INFO - H - 515
2018 - 2019

10 June 2019

Big Data – Scalable Analytics Part

Project Assignment – Phase II

Kaïs Albichari	000395807	M-INFOS
Tanguy d'Hose	000409718	M-INFOS

Introduction

The second phase of the assignment for the course on Big Data and Scalable Analytics requires the students to implement a scalable distributed online forecasting system for missing sensor measurements.

In a real-world smart city project, data emitted by sensors can be subject to multiple problems. Indeed, malfunctions of sensors could lead to erroneous, noisy or even missing data. By training machine learning models on incoming streams of data, these problems can drastically be reduced, if not eliminated, through replacement of problematic values by realistic predictions.

Such a system is materialized through implementation of three different models to predict missing (or future) values. The assignment specifies the first model is a simple persistence model. Two other approaches need to improve on this very simple model through the use of machine learning. A precondition is that one of these additional implementations is based on the RLS model. Consequently, the last required model is fully contingent on the students' choice. In our case, we have opted for a simple, but efficient method in streaming contexts, namely stochastic gradient descent.

Streaming architecture

We have chosen to split our project in three different streaming architectures: one for each prediction model. Indeed, the goal of this assignment is to implement online learning models. Our assumption is that the implementation of such models in real-life situations would not require updating and predicting simultaneously on multiple models. Additionally, splitting the architecture also makes our code more readable and less chaotic.

For each one of these models, data is duplicated mostly to handle fault-tolerance. Then, we define states that are updated as the streaming data come and contain the information the learning processes generate as well as the necessary information for said processes to continue their learning activity. In other words, the states regroup data such as features, truth values, predictions, weights and other learning parameters.

In terms of implementation, it would be possible to make use of the three models within the same streaming architecture. However, changes in the way the states are handled should be operated. The *updateStateByKey()* method matches the keys of our preprocessed batch data with the sensors we want to predict which are associated to the state as keys. Therefore, a more sophisticated definition of these keys should be created in order to duplicate the data along the different predictions models that are implemented.

The three architectures utilize same technologies: Kafka paired with zookeeper enables our senders to dispatch and recuperate data to and from the correct topic. Additionally, it permits to follow the "publish-subscribe" message queue paradigm. It is also these technologies that provide replication of information that is being streamed. Management of streamed data is handled by the Spark Streaming framework. This framework permits to obtain short-span data (mini-batches) that are transformed to RDDs. A great advantage of Spark Streaming is that multiple data instances can be handled simultaneously, as opposed to one-by-one management.

The collected RDDs are transformed based on the expectations of our learning models. Use of RDDs permit parallelization of operations. Indeed, instead of iterating over received data, operations can be applied to entire data. All learning algorithms also avoid iterative operations by making use of DataFrame or NumPy specific functions. We did not push parallelism further by multiplying the number of active jobs. It is known that Spark Streaming allows increase of concurrent jobs by modifying *spark.streaming.concurrentJobs* to a value greater than 1. The reason we did not provide experimental data on this matter is quite simple: if batch processing takes longer than the batch interval, weird situations can occur and correctness of learning can no longer be guaranteed. Indeed, for this project, it is imperative data is streamed in chronological order. If chronology is lost, learning is not necessarily applied on data of the same slots or even same days, yielding sub-optimal results. Parallelization can also be obtained through parametrization of the environment. In this context, the number of cores and executors per core, as well as the required batch interval must be analyzed.

Running the architecture correctly requires the execution of steps in a very precise sequence. Indeed, after having started Kafka and Zookeeper, the receiver is the first that needs to be launched. After a few moments, the sender can be launched. By doing so, it will initialize adequately and no errors or unexpected behavior will occur.

Scalability of our architecture will be discussed after having introduced and discussed our prediction models.

Prediction methods

First, recall some conventions that were defined in the assignment. Let $s_i(t)$ denote the temperature of a sensor i at time t and let $\hat{s}_i(t) = h(x_i, \theta_i)$ be the prediction of $s_i(t)$ using a prediction model h with input data x_i and parameters θ_i.

