Politecnico di Torino

Master's Degree in Data science and Engineering

Evaluation of the effectiveness of adopting MLops practices in an industrial context, the problem of concept drift and conformity with the new AI Act

Supervisors

Prof. Antonio VETRÒ

Candidate

Nicolò VERGARO

October 2023

Summary

This thesis project is about applying MLOps practices in an industrial environment. MLOps refers to the intersection of DevOps with Machine Learning through practices that aim at automating and handling the processes related to the life cycle of an ML application. These processes include developing, testing, deploying, and monitoring a machine learning model.

The focus of the thesis is not only MLOps but also two other important aspects that can be linked with MLOps and have become more important lately, especially the latter, which is highly recent. The first aspect is related to concept drift, which is a shift in the statistical properties of data while the model is in production. Concept drift has become more prominent, especially given the dynamic and fast-paced environment in which we generate data at an increasing rate with time. The second aspect concerns fairness, which refers to avoiding creating a machine learning system that embeds biases and discriminates in its predictions. A significant step towards fairness is the new AI Act, created by the European Union to be the first regulation concerning AI in the world. When enforced, it will separate AI applications into three levels based on the risk they pose to users, each of which will have consequences on the company that produces the application and on the application itself.

The proposed solution will address these two aspects while enforcing MLOPs practices by developing a pipeline on the partner platform Amazon Web Services (AWS). The pipeline will automatically handle the end-to-end lifecycle of an ML application, which starts from the ingestion of the dataset and ends with the deployment of the trained model on an endpoint where the end user can make predictions while continuously monitoring the model for drift. We will see how the solution brings value to potential clients and simultaneously considers concept drift and respects fairness.

Table of Contents

List of Figures

Acronyms

AI

Artificial Intelligence

MLOps

Machine Learning operations

AWS

Amazon Web Services

DAG

Directed acyclic graph

S3

Simple Storage Service

EKS

Elastic Kubernetes Service

ECR

Elastic Container Registry

SNS

Simple Notification Service

ARN

Amazon Resource Name

IAM

Identity and Access Management

Chapter 1

Introduction

1.1 About the company

This thesis project was performed during my employment as an intern at Data Reply, the Reply group company which offers a broad range of advanced analytics and AI-powered data services across different industries and business functions. The Reply group was founded in 1996 in Turin and has since grown into a multinational company. Each of the Reply's companies is also divided into Business Units, independent divisions of the company responsible for their profits and losses.

The goal of Data Reply is to achieve meaningful outcomes through the effective use of data. The company partners with Amazon Web Services, so it relies on AWS infrastructure to build Big Data platforms and implement ML and AI models, allowing clients to exploit the full potential of their data. The partnership with AWS also allows Data Reply access to the latest technologies and resources developed by Amazon, allowing them to iterate faster and incorporate emerging technologies into their products. The result of this project is offered as a service on AWS, namely SageMaker pipelines, which allowed us to have the foundation on which to develop our solution.

Even if the main focus of the company is doing consulting work for external clients, Data Reply has also begun lately to develop the so called" offerings", which are projects on new technologies that are developed internally, as a proof of concept, by a team of employees that work in the same business unit. The offering is developed over several months, and if it leads to satisfactory results, it is pitched and eventually sold to potential clients.

1.2 About the project

This project is an offering in Business Unit 3, Enterprise and Analytics (ENA), about **MLOps**. It aims to describe an MLOps solution that orchestrates the entire life cycle of an ML application. In order to do so, it leverages DevOps practices for the Machine Learning world to automate and simplify development, testing, training, deployment and continuous monitoring of the models created by Data Scientists. Furthermore, it focuses also on what revolves around MLOps and is becoming increasingly more important, namely **concept drift** and the upcoming EU regulation of AI, the **AI Act**.

1.3 Motivations of the work

This project was assigned by management to the group of employees I worked with to allow them to gain experience on different projects from those they were working on with clients and on relatively new themes. These topics are chosen considering what clients could find interesting and what they could buy in the future, along with the fact that there are not many projects about this topic. This topic, MLOps, meets all these criteria; therefore, it was structured into an offering.

What makes this project interesting is that given that machine learning systems are becoming increasingly prominent in many industries, there is a need for tools to standardize and ease the production of these systems, from data ingestion to model deployment and subsequent monitoring.

The project also focuses on the challenges that have arisen, especially in the last times. The first is that, given the dynamic nature of data, ML models in production are at risk of suffering from drift, so there is a need to keep these models updated or at least be aware when drift happens. The second is that with the upcoming AI regulation (AI Act), machine learning applications categorized as high risk (trivially, a system that classifies people as worthy of a loan are high risk) must produce a self assessment declaring that the system is fair and does not discriminate.

After developing this offer, Data Reply can sell the complete MLOps pipeline to customers, which can obtain the advantage of plugging in their database and, after minor tweaks to the configuration, obtain an endpoint which accepts data and makes predictions. Clients can save time and money and set up a machine learning system for their needs. The system that they will set up will also:

- Be compliant with the upcoming AI regulation since the system will compute fairness measures and produce a report which can be exported for later use. Furthermore, the system will automatically check if data is unbiased and, if it does not comply with specified thresholds, block the execution before the model is even trained.

- Be monitored after it is deployed to check if data drifts compared to the data it was trained on. The client can decide the frequency of the monitoring, set up notifications to be aware when drift happens, and set up the automatic retrain of the model after drift has been detected to update the model with new data

In conclusion, this project merged the commitment to innovation with anticipation of industry trends.

Chapter 2

Machine Learning in production

2.1 MLOps

Machine learning solutions are becoming increasingly spread across various industries, but adopting AI brings many challenges. As found by the IBM Global AI Adoption Index of 2022 [1], only 35% of companies have adopted AI, while an additional 42% is still exploring this possibility but facing many challenges such as insufficient expertise or knowledge on AI, price is too high or lack of tools to develop AI solutions. Furthermore, when a data science project is started, the path to its completion is full of dangers. As stated by [2], some reasons a data science project may fail are:

- Poor deployment planning: Usually, companies start data science projects because they want to do it; they do not have a clear goal in mind. Even when companies have a clear goal, they often start with a basic proof of concept that evolves into the final model to use, but they did not think about the infrastructure to move the model into production. What happens in those cases is that when predictions are needed, the data scientist manually calls the model from the testing environment.
- Poor maintenance planning: Since companies usually want to "move fast", they skip maintenance decisions, such as tracking experiments, versioning models or foreseeing model monitoring, in favour of obtaining the final model as soon as possible. These practices make maintaining these models over time difficult and expensive, accumulating technical debt [3].

When a company decides to adopt AI, even if it can thoroughly plan and avoid generating technical debt, it still has to embark on a very long journey composed of many phases that start with data ingestion and are completed with model monitoring. It does not finish there, as these steps should be executed again periodically or when drift is detected. Figure 2.1 shows an example of such steps.

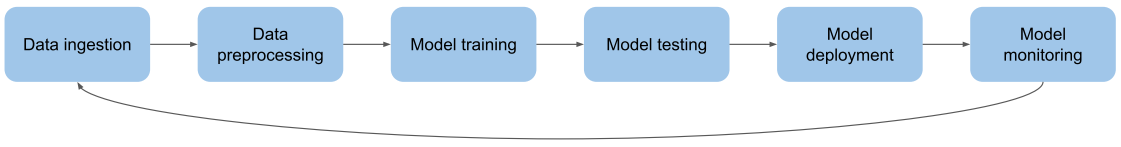


Figure 2.1: An example of operations to perform to have an ML model in production

To remove all the hassle and automate this process, we can leverage MLOps, which stands for Machine Learning Operations. As stated in [4], MLOps is the intersection between machine learning and DevOps, specifically, applying DevOps practices to the machine learning life cycle. DevOps practices [5] aim at minimizing the time from the inception of a software requirement to when it is deployed and available to the customer by focusing on Continuous Integration (CI) and Continuous Delivery (CD). Continuous Integration means that developers of different teams should frequently integrate their code, which should also happen with other applications and services. Continuous Delivery means that newly developed features should get to the customer quickly. A new paradigm for machine learning was necessary, as ML applications are not only code but belong to a broader system that also includes data which is not static but is subject to change. If the change happens after the application goes into production, it will produce wrong results. For this reason, a new component was introduced alongside CI and CD: CT (Continuous Training) [6]. Continuous Training refers to automatically retraining and serving the models. CT uses collected feedback and production data to keep the model up to date.

