

**REAL-TIME Anomaly detection using Machine Learning**

*An investigative study*



PIYUSH NANDI, SAYAN BACHHAR, goutam bose

*Final Year Project Report 2015-2019*

*Institute of engineering & management, kolkata*

**Real-Time Anomaly Detection Using Machine Learning – An Investigative Study**

A project report submitted for the partial fulfillment of the

Bachelor of Technology Degree in

**Computer Science & Engineering under**

**Maulana Abul Kalam Azad University of Technology by**

Sayan Bachhar

Roll No: 10400115150, Registration Number: 151040110150

Piyush Nandi

Roll No: 10400115111, Registration Number: 151040110111

Goutam Bose

Roll No: 10400115072, Registration Number: 151040110072

Academic Session: 2015-2019

Under the Supervision of

Prof. Amit Kumar Das



**Department of Computer Science and Engineering Institute of Engineering & Management**

Y-12, Salt Lake, Sector 5, Kolkata, Pin 700091, West Bengal, India Affiliated To



### Maulana Abul Kalam Azad University of Technology

BF 142, BF Block, Sector 1, Kolkata, West Bengal 700064

### May 2019



**CERTIFICATE**

**TO WHOM IT MAY CONCERN**

This is to certify that the project report titled **“Real-Time Anomaly Detection Using Machine Learning – An Investigative Study”**, submitted by **Sayan Bachhar, Roll No: 10400115150, Registration Number: 151040110150**, **Piyush Nandi, Roll No: 10400115111, Registration Number: 151040110111**, **Goutam Bose, Roll No: 10400115072, Registration Number: 151040110072** students of **Institute of Engineering & Management** in partial fulfillment of requirements for the award of the degree of **Bachelor of Technology in Computer Science & Engineering**, is a bona fide work carried out under the supervision of **Prof. Amit Kumar Das** during the final year of the academic session of 2015-2019. The content of this report has not been submitted to any other university or institute for the award of any other degree.

It is further certified that the work is entirely original and the performance has been found to be satisfactory.

**Prof. Amit Kumar Das**

Assistant Professor

Department of Computer Science and Engineering Institute of Engineering & Management

**Prof.(Dr.) Himadri Nath Saha**

H.O.D.

Department of Computer Science and Engineering Institute of Engineering & Management

**Prof.(Dr.) Amlan Kusum Nayak**

Principal

Institute of Engineering & Management



**INSTITUTE OF ENGINEERING & MANAGEMENT**



DECLARATION

FOR NON-COMMITMENT OF PLAGIARISM

We,  **Sayan Bachhar, Piyush Nandi** and **Goutam Bose**, students of B.Tech. in the Department of Computer Science and Engineering, Institute of Engineering & Management have submitted the project report in partial fulfillment of the requirements to obtain the above noted degree. We declare that we have not committed plagiarism in any form or violated copyright while writing the report and have acknowledged the sources and/or the credit of other authors wherever applicable. If subsequently it is found that we have committed plagiarism or violated copyright, then the authority has full right to cancel/reject/revoke our degree.

Name of the Student: SAYAN BACHHAR

Full Signature:

Name of the Student: PIYUSH NANDI

Full Signature:

Name of the Student: GOUTAM BOSE

Full Signature:

Date:

***Acknowledgements***

*This project would have been impossible to accomplish without the astute mentorship of Professor Amit Kumar Das who has proven to be a pillar of support in every aspect of our work. We have always been encouraged by him to participate in his personal projects which gave us countless vital insights into our field of study.*

*We would also like to express our special thanks of gratitude to our respected principal, Head of the department of Computer Science and Engineering and our esteemed faculty who gave us the golden opportunity to do this wonderful project on the topic Real-Time Anomaly Detection Using Machine Learning, and also helped us in doing a lot of research with constant guidance and feedback that facilitated us to complete the project within the stipulated time frame.*

**Contents**

|  |  |
| --- | --- |
| *Article* | *Page* |
| <1> Abstract | **1** |
| <1.1> Introduction | 1 |
| <1.2> Objectives | 1 |
| <1.2.1> Developing a customised time-series dataset | 1 |
| <1.2.2> Investigating potential forecasting models | 1 |
| <1.3> Scope of the project | 2 |
| <1.4> Expectations from the best solution | 2 |
| <2> Supervised vs unsupervised learning | **3** |
| <2.1> The hypothesis | 3 |
| <2.2> The big picture | 3 |
| <2.3> Inclining towards unsupervised learning | 4 |
| <2.4> In perspective of the investigation | 4 |
| <3> Continuous learning & real-time analysis | **5** |
| <3.1> Learning like humans do | 5 |
| <3.2> Introducing continuous learning | 5 |
| <3.3> Emerging need for real time analysis | 6 |
| <4> A literature survey | **7** |
| <4.1> Scope of alternative solutions | 7 |
| <4.1.1> SARIMA | 7 |
| <4.1.2> GArCH | 7 |
| <4.1.3> Isolation forest | 8 |
| <4.2> Preference of ARIMA | 8 |
| <4.3> Preference of LSTM | 9 |
| <4.4> Preference of HTM | 9 |
| <5> Experimental setup | **10** |
| <5.1> System specifications | 10 |
| <5.2> IDE specifications | 10 |
| <6> Developing the dataset | **11** |
| <6.1> About time-series | 11 |
| <6.2> Choosing univariate time-series | 11 |
| <6.3> Developing a time-series dataset | 11 |
| <6.3.1> Generating a sine trend | 12 |
| <6.3.2> Generating an exponential trend | 13 |
| <6.3.3> Additive superimposition of both trends | 13 |
| <6.3.4> Introducing seasonal variations | 14 |
| <6.3.5> Injecting noise | 15 |
| <6.3.6> Inserting anomalies | 16 |
| <7> Investigating ARIMA | **17** |
| <7.1> The models involved | 17 |
| <7.1.1> Autoregression | 17 |
| <7.1.2> Moving average | 18 |
| <7.2> Auto Regressive Integrated Moving Average | 18 |
| <7.3> Implementation | 19 |
| <7.3.1> Checking stationarity of the dataset | 19 |
| <7.3.2> Determining ARIMA parameters | 20 |
| <7.3.3> Preparing and executing the model | 22 |
| <7.3.4> Classifying anomalies | 23 |
| <7.3.5> Graphical analysis | 23 |
| <8> Investigating LSTM | **24** |
| <8.1> Recurrent Neural Networks | 24 |
| <8.2> Introducing LSTM | 24 |
| <8.3> Architecture of LSTM | 25 |
| <8.3.1> Forget gate | 26 |
| <8.3.2> Input gate | 26 |
| <8.3.3> Output gate | 26 |
| <8.4> Implementation | 27 |
| <8.4.1> Splitting the dataset | 27 |
| <8.4.2> Building the model | 28 |
| <8.4.3> Fitting the model | 28 |
| <8.4.4> Executing the model | 29 |
| <8.4.5> Graphical analysis | 29 |
| <9> Investigating HTM | **31** |
| <9.1> A brief introduction | 31 |
| <9.2> Methodology | 31 |
| <9.2.1> Sparse Distributed Representations | 31 |
| <9.2.1.1> Spatial pooling | 33 |
| <9.2.1.2> Active, inactive and predictive cells | 33 |
| <9.2.1.3> How do cells become active? | 33 |
| <9.2.1.4> How do cells become predictive? | 33 |
| <9.2.1.5> The output of a minicolumn | 34 |
| <9.2.2> Inference and online learning | 34 |
| <9.2.3> HTM for anomaly detection | 34 |
| <9.3> Implementation | 35 |
| <9.4> Graphical analysis | 36 |
| <10> Conclusions | **37** |
| <11> References | **38** |

**<1> Abstract**

**<1.1> Introduction**

Every industry with networking integrated into their infrastructure requires an autonomous system to monitor its respective network performance statistics and apprehend any situation as they arrive. This requirement hugely benefits from the advent of machine learning methodologies which help computers to closely emulate human analytical skills by autonomously gauging the performance and general traits of any network’s dataflow and pointing out potential information that can help in improving the said network’s statistics. Hence, the monitoring system should closely inspect such metrics (like network throughput, nature of incoming and outgoing packets of data, network and transmission control layer statistics, etc.) in real time and analyse them as desired. This project is aimed at proposing such a solution, i.e. a model, that will be able to understand legitimate trends, as and when they appear, of any given time series based statistical data of network metrics and detect anomalous behaviour out of the same in real time.

**<1.2> Objectives**

The two major phases of the project are described briefly as below.

**<1.2.1> Developing a customised time series dataset**

* Since procuring actual time series data of data traffic generated on industrial networks is not practically feasible for this project, we need the next best alternative to base our operations on.
* To evaluate each of the three proposed models in a uniform manner, create identical testing environments and converge onto common grounds of comparable forecasting and anomaly detection results, we will synthesize our own dataset with manually injected trends, noise, seasonality and anomalies.
* This dataset will be a customised univariate time series of packet flow which will mirror real life network trends as much practically as possible.

**<1.2.2> Investigating potential forecasting models**

* The primary goal of this project is to design an algorithmic statistical evaluation solution which will be able to detect anomalies in a time series dataset of network packet capture on a real time basis.
* This project will be aimed to investigate three potential machine learning models, i.e., the Auto-Regressive Integrated Moving Average (ARIMA) model, followed by Long Short Term Memory (LSTM) model and ultimately the Hierarchical Temporal Memory (HTM) model.

**<1.3> Scope of the project**

For the sake of simplicity, the scope of the project has been limited to the following aspects:

* The synthetic dataset that will be developed for comparison of the models for anomaly detection will be a univariate time series stream, i.e., there will be a single varying against time .
* Since streaming data is not available the models will be fed point by point during prediction to emulate the real-time prediction scenario.

**<1.4> Expectations from the best solution**

* Predictions must be made online, i.e., the algorithm must identify current state as normal or anomalous before receiving the subsequent state.
* The model must learn continuously without a requirement to store the entire stream.
* The model must run in an automated fashion, i.e., without data labels or manual database tweaking.
* Model must adapt to dynamic environments and concept drift, as the underlying statistics of the data stream is often non-stationary.
* Model should make anomaly detections as early as possible.
* Model should minimize false positives and false negatives (this is true for batch scenarios as well).

The three models proposed above will be examined closely to root out factors that make them beneficial in certain situations, and to show how HTM outperforms the other two models.

**<2> Supervised vs unsupervised learning**

**<2.1> The hypothesis**

In case of supervised learning, any and all trends/cycles of a typical time series dataset are explicitly exhibited to the respective model, as is expected from this mode of training. Hence, the model is generally expected not to identify anything outside the trained behaviour patterns as a part of the current pattern being analysed, i.e., normal. This is a matter of concern when it comes to analysis of real time streaming of data which may not always follow/reiterate any behaviour from the previous time cycles, and any new trend may surface which are radically different from anything the model has ever processed up till the present. We know this to be called as Concept Drift. So, in spite of the data being a normal result of some dynamic changes introduced manually and intentionally, it will almost always be flagged as anomalous. This is also true in case of the presence of seasonality in the time series data, which is usually not captured by supervised models even if such data is provided in training. This is not a desirable property we want in our best solution.

This is where our expectations from unsupervised learning algorithms need to be mentioned. Since all unsupervised learning follows classification of incoming new data values without any prior training, it is generally expected that the model will be able to adapt to concept drifts. Seasonality will be accurately observed and learned. Also, since unsupervised algorithms will not require any training data to boot, we expect such class of models to be the most suitable candidate for our requirements, i.e., real time data analysis, where initial availability of sufficient data (for training) is rather uncertain.

To clearly test this hypothesis of the prowess of unsupervised learning over supervised learning, we will be investigating three models of appropriately comparable calibre (in terms of time series forecasting) and examining its results to verify our expectations.