1. Persistence

As described in the assignment, the persistence model predicts the value of sensor solely based on the last observation:

$$\hat{s}_i(t) = \theta_i$$

With θ_i being $s_i(t_{last_i})$ and t_{last_i} the last instant for which data for sensor i was collected. It comes without surprise that the persistence model is not well-suited for the considered problem. Indeed, temperature values are expected to change numerous times during a span of 24 hours. The predictive quality of this model being poor, the goal of the assignment is aimed towards implementing models that are better-suited for predictions corresponding to our smart-city problem.

2. Recursive Least Squares

Recursive Least Squares (RLS) is a recursive application of the Least Squares regression algorithm. In this approach, each new observation is considered to correct a previous estimate of the supposed correlation between features and observation (the objective function). To do so, it aims to minimize the weighted linear least squares cost function.

3. Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is an approach that aims to identify the objective function that, in our case, defines temperature distributions through an iterative process. Using our formalism, the predictive model would be described as such:

$$\hat{s}_i(t) = \sum_i x_i w_i$$

where x_i is the observed temperature of the slot of sensor i (or neighboring sensors) the previous day and w_i is the weight associated to that feature. However, our approach poses the hypothesis that predictions are made for a 24-hour span, thus, the model can be updated for all slots simultaneously.

As mentioned in the assignment, the first day serves to collect data for the initial model with hopes to predict the next day. Evidently, it is impossible to predict very accurate values as no feed-back has been provided by measurements of the next day. For this purpose, it is important to initialize weights adequately. Indeed, in most cases weights can be initialized to zero as these will be subject to training before any prediction. In our approach, a first prediction is to be emitted without preceding feedback. By initializing weights to 1, we obtain a first prediction not too far off the real value. When considering multiple features, the initial weight is set to $\frac{1}{\#features}$.

Weights form the core of the learning method. It is precisely the update of these weights that permit to narrow the scope of predictions and steadily approach the objective function. More precisely, when feedback is provided (actual observations) weights are updated as follows:

$$w_i(t) = w_i(t-1) - \eta \nabla w_i(t-1)$$

Where the second term represents the derivative of loss with respect to the selected features. There are many possibilities of choice for a loss function. In our case, loss is the negative difference between observations and the hypothesis nuanced by a small regularization term. By subsequently deriving with respect to the features (temperature and bias) and dividing by the number of observations (slots in a day), the second term can be evaluated and weights updated adequately. Additional research provides support for weight normalization: weight normalization speeds up convergence of stochastic gradient descent [1], resulting in fewer iterations required to obtain realistic predictions.

It is important to know that, given current number of streamed values and architecture, all slot observations are used to update weights. In that perspective, stochastics of the method does not lie in the random selection of instances to update weights, but rather the one-pass training that is exercised on all provided instances.

Sensor Predictions

In this section, we have decided to include three graphs for each required sensor in each model. The first graph shows the prediction for that sensor, alongside the actual measured temperatures the next day for the entire week. The second graph focuses on the last day. Indeed, as training is continued on during the entire week, one would expect most optimal results would occur on the last day. By dedicating a graph to that day, we can easily see to what extent the considered model is capable to predict temperatures. Finally, we display a graph of the evolution of the MSE on the past 24 hours. In this case, models that are updated at each slot consider the last 2880 slots, whereas models that are updated daily can only dispose of that measure at each day's end.