To sum up, implementing an ML solution does not only mean writing code to train a model, but many more operations should be performed before and after this step. Many of these operations are complex and require a specific infrastructure to support the development and deployment of the solution. By adopting MLOps practices, companies can create pipelines that can automatize and standardize the end-to-end life cycle of an ML application, from data ingestion to model deployment and monitoring. Maintaining high performances is easier as models are continuously monitored and retrained when needed. Furthermore, MLOps helps lower technical debt by tracking and versioning models, code, hyperparameters, experiments, logs, and so on. Development and maintenance processes are streamlined and more efficient, resulting in teams being able to focus on new technologies and creating new models quickly [7].

2.2 Drift

After talking about MLOps, the next important topic is drift. It needs to be addressed as, in the last years, it has been estimated that the amount of data produced is in the order of the quintillion of bytes every day [8]. Many of the data sources are becoming faster, for example, sensors that collect and stream large amounts of data in short periods; in fact, one of the characteristics of this kind of data (Big Data) is velocity [9], along with volume and variety (3 V's of BigData).

It is often assumed that the process that generates this data is stationary. However, in many scenarios, the process is non-stationary, and data is affected by variations that change the statistical properties of data. Some proposed that we add at least two more V's to the definition of Big Data, one of which is Variability, which means that data structure and how people interpret it may change during time [10]. From these non-stationary processes we derive the idea of concept drift [11].

Concept drift is a phenomenon in which the statistic properties of the target variable (what the model predicts, the label) change arbitrarily during time, so the representation of data that was learnt by the model using data of past events may not be relevant for new events, resulting in the degraded performances of the model itself [12].

Concept drift is a very modern concept, given that we live in a fast evolving world where we produce large amounts of data. An example of an application that can be affected by drift is a recommender news system for a particular individual. What he finds interesting at a certain time might not be interesting after some time. We will see this specific example more in-depth later.

Following the notation introduced by [13], we can define $P_t(X, y)$ as the joint distribution, at time t, of the set of the input variables X and the target variable y. Changes in the data are caused by changes in the components of this relationship. We can write:

$$P_t(X, y) = P_t(X)P_t(y|X)$$

Where $P_t(X)$ is the marginal probability of X, which represents the probability of observing data without taking into consideration the label; instead, with $P_t(y|X)$ we refer to the conditional probability of y given X, which represents the probabilities of the labels y given that we observed X: this can also be interpreted as the *decision boundary* learnt by our model. Formally, we observe the phenomenon of drift if:

$$\exists t : P_t(X, y) \neq P_{t+1}(X, y)$$

We can analyze concept drift by analyzing the changes in the components:

- **Real** concept drift: $P_t(y|X) \neq P_{t+1}(y|X)$, while $P(X)$ does not change. In this case, we have the mutation of the decision boundary, while the probability of observing data does not change.

- **Virtual** concept drift: $P_t(X) \neq P_{t+1}(X)$, while $P(y|X)$ does not change. In this case, we have a variation of the input data, while the decision boundary does not change.

We can observe the two types of drift in Figure 2.2.

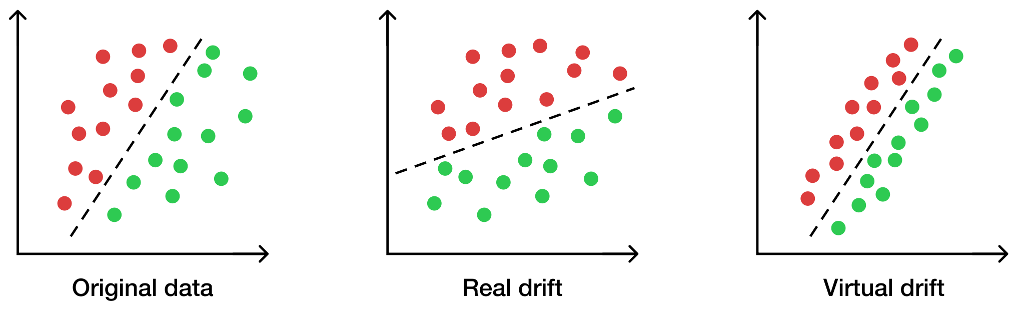


Figure 2.2: Types of concept drift

To better understand the difference between virtual and real drift, we can look at an example from [13]: Given a website on news about real estate, the task of the user is to classify the news as relevant or not relevant. If, at that moment, the user wants to buy a house, news about houses on sale will be relevant, while news about holiday houses will be irrelevant. If the editor of the news were to change, the style of the news might change too, but the user would still find relevant only news about houses on sale: this represents **virtual drift** ($P(X)$ changes, but not $P(y|X)$). If the user decides he does not want to buy a house anymore but prefers to go on holiday, the news will stay the same. However, the interests of the user changed, as he will find news about holiday houses relevant, while news about houses on sale will become irrelevant: this represents **real drift** ($P(y|X)$ changes, but not $P(X)$).

Furthermore, we can notice that the transition of data distribution can be of four different types [12]:

- Sudden: For example, changing a sensor with another with a different calibration.

- Incremental: For example, a sensor that degrades and slowly loses its precision.

- Gradual: This kind of drift is a variation of incremental drift. An example is a user who starts to find news about holiday houses more relevant than news about houses on sale but still finds relatively relevant news about houses on sale; he eventually loses interest in the latter.
- Reoccurring concept: We have this kind of drift when an old concept returns after a certain time. This type of drift can be associated with cyclical phenomena like seasons or irregular ones like inflation or market mood. Another example is Black Friday, which occurs regularly once per year [14].

We can observe the different types of drift in Figure 2.3.

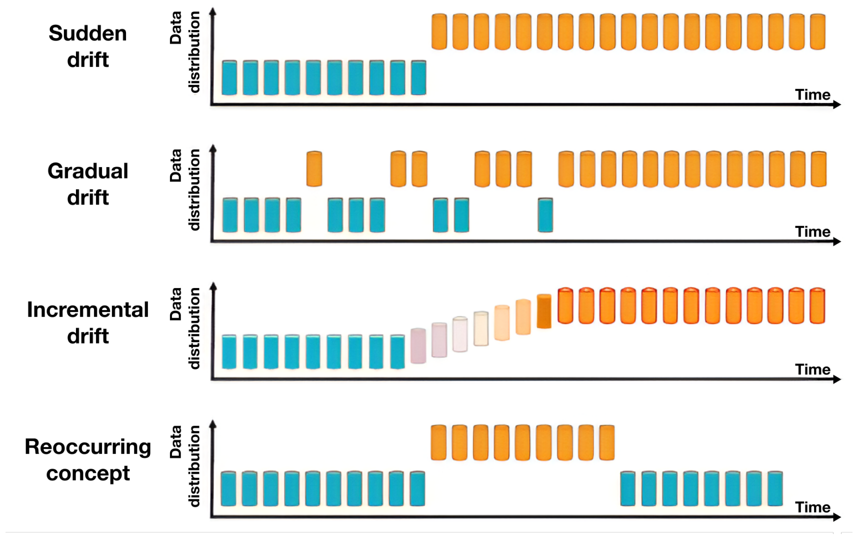


Figure 2.3: Types of concept drift, source [15]

To adapt to concept drift, we can decide to adopt two approaches, active or passive[11]:

- Active: We try to detect drift and adapt to the situation, for example, by retraining the model or understanding if there are errors in the data.
- Passive: The model is updated continuously, regardless of whether drift is present.

Passive approaches are effective with gradual or recurring drift, while active approaches are useful for sudden drift.

There are many algorithms to detect drift, but in this project, we focused on **data distribution-based drift detectors** since it is what AWS SageMaker gives us by default. With these kinds of methods, we monitor the distance between two data distributions. Section 4.9 will discuss more implementation details.