**<2.2> The big picture**

|  |  |  |
| --- | --- | --- |
|  | *Supervised Learning* | *Unsupervised Learning* |
| *Input Data* | uses known and labelled data as input | Uses unknown data as input |
| *Computational Complexity* | very complex | less computational complexity |
| *Real Time* | uses off-line analysis | uses real time analysis of data |
| *Number of Classes* | number of classes are known | number of classes are not known |
| *Accuracy of Results* | accurate and reliable results | moderately accurate and reliable results |

**<2.3> Inclining towards unsupervised learning**

In a nutshell, supervised machine learning algorithms are trained by example. We feed them datasets containing examples which are already labelled or categorized, and this enables the algorithm to build a general model of each category. Then, the algorithm processes the real (un-categorized) data and attempts to put each item into one of the pre-learned categories. Since a supervised algorithm only knows the categories on which it has been trained, and its training was conducted on pre-labelled examples, a supervised machine learning algorithm cannot place an item into a category it has not seen an example of. This means that an automated anomaly detection system built on such an algorithm would have to be given examples of every single possible type of anomaly on every possible data distribution, pattern and trend.

**<2.4> In perspective of the investigation**

In this project, ARIMA along with LSTM will be the supervised models standing in comparison with the unsupervised model of HTM. The comparison will show how supervised learning is dependent on availability of actual outcomes corresponding to the earlier predicted outcomes in order to detect outliers, whereas unsupervised learning models analyse the incoming stream of data in real time to produce instantaneous results.

**<3> Continuous learning & real-time analysis**

**<3.1> Learning like humans do**

We always learn to do something by continuously engaging with the activity to encounter new challenges and adapt for upcoming ones. Our brain reconfigures itself in order to incorporate the lessons learnt from these experiences as guidelines for any action to take in the future. Neural networks are one step closer to make “logical thinking” of machines emulate the working of the human brain.

Deep neural networks are currently the most successful machine learning technique for solving a variety of tasks including language translation, image classification and image generation. However, they have typically been designed to learn multiple tasks only if the data is presented all at once. As a network trains on a particular task, its parameters are adapted to solve the task. When a new task is introduced, new adaptations overwrite the knowledge that the neural network had previously acquired. This phenomenon is known in cognitive science as “catastrophic forgetting” and is considered one of the fundamental limitations of neural networks.

By contrast, our brains work in a very different way. We are able to learn incrementally, acquiring skills one at a time and applying our previous knowledge when learning new tasks.

**<3.2> Introducing continuous learning**

Continuous learning is an idea about learning continuously and adaptively about the external world and enabling the autonomous incremental development of ever more complex skills and knowledge. In the context of machine learning, it means being able to smoothly update the model to take into account extremely dynamic and radical changes in the observed input stream, adapting to them, but still being able to re-use and retain useful information from what has already been analysed in the past.

The role that continuous learning will play in this project will be to help the solution model make acceptable logical decisions over any extrinsic set of incoming data it encounters, and acknowledge its existence as a legitimate trend if so happens to be the case. But what it will also help in, is to retain the training from all the past trends it has already gone through, so as to be able to identify anything similar in the future cognitively.

Most importantly, continuous learning helps a model discover fresh actionable insights from new data, before those insights become redundant or outdated. This is one of the major factors of the keynote of this project, which is real-time anomaly detection.

**<3.3> Emerging need for real-time analysis**

Starting from the search engines that have become the first solution to every problem, online stores which sell almost everything you can imagine, to those annoying one-note advertisements that pop up on your browser – real-time data interpretation and prediction is omnipresent in the digital world of the internet. Development of instantaneous information with negligible margins for error has a tremendous demand in the commercial and technological sphere.

Deriving insights and reaching plausible conclusions instantaneously as soon as data is available for analysis can significantly increase the value of such information. Decision-making founded on such a model makes the analysis ‘immediately useful’, helping an infrastructure react and/or adapt rapidly and swiftly to any factors that may change without notice. This is more realisable in the case of network performance analysis, where even a single log of anomalous packet flow may indicate a transient defect in performance and delivery, which if not addressed immediately, may create larger, irreversible technical difficulties to deal with down the line.



**<4> A literature survey**

**<4.1> Scope of alternative solutions**

Considering that there are quite a number of models which specialise in time series analytics alongside ARIMA, LSTM and HTM in general, namely Seasonal ARIMA (SARIMA), Generalized Autoregressive Conditional Heteroskedasticity (GArCH), and Isolation Forest, we analyse each of them briefly.

**<4.1.1> SARIMA**

The Seasonal Autoregressive Integrated Moving Average (SARIMA) method models the next step in the sequence as a linear function of the differenced observations, errors, differenced seasonal observations, and seasonal errors at prior time steps. It combines the ARIMA model with the ability to perform the same autoregression, differencing, and moving average modelling at the seasonal level. The notation for the model involves specifying the order for the AR(p), I(d), and MA(q) models as parameters to an ARIMA function and AR(P), I(D), MA(Q) and m parameters at the seasonal level, e.g. SARIMA(p, d, q)(P, D, Q)m where “m” is the number of time steps in each season (the seasonal period). A SARIMA model can be used to develop AR, MA, ARMA and ARIMA models.

Yet, it is very important to note that this model’s capabilities to realise the seasonality of the dataset completely depends on the parameter m which (along with P, D and Q) needs to be manually deduced either by intuition or brute-force. This is an inconvenience considering that the seasonality of our timeseries in particular cannot be hardcoded into the model and also that the preliminary model only needs to support the requirement of neural networks.

**<4.1.2> GArCH**

Generalized Autoregressive Conditional Heteroskedasticity (GArCH) has a similar caveat to SARIMA. The focus of this particular model is conditional heteroskedasticity, which is generally used on financial series that exhibit volatility clustering ("large changes tend to be followed by large changes, of either sign, and small changes tend to be followed by small changes." - Benoit Mandelbrot, 1963). Conditional heteroskedasticity exists in finance because asset returns are volatile. A collection of random variables is heteroskedastic if there are subsets of variables within the larger set that have a different variance from the remaining variables. This is a little out of context of the case study in this project where we are considering a strictly univariate timeseries with seasonal cycles and trends.

With variance as a linear regression, the next lag step in time series is defined as:

(where is the corresponding white noise)

Note that (G)ArCH should only ever be applied to a series that has already had an appropriate model fitted sufficient to leave the residuals looking like discrete white noise. (G)ArCH should only ever be applied to series that do not have any trends or seasonal effects, i.e., that has no (evident) serial correlation. ARIMA is often applied to such a series, at which point (G)ArCH may not be a good fit.

**<4.1.3> Isolation Forest**

Isolation Forest, a method which in principle is similar to the well-known and popular Random Forest, which is different from other popular outlier detection methods, is that Isolation Forest explicitly identifies anomalies instead of profiling normal data points. Isolation Forest, like any tree ensemble method, is built on the basis of decision trees. In these trees, partitions are created by first randomly selecting a feature and then selecting a random split value between the minimum and maximum value of the selected feature.

In principle, outliers are less frequent than regular observations and are different from them in terms of values (they lie further away from the regular observations in the feature space). That is why by using such random partitioning they should be identified closer to the root of the tree (shorter average path length, i.e., the number of edges an observation must pass in the tree going from the root to the terminal node), with fewer splits necessary.

A normal point requires more partitions to be identified than an abnormal point.

As with other outlier detection methods, an anomaly score is required for decision making. In case of isolation forest it is defined as:

where is the path length of observation , is the average path length of unsuccessful search in a binary search tree and is the number of external nodes.

Each observation is given an anomaly score and the following decision can be made on its basis:

* Score close to 1 indicates anomalies
* Score much smaller than 0.5 indicates normal observations
* If all scores are close to 0.5 than the entire sample does not seem to have clearly distinct anomalies

**<4.2> Preference of ARIMA**

The reasons for choosing ARIMA are very simple. The foundational algorithm that ARIMA works on is very objective and fine tuning the model to fit the dataset is an exercise in intuition. There is no brute-force grid searching required to zero in on the optimum fit (although it is a viable option here too). Even though ARIMA only scratches the surface in terms of accurate forecasting, specifically its ignorance of seasonality, its simplicity in statistical outputs allows a lot of room for augmentations (in our case, anomaly detection criteria). In a nutshell, ARIMA is an excellent primer for a jumpstart into time series analysis.

**<4.3> Preference of LSTM**

To bridge the transition gap between the supervised learning of ARIMA and unsupervised neural network in our investigation, we need another model to introduce neural networking into the fray. Considering ARIMA is a statistical model, we wanted to keep our investigative exploration a little bit spaced out, specifically to a Recurrent Neural Network (RNN). RNNs are a powerful type of neural network designed to handle sequence dependence. They learn to selectively retain relevant information allowing them to capture dependencies across several time steps. This allows them to utilize both current input and past information while making future predictions, a feature that is very useful for predicting trends and detecting anomalies. We choose Long Short-Term Memory (LSTM) network specifically because it is an improved version of RNN. While keeping its core working aspect same, it successfully removes its short term memory for retaining information thus also fixing its problem of not adopting properly if a pattern keeps occurring at long time intervals, i.e., seasonality in data. Moreover, LSTM can forget unnecessary information as time progresses. All these are beneficial for fulfilling our objective.

**<4.4> Preference of HTM**

Although isolation forest and other unsupervised algorithms like random forest appear to be a good fit for outlier detection in many scenarios, when it comes to real-time streaming data, these solutions incur a major setback because of being tree ensemble methods which profile the outliers over the whole dataset instead of real-time instance. So considering the importance of on-the-fly inferencing as the main objective we choose Hierarchical Temporal Memory (HTM) as a candidate over the above algorithms that has the capability of online learning and on-the-fly Inferencing.

**<5> Experimental setup**

**<5.1> System specifications**

All experimental procedures of this investigative project have been executed on the system as specified below:

* Processor: Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz (8 CPUs), ~2.3GHz
* Memory: 8192 MB, DDR4
* Operating System: Windows 10 Home Single Language 64-bit (10.0, Build 17763)

**<5.2> IDE specifications**

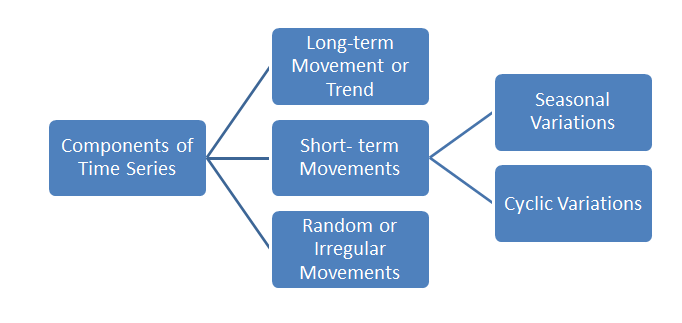
All code segments demonstrated in this project are based on the development platform as specified below:

* Python 3.7
* Anaconda 3
* JetBrains PyCharm Community Edition 2018.3.5

**<6> Developing the dataset**

**<6.1> About time series**

Time Series is a sequence of well-defined data points measured at consistent time intervals over a period of time. Data collected on an ad-hoc basis or irregularly does not form a time series. Time series analysis is the use of statistical methods to analyse time series data and extract meaningful statistics and characteristics about the data. Time series Analysis helps us understand what are the underlying forces leading to a particular trend in the time series data points and helps us in forecasting and monitoring the data points by fitting appropriate models to it.



**<6.2> Choosing univariate time series**

The term "univariate time series" refers to a time series that consists of single (scalar) observations recorded sequentially over equal time increments. Although a univariate time series data set is usually given as a single column of numbers, time is in fact an implicit variable in the time series. If the data are evenly spaced, the time variable, or index, does not need to be explicitly given. The time variable may sometimes be explicitly used for plotting the series. However, it is not used in the time series model itself. This is yet another advantage of such a series, where the actual index value of time for any particular observation is purely subjective.

