Introduction

Most of us would have heard about the new buzz in the market i.e. Cryptocurrency. Many of us would have invested in their coins too. But, is investing money in such a volatile currency safe? How can we make sure that investing in these coins now would surely generate a healthy profit in the future? We can't be sure but we can surely generate an approximate value based on the previous prices. Time series models is one way to predict them.

Source: Bitcoin

Besides Crypto Currencies, there are multiple important areas where time series forecasting is used for example: forecasting Sales,Call Volume in a Call Center, Solar activity, Ocean tides, Stock market behaviour, and many others .

Assume the Manager of a hotel wants to predict how many visitors he should expect next year to accordingly adjust the hotel's inventories and make a reasonable guess of the hotel's revenue. Based on the data of the previous years /months/days, (S)he can use time series forecasting and get an approximate value of the visitors. Forecasted value of visitors will help the hotel to manage the resources and plan things accordingly.

In this article, we will learn about multiple forecasting techniques and compare them by implementing on a dataset. We will go through different techniques and see how to use these methods to improve score.

Let's get started!

Table of Contents

Understanding the Problem Statement and Dataset

We are provided with a Time Series problem involving prediction of number of commuters of JetRail, a new high speed rail service by Unicorn Investors. We are provided with 2 years of data(Aug 2012-Sept 2014) and using this data we have to forecast the number of commuters for next 7 months.

Let's start working on the dataset downloaded from the above link. In this article, I'm working with train dataset only.

```
import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

#Importing data

df = pd.read_csv('train.csv')

#Printing head

df.head()
```

	ID	Datetime	Count
0	0	25-08-2012 00:00	8
1	1	25-08-2012 01:00	2
2	2	25-08-2012 02:00	6
3	3	25-08-2012 03:00	2
4	4	25-08-2012 04:00	2

```
#Printing tail

df.tail()
```

	ID	Datetime	Count
18283	18283	25-09-2014 19:00	868
18284	18284	25-09-2014 20:00	732
18285	18285	25-09-2014 21:00	702
18286	18286	25-09-2014 22:00	580
18287	18287	25-09-2014 23:00	534

As seen from the print statements above, we are given 2 years of data (2012-2014) at ***hourly level*** with the number of commuters travelling and we need to estimate the number of commuters for future.

In this article, I'm sub setting and aggregating dataset at daily basis to explain the different methods.

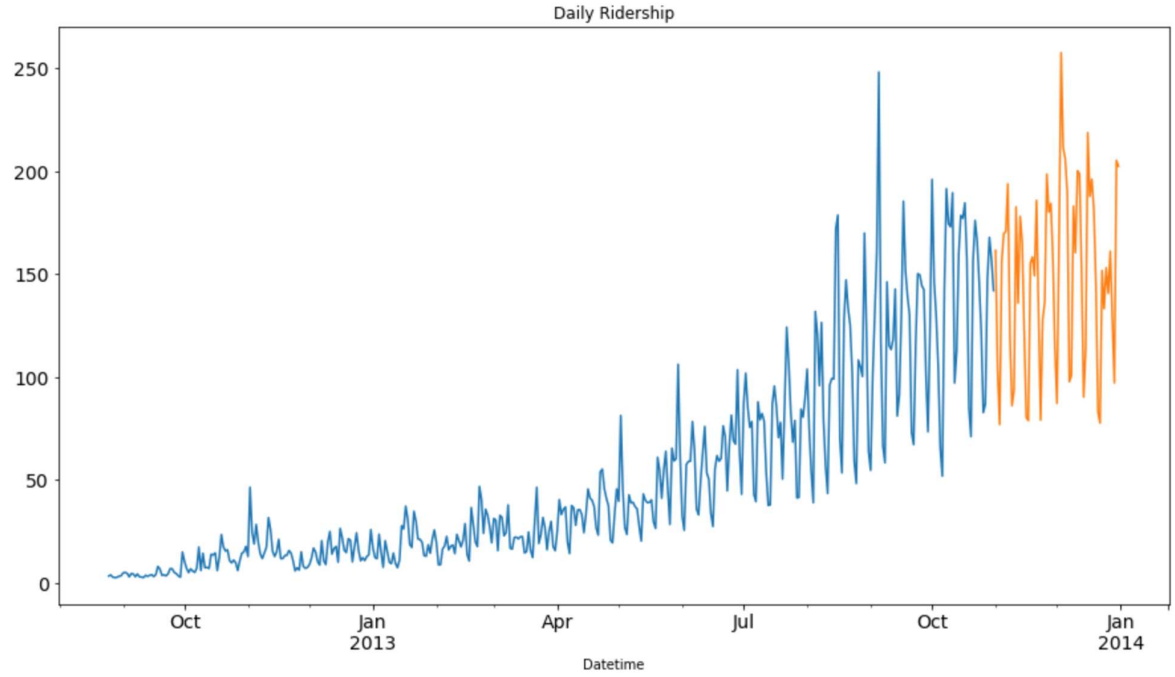
- Sub setting the dataset from (August 2012 – Dec 2013)
- Creating train and test file for modelling. The first 14 months (August 2012 – October 2013) are used as training data and next 2 months (Nov 2013 – Dec 2013) as testing data.
- Aggregating the dataset at daily basis

```
#Subsetting the dataset

#Index 11856 marks the end of year 2013

df = pd.read_csv('train.csv', nrows = 11856)



#Creating train and test set

#Index 10392 marks the end of October 2013

train=df[0:10392]
```

```
test=df[10392:]

#Aggregating the dataset at daily level

df.Timestamp = pd.to_datetime(df.Datetime,format='%d-%m-%Y %H:%M')

df.index = df.Timestamp

df = df.resample('D').mean()

train.Timestamp = pd.to_datetime(train.Datetime,format='%d-%m-%Y %H:%M')

train.index = train.Timestamp

train = train.resample('D').mean()

test.Timestamp = pd.to_datetime(test.Datetime,format='%d-%m-%Y %H:%M')

test.index = test.Timestamp

test = test.resample('D').mean()
```

Let’s visualize the data (train and test together) to know how it varies over a time period.

```
#Plotting data

train.Count.plot(figsize=(15,8), title= 'Daily Ridership', fontsize=14)

test.Count.plot(figsize=(15,8), title= 'Daily Ridership', fontsize=14)

plt.show()
```



