A Hands-On Introduction to Time Series Classification (with Python Code)

analyticsvidhya.com/blog/2019/01/introduction-time-series-classification

Aishwarya Singh An avid reader and blogger who loves exploring the endless world of data science and artificial intelligence. Fascinated by the limitless applications of ML and AI; eager to learn and discover the depths of data science.

January 7, 2019

Introduction

Classifying time series data? Is that really possible? What could potentially be the use of doing that? These are just some of the questions you must have had when you read the title of this article. And it's only fair – I had the exact same thoughts when I first came across this concept!

The time series data most of us are exposed to deals primarily with generating forecasts. Whether that's predicting the demand or sales of a product, the count of passengers in an airline or the closing price of a particular stock, we are used to leveraging tried and tested time series techniques for forecasting requirements.

But as the amount of data being generated increases exponentially, so does the opportunity to experiment with new ideas and algorithms. Working with complex time series datasets is still a niche field, and it's always helpful to expand your repertoire to include new ideas.

And that is what I aim to do in the article by introducing you to the novel concept of time series classification. We will first understand what this topic means and it's applications in the industry. But we won't stop at the theory part – we'll get our hands dirty by working on a time series dataset and performing binary time series classification. Learning by doing – this will help you understand the concept in a practical manner as well.

If you have not worked on a time series problem before, I highly recommend first starting with some basic forecasting. You can go through the below article for starters:

> A comprehensive beginner's guide to create a Time Series Forecast (with Codes in Python)

Table of contents

Introduction to Time Series Classification

Time series classification has actually been around for a while. But it has so far mostly been limited to research labs, rather than industry applications. But there is a lot of research going on, new datasets being created and a number of new algorithms being proposed. When I first came across this time series classification concept, my initial thought was – how can we classify a time series and what does a time series classification data look like? I'm sure you must be wondering the same thing.

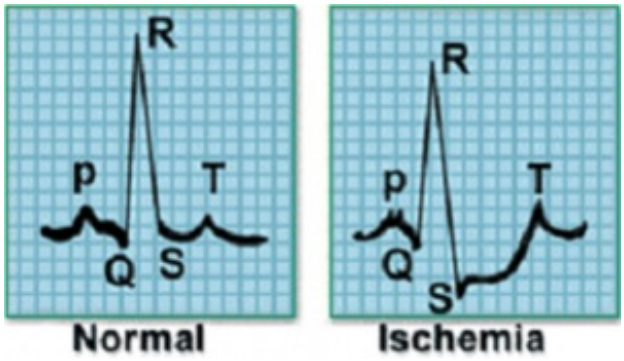
As you can imagine, time series classification data differs from a regular classification problem since the attributes have an ordered sequence. Let's have a look at some time series classification use cases to understand this difference.

1) Classifying ECG/EEG signals

ECG, or electrocardiogram, records the electrical activity of the heart and is widely be used to diagnose various heart problems. These ECG signals are captured using external electrodes.

For example, consider the following signal sample which represents the electrical activity for one heartbeat. The image on the left represents a normal heartbeat while the one adjacent to it represents a Myocardial Infarction.

The data captured from the electrodes will be in time series form, and the signals can be classified into different classes. We can also classify EEG signals which record the electrical activity of the brain.



2) Image Classification

Images can also be in a sequential time-dependent format. Consider the following scenario:

Crops are grown in a particular field depending upon the weather conditions, soil fertility, availability of water and other external factors. A picture of this field is taken daily for 5 years and labeled with the name of the crop planted on the field. Do you see where I'm going with this? The images in the dataset are taken after a fixed time interval and have a defined sequence, which can be an important factor in classifying the images.

3) Classifying Motion Sensor Data

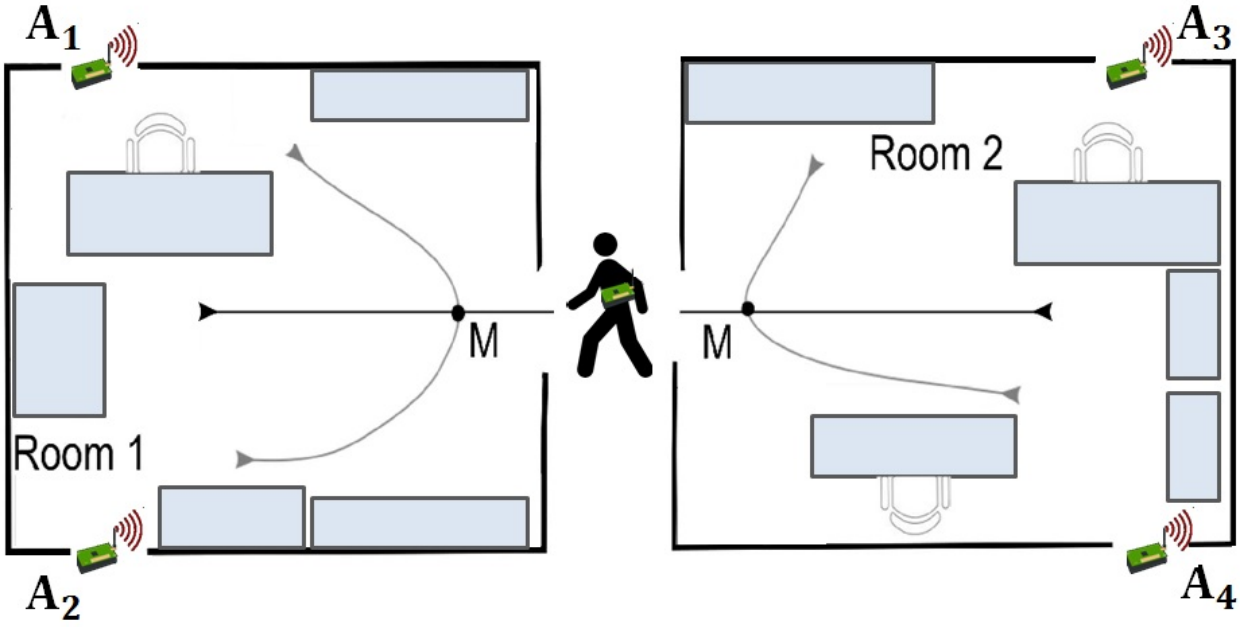
Sensors generate high-frequency data that can identify the movement of objects in their range. By setting up multiple wireless sensors and observing the change in signal strength in the sensors, we can identify the object's direction of movement.

What other applications can you think of where we can apply time series classification? Let me know in the comments section below the article.

Setting up the Problem Statement

We will be working on the '*Indoor User Movement Prediction*' problem. In this challenge, multiple motion sensors are placed in different rooms and the goal is to identify whether an individual has moved across rooms, based on the frequency data captured from these motion sensors.

There are four motion sensors (A1, A2, A3, A4) placed across two rooms. Have a look at the below image which illustrates where the sensors are positioned in each room. The setup in these two rooms was created in 3 different pairs of rooms (group1, group2, group3).



A person can move along any of the six pre-defined paths shown in the above image. If a person walks on path 2, 3, 4 or 6, he moves within the room. On the other hand, if a person follows path 1 or path 5, we can say that the person has moved between the rooms.

The sensor reading can be used to identify the position of a person at a given point in time. As the person moves in the room or across rooms, the reading in the sensor changes. This change can be used to identify the path of the person.

Now that the problem statement is clear, it's time to get down to coding! In the next section, we will look at the dataset for the problem which should help clear up any lingering questions you might have on this statement. You can download the dataset from this link: **Indoor User Movement Prediction.**

Reading and Understanding the Data

Our dataset comprises of 316 files:

- 314 ***MovementAAL*** csv files containing the readings from motion sensors placed in the environment
- A ***Target*** csv file that contains the target variable for each MovementAAL file
- One ***Group Data*** csv file to identify which MovementAAL file belongs to which setup group
- The ***Path*** csv file that contains the path which the object took

Let's have a look at the datasets. We'll start by importing the necessary libraries.

```
import pandas as pd
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
from os import listdir

from keras.preprocessing import sequence
import tensorflow as tf
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM

from keras.optimizers import Adam
from keras.models import load_model
from keras.callbacks import ModelCheckpoint
```

Before loading all the files, let's take a quick sneak peek into the data we are going to deal with. Reading the first two files from the movement data:

```
df1 = pd.read_csv('/MovementAAL/dataset/MovementAAL_RSS_1.csv')
df2 = pd.read_csv('/MovementAAL/dataset/MovementAAL_RSS_2.csv')
```

```
df1.head()
```

	#RSS_anchor1	RSS_anchor2	RSS_anchor3	RSS_anchor4
0	-0.90476	-0.48	0.28571	0.30
1	-0.57143	-0.32	0.14286	0.30
2	-0.38095	-0.28	-0.14286	0.35
3	-0.28571	-0.20	-0.47619	0.35
4	-0.14286	-0.20	0.14286	-0.20

```
df2.head()
```

	#RSS_anchor1	RSS_anchor2	RSS_anchor3	RSS_anchor4
0	-0.57143	-0.20	0.71429	0.50
1	-0.76190	-0.48	0.76190	-0.25
2	-0.85714	-0.60	0.85714	0.55
3	-0.76190	-0.40	0.71429	0.60
4	-0.76190	-0.84	0.85714	0.45

```
df1.shape, df2.shape
```

```
((27, 4), (26, 4))
```

The files contain normalized data from the four sensors – A1, A2, A3, A4. The length of the csv files (number of rows) vary, since the data corresponding to each csv is for a different duration. To simplify things, let us suppose the sensor data is collected every

second. The first reading was for a duration of 27 seconds (so 27 rows), while another reading was for 26 seconds (so 26 rows).

We will have to deal with this varying length before we build our model. For now, we will read and store the values from the sensors in a list using the following code block:

```
path = 'MovementAAL/dataset/MovementAAL_RSS_'
sequences = list()
for i in range(1,315):
    file_path = path + str(i) + '.csv'
    print(file_path)
    df = pd.read_csv(file_path, header=0)
    values = df.values
    sequences.append(values)


targets = pd.read_csv('MovementAAL/dataset/MovementAAL_target.csv')
targets = targets.values[:,1]
```

We now have a list *'sequences'* that contains the data from the motion sensors and 'targets' which holds the labels for the csv files. When we print *sequences[0]*, we get the values of sensors from the first csv file:

```
sequences[0]
```

```
array([[-0.90476 , -0.48    ,  0.28571 ,  0.3     ],
       [-0.57143 , -0.32    ,  0.14286 ,  0.3     ],
       [-0.38095 , -0.28    , -0.14286 ,  0.35    ],
       [-0.28571 , -0.2     , -0.47619 ,  0.35    ],
       [-0.14286 , -0.2     ,  0.14286 , -0.2     ],
       [-0.14286 , -0.2     ,  0.047619,  0.      ],
       [-0.14286 , -0.16    , -0.38095 ,  0.2     ],
       [-0.14286 , -0.04    , -0.61905 , -0.2     ],
       [-0.095238, -0.08    ,  0.14286 , -0.55    ],
       [-0.047619,  0.04    , -0.095238,  0.05    ],
       [-0.19048 , -0.04    ,  0.095238,  0.4     ],
       [-0.095238, -0.04    , -0.14286 ,  0.35    ],
       [-0.33333 , -0.08    , -0.28571 , -0.2     ],
       [-0.2381  ,  0.04    ,  0.14286 ,  0.35    ],
       [ 0.      ,  0.08    ,  0.14286 ,  0.05    ],
       [-0.095238,  0.04    ,  0.095238,  0.1     ],
       [-0.14286 , -0.2     ,  0.14286 ,  0.5     ],
       [-0.19048 ,  0.04    , -0.42857 ,  0.3     ],
       [-0.14286 , -0.08    , -0.2381  ,  0.15    ],
       [-0.33333 ,  0.16    , -0.14286 , -0.8     ],
       [-0.42857 ,  0.16    , -0.28571 , -0.1     ],
       [-0.71429 ,  0.16    , -0.28571 ,  0.2     ],
       [-0.095238, -0.08    ,  0.095238,  0.35    ],
       [-0.28571 ,  0.04    ,  0.14286 ,  0.2     ],
       [ 0.      ,  0.04    ,  0.14286 ,  0.1     ],
       [ 0.      ,  0.04    , -0.047619, -0.05    ],
       [-0.14286 , -0.6     , -0.28571 , -0.1     ]])
```

As mentioned previously, the dataset was collected in three different pairs of rooms – hence three groups. This information can be used to divide the dataset into train, test and validation sets. We will load the ***DatasetGroup*** csv file now:

```
groups = pd.read_csv('MovementAAL/groups/MovementAAL_DatasetGroup.csv', header=0)
groups = groups.values[:,1]
```

We will take the data from the first two sets for training purposes and the third group for testing.

Preprocessing Steps

Since the time series data is of varying length, we cannot directly build a model on this dataset. So how can we decide the ideal length of a series? There are multiple ways in which we can deal with it and here are a few ideas (I would love to hear your suggestions in the comment section):

- Pad the shorter sequences with zeros to make the length of all the series equal. In this case, we will be feeding incorrect data to the model
- Find the maximum length of the series and pad the sequence with the data in the last row
- Identify the minimum length of the series in the dataset and truncate all the other series to that length. However, this will result in a huge loss of data
- Take the mean of all the lengths, truncate the longer series, and pad the series which are shorter than the mean length

Let's find out the minimum, maximum and mean length:

```
len_sequences = []
for one_seq in sequences:
    len_sequences.append(len(one_seq))
pd.Series(len_sequences).describe()
```

```
count    314.000000
mean      42.028662
std       16.185303
min       19.000000
25%       26.000000
50%       41.000000
75%       56.000000
max      129.000000
dtype: float64
```

Most of the files have lengths between 40 to 60. Just 3 files are coming up with a length more than 100. Thus, taking the minimum or maximum length does not make much sense. The 90th quartile comes out to be 60, which is taken as the length of sequence for the data. Let's code it out:

```
#Padding the sequence with the values in last row to max length
to_pad = 129
new_seq = []
for one_seq in sequences:
    len_one_seq = len(one_seq)
    last_val = one_seq[-1]
    n = to_pad - len_one_seq

    to_concat = np.repeat(one_seq[-1], n).reshape(4, n).transpose()
    new_one_seq = np.concatenate([one_seq, to_concat])
    new_seq.append(new_one_seq)
final_seq = np.stack(new_seq)

#truncate the sequence to length 60
from keras.preprocessing import sequence
seq_len = 60
final_seq=sequence.pad_sequences(final_seq, maxlen=seq_len, padding='post', dtype='float',
truncating='post')
```

Now that the dataset is prepared, we will separate it based on the groups. Preparing the train, validation and test sets:

```
train = [final_seq[i] for i in range(len(groups)) if (groups[i]==2)]
validation = [final_seq[i] for i in range(len(groups)) if groups[i]==1]
test = [final_seq[i] for i in range(len(groups)) if groups[i]==3]

train_target = [targets[i] for i in range(len(groups)) if (groups[i]==2)]
validation_target = [targets[i] for i in range(len(groups)) if groups[i]==1]
test_target = [targets[i] for i in range(len(groups)) if groups[i]==3]

train = np.array(train)
validation = np.array(validation)
test = np.array(test)

train_target = np.array(train_target)
train_target = (train_target+1)/2

validation_target = np.array(validation_target)
validation_target = (validation_target+1)/2

test_target = np.array(test_target)
test_target = (test_target+1)/2
```

Building a Time Series Classification model

We have prepared the data to be used for an LSTM (Long Short Term Memory) model. We dealt with the variable length sequence and created the train, validation and test sets. Let's build a single layer LSTM network.

Note: You can get acquainted with LSTMs in this wonderfully explained tutorial. I would advice you to go through that first as it'll help you understand how the below code works.

```
model = Sequential()
model.add(LSTM(256, input_shape=(seq_len, 4)))
model.add(Dense(1, activation='sigmoid'))

model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 256)               267264
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 257
=================================================================
Total params: 267,521
Trainable params: 267,521
Non-trainable params: 0
_________________________________________________________________
```

We will now train the model and monitor the validation accuracy:

```
adam = Adam(lr=0.001)
chk = ModelCheckpoint('best_model.pkl', monitor='val_acc', save_best_only=True, mode='max',
verbose=1)
model.compile(loss='binary_crossentropy', optimizer=adam, metrics=['accuracy'])
model.fit(train, train_target, epochs=200, batch_size=128, callbacks=[chk], validation_data=
(validation,validation_target))


#loading the model and checking accuracy on the test data
model = load_model('best_model.pkl')

from sklearn.metrics import accuracy_score
test_preds = model.predict_classes(test)
accuracy_score(test_target, test_preds)
```

I got an accuracy score of 0.7884615384615384. It's quite a promising start but we can definitely improve the performance of the LSTM model by playing around with the hyperparameters, changing the learning rate, and/or the number of epochs as well.

End Notes

That brings us to the end of this tutorial. The idea behind penning this down was to introduce you to a whole new world in the time series spectrum in a practical manner.

A Multivariate Time Series Guide to Forecasting and Modeling (with Python codes)

analyticsvidhya.com/blog/2018/09/multivariate-time-series-guide-forecasting-modeling-python-codes

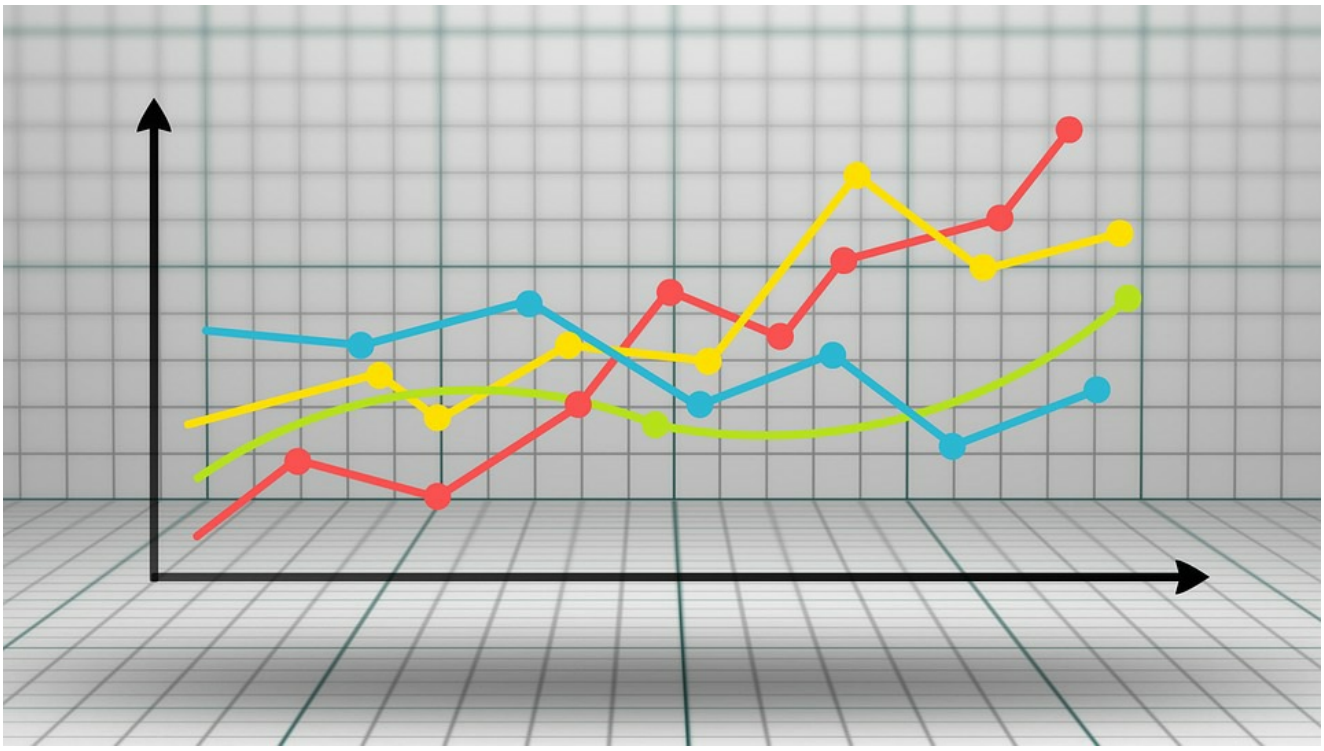
Aishwarya Singh An avid reader and blogger who loves exploring the endless world of data science and artificial intelligence. Fascinated by the limitless applications of ML and AI; eager to learn and discover the depths of data science.

September 27, 2018

Introduction

Time is the most critical factor that decides whether a business will rise or fall. That's why we see sales in stores and e-commerce platforms aligning with festivals. These businesses analyze years of spending data to understand the best time to throw open the gates and see an increase in consumer spending.

But how can you, as a data scientist, perform this analysis? Don't worry, you don't need to build a time machine! Time Series modeling is a powerful technique that acts as a gateway to understanding and forecasting trends and patterns.



But even a time series model has different facets. Most of the examples we see on the web deal with univariate time series. Unfortunately, real-world use cases don't work like that. There are multiple variables at play, and handling all of them at the same time is where a data scientist will earn his worth.

In this article, we will understand what a multivariate time series is, and how to deal with it. We will also take a case study and implement it in Python to give you a practical understanding of the subject.

Table of contents

1. Univariate versus Multivariate Time Series

This article assumes some familiarity with univariate time series, its properties and various techniques used for forecasting. Since this article will be focused on multivariate time series, I would suggest you go through the following articles which serve as a good introduction to univariate time series:

But I'll give you a quick refresher of what a univariate time series is, before going into the details of a multivariate time series. Let's look at them one by one to understand the difference.

1.1 Univariate Time Series

A univariate time series, as the name suggests, is a series with a single time-dependent variable.

For example, have a look at the sample dataset below that consists of the temperature values (each hour), for the past 2 years. Here, temperature is the dependent variable (dependent on Time).

If we are asked to predict the temperature for the next few days, we will look at the past values and try to gauge and extract a pattern. We would notice that the temperature is lower in the morning and at night, while peaking in the afternoon. Also if you have data for the past few years, you would observe that it is colder during the months of November to January, while being comparatively hotter in April to June.

Such observations will help us in predicting future values. Did you notice that we used only one variable (the temperature of the past 2 years,)? Therefore, this is called Univariate Time Series Analysis/Forecasting.

Time	Temperature
5:00 am	59 °F
6:00 am	59 °F
7:00 am	58 °F
8:00 am	58 °F
9:00 am	60 °F
10:00 am	62 °F
11:00 am	64 °F
12:00 pm	66 °F
1:00 pm	67 °F
2:00 pm	69 °F
3:00 pm	71 °F
4:00 pm	71 °F
5:00 pm	71 °F
6:00 pm	69 °F
7:00 pm	68 °F
8:00 pm	65 °F
9:00 pm	64 °F

1.2 Multivariate Time Series (MTS)

A Multivariate time series has more than one time-dependent variable. Each variable depends not only on its past values but also has some dependency on other variables. This dependency is used for forecasting future values. Sounds complicated? Let me explain.

Consider the above example. Now suppose our dataset includes perspiration percent, dew point, wind speed, cloud cover percentage, etc. along with the temperature value for the past two years. In this case, there are multiple variables to be considered to optimally predict temperature. A series like this would fall under the category of multivariate time series. Below is an illustration of this:

Time	Temperature	cloud cover	dew point	humidity	wind
5:00 am	59 °F	97%	51 °F	74%	8 mph SSE
6:00 am	59 °F	89%	51 °F	75%	8 mph SSE
7:00 am	58 °F	79%	51 °F	76%	7 mph SSE
8:00 am	58 °F	74%	51 °F	77%	7 mph S
9:00 am	60 °F	74%	51 °F	74%	7 mph S
10:00 am	62 °F	74%	52 °F	70%	8 mph S
11:00 am	64 °F	76%	52 °F	65%	8 mph SSW
12:00 pm	66 °F	80%	52 °F	60%	8 mph SSW
1:00 pm	67 °F	78%	52 °F	58%	10 mph SW
2:00 pm	69 °F	71%	52 °F	54%	10 mph SW
3:00 pm	71 °F	75%	52 °F	52%	11 mph SW
4:00 pm	71 °F	78%	52 °F	52%	11 mph SW
5:00 pm	71 °F	78%	52 °F	52%	12 mph SW
6:00 pm	69 °F	78%	52 °F	54%	11 mph SW
7:00 pm	68 °F	87%	53 °F	60%	12 mph SW
8:00 pm	65 °F	100%	54 °F	66%	11 mph SSW
9:00 pm	64 °F	100%	55 °F	72%	13 mph SSW

Now that we understand what a multivariate time series looks like, let us understand how can we use it to build a forecast.

