FEDERAL UNIVERSITY OF OURO PRETO
INSTITUTE OF EXACT AND BIOLOGICAL SCIENCES
COMPUTER DEPARTMENT

JONATAS MIGUEL AVELINO FERREIRA
Supervisor: Joubert de Castro Lima
Co-supervisor: Gabriel de Oliveira Ribeiro

# LEARNING ORCHESTRA AUTOML: PYCARET AND AUTOKERAS OPERATED USING A RESTFUL API AND MODELED AS ATOMIC PIPELINE STEPS

Ouro Preto, MG
2022

FEDERAL UNIVERSITY OF OURO PRETO
INSTITUTE OF EXACT AND BIOLOGICAL SCIENCES
COMPUTER DEPARTMENT

JONATAS MIGUEL AVELINO FERREIRA

# LEARNING ORCHESTRA AUTOML: PYCARET AND AUTOKERAS OPERATED USING A RESTFUL API AND MODELED AS ATOMIC PIPELINE STEPS

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 of Computer Science.

**Supervisor:** Joubert de Castro Lima
**Co-supervisor:** Gabriel de Oliveira Ribeiro

Ouro Preto, MG
2022

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIENCIAS EXATAS E BIOLOGICAS
DEPARTAMENTO DE COMPUTACAO

**FOLHA DE APROVAÇÃO**

**Jonatas Miguel Avelino Ferreira**

**Learning Orchestra AutoML: Pycaret and Autokeras operated using a RESTful API and modeled as atomic pipeline steps**

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 27 de Outubro de 2022.

Membros da banca

Joubert de Castro Lima (Orientador) - Doutor - Universidade Federal de Ouro Preto
Gabriel de Oliveira Ribeiro (Coorientador) - Bacharel - CI&T
Pedro Henrique Lopes Silva (Examinador) - Doutor - Universidade Federal de Ouro Preto
Rodrigo César Pedrosa Silva (Examinador) - Doutor - Universidade Federal de Ouro Preto

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 27/10/2022.

Documento assinado eletronicamente por **Joubert de Castro Lima**, **PROFESSOR 3 GRAU**, em 28/10/2022, às 08:19, 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 **0416026** e o código CRC **BE4F47BF**.



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

# Acknowledgements

First, I dedicate this work to my parents Andrea Maria and Giovanne Vicente for their love, encouragement and unconditional support, to my girlfriend Thaís Dias Mendonça, who never refused me love, support and encouragement, to my teacher and advisor Joubert De Castro Lima for her Guidance, teachings, support and trust, finally, I would like to thank Gabriel de Oliveira Ribeiro for his support in creating this work during the semester.

# Resumo

A atividade de um cientista de dados normalmente funciona em um processo iterativo composto pelas seguintes atividades: coleta de dados disponíveis, realização de análise exploratória de dados, limpeza/enriquecimento desses dados, construção de modelos, validação de suas previsões e implantação de resultados. As ferramentas de Automated Machine Learning (AutoML), como Katib, Pycaret, AutoKeras e ML-JAR, são utilizadas hoje em dia para simplificar ou mesmo eliminar algumas das atividades citadas anteriormente. Consequentemente, o cientista de dados pode se concentrar em validações e atividades de ajuste de parâmetros, por exemplo. Infelizmente, hoje em dia não existe uma API com suporte para múltiplas ferramentas AutoML. As mais modernas ferramentas de pipeline Machine Learning (ML), como TFX, Spark, Kubeflow ou Ray, são compatíveis com muitos ML-Toolkits, como TensorFlow, Scikit-learn, Pytorch, MXNet e outros, mas nenhum deles oferece suporte a diferentes ferramentas AutoML. Essa lacuna pode aumentar o custo de desenvolvimento e implantação de pipelines ML. Outro aspecto importante nas ferramentas de pipeline ML existentes, é sua incapacidade de executar etapas de pipeline separadas (transformação de dados, exploração, treinamento, ajuste ou qualquer outra etapa). Eles executam todo o pipeline e, muitas vezes, utilizando somente a memória RAM. Esse comportamento limita tanto a utilização da parada antecipada (Early Stopping) de etapas do pipeline quanto a reexecução de etapas do pipeline armazenadas em volumes de um ambiente de nuvem, o que significa que tais etapas não foram armazenados apenas na RAM. Este trabalho apresenta a primeira API para operar as ferramentas Pycaret e Autokeras AutoML. O cientista de dados agora pode desenvolver seus pipelines de dados usando notebooks e várias chamadas de API RESTful interoperáveis para serviços AutoML. Este trabalho também resolve todo o problema de execução do pipeline, que também é executado utilizando apenas a memória RAM por Pycaret e AutoKeras. Agora, etapas únicas do pipeline AutoML podem ser executadas separadamente em diferentes contêineres e salvas em um volume específico do cluster. Para implementar tais inovações e executar as etapas restantes do pipeline ML, como coletar dados, explorar, limpar e enriquecer dados, bem como o salvamento automático das saídas das etapas mencionadas anteriormente, estendemos o núcleo da ferramenta Learning Orchestra services, permitindo a execução de serviços AutoML. Learning Orchestra é uma ferramenta de pipeline ML com suporte para as ferramentas Spark MLlib, Scikit-learn e TensorFlow ML. Avaliamos o impacto dos serviços de API apresentados em relação à execução das ferramentas AutoML diretamente de notebooks, como Zeppelin e Jupyter. Os resultados demonstraram quase nenhum impacto em termos de tempo de execução do serviço de nossa API e um aumento médio de 250MB na utilização de RAM quando utilizado nossa solução distribuída.

**Palavras Chave**: Aprendizado de Máquina. Mineração de dados. Sistema distribuído. Arquitetura Orientada a Serviços. Computação em Nuvem. Pipeline. API. Orquestração de Contêineres. Aprendizado de máquina automatizado.

# Abstract

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 Automated Machine Learning (AutoML) tools, like Katib, Pycaret, AutoKeras and ML-JAR, are used nowadays to simplify or even eliminate some of the activities cited before. Consequently, the data scientist can concentrate in validations and parameter tuning activities, for instance. Unfortunately, nowadays there is no API with a support for multiple AutoML tools. The state-of-the-art Machine Learning (ML) pipeline tools, like TFX, Spark, Kubeflow or Ray, support many ML-Toolkits, like TensorFlow, Scikit-learn, Pytorch, MXNet and others, but none of them support distinct AutoML tools. Such a lacuna can increase the cost of developing and deploying ML pipelines. Another important aspect in the existing ML pipeline tools is their incapacity to run separated pipeline steps (data transformation, exploration, training, tuning or any other step). They run the entire pipeline and very often in RAM-only mode. This behavior limits both early pipeline stops and the re-execution of steps of the pipeline stored in volumes of a cloud environment, which means stored not only in RAM. This work presents the first RESTful API to operate Pycaret and Autokeras AutoML tools. The data scientist can now develop their data pipelines using notebooks and several interoperable RESTful API calls for AutoML services. This work also solves the entire pipeline run problem, which was performed in RAM-only mode. Now, single AutoML pipeline steps can run separately in different containers and saved in a specific volume of the cluster. To implement both improvements, we extend the Learning Orchestra tool core services, enabling it to perform also AutoML services. The Learning Orchestra is a ML pipeline tool with support for Spark MLlib, Scikit-learn and TensorFlow ML Toolkits. We have evaluated the impact of the presented API services against the execution of the AutoML tools directly from notebooks, like Zeppelin and Jupyter. The results demonstrated almost no impact in terms of service runtime of our API and an average increase of 250MB in RAM utilization for MongoDB support when our solution is selected.

**Keywords**: Machine Learning. Data Mining. Distributed System. Service-oriented Architecture. Cloud Computing. Pipeline. Container. Container Orchestration. Automated Machine Learning.

# List of Figures

Figure 3.1 – Learning Orchestra AutoML architecture. Adapted from (RIBEIRO, 2021) . 14
Figure 4.1 – Pycaret and Autokeras container deployment options in Learning Orchestra AutoML . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32

# List of Tables


Table 2.1 – Comparison table of AutoML tools. . . . . . . . . . . . . . . . . . . . . . . . . 12
Table 2.2 – Comparison table of ML type tasks support in AutoML tools. . . . . . . . . . 13
Table 4.1 – VMs configuration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
Table 4.2 – Run Time Comparison: Pycaret full automatic and optimized . . . . . . . . . 34
Table 4.3 – Run Time: Spark manual classifiers . . . . . . . . . . . . . . . . . . . . . . . . 34
Table 4.4 – Titanic model using Spark MLlib . . . . . . . . . . . . . . . . . . . . . . . . . 34
Table 4.5 – Prediction metrics - Best Titanic Pycaret Pipeline using optimizations in pre-processing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
Table 4.6 – Prediction metrics - Best Titanic Pycaret pipeline full automated . . . . . . . 35
Table 4.7 – Autokeras MNIST runtimes . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
Table 4.8 – Learning Orchestra Autokeras MNIST runtimes . . . . . . . . . . . . . . . . . 36
Table 4.9 – Autokeras MNIST RAM Consumption . . . . . . . . . . . . . . . . . . . . . . 37
Table 4.10–Autokeras and Learning Orchestra MNIST RAM consumption . . . . . . . . 37
Table 4.11–ML metrics from Autokeras MNIST experiments . . . . . . . . . . . . . . . . 38
Table 4.12–Keras+TensorFlow MNIST experiment (from (RIBEIRO, 2021)) . . . . . . . 38

# Acronyms

**AD** Anomaly Detection. 12, 13

**AI** Artificial Intelligence. 7

**API** Application Programming Interface. v, vi, xii, 2, 6–11, 13, 16–24, 27, 38–40

**AR** Association Rules. 13

**AuFS** Advanced Multi-Layered Unification Filesystem. 5

**AutoML** Automated Machine Learning. v–vii, xii, 1–4, 6–11, 13–23, 26, 27, 31, 33, 35, 38, 39

**BL** Black-Box Model. 7

**CASH** Combined Algorithm Selection and Hyperparameter Optimization. 8

**CL** Classification. 12, 13

**CLU** Clustering. 12, 13

**CNN** Convolutional Neural Network. 9

**CPU** Central Processing Unity. 4, 31

**CRUD** Create, Retrieve, Update and Delete operations. 18

**CSV** Comma-separated Values. 24, 26

**DBS** Deep Belief System. 9

**DL** Deep Learning. xii, 6, 8, 9, 11, 13

**DM** Data Mining. xii, 6

**DN** Deep Network. 9

**DNN** Deep Neural Network. 8

**DT** Decision Tree. 33, 34

**GB** Gradient-Boosted Tree. 33–35

**GCP** Google Cloud Platform. 2, 14, 31

**GPU** Graphics Processing Unity. 4

**GUI** Graphical User Interface. 9–11

**HTTP** Hypertext Transfer Protocol. 17, 18

**IaaS** Infrastructure as a Service. 4

**IC** Image Classification. 13

**IP** Internet Protocol. 31

**JSON** JavaScript Object Notation. 17, 18, 27

**KDD** knowledge discovery from data. 6

**LR** Logistic Regression. 33, 34

**LSTM** Long Short-Term Memory. 9

**LXC** Linux Container. 5

**ML** Machine Learning. v, vi, viii, xii, 1, 2, 6–11, 13–16, 18, 33, 35, 37–39

**MLaaS** Machine Learning as a Service. 10

**MLC** Multi-Label Classification. 12, 13

**MNN** Multilayer Neural Network. 9

**MOJO** Model Object Optimized. 11

**NAS** Neural Architecture Search. 8–10, 12, 13, 16, 18, 19

**NB** Naive Bayes. 33, 34

**NLP** Natural Language Processing. 12, 13

**NN** Neural Network. 6, 8–10

**OS** Operation System. 4, 5, 31

**PaaS** Platform as a Service. 4

**RAM** Random Access Memory. v, vi, 4, 31, 37

**RE** Regression. 12, 13

**REST** Representational State Transfer. 2, 17, 18, 22, 24, 27

**RF** Random Forest. 33, 34

**RNN** Recurrent Neural Network. 9

**Saas** Software as a Service. 4

**SDK** Software Development Kit. 11

**SL** Shallow Learning. xii, 8

**TS** Time Series. 13

**URL** Uniform Resource Locator. 16–18, 27

**VC** Video Classification. 13

**vCPU** Virtual CPU. 31

**VM** Virtual Machine. viii, xii, 4, 5, 10, 14, 31, 33, 39

**VMM** Virtual Machine Monitor. 4

**Web** Web Designs. 10, 11

**XAI** eXplainable Artificial Intelligence. 7

# Contents


1 Introduction . . . . . 1
1.1 Goals . . . . . 2
1.2 Out of Scope . . . . . 2
1.3 Work Organization . . . . . 2
2 Literature Review . . . . . 4
2.1 Basic Concepts . . . . . 4
2.1.1 Cloud Computing . . . . . 4
2.1.2 Virtual Machine (VM) . . . . . 4
2.1.3 Microservices . . . . . 5
2.1.4 Container . . . . . 5
2.1.5 Docker . . . . . 5
2.1.6 Docker Swarm . . . . . 5
2.1.7 Keras . . . . . 6
2.1.8 MongoDB - No SQL storage . . . . . 6
2.1.9 Data Mining and Machine Learning - concepts and differences . . . . . 6
2.1.10 Automated Machine Learning (AutoML) . . . . . 6
2.1.11 Learning Orchestra Tool . . . . . 7
2.1.12 Feature Engineering . . . . . 7
2.1.13 Explainability . . . . . 7
2.1.14 NAS . . . . . 8
2.2 Related Work . . . . . 8
2.2.1 Shallow Learning works . . . . . 8
2.2.2 Deep Learning Works . . . . . 9
2.2.3 AutoML pipeline design tools . . . . . 10
2.2.4 Discussion . . . . . 11
3 Development . . . . . 14
3.1 Architecture . . . . . 14
3.1.1 VM and Container Orchestration . . . . . 14
3.1.2 Containers . . . . . 14
3.1.3 Learning Orchestra AutoML Installation . . . . . 15
3.1.4 Existing AutoML tools . . . . . 15
3.1.5 Services . . . . . 15
3.1.6 API Gateway . . . . . 16
3.1.7 Learning Orchestra AutoML REST API . . . . . 17
3.1.8 Learning Orchestra Clients . . . . . 22
3.2 Pipeline Examples . . . . . 23