**<6.3> Developing a time-series dataset**

Any time series consists of the above mentioned four components:

* Trend / long-term movement
* Seasonal variations
* Cyclic variations
* Random/irregular movements or noise

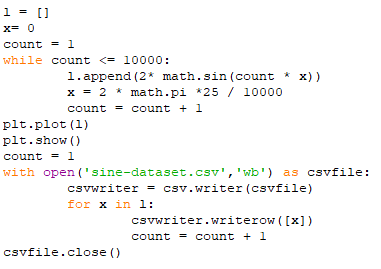
In order to create this type of a dataset, the below proposed methodology has been adapted:

* Create two individual trends of same size
* Merge the two created trends
* Introduce Periodic fluctuations in the merged trend
* Amplify the trend ununiformly in a random manner to introduce noise

Clearly the above methodology lacks cyclic fluctuations. In fact cyclic fluctuation is difficult to produce by hand.

**<6.3.1> Generating a sine trend**

First of all, for creating two individual trends we are considering the sinusoidal trend and the exponential trend. We write two separate functions for the same and publish them as dataframes.



The result is:

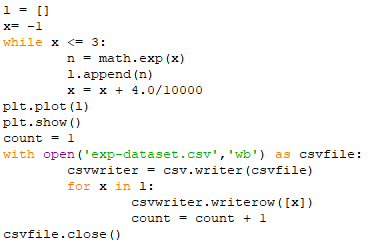
A screenshot of a cell phone

Description generated with very high confidence

sine trend created

**<6.3.2> Generating an exponential trend**

Similarly, we create an exponential trend and get it published:



The result is:

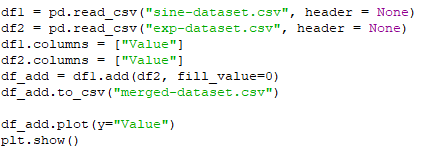
A screenshot of a cell phone

Description generated with very high confidence

exponential trend created

**<6.3.3> Additive superimposition of both trends**

We superimpose these two trends one upon another:



The corresponding result is:

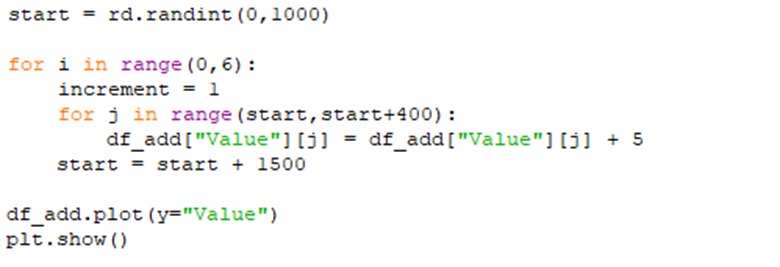
A screenshot of a cell phone

Description generated with very high confidence

both trends superimposed

**<6.3.4> Introducing seasonal variations**

We then introduce some seasonal variation. A random starting point is chosen and from that starting point we go on amplifying the values constantly up to a pre-determined time and periodically we do the same after certain interval until the end. This is shown below:



The result is:

A screenshot of a cell phone

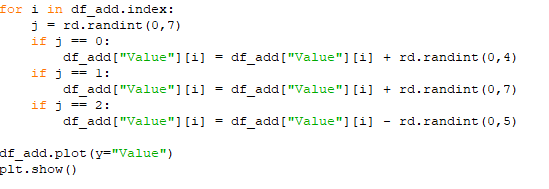
Description generated with very high confidence

seasonal variations added

**<6.3.5> Injecting noise**

Finally, we introduce noise. For that, we iterate over the time series with a flag, whose value changes randomly at each iteration point, and based on that flag value either the current datapoint is amplified or reduced by some constant value or it remains unchanged.

The corresponding code snippet:



The result is:

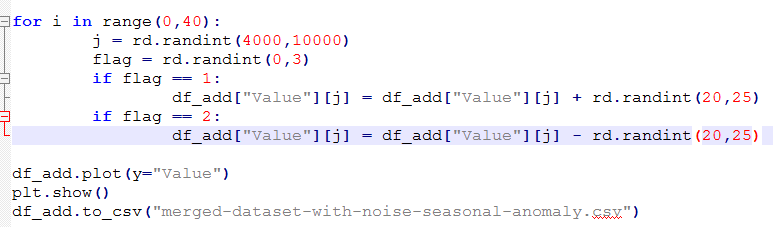
A close up of a tree

Description generated with high confidence

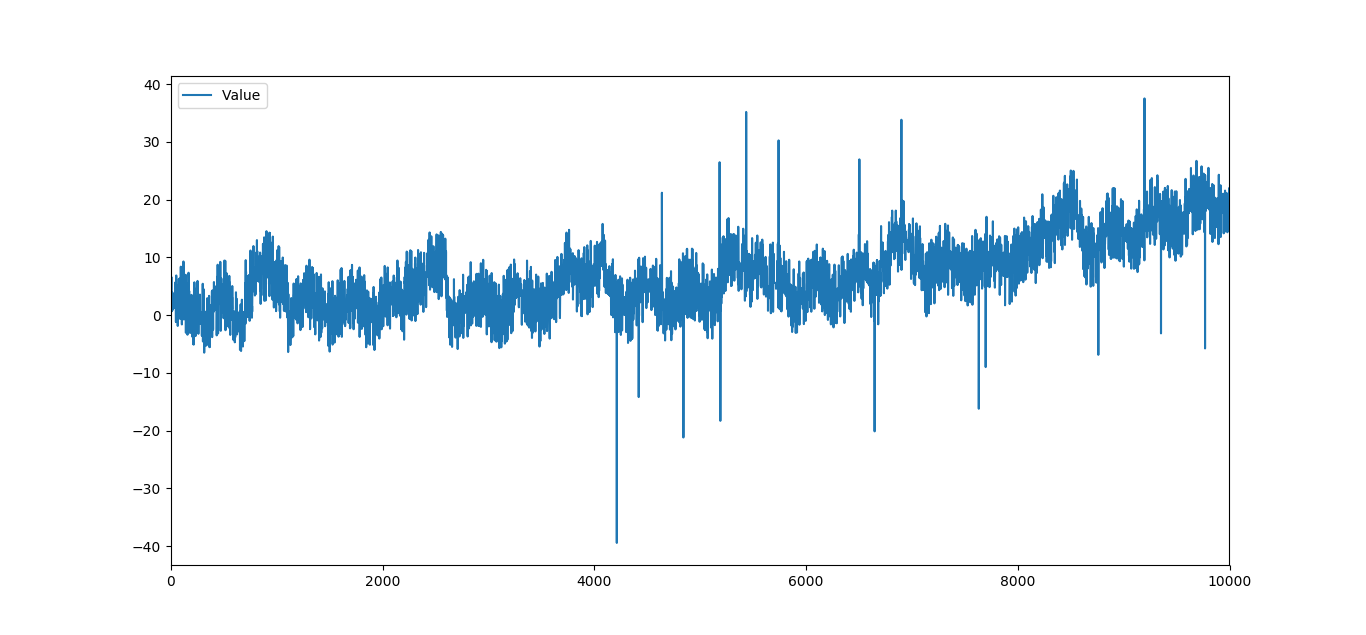
noise injected

**<6.3.6> Inserting anomalies**

We insert some random anomalies in the dataset careful enough to insert in the later phase of the dataset:



The final dataset when plotted is a perfect synthetic time series:



final dataset with anomalies

**<7> Investigating ARIMA**

**<7.1> The models involved**

The anagram ARIMA stands for Auto Regressive Integrated Moving Average, which is a reflection of the fact that ARIMA is an amalgamation of two time series forecasting models, namely autoregression (AR) and moving average (MA). To understand how ARIMA is the best of both worlds, it is important to know these two models individually first.

**<7.1.1> Autoregression**

Autoregression is a time series model that uses observations from previous time steps as input to a regression equation to predict the value at the next time step. It is a very simple idea that can result in accurate forecasts on a range of time series problems.

Now, a regression model (such as linear regression) models an output value based on a linear combination of input values. Taking a familiar equation for example:

Where is the prediction, and are coefficients found by optimizing the model on training data, and is an input value. This technique can be used on time series where input variables are taken as observations at previous time steps, called lag variables. Because the regression model uses data from the same input variable at previous time steps, it is referred to as an autoregression (regression of self).

So, we can predict the value for the next time step given the observations at the last time steps . As a regression model, this would look as follows:

Now, in order to determine this critical value (and also the order of the AR model) of , i.e., the number of lags which will give the optimum prediction, we need to find the relation of the output variable with its own predecessors to find out up to which lag does the current output variable strongly relate with the previous outputs. This relationship between variables is called correlation.

If both variables change in the same direction (e.g. go up together or down together), this is called a positive correlation. If the variables move in opposite directions as values change (e.g. one goes up and one goes down), then this is called negative correlation. We can use statistical measures to calculate the correlation between the output variable and values at previous time steps at various different lags. The stronger the correlation between the output variable and a specific lagged variable, the more weight that autoregression model can put on that variable when modelling.

Again, because the correlation is calculated between the variable and itself at previous time steps, it is called an autocorrelation. It is also called serial correlation because of the sequenced structure of time series data.

**<7.1.2> Moving average**

The moving average model, also known as moving average process, is a common approach for modelling 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. In terms of interpretation, an MA model simply means that the time series is a function of the error from previous periods. The moving-average model should not be confused with the moving average, a distinct concept despite some similarities.

A moving average model is similar to an autoregressive model, except that instead of being a linear combination of past time series values, it is a linear combination of the past white noise terms. Intuitively, this means that the MA model sees such random white noise "shocks" directly at each current value of the model. This is in contrast to an AR model, where the shocks are only seen indirectly, via regression onto previous terms of the series. The random shocks at each point are assumed to be mutually independent and to come from the same distribution, typically a normal distribution, with location at zero and constant scale.

A moving average model of order can be mathematically represented as:

where is the mean of the series, the  are, again, coefficients found by optimizing the model and the  are white noise error terms.

Here, the order is generally deduced using the partial autocorrelation of terms at n lags. A partial autocorrelation is a summary of the relationship between an observation in a time series with observations at prior time steps with the relationships of intervening observations removed.

**<7.2> Auto Regressive Integrated Moving Average**

The ARIMA forecasting model is a combination of both autoregression and moving average along with a third factor, integration. Unlike the name of the term may suggest, it represents the differencing of raw observations (e.g. subtracting an observation from the previous time step).

For example, differencing a data twice can be represented as:

Differencing in statistics is a transformation applied to time-series data in order to make it stationary (statistical properties like mean and standard deviation remain constant with time). This allows the properties to not depend on the time of observation, eliminating trend and seasonality and stabilizing the mean of the time series.

A serious resultant drawback of this is the model’s inability to support seasonal data. ARIMA expects data that is either not seasonal or has the seasonal component removed by differencing.

The non-differenced expression for the ARIMA model can be simply represented as:

**<7.3> Implementation**

Following is a detailed walkthrough of applying the ARIMA model onto the custom generated dataset and establishing the need for a superior model.

**<7.3.1> Checking stationarity of the dataset**

Reading the dataset .csv file as follows gives us some insight into the nature of the data that ARIMA has to deal with.

