Analytics on streaming data from Environmental Sensors

Rajrup Ghosh*, Shib Shankar Das †
*Department of Computational and Data Sciences
†Electronics Communicaton Engineering
Indian Institute of Science, Bangalore India
Email: rajrup.withbestwishes@gmail.com*, shibsankar.tech@gmail.com†

***Abstract*—Growth of the internet, Lead to Cloud Computing, Mobile Network and Internet of Things increasing rapidly, big data is becoming a hot-spot in recent years. Data generated from many sources made huge demand for storing, managing, processing and querying on various stream. With the advent of environmental sensors and high velocity data constantly streaming from them, scalable real-time analytics like mean, variance, higher order moments for each day statistical collection requires frameworks like Apache Storm. In this work we have implemented these simple statistical analysis on environmental sensor data obtained from different cities over the world. We have also given a real-time forecasting using ARIMA model for weather parameters like, temperature and humidity.**

I. Introduction

Information technology research has made significant advances over the past several years in cyberinfrastructure architectures for computational science investigation. These infrastructures, often assembled as loose collections of services, enable new forms of scientific investigation. The meteorology community is on the leading edge in motivating adaptivecyberinfrastructure research by means of use cases that involve the sensing and recognition of different weather phenomenon, and associated complex parallel and distributed analysis sequences that are invoked in response. On the otherhand with the rise in pollution becoming a global concern these days, finding out pollution hot spots, prediction of concentration levels of different pollutants is hugely required. These requirements can be met these days with the increasing availability of sensors and sensor technologies. This also poses a growing concern in terms of the velocity of sensor data produced each second and to do real-time analytics on top of the data being produced. This kind of high velocity data can be called as streams.

To address this high velocity data concern recent developments have been done to come up with scalable frameworks for doing real time data processing on streams. Real time data processing is quite difficult in batch systems like Apache Hadoop where a sufficient number of events or streams are queued for a map reduce job performing analytics on top of them, but incurs a huge I/O overhead to meet real-time deadlines. The main purpose of real time data processing and stream is to realize an entire system that can process mesh data in short time. The truth is the value of the Internet of things only comes from the astounding mass of data coming from different stream sources like sensors, mobiles, social media, health appliances and many more. It has been widely used in variety of fields such as Big Science, RFID, social networks, and internet search indexing astronomy and so on.

The Social network Facebook and twitter serve billions of page views per day and have billions of article and photos to store. The traditional application of data processing is distributed system. Stream processing is a technology that allows for the collection, integration, analysis, visualization, and system integration of data, all in real time, as the data is being produced, and without disrupting the activity of existing sources, storage, and enterprise systems. The Advantage of this system is high fault tolerance, low-Cost, high reliability, high extensibility and high efficiency.

Many modern data processing environments require processing complex computation on streaming data in real-time. This is particularly true for streaming applications like weather monitoring, pollutant concentration monitoring, Twitter streams where the use cases form complex decision making and real-time results of those use cases. Apache Storm [1] is such a real-time distributed stream data processing engine that powers the real-time stream data management tasks that are crucial to provide scalable stream analytics services.

We have performed different statistical analysis on some sensors like temperature, humidity and dust measuring sensors deployed for smart city data challenges held by Data Canvas [2]. In this project we have studied changes in weather parameters such as temperature and humidity across a period of time for different cites. Air pollutant like dust is taken into consideration for measuring the pollution level at particular location of a city. We have implemented a forecasting model like ARIMA to perform real-time forecast on temperature, humidity and dust concentration after each time interval. Such real-time forecasting may be of great use in different sectors of science, technology and business.

