Predictors for Trickling in WSN

Douglas Marcelino Beppler Martins

Lucas May Petry

Maike de Paula Santos

Motivation

Nowadays, the Internet of Things (IoT) is present in numerous kinds of devices, from residential automation to smart cities (INTERNET... 2017). In an IoT environment, the more devices there are, the bigger is the number of messages exchanged among them. This increase in communication can lead to several problems. One of these issues is the device's power consumption. Since many sensors make use of a battery as their power source, the excessive use of their wireless may drain a large portion of it (SOMASUNDARA et al., 2006). Moreover, a large amount of communication between devices implies a large spending in transmission infrastructure. Hence, tools and techniques that allow companies to reduce the previously mentioned factors become of great value.

Goals

In order to diminish the communication between a sensor and a gateway, we propose the implementation of SmartData predictors. The goal is to run the designed predictor on both the sensor and the gateway, so they talk less without losing information. Every time data becomes available on a sensor, the data will be sent to the gateway only if it differs from the predicted value, given a user-defined threshold. If that's not the case, the gateway will know the value beforehand via the predictor. Figure 1 below exemplifies the described environment.

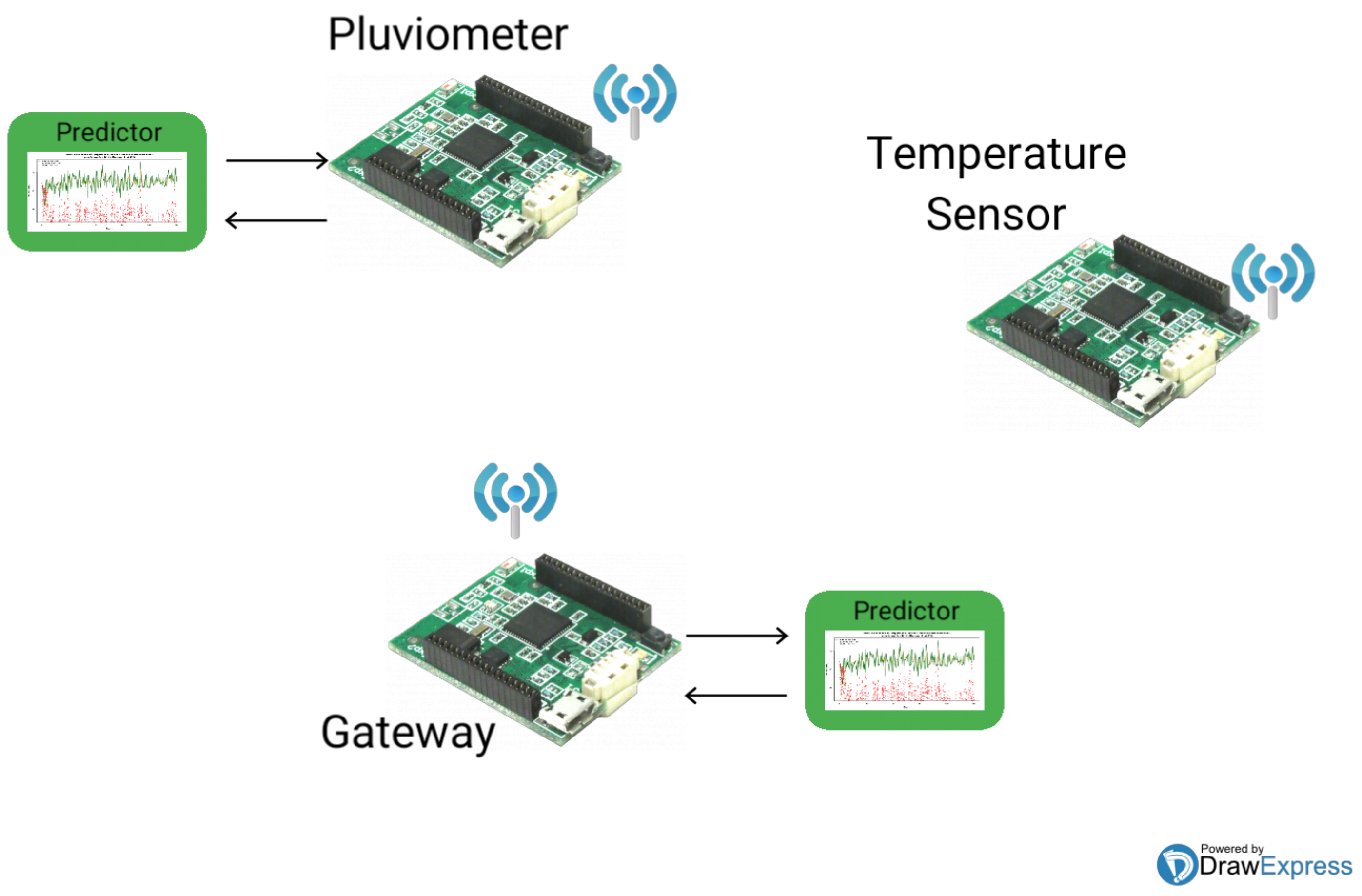


Figure 1 - Example of IoT environment with a predictive sensor.

Methodology

After studying the existing algorithms, we chose to program a linear regression with gradient descent (NEDRICH, 2014) and Elman Neural Networks (KOSKELA et al., 1997) as our predictors. We intend to prototype these algorithms in the C++ language for a general purpose operating system, so that we are able to test them with manual inputs and confirm the desired behavior. Subsequently, the algorithms will be ported to run on EPOSMote III, using real sensors to simulate a gateway-sensor environment and validate the efficiency of this approach. Initially, we will run tests in a controlled environment. Afterwards, data from the UFSC IoT Gateway will be used to demonstrate the proposed algorithms.

Tasks

1. Make project plan. ✔
2. Study system restrictions and limitations of EPOSMote III and elaborate implementation strategies, such as design patterns. ✔
3. Code and test the chosen prediction algorithms in C++ for a general purpose operating system. ✔
4. Code and test the **linear regression with gradient descent predictor** on EPOS. ✔

5. Code and test the **Elman neural network** on EPOS . ✔[1]
6. Deploy the final product using the UFSC IoT Gateway . ✔[1]
7. Present the final project and results. ✔

Deliverables

1. Project plan. ✔
2. Report on restrictions and limitations of embedded systems and the strategy used to tackle the problems. ✔
3. Code and test results of the prediction algorithms on a general purpose operating system. ✔
4. Code and test results of the **linear regression with gradient descent predictor** on EPOS . ✔
5. Code and test results of the **Elman neural network** on EPOS . ✔[1]
6. Deployed algorithm on UFSC IoT Gateway . ✔[1]
7. Project demonstration. ✔

Schedule

First week (W1): April 26th, 2017.
Last week (W10): June 28th, 2017.