2.3 FairML

As was repeatedly said, machine learning solutions are rapidly being used in many industries and for different applications. As machine learning algorithms improve in accuracy, many decisions become automated while human feedback becomes less valuable; as [16] shows, humans are more likely to overestimate the credibility of an algorithm and follow its decision even if it is incorrect and contradicting, with the side effect of providing a false sense of security that a human has overseen the decision. The problem with these practices arises when an algorithm produces unfair and biased decisions due to the unfair and biased training data.

For these reasons, research on fairness of machine learning has become a widely popular field, with many conferences started and many companies looking into this matter. It is crucial, given that automated decisions are taken in many fields nowadays, from harmless movie recommendations to more critical ones, for example, to decide whether a defendant is at risk of becoming a recidivist [17]. A wrong suggestion about a movie has no critical effects, as we can stop playing it; being labelled with a high risk of recidivism might limit your liberty [18].

Given all the previously cited problems with machine learning systems in the scope of fairness, the EU started working towards the world's first rules on AI, the *AI Act*, which was approved by the EU Parliament on June 14, 2023 [19]. The following steps involve talks with EU countries in the Council to decide the final form of the law, aiming at an agreement by the end of 2023. It brings a classification of AI systems based on the severity of the risk they pose to users. While there is a paragraph on the rapidly growing generative AIs, it also indicates three different levels of risk, which have different implications:

- Unacceptable risk: These applications are considered a threat to people and will be banned. Examples are estimations of the reliability of people (credit score) or systems that use real-time biometric identification for monitoring purposes.
- High risk: These applications negatively affect safety or fundamental rights. An example is scanning a candidate's CV to assign a score. They are divided into two main categories: The first includes applications already governed by existing product safety legislation (e.g., medical devices) that must conform to existing legal frameworks. The other category includes eight areas such as:
  - Employment, worker management and access to self-employment
  - Migration, asylum and border control management
  - Biometric identification and categorization of natural persons

  These systems have to present a self assessment before being put on the market and will also be assessed while on the market.

- Limited risk: These applications should comply with minimal transparency requirements to allow users to make informed decisions and decide whether to continue using the system. Users should be made aware that they are interacting with AI.

Given the recent approval of the AI Act, it would be interesting to link it with MLOps practices. For our project, we used a test dataset that is perfect for this case and can be used for real applications: fraud detection on loan requests. This dataset is "high risk", following the AI act definition, given that we use it to perform "categorization of natural persons"; furthermore, in the previous draft, it was explicitly considered high risk for "creditworthiness assessment", which is precisely our case. For these cases, it is necessary by law (when it will be approved in its final form plus within two more years for the law to enter into force) that the company developing the application presents a self-assessment stating that the application that uses AI is safe. For this reason, we can include an innovative function in our MLOps pipeline that computes fairness measures on the training dataset and stops the execution if some criteria are not satisfied. Otherwise, we can conclude that the data is safe. Furthermore, the AI act also has more links with MLOps, given that by using MLOps practices, we enforce traceability, version control (of model and data) and reproducibility, all of which improve transparency and accountability.

Talking instead about drift, we can detect drift in data with model monitoring; thus, we can notice when statistical properties of data change (and consequently, the performances of the model can degrade), restart the pipeline and compute fairness metrics again. This could make us aware that new bias in the data has arisen, also improving transparency; it also has the positive effect of updating the model with new data, preventing model performances from degrading substantially, and potentially taking decisions that can have a negative impact. Finally, drift can also be seen as "evaluation bias" as a cause of fairness problems. We have evaluation bias when evaluation data do not represent the target population well or when training and evaluation/operation data differ.

In conclusion, we saw how the importance of fairness in machine learning has grown, as ML systems are becoming widespread and automated decisions about critical topics can be unfair and influenced by biases in the data. The increasing research and the approval of the EU's AI Act can be seen as the need for transparency, accountability and awareness in this domain, with one of the most important examples being generative AI.

Given the existence of MLOps, it becomes clear that these practices will play a crucial role in ensuring compliance and fairness as it becomes easier to ensure transparency or integrate fairness checks. Moreover, it helps address drift, simultaneously preserving fairness and the model's performance.

The convergence of these two aspects will help create a framework that prioritizes ethics, transparency and accountability while adapting to the changing environment.

Chapter 3

Project

3.1 Overview of the project

In this section, we take a high-level look at the project by first introducing the provider on which we based the solution, Amazon Web Services, citing the various services we used to implement the pipeline. It is worth noting that to perform machine learning operations, the most used AWS service is SageMaker, which is a fully managed ML service. Among the various services that SageMaker offers, we have, for example:

- Services used to perform rapid data processing (ProcessingJob), model training (TrainingJob), model deployment (CreateEndpoint), monitoring, and other ML related operations. Moreover, we can use SageMaker's UI to inspect these processes; for example, we can check when they began, how long the processing took, inputs, outputs, and logs.
- The tool *SageMaker Model Building Pipelines* (from now on, we will refer to it as just *SageMaker Pipelines*), through which we can orchestrate ML pipelines. (we will see a more in-depth description in Section 3.2)
- The IDE SageMaker Studio is a web IDE based on Jupyter Lab. It allows the user to access many tools to manage the different phases of developing a machine learning model, such as Jupyter notebooks, terminals, sections to check the model registry and endpoints, etc. SageMaker Studio also allows us to analyze the pipelines being executed or executed in the past. We can examine the pipeline visually by analyzing its DAG (directed acyclic graph, a graph that represents the steps and how they are linked with each other) as we can see in Figure 3.1; this tool also allows us to inspect parameters, inputs, outputs and logs of the various steps and of the pipeline itself, for example we can see in Figure 3.2 some parameters of the training step.

- Pre-built Docker images based on some of the most common and used ML frameworks, such as scikit-learn, PyTorch, and TensorFlow. These images also include some standard libraries usually used in these scenarios, like Pandas and NumPy. SageMaker also makes available containers for some built-in algorithms like XGBoost, which we can use for training a model without further configurations. Furthermore, we can install more libraries if necessary for the specific use case; the only requirement is that they are on PyPi.

- SageMaker Python SDK, thanks to which we can use SageMaker services in Python, for example, to define a pipeline and its components, handle monitoring and experiment tracking, etc. The pipeline that we built was developed using this SDK

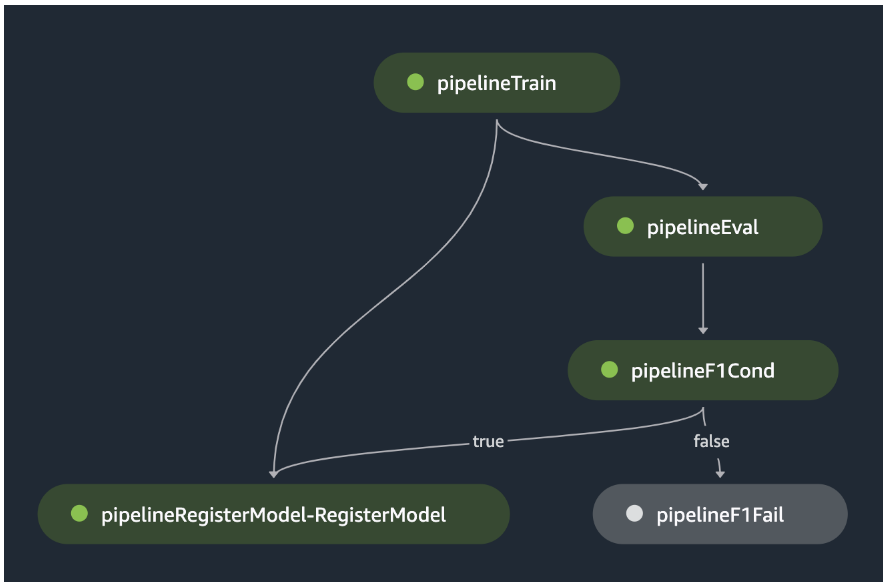
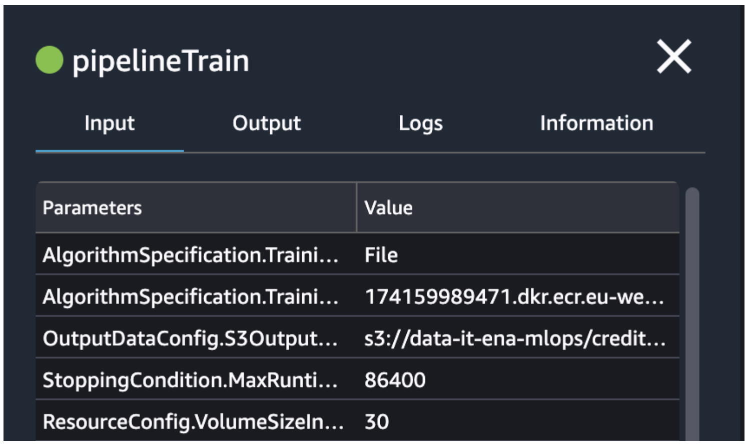


