POLITECNICO DI TORINO

DEPARTMENT OF CONTROL AND COMPUTER ENGINEERING

Master Of Science in Data Science and Engineering

Master Thesis

Streamline machine learning projects to production using cutting-edge MLOps best practices on AWS

Supervisors
Prof. Daniele Apiletti
Alessandro Bianchi
Alex Degiovanni

Candidate
Alessandro Palladini

ACADEMIC YEAR 2020/2021

Abstract

In the early years of its life, the use of machine learning was limited to academic research, where it had the opportunity to evolve. In recent years the transition to the industrial world has taken place, and nowadays in any field especially in the largest companies machine learning is assuming an increasingly central role. This process comes with challenges: every technological evolution, which in some cases can also be disruptive, involves work and organizational adaptations. Moreover, gaps and differences demark the distance among how academic research and real-world companies work.

The objective of machine learning operations (MLOps) is to shorten but also make more reliable the major phases that characterise the deployment and maintenance of machine learning model in production. It inherits from DevOps some of its key practices like continuous integration, continuous deployment and delivery while introducing practices unique to machine learning systems, like continuous training and data versioning.

In this work we built a MLOps framework and practices using AWS services, that can be used in various settings, and by different teams from different backgrounds, which broadens the horizons of MLOps, including in the framework new innovative strategies, such as monitoring jobs and the usage of feature store.

The MLOps pipeline is the main pillar of our work: we were able to build an highly modular and reliable pipeline that automates the entire machine learning life-cycle, bridging the gap between development and operation and enabling better collaboration and communication between different teams who are operating in the system.

The main difficulty and the main focus in the first phase of our work was to develop the pipeline, once it was ready we could focus on the pure machine learning experimentation phase of the different projects. This allow to deliver a model to production much faster from its conception. The two projects we present are in different setting: a classic binary classification and a cutting edge time series forecasting, which is not present in literature yet. In both the projects we show how the continuously train

and deliver of models in production work: the monitoring activities check at regular intervals whether drift between the statistics of the training dataset and serving data are present or model performances decay in terms of predefined metrics.

Acknowledgement

With this graduation, I momentarily finish my studies, that last 18 years. These years have been incredibly, and I finally find myself ready to do the big step, and be adult under any perspective. The years of my Master's degree has been characterized by historical facts, unfortunately, in a negative way. I have had the luck to do what I have always wished to do, and I arrive at my final day as a student with much gratitude. Gratitude to my family, that allowed me to achieve this important goal. In particular, I mention my grandmother, Anna, without her I would not be what I am, and I would probably never lived in Turin. I am thankful to my girlfriend, Erica, with whom I shared the last three years, without her everything would be different, worst, and I can not be happier to share this finish line with her. I thank all my friends, that have made these years of study extremely funny.

This work has been developed in collaboration with Data Reply IT. I entered this company as a student, and after one year I learnt a lot, and the objective of this thesis has been achieved with their needful collaboration. In particular, I thank a lot Alessandro and Alex, that each Thursday for almost one year supported me in the development of this thesis. I also would like to thank Michele and Emanuele, that gave me this opportunity. Last but not least I thank my supervisor Professor Daniele Apiletti, for its advices for my career.

Contents

Chapter 1

Introduction

Traditionally, machine learning (ML) has been approached from a perspective of individual scientific experiments which are predominantly carried out in isolation by data scientists. However, machine learning has evolved from purely an area of academic research to an applied field, and it has become part of real-world solutions and critical to business. This entails a shift of the perspective while approaching a machine learning project, to let the process be as easily accessible, reproducible and collaborative as possible. As shown in the Figure 1.1, only a small fraction of a real-world ML system is composed of the ML code. The multiple surrounding elements are required as much as the code, and generally, they set the most difficult challenges to tackle. To develop and operate complex systems like these, that are faced daily in a real-world ML system, we present this work, in which we apply some DevOps principles, while introducing some practices unique to ML systems: streamline ML projects using cutting-edge machine Learning Operations (MLOps) practices on AWS.

The aim of this thesis is to establish a set of practices that put tools, pipelines, and processes to build fast time-to-value machine learning development projects, but, at the same time, to make more reliable and modular the major phases that characterise the deployment and maintenance of machine learning model in production, using the chosen environment (AWS).

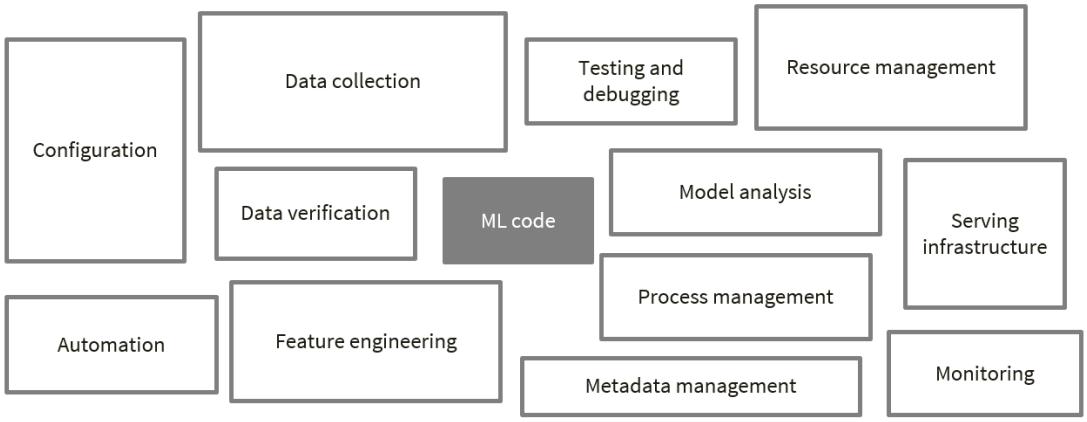

Figure 1.1: Only a small fraction of real-world ML systems is occupied by the ML code. The required surrounding infrastructure is vast and complex. Adapted from [1]

1.1 Chapter guide

1. **Chapter 1**. A general introduction.
2. **Chapter 2**. Contains the general background. It present the context of how machine learning works in production. It highlights the major challenges in traditional ML systems, hence the main reasons of this work.
3. **Chapter 3**. The state-of-art relevant to this thesis is presented in this Chapter. We discuss the relevant tools for this thesis.
4. **Chapter 4**. We introduce the key concepts that we implement in our framework, along with the AWS services used.
5. **Chapter 5**. In this Chapter we concretely describe the methodology that constitute the pillar of this thesis.
6. **Chapter 6**. This Chapter show the details of the implementation of two intrinsically different projects, that regardless share the the methodology presented in this work.
7. **Chapter 7**. The final conclusions and the possible future developments.

Chapter 2

Background

In this Chapter we present the background of our work, presenting the main reasons of why MLOps is so important, and concluding by showing the objectives of MLOps, and consequently of this thesis.

2.1 Differences between machine learning and traditional software

The first time "machine learning" came out, was back in 1952. Since then, the history of ML is deep and variegated, until its fundamental role in basically the majority of sectors in the contemporary economy. Traditional software and ML share lot. But the difference between them is basal: the former aims to answer a problem using a predefined set of rules or logic. In contrast, machine learning seeks to construct a model or logic for the problem by analyzing its input data and answers[15]. The additional dimension that differentiates ML and classical software is contained in the previous sentence, the data. It might sound self explanatory and simple but the addition of data, usually big data, brings entirely new challenges to the development process [11].

The nature of ML, because of the data, becomes multidisciplinary. To understand it is enough to look at some roles involved in real world ML projects: data Scientist,

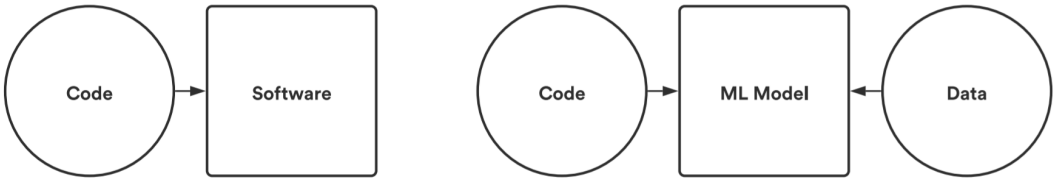

Figure 2.1: Data is a new dimension in ML systems w.r.t. traditional software.

data Engineer, machine learning engineer, DevOps engineer, business owner, manager are just some of the role that must collaborate in order to achieve their mandate, even if their origins are different. It complicates the practice.

While normal software can generally be developed locally with almost instant feedback on how a code change affects the end-result, in ML, to see the effects of a code change requires retraining a model.

Consider for example working with models that are trained with large datasets poses huge infrastructure challenges as either it needs a lot of waiting time to do it locally or cloud computing resources are needed. Let make another example to visualize better this distinction: a developer of classic software can just save a code change and see immediate results; while a data scientist save similar code change but then to see result he/she must deploy the code, maybe spinning up a cluster of GPUs, maybe transfer data and retraining a model before seeing any results.

Moreover, the introduction of data carries with it other differences in the development process: data itself is another aspect that can change. Indeed, in classical software development, a version of the code produces a version of the software, while in machine learning, a version of the code and a version of the data together produce a version of the ML model.

The fact that data can change increases the complexity drastically. To reproduce a model, or at least its result, code and data must be reproducible. This lead to the statement that the components (both data and code) must be versioned to ensure reproducibility. In this work we will come back to this concept multiple times, since has been crucial in our development. To get an idea of this concept, it is enough to think about a machine learning system (which have different and multiple steps

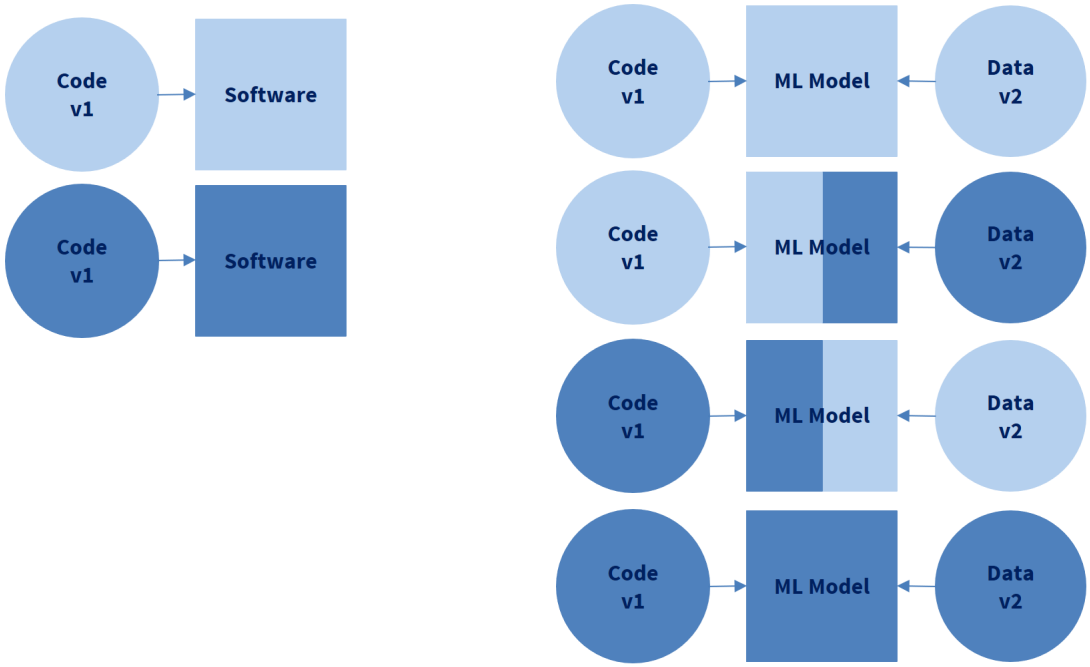

Figure 2.2: In classical software development, a version of the code produces a version of the software, while in machine learning, a version of the code and a version of the data together produce a version of the ML model.

for data preparation, augmentation and generation) and how the matter of versioning both data and code becomes increasingly complex.
We saw of data actively affect makes a difference between ML and traditional software. Here we sum up the aspect that characterizes ML systems and that we must consider in our work:

- Team heterogeneity: in an ML project, the team includes data scientists or ML researchers, who focus on their job like exploratory data analysis, model development, and experimentation. These people might not be able to deliver their product to production, and other members are needed, able to build production-class services.

- Project development: ML is experimental in nature. Experimentation is the key to find what best works for the task. Trying different features, algorithms, modeling techniques, and parameter configurations tracking them to consider what worked what did not is a challenge.