2. Dealing with a Multivariate Time Series – VAR

In this section, I will introduce you to one of the most commonly used methods for multivariate time series forecasting – **Vector Auto Regression (VAR)**.

In a VAR model, each variable is a linear function of the past values of itself and the past values of all the other variables. To explain this in a better manner, I'm going to use a simple visual example:

We have two variables, y1 and y2. We need to forecast the value of these two variables at time t, from the given data for past n values. For simplicity, I have considered the lag value to be 1.

For calculating y1(t), we will use the past value of y1 and y2. Similarly, to calculate y2(t), past values of both y1 and y2 will be used. Below is a simple mathematical way of representing this relation:

Variable y1	Variable y2
$y1_{t-n}$	$y2_{t-n}$
...	...
$Y1_{t-2}$	$Y2_{t-2}$
$Y1_{t-1}$	$Y2_{t-1}$
$y1_t$	$y2_t$

Variable y1	Variable y2
$y1_{t-n}$	$y2_{t-n}$
...	...
$Y1_{t-2}$	$Y2_{t-2}$
$Y1_{t-1}$	$Y2_{t-1}$
$y1_t$	$y2_t$

$$y_1(t) = a_1 + w_111 * y_1(t-1) + w_112 * y_2(t-1) + e_1(t-1)$$
$$y_2(t) = a_2 + w_121 * y_1(t-1) + w_122 * y_2(t-1) + e_2(t-1)$$

Here,

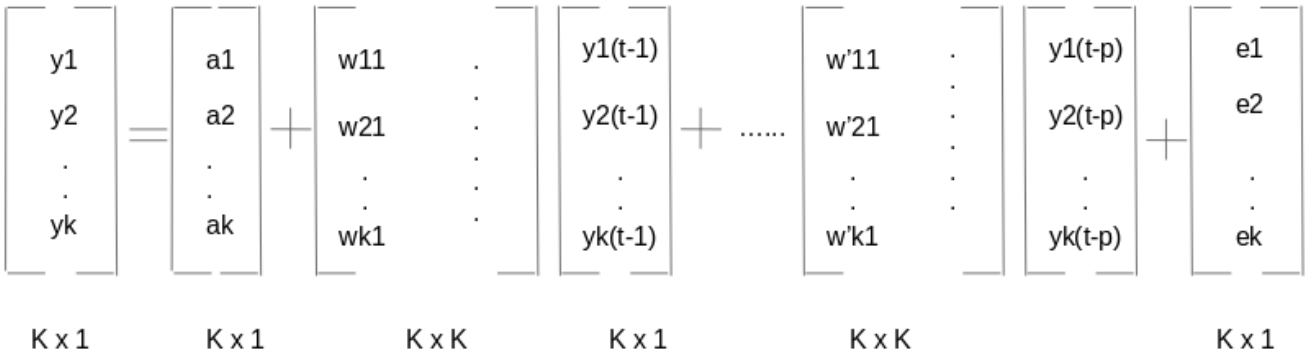
- a1 and a2 are the constant terms,
- w11, w12, w21, and w22 are the coefficients,
- e1 and e2 are the error terms

These equations are similar to the equation of an AR process. Since the AR process is used for univariate time series data, the future values are linear combinations of their own past values only. Consider the AR(1) process:

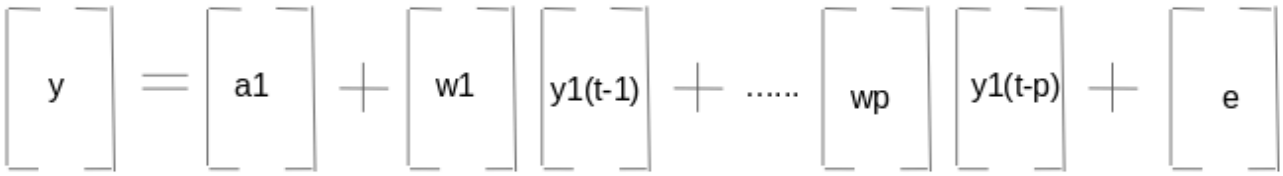
y(t) = a + w*y(t-1) +e

In this case, we have only one variable – y, a constant term – a, an error term – e, and a coefficient – w. In order to accommodate the multiple variable terms in each equation for VAR, we will use vectors. We can write the equations (1) and (2) in the following form :

The two variables are y1 and y2, followed by a constant, a coefficient metric, lag value, and an error metric. This is the vector equation for a VAR(1) process. For a VAR(2) process, another vector term for time (t-2) will be added to the equation to generalize for p lags:



The above equation represents a VAR(p) process with variables y1, y2 ...yk. The same can be written as:



$$y(t) = a + w_1 * y(t-1) + ... + w_p * y(t-p) + \varepsilon t$$

The term ε_t in the equation represents multivariate vector white noise. For a multivariate time series, ε_t should be a continuous random vector that satisfies the following conditions:

1. $E(\varepsilon_t) = 0$
   Expected value for the error vector is 0
2. $E(\varepsilon_{t1},\varepsilon_{t2}') = \sigma_{12}$
   Expected value of ε_t and ε_t' is the standard deviation of the series

3. Why Do We Need VAR?

Recall the temperate forecasting example we saw earlier. An argument can be made for it to be treated as a multiple univariate series. We can solve it using simple univariate forecasting methods like AR. Since the aim is to predict the temperature, we can simply remove the other variables (except temperature) and fit a model on the remaining univariate series.

Another simple idea is to forecast values for each series individually using the techniques we already know. This would make the work extremely straightforward! Then why should you learn another forecasting technique? Isn't this topic complicated enough already?

From the above equations (1) and (2), it is clear that each variable is using the past values of every variable to make the predictions. Unlike AR, **VAR is able to understand and use the relationship between several variables**. This is useful for describing the dynamic behavior of the data and also provides better forecasting results. Additionally, implementing VAR is as simple as using any other univariate technique (which you will see in the last section).

4. Stationarity of a Multivariate Time Series

We know from studying the univariate concept that a stationary time series will more often than not give us a better set of predictions. If you are not familiar with the concept of stationarity, please go through this article first: A Gentle Introduction to handling non-stationary Time Series.

To summarize, for a given univariate time series:

y(t) = c*y(t-1) + ε t

The series is said to be stationary if the value of |c| < 1. Now, recall the equation of our VAR process:

$$Iy(t) = a + w_1 * y(t-1) + w_2 * y(t-2) + ... + w_p * y(t-p) + \varepsilon$$

Note: I is the identity matrix.

Representing the equation in terms of Lag operators, we have:

$$Iy(t) = a + w_1 * L^1 y(t) + w_2 L^2 * y(t) + ... + w_p * L^p y(t) + \varepsilon$$

Taking all the y(t) terms on the left-hand side:

$$Iy(t) - w_1 * L^1 y(t) - w_2 L^2 * y(t) - ... - w_p * L^p y(t) = a + e$$

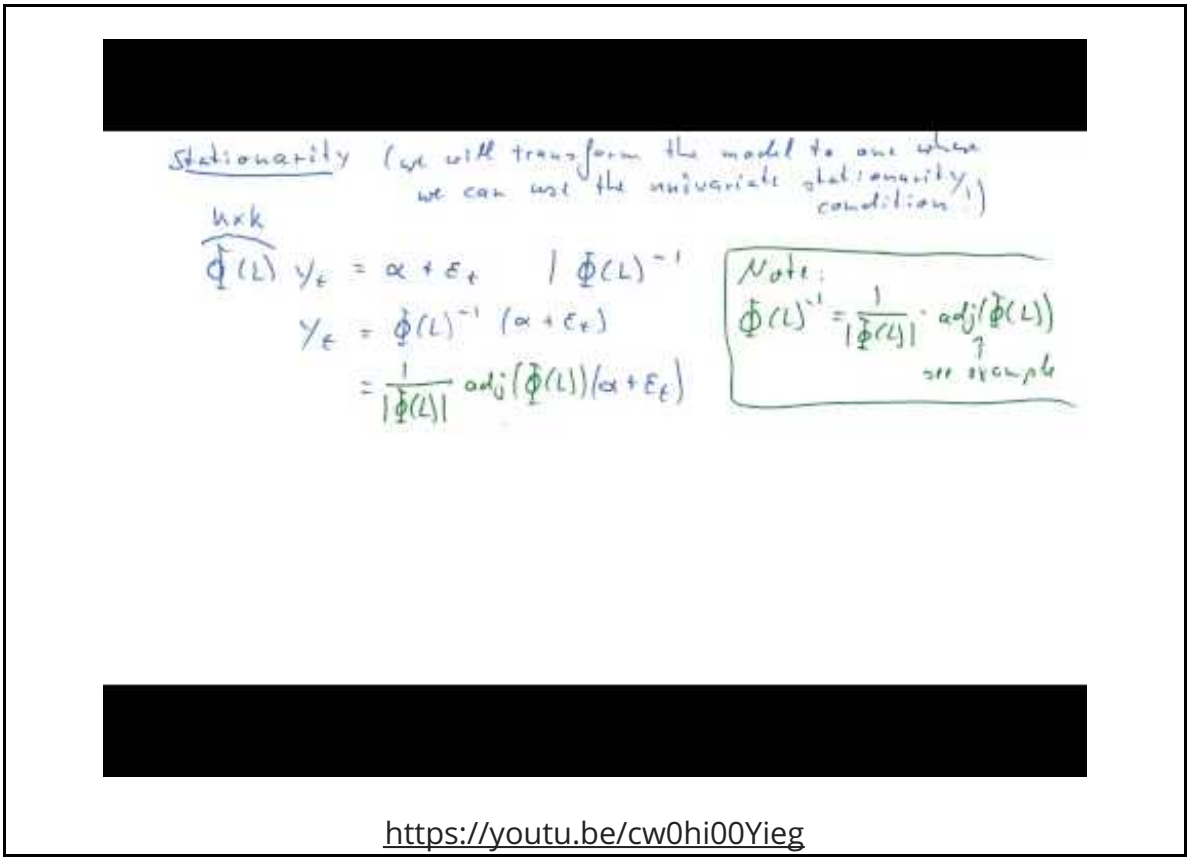
$$(I - w_1 * L^1 - w_2 L^2 * -... - w_p * L^p) y(t) = a + e$$

The coefficient of y(t) is called the lag polynomial. Let us represent this as Φ(L):

$$\Phi(L)y(t) = a + \epsilon$$

For a series to be stationary, the eigenvalues of $|\Phi(L)^{-1}|$ should be less than 1 in modulus. This might seem complicated given the number of variables in the derivation. This idea has been explained using a simple numerical example in the following video. I highly encourage watching it to solidify your understanding:

$$y(t) = \Phi(L)^{-1}(a + e)$$



https://youtu.be/cw0hi00Yieg

Similar to the Augmented Dickey-Fuller test for univariate series, we have Johansen's test for checking the stationarity of any multivariate time series data. We will see how to perform the test in the last section of this article.

5. Train-Validation Split

If you have worked with univariate time series data before, you'll be aware of the train-validation sets. The idea of creating a validation set is to analyze the performance of the model before using it for making predictions.

Creating a validation set for time series problems is tricky because we have to take into account the time component. One cannot directly use the *train_test_split* or *k-fold* validation since this will disrupt the pattern in the series. The validation set should be created considering the date and time values.

Suppose we have to forecast the temperate, dew point, cloud percent, etc. for the next two months using data from the last two years. One possible method is to keep the data for the last two months aside and train the model on the remaining 22 months.

Once the model has been trained, we can use it to make predictions on the validation set. Based on these predictions and the actual values, we can check how well the model performed, and the variables for which the model did not do so well. And for making the final prediction, use the complete dataset (combine the train and validation sets).

6. Python implementation

In this section, we will implement the Vector AR model on a toy dataset. I have used the Air Quality dataset for this and you can download it from here.

```
#import required packages
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

#read the data
df = pd.read_csv("AirQualityUCI.csv", parse_dates=[['Date', 'Time']])

#check the dtypes
df.dtypes
```

```
Date_Time        object
CO(GT)           int64
PT08.S1(CO)      int64
NMHC(GT)         int64
C6H6(GT)         int64
PT08.S2(NMHC)    int64
NOx(GT)          int64
PT08.S3(NOx)     int64
NO2(GT)          int64
PT08.S4(NO2)     int64
PT08.S5(O3)      int64
T                int64
RH               int64
AH               int64
dtype: object
```

The data type of the *Date_Time* column is *object* and we need to change it to *datetime*. Also, for preparing the data, we need the index to have *datetime*. Follow the below commands:

```
df['Date_Time'] = pd.to_datetime(df.Date_Time , format = '%d/%m/%Y %H.%M.%S')
data = df.drop(['Date_Time'], axis=1)
data.index = df.Date_Time
```

The next step is to deal with the missing values. Since the missing values in the data are replaced with a value -200, we will have to impute the missing value with a better number. Consider this – if the present dew point value is missing, we can safely assume that it will be close to the value of the previous hour. Makes sense, right? Here, I will impute -200 with the previous value.

You can choose to substitute the value using the average of a few previous values, or the value at the same time on the previous day (you can share your idea(s) of imputing missing values in the comments section below).

```
#missing value treatment
cols = data.columns
for j in cols:
    for i in range(0,len(data)):
       if data[j][i] == -200:
          data[j][i] = data[j][i-1]

#checking stationarity
from statsmodels.tsa.vector_ar.vecm import coint_johansen
#since the test works for only 12 variables, I have randomly dropped
#in the next iteration, I would drop another and check the eigenvalues
johan_test_temp = data.drop([ 'CO(GT)'], axis=1)
coint_johansen(johan_test_temp,-1,1).eig
```

Below is the result of the test:

```
array([ 0.17806667,  0.1552133 ,  0.1274826 ,  0.12277888,  0.09554265,
       0.08383711,  0.07246919,  0.06337852,  0.04051374,  0.02652395,
       0.01467492,  0.00051835])
```

We can now go ahead and create the validation set to fit the model, and test the performance of the model:

```
#creating the train and validation set
train = data[:int(0.8*(len(data)))]
valid = data[int(0.8*(len(data))):]

#fit the model
from statsmodels.tsa.vector_ar.var_model import VAR

model = VAR(endog=train)
model_fit = model.fit()

# make prediction on validation
prediction = model_fit.forecast(model_fit.y, steps=len(valid))
```

The predictions are in the form of an array, where each list represents the predictions of the row. We will transform this into a more presentable format.

```
#converting predictions to dataframe
pred = pd.DataFrame(index=range(0,len(prediction)),columns=[cols])
for j in range(0,13):
    for i in range(0, len(prediction)):
       pred.iloc[i][j] = prediction[i][j]

#check rmse
for i in cols:
    print('rmse value for', i, 'is : ', sqrt(mean_squared_error(pred[i], valid[i])))
```

Output of the above code:

```
rmse value for CO(GT) is :  1.4200393103392812
rmse value for PT08.S1(CO) is :  303.3909208229375
rmse value for NMHC(GT) is :  204.0662895081472
rmse value for C6H6(GT) is :  28.153391799471244
rmse value for PT08.S2(NMHC) is :  6.538063846286176
rmse value for NOx(GT) is :  265.04913993413805
rmse value for PT08.S3(NOx) is :  250.7673347152554
rmse value for NO2(GT) is :  238.92642219826683
rmse value for PT08.S4(NO2) is :  247.50612831072633
rmse value for PT08.S5(O3) is :  392.3129907890131
rmse value for T is :  383.1344361254454
rmse value for RH is :  506.5847387424092
rmse value for AH is :  8.139735443605728
```

After the testing on validation set, lets fit the model on the complete dataset

```
#make final predictions
model = VAR(endog=data)
model_fit = model.fit()
yhat = model_fit.forecast(model_fit.y, steps=1)
print(yhat)
```

A Gentle Introduction to Handling a Non-Stationary Time Series in Python

analyticsvidhya.com/blog/2018/09/non-stationary-time-series-python

Aishwarya Singh An avid reader and blogger who loves exploring the endless world of data science and artificial intelligence. Fascinated by the limitless applications of ML and AI; eager to learn and discover the depths of data science.

September 13, 2018

Introduction

What do these applications have in common: predicting the electricity consumption of a household for the next three months, estimating traffic on roads at certain periods, and predicting the price at which a stock will trade on the New York Stock Exchange?

They all fall under the concept of time series data! You cannot accurately predict any of these results without the 'time' component. And as more and more data is generated in the world around us, time series forecasting keeps becoming an ever more critical technique for a data scientist to master.

But time series is a complex topic with multiple facets at play simultaneously.

For starters, making the time series stationary is critical if we want the forecasting model to work well. Why? Because most of the data you collect will have non-stationary trends. And if the spikes are erratic how can you be sure the model will work properly?

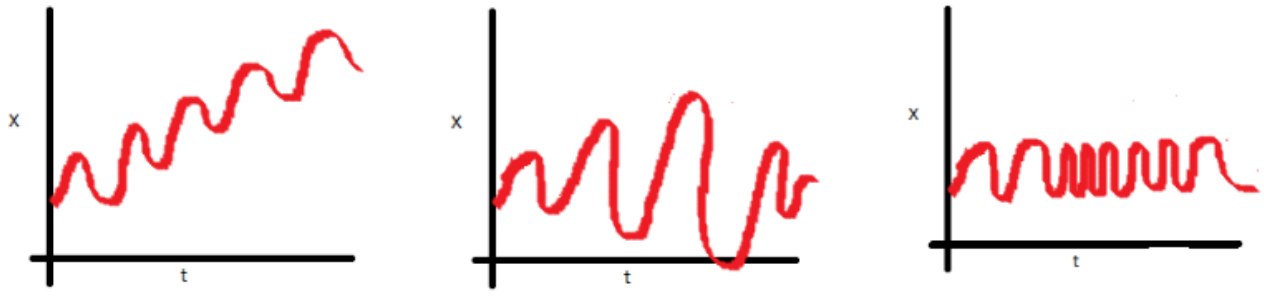
The focus of this article is on the methods for checking stationarity in time series data. This article assumes that the reader is familiar with time series, ARIMA, and the concept of stationarity. Below are some references to brush up on the basics:

Table of contents

1. Introduction to Stationarity

'Stationarity' is one of the most important concepts you will come across when working with time series data. **A stationary series is one in which the properties – mean, variance and covariance, do not vary with time.**

Let us understand this using an intuitive example. Consider the three plots shown below:



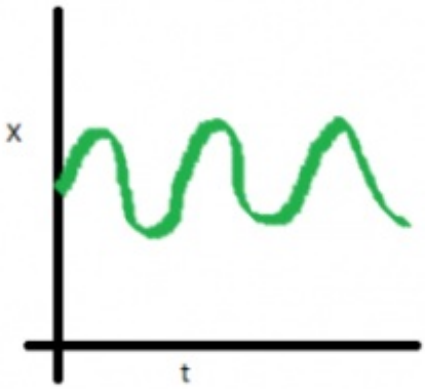
- In the first plot, we can clearly see that the mean varies (increases) with time which results in an upward trend. Thus, this is a non-stationary series. **For a series to be**

classified as stationary, it should not exhibit a trend.

- Moving on to the second plot, we certainly do not see a trend in the series, but the variance of the series is a function of time. As mentioned previously, a stationary series must have a constant variance.
- If you look at the third plot, the spread becomes closer as the time increases, which implies that the covariance is a function of time.

The three examples shown above represent non-stationary time series. Now look at a fourth plot:

In this case, the mean, variance and covariance are constant with time. This is what a stationary time series looks like.



Think about this for a second – predicting future values using which of the above plots would be easier? The fourth plot, right? Most statistical models require the series to be stationary to make effective and precise predictions.

So to summarize, a stationary time series is the one for which the properties (namely mean, variance and covariance) do not depend on time. In the next section we will cover various methods to check if the given series is stationary or not.

2. Loading the Data

In this and the next few sections, I will be introducing methods to check the stationarity of time series data and the techniques required to deal with any non-stationary series. I have also provided the python code for applying each technique. **You can download the dataset we'll be using from this link: AirPassengers.**

Before we go ahead and analyze our dataset, let's load and preprocess the data first.

```
#loading important libraries
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

#reading the dataset
train = pd.read_csv('AirPassengers.csv')

#preprocessing
train.timestamp = pd.to_datetime(train.Month , format = '%Y-%m')
train.index = train.timestamp
train.drop('Month',axis = 1, inplace = True)

#looking at the first few rows
#train.head()
```

1949-01-01	112
1949-02-01	118
1949-03-01	132
1949-04-01	129
1949-05-01	121

Looks like we are good to go!

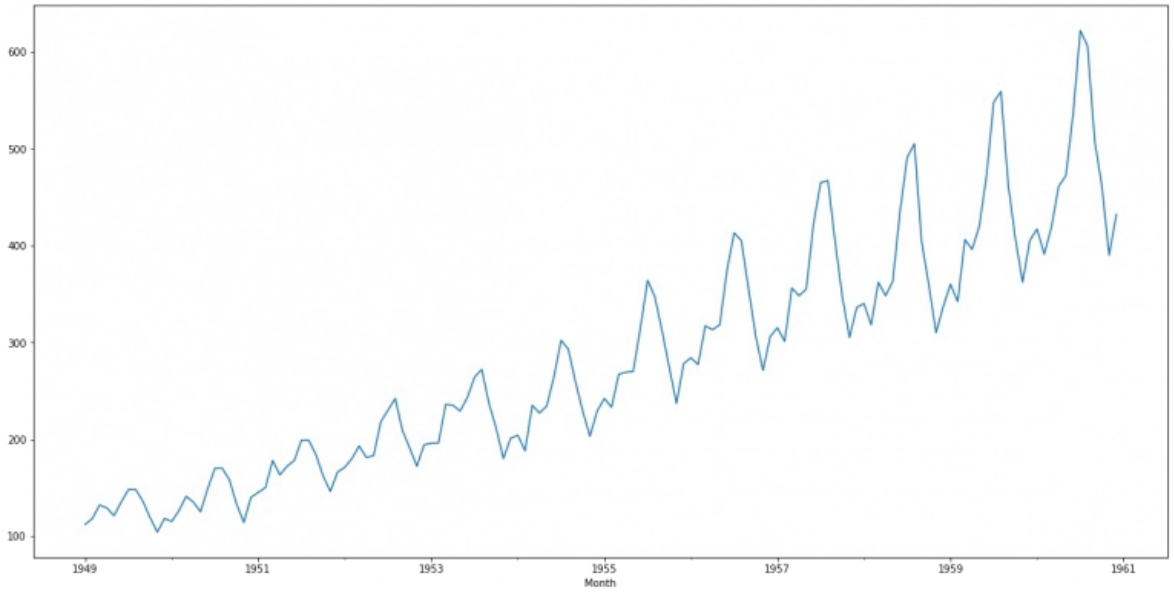
3. Methods to Check Stationarity

The next step is to determine whether a given series is stationary or not and deal with it accordingly. This section looks at some common methods which we can use to perform this check.

Visual test

Consider the plots we used in the previous section. We were able to identify the series in which mean and variance were changing with time, simply by looking at each plot. Similarly, we can plot the data and determine if the properties of the series are changing with time or not.

```
train['#Passengers'].plot()
```



Although its very clear that we have a trend (varying mean) in the above series, this visual approach might not always give accurate results. It is better to confirm the observations using some statistical tests.

Statistical test

Instead of going for the visual test, we can use statistical tests like the unit root stationary tests. Unit root indicates that the statistical properties of a given series are not constant with time, which is the condition for stationary time series. Here is the mathematics explanation of the same :

Suppose we have a time series :

$y_t = a*y_{t-1} + \varepsilon_t$

where y_t is the value at the time instant t and ε_t is the error term. In order to calculate y_t we need the value of y_{t-1}, which is :

$y_{t-1} = a*y_{t-2} + \varepsilon_{t-1}$

If we do that for all observations, the value of y_t will come out to be:

$y_t = a^n*y_{t-n} + \Sigma\varepsilon_{t-i}*a^i$

If the value of a is 1 (unit) in the above equation, then the predictions will be equal to the y_{t-n} and sum of all errors from t-n to t, which means that the variance will increase with time. This is knows as unit root in a time series. We know that for a stationary time series, the

variance must not be a function of time. The unit root tests check the presence of unit root in the series by checking if value of a=1. Below are the two of the most commonly used unit root stationary tests:

ADF (Augmented Dickey Fuller) 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 (we will understand more about difference stationary in the next section).