1. Persistence

1.1. Week

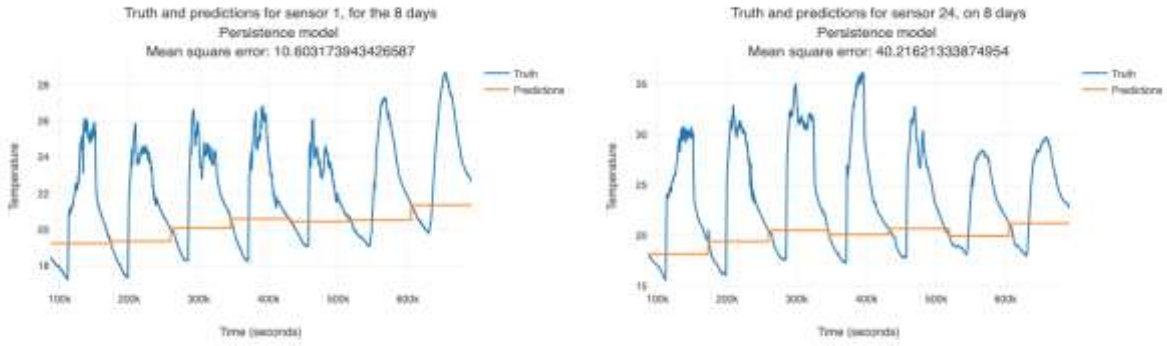

Fig 1. week predictions sensor 1(left) and 24(right), persistence model

As expected, the persistence model is not well-suited to the problem. We clearly see that predictions only rarely correspond to the real values. Consequently, we expect that other, smarter models will certainly outperform persistence.

1.2. Day 8

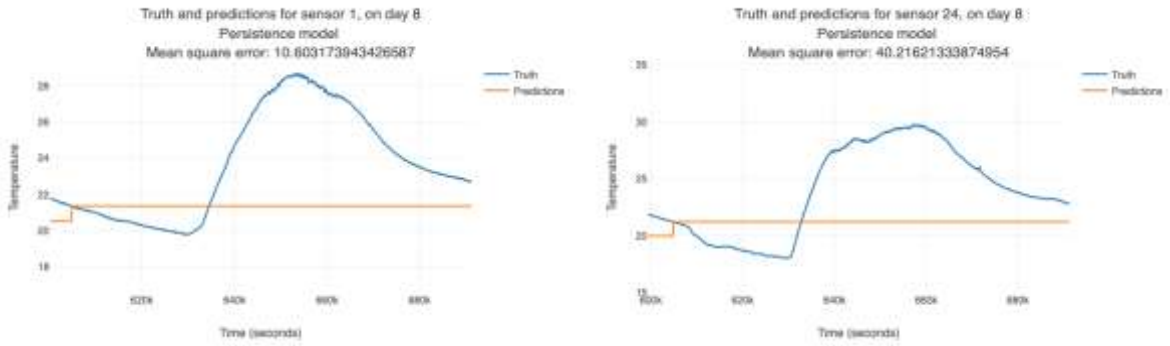

Fig 2. day 8 predictions sensor 1(left) and 24(right), persistence model

Highlight of the last of prediction shows that there only exist 2 intersections between prediction and truth. In terms of accuracy, the system dramatically fails as only 0,0007% of all predictions are correct, or only 1 minute throughout the day. Of course, accuracy is not a very meaningful measure in such a system, this is why MSE is key to the analysis.

1.3. MSE

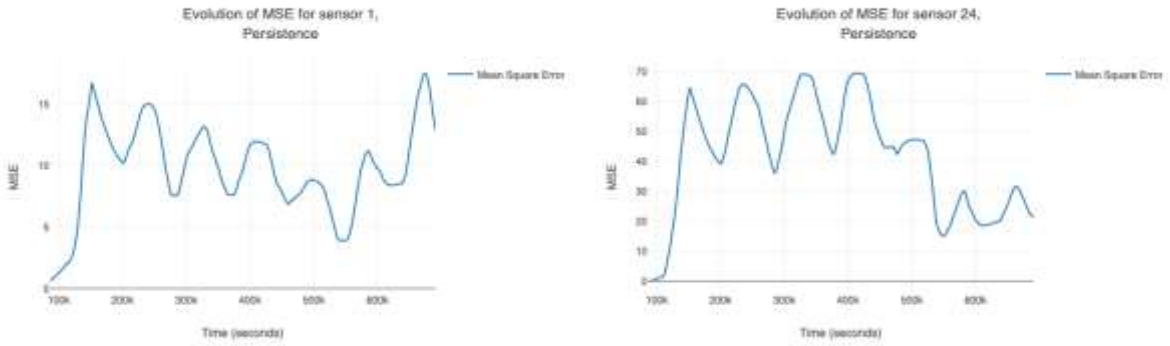

Fig 3. MSE of the week for sensor 1(left) and 24(right), persistence model

Analysis of the MSE shows a trend that really resembles the original predictions, only smoother and with less difference between minima and maxima. Evidently, the definition of MSE smooths out positive and negative differences between predictions.