Task	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10
1. Make project plan	X	D1								
2. Study system restrictions and limitations of EPOSMote III and elaborate implementation strategies, such as design patterns		X	D2							
3. Code and test the chosen prediction algorithms in C++ for a general purpose operating system			X	D3						
4. Code and test the **linear regression with gradient descent predictor** on EPOS				X	X	D4				
5. Code and test the **Elman neural network** on EPOS [1]						X	X	X	D5	
6. Deploy the final product on the UFSC IoT Gateway [1]									X	D6
7. Present the final project and results										D7

[1] See project changes

Changes & Issues

Changes

1. Due to the lack of material and information about Elman Neural Networks, in addition to the lack of support for it on the third-party software used to design and run the network, we decided to implement a **Multilayer Perceptron (MLP) Neural Network.**. Koskela et al. (1997) also describes the use of MLP networks for time series prediction. Furthermore, Multilayer Perceptron neural networks are very known by the scientific community and there is a lot of research and tools for dealing with them.
2. The second change in our project is related to task 6. After reaching out to UFSC personnel, we were informed that it would be difficult and not really feasible to deploy our product on the UFSC IoT Gateway . Instead, we collected data from the gateway at UFSC and simulated a real sensor.

Issues

1. The main issue that had been foreseen by us was the synchronization of predictors. A problem of communication or any other issue in the WSN could affect the synchronization of the predictors running on sensors and on the gateway. The first question was *how does the gateway know the real reason why it is not receiving data from the sensor?* We must be able to know at a certain point if the remote device has malfunctioned; or if a problem in the network happened and the message to the gateway was not delivered; or any other unexpected problem with communication. For that reason, we included a mechanism of synchronization in the implementation, so that the gateway knows the sensor is still "alive". This mechanism is further discussed in the project design section.
2. The second issue, not as important as the first one, was the lack of the exponential function in EPOS math library. Therefore, we had to implement the exponential function using *Taylor Series*.

EPOSMote III Restrictions and Project Design

Project Design

Our goal is to implement a **dynamic** linear regression predictor with gradient descent and a **static** MLP neural network predictor. When a gateway has interest on the data coming from a sensor and data is available, the sensor will broadcast the data to the network and it will reach the gateway. Therefore, we propose some modifications to the SmartData class and the design of a subclass **PredictiveSmartData**.

SmartData will implement the `send(Time t, Time_Offset expiry)` method, which essentially broadcasts the data to the network. The **PredictiveSmartData** will overwrite this method, checking the read value against the predictors' prediction and decide if data should be transmitted. The class diagram below summarizes the new classes and modifications to SmartData that are being proposed.