II. Apache Storm

There have been many new frameworks for stream processing like Aurora, Borealis, Apache Storm, Apache Samza, Google MillWheel, Amazon Kinesis and many more. Storm is a real-time distributed stream data processing engine at Twitter that powers the real-time stream data management tasks that are crucial to provide Twitter services. Storm is designed to be:

- **Scalable**:The application manager can add or remove nodes from the Storm cluster without disrupting existing data flows through Storm topologies.
- **Resilient**: Fault-tolerance is crucial to Storm as it is often deployed on large clusters, and hardware components can fail. The Storm cluster must continue processing existing topologies with a minimal performance impact
- **Extensible**: Storm topologies may call arbitrary external functions (e.g. looking up a MySQL service for the social graph), and thus needs a framework that allows extensibility.
- **Efficient**:Since Storm is used in real-time applications; it must have good performance characteristics. Storm uses a number of techniques, including keeping all its storage and computational data structures in memory.
- **Easy to Administer**: Since Storm is at that heart of user interactions on Twitter, end-users immediately notice if there are (failure or performance) issues associated with Storm. The operational team needs early warning tools and must be able to quickly point out the source of problems as they arise. Thus, easy-to-use administration tools are not a nice to have feature, but a critical part of the requirement.

The basic Storm data processing architecture consists of streams of tuples flowing through topologies. A topology is a directed graph where the vertices represent computation and the edges represent the data flow between the computation components. Vertices are further divided into two disjoint sets spouts and bolts. Spouts are tuple sources for the topology. Typical spouts pull data from queues, such as Kafka [3] or Kestrel. On the other hand, bolts process the incoming tuples and pass them to the next set of bolts downstream. Note that a Storm topology can have cycles. From the database systems perspective, one can think of a topology as a directed graph of operators.

Storm runs on a distributed cluster, and at Twitter often on another abstraction such as Mesos [4]. Clients submit topologies to a master node, which is called the Nimbus. Nimbus is responsible for distributing and coordinating the execution of the topology. The actual work is done on worker nodes. Each worker node runs one or more worker processes. At any point in time a single machine may have more than one worker processes, but each worker process is mapped to a single topology. Note more than one worker process on the same machine may be executing different part of the same topology. The high level architecture of Storm is shown in Figure 1.

Each worker process runs a JVM, in which it runs one or more executors. Executors are made of one or more tasks. The actual work for a bolt or a spout is done in the task. Data is shuffled from a producer spout/bolt to a consumer bolt (both producer and consumer may have multiple tasks). This shuffling is like the exchange operator in parallel databases. Storm supports the following types of partitioning strategies:

- **Shuffle** grouping, which randomly partitions the tuples.

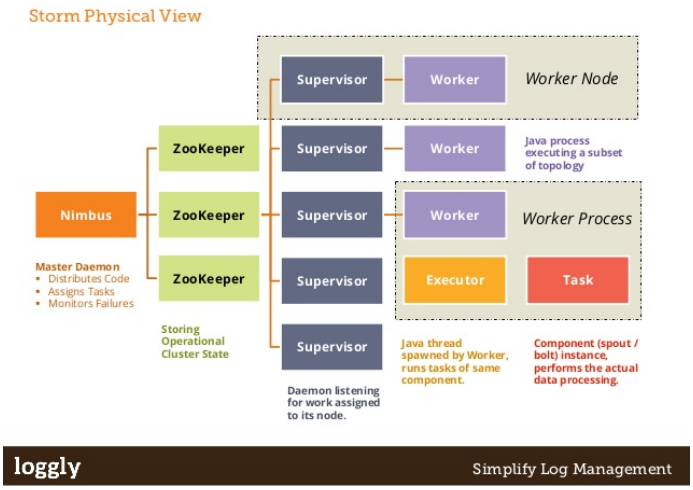

Fig. 1. Architecture of Storm

- **Fields** grouping, which hashes on a subset of the tuple attributes/fields.
- **All** grouping, which replicates the entire stream to all the consumer tasks.
- **Global** grouping, which sends the entire stream to a single bolt.
- **Local** grouping, which sends tuples to the consumer bolts in the same executor.