3.2.1 Pycaret and Titanic . . . 23
3.2.2 Autokeras and MNIST . . . 27
**4 Experiments . . . 31**
4.1 Setup . . . 31
4.1.1 Container setup for Pycaret and Autokeras . . . 31
4.2 Metrics . . . 32
4.3 Titanic Experiment . . . 33
4.3.1 Experiment Overview . . . 33
4.3.2 Dataset . . . 33
4.3.3 Runtime Results . . . 33
4.3.4 ML Metrics Results . . . 34
4.4 MNIST Experiment . . . 35
4.4.1 Experiment Overview . . . 35
4.4.2 Dataset . . . 36
4.4.3 Runtime Results . . . 36
4.4.4 RAM Memory Consumption . . . 37
4.4.5 ML Metrics Results . . . 37
**5 Conclusion . . . 39**

**Bibliography . . . 41**

**Appendix 45**
**APPENDIX A The entire content of Titanic in Pycaret . . . 46**
**APPENDIX B The entire content of Mnist in Autokeras . . . 49**

# 1 Introduction

The data scientist 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. With increase of data amount used in data scientist processing, as well as its complexity because of the number of parameters to be tuned in a model, different tools for Automated Machine Learning (AutoML), like Katib (GEORGE et al., 2020), Pycaret [1] and ML-JAR [2], or distributed tools designed to build Machine Learning (ML) pipelines, like TFX (BAYLOR et al., 2017), Spark MLlib (MENG et al., 2016), Kubeflow (BISONG, 2019), XGBoost [3] and Ray (MORITZ et al., 2018), are earning more visibility.

In a word, AutoML can be understood to involve the automated construction of a ML pipeline on the limited computational budget (HE; ZHAO; CHU, 2021). The automatic discover of the best hyper-parameters combination, a way to discover under-performance training steps and the best classification algorithms are, thus, part of the AutoML pipeline. The AutoML benefits are at least two: i) simple and fast prototype of ML models well optimized, thus useful for beginners in data science; ii) very useful for senior data scientists as well because the AutoML outputs can be used to a fine tuning activity or even to explore alternatives and combine some of them.

Due to the recent interest in AutoML, many research surveys were published (YAO et al., 2018; ELSHAWI; MAHER; SAKR, 2019; HE; ZHAO; CHU, 2021; ZÖLLER; HUBER, 2021). They reinforced current trends in AutoML. Several tools were developed for AutoML goals and they are: Auto-Weka (THORNTON et al., 2013), TPOT (OLSON; MOORE, 2016), Auto-Sklearn (FEURER et al., 2019), H2O for AutoML (LEDELL; POIRIER, 2020), Pycaret [4], ML-JAR [5], Auto-Net (MENDOZA et al., 2016), Auto-Pytorch (ZIMMER; LINDAUER; HUTTER, 2021), Autokeras (JIN; SONG; HU, 2019), Katib (GEORGE et al., 2020), Google cloud AutoML [6], Microsoft Azure AutoML[7], Amazon SageMaker Autopilot plus the automatic tuning (DAS et al., 2020; PERRONE et al., 2020) and H2O Driverless AI[8].

As we can see, there are alternatives providing the AutoML services and others providing the entire ML process mentioned previously. On both cases, the existing tools do not support multiple AutoML tools. Very often they support some ML Toolkits, like Scikit-learn, MXNet, TensorFlow and Pytorch, and just one AutoML tool, e.g., the Kubeflow with Katib support.

[1] <https://pycaret.org/>
[2] <https://mljar.com/automl/>
[3] <https://xgboost.readthedocs.io/en/stable/>
[4] <https://pycaret.org/>
[5] <https://mljar.com/automl/>
[6] <https://cloud.google.com/automl>
[7] <https://docs.microsoft.com/en-us/azure/machine-learning/>
[8] <https://www.h2o.ai/products/h2o-driverless-ai/>

Another important limitation in the existing ML or AutoML pipeline tools is their incapacity to model each data pipeline step as an atomic processing unit, enabling transfer such step using a communication network or we can store it in volumes of a cloud environment, which means not using RAM-only storage. The atomic pipeline step can be re-executed and this behavior avoids entire pipeline runs with high computational costs.

## 1.1 Goals

The first goal of this work is to reduce a bit more the existing multi-AutoML tools support lacuna. For that, it is presented a new AutoML RESTful API to operate Pycaret and Autokeras tools.

The second goal is to present an alternative to avoid entire AutoML pipelines runs. For that, it is presented a transparent way to decouple existing Pycaret and Autokeras pipeline steps, enabling their re-execution, their distribution into cloud containers and their storage in cloud volumes.

The results expected:

- An efficient API in terms of runtime and memory consumption if compared with Pycaret and Autokeras operated directly via notebooks, like Zeppelin and Jupyter;
- A simple way to operate Pycaret and Autokeras using a unique API syntax, a unique Google Cloud Platform (GCP) deployment script and with Web support, thus supporting REST interoperable services;
- A transparent way to re-execute steps of Pycaret and Autokeras pipelines in GCP;
- A transparent pipeline step storage mechanism in cloud volumes, which means not RAM-only.

## 1.2 Out of Scope

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

## 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

AutoML pipelines. Chapter 4 details the extensions made in Learning Orchestra tool, precisely its AutoML layer. Chapter 5 details the experiments and experimental results. Finally, in Chapter 6 the conclusion and future research directions are described.

# 2 Literature Review

In this chapter, we present the most similar works found in the literature about AutoML. In the first part of the chapter, we also present the basic concepts for a better reader understanding.

## 2.1 Basic Concepts

In this section, we present the fundamental concepts useful for a better understanding of the work.

### 2.1.1 Cloud Computing

Cloud computing is a computing model where any computing resources, CPU, GPU, RAM, storage, OS, and software are delivered on demand over the internet. Five characteristics are found in cloud computing: large-scale computing resources, high scalability, and elasticity shared resource pool, dynamic resource scheduling, and general purpose. Cloud computing is offered in three main models:

- Infrastructure as a Service (IaaS): In this model essential IT components are offered, such as network resources, computers (virtual or dedicated hardware), and data storage space;
- Platform as a Service (PaaS): Provides deployment into the cloud infrastructure, removing the responsibility of managing the hardware infrastructure and operating systems from the user, focusing only on the deployment and management of its applications;
- Software as a Service (Saas): Allows users to use a complete application, such as Web-Email, without worrying about maintenance, management, or infrastructure;

These resources can be easily deployed and used at scale with minimal effort (MELL; GRANCE et al., 2011).

### 2.1.2 Virtual Machine (VM)

A Virtual Machine (VM) is a computational resource that emulates a conventional physical machine through software. Several virtual machines, with their OS and applications, can be executed on the same physical machine called a host, and each VM is called a guest machine. Hypervisor (or Virtual Machine Monitor (VMM)) is computer software that allows many virtual machines (VMs) to run independently and concurrently with isolation and efficiency (GOLDBERG, 1974).

### 2.1.3 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 is a software architecture where the application is split into small, loosely coupled services that are highly maintainable and testable with independent deployment and communication via API. Unlike monolithic applications where all processes are highly coupled and run as a single service.

### 2.1.4 Container

Containers as well as VMs are used for virtualization, packaging the application with its dependencies, and providing isolated environments for software execution. By operating at the OS level and sharing the system kernel, the container is much lighter, using only a fraction of the memory required by VMs [1].

### 2.1.5 Docker

Docker was born as an open-source project aiming to simplify the creation and management of Linux containers. To isolate the user database, namespaces, and network, docker uses Linux Containers and Linux Container (LXC), a user-space control package for Linux Containers through kernel-level namespaces. Another function of this package is hardware resource control called Control Groups (cgroups), allowing docker to perform fair resource sharing. Docker adopts Advanced Multi-Layered Unification Filesystem (AuFS) as the filesystem in its containers. AuFS provides the ability to reuse an image for different containers and perform version control of images, keeping image sizes to a minimum by deploying only the changes from the previous image (MERKEL, 2014).

### 2.1.6 Docker Swarm

Docker Swarm is Docker's native tool for orchestrating containers in a cluster. Swarm is composed of two types of docker hosts, the Manager, and the Workers. The Host Manager runs all docker commands to manage the swarm state, while the Workers just receive and execute tasks. Docker swarm provides: scaling of services, constant cluster state monitoring,

[1] <https://cloud.google.com/learn/what-are-containers>

automatically assigning addresses to the containers on the overlay network, load balancing, TLS mutual authentication between itself and all other nodes, and rolling updates [2].

### 2.1.7 Keras

Running on top of the machine learning platform TensorFlow, Keras is a high-level Neural Network (NN) API. Written in Python with a focus on enabling fast experimentation, its API is Simple, Flexible, and powerful[3], making Keras the most widely adopted one (MOOLAYIL; MOOLAYIL; JOHN, 2019).

### 2.1.8 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). MongoDB is an open-source document-oriented database that supports large volumes of data. Its storage does not use only tables, so it is characterized as a NoSQL database.

### 2.1.9 Data Mining and Machine Learning - concepts and differences

Data Mining (DM) is the process of discovering interesting patterns, models, and other kinds of knowledge in large data sets (HAN; PEI; TONG, 2022).DM is also a particular step in knowledge discovery from data (KDD) process. The additional steps in the KDD process, such as data preparation, data selection, data cleaning, incorporation of appropriate prior knowledge, and proper interpretation of the results of mining ensure that useful knowledge is derived from the data (FAYYAD; PIATETSKY-SHAPIRO; SMYTH, 1996).

ML is the technique that improves system performance by learning from experience via computational methods (ZHOU, 2021). The similarity between DL and ML are: Both are analytics processes, are good at pattern recognition, and learn from data to improve decision-making.

Differences between DL and ML are: DL focuses on discovering properties of the data, while ML focuses on predicting and learning from the data and developing intelligent systems.

### 2.1.10 Automated Machine Learning (AutoML)

To reduce the onerous development costs in ML, a novel idea of automating the entire pipeline of ML has emerged (HE; ZHAO; CHU, 2021). There are various definitions of AutoML. For example, according to (ZÖLLER; HUBER, 2021), AutoML is designed to reduce the demand for data scientists and enable domain experts to automatically build ML applications without

[2] <https://docs.docker.com/engine/swarm/>
[3] <https://keras.io/about/>

much requirement for statistical and ML knowledge. In (YAO et al., 2018), AutoML is defined as a combination of automation and ML. In a word, AutoML can be understood to involve the automated construction of a ML pipeline on the limited computational budget (HE; ZHAO; CHU, 2021).