Python code:

```
#define function for ADF test
from statsmodels.tsa.stattools import adfuller
def adf_test(timeseries):
    #Perform Dickey-Fuller test:
    print ('Results of Dickey-Fuller Test:')
    dftest = adfuller(timeseries, autolag='AIC')
    dfoutput = pd.Series(dftest[0:4], index=['Test Statistic','p-value','#Lags Used','Number of Observations Used'])
    for key,value in dftest[4].items():
       dfoutput['Critical Value (%s)'%key] = value
    print (dfoutput)

#apply adf test on the series
adf_test(train['#Passengers'])
```

Results of ADF test: The ADF tests gives the following results – test statistic, p value and the critical value at 1%, 5% , and 10% confidence intervals. The results of our test for this particular series are:

```
Results of Dickey-Fuller Test:
Test Statistic                  0.815369
p-value                         0.991880
#Lags Used                     13.000000
Number of Observations Used    130.000000
Critical Value (1%)             -3.481682
Critical Value (5%)             -2.884042
Critical Value (10%)            -2.578770
dtype: float64
```

Test for stationarity: If the test statistic is less than the critical value, we can reject the null hypothesis (aka 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 not stationary).

In our above example, the test statistic > critical value, which implies that the series is not stationary. This confirms our original observation which we initially saw in the visual test.

2 . KPSS (Kwiatkowski-Phillips-Schmidt-Shin) Test

KPSS is another test for checking the stationarity of a time series (slightly less popular than the Dickey Fuller test). The null and alternate hypothesis for the KPSS test are opposite that of the ADF test, which often creates confusion.

The authors of the KPSS test have defined the null hypothesis as the process is trend stationary, to an alternate hypothesis of a unit root series. We will understand the *trend stationarity* in detail in the next section. For now, let's focus on the implementation and see the results of the KPSS test.

Null Hypothesis: The process is trend stationary.

Alternate Hypothesis: The series has a unit root (series is not stationary).

Python code:

```
#define function for kpss test
from statsmodels.tsa.stattools import kpss
#define KPSS
def kpss_test(timeseries):
    print ('Results of KPSS Test:')
    kpsstest = kpss(timeseries, regression='c')
    kpss_output = pd.Series(kpsstest[0:3], index=['Test Statistic','p-value','Lags Used'])
    for key,value in kpsstest[3].items():
    kpss_output['Critical Value (%s)'%key] = value
print (kpss_output)
```

Results of KPSS test: Following are the results of the KPSS test – Test statistic, p-value, and the critical value at 1%, 2.5%, 5%, and 10% confidence intervals. For the air passengers dataset, here are the results:

```
Results of KPSS Test:
Test Statistic           1.052175
p-value                  0.010000
Lags Used               14.000000
Critical Value (10%)     0.347000
Critical Value (5%)      0.463000
Critical Value (2.5%)    0.574000
Critical Value (1%)      0.739000
dtype: float64
```

Test for stationarity: If the test statistic is greater than the critical value, we reject the null hypothesis (series is not stationary). If the test statistic is less than the critical value, if fail to reject the null hypothesis (series is stationary). For the air passenger data, the value of the test statistic is greater than the critical value at all confidence intervals, and hence we can say that the series is not stationary.

I usually perform both the statistical tests before I prepare a model for my time series data. It once happened that both the tests showed contradictory results. One of the tests showed that the series is stationary while the other showed that the series is not! I got stuck at this part for hours, trying to figure out how is this possible. As it turns out, there are more than one type of stationarity.

So in summary, the ADF test has an alternate hypothesis of linear or difference stationary, while the KPSS test identifies trend-stationarity in a series.

3. Types of Stationarity

Let us understand the different types of stationarities and how to interpret the results of the above tests.

- **Strict Stationary**: A strict stationary series satisfies the mathematical definition of a stationary process. For a strict stationary series, the mean, variance and covariance are not the function of time. The aim is to convert a non-stationary series into a strict stationary series for making predictions.
- **Trend Stationary**: A series that has no unit root but exhibits a trend is referred to as a trend stationary series. Once the trend is removed, the resulting series will be strict stationary. The KPSS test classifies a series as stationary on the absence of unit root. This means that the series can be strict stationary or trend stationary.
- **Difference Stationary**: A time series that can be made strict stationary by differencing falls under difference stationary. ADF test is also known as a difference stationarity test.

It’s always better to apply both the tests, so that we are sure that the series is truly stationary. Let us look at the possible outcomes of applying these stationary tests.

- **Case 1**: Both tests conclude that the series is not stationary -> series is not stationary
- **Case 2**: Both tests conclude that the series is stationary -> series is stationary
- **Case 3**: KPSS = stationary and ADF = not stationary -> trend stationary, remove the trend to make series strict stationary
- **Case 4**: KPSS = not stationary and ADF = stationary -> difference stationary, use differencing to make series stationary

4. Making a Time Series Stationary

Now that we are familiar with the concept of stationarity and its different types, we can finally move on to actually making our series stationary. Always keep in mind that in order to use time series forecasting models, it is necessary to convert any non-stationary series to a stationary series first.

Differencing

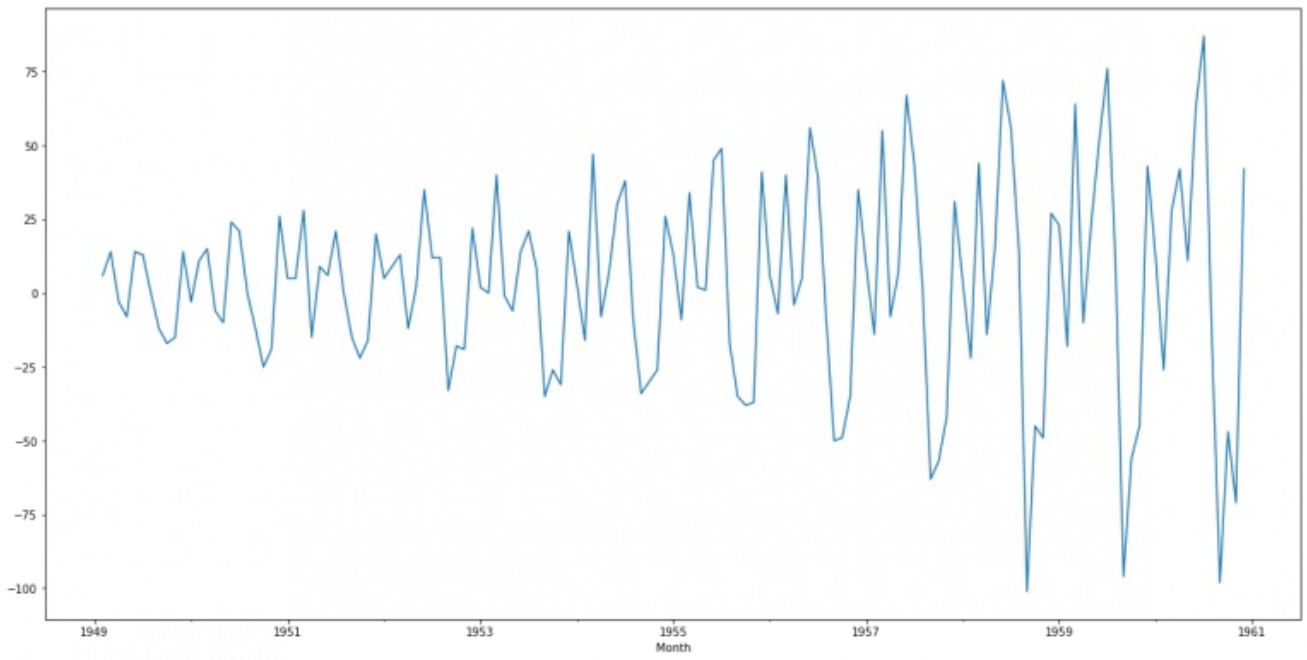
In this method, we compute the difference of consecutive terms in the series. Differencing is typically performed to get rid of the varying mean. Mathematically, differencing can be written as:

$y_t' = y_t - y_{(t-1)}$

where y_t is the value at a time t

Applying differencing on our series and plotting the results:

```
train['#Passengers_diff'] = train['#Passengers'] - train['#Passengers'].shift(1)
train['#Passengers_diff'].dropna().plot()
```



Seasonal Differencing

In seasonal differencing, instead of calculating the difference between consecutive values, we calculate the difference between an observation and a previous observation from the same season. For example, an observation taken on a Monday will be subtracted from an observation taken on the previous Monday. Mathematically it can be written as:

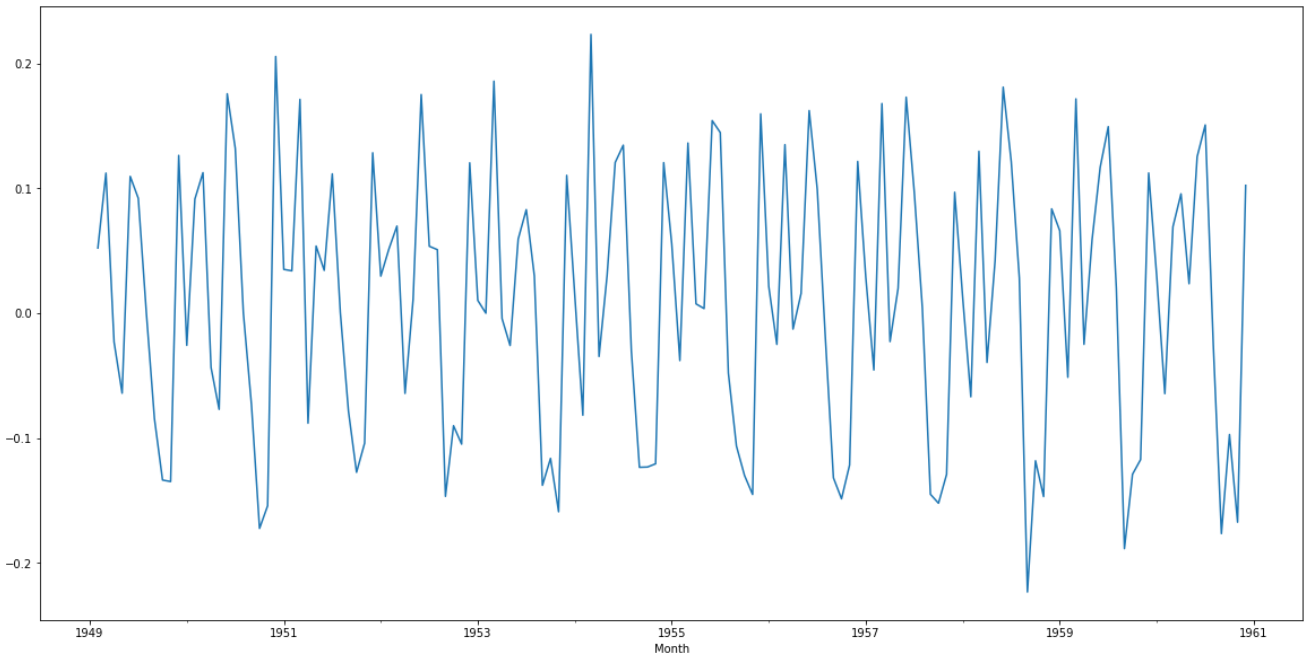
$y_t' = y_t - y_{(t-n)}$

```
n=7
train['#Passengers_diff'] = train['#Passengers'] - train['#Passengers'].shift(n)
```

Transformation

Transformations are used to stabilize the non-constant variance of a series. Common transformation methods include power transform, square root, and log transform. Let's do a quick log transform and differencing on our air passenger dataset:

```
train['#Passengers_log'] = np.log(train['#Passengers'])
train['#Passengers_log_diff'] = train['#Passengers_log'] - train['#Passengers_log'].shift(1)
train['#Passengers_log_diff'].dropna().plot()
```



As you can see, this plot is a significant improvement over the previous plots. You can use square root or power transformation on the series and see if they come up with better results. Feel free to share your findings in the comments section below!

End Notes

In this article we covered different methods that can be used to check the stationarity of a time series. But the buck doesn't stop here. The next step is to apply a forecasting model on the series we obtained. You can refer to the following article to build such a model: Beginner's Guide to Time Series Forecast.

Build High Performance Time Series Models using Auto ARIMA in Python and R

analyticsvidhya.com/blog/2018/08/auto-arima-time-series-modeling-python-r

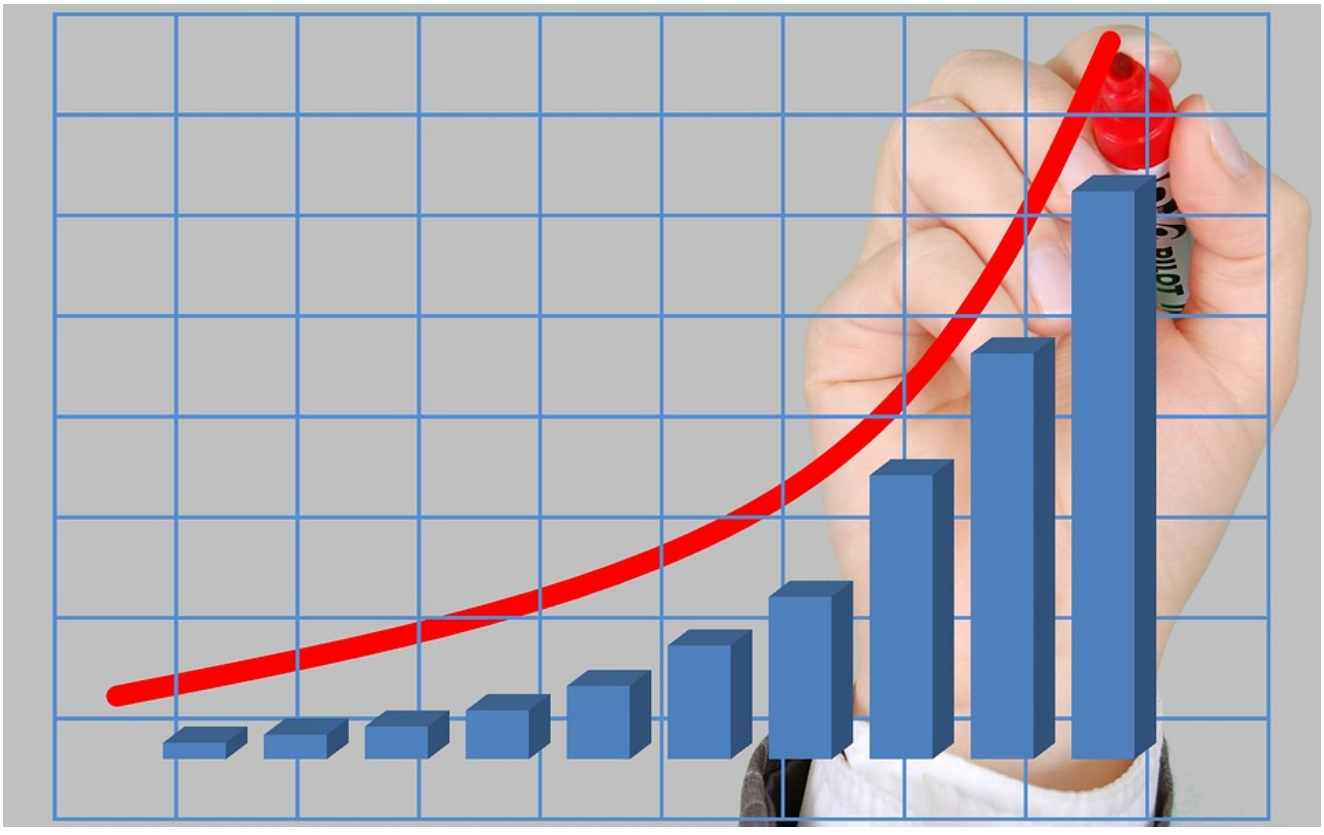
Aishwarya Singh An avid reader and blogger who loves exploring the endless world of data science and artificial intelligence. Fascinated by the limitless applications of ML and AI; eager to learn and discover the depths of data science.

August 30, 2018

Introduction

Picture this – You've been tasked with forecasting the price of the next iPhone and have been provided with historical data. This includes features like quarterly sales, month-on-month expenditure, and a whole host of things that come with Apple's balance sheet. As a data scientist, which kind of problem would you classify this as? Time series modeling, of course.

From predicting the sales of a product to estimating the electricity usage of households, time series forecasting is one of the core skills any data scientist is expected to know, if not master. There are a plethora of different techniques out there which you can use, and we will be covering one of the most effective ones, called Auto ARIMA, in this article.



We will first understand the concept of ARIMA which will lead us to our main topic – Auto ARIMA. To solidify our concepts, we will take up a dataset and implement it in both Python and R.

Table of content

If you are familiar with time series and it's techniques (like moving average, exponential smoothing, and ARIMA), you can skip directly to section 4. For beginners, start from the below section which is a brief introduction to time series and various forecasting techniques.

1. What is a time series ?

Before we learn about the techniques to work on time series data, we must first understand what a time series actually is and how is it different from any other kind of data. Here is the formal definition of time series – It is a series of data points **measured at consistent time intervals**. This simply means that particular values are recorded at a constant interval which may be hourly, daily, weekly, every 10 days, and so on. **What makes time series different is that each data point in the series is dependent on the previous data points.** Let us understand the difference more clearly by taking a couple of examples.

Example 1:

Suppose you have a dataset of people who have taken a loan from a particular company (as shown in the table below). Do you think each row will be related to the previous rows? Certainly not! The loan taken by a person will be based on his financial conditions and needs (there could be other factors such as the family size etc., but for simplicity we are considering only income and loan type) . Also, the data was not collected at any specific time interval. It depends on when the company received a request for the loan.

Loan ID	Date	Income per month	Loan type	Loan amount
ID207	15/07/18	25000	Car Loan	1000000
ID190	15/07/18	50000	Home Loan	2500000
ID007	22/07/18	70000	Personal Loan	1500000
ID433	29/07/18	45000	Education Loan	4500000
ID204	29/07/18	20000	Education Loan	5000000
ID611	08/08/18	80000	Business Loan	9000000
ID947	17/08/18	60000	Personal Loan	3700000
ID200	21/08/18	20000	Car Loan	500000
ID222	29/08/18	30000	Personal Loan	4300000

Example 2:

Let's take another example. Suppose you have a dataset that contains the level of CO2 in the air per day (screenshot below). Will you be able to predict the approximate amount of CO2 for the next day by looking at the values from the past few days? Well, of course. If you observe, the data has been recorded on a daily basis, that is, the time interval is constant (24 hours).

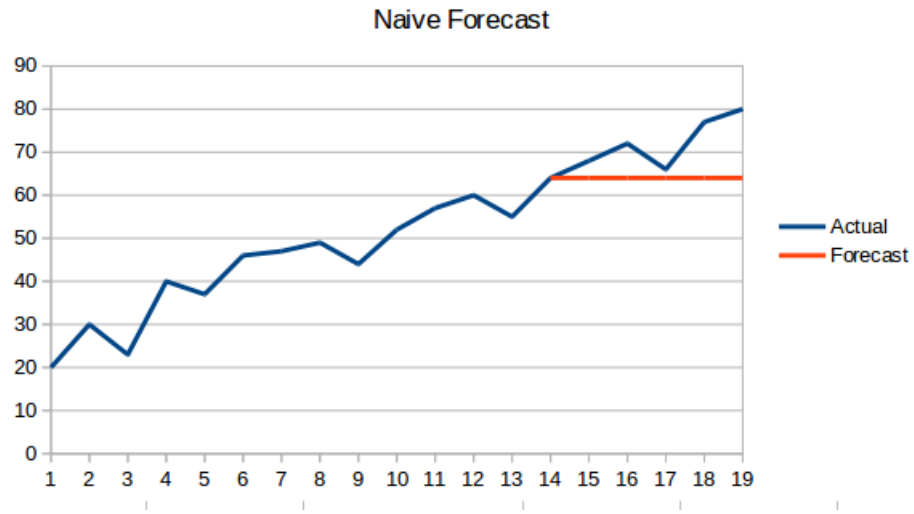
You must have got an intuition about this by now – the first case is a simple regression problem and the second is a time series problem. Although the time series puzzle here can also be solved using linear regression, but that isn't really the best approach as it neglects the relation of the values with all the relative past values. Let's now look at some of the common techniques used for solving time series problems.

Date	CO2 in air
20/08/18	313
21/08/18	322
22/08/18	330
23/08/18	337
24/08/18	343
25/08/18	350
26/08/18	359
27/08/18	366
28/08/18	371

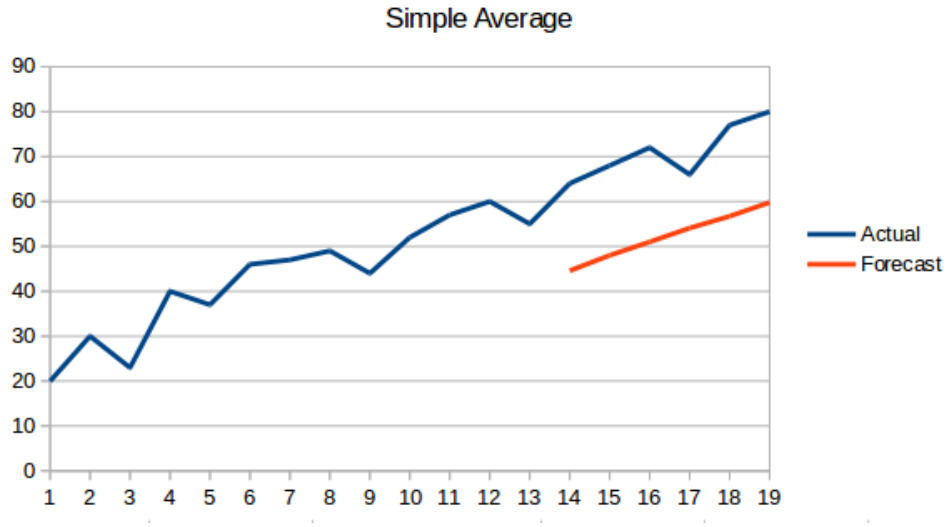
2. Methods for time series forecasting

There are a number of methods for time series forecasting and we will briefly cover them in this section. The detailed explanation and python codes for all the below mentioned techniques can be found in this article: 7 techniques for time series forecasting (with python codes).

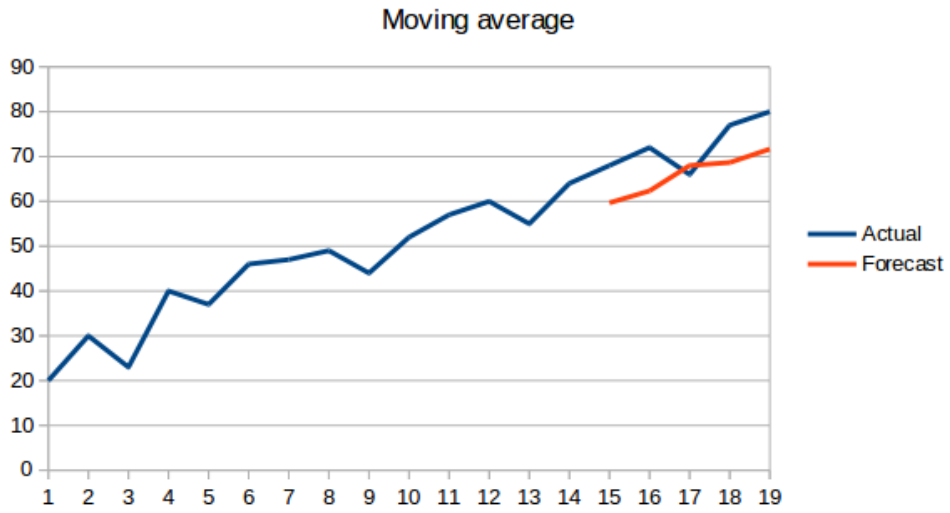
1. **Naive Approach:** In this forecasting technique, the value of the new data point is predicted to be equal to the previous data point. The result would be a flat line, since all new values take the previous values.