2. Recursive Least Squares

2.1. Week

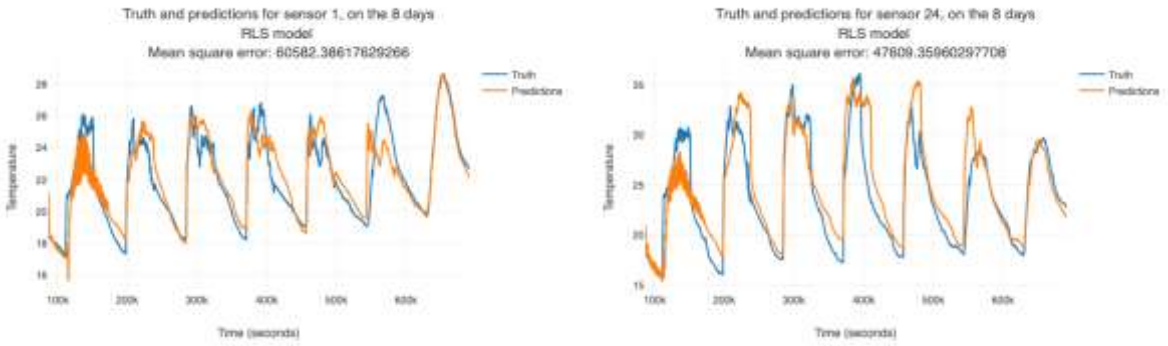

Fig 4. week predictions sensor 1(left) and 24(right), RLS model

It is important to mention the week prediction omits the first 10 predictions for plotting. Indeed, these are so off-balance that the scale becomes much greater and other results unclear. We do see that predicting the first day is quite chaotic, but stabilizes throughout the day. From then on, predictions are quite good. For the first sensor, predictions end on an MSE of 0,07. It important to notice that this potentially biased by the fact that day 8 has temperatures very similar do day 7 in terms of distributions. However, predictions for sensor 24 are a bit less precise even if the resemblance between day 7 and day 8 is also noticeable.

2.1.1. Day 8

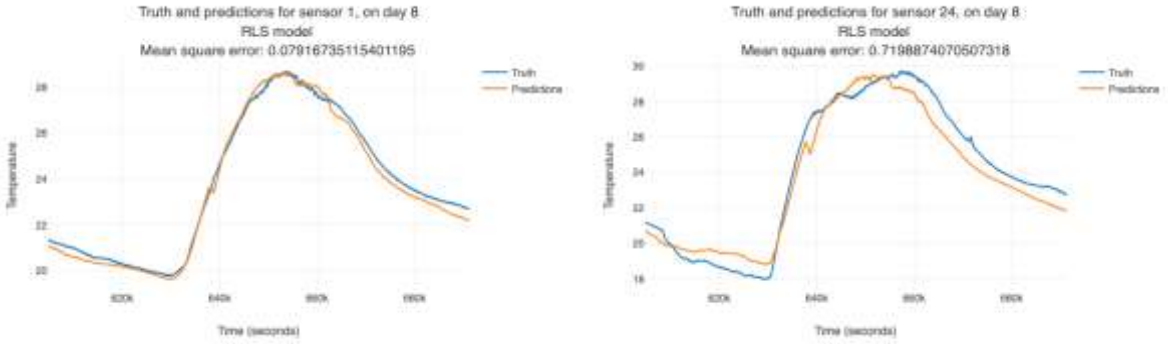

Fig 5. last day predictions sensor 1(left) and 24(right), RLS model

Focus on predictions for the last day portraits of qualitative these are. More particularly, RLS has no difficulty predicting the trends of evolution of the day as predictions are closely situated to the truth. Additionally, we see that deviation from the truth rarely exceeds 0,5°C. This is confirmed by the fact that the MSE is inferior 1. Using the provided MSEs, one can easily calculate that the mean deviations for sensors 1 and 24 are 0,28°C and 0,85°C respectively.