We will dig deep into all these concepts later in the work, starting from the next section.

2.2 Major challenges in real-world traditional machine learning systems

We carried out a study on the state and difficulties of machine learning systems in production [2], [3], [4], [5]. The goal is to find out challenges and difficulties in building and maintaining ML systems in a production environment, mainly studying ML traditional systems. We want to remind that ML in real-world system is completely another thing with respect to what it is in a local environment. Indeed, whether doing academic research or anything else in local means essentially working in a straight forward single environment, usually with a single programming language and a single software. ML in production focuses more on engineering. Not only focus on implementing ML models but also build the whole ML-related infrastructure within an industry.
As a machine learning system is not the same as the traditional software system, a set of ML-specific challenges needs to be taken into consideration. In the following bullet list we report a non comprehensive summary of those challenges:

- Versions management and organization: developing ML models relies on several components: data, algorithms code and/or parameters. An important phase during development of ML models is experimentation. In this phase these components might change overtime which leads to different versions. Different versions will lead to different model behaviors, a certain version of data, code and parameters generates a certain model. There should be certain strategies to manage each version of each component. Without certain strategies to handle different versions, lose control of the ML system becomes matter of time. Creating a frozen version of the data and parameters can help us keep track of different versions. Versioning carries its own costs and complications, for example, maintaining multiple versions over time, in particular when the amount of data used grow fast in time. As briefly explained in Section 2.1, traditional

software systems utilize modular design to maintain the whole system, allowing engineers to change one part without interfering with others. While compared to traditional software systems, ML systems have no such clear boundaries between components. Keeping track of any different components version, and build an ML pipelines able to work with data versions, algorithm code versions and/or different parameters is one of the objective of this work, and MLOps in general. Any change in these components mentioned before triggers new model versions.

- Model deployment and resource management: a machine learning model can only begin to add value to an organization when that model's insights routinely become available to the users for which it was built. After the experimentation phase once the "pure" data scientists have done their job, building a model able to reach certain metrics on the test data, the model is ready to be deployed. Once it gets deployed to production, it starts making predictions against live data. There are some challenges at this stage. First of all, the deployment phase in most cases is not directly straight forward, since complex and unstructured handover are carried on by data and machine learning engineers. Moreover, there are other factors to take into account, such as how frequently predictions should be generated and whether predictions should be generated for a single instance at a time or a batch of instances (online/offline Prediction) An appropriate framework and appropriate resources (that can change over time to put right costs/performances) to serve the model based on the mentioned consideration should be plan, to let the workflow be as smooth and flawless as possible.

- Monitoring: models usually degrades over time in production due to various factors like data drift, environment changes, etc. It's essential to have the information necessary to troubleshoot and fix a problem readily available to the teams is actively working on it. We will dig into monitoring more in this work.

- Multi model management level: in real-world ML systems usually the goal is running and maintaining dozens or hundreds of models simultaneously, which leads to managing challenges. For example, monitor the whole production

pipeline, or simply, do an A/B test that is essential in ML [7].

Other problems can be faced. that is why is essential to have a strategy or workflow that helps people involved in the process to tackle those challenges.
Eventually, the goal of MLOps is to reduce technical friction to get the model from an idea into production in the shortest possible time to market with as little risk as possible [10]
In literature, there are different level of evolution and automation. Depending of the level, the challenges differ. In this section we presented the challenges of a system built without the usage of MLOps practice and concepts, also known as MLOps level 0 or traditional ML.
In the following Chapters we will have the chance to discuss how different evolution of MLOps tackle basic and more evoluted challenges. Of course higher the level of MLOps lower the entity of the challenges faced.

2.3 State of machine learning

In 2020 and 2021, a survey on 330 data scientists, machine learning engineers and managers in a broad range of companies that apply ML in their activities, where asked about their focus on the following 3 months, and what major obstacles they were facing with[11]. More than one answer could be return. The scenario given by the survey depicts a portrait of the state of machine learning in 2020: although 20% of respondents said they are still focusing more on the experimentation and learning phase, half of the respondents said that they were focused on developing models for production use and over 40% said they would be deploying models for production.
Some of the key concepts we will present in this work, that are the basics concepts of MLOps, are still not relevant for the respondents. As shown by the chart 2.3, only automating retraining of models and monitoring models in production are relevant for a small percentage of them. Machine learning in production being relatively nascent for most in 2020.
The survey was carried out again at the beginning of 2021, on the same respondents. The differences are clearly visible: monitoring models in production had the

most significant growth with respect the previous year (13% to 31%). In working economies, when demands grows so does supply. Indeed, there are plenty of tools within the monitoring space that rose to prominence.

Automating retraining of models had similar gains in priority and goals (doubling from 2020), with 30% of respondents focusing on building machine learning pipelines for the purpose.

While being by no means exhaustive, the results support that teams have made strides towards MLOps in the past year, and implementing machine learning systems (as opposed to projects) is top of mind. Last but not least, we can see that optimizing models also seemed to take a significant step back in priority within the past year.

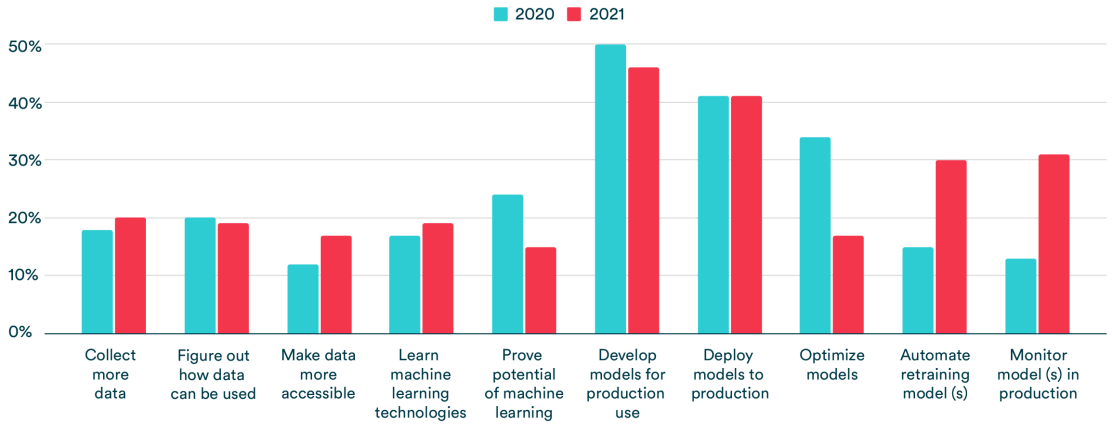

Figure 2.3: State of ML in 2020 and 2021. Adapted from [11]

2.4 The objectives of MLOps

There is a steadily rising interest in the topic MLOps, that can be easily found by looking at the search volume and news articles related to these aspects, that are becoming more and more relevant.

With this section we conclude this Chapter, in which we gave an overview of the background of our work, in which we presented how production models raise new challenges, not just for data scientists but the extended team of engineers, product managers and compliance officers, which will need to be solved collaboratively.

Figure 2.4: This chart show the interest in time of the world "MLOps", according to Google Trends

MLOps practices aims to face and resolve those challenges. As seen, in most real-world applications, the underlying data changes constantly and thus models need to be retrained or whole pipelines even need to be rebuilt to tackle feature drift. We also need to take into consideration that business and regulatory requirements can also change rapidly requiring a more frequent release cycle, and this might a huge pain the neck if not considered. This is where MLOps comes in to combine operational know-how with machine learning and data science knowledge. MLOps are basically a series of practices, that, if followed, bring the engineers (included us), to build a framework that is reliable, scalable, and represent a solid starting point of any machine learning tasks we will face in the future. This framework is mainly composed by pipelines, that are modular and reusable, also not throw-away as you can reuse parts when developing a model for a different purpose.

Chapter 3

State of the art

3.1 Differences and similarities between MLOps and DevOps

A DevOps pipeline is a set of automated processes and tools that allows both developers and operations professionals to work cohesively to build and deploy code to a production environment. It provides benefits such as shortening the development cycles, increasing deployment velocity, and dependable releases [14]. To achieve these benefits, two key and fundamental concepts are introduces in the software system development. In details:

- Continuous integration (CI): it is the practice where members team use a version control system and integrate their work frequently to the same location merging the changes to the code base to the main branch as often as possible. These changes are validated by creating a build and then verified by running automated tests against the build in order to detect any integration errors as quickly as possible. If these tests don't pass, the changes aren't merged, and developers avoid integration challenges that can happen.

- Continuous delivery and continuous deployment (CD): continuous Delivery takes CI one step further, since it enables automation to deploy all the code changes to an environment (e.g. stage, prod, etc.) after the changes have been merged. The artifact may be built as part of CI or as part of the continuous delivery process since the repository is reliable given your CI process. In other

words, this means that there is an automated release process on top of the automated testing process and that developers can deploy their changes at any time. Continuous deployment makes the process completely automated, releases are always on-going.

Similar practices can be applied to help guarantee reliably built and operating ML systems at scale. But, ML workflow is more complex as it includes several ML-specific fields, such as collecting datasets, building models, tuning hyperparameters, etc.
Bearing in mind also the differences between ML and traditional software delineated in Section 2.1, we can state that ML and traditional software systems are similar in continuous integration of source control, unit testing, integration testing, and continuous delivery of the software module or the package.
However, in ML, there are also notable differences and a new essential concept:

- CI is no longer only about testing and validating code and components. Also data, data schemas, and models must be tested and validated.

- CD is no longer about a single software package or a service, but a system (an ML training pipeline) that should automatically trigger the deploy of another service, the model prediction service.

- CT that stands for continuous training is a new property, unique to ML systems. It is essential that the model, the core business value, is always as good as possible. CT components aims to retrain the models automatically and constantly whenever needed as well as serving the models.

We can conclude this section stating that MLOps is DevOps for ML. Indeed, they have some similarities: some DevOps practices are applied in MLOps. Also, they both encourage and facilitate collaborations between data scientists, engineers and operations, to sleek the entire process and serve customers in best way.

3.2 Docker

One of the main pillars of our framework is the extensive use of Docker. Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allows the user, in this case us, to run many containers simultaneously. Containers are lightweight and contain everything needed to run the application, so we do not need to rely on what is currently installed on the host. The two main concepts that must be known while working with Docker, and that we will recall throughout the work are Docker image and Docker container. Here a synthesis of these concepts [8]:

- A Docker image is an unchangeable file that contains the source code, libraries, dependencies, tools, and other files needed for an application to run. Due to their read-only nature, the concept of image gain a great consistency, that is one of the great features of Docker. It allows developers to test and experiment software in stable, uniform conditions, without impact what is outside. An image is a read-only template with instructions for creating a Docker container, and it can not be started or ran. Rather, an image is a template employed as a base to build a container.

- A Docker container is ultimately just a running image. Once a container is created, it adds a writable layer on top of the immutable image, meaning it can now be modified. In other words, a Docker container is a virtualized run-time environment where it is possible to isolate applications from the underlying system. A great feature of the container, is that they are compact, portable units in which applications can run quickly and easily.

Docker streamlines the development life-cycle by allowing developers, in this case us, to work in standardized environments using local containers which provide our applications and services. Containers are great for continuous integration and continuous delivery (CI/CD) workflows, as we will show throughout the work.

3.3 Cloud computing and AWS introduction

Cloud computing is the on-demand availability of IT resources over the internet, without direct active management by the user. Unlike grid computing, clouds are capable with the scalability and flexibility metrics. Moreover, clouds provide service centric models to meet the user necessities. [21] AWS (Amazon Web Services) is a cloud computing platform subsidiary of Amazon.com, Inc., that includes a mixture of infrastructure as a service (IaaS), platform as a service (PaaS) and packaged software as a service (SaaS) offerings. AWS services can offer an organization tools such as compute power, database storage and content delivery services as well as customized application programming interfaces (API). This work has been entirely developed on AWS, and we are going to give a brief introduction of the key concepts used during the project implementation along with the main services used.

Chapter 4

Key concepts and AWS services used

In this Chapter we present the main concepts that drove our work along with the AWS services used to carry them out.
To give a brief introduction, we can say that we implement the entirety of our work on AWS. The motivation of this choice rely in the possibilities that it enables: a scalable, reliable, and secure global computing infrastructure [6]. Let's briefly scroll these 3 words:

- scalable means that there is no limitation on computational resources. AWS but cloud computing companies make available any kind of computational engines that can be scaled on top of each other to reach potentially extremely powerful machine, that would be super expensive if bought. In machine learning contexts that could be determinant: think about deep learning algorithms that need powerful GPUs to train. Buy them is over budget for most companies, especially start-ups, but AWS introduces the concepts of pay-as-you-go: you basically rent the resources you need, paying them only the usage time. And in ML this could be great, since the training time on an entire dataset can represent much less time than the entire experimentation phase, if the team is well organize and prepared. A strategy that we used to save money in this phase is to simply do some machine learning experimentation on a subset of the entire training dataset, where a simple instance with low computational power were enough, and when we were ready to train the whole dataset on a specific algorithm then

we used more power (hence spending more money).