Each worker node runs a Supervisor that communicates with Nimbus. The cluster state is maintained in Zookeeper [5], and Nimbus is responsible for scheduling the topologies on the worker nodes and monitoring the progress of the tuples flowing through the topology. Loosely, a topology can be considered as a logical query plan from a database systems perspective. As a part of the topology, the programmer specifies how many instances of each spout and bolt must be spawned. Storm creates these instances and also creates the interconnections for the data flow.

One of the key characteristics of Storm is its ability to provide guarantees about the data that it processes. It provides two types of semantic guarantees "at least once," and "at most once" semantics. At least once semantics guarantees that each tuple that is input to the topology will be processed at least once. With at most once semantics, each tuple is either processed once, or dropped in the case of a failure. To provide "at least once" semantics, the topology is augmented with an "acker" bolt that tracks the directed acyclic graph of tuples for every tuple that is emitted by a spout.

Storm attaches a randomly generated 64-bit "message id" to each new tuple that flows through the system. This id is attached to the tuple in the spout that first pulls the tuple from some input source. New tuples can be produced when processing a tuple; e.g. a tuple that contains an entire Tweet is split by a bolt into a set of trending topics, producing one tuple per topic for the input tuple. Such new tuples are assigned a new random 64-bit id, and the list of the tuple ids is also

retained in a provenance tree that is associated with the output tuple. When a tuple finally leaves the topology, a backflow mechanism is used to acknowledge the tasks that contributed to that output tuple. This backflow mechanism eventually reaches the spout that started the tuple processing in the first place, at which point it can retire the tuple.

III. Methodology

A. Calculating Moments

Suppose a stream consists of elements chosen from a universal set. Assume the universal set is ordered so we can speak of the i^{th} element for any i. Let m_i be the number of occurrences of the i^{th} element for any i. Then the kth-order moment (or just k^{th} moment) of the stream is the sum over all i of $(m_i)^k$ [6].

There is no problem computing moments of any order if we can afford to keep in main memory a count for each element that appears in the stream. However, also as in that section, if we cannot afford to use that much memory, then we need to estimate the k^{th} moment by keeping a limited number of values in main memory and computing an estimate from these values. For the case of distinct elements, each of these values were counts of the longest tail produced by a single hash function. We shall see another form of value that is useful for second and higher moments.

The Alon-Matias-Szegedy Algorithm for Second Moments: Let a stream has a particular length n. Suppose we do not have enough space to count all the m_is for all the elements of the stream. We can still estimate the second moment of the stream using a limited amount of space; the more space we use, the more accurate the estimate will be. We compute some number of variables. For each variable X, we store:

1 A particular element of the universal set, which we refer to as $X.element$, and
2 An integer X.value, which is the value of the variable. To determine the value of a variable X, we choose a position in the stream between 1 and n, uniformly and at random. Set X.element to be the element found there, and initialize $X.value$ to 1. As we read the stream, add 1 to $X.value$ each time we encounter another occurrence of $X.element$.

Alon-Matias-Szegedy Algorithm

1 $n = 0$
2 $mean(0) = 0$;
3 $std(0) = 0$;
4 $randomInterval = 5$;
5 $memorycapacity = 10000$
6 $HashMapelementFrequency = null$;
7 $execute(tuplet)$
8 $n = n + 1$
9 $t = t.temperature$
10 $mean(n) = (mean(n-1) * (n-1) + t)/n$;
11 $std(n) = (std(n-1) * (n-1) + (t - mean(n))^2)/n$;
12 $If\ t\ is\ already\ there\ in\ map\ then\ do$
13 $elementfrequency++$
14 $Else$
15 $If(count \mod randomInterval == 0)$ $andsizeof(elementFrequency) < memorycapacity$
16 $add\ element\ to\ map\ and\ frequency = 1$
17 $EndIf$
18 $EndIf$
19 $Foreach\ element\ in\ map\ do$
20 $sumsecondmoment+ = n * (2 * t.frequency - 1)$
21 $sumthirdmoment+ = n * (3 * t.frequency^3 - 3 * t.frequency^2 + 1)$
22 $EndFor$
23 $2ndmoment = 2ndmoment/size(elementFrequency)$
24 $3rdmoment = 3rdmoment/size(elementFrequency)$

B. Storm Topology

The Storm Topology we are using in this project is as follows:

1. Spout: ReadSensorSpout: This spout is reading tuple from the dataset which is stored in HDFS and emitting tuple in a given frequency (let's say 50 samples per second).