### 2.1.11 Learning Orchestra Tool

The Learning Orchestra tool (RIBEIRO, 2021) is a cloud solution providing a RESTfull API to develop ML pipelines using Spark MLlib, Scikit-learn and TensorFlow. Besides the possibility to operate several ML Toolkits using a single API syntax, the Learning Orchestra provides atomic pipeline steps for TensorFlow and Scikit-learn Toolkits. The consequence of an atomic pipeline step is the capacity to re-execute some steps and not the entire pipeline all the times. Furthermore, these steps can be stored in cloud volumes and transmitted via communication networks.

In terms of pipeline steps, Learning Orchestra offered eleven services to implement all ML models. The services are: Dataset, Model, Transform, Explore, Tune, Training, Evaluate, Predict, Builder, Function and Observe. In this work we extended some of those eleven services to provide AutoML capabilities.

### 2.1.12 Feature Engineering

According to (DONG; LIU, 2018), in ML, Data Mining, and Data Analytics, a feature is an attribute or variable used to describe some aspect of individual data objects. Feature Engineering aims to maximize resource extraction from raw data for utilization by the classification algorithms and models (HE; ZHAO; CHU, 2021).

### 2.1.13 Explainability

The evolution of ML techniques has brought increasingly complex and accurate models; however, with less transparent internal mechanisms. Due to this characteristic they are called Black-Box Model (BL). Understanding the internal mechanisms of the model is very important to detect biases and failures, understand when the model predictions have good or bad results, obtain insights into the applied problem and justify business decisions based on AI models, thus increasing the robustness and confidence of the results (MOLNAR et al., 2022). Due to the need to understand how complex models work, the area of eXplainable Artificial Intelligence (XAI) has emerged. XAI aims to increase the explainability of AI systems, making them more understandable to humans.

### 2.1.14 NAS

Recently, many computational problems have been solved with Deep Neural Network (DNN) applications. We realized that the success of these applications came from an arduous process of trial and error, of developing an efficient Neural Network (NN) architecture for the specific problem (REN et al., 2021). Neural Architecture Search (NAS) is a technology that aims to find the best NN architecture for a given learning task and dataset, making it an important improvement to AutoML (JIN; SONG; HU, 2019).

## 2.2 Related Work

There are AutoML works designed for Shallow Learning (SL) and for Deep Learning (DL). There are also some solutions working only with specific data type, e.g., images, text, stream, tabular and so forth. In this section, we divided the literature works according to the learning type. We also present solutions like Learning Orchestra in a separated section, i.e., ML pipeline tools with pre-processing (including explore and transform), monitoring/debugging and some other pipeline steps that are not present in AutoML core works, but some of these tools support AutoML pipelines.

### 2.2.1 Shallow Learning works

Auto-Weka (THORNTON et al., 2013) is one of the most well known tool to solve the problem named Combined Algorithm Selection and Hyperparameter Optimization (CASH). It uses Bayesian techniques combined with Random Forest based data organization to enable multiple search spaces for the ML algorithms. The users can explore many hyper-parameters automatically, resulting in a suggestion of the best algorithm and their best parameters combination. Auto-Weka was developed using the Weka API (HALL et al., 2009) facilities and its vast amount of classification algorithms. In (KOTTHOFF et al., 2017), the authors introduced the support for regression algorithms and the adoption of all performance metrics of Weka. Besides that, Auto-Weka 2.0 can run in parallel, i.e., on a multicore single machine.

More recent works were done and some of them use the Scikit-learn ML toolkit (PEDREGOSA et al., 2011). The first one is the TPOT (OLSON; MOORE, 2016) (Tree-based Pipeline Optimization Tool), which uses stochastic algorithms in conjunction with genetic ones. One of the Scikit-learn classification algorithms is selected with its best parameters combination. The Auto-Sklearn tool (FEURER et al., 2019) is a second Scikit-learn AutoML tool, which is a Bayesian approach, resulting in a combination of fifteen classifiers with fourteen pre-processing methods, thus a robust alternative for the entire ML pipeline and not just the learning steps. In 2020, the Auto-Sklearn announced two improvements: portfolios instead of meta-features to represent similarities; and an anticipated stop condition detection on each ML pipeline.

The H2O for AutoML (LEDELL; POIRIER, 2020) is a solution designed for structured data and it is an extension of the H2O open source ML platform (H2O.AI, 2020). It uses the combination of random search and stacked ensembles to produce a large number of ML models in a short period of time. There are H2O AutoML API for the following programming languages: R, Python, Java and Scala. During the model tests it is possible to make fine adjustments via command line or Graphical User Interface (GUI) using the Flow tool [4]. At the end of the AutoML pipeline, it is presented the ranking of the ML algorithms based on the accuracy of the models.

Deployment is a fundamental step of a data scientist activity, but few AutoML solutions presented a way to deploy a model in production in a public cloud environment. Pycaret [5] is a library for the automation of the ML pipeline, including the deployment of the selected model in Amazon AWS cloud environment [6]. The Pycaret is not only used for AutoML or classification problems, but also for clustering, regression, outlier detection, natural language processing and association rules problems, thus it can be considered a Data Mining library with AutoML support. It also has hyper-parameter tuning, feature extraction, ensembles creation and model comparisons.

### 2.2.2 Deep Learning Works

Very often the data scientist is trying to find Multilayer Neural Network (MNN) architectures to solve DL problems. Such architectures can discover complex patterns and handle large amount of data [7]. Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), Deep Networks (DNs) and Deep Belief Systems (DBSs) are all examples of Multilayer Neural Networks (MNNs). Thus, the AutoML works use such useful libraries, frameworks and middlewares internally.

Auto-Net (MENDOZA et al., 2016) in its first version was developed as an extension of the Auto-Sklearn (FEURER et al., 2019). Its discover process adopts a Bayesian optimization and some steps of the AutoML pipeline discover is done by Auto-Sklearn (FEURER et al., 2019); this way, it incorporates regression and classification components into its solution. Some modern DL techniques, like feed-forward networks such as CNN, RNN including Long Short-Term Memory (LSTM) and support of CPUs and GPUs due to (Theano Development Team, 2016) expression compiler, are performed by the library Lasagne (DIELEMAN et al., 2015). In your second version (MENDOZA et al., 2019) as developed over Pytorch (PASZKE et al., 2019) and it is used to perform Neural Architecture Search (NAS).

The Auto-Pytorch (ZIMMER; LINDAUER; HUTTER, 2021) is a kind of successor of Auto-Net and one of its advantage is the hyper-parameter tuning in conjunction with the NN architecture discover process, completing even more the DL pipeline. A second contribution of Auto-Pytorch when compared with Auto-Net is the support for tabular and image data types.

[4] <https://docs.h2o.ai/h2o/latest-stable/h2o-docs/flow.html>
[5] <https://pycaret.org/>
[6] <https://aws.amazon.com>
[7] Techopedia - <https://www.techopedia.com/definition/33268/multi-layer-neural-network>

Auto-Keras (JIN; SONG; HU, 2019) is another important AutoML tool that also uses Bayesian optimization to perform NAS. One of its advantage is its simple API design, where only three API calls are sufficient to start the AutoML pipeline. To perform fine adjustments in the results obtained by Auto-Keras, there are other API services. Its single machine deployment limitation imposes impossibilities to handle large models or discover a large number of architectures due to the time consumed for them.

One of the first AutoML solution designed to be deployed in cloud environments, including the support for containerization, is Katib (GEORGE et al., 2020). It integrates with Kubeflow (BISONG, 2019), using its capacity to run ML pipelines over a cluster of Docker containers. Katib supports the following ML toolkits: Tensorflow, MXNet and PyTorch. Besides Kubeflow, Katib integrates with XGBoost, enabling other pipelines over a cluster. In terms of AutoML, it supports the hyper-parameter tuning, early-stop and NAS. The data scientist can monitor the pipeline via a Web GUI.

### 2.2.3 AutoML pipeline design tools

There are several pipeline steps, like transformation and explore ones, not performed by the works detailed before, thus in this section we present the state-of-art in terms of AutoML pipeline design tools, where, for instance, collaboration of models among data scientists worldwide and monitoring of the training step after the AutoML discover of the best Neural Network architecture are both feasible. In summary, they represent a natural way to increase complexity in the code development to deliver even more facilities for the data scientists.

The data scientist can deploy their own AutoML tool or even an entire ML pipeline design tool using the existing public cloud environments, but several companies are now offering the AutoML and not only the VMs or containers services on demand. In summary, the users pay to use Machine Learning as a Service (MLaaS), which are offered on demand and with many facilities regarding deployment and configuration issues.

Google has its Cloud AutoML [8], which is a set of three service domains: vision, natural language and tabular. The vision service is useful to detect, locate and quantify objects on images or videos using existing models previously trained and customized. The natural language service extracts entities, performs sentimental analysis and has support for more than 50 languages. Finally, the tabular service offers a high quality training step, extracts features and enables hyper-parameter tuning. Since it is a Google platform, it has integration with Dataproc [9] and BigQuery [10] products.

Besides the Google AutoML services, it has also the entire ML platform named Google

[8] Cloud AutoML - <https://cloud.google.com/automl>
[9] Dataproc - <https://cloud.google.com/dataproc>
[10] BigQuery - <https://cloud.google.com/bigquery>

Vertex AI [11]. The platform enables training the existing code or personalized ones and models comparisons using the Google Cloud AutoML explained previously. All models are stored in a unique ML model storage, thus they can be deployed easily at the same end-points offered by Vertex AI.

Microsoft has its Azure AutoML[12], which is useful to automate the ML pipeline. For that, it adopts a collaborative filtering and a hyper-parameter tuning service using a Bayesian optimization. The collaborative filtering tries to find pipelines from an existing database with million of experiments with different datasets. Other important service is the feature extraction. The Azure AutoML has also an offline version with a Software Development Kit (SDK) in Python. It has Web GUI for a better training understanding.

SageMaker (JOSHI, 2020) is an Amazon platform with several tools inside. All of them are available online, so it can be considered the first ML pipeline design tool as a service. The tools inside SageMaker are: Ground Truth (labeling), Data Wrangler (transformation), Feature Store (storage), Processing (resource engineering), Clarify (detection of statistical trends and explainability) (HARDT et al., 2021), Autopilot (AutoML) (DAS et al., 2020), Training (PERRONE et al., 2020), Adjust, AWS (hosting), Monitor (monitoring and logs) and Pipelines (continuous integration and development - CI/CD). The data scientist uses Sagemaker through API or its integrated development environment, named Sagemaker studio, the last accessed also through Web. Besides the Amazon internal tools, SageMaker also operates with existing DL toolkits, like MXNet, Pytorch and Tensorflow.

H2O Driverless AI[13] is the commercial version of the H20 AutoML tool (LEDELL; POIRIER, 2020). It offers a Web interface, accompanied by a Python and R API for the platform usage. Unlike its open-source version, it supports not only tabular data, but also stream, raw text, natural language processing, image and video datasets. H2O Driverless AI is data read agnostic, which means it can receive data from any source. For data explainability services, it has automatic plotting of graphs and tables, as well as detailed reports about data quality. As the final result of the ML process, the platform provides the automatic creation of a Model Object Optimized (MOJO)[14] object or a client/server application, both useful to obtain the pipeline metadata with the best overall score of different models, which can be used in production.

### 2.2.4 Discussion

In this chapter, a brief analysis will be made of the characteristics of the tools discussed in the previous sections, making a comparison of the functionalities delivered by them.

As we can see in the table 2.1, Auto-Weka (THORNTON et al., 2013) is the only tool

[11] Vertex AI - <https://cloud.google.com/>
[12] <https://docs.microsoft.com/en-us/azure/machine-learning/>
[13] <https://www.h2o.ai/products/h2o-driverless-ai/>
[14] <https://docs.h2o.ai/h2o/latest-stable/h2o-docs/productionizing.html>

that does not have feature engineering, all the others do. All analyzed works have multi-core execution, but the only ones to present distributed execution are the tools described in the section: ML pipeline design tools and Katib (GEORGE et al., 2020).

All also have a certain level of explainability, that is, a way of plotting some metadata about the trained models.

Pycaret [15] and H20 (LEDELL; POIRIER, 2020) stand out for natively offering a way to deploy, among shallow learning tools.

