Smart System for notification and updates of Health Issues and their prevention.

Submitted to
Dr Bhaskar Biswas
Dept. of Computer Science and Engineering
IIT (BHU)-Varanasi

Table of Contents

Acknowledgement

We would like to express our special thanks of gratitude to our Mentor, **Dr Bhaskar Biswas**, who gave us the golden opportunity to do this wonderful project.

We also want to thank **M.S. -S.S.H**, who gave us the required data to complete our project.

By this project, we came to know about so many new things and we are really thankful to them.

Abstract

Maintaining drug inventory while providing appropriate patient care remains a daily challenge for Hospitals. Many times, Hospitals fall short of Medicines, which their doctors prescribe to the patients, in such cases the patient has to suffer, we have developed a system using Machine Learning Algorithms for Hospitals which will predict the quantity of medicines and trend of diseases.

This model also predicts disease trends by mapping them with their respective medicines trends using python dictionary having disease as key and medicines as their values.

Also by comparing with different methods, this one proves to have better results of trends and amounts of medicine. Errors are calculated in rms values and trends of diseases are scaled as number of patients affected in a particular month of a year per maximum number of patients affected in a particular month of a year since 2004.

AIM OF THE PROJECT:

"Amount of medicines" required in-hand by the hospitals so that they will not fall short of the medicines for a month or two and all the patients will be diagnosed properly.

We can also predict which diseases are going to spread in particular area for a particular time of the year and so that necessary measures can be taken to prevent the spreading of those particular diseases.

Neural Network:

A Neural Network (NN) is a computational model that is inspired by the way biological neural networks in the human brain process information. Artificial Neural Networks have generated a lot of excitement in Machine Learning research and industry, thanks to many breakthrough results in speech recognition, computer vision and text processing. In this blog post we will try to develop an understanding of a particular type of Artificial Neural Network called the Multi Layer Perceptron.

A Single Neuron

The basic unit of computation in a neural network is the **neuron**, often called a **node** or **unit**. It receives input from some other nodes, or from an external source and computes an output. Each input has an associated **weight** (w), which is assigned on the basis of its relative importance to other inputs. The node applies a function ***f*** (defined below) to the weighted sum of its inputs as shown in Figure 1 below:

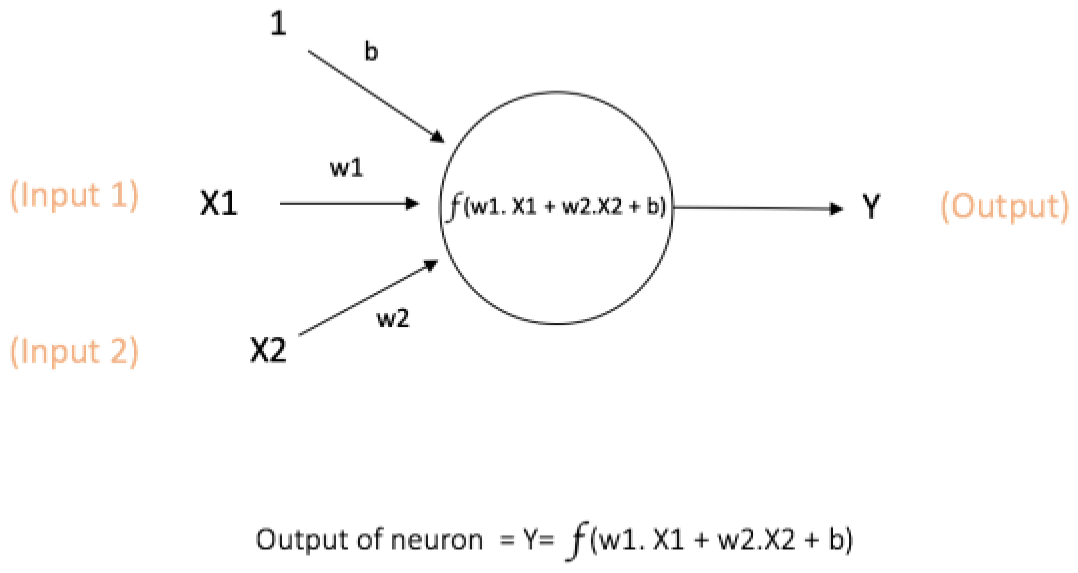


Figure 1: a single neuron

The above network takes numerical inputs **X1** and **X2** and has weights **w1** and **w2** associated with those inputs. Additionally, there is another input **1** with weight **b** (called the **Bias**) associated with it. We will learn more details about role of the bias later.

The output **Y** from the neuron is computed as shown in the Figure 1. The function ***f*** is non-linear and is called the **Activation Function**. The purpose of the activation function is to introduce non-linearity into the output of a neuron. This is important because most real world data is nonlinear and we want neurons to *learn* these nonlinear representations.

Step 1: Forward Propagation

All weights in the network are randomly assigned. Let's consider the hidden layer node marked **V** in Figure 5 below. Assume the weights of the connections from the inputs to that node are w1, w2 and w3 (as shown).

The network then takes the first training example as input (we know that for inputs 35 and 67, the probability of Pass is 1).

- Input to the network = [35, 67]
    - Desired output from the network (target) = [1, 0]

Then output V from the node in consideration can be calculated as below (***f*** is an activation function such as sigmoid):

$$V = \boldsymbol{f}\,(1{*}w1 + 35{*}w2 + 67{*}w3)$$

Similarly, outputs from the other node in the hidden layer is also calculated. The outputs of the two nodes in the hidden layer act as inputs to the two nodes in the output layer. This enables us to calculate output probabilities from the two nodes in output layer.

Suppose the output probabilities from the two nodes in the output layer are 0.4 and 0.6 respectively (since the weights are randomly assigned, outputs will also be random). We can see that the calculated probabilities (0.4 and 0.6) are very far from the desired probabilities (1 and 0 respectively), hence the network in Figure 5 is said to have an 'Incorrect Output'.

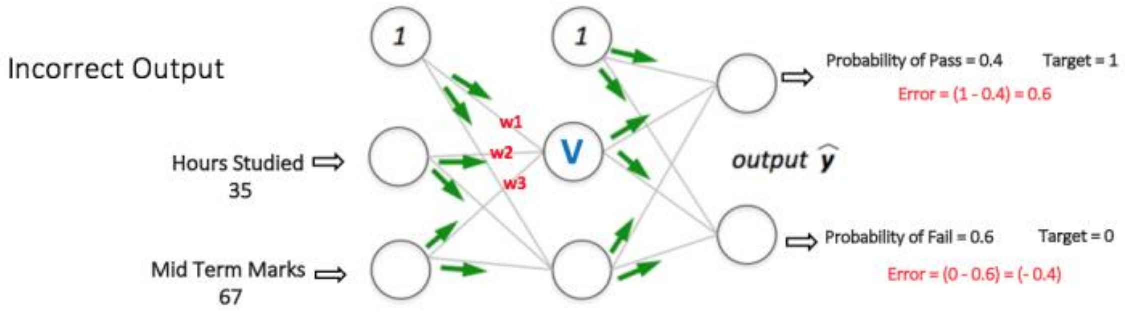
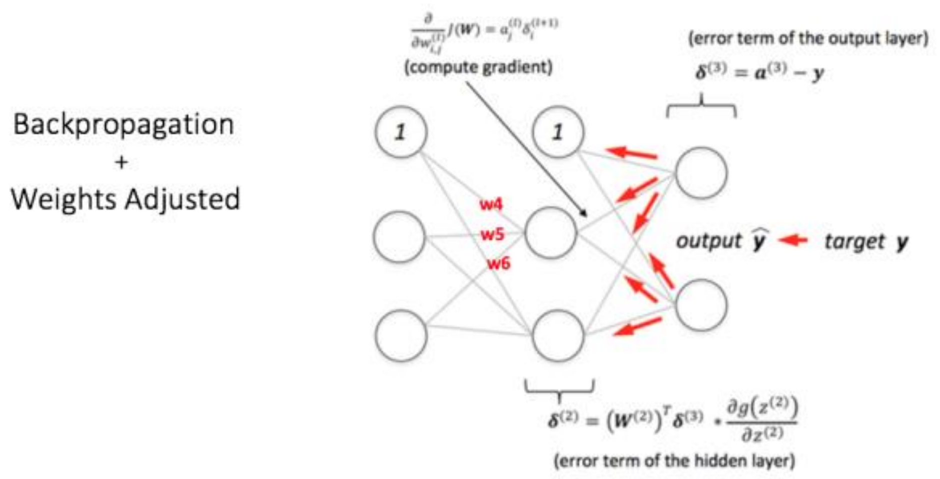


Figure 5: forward propagation step in a multi-layer perceptron

Step 2: Back Propagation and Weight Updating

We calculate the total error at the output nodes and propagate these errors back through the network using Backpropagation to calculate the *gradients*. Then we use an optimization method such as *Gradient Descent* to 'adjust' **all** weights in the network with an aim of reducing the error at the output layer. This is shown in the Figure 6 below (ignore the mathematical equations in the figure for now).

Suppose that the new weights associated with the node in consideration are w4, w5 and w6 (after Backpropagation and adjusting weights).



$\frac{\partial}{\partial w_{i,j}^{(l)}} J(W) = a_j^{(l)} \delta_i^{(l+1)}$
(compute gradient)
(error term of the output layer)
$\delta^{(3)} = a^{(3)} - y$
Backpropagation
+
Weights Adjusted
w4
w5
w6
1
1
output $\hat{y}$
target y
$\delta^{(2)} = (W^{(2)})^T \delta^{(3)} \cdot \frac{\partial g(z^{(2)})}{\partial z^{(2)}}$
(error term of the hidden layer)

Figure 6: backward propagation and weight updation step in a multi layer perceptron

If we now input the same example to the network again, the network should perform better than before since the weights have now been adjusted to minimize the error in prediction. As shown in Figure 7, the errors at the output nodes now reduce to [0.2, -0.2] as compared to [0.6, -0.4] earlier. This means that our network has learnt to correctly classify our first training example.