2. Bolt: FilterDataBolt: We are using following 3 parameter from our dataset

- Temperature
- Humidity
- Dust

We have implemented Kalman filter for each parameter separately so that we can remove unexpected noise from the data. Temperature and Humidity data is not fluctuating much. So, Kalman filter is not much useful here for temperature and humidity. But as Dust has noise in the data that is removed significantly after filtering.

3. Bolt: CalculateParameterBolt: The bolt is receiving filtered data of Temperature,Humidity and Dust. Then we are estimating mean,standard deviation, second moment about 0 and third moment about 0. We have used Alon-Matias-Szegedy Algorithm for calculating estimation of second moment and third moment about 0.

C. Time Series Data Models

Time series models use the past movements of variables in order to predict their future values. Unlike structural models that relate the variable we want to forecast with a set of other variables, the time series model is not based on economic theory. However, in term of forecasting, the reliability of the estimated equation should be based on out-of-sample performance. The time series model can mostly produce quite accurate forecasts, especially in case that there are multi-dimensional relationships among variables. Because of the complexity of international economic relations, large structural models are likely to suffer from omitted variable bias, mis-specifications, simultaneous causality and other problems leading to substantial forecasting errors.

The time series model using Box-Jenkins approach has been proposed by Box and Jenkins [7]. This approach has been widely used in the literature because of its performance and

simplicity. Most time series can be described by Autoregressive Moving Average (ARMA) model. The stationary series Y_t is said to be $ARMA(p;q)$ if

$$Y_t - \phi_1 Y_{t-1} - ...\phi_p Y_{t-p} = \epsilon_t + \theta_1 \epsilon_{t-1} + ... + \theta_q \epsilon_{t-q} \quad (1)$$

where $_t$ is white noise and there is no common factor between autoregressive polynomial,$(1 - \phi_1 L - \phi_2 L^2 - ... - \phi_p L^p)$, and moving average polynomial, $(1 + \theta_1 L + ... + \theta_q L^q)$, where L is a lag operator. Also, these polynomials can be represented by $\phi(L)$ and $\theta(L)$, respectively. If the series is difference-stationary, the integrated autoregressive moving average (ARIMA) model is implemented. The series Y_t is said to be $ARIMA(p;d;q)$ if

$$\phi(L)(1-L)^d Y_t = \theta(L)\epsilon_t \quad (2)$$

where d is the d^{th} difference operator.

The ARIMA model will be extended into ARIMA model with explanatory variable (X), called $ARIMAX(p;d;q)$. Specifically, $ARIMAX(p;d;q)$ can be represented by

$$\phi(L)(1-L)^d Y_t = \theta(L)X_t + \theta(L)\epsilon_t \quad (3)$$

where X_t is trade partners CLI. Moreover, some export commodities, such as rice and agricultural goods, have seasonal feature. We employ seasonal ARIMA and seasonal ARIMAX model to capture seasonality.

The difference ARMA and ARIMA model is that ARIMA Model converts a non-stationary data to a stationary data before working on it. ARIMA model is widely used to predict linear time series data. The ARIMA models are often referred to as Box-Jenkins models as ARIMA approach was first popularized by Box and Jenkins. ARIMA model is often referred to as ARIMAX model when it includes other time series as input variables. The ARIMA procedure offers great flexibility in univariate time series model identification, parameter estimation, and forecasting.