About the tools described in the ML pipeline design tools section: they all offer a complete environment for the data scientist covering the entire production cycle of a data model. The only one that doesn't offer NAS is Microsoft's[16] solution. As they work as a service, they abstract the complexity of configuring and deploying the cluster from the user.

| Tool | Multi-core | Distributed | Explainability | Deployment | Feature engineering |
|---|---|---|---|---|---|
| Auto-Weka | ✓ | ✗ | ✓ | ✗ | ✗ |
| TPOT | ✓ | ✗ | ✓ | ✗ | ✓ |
| Auto-Sklearn | ✓ | ✗ | ✓ | ✗ | ✓ |
| H2O AutoML | ✓ | ✗ | ✓ | ✓ | ✓ |
| Pycaret | ✓ | ✗ | ✓ | ✓ | ✓ |
| Auto-Net | ✓ | ✗ | ✓ | ✗ | ✓ |
| Auto-Pytorch | ✓ | ✗ | ✓ | ✗ | ✓ |
| Katib | ✓ | ✓ | ✓ | ✓ | ✓ |
| Auto-Keras | ✓ | ✗ | ✓ | ✗ | ✓ |
| Google AutoML | ✓ | ✓ | ✓ | ✓ | ✓ |
| Azure | ✓ | ✓ | ✓ | ✓ | ✓ |
| SageMaker | ✓ | ✓ | ✓ | ✓ | ✓ |
| H2O Driverless AI | ✓ | ✓ | ✓ | ✓ | ✓ |

Table 2.1 – Comparison table of AutoML tools.

The tools mentioned in the previous sections support the following ML-Tasks:

- Classification (CL)
- Regression (RE)
- Multi-Label Classification (MLC)
- Clustering (CLU)
- Anomaly Detection (AD)
- Natural Language Processing (NLP)

[15] <https://pycaret.org/>
[16] <https://docs.microsoft.com/en-us/azure/machine-learning/>

- Association Rules (AR)
- Image Classification (IC)
- Video Classification (VC)
- Time Series (TS)
- Neural Architecture Search (NAS)

| Tool | ML-Tasks |
| --- | --- |
| Auto-Weka | CL, RE |
| TPOT | CL, RE |
| Auto-Sklearn | CL, RE |
| H2O AutoML | CL, RE |
| Pycaret | CL, RE, CLU, AD, NLP, AR |
| Auto-Net | NAS, CL, RE, IC |
| Auto-Pytorch | NAS, CL, RE, IC |
| Katib | NAS, CL, RE, IC |
| AutoKeras | NAS, CL, RE, IC |
| Google AutoML | CL, RE, NAS, IC, VC, NLP |
| Azure | CL,RE |
| SageMaker | CL, RE, MLC, IC, NAS |
| H2O Driverless AI | CL, RE, MLC, NLP, NAS |

Table 2.2 – Comparison table of ML type tasks support in AutoML tools.

In all related work section, the tool with the widest range of supported task types is Pycaret, supporting clustering tasks, anomaly detection, natural language processing and association rules.

Regarding AutoML tools for DL, Katib is the most complete with options for distributed execution, deployment, feature engineering and explainability tools, but as it is a Kubernetes[17] native project, thus its deployment cannot be performed together with Learning Orchestra, which is limited to Docker Swarm[18] as the container orchestrator. The best remaining options are Auto-Pytorch and Autokeras, both of which have the same characteristics. Autokeras was chosen to be the AutoML tool for DL in this work, as it has a simpler API to use, meeting one of AutoML's characteristics, the democratization of creating ML systems.

[17] <https://kubernetes.io/>
[18] <https://docs.docker.com/engine/swarm/>

# 3 Development

In this chapter, it is detailed the Learning Orchestra AutoML architecture. Besides the architecture, an example of how we can use the Learning Orchestra AutoML services is presented at the end of the chapter.

## 3.1 Architecture

The Learning Orchestra architecture is organized into layers, precisely into eight layers as Figure 3.1 illustrates. The AutoML extensions were made on almost all layers, thus we use a similar architecture presented in (RIBEIRO, 2021). Fewer pipeline steps were implemented (precisely, seven API services) for AutoML when compared with the eleven services implemented for the entire ML API possibilities.

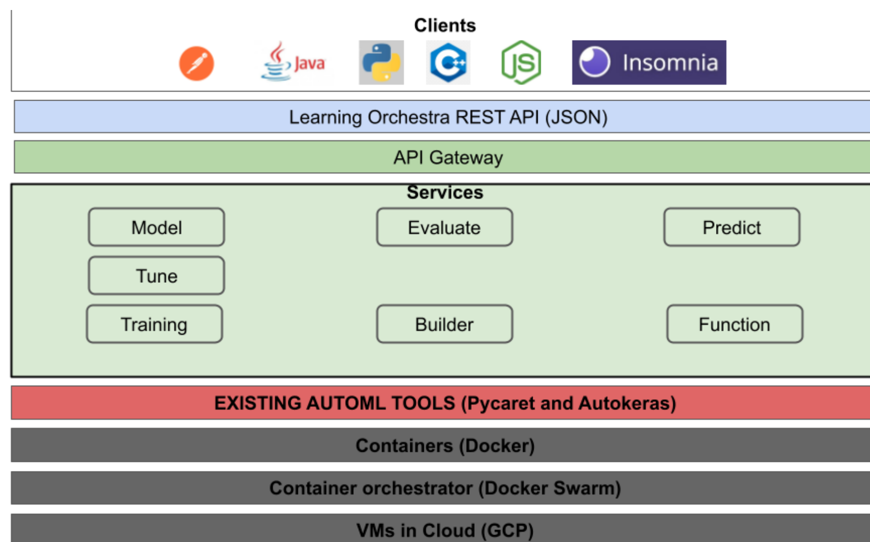


Figure 3.1 – Learning Orchestra AutoML architecture. Adapted from (RIBEIRO, 2021)

### 3.1.1 VM and Container Orchestration

These two layers use the same GCP and Docker Swarm scripts presented in the first version of Learning Orchestra.

### 3.1.2 Containers

In this work, we added PyCaret and Autokeras container images and they also represent a processing container type, this way they can be replicated over the deployed VMs and managed by the Docker Swarm, identical to any other processing container (Spark, Scikit-learn, TensorFlow or

MongoDB) of the previous Learning Orchestra version (RIBEIRO, 2021). The AutoML services were coded also in Python, so the language interpreter is present in processing containers.

### 3.1.3 Learning Orchestra AutoML Installation

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 Orchestra AutoML[1] and, as root user, you have to run the file $run.sh$, located on root directory. This file deploys the Learning Orchestra AutoML system on Docker Swarm.

### 3.1.4 Existing AutoML tools

This is not really a layer, but it is important to explain due to Learning Orchestra interoperability capacity. It represents the wrapped solutions, regardless it is a set of ML toolkits, a set of AutoML tools, a set of hyper-parameter tuning solutions or a set of monitoring/debugging tools. On this work, we extended the original supported tools and added the Pycaret and Autokeras tools. We understand that the catalog of possibilities for the data scientist is a fundamental requirement for the Learning Orchestra pipeline tool.

### 3.1.5 Services

There are eleven services types in Learning Orchestra AutoML, which means eleven pipeline steps possibilities to use and they are: Dataset, Model, Transform, Explore, Tune, Training, 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 following pipeline steps were not changed for AutoML, thus not present in Figure 3.1: Dataset, Transform, Explore and Observe. More information about them in (RIBEIRO, 2021). The pipeline steps: Model, Training, Tune, Evaluate, Predict, Builder and Function were extended to operate Pycaret and Autokeras tools, thus described in this work.

The service types details:

- Model: Responsible for loading machine learning models from existing repositories. In terms of AutoML, it can be used to create new models to be combined with searched ones (model is also referred as neural architectures) by the AutoML tools.

[1] <https://github.com/learningOrchestra/automl>

- Training: Probably it is the most computational expensive service of an ML or AutoML pipeline, because the models will be trained for best learn the subjacents patterns on data. A diversity of algorithms can be executed and compared, like Support Vector Machine (SVM), Random Forest, Bayesian inference, K-Nearest Neighbors (KNN), Deep Neural Networks (DNN), and many others. In AutoML, it represents the NAS pipeline step or sometimes called models comparisons step of AutoML pipelines.

- Tune: Performs the search for an optimal set of hyper-parameters for a given model. It can be made through strategies like grid-search, random search, or Bayesian optimization. For AutoML, it is normally called after the NAS or models comparisons or any other sort of training step.

- 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. In AutoML, there is always an evaluate step to produce metrics, which runs after the tune pipeline step. In PyCaret tool, for instance, the method $finalize$ is called for such purpose.

- Predict: The model must be used to predict, this way it runs and classifies new dataset instances. In Titanic model, it classifies if the passenger will live or not after the terrible accident. In MNIST model, it identifies letters from images. As we can see, prediction is the final step of a ML pipeline, thus present in AutoML pipelines as well.

- Builder: Responsible to execute Autokeras or PyCaret entire pipelines in Python, offering an alternative way to use the Learning Orchestra AutoML system just as a deployment alternative and not an environment for building pipelines.

- Function: Responsible to wrap a Python function, representing a wildcard for the data scientist when there is no Learning Orchestra support for a specific AutoML service. It is different from Builder service, since it does not run the entire pipeline. Instead, it runs just a Python function of Autokeras or PyCaret APIs on a cluster of containers.

### 3.1.6 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 - '$IPaddress/api/learningOrchestra/v1/dataset/csv/$'. 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. Basically, the PyCaret and Autokeras routes were added into KrakenD gateway to provide AutoML services.

### 3.1.7 Learning Orchestra AutoML REST API

There are seven services types on Figure 3.1 and all of them are called following three simple rules. The rules for URL syntax are:

- *rule one*: PyCaret or Autokeras services have the tool name on the URL

  Ex:

  - *IPaddress/api/learningOrchestra/v1/evaluate/pycaret/*
  - *IPaddress/api/learningOrchestra/v1/builder/autokeras/*
  - *IPaddress/api/learningOrchestra/v1/model/pycaret/*

  Since these tools implement several alternatives for tune, training, predict, evaluate and other service types, the URL does not contain the service name on such API calls. The JSON message has all the AutoML tool 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/*
  - *IPaddress/api/learningOrchestra/v1/dataset/csv/*

- *rule three*: The Function service is a Python alternative for the data scientist. The tool to be used by Learning is inserted into JSON and not in the URL. Other programming languages will be supported by Learning Orchestra in future versions, so we decided to inform the language adopted on the URL

  - *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. training 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 training, evaluate or tune AutoML step 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 training, 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 AutoML API using a specification. We decided to adopt the Open API Initiative (API, 2021). The server used to code and host the Learning Orchestra AutoML system API is Swagger Hub (HUB, 2021). Function REST CRUD services, as well as the other API service types illustrated in Figure 3.1 are detailed at - <https://app.swaggerhub.com/apis-docs/learningOrchestra/automl/1.0.0>.

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 training, predict or any other AutoML API call. There is also a REST API call using JSON objects to observe the conclusion of an AutoML step submitted previously URL

- *IPaddress/api/learningOrchestra/v1/observe/{pipeline_step_name}*

The Observer is implemented using MongoDB client, where notifications of collections updates are performed without blocking the client with successive results checks. The Observer service was not changed, thus not described in this work. For more information about this API service and other ML useful services, like Dataset, Transform and Explore, see the work (RIBEIRO, 2021).

The Model API service summary:

- method POST: IPaddress/api/learningOrchestra/v1/model/pycaret - starts the creation or import of an existing model during the AutoML pipeline. It is useful to improve the NAS step. For that, it invokes an existing PyCaret *create_model* method. The output is a well trained model. Any successive API call must use PyCaret, since the stored binary object format is only compatible with such a tool;
- method GET: IPaddress/api/learningOrchestra/v1/model/pycaret - retrieves all models created or imported previously using PyCaret, i.e., all models not found by the NAS step.

- method GET: IPaddress/api/learningOrchestra/v1/model/pycaret/{modelname} - retrieves a specific model metadata created or imported using PyCaret;
- method DELETE: IPaddress/api/learningOrchestra/v1/model/pycaret/{modelname} - deletes a specific model metadata created or imported using PyCaret;
- method PATCH: IPaddress/api/learningOrchestra/v1/model/pycaret/{modelname} - updates a specific model metadata created or imported using PyCaret. The model creation is re-executed via another $create_model$ method call.
- method POST: IPaddress/api/learningOrchestra/v1/model/autokeras - starts the creation or import of an existing model during the AutoML pipeline. It just returns a model to be trained automatically, i.e., it combines tune and training into a single pipeline step. Any successive API call must use Autokeras, since the stored binary object format is only compatible with such a tool;
- method GET: IPaddress/api/learningOrchestra/v1/model/autokeras - retrieves all models created or imported previously using Autokeras, i.e., all models not found by the NAS step.
- method GET: IPaddress/api/learningOrchestra/v1/model/autokeras/{modelname} - retrieves a specific model metadata created or imported using Autokeras;
- method DELETE: IPaddress/api/learningOrchestra/v1/model/autokeras/{modelname} - deletes a specific model metadata created or imported using Autokeras;
- method PATCH: IPaddress/api/learningOrchestra/v1/model/autokeras/{modelname} - updates a specific model metadata created or imported using Autokeras. The model creation is re-executed via another $create_model$ method call.

