cCube: A Cloud Microservices Architecture for Evolutionary Machine Learning Classification

Pasquale Salza
University of Salerno, Italy
psalza@unisa.it

Erik Hemberg
Massachusetts Institute of Technology, USA
hembergerik@csail.mit.edu

Filomena Ferrucci
University of Salerno, Italy
fferrucci@unisa.it

Una-May O'Reilly
Massachusetts Institute of Technology, USA
unamay@csail.mit.edu

ABSTRACT

We present *cCube*, a microservices open source architecture used to automatically create an application of one or more Evolutionary Machine Learning (EML) classification algorithms that can be deployed to the cloud with automatic data factorization, training, result filtering and fusion.

CCS CONCEPTS

•Computing methodologies → Machine learning; •Theory of computation → *Evolutionary algorithms;* **•Software and its engineering →** *Cloud computing;*

KEYWORDS

Evolutionary Machine Learning, Microservices, Cloud Computing

ACM Reference format:
Pasquale Salza, Erik Hemberg, Filomena Ferrucci, and Una-May O'Reilly. 2017. cCube: A Cloud Microservices Architecture for Evolutionary Machine Learning Classification . In *Proceedings of GECCO '17 Companion, Berlin, Germany, July 15-19, 2017,* 2 pages.
DOI: http://dx.doi.org/10.1145/3067695.3076089

1 INTRODUCTION

We present *cCube* (compare, compete, collaborate), an open source architecture helping its users develop an application to deploy EML algorithms to the cloud. The source code is shared at the address https://github.com/ccube-eml, under the terms of the *MIT License*.

cCube supports competition and collaboration as with FCUBE [1], a prior project that has addressed the challenge of the EML community, collaboratively developing a compendium solution to a "big data" classification problem. Despite being noteworthy, FCUBE architecture is "locked" into one cloud provider, i.e., *Amazon AWS*, and lacks both automation and robust fault tolerance.

A *cCube* EML application factors data, handles parameter configuration, tasks parallel classifier training with different algorithms, filtering and fusing classifier results into a final ensemble model. *cCube* serves 3 types of users: EML algorithm designer, multi-algorithm EML application manager and "black box" end users. In *cCube* researchers can run different EML algorithms, their own as well as others', developed in different programming languages, without inserting any code into them to accommodate cloud scaling. *cCube* also supports crowdfunding, i.e., the sharing of costs by a collaborative group that wishes to execute a large multi-learner, factor, filter, and fuse application.

Instead of being monolithic, *cCube* has a *microservices* architecture [4], each running its own process and communicating with lightweight protocols, e.g., HTTP resource API and message queues. Each service is independently deployable in a fully automated way making applications easier to scale and more fault tolerant. For all of its microservices, *cCube* uses lightweight runtime environments in the form of "software containers" [2], automatically deployed using *Docker* and *Docker Swarm*. By using Docker, we implicitly made *cCube* flexible against the limitations of the quantity of machines the providers usually impose upon their users, through an infrastructure definable as "multi-cloud", i.e., based on the allocation of instances by different cloud providers but participating to the same system [3].

2 *cCube* ARCHITECTURE

We designed the *cCube* architecture devising several microservices, all managed by the client `Orchestrator`. It creates and provisions the compute units, initiating and directing the symphony of microservices. It uses a bridge pattern [5] to interface with different cloud providers, e.g., *OpenStack* and *Amazon EC2*.

To build a *cCube* application, the EML developers copy a template from *cCube*'s repository and customize configuration, e.g., EML algorithm invocation. Then, *cCube* runs as daemon instructed to manage the communication with the other microservices of the system. The only requirement for the connection with *cCube* is the definition of some predefined environment variables, defining how the *cCube* daemon can interact with the EML algorithm and its outcome for both the learning and prediction phases.

The developers then become end users and start the `Orchestrator` on their machine, provide keys for authorization, thus keeping their sensitive information local and secure. The client, through Docker, starts the application after provisioning resources, running resource discovery and set up. All the produced output is stored into a support database, e.g., *MongoDB*.

In detail, we defined the following microservices, whose overview is shown in Figure 1.

GECCO '17 Companion, Berlin, Germany
 978-1-4503-4939-0/17/07…$15.00
DOI: http://dx.doi.org/10.1145/3067695.3076089