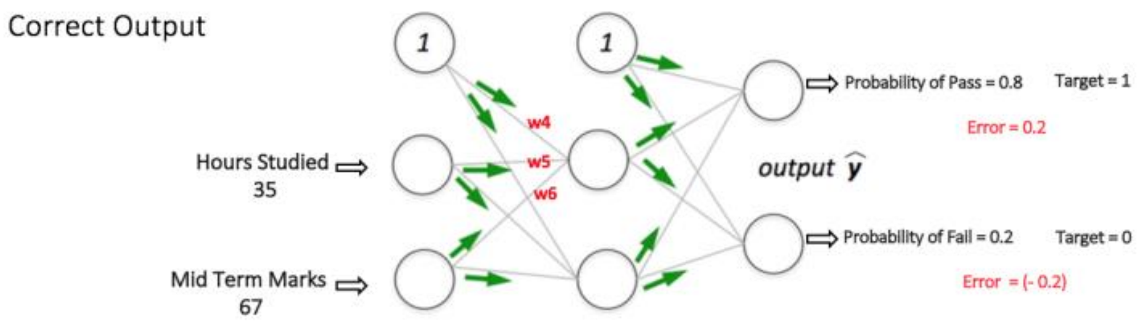


Figure 7: the MLP network now performs better on the same input

We repeat this process with all other training examples in our dataset. Then, our network is said to have *learnt* those examples.

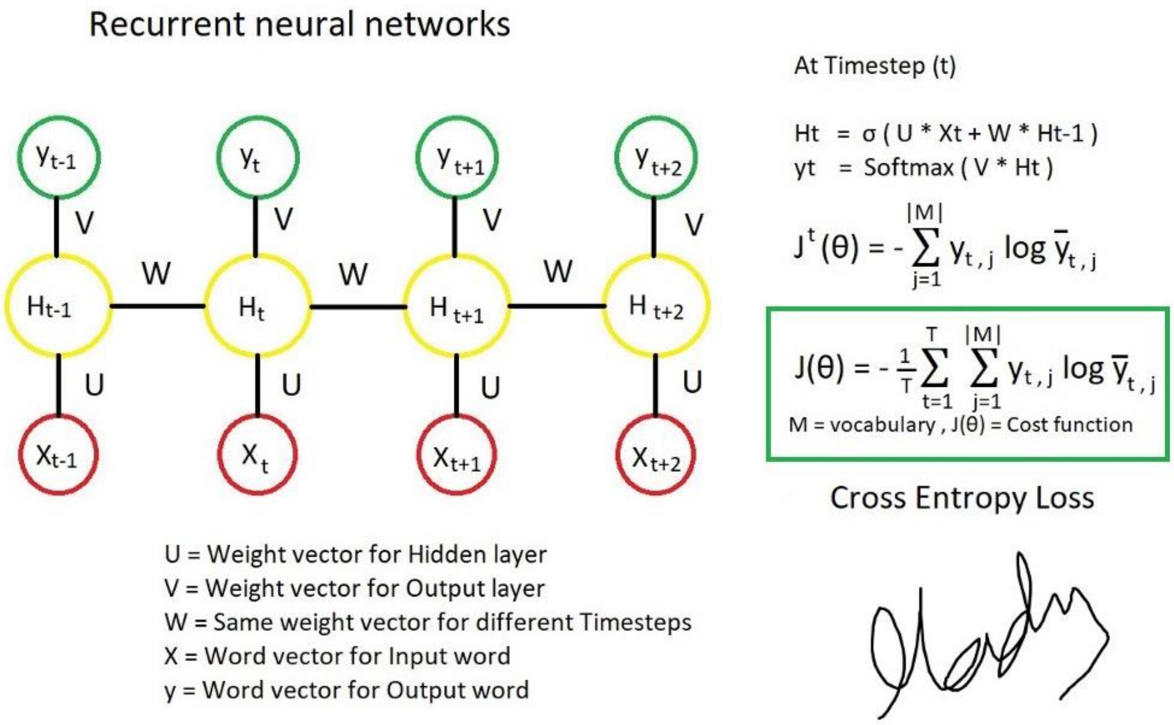
Why Recurrent Neural Network?

A neural network usually takes an independent variable X (or a set of independent variables) and a dependent variable y then it learns the mapping between X and y **(**we call this Training**),** Once training is done, we give a new independent variable to predict the dependent variable. But **what if the order of data matters?** Just imagine what if the order of all independent variables matters?

A Recurrent Neural Network has two inputs, the present and the recent past. This is important because the sequence of data contains crucial information about what is coming next, which is why a RNN can do things other algorithms can't.

Recurrent Neural Network:

A recurrent neural network (RNN) is a class of artificial neural network where connections between nodes form a directed graph along a sequence. This allows it to exhibit temporal dynamic behaviour for a time sequence. Unlike feedforward neural networks, RNNs can use their internal state (memory) to process sequences of inputs. This makes them applicable to tasks such as unsegmented, connected handwriting recognition or speech recognition.


Recurrent neural networks
y_{t-1} y_t y_{t+1} y_{t+2}
V
W
H_{t-1} H_t H_{t+1} H_{t+2}
U
X_{t-1} X_t X_{t+1} X_{t+2}
U = Weight vector for Hidden layer
V = Weight vector for Output layer
W = Same weight vector for different Timesteps
X = Word vector for Input word
y = Word vector for Output word
At Timestep (t)
Ht = σ (U * Xt + W * Ht-1)
yt = Softmax (V * Ht)
$J^t(\theta) = -\sum_{j=1}^{|M|} y_{t,j} \log \bar{y}_{t,j}$
$J(\theta) = -\frac{1}{T}\sum_{t=1}^{T}\sum_{j=1}^{|M|} y_{t,j} \log \bar{y}_{t,j}$
M = vocabulary , J(θ) = Cost function
Cross Entropy Loss

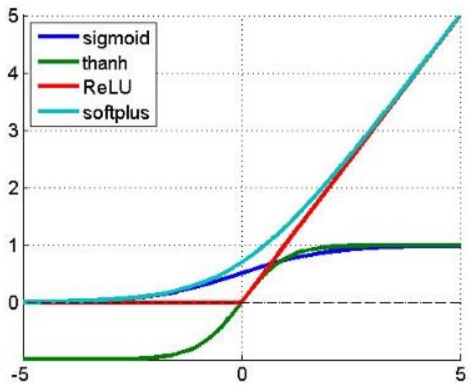
[illegible]

In the picture we are calculating the Hidden layer time step (t) values so Ht = Activatefunction(input * Hweights + W * Ht-1)

yt = softmax(Hweight* Ht)

Ht-1 is the previous time step and as i said W's are same for all time steps. The activation function can be Tanh, Relu, Sigmoid, etc.

	Propagation
Sigmoid	$y_s = \frac{1}{1+e^{-x_s}}$
Tanh	$y_s = \tanh(x_s)$
ReLu	$y_s = \max(0, x_s)$



Above we calculated only for Ht similarly we can calculate for all other time steps.

Steps:

1. Calculate **Ht-1** from **U** and **X**
2. Calculate **yt-1** from **Ht-1** and **V**
3. Calculate **Ht** from **U,X,W** and **Ht-1**
4. Calculate **yt** from **V** and **Ht** and so on…

Note:

1. **U** and **V** are weight vectors, different for every time step.

2. We can even calculate hidden layer (all time steps) first then calculate y values.

3. Weight vectors are random initially.

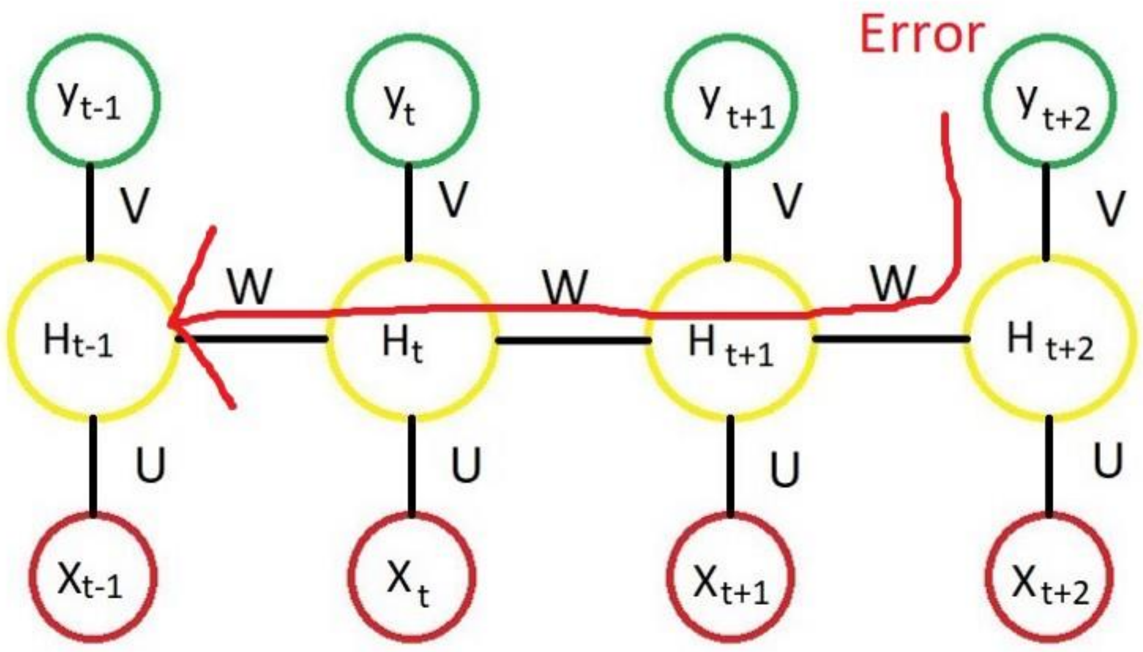
Once Feed forwarding is done then we need to calculate the error and back propagate the error using back propagation.