The Training API service summary. Note that, Pycaret does not implement Training explicitly. This way, there is no API Training service for Pycaret. Instead, it returns a model well trained, requiring sometimes just an extra tuning pipeline step before the Prediction API service call:

- method POST: IPaddress/api/learningOrchestra/v1/train/autokeras - creates a models comparisons step of a AutoML pipeline using the Autokeras $compare_models$ method. Any successive API call must use Autokeras, since the stored binary object format is only compatible with such a tool;
- method GET: IPaddress/api/learningOrchestra/v1/train/autokeras - retrieves all training metadata performed previously using Autokeras;
- method GET: IPaddress/api/learningOrchestra/v1/train/autokeras/{objectname} - retrieves a specific training metadata performed previously using Autokeras;

- method DELETE: IPaddress/api/learningOrchestra/v1/train/autokeras/{objectname} - deletes a specific training object created previously using Autokeras;
- method PATCH: IPaddress/api/learningOrchestra/v1/train/autokeras/{objectname} - updates a specific training object created previously using Autokeras. The models comparisons operation is re-executed.

The Tune API service summary. Note that, Autokeras does not implement tune explicitly, thus there is no Tune API call for Autokeras. Instead, during the training step the hyper-parameters are tuned to perform better training results:

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

The Predict API service summary:

- method POST: IPaddress/api/learningOrchestra/v1/predict/pycaret - creates a predict binary object in a volume using a previous object from a Tune or other AutoML pipeline step. For that, it invokes the predict method from PyCaret. Any successive API call must use PyCaret, since the stored binary object format is only compatible with such a tool;
- method GET: IPaddress/api/learningOrchestra/v1/predict/pycaret - retrieves all predict metadata performed previously using PyCaret;
- method GET: IPaddress/api/learningOrchestra/v1/predict/pycaret/{objectname} - retrieves a specific predict metadata performed previously using PyCaret;
- method DELETE: IPaddress/api/learningOrchestra/v1/predict/pycaret/{objectname} - deletes a specific predict object created previously using PyCaret;

- method PATCH: IPaddress/api/learningOrchestra/v1/predict/pycaret/{objectname} - updates a specific predict object created previously using PyCaret. The predict operation is re-executed.

  The existing Tensorflow and Scikit-learn Predict API calls are maintained and to understand them a complete description is provided by the work (RIBEIRO, 2021).

The Evaluate API service summary:

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

  The existing Tensorflow and Scikit-learn Evaluate API calls are maintained and to understand them a complete description is provided by the work (RIBEIRO, 2021).

The Builder API service summary:

- method POST: IPaddress/api/learningOrchestra/v1/builder/pycaret - creates a builder pipeline binary object in a volume. For that, it invokes several existing PyCaret AutoML methods to run the entire pipeline;
- method GET: IPaddress/api/learningOrchestra/v1/builder/pycaret - retrieves all builder metadata performed previously using PyCaret;
- method GET: IPaddress/api/learningOrchestra/v1/builder/pycaret/{objectname} - retrieves a specific builder metadata performed previously using PyCaret;
- method DELETE: IPaddress/api/learningOrchestra/v1/builder/pycaret/{objectname} - deletes a specific builder object created previously using PyCaret;

- method PATCH: IPaddress/api/learningOrchestra/v1/builder/pycaret/{objectname} - updates a specific builder object created previously using PyCaret. The builder pipeline operation is re-executed.

  The existing Spark MLlib Builder API calls are maintained and to understand them a complete description is provided by the work (RIBEIRO, 2021).

The Function API service summary:

- method POST: IPaddress/api/learningOrchestra/v1/function/python - executes a Python function, regardless it is used Pycaret or Autokeras methods inside such a function. The output is a binary object stored in a volume. Any successive API call must use Pycaret or Autokeras, 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.

### 3.1.8 Learning Orchestra Clients

The last layer of Figure 3.1 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.

Learning Orchestra provides 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/tree/automl>) and (documentation - <https://github.com/learningOrchestra/pythonClient/tree/automl>. We omit its details in this section, since on Section 3.2 we have explained its utilization during the examples explanation. Minor extensions were made to the Python Client for use with the new AutoML services. The Python Client design pattern is to extend base classes of each service to all supported tools, so the $Model$, $Evaluate$, $Predict$, $Tune$, $Training$ and $Builder$ classes have been extended to the respective derived classes: $ModelPycaret$, $ModelAutokeras$,

$EvaluatePycaret$, $EvaluateAutokeras$, $PredictPycaret$, $PredictAutokeras$, $TuneAutokeras$, $TrainingPycaret$, $TrainingAutokeras$, $BuilderPycaret$, $BuilderAutokeras$. In summary, to add a new tool support in Learning Orchestra, you need to maintain the API services and configure new routes on the gateway of Learning Orchestra, and update a Python Client.

## 3.2 Pipeline Examples

In this section, we present how to operate Pycaret and Autokeras using the Python client of Learning Orchestra. The atomic AutoML pipeline steps are highlighted in the code.

### 3.2.1 Pycaret and Titanic

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.
- 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.

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

- 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 DataType Learning Orchestra pipeline steps were used to build the Titanic pipeline using the Python Client to simplify even more the REST API access. The following imports are required.

```
1  from learning_orchestra_client.builder.builder import BuilderPycaret
2  from learning_orchestra_client.dataset.csv import DatasetCsv
3  from learning_orchestra_client.transform.data_type import TransformDataType
```

The cluster IP is also required to provide an access.

```
8  CLUSTER_IP = "http://34.151.210.120"
```

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 to download the training 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 step. Line 18 illustrates the wait condition for the dataset download part of code.

```
10 dataset_csv = DatasetCsv(CLUSTER_IP)
11
12 dataset_csv.delete_dataset("train")
13 dataset_csv.insert_dataset_async(
14     url="https://raw.githubusercontent.com/JonatasMiguel/"
15         "PycaretTitanic/main/train.csv",
16     dataset_name="train",
17 )
18 dataset_csv.wait(dataset_name="train",
```

Next, in the Transform service, we need to change some attribute types, since some of them must be numbers. The Client method *update_dataset_type_async* is called for the training dataset (see Lines 29 to 31). The attributes $Age$, $Fare$, $Parch$, $Pclass$ and $SipSp$ are updated. A wait condition API call is performed (Lines 33 to 34), similar to other steps.

```
transform_data_type = TransformDataType(CLUSTER_IP)
type_fields = {
    "Age": "number",
    "Fare": "number",
    "Parch": "number",
    "Pclass": "number",
    "SibSp": "number"
}
transform_data_type.update_dataset_type_async(
    dataset_name=f"train",
    types=type_fields)

transform_data_type.wait(dataset_name=f"train",
                          timeout=1)
```

PyCaret pipeline is implemented from lines 37 to 65. The Learning Orchestra runs this pipeline as a Builder service. PyCaret has six modules available, which can be imported depending on the type of experiment you want to perform. They are: Classification, Regression, Clustering, Anomaly Detection, Natural Language Processing and Association Rule Mining. In Titanic experiment, PyCaret Classification module must be imported (Line 38).

```
from pycaret import classification
```

Next, we set up our experiment (Lines 40 to 47). The $setup$ method represents a Transformation in Learning Orchestra, since it is the first step of a PyCaret pipeline and must be called before executing any other function. In this step, many options to configure a pipeline is supported and in Titanic the following parameters are configured: Data Type Inference, Data Cleaning and Preparation, Data Sampling, Training Test Split and Assigning Session ID as seeds. The parameters $data$ and $target$, are the only mandatory ones, which respectively define the training data and the feature that we want to predict (Lines 41 to 42).

In the same way as (RIBEIRO, 2021), the column $Ticket$ has been removed from the training process, as it does not provide relevant information and reduces the accuracy of the model (Line 44).

PyCaret automatically detects the data type of each feature. However, this inference is not perfect. This way, we decided to inform the data types because it is an important information used by several algorithms, for instance, the missing value imputation for numeric and categorical features must be performed separately. In our experimental evaluations, we observed an improvement in the Accuracy and F1-score metrics when types are set for $Age$, $Fare$, $Parch$, $Pclass$ and $SipSp$.

The *setup* method is also responsible to split the training dataset into training and testing datasets with the default 70:30 ratio. In PyCaret, the evaluation of a trained model and hyper-parameter optimization are performed using the k-fold cross-validation.

For the reproduction of the experiment, a seed is set (Line 45). To disable input confirmation, *silent* is set True (Line 46).

```
40 clas = classification.setup(
41     data=train,
42     target='Survived',
43     ignore_features=['Ticket'],
44     numeric_features=['Age','Fare','Parch','Pclass','SibSp'],
45     session_id = 1,
46     silent=True)
```

In sequence, the pipeline trained all models available in Pycaret with default hyper-parameters and evaluate performance metrics using cross-validation with 10 $folds$. By default, the model with the best accuracy result is returned. To enable a comparison of models that have a long runtime, we set the parameter *turbo* to False.

```
48 best = classification.compare_models(turbo = False)
```

The next step in the AutoML pipeline is the hyper-parameter tuning (lines 50-53). The tuning step is done with 100 iterations to improve the performance of the model, and it returns the model with the best k-fold validated cross-scores. By default, as mentioned before, the classification tasks optimize accuracy.

```
50 best_tuned = classification.tune_model(
51     best,
52     n_iter = 100,
53     choose_better=True)
```

To create a grid with a k-fold validation cross-score from the most recent model with all metrics and means, the $classification.pull$ method call is done (line 57). This grid is stored in PyCaret as a CSV file, thus Learning Orchestra inserts it into a container for later use.

```
57 score = classification.pull()
58 score.to_csv('score', sep='\t', encoding='utf-8')
```

To predict, PyCaret offered the $finalize_model$ method.

```
60 final_gbr = classification.finalize_model(best_tuned)
```

In the last step of a pipeline, the entire transformation done during the pipeline steps, the trained model and the predictions objects are saved as a transferable binary pickle file for later use.

```
62 classification.save_model(final_gbr, 'titanic_pycaret')
```

To finish the pipeline execution using the Learning Orchestra tool, we need to run a Builder service. For that, we need a pipeline code, which is a PyCaret's code inserted in a JSON tag. The data scientist can opt to insert the code URL, this way the Learning Orchestra downloads and runs it. The Builder service needs only the training dataset, the AutoML code and a name to identify such step in Learning Orchestra. The previously stored k-fold validated cross-score grid is retrieved via the $search_builder_register_predictions$ function (Line 80). The entire Titanic code in PyCaret can be found at - [4]

```
68 builder.run_pycaret_async(
69     name=name,
70     parameters={
71         "train": "$train"
72     },
73     code=code)
74 builder.wait(name, 1)
75
76 print(builder.search_execution_content(
77     name=name,
78     pretty_response=True))
79
80 print(builder.search_builder_register_predictions(file_name='score'))
```

### 3.2.2 Autokeras and MNIST

In this section, we illustrate how to use several Learning Orchestra AutoML pipeline steps via Python. The Dataset, Model, Training, Predict, Evaluate and Function Learning Orchestra pipeline steps were used to build the Minist pipeline using the Python Client to simplify even more the REST API access.

[4] Titanic Pycaret code - <https://github.com/learningOrchestra/pythonClient/blob/automl/pipeline/titanicPycaret.py>

We adapted a code[5] that uses the Autokeras tool to construct a multiclass image classification model. We used the Learning Orchestra API to illustrate how to operate the popular Autokeras tool in cloud. 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 the core ideas. For more detailed information, the entire code can be found at Github repository[6].