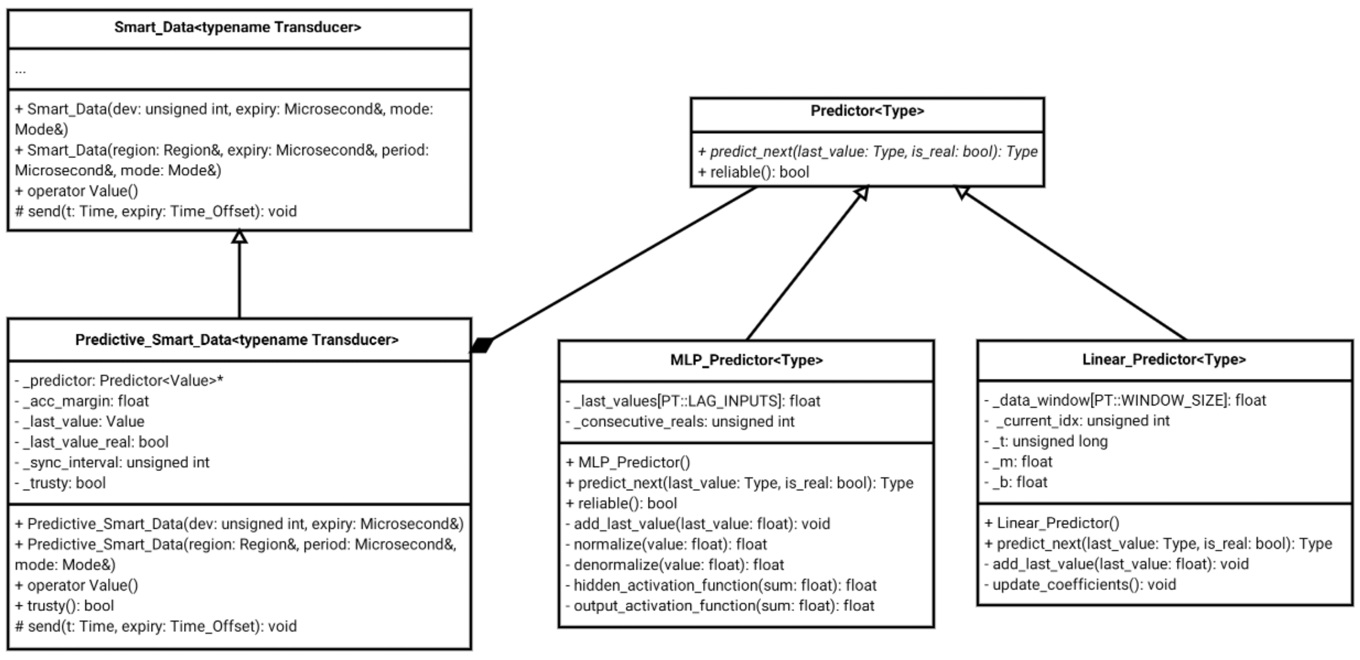

Smart_Data<typename Transducer>
...
+ Smart_Data(dev: unsigned int, expiry: Microsecond&, mode: Mode&)
+ Smart_Data(region: Region&, expiry: Microsecond&, period: Microsecond&, mode: Mode&)
+ operator Value()
# send(t: Time, expiry: Time_Offset): void
Predictor<Type>
+ predict_next(last_value: Type, is_real: bool): Type
+ reliable(): bool
Predictive_Smart_Data<typename Transducer>
- _predictor: Predictor<Value>*
- _acc_margin: float
- _last_value: Value
- _last_value_real: bool
- _sync_interval: unsigned int
- _trusty: bool
+ Predictive_Smart_Data(dev: unsigned int, expiry: Microsecond&)
+ Predictive_Smart_Data(region: Region&, period: Microsecond&, mode: Mode&)
+ operator Value()
+ trusty(): bool
# send(t: Time, expiry: Time_Offset): void
MLP_Predictor<Type>
- _last_values[PT::LAG_INPUTS]: float
- _consecutive_reals: unsigned int
+ MLP_Predictor()
+ predict_next(last_value: Type, is_real: bool): Type
+ reliable(): bool
- add_last_value(last_value: float): void
- normalize(value: float): float
- denormalize(value: float): float
- hidden_activation_function(sum: float): float
- output_activation_function(sum: float): float
Linear_Predictor<Type>
- _data_window[PT::WINDOW_SIZE]: float
- _current_idx: unsigned int
- _t: unsigned long
- _m: float
- _b: float
+ Linear_Predictor()
+ predict_next(last_value: Type, is_real: bool): Type
- add_last_value(last_value: float): void
- update_coefficients(): void

Figure 2 - Class diagram.

PredictiveSmartData encapsulates a **Predictor**. Any new predictor to be developed can be easily attached to EPOS by implementing the **Predictor** interface methods `predict_next(float last_value, bool is_real)` and `reliable()`. The first method is called by **PredictiveSmartData** to request a prediction. The latter is the reliability of the predictor disregarding any other information outside the predictor. This method is used by **PredictiveSmartData** to recover from a desynchronization between gateway and sensor.

Whenever a **PredictiveSmartData** is instantiated on the sensor-side, the set operation mode will always be `ADVERTISED`, given that we're only interested in sensors being monitored by the gateway. *_acc_margin* is a user-defined parameter in **PredictiveSmartData**, meaning how acceptable a predicted value *y* is, given the value *x* read from the sensor. *y* is acceptable if it lies between *x* - *_acc_margin* and *x* + *_acc_margin*. We solve the synchronization issue described earlier by forcing a synchronization interval (attribute *_sync_interval*). *_sync_interval* is the maximum number of consecutive correctly predicted values without transmitting data to the gateway. After that period, regardless of what the prediction on the sensor-side was, data will be sent to the gateway. Hence, the gateway knows when he is supposed to receive data and sets *_trusty* to false if it did not, meaning that predicted data is not trustful anymore.

The predictive smart data, linear regression predictor and MLP predictor parameters are described below. Some of them are not shown in the class diagram because they sit on Traits files.

Predictive Smart Data Parameters

- `ACC_MARGIN` : the acceptance margin for the predicted data to be considered;
- `PREDICTOR` : the predictor used by the Predictive Smart Data. It can be `LINEAR` or `MLP` ;
- `SYNC_INTERVAL` : the synchronization interval.

Linear Regression Predictor Parameters

- `WINDOW_SIZE` : number of data points that will be considered to update the coefficients `M` and `B` ;
- `LRATE` : learning rate of the gradient descent algorithm;
- `GD_ITERATIONS` : stop criteria for the gradient descent algorithm;
- `M` and `B` : initial coefficients of the linear function *y* = *mx* + *b*.

MLP Neural Network Predictor Parameters

- `HIDDEN_UNITS` : number of hidden units in the network;
- `LAG_INPUTS` : number of inputs of the network;
- `HIDDEN_WEIGHTS` : weights from inputs to the hidden units;
- `HIDDEN_BIASES` : biases to the hidden units;
- `OUTPUT_WEIGHTS` : weights from hidden units to the output unit;
- `OUTPUT_BIAS` : bias to the output unit;
- `NORMALIZATION` : flag indicating if data should be normalized or not;
- `NORMALIZATION_MIN` : the minimum value for normalization;
- `NORMALIZATION_MAX` : the maximum value for normalization.

Figure 3 shows the overall operation on sensor and gateway (implementation details were abstracted).

