

GABRIEL DE OLIVEIRA RIBEIRO

Advisor: Joubert de Castro Lima

Co-Advisor: Lauro Moraes

**LEARNING ORCHESTRA: BUILDING MACHINE
LEARNING WORKFLOWS ON SCALABLE
CONTAINERS**

Ouro Preto
April of 2021

FEDERAL UNIVERSITY OF OURO PRETO
INSTITUTE OF EXACT SCIENCES AND BIOLOGY
UNDERGRADUATE PROGRAM IN COMPUTER SCIENCE

Monograph presented to the Undergraduate
Program in Computer Science of the Federal
University of Ouro Preto in partial fulfillment
of the requirements for the degree of Bachelor
in Computer Science.

GABRIEL DE OLIVEIRA RIBEIRO

Ouro Preto
April of 2021



MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO



FOLHA DE APROVAÇÃO

Gabriel de Oliveira Ribeiro

Learning Orchestra: building Machine Learning workflows on scalable containers

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como requisito parcial para obtenção do título de Bacharel em Ciência da Computação

Aprovada em 22 de Abril de 2021.

Membros da banca

Joubert de Castro Lima (Orientador) - Doutor - Universidade Federal de Ouro Preto
Lauro Ângelo Gonçalves de Moraes (Coorientador) - Mestre - Universidade Federal de Ouro Preto
Milton Stiilpen Júnior (Examinador) - Mestre - Stilingue
Felipe Melo (Examinador) - Mestre - Absa Group

Joubert de Castro Lima, orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 22/04/2021.



Documento assinado eletronicamente por **Joubert de Castro Lima, PROFESSOR 3 GRAU**, em 23/04/2021, às 10:01, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do [Decreto nº 8.539, de 8 de outubro de 2015](#).



A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **0160354** e o código CRC **C610A273**.

Referência: Caso responda este documento, indicar expressamente o Processo nº 23109.003505/2021-56

SEI nº 0160354

R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35400-000
Telefone: 3135591692 - www.ufop.br

Resumo

Esforços interessantes foram realizados nas últimas duas décadas para construir ferramentas que facilitam e agilizam o desenvolvimento de workflows de Machine Learning (ML) compostos de vários pipelines. De scripts Unix à componentes de ML desenhados para Web e soluções para automatizar e orquestrar etapas de ML e Data Mining (DM), nós tentamos muitos serviços de alto nível para o processo iterativo do cientista de dados. De outro lado, nós temos os serviços de baixo nível sendo investigados com os mesmos propósitos, como ambientes em nuvem, orquestração de contêineres, tolerância a falhas, etc. Normalmente, scripts são produzidos para simplificar as operações dos serviços de baixo nível. Infelizmente, nenhuma solução existente coloca ambos os serviços - alto e baixo níveis - na mesma instalação. Além disso, nenhum deles possibilita a utilização de diferentes ferramentas durante a construção de um único workflow. Tais soluções não são, portanto, flexíveis o suficientes para permitir uma ferramenta, como o Scikit-Learn, construir um pipeline, e outras ferramentas de ML, como TensorFlow ou Spark MLlib, construírem outros pipelines. Uma Application Programming Interface (API) Representational State Transfer (REST) interoperável é muito útil para expor esses serviços, mas apenas algumas alternativas atendem esse requisito. Para suprir tais limitações, nós apresentamos um sistema open source denominado Learning Orchestra, uma solução para construir workflows complexos usando diferentes ferramentas de ML de forma transparente. Com uma única e interoperável API é possível construir fluxos analíticos que podem ser instalados em ambientes com contêineres em nuvens computacionais capazes de escalar e tolerar falhas. Experimentos demonstraram que nosso sistema é uma alternativa promissora e inovadora para o problema de simplificar e agilizar o processo iterativo de ML.

Palavras Chave: Aprendizado de Máquina. Mineração de Dados. Sistemas Distribuídos. Arquitetura de Microserviços. Computação em Nuvem. Workflow. Pipeline. Contêiner-

ização.

Abstract

Interesting efforts were done to construct tools to facilitate and streamline the development of Machine Learning (ML) workflows composed of several pipelines in the last two decades. From Unix scripts to Web based ML components and solutions to automate and orchestrate ML and Data Mining (DM) pipes, we have tried many high level services for the data scientist iterative process. On the other hand, we have the low level services being investigated, like cloud environments, container orchestration, fault tolerance service and so forth. Normally, scripts are produced to simplify such low level services operations. Unfortunately, no existing solution put both low and high level services on a unique service stack. Furthermore, none of them enables the utilization of different existing tools during the construction of a single workflow, i.e., they are not flexible enough to permit a tool, like Scikit-learn, to build one pipeline and other ML tools, such as Spark MLlib or TensorFlow, to build other pipes. A Representational State Transfer (REST) interoperable Application Programming Interface (API) is very useful to expose these tools services, but few existing alternatives attend this requirement. To address these limitations, we present the open source Learning Orchestra system, a solution to construct complex workflows using different ML tools or players transparently, i.e., from a single interoperable API we can build interesting analytical flows. The workflows can be deployed on a containerized cloud environment capable to scale and be resilient. Experiments demonstrated that our system is a promising and innovative alternative for the problem of simplify and streamline the ML iterative process.

Keywords: Machine Learning. Data Mining. Distributed System. Microservice Architecture. Cloud Computing. Workflow. Pipeline. Containerization.

I dedicate this hard work to my mother, Silvana, and my father, Eleotério, who support me on all undergraduate period.

Acknowledgment

I want to thank the Universidade Federal de Ouro Preto (UFOP) for the high-level quality computer science course and my advisors Joubert and Lauro for the good moments and the great teaching. This project can be finished sometime in the future, but the friendship among us will be kept. I also thank the Laboratório Multiusuários de Bioinformática of NUPEB for the support and CAPES to allow access to the great academic publications used in this work.

Contents

1	Introduction	1
1.1	Goal	2
1.2	Scope	2
1.3	Work Organization	3
2	Basic Concepts	4
2.1	Cloud Computing	4
2.2	Virtual Machine	4
2.3	Application Programming Interface	5
2.4	Microservices	5
2.5	Service Orchestration and Choreography	6
2.6	Container	6
2.6.1	Docker	6
2.6.2	Docker Swarm	7
2.7	Spark processing solution	7
2.8	MongoDB - No SQL storage	7
2.9	Kafka publishers and subscribers	8
2.10	Scikit-learn Python solution	8
2.11	TensorFlow - ML at scale	8
2.12	Keras	9
2.13	Data Mining and Machine Learning - concepts and differences	9
3	Related Work	11
4	Development	18
4.1	Architecture	18
4.1.1	The Virtual Machine (VM) Layer	19

4.1.2	Container Orchestrator	20
4.1.3	Containers	20
4.1.4	Existing Libraries, Frameworks and Middlewares	22
4.1.5	Services	22
4.1.6	API Gateway	24
4.1.7	Learning Orchestra REST API	25
4.1.8	Learning Orchestra Clients	37
4.2	Pipeline Examples	37
4.2.1	A Pipeline using the Builder service	37
4.2.2	A Text Classification Pipeline Example	47
4.2.3	An Image Classification Pipeline Example	54
5	Experiments	59
5.1	Setup	59
5.1.1	Container setup for Titanic pipeline	59
5.1.2	Container setup for IMDb and MNIST pipelines	61
5.2	Metrics	61
5.3	Classifiers	62
5.4	Titanic Experiment	63
5.4.1	Dataset	63
5.4.2	Runtime Results	64
5.4.3	Memory Consumption Results	66
5.4.4	ML Metrics Results	67
5.4.5	Samples Prediction Results	67
5.4.6	Ranking survived Titanic results	68
5.5	IMDb Experiment	68
5.6	MNIST Experiment	70
6	Conclusion	71
6.1	Discussion	72
6.1.1	The good	72
6.1.2	The bad	72
6.1.3	The ugly	73
6.2	Future Directions	73

List of Figures

4.1	Learning Orchestra architecture	18
4.2	GCP cluster configuration example	19
4.3	Learning Orchestra container types	21
4.4	Learning Orchestra workflow composed of five pipelines	25
4.5	IMDb sample	47
5.1	First container configuration	60
5.2	Second container configuration	60
5.3	Third container configuration	61

List of Tables

5.1	VMs configuration	59
5.2	Classifiers training runtime	65
5.3	Classifiers train-validation runtime	65
5.4	Classifiers test runtime	65
5.5	Titanic runtime	66
5.6	Classifiers training runtime with one Spark W instance	66
5.7	Memory consumption	67
5.8	Prediction metrics comparison	67
5.9	Tuples Prediction Results	68
5.10	Classifiers score on Kaggle Challenge	68
5.11	IMDb solvers runtime and memory consumption	69
5.12	Metrics Scores for MNIST experiment	70

Acronyms

API Application Programming Interface. i, iii, vi, vii, 1–3, 5, 8, 16, 24–39, 41, 49, 53, 71, 72, 74

AuFS Advanced Multi-Layered Unification Filesystem. 7

CLI Command Line Interface. 22

CON Containerized, elastic and scalable. 12

CPU Central Processing Unity. 59, 66

CRUD Create, Retrieve, Update and Delete operations. 26

DM Data Mining. i, iii, vi, 9, 11–13

DT Decision Tree. 60, 64–68

FAULT Fault tolerance. 11

GB Gradient-Boosted Tree. 60, 64–68

GCP Google Cloud Platform. 59, 64

HTTP Hypertext Transfer Protocol. 5, 26, 73

INT Interoperability. 12

IP Internet Protocol. 59

JSON JavaScript Object Notation. 2, 5, 21, 25–27, 42, 71–73

KDD knowledge discovery from data. 9

LPMN Language Processing Modeling Notation. 13

LR Logistic Regression. 60, 64–68

LXC Linux Container. 6

ML Machine Learning. i, iii, vi, 1–3, 7, 9, 11–13, 19–25, 27, 31, 32, 36, 42, 47, 49, 66, 69–74

NAS Neural Architecture Search. 3

NB Naive Bayes. 60, 64–68

NLP Natural Language Processing. 12, 13

ORCH Service Orchestration or Choreography. 12

OS Operation System. 4–6, 19, 59, 62

RAM Random Access Memory. 59, 67

RDD resilient distributed dataset. 7

REST Representational State Transfer. i, iii, 2, 5, 26, 37, 38, 71

RF Random Forest. 60, 64–68

SML Specific ML components to build workflows. 12

SOAP Simple Object Access Protocol. 5

SQL Structured Query Language. 7

TS Timers and schedulers. 11

URL Uniform Resource Locator. 23–27, 42

vCPU Virtual CPU. 59

VM Virtual Machine. vi, x, 4, 6, 19, 20, 48, 49, 59, 61, 64–66, 69, 71–73

WEB Web Designs. 12

WEKA Waikato Environment for Knowledge Analysis. 12, 13

WP Workflow programming. 11

XML Extensible Markup Language. 5

Chapter 1

Introduction

The data scientist activity normally works in an iterative process composed of the following activities: gathering available data, performing exploratory data analysis, cleaning/enriching those data, building models, validating their predictions, and deploying results. The utilization of standalone tools to build workflows composed of analytical pipelines, like WEKA Hall et al. (2009), is not sufficient nowadays because the volume of data and the number of algorithms for classification, pre-processing, tuning and so forth increased a lot in the last decade. Even the distributed version of WEKA Koliopoulos et al. (2015) is not so realist because the deployment and other important low level activities are complex and very often not familiar for the data scientist.

Interesting efforts were done to construct tools to facilitate and streamline the development of Machine Learning (ML) workflows composed of several pipelines in the last two decades. From UNIX scripts Breck (2008) to general purpose containerized extensions for Pegasus Vahi et al. (2019) and Web based drag and drop R/Python components McClure (2018), we have tried to deliver many alternatives for the data scientist. Some works tried to automate the workflow construction, i.e., the orchestration of its pipes using annotations or any sort of high level descriptions from the specialist. These alternatives adopt planning and learning algorithms to suggest analytical workflows Beygelzimer et al. (2013); Walkowiak (2017); He et al. (2019).

We understand that the goal to facilitate and streamline the data scientist iterative process explained previously continues to have a gap. Users should load, optimize and execute useful existing ML models from companies like Google and Facebook, but they should also have an Application Programming Interface (API) to build their own pipelines in a traditional way. These efforts normally do not coexist with a simple deploy and execution

of an analytical infrastructure capable to be fault tolerant, scalable and other important issues that are complex for non specialists. In terms of heterogeneity, where interoperable ML microservices can be called via a Representational State Transfer (REST) Application Programming Interface (API) using different ML players, like Spark MLlib Meng et al. (2016), Scikit-learn Pedregosa et al. (2011) or TensorFlow Abadi et al. (2016), producing several pipelines deployed over containers on different cloud platforms, the literature alternatives do not present any system with all these benefits.

1.1 Goal

The goal of this work is to develop a tool, named Learning Orchestra, to reduce a bit more the existing gap in facilitate and streamline the data scientist iterative process composed of gather data, perform exploratory analyze, clean/enrich those data, build models, validate their predictions and deploy the results.

For that, the Learning Orchestra system enables the deployment and execution of several layers of software over several hardware architectures. It can be deployed/executed on market-leaders cloud platforms, resulting in elastic and on demand services. It is a containerized system, thus resilient, lightweight and fair in terms of workload. It offers containers orchestration transparently. It implements several ready to use services for ML entire pipelines or pipelines steps, like dataset, model, transform, tune, train, predict and observe, the last for notification purpose. It is interoperable, since its API sends/receives messages written in JavaScript Object Notation (JSON), so data scientists with little programming skills in any language can build complex analytical workflows. Finally, it puts together players, like Spark MLlib, Scikit-learn and TensorFlow in a single ML solution.

1.2 Scope

As we can see, it is out of scope of this work the development of new algorithms, methods or approaches for ML or any pipe of the previous presented iterative process, precisely gather available data, clean/enrich those data, build models, validate their predictions and deploy results.

It is out of scope any auto ML facility via Learning Orchestra, precisely the automatic execution of the pipeline presented on the auto ML survey work He et al. (2019). Of-course, if the ML tool, like Scikit-learn, Spark MLlib or TensorFlow, provides services like

Neural Architecture Search (NAS) Zoph and Le (2016) or hyperparameters optimization, for instance, the Learning Orchestra calls it like it calls any other model configuration strategy, being it manual or automatic. The data scientist needs just to configure the model and perform another API call to run it, but there is no auto ML pipe on our system API. It is part of future research investigations the support for such requirement, similar to Katib Zhou et al. (2019).

The Learning Orchestra system did not has specific API microservices to build neural networks from scratch, this way the data scientists can load existing ones on their pipelines. There is a service called Builder, where the data scientist can run an entire ML pipeline using the Python programming language, so we assume it can be also used to construct a neural network in TensorFlow, for instance, and run it after the building phase, but without any API facility in terms of deep learning on the Learning Orchestra.

1.3 Work Organization

The rest of this work is organized as follows: Chapter 2 discusses the basic concepts for a better understanding of the work. Chapter 3 presents the related work about tools for building analytical workflows. Chapter 4 details the architecture and design of the Learning Orchestra system. Chapter 5 details the experiments and experimental results. Finally, in Chapter 6 the conclusion and future research directions are described.

Chapter 2

Basic Concepts

In this chapter, all the basic concepts that are useful for a better understanding of the work are presented.

2.1 Cloud Computing

Cloud Computing offers a shared set of flexible and configurable computing resources, such as network, CPU, GPU, RAM, hard disk, but also software layers, such as Operation System (OS), basic software (Ex. database, compilers, etc.) and applications (Ex. stream processing tools, graph tools and so forth). These resources can be easily deployed and used at scale with minimal efforts Mell et al. (2011).

A single hardware can be abstracted as several small devices or a cluster of machines can be abstracted as a single device. The elasticity is another fundamental property in cloud environments, so the user can increase the processing or storage capacities, as well as any other hardware or software configuration on-the-fly, thus cloud computing became very useful for high availability demands.

2.2 Virtual Machine

The Virtual Machine (VM) is the core of cloud computing and it is an emulated computer system with all the layers (hardware, OS and applications) running over another computer system. This architectural design provides efficiency and isolation for users Goldberg (1974).

A hypervisor (or virtual machine monitor - VMM) is a computer software, firmware or hardware that creates and runs VMs. A computer on which a hypervisor runs one or more VMs is called a host machine, and each VM is called a guest machine. The hypervisor presents the guest OS with a virtual operating platform and manages the execution of the guest OS. Multiple instances of a variety of OS may share the virtualized hardware resources. Any application can be deployed over such OS, being also virtualized.

2.3 Application Programming Interface

The Application Programming Interface (API) is used by computer systems for their communications, thus it defines how to make requests, the used data types, and how to interact to allow system communication ¹. The API can be customized for specific utilization, such as interoperability or transaction services.

In Web context, the API use Hypertext Transfer Protocol (HTTP) protocol to send the content of requests and responses. The common used format is the Extensible Markup Language (XML) or JavaScript Object Notation (JSON). The communication mechanism of Web APIs are migrating from Simple Object Access Protocol (SOAP) to Representational State Transfer (REST), thus a transaction or a request can take long periods to occur (hours, days or even months), a fundamental requirement for justice services, government services, scientific experiments and many more.

