Online Metrics Prediction in Monitoring Systems

Matthieu Caneill*, Noël De Palma*, Ali Ait-Bachir†, Bastien Dine†, Rachid Mokhtari† and Yagmur Gizem Cinar*
*Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG, 38000 Grenoble, France
Email: {caneill, depalma, yagmur.cinar}@imag.fr
†Coservit
Email: {ali.ait-bachir, bastien.dine, rachid.mokhtari}@coservit.fr

Abstract—**Monitoring thousands of machines and services in a datacenter produces a lot of time series points, giving a general idea of the health of a cluster. However, there is a lack of tools to further exploit this data, for instance for prediction purposes. We propose to apply linear regression algorithms to predict the future behavior of monitored systems and anticipate downtimes, giving system administrators the information they need ahead of the problems arising. This problem is quite challenging when dealing with a high number of monitoring metrics, given our three main constraints: a low number of false positives (thus blacklisting volatile metrics), a high availability (due to the nature of monitoring systems), and a good scalability. We implemented and evaluated such a system using production metrics from Coservit, a company specialized in infrastructure monitoring.**

The results we obtained are promising: sub-second latency per predicted metric per CPU core, for the entire end-to-end process. This latency is constant when scaling the system up to 125 cores on 4 machines dedicated for monitoring predictions, and the performances don't decrease with time: during 15 minutes, it is able to handle more than 100 000 monitoring metrics.

I. Introduction

Monitoring machines ensures a system is running correctly, and triggers human intervention as soon as it is needed. A lot of monitoring tools exist nowadays, ranging from in-house scripts to complete software suites, distributed on many servers. Monitoring software usually collect metrics (e.g. CPU load, available memory), check them against defined thresholds (e.g. 80% of the maximum), raise alerts to system administrators, and generate reports about resources consumption and error rates.

There are different kinds of collected metrics: system-related (CPU load, network speed, etc.), services-related (database uptime, web server open connections, etc.), or statuses (up, down, unknown). These metrics are timestamped, and are collected as time series.

Coservit is a company which has collected about half a million different metrics for more than 5 years, across dozens of thousands of physical hosts. We used this dataset to experiment different approaches, aiming to predict the future behavior of these metrics.

The goal of our system is to make predictions about the health of online services in the near future, in order for system administrators to perform preventive maintenance, instead of reacting to problems and fixing them when they occur. This is why we focused on a short time horizon, typically a few hours. We applied linear regressions on the different collected metrics, using historical data to train our system. We found linear regression to be the best fit for this kind of data and this horizon, thanks to its ability to identify local trends.

We added the constraint of creating a scalable system that is not limited to a maximum number of metrics, or a single host. It must also be as CPU-efficient as possible, and for instance not waste resources learning metrics which are too difficult to predict due to their volatile aspect.

The challenges involved to develop this system were the huge amount of collected metrics, the processing time per metric which had to be fast enough to be useful, and the need to avoid false positives. Our contributions are the description of a scalable system, some optimizations we applied to it, and its evaluation with industrial production data.

We found almost 60% of the observed metrics are suitable for linear regression prediction; as detailed in Section II-D the other ones are too volatile to be accurately predicted with this method. The entire process of retrieving measurements, building the learning parameters, predicting the values, and storing the results takes about one second per metric. Moreover, it scales linearly up to all the CPU cores we had at our disposal for evaluating this system.

The rest of this paper is organized as follows: we first detail our solution in Section II, before evaluating it in Section III. We then present the related work in Section IV, and conclude in Section V.

II. System description

A. Architecture

We base our architecture around a Cassandra database, used for storing monitored metrics, predicted values, and prediction error rates. Cassandra [1] is a column-oriented, distributed database, designed for fault tolerance and scalability. Another important framework we use to distribute work among different machines is Spark [2]: it provides the same properties of fault tolerance and scalability, along a rich API for data transformations and machine learning.