- reliable stands in the fact that AWS is the largest cloud company at the time we write these lines, 2022. Their services are used worldwide and have been tested by a multitude of testers internally and users externally, and we are almost sure that these services are bug free. In most cases the documentation is exhaustive, and an intuitive API is provided for most of the services. These statement lead to a consideration: the amount of time spared thanks to the AWS choice is not negligible. Moreover, on the internet are present a good number of blogs that help you going through encountered problems.
- this work will not consider the security aspect as relevant for a main reason that solve all the possible problems: security is always managed by AWS. We can chose the level of security we desire, considering the security level a parameter in our development.

The consideration we make are of our own, and we do not consider to sponsor AWS, but, we just appreciate our choice while writing this final dissertation. From past experiences we would like also to say that without AWS we estimate that this work would have taken a lot more professional experience, competencies and more than all a lot of extra time to present the results achieved.

4.1 Code/data version control and CodeCommit/S3

In software engineering, the practice of data version control is well know. It allows you to control and revert code or even an entire project back to a previous state and compare changes over time, see the commit that might be causing a problem, and more. The most known open source software that allow this practice is Git, that has been widely used on the marketplace. In machine learning, as we have already presented, not only code but also data can change, and this increases complexity. The same practices followed in data version control applies also with data version control, with the exception that, depending on the type of data, in some cases a lot more memory is needed. The tool that has made really appreciable Git is the wide

spread of the concept of repository, that enables the practices of version control. CodeCommit is the source code storage and version-control service of AWS that contains all the code written for this project. Regarding the data, AWS is really appreciate for its storage service, S3, that, among other functions, allows data version control. The main takeaway from this section is that thanks to the structure of both CodeCommit and S3 we can easily implement triggers on the way. If we want, whenever a new version of the data is uploaded or a new commit of the code is pushed in our repositories the triggers will manage the subsequent actions: either kick off the pipelines or send alerts.

4.2 Sagemaker

The main service we used is AWS Sagemaker, the service specifically though to be mainly involved and central in machine learning using AWS. To be more precise, amazon Sagemaker is the cloud machine-learning platform of AWS that support users in building, training, tuning and deploying machine learning models in a production ready hosted environment. Sagemaker contains lot of functionalities within itself, that are still called services, in this Chapter we will explain those mainly used for our task.

4.3 ML development environment and Sagemaker Studio

Any project involved in machine learning needs its IDE to perform various task, like in a local environment. Amazon Sagemaker Studio is in not only an IDE that where classic machine learning tasks can be carried out, it also allow users to efficiently deploy jobs and resources to monitor models. It is basically a single unified visual interface where we perform the following tasks:

- Write and execute code, like in local Jupyter notebooks.
- Build and train machine learning models.
- Deploy the models and monitor the performance of their predictions.

- Carry out and track experiments.

4.4 CT ML Pipelines and Sagemaker Pipelines

One of the strength of MLOps is the reusability of the pipeline, the orderly methodology and agile modularity. To meet this conditions, the usage of pipelines in necessary in any step of the process, as we will see. To introduce the concept, let's think about the training of a model, even in a local academic system: the steps that must be always present in a ML project to do things correctly are: preprocess, model training on the train set, model evaluation on the validation set, and finally test the performances on the test set with precise metrics. The process is iterative until we find a setting of the preprocess, model and hyperparameters that satisfy the predefined conditions that our model must meet. Since we know the no free lunch theorem [16], we know that experimentation must be done, and in real work environment it is better to have modular components instead of a monolithic one, especially when experimentation is part of the process. Moreover, we desire that each step has its own configuration, because each step can have different requirements (for example a preprocessing step might be computationally expensive while a condition checking step needs less computational power. Taking the previous consideration, now we want to introduce a service that we used in the first part of the MLOps pipeline infrastructure we built for this work, namely the model build pipeline, see subsection 5.2. This service is called Sagemaker Pipelines, and it uses a Python SDK to build machine learning pipelines. With it, we can define a JSON pipeline definition and construct pipelines as a series of interconnected steps described by the JSON. This JSON pipeline definition encodes a pipeline using a directed acyclic graph (DAG), that gives information on the technical requirements for and relationships between each step of the pipeline. The structure of a pipeline's DAG is determined by the data dependencies between steps. These data dependencies are created when the properties of a step's output are passed as the input to another step [23], and the steps are executed in their dependencies order: when a step finish the following/followings are triggered. Moreover, Sagemaker Pipelines is a fully managed tool, which means that

it creates and manages resources automatically. Its utility rely in the orchestration of the steps provided by us, handling the step creations and management. Furthermore, it is possible to implement a lineage tracking logic, that lets us analyzing the data history, the insights of the pipeline as well as the various versions of the ML models we generate using the pipeline.
Just as a reminder, in MLOps, the pipelines are the real value.

4.5 Model registry and Sagemaker Model Registry

When a ML Pipelines correctly terminate it returns a trained model. To keep track of all the model's information, model versions, as well as the model itself we introduce the concept of model registry. In particular, a model registry is a centralized which stores the model metadata, such as training and validation metrics and data lineage, manages model versions and the approval status of the model. Sagemaker offers Model Registry that is integrated in Sagemaker Studio, hence pretty straight forward to use. Each project can have multiple model contemporary serving in production. Each model has a dedicated model registry. A crucial pro point is that it allows to automate model deployment with CI/CD.

4.6 CI/CD deployment pipelines and Codepipeline

Once model is ready to be deployed, it must have an infrastructure that make possible for customers to get prediction from it, in the fast and smooth possible way. We make use of serverless applications provided by AWS to construct a CI/CD pipeline for deploying a machine learning model as an endpoint. The reasons of this choice are simple: CI and CD are key points of MLOps, we know their fundamental utility. The main service that allow use to reach our goal is CodePipeline, a continuous delivery service used to automate all the steps required to release continuously a software whenever needed [24], in our case a ML one. Again, we are making use o pipeline: modularization is essential. Let's disclose the high level usage we make in this work thank to this service; this pipeline reads all code from an ad-hoc repository and

triggers CodeBuild (a service we will explain later). This CodeBuild uses serverless framework to deploy the code in the repository to a serverless function which will be directly used to make prediction by the customers. The goal is to trigger the execution of this pipeline, hence to update the endpoint whenever desired. We will our implementation and the steps chosen in Section 5.3 and explain how we set our trigger in each project in Chapter 6.

4.7 Feature Store and Sagemaker FeatureStore

Managing the feature lifetime, getting features to production is particularly hard. ML models require a consistent view of features across training and serving. The definitions of features used to train a model must exactly match the features provided in online serving. When they don't match, training-serving skew is introduced which can cause hard-to-debug model performance problems [9]. As an example, just imagine a situation where you have built your model using spark, but during serving you receive data as pandas object. It would be a catastrophic event to maintain. Traditionally, data scientists pass their feature transformations to data engineers to re-implement the feature pipelines with production-hardened code, significantly slowing down the process. These are the principal reasons why the concept and implementation of feature store has born and started to grow in the latest years. Feature store is a centralized store for features and associated metadata. It serves as the single source of truth to store, retrieve, remove, track, share, discover, and control access to features [12]. Doing so, it makes easy to create, share, and manage features for machine learning development. It accelerates this process by reducing repetitive data processing and curation work required to convert raw data into features that are subsequently used for both training and inference, reducing the training-serving skew. AWS provide Sagemaker FeatureStore, that has the capability to ingest data via a high TPS API and data consumption via the online and offline stores. The online store is used for low latency real-time inference use cases, and the offline store is used for training and batch inference.

4.8 Monitoring and Sagemaker Model Monitor

Once the model has been deployment and is serving it must be monitor under different perspective to assure the expected performance [13]. It is important to remind that any disruption to model performance directly translates to actual business loss. Let first introduce the concept of *data capture*: it is the practice of saving the data received during inference, when possible by the legislation. Doing so, we have the data our model is predicting stored in our memory, that can be eventually used in various useful ways (e.g. continuous training).
Here we show a comprehensive list of the most important things to monitor:

- **Data monitor**: data used while training are usually curated and can differ from the data that the model receives while in production. If the statistical nature of the data received while in production drifts away from the data the model was trained on, the model begins to lose accuracy in its predictions[22].

- **Model quality monitor**: when a model is deployed and serving, it is expected to perform as it does on the test set, according to chosen metrics, depending on the type of the problem (e.g. classification, regression, etc.). If it does not happen, something is changed and some action must be taken. Model quality monitor is a practice to monitor the performance of a model by comparing the predictions that the model makes with the actual ground truth labels that the model attempts to predict. To do this, model quality monitoring merges data that is captured from real-time inference with actual labels that are stored, and then compares the predictions with the actual labels. The practice that retrieves the actual labels is very often a tedious process, that sometimes is not possible without manual labor.

- **Monitor bias drift**: bias monitoring is the practice of looking for monitor predictions with bias. Bias can be introduced or exacerbated in deployed ML models when the training data differs from the data that the model sees during deployment. These kinds of changes are usually temporary, due to some short-lived events. But in any case it might be important to detect these changes.

- **Monitor computational performance and traffic**: monitoring the usage and computational performances of a model in production is essential to always visualize how our service is behaving with our clients and the cost-effective cloud cost of the service. To do so, we collect system usage metrics like GPU memory allocation, network traffic, and disk usage.

We will present our choices about the implementation set up to tackle the first three bullets, namely data quality, model quality and bias drift in Section 5.4. To end this section we want to discuss about the last remaining bullet. Thanks to AWS this is a lot more straight forward with respect to a normal on-premises cluster, because AWS has a native service that allows to monitor computational traffic with a lot of charts and accurate metrics, this service is called CloudWatch and serves as an hub to collect various logs and metrics of the general AWS services currently in use and used in the past. Moreover, AWS Auto Scaling monitors running applications and automatically adjusts capacity to maintain steady, predictable performance at the lowest possible cost impact. Eventually we would like to state that the choosing of cloud computing platform such as AWS (but also the other main companies provide similar services) could reduce the implementation work in certain areas, such as in this case.

4.9 Logs, trigger and Cloudwatch, EventBridge

In this work we used, of course, massive amount of code to implement all the needed resources and infrastructure by the concept of IaaS presented at the beginning of this Chapter. When writing and managing code and infrastructure logs have the purpose tracking error and report them along with related data, if possible in a centralized way. We also need to know of our resources are being used, to track the activity on our servers and the number of income requests that our model in production is receiving. AWS Cloudwatch is the service that helps us to achieve these needs by collecting data, logs and actionable insights to monitor running applications, and in practice helped us to debug errors encountered.
Here we also introduce a key concept, the triggers: they are used to automatically

launch some steps of our pipeline to enable the continuous training concept explained in Chapter 3. When there is an event, a malfunction, or violations of the constrains in the monitoring job, it must be taken some actions automatically, to further render the framework agile. EventBridge is an AWS service that can be event-driven, and it is used in this project to trigger actions whenever needed.

4.10 Other services used

Presenting all the services used and their details of why and how we use them is practically impossible, because they are many and it is not the objective of this work. Here we mention other services along with a short description that we use to reach our objective: build reliable and reusable pipelines that can be rapidly used in any context. These services are:

- CodeBuild, that allows us to build and test code with continuous scaling.
- Cloudformation lets us to model, provision, and manage resources by treating infrastructure as code.
- ECR, where we create and store some ad-hoc containers.
- API Gateway, used to connect the AWS environment with the outside world.
- Lambda, a serverless service that allows us to run code at inference time.
- Athena, that allow us to correctly use the feature store by creating feature groups, but also to query data in S3.

To learn more, we recommend the AWS website, in which further details are provided.

Chapter 5

Methodology

5.1 The repositories

Independently on the project, the code used is always stored in two different repository that serves as central hubs. The model build and model deploy pipelines, that form the basic structure of our work have two different dedicated repo, that contains their code.