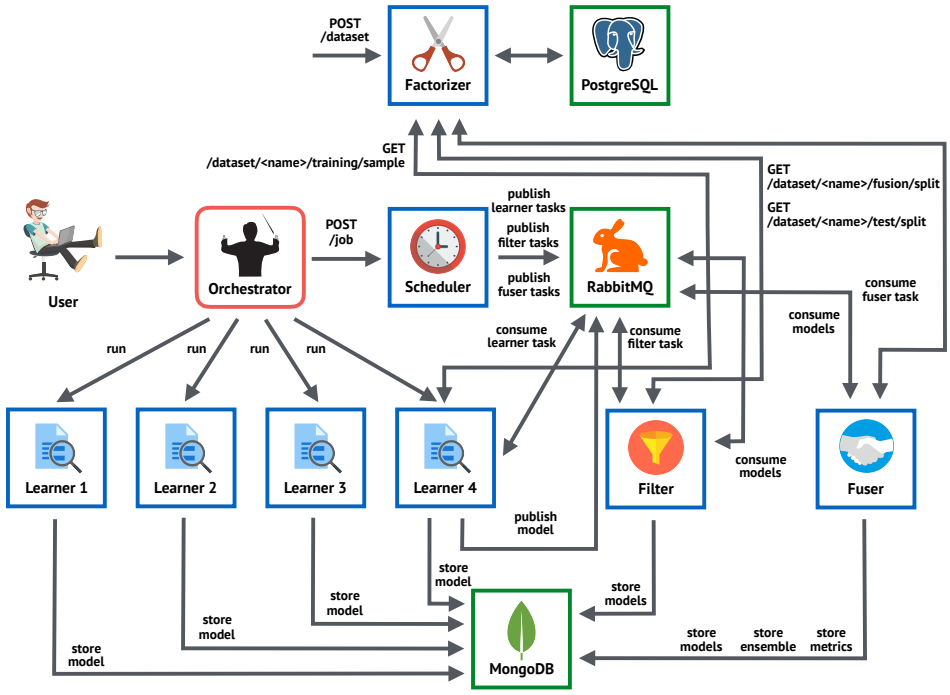

Figure 1: Under the hood of the *cCube* black box there is a microservices and containers architecture.

`Factorizer`: it interfaces with the storage, e.g., *PostgreSQL*, exposing its services through a REST interface, thus the communication happens using HTTP. This allows the data set upload and then the data set can be split into separate parts on demand and following the components needs, e.g., to use for training, fusion, and testing.

`Scheduler`: it accepts jobs through a REST interface. A job is considered as a chain of processes executed on a *cCube* cluster, involving all the components, i.e., microservices. Once a new job is requested, the `Scheduler` creates the tasks for the `Learners`, `Filter` and `Fuser` and publish them on different message queues in `JSON` format. In case of a failed microservice, the task would be put again into the queue and that computation run again by another container. Moreover, the tasks contain all the relevant information needed for running an activity, e.g., the target dataset name, the parameters of separation for training, fusion and testing, features to include/exclude, duration. A producer/consumer pattern is controlled by a "queue manager", e.g., *RabbitMQ*.

`Learner`: each of the multiple learners consumes a task, training data sample and parameters, which are possibly generated at random when the task is created by the `Scheduler`, allowing also a parameters factorization. First, a data sample for training is given by the `Factorizer` and, once received, the `Learner` executes the EML algorithm according to a given duration, i.e., one of the parameters included in the task. Then, the computed model is compressed and stored in a message and sent to a queue for outputs.

`Filter`: it consumes the outputs until the last expected model arrives. Then, the models are executed on a split of the data that has been set aside, according to a filter policy. After the set of filtered models is complete, it is published as a single message to another queue for fusing.

`Fuser`: filtered models are eventually fused together to collaborate in an ensemble model. A fusion task is consumed, providing the needed information. Data for fusion comes from the `Factorizer`, i.e., the same split used during the filter phase. Then, another data split is used to test the ensemble and computing some predictive performance metrics.

3 CONCLUSIONS AND FUTURE WORK

We presented *cCube*, an open source architecture for the comparison, competition and multi-party collaboration of EML algorithms and users. As future work, we plan to implement and experiment *cCube*, using different use cases, algorithms, and cloud providers.

REFERENCES

[1] Ignacio Arnaldo, Kalyan Veeramachaneni, Andrew Song, and Una-May O'Reilly. 2015. Bring Your Own Learner: A Cloud-Based, Data-Parallel Commons for Machine Learning. *IEEE Computational Intelligence Magazine* 10, 1 (Feb. 2015), 20–32.

[2] Erhan Ekici. 2014. Microservices Architecture, Containers and Docker. (Dec. 2014). https://www.ibm.com/developerworks/community/blogs/1ba56fe3-efad-432f-a1ab-58ba3910b073/entry/microservices_architecture_containers_and_docker

[3] Nicolas Ferry, Alessandro Rossini, Franck Chauvel, Brice Morin, and Arnor Solberg. 2014. Towards Model-Driven Provisioning, Deployment, Monitoring, and Adaptation of Multi-Cloud Systems. In *IEEE International Conference on Cloud Computing (CLOUD)*. 887–894.

[4] James Lewis and Martin Fowler. 2014. Microservices, a Definition of This New Architectural Term. (March 2014). https://martinfowler.com/articles/microservices.html

[5] John Vlissides, Richard Helm, Ralph Johnson, and Erich Gamma. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.