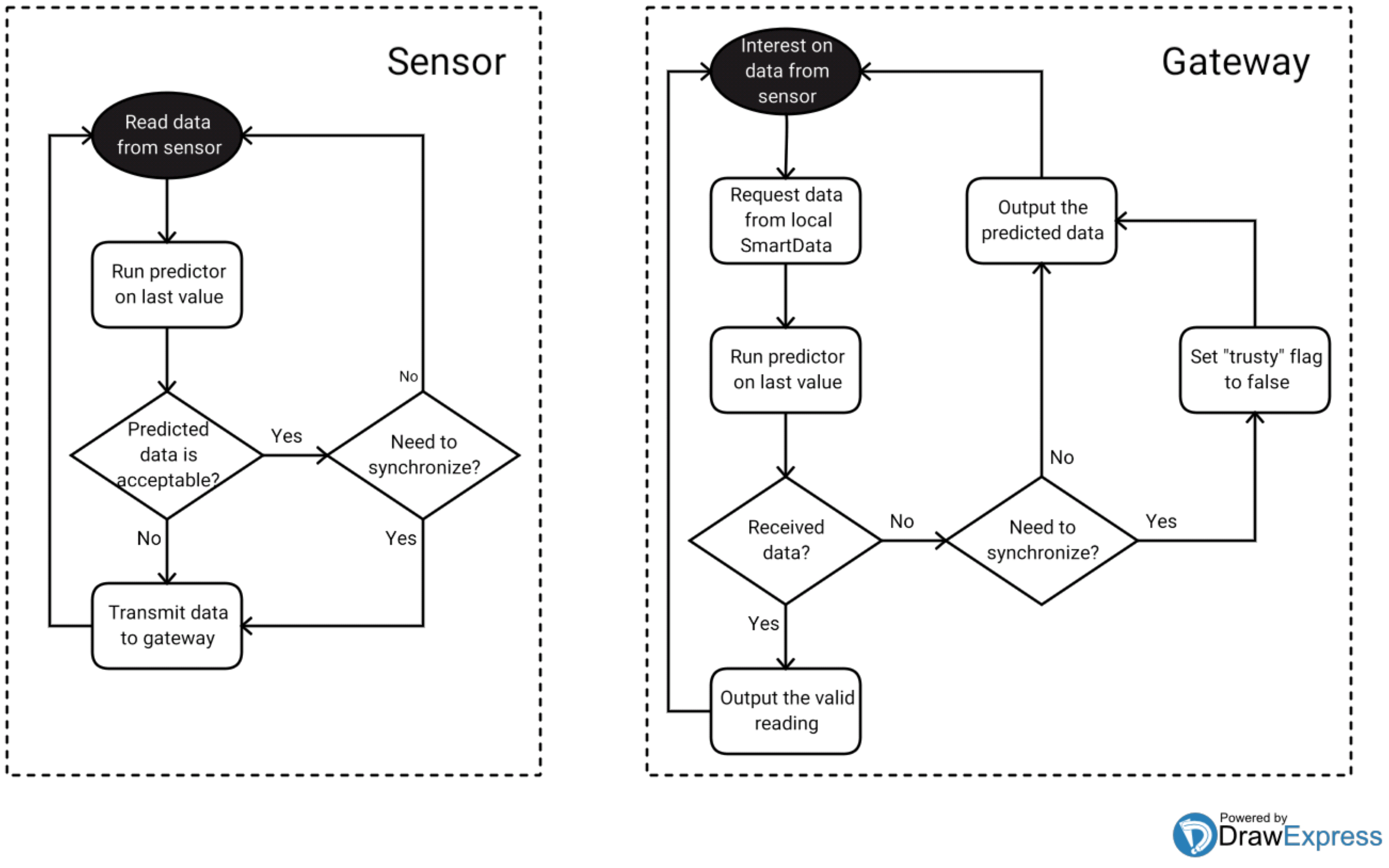

Sensor
Read data from sensor
Run predictor on last value
Predicted data is acceptable?
Yes
No
Need to synchronize?
No
Yes
Transmit data to gateway
Gateway
Interest on data from sensor
Request data from local SmartData
Run predictor on last value
Received data?
No
Yes
Need to synchronize?
No
Yes
Output the valid reading
Output the predicted data
Set "trusty" flag to false
Powered by DrawExpress

Figure 3 - Flow chart of sensor and gateway operation.

Hardware Restrictions

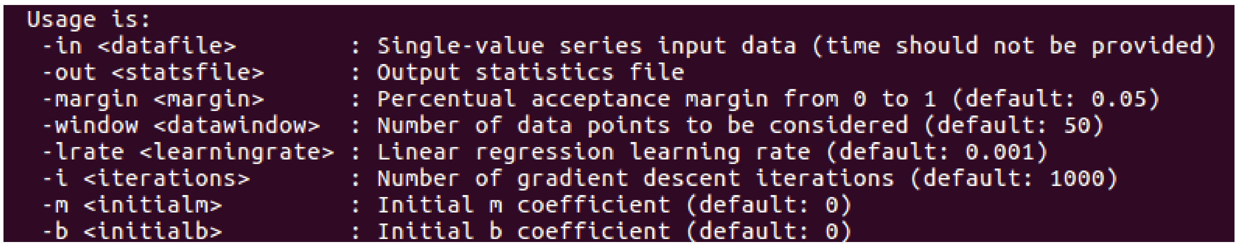
The most concerning hardware configuration for our project is ROM and RAM, and CPU clock speed. EPOSMote III provides the following configuration:

- 512Kbyte, 256Kbyte or 128Kbyte flash memory (ROM);
- 32Kbyte RAM;
- Up to 32-MHz clock speed.

Due to the fact that the linear predictor is dynamic, we'll need to store some parameters and a limited number of data values in RAM. On the other hand, the parameters used by the MLP predictor can be stored in ROM.

Code and Test Results of the Linear Regression with Gradient Descent Predictor

The linear regression predictor was implemented and is at revision 4572 on the SVN server . It was implemented with gradient descent so the underlying function representing the data to be predicted is adjusted as more data becomes available. The code and interface developed can be useful for simulating the predictor behavior on a dataset before flashing it to EPOSMote III . A simple guide to using the program can be found by typing `./linear -h`.



```
Usage is:
  -in <datafile>         : Single-value series input data (time should not be provided)
  -out <statsfile>       : Output statistics file
  -margin <margin>       : Percentual acceptance margin from 0 to 1 (default: 0.05)
  -window <datawindow>   : Number of data points to be considered (default: 50)
  -lrate <learningrate>  : Linear regression learning rate (default: 0.001)
  -i <iterations>        : Number of gradient descent iterations (default: 1000)
  -m <initialm>          : Initial m coefficient (default: 0)
  -b <initialb>          : Initial b coefficient (default: 0)
```

Figure 4 - Usage of linear regression predictor.

We ran tests on Ubuntu over two different datasets.

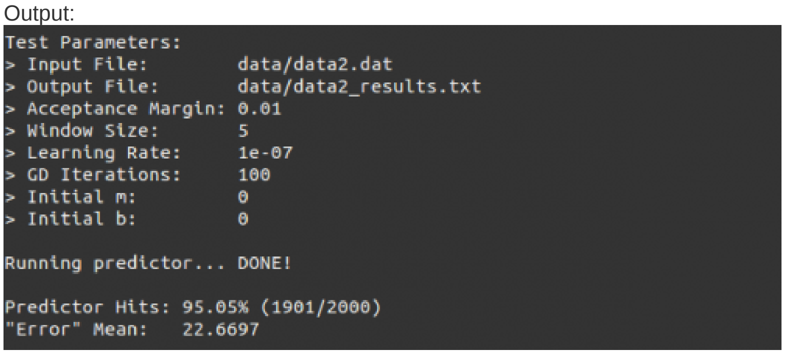
Dataset 1: Linear Function y = 3x + 7

We created a dataset of 1000 points corresponding to the function `y = 3x + 7` evaluated from 1 to 1000. After trying out different sets of parameters, we achieved excellent results running the folowing command (and parameters).

Command:

```
./linear -in data/data2.dat -out data/data2_results.txt -margin 0.01 -window 5 -lrate 0.0000001 -i 100 -m 0 -b 0
```

Output:



```
Test Parameters:
> Input File:        data/data2.dat
> Output File:       data/data2_results.txt
> Acceptance Margin: 0.01
> Window Size:       5
> Learning Rate:     1e-07
> GD Iterations:     100
> Initial m:         0
> Initial b:         0

Running predictor... DONE!

Predictor Hits: 95.05% (1901/2000)
"Error" Mean:    22.6697
```

Figure 5 - Output of linear regression predictor tests for a linear function.