2.4 Microservices

One of several definitions of Microservices is:

"A Microservice is an independently deployable component of bounded scope that supports interoperability through message-based communication. Microservice Architecture is a style of engineering highly automated and where software systems are made up of capability-aligned microservices." Nadareishvili et al. (2016)

Microservices are small in size, being more cohesive, if compared with a monolithic architecture. Web APIs are often used to intercommunicate microservices. This software architecture enables the development of low coupling components and each of them can be developed at different stage. Normally, each microservice has its metadata that is very

¹https://en.wikipedia.org/wiki/Application_programming_interface

useful for both service discovery and service composition, a fundamental activity of service orchestration explained further in this chapter.

2.5 Service Orchestration and Choreography

Service orchestration refers to a centralized process responsible to interact and manage microservices. The interactions happen at message level and the orchestration includes business logic, including service composition, and task execution order Peltz (2003), this way the microservices must be flexible and adaptable to business needs.

Choreography services represent a decentralized architecture, where each participant, i.e., each microservice has a communication pattern focused in collaboration, thus elections and other decision strategies are used Busi et al. (2005). The same orchestration responsibilities (service discover, service composition, task ordering, etc.) are included in choreography, but in a distributed way.

2.6 Container

The container is also a virtualization layer and used together with VMs, since the containers operate at OS level Merkel (2014). In containers, there are abstraction layers for process, network, hard disks and so forth. Containers running on the same machine preserve the isolation, thus memory, processor and disk are not visible among them.

As we can see, containers are more light than VMs, this way faster during deployment, creation and destruction of computer environments. The VM requires kernel level, OS libraries, drivers and many more, so even when it is not running it consumes a huge amount of memory. Normally, containers are fault tolerant, so if a containerized application crashes for any reason, the container can call any procedure to restart it. The recover service in terms of backlogs and operations ordering are domain application responsibilities.

2.6.1 Docker

Docker is one of the market-leaders container technology, which uses few features from Linux kernel to provide a lightweight tool to manage the life-cycle of many applications Merkel (2014). Docker uses Linux containers and Linux Container (LXC), a package from Linux containers to provide user-namespaces, separating container's database and network

from host. Docker adopts Advanced Multi-Layered Unification Filesystem (AuFS) as its file system. It is a layered file system, enabling Docker containers to deploy several customized images derived from the same image. The LXC also provides *cgroups*, a package used to both analyze and limit containers resources, such as memory usage, disk space and I/O bandwidth.

Docker also provides a Web repository where there are images from several frameworks, OS, and programming languages (Ex. Python and MongoDB images). The most common way to manage Docker containers is through command line, but it also has a REST API.

2.6.2 Docker Swarm

Docker Swarm is a clustering tool, orchestrating Docker containers (nodes) into a virtual Docker system Naik (2016). It adopts nodes to provide a redundancy system in the case of processing failure. The Docker Swarm has Manager and Worker nodes, where the manager node allows the cluster state management, precisely the node deploy, update, and remove services in an existing cluster of nodes. On the other hand, the worker nodes are responsible for running tasks.

2.7 Spark processing solution

Spark is a cluster computing framework which supports applications with specific tasks, such as ML, stream and Structured Query Language (SQL) ones. It also provides scalability and fault tolerance services transparently, being faster than another market-leader named Hadoop in ML operations Zaharia et al. (2010). The main abstraction of Spark is the resilient distributed dataset (RDD), which stores an object collection across machines in a cluster.

2.8 MongoDB - No SQL storage

MongoDB is a powerful, flexible, and scalable general-purpose database. It combines the ability to scale out with features, such as secondary indexes, range queries, sorting, aggregations, and geo-spatial indexes Chodorow (2013). The MongoDB, as Non-relational database, uses the document concept to storage the rows or registers. The document has no predefined schemas, this way the document keys do not require prefixed types and sizes, thus they allow insertions or removals of fields more freely.

2.9 Kafka publishers and subscribers

Kafka is an open-source distributed event streaming platform developed for collecting and delivering high volumes of log data with low latency Kreps et al. (2011). It joins the benefits of traditional messaging systems and log aggregators, providing an API similar to message systems, allowing applications to consume log events in real time. Inside the main Kafka concepts, there are publishers, topics, brokers, partitions and subscribers, where the publishers publish messages to specific abstractions called topics. The topic content is stored in a broker, which is a set of Kafka servers. A topic can be fragmented into several partitions deployed on several brokers to provide scalable write/read operations. Finally, the subscriber can register to a topic to be notified when the topic changes.

2.10 Scikit-learn Python solution

Scikit-learn is a rich environment to provide state-of-the-art implementations of many well known machine learning algorithms, while maintaining an easy-to-use interface tightly integrated with the Python language Pedregosa et al. (2011).

It differs from other machine learning toolboxes in Python for various reasons: i) it is distributed under the BSD license ii) it incorporates compiled code for efficiency, iii) it depends only on numpy and scipy to facilitate easy distribution, and iv) it focuses on imperative programming, not using, for instance, data-flow based approaches.

2.11 TensorFlow - ML at scale

TensorFlow is a machine learning system that operates at large scale and in heterogeneous environments. TensorFlow uses data-flow graphs to represent computation, shared state, and the operations that mutate that state. It maps the nodes of a data-flow graph across many machines in a cluster, and within a machine across multiple computational devices, including multicore CPUs, general purpose GPUs, and custom-designed ASICs known as Tensor Processing Units (TPUs) Abadi et al. (2016).

Unlike traditional data-flow systems, in which graph vertices represent functional computation on immutable data, TensorFlow allows vertices to represent computations that own or update mutable state. Edges carry tensors (multi-dimensional arrays) between nodes, and TensorFlow transparently inserts the appropriate communication between dis-

tributed subcomputations. This way, TensorFlow allows programmers to experiment with different parallelization schemes that, for example, offload computation onto the servers that hold the shared state to reduce the amount of network traffic.

2.12 Keras

Keras is an open-source software library that provides a Python API that is simple, flexible, and consistent API for developing deep learning workflows. Keras acts as an interface for the TensorFlow solution.

Up until version 2.3 Keras supported multiple backends, but as of version 2.4, only TensorFlow is supported. Designed to enable fast experimentation with deep neural networks, it focuses on being user-friendly, modular, and extensible. Keras (2021)

2.13 Data Mining and Machine Learning - concepts and differences

Data Mining (DM) is the process of discovering interesting patterns and knowledge from data. The data sources can include databases, data warehouses, the Web, other information repositories, or data that are streamed into the system dynamically Han et al. (2011). The DM task is a step in a knowledge discovery from data (KDD) process composed of several steps, like selection, pre-processing, transformation, interpretation and so forth. As a highly application-driven domain, DM has incorporated many techniques from other domains such as statistics, Machine Learning, pattern recognition, database and data warehouse systems, information retrieval, visualization, algorithms, high- performance computing, and many application domains Han et al. (2011).

ML investigates how computers can learn (or improve their performance) based on data. It is divided into four main classes:

- Supervised learning - a set of labeled data is used as input to ML algorithms, so they can learn the patterns previously, making predictions on non labeled data based on a train process;
- Unsupervised learning - these ML algorithms can build an internal structure to recognize the underlying patterns on unlabeled data;

-
- Semi-supervised learning - the algorithms make use of both labeled and unlabeled examples when learning a model. In one approach, labeled examples are used to learn class models and unlabeled examples are used to refine the boundaries between classes;
 - Active learning - it is a type of iterative supervised learning approach that lets users play an active role in the learning process. An active learning approach can ask a user (e.g., a domain expert) to label an example, which may be from a set of unlabeled examples or synthesized by the learning program. The goal is to optimize the model quality by actively acquiring knowledge from human users, given a constraint on how many examples they can be asked to label Han et al. (2011).

Chapter 3

Related Work

In this chapter, it is detailed the most similar papers of literature, where the core topic is the development of Machine Learning/Data Mining workflows. We are interested in manual or automated, visual or programmatic development of analytical workflow services, such as gathering available data, cleaning/preparing those data, building models, validating their predictions and deploying results. Besides that, we are also interested in low level service facilities, such as cloud services, containerized solutions, fault tolerance, scalability and container orchestration services provided for data scientists transparently.

We have searched the following paper repositories to build the related work: IEEE, ACM and ISI Web of Science. The Google search engine was also used to find white papers and magazine reports. The keywords searched were:

(workflow OR pipeline) AND (machine learning OR data mining) AND (middleware OR framework OR library OR tool)

All the related works are evaluated according to the following requirements:

- Fault tolerance (FAULT)- evaluates if the solution is resilient regarding data storage and service processing;
- Timers and schedulers (TS) - evaluates if it is possible to attach timer and scheduler components on ML/DM workflows;
- Workflow programming (WP) - evaluates if the user can build workflows programmatically or visually;

- Specific ML components to build workflows (SML) - evaluates if the solution has ready and configurable ML/DM services to build analytical workflows;
- Interoperability (INT) - evaluates if the solution enables several existing ML/DM solutions to work together, i.e., to interoperate on workflows, precisely on several pipelines;
- Web Designs (WEB) - evaluates if the solution is designed for Web;
- Containerized, elastic and scalable (CON) - evaluates if the solution is containerized, elastic and scalable;
- Service Orchestration or Choreography (ORCH) - evaluates if the solution provides service discovery and service composition of ML/DM services to build analytical workflows. Auto ML issues are included.

The seminal visual DM workflow tool is Waikato Environment for Knowledge Analysis (WEKA) Hall et al. (2009), a framework developed for data scientists, where a set of learning and data pre-processing algorithms can be adopted. The framework has an API for specific learning algorithms, such as deep learning ones, this way programming skills are required. WEKA turns feasible and popular the development of visual analytical workflows. A large number of DM algorithms, the WEKA knowledge flow component and the fact that WEKA is open source allowed its maintenance and longevity, since it is active with a large community for almost three decades.

The WEKA project entered the era of large scale analytical services, and for that, it was integrated with Spark Apache solution Zaharia et al. (2010). These improvements were detailed in Koliopoulos et al. (2015). Unfortunately, WEKA is not designed for Web and it cannot discover ML/DM service alternatives or enable service compositions using its API. Furthermore, it is not simple to interoperate its algorithms with other ML/DM tools. Finally, its is not containerized, thus no container orchestration is supported, and it is not a cloud solution with deploy and execution facilities, for instance.

In Breck (2008), it was presented a workflow system tool for ML and Natural Language Processing (NLP) processing, named Zymake. The main goal of it is to simplify the workflow execution system for users. It is a UNIX shell based solution, so scripts impose a programming complexity for ordinary users. Since it is not a distributed solution, Zymake is not resilient, not containerized, cannot run in a browser and it does not have user interface. The interoperability and orchestration features are not mentioned.

The Beygelzimer et al. (2013) work presented an analytic workflow orchestrator to handle ML tasks. The core idea is to create an analytic workflow set and in sequence uses a meta ML kernel to predict the best workflow to be deployed and executed. It demonstrated the usage of WEKA tool in a workflow, where a prediction task of a healthcare dataset was performed. The Web design is not present and there is no support for elastic cloud environments, including containers, container orchestrator and so forth. Timers and schedulers are also omitted on its workflows. There is no catalog of existing ML/DM tools to be used, so interoperability was not addressed. In summary, the automatic discover of ML tasks to compose a workflow is the core advantage of this work.

An important work to build Natural Language Processing (NLP) micro-services orchestration is Walkowiak (2017). A Language Processing Modeling Notation (LPMN) was introduced and it has three main stages on its orchestrator: the first one loads and transforms the data, the second partitions the data, and the third stage starts parallel NLP micro-services to handle the partitioned data, joining the results into a unique file. The LPMN was used to create several web applications ¹. The LPMN was designed for several programming languages and more than 500000 engine tasks were processed in half-year, at the paper publication date. The Walkowiak (2017) tool is not designed for Web, no visual workflows is supported, no containers and other elastic cloud features were detailed. The fault tolerance support was not mentioned. Interoperability among existing tools is also absent.

In 2018, Sean McClure wrote a magazine report with the title "GUI-fying the Machine Learning Workflow: Towards Rapid Discovery of Viable Pipelines" McClure (2018). He developed a Web rapid prototype solution for data scientists, where R and Python functions are wrapped and added to a pipeline using a simple Graphical User Interface (GUI). The solution is a Web based alternative where drag-and-drop operations are possible and it runs in the cloud using containers, thus fault tolerant. The Gui-fy solution supports some specific ML components for training, evaluation, pre-processing and visualization ML steps. Sometimes the data scientist is responsible for selecting and writing their own ML components. No automatic orchestration of services was done. There is no catalog of existing ML/DM tools to be used, so interoperability was not addressed. No unsupervised or semi-supervised service discovery and service composition of analytical components for building workflows were mentioned.

In Vahi et al. (2019), the authors presented a scientific workflow tool for Pegasus Work-

¹<http://ws.clarin-pl.eu>

flow Management System Deelman et al. (2015). The idea is to deploy, thus reproduce, scientific workflows in several cloud environments, so independent of OS and Computer architectures. The tool is containerized, supporting technologies such as Docker Merkel (2014), Singularity Kurtzer et al. (2017) and Shifter Jacobsen and Canon (2015). Unfortunately, all containers images are transferred via data network to the execution host and not build on host. The authors did not inform the existence of specific ML/DM components to be used on the workflows, simplifying the development. The user development tool is not designed for Web, thus it requires manual installations and updates, as well as it does not run in all devices, specially the mobile ones. Furthermore, no orchestration service was mentioned in such a work (discovery and composition services).

Recently, the platform Kubeflow was presented Bisong (2019). It is created to enhance and simplify the process of deploying ML workflows on Kubernetes Hightower et al. (2017). Thus, it is a containerized solution designed for cloud environments and with container orchestration service. It enables a visual Web interface for deployments of existing workflows. One of its main advantage is its interoperability, working with Chainer Tokui et al. (2015), Jupyter Kluyver et al. (2016), Katib Zhou et al. (2019), MXNet Chen et al. (2015), PyTorch Paszke et al. (2019), TensorRT Vanholder (2016) and TensorFlow Abadi et al. (2016). Scheduler and timer components are feasible in Kubeflow. Unfortunately, the pipelines are coded on these existing tools and only deployed on Kubeflow, so there is no way to develop the pipelines on it. This way, workflows with multiple pipelines coded using several ML tools are impossible. Another limitation is the absence of ML orchestration services from the existing ecosystem tools and this is because Kubeflow only deploys existing workflows as black boxes.

The TensorFlow Extended (TFX) Baylor et al. (2017) is a general-purpose machine learning platform implemented at Google. One of its main goals is to produce one machine learning platform for many learning tasks and for that it adopts the TensorFlow tool Abadi et al. (2016), which provides full flexibility for implementing any type of model architecture. TFX supports several continuation strategies that result from the interaction between data visitation and warm-starting options. Data visitation can be configured to be static or dynamic (over a rolling range of directories). Warm-starting initializes a subset of model parameters from a previous state. Model validation must be coupled with data validation in order to detect corrupted training data and thus prevent bad models from reaching production. On this direction, TFX provides implementations of these components that encode best practices observed in many production pipelines. Since TFX is a TensorFlow

extension, it adopts the same programming style, thus no visual pipeline is supported, including visual Web development support. Heterogeneity is not the core of TFX, so it does not have an interoperable REST API to invoke ML services and it does not handle different ML tools, like PyTorch or Scikit-learn, in a single workflow. No scheduler and observer services are performed for each pipe of a single pipeline, this way the data scientist cannot observe the state or schedule the pipe of complex workflows. Finally, there is no evidence of service orchestration on TFX.

Airflow Apache (2021) is a lightweight workflow manager, initially developed by AirBnB Guttentag (2019), and in sequence supported by Apache Incubator. It executes each workflow as a directed acyclic graph (DAG) of tasks. The tasks are usually atomic and are not supposed to share any resources with each other; therefore, they can be run independently. The DAG describes relationships between the tasks and defines their order of execution. The DAG objects are initiated from Python scripts placed in a designated folder. Airflow has a modular architecture and can distribute tasks to an arbitrary number of workers and across multiple servers while adhering to the task sequence and dependencies specified in the DAG. Unlike many of the more complicated platforms, Airflow imposes little overhead, is easy to install, and can be used to run task-based workflows in various environments ranging from stand-alone desktops and servers to Amazon or Google cloud platforms. Unfortunately, the data scientist cannot adopt multiple ML tools, like TensorFlow, Scikit-learn or PyTorch, to produce their pipelines in Python. They also cannot observe the pipelines steps executions remotely, transparently and using an interoperable strategy. There is no visual development of ML pipelines on the Web, so only Python code, including reusable analytical components, are feasible. It does not have a REST API for interoperable services and there is no evidence of service orchestration on it. On the other hand, it has schedulers and a vast amount of third-party integrator components.

The Amazon SageMaker Liberty et al. (2020) is another market-leader in terms of at scale ML platforms for pipeline deployment and development. It supports incremental training, resumable and elastic learning as well as automatic hyper-parameter optimization. SageMaker assumes distributed streaming data and a shared model state. The combination of a streaming model and shared state enables a convenient and simple abstraction for distributed learning algorithms. As a result, large scale learning algorithms can be integrated by implementing only three functions, namely, initialize, update, and finalize. This way, no visual ML components are possible to build Web pipelines. Furthermore, the update function is simultaneously executed on many machines in parallel, and must

be implemented such that the updates on partitions of the input stream can be merged together. This requirement is very complex for many data scientists, once parallelism is explicitly introduced and it is well known that concurrence, data partition and some other high performance programming skills are not very simple. The heterogeneity is performed in some level, since any existing ML algorithm in any tool can be executed, but an interoperable REST API for multi-language service calls is not addressed. The SageMaker is from Amazon, so it runs on AWS cloud environment and we do not have information about its cloud portability or even its container or container orchestrator solutions portability.