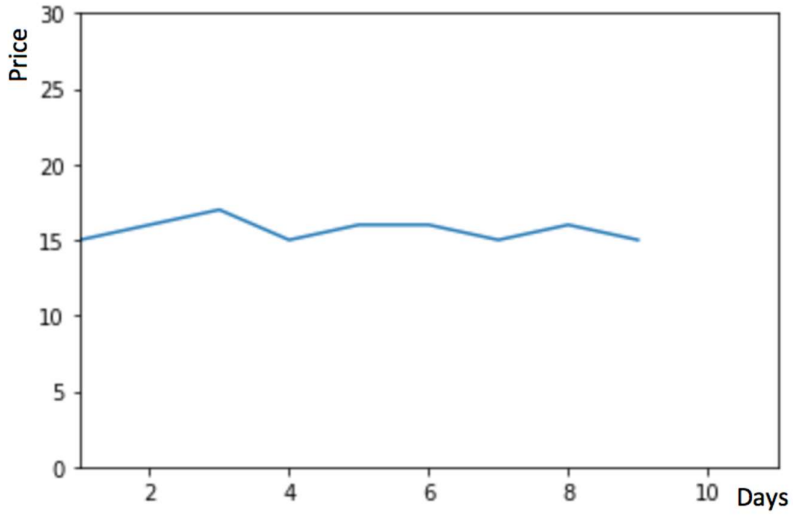
Installing library (statsmodels)

The library which I have used to perform Time series forecasting is statsmodels. You need to install it before applying few of the given approaches. statsmodels might already be installed in your python environment but it doesn't support forecasting methods. We will clone it from their repository and install using the source code. Follow these steps :-

1. Use pip freeze to check if it's already installed in your environment.
2. If already present, remove it using "conda remove statsmodels"
3. Clone the statsmodels repository using "git clone git://github.com/statsmodels/statsmodels.git". Initialise the Git using "git init" before cloning.
4. Change the directory to statsmodels using "cd statsmodels"
5. Build the setup file using "python setup.py build"
6. Install it using "python setup.py install"
7. Exit the bash/terminal
8. Restart the bash/terminal in your environment, open python and execute "from statsmodels.tsa.api import ExponentialSmoothing" to verify.

Method 1: Start with a Naive Approach

Consider the graph given below. Let's assume that the y-axis depicts the price of a coin and x-axis depicts the time (days).



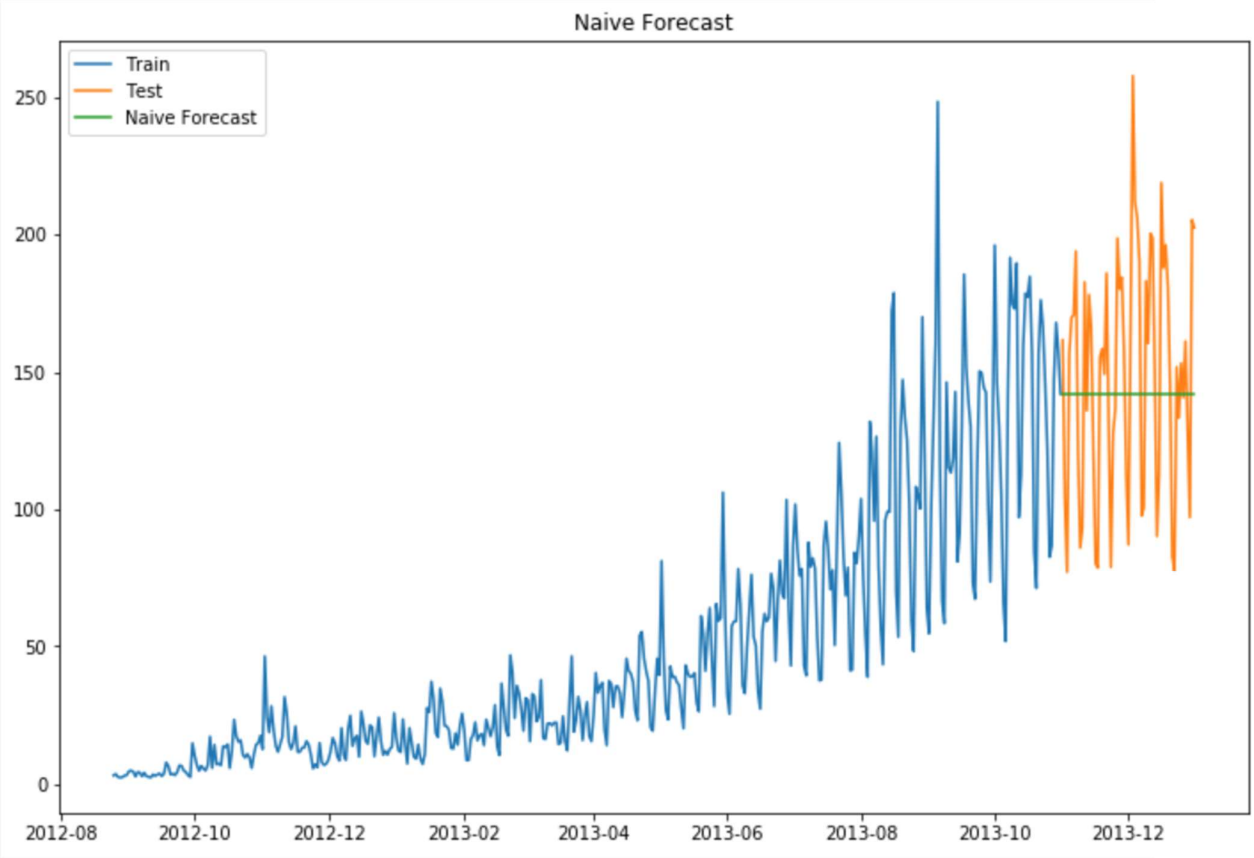
We can infer from the graph that the price of the coin is stable from the start. Many a times we are provided with a dataset, which is stable throughout it's time period. If we want to forecast the price for the next day, we can simply take the last day value and estimate the same value for the next day. Such forecasting technique which assumes that the next expected point is equal to the last observed point is called **Naive Method.**

Hence $\hat{y}_{t+1} = y_t$.

Now we will implement the Naive method to forecast the prices for test data.

```
dd= np.asarray(train.Count)

y_hat = test.copy()

y_hat['naive'] = dd[len(dd)-1]

plt.figure(figsize=(12,8))

plt.plot(train.index, train['Count'], label='Train')

plt.plot(test.index,test['Count'], label='Test')
```

```
plt.plot(y_hat.index,y_hat['naive'], label='Naive Forecast')

plt.legend(loc='best')

plt.title("Naive Forecast")

plt.show()
```



We will now calculate RMSE to check to accuracy of our model on test data set.

```
from sklearn.metrics import mean_squared_error

from math import sqrt
```