ARIMA models are generally used to analyze time series data for better understanding and forecasting.Initially, the appropriate ARIMA model has to be identified for the particular datasets and the parameters should have smallest possible values such that it can analyze the data properly and forecast accordingly. [8] The Akaike Information Criteria (AIC) is a widely used measure of a statistical model. It is used to quantify the goodness of fit of the model. When comparing two or more models, the one with the lowest AIC is generally considered to be closer with real data. AICc is AIC with a correction for finite sample sizes.

The AIC does not penalize model complexity as heavily as the BIC (Bayesian Information Criterion) does. Burnham & Anderson shows that AIC and AICc and BIC all can be derived in the same framework and using AIC/AICc for model selection is theoretically proved to be more advantageous than using BIC for selecting a model. AIC is asymptotically optimal in selecting the model, under the assumption that the true model is not in the candidate set (as is virtually always the case in practice); BIC relies on the assumption that the true model is in the candidate set which makes it asymptotically less optimal.

Often for large datasets choice of parameters for the ARIMA model is quite difficult. It is required to perform tests for AIC, RMSE, MAPE for number of times with different p and q. To automate the process Rob J. Hyndman [9] came up auto ARIMA implementation which makes life of statistician and novice learner's task easier.

The algorithms and modelling frameworks for automatic univariate time series forecasting are implemented in the **fpp, forecast** package in R. The $auto.arima()$ function automatically calculates the seasonal trend and difference in the data. For forecasting we have integrated R with our JAVA code using JRI. The forecasting script is written in R, which has 2 parts: $Init\_Script.R$ and $Run\_Script.R$. The dataset has been divided into two parts - Training data for the model and Testing data for model accuracy. The choice of size of training data is a hyper-parameter to be decided. We trained the model in the prepare stage of a spout. In the $nextTuple()$ function we call the $Run\_Script.R$ to get a forecast after each time interval. After a time window as given by the user we retrain the model by incorporating the true test data for that time period. This method is called forecasting with re-estimation. As retraining may take a bit of latency, we forecast for the time window at once and enter the data into a queue from where the $nextTuple()$ function dequeue forecast one by one while the model gets trained for the previous window time. This makes the forecast implementation more flexible and scalable.

$Init\_Script.R$

```
> library(fpp)
> temperature <- scan(file = filepath)
> n <- length(temperature)
> t <- ts(temperature, start=0, deltat = 1)
> train <- window(t, end=end)
> test <- window(t, start=start)
> fit <- auto.arima(train)
> order <- arimaorder(fit)
```

$Run\_Script.R$

```
> library(fpp)
> fc <- ts(numeric(length(test)), start=start1, deltat = 1)
>for(wind in 1:num_retrain)
+{
+  x <- window(temp, end=end1 + wind_size*(wind-1))
+  refit <- Arima(x, model = fit)
+  temp_forecast <- forecast(refit, h=10)$mean
+  for(i in 1:wind_size)
+  {
+    fc[wind_size*(wind-1)+i] <- temp_forecast[i]
+  }
+}
> accuracy(test,fc)
```

IV. Experiments and Results

Based on the change in parameters like mean,standard deviance,second moments and third moments we can have a comparison between weather variation of two city. As we have shown there is significant change in mean of temperature for Bangalore and Singapore. Actual temperature vs predicted temperature graph clearly shows that the predicted temperature

TABLE I
ARIMA FORECAST TABLE

p	d	q	AIC	AICc	BIC
2	1	3	-274083.7	-274083.7	-274027.8

ME	RMSE	MAE	MPE	MAPE
0.001849033	0.124834	0.05049333	0.009608752	0.3100791

is lying on the actual temperature and the absolute difference between actual and predicted temperature is very less.