BPTT (BACK PROPAGATION THROUGH TIME)

We need to calculate the below terms

1. How much does the ***total error*** change with respect to the ***output (hidden and output units)?*** (or how much is a change in ***output***)
2. How much does the ***output*** change with respect to ***weights (U, V, W)***? (or how much is a change in ***weights***)

Since W's are same for all time steps we need to go all the way back to make an update.



Here Current time step is calculated based on the previous time step so we have to traverse all the way back.

TWO ISSUES OF STANDARD RNN'S

There are two major obstacles RNN's have or had to deal with. But to understand them, you first need to know what a gradient is.

A gradient is a partial derivative with respect to its inputs. If you don't know what that means, just think of it like this: A gradient measures how much the output of a function changes, if you change the inputs a little bit.

You can also think of a gradient as the slope of a function. The higher the gradient, the steeper the slope and the faster a model can learn. But if the slope is zero, the model stops to learning. A gradient simply measures the change in all weights with regard to the change in error.

Exploding Gradients

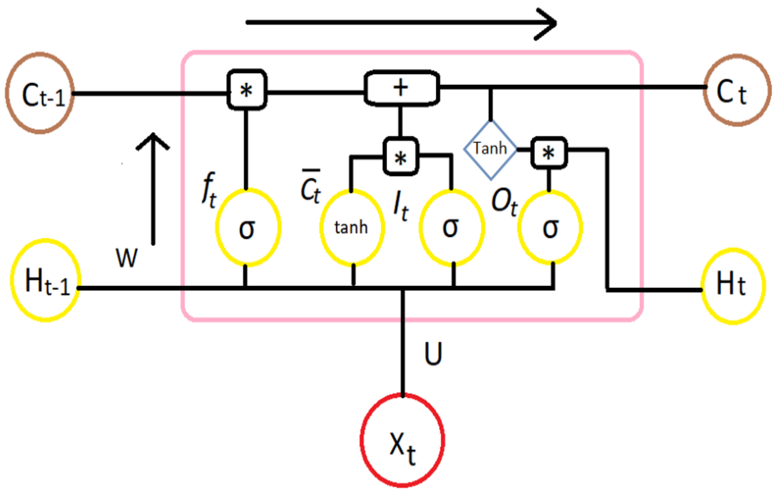
We speak of „Exploding Gradients "when the algorithm assigns a stupidly high importance to the weights, without much reason. But fortunately, this problem can be easily solved if you truncate or squash the gradients.

Vanishing Gradients

We speak of „Vanishing Gradients "when the values of a gradient are too small and the model stops learning or takes way too long because of that. This was a major problem in the 1990s and much harder to solve than the exploding gradients. Fortunately, it was solved through the concept of LSTM by Sepp Hoch Reiter and Juergen Schmidhuber, which we will discuss now.

LSTM

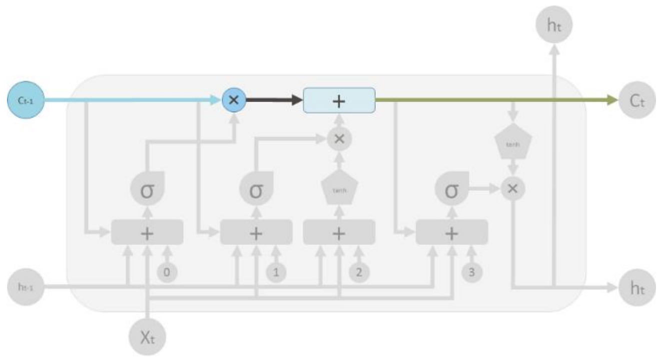
Then later, LSTM (Long Short Term Memory) to solve this issue by explicitly introducing a memory unit, called the cell into the network. This is the diagram of a LSTM building block.

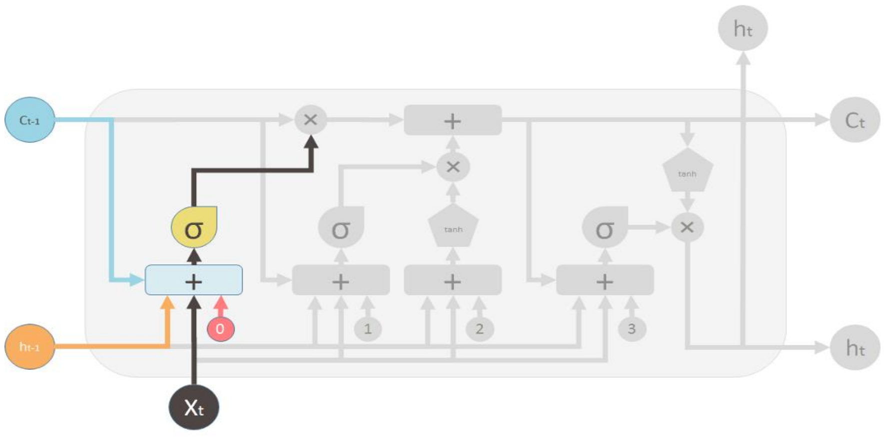


At a first sight, this looks intimidating. Let's ignore the internals, but only look at the inputs and outputs of the unit. The network takes three inputs. X_t is the input of the current time step. H(t-1) is the output from the previous LSTM unit and C_t-1 is the "memory" of the previous unit, which I think is the most important input. As for outputs, h_t is the output of the current network. C_t is the memory of the current unit.

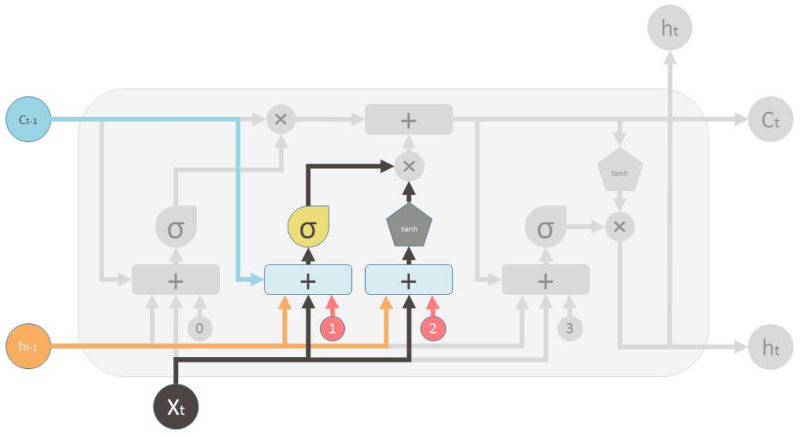
Therefore, this single unit makes decision by considering the current input, previous output and previous memory. And it generates a new output and alters its memory. The way its internal memory C_t changes is pretty similar to piping water through a pipe. Assuming the memory is water, it flows into a pipe. You want to change this memory flow along the way and this change is controlled by two valves. The first valve is called the forget valve. If you shut it, no old memory will be kept. If you fully open this valve, all old memory will pass through. The second valve is the new memory valve. New memory will come in through a T shaped joint like above and merge with the old memory. Exactly how much new memory should come in is controlled by the second valve.

On the LSTM diagram, the top "pipe" is the memory pipe. The input is the old memory (a vector). The first cross ✕ it passes through is the forget valve. It is actually an element-wise multiplication operation. So if you multiply the old memory C_t-1 with a vector that is close to 0, that means you want to forget most of the old memory. You let the old memory goes through, if your forget valve equals 1. Then the second operation the memory flow will go through is this + operator. This operator means piece-wise summation. It resembles the T shape joint pipe. New memory and the old memory will merge by this operation. How much new memory should be added to the old memory is controlled by another valve, the ✕ below the + sign. After these two operations, you have the old memory C_t-1 changed to the new memory C_t.



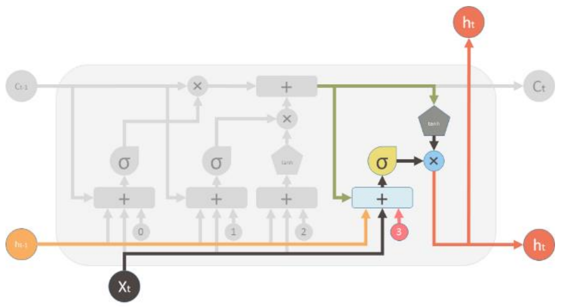


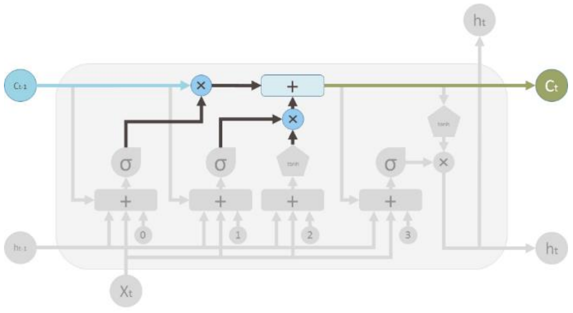
Now let’s look at the valves. The first one is called the forget valve. It is controlled by a simple one layer neural network. The inputs of the neural network is h_t-1, the output of the previous LSTM block, X_t, the input for the current LSTM block, C_t-1, the memory of the previous block and finally a bias vector b_0. This neural network has a sigmoid function as activation, and its output vector is the forget valve, which will applied to the old memory C_t-1 by element-wise multiplication.



Now the second valve is called the new memory valve. Again, it is a one layer simple neural network that takes the same inputs as the forget valve. This valve controls how much the new memory should influence the old memory.

The new memory itself, however is generated by another neural network. It is also a one layer network, but uses tanh as the activation function. The output of this network will element-wise multiple the new memory valve, and add to the old memory to form the new memory.