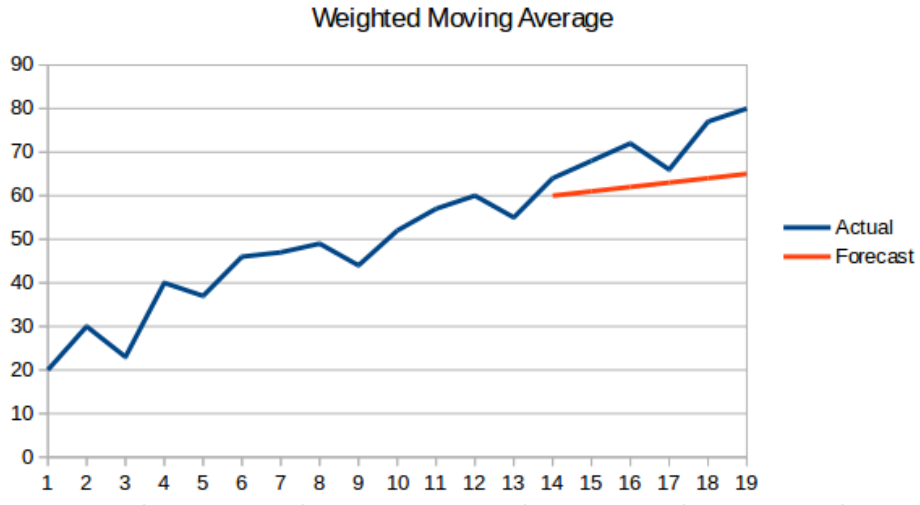
2. **Simple Average:** The next value is taken as the average of all the previous values. The predictions here are better than the 'Naive Approach' as it doesn't result in a flat line but here, all the past values are taken into consideration which might not always be useful. For instance, when asked to predict today's temperature, you would consider the last 7 days' temperature rather than the temperature a month ago.



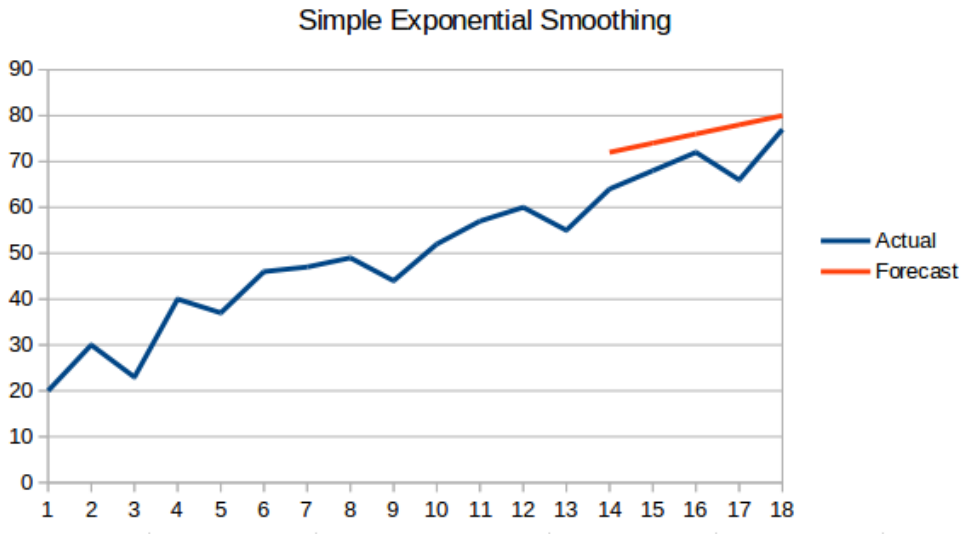
3. **Moving Average :** This is an improvement over the previous technique. Instead of taking the average of all the previous points, the average of 'n' previous points is taken to be the predicted value.



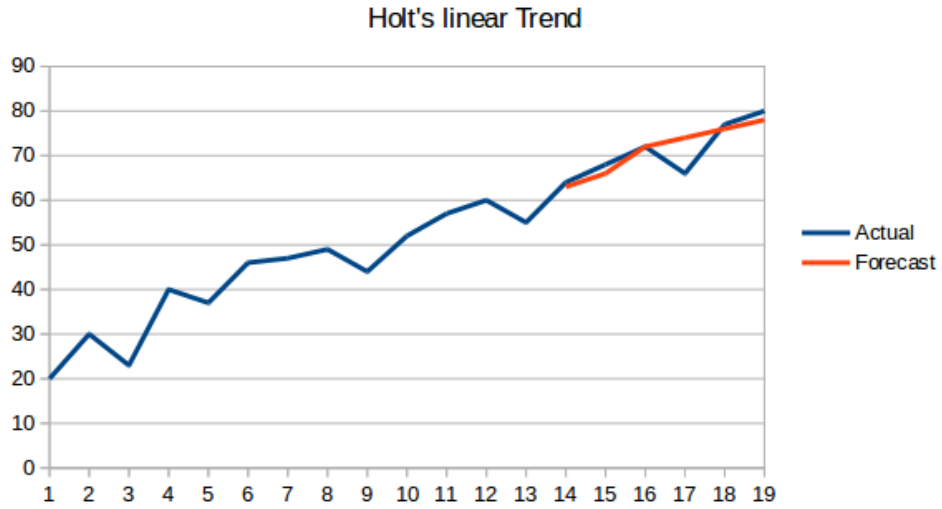
4. **Weighted moving average :** A weighted moving average is a moving average where the past 'n' values are given different weights.



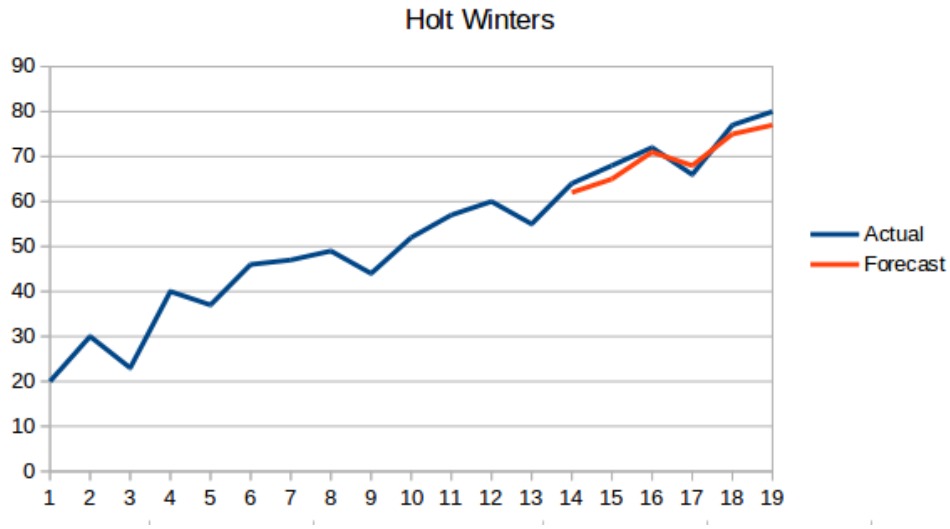
5. **Simple Exponential Smoothing:** In this technique, larger weights are assigned to more recent observations than to observations from the distant past.



6. **Holt's Linear Trend Model:** This method takes into account the trend of the dataset. By trend, we mean the increasing or decreasing nature of the series. Suppose the number of bookings in a hotel increases every year, then we can say that the number of bookings show an increasing trend. The forecast function in this method is a function of level and trend.



7. **Holt Winters Method:** This algorithm takes into account both the trend and the seasonality of the series. For instance – the number of bookings in a hotel is high on weekends & low on weekdays, and increases every year; there exists a weekly seasonality and an increasing trend.



8. **ARIMA:** ARIMA is a very popular technique for time series modeling. It describes the correlation between data points and takes into account the difference of the values. An improvement over ARIMA is SARIMA (or seasonal ARIMA). We will look at ARIMA in a bit more detail in the following section.

3. Introduction to ARIMA

In this section we will do a quick introduction to ARIMA which will be helpful in understanding Auto Arima. A detailed explanation of Arima, parameters (p,q,d), plots (ACF PACF) and implementation is included in this article : Complete tutorial to Time Series.

ARIMA is a very popular statistical method for time series forecasting. ARIMA stands for **Auto-Regressive Integrated Moving Averages**. ARIMA models work on the following assumptions –

- The data series is stationary, which means that the mean and variance should not vary with time. A series can be made stationary by using log transformation or differencing the series.
- The data provided as input must be a univariate series, since arima uses the past values to predict the future values.

ARIMA has three components – AR (autoregressive term), I (differencing term) and MA (moving average term). Let us understand each of these components –

- AR term refers to the past values used for forecasting the next value. The AR term is defined by the parameter 'p' in arima. The value of 'p' is determined using the PACF plot.
- MA term is used to defines number of past forecast errors used to predict the future values. The parameter 'q' in arima represents the MA term. ACF plot is used to identify the correct 'q' value.
- Order of differencing specifies the number of times the differencing operation is performed on series to make it stationary. Test like ADF and KPSS can be used to determine whether the series is stationary and help in identifying the d value.

4. Steps for ARIMA implementation

The general steps to implement an ARIMA model are –

1. **Load the data:** The first step for model building is of course to load the dataset
2. **Preprocessing:** Depending on the dataset, the steps of preprocessing will be defined. This will include creating timestamps, converting the dtype of date/time column, making the series univariate, etc.
3. **Make series stationary:** In order to satisfy the assumption, it is necessary to make the series stationary. This would include checking the stationarity of the series and performing required transformations
4. **Determine d value:** For making the series stationary, the number of times the difference operation was performed will be taken as the d value
5. **Create ACF and PACF plots:** This is the most important step in ARIMA implementation. ACF PACF plots are used to determine the input parameters for our ARIMA model

6. **Determine the p and q values:** Read the values of p and q from the plots in the previous step
7. **Fit ARIMA model:** Using the processed data and parameter values we calculated from the previous steps, fit the ARIMA model
8. **Predict values on validation set:** Predict the future values
9. **Calculate RMSE:** To check the performance of the model, check the RMSE value using the predictions and actual values on the validation set

5. Why do we need Auto ARIMA?

Although ARIMA is a very powerful model for forecasting time series data, the data preparation and parameter tuning processes end up being really time consuming. Before implementing ARIMA, you need to make the series stationary, and determine the values of p and q using the plots we discussed above. Auto ARIMA makes this task really simple for us as it eliminates steps 3 to 6 we saw in the previous section. Below are the steps you should follow for implementing auto ARIMA:

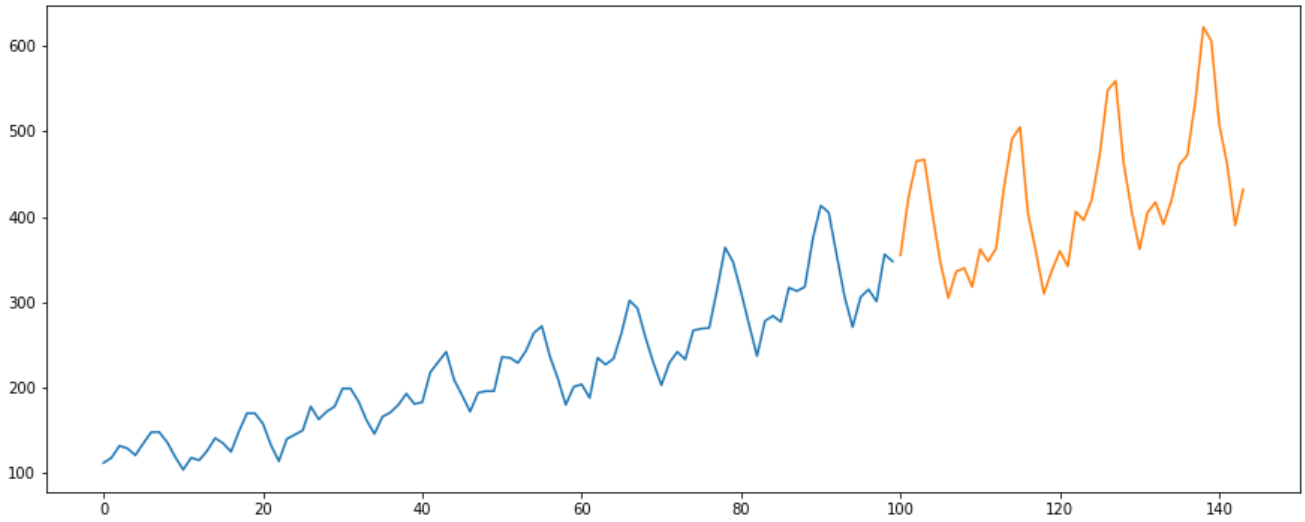
1. Load the data: This step will be the same. Load the data into your notebook
2. Preprocessing data: The input should be univariate, hence drop the other columns
3. Fit Auto ARIMA: Fit the model on the univariate series
4. Predict values on validation set: Make predictions on the validation set
5. Calculate RMSE: Check the performance of the model using the predicted values against the actual values

We completely bypassed the selection of p and q feature as you can see. What a relief! In the next section, we will implement auto ARIMA using a toy dataset.

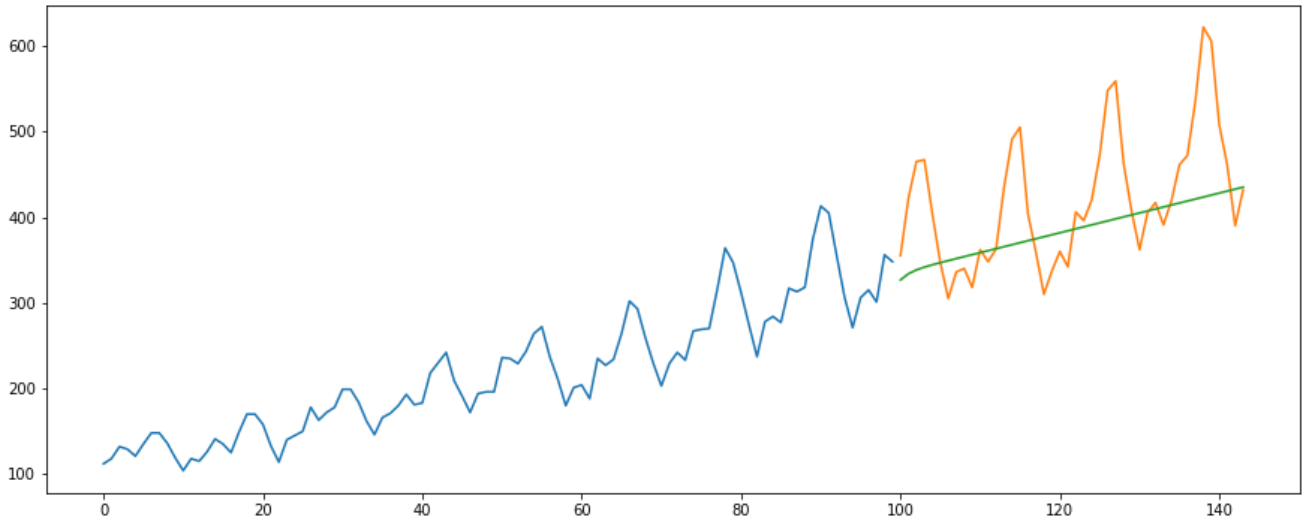
6. Implementation in Python and R

We will be using the International-Air-Passenger dataset. This dataset contains monthly total of number of passengers (in thousands). It has two columns – month and count of passengers. You can download the dataset from this link.

```
#load the data
data = pd.read_csv('international-airline-passengers.csv')

#divide into train and validation set
train = data[:int(0.7*(len(data)))]
valid = data[int(0.7*(len(data))):]

#preprocessing (since arima takes univariate series as input)
train.drop('Month',axis=1,inplace=True)
valid.drop('Month',axis=1,inplace=True)

#plotting the data
train['International airline passengers'].plot()
valid['International airline passengers'].plot()
```




```
#building the model
from pyramid.arima import auto_arima
model = auto_arima(train, trace=True, error_action='ignore', suppress_warnings=True)
model.fit(train)

forecast = model.predict(n_periods=len(valid))
forecast = pd.DataFrame(forecast,index = valid.index,columns=['Prediction'])

#plot the predictions for validation set
plt.plot(train, label='Train')
plt.plot(valid, label='Valid')
plt.plot(forecast, label='Prediction')
plt.show()
```




```
#calculate rmse
from math import sqrt
from sklearn.metrics import mean_squared_error

rms = sqrt(mean_squared_error(valid,forecast))
print(rms)
```

output -
76.51355764316357

Below is the R Code for the same problem:

```
# loading packages
library(forecast)
library(Metrics)

# reading data
data = read.csv("international-airline-passengers.csv")

# splitting data into train and valid sets
train = data[1:100,]
valid = data[101:nrow(data),]

# removing "Month" column
train$Month = NULL

# training model
model = auto.arima(train)

# model summary
summary(model)

# forecasting
forecast = predict(model,44)

# evaluation
rmse(valid$International.airline.passengers, forecast$pred)
```

7. How does Auto Arima select the best parameters

In the above code, we simply used the *.fit()* command to fit the model without having to select the combination of p, q, d. But how did the model figure out the best combination of these parameters? Auto ARIMA takes into account the AIC and BIC values generated (as you can see in the code) to determine the best combination of parameters. AIC (Akaike Information Criterion) and BIC (Bayesian Information Criterion) values are estimators to compare models. The lower these values, the better is the model.

Check out these links if you are interested in the maths behind AIC and BIC.

8. End Notes and Further Reads

I have found auto ARIMA to be the simplest technique for performing time series forecasting. Knowing a shortcut is good but being familiar with the math behind it is also important. In this article I have skimmed through the details of how ARIMA works but do make sure that you go through the links provided in the article. For your easy reference, here are the links again:

I would suggest practicing what we have learned here on this practice problem: Time Series Practice Problem. You can also take our training course created on the same practice

problem, Time series forecasting, to provide you a head start.

Generate Quick and Accurate Time Series Forecasts using Facebook's Prophet (with Python & R codes)

analyticsvidhya.com/blog/2018/05/generate-accurate-forecasts-facebook-prophet-python-r

Ankit Choudhary IIT Bombay Graduate with a Masters and Bachelors in Electrical Engineering. I have previously worked as a lead decision scientist for Indian National Congress deploying statistical models (Segmentation, K-Nearest Neighbours) to help party leadership/Team make data-driven decisions. My interest lies in putting data in heart of business for data-driven decision making.

May 10, 2018

Introduction

Understanding time based patterns is critical for any business. Questions like how much inventory to maintain, how much footfall do you expect in your store to how many people will travel by an airline – all of these are important time series problems to solve.

This is why time series forecasting is one of the must-know techniques for any data scientist. From predicting the weather to the sales of a product, it is integrated into the data science ecosystem and that makes it a mandatory addition to a data scientist's skillset.

If you are a beginner, time series also provides a good way to start working on real life projects. You can relate to time series very easily and they help you enter the larger world of machine learning.

Prophet is an open source library published by Facebook that is based on **decomposable (trend+seasonality+holidays) models**. It provides us with the ability to make time series predictions with good accuracy using simple intuitive parameters and has support for including impact of custom seasonality and holidays!

In this article, we shall cover some background on how Prophet fills the existing gaps in generating fast reliable forecasts followed by a demonstration using Python. The final results will surprise you!

Table of Contents

What's new in Prophet?

When a forecasting model doesn't run as planned, we want to be able to tune the parameters of the method with regards to the specific problem at hand. Tuning these methods requires a thorough understanding of how the underlying time series models

work. The first input parameters to automated ARIMA, for instance, are the maximum orders of the differencing, the auto-regressive components, and the moving average components. A typical analyst will not know how to adjust these orders to avoid the behaviour and this is the type of expertise that is hard to acquire and scale.

The Prophet package provides intuitive parameters which are easy to tune. Even someone who lacks expertise in forecasting models can use this to make meaningful predictions for a variety of problems in a business scenario.

The Prophet Forecasting Model

We use a decomposable time series model with three main model components: trend, seasonality, and holidays. They are combined in the following equation:

$$y(t) = g(t) + s(t) + h(t) + \epsilon_t$$

- **g(t)**: piecewise linear or logistic growth curve for modelling non-periodic changes in time series
- **s(t)**: periodic changes (e.g. weekly/yearly seasonality)
- **h(t)**: effects of holidays (user provided) with irregular schedules
- **ϵ_t**: error term accounts for any unusual changes not accommodated by the model

Using time as a regressor, Prophet is trying to fit several linear and non linear functions of time as components. Modeling seasonality as an additive component is the same approach taken by exponential smoothing in Holt-Winters technique . We are, in effect, framing the forecasting problem as a curve-fitting exercise rather than looking explicitly at the time based dependence of each observation within a time series.

Trend

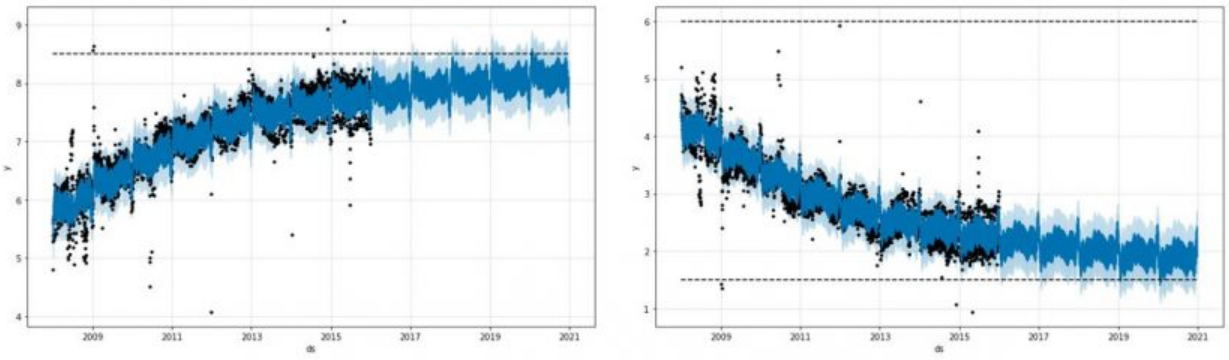
Trend is modelled by fitting a piece wise linear curve over the trend or the non-periodic part of the time series. The linear fitting exercise ensures that it is least affected by spikes/missing data.

Saturating growth

An important question to ask here is – Do we expect the target to keep growing/falling for the entire forecast interval?

More often than not, there are cases with non-linear growth with a running maximum capacity. I will illustrate this with an example below.

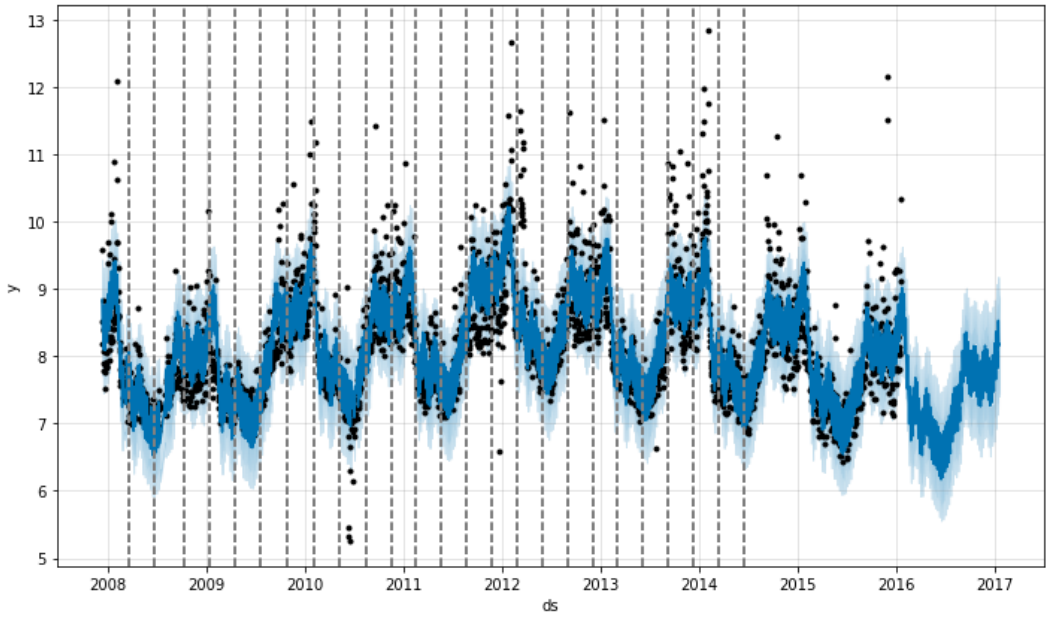
Let's say we are trying to forecast number of downloads of an app in a region for the next 12 months. The maximum downloads is always capped by the total number of smartphone users in the region. The number of smartphone users will also, however, increase with time.