5.2 Pipeline I: model build

We already gave a motivation about the choice of working with pipelines, that became the core of our work in each phase, including the first one in which a ML model is built, see Section 4.4.
In this section we want to explain each steps of our pipeline, the reasons, technical details and its integration in the whole framework and its importance.
The first repository (see Section 5.1) is the repository in which we store the code to manage our multi-step model building pipeline. This pipeline must be agile and modular given the fact that in a preproduction phase we want to have rapid experiments, and modular components that can be reusable and composable. While in production, a key challenge of MLOps is continuos training (CT): the model is automatically trained every time there is a trigger. Keep in mind that CT enable to have continuous delivery of models.

The backbone of this phase are the Sagemaker Pipelines, that meet these requirements, allowing resilience and reproducibility of the ML workflows. We define the following ordered steps in our model build pipeline:

1. Preprocessing step: the first step in our workflow is the process of input data. Within this step, we create a processing job for data processing. This job works as following: it copies the data from a location in S3 (the location is given as input of the step) and a python script that contains the code of the aimed preprocess. Afterword the job pulls a processing container, which contains a chosen image (see Chapter 6 to get more details about the chosen images in our implementation). Once the job completes, the outputs, namely the different datasets (train, validation and test set), are stored in S3, in a specified location, and the cluster resources provisioned for the duration of the job are shut down.

2. Model training: next comes the training of a model. This step creates a training job, that receives the information regarding the location of the outputs of the previous step (the preprocess datasets), the computational instances needed, and the S3 location where we want to store the output of the job. Of course, an algorithm is required. To provide it to the job we also pass the Elastic Container Registry path where the training code is store. Elastic Container Registry is the AWS Docker registry. Hence, a container is needed. This container must contain the information needed to use both in train and serve mode, since in the training job the model must be trained, while elsewhere it must predict the input data. This information can be encoded in the script provided to the container. This script will have the methods `train` and `model_fn`, that will be invoked in the two different respective phases of the job.
   During the training phase, once this job is triggered, it launches the ML compute instances and uses the training code and the dataset to train the model. It saves the resulting model artifacts and other output metadata in the specified locations.

3. Model evaluation: the trained model must be evaluated on the test set. Like in the preprocessing step we make use of a processing job. This job works

similarly to the mentioned. It copies the test set from S3 (the location in this case is automatically provided, since it is an output of a previous job) and the code to evaluate the model. Using this code, this processing job is able to retrieve the trained model and do prediction on the test set, and save the results (in our case the metrics).

4. Condition check : the results of the previous step must be compared with thresholds set apriori. We construct a conditional step, that support conditional branching. This way we assess whether our model satisfy predefined conditions, and, if so, we proceed to the following step. Otherwise the pipeline stops.

5. Model registration: once we are happy with the trained model, we can finally register the model in the model registry. We do that by creating a model version that belongs to a specific model group (every different task has its model registry). A model version must include both the model artifacts (the trained weights of a model) and the inference code for the model.

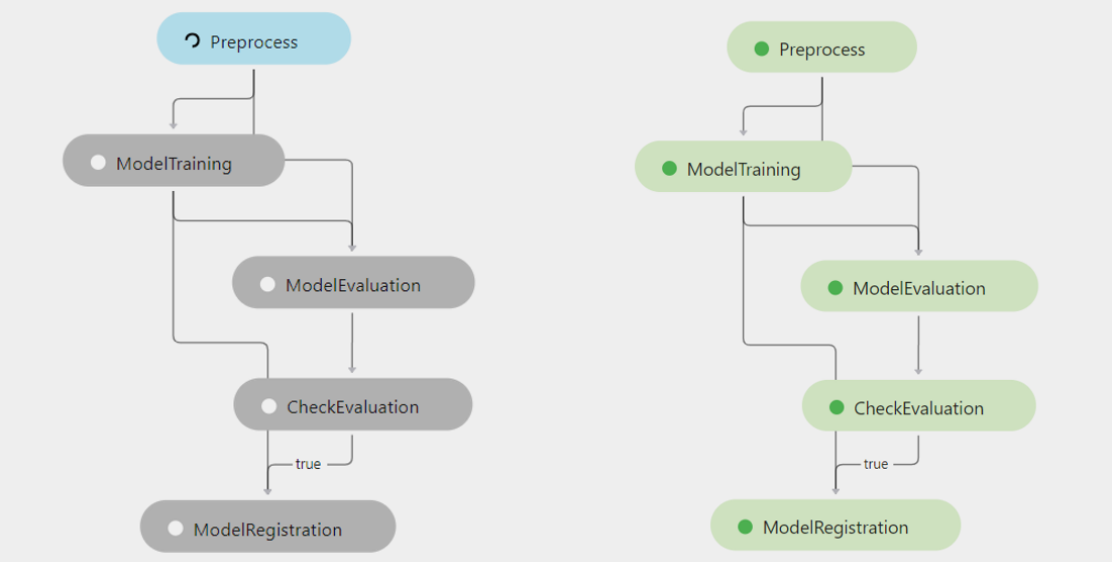

Figure 5.1: Model build graphs: the left one is a representation of the pipeline as a DAG, while it is executing, the right one once the execution is finished. Each step start only when its input step has correctly executed. The green color means that the steps run correctly. As a result the trained model has been added to its model registry

Once a version just trained of the model is added to its model registry, we decided

that, we, as "human", must always monitor what is going on, so the approval of the first pipeline is manual, and this step is managed through Sagemaker Studio, in the model registry tab. Only once we give our approval the second pipeline is triggered and the model deployment process begins.
The main take away of this section is the importance of the two components that improve the operational resilience and reproducibility of the ML workflows: pipelines and model registry. These workflow automation components enable to easily scale the ability to build, train, test, and deploy models in production, iterate faster, reduce errors due to manual orchestration, and build repeatable mechanisms.

5.3 Pipeline II: model deploy

Once the model is manually approved in the first pipeline, it is ready to be deployed to an endpoint. A good practice in real world systems is to have more than one environment. The advantages are many, like more security, faster product delivery and easier testing of innovative products. In our work we decided to set up the pipeline that will deploy the model to two different endpoints, one called `endpoing-staging` and the other called `endpoint-production`, to simulate the two different environment desired.
Our cutting edge solution implements a CI/CD pipeline using as a main service CodePipeline: we construct a pipeline able to retrieve the last version of the model just registered in its own model registry, getting all the resources information required from our second repository, and deploy the model. Let's explain the details going through the steps of this pipeline:

1. Source step: the first step of a pipeline must be a step that retrieves where the information (e.g. the code) of the actions that the pipeline have to perform is stored, in our case a repository. We decide to give the description of some of the artifacts that belongs to this repository here, to make easier the comprehension of the next steps:

    - build.py: this python file is invoked from the build stage; it contains code

to get the latest approve package arn and exports staging and configuration files.

- buildspec.yml: this file is used by the build stage to build a CloudFormation template, it contains everything needed such as commands and installation requirements.
- endpoint-config-template.yml : this file is built and deployed by the infrastructure pipeline in the required environment. It specifies the resources that need to be created, like the endpoint's.
- staging-config.json/prod-config.json: these configuration files are used to customize staging/prod stage in the pipeline. Through them we configure the instance type and number of instances desired.
- test_buildspec.yml: this file is used by the CodePipeline's staging stage to run the test code of the following python file.
- test_test.py: this python file contains code to describe and invoke the staging endpoint. We can write the code to perform all the desired tests on our model as an endpoint before deploying to production.

2. Build step: this step is the core of this pipeline, it packages the endpoint infrastructure as code defined in endpoint-config-template.yml by using CloudFormation, running a build using codeBuild. to retrieve the right model to deploy we pass to this step the build.py, that specified a model registry is able to retrieve the latest version. Once this process is done we are close to the end of our pipeline. Indeed, now we got a container ready to be deployed and ready to make predictions.

3. Deploy staging: in this step we includes three steps:

   - Deploy staging resources: this is the step that launches the endpoint in the staging environment, retrieving from staging-config.json the configuration we require.
   - Test staging: we now want to test our endpoint in a safe environment to find all the flaws, and in general the behaviour of our endpoint. Also in

this step we use CodeBuild, so we need a buildspec.yml also here in which we specify the commands to be executed to build the correct Docker image and also the script python containing the code that can test our endpoint.

- Manual approval: once we are happy with our test we can finally deploy it to production. Finally our model can bring value to our company, doing its job. Perfect! But again, as in the final step of the first pipeline, we set a manual approval to do so. Again, we always monitoring what is going on, a crucial aspect of our work.

4. Deploy production: this step does exactly the same work as the deploy staging resources; it deploys the endpoint but this time to production, the infrastructure that support the endpoint is specified in the pointed JSON.

Eventually we reach the conclusion of our entire pipeline, getting a model ready to serve to production. Depending on the real-world way of working, the possibilities to send request to the production endpoint are many. We simulate two typical situation, the main difference between them relies in the origin of the request:

- internal AWS request: in this case the request comes directly from AWS, and it is pretty straight forward as a solution. The endpoint lives in Sagemaker and it is possible to invoke it from the services in which is useful to contact an endpoint, for example Sagemaker itself or a simple lambda function (a serverless function).

- external request: now the solution becomes more complicated, but it simulates the used solution. It is the case where we want yo make our endpoint invoked by a website for example, for customers usage. Given the securities and operation policies of AWS, we must set up an intermediate layer which is knows as a gateway. The API Gateway AWS service comes in help. We specifically present the details of the tool in Section 5.3.1. Here, we specify that we construct a lambda function able to invoke the endpoint. This function accepts as parameter the name of the endpoint, the body of the request (the data we want to predict) and return the model's prediction. We connect this lambda function to API

Gateway to be able to send request from an external service, such as Postman, using a POST method.

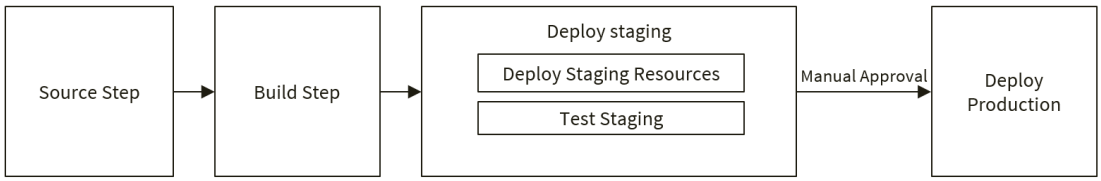

Figure 5.2: Model pipeline graph

We conclude this section where we explained the backbone structure of our work. We remark that the pipeline we built (both the model build and model deploy) are reusable and highly modular, so when a new ML project shows off we have a solid MLOps pipeline that can helps us to reach the market in the fastest time, giving to the future developers the opportunity to focus more on the experimentation part of the pipeline.
Moreover, this pipeline enables the fundamental CI/CD/CT: we are able to state that the entire workflow, even if highly modular behaves as a whole, being the components entirely connected. Whenever needed (for example when a commit is done to the central repository) the pipeline can be automatically triggered, like we do, and it executes again this time using the new code/data version.

5.3.1 API

The endpoint in production must be reached in some way by the external world, to allow users to effectively be served by the created endpoint. APIs act as the "front door" to access our endpoint from the outside. Specifically, we created a web API with an HTTP endpoint for a Lambda function that manage the incoming requests, by using Amazon API Gateway. Hence, this tool allow us to create a web APIs that route HTTP requests to the Lambda function. Once the Lambda is called, it process the incoming request and invoke the endpoint in production, to get the predictions of the model. The prediction does the reverse way to the user, that receive the feedback of the model for the data he/she provides.

5.4 Monitoring

One of the main pillars of this work is the monitor phase. The two pipelines just presented (model build and model deploy) run and an endpoint is correctly serving in production. We show in this section how to correctly set up monitoring activities to monitor the quality of our predictions. These pipelines we present here can launch triggers to alert us whether our predictions do not satisfy certain requirements, both under statistical and quantitative perspective. The explanations about motivation of these pipeline are presented in Section 4.8, here we present our implementation of them. We use the plural since we implement two different pipelines:

- data monitor pipeline: it will generate triggers and alerts every time the statistical nature of the data in the training will differ significantly from the statistical data of the data received in production. The first ingredient that must be present to make the monitor possible is the data capture. To make this ingredient available we configured our real-time inference endpoint to capture data from requests and store the captured data in S3. After that we follow these steps:

    1. Create a baseline. We need reference statistics that will be used as baseline to confront incoming data to monitor if data drift and other data quality issues can be detected. There are different techniques to do so, but the main idea is the following: analyzing the training dataset to build baseline and constrains for each feature in the dataset. This is done because first of all the training dataset data schema and the inference dataset schema should exactly match, but also the statistics of each feature calculated during training should correspond to the statistics calculated on the same feature using batch of data captured during inference. So after this step we got two files: baseline and constrains, that contains all the information needed for the next steps. The logic to construct these two files change depending on the needs, as we will see in our different projects.