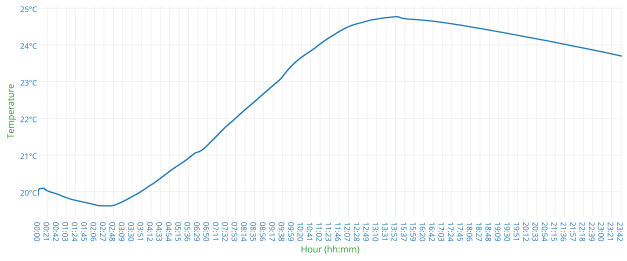

Fig. 2. Mean Temperature of Bangalore,Date:15/01/2015

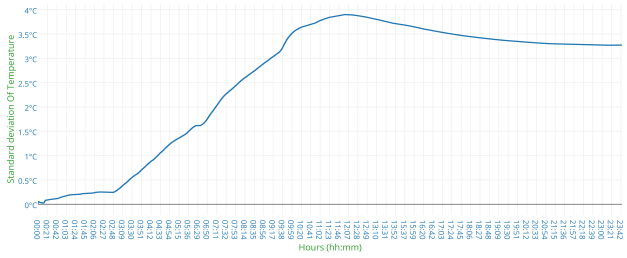

Fig. 3. Standard Deviation of Temperature of Bangalore,Date:15/01/2015

V. CONCLUSIONS

In this project we have learned real-time analytics over high velocity streaming data. Apache Storm scaled nicely with different input stream rates for performing statistical analysis on the sensor data. As we haven't come across any free licensed JAVA implementation of ARIMA, we used R packages which has some overhead on end to end latency for streaming applications. Even in cluster environment this type of R procedure calls require the availability of R in every machine with classpath set for all the required jars, which may become cumbersome for streaming applications to work. But it can be observed from the results obtained that forecasting for weather data worked very well using $auto.arima()$ with periodic retraining, giving error less than 1%. This work can be extended to perform extreme scalability tests for the statistical parameter estimation algorithms. Moreover, ARIMA can be implemented in JAVA so the strom application can better

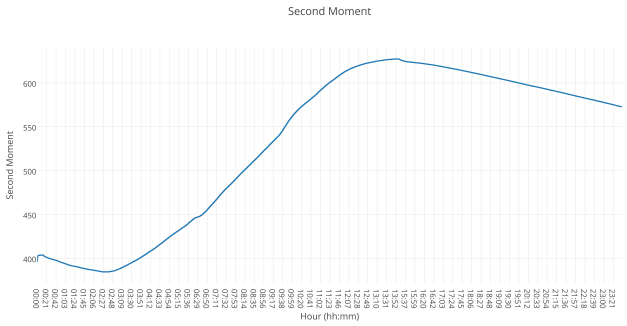

Fig. 4. Second moment about 0 of Temperature of Bangalore,Date:15/01/2015

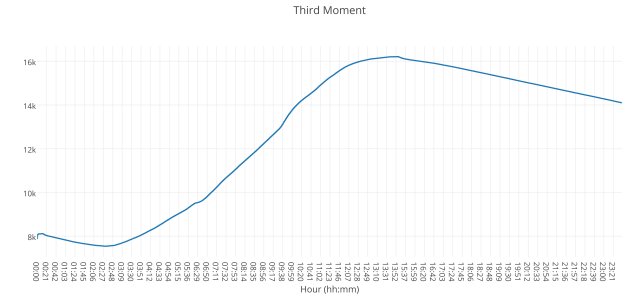

Fig. 5. Third moment about 0 of Temperature of Bangalore,Date:15/01/2015

scale with forecasting, decreasing the end to end latency as a whole. Another important study may involve comparison with different forecasting methods and how they scale with streaming applications. All these study has a huge application in weather stations and pollution monitoring centers for dealing with numerous sensor data and is of a growing importance in developing future SMART CITIES.

REFERENCES

[1] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm@twitter," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: ACM, 2014, pp. 147–156. [Online]. Available: http://doi.acm.org/10.1145/2588555.2595641

[2] D. Canvas, "Data Canvas Dataset," urlhttp://datacanvas.org/sense-your-city/, 2015–2016.

[3] N. Garg, *Apache Kafka*. Packt Publishing, 2013.