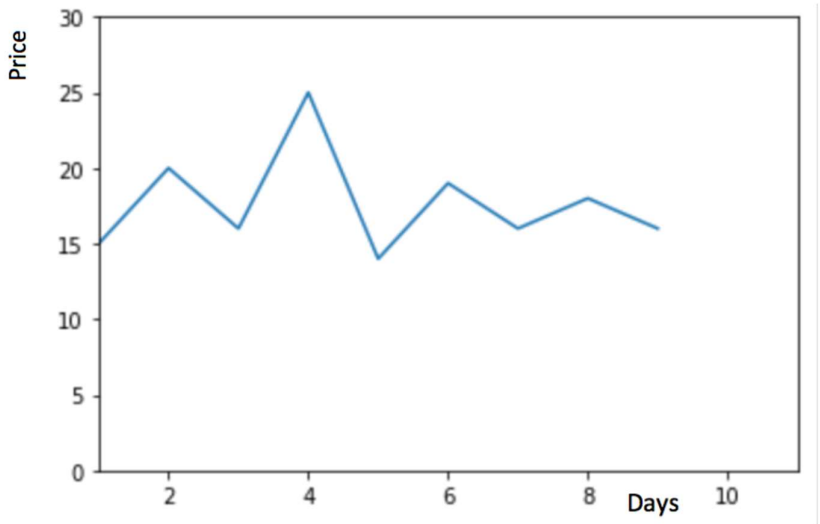
```
rms = sqrt(mean_squared_error(test.Count, y_hat.naive))

print(rms)
```

RMSE = 43.9164061439

We can infer from the RMSE value and the graph above, that Naive method isn't suited for datasets with high variability. It is best suited for stable datasets. We can still improve our score by adopting different techniques. Now we will look at another technique and try to improve our score.

Method 2: – Simple Average

Consider the graph given below. Let's assume that the y-axis depicts the price of a coin and x-axis depicts the time(days).



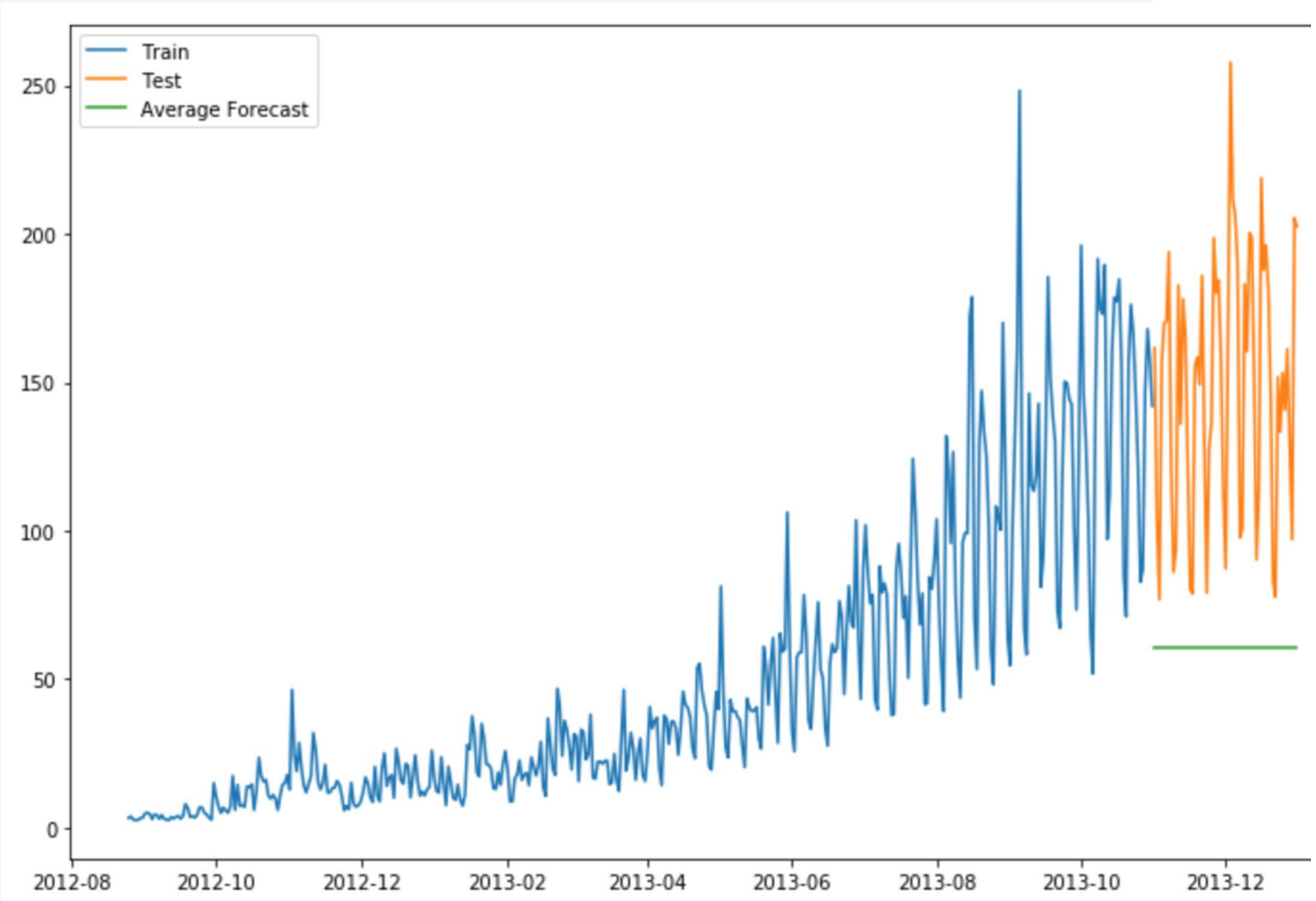
We can infer from the graph that the price of the coin is increasing and decreasing randomly by a small margin, such that the average remains constant. Many a times we are provided with a dataset, which though varies by a small margin throughout it's time period, but the average at each time period remains constant. In such a case we can forecast the price of the next day somewhere similar to the average of all the past days.

Such forecasting technique which forecasts the expected value equal to the average of all previously observed points is called Simple Average technique.

Hence $\hat{y}_{x+1} = \frac{1}{x}\sum_{i=1}^{x} y_i$

We take all the values previously known, calculate the average and take it as the next value. Of course it won't be it exact, but somewhat close. As a forecasting method, there are actually situations where this technique works the best.

```
y_hat_avg = test.copy()

y_hat_avg['avg_forecast'] = train['Count'].mean()

plt.figure(figsize=(12,8))

plt.plot(train['Count'], label='Train')

plt.plot(test['Count'], label='Test')

plt.plot(y_hat_avg['avg_forecast'], label='Average Forecast')

plt.legend(loc='best')

plt.show()
```



We will now calculate RMSE to check to accuracy of our model.

```
rms = sqrt(mean_squared_error(test.Count, y_hat_avg.avg_forecast))

print(rms)
```

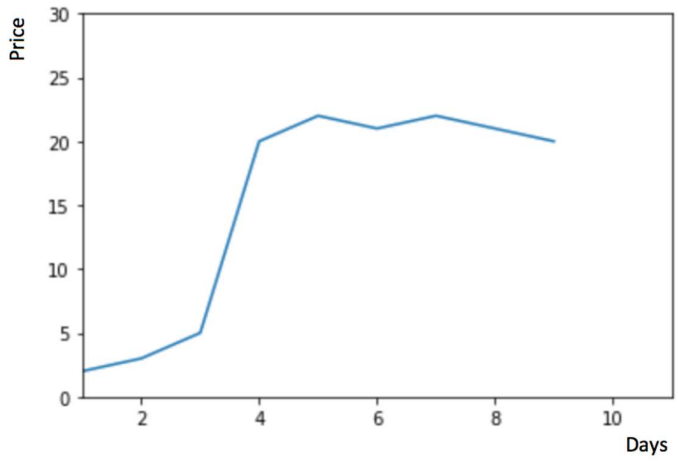
RMSE = 109.545990803

We can see that this model didn't improve our score. Hence we can infer from the score that this method works best when the average at each time period remains constant. Though the score of Naive method is better than Average method, but this does not mean that the Naive

method is better than Average method on all datasets. We should move step by step to each model and confirm whether it improves our model or not.