These two ✕ signs are the forget valve and the new memory valve. And finally, we need to generate the output for this LSTM unit. This step has an output valve that is controlled by the new memory, the previous output h_t-1, the input X_t and a bias vector. This valve controls how much new memory should output to the next LSTM unit.

The above diagram is inspired by Christopher's blog post. But most of the time, you will see a diagram like below. The major difference between the two variations is that the following diagram doesn't treat the memory unit C as an input to the unit. Instead, it treats it as an internal thing "Cell".

I like the Christopher's diagram, in that it explicitly shows how this memory C gets passed from the previous unit to the next. But in the following image, you can't easily see that C_t-1 is actually from the previous unit. And C_t is part of the output.

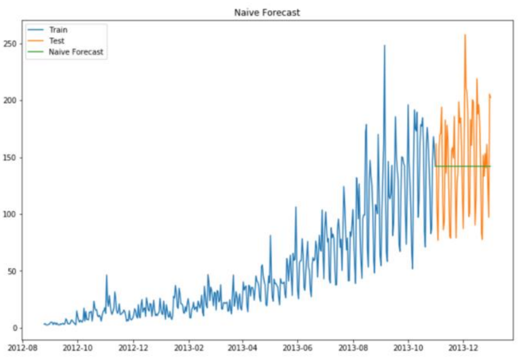
Second reason I don't like the following diagram is that the computation you perform within the unit should be ordered, but you can't see it clearly from the following diagram. For example to calculate the output of this unit, you need to have C_t, the new memory ready. Therefore, the first step should be evaluating C_t.

The following diagram tries to represent this "delay" or "order" with dash lines and solid lines (there are errors in this picture). Dash lines means the old memory, which is available at the beginning. Some solid lines means the new memory. Operations require the new memory have to wait until C_t is available.

Comparison between different models for improving RMSE:

Method 1: Start with a Naive Approach: 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 its 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.

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

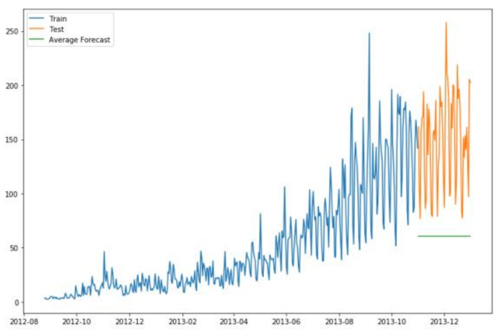


Method 2: – Simple Average

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.

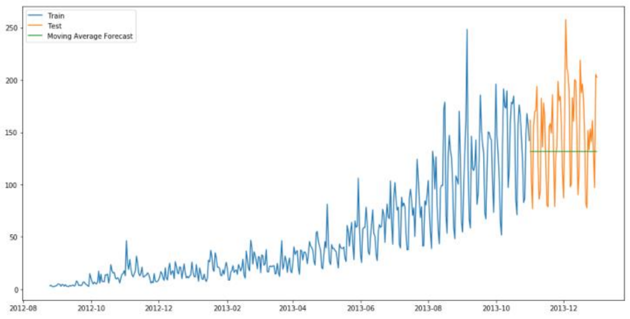
$$\text{Hence } \hat{y}_{x+1} = \frac{1}{x}\sum_{i=1}^{x} y_i$$



Method 3 – Moving Average:

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.

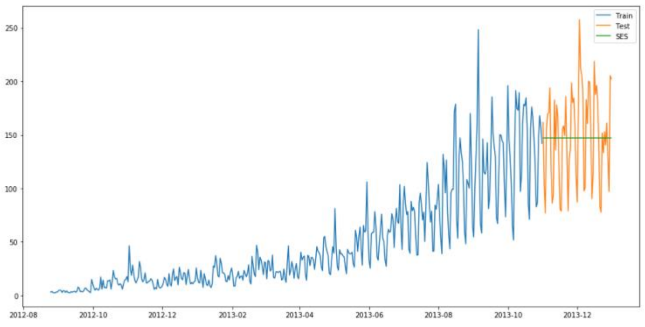
$$\hat{y}_i = \frac{1}{p}(y_{i-1} + y_{i-2} + y_{i-3} \ldots\ldots + y_{i-p})$$



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:

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



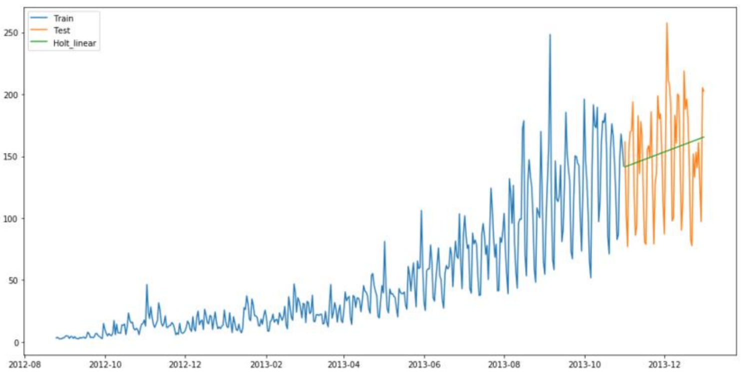
Method 5 – Holt’s Linear Trend method:

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 its components which are Trend, Seasonality and Residual. Any dataset that follows a trend can use Holt’s linear trend method for forecasting.

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

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

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

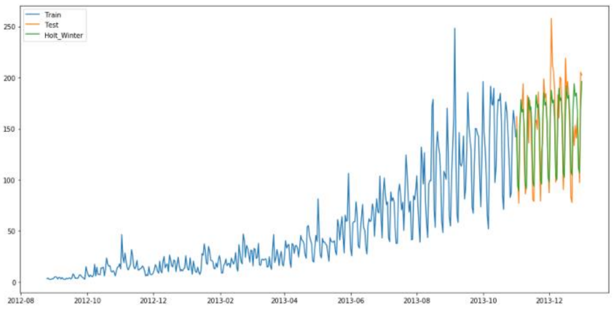


Method 6 – Holt-Winters Method:

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.

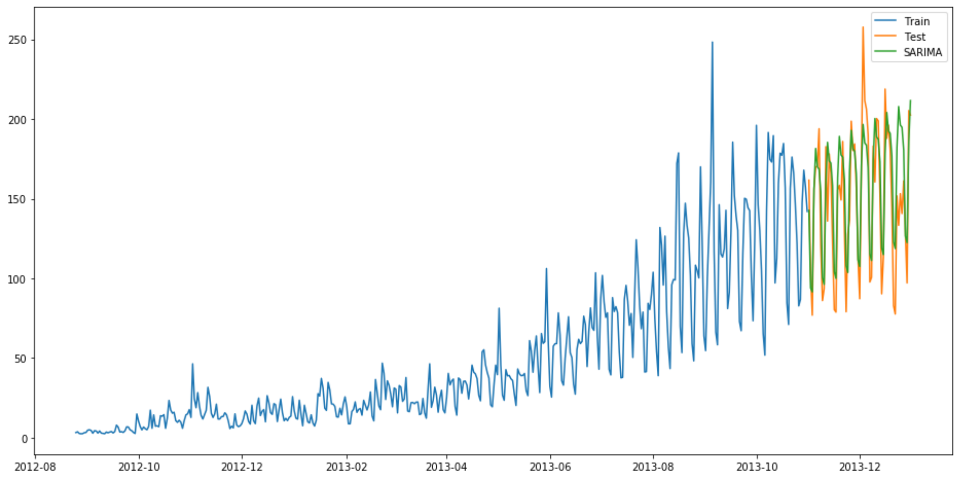
Using Holt’s winter method will be the best option among the rest of the models because of the seasonality factor. The Holt-Winters seasonal method comprises the forecast equation and three smoothing equations — one for the level ℓ_t, one for trend b_t and one for the seasonal component denoted by s_t,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

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



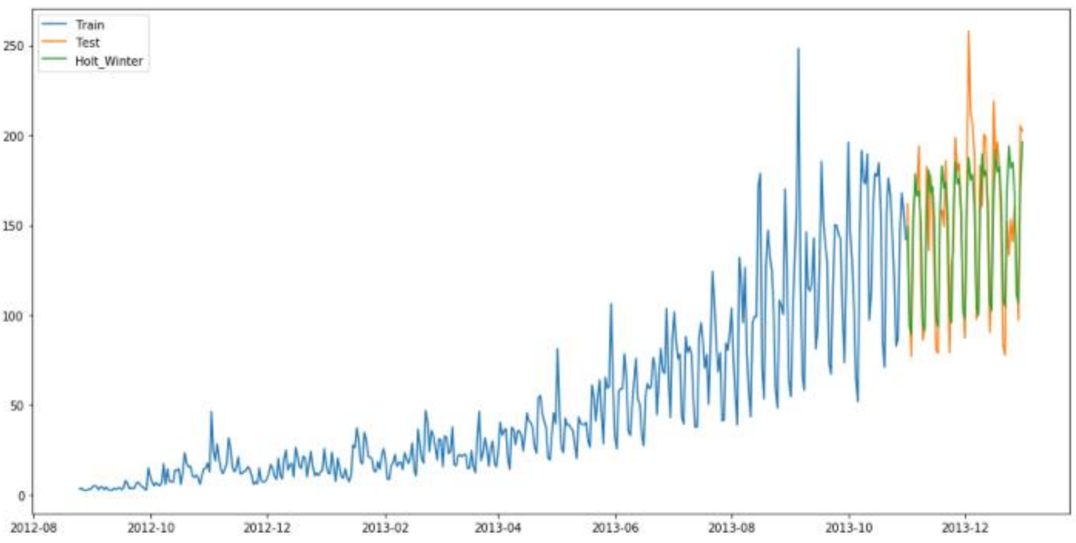
Method 7 – ARIMA:

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.