Mahout was first announced in 2008 and at that time it focused on MapReduce programming style. Recently, a new release Anil et al. (2020) offered a domain-specific language for declarative machine learning in cluster environments and its name is Samsara Schelter et al. (2016). Samsara allows its users to specify programs using a set of common matrix abstractions and linear algebraic operations, which at the same time integrate with existing dataflow operators. Mahout does not consider visual ML components. Low level services deployment are not addressed. It operates over clouds and containers, but without orchestration services. Multiple ML solutions are not feasible in a single workflow. Web designs, including a REST API, are not included. The Samsara programming skills are the main drawback of Mahout because the new matrix and linear algebra features integration with the old MapReduce features is not quite simple.

The scope of this work does not include auto ML, but it is important to highlight that we have found just one paper describing a library to write pipelines for such a purpose and its name is Katib Zhou et al. (2019). It can be deployed on cloud using Kubeflow Bisong (2019), thus it runs on containers and is designed for large scale. Basically, Katib implements the following pipes for its data scientists: "*Suggestion*, which provides the topology of the next candidate or the modification decisions of the previous architecture to *Model-Manager*, which constructs the model and sends it to *Trial*. Then the model is evaluated and the performance metrics are fed back to *Suggestion*, starting a new iteration" Zhou et al. (2019). A detailed study about the algorithms used to implement auto ML emerging technology is found in He et al. (2019).

The Learning Orchestra system differentiates from the literature in terms of providing a full stack of services to facilitate and streamline the data scientist activities or pipeline steps, varying from low level services, like cloud, container and container orchestration, to high level services, like ready to use model API services, tuning and training services and so forth. These low and high level services are implemented using three existing players

established worldwide: Scikit-learn, Spark-ML and TensorFlow. A single API enables an integration layer of high level services and several scripts enable the deploy and execution of low level services. Complex workflows can emerge from several pipelines using several tools and deployed on different cloud environments running lightweight containers.

Traditional pipelines, composed of tasks like data cleaning, data transformation, data enrichment, model training, hyperparameters tuning, model evaluating, and model's prediction, the last representing the production phase, can coexist with existing models trained and tuned previously by Google and Facebook, for instance, on Learning Orchestra. In the literature, this flexibility is quite rare. Very often, only low level or high level services are investigated, as we detailed in this chapter. Sometimes only the automation of services is the interest, like in Katib. None of the existing works enable different ML tools to be used during a single workflow construction using the same interoperable API. As we can see, the literature is different from the present works.

Two limitations of our presented system are: it is not designed to build neural networks from scratch and it does not support auto ML on its API. For the first limitation, the Learning Orchestra API has the *builder* service to encapsulate Spark-ML or TensorFlow codes that could be the construction of a deep neural network, but as we can see the API is not designed for that. In the next chapter, we detail how the Learning Orchestra system architecture is organized, how its services communicate and how its scripts are executed/configured.

Chapter 4

Development

In this chapter, it is detailed the Learning Orchestra architecture, this way how its components are organized and how they interact. Besides the architecture, several examples of how we can use the Learning Orchestra services are presented at the end of the chapter.

4.1 Architecture

The Learning Orchestra architecture is organized into layers, precisely into eight layers as Figure 4.1 illustrates.

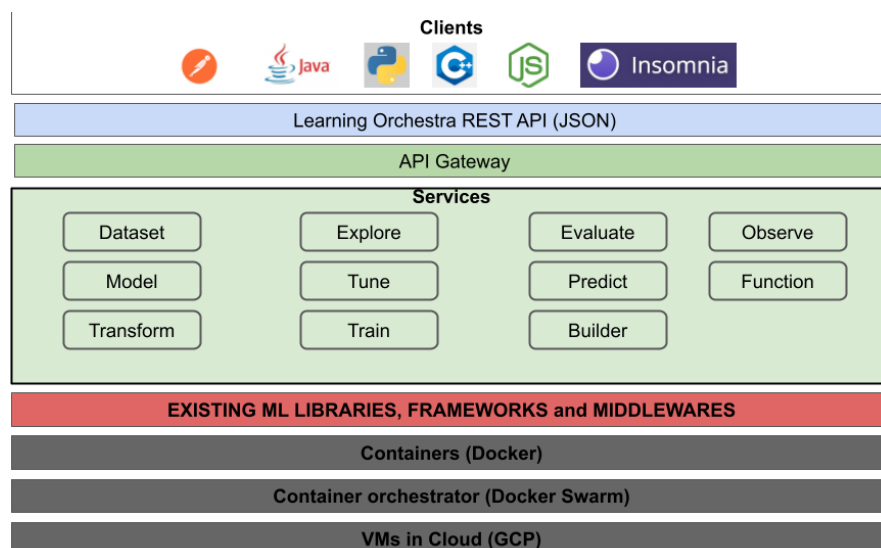


Figure 4.1: Learning Orchestra architecture

4.1.1 The Virtual Machine (VM) Layer

The VM represents the first layer, this way the lowest abstraction level of the Learning Orchestra system where the OS type, network configuration, memory capacity, number of cores on each virtual CPU or GPU and disk technology are applied. The Cloud environment turns the deploy and the maintainability of the systems a bit easier due to its elastic property. The Google Cloud Platform (GCP) and the AWS Platform are the two biggest players nowadays in Cloud Computing industry, so we decided to operate using GCP, but the Learning Orchestra is compatible with AWS infrastructure and any other public Cloud Computing platform.

The Learning Orchestra needs to be executed on Linux Debian and requires at least a cluster composed of VMs with 4GB of RAM and 64GB of disk space. It is important to stress that the ML model drives the deployment, so if the model is huge or if the input dataset is huge, the amount of disk space and RAM capacity, as well as the number of cores and VMs will increase. From the data scientist perspective, this layer is almost transparent, i.e., no detailed configuration is necessary. In summary, we just need to configure an elastic cluster environment, composed of VMs of any size, similar to Figure 4.2.

← VM instance details EDIT RESET

☐ Enable CPU overcommit ?

Labels

+ Add label

Creation time

Jul 21, 2020, 6:32:46 AM

Network interfaces ?

nic0: default default

+ Add item

Firewalls

- ☒ Allow HTTP traffic
- ☒ Allow HTTPS traffic

Network tags

http-server https-server

Deletion protection

☐ Enable deletion protection

When deletion protection is enabled, instance cannot be deleted. [Learn more](#)

Figure 4.2: GCP cluster configuration example

4.1.2 Container Orchestrator

The second layer is responsible for the container orchestration, where services like discovery of ML services, precisely the gateway routes, cluster deployment, on demand cluster resize, services migration from one container to other and issues related with network are covered. There are two leaders nowadays: Docker Swarm Naik (2016) and Kubernetes Hightower et al. (2017). We decided to use Docker Swarm to scale the Learning Orchestra services.

To simplify all containers management, we decided to use Portainer Portainer (2021). It enables to manage the global cluster state, including delete images and volumes, track the memory/CPU usage on each container, and observe the microservices and VMs of the cluster. It is deployed with Learning Orchestra, so transparent for the data scientist. It is available through the IP address and port 9000. All these software dependencies and Docker cluster tuning issues are configured via Docker Swarm configuration file, so Learning Orchestra has its own Docker Swarm configuration file.

4.1.3 Containers

Once we have deployed the container orchestrator, its necessary to deploy the containers. We have choose the Docker container technology Merkel (2014) due to its popularity and simplicity. Besides that, Docker Swarm and Docker containers are build from the same company, thus with high compatibility.

Figure 4.3 details the idea inside a container in the Learning Orchestra system. Basically, we have processing containers and storage containers. On each container we have an image previously mounted, so we have Spark Zaharia et al. (2010), Scikit-learn Pedregosa et al. (2011), TensorFlow Abadi et al. (2016) and MongoDB Chodorow (2013) images on two containers type that can be replicated over the deployed VMs and managed by the Docker Swarm, as explained previously. The system backend was developed using Python Van Rossum and Drake (2011), this way all images contain such a language interpreter.

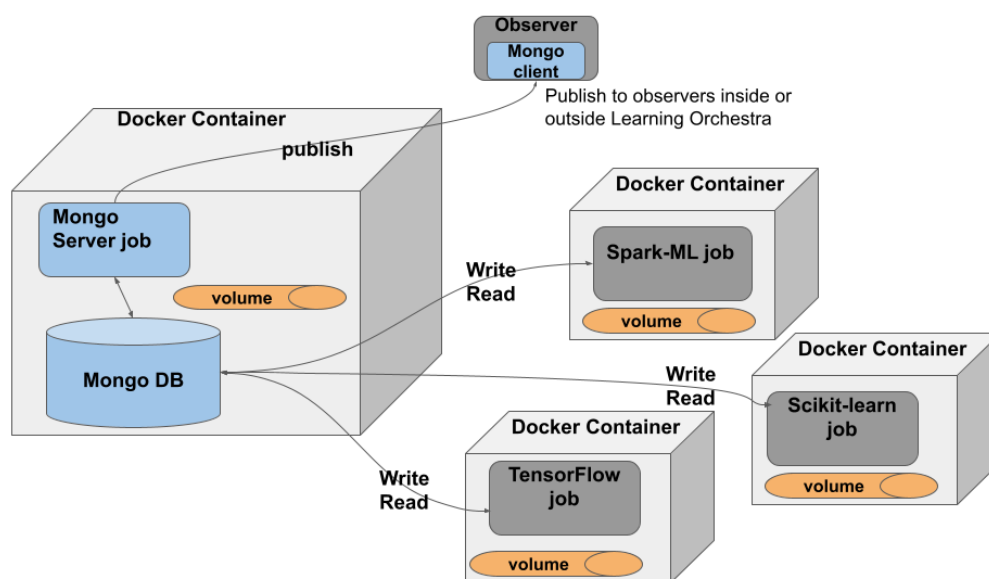


Figure 4.3: Learning Orchestra container types

The processing container has a Spark-ML, Scikit-learn or TensorFlow job doing any of their responsibilities. There are processing jobs running only Python code and it is because the data scientists need customized data transformations and dataset manipulations. Very often the processing jobs read the inputs from MongoDB and store the outputs back to the database or in a volume transparently managed by Docker Swarm, i.e., Docker Swarm manages the volume access on the network, so any job can access any volume transparently. This strategy is useful for two reasons: first, it maintains the intermediate information safe during a ML pipeline; second, it enables notifications via MongoDB.

The MongoDB storage system is responsible for publishing updates to components named Observers (explained later in this chapter). These updates occur on MongoDB and using a collection concept. A collection is a set of documents and each document is a set of entries. MongoDB is well designed for JSON format and it stores collections of documents. When a collection is updated, MongoDB notifies its clients, so we created a way for Learning Orchestra users observe collections changes pretty simple. Learning Orchestra supports storage fault tolerance and for that there is always two copies of MongoDB running, one for read/write operations and the other just for read operations. Synchronization aspects are done by a microservice, named referee, on a separate container, thus transparent for the data scientist.

From the data scientist perspective, the container configuration of the Learning Orchestra is performed as follows: First, it is necessary to have a Docker Swarm cluster,

configured according to any existing Web tutorial. On the manager machine of the cluster, you must install the Docker Compose configuration file. Next, you need to clone the source code from Learning Orchestra¹ and, as root user, you have to run the file *run.sh*, located on root directory. This file deploys the Learning Orchestra system on Docker Swarm.

To improve the performance of the microservices running on Spark cluster, it is necessary to scale the Spark worker and for that there are two ways: you can use the Docker Command Line Interface (CLI) or you can edit the configuration file. We illustrate the option using the configuration file. Before you run the *run.sh* file, you must set the number of replicas of the Spark worker on the *docker – compose.yml* file, located on root directory, precisely at line 155 of such a file. The default value are three replicas, but this number must be changed according to your ML model requirements and cluster resources. After this configuration setup, you just need to run the *run.sh* file.

4.1.4 Existing Libraries, Frameworks and Middlewares

This is not really a layer, but it is important to explain due to Learning Orchestra heterogeneity capacity. It represents the wrapped solutions, regardless it is a model configuration step or a dataset transform step in a pipeline. There are several examples of existing tools (libraries, frameworks and middlewares) to develop the pipes of a ML pipeline. On this work, we adopt the algorithms from Spark MLlib Meng et al. (2016), Scikit-learn Pedregosa et al. (2011) and TensorFlow Abadi et al. (2016). We understand that the catalog of possibilities for the data scientist is a fundamental requirement for the Learning Orchestra system.

4.1.5 Services

There are eleven services types and they are: Dataset, Model, Transform, Explore, Tune, Train, Evaluate, Predict, Builder, Observe and Function. Each service is composed at least of four microservices: one to insert or run the service, one to search for metadata or content about the service previously executed, one to delete the service from Learning Orchestra and one to update some information about the service.

The service types details:

¹<https://github.com/learningOrchestra/learningOrchestra>

- **Dataset:** Responsible to obtain a dataset. External datasets are stored on MongoDB or on volumes using an Uniform Resource Locator (URL). Dataset service enables the use of csv format datasets or generic format datasets.
- **Model:** Responsible for loading machine learning models from existing repositories. It is useful to be used to configure a TensorFlow or Scikit-learn object with a tuned and pre-trained models, like the pre-trained deep learning models provided by Google or Facebook, trained on huge instances, for example. On the other hand, it is also useful to load a customized/optimized neural network developed from scratch by a data scientist team.
- **Transform:** Responsible for a catalog of tasks, including embedding, normalization, text enrichment, bucketization, data projection and so forth. Learning Orchestra has its own implementations for some services and implement other transform services from TensorFlow and Scikit-learn.
- **Explore:** The data scientist must perform exploratory analysis to understand their data and see the results of their executed actions. So, Learning Orchestra supports data exploration using the catalog provided by TensorFlow and Scikit-learn tools, including histogram, clustering, t-SNE, PCA, and others. All outputs of this step are plottable.
- **Tune:** Performs the search for an optimal set of hyperparameters for a given model. It can be made through strategies like grid-search, random search, or Bayesian optimization.
- **Train:** Probably it is the most computational expensive service of an ML pipeline, because the models will be trained for best learn the subjacents patterns on data. A diversity of algorithms can be executed, like Support Vector Machine (SVM), Random Forest, Bayesian inference, K-Nearest Neighbors (KNN), Deep Neural Networks (DNN), and many others.
- **Evaluate:** After training a model, it is necessary to evaluate it's power to generalize to new unseen data. For that, the model needs to perform inferences or classification on a test dataset to obtain metrics that more accurately describe the capabilities of the model. Some common metrics are precision, recall, f1-score, accuracy, mean squared error (MSE), and cross-entropy. This service is useful to describe the generalization power and to detect the need for model calibrations.

- **Predict:** The model can run indefinitely. Sometimes feedbacks are mandatory to reinforce the train step, so the Evaluate services are called multiple times. This is the main reason for a production pipe and, consequently, a service of such a type.
- **Builder:** Responsible to execute Spark-ML entire pipelines in Python, offering an alternative way to use the Learning Orchestra system just as a deployment alternative and not an environment for building ML workflows composed of pipelines.
- **Observe:** Represents a catalog of collections of Learning Orchestra and a publish/-subscribe mechanism. Applications can subscribe to these collections to receive notifications via observers.
- **Function:** Responsible to wrap a Python function, representing a wildcard for the data scientist when there is no Learning Orchestra support for a specific ML service. It is different from Builder service, since it does not run the entire pipeline. Instead, it runs just a Python function of Scikit-learn or TensorFlow models on a cluster container. It is part of future plans the support of functions written in *R* language.

Figure 4.4 illustrates how complex the pipelines can become. They form a workflow of services. Note that, each pipeline has a logical clock, precisely the Dataset and Model service, thus it can schedule its execution for the future, enabling the data scientist to program pipelines at different time. The formed graph has dependencies among pipes, but it also has concurrency opportunities to run the pipes in parallel or distributed over multiple containers. The Learning Orchestra API is asynchronous, thus useful for concurrent development.

The observers enable notifications at any step of any pipeline, i.e., at any MongoDB collection update after some pipes runs. As we can see, many opportunities emerge from the ten service types. The Function service is not present, since it can be called before or after any step of the five pipelines illustrated in Figure 4.4. As we mentioned, it is a wildcard very useful when there is no Learning Orchestra support for a specific service.

4.1.6 API Gateway

This layer represents a way to simplify the URL syntax, avoiding to expose internal design or implementation issues. For instance, to load a dataset the data scientist must perform a POST API call like this - `'IPaddress/api/learningOrchestra/v1/dataset/csv/'`.