In the first step of the pipeline, we perform the dataset download [7] (Lines 10 to 16) . The obtained MNIST dataset is divided into four parts: $train_images$, $test_images$, $train_labels$, and $test_labels$ and store them into $NumPy$ objects to be processed by our Autokeras model. For this task, we implemented the $mnist_load_data$ function, which receives the file objects of an image and its label file, and outputs a $NumPy$ array for each of them (Lines 21 to 36). We omit such code in this work because the code is identical with the original Autokeras MNIST.

```
10 dataset_generic = DatasetGeneric(CLUSTER_IP)
11 dataset_generic.insert_dataset_async(
12     dataset_name=f"mnist_dataset_autokeras",
13     url="https://storage.googleapis.com/tensorflow/"
14         "tf-keras-datasets/mnist.npz",
15 )
16 dataset_generic.wait(f"mnist_dataset_autokeras")
17 
18 function_python = FunctionPython(CLUSTER_IP)
19 mnist_load_data = '''
```

A $runFunctionAsync$ method call requires a name for the task, a set of parameters, which can be only the previous pipeline step name (the dataset $mnist_dataset_autokeras$, for instance), and the code to be executed. Since Learning Orchestra API is asynchronous, we performed a synchronization barrier (Line 44).

```
38 function_python.run_function_async(
39     name=f"mnist_load_data",
40     parameters={
41         "mnist_dataset_autokeras": f"$mnist_dataset_autokeras"
42     },
```

5 Minist in Autokeras - <https://autokeras.com/tutorial/image_classification/>

6 Learning Orchestra MNIST for Autokeras <https://github.com/learningOrchestra/pythonClient/blob/automl/pipeline/autokeras_mnist.py>

7 Minist dataset - <https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz>

```
43      code=mnist_load_data)
44  function_python.wait(f"mnist_load_data")
```

A common technique to make neural networks converge faster is to use data normalization in the pre-processing stage, as Learning Orchestra previous work (RIBEIRO, 2021) presented using a TensorFlow+Keras MNIST example. In this work with Autokeras we do not need to worry about normalization issues because AutoKeras performs them automatically. During the model creation all these pre-processing steps are performed (Lines 47 to 55).

The model defined in this example uses a Multi-Class Image Classification, provided by the $autokeras.tasks.image$ module (Line 49). The $overwrite$ parameter (Line 52) was set to True for not reusing files created for similar models.

```
46  model_autokeras = ModelAutoKeras(CLUSTER_IP)
47  model_autokeras.create_model_async(
48      name=f"mnist_model_autokeras",
49      module_path="autokeras.tasks.image",
50      class_name="ImageClassifier",
51      class_parameters={
52          "overwrite": True,
53          "max_trials": 10
54      }
55  )
```

Next, the search for the best neural architecture for the previous created model started (Lines 59 to 70). We defined six epochs for training each model during the search. The fraction of 0.1 of the training data are used to evaluate the loss and model metrics. The Learning Orchestra API requires the previous pipeline step - $mnist_model_autokeras$, the training method name - $fit$, and its parameters, where the number of epochs, the image labels and the images are defined (Lines 64 to 68). Training service is asynchronous, thus a wait condition is performed in Line 71.

```
58  train_autokeras = TrainAutokeras(CLUSTER_IP)
59  train_autokeras.create_training_async(
60      name=f"mnist_model_trained_autokeras",
61      model_name=f"mnist_model_autokeras",
62      parent_name=f"mnist_model_autokeras",
63      method_name="fit",
64      parameters={
65          "x": f"$mnist_load_data.train_images",
66          "y": f"$mnist_load_data.train_labels",
```

```
67          "epochs": 6,
68          "validation_split": 0.1,
69      }
70  )
71  train_autokeras.wait(f"mnist_model_trained_autokeras")
```

Then, the prediction pipeline step is defined (Lines 73 to 82. The predictions are made on the test data and using the best neural network found in the previous training step. The prediction API call is similar to the training API call, where the previous pipeline step, the name for a prediction, the method of Autokeras to be invoked and its parameters must be defined.

```
72
73  predict_autokeras = PredictAutokeras(CLUSTER_IP)
74  predict_autokeras.create_prediction_async(
75      name=f"mnist_model_predicted_autokeras",
76      model_name=f"mnist_model_autokeras",
77      parent_name=f"mnist_model_trained_autokeras",
78      method_name="predict",
79      parameters={
80          "x": f"$mnist_load_data.test_images"
81      }
82  )
83
```

Finally, the model evaluation is performed using the metrics: loss and cross-entropy accuracy. The test images and labels are the unique parameters for Autokeras (Lines 92 to 95).

```
85
86  evaluate_autokeras = EvaluateAutokeras(CLUSTER_IP)
87  evaluate_autokeras.create_evaluate_async(
88      name=f"mnist_model_evaluated_autokeras",
89      model_name=f"mnist_model_autokeras",
90      parent_name=f"mnist_model_trained_autokeras",
91      method_name="evaluate",
92      parameters={
93          "x": f"$mnist_load_data.test_images",
94          "y": f"$mnist_load_data.test_labels"
95      }
96  )
97
```

# 4 Experiments

This chapter presents the cluster setup where we executed the Learning Orchestra system and the performed experiments. The pipeline for Titanic and MNIST models, described in Section 3.2, were evaluated using the same cluster setup and one pipeline at the time.

## 4.1 Setup

The Learning Orchestra AutoML 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 4.1.

| VM | OS | CPU Model | CPU Cores | System RAM | System Storage |
|---|---|---|---|---|---|
| 1 | Debian 10 | On Demand | 2 vCPUs not shared | 8GB | 120GB |
| 2 | Debian 10 | On Demand | 2 vCPUs not shared | 8GB | 120GB |
| 3 | Debian 10 | On Demand | 2 vCPUs not shared | 8GB | 120GB |

Table 4.1 – VMs configuration

### 4.1.1 Container setup for Pycaret and Autokeras

The Learning Orchestra AutoML system offers too deployment alternatives for Pycaret and Autokeras tools. One where the pipeline steps run in a single container deployed in the same VM where the master Docker Swarm is deployed. The second alternative enables Training, Model, Evaluate, Function and Predict AutoML pipeline steps to run on different containers on different VMs, thus in a distribute way - Figure 4.1. The existing steps using TensorFlow or Scikit-learn can also run distributed.

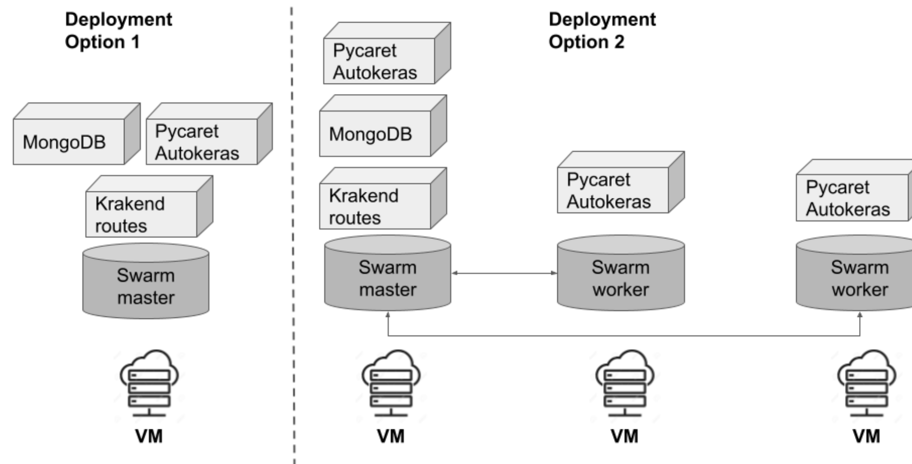


Figure 4.1 – Pycaret and Autokeras container deployment options in Learning Orchestra AutoML

## 4.2 Metrics

We calculate some metrics in our experiments: runtime, memory consumption, accuracy 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 Builder API call or from several API calls to produce a pipeline with atomic steps.

  beginitemize

- Memory consumption: the amount of RAM memory the API service used, including container, VM and any other virtualization overheads.
- 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 metz1978basic. 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.
- 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), lipton2014thresholding.

## 4.3 Titanic Experiment

### 4.3.1 Experiment Overview

The results presented in this section represent Learning Orchestra AutoML running Pycaret, precisely the Titanic model, using the single VM deployment option illustrated in Figure 4.1. In Learning Orchestra previous work (RIBEIRO, 2021), the authors performed Titanic experiments, but using Spark MLlib. We decided to evaluate an AutoML solution against a ML solution where several classifiers are used. These results are important in terms of runtime and ML metrics, since AutoML tools require less data scientist interference.

The Builder API service was used in both Pycaret and Spark Titanic experiments. Dataset, Datatype and Projection API services were used to pre-process the Titanic dataset. The POST microservice runs five classifiers (Logistic Regression (LR), Gradient-Boosted Tree (GB), Decision Tree (DT), Naive Bayes (NB) and Random Forest (RF)) for Spark experiment, identical to the scenarios evaluated in (RIBEIRO, 2021). Pycaret is an AutoML tool, this way it transparently uses many classifiers internally during its models comparisons. The data scientist does not inform the classifiers and their parameters for Pycaret. An important issue is that Pycaret performs the models comparisons in parallel in a single VM and Spark runs all the five classifiers in a distribute way, which is similar to the second deployment option (Figure 4.1) of Learning Orchestra AutoML. The Learning Orchestra is not responsible for the Spark ML tasks distribution.

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

### 4.3.2 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 70% for training, and 30% 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.

Pycaret by default uses k-folds cross-validation in evaluation steps, precisely it uses the 10 $folds$.

### 4.3.3 Runtime Results

It was measured 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 4.2 illustrates the Pycaret runtimes using the dataset with no transformations and second run where Pycaret uses a pre-processed dataset. The AutoML process is attenuated in terms of processing when the data scientist informs features and that is the main reason for a better runtime of the optimized Pycaret Titanic model in Table 4.2.

| Execution | Pycaret | Pycaret Opt |
|---|---|---|
| 1 | 394.68 s | 354.84 s |
| 2 | 395.71 s | 352.72 s |
| 3 | 393.39 s | 353.79 s |
| 4 | 392.96 s | 354.16 s |
| 5 | 392.90 s | 353.19 s |
| Standard deviation | 1.097 s | 0.739 s |
| Average | 393.92 s | 353.74 s |

Table 4.2 – Run Time Comparison: Pycaret full automatic and optimized

Unfortunately, when compared with the distributed Spark solution, the Pycaret model search took more time (see 4.3). In summary, Spark can use all six cores available in the cluster and Pycaret only two cores of a VM, which are insufficient for all combinations of classifiers and parameter tuning tasks running concurrently. It is important to consider that Pycaret has a vast catalog of algorithms to search and not only the five used by Spark, thus, some of them can take more time than others.

| Execution | Spark Application |
|---|---|
| 1 | 162.40 s |
| 2 | 88.25 s |
| 3 | 78.92 s |
| 4 | 77.82 s |
| 5 | 78.26 s |
| Standard deviation | 29.99 s |
| Average | 97.13 s |

Table 4.3 – Run Time: Spark manual classifiers

### 4.3.4 ML Metrics Results

Table 4.4 illustrates the validation result for the five classifiers in terms of accuracy and F1-score. We have just repeated the Spark experiment done in (RIBEIRO, 2021).

The model using GB classifier 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 NB classifier, with $63.3\%$ for the accuracy and $62.5\%$ for the F1-score.

| Classifier | accuracy rate | F1 score |
|---|---|---|
| LR | 0.808 | 0.807 |
| DT | 0.796 | 0.795 |
| RF | 0.802 | 0.801 |
| GB | 0.825 | 0.824 |
| NB | 0.633 | 0.625 |

Table 4.4 – Titanic model using Spark MLlib

We have executed several Pycaret runs over the container cluster and the reasons are: i) we have a simple way to inform features, like attribute type, to improve the model metrics; and ii) we have some data transformations performed by the previous Learning Orchestra work (RIBEIRO, 2021), which can be easily incorporated to the Pycaret pipeline to improve the model. Table 4.4 shows the best classifier algorithm discovered by Pycaret for accuracy and F1-score metrics using transformations and features. On Table 4.6, we present the best algorithm discovered by Pycaret using the Titanic dataset without modifications, which means realy automated.

The Pycaret searched a model with accuracy of $0.855$, thus better than the Spark experiment $0.825$. Even the Pycaret configured without interventions produced a better accuracy than Spark (see Table 4.6).

| Classifier | accuracy rate | F1 score |
|---|---|---|
| GB | 0.855 | 0.783 |

Table 4.5 – Prediction metrics - Best Titanic Pycaret Pipeline using optimizations in pre-processing

| Classifier | accuracy rate | F1 score |
|---|---|---|
| GB | 0.831 | 0.744 |

Table 4.6 – Prediction metrics - Best Titanic Pycaret pipeline full automated

## 4.4 MNIST Experiment

### 4.4.1 Experiment Overview

The results presented in this section represent Learning Orchestra AutoML running Autokeras, precisely the MNIST model, using the single VM deployment option illustrated in Figure 4.1. In Learning Orchestra's previous work (RIBEIRO, 2021), the authors performed MNIST experiments but using Keras and TensorFlow. This way, we can test the ML results of an AutoML solution against a traditional ML solution using the MNIST dataset, as we did in the Titanic experiment.

