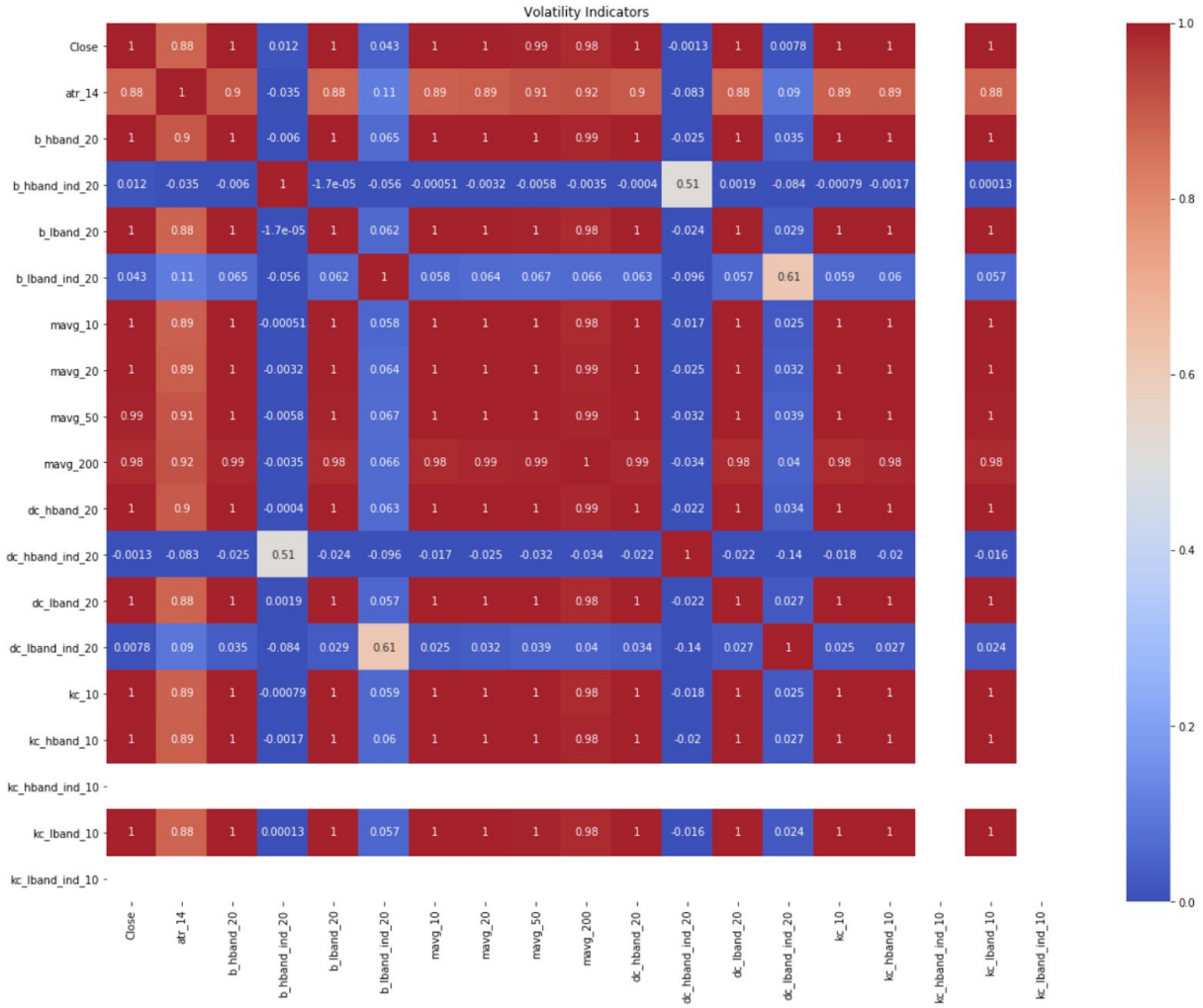


I’d like to mention that adding technical indicators didn’t improve the predictions, and actually worsened them. In practice these indicators are used by traders to try to figure out price movements, but in practice, and this can be seen in the Jupyter notebooks, they didn’t do a good job. My theory is that most of them are calculated from the actual price and these features are highly correlated, which is confusing the model. I used a heatmap to plot this correlation between features:

```
cols = ['Open', 'High', 'Low', 'Close']
plot_heatmap_corr(featured_series[cols], title='Open/High/Low/Close', figsize=(8, 6))
```



We can see the highly correlation between Open, High, Low and Close price. I also used mostly [volatility indicators](), which were selected as important features by XGBoost:



This correlating makes them, in my opinion, useless features, and as seen in practice, they're simply not helping the model with the predictions.