Figure 1 shows our architecture. We consider the monitoring agents as black boxes, geographically scattered in different datacenters, reporting metrics about the systems to a monitoring broker. All the metrics are stored in Cassandra for further processing.

Spark workers find the metric identifiers ready to be processed in a RabbitMQ message queue, read all the data points

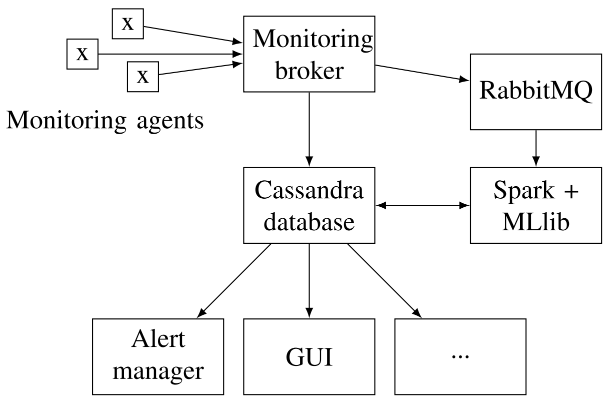


Fig. 1: System architecture

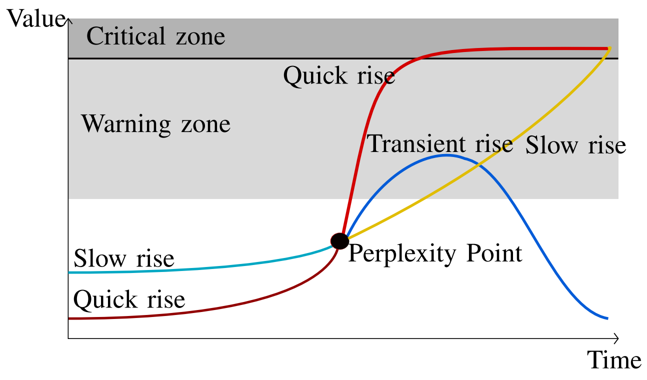


Fig. 2: Metric trend cases

associated to them from Cassandra, and run prediction algorithms with MLlib (the machine learning library bundled with Spark). The message queue serves as the producer/consumer middleware, and is useful to regulate the flow of predicted metrics and which metrics need to be processed.

Finally, end-users can use different front-ends, most notably a web-based graphical interface alerting them about future problems.

This architecture resembles a lambda-architecture [3]: it leverages historical and real-time data to provide up-to-date predictions combining both sources.

B. Data model

We use a Cassandra cluster to store all the collected metrics, as well as the different computed predictions on them. The relevant tables are:

- *metrics*: stores the metric names (e.g. `open_sockets`, `disk_freespace`).
- *metric_measurements*: each row represents a measurement, and consists of a metric id and a value, as well as a timestamp, a unit, the warning and critical thresholds, and if relevant the minimum and maximum possible values.
- *metric_predictions*: each row stores two predicted points (a point is a value and its timestamp). The first point represents the metric in the near future (a few seconds ahead), while the second point is the metric in a more distant future, determined by the prediction horizon.
- *metric_errors*: each row stores the root-mean-square error (RMSE) calculated in the blacklisting process. This permits to later filter out metrics whose measured values don't align with the predictions.

The stored metrics are flattened: they don't represent the hierarchy of hosts, services and metrics found in some monitoring engines. This is on purpose: Cassandra is not used to store this kind of information, which is better managed by specialized tools. In fact, this schema is monitoring engine-agnostic: it is meant to be used by different engines as a sink for their metrics.

C. Linear regression

We first discuss in this section why linear regression makes sense compared to the legacy threshold methods, and then describe how we applied it on our metrics.