Figure 3.1: An example of the DAG of a pipeline

To implement MLOps practices, we used the tool SageMaker Pipelines, with which we were able to create a smooth automated process that starts from data ingestion and ends at model deployment; the process continues under the hood by monitoring the model to detect any drift, and, if detected, sends a notification to alarm the user and starts the pipeline again, with updated data.

To create and orchestrate pipelines on Amazon Web Services, we have multiple choices. Among the various approaches, we explored Amazon SageMaker Pipelines and KubeFlow. We decided to explore these two solutions since my coworkers already had some experience with KubeFlow, while SageMaker Pipelines is the native solution of the provider AWS. After having identified these two techniques,



Parameters	Value
AlgorithmSpecification.Traini...	File
AlgorithmSpecification.Traini...	174159989471.dkr.ecr.eu-we...
OutputDataConfig.S3Output...	s3://data-it-ena-mlops/credit...
StoppingCondition.MaxRunti...	86400
ResourceConfig.VolumeSizeIn...	30

Figure 3.2: An example of the metadata for a training step

we carried out a preliminary research phase, which allowed us to have a clear picture of both solutions and understand the pros and cons of using a native solution versus using an external framework. In the following sections, we will describe in more detail these two approaches, and eventually, we will discuss the choice made.

3.2 SageMaker pipelines

SageMaker Model Building Pipelines is a tool for building machine learning pipelines that take advantage of direct integration with Amazon SageMaker. SageMaker Pipelines represent the native solution on Amazon Web Services to manage the life cycles of machine learning models. Since this tool is also integrated with SageMaker Python SDK, we can create our pipelines using Python.

SageMaker pipelines allow us to create steps and manage them easily and efficiently. Since it is a fully managed service, we can orchestrate the pipeline at a high level without worrying about handling resources, for example, the underlying physical machines, since SageMaker creates and handles the resources for us. Furthermore, as we said before, this tool is integrated with SageMaker Studio, allowing us to write code, test it, and analyze the results without leaving the environment.

3.2.1 The pipeline object

SageMaker Pipelines supports different types of steps, which describe the actions that the pipeline takes and the relationships between steps. To define the steps the pipeline will execute, we use an object of the class *Pipeline*. We define a Pipeline object through its name, parameters and steps. The name of the pipeline must be unique inside the `(account, region)` pair. A basic example of the Pipeline object is:

```
pipeline = Pipeline(
        name=<pipeline-name>",
        parameters=[parameters],
        steps=[step_preprocessing, step_train, step_eval, ...],
    )
```

A pipeline can be parameterized by introducing variables whose value can be accessed at runtime, i.e. during the pipeline execution. SageMaker Pipelines support the following parameter types: `ParameterString`, `ParameterInteger`, `ParameterFloat`, `ParameterBoolean`. Examples of parameters are the kind of machine to use for inference or the number of instances of the machine to allocate:

```
instance_type = ParameterString(
        name= "InferenceInstanceType",
        default_value= "ml.m4.xlarge"
    )
```

All the parameters we reference when defining a step must be defined when we create the Pipeline object. These parameters are defined with default values that must match the parameter type. The default value can be overridden when a pipeline is started using the method `start()` and defining a dictionary that contains the parameters we want to modify:

```
execution = pipeline.start(
    parameters=dict(
        ProcessingInstanceCount= "2",
        InferenceInstanceType= "ml.t3.medium"
    )
)
```

When the pipeline will be executed on a machine instance, it will interface with a new filesystem, which is the folder `/opt/ml`, in which there are subfolders for each step, for example, the training step accesses `/opt/ml/model`. In contrast, the processing step accesses `/opt/ml/processing`. What is important to understand is that to make each step communicate with the outside, for instance, with Amazon S3 [1], the files must go through this particular folder. For example, to allow the evaluation step (which is a processing step) to use the model, it must be moved from S3 to the folder `/opt/ml/processing`, while every output will be at first

[1]Simple Storage Service, an Amazon service that provides object storage service on the cloud

saved "locally" in `/opt/ml/processing` and then moved into the desired folder on S3. We will see this mechanism in more detail in the sections describing the various steps. More steps and components that we will need during the pipeline development will be explored in Section 3.6.

Another important aspect of the pipeline we cited before is the dependencies among the steps and how to specify which step comes after another. The order in which we specify the steps list in the pipeline definition is not important, as SageMaker Pipelines can automatically resolve the relationship between steps and determine the order of execution. SageMaker can handle dependencies for us thanks to a special attribute of each step called *properties*, which is resolved at runtime, for example, the S3 URI of the trained model obtained as output from the training step:

```
    train_step.properties.ModelArtifacts.S3ModelArtifacts
```

As we can see, to obtain the desired nested property, we use JsonPath [2]notation to traverse the JSON property object. Properties are used to create data dependencies between pipeline steps by passing the properties of a step as input for another step. SageMaker then uses these dependencies to create the DAG of the pipeline and make sure that a step that takes as input the property of another step is not started before the latter has finished its execution.

There is also the case when no properties are passed from one step to another, but we know that one step must necessarily start after another step has finished its execution. For these cases, we can also decide to specify custom dependencies. To define this kind of dependency we use the method `add_depends_on()`, for example:

```
training_step = TrainingStep(...)
processing_step = ProcessingStep(...)
training_step.add_depends_on([processing_step])
```

Adding custom dependencies must respect that the pipeline is a DAG, so creating cyclic dependencies is forbidden and will throw an exception.

3.3 KubeFlow

KubeFlow [20] is an open source platform created by Google based on Kubernetes [3], used to orchestrate ML pipelines. The project's goal is to make deployments of machine learning (ML) workflows on Kubernetes simple, portable and scalable. This

[2]JsonPath implementation

[3]open source platform created by Google to deploy, scale, and manage containerized applications

means that whenever we can run Kubernetes, we can run KubeFlow, thus effectively making KubeFlow platform-agnostic. One of the components of KubeFlow is KubeFlow Pipelines (KFP), a platform for building and deploying portable and scalable machine learning (ML) workflows using Docker containers. It allows us to write an ML pipeline in Python by writing entirely custom components or using pre-built ones and execute the pipeline on any KubeFlow Pipeline conformant backend based on Kubernetes or Google Cloud Platform. KubeFlow also allows us to interact with pipelines through its UI, which lets us see or start runs, explore the configuration, graph, and output of a pipeline run, compare the results of one or more runs, etc.

KubeFlow was taken into consideration for implementing a pipeline on AWS for two main reasons:

- We already had prior knowledge of KubeFlow to orchestrate a pipeline.
- It can be executed on AWS thanks to the service Amazon EKS (Elastic Kubernetes Service), which is a managed service that can be used to execute Kubernetes on AWS. By using EKS, we can also access other AWS services, such as S3 for storage.