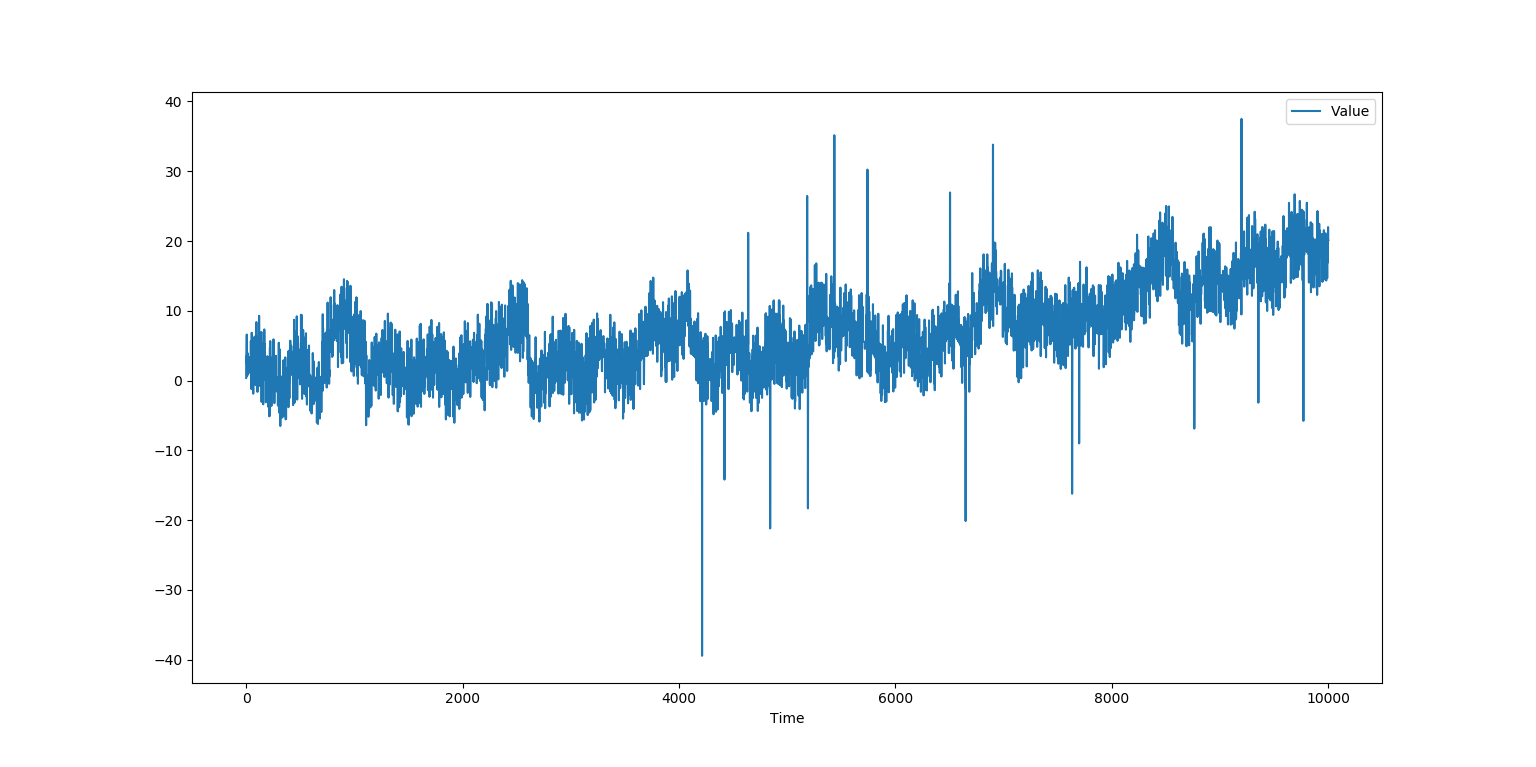
from pandas import read\_csv

from matplotlib import pyplot

series = read\_csv('custom.csv', header=0,index\_col=0)

series.plot()

pyplot.show()



It is quite obvious from the plot of the time series that it is not stationary.

To have a consolidated proof of the same, we map some basic statistical properties of the data.

rolmean=series.rolling(window=100).mean()

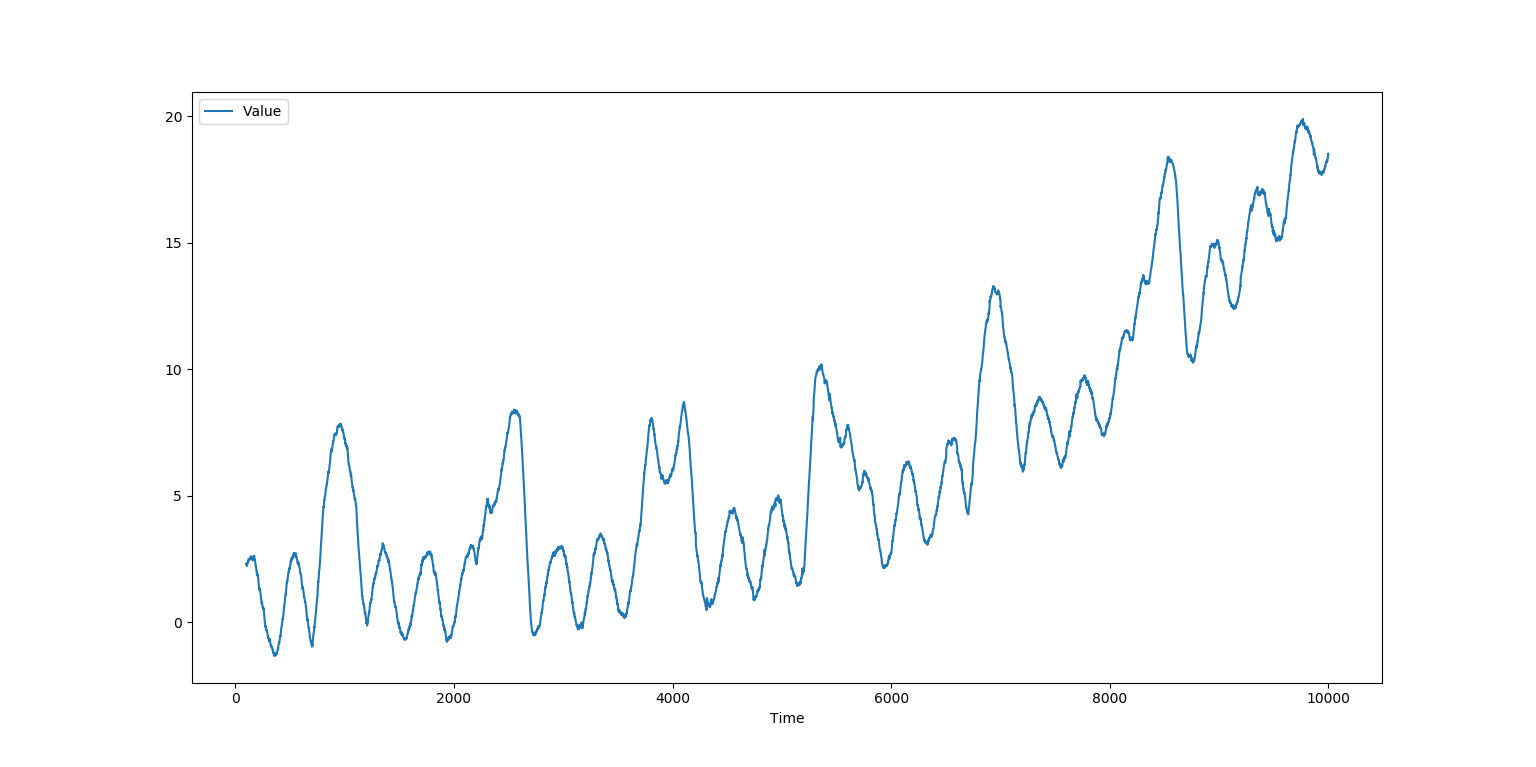
rolstd=series.rolling(window=100).std()

rolmean.plot()

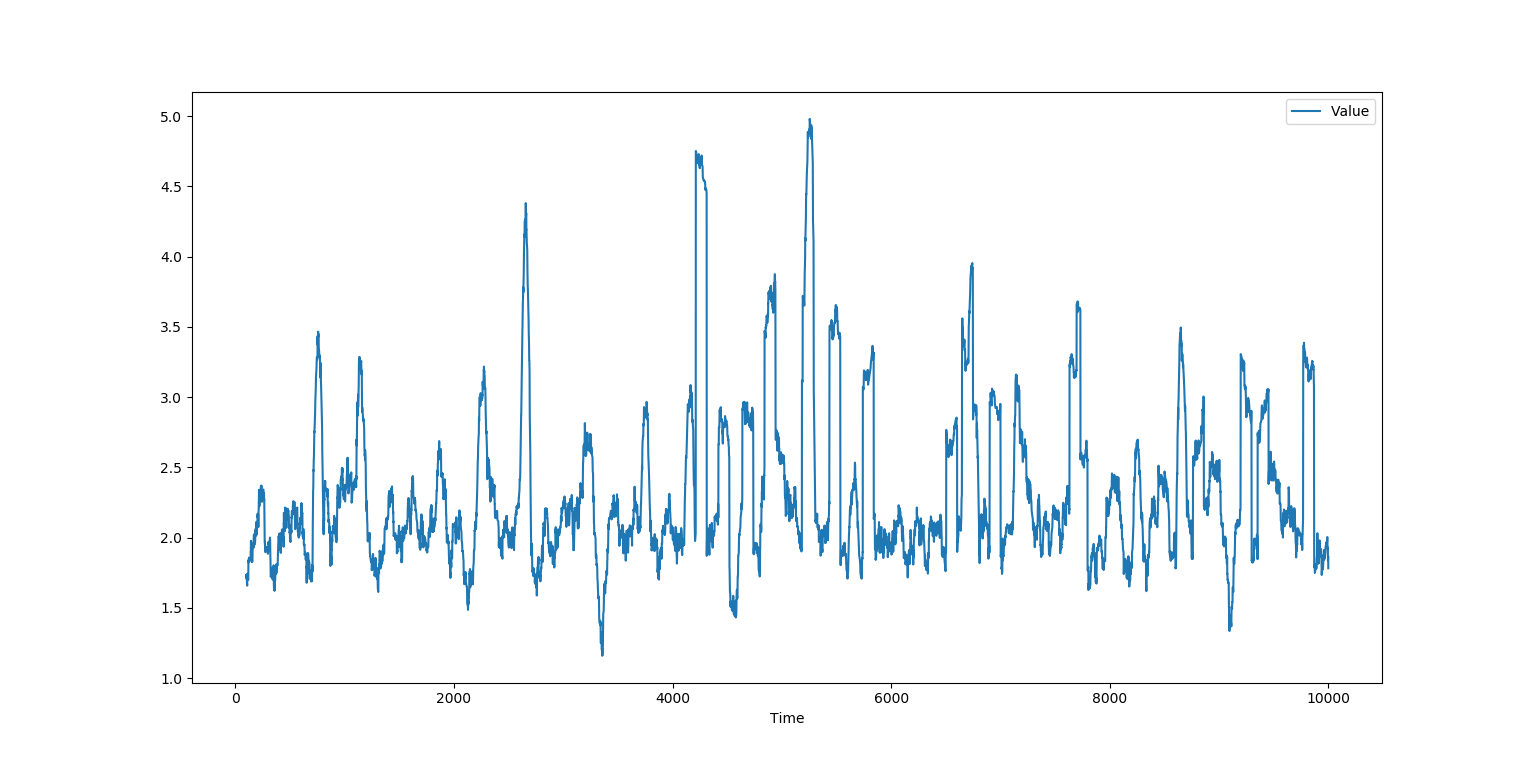
rolstd.plot()

pyplot.show()

Rolling mean:



Rolling standard deviation:



The rolling mean and standard deviation essentially plots the mean and standard deviation of a group of consecutive values in the series (groups of 100 values as indicated by the window parameter) as a sliding window. It can be seen that the values are not quite stable and rolling mean has a generally increasing trend.

We also perform an obligatory augmented Dickey-Fuller test to show the stationarity of the data. Without delving into specifics, it is only sufficient to realise that the p-value needs to be even closer to zero than it is here in this case.

from statsmodels.tsa.stattools import adfuller

dftest=adfuller(series['Value'],autolag='AIC')

dfoutput=pd.Series(dftest[0:4],index=['test\_statistic','pval','lags\_used','no\_of\_obs'])

for key,value in dftest[4].items():

dfoutput['critical\_val (%s)'%key]=value

print(dfoutput)

**test\_statistic -1.638380**

**pval 0.463107**

**lags\_used 27.000000**

**no\_of\_obs 9972.000000**

**critical\_val (1%) -3.431006**

**critical\_val (5%) -2.861830**

**<7.3.2> Determining ARIMA parameters**

ARIMA being a combination of AR and MA, we need to estimate the closest possible orders of both the models which represent the dataset in the best possible way.