Method 3 – Moving Average

Consider the graph given below. Let's assume that the y-axis depicts the price of a coin and x-axis depicts the time(days).



We can infer from the graph that the prices of the coin increased some time periods ago by a big margin but now they are stable. Many a times we are provided with a dataset, in which the prices/sales of the object increased/decreased sharply some time periods ago. In order to use the previous Average method, we have to use the mean of all the previous data, but using all the previous data doesn't sound right.

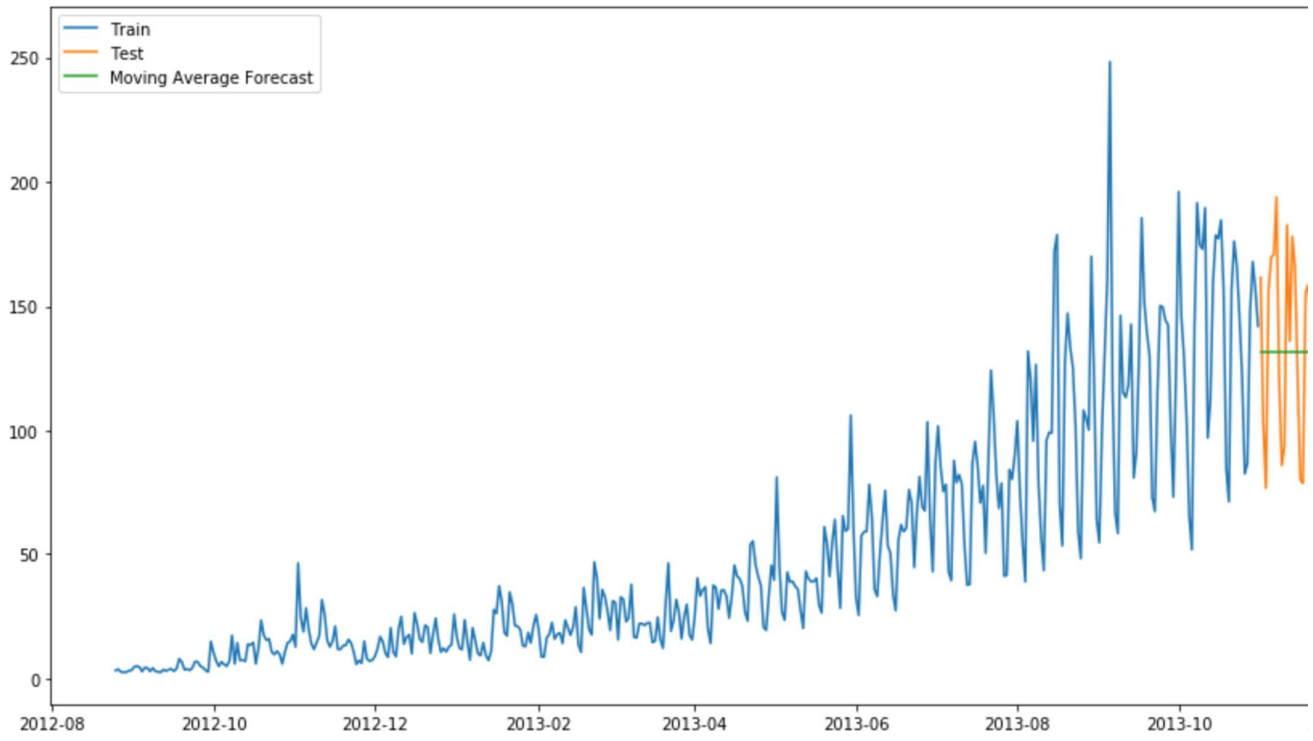
Using the prices of the initial period would highly affect the forecast for the next period. Therefore as an improvement over simple average, we will take the average of the prices for last few time periods only. Obviously the thinking here is that only the recent values matter. Such forecasting technique which uses window of time period for calculating the average is called Moving Average technique. Calculation of the moving average involves what is sometimes called a "sliding window" of size n.

Using a simple moving average model, we forecast the next value(s) in a time series based on the average of a fixed finite number 'p' of the previous values. Thus, for all i > p

$$\hat{y}_i = \frac{1}{p}(y_{i-1} + y_{i-2} + y_{i-3} \ldots\ldots + y_{i-p})$$

A moving average can actually be quite effective, especially if you pick the right p for the series.

```
y_hat_avg = test.copy()

y_hat_avg['moving_avg_forecast'] = train['Count'].rolling(60).mean().iloc[-1]

plt.figure(figsize=(16,8))

plt.plot(train['Count'], label='Train')

plt.plot(test['Count'], label='Test')

plt.plot(y_hat_avg['moving_avg_forecast'], label='Moving Average Forecast')

plt.legend(loc='best')

plt.show()
```



We chose the data of last 2 months only. We will now calculate RMSE to check to accuracy of our model.

```
rms = sqrt(mean_squared_error(test.Count, y_hat_avg.moving_avg_forecast))

print(rms)

RMSE = 46.7284072511
```

We can see that Naive method outperforms both Average method and Moving Average method for this dataset. Now we will look at Simple Exponential Smoothing method and see how it performs.

An advancement over Moving average method is **Weighted moving average** method. In the Moving average method as seen above, we equally weigh the past 'n' observations. But we might encounter situations where each of the observation from the past 'n' impacts the forecast in a different way. Such a technique which weighs the past observations differently is called Weighted Moving Average technique.

A weighted moving average is a moving average where within the sliding window values are given different weights, typically so that more recent points matter **more**. Ins

$$\text{Hence, } \hat{y_i} = \frac{1}{m}(w_1 * y_{i-1} + w_2 * y_{i-2} + w_3 * y_{i-3} \ldots\ldots + w_m * y_{i-m})$$

tead of selecting a window size, it requires a list of weights (which should add up to 1). For example if we pick [0.40, 0.25, 0.20, 0.15] as weights, we would be giving 40%, 25%, 20% and 15% to the last 4 points respectively.