    2. Schedule data monitoring job. With this step the concept of "continuously"

really comes to light. We continuously monitor what is going on by kicking off processing jobs at a specified frequency to analyze the data collected during a given period. In other words we group the data collected in each period and make the same passages used in the previous step to compute the baseline file. We then compares the dataset for the current analysis with the baseline statistics, constraints provided and if violation are present we generate a report. If the violation file is created we immediately generate a trigger that will send us this file but we can also trigger the execution of the main pipeline, starting from the model build pipeline, and the process start again. Here comes the core of CT. Under the hood we enable Sagemaker to launch resources, when needed (for example if we schedule the job once a day, once a day the needed resources will be launched), and use a container to do the passages explained. Once the comparison is carried out the resources are shutted down.

- model monitor pipeline: this time we want to monitor the prediction performances of our model in production. To do this we need to know how our model behaved during training. For example if a model get a 50% accuracy on the test set but only a 30% during serving, an alert must be triggered. The logic implemented in the data monitor pipeline is emulated in the model monitor pipeline with some changes. The steps are the following:

  1. Create a baseline. In this case we launch a baseline job that takes the model predictions and compares them versus ground truth labels. It calculates metrics for the model and suggests statistical rules and constraints to use to monitor model quality drift by defining thresholds against which the model performance is evaluated, and store these information in a baseline file.

  2. Schedule model quality jobs. This passage is very similar to the one presented in the data quality pipeline. A processing job is kicked off at a given frequency to group the model prediction of a given period. The difference this time is that to carry out the monitoring analysis we need the ground

truth of the serving data. The strategy to label incoming data changes depending by the context we are working in, but in general is one of the main tedious process we can face in machine learning. Not in few cases human work is required to address this task that can be costly and time consuming. Once we got the labels of the prediction our model made, the same logic can again be carried out on the training dataset, and the comparison will bring out eventual flaws in the predictions alert will be triggered and the training pipeline will be kicked off.

Generally speaking, monitoring jobs are one of the most useful tool presented in this works, because they contribute to the correctly meet the needs' projects we are working on, and consequently to bring values to our company without massive human intervention.

5.5 Feature store management

We decided to present the work we have done regarding the feature store following our approach, that is detached from the pipelines. In both our projects we followed the same path, among the various that exists to build a feature store. The reason of this choice relies in the following: one of the main utilities of a feature store is to have a shared feature orchestrator among the organization we are working in. Hence, this concept might come in help in our context, MLOps, but it is not the only context where it is used.

In this section we want to give an overview of the main procedures to build feature stores, that are shared among projects, by a more theoretical point of view and later also by a concrete example. First, we introduce some terminology that will help the readers to understand all the passages: a FeatureGroup is the resource that contains the metadata for all the data stored in the feature store. Mainly, it contains a list of FeatureDefinitions. A FeatureDefinition consists of a name and a data type. Under the hood in the FeatureGroup is encrypted an OnlineStoreConfig and an OfflineStoreConfig to control where the data is stored, that will be used whether the feature store is used in batch (offline) or online mode.

Now we can start to explain the process. First we need to deeply inspect the starting dataset (we can think about the training dataset, they are often the same) and its features, and decide how we want to manipulate our feature in order to have them shared and usable in the various phase of the project. Once we know how to manipulate them we can concretely do this step locally, in our notebook. At this point we have the raw starting dataset and the processed one. Now we can define and instantiate the FeatureGroup and the FeatureDefinitions. The latter will be nothing else than the formal definition of the resulting features after the manipulation phase. Once a FeatureGroup is created, data can be added as records. Records can be thought of as a row in a table. Each record will have a unique identifier along with values for all the FeatureDefinitions, that are the result of the processing phase of the raw dataset. In order to be clearer, with an abuse of terminology, we can imagine that once the process is complete the feature store will contain a dataset composed by unique combinations of processed features ready to be quickly retrieved in any process of the entire framework.

Let's make an example. Imagine in our dataset we have a text feature that encodes different categories. We can have problems on why and how we can use the feature store: some algorithms may have problems when dealing with this kind of feature, to prevent them we can for example use some kind of preprocessing techniques, such as the one-hot encoding. The number of the created columns using the mentioned techniques is unique, because it is the number of categories we have, but their order it is not unique, it may vary in different phases. To avoid this problem we can order the created columns. One may think that the problems are resolved. But what if a new category appears? This may cause a problem if this preprocessing techniques is repeated in various stages of the framework, because in different phases some categories might not appear. But also, the biggest problem comes to the surface when we have combination of categorical features that may not appear in the starting dataset, see Figure 5.3 to have a clear insight. To avoid any kind of problem we rely on the concept of feature store, where we carry out the preprocess phase just once and to obtain a central shared feature repository ready to be used in the entire organization.

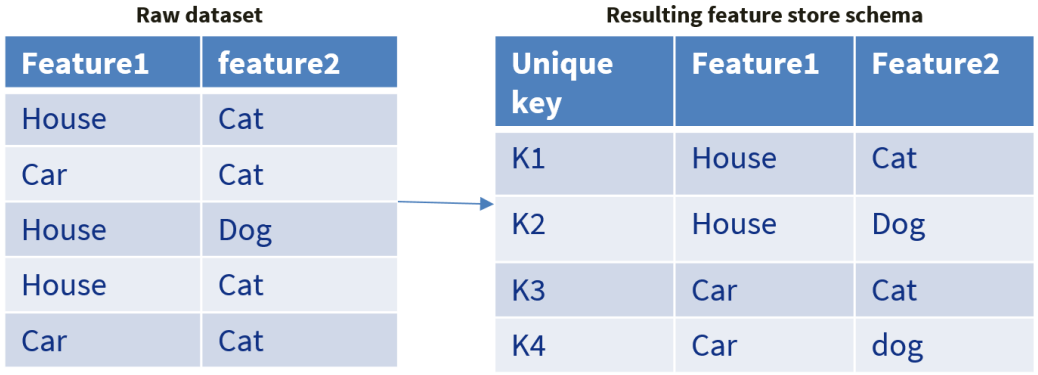

Figure 5.3: In the raw dataset we have repeated rows and not all the combination of the categorical features appear (the combination car, dog is missing). In the feature store there will be unique rows where all the possible combinations are present and identified by a unique identifier, that can also be a string containing the combination(e.g. "car,dog").

The feature store can be used in online and offline mode.

- In online mode, features are read with low latency reads and used for high throughput predictions. To retrieve them, we must call the defined feature group of interest from the service we are using (for example a Lambda function) and provide to it the unique keys of the defined group. It return the related features. This mode is usually employed during serving time through an endpoint.
- In offline mode, large streams of data are available through the offline store. Using this configuration, to retrieve them we must provide the unique keys of interest, but this though Athena. Indeed, the offline configuration works in conjunction with S3, where the data of the feature group of interest are stored. It can return a table object. This mode can be used training time or at and batch inference.

5.6 Infrastructure

We conclude this Chapter by showing how we link all the pieces together, in Figure 5.4.

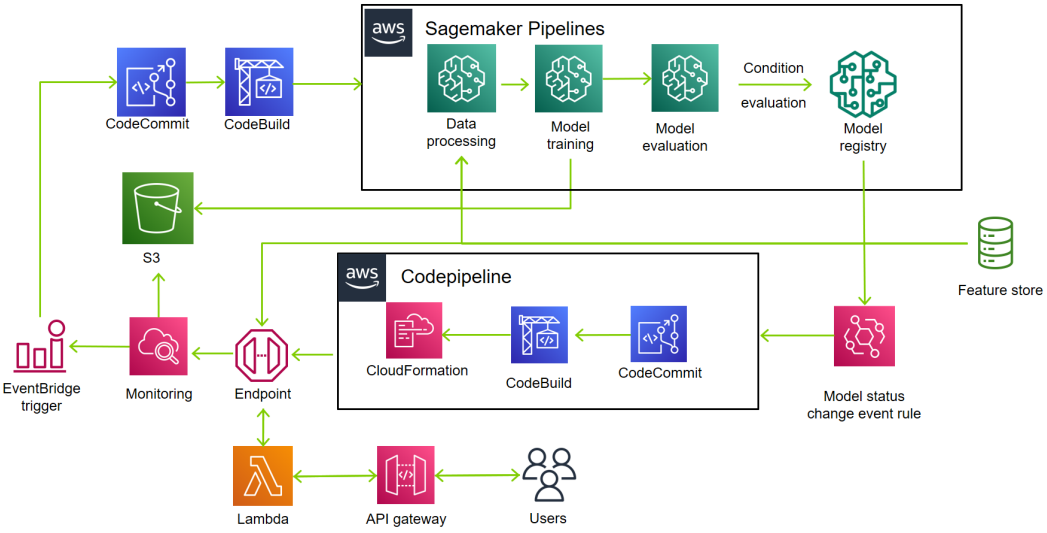

Figure 5.4: The main infrastructure

Chapter 6

Projects implementation

The objective of this work is to show how MLOps practices effectively work and how we can benefit from them.
In this Chapter we present how the entire workflow behave in practice through two different projects, that have different characteristics. This way we can appreciate even further the benefits of implementing MLOps best practices: our framework can be easily and quickly adapted in different contexts that don't have much in common, such as binary classification and time series analysis and forecasting using deep learning models. We expose here the technical details that characterize each project, and how our implementation help us to streamline their delivery to production, helping our company to have real-value benefits using machine learning without having to wait the time needed using a "standard" machine learning framework (e.g. without implementing MLOps practices).
We will present the projects in the order we followed, step by step. Even if there is not a fixed pattern to follow that is universally the best, we can state that a following our steps order can be a good practice.
We highlight here that the data used have been retrieved from public dataset or synthetically modified to avoid any data property related problems.

6.1 Binary classification

The first case we present is one of the "easier" setting to deal with among the total machine learning methods. Still, in a true production environment the challenges to update a binary classification algorithm can be a lot and it can requires years to keep up the model with business requirements and/or world changes (that are reflects in data changes). We present how our framework helps to streamline the workflow.

6.1.1 Data exploration

First, we show the dataset, and its main characteristics, to be able to understand our task and possible problems we can face. The starting dataset in question is a bank dataset [17], that has been slightly modified for achieve our purposed in a better way. For completeness, we cite that, as the name suggests, this dataset was created for a bank marketing campaign. Each row represents a possible client along with its information, like for example age, job and marital status. The desired target is to predict whether the client will adhere to some campaign.
Since our framework can adapt to different settings and different tasks, we decided to chose a dataset that can be helpful for our purpose, that is showing how MLOps work and how much can help streamline the workflow to deliver ML models in production, so we will not dig too much in the data exploration part, just showing the essentials and what matters for this thesis. Hence, we chose this dataset for its characteristics, indeed, it is composed by both numerical and categorical features, and the combination of them can be interested for our showing purposes. The total number of features is 14 plus the outcome variable, the y, while the total number of rows is 40689. Since we are not really interested in the real meaning of this features, and what they represents, we The characteristics of these 14 variables are divided as follow :

- categorical features: features that can only take a limited number of possible values. We list them here in the format name_of_the_column: [possible_values_it_can_take]

  - job : type of job: [blue-collar, management, technician, admin., services, retired, self-employed, entrepreneur, unemployed, housemaid, student, unknown]
  - marital : [married, single, divorced]
  - education:[secondary, tertiary, primary, unknown]
  - default: [no, yes]
  - housing: [no, yes]
  - loan: [no,yes]
  - contact: [cellular, telephone, unknown]
  - month: [jan, feb, mar, ..., nov, dec]

We print the head of this subset in Figure 6.1.
Also, we would like to highlight a concept here. The combination of all these feature is possible, and must be taken into account for the next steps, even the number of possible combinations (41472) is higher than the actual number of sample presents in the training dataset.

	job	marital	education	default	housing	loan	contact	month	poutcome
0	blue-collar	married	primary	no	no	no	cellular	aug	unknown
1	self-employed	married	tertiary	no	yes	no	cellular	sep	unknown
2	services	married	secondary	no	yes	yes	cellular	may	failure
3	admin.	single	secondary	no	yes	no	cellular	apr	failure
4	management	single	secondary	no	yes	yes	cellular	aug	unknown

Figure 6.1: Head of Pandas DataFrame containing only categorical features

- numerical features: feature that can take a numeric value.
  - age
  - balance
  - day
  - duration