So given a model ARIMA(dataset, order=(p, d ,q)) :

* **p**: number of lag observations included in the model, i.e., order of autoregression
* **d**: number of times that the raw observations are differenced
* **q**: size of the moving average window, i.e., order of moving-average

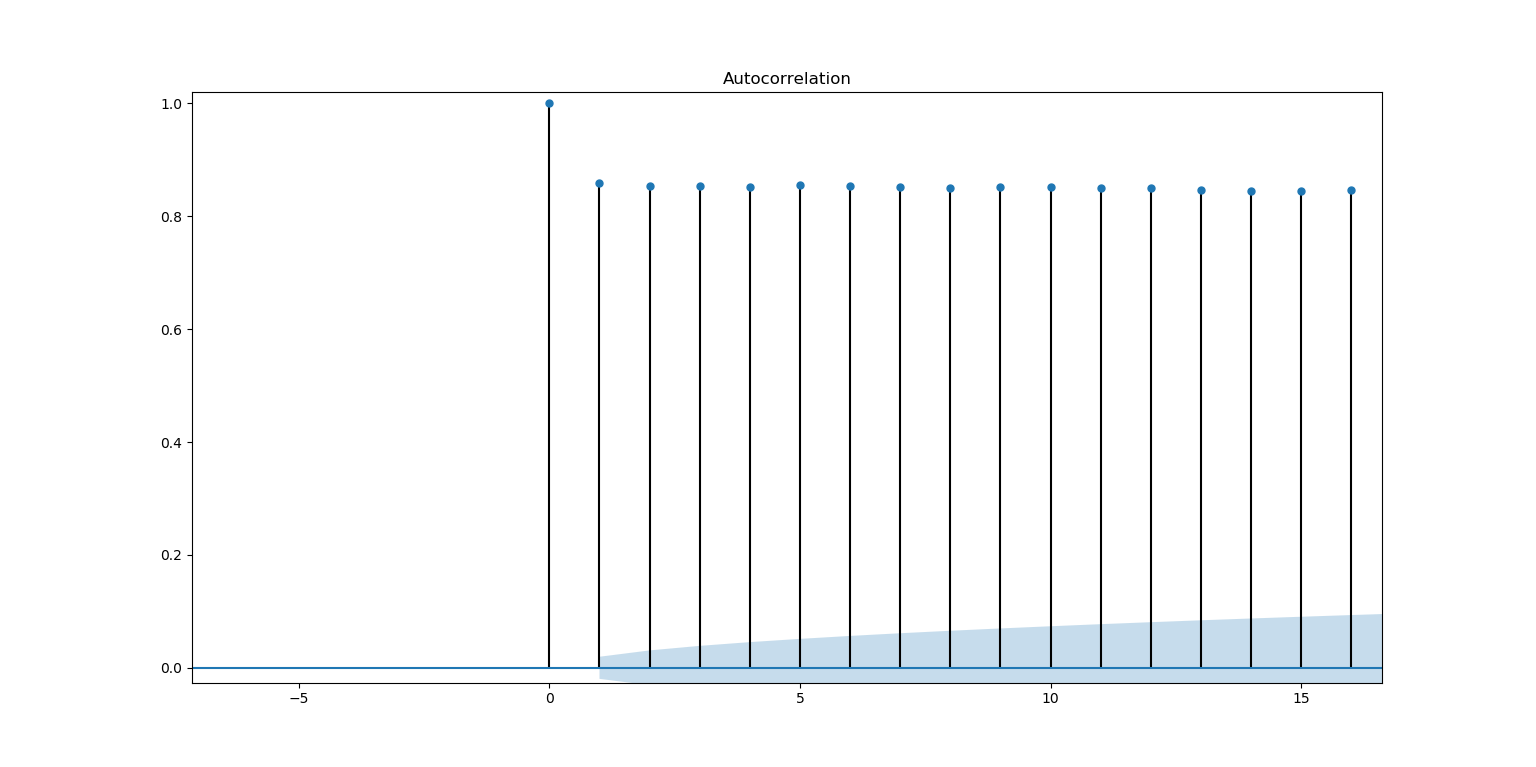
As explained earlier, we need to refer to the autocorrelation and partial autocorrelation plots (ACF and PACF respectively) of the dataset to reach at a conclusion for the above parameters.

from statsmodels.graphics.tsaplots import plot\_acf, plot\_pacf

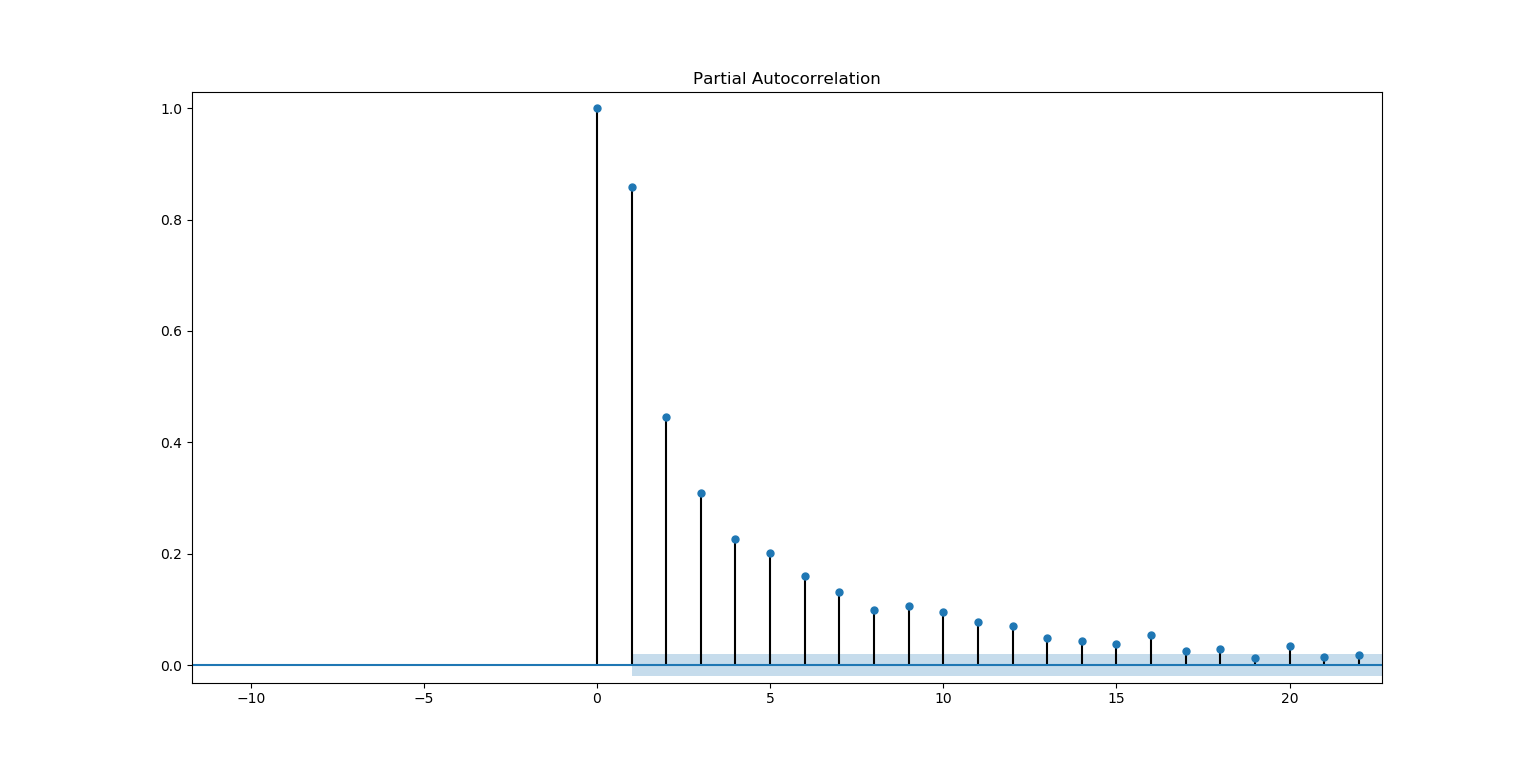
plot\_acf(series)

plot\_pacf(series, lags=500)

pyplot.show()



From the ACF plot, it is clearly evident that the 0 interval of lags produces the highest correlation factor and is significantly better compared to its successors in the plot. Lets appoint an order of 0 to parameter p.



The PACF plot shows that the first two lags (0 through 1) have characteristically desired correlation factors, so we appoint an order of 1 to parameter q.

It should be noted that while the ACF plot is fed the entire set as input, the PACF plot is much more time-consuming to produce, hence we reduce the number of lags it needs to scan through so as to hasten the process. As any observation-worthy information is discernible within the initial segments of any dataset, this reduction has absolutely no effect on the quality of the results.

Coming to the differencing order d, it is very difficult to arrive at a definitive value considering that the dataset is not even stationary to begin with. To keep the inspection of forecasting results comparable with the next two models, we follow conventions to keep the order to a default of 1 which after multiple trials proved to be giving the best results.

**<7.3.3> Preparing and executing the model**

Now that we have our parameters fixed, we can begin formulating the model. First we segregate the training (1/3rd) and testing (2/3rd) parts from the raw dataset.

X = series.values

size = int(len(X) \* 0.33)

train, test = X[0:size], X[size:len(X)]

history = [x for x in train]

The history set is nothing but the training subset converted from ndarray format of train for compatibility reasons.

Next, we initiate the forecasting process by iterating the decided model over the testing subset with order (0, 1, 1).

from statsmodels.tsa.arima\_model import ARIMA

predictions = []

ucl = []

lcl = []

std\_err = []

for t in range(len(test)):

print("Forecasting: ",(t+1)\*100/len(test), "% complete...")

model = ARIMA(history, order=(0,1,1))

model\_fit = model.fit(disp=0)

output, stderr, conf = model\_fit.forecast()

ucl.append(conf[0][1])

lcl.append(conf[0][0])

std\_err.append(stderr)

predictions.append(output)

obs = test[t]

history.append(obs)

For every value in the testing subset, the ARIMA model is trained with the current history set and the model is fitted by model.fit() onto the underlying dataset. Then, a single value forecast is executed with model\_fit.forecast() storing its results into three variables.

output – contains the predicted value

stderr – contains the standard error value of the prediction

conf – the upper and lower limits of the prediction range

The lists ucl and lcl contain all the respective upper/lower limits of prediction to be represented as confidence levels of the prediction plot. The predictions list contains all the predicted values from the variable output. The std\_err list contains all the values from the variable stderr. Before the next iteration, the history set is updated with the present experience, i.e., the currently tested actual expected value.