[4] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 295–308. [Online]. Available: http://dl.acm.org/citation.cfm?id=1972457.1972488

[5] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 11–11. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855840.1855851

[6] A. Rajaraman and J. D. Ullman, *Mining of Massive Datasets*. New York, NY, USA: Cambridge University Press, 2011.

[7] G. E. P. Box and G. Jenkins, *Time Series Analysis, Forecasting and Control*. Holden-Day, Incorporated, 1990.

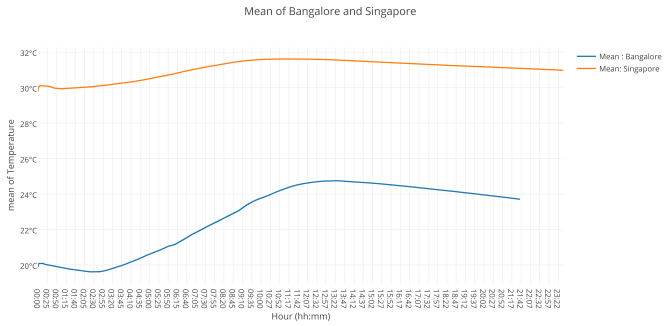

Fig. 6. Mean Temperature of Bangalore and Singapore

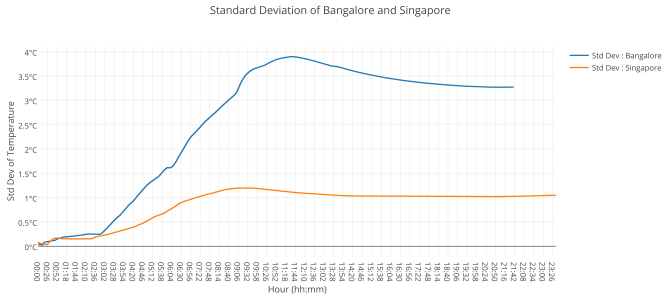

Fig. 7. Standard Deviation of Temperature of Bangalore and Singapore

[8] Y. Sakamoto, M. Ishiguro, and G. Kitagawa, *Akaike information criterion statistics*, ser. Mathematics and its applications. Japanese series. Tokyo: KTK Dordrecht, 1986, includes index. [Online]. Available: http://opac.inria.fr/record=b1135735

[9] R. J. Hyndman and Y. Khandakar, "Automatic time series forecasting: the forecast package for R," *Journal of Statistical Software*, vol. 26, no. 3, pp. 1–22, 2008. [Online]. Available: http://www.jstatsoft.org/article/view/v027i03

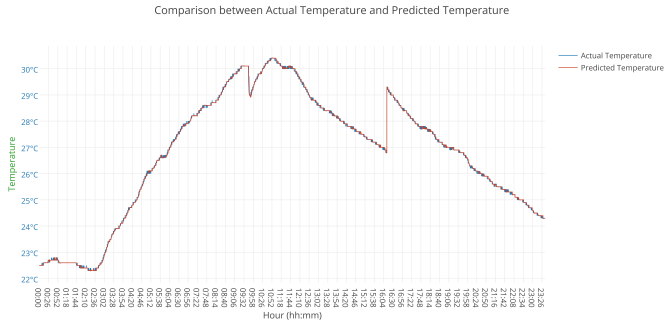

Fig. 8. Actual and Predicted Temperature of Bangalore

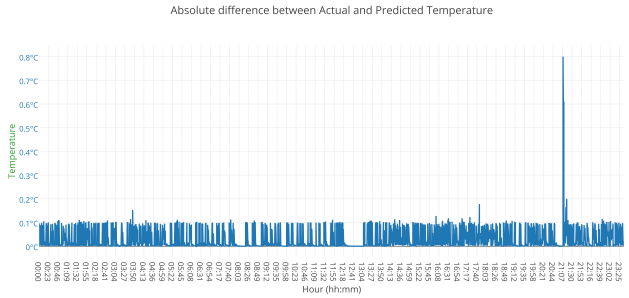

Fig. 9. Absolute difference between Actual and Predicgted temperature