Method 4 – Simple Exponential Smoothing

After we have understood the above methods, we can note that both Simple average and Weighted moving average lie on completely opposite ends. We would need something between these two extremes approaches which takes into account all the data while weighing the data points differently. For example it may be sensible to attach larger weights to more recent observations than to observations from the distant past. The technique which works on this principle is called Simple exponential smoothing. Forecasts are calculated using weighted averages where the weights decrease exponentially as observations come from further in the past, the smallest weights are associated with the oldest observations:

$$\hat{y}_{T+1|T} = \alpha y_T + \alpha(1-\alpha)y_{T-1} + \alpha(1-\alpha)^2 y_{T-2} + \cdots$$

where $0 \leq \alpha \leq 1$ is the **smoothing** parameter.

The one-step-ahead forecast for time T+1 is a weighted average of all the observations in the series y1,…,yT. The rate at which the weights decrease is controlled by the parameter α.

If you stare at it just long enough, you will see that the expected value $\hat{y}_x$ is the sum of two products: $\alpha \cdot y_t$ and $(1-\alpha) \cdot \hat{y}_{t-1}$.

Hence, it can also be written as :

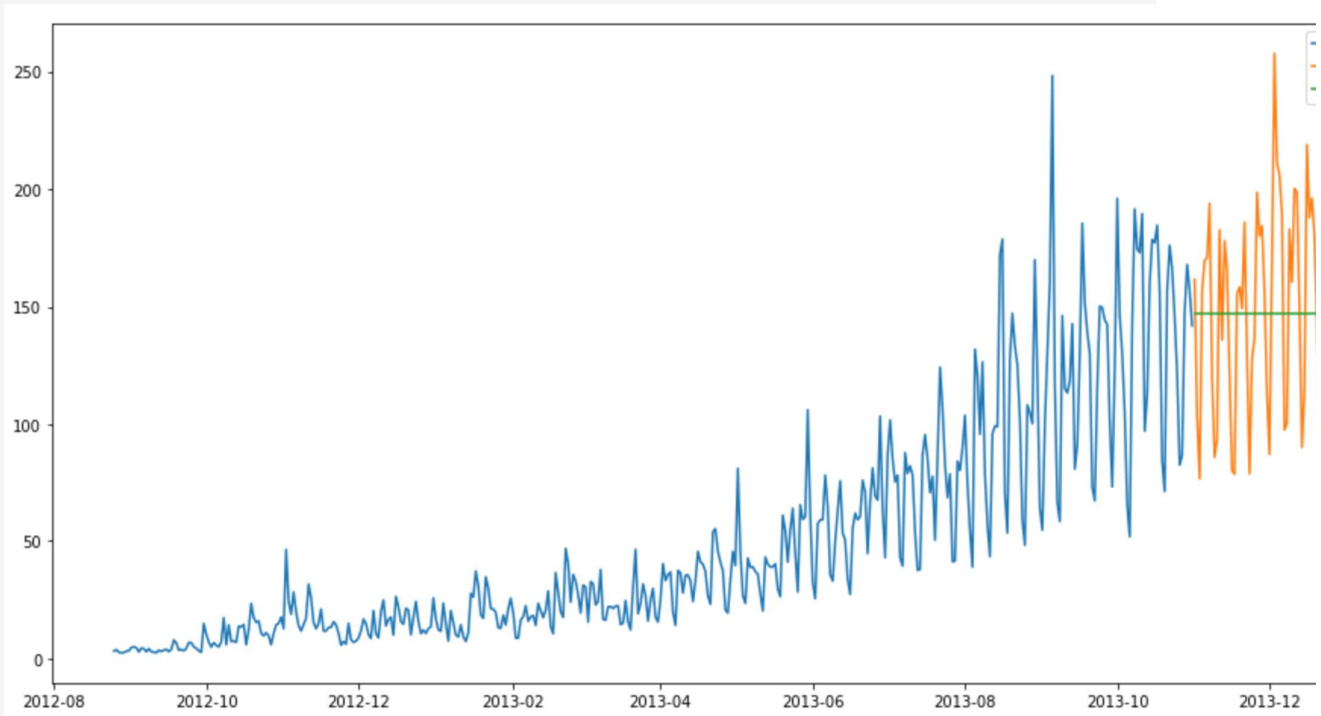
$$\hat{y}_{t+1|t} = \alpha * y_t + (1-\alpha) * \hat{y}_{t|t-1}$$

So essentially we've got a weighted moving average with two weights: α and 1−α.

As we can see, 1−α is multiplied by the previous expected value $\hat{y}_{x-1}$ which makes the expression recursive. And this is why this method is called **Exponential**. The forecast at time t+1 is equal to a weighted average between the most recent observation y_t and the most recent forecast $\hat{y}_{t|t-1}$.

```
from statsmodels.tsa.api import ExponentialSmoothing, SimpleExpSmoothing, Holt
```

```
y_hat_avg = test.copy()

fit2 = SimpleExpSmoothing(np.asarray(train['Count'])).fit(smoothing_level=0.6,optimized=False)

y_hat_avg['SES'] = fit2.forecast(len(test))

plt.figure(figsize=(16,8))

plt.plot(train['Count'], label='Train')

plt.plot(test['Count'], label='Test')

plt.plot(y_hat_avg['SES'], label='SES')

plt.legend(loc='best')

plt.show()
```



We will now calculate RMSE to check to accuracy of our model.

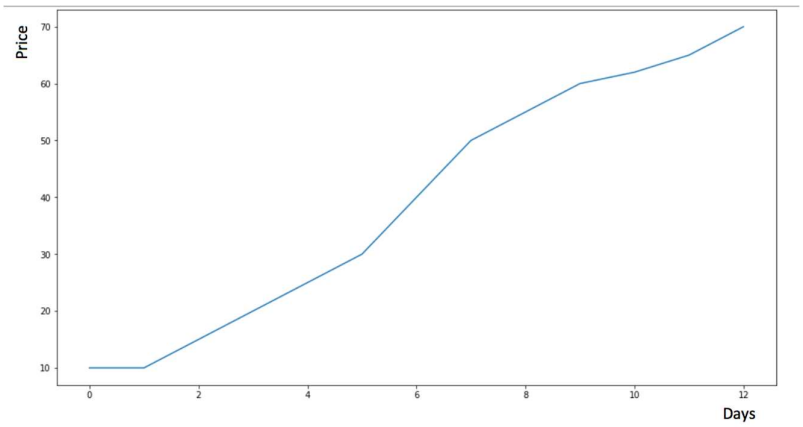
```
rms = sqrt(mean_squared_error(test.Count, y_hat_avg.SES))

print(rms)
```

RMSE = 43.3576252252

We can see that implementing Simple exponential model with alpha as 0.6 generates a better model till now. We can tune the parameter using the validation set to generate even a better Simple exponential model.

Method 5 – Holt's Linear Trend method

We have now learnt several methods to forecast but we can see that these models don't work well on data with high variations. Consider that the price of the bitcoin is increasing.