– campaign

– previous

We print the head of this subset in Figure 6.2

	age	balance	day	duration	campaign	previous
0	45	185	25	78	4	0
1	32	1676	9	223	2	0
2	39	-762	8	40	6	2
3	48	3531	20	81	2	6
4	37	24	11	232	4	0

Figure 6.2: Head of Pandas DataFrame containing only numerical features

6.1.2 Creation of the Feature Store

In the previous Chapters we explained why the concept of feature store can be crucial, and how to use it depending on the step. Indeed, we first need to instantiate the feature store, create the needed feature groups, fill them, and once it contains all the records we desired it can be ready to used at training time (using the offline mode) and at serving time (using online mode in this case).
In this section we present why the feature store is needed in this specific project and how we use it. We have just seen that our raw dataset has 8 different categorical features. The combination of the values that these features if higher than the actual number of rows in the whole dataset. It means for sure that not all the possible combination are present. If we would be in "local" framework (e.g. carried out with a simple personal computer) a part of the preprocess phase would be to simply split the entire dataset in train, validation and test set and fit and transform the subset of the train set with the one-hot encoding techniques. In this way we process these 8 categorical features obtaining a processed train dataset that has more columns, namely the sum of the possible values of all the categorical features (in our case 40). After that we can transform the validation and the test set according to the fit on the train

set. Even if really basic, this techniques can avoid a lot of problems when dealing with categorical features, in particular when the categorical features are strings. Indeed some models, like XGBoost, can not deal with strings. Furthermore, some models can benefit thanks to this practice, and so better results can be obtained.
In a real-world example, this is not that straight forward, because of some reasons: first of all the two fit/transform (the train set) and transform (data to predict) are not one after the other but basically at the beginning and at the end of the workflow. So the fit/transform job should be in some way fitted at the beginning and applied at the end. It is doable, but with some pain in the neck and lot of problems may arise. Just think that every time there is a slight change all the process must be manually done again. Another problem of this approach is the following: it can happen that the categorical values of a feature that appear in the train set are not all the possible values the feature can assume. Again, all the process must be one from scratch, with a lot of time wasting.
To overcome these challenges we create a feature store to store the one hot encoding of our categorical features. This feature store must contain all the possible combination of these categorical features, even if they do not appear in the entire dataset. To do that we take all the possible combinations of the categorical features and fit the one hot encoding transformation on this resulting dataset. Also, this dataset must contain a column that identify any row, e.g. the unique key. We decided that this unique key is a string that contains the concatenation of the text values of the combination. Once we have this dataset (the head of this dataset is shown in Figure 6.4) we are ready to followed the steps described in Section 5.5: first we define and create the feature group, second we instantiate the feature definition (meaning that we indicate the column that contains the unique keys, and the data type of each column in the desired feature group we are creating. Now we are ready to ingest the processed dataset. Once the ingestion is terminated without errors we obtain our feature store ready to be used in offline and online mode. Since we have our beautiful feature store, we can exploit it as much as possible. Indeed, among our trials, we used the offline mode, in particular in the training phase, but also, since we chose to implement a real-time inference setting, we will use the online mode during inference. We

	combinazioni	job_admin.	job_blue-collar	...	poutcome_other	poutcome_success	poutcome_unknown
0	admin.,divorced,primary,no,no,no,cellular,apr,...	1	0	...	0	0	0
1	admin.,divorced,primary,no,no,no,cellular,apr,...	1	0	...	1	0	0
2	admin.,divorced,primary,no,no,no,cellular,apr,...	1	0	...	0	1	0
3	admin.,divorced,primary,no,no,no,cellular,apr,...	1	0	...	0	0	1
4	admin.,divorced,primary,no,no,no,cellular,aug,...	1	0	...	0	0	0

Figure 6.3: The categorical processed dataset ready to be fed into the feature store. The column "combinazioni" is the column that contains the unique keys.

will comment the next phases in the following sections.

The creation of the feature store is automatic, that means for whatever reasons the notebook that contains the code to create the feature store can be executed automatically. The results will be to have another adjourned feature group. The following steps of the whole framework we present are implemented to retrieve the data from the latest creation of the feature group for a given dataset (e.g. in our case project). In this way the process is always coordinated and up to date. Still, the deletion of an old feature group that can be useless must be done manually, to always check if the process is working as expected, avoiding no come back actions.

6.1.3 Model build pipeline

In this section we present the project's implementation of the pipeline presented in Section 5.2, that contains general insights and technical and theoretical details. The aim will be to have a trained model on adjourned data that, if the chosen conditions are met, will be registered in the appropriate model registry and ready to be deployed in staging, and eventually in production.

This pipeline manage to retrieve from a specific location in S3 the training data at a certain time. The data on S3 can be updated whenever needed in this location, so if we obtain a new adjourned dataset that will be uploaded in this location EventBridge will trigger the execution of the entire process that eventually can update the model in production, if all the automatic and manual conditions are met

Another trigger that can execute the process is a push to the central model build repository, that contains all the code that manages this pipeline.

The triggers work simply by executing the lines of code of the update model build

repository.
Once we have the training dataset the execution of the pipeline is managed by a central python file that contains all the useful information of the steps we want to execute, including resources information and estimators type and location, and where to retrieve the code needed stored in the rest of the repository.
We highlight the usage of Sagemaker Pipelines, that allow us to have modular steps that in most cases are handled by class provided by AWS. Indeed, as already mention, AWS provides a lot of functionality that in some cases can be just executed and that integrate in their environment. In this specific binary classification projects the steps that characterize the model build pipeline are the following:

- Preprocessing step: this step takes as input the entire training dataset, the code that will be run to materially carry out the preprocess and the resources detail chosen that will be used to initialize an instance that will be used to execute this step. Moreover we pass as input the output paths that will contain the output artifacts of this step, that will be used as input of the next step. The resources we chose are pretty standard, called `ml.m4.xlarge`.

  We can now focus on the true preprocessing part, analyzing our code. Since we decided to use XGBoost, we need the one-hot encoded features, that are already created in the feature store. To retrieve them, we call the feature group. In the offline mode, the data of interested can be selected in a SQL style through Athena (since they are in S3). Now we got the processed subset of the categorical features, so we can concatenate for each sample the raw numerical features and the processed categorical ones. Since we have a dataset having much more columns than before, in the experimentation phase of the project we tried different approaches, such as PCA, but to apply it in production we should deepen some concept, as we will see in the next step.

  Of course, the goal of this thesis is not to find out the best preprocessing pipeline, or the best model that fit the data, but to show how the practices we present streamline the flow. Now we have a numerical dataset that can be carved in a countless ways, just experimenting, and then figuring out how to avoid any training-serving skew problem either using the feature store or another agile

method, such as the usage of Docker containers.
The last lines of code of the script ingested by the preprocessing step is to split the resulting dataset in training validation and test set. They will be the output of this step, and will be used as the input of the next ones.

- Training step: the aim of this step is to generate a model artifact. This step takes as input the training and the validation set generated from the previous step, and the estimator with the related resources desired along with a Docker image. The image is the crucial part of this step: it contains the ML algorithm to be used in this project. We tried out different images. These images can be divided in categories:
  - Prebuilt Docker images: AWS provides containers for its built-in algorithms but also prebuilt Docker images for some of the most common machine learning frameworks and algorithms. XGBoost is included in them.
  - Ad-hoc Docker images: for several reasons, we may want to use an ad-hoc image that can perfectly fits our task. We can either extend the prebuild container provided by AWS or create a container from scratch.

  In the first attempt, to familiarize with the tool, we used a prebuilt image having as framework XGBoost. This way, when executed, this step returns a model artifact without any further effort. This way life may seems simple. But in real-case example it is not. Hence, we carried out different experiments also providing ad-hoc images.
  Of course, the image must be created. In AWS there are lot of ways to do that, also in Sagemaker. We briefly explain our implementation strategy: we create a notebook, that is of course separate from the entire MLOps workflow, where we create it. If we want to extend a prebuild container here are the requirements needed from AWS: we need a text file and a python script. The text file contains the EMR specification of the image to extend and some other configuration specification. The python script contains the code of interest, which,among the others, must have two functions that specifies the training and the predict logic, respectively. In the preprocessing step we cite that, since we have a lot

of numerical columns, a processing technique such as the principal component analysis (PCA) can enhance the overall performance of the model. The PCA must be fit on the trained dataset and later applied on the test/serving data. Again, these two phases can not be done sequentially in a simple environment (e.g. like a notebook in a local personal computer), and a strategy must be thought. Our strategy can be carried out simply extending a prebuilt image, and it is the following: we extend the XGBoost image with a script that instead of just train XGBoost's algorithm train a SkLearn pipeline with two steps in it:

1. Fit and transform of the training dataset with PCA.
2. XGBoost model training.

This way, in the serving phase, through the predict function we call the SkLearn pipeline predict method, and the game is done. Again, any training-serving skew problem is avoided observing this strategy.
Independently on the type of chosen image, the output of this step is a model artifact, that is stored in a specific location in S3 in a `tar.gz` format.

- Evaluation step: this step receive as input the model artifact generated in the previous step, the test set and of course resource and infrastructure information to let the flow be as smooth as possible. As in the preprocess step, a python script is required. Indeed, in this step we want to evaluate the artifact on data never seen before by the model, the test set. In the evaluation script we load the model and the test data and we call the predict function of the model. We compare the prediction values with the ground truth, calculating the accuracy. Moreover, within this step we create an evaluation report. It is nothing more than a python dictionary that will be saved in a specific S3 path. This dictionary contains the accuracy just calculated, and it is stored for lineage purposes, since it can be useful in the future, but, most importantly, this dictionary is the output of this step that will be used in the next one.

- Condition check step: this step reads the output of the previous step, the dictionary containing the model accuracy and compare it to a threshold we define

apriori. In our case we chose to set a threshold that would be easily met every time this pipeline is executed, since we do not care about the actual performance, but only on the pipeline itself. This step accepts two hyperparameters, respectively "if_step" and "else_step": it means that if the condition is met, the specified argument in the "if_step" (e.g. another step) is executed, otherwise the "else_step". In the former step we specified the registration step, while in the else step we do not specify nothing, since the if step is always met in our setting.

- Registration step: this step loads the artifact in the appropriate model registry. If the model registry does not exists it will be created. The model registry assigns the version number to the artifact that is being registered. Of course, the first artifact has the version number 1, the second the version number 2 and so on. We specify in this step that a manual approval is required to release the selected version to flow through the next pipeline, the model deploy pipeline.

Once all these steps have been executed, the manual approval is required to trigger the following model deploy pipeline. The DAG that represent the model build pipeline in this project can be seen at Figure 6.4.
We mention here another step that has been tried out, even if we decide to not include it in the final version of this thesis, that is the processing step where we tune the hyperparameters of the chosen model, XGBoost. This step can be easily integrated in our workflow. Doing so our pipeline will become more deep, having one more step, and since it is unnecessary for our purpose to have an highly tuned model we decide to try it out to study how it works but not exploit it too much to spare time and money. Using the steps provided by AWS is not the only way to carry out the hyperparameter tuning phase. Indeed, it can be also carried out inside the container image, or in a separate location (not within the pipeline), because it can require a lot of time. It is basically the core of the experimentation phase.
Eventually, if dealing with big datasets, we recommend to experiment a bit on a subset of the entire dataset in a separate location (we did it in a Sagemaker notebook), just to get insights of the data. Then the usage of MLOps practices can enforce the

tidiness of the workflow, and help to keep track of all the experimentation carried out. The modularity of the steps would help to always arrange a better setting that working "randomly" on a local notebook.