With domain knowledge at his/her disposal, an analyst can then define a varying **capacity *C(t)*** for the time series forecasts he/she is trying to make.

Changepoints

Another question to answer is whether my time series encounters any underlying changes in the phenomena e.g. a new product launch, unforeseen calamity etc. At such points, the growth rate is allowed to change. These changepoints are automatically selected. However, a user can also feed the changepoints manually if it is required. In the below plot, the dotted lines represent the changepoints for the given time series.



As the number of changepoints allowed is increased the fit becomes more flexible. There are basically 2 problems an analyst might face while working with the trend component:

- Overfitting
- Underfitting

A parameter called changepoint_prior_scale could be used to adjust the trend flexibility and tackle the above 2 problems. Higher value will fit a more flexible curve to the time series.

Seasonality

To fit and forecast the effects of seasonality, prophet relies on fourier series to provide a flexible model. Seasonal effects s(t) are approximated by the following function:

$$s(t) = \sum_{n=1}^{N} \left(a_n \cos\left(\frac{2\pi nt}{P}\right) + b_n \sin\left(\frac{2\pi nt}{P}\right) \right)$$

P is the period (365.25 for yearly data and 7 for weekly data)

Parameters [a_1, b_1, ..., a_N, b_N] need to be estimated for a given N to model seasonality.

The fourier order N that defines whether high frequency changes are allowed to be modelled is an important parameter to set here. For a time series, if the user believes the high frequency components are just noise and should not be considered for modelling, he/she could set the values of N from to a lower value. If not, N can be tuned to a higher value and set using the forecast accuracy.

Holidays and events

Holidays and events incur predictable shocks to a time series. For instance, Diwali in India occurs on a different day each year and a large portion of the population buy a lot of new items during this period.

Prophet allows the analyst to provide a custom list of past and future events. A window around such days are considered separately and additional parameters are fitted to model the effect of holidays and events.

Prophet in action (using Python)

Currently implementations of Prophet are available in both Python and R. They have exactly the same features.

Prophet() function is used do define a Prophet forecasting model in Python. Let us look at the most important parameters:

3.1 Trend parameters

Parameter	Description

growth	linear' or 'logistic' to specify a linear or logistic trend
changepoints	List of dates at which to include potential changepoints (automatic if not specified)
n_changepoints	If changepoints in not supplied, you may provide the number of changepoints to be automatically included
changepoint_prior_scale	Parameter for changing flexibility of automatic changepoint selection

3.2 Seasonality & Holiday Parameters

Parameter	Description
yearly_seasonality	Fit yearly seasonality
weekly_seasonality	Fit weekly seasonality
daily_seasonality	Fit daily seasonality
holidays	Feed dataframe containing holiday name and date
seasonality_prior_scale	Parameter for changing strength of seasonality model
holiday_prior_scale	Parameter for changing strength of holiday model

yearly_seasonality, weekly_seasonality & daily_seasonality can take values as True, False and no of fourier terms which was discussed in the last section. If the value is True, default number of fourier terms (10) are taken. Prior scales are defined to tell the model how strongly it needs to consider the seasonal/holiday components while fitting and forecasting.

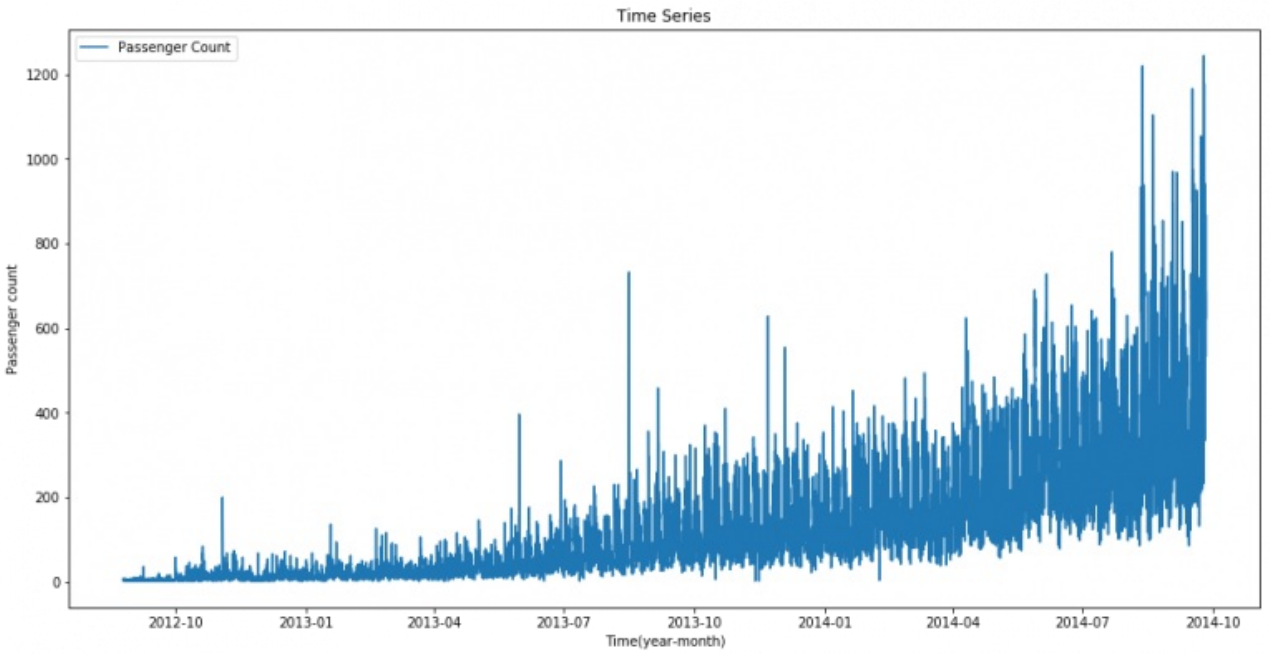
Predicting passsenger traffic using Prophet

Now that we are well versed with nuts and bolts of this amazing tool. Lets dive into a real dataset to see its potential. Here I have used Prophet in python for one of the practice problems available on datahack platform at this link.

The dataset is a univariate time series that contains hourly passenger traffic for a new public transport service. We are trying to forecast the traffic for next 7 months given historical traffic data of last 25 months. Basic EDA for this can be accessed from this course.

Import necessary packages and reading dataset

```
# Importing datasets
import pandas as pd
import numpy as np
from fbprophet import Prophet
# Read train and test
train = pd.read_csv('Train_SU63ISt.csv')
test = pd.read_csv('Test_0qrQsBZ.csv')

# Convert to datetime format
train['Datetime'] = pd.to_datetime(train.Datetime,format='%d-%m-%Y %H:%M')
test['Datetime'] = pd.to_datetime(test.Datetime,format='%d-%m-%Y %H:%M')
train['hour'] = train.Datetime.dt.hour
```



We see that this time series has a lot of noise. We could re-sample it day wise and sum to get a new series with reduced and noise and thereby easier to model.

```
# Calculate average hourly fraction
hourly_frac = train.groupby(['hour']).mean()/np.sum(train.groupby(['hour']).mean())
hourly_frac.drop(['ID'], axis = 1, inplace = True)
hourly_frac.columns = ['fraction']


# convert to time series from dataframe
train.index = train.Datetime
train.drop(['ID','hour','Datetime'], axis = 1, inplace = True)

daily_train = train.resample('D').sum()
```

Prophet requires the variable names in the time series to be:

- y – Target
- ds – Datetime

So, the next step is to convert the dataframe according to the above specifications

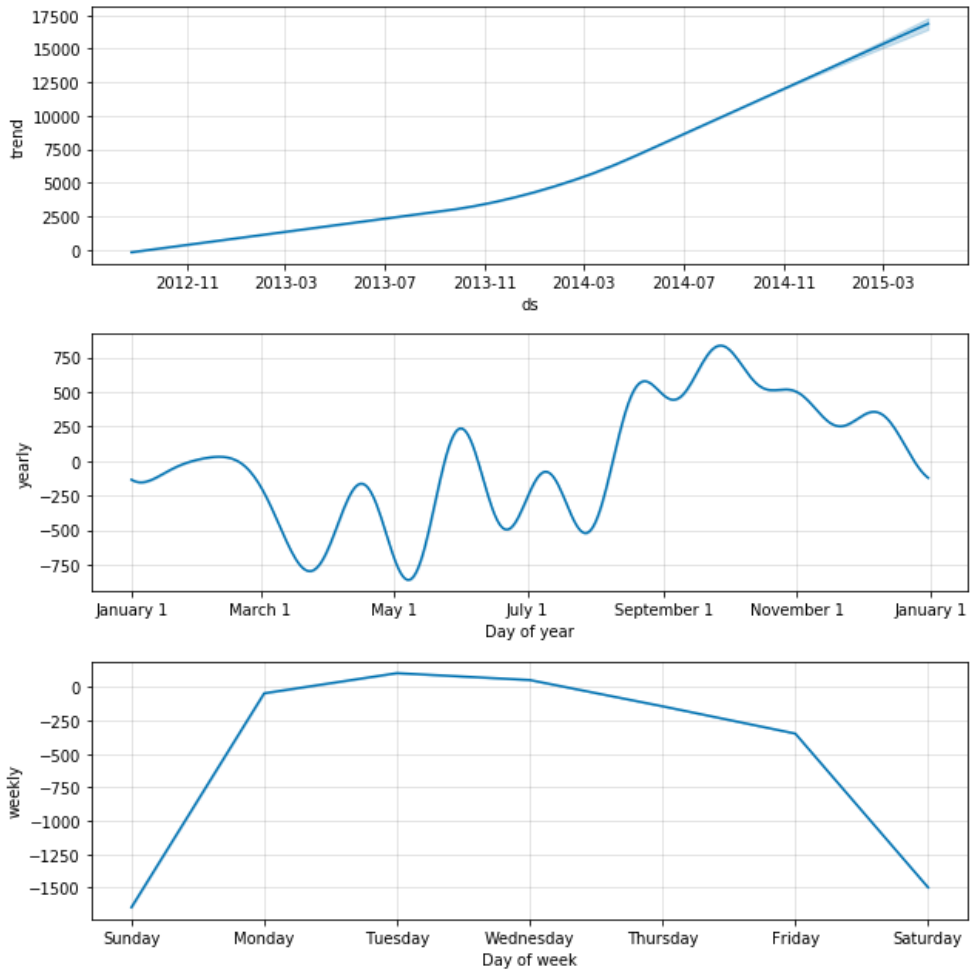
```
daily_train['ds'] = daily_train.index
daily_train['y'] = daily_train.Count
daily_train.drop(['Count'],axis = 1, inplace = True)
```

Fitting the prophet model:

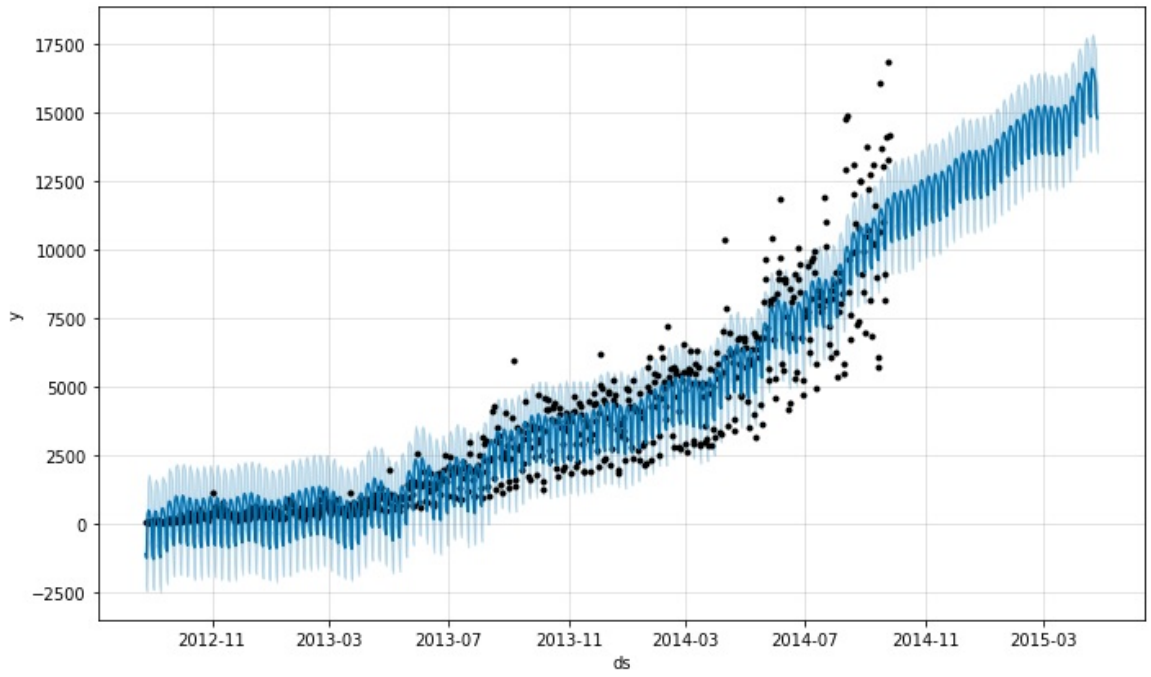
```
m = Prophet(yearly_seasonality = True, seasonality_prior_scale=0.1)
m.fit(daily_train)
future = m.make_future_dataframe(periods=213)
forecast = m.predict(future)
```

We can look at the various components using the following command:

```
m.plot_components(forecast)
```



Using the mean hourly fraction for each hour from 0 to 23, we could then convert the daily forecasts into hourly forecasts make submission. This is how our forecasts over the daily data looks like.




```
# Extract hour, day, month and year from both dataframes to merge
for df in [test, forecast]:
    df['hour'] = df.Datetime.dt.hour
    df['day'] = df.Datetime.dt.day
    df['month'] = df.Datetime.dt.month
    df['year'] = df.Datetime.dt.year

# Merge forecasts with given IDs
test = pd.merge(test,forecast, on=['day','month','year'], how='left')
cols = ['ID','hour','yhat']
test_new = test[cols]

# Merging hourly average fraction to the test data
test_new = pd.merge(test_new, hourly_frac, left_on = ['hour'], right_index=True, how = 'left')
# Convert daily aggregate to hourly traffic
test_new['Count'] = test_new['yhat'] * test_new['fraction']
test_new.drop(['yhat','fraction','hour'],axis = 1, inplace = True)
test_new.to_csv('prophet_sub.csv',index = False)
```

This gets a score of 206 on the public leaderboard and does produce a stable model. Readers can go ahead and tweak the hyperparameters (fourier order for seasonality/changeover) to get a better score. Reader could also try and use a different technique to convert the daily predictions to hourly data for submission and may get a better score.

R Code

Implementation in R for the same problem statement is given below.

```
library(prophet)
library(data.table)
library(dplyr)
library(ggplot2)

# read data
train = fread("Train_SU63ISt.csv")
test = fread("Test_0qrQsBZ.csv")

# Extract date from the Datetime variable
train$Date = as.POSIXct(strptime(train$Datetime, "%d-%m-%Y"))
test$Date = as.POSIXct(strptime(test$Datetime, "%d-%m-%Y"))

# Convert 'Datetime' variable from character to date-time format
train$Datetime = as.POSIXct(strptime(train$Datetime, "%d-%m-%Y %H:%M"))
test$Datetime = as.POSIXct(strptime(test$Datetime, "%d-%m-%Y %H:%M"))

# Aggregate train data day-wise
aggr_train = train[,list(Count = sum(Count)), by = Date]

# Visualize the data
ggplot(aggr_train) + geom_line(aes(Date, Count))

# Change column names
names(aggr_train) = c("ds", "y")

# Model building
m = prophet(aggr_train)
future = make_future_dataframe(m, periods = 213)
forecast = predict(m, future)

# Visualize forecast
plot(m, forecast)

# proportion of mean hourly 'Count' based on train data
mean_hourly_count = train %>%
 group_by(hour = hour(train$Datetime)) %>%
 summarise(mean_count = mean(Count))

s = sum(mean_hourly_count$mean_count)
mean_hourly_count$count_proportion = mean_hourly_count$mean_count/s

# variable to store hourly Count
test_count = NULL

for(i in 763:nrow(forecast)){
 test_count = append(test_count, mean_hourly_count$count_proportion * forecast$yhat[i])
}

test$Count = test_count
```

‘Anomalize’ is a R Package that Makes Anomaly Detection in Time Series Extremely Simple and Scalable

analyticsvidhya.com/blog/2018/04/anomalize-r-package-makes-anomaly-detection-extremely-simple-scalable

Pranav Dar Senior Editor at Analytics Vidhya. Data visualization practitioner who loves reading and delving deeper into the data science and machine learning arts. Always looking for new ways to improve processes using ML and AI.

April 9, 2018

Overview

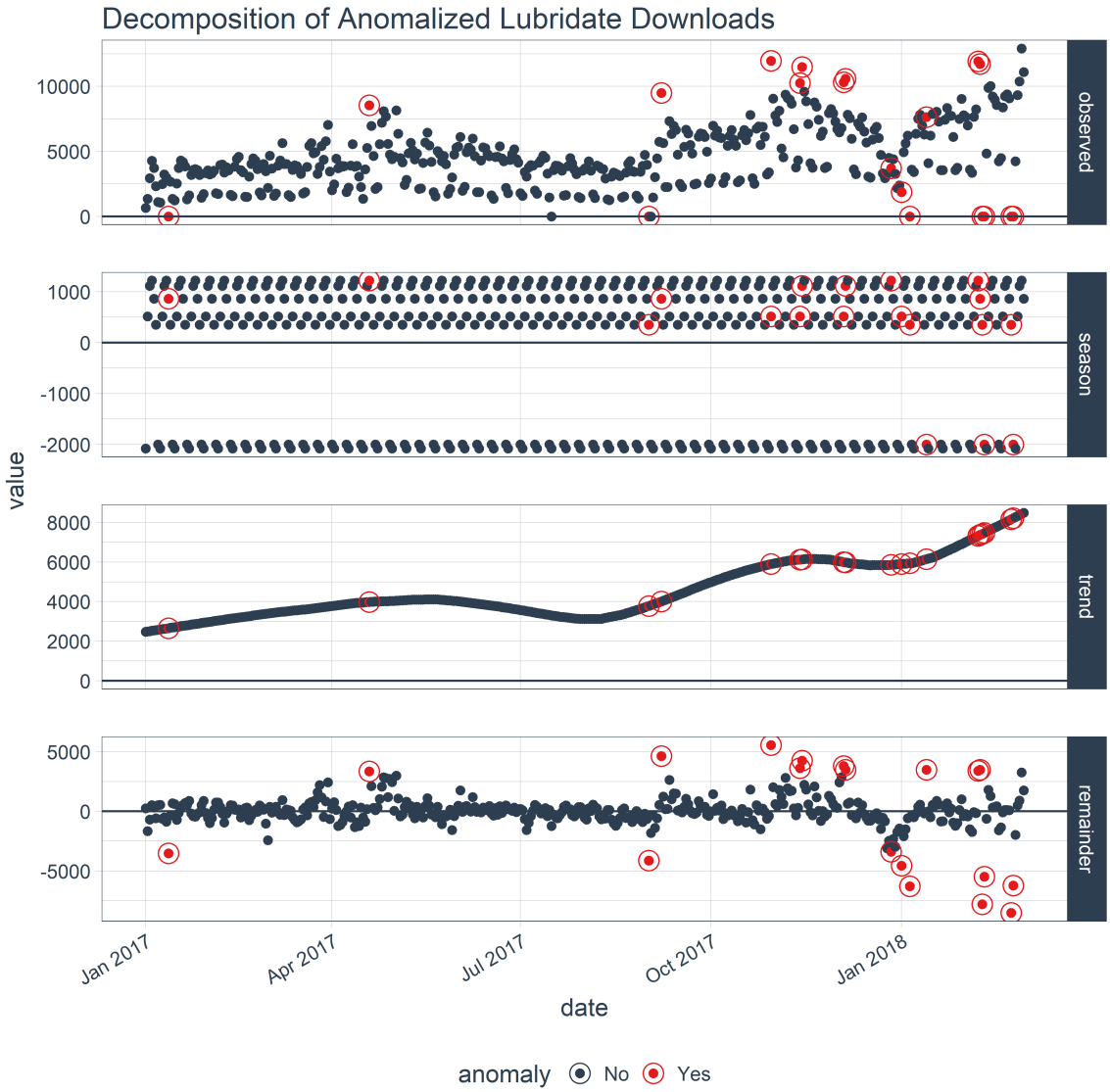
- The ‘anomalize’ package makes it really easy and scalable to detect anomalies in your time series data
- It has three functions – time_decompose(), anomalize(), and time_recompose()
- It operates within the tidyverse universe

Introduction

Anyone familiar with the machine learning world has been introduced to, or works with, time series forecasting. It’s one of the most popular fields and has diverse and vast applications.

But actually performing a time series analysis is not a straightforward task. Even with the packages currently available, there is still a bit of work that goes into making a time series model ready for the eventual analysis and for building a model. With the manual effort that goes in, the chances of missing anomalies and making errors increases.

The ‘anomalize’ package makes it really simple, easy and scalable to detect anomalies in your data. It has three functions (mentioned in this article below) and together, they make it a straightforward process to decompose, the given time series, detect any anomalies, and finally create “bands” that separate the normal data from the anomalous one.



The package has three main functions, namely:

- **time_decompose():** Separates the time series data into seasonal, trend, and remainder components
- **anomalize():** This applies anomaly detection methods to the remainder component
- **time_recompose():** This calculates limits that separate the expected normal data from the anomalies

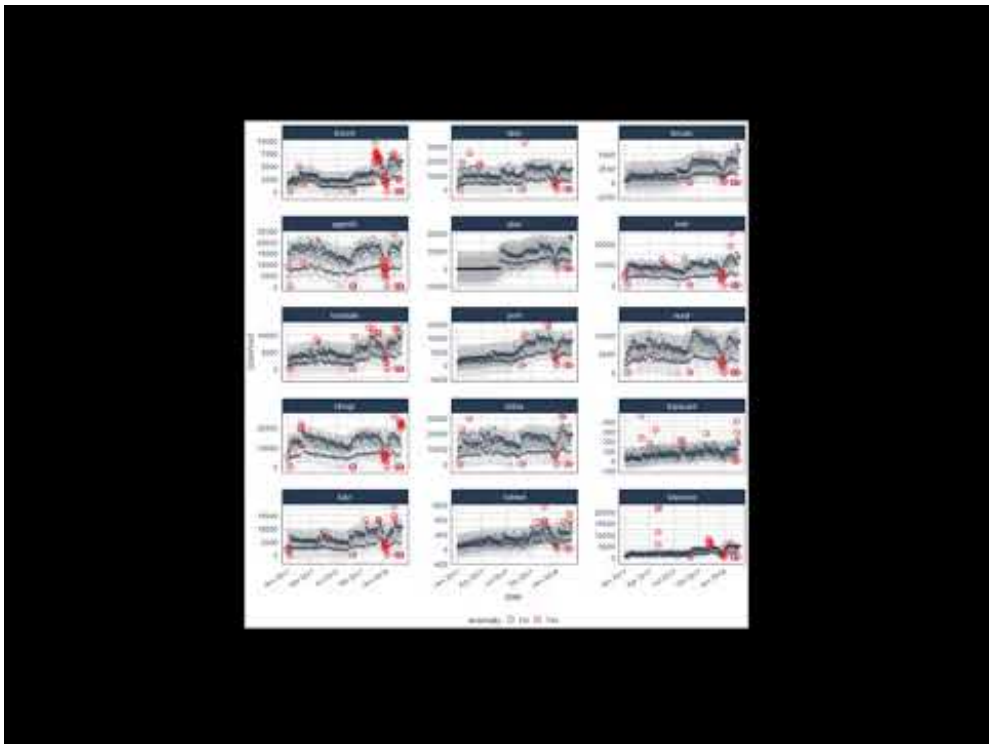
In order to use this package, you need to have the *tidyverse* package installed and loaded as well. You can install the 'anomalize' package either from *devtools*:

```
# devtools::install_github("business-science/anomalize")
```

or from the CRAN itself:

```
install.packages("anomalize")
```

You can check out the GitHub repository here and view the below video for further details on this package.



https://youtu.be/Gk_HwjhlQJs

Our take on this

There have been packages built for anomaly detection previously, namely Twitter's *AnomalyDetection* and the *tsoutliers()* packages. But 'anomalize' takes that to the next level by making it even more simpler and scalable within the tidyverse universe.