Besides the comparative results using the previous Learning Orchestra ML version, we can evaluate the impact of Learning Orchestra RESTful AutoML API and for that, we tested Autokeras without Learning Orchestra integration against a version with Learning Orchestra. In MNIST experiments using Learning Orchestra, the pipeline used several API services and not only the Builder. Thus, in this section, we illustrate the atomic pipeline steps benefits when the pipeline steps are stored into volumes and when it is possible to re-run steps of the pipeline and not only the entire pipeline, as Autokeras does when executed alone. In these MNIST experiments the following API services were used: Dataset, Function, Model, Training, Predict and Evaluate.

The MNIST pipeline steps were coded using the Python Client to simplify a bit more the Learning Orchestra usage.

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

### 4.4.2 Dataset

This challenge is part of the computer vision area. Its objective is to correctly identify the handwritten numeric digits in a data set. This dataset is known as: Modified National Institute of Standards and Technology (MNIST) - [1], this is the premier dataset for computer vision beginners. MNIST contains 60,000 images with a resolution of 28x28 pixels.

### 4.4.3 Runtime Results

Tables 4.7 and 4.8 illustrate the runtimes of MNIST using Autokeras alone and integrated with Learning Orchestra, respectively. As we can see, the impact of Learning Orchestra REST services is irrelevant. In these results the automated store of pipeline steps is performed by Learning Orchestra.

| Execution | Autokeras MNIST runtime |
|---|---|
| 1 | 1475.96 s |
| 2 | 1474.32 s |
| 3 | 1462.12 s |
| 4 | 1432.78 s |
| 5 | 1430.04 s |
| Standard deviation | 19.89 s |
| Average | 1455,04 s |

Table 4.7 – Autokeras MNIST runtimes

| Execution | Learning Orchestra Autokeras MNIST runtime |
|---|---|
| 1 | 1498.50 s |
| 2 | 1474.32 s |
| 3 | 1441.03 s |
| 4 | 1436.77 s |
| 5 | 1433.44 s |
| Standard deviation | 25.46 s |
| Average | 1456,81 s |

Table 4.8 – Learning Orchestra Autokeras MNIST runtimes

[1] MNIST - <http://yann.lecun.com/exdb/mnist/>

### 4.4.4 RAM Memory Consumption

We have collected the memory consumption before the experiments and after them, the same methodology applied in (RIBEIRO, 2021). In order to have a basis for comparison, the MNIST pipeline was executed in a container inside one of the VM's of the cluster, using only the AutoKeras tool. It is observed in Table 4.9 that, on average, 537.2 MB of RAM is used to find an optimized neural network without Learning Orchestra costs.

| Autokeras MNIST RAM Consumption | |
|---|---|
| Execution | RAM Running |
| 1 | 541 MB |
| 2 | 545 MB |
| 3 | 528 MB |
| 4 | 529 MB |
| 5 | 543 MB |
| Standard deviation | 7.22 MB |
| Average | 537.2 MB |

Table 4.9 – Autokeras MNIST RAM Consumption

Table 4.10 – Autokeras and Learning Orchestra MNIST RAM consumption

| Average Consumption | MongoDB | Function | Model | Training/Evaluate |
|---|---|---|---|---|
| RAM idle | 127.8 MB | 176,6 MB | 217.3 MB | 168.8 MB |
| RAM running | 152.5 MB | 208 MB | 270 MB | 590,35 MB |

In terms of memory consumption when using Autokeras through Learning Orchestra, we see that the system needs more memory to maintain the distributed API services. MongoDB and Krakend are deployed in separated containers and their memory usage are introduced by Learning Orchestra. In terms of pipeline execution, the memory consumption is similar (around 500 MB) and the training step is the more costly one, which is an expected result.

### 4.4.5 ML Metrics Results

On these experiments the training pipeline step used six epochs and 20% data split percentage for validation. We have compared the MNIST results from the work (RIBEIRO, 2021) because it represents a traditional ML pipeline developed with Keras + TensorFlow, which means without automated discoveries.

The automated version produced a cross-entropy of 0.0358 (Table 4.11) and the traditional pipeline construction 0.069 (Table 4.12), indicating a better result for Autokeras against Keras + TensorFlow. In terms of accuracy, the automated version produced 0.9882 (Table 4.11) and the Keras version 0.979 (Table 4.12). Again, Autokeras produced a better result.

| Execution | Cross-entropy loss | Accuracy |
|---|---|---|
| 1 | 0.0354 | 0.9884 |
| 2 | 0.0360 | 0.9883 |
| 3 | 0.0365 | 0.9882 |
| 4 | 0.0359 | 0.9876 |
| 5 | 0.0354 | 0.9884 |
| Standard deviation | 0.0004 | 0.0002 |
| Average | 0.0358 | 0.9882 |

Table 4.11 – ML metrics from Autokeras MNIST experiments

| Execution | Cross-entropy loss | Accuracy |
|---|---|---|
| Average | 0.069 | 0.979 |

Table 4.12 – Keras+TensorFlow MNIST experiment (from (RIBEIRO, 2021))

As mentioned before, this work is not about ML models quality. Still, these kinds of experiments demonstrated that Learning Orchestra does not interfere in Pycaret and Autokeras results, and it enables both ML and AutoML pipeline development using the same API syntax.

# 5 Conclusion

In this work, we present important contributions for the interoperability of AutoML tools. We have presented a new version for the Learning Orchestra, a system to operate, using a RESTful API, several common services of ML and AutoML areas. The new version enables operate PyCaret and Autokeras AutoML tools transparently using a REST API or a Python client.

The current Learning Orchestra AutoML system guarantees the atomic pipeline steps, which means the capacity to transfer serializable objects through the communication network or the capacity to store them in volumes of cloud environments. This way, Learning Orchestra continues to be one of the only ML solution used to build pipelines, including the AutoML ones, where their steps are atomic and serializable.

Another important aspect of an integration with Learning Orchestra is that it distributes the pipeline steps, enabling a transparent deployment over many VMs and, consequently, many containers. We have evaluated the presented solution in different scenarios using different models and AutoML tools. Basically, the REST API does not interfere in the runtimes of Pycaret and Autokeras services when they are intercepted by Learning Orchestra. The API syntax was maintained to provide AutoML services. Finally, the AutoML pipelines developed for Titanic and MNIST models proved to produce better results in terms of accuracy and cross-entropy when compared with a traditional way to build ML pipelines, where the data scientist provides several mandatory information before training and tuning a model.

This work represents a lacuna reduction in terms of operability and heterogeneity, but it still requires improvements. The first one is the experiments using the Learning Orchestra distributed deployment. Titanic and MINIST must be tested using such deployment option illustrated in Figure 4.1. Other AutoML tools, like Katib and Auto-pytorch, should be operated via Learning Orchestra without API syntax changes. Furthermore, an alternative to implement distributed training and tuning steps should be available in Learning Orchestra catalog of tools. Nowadays the pipeline can be distribute, but the training or tuning steps run, in the best cases, only in parallel in a single VM. Horovod (SERGEEV; BALSO, 2018) and TensorFlow distributed, for instance, should be part of the Learning Orchestra tools catalog.

Identifying underperformance training steps require constant attention and deep domain expertise. As state-of-the-art models grow in size and complexity, debugging becomes increasingly difficult and unfortunately few workflow design tools support monitoring or debugging services. We have found SageMaker (JOSHI, 2020), from Amazon, TFX (BAYLOR et al., 2017) from the TensorFlow project, and Kubeflow (BISONG, 2019), from the Kubernetes team, as alternatives with services for debug/monitor demands. Only SageMaker has both debug and monitor, where the first gives feedbacks about training jobs, and the last also stops underperformance jobs. We

understand that a monitoring and debugging services should be provided in the current REST API of Learning Orchestra. The actual Observer service can be remodeled to bring part of this feature. An integration with, for instance, TensorBoard (LUUS; KHAN; AKHALWAYA, 2019) can be an option to bring these debug and monitor benefits.

# Bibliography


API, O. *Open API Specification*. 2021. 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. Tfx: A tensorflow-based production-scale machine learning platform. In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. [S.l.: s.n.], 2017. p. 1387–1395.

BISONG, E. Kubeflow and kubeflow pipelines. In: *Building Machine Learning and Deep Learning Models on Google Cloud Platform*. [S.l.]: Springer, 2019. p. 671–685.

CHODOROW, K. *MongoDB: the definitive guide: powerful and scalable data storage*. [S.l.]: " O'Reilly Media, Inc.", 2013.

DAS, P.; IVKIN, N.; BANSAL, T.; ROUESNEL, L.; GAUTIER, P.; KARNIN, Z.; DIRAC, L.; RAMAKRISHNAN, L.; PERUNICIC, A.; SHCHERBATYI, I. et al. Amazon sagemaker autopilot: a white box automl solution at scale. In: *Proceedings of the Fourth International Workshop on Data Management for End-to-End Machine Learning*. [S.l.: s.n.], 2020. p. 1–7.

DIELEMAN, S.; SCHLüTER, J.; RAFFEL, C.; OLSON, E.; SøNDERBY, S. K.; NOURI, D. et al. *Lasagne: First release.* 2015. Disponível em: <http://dx.doi.org/10.5281/zenodo.27878>.

DONG, G.; LIU, H. *Feature engineering for machine learning and data analytics*. [S.l.]: CRC Press, 2018.

ELSHAWI, R.; MAHER, M.; SAKR, S. Automated machine learning: State-of-the-art and open challenges. *arXiv preprint arXiv:1906.02287*, 2019.

FAYYAD, U.; PIATETSKY-SHAPIRO, G.; SMYTH, P. The kdd process for extracting useful knowledge from volumes of data. *Communications of the ACM*, ACM New York, NY, USA, v. 39, n. 11, p. 27–34, 1996.

FEURER, M.; KLEIN, A.; EGGENSPERGER, K.; SPRINGENBERG, J. T.; BLUM, M.; HUTTER, F. Auto-sklearn: efficient and robust automated machine learning. In: *Automated Machine Learning*. [S.l.]: Springer, Cham, 2019. p. 113–134.

GEORGE, J.; GAO, C.; LIU, R.; LIU, H. G.; TANG, Y.; PYDIPATY, R.; SAHA, A. K. *A Scalable and Cloud-Native Hyperparameter Tuning System*. 2020.

GOLDBERG, R. P. Survey of virtual machine research. *Computer*, IEEE, v. 7, n. 6, p. 34–45, 1974.

H2O.AI. *H2O: Scalable Machine Learning Platform*. [S.l.], 2020. Version 3.30.0.6. Disponível em: <https://github.com/h2oai/h2o-3>.

HALL, M.; FRANK, E.; HOLMES, G.; PFAHRINGER, B.; REUTEMANN, P.; WITTEN, I. H. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, ACM New York, NY, USA, v. 11, n. 1, p. 10–18, 2009.

HAN, J.; PEI, J.; TONG, H. *Data mining: concepts and techniques*. [S.l.]: Morgan kaufmann, 2022.

HARDT, M.; CHEN, X.; CHENG, X.; DONINI, M.; GELMAN, J.; GOLLAPROLU, S.; HE, J.; LARROY, P.; LIU, X.; MCCARTHY, N. et al. Amazon sagemaker clarify: Machine learning bias detection and explainability in the cloud. *arXiv preprint arXiv:2109.03285*, 2021.

HE, X.; ZHAO, K.; CHU, X. Automl: A survey of the state-of-the-art. *Knowledge-Based Systems*, Elsevier, v. 212, p. 106622, 2021.

HUB, S. *Swagger Hub*. 2021. Available at: <https://swagger.io/tools/swaggerhub/>.

INSOMNIA. *Insomnia Collaborative API Design Editor*. 2021. Available at: <https://insomnia.rest/>.

JIN, H.; SONG, Q.; HU, X. Auto-keras: An efficient neural architecture search system. In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. [S.l.: s.n.], 2019. p. 1946–1956.

JOSHI, A. V. Amazon's machine learning toolkit: Sagemaker. In: *Machine Learning and Artificial Intelligence*. [S.l.]: Springer, 2020. p. 233–243.

KOTTHOFF, L.; THORNTON, C.; HOOS, H.; HUTTER, F.; LEYTON-BROWN, K. Auto-weka 2.0: Automatic model selection and hyperparameter optimization in weka. *Journal of Machine Learning Research*, v. 18, p. 1–5, 03 2017.

KRAKEND. *KrakenD*. 2021. Available at: <https://www.krakend.io/>.