Additionally, we plotted the results to a graph for visualization purposes, using the R language:

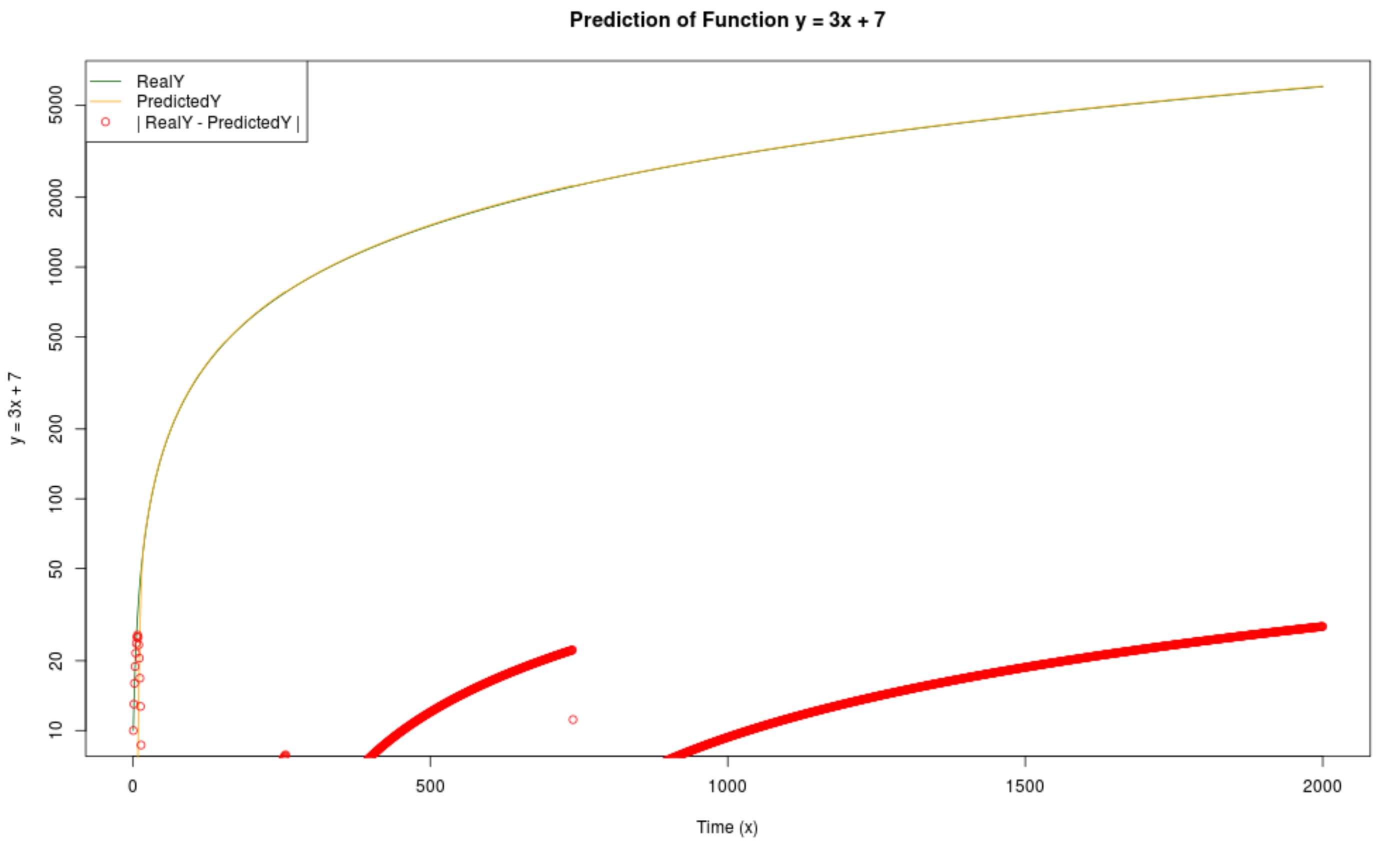


Figure 6 - Plotted results of linear regression predictor tests for a linear function.

As expected, the linear regression predicts inherently linear data very well.

Dataset 2: Hourly Air Temperature at SFO Airport

The second dataset comprised 26444 temperature values measured hourly at the San Francisco International Airport between January 1st, 2015 and December 31st, 2016 (NOAA, 2017). This dataset was chosen because our goal was to verify the predictor performance on real-world data. Once again, after a few trials we ended up with the command and parameters below.

Command:

```
./linear -in data/data1.dat -out data/data1_results.txt -margin 0.08 -window 30 -lrate 0.000000001 -i 200 -m 0 -b
0
```

Plotted results:

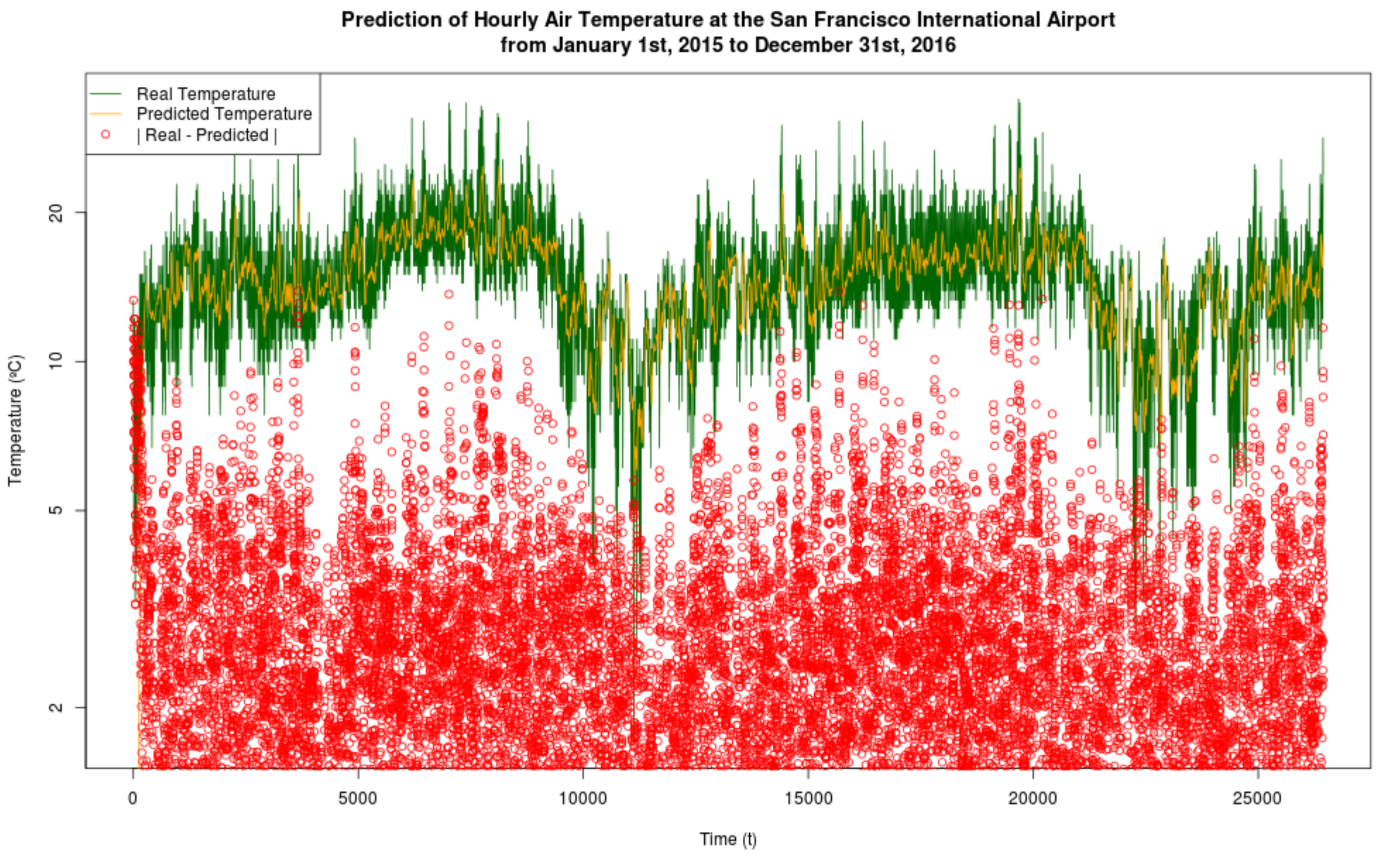


Figure 7 - Plotted results of linear regression predictor tests on temperature dataset.

To have a more detailed visualization, we also plotted the first 5000 points alone:

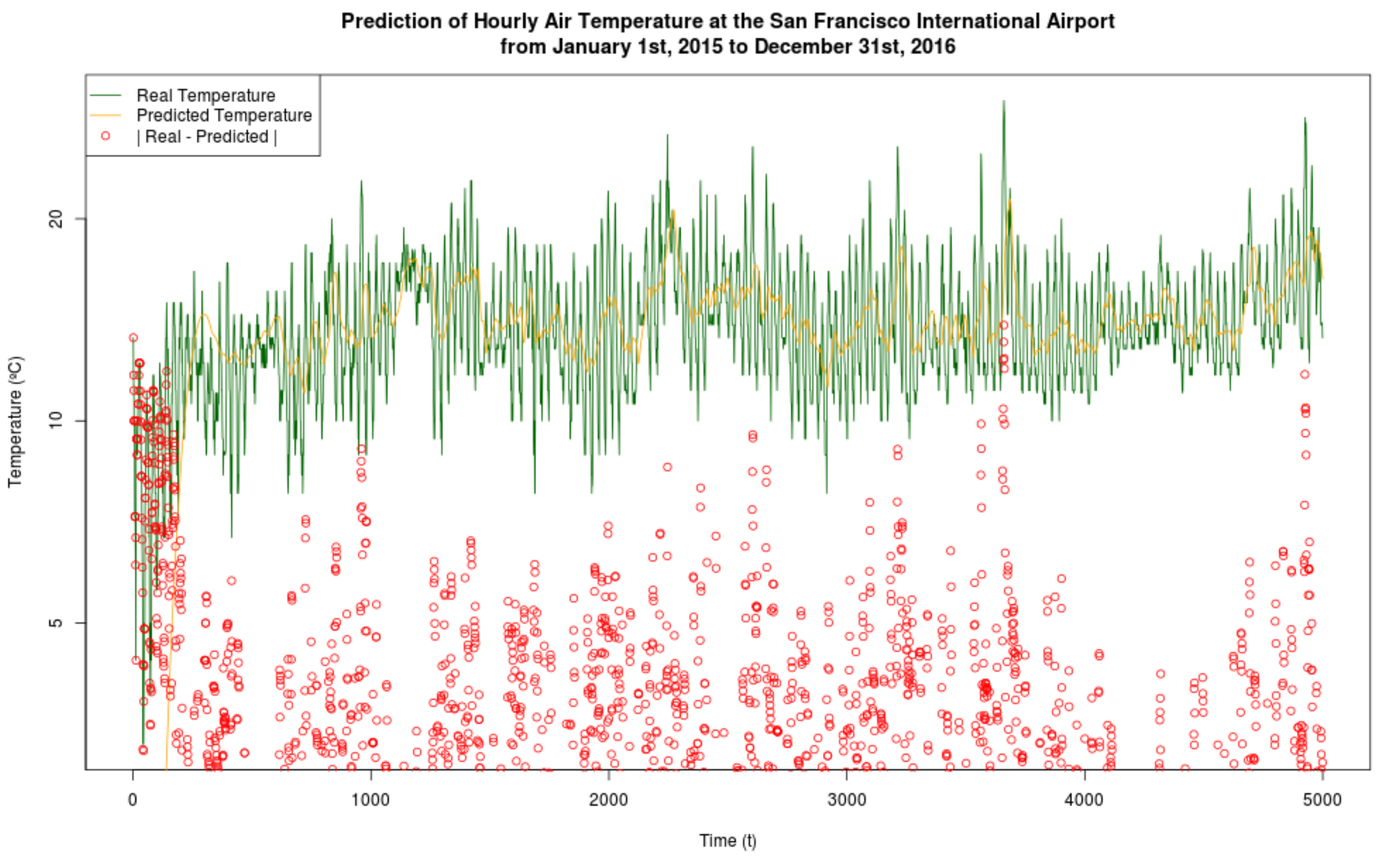


Figure 8 - First 5000 points of linear regression predictor tests on temperature dataset.

The parameter `-margin` is the percentage acceptance margin. More accurate predictions (restrictive margins) will result in less hits by the predictor. For this temperature dataset we tried these three different margins.

Margin: 5%

- Predictor Hits: 20.4092% (5397/26444)
- "Error" Mean: 2.08872

Margin: 8%

- Predictor Hits: 32.098% (8488/26444)
- "Error" Mean: 2.08895

Margin: 10%

- Predictor Hits: 39.9524% (10565/26444)
- "Error" Mean: 2.0918

The linear regression attempts to "fit" the data in a linear function. Temperature data, though, is not linear. Considering the 5% acceptance margin, our predictor was still able to correctly predict about 20% of all the data, which would roughly mean a 20% decrease in communication between sensors.