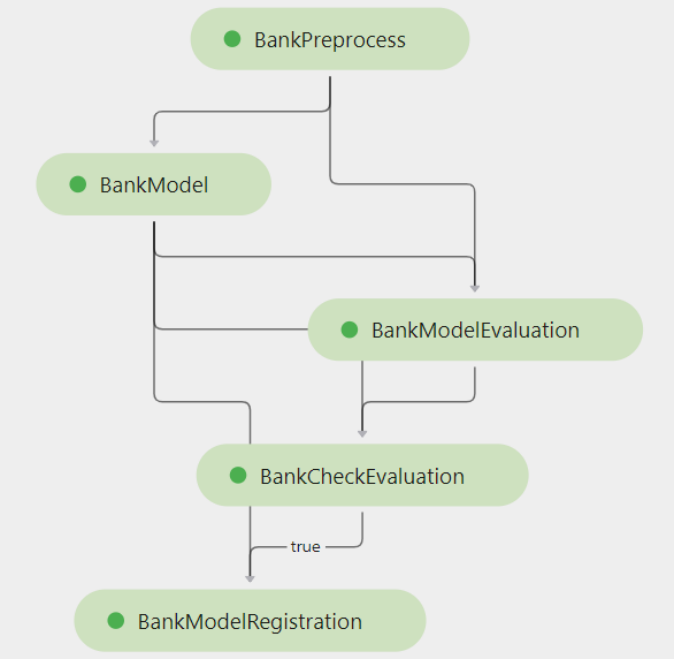

Figure 6.4: Graph containing the steps of the model build pipeline for the binary classification project

6.1.4 Model deploy pipeline

In this section we explain the usage of this pipeline in this first binary classification project along with its peculiarities. The aim of this second pipeline is to deploy and endpoint from which is it possible to predict incoming data through a POST request. Before going into the details we can state that this pipeline is quite solid, and the innovations it bring can be easily adapted on the project we are working on, without much effort. Anyway, we look at the its singular implementation in this project. The code that will be execute one this pipeline is triggered is in the model pipeline repository. As shown in Section 5.3 the repository contains some crucial files that manage the pipeline flow. This repository is really reusable thanks to its parametrization. This way, the yml scrips contained in the repository are ready-to-use without having to change them, except for some parameters such as the name of the model

registry of interest. Depending on the project, we also need to specify the resources desired as usual when working in the cloud. For this project we chose really standard engines, such as `ml.m4.xlarge` and `ml.t3.medium`, since our dataset is not too large and the service load would clearly be not demanding, since we are doing a thesis demonstration. In particular, both in staging and production we use a single `ml.t3.medium` instance for the respective endpoint. Whereas regarding the python file, we have two main python file: build.py and test.py. As explained in Section 5.3 the former is more similar to a configuration script, and it is highly parametrized as well, while the latter is used to test the endpoint in staging. Hence, we can exploit all its use-fullness in this project. We can carry out different tests, that belong to three main categories:

- Performance testing: this type of testing examines responsiveness, stability, scalability, reliability, speed and resource usage of the infrastructure.

- Load testing: it measures how the system handle heavy load volumes.

- Stress testing: it is a type of performance test that checks the upper limits of our system by testing it under extreme loads. Stress tests examine how the system behaves under intense loads and how it recovers when going back to normal usage.

In the test.py script we check whether the endpoint is reliable, meaning that the expected type of response depending on some particular requests is satisfied. Hence, performance testing. Regarding the reliability of the system, that must always ensure responsiveness to an incoming request we are supported by AWS. Of course we must chose an instance that is proportional with the expected load. Nonetheless, AWS provides a lot of option to adopt a scalable systems always ready to scale depending on the incoming traffic. We will not deepen this topic.
Once we are sure that the endpoint in the staging environment works as expected we can finally deploy the endpoint in production. From now on our work can generate true value for our company.

API

To serve incoming requests, we must use an API to allow users access the predictions of our model, given input data. We linked to our API a serverless function (Lambda function), that receive input data and process them to avoid any train-serving skew problem. Indeed, here is where the feature store plays a central role. Once receiving the data, we call the online store of the feature group, and retrieve the correspondent data for the categorical features. Now we invoke the endpoint in production, and the user can finally see the prediction of our model for its data.

6.1.5 Monitoring

Once we are out in production we must be sure that the model behave as expected, to always deliver an high quality service. To do so we employ monitoring activities that will be able to automatically check whether everything is working as expected, and possibly trigger the execution of the entire process in case some violations are detected. In this section we show the implementation of these pipeline in this specific project, moreover we simulate with the usage of toy data some situation in which we expect these pipeline to report the violations and consequently we expect the process to restart again, from the model build pipeline. In Section 4.8 we gave the concept overview and the reasons why we implement these pipeline, while in Section 5.4 we present technical details. Here we adapt them in this binary classification context. Like the model deploy pipeline, these pipelines are highly reusable in various project. The major difference in the implementation of these pipelines among different projects are the baseline dataset used. Let see in details both the data monitor and model monitor pipelines for this project, following the structure of Section 5.4:

- data monitor pipeline:

    1. We create the baseline starting from the training dataset after the preprocess step, that can be retrieved from S3. The two file that have to be created as baseline, the constrains a statistics json, are created using the Deequ library: Deequ is a library built on top of Apache Spark for defining "unit tests for data", which measure data quality in large datasets.

2. Once we have this two files we can can set the monitoring schedule as desired. We set it to run every hour. So every hour we monitor whether incoming data are changing in their statistics as explained in Section 5.4.

- model monitor pipeline:

1. Again, the baseline is created starting from the training dataset. The way we create this file is pretty simple, indeed it contains basics statistics such as the class balance (e.g. the relative percentage of the classes) and the type of prediction (either 0 or 1 since we are in a binary setting). If the model will drift in its predictions towards a class, then alerts will be triggered.
2. Regarding this pipeline, every time the monitor quality job run there should be the ground truth of the incoming data already collected in S3. In the following subsections we will explain how we carried out some tests, to give better insights about the process. Nevertheless we schedule the job to run every hour. When new ground truth data are not present in the given path, the job will simply return an alert that no file have been found.

Data monitor pipeline tests

The objective of this paragraph is to present how we check whether data monitor pipeline works as expected. When we want to carry out these tests, we must have an endpoint ready to serve our requests, with the data capture enabled. We create synthetic samples to complete this task. We carried out two different type of tests:

- Check whether the data type do not differ from the data type of the baseline: we fed to the endpoint data with the same number of attributes, but with different data types.
- Monitor data drift: To check whether the data monitoring job works properly, we create a fake dataset with manually modify attributes. As an example, in one of our tests we took 100 samples of the test set and we randomly change it by adding values to the "age" column, by the formula $age_i = age_i + random.randint(0, 50)$. Afterwards we are going to request predictions

for this 100 sample through our endpoint. Doing so, when the monitoring job computes the statistics for the column age, it should detected a data drift.

This pipeline behave as expected, and once violations are reported, it triggers the execution of the model build pipeline to deploy a model that can be trained on new data (if the ground truth is available for the serving data).

Model monitor pipeline tests

In this pipeline we explain how we check whether the model monitor pipeline works as expected. Again, while carrying out these tests, we have an endpoint ready to predict incoming data. We report in this thesis one of the tests we carried out: we fed 100 samples to the endpoint, that has the data capture enabled. Each sample will have a unique when captured, and the predicted values will be associated and saved along with its features. Now we have to associate to each sample that has been predicted its ground truth through the needed key. This time, to trigger the violations of the monitor pipeline, we generate fake ground truths that do not match the predicted values. Again when the job run, it correctly generates a violation report, and consequently the execution of the model build pipeline starts.

6.2 Time series forecasting

The real value of the entire work is the framework, not the specific use case. Our main objective is to show is reliability and re-usability, independently on the specific settings, either binary classification or something else. To show this, we present a second project implementation, that as nearly nothing in common with the previous one in its basic ML concepts, like time series analysis and forecasting, applying machine learning models.
Hence, since our objective is not to build the best machine learning model for this specific task, we do not intend to dig deep into the "data science" part of the project, but more to highlight and appreciate the goodness of the framework presented. Furthermore, we remark here that the time series analysis and forecasting is in general more difficult and require more attention than a simple binary classification context,

so the parts of the next section that uniquely belongs to this context and the chosen dataset can be surely studied more deep, and it is a part of the future studies of this thesis. Also, we use this project as an opportunity to study deep learning models in the context of time series, but also to learn how to use external data sources to enrich the starting dataset, as we will show in Section 6.2.2.

6.2.1 Data exploration

The dataset we used for this project is a synthetic dataset, that we created in order to avoid any possible problem. Still, it retraces a true dataset that inspired us to implement this project. Let's think that we collaborate with a chain company that wants to predict its income by analysing its data. This chain has several stores, that sell several products. Each product belongs to a category, and has a unit price that can change over time. In particular, we have 6 stores that can sell 17 different products, hence, we can have 102 different time series that model the selling quantity rate of each available product in each store. To add more complexity, but also to reproduce faithfully the reality, we built 98 different times series. Indeed, in the real world it can happen that different stores sell different products, even if they belong the the same chain company. The time span is two years. This company provides us a typical dataset that chain store collect. In particular, the dataset has the following columns:

- date.
- store key, that is a unique key for the respective store.
- item key, that is the unique identifier of an item
- unit price, the price of the item for the respective date in the respective store.
- item category, each item belongs to a category.
- quantity sold, the quantity of items sold in store for the given date, that, is our target.

	date	store key	item key	quantity sold	unit price	item category
0	2018-12-30	60	349	33	26,3	21
1	2017-10-25	60	347	59	26,25	21
2	2019-02-26	60	351	62	19,42	21
3	2019-03-11	60	348	104	20,19	21
4	2019-03-20	60	433	96	27,86	21

Figure 6.5: Head of Pandas DataFrame

The main flaw of this dataset is the presence of missing data. Depending on the location, and on the week days, some shops are closed while others are not. So data are not homogeneous under this point of view, and in this context that may arise big problems in the machine learning model development. We will show how we tackle this problem in the model build pipeline subsection.
Moreover, we know the geographical location for each store, having the longitude and latitude. We will show how we use this information in the feature store in Section 6.2.2.

6.2.2 OSM and creation of the Feature Store

This dataset is pretty basic in its features, and can be enriched to have a better insights on the numbers.
Since we have the geographical coordination we can exploit them. Intuitively, the location of a shop can be related to the quantity sold of certain items respect to others (for example a shop located in a city center can sell more items related to the tourism). Among the many ways to extract locations information from the internet, we make use of OpenStreetMap (OSM).
OpenStreetMap is a free, editable map of the world, created and maintained by volunteers and available for use under an open license. Anyone can use OSM to provide maps, directions, and geographic context to users around the world [18]. We will not explain the details, but we manage to retrieve through Athena the OSM informations

about the location of our stores in a 1500 meters range. It means that for every store, we got the OSM specifics in a 1.5 kilometers radius. Through an ad-hoc preprocessing, we retrieve the number of point of interests (POI) in each area. In particular, for our purpose we obtained the number of amenities and tourists POI only. The Processing job along with the script used in this phase can be automated as well, in order to run this process every time is needed without human intervention, but, but, since we can safely think that the location of the stores will not change. If new stores will be launched or we want to check if OSM data changed than human intervention could be even useful. Once we have the dataset processed, we can follow the exact same steps explained in the previous project. We want to stress out that the only differences among the creation of the feature stores in the two presented projects is the specific data manipulation, that is of course specific for each different dataset. We can see in Figure 6.6 the processed dataset. We did not remove the longitude and latitude features, even if they are not relevant in a machine learning context, because in a real-world case, they can be useful to other team of people.

	store	latitude	longitude	amenity_count	tourism_count
0	12	51.206599	-1.596419	10	1
1	13	51.118627	-2.967112	3	0
2	14	52.191839	-1.415261	1	0
3	15	51.356004	0.670460	20	0
4	16	51.022622	0.912628	0	0

Figure 6.6: The dataset ready to be fed into the feature store. For each store we will have the features latitude longitude and the count of amenity and tourism POI places nearby

6.2.3 Model build pipeline

At this point, we already know the theoretical points this pipeline address. As said in the introduction of this section, this projects has been also chosen to experiment and study how machine and deep learning models tackle the problem of time series forecasting.

Before going into the details of our final implementation, we want to make a little

digression about the models tried, and our final choice. The research in this field has gone really fast in the latest times, and the state-of-the-art in the context of forecasting time series is now carried out with deep learning models. Among the trials, we report here two models that have attracted our focus:

- DeepAR: the authors of this work propose a methodology for producing accurate probabilistic forecasts, based on training an auto-regressive recurrent network model on a large number of related time series. They demonstrate how by applying deep learning techniques to forecasting, one can overcome many of the challenges faced by widely-used classical approaches to the problem [19].
- Temporal Fusion Transformers: the authors of this work exploit cutting-edge methods to build a powerful and interpretable deep learning model. In particular, the temporal fusion transformer (TFT) is an attentionbased architecture which combines high-performance multi-horizon forecasting with interpretable insights into temporal dynamics. To learn temporal relationships at different scales, TFT uses recurrent layers for local processing and interpretable self-attention layers for long-term dependencies [20].