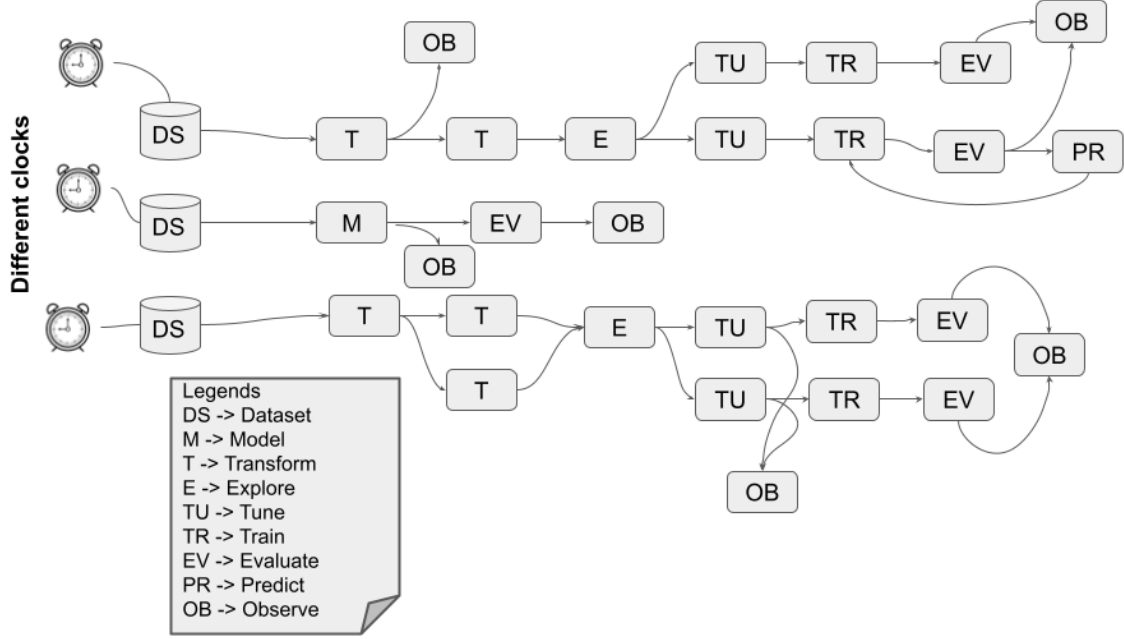


Figure 4.4: Learning Orchestra workflow composed of five pipelines

Internally, the Learning Orchestra system can call any microservice implementation, including third party alternatives. It is an elegant way to isolate user demands from technical issues, improving maintainability, extensibility and testability.

We have selected the KrakenD gateway KrakenD (2021) and it is configured on *run.sh* file of Learning Orchestra, so the data scientist does not need extra configuration demands.

4.1.7 Learning Orchestra REST API

There are eleven services types on Figure 4.1 and all of them are called following three simple rules. The rules for URL syntax are:

- *rule one*: Scikit-learn, TensorFlow or Spark-ML services have the tool name on the URL (Ex. `'IPaddress/api/learningOrchestra/v1/transform/scikitlearn/'`, `'IPaddress/api/learningOrchestra/v1/builder/sparkml/'` and `'IPaddress/api/learningOrchestra/v1/train/tensorflow/'`). Since these tools implement several alternatives for tune, train, predict, transform and other service types, the URL does not contain the service name on such API calls. The JSON message has all the tool ML method call particularities;

- *rule two*: The URL suffix can contain or not the service name when such a service is implemented internally, i.e., a service implemented by Learning Orchestra team. Ex. Dataset service and Transform/Explore services are also provided internally, thus they have the following URLs: `'IPaddress/api/learningOrchestra/v1/transform/projection/'`, `'IPaddress/api/learningOrchestra/v1/explore/histogram/'` and `'IPaddress/api/learningOrchestra/v1/dataset/csv/'`;
- *rule three*: The Function service is a Python alternative for the data scientist. Other programming languages will be supported by Learning Orchestra in future versions, so we decided to inform the language adopted on the URL (Ex. `'IPaddress/api/learningOrchestra/v1/function/python/'`).

The REST API calls are composed of JSON objects on their requests and responses. There are POST, GET, PUT, PATCH and DELETE HTTP requests. Each service type (Ex. train or tune) is composed of at least four microservices, as mentioned before, representing the Create, Retrieve, Update and Delete operations (CRUD) for each service type. The POST request is used to proceed an operation, i.e., a train, transform or explore pipe of a pipeline. All POST requests are asynchronous, this way the API caller receives an acknowledgment, indicating the URL to obtain the final result forward. Each service type has at least one GET HTTP request to search for an existing train, predict or any other pipeline step metadata information. Besides that, all service types have a DELETE and PUT/PATCH requests to enable all CRUD operations. Only the GET request is synchronous because the other requests may take long execution periods.

This work is not a tutorial about Learning Orchestra API microservices, thus we decided to write the API using a specification. We decided to adopt the Open API Initiative API (2021). The server used to code and host the Learning Orchestra system API is Swagger Hub Hub (2021). Observer REST CRUD services, as well as the other API service types illustrated in Figure 4.1 are detailed at - <https://app.swaggerhub.com/apis-docs/learningOrchestra/learningOrchestra/v1.0>.

There are two ways to wait for an API call: you can use the acknowledgment message with the URL to get the final result and proceed successive HTTP requests of GET type until a valid result is returned. There is a more elegant and efficient way to adopt an Observer, where the Learning Orchestra offers two Observer options: the Python Observer component to be imported on data scientist code and used accordingly to wait

for train, projection, tune or any other ML API call. There is also a REST API call using JSON objects to observe the conclusion of a ML job submitted previously - `IPaddress/api/learningOrchestra/v1/observe/`. The Observer is implemented using MongoDB client, where notifications of collections updates are performed without blocking the client with successive job results checks.

The Observe API service summary:

- method POST: `IPaddress/api/learningOrchestra/v1/observe/` - informs a pipe to be observed (tune, predict, train, evaluate or any other pipe of a pipeline);
- method GET: `IPaddress/api/learningOrchestra/v1/observe/` - retrieves all observed pipes metadata stored in Learning Orchestra;
- method GET: `IPaddress/api/learningOrchestra/v1/observe/{pipename}` - retrieves a specific observed pipe metadata stored in Learning Orchestra;
- method DELETE: `IPaddress/api/learningOrchestra/v1/observe/{pipename}` - deletes a specific observed pipe stored in Learning Orchestra.

The Dataset API service summary:

- method POST: `IPaddress/api/learningOrchestra/v1/dataset/csv` - downloads a csv format dataset using an URL;
- method GET: `IPaddress/api/learningOrchestra/v1/dataset/csv` - retrieves all csv datasets metadata stored in Learning Orchestra;
- method GET: `IPaddress/api/learningOrchestra/v1/dataset/csv/{datasetname}` - retrieves a specific csv dataset metadata stored in Learning Orchestra;
- method DELETE: `IPaddress/api/learningOrchestra/v1/dataset/csv/{datasetname}` - deletes a specific csv dataset stored in Learning Orchestra;
- method POST: `IPaddress/api/learningOrchestra/v1/dataset/generic` - downloads a generic dataset in any format from an url;
- method GET: `IPaddress/api/learningOrchestra/v1/dataset/generic` - retrieves all generic datasets metadata downloaded and stored in Learning Orchestra;

- method GET: `IPaddress/api/learningOrchestra/v1/dataset/generic/{datasetname}`
- retrieves a specific dataset metadata downloaded and stored in Learning Orchestra;
- method DELETE: `IPaddress/api/learningOrchestra/v1/dataset/generic/{datasetname}`
- deletes a specific dataset in Learning Orchestra;

The Transform API service summary:

- method POST: `IPaddress/api/learningOrchestra/v1/transform/projection` - creates a projected dataset from a previous inserted dataset;
- method GET: `IPaddress/api/learningOrchestra/v1/transform/projection` - retrieves all projections metadata performed previously;
- method GET: `IPaddress/api/learningOrchestra/v1/transform/projection/{datasetname}`
- retrieves a specific projection metadata performed previously;
- method DELETE: `IPaddress/api/learningOrchestra/v1/transform/projection/{datasetname}`
- deletes a specific projection created previously;
- method PATCH: `IPaddress/api/learningOrchestra/v1/transform/dataType` - updates the types of some dataset fields created previously;
- method POST: `IPaddress/api/learningOrchestra/v1/transform/scikitlearn` - creates a transformed binary object in a volume using a previous inserted dataset. For that, it invokes an existing Scikit-learn transform method. Any successive API call must use Scikit-learn, since the stored binary object format is only compatible with such a tool;
- method GET: `IPaddress/api/learningOrchestra/v1/transform/scikitlearn` - retrieves all transformations metadata performed previously using Scikit-learn;
- method GET: `IPaddress/api/learningOrchestra/v1/transform/scikitlearn/{datasetname}`
- retrieves a specific transformation metadata performed previously using Scikit-learn;
- method DELETE: `IPaddress/api/learningOrchestra/v1/transform/scikitlearn/{datasetname}`
- deletes a specific transformation created previously using Scikit-learn;

- method PATCH: `IPaddress/api/learningOrchestra/v1/transform/scikitlearn/{datasetname}`
- updates a specific transformation created previously using Scikit-learn. The transform operation is re-executed;
- method POST: `IPaddress/api/learningOrchestra/v1/transform/tensorflow` - creates a transformed binary object in a volume using a previous inserted dataset. For that, it invokes an existing TensorFlow transform method. Any successive API call must use TensorFlow, since the stored binary object format is only compatible with such a tool;
- method GET: `IPaddress/api/learningOrchestra/v1/transform/tensorflow` - retrieves all transformations metadata performed previously using TensorFlow;
- method GET: `IPaddress/api/learningOrchestra/v1/transform/tensorflow/{datasetname}`
- retrieves a specific transformation metadata performed previously using TensorFlow;
- method DELETE: `IPaddress/api/learningOrchestra/v1/transform/tensorflow/{datasetname}`
- deletes a specific transformation created previously using TensorFlow;
- method PATCH: `IPaddress/api/learningOrchestra/v1/transform/tensorflow/{datasetname}`
- updates a specific transformation created previously using TensorFlow. The transform operation is re-executed.

The Function API service summary:

- method POST: `IPaddress/api/learningOrchestra/v1/function/python` - executes a Python function, regardless it is used Scikit-learn or TensorFlow methods inside such a function. The output is a binary object stored in a volume. Any successive API call must use Scikit-learn or TensorFlow, since the stored binary object format is only compatible with such tools;
- method GET: `IPaddress/api/learningOrchestra/v1/function/python` - retrieves all Python functions metadata executed previously;
- method GET: `IPaddress/api/learningOrchestra/v1/function/python/{functionname}`
- retrieves a specific Python function metadata executed previously;
- method DELETE: `IPaddress/api/learningOrchestra/v1/function/python/{functionname}`
- deletes a specific function result created previously;

- method PATCH: `IPaddress/api/learningOrchestra/v1/function/python/{functionname}`
- updates a specific function executed previously. The function is re-executed.

The Explore API service summary:

- method POST: `IPaddress/api/learningOrchestra/v1/explore/tensorflow` - creates an explore image plot visualization in a volume using a previous object. For that, it invokes an existing TensorFlow explore method and converts the result in an image plot;
- method GET: `IPaddress/api/learningOrchestra/v1/explore/tensorflow` - retrieves a specific explore image plot performed previously using TensorFlow;
- method GET: `IPaddress/api/learningOrchestra/v1/explore/tensorflow/{objectname}/metadata`
- retrieves all explore metadata performed previously using TensorFlow;
- method GET: `IPaddress/api/learningOrchestra/v1/explore/tensorflow/{objectname}`
- retrieves a specific explore image plot performed previously using TensorFlow;
- method DELETE: `IPaddress/api/learningOrchestra/v1/explore/tensorflow/{objectname}`
- deletes a specific explore image plot created previously using TensorFlow;
- method PATCH: `IPaddress/api/learningOrchestra/v1/explore/tensorflow/{objectname}`
- updates a specific explore image plot created previously using TensorFlow. The explore operation is re-executed.
- method POST: `IPaddress/api/learningOrchestra/v1/explore/scikitlearn` - creates an explore image visualization in a volume using a previous object. For that, it invokes an existing Scikit-learn explore method and converts the result in an image plot.
- method GET: `IPaddress/api/learningOrchestra/v1/explore/scikitlearn` - retrieves all explore metadata performed previously using Scikit-learn;
- method GET: `IPaddress/api/learningOrchestra/v1/explore/scikitlearn/{objectname}/metadata`
- retrieves a specific explore metadata performed previously using Scikit-learn;
- method GET: `IPaddress/api/learningOrchestra/v1/explore/scikitlearn/{objectname}`
- retrieves a specific explore image plot performed previously using Scikit-learn;

- method DELETE: IPaddress/api/learningOrchestra/v1/explore/scikitlearn/{objectname}
- deletes a specific explore image plot created previously using Scikit-learn;
- method PATCH: IPaddress/api/learningOrchestra/v1/explore/scikitlearn/{objectname}
- updates a specific explore image plot created previously using Scikit-learn. The explore operation is re-executed;
- method POST: IPaddress/api/learningOrchestra/v1/explore/histogram - creates a histogram from a previous inserted dataset;
- method GET: IPaddress/api/learningOrchestra/v1/explore/histogram - retrieves all histograms metadata performed previously;
- method GET: IPaddress/api/learningOrchestra/v1/explore/histogram/{datasetname}
- retrieves a specific histogram metadata performed previously;
- method DELETE: IPaddress/api/learningOrchestra/v1/explore/histogram/{datasetname}
- deletes a specific histogram created previously;

The Model API service summary:

- method POST: IPaddress/api/learningOrchestra/v1/model/tensorflow - loads a ML model in a volume using the TensorFlow API. For that, it invokes an existing TensorFlow method. Any successive API call must use TensorFlow, since the stored binary object format is only compatible with such a tool;
- method GET: IPaddress/api/learningOrchestra/v1/model/tensorflow - retrieves all models metadata created previously using TensorFlow;
- method GET: IPaddress/api/learningOrchestra/v1/model/tensorflow/{modelname}
- retrieves a specific model metadata created previously using TensorFlow;
- method DELETE: IPaddress/api/learningOrchestra/v1/model/tensorflow/{modelname}
- deletes a specific model object created previously using TensorFlow;
- method PATCH: IPaddress/api/learningOrchestra/v1/model/tensorflow/{modelname}
- updates a specific model object created previously using TensorFlow. The model creation operation is re-executed.

- method POST: `IPaddress/api/learningOrchestra/v1/model/scikitlearn` - loads a ML model in a volume using the Scikit-learn API. For that, it invokes an existing Scikit-learn method. Any successive API call must use Scikit-learn, since the stored binary object format is only compatible with such a tool;
- method GET: `IPaddress/api/learningOrchestra/v1/model/scikitlearn` - retrieves all models metadata created previously using Scikit-learn;
- method GET: `IPaddress/api/learningOrchestra/v1/model/scikitlearn/{modelname}` - retrieves a specific model metadata created previously using Scikit-learn;
- method DELETE: `IPaddress/api/learningOrchestra/v1/model/scikitlearn/{modelname}` - deletes a specific model object created previously using Scikit-learn;
- method PATCH: `IPaddress/api/learningOrchestra/v1/model/scikitlearn/{modelname}` - updates a specific model object created previously using Scikit-learn. The model creation operation is re-executed.

The Tune API service summary:

- method POST: `IPaddress/api/learningOrchestra/v1/tune/tensorflow` - creates a tune binary object in a volume using a previous object. For that, it invokes an existing TensorFlow tune method. Any successive API call must use TensorFlow, since the stored binary object format is only compatible with such a tool;
- method GET: `IPaddress/api/learningOrchestra/v1/tune/tensorflow` - retrieves all tune metadata performed previously using TensorFlow;
- method GET: `IPaddress/api/learningOrchestra/v1/tune/tensorflow/{objectname}` - retrieves a specific tune metadata performed previously using TensorFlow;
- method DELETE: `IPaddress/api/learningOrchestra/v1/tune/tensorflow/{objectname}` - deletes a specific tune object created previously using TensorFlow;
- method PATCH: `IPaddress/api/learningOrchestra/v1/tune/tensorflow/{objectname}` - updates a specific tune object created previously using TensorFlow. The tune operation is re-executed.

- method POST: `IPaddress/api/learningOrchestra/v1/tune/scikitlearn` - creates a tune binary object in a volume using a previous object. For that, it invokes an existing Scikit-learn tune method. Any successive API call must use Scikit-learn, since the stored binary object format is only compatible with such a tool;
- method GET: `IPaddress/api/learningOrchestra/v1/tune/scikitlearn` - retrieves all tune metadata performed previously using Scikit-learn;
- method GET: `IPaddress/api/learningOrchestra/v1/tune/scikitlearn/{objectname}` - retrieves a specific tune metadata performed previously using Scikit-learn;
- method DELETE: `IPaddress/api/learningOrchestra/v1/tune/scikitlearn/{objectname}` - deletes a specific tune object created previously using Scikit-learn;
- method PATCH: `IPaddress/api/learningOrchestra/v1/tune/scikitlearn/{objectname}` - updates a specific tune object created previously using Scikit-learn. The tune operation is re-executed.

The Train API service summary:

- method POST: `IPaddress/api/learningOrchestra/v1/train/tensorflow` - creates a train binary object in a volume using a previous object. For that, it invokes an existing TensorFlow train method. Any successive API call must use TensorFlow, since the stored binary object format is only compatible with such a tool;
- method GET: `IPaddress/api/learningOrchestra/v1/train/tensorflow` - retrieves all train metadata performed previously using TensorFlow;
- method GET: `IPaddress/api/learningOrchestra/v1/train/tensorflow/{objectname}` - retrieves a specific train metadata performed previously using TensorFlow;
- method DELETE: `IPaddress/api/learningOrchestra/v1/train/tensorflow/{objectname}` - deletes a specific train object created previously using TensorFlow;
- method PATCH: `IPaddress/api/learningOrchestra/v1/train/tensorflow/{objectname}` - updates a specific train object created previously using TensorFlow. The train operation is re-executed.