2.1.2. MSE

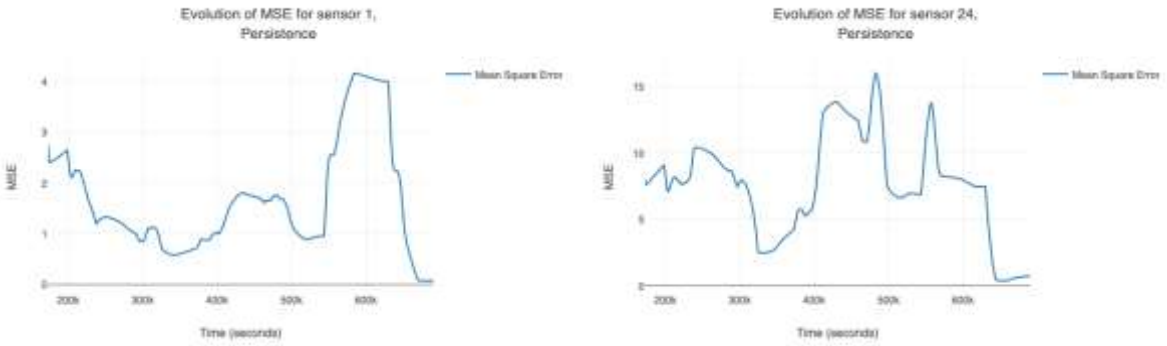

Fig 6. MSE of the week sensor 1(left) and 24(right), RLS

As in our weekly predictions, the first day's values have been omitted in this plot to favor readability and better identify the MSE through the 7 days of prediction. The first sensor does present a decrease in MSE in the first part of the week. Afterwards a small increase is followed by a severe drop and abrupt increase for the 6th day, to finally end on an absolute minimum. Seeing the predictions for the week, it is clear why this occurred: day 7 is an outlier in terms of distributions so far. Hence, the model could not have predicted it better. The second sensor does have MSEs that are situated much higher at any moment and reaching a maximum more than three times as important as sensor 1. Looking at the truth values for this sensor, one can notice there less similarities between days and more differences in temperatures day to day. Consequently, the model has issues finding the weights corresponding to the objective function.

3. Stochastic Gradient Descent

3.1. Week

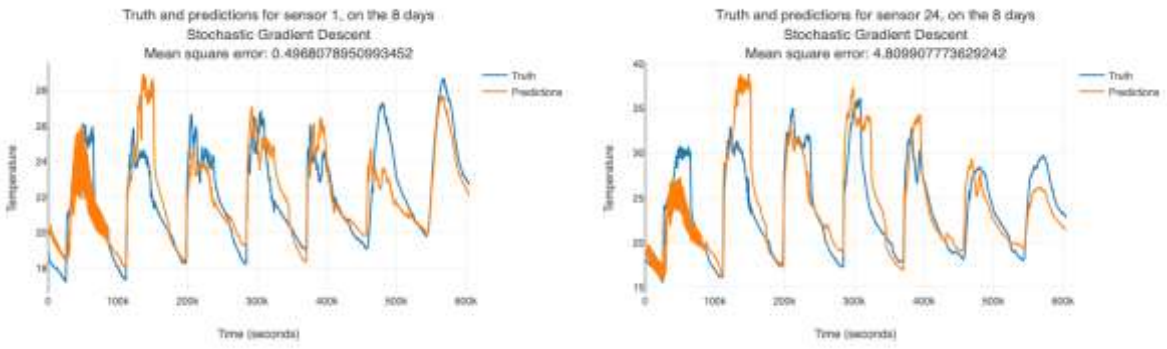

Fig 7. week predictions sensor 1(left) and 24(right), SGD

Weekly predictions by the SGD model show that the SGD is very dependent on previous observations for future predictions. Unfortunately, that strategy does not pay off very well in this case. Indeed, temperatures can severely vary from day to day, meaning that correlations or linear tendencies do not necessarily exist. However, as one can see from the last 2 days for sensor 1, when a correlation exists the predictor does do very well.