Code and Test Results of the Multilayer Perceptron Neural Network Predictor

The Multilayer Perceptron Neural Network predictor was implemented and is at revision 4572 on the SVN server. Based on the work of Koskela (1997), we decided to implement a feed-forward network (MLP). As stated in the previous tasks, the network should be designed and trained using additional software and historical data of the series we want to predict. We used MATLAB in our tests.

The following diagram represents the layout of the neural network used. The inputs are the last values of the time series being used to predict the next value. As mentioned before, `LAG_INPUTS` parameters define the number of inputs in the network.

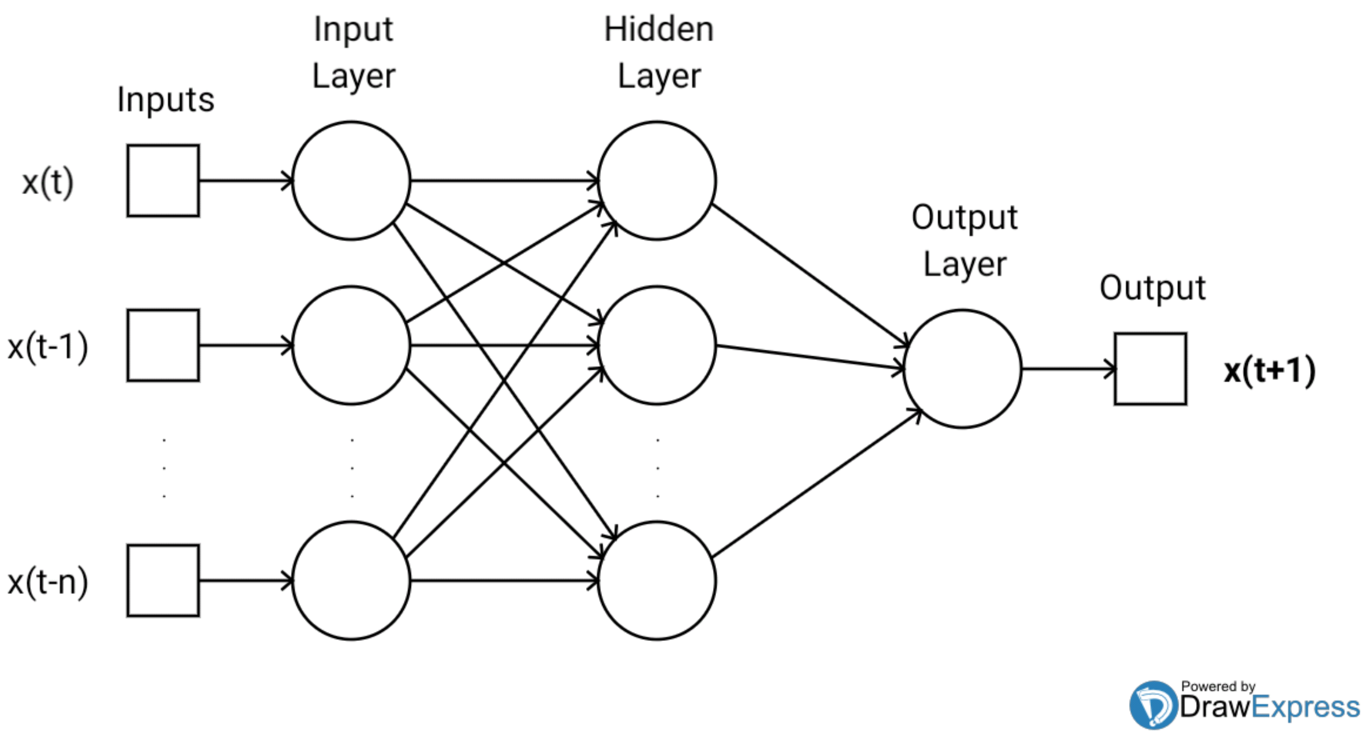


Figure 9 - Multilayer perceptron neural network.

The neural network was trained and tested on MATLAB using room lighting data collected from the UFSC IoT Gateway . Figure 10 shows the expected output and the trained network output.

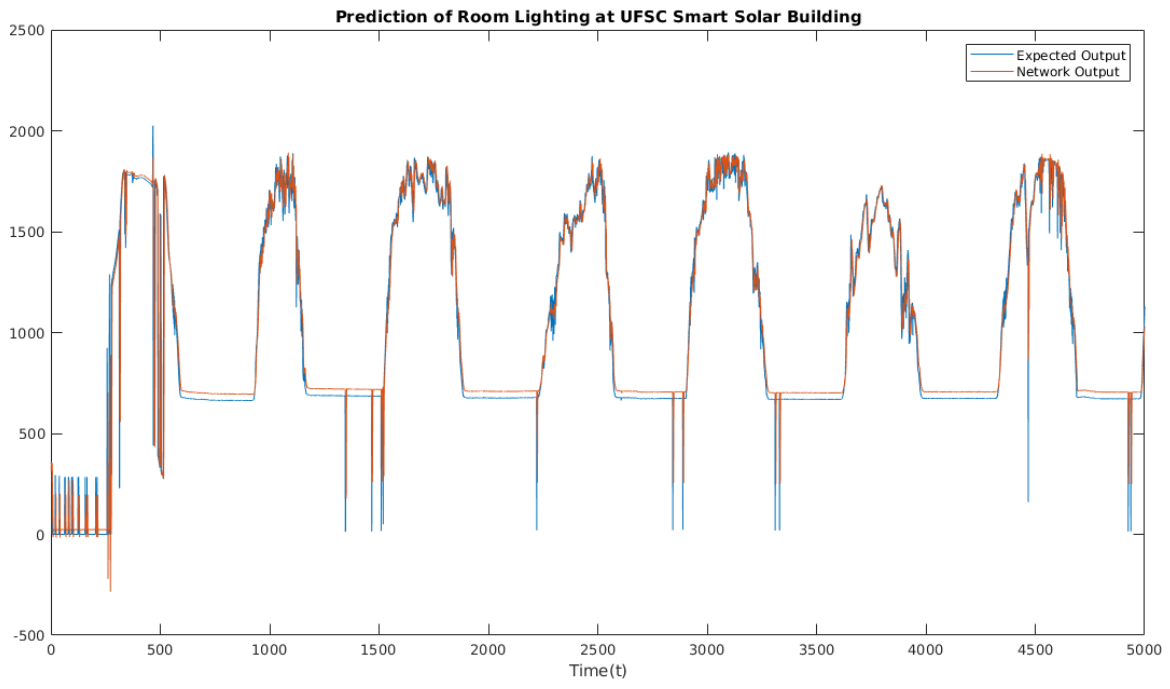


Figure 10 - Plotted results of trained MLP network on test set of room lighting data.