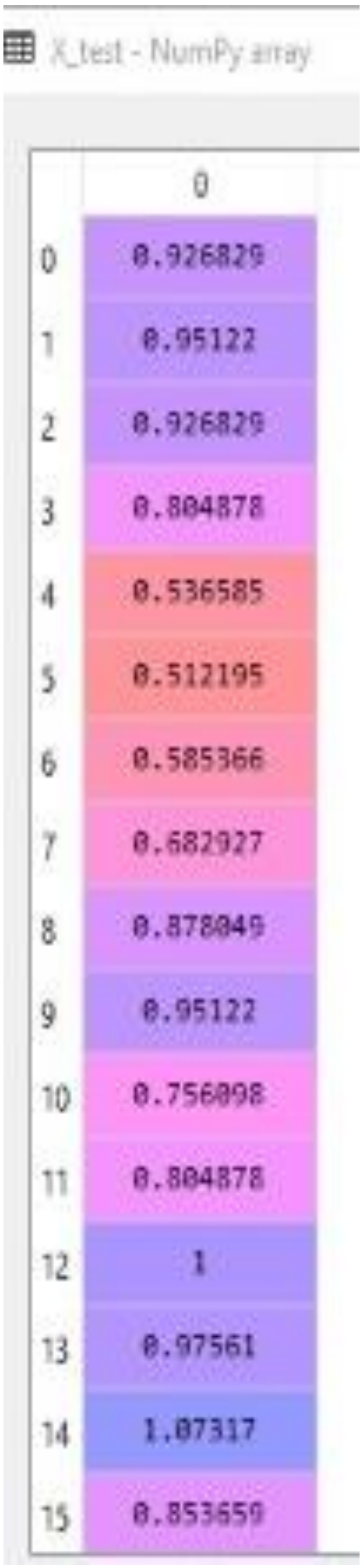
We can compare these models on the basis of their RMSE scores:

<u>Method 8– RNN:</u>

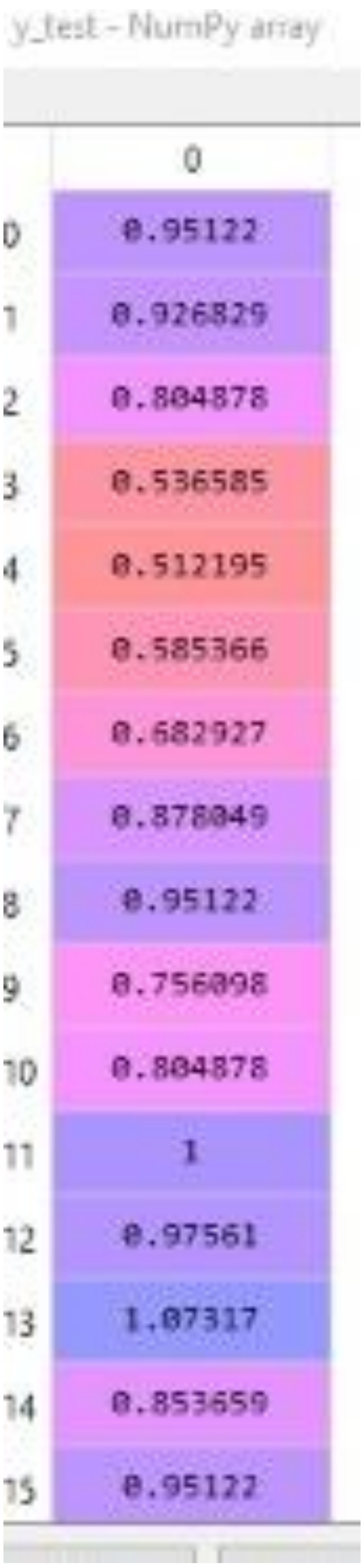


Model	**RMSE**
Naïve Method	43.9
Simple Average	109.5
Moving Average	46.72
Simple Exponential Smoothing	43.35
Holt's Linear Trend	43.05
Holt's Winter	23.96
ARIMA	26.06
RNN	**25.02**

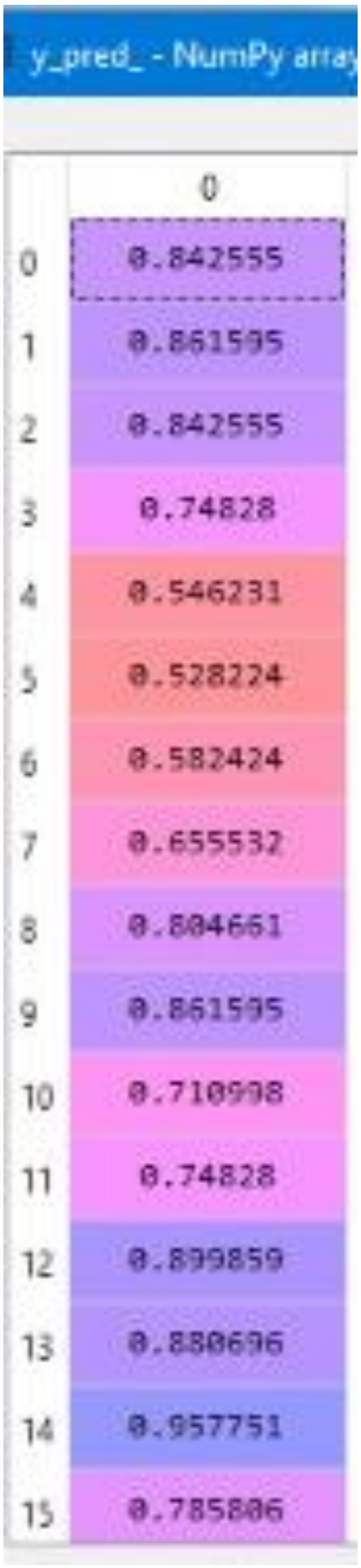
Predicted Values:


X_test - NumPy array

	0
0	0.926829
1	0.95122
2	0.926829
3	0.804878
4	0.536585
5	0.512195
6	0.585366
7	0.682927
8	0.878049
9	0.95122
10	0.756098
11	0.804878
12	1
13	0.97561
14	1.07317
15	0.853659

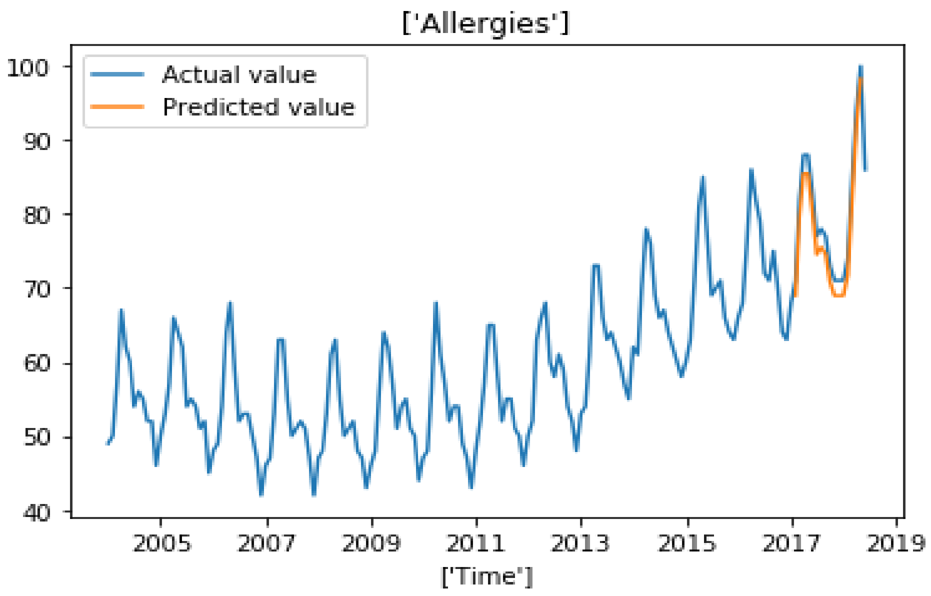

y_test - NumPy array

	0
0	0.95122
1	0.926829
2	0.804878
3	0.536585
4	0.512195
5	0.585366
6	0.682927
7	0.878049
8	0.95122
9	0.756098
10	0.804878
11	1
12	0.97561
13	1.07317
14	0.853659
15	0.95122

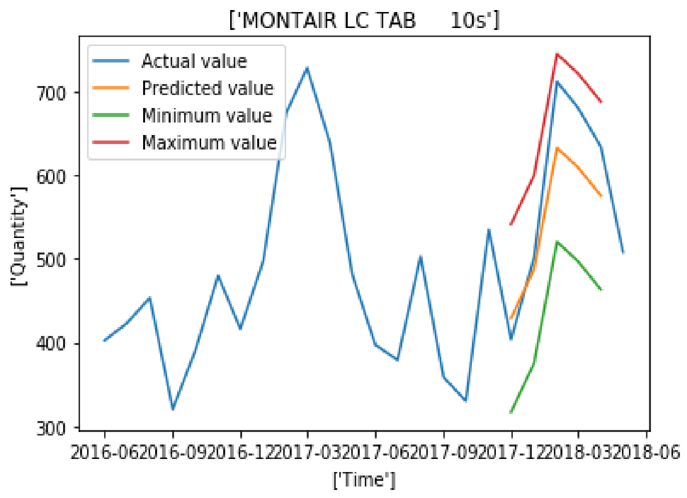

y_pred_ - NumPy array

	0
0	0.842555
1	0.861595
2	0.842555
3	0.74828
4	0.546231
5	0.528224
6	0.582424
7	0.655532
8	0.804661
9	0.861595
10	0.710998
11	0.74828
12	0.899859
13	0.880696
14	0.957751
15	0.785806