The most important strength of choosing this solution is given by the **SageMaker components for KubeFlow** [21], which is a series of components created to integrate Amazon SageMaker with KubeFlow. Thanks to these components, we can exploit SageMaker's functionalities for managing the life cycle of an ML model, such as training or deploying a model on an endpoint, using KubeFlow to orchestrate the pipeline. This approach offers the possibility to use the managed infrastructure of SageMaker and its optimized algorithms while keeping the pipeline flexible and platform-agnostic. Furthermore, by abstracting the components from the platform, we can create reusable pipelines that can be executed independently from the underlying platform, for example, GCP or AWS.

3.3.1 SageMaker components for KubeFlow

Generally, a KubeFlow component is a step of the pipeline represented by a Python module built into a Docker image. When we run the pipeline, the container of the component is instantiated on one of the nodes of the Kubernetes cluster that runs KubeFlow; then the code is executed. Each component can generate outputs that are read by the following components. Using SageMaker components for KubeFlow instead, we do not encapsulate the logic of the component into a container. Instead, we create and monitor native SageMaker processes (such as training, tuning, or deploying to an endpoint) directly from the KubeFlow pipeline. In this case, each component invokes SageMaker jobs, through which we can leverage SageMaker's fully managed infrastructure. This also allows us to monitor processes (e.g., processing, training, or deployment) directly from the SageMaker UI without having to move to the KubeFlow UI.

The components come in two versions, v1 and v2; some come in both versions, others only in v1. As far as possible, it is best to use the latest available version of each component, as v2 leverages the SageMaker Operators for Kubernetes (ACK), which allows us to define and use AWS services from the Kubernetes cluster. Also, by deploying the "full" version of *KubeFlow on AWS*, the v2 components do not need any additional configuration to access SageMaker, as this configuration is part of the deployment itself.

In conclusion, when deciding to use KubeFlow and AWS as a cloud provider, it is recommended to use SageMaker components for KubeFlow, as they offer seamless integration with AWS SageMaker infrastructure while still orchestrating with KubeFlow.

3.3.2 KubeFlow deployment on AWS

The vanilla deployment option is the most straightforward way to deploy KubeFlow on AWS, with minimal changes and optimized for Amazon EKS. Before beginning with the deployment, we created an Ubuntu environment using AWS Cloud9, which is a cloud-based integrated development environment (IDE) that lets us write, run, and debug code within a browser; furthermore, it also includes a debugger and terminal; other than the code editor. After cloning the repository containing the necessary tools for the deployment, we had to configure some options like region, KubeFlow and AWS versions and install the necessary tools using the command `make` on the makefile `install-tools`.

After performing these operations, we proceeded to perform the deployment using Terraform, an infrastructure as code tool used to define both cloud and on-prem resources in human-readable configuration files that can be versioned, reused, and shared. It can create and manage resources on cloud platforms and other services through their application programming interfaces (APIs). This makes Terraform able to work with virtually any platform or service with an accessible API. It already supports many providers such as Amazon Web Services (AWS), Azure, Google Cloud Platform (GCP), etc. The workflow of Terraform consists of three main stages: first we **write**, in which we define the resources; second we **plan**, in which Terraform creates an execution plan in which it describes the step it will take based on the platform it is executed on; finally, we **apply**, in which Terraform performs the proposed operations in the correct order.

We chose to use Terraform because it automatically created a VPC, created an EKS cluster, and deployed the vanilla distribution of KubeFlow on AWS. We had to perform some of these manually with the other installation options, which would have added more overhead.

After manually defining variables like the cluster region and cluster name and saving these to a `.tfvars` file, we performed the command

```
terraform init && terraform plan
```

with which we initialized Terraform and executed the "plan" step we saw before. Finally, we used again the command `make` on the makefile called `deploy`, which contains the "apply" steps of Terraform.

After deploying KubeFlow, we just need to execute two more commands to connect to the KubeFlow UI, which consist of making a port forward through the commands:

```
$(terraform output -raw configure_kubectl)
make port-forward
```

After following the whole procedure, we can access KubeFlow UI on the built-in Cloud9 browser at `localhost:8080` by logging in with the default credentials email: user@example.com and password: 12341234.

3.4 Framework choice

After discussing both frameworks, we can analyze their advantages and disadvantages to make an informed choice. In our choice, we did not only consider the two solutions per se but also the resources made available to the project by the company.

Amazon SageMaker Pipelines is the fully-managed native AWS solution, so it is deeply integrated with the AWS environment; in fact, we can start using it right away with SageMaker Studio, which provides us with a development environment and tools to analyze the pipeline: all of this is given without requiring any additional configurations. The disadvantage is that it is not cloud agnostic, so it is not a portable solution should you decide to change cloud providers.

KubeFlow, on the other hand, is cloud agnostic, so there would be no need to completely rewrite the pipeline by changing cloud providers; it is also an open source project, very flexible, and on AWS it can leverage the SageMaker Components for KubeFlow, thus allowing you to use the AWS infrastructure to run the different steps. The major disadvantage is that it requires heavy configuration, management and maintenance, as well as specific skills to manage a Kubernetes cluster, as it is a custom solution not provided by AWS.

Although KubeFlow is a very viable option, based on the boundaries and purposes of the project, we decided to opt for SageMaker pipelines, accepting a compromise between the need for customization and convenience in management.

3.5 Versioning and deployment techniques

To guarantee adequate code versioning and ease the release management, we adopted GitHub as a code versioning system, as it is widespread across the organization, provides an easy web interface, and all of us had experience with it. Our GitHub repository was linked to CodeBuild, which is a fully managed continuous integration service that compiles source code, runs tests, and produces ready-to-deploy software packages.

We linked these two services using CodeBuild triggers, which constantly monitor push events in the GitHub repository. Triggers are configured to react exclusively to push events associated with tags that match a specific regular expression that we defined. In particular, we set up two triggers that have two different regex. The first captures all push actions on the main branch with a tag that starts with "release", followed by the name of the use case and the version number preceded by "-v"; an example could be `'release-use_case-v1.05'`. The second trigger instead has a similar structure but captures all push actions on any branch that starts with "feature/" and on any push that starts with "prerelease", then it includes the use case and version number as before. The two triggers are linked to the same `buildspec.yaml`, in which we describe the steps to execute before running the pipeline.

When one of the two triggers is triggered, the pipeline process is automatically started through CodeBuild, which guarantees scalability and flexibility, allowing us to execute build processes in parallel and manage the resources efficiently, thus respecting the scalability needs of the project. This is particularly handy in an ML project, where you should usually manage large quantities of data and computationally intensive processes, but more importantly, where based on the use case, architectural needs can vary broadly.

In particular, the steps executed when one of the two triggers is triggered are:

- Installation of a certain Python version (3.11), followed by the installation of the dependencies specified in the "requirements.txt" file. This is important to configure a coherent and reproducible environment for our pipeline.

- Export of three variables: the tag name, the use case and the version number. The last two variables are obtained by splitting on the right points the tag variable.

- Updates the Docker image of the training step by accessing the folder that contains the training script and Dockerfile. This is useful because the pipeline will always use the latest available training image, unless we specify another one in its parameters. We will talk about this more in-depth in the following sections.

- The pipeline is started using the use case extracted previously. This means that we can change which pipeline to launch just based on this parameter.

After completing these steps, the pipeline is running, and we can see its execution by going into the Pipeline section of SageMaker Studio. Our pipeline architecture (which we will see in the next sections) and the triggers mechanism guarantee an efficient and unified handling of the ML workflows, allowing the Data Scientist to focus on working on the code and formatting in the right way the tag associated with the push on the branch; what comes next is handled autonomously by the MLOps process.

3.6 SageMaker pipelines step description

Having chosen SageMaker Pipelines as the framework to orchestrate the pipeline, let us learn more about it by describing the different steps we are going to use, to get a clearer idea when we will describe the logical steps of the pipeline. Before proceeding with the description of the steps, we will talk about an important aspect of AWS in general, which we will name later during the description of the steps and in the next section: AWS roles and resources names (ARNs).