1) Comparison with threshold: Most metrics have two thresholds: warning, and critical. A system in the warning state continues to properly run, but it is a signal that some components might break. For instance a CPU whose load is 80% is still working, but approaching its maximum capacity. Most systems will emit a notification when a metric is above its warning threshold, but it doesn't mean the trend will continue and the metric will enter its critical state.

Looking at the evolution of data points among different kinds of metrics, we identified 6 common scenarios, represented in Figure 2. Two situations lead to the *perplexity point*, the point in time when metrics get closer to the warning threshold, and from where three situations can arise. We evaluate the benefits of linear regression versus the warning threshold for each of them:

- *Slow rise* followed by *slow rise*: linear regression is a perfect fit for this situation, as it will easily identify the trend even before the warning zone is reached.
- *Slow rise* followed by *quick rise*: both linear regression and the threshold system will be efficient if they refresh their measurements often enough. If the rise is too fast, they will both predict the issue too late.
- *Slow rise* followed by *transient rise*: if linear regression can predict the future decrease, it will avoid sending a false positive alert.
- *Quick rise* followed by *slow rise*: both systems might give a false positive, or at least predict the issue too early. However, linear regression will be better at predicting the change to normal state again.
- *Quick rise* followed by *quick rise*: depending of the frequency of the measurements, the threshold system might alert too late about a fast arising problem, whereas linear regression can predict it.
- *Quick rise* followed by *transient rise*: again, false positives might be sent by both systems, but linear regression is better at anticipating the return to normal state.

Linear regression is better than the legacy threshold system in 4 out of 6 scenarios, and better or equal in the other 2. That makes it a good candidate for predicting the behavior of many metrics.

2) Linear regression on monitoring metrics: We use MLlib (bundled with Spark) to perform a linear regression on data distributed on different machines. The training phase is performed independently for each metric, using stochastic gradient descent and a history size of 30 points. We found this value to be optimal: it gives good results (described in more details in Section III) while keeping the training phase to a reasonable time (~100 ms). An advantage of using stochastic gradient descent w.r.t. other methods such as ARIMA is the avoidance of tuning parameters for every metric. Once performed, we store the prediction results back into Cassandra in the *metric_predictions* table, available for consumption by different front-ends, and by the error evaluation process (Section II-D).

We didn't explore in this work the potential correlations between different metrics. We observed it often happens that multiple errors are reported from the same machine in case of a failure (if it runs multiple monitored services), but this is less obvious when predicting problems in advance.

3) Prediction horizon: The prediction horizon (the estimation of how long the predictions are valid) is complex; as it is generally less and less correct over time. However, we noticed that a maximum horizon of 8 hours is a good metric, and this is the amount needed for reliability engineers to not wake up overnight to fix services. The predictions are given as "best effort": they represent the best values obtained by the system, but can't give guarantees about their veracity. It is important to note they are continuously recomputed, and hence never out-of-date.

D. Metrics selection

Not all metrics are good candidates for prediction. Some metrics don't show any pattern, and never respect their predicted values. We use a blacklisting algorithm to eliminate them, in order to save computing resources and avoid false positives. A weekly batch script performs an error evaluation (RMSE, Root Mean Square Error) of the predicted values, which are compared against the observed values for the week. If the RMSE is higher than a given threshold, the metric is blacklisted.

E. Optimizations

This section describes some interesting optimizations that helped us greatly reduce the time and resources needed.

1) Caching: Spark implements a persistence mechanism, which allows to cache data either on memory, and/or on disk. By analyzing the data processing chain, one can spot the states where saving data brings computation time benefits, typically when one Dataframe is to be used by many other functions. As sometimes persistence can reduce performances, when the cost of caching data is greater than the benefits it provides, comparing the performances both with and without persistence