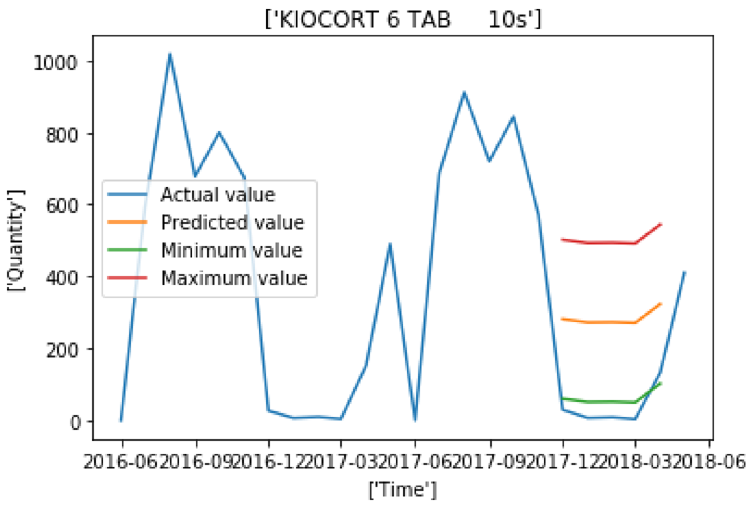
RESULT:



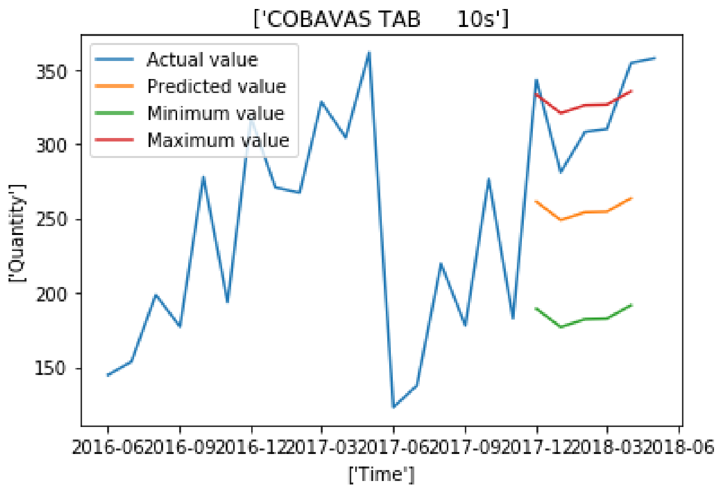
['Allergies']
Actual value
Predicted value
100
90
80
70
60
50
40
2005
2007
2009
2011
2013
2015
2017
2019
['Time']



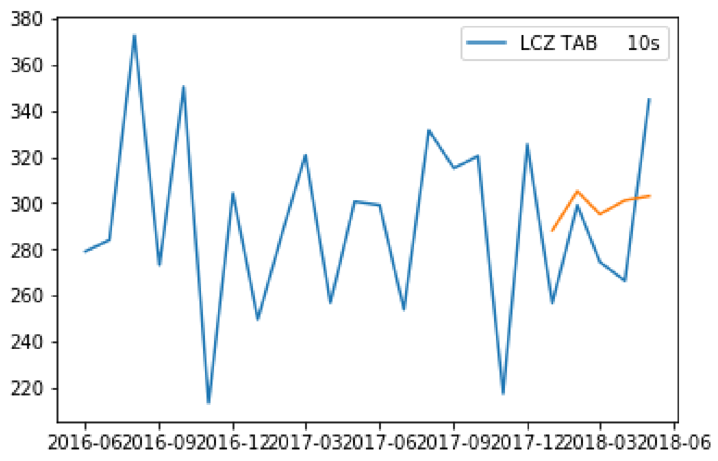
['MONTAIR LC TAB 10s']
Actual value
Predicted value
Minimum value
Maximum value
['Quantity']
['Time']



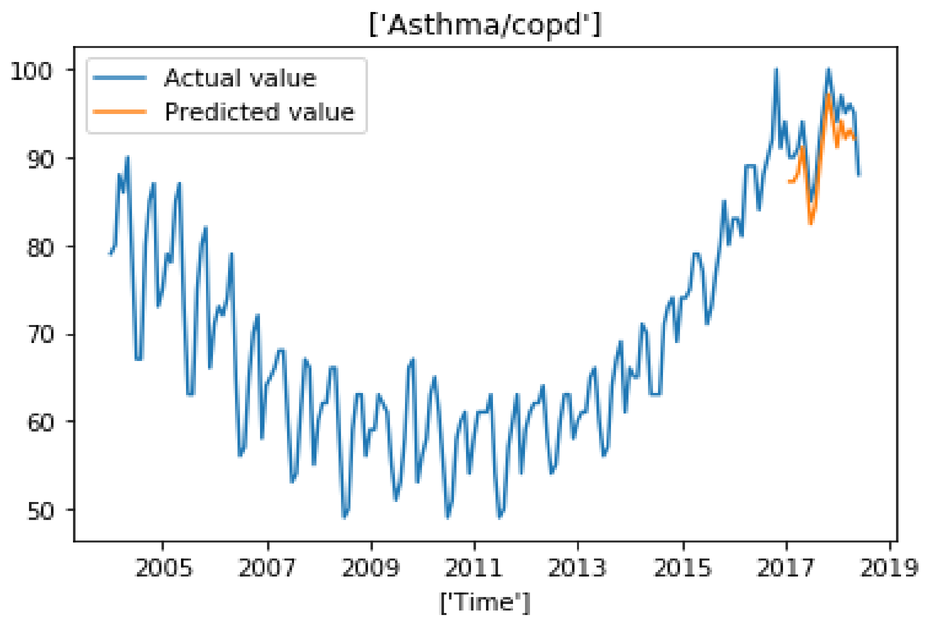
['KIOCORT 6 TAB 10s']
Actual value
Predicted value
Minimum value
Maximum value
['Quantity']
['Time']

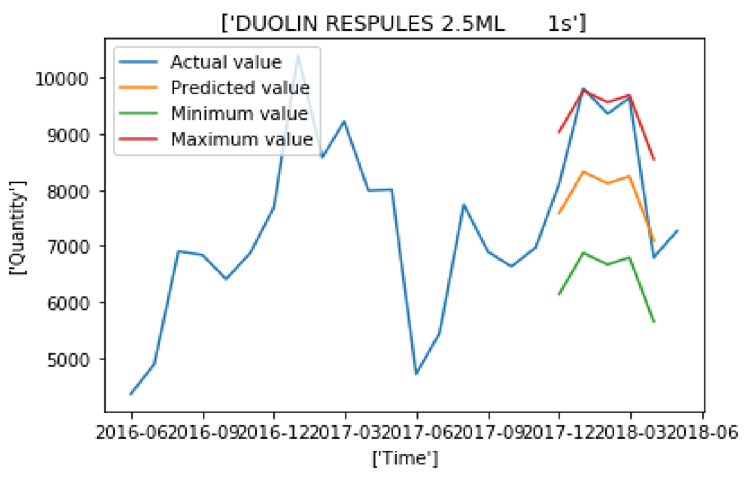


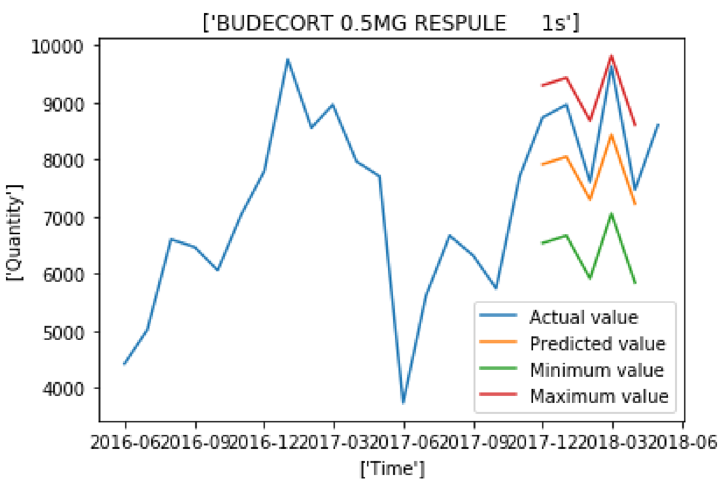
['COBAVAS TAB 10s']
Actual value
Predicted value
Minimum value
Maximum value
['Quantity']
['Time']