Reflection

Time series forecasting is definitely a fascinating are of Machine Learning. In the last two months I learnt a lot with this project. Unlike the other Machine Learning problems, classification and regression, time series adds the temporal component, which makes it more complex to tackle, as I could experience myself. I had to learn about the taxonomy of time series forecasting, univariate vs multivariate, single step vs multi-step. Also about the different components of a time series, such as trend, seasonality, noise. The importance of stationarity when dealing with time series, and how to make them stationary. Re-framing the problem as Supervised Learning was also quite interesting. I also learnt a lot about statistical models used for time series such as Autoregressive models (AR) and Moving Average models (MA). And more sophisticated ones ARMA, and ARIMA where integration is incorporated into the picture to make the series stationary.

I can summarize the entire process and all the steps I took for this project in the following points:

1. Obtain the dataset, our time series, from one of the many sources there are on the internet. I personally recommend Yahoo! Finance
2. Data exploration and visualization to understand the time series. This includes exploring stationarity, which is an important aspect to consider in this kind of problems. In practice when using Neural Networks, this doesn't seem to affect the performance of the models though.
3. Identifying potential issues such as missing values, or anomalies and try to correct them.
4. Explore different ways the dataset can be enhanced, such as doing feature engineering, in this case, adding technical indicators to the dataset. In practice, as I mentioned in Free-Form Visualization section, these indicators turned out to be not very helpful. At least not in my experiments.
5. Explore the distribution of the Close price (our target label), and the different ways it can be transformed to help the model, for example using power transformations. It's worth mentioning that I found Cox-Box test the best way to transform. Unfortunately, because I used indicators with negative values, I had to resort to a different algorithm to transform my data when using features. I used Yeo-Johnson transformation, provided by sklearn.preprocessing.PowerTransformer, which supports 0 and negative values.
6. Build the benchmarks and metrics, and use them to make naive predictions. The results will be used to be compared with the actual solutions.
7. Build the models, simple ones, train them and run a walk-forward validation on them to see the results. GANs took me very long time to understand and implement. Lots of research, trials and errors before I got something more or less working. I ran into many issues like Mode Collapse as I explained before. I tried many different architectures. Using latent noise, or simple real inputs. Both actually achieve similar results. Changing the way generator and discriminator are connected. I also tried some solutions out there like Wasserstein GAN (WGAN), but I didn't succeed in improving the prediction. I wasn't

even sure this solutions could be applied to time series forecasting problems since all the papers about GAN and their different architectures are related to images.

8. Hyperparameter tuning is the next step to try to improve the performance of these models. Because of the low computer power I have, unfortunately I couldn't play much with this. I always had to run around 100 epochs because it was taking quite a long time for some models to train, above all LSTMs and GANs.
9. Trying different strategies to make predictions, not only output strategy, but also recursive multi-step. Surprisingly, this last strategy was the best option for GAN model
10. Interpret the results and evaluate against benchmarks.

The results were not what I expected when I decided to go for this problem. I had big hopes that NN models could do a great job here, but as I could see myself, it's not the case. I think one of the reasons is simply because Stock market prices are highly volatile and unpredictable. Models cannot really get much from historical prices. They're missing lots of inputs such as industry performance, news, market sentiment, different fundamental and technical factors, etc. They don't have this information that are moving the Stock prices.

Improvement

I'm sure that with a more thoroughly hyperparameter tuning, longer training, and different architectures, with more stacked layers, we could achieve better prediction power than ARIMA model. Unfortunately time was a bit against this experimentation.

Regularization didn't improve at all the models and actually it was in detriment, but maybe with a longer training, a couple of thousands epochs, the models could generalize better, and improve, although I'm still not convinced that regularization help here due to the random nature of this timeseries, where no patterns seem to exist.

A Direct-Recursive Hybrid Strategies could help to make better predictions, where you build as many models as days in the future you want to predict, and use the prediction of models predicting previous days as inputs for the models in charge to predict next days, as followed:
prediction(t+1) = model(obs(t), obs(t-1), obs(t-2), ..., obs(t-n))
prediction(t+2) = model(prediction(t+1), obs(t), obs(t-1), ..., obs(t-n))
…
prediction(t+5) = model(prediction(t+4), prediction(t+3), prediction(t+2), ..., obs(t-n))

More research and experimentation with GANs. There are many tricks and tips to make GANs work, such as increasing kernel size and adding more filters. Sometimes, flipping labels assignment while training the discriminator. That means, generated prices are real, actual real ones are fake. This sounds very strange at the beginning, but it seems to help with the gradient flow. Using soft and noisy labels preventing the discriminator from getting overconfident while classifying real and fake prices, which kills the training at the very beginning, approaching its loss very rapidly. I read in some places that batch normalization in the discriminator also helps

with the training, but I got worse results with this layer before the nonlinear activation, so I decided to drop it.

There is definitely lots of room for improvement, but I personally think I have achieved quite a lot in this project, and even though the NN models couldn't beat the statistical model ARIMA, I believe they can do a much better job in other types of time series where there are patterns hidden in noise. Also, if we could feed our models with more inputs like the result of sentiment analysis, and other variables that move the stock prices, then NN models can perform really predict, maybe not prices, but instead market movements with high accuracy.

References

Annexed references