The developers have mentioned that they are also looking into the possibility of making a python library for this. The initial reaction in the ML community has been extremely positive. This is a brilliant tool for QA analysis in any field, from marketing to manufacturing. Go ahead and try it out.

7 methods to perform Time Series forecasting (with Python codes)

analyticsvidhya.com/blog/2018/02/time-series-forecasting-methods

Gurchetan Singh Budding Data Scientist from MAIT who loves implementing data analytical and statistical machine learning models in Python. I also understand big data technology like Hadoop and Alteryx.

February 8, 2018

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 modeling is one way to predict them.

Source: Bitcoin

Besides Cryptocurrencies, there are multiple important areas where time series forecasting is used – 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 should he 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()
```

```
#Printing tail
df.tail()
```

	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

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 subsetting and aggregating dataset at daily basis to explain the different methods.

- Subsetting the dataset from (August 2012 – Dec 2013)

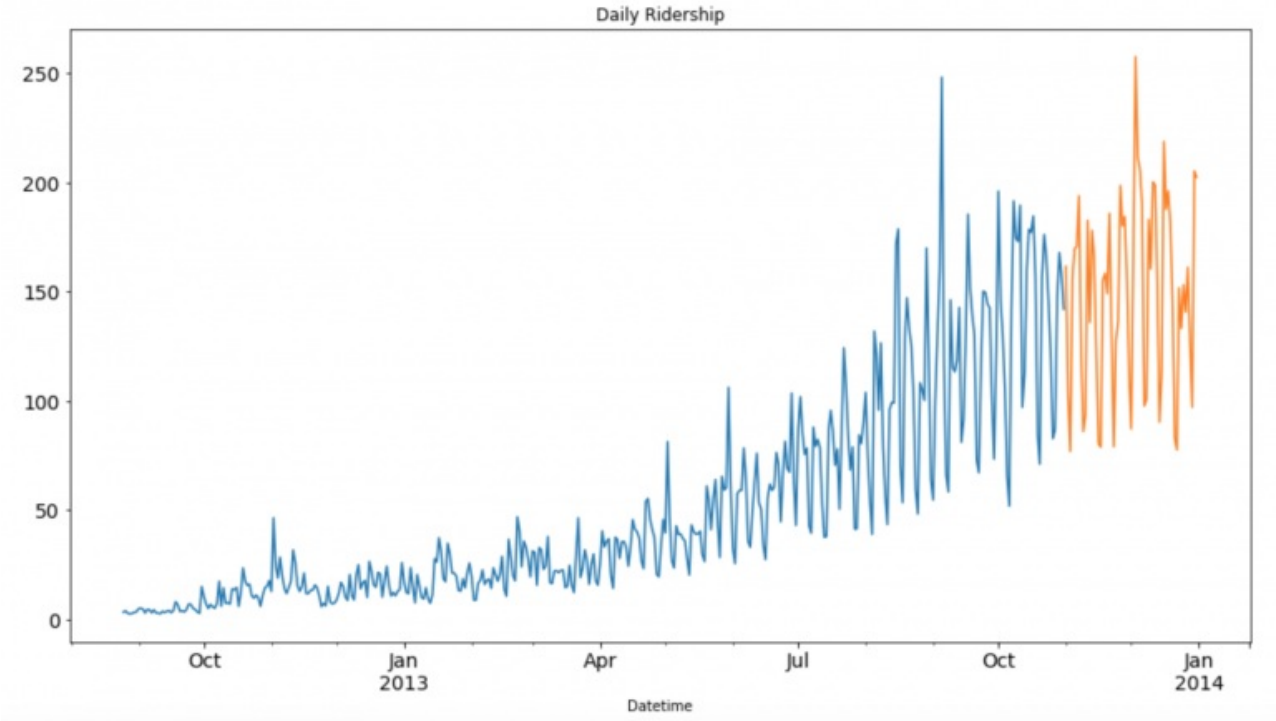
- Creating train and test file for modeling. The first 14 months (August 2012 – October 2013) are used as training data and next 2 months (Nov 2013 – Dec 2013) as testing data.
- Aggregating the dataset at daily basis

	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

```
#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()
```



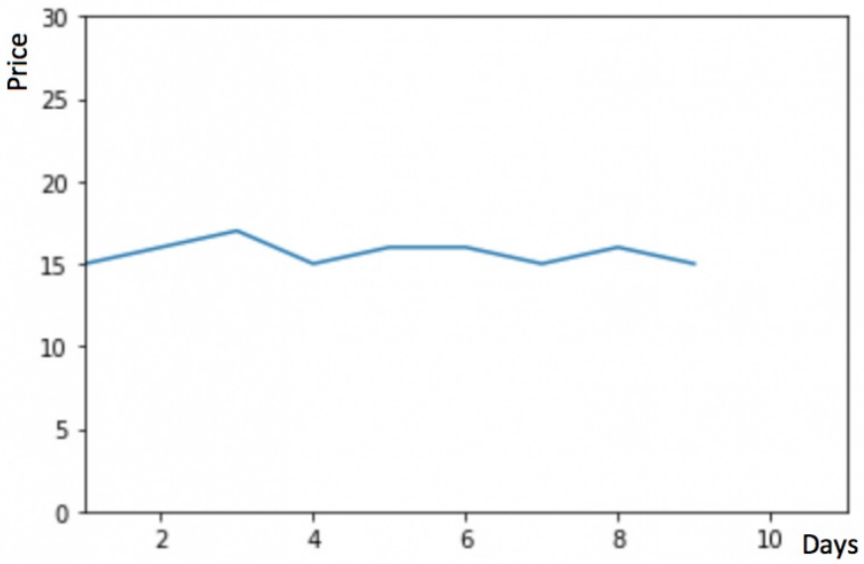
Installing library(statsmodels)

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.
    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).

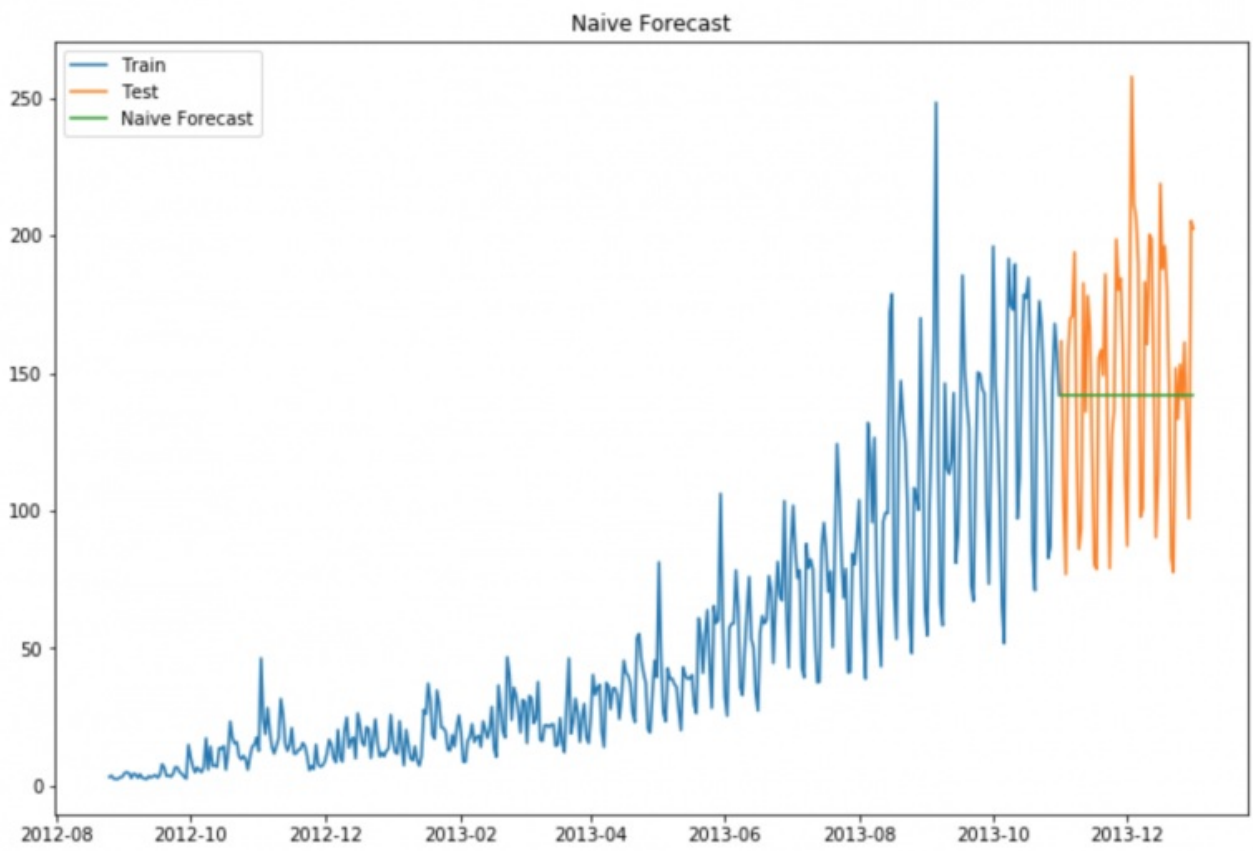


We can infer from the graph that the price of the coin is stable from the start. Many a times we are provided with a dataset, which is stable throughout it's time period. If we want to forecast the price for the next day, we can simply take the last day value and estimate the same value for the next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

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

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

```
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.

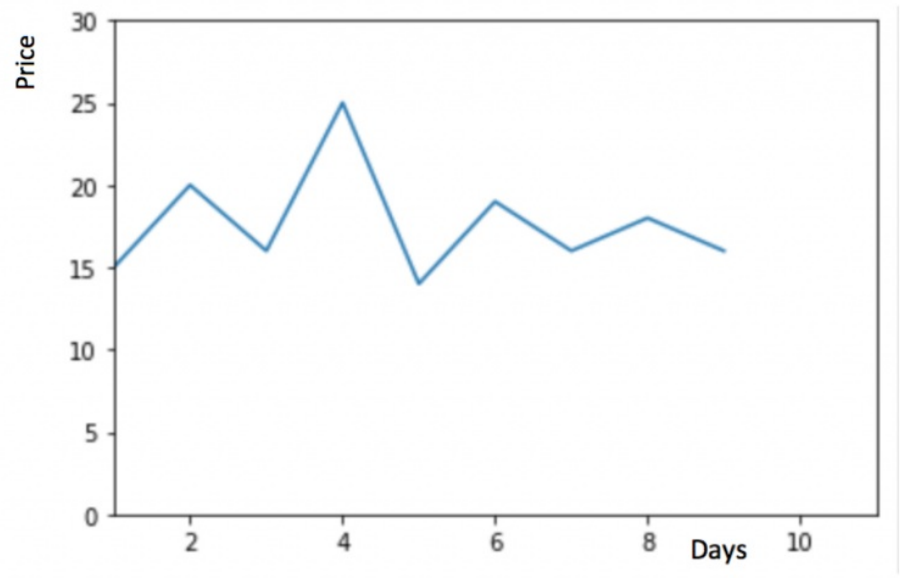
```
from sklearn.metrics import mean_squared_error
from math import sqrt
rms = sqrt(mean_squared_error(test.Count, y_hat.naive))
print(rms)
```

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).



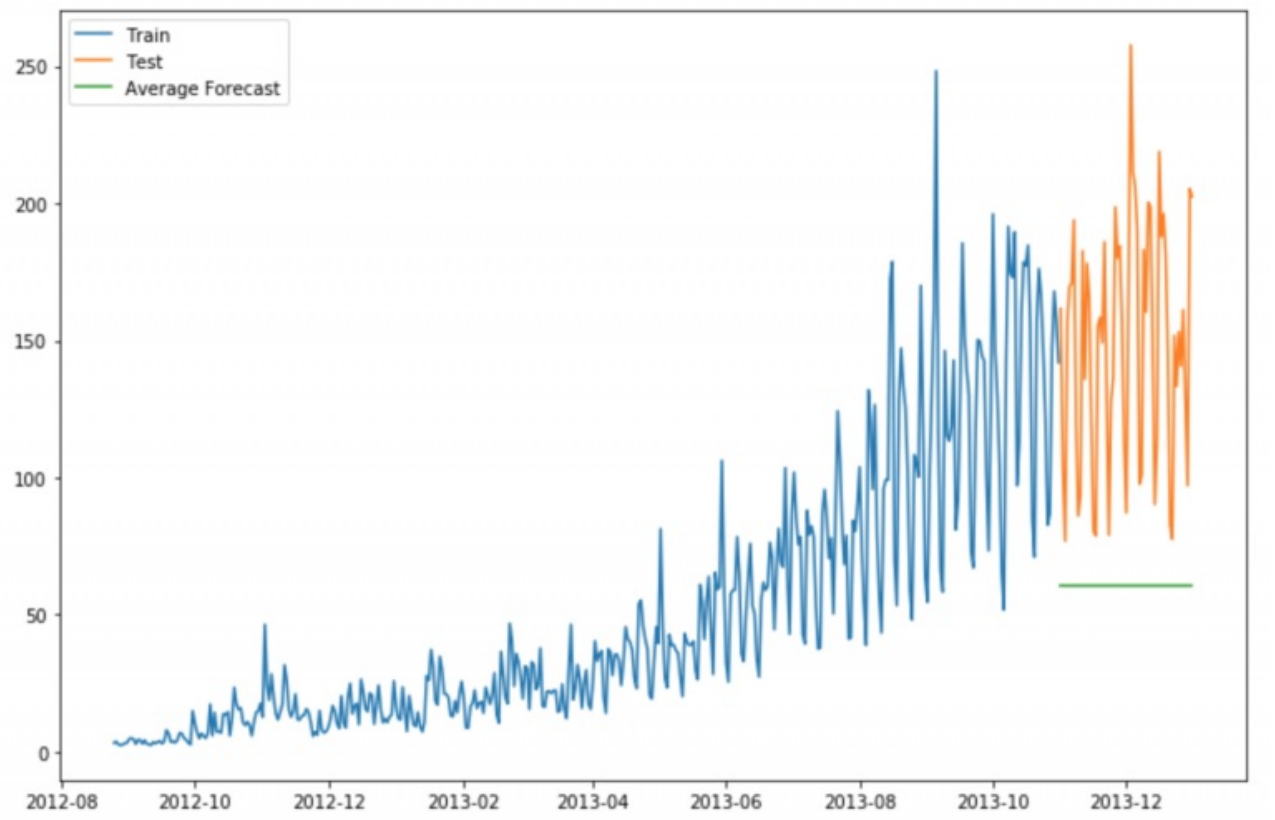
We can infer from the graph that the price of the coin is increasing and decreasing randomly by a small margin, such that the average remains constant. Many a times we are provided with a dataset, which though varies by a small margin throughout it's time period, but the average at each time period remains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

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.

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

```
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.

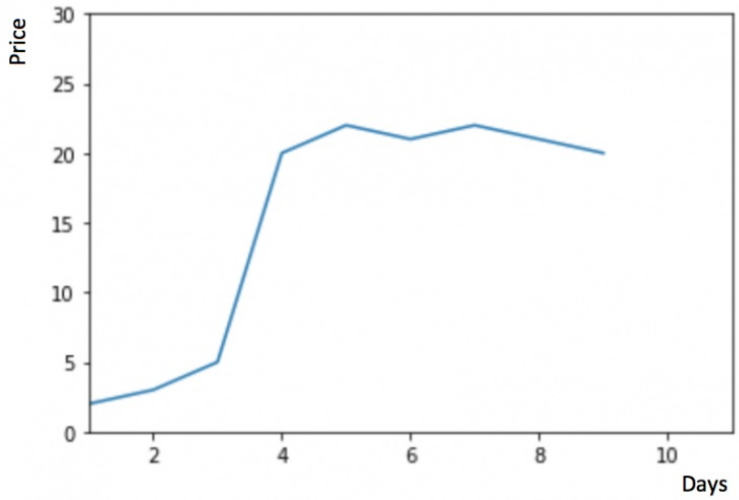
```
rms = sqrt(mean_squared_error(test.Count, y_hat_avg.avg_forecast))
print(rms)
```

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.

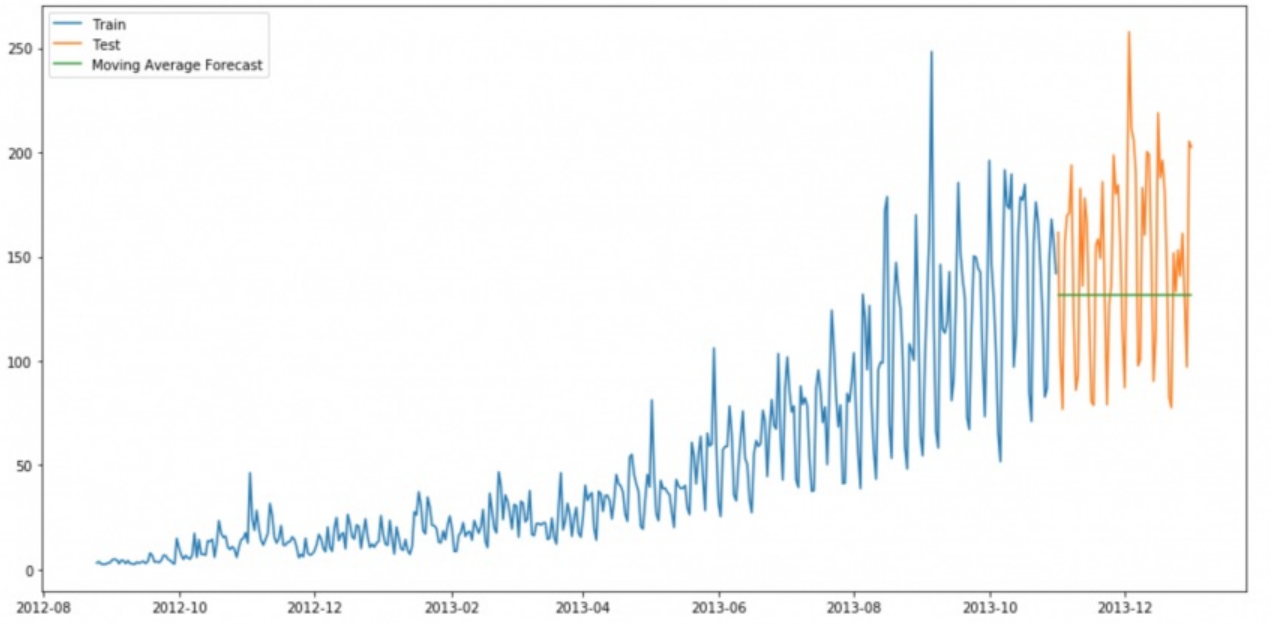
Using the prices of the initial period would highly affect the forecast for the next period. Therefore as an improvement over simple average, we will take the average of the prices for last few time periods only. Obviously the thinking here is that only the recent values matter. Such forecasting technique 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

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

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 $(1-\alpha) \cdot \hat{y}_{t-1}$.

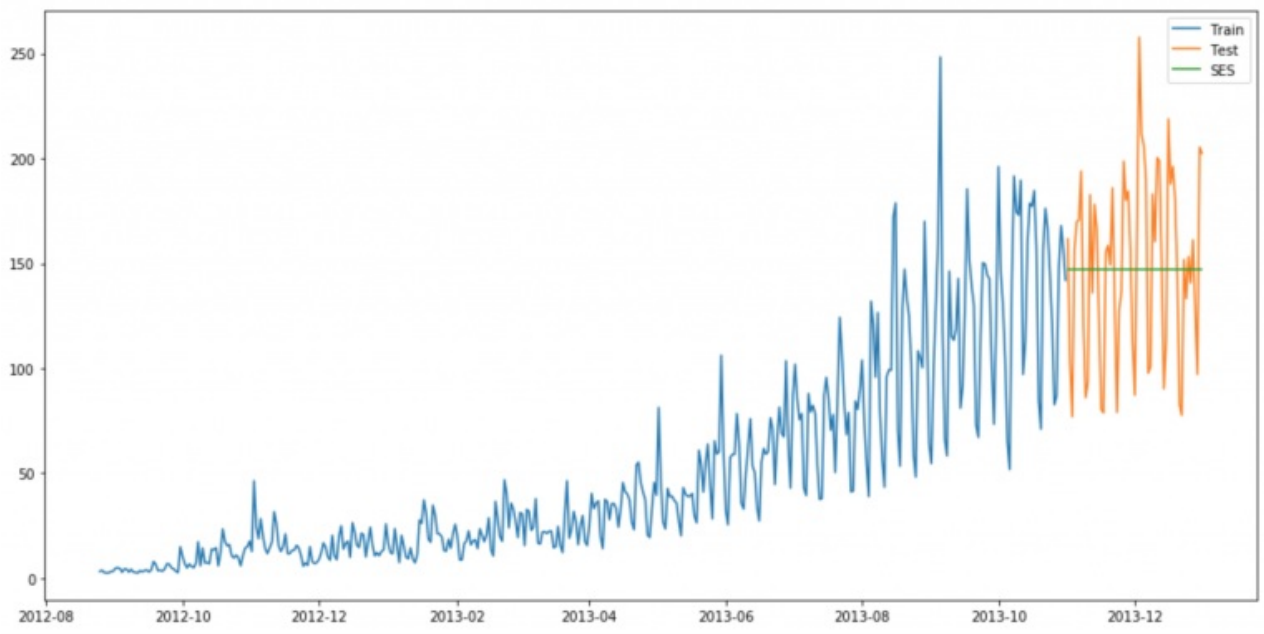
Hence, it can also be written as :

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

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

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.

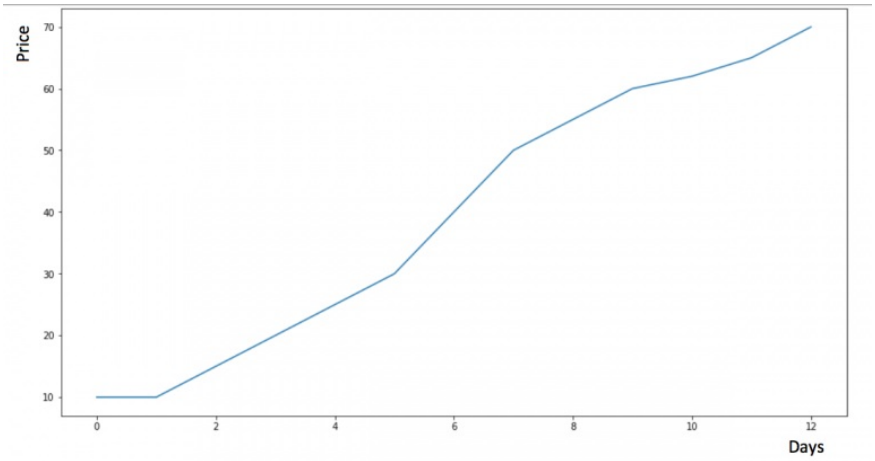
```
rms = sqrt(mean_squared_error(test.Count, y_hat_avg.SES))
print(rms)
```

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.



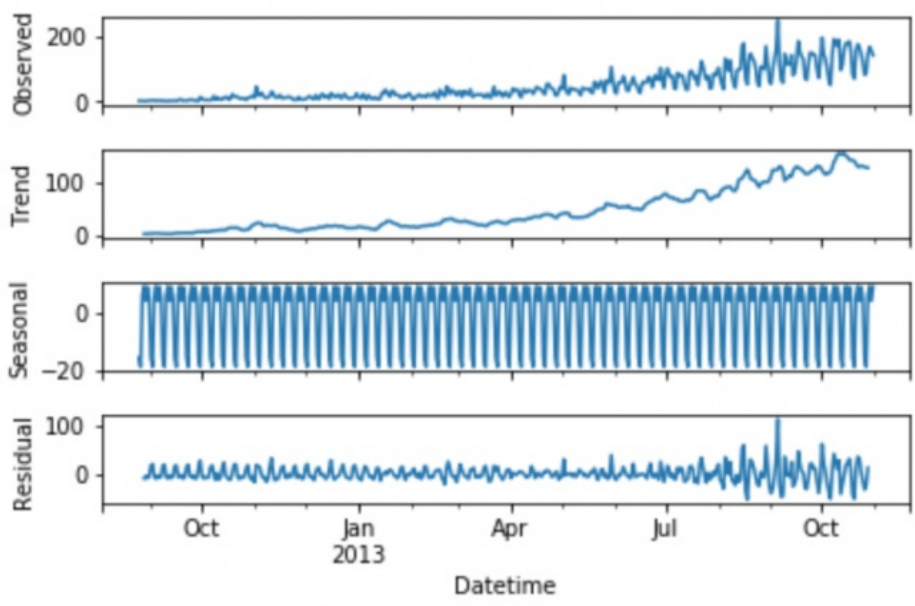
If we use any of the above methods, it won't take into account this trend. Trend is the general pattern of prices that we observe over a period of time. In this case we can see that there is an increasing trend.

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

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

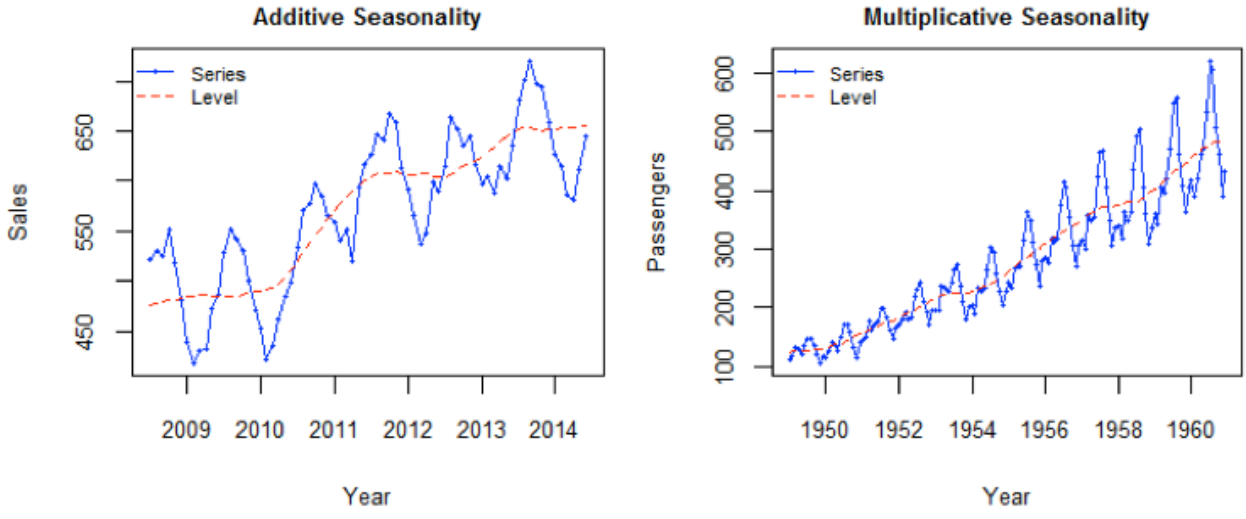
$$\text{Level equation : } \ell_t = \alpha y_t + (1-\alpha)(\ell_{t-1}+b_{t-1})$$

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

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.

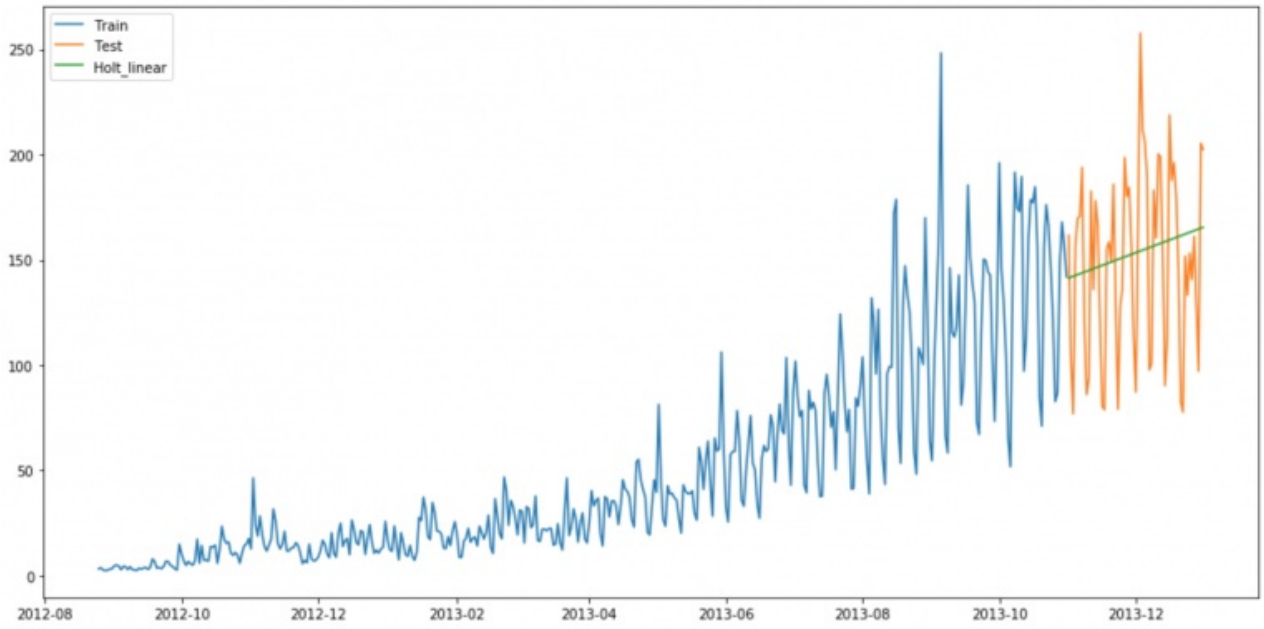
As with simple exponential smoothing, the level equation here shows that it is a weighted average of observation and the within-sample one-step-ahead forecast The trend equation shows that it is a weighted average of the estimated trend at time t based on ℓ(t)−ℓ(t−1) and b(t−1), the previous estimate 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)
```