3.2. Day 8

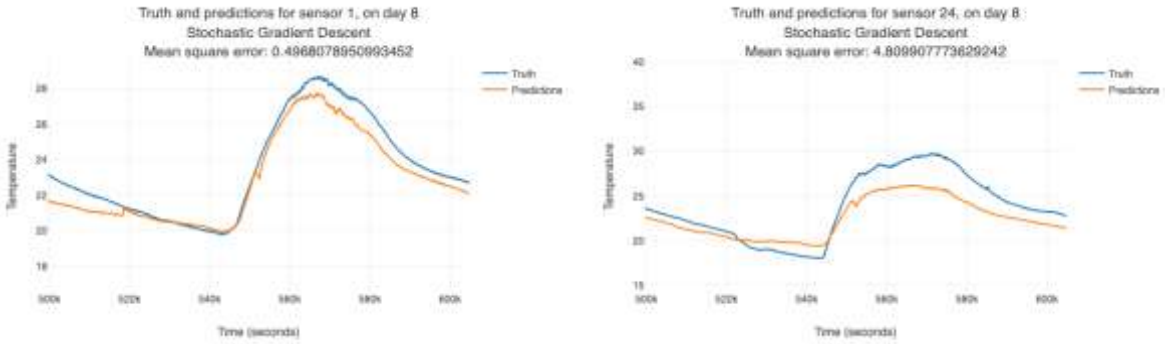

Fig 8. last day predictions sensor 1(left) and 24(right), RLS

A highlight of the last day's predictions show how SGD performs in the context of this problem. We see that sensor 1's predictions are very close to reality on the last day. This is not the case for sensor 24. Indeed, the model is able to identify the tendency of the day, but approaching correct temperatures is a goal that has not been realized on that day. In fact, the MSE for that day shows that the mean deviation with respect to reality exceeds 2°C.

3.3. MSE

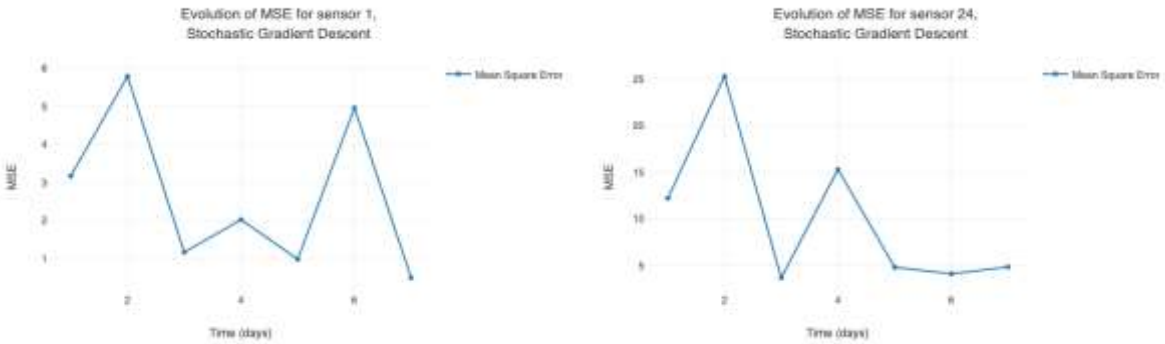

Fig 9. MSE of the week sensor 1(left) and 24(right), RLS

MSEs associated to our SGD model provide more insight on performance. The first sensor presents relatively small MSEs, indicating a temperature distribution that has more similarities throughout the week. One can also notice a tendency where the MSEs reach a new absolute minimum every 2 updates, possibly indicating a correct path to long-term convergence. For sensor 24, the observations are quite similar in the sense that minima seem to occur (almost) every 2 updates. Unfortunately, these minima are never inferior to 4 or less. This is a clear indication that the model requires more time to converge.

Scalability

The level of scalability of the system is defined by the number of sensors it can accommodate without requiring vast amounts of changes in implementation. As mentioned in the assignment, predictions for sensors 1 and 24 are expected. For this purpose, we have chosen to lighten the load that is sent by the Kafka producer and restrict it to measures of the considered sensors, as well as measures from their five nearest neighbors.

Evidently, a first level of scalability would revolve around the inclusion of all other sensors. To enable this, only few changes have to be applied to the sender's procedure to avoid restrictions on the information that is being sent. Indeed, send and receive procedures would be confronted to larger batch sizes. In case of the persistence model and versions of other models having only one feature, under normal conditions this would imply linear increase in sent data. When RLS and SGD consider more features (larger number of neighbors), increases more drastically. The exact manner in which it does increase depends on the exact combination of neighbors and whether some are redundant.