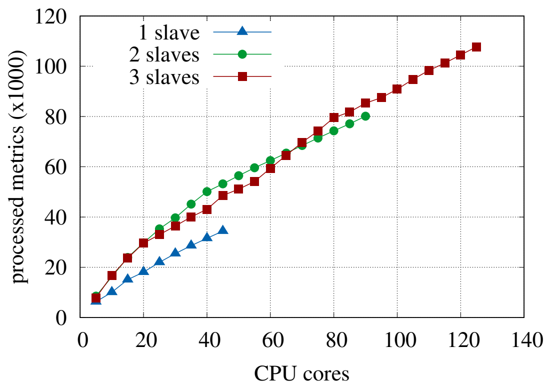


Fig. 3: Number of metrics handled in 15 minutes.

is rather necessary. In our case, we found that persisting the measurements data for a metric after it was retrieved from Cassandra greatly reduced the processing time.

2) Dataframes: Spark provides different APIs to developers, the main one being Resilient Distributed Dataset (RDD). RDDs are suitable to apply transformations on large, unstructured datasets. On the other hand, Dataframes, another container for distributed data, relies on tabular data organized into columns and associated to a schema, like an SQL table. For relatively complex queries, using Dataframes brings consequent speed-ups [4], because they benefit from advanced optimizations such as a query planner (the Catalyst Optimizer). Since our data is already organized into tables, we could compare both approaches, RDDs and Dataframes, and the latter were the fastest.

3) Cassandra optimizations: Cassandra being a distributed, column-oriented database, the key to obtain good performances is to design a schema around the expected types of queries it will get, at the price of duplicating data. Cassandra is optimized for writes, and in order to get good performances for reads, it is necessary to well partition the data. Good practices recommend two goals for that purpose: balance data evenly across machines, and minimize the number of partitions that need to be accessed for one query. It is recommended for a partition to be a few hundreds of megabytes, and contain a few hundreds of thousands of values. Our biggest table containing the measurements as reported by the monitoring engine, and every unique monitored service having up to a few metrics updated at most every minute, we decided to keep monthly partitions per service, which would weigh around 60 MB. It is to be noted that yearly partitions would be sustainable by Cassandra as well.

III. Evaluation

A. Setup

We ran the experiments on 4 physical machines (HPE Proliant DL380), with 16–28 hyper-threaded cores, and 128–256 GB of memory. We installed Debian stretch 9.0, Spark 2.1.0, and Cassandra 3.0.9.

One machine is the master and the three others are slaves. We replayed the production load triggered by the monitoring

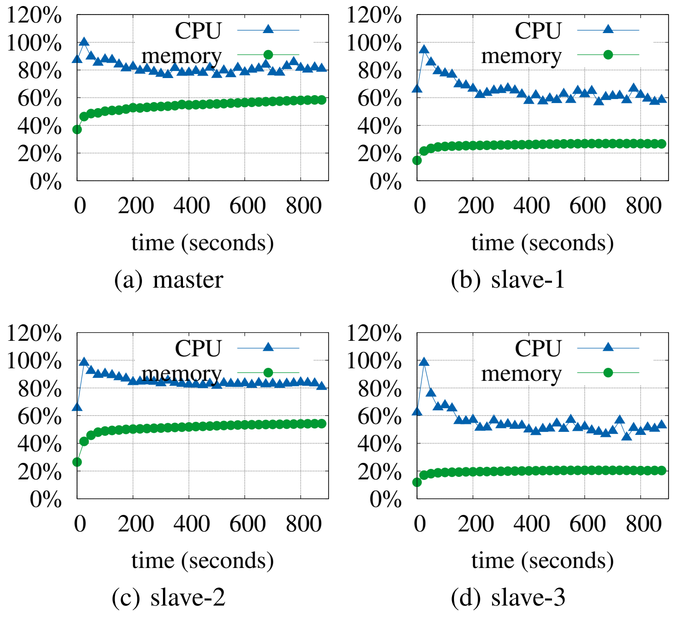


Fig. 4: CPU load and memory consumption, when running on 100 cores for 15 minutes.

boxes reporting metrics, and measured different parameters under different conditions.