Even if the scope of this work is not to find the best model for the specific chosen dataset, we carried out some experiments using both models. They are the state-of-the-art in this context, and we had some excited moments tried them. At the end, we decided to go with the former, DeepAR, since it is implemented in the `gluonts` library, that better integrates in our framework, but more importantly because it better models the relations between input time series given. Indeed, we have for each combination of item and store a different time series, that of course are related.
Once chosen the algorithm we explain the details of the steps of this pipeline. We highlight that for a demonstrative purpose the steps chosen do not change with respect to the first project presented, regardless of the context.
Without repeating the generics, the steps in details of this pipeline are the following:

- Preprocessing step: this step takes as input the entire training dataset, the preprocessing code, and infrastructure details.
  A dataset can be usually enriched, we already built a feature group to integrate

geographical information to reach better performances. Another feature that might be useful in our context is a simple binary column that indicates whether a day is an holiday. For the majority of shops, holidays correspond income increase. We can manage to add this feature in the preprocessing step, even if we could create an apposite feature group, but since standard libraries provide this information, we avoid to waste time and resources. Now, we process our dataset to be easily ingested by DeepAR, implemented by the library `gluonts`. Indeed, this model requires to have separate time series for each available feature. First of all we manipulate data to have, for each feature, on the row the unique combination of item and product (because they represent a unique time series) and on the columns the date. The unique key, is though to be easily retrieved also during the serving phase, and it is the combination of the item key and the store key. Also the target feature, the quantity sold, is preprocessed in the same way. Afterwords we split the dataset in two different subset, training and test set. The test set will contain the last 30 days, to confront it with prediction values.

- Training step: the aim of this step is to generate a model artifact. This step takes as input the training set generated from the previous step, and the desired estimator, in this case DeepARestimator. This estimator accepts missing data, and this is the reason why we did not manage them in the preprocessing step.

store_item	2018-11-06	2018-11-07	2018-11-08	2018-11-09	2018-11-10	...	2018-12-21	2018-12-22	2018-12-23	2018-12-24	2018-12-26
16_410	70.0	88.0	95.0	105.0	68.0	...	91.0	75.0	17.0	NaN	69.0
16_415	97.0	116.0	137.0	143.0	97.0	...	154.0	85.0	38.0	NaN	87.0
16_416	246.0	274.0	345.0	377.0	305.0	...	303.0	217.0	70.0	NaN	178.0
16_419	91.0	97.0	116.0	136.0	114.0	...	154.0	80.0	30.0	NaN	84.0
16_420	87.0	113.0	120.0	125.0	92.0	...	128.0	74.0	18.0	NaN	81.0
...	...	...	...	...	...	...	...	...	...	...	...
77_526	102.0	90.0	121.0	125.0	104.0	...	196.0	130.0	120.0	47.0	97.0
77_527	25.0	23.0	22.0	27.0	29.0	...	22.0	22.0	33.0	18.0	25.0
77_528	87.0	93.0	110.0	115.0	82.0	...	178.0	101.0	82.0	21.0	86.0
77_539	53.0	56.0	73.0	63.0	50.0	...	121.0	73.0	80.0	41.0	65.0
77_561	13.0	18.0	11.0	27.0	21.0	...	21.0	28.0	24.0	11.0	17.0

Figure 6.7: Pandas DataFrame containing the target feature, the quantity sold, in the format presented. On the columns we have the date while on the rows we have the unique key in the format store_item. All the other features, used to train the model, are processed to obtain the same format.

- Evaluation step: this step receive as input the model artifact generated in the previous step, the test set and the resource and infrastructure information, along with a python script that contains instruction to evaluate the model artifact generated in the previous step. In this step we forecast the 30 days of the test set. We calculate the performance of our model with three metrics: mean square error, absolute error, mean absolute percentage error. Within this step we create an evaluation report containing the values of these three metrics, in the form of a python dictionary, and save it to S3.
- Condition check step: this step reads the output of the previous step, and compare it to a threshold we define apriori. Also in this project we chose to set a threshold that would be easily met every time this pipeline is executed, since we do not care about the actual performance. This step accepts two hyperparameters, respectively "if_step" and "else_step": it means that if the condition is met, the specified argument in the "if_step" (e.g. another step) is executed, otherwise the "else_step". In the former step we specified the registration step, while in the else step we do not specify nothing, since the if step is always met in our setting.
- Registration step: this step loads the artifact in the appropriate model registry. If the model registry does not exists it will be created. As explained in the first project, the model registry assigns the version number to the artifact that is being registered. Of course, the first artifact has the version number 1, the second the version number 2 and so on.

Once an artifact is registered in its model registry a manual step is required to trigger the execution of the model deploy pipeline.

6.2.4 Model deploy pipeline

The strength of our work rely also in its reusability. This pipeline indeed is exactly the same used in the first project. The only differences are the test.py script, that is tailored for each different project to test the endpoint produced, and the resources specified, that are simply parameters to fill. To not repeat ourselves we do not explain

what already said in Section 6.1.4, because, again, we would only explain the same process twice.
Moreover, we specify that the resources employed corresponds to the ones used in the first project, since we do not need a lot of computational power.

API

Again, the logic is the same with respect to the previous project also for the gateway to let users employ the model. The lambda function that manages the raw data to be processed, uses the online store of the feature group created during serving time to retrieve geographical information, and process the data to be concordant with the training data. The user has to provide the time series he/she wants to predict (e.g. the store and item) along with all the features for each date he/she wants to get predicted. We configured the lambda to return the median value of the predictions of the correspondents number of input days. It means that the endpoint will return precise predictions, without confidence intervals. This can be improved, since in this context can be really helpful to visualize results with dashboards. We will talk about it later in Section 7.1.

6.2.5 Monitoring

Regarding the monitoring context, there are some changes with respect to the binary classification project. First of all, the features of the data are all static features, except for the price. It means that there can not be data drift for all the features except for the price (we assume that the category of an item will not change over time, if it changes it is only because the user make a mistake while fed data to the model). Moreover, data to be predicted are usually data "thought" by the user, because they are guesses on the future's price. Let's make an example. We deploy a model to production trained on the time series having as last date today. If a user wants to employ the model to make predictions, the data he/she will provide to the model are his/her guesses to the price, and it is very likely that the user will employ the model to make several forecasting while changing the price, to take business decision with

the support of the model. Hence, we conclude that the data monitoring in this specific context does not make sense to be implemented.
Whereas, the model monitor pipeline make a lot of sense, because we want to monitor whether our pipeline is supporting business decision in the correct way. This is a very difficult task to achieve, because there is no ground truth baseline. We only have the time series trends. The only thing that we were able to correctly implement is to daily check whether model predictions respects some chosen threshold when compared to the real value of the day considered. It is similar to the condition check step of the model build pipeline.
Still, we also believe that in this context, the concept of CT is required and it is more that in any other context, because the factors that affect the sales trend can change daily. That is also why we believe that in the TS context enriching the raw dataset is the key to succeed, and MLOps can really help to achieve both requirements.
Without adding a paragraph to explain the tests and their results, we only mention that they behave as expected, also for this project.

Chapter 7

Discussion and conclusion

This Chapter aims to recap the main findings discussed through the thesis, possible further developments, and conclude the work. The first part of this study focused on the differences that characterize ML systems with respect to traditional ones, and the main challenges that real-world ML systems face when bringing ML models to production. In particular we study the ML-specific challenges that needs to be taken into consideration when planning the practices to implement which can overcome them. We show the state of machine learning in production, and how the focus of major companies is flowing into the concepts related to our work. We make use of some tools that, in conjunction with the concepts presented in Chapter 4 allowed us to present a framework that overcome the challenges that ML architectures without MLOps practices implemented encounter. The tools and technologies used within this MLOps architecture are all based on AWS. This choice avoid us to do additional work when dealing with infrastructural settings, and speed up our implementation process. Our framework can be easily employed in different projects without much effort, since it is developed with the objective of being task unrelated. The main characteristics of this framework are: reliability, modularity and fastness. Indeed, the infrastructural backbone introduces CI/CD/CT pipelines, that speed up the deployment process with high automation, but at the same time we allow modular steps to meet different projects need. Among the various concepts, in this discussion we highlight the usage of the feature store and monitoring jobs that push our work to be really competitive in the MLOps framework market.

7.1 Future developments

The presented framework represents a reliable structure, although it can be enriched. The first improvement we will implement is the usage of dashboards, to visualize results. For example, in the second project presented (Section 6.2), we consider more appropriate to provide users with a dashboard to visualize the predictions, and it will be implemented when future customers will use our product.
There are some other competitive tools to emulate what we presented. We made our infrastructural choices based on the team's experience and requirements, but another possible development would be to use other platforms, maybe open source. Indeed, it is also possible to integrate open source tools in our work, and we are already considering that as a future development, to make our product more accessible.

7.2 Conclusion

At the start of this thesis, we set ourselves the objective of streamlining machine learning to production with an innovative framework that can be make the machine learning development process more reliable, scalable, secure and traceable. We proposed a framework and a workflow strategy that fulfilled our objective, using cutting-edge MLOps practices using one of the best platform such as AWS. A standard ML life-cycle has been established. Now we have an MLOps architecture and workflow strategy that will help us to manage our ML development and deployment process in future projects, regardless of the specific task. Data, ML artefacts life-cycle, and ML experiment details can be tracked and saved, no more than necessary manual handover is needed when deploying models into production.

List of Figures

Bibliography

[1] D. Sculley et al. Hidden Technical Debt in Machine Learning Systems. *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, Cambridge, MA, USA, 2015. MIT Press.

[2] Y. Zhao. Machine Learning in Production: A Literature Review. *https://staff.fnwi.uva.nl/a.s.z.belloum/LiteratureStudies/Reports/2021-LiteratureStudy-report-Yizhen.pdf*, March 2021.

[3] A. Paleyes et al. Challenges in Deploying Machine Learning: a Survey of Case Studies. *arXiv e-prints*, November 2020.

[4] L. Patruno. 5 Challenges to Running Machine Learning Systems in Production. *https://mlinproduction.com/5-challenges-to-ml-in-production-solve-them-with-aws-Sagemaker/*, April 2020.

[5] A. Shah. Challenges Deploying Machine Learning Models to Production. *https://towardsdatascience.com/challenges-deploying-machine-learning-models-to-production-ded3f9009cb3* , June 2020

[6] I. Pölöskei. MLOps approach in the cloud-native data pipeline design. *Acta Technica Jaurinensis*, April 2021.

[7] C. Shi. Dynamic Causal Effects Evaluation in A/B Testing with a Reinforcement Learning Framework. *arXiv:2002.01711*, February 2020.

[8] Docker. Docker overview. *https://docs.docker.com/get-started/overview/*.

[9] Google Cloud. MLOps: Continuous delivery and automation pipelines in machine learning. *https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning*, January 2020.

[10] Y. Liu et al. Building A Platform for Machine Learning Operations from Open

Source Frameworks. *IFAC PapersOnLine 53-5 (2020) 704–709*, 2020.

[11] Valohai. Practical MLOps How to Get Ready for Production Models. *https://valohai.com/mlops-ebook/*, 2020.

[12] L. Orr. Managing ML Pipelines: Feature Stores and the Coming Wave of Embedding Ecosystems. *PVLDB, 14(12): 3178-3181, 2021.*, August 2021.

[13] E. Breck et al. The ML Test Score: A Rubric for ML Production Readiness and Technical Debt Reduction. *Proceedings of IEEE Big Data (2017)*, 2017

[14] M. Gokarna et al. DevOps A Historical Review and Future Works. *International Conference on Computing, Communication, and Intelligent Systems (ICCCIS)*, December 2020.

[15] Z. Wan et al. How does Machine Learning Change Software Development Practices?. IEEE Transactions on Software Engineering (Volume: 47, Issue: 9, Sept. 1 2021), August 2019.

[16] D. H. Wolpert et al. No Free Lunch Theorems for Optimization. *IEEE Transaction On Evolutionary Computation*, April 1997.

[17] Moro et al. A Data-Driven Approach to Predict the Success of Bank Telemarketing. *Decision Support Systems, Elsevier, 62:22-31*, June 2014

[18] OpenStreetMap Foundation. *https://wiki.osmfoundation.org/wiki/Main_Page*.

[19] D. Salinas et al. *arXiv:1704.04110*, February 2019.

[20] B. Lim et al. *arXiv:1912.09363* , September 2020.

[21] P.P. Ray. An Introduction to Dew Computing: Definition, Concept and Implications. *IEEE Journals Magazine*, November 2017.

[22] AWS. Monitor models for data and model quality, bias, and explainability. *https://docs.aws.amazon.com/Sagemaker/latest/dg/model-monitor.html*.

[23] AWS. Sagemaker Pipelines Overview. *https://docs.aws.amazon.com/Sagemaker/latest/dg/build-and-manage-pipeline.html*.

[24] H. Devanathan. AWS Codepipeline. *https://towardsdatascience.com/tagged/aws-codepipeline?p=2c250ae5006a*