- method POST: IPaddress/api/learningOrchestra/v1/train/scikitlearn - creates a train binary object in a volume using a previous object. For that, it invokes an existing Scikit-learn train method. Any successive API call must use Scikit-learn, since the stored binary object format is only compatible with such a tool;
- method GET: IPaddress/api/learningOrchestra/v1/train/scikitlearn - retrieves all train metadata performed previously using Scikit-learn;
- method GET: IPaddress/api/learningOrchestra/v1/train/scikitlearn/{objectname} - retrieves a specific train metadata performed previously using Scikit-learn;
- method DELETE: IPaddress/api/learningOrchestra/v1/train/scikitlearn/{objectname} - deletes a specific train object created previously using Scikit-learn;
- method PATCH: IPaddress/api/learningOrchestra/v1/train/scikitlearn/{objectname} - updates a specific train object created previously using Scikit-learn. The train operation is re-executed.

The Predict API service summary:

- method POST: IPaddress/api/learningOrchestra/v1/predict/tensorflow - creates a predict binary object in a volume using a previous object. For that, it invokes an existing TensorFlow predict method. Any successive API call must use TensorFlow, since the stored binary object format is only compatible with such a tool;
- method GET: IPaddress/api/learningOrchestra/v1/predict/tensorflow - retrieves all predict metadata performed previously using TensorFlow;
- method GET: IPaddress/api/learningOrchestra/v1/predict/tensorflow/{objectname} - retrieves a specific predict metadata performed previously using TensorFlow;
- method DELETE: IPaddress/api/learningOrchestra/v1/predict/tensorflow/{objectname} - deletes a specific predict object created previously using TensorFlow;
- method PATCH: IPaddress/api/learningOrchestra/v1/predict/tensorflow/{objectname} - updates a specific predict object created previously using TensorFlow. The predict operation is re-executed.

- method POST: `IPaddress/api/learningOrchestra/v1/predict/scikitlearn` - creates a predict binary object in a volume using a previous object. For that, it invokes an existing Scikit-learn predict method. Any successive API call must use Scikit-learn, since the stored binary object format is only compatible with such a tool;
- method GET: `IPaddress/api/learningOrchestra/v1/predict/scikitlearn` - retrieves all predict metadata performed previously using Scikit-learn;
- method GET: `IPaddress/api/learningOrchestra/v1/predict/scikitlearn/{objectname}` - retrieves a specific predict metadata performed previously using Scikit-learn;
- method DELETE: `IPaddress/api/learningOrchestra/v1/predict/scikitlearn/{objectname}` - deletes a specific predict object created previously using Scikit-learn;
- method PATCH: `IPaddress/api/learningOrchestra/v1/predict/scikitlearn/{objectname}` - updates a specific predict object created previously using Scikit-learn. The predict operation is re-executed.

The Evaluate API service summary:

- method POST: `IPaddress/api/learningOrchestra/v1/evaluate/tensorflow` - creates an evaluate binary object in a volume using a previous object. For that, it invokes an existing TensorFlow evaluate method. Any successive API call must use TensorFlow, since the stored binary object format is only compatible with such a tool;
- method GET: `IPaddress/api/learningOrchestra/v1/evaluate/tensorflow` - retrieves all evaluate metadata performed previously using TensorFlow;
- method GET: `IPaddress/api/learningOrchestra/v1/evaluate/tensorflow/{objectname}` - retrieves a specific evaluate metadata performed previously using TensorFlow;
- method DELETE: `IPaddress/api/learningOrchestra/v1/evaluate/tensorflow/{objectname}` - deletes a specific evaluate object created previously using TensorFlow;
- method PATCH: `IPaddress/api/learningOrchestra/v1/evaluate/tensorflow/{objectname}` - updates a specific evaluate object created previously using TensorFlow. The evaluate operation is re-executed.

- method POST: IPaddress/api/learningOrchestra/v1/evaluate/scikitlearn - creates an evaluate binary object in a volume using a previous object. For that, it invokes an existing Scikit-learn evaluate method. Any successive API call must use Scikit-learn, since the stored binary object format is only compatible with such a tool;
- method GET: IPaddress/api/learningOrchestra/v1/evaluate/scikitlearn - retrieves all evaluate metadata performed previously using Scikit-learn;
- method GET: IPaddress/api/learningOrchestra/v1/evaluate/scikitlearn/{objectname} - retrieves a specific evaluate metadata performed previously using Scikit-learn;
- method DELETE: IPaddress/api/learningOrchestra/v1/evaluate/scikitlearn/{objectname} - deletes a specific evaluate object created previously using Scikit-learn;
- method PATCH: IPaddress/api/learningOrchestra/v1/evaluate/scikitlearn/{objectname} - updates a specific evaluate object created previously using Scikit-learn. The evaluate operation is re-executed.

The Builder API service summary:

- method POST: IPaddress/api/learningOrchestra/v1/builder/sparkml - creates a builder pipeline binary object in a volume. For that, it invokes several existing Spark MLlib ML methods;
- method GET: IPaddress/api/learningOrchestra/v1/builder/sparkml - retrieves all builder metadata performed previously using Spark MLlib;
- method GET: IPaddress/api/learningOrchestra/v1/builder/sparkml/{objectname} - retrieves a specific builder metadata performed previously using Spark MLlib;
- method DELETE: IPaddress/api/learningOrchestra/v1/builder/sparkml/{objectname} - deletes a specific builder object created previously using Spark MLlib;
- method PATCH: IPaddress/api/learningOrchestra/v1/builder/sparkml/{objectname} - updates a specific builder object created previously using Spark MLlib. The builder pipeline operation is re-executed.

4.1.8 Learning Orchestra Clients

The last layer of Figure 4.1 is represented by several clients in different programming languages or tools, like Insomnia Insomnia (2021), to test REST APIs. This layer represents a way to simplify even more the Learning Orchestra API microservices presented on previous section. Each independent company or volunteer can design and implement its own clients with different microservices composition strategies.

We have developed one client in Python and it is used on the next section to develop the pipeline examples of this work. Such a Python client interfaces documentation and code are available at (repository - <https://github.com/learningOrchestra/pythonClient>) and (documentation - <https://learningorchestra.github.io/pythonClient/>). We omit its details in this section, since on Section 4.2 we have explained its utilization during the examples explanation.

4.2 Pipeline Examples

In this section, we present three examples to illustrate how to write pipelines for Learning Orchestra using a Python Client.

4.2.1 A Pipeline using the Builder service

The first example uses the Titanic dataset obtained from Kaggle repository ², which is a web platform containing thousands of datasets and several challenges of data science. The Titanic dataset is the content for one challenge ³, where the target is to predict which passenger survived in the Titanic disaster using the data of each passenger and the label survived or not. The dataset contains the fields:

- PassengerId - The id from each tuple, beginning in id 0 and finishing in last tuple count.
- Pclass - Ticket class, value 1 to first class, 2 to second class and 3 to third class.
- Name - The name of passenger, where each tuple has a unique name.
- Sex - The passenger sex, so the field has the "male" or "female" values.

²<https://www.kaggle.com/>

³<https://www.kaggle.com/c/titanic/overview>

- Age - Age in years of a passenger, and the training dataset has 89 unique age values, but the test dataset has 80 unique age values.
- SibSp - Number of siblings and spouses aboard the Titanic and there are 7 unique values.
- Parch - Number of parents and children aboard the Titanic. The training dataset has 7 unique values and the test dataset has 8 unique values.
- Ticket - Ticket number, where the training dataset has 681 unique values and the test dataset has 363 unique values.
- Fare - Passenger fare, where the training dataset has 248 unique values and the test dataset has 170 unique values.
- Cabin - Cabin number, where the training dataset has 148 unique values and the test dataset has 77 unique values.
- Embarked - Port of Embarkation, where C value represents Cherbourg, Q value represents Queenstown and S value represents Southampton

The Titanic was splitted out into two datasets, the training and the test ones, where the training has 891 tuples and the test has 418 tuples.

The Dataset, Builder, Projection and Data Type Learning Orchestra services are used to build the Titanic pipeline using the Python Client to simplify even more the REST API access. The cluster IP is also required to provide an access (see Lines 1 to 6).

```
1 from learning_orchestra_client.dataset.csv import DatasetCsv
2 from learning_orchestra_client.transform.projection import TransformProjection
3 from learning_orchestra_client.transform.data_type import TransformDataType
4 from learning_orchestra_client.builder import BuilderSparkML

6 CLUSTER_IP = "http://35.226.126.122"
```

After import all Python Client modules and define the cluster IP, we start to download the datasets. For that, the Client method call *insert_dataset_async* is performed twice,

one to download the train csv dataset and the other to download the test csv dataset. Most of API services of Learning Orchestra are asynchronous, this way a wait synchronization barrier is fundamental to guarantee a correct execution of any pipe step. Lines 19 and 20 illustrate the wait conditions for the datasets download step.

```

8 dataset_csv = DatasetCsv(CLUSTER_IP)
9
10 dataset_csv.insert_dataset_async(
11     url="https://filebin.net/r4b6z6sganz2opsh/train.csv?t=9d3lp7jm",
12     dataset_name="titanic_training",
13 )
14 dataset_csv.insert_dataset_async(
15     url="https://filebin.net/r0c41p538us5fcrz/test.csv?t=td68r02h",
16     dataset_name="titanic_testing"
17 )
18
19 dataset_csv.wait(dataset_name="titanic_training")
20 dataset_csv.wait(dataset_name="titanic_testing")
21
22 print(dataset_csv.search_dataset_content("titanic_training", limit=1,
23                                         pretty_response=True))
24 print(
25     dataset_csv.search_dataset_content("titanic_testing", limit=1,
26                                         pretty_response=True))

```

Next, we need to perform two projection activities, one to create a new dataset from the original train dataset using only the attributes *Passenger_Id*, *Pclass*, *Age*, *SibSp*, *Parch*, *Fare*, *Name*, *Sex*, *Embarked* and *Survived* (see Lines 24 to 40). The remaining test dataset attributes are removed when we call the Python Client method *remove_dataset_attributes_async* (see Lines 42 to 45). In sequence, we perform the second projection activity, where the attribute *Survived* is removed from test dataset. For that, same method call is performed, i.e., the same set of attributes defined on variable *required_columns* is used to create a new test dataset, excluding the attribute *Survived* (see Lines 49 to 52). The synchronization barriers are done using the wait method calls

(Lines 54 and 55) and we print the pipeline step results (Lines 57 to 63).

```
28 transform_projection = TransformProjection(CLUSTER_IP)
29 required_columns = [
30     "PassengerId",
31     "Pclass",
32     "Age",
33     "SibSp",
34     "Parch",
35     "Fare",
36     "Name",
37     "Sex",
38     "Embarked",
39     "Survived"
40 ]
41
42 transform_projection.remove_dataset_attributes_async(
43     dataset_name="titanic_training",
44     projection_name="titanic_training_projection3",
45     fields=required_columns)
46
47 required_columns.remove("Survived")
48
49 transform_projection.remove_dataset_attributes_async(
50     dataset_name="titanic_testing",
51     projection_name="titanic_testing_projection",
52     fields=required_columns)
53
54 transform_projection.wait(projection_name="titanic_training_projection")
55 transform_projection.wait(projection_name="titanic_testing_projection")
56
57 print(transform_projection.search_projection_content(
58     projection_name="titanic_training_projection", limit=1,
59     pretty_response=True))
60
```



```
61 print(transform_projection.search_projection_content(  
62     projection_name="titanic_testing_projection", limit=1,  
63     pretty_response=True))
```

Before the Builder service we need to change some attribute types, since some of them must be numbers. The Client method *update_dataset_type_async* is called twice, one for the test dataset (see Lines 75 to 77) and other for the train dataset (see Lines 81 to 83). The attributes *Age*, *Fare*, *Parch*, *PassengerId*, *Pclass* and *SibSp* are used for the test dataset and we include the *Survived* attribute for the train dataset. Wait conditions API calls are performed similar to other pipe steps (Lines 85 and 86).

```
65 transform_data_type = TransformDataType(CLUSTER_IP)  
66 type_fields = {  
67     "Age": "number",  
68     "Fare": "number",  
69     "Parch": "number",  
70     "PassengerId": "number",  
71     "Pclass": "number",  
72     "SibSp": "number"  
73 }  
74  
75 transform_data_type.update_dataset_type_async(  
76     dataset_name="titanic_testing_projection",  
77     types=type_fields)  
78  
79 type_fields.update({"Survived": "number"})  
80  
81 transform_data_type.update_dataset_type_async(  
82     dataset_name="titanic_training_projection",  
83     types=type_fields)  
84  
85 transform_data_type.wait(dataset_name="titanic_testing_projection")  
86 transform_data_type.wait(dataset_name="titanic_training_projection")
```

Finally, we call the Builder service to make predictions on test dataset and using the training dataset. Such a service requires a modeling code, which is a Pyspark code inserted in a JSON tag. The data scientist can opt to insert the code URL, this way the Learning Orchestra download and run it. The code represents a Spark entire pipeline ready to be deployed and executed.

The Builder service needs only the train dataset, the test dataset, the modeling code (the user defined pre-processing code to prepare the input data for fed the classifiers) and finally the classifiers to be used. In our example, five classifiers (*lr*, *dt*, *gb*, *rf*, and *nb*) run in parallel over all available cluster containers and transparently (line 193).

```

188 builder = BuilderSparkMl(CLUSTER_IP)
189 result = builder.run_spark_ml_async(
190     train_dataset_name="titanic_training_projection",
191     test_dataset_name="titanic_testing_projection",
192     modeling_code=modeling_code,
193     model_classifiers=["LR", "DT", "GB", "RF", "NB"])
194
195 for prediction in result["result"]:
196     builder.wait(dataset_name=prediction)
197     print(builder.search_builder_register_predictions(
198         builder_name=prediction, limit=1, pretty_response=True))

```

The Pyspark code was inspired from a web page ⁴, but to avoid code duplication, it was refactored. First, several libraries must be imported, precisely some SQL functions and ML features (Lines 88 to 97).

```

88 modeling_code = '''
89 from pyspark.ml import Pipeline
90 from pyspark.sql.functions import (
91     mean, col, split,
92     regexp_extract, when, lit)

```

⁴<https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bfcf/5722190290795989/3865595167034368/8175309257345795/latest.html>

```
93
94 from pyspark.ml.feature import (
95     VectorAssembler,
96     StringIndexer
97 )
```

Next, we need to rename the "Survived" column in both datasets for the default name "label", used by Pyspark for the predictions (Lines 102 and 103). A list to handle both datasets was created to facilitate the remaining operations in the code (Line 104).

```
99 TRAINING_DF_INDEX = 0
100 TESTING_DF_INDEX = 1
101
102 training_df = training_df.withColumnRenamed('Survived', 'label')
103 testing_df = testing_df.withColumn('label', lit(0))
104 datasets_list = [training_df, testing_df]
```

In Titanic datasets, there are misspelled initials in "Name" field, so we created a new field named "Initial" and extracted the initial names from the "Name" field using a regex (Lines 105 to 110). Next, we use two catalogs of initials, one miss spelled (Lines 112 to 117) and one correct (Lines 118 to 123). They are used to replace incorrect occurrences in both datasets (Lines 124 to 126).

```
106 for index, dataset in enumerate(datasets_list):
107     dataset = dataset.withColumn(
108         "Initial",
109         regexp_extract(col("Name"), "([A-Za-z]+)\.", 1))
110     datasets_list[index] = dataset
111
112 misspelled_initials = [
113     'Mlle', 'Mme', 'Ms', 'Dr',
114     'Major', 'Lady', 'Countess',
115     'Jonkheer', 'Col', 'Rev',
116     'Capt', 'Sir', 'Don'
```

```
117 ]
118 correct_initials = [
119     'Miss', 'Miss', 'Miss', 'Mr',
120     'Mr', 'Mrs', 'Mrs',
121     'Other', 'Other', 'Other',
122     'Mr', 'Mr', 'Mr'
123 ]
124 for index, dataset in enumerate(datasets_list):
125     dataset = dataset.replace(misspelled_initials, correct_initials)
126     datasets_list[index] = dataset
```

The "Age" field has several null values, so instead of using null values we decided to approximate the ages. If the "Initials" field contains *Miss*, the age will be 22, *Mr* will be replaced by 33 years and so forth (Lines 129 to 133). Only the null values are replaced (Lines 134 to 141).

```
129 initials_age = {"Miss": 22,
130                 "Other": 46,
131                 "Master": 5,
132                 "Mr": 33,
133                 "Mrs": 36}
134 for index, dataset in enumerate(datasets_list):
135     for initial, initial_age in initials_age.items():
136         dataset = dataset.withColumn(
137             "Age",
138             when((dataset["Initial"] == initial) &
139                 (dataset["Age"].isNull()), initial_age).otherwise(
140                     dataset["Age"]))
141     datasets_list[index] = dataset
```

The "Embarked" field also has null values, but as the histogram highlighted, the most frequent value is *S*, so we decided to replace the null values with *S* value.

```
144 for index, dataset in enumerate(datasets_list):
145     dataset = dataset.na.fill({"Embarked": 'S'})
146     datasets_list[index] = dataset
```

To analyze the survived rate and compare it with the family size of a passenger, we created the fields "Family_Size" and "Alone", where the "Family_Size" is the sum of "SibSp" and "Parch" fields, and the "Alone" has the value 1 when "Family_Size" has no member.

```
149 for index, dataset in enumerate(datasets_list):
150     dataset = dataset.withColumn("Family_Size", col('SibSp')+col('Parch'))
151     dataset = dataset.withColumn('Alone', lit(0))
152     dataset = dataset.withColumn(
153         "Alone",
154         when(dataset["Family_Size"] == 0, 1).otherwise(dataset["Alone"]))
155     datasets_list[index] = dataset
```

The string columns "Sex", "Embarked" and "Initial" must be converted to numbers and for that we use Pyspark StringIndexer. The Learning Orchestra Data Type service converts numbers typed as strings into numbers, so it is quite different from Pyspark StringIndexer.

```
158 text_fields = ["Sex", "Embarked", "Initial"]
159 for column in text_fields:
160     for index, dataset in enumerate(datasets_list):
161         dataset = StringIndexer(
162             inputCol=column, outputCol=column+"_index").\
163             fit(dataset).\
164             transform(dataset)
165         datasets_list[index] = dataset
```

To avoid fit a model with string fields, it will be removed the string columns used by StringIndexer in the previous step.

```
168 non_required_columns = ["Name", "Embarked", "Sex", "Initial"]
169 for index, dataset in enumerate(datasets_list):
170     dataset = dataset.drop(*non_required_columns)
171     datasets_list[index] = dataset
```

4.2.2 A Text Classification Pipeline Example

In this example, we adapt a sentiment analysis code available on Kaggle ⁵ to provide a text classification example using the IMDb dataset. We have adapted the Python code that uses the Scikit-learn library using the Learning Orchestra API to illustrate its integration with this ML solution.