LEDELL, E.; POIRIER, S. H2o automl: Scalable automatic machine learning. In: *Proceedings of the AutoML Workshop at ICML*. [S.l.: s.n.], 2020. v. 2020.

LUUS, F.; KHAN, N.; AKHALWAYA, I. Active learning with tensorboard projector. *arXiv preprint arXiv:1901.00675*, 2019.

MELL, P.; GRANCE, T. et al. The nist definition of cloud computing. Computer Security Division, Information Technology Laboratory, National …, 2011.

MENDOZA, H.; KLEIN, A.; FEURER, M.; SPRINGENBERG, J. T.; HUTTER, F. Towards automatically-tuned neural networks. In: PMLR. *Workshop on Automatic Machine Learning*. [S.l.], 2016. p. 58–65.

MENDOZA, H.; KLEIN, A.; FEURER, M.; SPRINGENBERG, J. T.; URBAN, M.; BURKART, M.; DIPPEL, M.; LINDAUER, M.; HUTTER, F. Towards automatically-tuned deep neural networks. In: *Automated Machine Learning*. [S.l.]: Springer, Cham, 2019. p. 135–149.

MENG, X.; BRADLEY, J.; YAVUZ, B.; SPARKS, E.; VENKATARAMAN, S.; LIU, D.; FREEMAN, J.; TSAI, D.; AMDE, M.; OWEN, S. et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, JMLR. org, v. 17, n. 1, p. 1235–1241, 2016.

MERKEL, D. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, v. 2014, n. 239, p. 2, 2014.

MOLNAR, C.; KÖNIG, G.; HERBINGER, J.; FREIESLEBEN, T.; DANDL, S.; SCHOLBECK, C. A.; CASALICCHIO, G.; GROSSE-WENTRUP, M.; BISCHL, B. General pitfalls of model-agnostic interpretation methods for machine learning models. In: SPRINGER. *International Workshop on Extending Explainable AI Beyond Deep Models and Classifiers*. [S.l.], 2022. p. 39–68.

MOOLAYIL, J.; MOOLAYIL, J.; JOHN, S. *Learn Keras for deep neural networks*. [S.l.]: Springer, 2019.

MORITZ, P.; NISHIHARA, R.; WANG, S.; TUMANOV, A.; LIAW, R.; LIANG, E.; ELIBOL, M.; YANG, Z.; PAUL, W.; JORDAN, M. I. et al. Ray: A distributed framework for emerging {AI} applications. In: *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. [S.l.: s.n.], 2018. p. 561–577.

NADAREISHVILI, I.; MITRA, R.; MCLARTY, M.; AMUNDSEN, M. *Microservice architecture: aligning principles, practices, and culture*. [S.l.]: " O'Reilly Media, Inc.", 2016.

OLSON, R. S.; MOORE, J. H. Tpot: A tree-based pipeline optimization tool for automating machine learning. In: PMLR. *Workshop on automatic machine learning*. [S.l.], 2016. p. 66–74.

PASZKE, A.; GROSS, S.; MASSA, F.; LERER, A.; BRADBURY, J.; CHANAN, G.; KILLEEN, T.; LIN, Z.; GIMELSHEIN, N.; ANTIGA, L.; DESMAISON, A.; KOPF, A.; YANG, E.; DEVITO, Z.; RAISON, M.; TEJANI, A.; CHILAMKURTHY, S.; STEINER, B.; FANG, L.; BAI, J.; CHINTALA, S. Pytorch: An imperative style, high-performance deep learning library. In: WALLACH, H.; LAROCHELLE, H.; BEYGELZIMER, A.; ALCHé-BUC, F. d'; FOX, E.; GARNETT, R. (Ed.). *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019. p. 8024–8035. Disponível em: <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>.

PEDREGOSA, F.; VAROQUAUX, G.; GRAMFORT, A.; MICHEL, V.; THIRION, B.; GRISEL, O.; BLONDEL, M.; PRETTENHOFER, P.; WEISS, R.; DUBOURG, V. et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, JMLR. org, v. 12, p. 2825–2830, 2011.

PERRONE, V.; SHEN, H.; ZOLIC, A.; SHCHERBATYI, I.; AHMED, A.; BANSAL, T.; DONINI, M.; WINKELMOLEN, F.; JENATTON, R.; FADDOUL, J. B. et al. Amazon sagemaker automatic model tuning: Scalable black-box optimization. *arXiv preprint arXiv:2012.08489*, 2020.

REN, P.; XIAO, Y.; CHANG, X.; HUANG, P.-Y.; LI, Z.; CHEN, X.; WANG, X. A comprehensive survey of neural architecture search: Challenges and solutions. *ACM Computing Surveys (CSUR)*, ACM New York, NY, USA, v. 54, n. 4, p. 1–34, 2021.

RIBEIRO, G. d. O. Learning orchestra: Building machine learning workflows on scalable containers. 2021.

SERGEEV, A.; BALSO, M. D. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, maio 2016. Disponível em: <http://arxiv.org/abs/1605.02688>.

THORNTON, C.; HUTTER, F.; HOOS, H. H.; LEYTON-BROWN, K. Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. In: *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. [S.l.: s.n.], 2013. p. 847–855.

YAO, Q.; WANG, M.; CHEN, Y.; DAI, W.; LI, Y.-F.; TU, W.-W.; YANG, Q.; YU, Y. Taking human out of learning applications: A survey on automated machine learning. *arXiv preprint arXiv:1810.13306*, 2018.

ZHOU, Z.-H. *Machine learning*. [S.l.]: Springer Nature, 2021.

ZIMMER, L.; LINDAUER, M.; HUTTER, F. Auto-pytorch: Multi-fidelity metalearning for efficient and robust autodl. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, IEEE, 2021.

ZÖLLER, M.-A.; HUBER, M. F. Benchmark and survey of automated machine learning frameworks. *Journal of Artificial Intelligence Research*, v. 70, p. 409–472, 2021.

# Appendix

# APPENDIX A – The entire content of Titanic in Pycaret

```
from learning_orchestra_client.builder.builder import BuilderPycaret
from learning_orchestra_client.dataset.csv import DatasetCsv
from learning_orchestra_client.transform.data_type import TransformDataType
import time

start = time.time()

CLUSTER_IP = "http://34.151.210.120"

dataset_csv = DatasetCsv(CLUSTER_IP)

dataset_csv.delete_dataset("train")
dataset_csv.insert_dataset_async(
    url="https://raw.githubusercontent.com/JonatasMiguel/"
        "PycaretTitanic/main/train.csv",
    dataset_name="train",
)
dataset_csv.wait(dataset_name="train",
                 timeout=1)

transform_data_type = TransformDataType(CLUSTER_IP)
type_fields = {
    "Age": "number",
    "Fare": "number",
    "Parch": "number",
    "Pclass": "number",
    "SibSp": "number"
}
transform_data_type.update_dataset_type_async(
    dataset_name=f"train",
    types=type_fields)

transform_data_type.wait(dataset_name=f"train",
```

```
34                              timeout=1)
35
36 name = f'titanicPycaret'
37 code = """
38 from pycaret import classification
39
40 clas = classification.setup(
41     data=train,
42     target='Survived',
43     ignore_features=['Ticket'],
44     numeric_features=['Age','Fare','Parch','Pclass','SibSp'],
45     session_id = 1,
46     silent=True)
47
48 best = classification.compare_models(turbo = False)
49
50 best_tuned = classification.tune_model(
51     best,
52     n_iter = 100,
53     choose_better=True)
54
55 best_tuned = classification.create_model(best_tuned)
56
57 score = classification.pull()
58 score.to_csv('score', sep='\t', encoding='utf-8')
59
60 final_gbr = classification.finalize_model(best_tuned)
61
62 classification.save_model(final_gbr, 'titanic_pycaret')
63
64 response = None
65 """
66 builder = BuilderPycaret(CLUSTER_IP)
67
68 builder.run_pycaret_async(
69     name=name,
70     parameters={
71         "train": "$train"
72     },
```

```
        code=code)
builder.wait(name, 1)

print(builder.search_execution_content(
        name=name,
        pretty_response=True))

print(builder.search_builder_register_predictions(file_name='score'))
```

# APPENDIX B – The entire content of Mnist in Autokeras

```python
from learning_orchestra_client.function.python import FunctionPython
from learning_orchestra_client.dataset.generic import DatasetGeneric
from learning_orchestra_client.model.autokeras import ModelAutoKeras
from learning_orchestra_client.train.autokeras import TrainAutokeras
from learning_orchestra_client.predict.autokeras import PredictAutokeras
from learning_orchestra_client.evaluate.autokeras import EvaluateAutokeras

CLUSTER_IP = "http://34.125.22.143"

dataset_generic = DatasetGeneric(CLUSTER_IP)
dataset_generic.insert_dataset_async(
    dataset_name=f"mnist_dataset_autokeras",
    url="https://storage.googleapis.com/tensorflow/"
        "tf-keras-datasets/mnist.npz",
)
dataset_generic.wait(f"mnist_dataset_autokeras")

function_python = FunctionPython(CLUSTER_IP)
mnist_load_data = '''

def load_data(path):
    import numpy as np
    with np.load(path) as f:
        x_train, y_train = f['x_train'], f['y_train']
        x_test, y_test = f['x_test'], f['y_test']
        return (x_train, y_train), (x_test, y_test)

(x_train, y_train), (x_test, y_test) = load_data(mnist_dataset_autokeras)

response = {
    "test_images": x_test,
    "test_labels": y_test,
    "train_images": x_train,
```

```
    "train_labels": y_train
}
'''

function_python.run_function_async(
    name=f"mnist_load_data",
    parameters={
        "mnist_dataset_autokeras": f"$mnist_dataset_autokeras"
    },
    code=mnist_load_data)
function_python.wait(f"mnist_load_data")

model_autokeras = ModelAutoKeras(CLUSTER_IP)
model_autokeras.create_model_async(
    name=f"mnist_model_autokeras",
    module_path="autokeras.tasks.image",
    class_name="ImageClassifier",
    class_parameters={
        "overwrite": True,
        "max_trials": 10
    }
)
model_autokeras.wait(f"mnist_model_autokeras")

train_autokeras = TrainAutokeras(CLUSTER_IP)
train_autokeras.create_training_async(
    name=f"mnist_model_trained_autokeras",
    model_name=f"mnist_model_autokeras",
    parent_name=f"mnist_model_autokeras",
    method_name="fit",
    parameters={
        "x": f"$mnist_load_data.train_images",
        "y": f"$mnist_load_data.train_labels",
        "epochs": 6,
        "validation_split": 0.1,
    }
)
train_autokeras.wait(f"mnist_model_trained_autokeras")

```

```
predict_autokeras = PredictAutokeras(CLUSTER_IP)
predict_autokeras.create_prediction_async(
    name=f"mnist_model_predicted_autokeras",
    model_name=f"mnist_model_autokeras",
    parent_name=f"mnist_model_trained_autokeras",
    method_name="predict",
    parameters={
        "x": f"$mnist_load_data.test_images"
    }
)

predict_autokeras.wait(f"mnist_model_predicted_autokeras")

evaluate_autokeras = EvaluateAutokeras(CLUSTER_IP)
evaluate_autokeras.create_evaluate_async(
    name=f"mnist_model_evaluated_autokeras",
    model_name=f"mnist_model_autokeras",
    parent_name=f"mnist_model_trained_autokeras",
    method_name="evaluate",
    parameters={
        "x": f"$mnist_load_data.test_images",
        "y": f"$mnist_load_data.test_labels"
    }
)

evaluate_autokeras.wait(f"mnist_model_evaluated_autokeras")

show_mnist_predict = '''
print(mnist_predicted)
response = None
'''
function_python.run_function_async(
    name=f"mnist_model_predicted_print",
    parameters={
        "mnist_predicted": f"$mnist_model_predicted_autokeras"
    },
    code=show_mnist_predict
)

```

```
show_mnist_evaluate = '''
print(mnist_evaluated)
response = None
'''
function_python.run_function_async(
    name=f"mnist_model_evaluated_print",
    parameters={
        "mnist_evaluated": f"$mnist_model_evaluated_autokeras"
    },
    code=show_mnist_evaluate
)

function_python.wait(f"mnist_model_evaluated_print")
function_python.wait(f"mnist_model_predicted_print")

print(function_python.search_execution_content(
    name=f"mnist_model_predicted_print",
    limit=1,
    skip=1,
    pretty_response=True))

print(function_python.search_execution_content(
    name=f"mnist_model_evaluated_print",
    limit=1,
    skip=1,
    pretty_response=True))
```