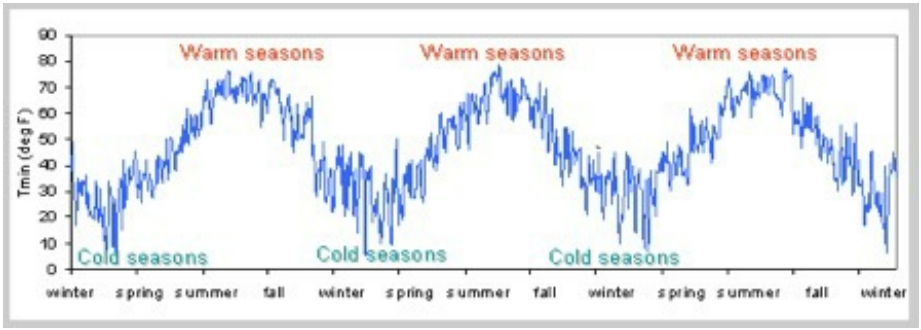
RMSE = 43.0562596115

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.

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

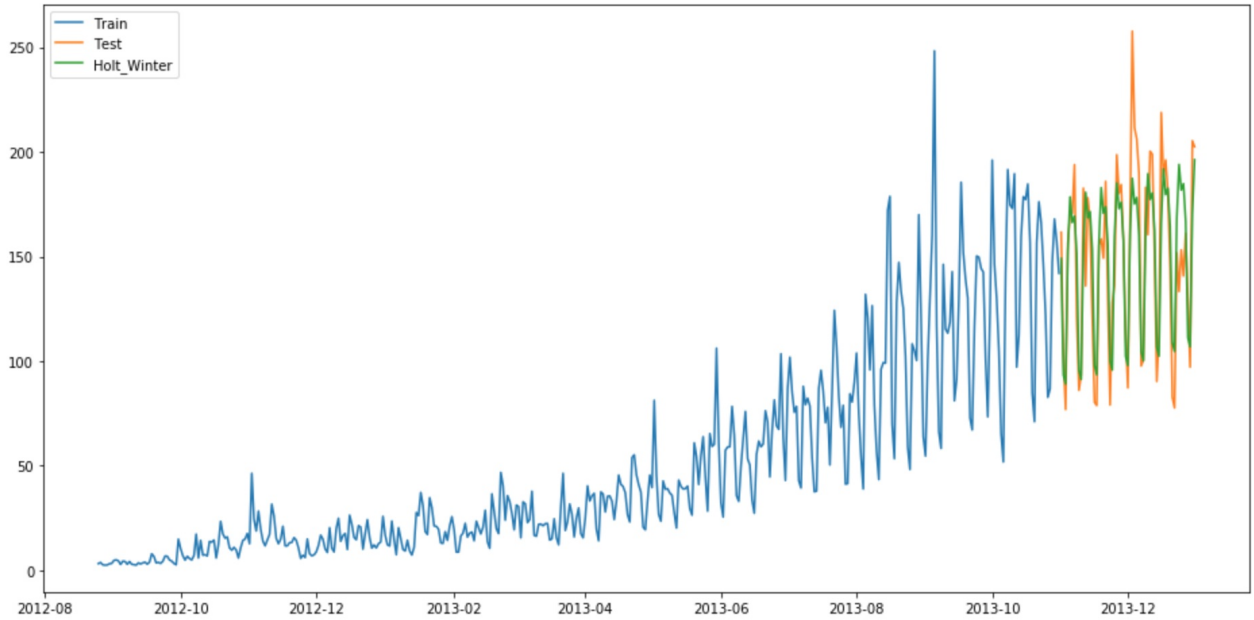
source

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',
seasonal='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

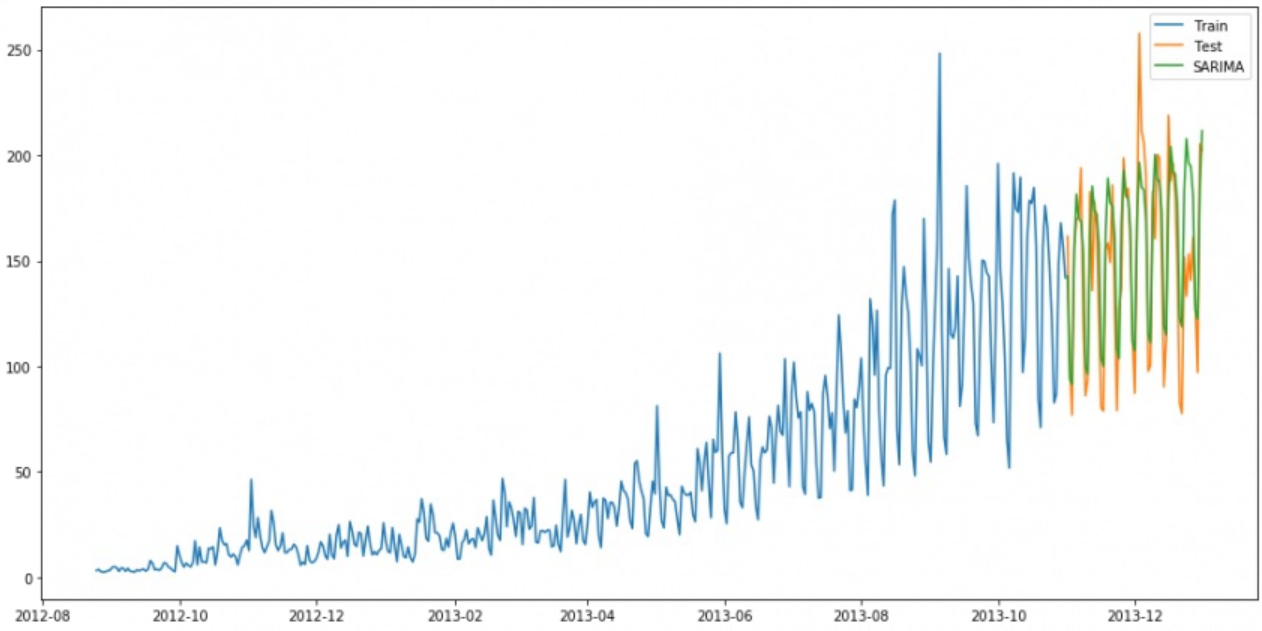
We can see from the graph that mapping correct trend and seasonality provides a far better solution. We chose seasonal_period = 7 as data repeats itself weekly. Other parameters can be tuned as per the dataset. I have used default parameters while building this model. You can tune the parameters to achieve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

Projects

Now, its time to take the plunge and actually play with some other real datasets. So are you ready to take on the challenge? Test the techniques discussed in this post and accelerate your learning in Time Series Analysis with the following Practice Problem:

	Practice Problem: Food Demand Forecasting Challenge	Forecast the demand of meals for a meal delivery company

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.

You can also explore **forecast** package built for Time series modelling in R language. You may also explore Double seasonality models from forecast package. Using double seasonality model on this dataset will generate even a better model and hence a better score.

Did you find this article helpful? Please share your opinions / thoughts in the comments section below.

40 Questions to test a data scientist on Time Series [Solution: SkillPower – Time Series, DataFest 2017]

analyticsvidhya.com/blog/2017/04/40-questions-on-time-series-solution-skillpower-time-series-datafest-2017

saurabh.jaju2 Saurabh is a data scientist and software engineer skilled at analyzing variety of datasets and developing smart applications. He is currently pursuing a Masters degree in Information and Data Science from University of California,Berkeley and is passionate about developing data science based smart resource management systems.

April 10, 2017

Introduction

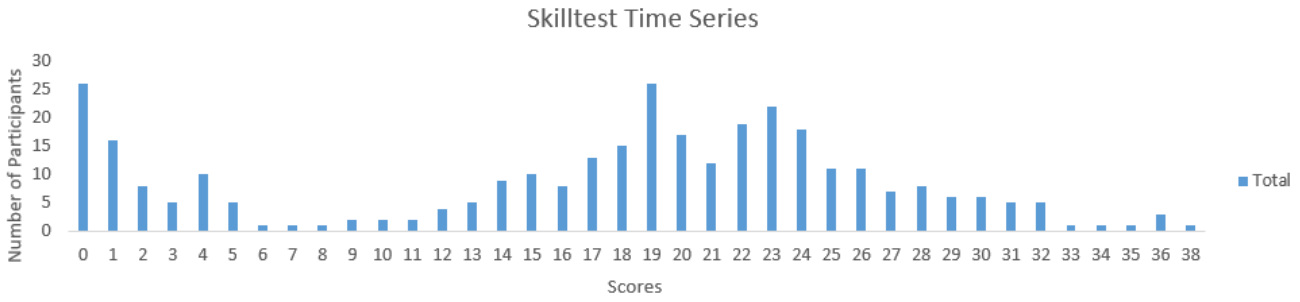
Time Series forecasting & modeling plays an important role in data analysis. Time series analysis is a specialized branch of statistics used extensively in fields such as Econometrics & Operation Research. This skilltest was conducted to test your knowledge of time series concepts.

A total of 1094 people registered for this skill test. The test was designed to test you on the basic & advanced level of time series. If you are one of those who missed out on this skill test, here are the questions and solutions. You missed on the real time test, but can read this article to find out how many could have answered correctly.

Here are the leaderboard ranking for all the participants.

Overall Scores

Below are the distribution scores, they will help you evaluate your performance.



You can access the scores here. More than 300 people participated in the skill test and the highest score obtained was 38. Here are a few statistics about the distribution.

Mean Score: 17.13

Median Score: 19

Mode Score: 19

Useful Resources

A Complete Tutorial on Time Series Modeling in R

A comprehensive beginner's guide to create a Time Series Forecast (with Codes in Python)

1) Which of the following is an example of time series problem?

1. Estimating number of hotel rooms booking in next 6 months.
2. Estimating the total sales in next 3 years of an insurance company.
3. Estimating the number of calls for the next one week.

A) Only 3
B) 1 and 2
C) 2 and 3
D) 1 and 3
E) 1,2 and 3

Solution: **(E)**

All the above options have a time component associated.

2) Which of the following is not an example of a time series model?

A) Naive approach
B) Exponential smoothing
C) Moving Average
D)None of the above

Solution: **(D)**

Naïve approach: Estimating technique in which the last period's actuals are used as this period's forecast, without adjusting them or attempting to establish causal factors. It is used only for comparison with the forecasts generated by the better (sophisticated) techniques.

In exponential smoothing, older data is given progressively-less relative importance whereas newer data is given progressively-greater importance.

In time series analysis, the moving-average (MA) model is a common approach for modeling univariate time series. The moving-average model specifies that the output variable depends linearly on the current and various past values of a stochastic (imperfectly predictable) term.

3) Which of the following can't be a component for a time series plot?

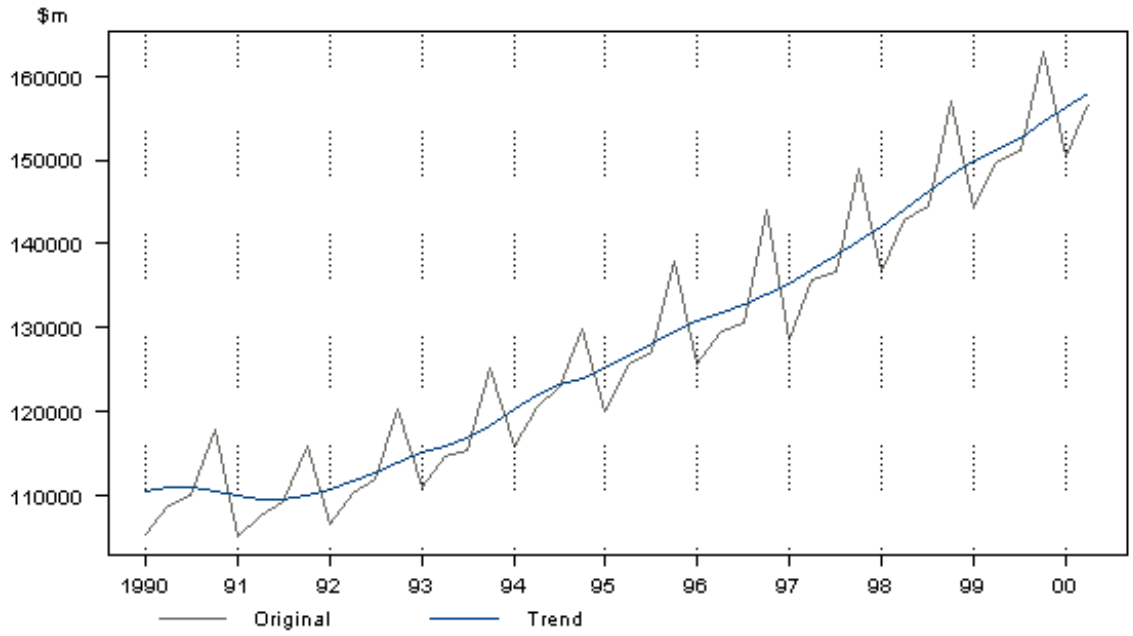
A) Seasonality
B) Trend
C) Cyclical
D) Noise
E) None of the above

Solution: **(E)**

A seasonal pattern exists when a series is influenced byseasonal factors (e.g., the quarter of the year, the month, or day of the week). Seasonality is always of a fixed and known period. Hence, seasonal time series are sometimes called periodic time series

Seasonality is always of a fixed and known period. A cyclic pattern exists when data exhibit rises and falls that are not of fixed period.

Trend is defined as the 'long term' movement in a time series without calendar related and irregular effects, and is a reflection of the underlying level. It is the result of influences such as population growth, price inflation and general economic changes. The following graph depicts a series in which there is an obvious upward trend over time.



Quarterly Gross Domestic Product

Noise: In discrete time, white noise is a discrete signal whose samples are regarded as a sequence of serially uncorrelated random variables with zero mean and finite variance.

Thus all of the above mentioned are components of a time series.

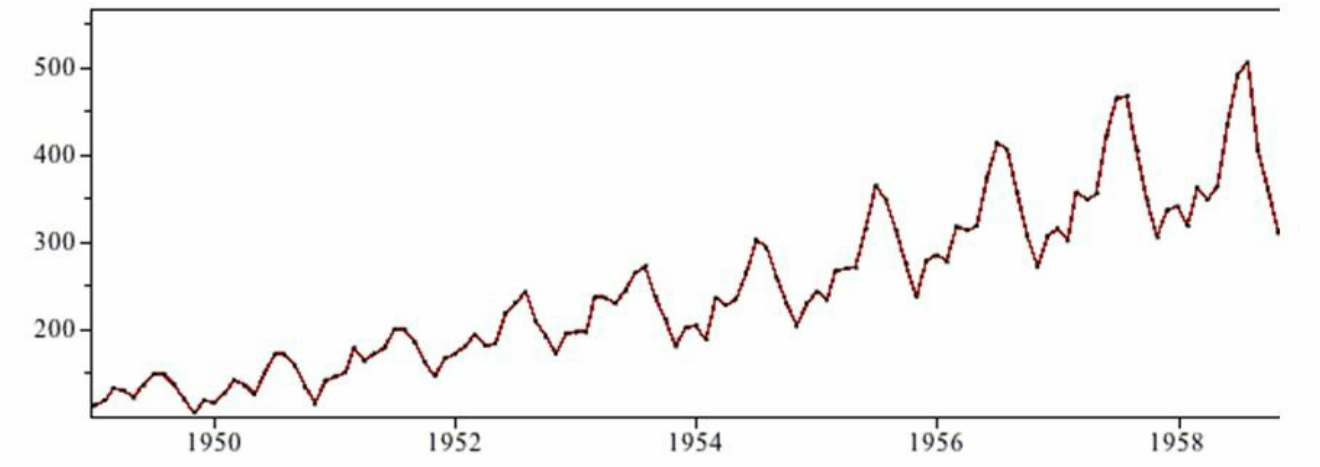
4) Which of the following is relatively easier to estimate in time series modeling?

A) Seasonality
B) Cyclical
C) No difference between Seasonality and Cyclical

Solution: **(A)**

As we seen in previous solution, as seasonality exhibits fixed structure; it is easier to estimate.

5) The below time series plot contains both Cyclical and Seasonality component.



A) TRUE
B) FALSE

Solution: **(B)**

There is a repeated trend in the plot above at regular intervals of time and is thus only seasonal in nature.

6) Adjacent observations in time series data (excluding white noise) are independent and identically distributed (IID).

A) TRUE

B) FALSE

Solution: **(B)**

Clusters of observations are frequently correlated with increasing strength as the time

intervals between them become shorter. This needs to be true because in time series forecasting is done based on previous observations and not the currently observed data unlike classification or regression.

7) Smoothing parameter close to one gives more weight or influence to recent observations over the forecast.

A) TRUE
B) FALSE

Solution: **(A)**

It may be sensible to attach larger weights to more recent observations than to observations from the distant past. This is exactly the concept behind 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:

$\hat{y}_{T+1|T}=\alpha y_T+\alpha(1-\alpha)y_T-1+\alpha(1-\alpha)2y_{T-2}+\cdots,(7.1)$

where $0\leq\alpha\leq10\leq\alpha\leq1$ is the smoothing parameter. The one-step-ahead forecast for time T+1T+1 is a weighted average of all the observations in the series $y_1,\ldots,y_T$. The rate at which the weights decrease is controlled by the parameter αα.

8) Sum of weights in exponential smoothing is ____.