3.6.1 Amazon roles

To securely control access to AWS resources, we use AWS Identity and Access Management (IAM), a web service to centrally manage permissions to control which AWS resources users can access. An IAM identity represents a user or workload that can access AWS services and perform actions. An IAM role is an IAM identity which, instead of being associated with one person, is intended to be assumable by anyone who needs it. Each IAM identity can be associated with one or more policies that determine what actions a user, role, or user group member can perform, on which AWS resources, and under what conditions.

3.6.2 Amazon Resource Names

Amazon Resource Names (ARNs) uniquely identify AWS resources. ARNs are used when it is required to specify a resource unambiguously across all of AWS. They have a general format; in fact, they all start with `arn:` and are followed by different information, each separated by a colon, for example:

```
arn:aws:iam::174159989471:role/bu3-mlops-deploy-model-lambda-role
```

which represents a role that was created to execute a Lambda function. The number `174159989471` represents the account ID that owns the resource. We will also use an ARN, for example, when we need to restart the pipeline, to specify the resource to call. When we will see roles in later code, it refers to the ARN of the role.

3.6.3 ProcessingStep

The *ProcessingStep* is used to create a SageMaker *Processing Job*, which is useful to process data, e.g. perform data validation, data preprocessing or model validation. This step needs an object of type Processor with which we describe the characteristics of the processing job, such as the type of machine on which it will run (instance type), the number of these machines, the Docker image on which it is based, etc. To define the ProcessingStep, we also need to specify inputs and outputs (if any). In our case, we will use as a Processor an object that inherits from the Processor class called ScriptProcessor. The advantage of this type of Processor is that it allows us to provide a script to execute during the processing job and a command to execute the script (e.g. "python3"). Other typical Processors are, for example, SKLearnProcessor or PySpark processor, in which we do not need to pass the base image but just the framework version and the code, and it will automatically obtain the correct Docker image on which to execute the code. The mechanism that governs input and output allows the script that we define to be platform agnostic, using the `/opt/ml/processing` directory. Any input we want to use in the script must be moved from S3 to that folder, while any output generated by the script is initially saved in the folder and later moved to S3. In more detail, we describe input and output separately.

- input: Any input is passed as *ProcessingInput*. To define such an object, we specify the path to the folder where the file is located on S3 at run time via the `source` parameter; we also specify where it should be moved to be used by the script via the `destination` parameter.

- output: Any output generated by our script is defined as *ProcessingOutput*. We specify with the `source` parameter the path to the folder where the file from the script is saved; with the `destination` parameter, we instead define the destination of that output on S3, so where it will be moved on S3. We also have a third parameter, called `output_name`, by which we define a symbolic name for the output.

From a ProcessingStep, we can obtain as output many different types of files, for example, a file that represents the dataset after preprocessing or a file that contains the evaluation metrics of the model. This last example, in particular, can be saved as a **property file** which is generally used to store output information from the ProcessingStep. This method of saving outputs will be useful if we want to add a conditional step to decide which actions to execute based on the results stored inside this file. To generate this kind of file, we pass the parameter `property_files` to the ProcessingStep, which is an array of objects of type *PropertyFile*. This object is instantiated by passing a name, the path of the JSON file saved inside the script and finally, an `output_name`, which is the same we saw earlier when we defined the *ProcessingOutput*.

To instantiate the *ProcessingStep*, we define its name with the parameter `name` and its arguments through the parameter `step_args`. The latter is the result of the `run` method on the previously defined *Processor*, and it is with this method that we define inputs, outputs and the code that will be executed. An example of *ProcessingStep* is:

```
script_processor = ScriptProcessor(
            image_uri=<image>,
            role=<role>,
            instance_type=<instance_type>,
            instance_count=<instance_count>,
            sagemaker_session=<pipeline_session>)

step_args = script_processor.run(
    inputs=[
        ProcessingInput(
            source=<input_data>,
            destination="/opt/ml/processing/input"
            )
        ],
    outputs=[
        ProcessingOutput(
            output_name= "train",
            source="/opt/ml/processing/train",
            destination=<path_on_s3>
            )
        ],
    code= "process.py"
)

step_process = ProcessingStep(
    name=<step_name>,
    step_args=step_args,
)
```

3.6.4 TrainingStep

The *TrainingStep* is used to create a SageMaker *Training Job*, which is useful for model training. To define the *TrainingStep*, we need input data and an object of type *Estimator*, which is used to describe the characteristics of the training job, such as the Docker image on which the training will run, the role, hyperparameters, etc. The output of this step is an object called `model.tar.gz`, which contains all the outputs of the step, in our case, the trained model, saved as a pickle or joblib file.

Inputs are passed to the training job through channels; a channel is a named input source that training algorithms can consume. For each input, we specify the name of the channel, for example, "train" or "validation", and associate an object of the class *TrainingInput* to describe the channel. To create the *TrainingInput*, we have many parameters; in our case, we used just two that are enough: `s3_data` and `content_type`. With the former, we specify where the input is located on S3; if this is the result of a previous step, for example, data preprocessing, we can access this information using the properties of the *ProcessingStep*, for example

```
step_process.properties.ProcessingOutputConfig.Outputs[
                "train"
            ].S3Output.S3Uri
```

where "train" is the `output_name` parameter that we saw before. On the other hand, `content_type` is used to describe the MIME type of the input data.

Another interesting thing about Estimators is about the Docker image on which the training job will be run. We can use a base Estimator and specify the image by passing the Amazon Elastic Container Registry (Amazon ECR) path of the Docker image containing the training algorithm or decide to use *framework Estimators*. With these Estimators, we just need to specify the framework version and other specific parameters, and SageMaker will automatically get the right Docker image for us. Examples of framework Estimators are XGBoost, PyTorch, SKLearn. Moreover, we can also decide to pass a custom Docker image to create our own training container; we will discuss this in more detail in Section 4.2.1.

An example of *TrainingStep* is the following:

```
    sklearn_estimator = SKLearn(
            entry_point=<train_code.py>,
            instance_type=<instance_type>,
            framework_version=<sklearn_version>,
            hyperparameters = {<name>: <value>},
            sagemaker_session=<pipeline_session>)

    sklearn_estimator.fit({'train': <s3_data_path>)
```

```
step_train = TrainingStep(
    name=<step_name>,
    step_args=step_args,
)
```

3.6.5 QualityCheckStep e ClarifyCheckStep

QualityCheckStep and ClarifyCheckStep can be used to both create new baselines and to check against previously computed baselines to detect drift. The former is focused on data quality or model quality; the latter is focused on bias analysis and model explainability. In our project, we will use these steps only to compute baselines for data quality and bias analysis, as data quality drift will be checked by a monitoring schedule that will be discussed later. To achieve this, we can use two parameters that are common to both steps:

- `skip_check`: This parameter indicates whether the drift check against the previous baseline is skipped. We set it to True, so no check is performed.
- `register_new_baseline`: This parameter indicates whether a new baseline should be computed or a newly calculated one is available. We set it to True, so a new baseline is computed when the step is executed.

For both steps, we need one common object that specifies technical information about the jobs, which is called `CheckJobConfig`; an example is:

```
check_job_config = CheckJobConfig(
    role=<role>,
    instance_count=<instance_count>,
    instance_type=<instance_type>,
    volume_size_in_gb=<size>,
    sagemaker_session=<sagemaker_session>,
)
```

QualityCheckStep

With *QualityCheckStep*, we generate our baseline based on the training dataset. Generating a baseline in this context means creating two files:

- `statistics.json`: This file contains statistics about the dataset and about each feature. It computes common statistics for each feature, like the number of missing values and the distribution, by creating a histogram with ten bins. For numerical features, it also computes statistics like mean, standard deviation, min, max, etc. These statistics are computed using KLL [22], which is an optimal and accurate quantile sketch. Quantile sketches are algorithms that let us estimate the distribution of values in a stream.

- `constraints.json`: This file expresses the constraints that a dataset must satisfy. For each feature, it specifies the name and the completeness (ratio of present values over the total number of values, 1.0 means there are no missing values); if a feature is numerical, it specifies if it is non negative, while if it is categorical it specifies the domain of the feature, which means what are the possible values that the feature can take.