We replay a dataset made of production data recorded on Coservit's servers. It represents two weeks of data, for 424 206 unique metrics. In total, there are 1 500 335 458 data points, whose size is about 15 GB in total. To get all these data points, 25 070 machines were monitored. An interesting deduction we can make is that, on average, there are about 17 monitored metrics per machine (however the standard deviation is about 30, so it highly depends on the type of machine).

B. Scaling

To check the system scaling performances, we measured the amount of metrics which can be processed in a 15 minute time range, varying the amount of slaves between 1 and 3, and the number of CPU cores between 5 and 125. Figure 3 shows the results. When using only 5 cores, the system could work on about 7000 metrics ($\pm$ 1000, depending on the number of slaves) in the given time range. This value scales linearly with the number of CPU cores, for different amount of slave machines; that's because all metrics are independent from each other and Spark manages this kind of setup very well. The maximum performance obtained is when using the 125 CPU cores at our disposal, and corresponds to about 108 000 predicted metrics in 15 minutes, or 120 per second. In conclusion, one metric takes about one second to be predicted on one CPU core, end-to-end on the processing chain.

Figure 4 shows the CPU load and the memory used when running the experiment with 100 cores. The machines are not overloaded, which leaves room for other high-consumption processes such as Cassandra. Two things are worth noting: the CPU load reaches 100% at the beginning, which is due to

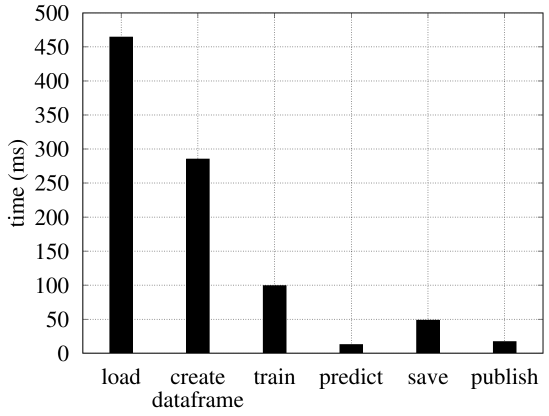


Fig. 5: Time repartition of the end-to-end process for predicting a metric (average with 90 000 metrics).

the Spark jobs start-up. Afterwards, both CPU and memory stay very stable: this is expected given the same work is done for every metric.

C. Time repartition

We also instrumented the different components of the processing chain, measuring the time taken by each of them, and averaging it on all the metrics. Figure 5 shows the results. Loading the data from Cassandra is what takes about half of the total time; this is expected since Cassandra is optimized for writes. Moreover, the network adds up to the latency. Creating a Spark Dataframe is quite resource-consuming too, but once it's created it's fast to work on it: the training and prediction times are relatively short. Finally, saving the data back into Cassandra and publishing an acknowledgement message to RabbitMQ is fast. End-to-end, the processing time of one metric on one CPU core is about one second. This is an acceptable time given our requirements, and is way below the prediction horizon (a few hours).

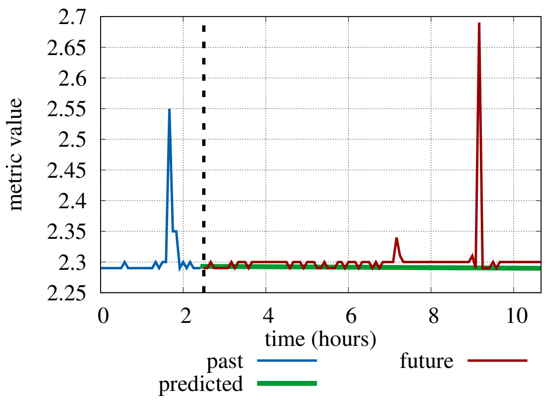
D. Load handling

Using these previous results:

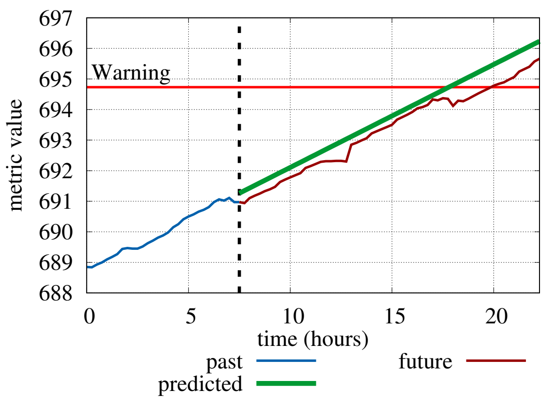
- It takes 1 second to predict a metric (end-to-end);
- There are on average 17 monitored services per machine (in the case of our dataset, a machine is either physical or virtual);

and taking a pessimist average of 1 minute for the metric sampling period (1 minute is usually the minimum period, and most metrics don't gain to be checked that often), we deduce we can handle $60 \times 24 = 1440$ metrics on a 24-core server. That means such a server can handle all the predictions for about 85 machines, which is a very acceptable ratio. Since this system scales linearly, increasing the number of cores will automatically increase the metrics load a server can handle.

It is important to note that the described servers have monitoring storage and prediction as their only role: they do not run other monitoring software, user interface dashboards, etc. If a 24-core server seems a lot to handle 85 machines (or 1440 metrics), it appears the performances are better in



(a) swap memory



(b) disk partition

Fig. 6: Measurements and predictions for two different metrics.

practice: the period is higher than 1 minute for most metrics, and the black-listing process will reduce the load in any case for less predictable metrics.

E. Predictions accuracy

Finally, we measured the root-mean-square error (RMSE) for every predicted metric. Due to outliers and too volatile metrics, it is quite high: its average is 822, with a standard deviation of $23\,687$. However its median is at 0.0001: this tells us most of the metrics have a very low RMSE. If we decide a good prediction has an $RMSE < 0.02$, when we filter the results to keep only those below this value, we get an average at 0.001, and a standard deviation at 0.003. More interestingly, 58.5% of the metrics fall within that range, which is promising for the benefits of linear regression over this system.

Figure 6 shows two examples of metrics measurements and predictions. A vertical dotted line separates the training values from both the predicted ones and what was actually measured. The first one is the swap memory of a machine, with up spikes probably due to the kernel swapping memory before freeing it immediately afterwards. Linear regression can't predict spikes, and that is something we intend to try with other, more complex, machine learning algorithms. Note that the y-scale doesn't begin at 0, hence the spikes are smaller than what they appear. In these cases, not predicting spikes is a feature, as they are sudden increases (or decreases) that are back to normal almost immediately afterwards; hence we avoid raising false positive alerts.

The second figure represents how full a disk partition is. This time an actual problem is detected: the warning threshold is reached. It is predicted a bit sooner than the actual problem occurence, but this difference would have trimmed down when getting new data and updating predictions.

IV. Related work

A. Time series

Time series forecasting is essential in many fields. Auto regressive (AR) and moving average (MA) are two of the very first linear statistical approaches of time series forecasting [5]. Auto Regressive Integrated Moving Average (ARIMA) is a linear forecasting model, which includes both AR and MA while considering trends in the time series [6], [5]. With advances in probabilistic machine learning, many studies utilize machine learning algorithms for time series forecasting [7] along with statistical approaches [8]. Support vector machines generalize well in high dimension [9]. With regression extension, many studies used support vector regression for time series prediction [10], [11], [12], [13], [14], [15]. Random Forest is an ensemble of weak learners [16] which results in good generalization. They are used in many time series prediction applications [17], [18], [15].