**<7.3.4> Classifying anomalies**

Now that we have our list of predictions, we can point out anything that the model may consider as anomalous behaviour from the corresponding actual observations. Here, we decide to classify a value as an anomaly which lies outside the confidence interval with an admissible margin of error (using std\_err).

anomaly = list()

for i in range(len(test)):

if test[i] > (ucl[i]+2\*std\_err[i]) or test[i] < (lcl[i]-2\*std\_err[i]):

anomaly.append(test[i])

else:

anomaly.append(np.NaN)

**<7.3.5> Graphical analysis**

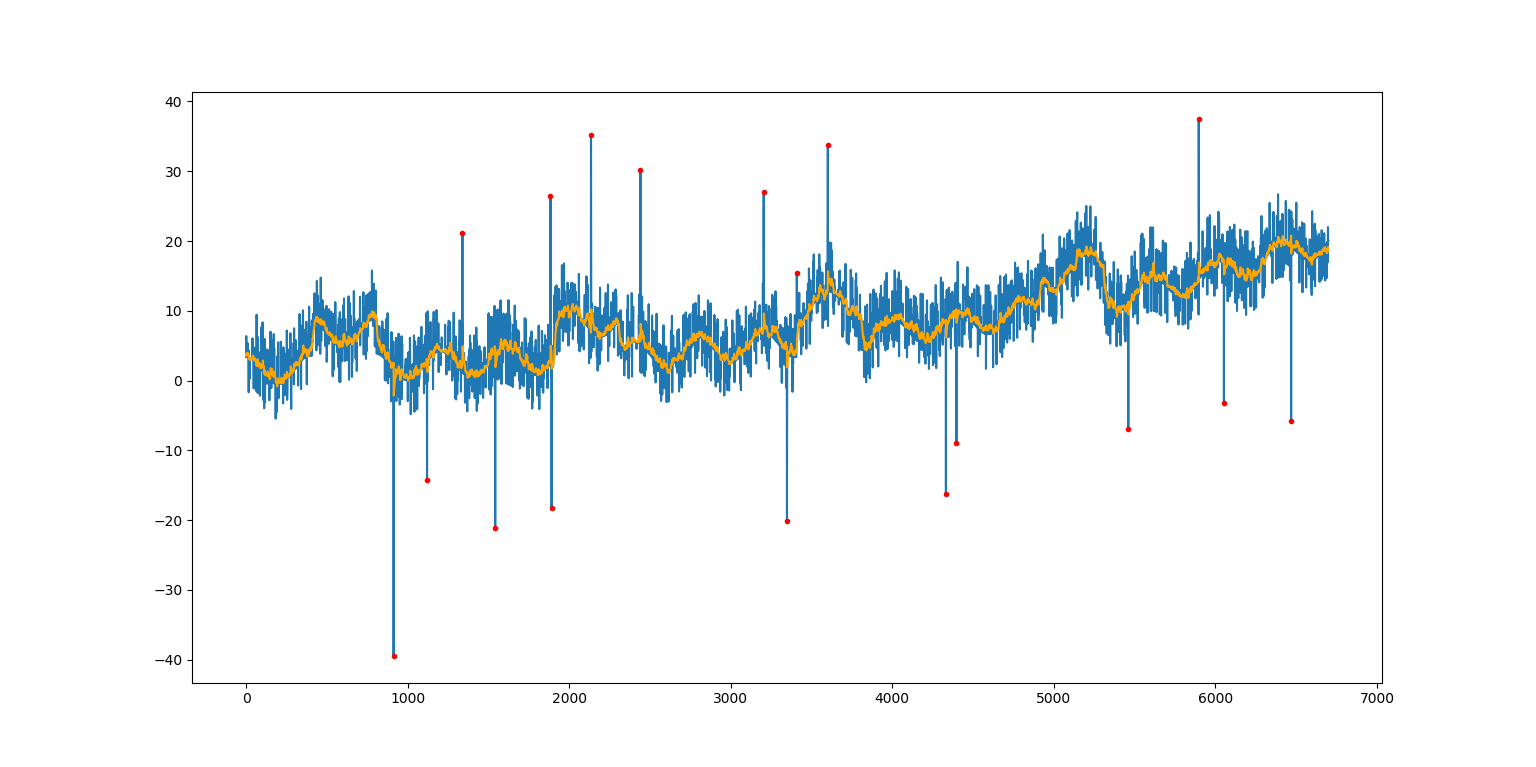
Plotting the expected values of test in blue, predicted values of predictions in orange and the flagged anomalies from anomaly as red dots, we get a conclusive forecast analysis of the ARIMA model.

pyplot.plot(test)

pyplot.plot(anomaly, marker='.', linestyle='', color='red')

pyplot.plot(predictions, color='orange')

pyplot.show()



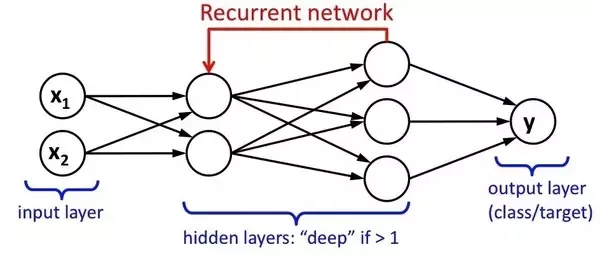
Confusion matrix and measurements:

|  |  |  |  |
| --- | --- | --- | --- |
| *Total(N) = 6700* | *Predicted Normal* | *Predicted Anomaly* |  |
| *Actual Normal* | 6682 (TN) | 13 (FP) | 6695 |
| *Actual Anomaly* | 0 (FN) | 5 (TP) | 5 |
|  | 6682 | 18 |  |

**<8> Investigating LSTM**

**<8.1> Recurrent Neural Networks**

Recurrent Neural Networks (RNN) are a powerful and robust type of neural networks and belong to the most promising algorithms out there at the moment because they are the only ones with an internal memory. Because of their internal memory, RNN’s are able to remember important things about the input they received, which enables them to be very precise in predicting what’s coming next. In a RNN, the information cycles through a loop. When it makes a decision, it takes into consideration the current input and also what it has learned from the inputs it received previously.



**<8.2> Introducing LSTM**

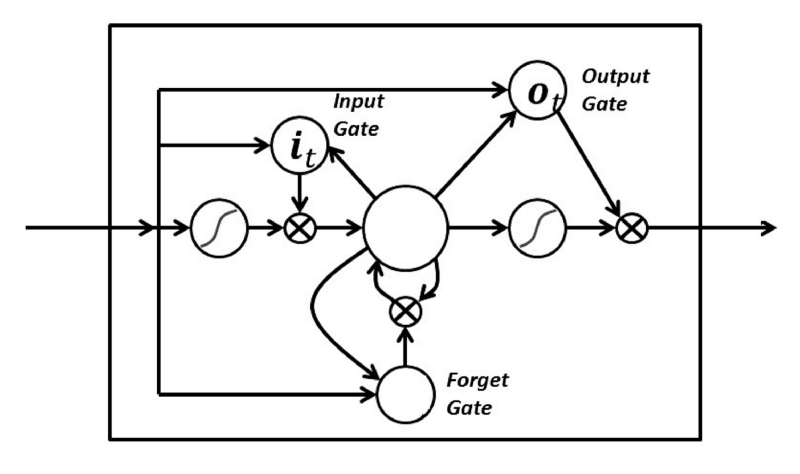
Long Short-Term Memory (LSTM) was invented because of the increase in available computational power and the massive amounts of data that we have nowadays. An usual RNN has a short-term memory. LSTM networks are an extension for recurrent neural networks, which basically extends their memory. Therefore it is well suited to learn from important experiences that have very long time lags in between. RNNs work just fine when we are dealing with short-term dependencies but fails when the information needs to be kept in mind for a long duration. That is where LSTM comes into play.

There are two major inherent problems that RNNs have or had to deal with i.e., vanishing and exploding gradients. A gradient measures how much the output of a function changes, if you change the inputs a little bit. Exploding gradients are when the algorithm assigns a stupidly high importance to the weights, without much reason. On the other hand, vanishing gradients are when the values of a gradient are too small and the model stops learning or takes way too long because of that. Exploding gradients can be easily solved by truncating or squashing the gradients. The problematic issues of vanishing gradients is solved through LSTM because it keeps the gradients steep enough and therefore the training relatively short and the accuracy high.

Thus, for a RNN, in order to add a new information, it transforms the existing information completely by applying a function. Because of this, the entire information is modified, on the whole, i.e. there is no consideration for *‘important’* information and *‘not so important’* information. LSTMs on the other hand, make small modifications to the information by multiplications and additions. This way, LSTMs can selectively remember or forget things.

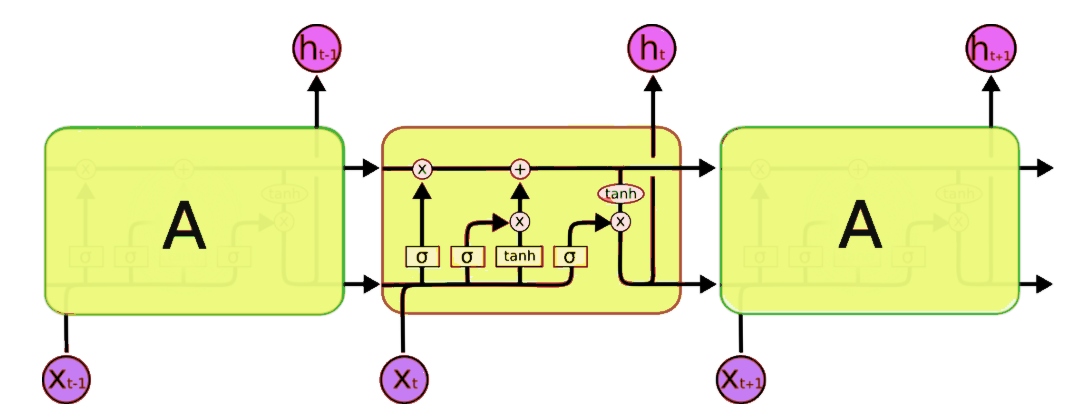
**<8.3> Architecture of LSTM**

As discussed earlier, LSTM can read, write and delete information from its memory. This memory can be seen as a gated cell, where gated means that the cell decides whether or not to store or delete information (e.g. if it opens the gates or not), based on the importance it assigns to the information. The assigning of importance happens through weights, which are also learned by the algorithm through the training dataset. You can see an illustration of a LSTM with its three gates below:



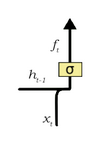
The gates in LSTM are analog, in the form of sigmoids, meaning that they range from 0 to 1. The fact that they are analog, enables them to do backpropagation with it.

Now let’s get into the details of the architecture of LSTM network even further by expanding on the simplistic figure above.

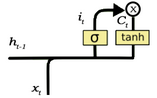


LSTM network is comprised of different memory blocks called **cells** (the rectangles that we see in the image). There are two states that are being transferred to the next cell; the **cell state** and the **hidden state**. The memory blocks are responsible for remembering things and manipulations to this memory is done through three major mechanisms, called **gates.** Each of them is being discussed below.

**<8.3.1> Forget gate**

A forget gate is responsible for removing information from the cell state. The information that is no longer required for the LSTM to understand things or the information that is of less importance is removed via multiplication of a filter. This is required for optimizing the performance of the LSTM network. The sigmoid function is responsible for deciding which values to keep and which to discard. If a ‘0’ is output for a particular value in the cell state, it means that the forget gate wants the cell state to forget that piece of information completely. Similarly, a ‘1’ means that the forget gate wants to remember that entire piece of information. This vector output from the sigmoid function is multiplied to the cell state.

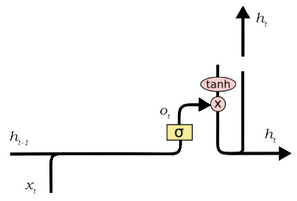
**<8.3.2> Input gate**

The input gate is responsible for the addition of information to the cell state. This addition of information is basically three-step process:

1. Regulating what values need to be added to the cell state by involving a sigmoid function.
2. Creating a vector containing all possible values that can be added to the cell state. This is done using the tanhfunction.
3. Multiplying the value of the regulatory filter (the sigmoid gate) to the created vector (the tanh function) and then adding this useful information to the cell state via addition.

Once this three-step process is complete, we ensure that the information which is added to the cell state is *important* and not *redundant.*

**<8.3.3> Output gate**

The job of selecting useful information from the current cell state and showing it out as an output is done via the output gate. The functioning of an output gate can again be broken down to three steps:

1. Creating a vector after applying tanh function to the cell state, thereby scaling the values to the range -1 to +1.
2. Making a filter such that it can regulate the values that need to be output from the vector created above. This filter again employs a sigmoid function.
3. Multiplying the value of this regulatory filter to the vector created in step 1, and sending it out as output and also to the hidden state of the next cell.

**<8.4> Implementation**

Following is a walkthrough of applying the LSTM model onto the custom generated dataset for anomaly detection.