  This file also contains an object called `monitoring_config`, which is automatically created and contains options for the monitoring job, specifying, for example, whether to check for datatypes violations or domain violations or how to perform the data distribution comparison during model monitoring and what are the thresholds. These last options, in particular, will be discussed in more detail in Section 4.9.

We can then generate and register our baselines with the `model.register()` method and pass the output of that method to Model Step using `step_args`, so we can access the baseline from the Model Registry when we perform model monitoring; furthermore, we can also visualize them from SageMaker Studio. The QualityCheck step leverages the Amazon SageMaker Model Monitor pre-built container, which has a range of model monitoring capabilities, including constraint suggestion, statistics generation, and constraint validation against a baseline. This container is also used when the monitoring job is executed. To define the *QualityCheckStep*, we need to create another object before, which is called `DataQualityCheckConfig` that is used to describe the input dataset and where the constraints and statistics JSON files will be saved on S3. We can see an example of *QualityCheckStep* in the following example:

```
data_quality_check_config = DataQualityCheckConfig(
    baseline_dataset=<baseline_dataset_path>,
    dataset_format=<dataset_format>,
    output_s3_uri=<s3_path>
)

step_data_quality_check = QualityCheckStep(
    name=<step_name>,
    skip_check=<True/False>,
    register_new_baseline=<True/False>,
    quality_check_config=data_quality_check_config,
    check_job_config=check_job_config,
    model_package_group_name=<model_package_group_name>,
)
```

ClarifyCheckStep

With *ClarifyCheckStep*, we generate our baseline based on the training dataset. In this context, generating the baseline means creating one machine readable file, `analysis.json` and a report in multiple formats, which are pdf, html and ipynb (Jupyter Notebook), which is instead intended for human readers.

- `analysis.json`: This file contains bias metrics and (if computed) feature importance in JSON format. This file contains two sections, one for pre-training bias metrics and one for post-training ones; we computed only the first ones in our case.
- `report`: This file contains visualizations and explanations of bias metrics and (if computed) feature importance. It includes, for example, the distribution of label values, table of bias metrics and their descriptions, etc.

Both files contain the metrics computed on a certain protected attribute and on a certain value of said attribute. To specify this information and further information needed by the step, we create a configuration in which we define:

- dataset type: This parameter specifies what is the dataset format, for example, `text/csv` or `application/json`, etc.
- label: This parameter is used to define the name of the column containing the target attribute.
- label values or threshold: This is an array of label values or a threshold number that represents the positive outcome of the label.
- facet: This is an array of facet objects, which are the sensitive attributes against which bias is measured.
  - name or index: the name of the column of the protected attribute.
  - value or threshold: an array of facet values or a threshold number that indicates the sensitive demographic groups that are disadvantaged.
- methods: This parameter specifies which methods should be used for the analysis. We can specify "all" and let the step compute all methods.

As we saw before for the *QualityCheckStep*, also in this case we can register our baselines with `model.register()` and pass the output to Model Step using its `step_args`, so these metrics can be accessed from the Model Registry when we perform model monitoring and we can also visualize them.

The ClarifyCheck step leverages the Amazon SageMaker Clarify pre-built container, that provides a range of model monitoring capabilities, including constraint suggestion and constraint validation against a given baseline.

To define the *ClarifyCheckStep*, we need to create different objects before, which are:

- BiasConfig: corresponds to the `facet` object that we saw before in the configuration;
- DataConfig: comprehends all the other objects but `facet`.
- DataBiasCheckConfig: this object is used to group the previous two configurations together

We can see an example of *ClarifyCheckStep* in the following example:

```
data_config = DataConfig(
    s3_data_input_path = <s3_path>,
    dataset_type=<dataset_type>,
    label=<target_column_name>,
    s3_output_path=<s3_path>,
    s3_analysis_config_output_path=<s3_path>
)

data_bias_config = BiasConfig(
    label_values_or_threshold=<positive_label>,
    facet_name=<facet_name>,
    facet_values_or_threshold=<disadvantaged_value>,
)

data_bias_check_config = DataBiasCheckConfig(
    data_config=data_config,
    data_bias_config=data_bias_config,
)

step_data_bias_check = ClarifyCheckStep(
    name=<step_name>,
    clarify_check_config=data_bias_check_config,
    check_job_config=check_job_config,
    skip_check=<True/False>,
    register_new_baseline=<True/False>,
    model_package_group_name=<model_package_group_name>
)
```

3.6.6 ConditionStep

The ConditionStep allows the pipeline to support the execution of other steps based on the evaluation of a condition. Before defining the step, we need to specify one or more conditions; we can do this thanks to the conditions that SageMaker provides us, some of which are: `ConditionEquals`, `ConditionGreaterThanOrEqualTo`, `ConditionOr`, `ConditionNot`. An example of condition definition is the following:

```
    ConditionGreaterThanOrEqualTo(
        left=JsonGet(
            step_name=<step_name>.name,
            property_file=<property_file_name>,
            json_path= <jsonpath_to_metric>
        ),
        right=<value>
    )
```

At a higher level, we can see that conditions accept two parameters, which are `left` and `right`, with which we define what are the values that will be on the left and on the right of the operator. In the example, the condition is the following:

$$\text{metric} \geq \text{value}$$

As we can see in the example, we used the function *JsonGet*, thanks to which we can explore the content of the property file that we obtained as output of a previous step. We use JsonPath notation to query the property JSON file to obtain the metric we need. The condition is finally evaluated by the *ConditionStep*, which decides to execute the (*if_steps*) or the (*else_steps*).

To define the ConditionStep we need a list of conditions, the steps to be executed in case the specified conditions turn out to be true (*if steps*) and the steps to be executed in the opposite case (*else steps*). If multiple conditions are present, they are linked by a logical AND. An example of *ConditionStep* is the following:

```
    step_cond = ConditionStep(
            name=<step_name>,
            conditions=[cond_1, cond_2, ...],
            if_steps=[step_1, step_2, ...],
            else_steps=[step_3, step_4],
    )
```

3.6.7 FailStep

The *FailStep* stops the pipeline execution when a condition is not met. When the pipeline enters this step, the execution is logged as failed. This step also allows us to enter a message to communicate the error, either as a static string or by using SageMaker's *Join* function to concatenate a string with pipeline parameters. This step is always the last one that is executed, and there can be no steps that depend on it. An example of the *FailStep* is the following:

```
step_fail = FailStep(
    name=<step_name>,
    error_message=Join(
        on=" ",
        values=[
        "Pipeline failed due to <metric> <",
        <pipeline_param>
        ]
    ),
)
```

In the example, we can see that the error message is constructed using the *Join* function because we had to include a parameter of the pipeline (described in Section 3.2.1).

3.6.8 ModelStep

The *ModelStep* can be used for two different objectives: to create a model or register a model on SageMaker's Model registry. Whether we want to create a model so we can deploy it right away or we want to register a model on the Model Registry, we first have to create an object of type *Model* to which we pass the URI of the image we used for training, the data of the model generated after the training step, etc. We can create a Model in this way:

```
model = Model(
    image_uri=<image_uri>,
    model_data=
        step_train.properties.ModelArtifacts.S3ModelArtifacts,
    sagemaker_session=<pipeline_session>,
    role=<role>,
)
```

Also, the *ModelStep* itself is the same for both objectives except for the parameter `step_args`, which are different based on the fact that we are creating or registering the model, with the methods `create()` or `register()` respectively on the *Model* object. An example of the *ModelStep* is the following:

```
step_model = ModelStep(
   name=<step_name>,
   step_args=<step_args>,
)
```

This step can be inserted as a step in the pipeline or as a `if_step` in a *ConditionStep* to register or create the model just in case one or more conditions (for example, on evaluation metrics, as we saw before) are satisfied.

Model registration

Before discussing how the model registration happens, let us first discuss the Model Registry. The Model Registry is a SageMaker tool that allows us to perform operations like:

- manage trained model versions
- manage its approval status
- associate it with training metadata such as accuracy, confusion matrix, etc
- deploy a model to an endpoint

A model in this context is called a *Model Package*. Many Model packages are stored in collections called **Model Package Groups**, which are groups created to track all models we train to solve a particular problem. When we add a new model to a package group, SageMaker automatically creates a new version. A *ModelPackage* represents a reusable abstraction of the model's components that are necessary for inference.