B. Monitoring

Some industrial companies have implemented prediction systems to prevent failures. Zabbix [19] uses different models to predict the future value of a given metric, and hence predict when a critical threshold will be reached. Triggers compute the predicted values each time a new metric value arrives. However, the choice of the model and the parameters tuning has to be done manually for each metric, which is resource-consuming and easily leads to errors. In our system, we use cross-validation to automatically tune the parameters and we blacklist metrics that are not good candidates for this system.

Other prediction systems focus on parallel problems. For example, Chalermarrewong et al. use time series analysis for hardware failure prediction [20]. They use self adjusting multi-step ARMA to predict the future values which updates the model according to paired t-test.

In the capacity planning domain, Azure [21] defines a set of machine learning algorithms to compute predicted values; which are however not predicted in real time.

Thermocast [22] focuses on predicting the thermal parameters of datacenters. Their approach is similar, but it solves a different issue and doesn't look at server problems but rather the temperature of the various equipment.

Many systems (e.g. [23], [24]) leverage elastic computing to predict Service-Level Agreement issues and provision enough resources for anticipated workloads. Our solution is orthogonal to this problem, since some monitoring issues can be consequences of changes in workloads, but not all of them.

Singh et al. describe an architecture for storing monitoring data and update a wiki engine with the collected information [25]. They present some similarities with our work in their choice of distributed engines, notably Spark and MLlib.

V. Conclusion and future work

Monitoring machines helps determining which services are down, and which resources need to be upgraded. It also generates a lot of time series points: thousands of metrics, as diverse as a CPU load or a database latency, constitute a set of metrics which can be leveraged for prediction purposes. However, predicting the future behavior of monitoring metrics poses a few challenges, the noteworthy ones being *accuracy* and *scalability*.

We described and evaluated a system for leveraging this opportunity, choosing linear regression as the main prediction algorithm, for its simplicity and relevance for most of the observed metrics. We detailed the inner workings of linear regression, as opposed to the threshold system in place within most monitoring software suites; as well as the main components revolving around it: persistence storage in a Cassandra database, work distribution among servers with Spark, and blacklisting of less predictable metrics. The detailed evaluation gave us great insights on this system, notably about its scalability (it scales linearly up to at least 125 cores) and speed (the end-to-end processing chain takes about one second on one CPU core to predict one metric). We believe this system adds a great value to monitoring tools, by giving system administrators information about potential future problems and downtimes.

In the future, we plan to experiment with deep learning, in order to focus on long term global trends, rather than local ones identified by linear regression. For blacklisted metrics which are not good fits for linear regression, we plan to implement other machine learning algorithms, and compare the performances obtained. Lastly, we envisage to plug this system to a ticketing mechanism used by clients to report problems, in order to match them with their host errors.

Acknowledgments

This work has been supported by the FEDER EU projects: Smart Support Center and Virtual Studio.

References

[1] A. Lakshman and P. Malik, “Cassandra: a decentralized structured storage system,” *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[3] N. Marz and J. Warren, *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*, 1st ed. Greenwich, CT, USA: Manning Publications Co., 2015.

[4] Adsquare. (2016). [Online]. Available: http://www.adsquare.com/comparing-performance-of-spark-dataframes-api-to-spark-rdd/

[5] C. Chatfield, *The Analysis of Time Series: An Introduction, Sixth Edition*, ser. Chapman & Hall/CRC Texts in Statistical Science. Taylor & Francis, 2003. [Online]. Available: https://books.google.fr/books?id=I367jgEACAAJ

[6] G. Box and G. Jenkins, *Time series analysis: forecasting and control*, ser. Holden-Day series in time series analysis. Holden-Day, 1970. [Online]. Available: https://books.google.fr/books?id=qsnaAAAAMAAJ

[7] G. Bontempi, S. B. Taieb, and Y.-A. Le Borgne, “Machine learning strategies for time series forecasting,” in *Business Intelligence*. Springer, 2013, pp. 62–77.

[8] J. G. De Gooijer and R. J. Hyndman, “25 years of time series forecasting,” *International journal of forecasting*, vol. 22, no. 3, pp. 443–473, 2006.