**<8.4.1> Splitting the dataset**

The class Dataloader is used for reading the values from the supplied dataset and then dividing the data into training and test datasets in the ratio 1:3.

import pandas as pd

import numpy as np

class Dataloader:

def \_\_init\_\_(self,window\_size,filename,column):

df = pd.read\_csv(filename)

df = pd.DataFrame(df[column])

isplit = round(len(df)\*0.25)

self.data\_train = df.get(column).values[:isplit]

self.data\_test = df.get(column).values[isplit:]

self.len\_train = len(self.data\_train)

self.len\_test = len(self.data\_test)

self.window\_size = window\_size

def normalise(self,window\_data):

window\_data = window\_data.reshape(-1,1)

scaler = preprocessing.MinMaxScaler()

return scaler.fit\_transform(window\_data)

def get\_train\_data(self):

data\_x = []

data\_y = []

for i in range(self.len\_train-self.window\_size):

window = self.data\_train[i:i+self.window\_size+1]

normalised\_window = self.normalise(window)

x = normalised\_window[:-1]

y = normalised\_window[-1]

data\_x.append(x)

data\_y.append(y)

data\_x = np.array(data\_x)

data\_y = np.array(data\_y)

return data\_x.reshape(data\_x.shape[0],data\_x.shape[1],1), data\_y

def get\_test\_data(self):

data\_x = []

data\_y = []

for i in range(self.len\_test-self.window\_size):

window = self.data\_test[i:i+self.window\_size+1]

normalised\_window = self.normalise(window)

x = normalised\_window[:-1]

y = normalised\_window[-1]

data\_x.append(x)

data\_y.append(y[0])

return data\_x, data\_y

**<8.4.2> Building the model**

We are using the library Keras,which is a high-level API for neural networks and works on top of TensorFlow. Models in Keras are defined as a sequence of layers. We create a Sequential model and add layers one at a time until we are happy with our network topology.

from keras.layers import Dense, Activation, Dropout, LSTM

from keras.models import Sequential, load\_model

from sklearn.metrics import mean\_squared\_error as mse

def build\_model(layers):

model = Sequential()

model.add(LSTM(input\_dim =1,output\_dim=50))

model.add(Dropout(0.5))

model.add(Dense(1))

model.compile(loss="mse",optimizer="adam")

model.summary()

return model

**<8.4.3> Fitting the model**

Now that we have defined our model and compiled it ready for efficient computation, we can train or fit our model on our loaded data by calling the **fit()** function on the model. The training process will run for a fixed number of iterations through the dataset called epochs. We can also set the number of instances that are evaluated before a weight update in the network is performed, called the batch\_size. We can also set a parameter called validation\_split in which case the model will set apart this fraction of the training data, will not train on it, and will evaluate the loss and any model metrics on this data at the end of each epoch.

seq\_len = 400

data=Dataloader(seq\_len,'merged-dataset-with-noise-and-seasonal(datetime) .csv','Value')

x\_train,y\_train = data.get\_train\_data()

epochs = 15

model = lstm.build\_model([1, 50, 100, 1])

model.fit(

x\_train,

y\_train,

batch\_size=32,

nb\_epoch=epochs,

validation\_split=0.05)

**<8.4.4> Executing the model**

The model is now ready to predict the output based on the data it has been trained on. The test dataset is provided to the model. The model will predict each timestep given the last sequence of true data, in effect only predicting 1 step ahead each time.

def predict\_point\_by\_point(model,x\_test,y\_test,x\_train,y\_train):

predicted\_data = []

for i in range(len(x\_test)):

data = x\_test[i].reshape(1,x\_test[i].shape[0],1)

x\_train = np.append(x\_train,data,axis=0)

x\_train = np.delete(x\_train,0,0)

y\_train = np.append(y\_train,y\_test[i])

y\_train = np.delete(y\_train,0,0)

predicted = model.predict(data)[0][0]

predicted\_data.append(predicted)

if (i+1)%50==0:

if mse(y\_test[i-49:i],predicted\_data[i-49:i])>0.06:

model.fit(x\_train,y\_train,batch\_size=90,nb\_epoch=4, validation\_split=0.05)

print (i)

return predicted\_data

**<8.5> Graphical analysis**

We have calculated the difference between predicted data and actual data for each point in the testing dataset to create an error dataset. From there, we have set a safe margin above which the value is flagged as an anomaly. From the plotted graph shown below, we get a conclusive forecast analysis of the LSTM model.

for i in range(len(predicted\_data)):

error = (abs(predicted\_data[i]-y\_test[i]))

errordata.append(error)

if (error > 0.2):

x.append(i)

y.append(y\_test[i])

mse = mean\_squared\_error(predicted\_data, y\_test)

rmse = sqrt(mse)

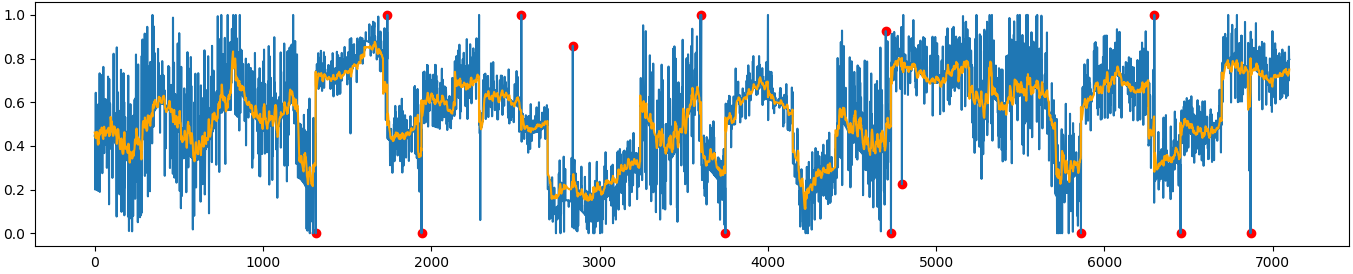
print('RMSE: %f'%rmse)

pyplot.plot(y\_test,label='Actual',color='blue')

pyplot.plot(predicted\_data,label='Predicted',color='orange')

pyplot.scatter(x, y,label ="Anomalies",color='red')

pyplot.show()



In the above graph, the expected values of test dataset are in blue, the values of predicted dataset are in orange and the flagged anomalies are shown as red dots.

Confusion Matrix and measurements:

|  |  |  |  |
| --- | --- | --- | --- |
| *Total(N) = 7100* | *Predicted Normal* | *Predicted Anomaly* |  |
| *Actual Normal* | 7082 (TN) | 9 (FP) | 7091 |
| *Actual Anomaly* | 4 (FN) | 5 (TP) | 9 |
|  | 7086 | 14 |  |

**<9> Investigating HTM**

**<9.1> A brief introduction**

Hierarchical temporal memory (HTM) is a biologically constrained theory (or model) of intelligence, originally described in the 2004 book *On Intelligence* by Jeff Hawkins with Sandra Blakeslee. HTM is based on neuroscience and the physiology and interaction of pyramidal neurons in the neocortex of the mammalian (in particular, human) brain.

At the core of HTM are learning algorithms that can store, learn, infer and recall high-order sequences. Unlike most other machine learning methods, HTM learns (in an unsupervised fashion) time-based patterns in unlabeled data on a continuous basis. HTM is robust to noise, and it has high capacity, meaning that it can learn multiple patterns simultaneously. When applied to computers, HTM is well suited for prediction, anomaly detection, classification and ultimately sensorimotor applications.

**<9.2> Methodology**

HTM learning algorithms, often referred to as cortical learning algorithms (CLA). It relies on a data structure called sparse distributed representations (that is, a data structure whose elements are binary, 1 or 0, and whose number of 1 bits is small compared to the number of 0 bits) to represent the brain activity and a more biologically-realistic neuron model (often also referred to as cell, in the context of the HTM theory). There are two core components in this HTM theory: a spatial pooling algorithm, which outputs sparse distributed representations (SDR), and a sequence memory algorithm, which learns to represent and predict complex sequences.

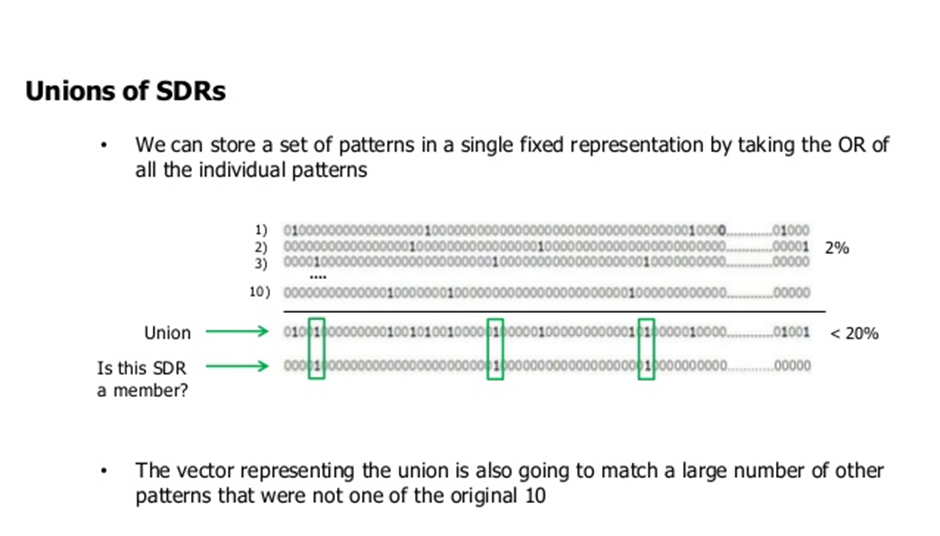
In this new generation, the layers and minicolumns of the cerebral cortex are addressed and partially modeled. Each HTM layer (not to be confused with an HTM level of an HTM hierarchy, as described [above](https://en.wikipedia.org/wiki/Hierarchical_temporal_memory#HTM_structure_and_algorithms)) consists of a number of highly interconnected minicolumns. An HTM layer creates a sparse distributed representation from its input, so that a fixed percentage of *minicolumns* are active at any one time. A minicolumn is understood as a group of cells that have the same receptive field. Each minicolumn has a number of cells that are able to remember several previous states. A cell can be in one of three states: *active*, *inactive* and *predictive* state.

**<9.2.1> Sparse Distributed Representations**

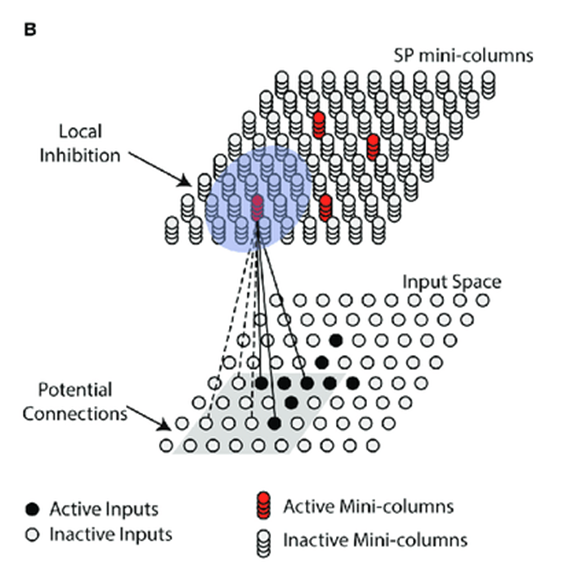
Integrating memory component with neural networks has a long history dating back to early research in distributed representations and self-organizing maps. For example, in sparse distributed memory (SDM), the patterns encoded by neural networks are used as memory addresses for content-addressable memory, with "neurons" essentially serving as address encoders and decoders.

Computers store information in "dense" representations such as a 32 bit word where all combinations of 1s and 0s are possible. By contrast, brains use sparse distributed representations (SDR). The human neocortex has roughly 16 billion neurons, but at any given time only a small percent are active. The activity of neurons are like bits in a computer, and therefore the representation is sparse.