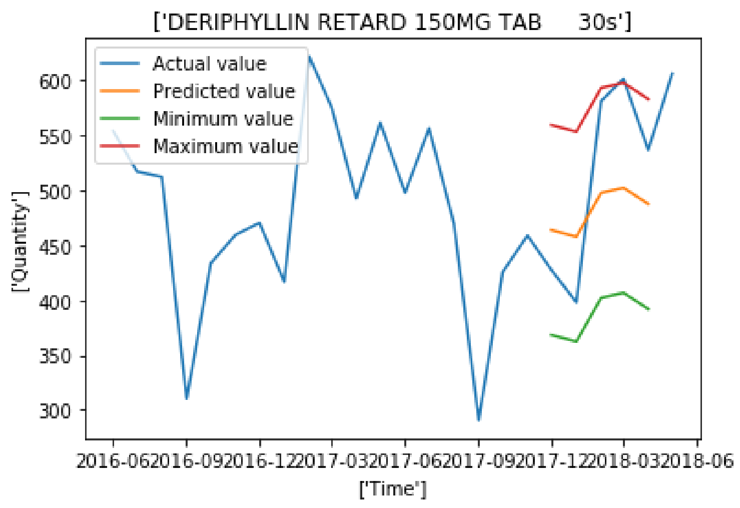


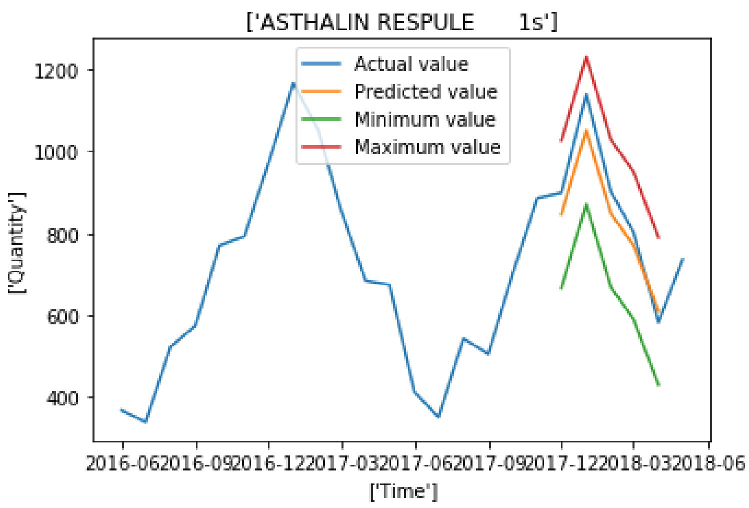
LCZ TAB 10s

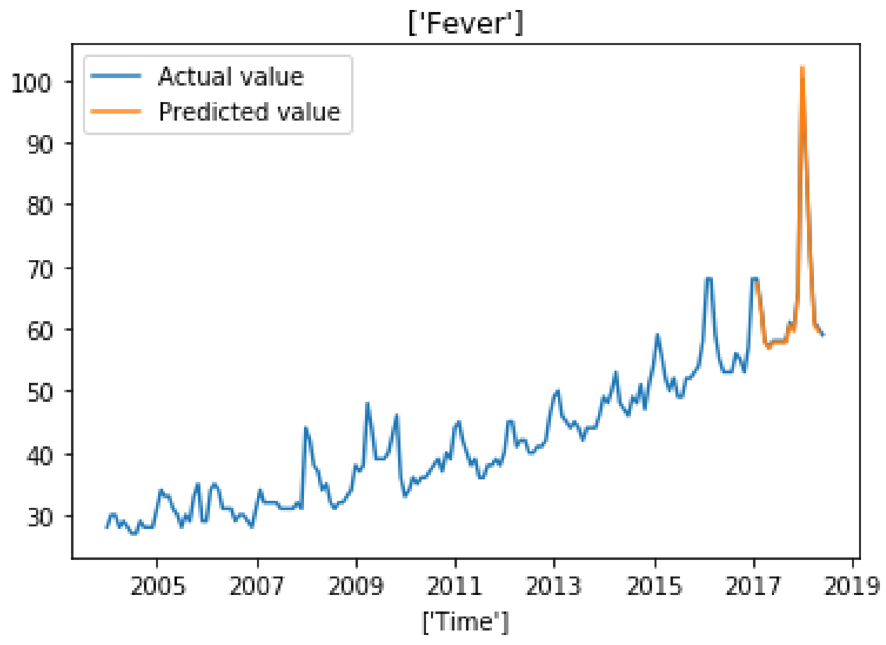

['Asthma/copd']
Actual value
Predicted value
100
90
80
70
60
50
2005
2007
2009
2011
2013
2015
2017
2019
['Time']

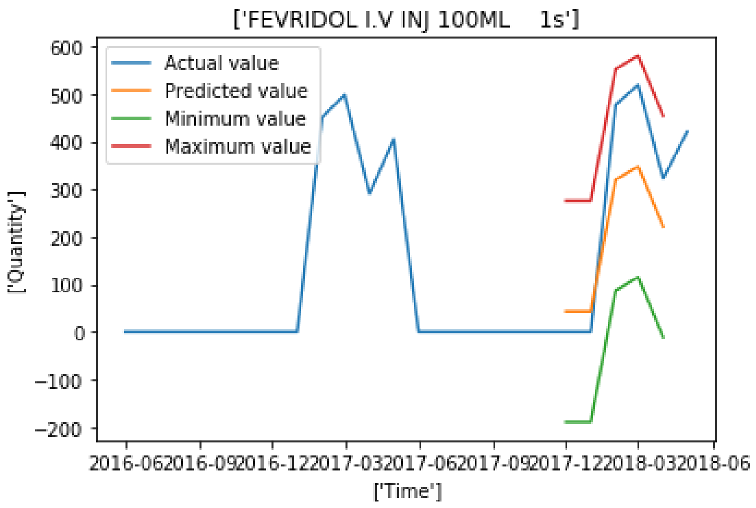

['DUOLIN RESPULES 2.5ML   1s']
Actual value
Predicted value
Minimum value
Maximum value
['Quantity']
10000
9000
8000
7000
6000
5000
2016-06
2016-09
2016-12
2017-03
2017-06
2017-09
2017-12
2018-03
2018-06
['Time']

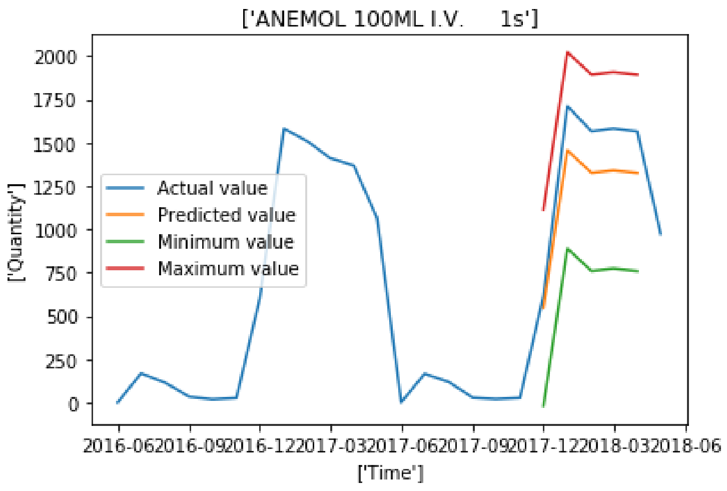

['BUDECORT 0.5MG RESPULE   1s']
Actual value
Predicted value
Minimum value
Maximum value
['Quantity']
10000
9000
8000
7000
6000
5000
4000
2016-06
2016-09
2016-12
2017-03
2017-06
2017-09
2017-12
2018-03
2018-06
['Time']

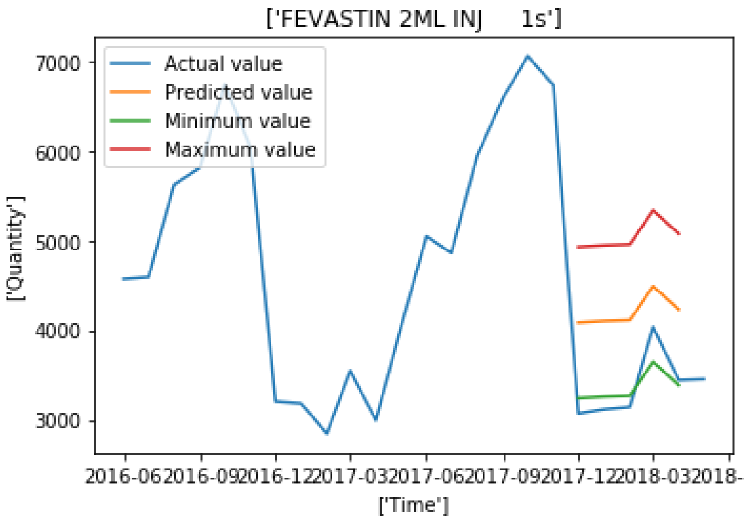

['DERIPHYLLIN RETARD 150MG TAB   30s']
Actual value
Predicted value
Minimum value
Maximum value
['Quantity']
600
550
500
450
400
350
300
2016-06
2016-09
2016-12
2017-03
2017-06
2017-09
2017-12
2018-03
2018-06
['Time']

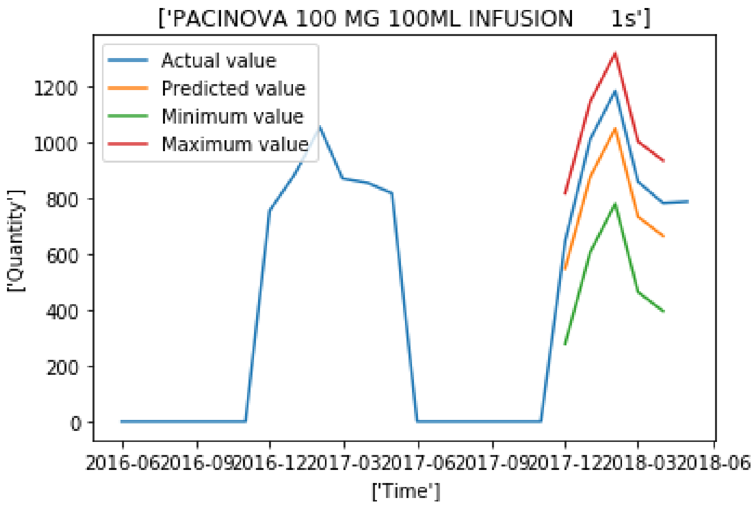

['ASTHALIN RESPULE   1s']
Actual value
Predicted value
Minimum value
Maximum value
['Quantity']
1200
1000
800
600
400
2016-06
2016-09
2016-12
2017-03
2017-06
2017-09
2017-12
2018-03
2018-06
['Time']