[9] V. Vapnik, “The nature of statistical learning theory,” 1995.

[10] K.-R. Müller, A. J. Smola, G. Rätsch, B. Schölkopf, J. Kohlmorgen, and V. Vapnik, “Predicting time series with support vector machines,” in *International Conference on Artificial Neural Networks*. Springer, 1997, pp. 999–1004.

[11] S. Mukherjee, E. Osuna, and F. Girosi, “Nonlinear prediction of chaotic time series using support vector machines,” in *Neural Networks for Signal Processing VII. Proceedings of the 1997 IEEE Signal Processing Society Workshop*, Sep 1997, pp. 511–520.

[12] T. Raicharoen, C. Lursinsap, and P. Sanguanbhoki, “Application of critical support vector machine to time series prediction. circuits and systems. iscas’03,” in *Proceedings of the 2003 International Symposium on*, vol. 5, 2003, pp. 25–28.

[13] T. V. Gestel, J. A. K. Suykens, D. E. Baestaens, A. Lambrechts, G. Lanckriet, B. Vandaele, B. D. Moor, and J. Vandewalle, “Financial time series prediction using least squares support vector machines within the evidence framework,” *IEEE Transactions on Neural Networks*, vol. 12, no. 4, pp. 809–821, Jul 2001.

[14] Y. Fan, P. Li, and Z. Song, “Dynamic least squares support vector machine,” in *Intelligent Control and Automation, 2006. WCICA 2006. The Sixth World Congress on*, vol. 1. IEEE, 2006, pp. 4886–4889.

[15] H. Borchani, G. Varando, C. Bielza, and P. Larrañaga, “A survey on multi-output regression,” *Wiley Int. Rev. Data Min. and Knowl. Disc.*, vol. 5, no. 5, pp. 216–233, Sep. 2015.

[16] L. Breiman, “Random forests,” *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[17] G. G. Creamer and Y. Freund, “Predicting performance and quantifying corporate governance risk for latin american adrs and banks,” *Financial Engineering and Applications*, 2004.

[18] M. J. Kane, N. Price, M. Scotch, and P. Rabinowitz, “Comparison of arima and random forest time series models for prediction of avian influenza H5N1 outbreaks,” *BMC Bioinformatics*, vol. abs/1412.6980, 2014.

[19] Z. P. triggers. (2017). [Online]. Available: https://www.zabbix.com/documentation/3.0/manual/config/triggers/ prediction

[20] T. Chalermarrewong, T. Achalakul, and S. C. W. See, “Failure prediction of data centers using time series and fault tree analysis,” in *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, Dec 2012, pp. 794–799.

[21] M. C. Azure. (2017). [Online]. Available: https://docs.microsoft.com/en-us/azure/machine-learning/machine-learning-algorithm-choice

[22] L. Li, C.-J. M. Liang, J. Liu, S. Nath, A. Terzis, and C. Faloutsos, “Thermocast: A cyber-physical forecasting model for datacenters,” in *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD ’11. New York, NY, USA: ACM, 2011, pp. 1370–1378. [Online]. Available: http://doi.acm.org/10.1145/2020408.2020611

[23] P. Leitner, A. Michlmayr, F. Rosenberg, and S. Dustdar, “Monitoring, prediction and prevention of sla violations in composite services,” in *2010 IEEE International Conference on Web Services*, July 2010, pp. 369–376.

[24] W. Fang, Z. Lu, J. Wu, and Z. Cao, “Rpps: A novel resource prediction and provisioning scheme in cloud data center,” in *2012 IEEE Ninth International Conference on Services Computing*, June 2012, pp. 609–616.

[25] S. Singh and Y. Liu, “A cloud service architecture for analyzing big monitoring data,” *Tsinghua Science and Technology*, vol. 21, no. 1, pp. 55–70, Feb 2016.