The IMDb dataset has basically two important columns for this pipeline, one with the text containing the fan/critical reviews and the other with a label indicating positive or negative reviews. Figure 4.5 illustrates a sample of the dataset content. The label 1 indicates a positive post and the label 0 indicates a negative post.

	text	label
0	I grew up (b. 1965) watching and loving the Th...	0
1	When I put this movie in my DVD player, and sa...	0
2	Why do people who do not know what a particula...	0
3	Even though I have great interest in Biblical ...	0
4	Im a die hard Dads Army fan and nothing will e...	1

Figure 4.5: IMDb sample

The Dataset, Function, Model, Train e Prediction Learning Orchestra services was imported by the Python Client to execute the IMDb pipeline. Besides that, a cluster IP was defined (see Lines 1 to 7).

```

1 from learning_orchestra_client.dataset.csv import DatasetCsv
2 from learning_orchestra_client.function.python import FunctionPython
3 from learning_orchestra_client.model.scikitlearn import ModelScikitLearn
4 from learning_orchestra_client.train.scikitlearn import TrainScikitLearn
5 from learning_orchestra_client.predict.scikitlearn import PredictScikitLearn
6
7 CLUSTER_IP = "http://34.68.100.96"
```

⁵<https://www.kaggle.com/avnika22/imdb-perform-sentiment-analysis-with-scikit-learn>

The first activity is the dataset download and for that the Dataset service is used, precisely the Client *insert_dataset_sync* method (see Lines 10 to 16). The original dataset was reduced due to the cluster VMs size used on our experiments, so instead of 40k text reviews, we have used 1k text reviews. We have maintained the original percentage of positive and negative posts to guarantee a similar result with the literature using this dataset.

```
10 dataset_csv = DatasetCsv(CLUSTER_IP)
11
12 dataset_csv.insert_dataset_sync(
13     dataset_name="sentiment_analysis",
14     url="https://drive.google.com/u/0/uc?"
15         "id=1PSLWHbKR_cuKvGKeOSl913kCfs-DJE2n&export=download",
16 )
```

After download the dataset, we have analyzed the percentage of positive and negative reviews on the dataset. A Function service is called (see Lines 32 to 35) using the Client *run_function_sync* method. The Function Learning Orchestra service receives a Python code and executes it. An important observation is the utilization of the wildcard \$ on the value of the *parameters* function tag. It indicates that the Learning Orchestra kernel must access a collection named *sentiment_analysis* on MongoDB and execute the Python code using it. The result of the percentage of positive and negative reviews is printed on Lines 18 and 41.

```
18 function_python = FunctionPython(CLUSTER_IP)
19
20 explore_dataset = '''
21 pos=data[data["label"]=="1"]
22 neg=data[data["label"]=="0"]
23
24 total_rows = len(pos) + len(neg)
25
26 print("Positive = " + str(len(pos) / total_rows))
27 print("Negative = " + str(len(neg) / total_rows))
28
```



```
29 response = None
30 '''
31
32 function_python.run_function_sync(
33     name="sentiment_analysis_exploring",
34     parameters={"data": "$sentiment_analysis"},
35     code=explore_dataset)
36
37 print(function_python.search_execution_content(
38     name="sentiment_analysis_exploring",
39     limit=1,
40     skip=1,
41     pretty_response=True))
```

In sequence, another Python function is executed by the Function service using the *run_function_sync* Client (see Lines 96 to 102). As mentioned before, the Learning Orchestra API services are most asynchronous, but the Client offered synchronous and asynchronous method calls to facilitate the data scientist ML pipeline development. On this IMDb pipeline example we have choose the synchronous option to illustrate both options.

The pre-processing activity executed by the Function service is a Python code named *dataset_preprocessing* and it is responsible, in the beginning, for removing the HTML tags and the emoticons (Lines 47 to 56). Next, the *nltk.stem.porter* module is used, so all words are converted to their base meaning (Ex. running is converted into run). This is for simplifying the data and removing unnecessary complexities in the text data (Lines 58 to 65). The *TfidfVectorizer* is imported on Line 70 and it is responsible to convert the collection of raw texts into a matrix of TF-IDF features (Lines 70 to 79). The train and test subsets are created from the matrix data to prepare the input for the classifiers (Lines 81 to 93). Finally, the Learning Orchestra Function service is called by the Client using the *run_function_sync*, so the previous pre-processing steps are executed on a specific VM container.

```
43 dataset_preprocessing = '''
44 import re
45
46
47 def preprocessor(text):
48     global re
49     text = re.sub("<[^>]*>", "", text)
50     emojis = re.findall("(?:\:|;|=)(?:-)?(?:\:|\\(|\\D|P)", text)
51     text = re.sub("[\\W]+", " ", text.lower()) + \
52         " ".join(emojis).replace("-", "")
53     return text
54
55
56 data["text"] = data["text"].apply(preprocessor)
57
58 from nltk.stem.porter import PorterStemmer
59
60 porter = PorterStemmer()
61
62
63 def tokenizer_porter(text):
64     global porter
65     return [porter.stem(word) for word in text.split()]
66
67
68 from sklearn.feature_extraction.text import TfidfVectorizer
69
70 tfidf = TfidfVectorizer(strip_accents=None,
71                         lowercase=False,
72                         preprocessor=None,
73                         tokenizer=tokenizer_porter,
74                         use_idf=True,
75                         norm="l2",
76                         smooth_idf=True)
```

```
77
78 y = data.label.values
79 x = tfidf.fit_transform(data.text)
80
81 from sklearn.model_selection import train_test_split
82
83 X_train, X_test, y_train, y_test = train_test_split(x, y,
84                                                    random_state=1,
85                                                    test_size=0.5,
86                                                    shuffle=False)
87
88 response = {
89     "X_train": X_train,
90     "X_test": X_test,
91     "y_train": y_train,
92     "y_test": y_test
93 }
94 '''
95
96 function_python.run_function_sync(
97     name="sentiment_analysis_preprocessed",
98     parameters={
99         "data": "$sentiment_analysis"
100     },
101     code=dataset_preprocessing
102 )
```

After run the pre-processing steps, we need to build a model and for that we can use the Learning Orchestra Model service. The Client method *create_model_sync* is used for such a responsibility. In this pipeline, the Logistic Regression classifier is used to build the model, so all of its parameters must be set during the model construction.

```
104 model_scikitlearn = ModelScikitLearn(CLUSTER_IP)
105
```

```
106 model_scikitlearn.create_model_sync(  
107     name="sentiment_analysis_logistic_regression_cv",  
108     module_path="sklearn.linear_model",  
109     class_name="LogisticRegressionCV",  
110     class_parameters={  
111         "cv": 6,  
112         "scoring": "accuracy",  
113         "random_state": 0,  
114         "n_jobs": -1,  
115         "verbose": 3,  
116         "max_iter": 500  
117     }  
118 )  
119 )
```

Once the model has being created, the train step can be executed. The Learning Orchestra has integration with Scikit-learn and TensorFlow, so in this pipeline the Scikit-learn is used and for that the Client method *create_train_sync* is executed (Lines 122 to 130). As mentioned before, the wildcard \$ indicates that the parameters *X* and *Y* are downloaded from MongoDB using the respective values *sentiment_analysis_preprocessed.X.train* and *sentiment_analysis_preprocessed.Y.train*. These values are generated on previous steps of the pipeline and stored on MongoDB transparently.

```
121 train_scikitlearn = TrainScikitLearn(CLUSTER_IP)  
122 train_scikitlearn.create_training_sync(  
123     parent_name="sentiment_analysis_logistic_regression_cv",  
124     name="sentiment_analysis_logistic_regression_cv_trained",  
125     method_name="fit",  
126     parameters={  
127         "X": "$sentiment_analysis_preprocessed.X_train",  
128         "y": "$sentiment_analysis_preprocessed.y_train",  
129     }  
130 )
```

After a train step, we could call a predict step, so the Client method *create_predict_sync* is performed using the *X.test* as parameter (Lines 133 to 141).

```
132 predict_scikitlearn = PredictScikitLearn(CLUSTER_IP)
133 predict_scikitlearn.create_prediction_sync(
134     parent_name="sentiment_analysis_logistic_regression_cv_trained",
135     name="sentiment_analysis_logistic_regression_cv_predicted",
136     method_name="predict",
137     parameters={
138         "X": "$sentiment_analysis_preprocessed.X_test",
139     }
140
141 )
```

Finally, we evaluate the accuracy of the trained and tested model using the *X.test* and *Y.test* outputs. For that, a Function Learning Orchestra API call is done using the Client *run_function_sync* method (Lines 150 to 157). A print final step is done to see the accuracy results (Lines 159 and 161).

```
143 logistic_regression_cv_accuracy = '''
144 from sklearn import metrics
145
146 print("Accuracy: ",metrics.accuracy_score(y_test, y_pred))
147
148 response = None
149 '''
150 function_python.run_function_sync(
151     name="sentiment_analysis_logistic_regression_cv_accuracy",
152     parameters={
153         "y_test": "$sentiment_analysis_preprocessed.y_test",
154         "y_pred": "$sentiment_analysis_logistic_regression_cv_predicted"
155     },
156     code=logistic_regression_cv_accuracy
157 )
158
```

```
159 print(function_python.search_execution_content(  
160     name="sentiment_analysis_logistic_regression_cv_accuracy",  
161     pretty_response=True))
```

4.2.3 An Image Classification Pipeline Example

In this example, we adapted a code⁶ that uses the Keras framework to construct a multiclass image classification model. We used the Learning Orchestra API to illustrate an integration between our tool and the popular TensorFlow/Keras. Using the MNIST dataset, we trained and evaluated a simple deep neural network.

In this work, not all the pipeline steps are present in an explicit manner because its code length is extensive. To avoid showing all the code lines, we prefer to show only a few most significant lines. For more detailed information, the entire code can be found at Git repository⁷.

The obtained MNIST dataset is divided into four parts: *train_images*, *test_images*, *train_labels*, and *test_labels*. Each part is compressed in an IDX file format, that is is a format for vector and multidimensional matrices of numerical types.

In the first step of the pipeline, we download these four image dataset files (Lines 10 to 33). We need to extract these files' information and store them into *NumPy* objects to be processed by our Keras/TensorFlow model. For this task, we implemented the "*treat_dataset*" function, which receives the file objects of an image file and a label file and outputs a *NumPy* array for each of them (Lines 35 to 128).

A good practice to make a neural network's convergence rate faster is normalizing the input images' pixels values. We convert the integer values of the interval $[0, 255]$ of each pixel to a new float interval of $[0.0, 1.0]$ (Lines 130 to 155).

```
131 mnist_datasets_normalization = '''  
132 import numpy as np  
133 from tensorflow.keras.utils import to_categorical  
134  
135 test_images = test_images / 255.
```

⁶https://www.tensorflow.org/datasets/keras_example

⁷<https://github.com/learningOrchestra/pythonClient/blob/main/examples/mnist.py>

```
136 train_images = train_images / 255.
137
138
139 response = {
140     "test_images": test_images,
141     "test_labels": test_labels,
142     "train_images": train_images,
143     "train_labels": train_labels
144 }
145 '''
146
147 function_python.run_function_sync(
148     name="mnist_datasets_normalized",
149     parameters={
150         "train_images": "$mnist_datasets_treated.train_images",
151         "train_labels": "$mnist_datasets_treated.train_labels",
152         "test_images": "$mnist_datasets_treated.test_images",
153         "test_labels": "$mnist_datasets_treated.test_labels"
154     },
155     code=mnist_datasets_normalization)
```

After normalization, we defined simple neural network architecture with one hidden layer. As we receive an input of shape 28 x 28, we defined a Flatten layer as the first layer of our network to plain these values into 784 neurons. Next, a dense hidden layer with 128 neurons with ReLU activation function is set up. The hidden layer's output is fed into our output layer, a dense layer with ten neurons with softmax activation function, one neuron for each target class, that is, on neuron for each possible digit (Lines 157 to 170).

```
158 model_tensorflow = ModelTensorflow(CLUSTER_IP)
159 model_tensorflow.create_model_sync(
160     name="mnist_model",
161     module_path="tensorflow.keras.models",
162     class_name="Sequential",
163     class_parameters={
```

```
164         "layers":
165             [
166                 "#tensorflow.keras.layers.Flatten(input_shape=(28, 28))",
167                 "#tensorflow.keras.layers.Dense(128, activation='relu')",
168                 "#tensorflow.keras.layers.Dense(10, activation='softmax')",
169             ]}
170     )
```

In the next pipeline step, we set up the compilation. Our model weights converge guided by the Adam optimizer set up with an initial learning rate of 0.001. We compile our model defining the loss function as the *SparseCategoricalCrossentropy*, used in multi-class classification problems to compute the cross-entropy loss between the labels and predictions. We defined this same function as the metric to be calculated during the training phase. These functions are from the Keras framework.

```
174 model_compilation = '''
175 import tensorflow as tf
176 model.compile(
177     optimizer=tf.keras.optimizers.Adam(0.001),
178     loss=tf.keras.losses.SparseCategoricalCrossentropy(),
179     metrics=[tf.keras.metrics.SparseCategoricalAccuracy()],
180 )
181
182 response = model
183 '''
184
185 function_python.run_function_sync(
186     name="mnist_model_compiled",
187     parameters={
188         "model": "$mnist_model"
189     },
190     code=model_compilation)
```


Using the Keras models' fit function, we split the training dataset in 10% for the validation set and the remaining 90% for the training set and set up our model's training phase for six epochs long.

```
194 train_tensorflow = TrainTensorflow(CLUSTER_IP)
195 train_tensorflow.create_training_sync(
196     name="mnist_model_trained",
197     model_name="mnist_model",
198     parent_name="mnist_model_compiled",
199     method_name="fit",
200     parameters={
201         "x": "$mnist_datasets_normalized.train_images",
202         "y": "$mnist_datasets_normalized.train_labels",
203         "validation_split": 0.1,
204         "epochs": 6,
205     }
206 )
```

After training, we predicted the labels for the test images dataset, and for that, a *create_prediction* Client method call is performed (Lines 208 to 218). The prediction uses our trained model to infer each sample's probabilities to belong to each digit class using the Keras framework's predict function. So, the output will return an array of ten probabilities for each sample image. It is most likely an image belong to the class with the most higher inferred probability value.

```
209 predict_tensorflow = PredictTensorflow(CLUSTER_IP)
210 predict_tensorflow.create_prediction_sync(
211     name="mnist_model_predicted",
212     model_name="mnist_model",
213     parent_name="mnist_model_trained",
214     method_name="predict",
215     parameters={
216         "x": "$mnist_datasets_normalized.test_images"
217     }
218 )
```

The last step of our implemented pipeline is the evaluation of the model. So, we calculated the metrics of loss and cross-entropy accuracy using the Keras framework. To do that, we used our trained model, the test images dataset, and the correct test labels as input (Lines 220 to 231).

```
221 evaluate_tensorflow = EvaluateTensorflow(CLUSTER_IP)
222 evaluate_tensorflow.create_evaluate_sync(
223     name="mnist_model_evaluated",
224     model_name="mnist_model",
225     parent_name="mnist_model_predicted",
226     method_name="evaluate",
227     parameters={
228         "x": "$mnist_model_predicted",
229         "y": "$$mnist_datasets_normalized.test_labels"
230     }
231 )
```

Chapter 5

Experiments

This chapter presents the cluster setup where we executed the Learning Orchestra system and the performed experiments. All the pipelines described in Section 4.2 were evaluated using the same cluster setup and one pipeline at the time.