If one considered the execution of RLS alongside 5 closest neighbors, the streaming infrastructure would be submitted to the following load:

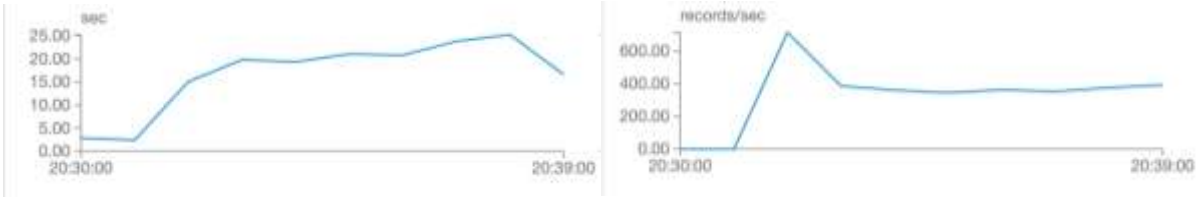

Fig 10. delay per batch (left) and records/sec (right)

We see that for the analysis of 2 sensors, delays can reach almost 25 seconds. The number of records spikes to 600 for the first batch, and then stabilizes around 400 records per second. In these conditions, the architecture already fails to satisfy the speedup conditions and intervals had to be increased.

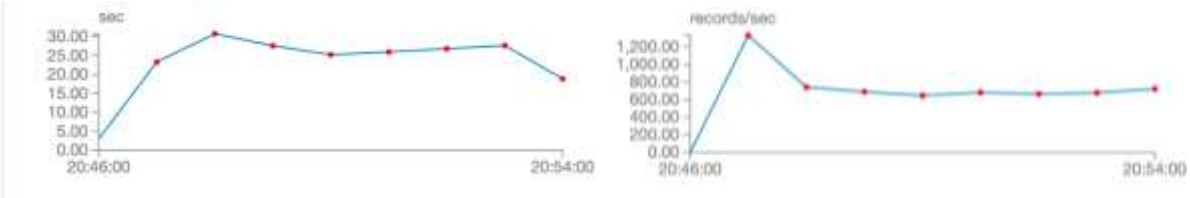

Fig 11. delay per batch (left) and records/sec (right)

The above graphs represent a situation in which we double the number of observed sensors. In this case, there is a slight increase in seconds required per processed batch. In fact, almost all of them now

require a delay larger than 25 seconds to be processed. The graph on the right shows that the number of records per second simply doubled since last time.

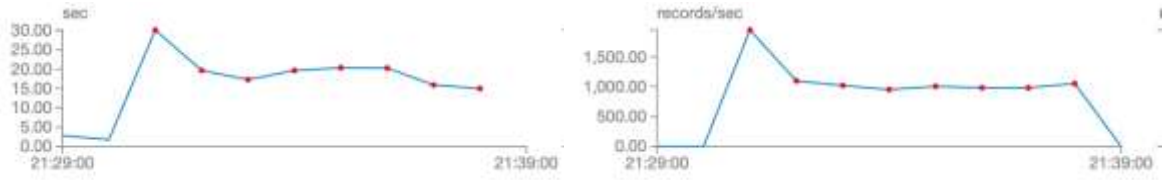

Fig 12. delay per batch (left) and records/sec (right)

When adding 2 additional sensors, delays per batch stabilize in some way and do not require much more time. Contrarily, the number of records that need to be processed per second dramatically increases. Additionally, analysis of the sender's side (Table 1) shows that the more sensors are analyzed, the more time is required for pre-processing and sending data. In fact, if all known sensors would be sent, the increase in time would be considerable. Though original speed-up conditions are not respected, evidence suggests the system would perform fine at normal speed.

day	*2 sensors*	*4 sensors*	*6 sensors*
0	22,81	32,66	48,81
1	19,19	39,90	48,09
2	16,74	30,60	35,88
3	16,25	26,70	42,40
4	21,41	26,65	38,78
5	16,41	26,15	35,64
6	16,64	24,78	38,95
7	18,35	24,63	29,07
average	**18,48**	**29,01**	**40,95**

Table 1. send time per number of sensors.