A) <1
B) 1
C) >1
D) None of the above

Solution: **(B)**

Table 7.1 shows the weights attached to observations for four different values of αα when forecasting using simple exponential smoothing. Note that the sum of the weights even for a small αα will be approximately one for any reasonable sample size.

yT	0.2	0.4	0.6	0.8
yT−1	0.16	0.24	0.24	0.16
yT−2	0.128	0.144	0.096	0.032
yT−3	0.102	0.0864	0.0384	0.0064

yT−4	(0.2)(0.8)	(0.4)(0.6)	(0.6)(0.4)	(0.8)(0.2)
yT−5	(0.2)(0.8)	(0.4)(0.6)	(0.6)(0.4)	(0.8)(0.2)

9) The last period's forecast was 70 and demand was 60. What is the simple exponential smoothing forecast with alpha of 0.4 for the next period.

A) 63.8
B) 65
C) 62
D) 66

Solution: **(D)**

Y_{t-1}= 70

S_{t-1}= 60

Alpha = 0.4

Substituting the values we get

0.4*60 + 0.6*70= 24 + 42= 66

10) What does autocovariance measure?

A) Linear dependence between multiple points on the different series observed at different times
B)Quadratic dependence between two points on the same series observed at different times
C) Linear dependence between two points on different series observed at same time
D) Linear dependence between two points on the same series observed at different times

Solution: **(D)**

Option D is the definition of autocovariance.

11) Which of the following is not a necessary condition for weakly stationary time series?

A) Mean is constant and does not depend on time
B) Autocovariance function depends on s and t only through their difference |s-t| (where t and s are moments in time)
C) The time series under considerations is a finite variance process
D) Time series is Gaussian

Solution: **(D)**

A Gaussian time series implies stationarity is strict stationarity.

12) Which of the following is not a technique used in smoothing time series?

A) Nearest Neighbour Regression
B) Locally weighted scatter plot smoothing
C) Tree based models like (CART)
D) Smoothing Splines

Solution: **(C)**

Time series smoothing and filtering can be expressed in terms of local regression models. Polynomials and regression splines also provide important techniques for smoothing. CART based models do not provide an equation to superimpose on time series and thus cannot be used for smoothing. All the other techniques are well documented smoothing techniques.

13) If the demand is 100 during October 2016, 200 in November 2016, 300 in December 2016, 400 in January 2017. What is the 3-month simple moving average for February 2017?

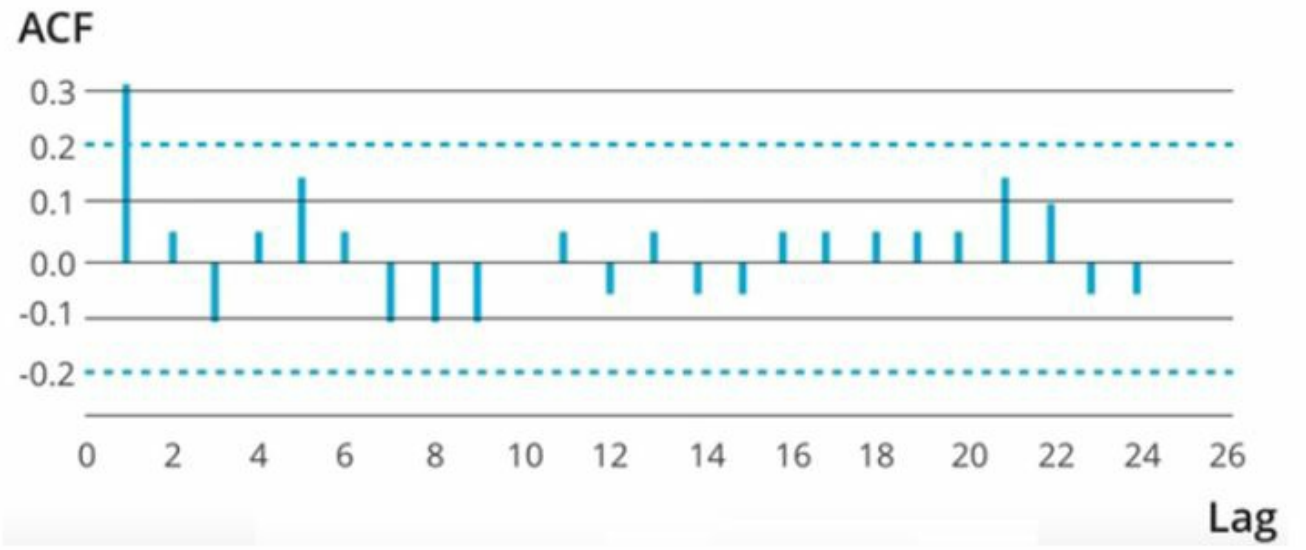
A) 300
B) 350
C) 400
D) Need more information

Solution: **(A)**

$X` = (x_{t-3} + x_{t-2} + x_{t-1}) /3$

(200+300+400)/ 3 = 900/3 =300

14) Looking at the below ACF plot, would you suggest to apply AR or MA in ARIMA modeling technique?



A) AR
B) MA
C) Can't Say

Solution: **(A)**

MA model is considered in the following situation, If the autocorrelation function (ACF) of the differenced series displays a sharp cutoff and/or the lag-1 autocorrelation is negative–i.e., if the series appears slightly "overdifferenced"–then consider adding an MA term to the model. The lag beyond which the ACF cuts off is the indicated number of MA terms.

But as there are no observable sharp cutoffs the AR model must be preffered.

15) Suppose, you are a data scientist at Analytics Vidhya. And you observed the views on the articles increases during the month of Jan-Mar. Whereas the views during Nov-Dec decreases.

Does the above statement represent seasonality?

A) TRUE
B) FALSE
C) Can't Say

Solution: **(A)**

Yes this is a definite seasonal trend as there is a change in the views at particular times.

Remember, Seasonality is a presence of variations at specific periodic intervals.

16) Which of the following graph can be used to detect seasonality in time series data?

1. Multiple box
2. Autocorrelation

A) Only 1
B) Only 2
C) 1 and 2
D) None of these

Solution: **(C)**

Seasonality is a presence of variations at specific periodic intervals.

The variation of distribution can be observed in multiple box plots. And thus seasonality can be easily spotted. *Autocorrelation plot* should show spikes at lags equal to the period.

17) Stationarity is a desirable property for a time series process.

A) TRUE
B) FALSE

Solution: **(A)**

When the following conditions are satisfied then a time series is stationary.

1. Mean is constant and does not depend on time
2. Autocovariance function depends on s and t only through their difference |s-t| (where t and s are moments in time)
3. The time series under considerations is a finite variance process

These conditions are essential prerequisites for mathematically representing a time series to be used for analysis and forecasting. Thus stationarity is a desirable property.

18) Suppose you are given a time series dataset which has only 4 columns (id, Time, X, Target).

id	Time	X	Target
1	1	100	10
2	2	200	20
3	3	300	30
1	4	400	40
2	5	500	50
3	6	600	60
1	7	500	50
2	8	400	40
3	9	500	30
4	10	700	20

What would be the rolling mean of feature X if you are given the window size 2?

Note: X column represents rolling mean.

A)

Id	Time	X'	Target
1	1	NaN	10
2	2	200	20
3	3	150	30
1	4	250	40
2	5	350	50
3	6	450	60
1	7	550	50
2	8	550	40
3	9	450	30
4	10	450	20

B)

Quater	Time	X'	Target
1	1	NaN	10
2	2	NaN	20
3	3	150	30
1	4	250	40
2	5	350	50
3	6	450	60
1	7	550	50
2	8	550	40
3	9	450	30
4	10	450	20

C)

Id	Time	X'	Target
1	1	NaN	10
2	2	200	20
3	3	150	30
1	4	250	40
2	5	350	50
3	6	450	60
1	7	550	50
2	8	550	40
3	9	450	30
4	10	NaN	20

D) None of the above

Solution: **(B)**

$X` = x_{t-2} + x_{t-1} /2$

Based on the above formula: (100 +200) /2 =150; (200+300)/2 = 250 and so on.

19) Imagine, you are working on a time series dataset. Your manager has asked you to build a highly accurate model. You started to build two types of models which are given below.

Model 1: Decision Tree model

Model 2: Time series regression model

At the end of evaluation of these two models, you found that model 2 is better than model 1. What could be the possible reason for your inference?

A) Model 1 couldn't map the linear relationship as good as Model 2
B) Model 1 will always be better than Model 2
C) You can't compare decision tree with time series regression
D) None of these

Solution: **(A)**

A time series model is similar to a regression model. So it is good at finding simple linear relationships. While a tree based model though efficient will not be as good at finding and exploiting linear relationships.

20) What type of analysis could be most effective for predicting temperature on the following type of data.

Date	Temperature	precipitation	temperature/precipitation
12/12/12	7	0.2	35
13/12/12	9	0.123	73.1707317073
14/12/12	9.2	0.34	27.0588235294
15/12/12	10	0.453	22.0750551876
16/12/12	12	0.33	36.3636363636
17/12/12	11	0.8	13.75

A) Time Series Analysis
B) Classification

C) Clustering
D) None of the above

Solution: **(A)**

The data is obtained on consecutive days and thus the most effective type of analysis will be time series analysis.

21) What is the first difference of temperature / precipitation variable?

Date	Temperature	precipitation	temperature/precipitation
12/12/12	7	0.2	35
13/12/12	9	0.123	73.1707317073
14/12/12	9.2	0.34	27.0588235294
15/12/12	10	0.453	22.0750551876
16/12/12	12	0.33	36.3636363636
17/12/12	11	0.8	13.75

A) 15,12.2,-43.2,-23.2,14.3,-7
B) 38.17,-46.11,-4.98,14.29,-22.61
C) 35,38.17,-46.11,-4.98,14.29,-22.61
D) 36.21,-43.23,-5.43,17.44,-22.61

Solution: **(B)**

73.17-35 = 38.17

27.05-73.17 = – 46.11 and so on..

13.75 – 36.36 = -22.61

22) Consider the following set of data:

{23.32 32.33 32.88 28.98 33.16 26.33 29.88 32.69 18.98 21.23 26.66 29.89}

What is the lag-one sample autocorrelation of the time series?

A) 0.26
B) 0.52
C) 0.13

D) 0.07

Solution: **(C)**

$\hat{\rho}_1 = PT_{t=2}(x_{t-1}-\bar{x})(x_t-\bar{x})\ PT_{t=1}(xt-\bar{x})^2$

$= (23.32-\bar{x})(32.33-\bar{x})+(32.33-\bar{x})(32.88-\bar{x})+\cdots\ PT_{t=1}(x_t-\bar{x})^2$

= 0.130394786

Where $\bar{x}$ is the mean of the series which is 28.0275

23) Any stationary time series can be approximately the random superposition of sines and cosines oscillating at various frequencies.

A) TRUE
B) FALSE

Solution: **(A)**

A weakly stationary time series, xt, is a finite variance process such that

> The mean value function, μt, is constant and does not depend on time t, and (ii) the autocovariance function, γ(s,t), defined in depends on s and t only through their difference |s−t|.

random superposition of sines and cosines oscillating at various frequencies is white noise. white noise is weakly stationary or stationary. If the white noise variates are also normally distributed or Gaussian, the series is also strictly stationary.

24) Autocovariance function for weakly stationary time series does not depend on _____ ?

A) Separation of xs and xt
B) h = | s – t |
C) Location of point at a particular time

Solution: **(C)**

By definition of weak stationary time series described in previous question.

25) Two time series are jointly stationary if ____ ?

A) They are each stationary
B) Cross variance function is a function only of lag h

A) Only A
B) Both A and B

Solution: **(D)**

Joint stationarity is defined based on the above two mentioned conditions.

26) In autoregressive models _____ ?

A) Current value of dependent variable is influenced by current values of independent variables
B) Current value of dependent variable is influenced by current and past values of independent variables
C) Current value of dependent variable is influenced by past values of both dependent and independent variables
D) None of the above

Solution: **(C)**

Autoregressive models are based on the idea that the current value of the series, xt, can be explained as a function of p past values, xt−1,xt−2,...,xt−p, where p determines the number of steps into the past needed to forecast the current value. Ex. xt = xt−1 −.90xt−2 + wt,

Where xt-1 and xt-2 are past values of dependent variable and wt the white noise can represent values of independent values.

The example can be extended to include multiple series analogous to multivariate linear regression.

27) For MA (Moving Average) models the pair σ = 1 and θ = 5 yields the same autocovariance function as the pair σ = 25 and θ = 1/5.

$$x_t = w_t + \frac{1}{5}w_{t-1}, \quad w_t \sim \text{iid} \quad \text{N}(0, 25)$$

$$y_t = v_t + 5v_{t-1}, \quad v_t \sim \text{iid} \quad \text{N}(0, 1)$$

A) TRUE
B) FALSE

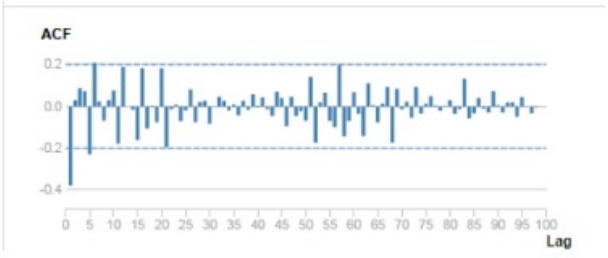
Solution: **(A)**

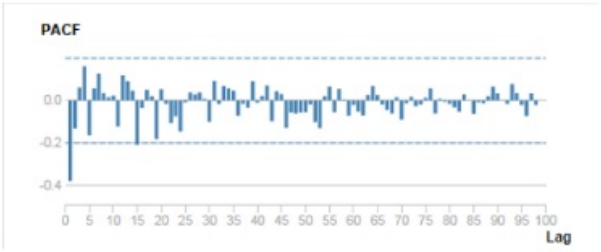
True, because autocovariance is invertible for MA models

note that for an MA(1) model, ρ(h) is the same for θ and 1 /θ

try 5 and 1 5, for example. In addition, the pair σ^2 w = 1 and θ = 5 yield the same autocovariance function as the pair σ^2 w = 25 and θ = 1/5.

28) How many AR and MA terms should be included for the time series by looking at the above ACF and PACF plots?





A) AR (1) MA(0)
B) AR(0)MA(1)
C) AR(2)MA(1)
D) AR(1)MA(2)
E) Can't Say

Solution: **(B)**

Strong negative correlation at lag 1 suggest MA and there is only 1 significant lag. Read this article for a better understanding.

29) Which of the following is true for white noise?

A) Mean =0
B) Zero autocovariances
C) Zero autocovariances except at lag zero
D) Quadratic Variance

Solution: **(C)**

A white noise process must have a constant mean, a constant variance and no autocovariance structure (except at lag zero, which is the variance).

30) For the following MA (3) process $y_t = \mu + E_t + \theta_1 E_{t-1} + \theta_2 E_{t-2} + \theta_3 E_{t-3}$, where σ_t is a zero mean white noise process with variance σ^2

A) ACF = 0 at lag 3
B) ACF =0 at lag 5
C) ACF =1 at lag 1

D) ACF =0 at lag 2
E) ACF = 0 at lag 3 and at lag 5

Solution: **(B)**

Recall that an MA(q) process only has memory of length q. This means that all of the autocorrelation coefficients will have a value of zero beyond lag q. This can be seen by examining the MA equation, and seeing that only the past q disturbance terms enter into the equation, so that if we iterate this equation forward through time by more than q periods, the current value of the disturbance term will no longer affect y. Finally, since the autocorrelation function at lag zero is the correlation of y at time t with y at time t (i.e. the correlation of y_t with itself), it must be one by definition.

31) Consider the following AR(1) model with the disturbances having zero mean and unit variance.

$$y_t = 0.4 + 0.2y_{t-1} + u_t$$

The (unconditional) variance of y will be given by ?

A) 1.5
B) 1.04
C) 0.5
D) 2

Solution: **(B)**

Variance of the disturbances divided by (1 minus the square of the autoregressive coefficient

Which in this case is : 1/(1-(0.2^2))= 1/0.96= 1.041

32) The pacf (partial autocorrelation function) is necessary for distinguishing between _____ ?

A) An AR and MA model is_solution: False
B) An AR and an ARMA is_solution: True
C) An MA and an ARMA is_solution: False
D) Different models from within the ARMA family

Solution: **(B)**

Table 3.1. Behavior of the ACF and PACF for ARMA Models

	AR(p)	MA(q)	ARMA(p, q)
ACF	Tails off	Cuts off after lag q	Tails off
PACF	Cuts off after lag p	Tails off	Tails off

33) Second differencing in time series can help to eliminate which trend?

A) Quadratic Trend
B) Linear Trend
C) Both A & B
D) None of the above

Solution: (A)

The first difference is denoted as $\nabla x_t = x_t - x_t - 1$. (1)

As we have seen, the first difference eliminates a linear trend. A second difference, that is, the difference of (1), can eliminate a quadratic trend, and so on.

34) Which of the following cross validation techniques is better suited for time series data?

A) k-Fold Cross Validation
B) Leave-one-out Cross Validation
C) Stratified Shuffle Split Cross Validation
D) Forward Chaining Cross Validation

Solution: **(D)**

Time series is ordered data. So the validation data must be ordered to. Forward chaining ensures this. It works as follows:

- fold 1 : training [1], test [2]
- fold 2 : training [1 2], test [3]
- fold 3 : training [1 2 3], test [4]
- fold 4 : training [1 2 3 4], test [5]
- fold 5 : training [1 2 3 4 5], test [6]

35) BIC penalizes complex models more strongly than the AIC.

A) TRUE
B) FALSE

Solution: **(A)**

AIC = -2*ln(likelihood) + 2*k,

BIC = -2*ln(likelihood) + ln(N)*k,

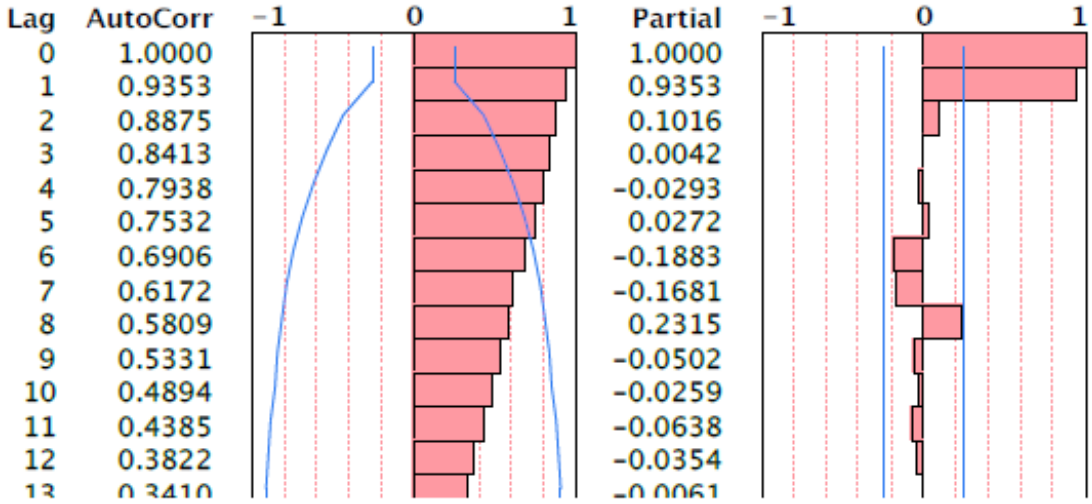
where:

k = model degrees of freedom

N = number of observations

At relatively low N (7 and less) BIC is more tolerant of free parameters than AIC, but less tolerant at higher N (as the natural log of N overcomes 2).

36) The figure below shows the estimated autocorrelation and partial autocorrelations of a time series of n = 60 observations. Based on these plots, we should.



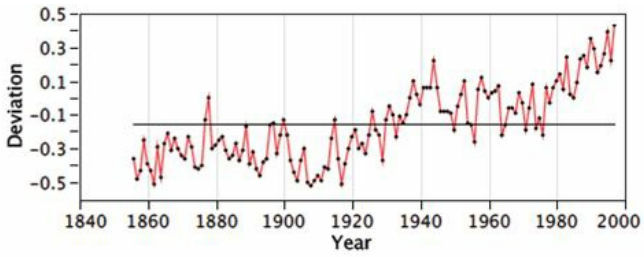
A) Transform the data by taking logs
B) Difference the series to obtain stationary data
C) Fit an MA(1) model to the time series

Solution: **(B)**

The autocorr shows a definite trend and partial autocorrelation shows a choppy trend, in such a scenario taking a log would be of no use. Differencing the series to obtain a stationary series is the only option.

Question Context (37-38)

The remaining questions consider a time series model for annual global temperature. The data for the time series in this analysis begin in 1856 and run through 1997 (n = 142). The measurements give the deviation from typical temperature in degrees Celsius. (Zero would be considered consistent with the long-run average.)



Model Summary

DF	140.0000
Sum of Squared Errors	1.7726
Variance Estimate	0.0127
Standard Deviation	0.1125
Akaike's 'A' Information Criterion	-214.4648
Schwarz's Bayesian Criterion	-211.5160
RSquare	0.7328

Parameter Estimates

Term	Estimate	Std Error	t Ratio	Prob>|t|
Level Smoothing Weight	0.39680005	0.0900926	4.40	<.0001*

	Actual Deviation	Year	Predicted Deviation
133	0.25	1988	0.13245007
134	0.18	1989	0.17909389
135	0.35	1990	0.17945343
136	0.29	1991	0.24712632
137	0.15	1992	0.2641386
138	0.19	1993	0.2188484
139	0.26	1994	0.20740135
140	0.39	1995	0.2282725
141	0.22	1996	0.29244598
142	0.43	1997	0.26369941

37) Use the estimated exponential smoothening given above and predict temperature for the next 3 years (1998-2000)

These results summarize the fit of a simple exponential smooth to the time series.

A) 0.2,0.32,0.6
B) 0.33, 0.33,0.33
C) 0.27,0.27,0.27
D) 0.4,0.3,0.37

Solution: **(B)**

The predicted value from the exponential smooth is the same for all 3 years, so all we need is the value for next year. The expression for the smooth is

$\text{smooth}_t = \alpha\, y_t + (1 - \alpha)\, \text{smooth}_{t-1}$

Hence, for the next point, the next value of the smooth (the prediction for the next observation) is

$\text{smooth}_n = \alpha\, y_n + (1 - \alpha)\, \text{smooth}_{n-1}$

$= 0.3968*0.43 + (1 - 0.3968)* 0.3968$

$= 0.3297$

38) Find 95% prediction intervals for the predictions of temperature in 1999.

These results summarize the fit of a simple exponential smooth to the time series.

A) 0.3297 2 * 0.1125
B) 0.3297 2 * 0.121
C) 0.3297 2 * 0.129
D) 0.3297 2 * 0.22

Solution: **(B)**

The sd of the prediction errors is

1 period out 0.1125

2 periods out 0.1125 sqrt(1+α2) = 0.1125 * sqrt(1+ 0.39682) ≈ 0.121

39) Which of the following statement is correct?

1. If autoregressive parameter (p) in an ARIMA model is 1, it means that there is no auto-correlation in the series.
2. If moving average component (q) in an ARIMA model is 1, it means that there is auto-correlation in the series with lag 1.
3. If integrated component (d) in an ARIMA model is 0, it means that the series is not stationary.

A) Only 1
B) Both 1 and 2
C) Only 2
D) All of the statements

Solution: **(C)**

Autoregressive component: AR stands for autoregressive. Autoregressive parameter is denoted by p. When p =0, it means that there is no auto-correlation in the series. When p=1, it means that the series auto-correlation is till one lag.

Integrated: In ARIMA time series analysis, integrated is denoted by d. Integration is the inverse of differencing. When d=0, it means the series is stationary and we do not need to take the difference of it. When d=1, it means that the series is not stationary and to make it stationary, we need to take the first difference. When d=2, it means that the series has been differenced twice. Usually, more than two time difference is not reliable.

Moving average component: MA stands for moving the average, which is denoted by q. In ARIMA, moving average q=1 means that it is an error term and there is auto-correlation with one lag.

40) In a time-series forecasting problem, if the seasonal indices for quarters 1, 2, and 3 are 0.80, 0.90, and 0.95 respectively. What can you say about the seasonal index of quarter 4?

A) It will be less than 1
B) It will be greater than 1
C) It will be equal to 1
D) Seasonality does not exist
E) Data is insufficient

Solution: **(B)**

The seasonal indices must sum to 4, since there are 4 quarters. .80 + .90 + .95 = 2.65, so the seasonal index for the 4^{th} quarter must be 1.35 so B is the correct answer.