5.1 Setup

The Learning Orchestra system was configured in a small cluster deployed in the GCP. It is composed of three VMs running in the same network, and each VM has its own external Internet Protocol (IP). The cluster configuration is presented in Table 5.1.

Table 5.1: VMs configuration

VM	OS	CPU Model	CPU Cores	System RAM	System Storage
1	Debian 10	On Demand	2 vCPUs not shared	8GB	100GB
2	Debian 10	On Demand	2 vCPUs not shared	8GB	100GB
3	Debian 10	On Demand	2 vCPUs not shared	8GB	100GB

5.1.1 Container setup for Titanic pipeline

On such a cluster, we have deployed two different container configurations for different Titanic pipeline experiments. One configuration with three Spark workers (Spark W) - Figure 5.1 and a second configuration with just one Spark worker- Figure 5.2. We adopt one Docker Swarm master and two Worker nodes on both configurations, i.e., one Docker instance per VM. The other Spark containers named Spark C and Spark M are

mandatory, and they represent the Spark client and master, respectively. These container configurations are used to analyze the runtime when the number of containers increases. LR, DT, GB, NB and RF represent the five classifiers executed transparently over the containers and they are: Logistic Regression, Decision Tree, Gradient-Boosted Tree, Naive Bayes and Random Forest. On both configurations, we deployed the MongoDB container in the Docker Swarm master to avoid storage interference on the experiments.

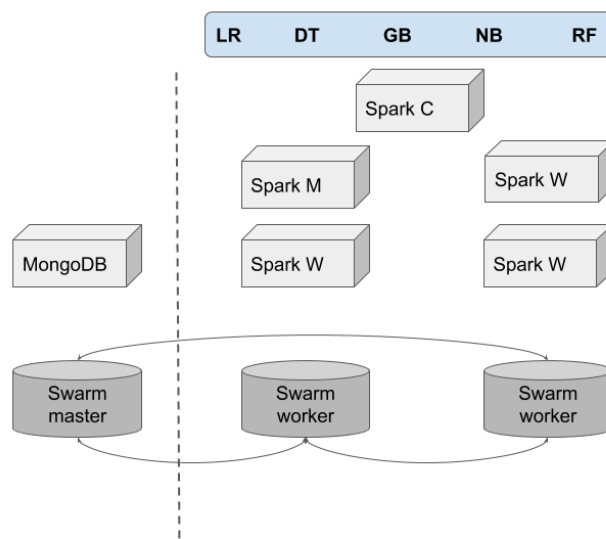


Figure 5.1: First container configuration

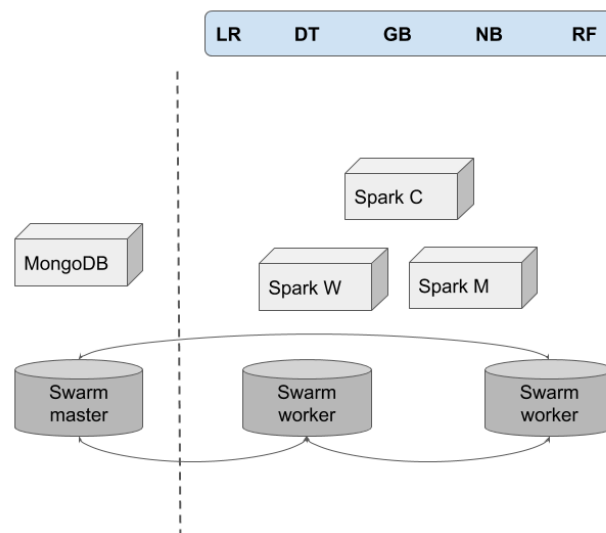


Figure 5.2: Second container configuration

5.1.2 Container setup for IMDb and MNIST pipelines

The remaining two pipelines described in Section 4.2 adopt the container configuration illustrated by Figure 5.3. Basically, Scikit-learn and TensorFlow images are deployed, and pipelines can use them accordingly. MongoDB is used to store metadata and datasets produced by several steps. One container is deployed per each pipe of the pipeline; this way, IMDb and MNIST have their own container setups using Model container or a Dataset container, for instance.

The applications can operate concurrently and over several containers and VMs, but this issue is the data scientist responsibility. We demonstrate some ways to achieve this container operation setup during IMDb and MNIST explanations. If no intelligent pipeline execution strategy is developed, the Learning Orchestra just distribute the containers over the VMs, but the pipes run sequentially.

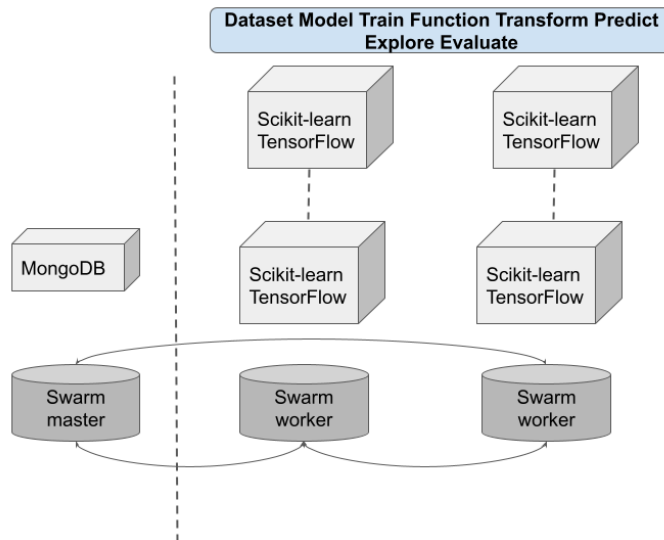


Figure 5.3: Third container configuration

5.2 Metrics

We calculate some metrics in our experiments: runtime, memory consumption, accuracy, loss, and F1 score. Each experiment used some of these metrics but not all of them.

- Runtime: represents the period during which a computer program is executing. In our experiments, the runtimes are obtained from a single run in a single container

running a single classifier or from the entire cluster, i.e., from all containers and all classifiers concurrent runs.

- **Memory Consumption:** also called memory footprint, represents the main memory that a program uses or references while running. In our experiments, the memory consumption is obtained when the container is idle and running any program. The OS memory usage and the Docker and any other infrastructure tool were not considered in the results.
- **Cross-entropy (Loss):** commonly used in deep neural networks. It is the difference between the predicted value and the true value. Can be defined as $-\sum_{i=1}^n \sum_{j=1}^m y_{i,j} \log(p_{i,j})$, where $y_{i,j}$ equals 1 if sample i belongs to class j , otherwise, it is equals 0. $p_{i,j}$ is model's predicted probability of sample i belongs to class j (Janocha and Czarnecki, 2017).
- **Accuracy:** Accuracy is also used as a statistical measure of how well a classification test correctly identifies or excludes a condition. That is, the accuracy is the proportion of correct predictions (both true positives and true negatives) among the total number of cases examined (Metz, 1978). The formula for quantifying accuracy is: $(TP + TN)/(TP + TN + FP + FN)$, where TP is True positive; FP is False positive; TN is True negative; FN is False negative. We want to predict if a passenger will survive or not, thus a binary classification in the Titanic dataset.
- **F1 score:** The F1 score is the harmonic mean of precision and recall, is defined as $F1 = (2 \times (P \times R))/(P + R)$. P is the fraction of all positives predictions, which are true positives, defined as $P = TP/(TP + FP)$. The recall R is the fraction of all actual positives that are predicted positive, and it is defined as $R = TP/(TP + FN)$, (Lipton et al., 2014).

5.3 Classifiers

It was used the following classifiers, but not on all pipelines: Logistic Regression, Decision Tree, Random Forest, Gradient Boosted Tree and Naive Bayes.

- **Logistic Regression (LR):** The logistic regression as a general statistical model was originally developed and popularized primarily by Joseph Berkson Cramer (2002). It is used to model the probability of a certain class or event occur, such as pass/fail,

win/lose, alive/dead or healthy/sick. The method can be extended to determine whether an image contains a cat, dog, lion and so forth. Each object being detected in the image would be assigned a probability between 0 and 1, with a sum of one.

- **Decision Tree (DT):** It uses a decision tree (as a predictive model) to go from observations about an item (represented in the branches) to conclusions about the item's target value (represented in the leaves). One common algorithm was presented in 1986 by Quinlan Quinlan (1986). It is a top-down induction of decision trees (TDIDT), considered by far the most common strategy for learning decision trees from data.
- **Random Forest (RF):** Random decision forests are an ensemble learning method for classification, regression and other tasks that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees Ho (1998). The first algorithm for random decision forests was created by Tin Kam Ho Ho (1995).
- **Gradient Boosted Tree (GB):** It is a machine learning technique for regression and classification problems, which produces a prediction model in the form of an ensemble of weak prediction models, typically decision trees. The idea of gradient boosting originated by Leo Breiman Breiman (1997).
- **Naive Bayes (NB):** In statistics, Naive Bayes classifiers are a family of simple "probabilistic classifiers" based on applying Bayes' theorem with strong (naïve) independence assumptions between the features, i.e., the models that assign class labels to problem instances, represented as vectors of feature values, assume that the value of a particular feature is independent of the value of any other feature, given the class variable. Unfortunately, in 2006 a study showed that Bayes classification is outperformed by other approaches, such as boosted trees or random forests Caruana and Niculescu-Mizil (2006).

5.4 Titanic Experiment

5.4.1 Dataset

Kaggle offers the Titanic data as two datasets, the training, and the test ones, where the training has 891 tuples, and the test has 418 tuples. We split the training dataset

at a ratio of 90% for training, and 10% of it was used to validate the training step. The test dataset does not have labels, and we used it to submit the predictions to the Kaggle platform.

We do not use any k-folds cross-validation in our evaluation steps to explain a model's generalization power. Still, it is part of future work to develop such methodology in the Learning Orchestra's evaluation step. We evaluate models using a single test dataset to calculate the experiment metrics.

To report the memory consumption and runtime results, each run was repeated five times and an average was calculated.

5.4.2 Runtime Results

The service Builder, precisely the POST microservice, runs five classifiers (LR, GB, DT, NB and RF) in two VMs with four vCPUs in total, since the master Docker Swarm node is responsible for MongoDB container, as mentioned before. The pipeline is compounded by training, including the validation step and the test steps, and is performed for each classifier sequentially. Each pipeline is performed concurrently with the others in a single container and over the three container instances previously explained. In summary, an easy alternative to perform concurrent pipeline executions is when you use different classifiers to build different models.

The five runs for the training step, not including its validation, are illustrated in Table 5.2. In general, when we deploy three Spark W containers - Figure 5.1 - the runtimes are quite similar. The last run took less time than the others, since the GCP maintains some cache and other optimizations, even when we stop and suspend the containers after each run. The Naive Bayes classifier proves to be a bit faster than the others and the literature reinforced this behavior - "Maximum-likelihood training can be done by evaluating a closed-form expression, which takes linear time, rather than by expensive iterative approximation as used for many other types of classifiers." Russell and Norvig (2002).

The runtimes of the training-validation step are illustrated in Table 5.3. Since only 10% of the dataset was used during the validation, the runtimes are very small, but quite similar as observed previously during the training step. The container configuration was the same, i.e., three Spark W instances.

Finally, in Table 5.4 there are the runtimes of the test step, where the predictions were done. This step is extremely fast since the models are already trained.

Table 5.2: Classifiers training runtime

Execution	LR	GB	DT	NB	RF
1	69.96 s	69.95 s	59.3 s	56.6 s	59.3 s
2	57.06 s	56.33 s	50.5 s	48.47 s	51.85 s
3	58.59 s	58.48 s	49.53 s	45.55	43.46 s
4	50.75 s	50.95 s	41.08 s	36.38 s	44.71 s
5	46.21 s	46.79 s	39.96 s	34.35 s	43.12 s
Average	56.51 s	56.5 s	48.07 s	44.27 s	48.48 s

Table 5.3: Classifiers train-validation runtime

Execution	LR	GB	DT	NB	RF
1	11.81 s	10.51 s	13.18 s	13.71 s	11.51 s
2	11.26 s	7.73 s	12.25 s	12.94 s	11.95 s
3	9.66 s	11.29 s	12.14 s	12.64 s	14.11 s
4	9.32 s	10.51 s	11.14 s	12.59 s	9.31 s
5	8.12 s	8.81 s	10.57 s	8.67 s	10.82 s
Average	10.03 s	9.77 s	11.85 s	12.11 s	11.54 s

Table 5.4: Classifiers test runtime

Execution	LR	GB	DT	NB	RF
1	0.47 s	0.34 s	0.12 s	1.53 s	0.17 s
2	0.09 s	0.55 s	0.06 s	0.08 s	0.08 s
3	0.09 s	0.15 s	0.14 s	0.44 s	0.79 s
4	0.08 s	0.04 s	0.10 s	0.14 s	0.10 s
5	0.15 s	0.13 s	0.19 s	0.13 s	0.30 s
Average	0.17 s	0.24 s	0.12 s	0.46 s	0.28 s

The previous runtimes were collected per process, i.e., per classifier in the container it executed. It is interesting to verify the total runtime, i.e., from the client perspective and including all network overheads, as well as the containers, VMs and other infrastructure overheads. Table 5.5 illustrates these runtimes. In general, training, validation and test steps took near 1.13 minute, but the total time took near 1.5 minute, so the overhead is not so drastic, but this result is not conclusive. Larger models will better demonstrate if this overhead continues to be small. The total time is collected in Spark C container, where the Titanic application starts its execution.

Table 5.5: Titanic runtime

Execution	Spark Application
1	2 m
2	1.5 m
3	1.5 m
4	1.3 m
5	1.3 m
Average	1.52 m

Table 5.6 illustrates the training runtimes using the container configuration with just one Spark W instance. As we can observe, the cluster running one processing container can have similar runtimes or even be faster when it is compared with a cluster with three times more processing containers - Table 5.2. This is because the number of tuples in Titanic dataset is small and the concurrence of five classifiers in a single container over a VM with 2 vCPUs did not degrade the runtimes of them. To reinforce this assumption, we created a hypothetical Titanic dataset with ten times more tuples to train. In this new scenario, the container configuration with three Spark W instances was near 30% faster. In summary, not always the decentralized container configuration is the best choice for running your ML model.

Table 5.6: Classifiers training runtime with one Spark W instance

Execution	LR	GB	DT	NB	RF
1	79.29 s	79.76 s	57.29 s	50.67 s	59.64 s
2	49.59 s	56.94 s	39.81 s	28.39 s	42.25 s
3	49.24 s	53.34 s	36.14 s	27.38 s	38.65 s
4	47.93 s	53.25 s	34.54 s	27.60 s	36.52 s
5	40.55 s	50.28 s	33.73 s	30.37 s	35.97 s
Average	53.18 s	58.71 s	40.3 s	32.88 s	42.6 s

5.4.3 Memory Consumption Results

During the experiments with three Spark W containers we decided to collect the memory consumption of them before the classifiers run and during an execution. The results are presented in Table 5.7. RAM idle indicates when the cluster is not running and RUN running when the classifiers are running. On average, the Spark W instances increased three times their memory consumption, proving that the Titanic model is very small. The

Spark C container increased ten times its memory consumption, since it is responsible for the JSON responses of all classifiers. The Spark master (Spark M) container maintained its memory stable, as expected.

Table 5.7: Memory consumption

Consumption	Spark C	Spark M	Spark W 1	Spark W 2	Spark W 3
RAM idle	68.86 MB	204.3 MB	189.4 MB	184.1 MB	166 MB
RAM running	767.8 MB	234.8 MB	768 MB	700 MB	720 MB

5.4.4 ML Metrics Results

Table 5.8 illustrates the validation result of the entire dataset for the five classifiers in terms of accuracy and F1 score. The goal was to predict if a passenger will survive or not.

We used 10% of the original training dataset as a validation dataset and the remaining samples for training. The implemented solution to this problem achieved good results in the validation tests. The Gradient Boosting (GB) model obtained the best metrics scores, with 82.5% for the accuracy and 82.4% for the F1 score. The worse results are obtained by the Naive Bayes (NB) model, with 63.3% for the accuracy and 62.5% for the F1 score.

Table 5.8: Prediction metrics comparison

Classifier	accuracy rate	F1 score
LR	0.808	0.807
DT	0.796	0.795
RF	0.819	0.816
GB	0.825	0.824
NB	0.633	0.625

5.4.5 Samples Prediction Results

Table 5.9 illustrates the probability of prediction on the first 10 tuples from the Titanic test dataset. In this example, we collected the probabilities arrays from RF classifier results. The "probability 0" column has the predicted probability of sample belongs to class 0 (Not Survived), and the "probability 1" column has the predicted probability of sample belongs to class 1 (Survived).

Table 5.9: Tuples Prediction Results

PassengerId	probability 0	probability 1	prediction
892	0.98808	0.01191	0.0
893	0.88695	0.11304	0.0
894	0.99681	0.00318	0.0
895	0.98299	0.01700	0.0
896	0.88695	0.11304	0.0
897	0.98299	0.01700	0.0
898	0.89960	0.10039	0.0
899	1.0	0.0	0.0
900	0.83837	0.16162	0.0
901	0.98968	0.01031	0.0

5.4.6 Ranking survived Titanic results

The Titanic results was submitted to a worldwide competition organized by Kaggle. This competition group several predictions submissions on Kaggle platform, making a ranking based on accuracy results. When this text was edited, the competition had 34193 teams, and the Learning Orchestra titanic prediction submission was on 10652 position, on Table 5.10.