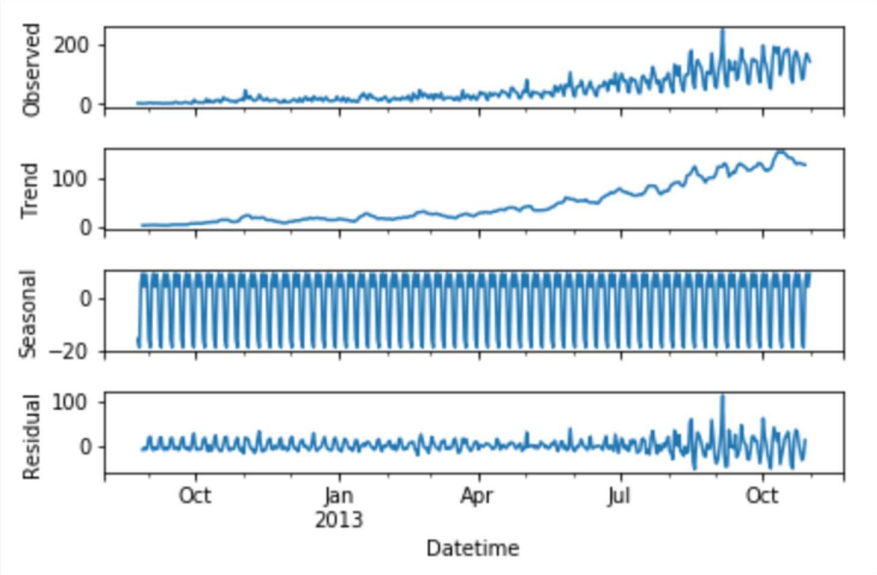
If we use any of the above methods, it won't take into account this trend. Trend is the general pattern of prices that we observe over a period of time. In this case we can see that there is an increasing trend.

Although each one of these methods can be applied to the trend as well. E.g. the Naive method would assume that trend between last two points is going to stay the same, or we could average all slopes between all points to get an average trend, use a moving trend average or apply exponential smoothing.

But we need a method that can map the trend accurately without any assumptions. Such a method that takes into account the trend of the dataset is called Holt's Linear Trend method.

Each Time series dataset can be decomposed into it's componenets which are Trend, Seasonality and Residual. Any dataset that follows a trend can use Holt's linear trend method for forecasting.

```
import statsmodels.api as sm

sm.tsa.seasonal_decompose(train.Count).plot()

result = sm.tsa.stattools.adfuller(train.Count)

plt.show()
```



We can see from the graphs obtained that this dataset follows an increasing trend. Hence we can use Holt's linear trend to forecast the future prices.

Holt extended simple exponential smoothing to allow forecasting of data with a trend. It is nothing more than exponential smoothing applied to both level(the average value in the series) and trend. To express this in mathematical notation we now need three equations: one for level, one for the trend and one to combine the level and trend to get the expected forecast ŷ

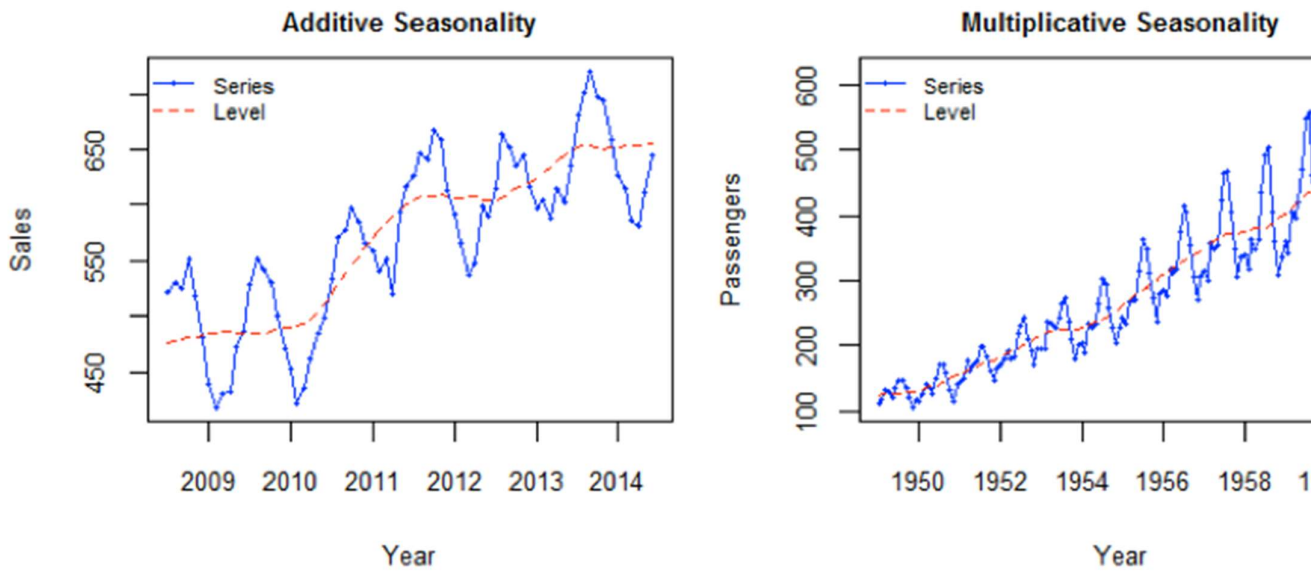
$$\text{Forecast equation : } \hat{y}_{t+h|t} = \ell_t + h\, b_t$$

$$\text{Level equation : } \ell_t = \alpha y_t + (1-\alpha)(\ell_{t-1}+b_{t-1})$$

$$\text{Trend equation : } b_t = \beta*(\ell_t - \ell_{t-1})+(1-\beta)b_{t-1}$$

The values we predicted in the above algorithms are called Level. In the above three equations, you can notice that we have added level and trend to generate the forecast equation.