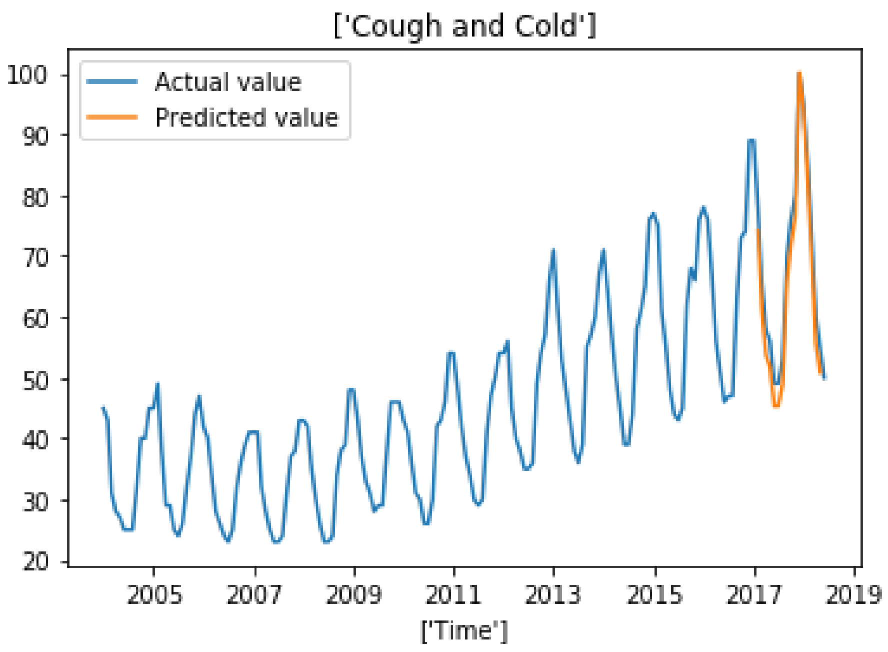

['Fever']
Actual value
Predicted value
['Time']

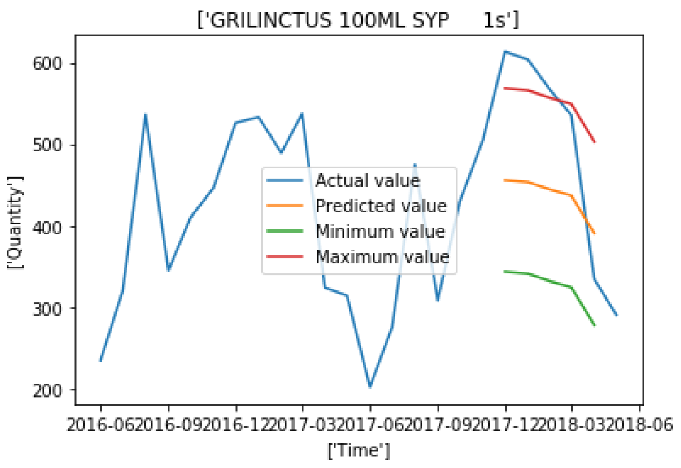

['FEVRIDOL I.V INJ 100ML 1s']
Actual value
Predicted value
Minimum value
Maximum value
['Quantity']
['Time']

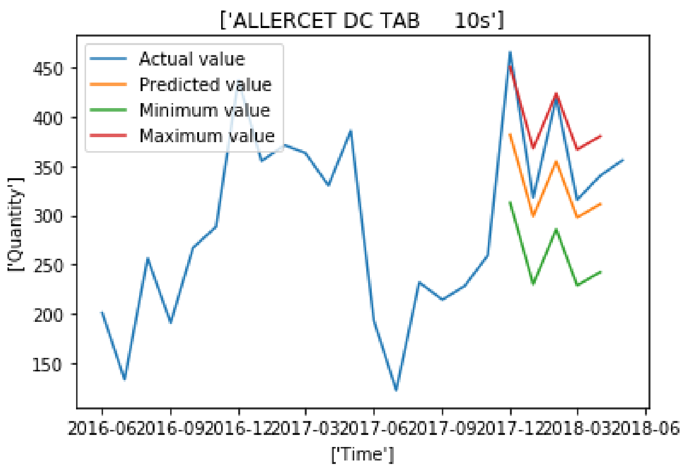

['ANEMOL 100ML I.V. 1s']
Actual value
Predicted value
Minimum value
Maximum value
['Quantity']
['Time']

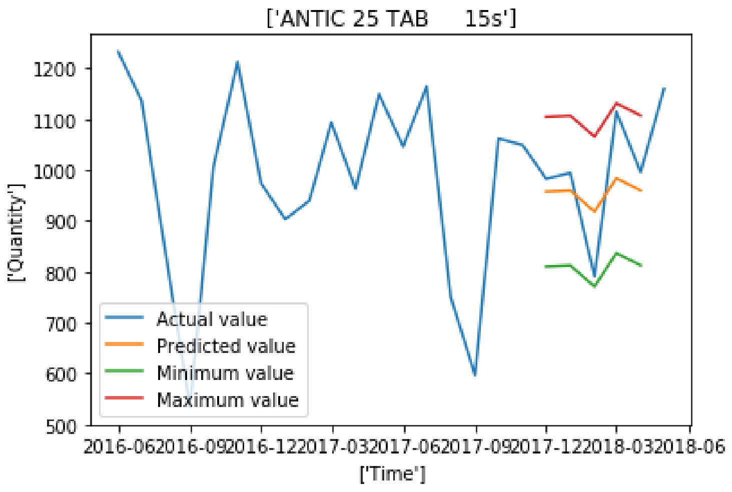

['FEVASTIN 2ML INJ 1s']
Actual value
Predicted value
Minimum value
Maximum value
['Quantity']
['Time']


['PACINOVA 100 MG 100ML INFUSION 1s']
Actual value
Predicted value
Minimum value
Maximum value
['Quantity']
['Time']


['Cough and Cold']
Actual value
Predicted value
100
90
80
70
60
50
40
30
20
2005
2007
2009
2011
2013
2015
2017
2019
['Time']


['GRILINCTUS 100ML SYP 1s']
Actual value
Predicted value
Minimum value
Maximum value
600
500
400
300
200
['Quantity']
2016-06
2016-09
2016-12
2017-03
2017-06
2017-09
2017-12
2018-03
2018-06
['Time']


['ALLERCET DC TAB 10s']
Actual value
Predicted value
Minimum value
Maximum value
450
400
350
300
250
200
150
['Quantity']
2016-06
2016-09
2016-12
2017-03
2017-06
2017-09
2017-12
2018-03
2018-06
['Time']


['ANTIC 25 TAB 15s']
Actual value
Predicted value
Minimum value
Maximum value
1200
1100
1000
900
800
700
600
500
['Quantity']
2016-06
2016-09
2016-12
2017-03
2017-06
2017-09
2017-12
2018-03
2018-06
['Time']

Conclusion:

By using RNN, we have devised a model for predicting the amount of medicines required by hospitals with their diversity of diseases spreading across a particular region with months which **is better than the approaches previously used by others for this purpose in terms of accuracy and error.**

Future Work:

After having large scale data of almost 10-15 years and access to medical data of different regions across the country, we can improve the accuracy.

If we analyse different diseases and their affecting viruses we can classify them accordingly and prevention would be far more précised.

Also if we know the viruses of any new epidemic or disease we can predict their medicines for preliminary cure.

References:

1. https://towardsdatascience.com/using-lstms-to-forecast-time-series-4ab688386b1f
2. https://www.analyticsvidhya.com/blog/2018/02/time-series-forecasting-methods/
3. https://medium.com/deep-math-machine-learning-ai/chapter-10-1-deepnlp-lstm-long-short-term-memory-networks-with-math-21477f8e4235
4. https://www.coursera.org/lecture/nlp-sequence-models/recurrent-neural-network-model-ftkzt
5. https://machinelearningmastery.com/time-series-prediction-lstm-recurrent-neural-networks-python-keras/
6. https://www.superdatascience.com/category/machine-learning/
7. https://www.kickstarter.com/projects/kirilleremenko/deep-learning-a-ztm-online-course

CODE:

```
#__________________________________________________________________________________
#importing libraries
import pandas as pd
import numpy as np
import math
import datetime
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
#__________________________________________________________________________________
#importing data
data = pd.read_csv('diseases.csv')
split_date = datetime.datetime(2017, 11, 1)
data['Month']=pd.to_datetime(data['Month'])
pred_data = data['Month']
pred_data = pd.DataFrame(pred_data)
pred_data = pred_data[pred_data.Month > split_date]
pred_data= pred_data.iloc[1:,:].reset_index()
pred_data.drop(['index'], axis = 1, inplace = True)
data = data.set_index('Month')
m,n  = data.shape


for i in range(n):
#__________________________________________________________________________________
#splitting data
    df = data.iloc[:,i:i+1]
    split_date = datetime.datetime(2017, 11, 1)
    train = df[df.index < split_date]
    test = df[df.index > split_date]
#__________________________________________________________________________________
#Scaling data
    sc = MinMaxScaler()
    train_sc = sc.fit_transform(train)
    test_sc = sc.transform(test)

    time_step = 1
    X_train = train_sc[:-time_step]
    y_train = train_sc[time_step:]
    test_sc = sc.transform(test)

    time_step = 1
    X_train = train_sc[:-time_step]
    y_train = train_sc[time_step:]

    X_test = test_sc[:-time_step]
    y_test = test_sc[time_step:]

    m,n = X_train.shape
    X_train = np.reshape(X_train,(m,n,1))
    m,n = X_test.shape
    X_test = np.reshape(X_test,(m,n,1))
#_____________________________________________________________________________
#Training the model
    model = Sequential()
    model.add(LSTM(12,input_shape=(None,1),activation='relu'))
    model.add(Dense(6))
    model.add(Dense(1))
    model.compile(loss='mean_squared_error',optimizer='adam')
    model.summary()
    model.fit(X_train,y_train,epochs=80,batch_size=1,verbose=1)

    y_pred = model.predict(X_test)
    y_pred = sc.inverse_transform(y_pred)
    y = pd.DataFrame(sc.inverse_transform(y_test))

    y_pred = pd.DataFrame(y_pred)
    y_pred.columns = train.columns
    pred_data = pred_data.join(y_pred)
    y_pred = y_pred.iloc[time_step:,:]
    y_pred = y_pred.reset_index()
    y_pred.drop(['index'], axis = 1, inplace = True)
    plt.plot(y)
    plt.plot(y_pred)
#_____________________________________________________________________________
# Calulating error
    rmse = math.sqrt(mean_squared_error(y,y_pred))

#_____________________________________________________________________________
#exporting data
pred_data = pred_data.set_index('Month')
pred_data.to_excel('predicted_diseases.xlsx')
```