After training the neural network, we extracted its units' weights and biases and statically coded in the corresponding traits file. Tests on the network prediction and its ability to recover from desynchronizations were performed on EPOS using EPOSMote III . We also trained and tested a neural network on the Air Temperature at SFO Airport dataset. Figure 11 shows the neural network predictions on the first 5000 data points.

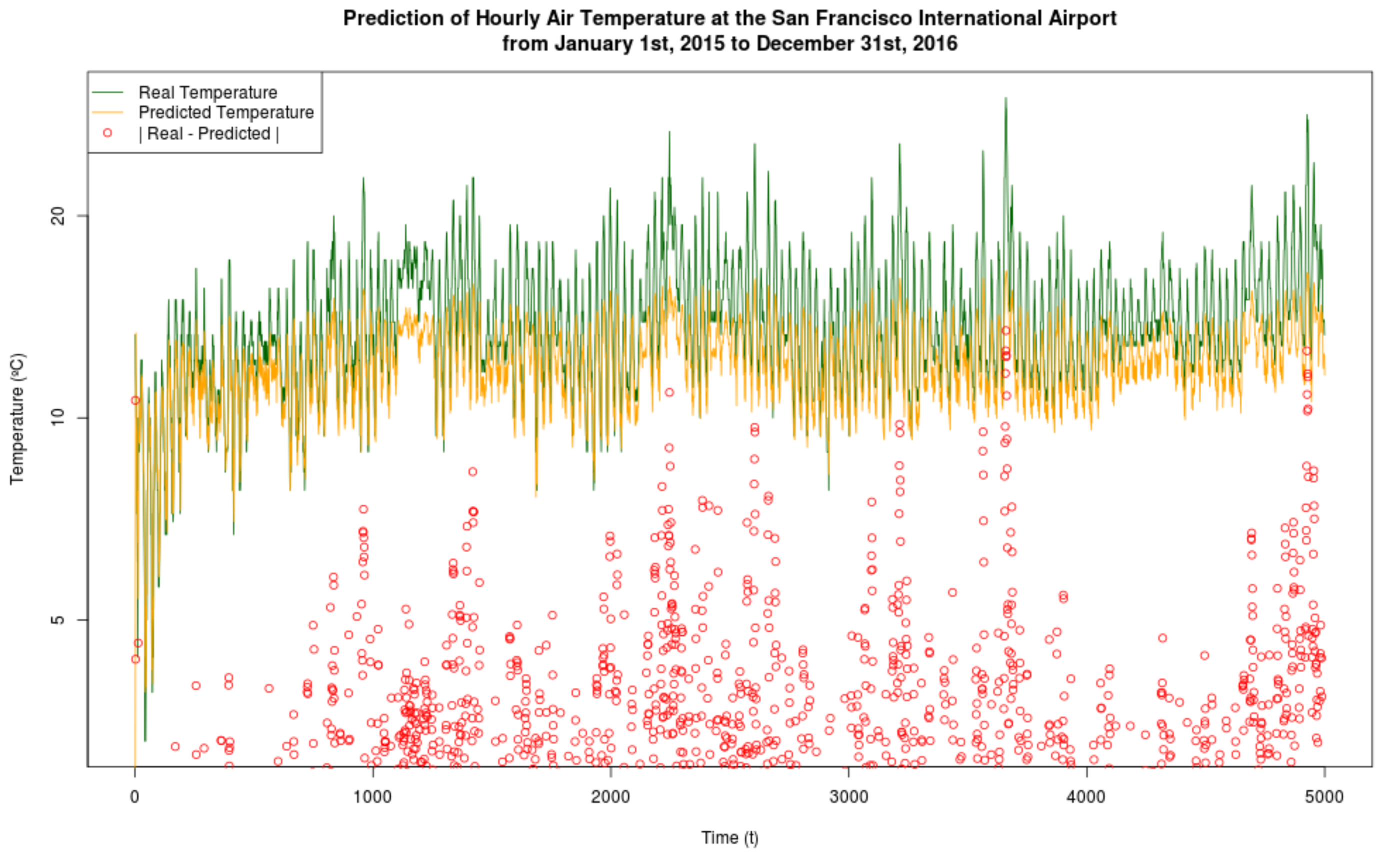


Figure 10 - First 5000 points of MLP predictor tests on temperature dataset.

Linear Regression vs. MLP Predictor

Feature	Linear Regression Predictor	MLP Predictor
Type	Dynamic	Static
Online learning	✔	
Needs series historical data		✔
Resilient to desynchronization		✔
No preprocessing needed	✔	

Conclusions & Future Work

We conclude by highlighting that **all the project requirements were fulfilled** (according to the changes described) and that **we achieved satisfying results**. Two different predictors were implemented and may be used for different problems in various contexts.

Some of the future work that can be done includes:

- Expand the MLP predictor to include networks with more layers, different activation functions, etc;
- Develop other types of predictors.

Bibliography

1. INTERNET of Things (IoT) Market : Global Demand, Growth Analysis & Opportunity Outlook 2023. 2017. Available at: <http://www.researchnester.com/reports/internet-of-things-iot-market-global-demand-growth-analysis-opportunity-outlook-2023/216>. Accessed on: 25 apr. 2017.
2. SOMASUNDARA, A.a. et al. **Controllably mobile infrastructure for low energy embedded networks**. 2006. Available at: <http://ieeexplore.ieee.org/abstract/document/1644743/>. Accessed on: 25 apr. 2017.
3. NEDRICH, Matt. **An Introduction to Gradient Descent and Linear Regression.** 2014. Available at: <https://spin.atomicobject.com/2014/06/24/gradient-descent-linear-regression>. Accessed on: 1 may 2017.
4. KOSKELA, Timo et al. **Time Series Prediction with Multilayer Perceptron, FIR and Elman Neural Networks.** 1997. Available at: <https://www.researchgate.net/publication/2638968_Time_Series_Prediction_with_Multilayer_Perceptron_FIR_and_Elman_Neural_Networks>. Accessed on: 1 may 2017.
5. NOAA Satellite And Information Service. National Climatic Data Center. **NNDC Climate Data Online**: Accessing data selection screen for Surface Data Hourly Global (DS3505). Available at: <https://www7.ncdc.noaa.gov/CDO/cdopoemain.cmd?datasetabbv=DS3505&countryabbv;=&georegionabbv;=&resolution=40>. Accessed on: 21 may 2017.