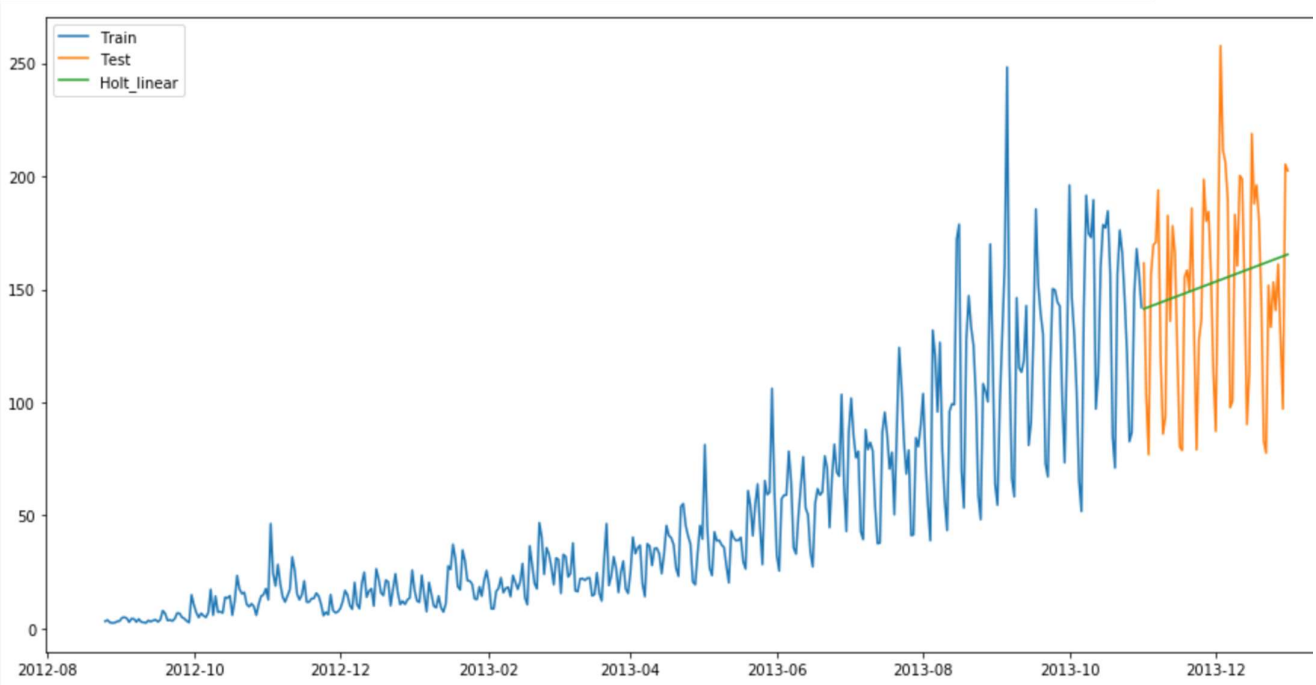
As with simple exponential smoothing, the level equation here shows that it is a weighted average of observation and the within-sample one-step-ahead forecast The trend equation shows that it is a weighted average of the estimated trend at time t based on $\ell(t)-\ell(t-1)$ and $b(t-1)$, the previous estimate of the trend.
We will add these equations to generate Forecast equation. We can also generate a multiplicative forecast equation by multiplying trend and level instead of adding it. When the trend increases or decreases linearly, additive equation is used whereas when the trend increases of decreases exponentially, multiplicative equation is used.Practice shows that multiplicative is a more stable predictor, the additive method however is simpler to understand.



source

```
y_hat_avg = test.copy()



fit1 = Holt(np.asarray(train['Count'])).fit(smoothing_level = 0.3,smoothing_slope = 0.1)

y_hat_avg['Holt_linear'] = fit1.forecast(len(test))



plt.figure(figsize=(16,8))

plt.plot(train['Count'], label='Train')

plt.plot(test['Count'], label='Test')

plt.plot(y_hat_avg['Holt_linear'], label='Holt_linear')
```

```
plt.legend(loc='best')

plt.show()
```



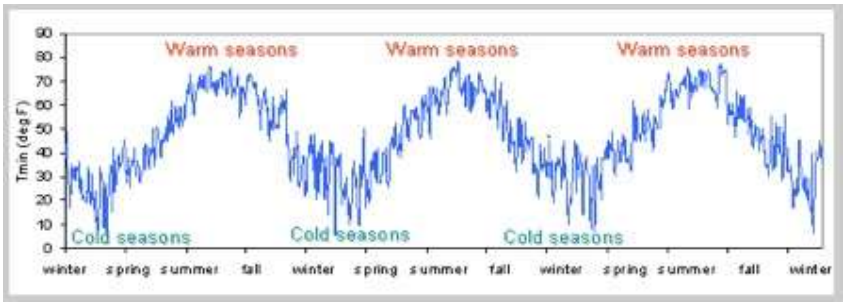
We will now calculate RMSE to check to accuracy of our model.

```
rms = sqrt(mean_squared_error(test.Count, y_hat_avg.Holt_linear))

print(rms)

RMSE = 43.0562596115
```

We can see that this method maps the trend accurately and hence provides a better solution when compared with above models. We can still tune the parameters to get even a better model.

Method 6 – Holt-Winters Method

So let's introduce a new term which will be used in this algorithm. Consider a hotel located on a hill station. It experiences high visits during the summer season whereas the visitors during the rest of the year are comparatively very less. Hence the profit earned by the owner will be far better in summer season than in any other season. This pattern will repeat itself every year. Such a repetition is called Seasonality. Datasets which show a similar set of pattern after fixed intervals of a time period suffer from seasonality.



source

The above mentioned models don't take into account the seasonality of the dataset while forecasting. Hence we need a method that takes into account both trend and seasonality to forecast future prices. One such algorithm that we can use in such a scenario is Holt's Winter method. The idea behind triple exponential smoothing(Holt's Winter) is to apply exponential smoothing to the seasonal components in addition to level and trend.

Using Holt's winter method will be the best option among the rest of the models beacuse of the seasonality factor. The Holt-Winters seasonal method comprises the forecast equation and three smoothing equations — one for the level ℓ_t, one for trend b_t and one for the seasonal component denoted by s_t, with smoothing parameters α, β and γ.

$$\begin{aligned}
\text{level} \quad L_t &= \alpha(y_t - S_{t-s}) + (1-\alpha)(L_{t-1} + b_{t-1}); \\
\text{trend} \quad b_t &= \beta(L_t - L_{t-1}) + (1-\beta)b_{t-1}, \\
\text{seasonal} \quad S_t &= \gamma(y_t - L_t) + (1-\gamma)S_{t-s} \\
\text{forecast } F_{t+k} &= L_t + kb_t + S_{t+k-s},
\end{aligned}$$