The SDRs used in HTM are binary representations of data consisting of many bits with a small percentage of the bits active (1s); a typical implementation might have 2048 columns and 64K artificial neurons where as few as 40 might be active at once. Although it may seem less efficient for the majority of bits to go "unused" in any given representation, SDRs have two major advantages over traditional dense representations. First, SDRs are tolerant of corruption and ambiguity due to the meaning of the representation being shared (*distributed*) across a small percentage (*sparse*) of active bits. In a dense representation, flipping a single bit completely changes the meaning, while in an SDR a single bit may not affect the overall meaning much. This leads to the second advantage of SDRs: because the meaning of a representation is distributed across all active bits, similarity between two representations can be used as a measure of [semantic](https://en.wikipedia.org/wiki/Semantic) similarity in the objects they represent. That is, if two vectors in an SDR have 1s in the same position, then they are semantically similar in that attribute. The bits in SDRs have semantic meaning, and that meaning is distributed across the bits.



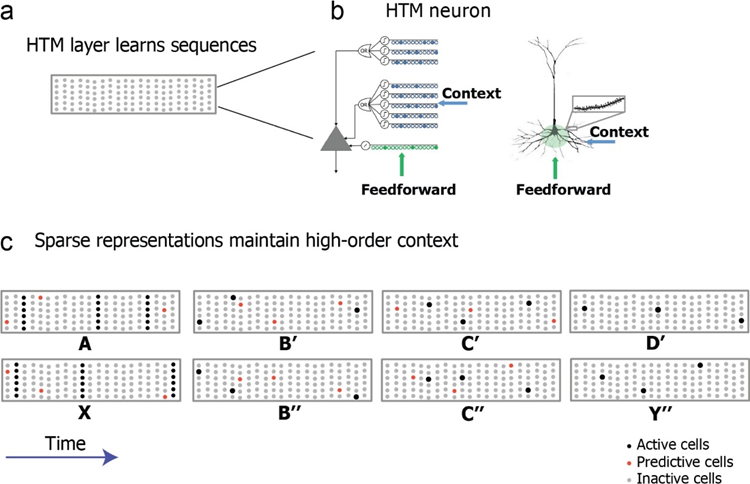
**<9.2.1.1> Spatial pooling**

The receptive field of each minicolumn is a fixed number of inputs that are randomly selected from a much larger number of node inputs. Based on the (specific) input pattern, some minicolumns will be more or less associated with the active input values. **Spatial pooling** selects a relatively constant number of the most active minicolumns and inactivates (inhibits) other minicolumns in the vicinity of the active ones. Similar input patterns tend to activate a stable set of minicolumns. The amount of memory used by each layer can be increased to learn more complex spatial patterns or decreased to learn simpler patterns.

**<9.2.1.2> Active, inactive and predictive cells**

As mentioned above, a cell (or a neuron) of a minicolumn, at any point in time, can be in an active, inactive or predictive state. Initially, cells are inactive.

**<9.2.1.3> How do cells become active?**

If one or more cells in the active minicolumn are in the *predictive* state, they will be the only cells to become active in the current time step. If none of the cells in the active minicolumn are in the predictive state (which happens during the initial time step or when the activation of this minicolumn was not expected), all cells are made active.

**<8.2.1.4> How do cells become predictive?**

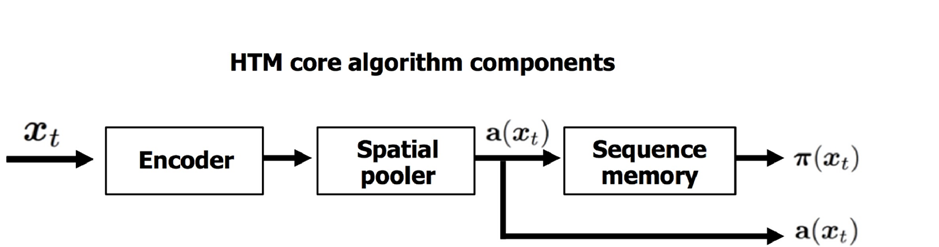
When a cell becomes active, it gradually forms connections to nearby cells that tend to be active during several previous time steps. Thus a cell learns to recognize a known sequence by checking whether the connected cells are active. If a large number of connected cells are active, this cell switches to the *predictive* state in anticipation of one of the few next inputs of the sequence.

**<9.2.1.5> The output of a minicolumn**

The output of a layer includes minicolumns in both active and predictive states. Thus minicolumns are active over longer periods of time, which leads to greater temporal stability seen by the parent layer.

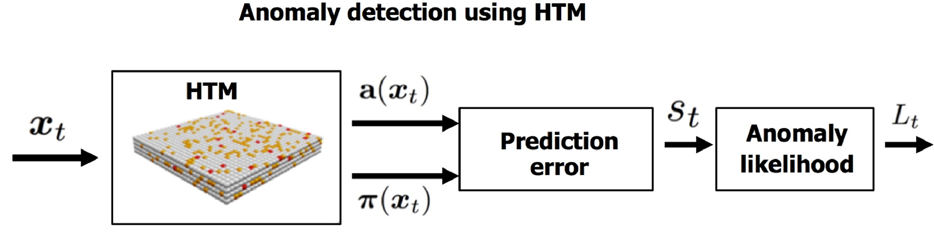
**<9.2.2> Inference and online learning**

Cortical learning algorithms are able to learn continuously from each new input pattern, therefore no separate inference mode is necessary. During inference, HTM tries to match the stream of inputs to fragments of previously learned sequences. This allows each HTM layer to be constantly predicting the likely continuation of the recognized sequences. The index of the predicted sequence is the output of the layer. Since predictions tend to change less frequently than the input patterns, this leads to increasing temporal stability of the output in higher hierarchy levels. Prediction also helps to fill in missing patterns in the sequence and to interpret ambiguous data by biasing the system to infer what it predicted.



**<9.2.3> HTM for anomaly detection**

We use the online learning of HTM to make predictions time-steps ahead and if the prediction differs from the actual value, we estimate a raw anomaly score. Finally we estimate an anomaly likelihood of the anomaly score based on the anomaly scores for a previous time window. If that likelihood score exceeds a pre-determined confidence interval then it is assumed that there is a considerable anomaly at that timestamp.

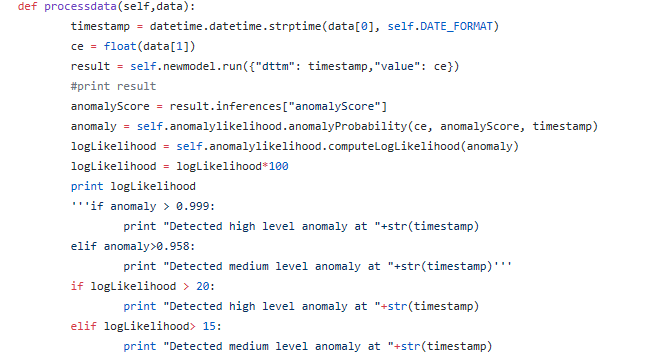


**<9.3> Implementation**

The following is a walkthrough of application of the aforementioned concept of HTM for anomaly detection:

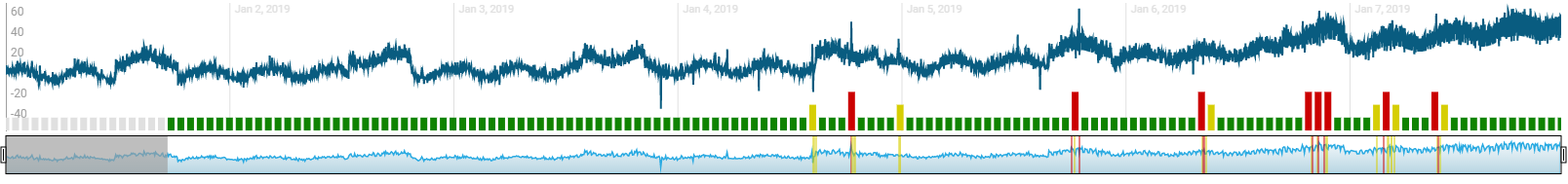


First we build the required model for predictions using pre-determined model parameters. We enable online learning and continuous prediction.



We then feed the model data one at a time as a tuple of (timestamp, value), and simultaneously retrieve the anomaly score as an output. Next we compute the Logarithmic Likelihood of the anomaly score retrieved and scale it on a scale of 100. Finally, we take a pre-determined confidence that if the anomaly likelihood exceeds 20 then it is a highly likely anomaly else if it is within 15 then it is moderately likely anomaly. Else it is not an anomaly.

**<9.4> Graphical analysis**



From the plot, 8 anomalies and 6 moderately likely anomalies have been reported. The blue line shown is the actual data and the red bars shown stands for Highly likely anomalies whereas Yellow bar shows the Moderately likely anomalies. Green bars are normal datapoints and the grey bars are initial phase when HTM tries to learn the trend and is not capable of drawing any inference.

Confusion Matrix and measurements:

|  |  |  |  |
| --- | --- | --- | --- |
| *Total(N) = 10000* | *Predicted Normal* | *Predicted Anomaly* |  |
| *Actual Normal* | 9986 (TN) | 3 (FP) | 9989 |
| *Actual Anomaly* | 1 (FN) | 10 (TP) | 11 |
|  | 9987 | 13 |  |

**<10> Conclusions**

|  |  |  |  |
| --- | --- | --- | --- |
| *Model* | *HTM* | *LSTM* | *ARIMA* |
| *Accuracy* | 0.9996 | 0.9982 | 0.9980 |
| *Sensitivity* | 0.909 | 0.5555 | 1 |
| *Precision* | 0.7692 | 0.3571 | 0.2778 |
| *False Positive Rate* | 0.0003 | 0.0012 | 0.0019 |
| *Misclassification Rate* | 0.0004 | 0.0018 | 0.0019 |
| *Total Execution Time* |  |  | 33.78 seconds |

From the Observations and Measurements, it is clearly evident that:

* **In case of non-stationary real-time streaming data, HTM is the best choice for Anomaly detection as it predicts the fastest point by point and adapts to the concept drift as evident from the minimum false positives.**
* **LSTM is a weaker choice when the nature of trend in the time series is quickly changing**
* **ARIMA is capable of quickly identifying outliers but fails to recognize the seasonal variations.**

Our expectations from the best model were:

1. Predictions must be made online, i.e., the algorithm must identify state as normal or anomalous before receiving the subsequent state
   1. The model must learn continuously without a requirement to store the entire stream.
   2. The model must run in an automated fashion, i.e., without data labels or manual parameter tweaking.
2. Model must adapt to dynamic environments and concept drift, as the underlying statistics of the data stream is often non-stationary.
3. Model should make anomaly detections as early as possible.

Compared to our expectations, **HTM satisfies all the expectations at its best**. As is verified from our above drawn conclusions, we can see:

* **HTM is capable of predicting online (point by point).**
* **HTM gives the least false positive in our synthetic data where the concept drifts had been intentionally incorporated. So it is clearly evident that it adapts to the concept drifts and seasonal variations.**
* **HTM provides the fastest predictions.**

Considering all the above conclusions and from the perspective of expectations satisfied, **HTM appears to be the most suitable model for the case study.**

**<11> Future Scope**

The following improvements could have been made to this endeavour –

* The whole solution could have been served as a complete packaged solution with GUI, middleware and the algorithm as a backend along with the facility to take streaming data as an input and the anomaly detected as a notification at output.
* The LSTM used is an open source Keras implementation. Our own version of LSTM could have been used with necessary changes in gradient descent algorithm for better comparison. Also changing hyper-parameters of LSTM might have yielded better results.
* ARIMA could have been fine-tuned.
* HTM by NuPIC used here is licensed product when used for production purposes. To cut down cost, own HTM could have been tried for implementation.

The above improvements became subject to limited time constraints however they can be the future scope of the project.

**<12> References**

The following papers and resources were used for the research purposes in this project –