Of course, one could also imagine that additional sensors are added to the project. This would translate in potentially much heavier loads being sent and received. As the current architecture is able to perform relatively fast under speed-up conditions, it safe to assume it could handle a higher load of information in normal speed conditions. However, if growth of sensors is such that the system becomes overloaded, Spark Streaming permits to handle data in parallel and, hence, reduce latencies introduced by these new readings.

By modifying the number of cores and executors per core, the time required to process the batches can severely by reduced. Nonetheless, defining the correct parameters for spark streaming is a process that

needs to be executed correctly. Particularly in this case, the number of cores and executors should be defined considering possible future scaling of the project.

Discussion

1. Problems encountered

As expected, a lot of problems were encountered during the translation of our terminal code into its streaming counterpart. Unfortunately, we forgot that Spark Streaming comes accompanied with Spark UI that enables users to dispose over useful information for debug. Once with recall this information, implementation was a lot smoother.

Furthermore, we had some difficulties grasping the way in which RLS worked. Many available resources do not dive into the internals of the algorithm or do not provide sufficient contextual information to understand.

2. Limits of the solutions

The persistence model is known to be a very limited solution. In fact, the persistence predictor is considered to be a very naïve forecast best suited as baseline. Its inefficiency is quite clear in the context of this problem: the size of the range of temperatures on 24h as well as the somewhat Gaussian form of the objective function result in predictions that are rarely a reflection of reality. Better predictions could be obtained with variants of persistence (persistence per hour for instance), but this would still not be appropriated to the considered system.

Recursive Least Squares belongs to the family of LMS-type stochastic gradient descent algorithms. Though quite simple, robust and efficient, it does remain subject to limitations of this family. It is known that LMS-type SGD algorithms have slow convergence. This problem is further highlighted when the input signal is correlated. RLS does provide better convergence than other algorithms of the LMS-type family. In the case of this application, RLS provides decent results, but room for improvement exists.
Though RLS is a very popular method due to its fast convergence, computational complexity of this algorithm can become a hurdle in some applications. If the project was to include other prediction tasks or more complex information, the algorithm could become ill-suited for online learning.

There are several approaches to Stochastic Gradient Descent. In this paper, we have opted for an online version of the algorithm that is updated once a day based on predictions of the previous day and the obtained observations. We do notice that the model can quickly pickup on similarities between days if

these exist. However, it does not generalize well. Typically, Gradient Descent algorithms require a vast amount of iterations to identify the objective function and adapt weights suitably. In online versions of the algorithm, weights are updated using solely one pass over data. In other words: the algorithm will require more updates to converge. This supports our previous claim on SGD-type algorithms.

Our current devised architecture performs using fairly standard streaming parameters. As described previously, some parameters can be optimized to better suit the problem. In most cases, modification of the number of cores, executors per core and interval per batch suffices to increase performance and put a first step towards scalability.

3. Future work

As problems were identified with the SGD approach, namely slow convergence, it would be very interesting to see how much updates SGD would require to converge. By running the algorithm on long periods of time, we would be able to see whether standard online SGD outperforms RLS and if some kind of convergence time – precision tradeoff exists.

Additionally, RLS and SGD both use hyperparameters such as forgetting factor, regularization factor and learning rate. The values were chosen based on typical values that work well for these algorithms. However, it must be said that this means they can potentially not be optimal for the problem. For this purpose, hyperparameter optimization is a step that still needs to be performed if the application would be expected to go live.

The last limitation of our problem, concerning parameters of Spark Streaming, is prone to future work. Indeed, if one was to analyze the characteristics of the problem in more detail, better parameters could be obtained. To this purpose, the most promising parameters are number of cores, number of executors per core and the batch interval. As seen through some of our simulations, modification of batch interval permits accommodation of greater number of items. If this work was to continue, a more detailed analysis on these elements would benefit the project.

References

1. Salimans, T., & Kingma, D. P. (2016). Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *Advances in Neural Information Processing Systems* (pp. 901-909).