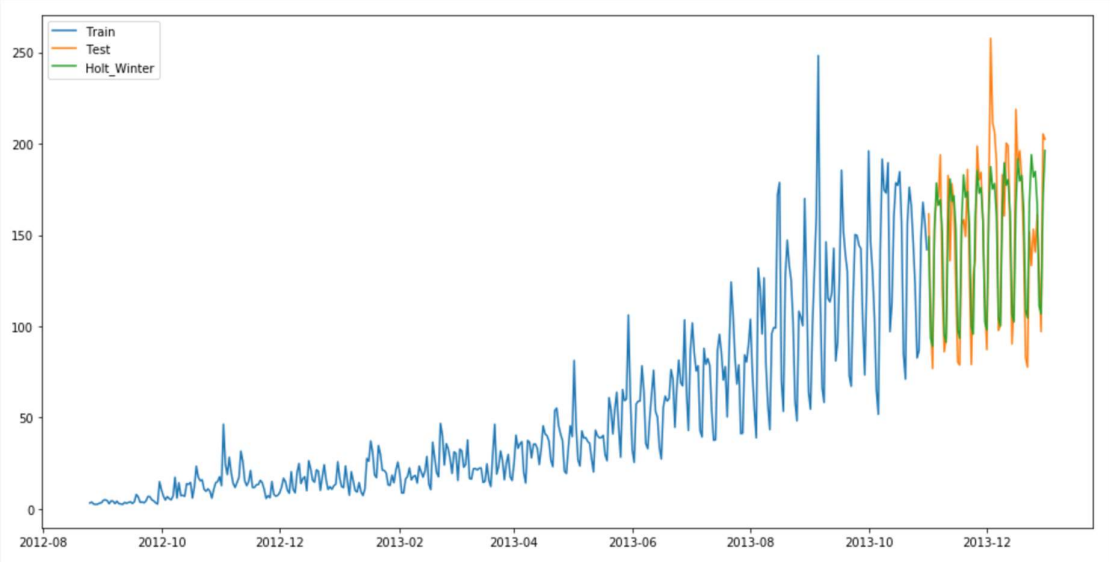
source

where s is the length of the seasonal cycle, for $0 \le \alpha \le 1$, $0 \le \beta \le 1$ and $0 \le \gamma \le 1$.

The level equation shows a weighted average between the seasonally adjusted observation and the non-seasonal forecast for time t. The trend equation is identical to Holt's linear method. The seasonal equation shows a weighted average between the current seasonal index, and the seasonal index of the same season last year (i.e., s time periods ago).

In this method also, we can implement both additive and multiplicative technique. The additive method is preferred when the seasonal variations are roughly constant through the series, while the multiplicative method is preferred when the seasonal variations are changing proportional to the level of the series.

```
y_hat_avg = test.copy()

fit1 = ExponentialSmoothing(np.asarray(train['Count']) ,seasonal_periods=7 ,trend='add', seas
onal='add',).fit()

y_hat_avg['Holt_Winter'] = fit1.forecast(len(test))

plt.figure(figsize=(16,8))

plt.plot( train['Count'], label='Train')

plt.plot(test['Count'], label='Test')

plt.plot(y_hat_avg['Holt_Winter'], label='Holt_Winter')

plt.legend(loc='best')

plt.show()
```



We will now calculate RMSE to check to accuracy of our model.

```
rms = sqrt(mean_squared_error(test.Count, y_hat_avg.Holt_Winter))

print(rms)
```

RMSE = 23.9614925662

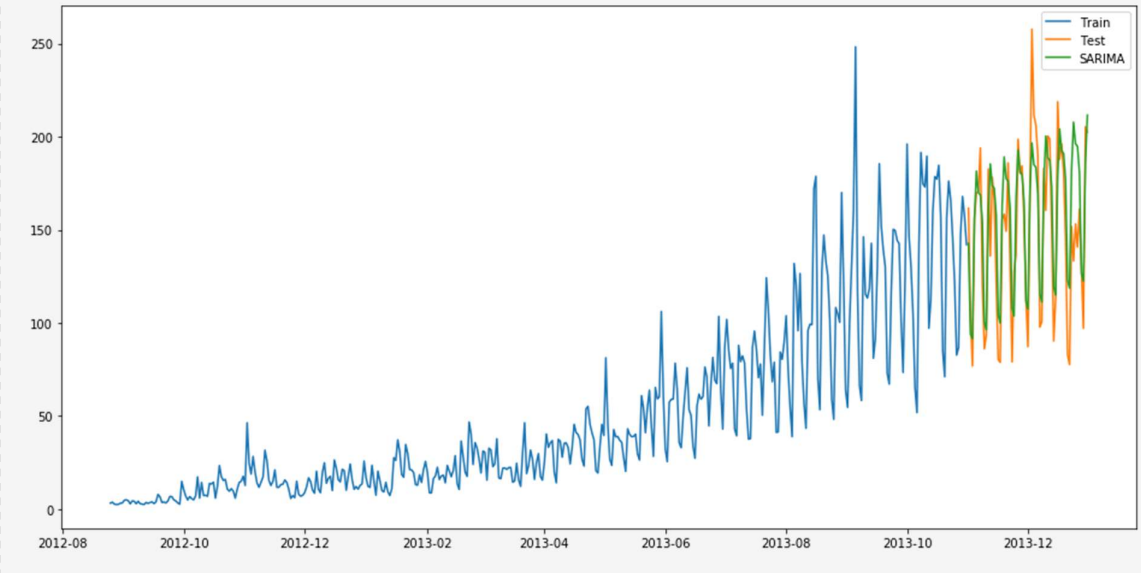
We can see from the graph that mapping correct trend and seasonality provides a far better solution. We chose seasonal_period = 7 as data repeats itself weekly. Other parameters can be tuned as per the dataset. I have used default parameters while building this model. You can tune the parameters to achieve a better model.

Method 7 – ARIMA

Another common Time series model that is very popular among the Data scientists is ARIMA. It stand for **Autoregressive Integrated Moving average**. While exponential smoothing models were based on a description of trend and seasonality in the data, ARIMA models aim to describe the correlations in the data with each other. An improvement over ARIMA is Seasonal ARIMA. It takes into account the seasonality of dataset just like Holt' Winter

method. You can study more about ARIMA and Seasonal ARIMA models and it's pre-processing from these articles (1) and (2).

```
y_hat_avg = test.copy()

fit1 = sm.tsa.statespace.SARIMAX(train.Count, order=(2, 1, 4),seasonal_order=(0,1,1,7)).fit()

y_hat_avg['SARIMA'] = fit1.predict(start="2013-11-1", end="2013-12-31", dynamic=True)

plt.figure(figsize=(16,8))

plt.plot( train['Count'], label='Train')

plt.plot(test['Count'], label='Test')

plt.plot(y_hat_avg['SARIMA'], label='SARIMA')

plt.legend(loc='best')

plt.show()
```



We will now calculate RMSE to check to accuracy of our model.

```
rms = sqrt(mean_squared_error(test.Count, y_hat_avg.SARIMA))

print(rms)
```

RMSE = 26.035582877

We can see that using Seasonal ARIMA generates a similar solution as of Holt's Winter. We chose the parameters as per the ACF and PACF graphs. You can learn more about them from the links provided above. If you face any difficulty finding the parameters of ARIMA model, you can use **auto.arima** implemented in R language. A substitute of auto.arima in Python can be viewed here.

We can compare these models on the basis of their RMSE scores.

Model	RMSE
Naive Method	43.9
Simple Average	109.5
Moving Average	46.72
Simple Exponential smoothing	43.35
Holt's linear Trend	43.05
Holt's Winter	23.96
ARIMA	26.06

End Notes

I hope this article was helpful and now you'd be comfortable in solving similar Time series problems. I suggest you take different kinds of problem statements and take your time to solve them using the above-mentioned techniques. Try these models and find which model works best on which kind of Time series data.

One lesson to learn from these steps is that each of these models can outperform others on a particular dataset. Therefore it doesn't mean that one model which performs best on one type of dataset will perform the same for all others too.

.