Table 5.10: Classifiers score on Kaggle Challenge

Classifier	Score
LR	0.77751
DT	0.76555
RF	0.77511
GB	0.75358
NB	0.65550

5.5 IMDb Experiment

IMDb (an acronym for Internet Movie Database) is an online database of information related to films, television programs, home videos, video games, and streaming content online – including cast, production crew and personal biographies, plot summaries, trivia, ratings, and fan and critical reviews. The original dataset has 40000 review samples. As described in Chapter 4, we have used a Logistic Regression Classifier model to evaluate the

IMDb and calculate only the accuracy metric, achieving a score of 0.8904. It is a reasonable result to a simple benchmark model. The Scikit-learn was used as the ML tool.

To evaluate the Learning Orchestra system in terms of runtime and memory consumption, one simple alternative for the data scientist is evaluate the same ML pipeline in terms of classifiers, but adopting different hyper-parameters on each classifier. This strategy can evaluate if different Logistic Regression configurations can achieve similar ML results. These IMDb pipelines are executed concurrently and distributed over the cluster of containers illustrated by Figure 5.3. We have evaluated IMDb pipeline with different values for the solver hyper-parameter, precisely using $\text{solver} = \{\text{newton-cg}, \text{lbfgs}, \text{liblinear}, \text{sag} \text{ and } \text{saga}\}$. More details about the solver and other hyper-parameters of Logistic Regression using Scikit-learn ¹.

Table 5.11: IMDb solvers runtime and memory consumption

Solver	Runtime	Memory
newton-cg	68.12s	1.791 GB
lbfgs	68.76s	1.641 GB
liblinear	70.7s	2.175 GB
sag	68.54s	2.340 GB
saga	69s	2.405 GB
all	5min 29s	2.833 GB

The results presented on Table 5.11 illustrated all IMDb pipeline runs. First, we performed five pipeline runs using each solver of the Logistic Regression classifier and the results demonstrated a similar runtime (approximately 70 seconds). In terms of memory consumption, each pipeline consumed around 2GB of RAM. In the second experimental round, we executed all previous five pipelines concurrently and the result was, as expected, better than successive sequential runs. In summary, all five pipelines took around six minutes sequentially and five minutes and half concurrently over only two VMs with two cores each. The memory consumption benefit was considerable, being the concurrent version only 30% worse, but it runs five pipelines and not only one.

¹https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

5.6 MNIST Experiment

The MNIST database (Modified National Institute of Standards and Technology database) Deng (2012) is an image database of handwritten digits. This database is widely used as a benchmark dataset for testing ML models for image processing systems in the literature. It is composed of 60,000 training images and 10,000 testing images. Each sample is a 28×28 pixels image of grayscale levels, so each pixel is represented by an integer value between 0 and 255. There are a total of ten classes, one for each possible digit from 0 to 9.

We constructed a simple deep neural network as showed in 4.2.3. The training step was performed for six epochs long using the training dataset split into 10% for validation and the remaining samples as a training dataset. To evaluate the trained example model, we used the testing dataset. We calculated the cross-entropy loss and the accuracy and obtained the results, presenting them on Table 5.12. The obtained results are good for a simple example of a deep neural network model, showing that only with one hidden layer and without a single convolutional layer to extract features, the model achieved a reasonable generalization power.

Table 5.12: Metrics Scores for MNIST experiment

Metric	Score
Cross-entropy loss	0.069
Accuracy	0.979

Chapter 6

Conclusion

In this work, we present the Learning Orchestra system for ML development and deployment. The alternative solution incentives the heterogeneity, since it has a REST API where only JSON objects are exchanged. Furthermore, it has transparent support for Scikit-Learn, TensorFlow and Spark MLlib, which are three standards of ML products worldwide. A set of eleven useful services enable the construction of complex ML flows composed of several pipelines. A unique script to deploy the Learning Orchestra over a set of VMs simplifies the data scientist low level activity.

To demonstrate the system services expressibility, we developed three pipelines: Titanic, IMDb and MNIST. The Learning Orchestra API is asynchronous, so we decided to implement a Python client with extra facilities, including synchronous and asynchronous interoperable ML services. Many other clients using many other programming languages are feasible with a REST API.

We have evaluated three scalable and simple techniques during the pipeline experiments. The first one test multiple classifiers to produce a more accurate ML model. The second technique tests the hyper-parameter values to identify the best solver in terms of accuracy. Finally, we demonstrate how simple is to download multiple datasets concurrently and over many containers, thus over several VMs. Multiple files are very normal when you want to build a deep learning model. These three scalable techniques are very useful for data scientists, regardless their interests.

The experiments conducted to evaluate the Learning Orchestra system considered ML metrics, as well as runtime and memory consumption metrics. In general, the system scales well and the techniques mentioned before really accelerates the pipelines. The memory consumption of concurrent solutions very often consume more memory than sequential

solutions, but they enable multiple pipeline runs faster. In terms of ML, the results presented in this work are also competitive. For instance, the Titanic results achieved the position 10652 in a competition with 34193 teams.

The rest of this chapter discusses the good, the bad and the ugly aspects of Learning Orchestra and the last part of Conclusion details the future directions.

6.1 Discussion

In this section, the Learning Orchestra system is discussed in terms of design, implementation, deploy and tests. The good, bad and ugly aspects of the system are highlighted.

6.1.1 The good

The presented open-source solution works with different ML marked-leaders alternatives, enabling the development of different pipelines using these alternatives into a single workflow. The data scientists can have different programming skills, varying from a shallow Python understanding where a loaded model of a previously trained dataset can be used to work with a test dataset, from an advanced Python skill, where the data scientist can use the Learning Orchestra API to tune, transform, explore, observe and model complex deep learning flows.

The container architectural design enables fault tolerance, workload among VMs, and the concurrent job runs. This last feature opens interesting opportunities to accelerate the system. JSON technology has its good and bad aspects. The good is its programming language interoperability.

Achieving good results with ML models on Titanic, IMDb, and MNIST datasets were not the goals of this work, as cited in 1 Chapter, but they achieved reasonable results. Thus they reinforced that the Learning Orchestra system produces correct ML results. Besides, develops and deploy ML workflows over multiple container executions.

6.1.2 The bad

The bad issues related to Learning Orchestra are: JSON is verbose, thus predict, tune, transform, explore, model, dataset, and all eleven services require a huge amount of code to be configured, precisely the calls for the methods of the Scikit-learn and the TensorFlow libraries.

As many layers of abstraction a system has, as slower the system becomes. Gateways, containers, VMs, JSON, HTTP protocol and so forth make the Learning Orchestra a bit slower, but the world understand that the benefits of hardware in terms of price and quality compensate.

The deployment of Learning Orchestra works only with Docker technology, so we do not used Kubernetes deployment tool, like Kubeflow does.

The algorithms of the Scikit-learn and the TensorFlow libraries run in a single thread and are not distributed. There are GPU extensions for TensorFlow, precisely for NVidia boards, but it still runs in a single VM. TensorFlow integrates with Spark, but for distributed storage purposes. Its extension TFX Baylor et al. (2017) and the Horovod library Sergeev and Del Balso (2018) alternative are good examples for both distributed train step and large-scale train-predict-evaluate production pipelines for ML.

The Experiments Evaluations considered a very small cluster, thus very small datasets. Bigger VMs and larger clusters to run, for instance, the 40k images of MNIST are fundamental.

6.1.3 The ugly

In Learning Orchestra, sometimes the VM volume is used to store binary files containing Scikit-learn or TensorFlow information. There are two ways to map a volume for the eleven ML service types: (i) the manager Docker Swarm container becomes the unique valid volume. Thus all jobs use it to store their files. (ii) Each container has its own local volume. The second alternative has a severe limitation. The Docker Swarm capabilities to rearrange the containers, depending on the workload of each VM, and the fault-tolerance recovery aspect, where containers return to the cluster on a different VM, cannot be used because the ML jobs waste their volume references. A Distributed File System (DFS) layer will fix alternative (ii) limitations.

MongoDB deployment nowadays is fault-tolerant in Learning Orchestra, but the current system version does not include a MongoDB deployment using sharding improvements, thus with no efficient distributed collection storage.

6.2 Future Directions

The Learning Orchestra system can be improved on different directions:

- Besides Scikit-learn, Spark MLlib and TensorFlow core, the Learning Orchestra kernel should support TFX and Horovod benefits;
- MongoDB deployment must include sharding option;
- A DFS layer using several containers must represent a virtual distributed volume. The Hadoop HDFS Borthakur et al. (2008), for instance, can be used;
- Kubeflow Bisong (2019) deployment support should be included;
- New clients for different programming languages, like Java and JavaScript, are very useful;
- The Function API service can include *R* support and not only Python;
- A visual Web interface to build and deploy ML workflows are mandatory;
- The support for pipe, pipeline and workflow collaboration via shared-resources among multiple data scientists is a useful issue;
- Auto-ML flows should be supported by Learning Orchestra, thus services, similar to Katib Zhou et al. (2019), are interesting alternatives;
- Larger experiments and different pipelines will discover new system strengths and weakness.

Bibliography

- Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283.
- Anil, R., Capan, G., Drost-Fromm, I., Dunning, T., Friedman, E., Grant, T., Quinn, S., Ranjan, P., Schelter, S., and Yilmazel, O. (2020). Apache mahout: Machine learning on distributed dataflow systems. *Journal of Machine Learning Research*, 21(127):1–6.
- Apache (2021). Apache air flow. Available at: <https://airflow.apache.org/>.
- API, O. (2021). Open api specification. Available at: <https://www.openapis.org/>.
- Baylor, D., Breck, E., Cheng, H.-T., Fiedel, N., Foo, C. Y., Haque, Z., Haykal, S., Ispir, M., Jain, V., Koc, L., et al. (2017). Tfx: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1387–1395.
- Beygelzimer, A., Riabov, A., Sow, D., Turaga, D. S., and Udrea, O. (2013). Big data exploration via automated orchestration of analytic workflows. In *Proceedings of the 10th International Conference on Autonomic Computing ({ICAC} 13)*, pages 153–158.
- Bisong, E. (2019). Kubeflow and kubeflow pipelines. In *Building Machine Learning and Deep Learning Models on Google Cloud Platform*, pages 671–685. Springer.
- Borthakur, D. et al. (2008). Hdfs architecture guide. *Hadoop Apache Project*, 53(1-13):2.
- Breck, E. (2008). zymake: A computational workflow system for machine learning and natural language processing. In *Software Engineering, Testing, and Quality Assurance for Natural Language Processing*, pages 5–13. Association for Computational Linguistics.

- Breiman, L. (1997). Arcing the edge. Technical report, Technical Report 486, Statistics Department, University of California at
- Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., and Zavattaro, G. (2005). Choreography and orchestration: A synergic approach for system design. In *International Conference on Service-Oriented Computing*, pages 228–240. Springer.
- Caruana, R. and Niculescu-Mizil, A. (2006). An empirical comparison of supervised learning algorithms. In *Proceedings of the 23rd international conference on Machine learning*, pages 161–168.
- Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. (2015). Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*.
- Chodorow, K. (2013). *MongoDB: the definitive guide: powerful and scalable data storage*. " O'Reilly Media, Inc."
- Cramer, J. S. (2002). The origins of logistic regression.
- Deelman, E., Vahi, K., Juve, G., Rynge, M., Callaghan, S., Maechling, P. J., Mayani, R., Chen, W., Da Silva, R. F., Livny, M., et al. (2015). Pegasus, a workflow management system for science automation. *Future Generation Computer Systems*, 46:17–35.
- Deng, L. (2012). The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, 29(6):141–142.
- Goldberg, R. P. (1974). Survey of virtual machine research. *Computer*, 7(6):34–45.
- Guttentag, D. (2019). Progress on airbnb: a literature review. *Journal of Hospitality and Tourism Technology*.
- Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18.
- Han, J., Pei, J., and Kamber, M. (2011). *Data mining: concepts and techniques*. Elsevier.
- He, X., Zhao, K., and Chu, X. (2019). Automl: A survey of the state-of-the-art. *arXiv preprint arXiv:1908.00709*.

- Hightower, K., Burns, B., and Beda, J. (2017). *Kubernetes: up and running: dive into the future of infrastructure*. " O'Reilly Media, Inc."
- Ho, T. K. (1995). Random decision forests. In *Proceedings of 3rd international conference on document analysis and recognition*, volume 1, pages 278–282. IEEE.
- Ho, T. K. (1998). The random subspace method for constructing decision forests. *IEEE transactions on pattern analysis and machine intelligence*, 20(8):832–844.
- Hub, S. (2021). Swagger hub. Available at: <https://swagger.io/tools/swaggerhub/>.
- Insomnia (2021). Insomnia collaborative api design editor. Available at: <https://insomnia.rest/>.
- Jacobsen, D. M. and Canon, R. S. (2015). Contain this, unleashing docker for hpc. *Proceedings of the Cray User Group*, pages 33–49.
- Janocha, K. and Czarnecki, W. M. (2017). On loss functions for deep neural networks in classification. *arXiv preprint arXiv:1702.05659*.
- Keras (2021). Keras. Available at: <https://en.wikipedia.org/wiki/Keras>.
- Kluyver, T., Ragan-Kelley, B., Pérez, F., Granger, B. E., Bussonnier, M., Frederic, J., Kelley, K., Hamrick, J. B., Grout, J., Corlay, S., et al. (2016). Jupyter notebooks-a publishing format for reproducible computational workflows. In *ELPUB*, pages 87–90.
- Koliopoulos, A.-K., Yiapanis, P., Tekiner, F., Nenadic, G., and Keane, J. (2015). A parallel distributed weka framework for big data mining using spark. In *2015 IEEE international congress on big data*, pages 9–16. IEEE.
- KrakenD (2021). Krakend. Available at: <https://www.krakend.io/>.
- Kreps, J., Narkhede, N., Rao, J., et al. (2011). Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7.
- Kurtzer, G. M., Sochat, V., and Bauer, M. W. (2017). Singularity: Scientific containers for mobility of compute. *PloS one*, 12(5).
- Liberty, E., Karnin, Z., Xiang, B., Rouesnel, L., Coskun, B., Nallapati, R., Delgado, J., Sadoughi, A., Astashonok, Y., Das, P., et al. (2020). Elastic machine learning algorithms in amazon sagemaker. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 731–737.

- Lipton, Z. C., Elkan, C., and Narayanaswamy, B. (2014). Thresholding classifiers to maximize f1 score.
- McClure, S. (2018). Gui-fying the machine learning workflow: Towards rapid discovery of viable pipelines.
- Mell, P., Grance, T., et al. (2011). The nist definition of cloud computing.
- Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D., Amde, M., Owen, S., et al. (2016). Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241.
- Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2.
- Metz, C. E. (1978). Basic principles of roc analysis. In *Seminars in nuclear medicine*, volume 8, pages 283–298. WB Saunders.
- Nadareishvili, I., Mitra, R., McLarty, M., and Amundsen, M. (2016). *Microservice architecture: aligning principles, practices, and culture*. " O'Reilly Media, Inc."
- Naik, N. (2016). Building a virtual system of systems using docker swarm in multiple clouds. In *2016 IEEE International Symposium on Systems Engineering (ISSE)*, pages 1–3. IEEE.
- Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019). Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037.
- Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. (2011). Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830.
- Peltz, C. (2003). Web services orchestration and choreography. *Computer*, 36(10):46–52.
- Portainer (2021). Portainer. Available at: <https://www.portainer.io/>.
- Quinlan, J. R. (1986). Induction of decision trees. *Machine learning*, 1(1):81–106.
- Russell, S. and Norvig, P. (2002). Artificial intelligence: a modern approach.

- Schelter, S., Palumbo, A., Quinn, S., Marthi, S., and Musselman, A. (2016). Samsara: Declarative machine learning on distributed dataflow systems. In *NIPS Workshop ML-Systems*.
- Sergeev, A. and Del Balso, M. (2018). Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*.
- Tokui, S., Oono, K., Hido, S., and Clayton, J. (2015). Chainer: a next-generation open source framework for deep learning. In *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, volume 5, pages 1–6.
- Vahi, K., Rynge, M., Papadimitriou, G., Brown, D. A., Mayani, R., da Silva, R. F., Deelman, E., Mandal, A., Lyons, E., and Zink, M. (2019). Custom execution environments with containers in pegasus-enabled scientific workflows. *arXiv preprint arXiv:1905.08204*.
- Van Rossum, G. and Drake, F. L. (2011). *The python language reference manual*. Network Theory Ltd.
- Vanhoder, H. (2016). Efficient inference with tensorrt.
- Walkowiak, T. (2017). Language processing modelling notation–orchestration of nlp microservices. In *Advances in Dependability Engineering of Complex Systems*, pages 464–473. Springer.
- Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., Stoica, I., et al. (2010). Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95.
- Zhou, J., Velichkevich, A., Prosvirov, K., Garg, A., Oshima, Y., and Dutta, D. (2019). Katib: A distributed general automl platform on kubernetes. In *2019 {USENIX} Conference on Operational Machine Learning (OpML 19)*, pages 55–57.
- Zoph, B. and Le, Q. V. (2016). Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*.