After defining the Model, we use the method `register()` to generate the `step_args` that we will pass to the ModelStep to register the model on the model registry.

```
register_model_step_args = model.register(
    content_types=["application/json"],
    response_types=["application/json"],
    inference_instances=[instance_type],
    transform_instances=[instance_type],
    model_package_group_name=<model_package_group_name>,
)
```

With this method, we define the specifications of the model that will be saved on the Model Registry.

Model creation

If we instead want to create a model, after defining the Model object, we use the method `create()` with which we generate the `step_args`.

```
    create_model_step_args = model.create(
        instance_type=<instance_type>
    ),
```

With this method, we create the SageMaker model that will be deployed on an endpoint. When we create the model with this methodology we see the model in the SageMaker UI, ready to be deployed, while if we register it on the Model Registry, the model does not appear on the UI.

3.6.9 LambdaStep

The *LambdaStep* is used to execute an AWS Lambda function. AWS Lambda is a serverless, event-driven compute service that allows us to run code without provisioning or managing servers. It is up to the user to provide its custom code, while Lambda runs it on high availability compute infrastructure and performs all the administration of compute resources, such as server and operating system maintenance, automatic scaling, security patch deployment, code logging, etc.

We can choose to execute an existing Lambda function or create one via SageMaker. In order to execute a Lambda function using the LambdaStep, we need to create an object of type Lambda. The Lambda type object can be created from an existing Lambda function via the `function_arn` parameter, for example:

```
Lambda(
    function_arn="arn:aws:lambda:..."
),
```

We also have the option of having SageMaker create the Lambda function for us, via the `handler` parameter, with the format `file_name.function_name`, and one between:

- `zipped_code_dir`: the path of the lambda function in .zip format.
- `s3_bucket`: the bucket of s3 where `zipped_code_dir` is loaded.
- `script`: the path to the lambda function script.

. An example is:

```
lambda=Lambda(
    function_name=<lambda_function_name>,
    execution_role_arn=<lambda_role>,
    script=<code_path>,
    handler=script_name.function_name
)
```

Once we have created the Lambda type object, we can execute the *Lambda Step* by passing the Lambda via the `lambda_func` parameter.

Our Lambda function can also process inputs and generate outputs. To define inputs we create a dictionary of key-value pairs, where values can only be primitive types (string, integer, or float) and nested objects are not supported. For the outputs, we instead define a list of keys, which are the keys defined in the dictionary returned as output from the Lambda function; also in this case, only primitive types are supported.

An example of a Lambda Step is the following:

```
step_lambda = LambdaStep(
    name=<step_name>,
    lambda_func=lambda
    inputs={
        <key1> = <value1>,
        <key2> = <value2>,
    },
    outputs=[
        <ret_key1>, <ret_key2>
    ]
)
```

3.6.10 SageMaker session and Pipeline Session

We might have noticed that sometimes in the parameters of the objects we saw until now, the parameter `session` may have appeared, for example, in the definition of *Model*. In that particular case, the parameter is called `sagemaker_session`, but the placeholder says `pipeline_session`. There is actually a difference between these two types of session; in fact, Pipeline Session is an extension of SageMaker Session. SageMaker Session manages the interactions between SageMaker APIs and AWS services like Amazon S3 and provides convenient methods for manipulating entities and resources that Amazon SageMaker uses, such as training jobs, endpoints, and S3 input datasets.

The main difference is that sometimes we need to call a method that is needed by the steps; for example, for *ModelStep*, we have to call `model.register()` or for *TrainingStep*, we have to call `estimator.fit()`. Pipeline session makes sure that the `fit()` method does not immediately start a training job, but the method is executed during a pipeline execution.

Processing and training components

Some components, like Processing and Training ones, need code that specifies what operations the component should perform. For example, for a processing component, we have to write the code for data processing or model evaluation, while for a training component we could, for example, perform some operations before starting the training, like transforming some columns. In this case, we have the possibility to choose how to provide these instructions.

- Script mode: In *script mode*, we write our code in a file, for example a .py or .sh file, and we pass it to the component, to which we will also pass a base image and the command to use to execute the script, such as `python3` or `sh`. As we saw in Section 3.1, we can use as base images pre-built Docker images that SageMaker has created for us and already has installed many common libraries. We can find these images in lists divided by region or through the method `image_uris.retrieve` that we can find in Python SDK.

- Bring your own container [23]: In this case, we create or already have an image that contains our training script. If we create the image, we can choose one pre-built container by SageMaker as base image or a Docker base image on which we will install the necessary libraries for our script. This mode is used when we want to install libraries not on PyPi, if the framework we want to use is not already provided by SageMaker or if we want to customize the environment. To use this method on SageMaker, we need to use the **training toolkit** to make the image compatible with SageMaker and the **inference toolkit**, which allows us to deploy the model on SageMaker. Both will be discussed more deeply in Section 4.2.1.

  In our project, we chose to use this strategy for another reason, which is that by the containerization of the training script we can obtain traceability (thanks to the versioning of the images enabled by default on ECR) and transparency (since we can obtain the whole content of any image to inspect it or to rerun the pipeline with an old image), which are two very important aspects in the context of MLOps and AI Act. Also, we obtain the advantage of using an algorithm not provided by SageMaker.

Chapter 4

Architecture of the solution

In this section I will talk about the implementation details of the solution. We will discuss each logical step of the pipeline, describing in detail which kind of services and steps we used, what are the inputs, outputs, and dependencies.

4.1 General setup

4.1.1 IDE

We implemented the pipeline using the IDE **SageMaker Studio**. As we said before, this allows us to perform operations like writing code, checking the execution status of the pipeline, its steps and their inputs and outputs, other than many more operations. This can all be done in SageMaker Studio without having to move back and forth to the SageMaker UI. To test the pipeline, we used the built in Terminal and we launched the pipeline from the command line as a regular Python script with:

```
python3 pipeline.py --use-case <use_case>
```

In a production environment, instead, the pipeline is launched automatically using the two triggers we defined on the codebase, eliminating the need for manual intervention.

4.1.2 Data storage configuration

The pipeline is set up to execute more use cases, which we pass as a command line argument. For this purpose, the S3 bucket where all the data generated and that is needed by the pipeline is organized by the following schema:

.

```
└─ data-it-ena-mlops/
   └─ use-case/
      ├─ data/
      └─ pipeline-name/
         └─ timestamp/
            ├─ step-name1
            ├─ step-name2
            ├─ ...
            └─ Logs
```

The timestamp represents the moment in which the pipeline execution begins. To maintain general coherence, the timestamp generated at the beginning of the pipeline execution is passed down to all the steps so they can all refer to the same folder. Inside the `timestamp` folder, other subdirectories store the outputs of the steps, like the trained model or the evaluation file; it also contains a subfolder called `Logs` where logs files produced during the execution of the pipeline are stored.

4.1.3 Configuration files

All of the configurations needed by the components of the pipeline are contained inside a file called `config.yaml`, which has sections for each component that needs configurations. Inside each section are the configuration values. This file was created to make the pipeline as customizable as possible with the least effort possible, as everything that needs to be changed is inside this file. For example, suppose we want to adapt the pipeline to a new use case. In that case, we only have to adapt the configuration file by changing the name of the dataset file, what represents the positive outcome in the label, configurations for bias measures computations, etc. To access the configuration file, we created a utility script that loads it wherever needed.

4.1.4 The Pipeline object in-depth

Starting from a high level, we can discuss the Pipeline object more deeply. As we described earlier in Section 3.2.1, we first define a *Pipeline* object; then, we start the pipeline with `pipeline.start()` where we can also override some parameters. There are three more methods that we have used in our solution, which are:

- `pipeline.describe()`: This method describes the details of the pipeline, for example: its ARN, the datetime it was created and run for the last time, the details of who created it and who modified it, etc.
- `pipeline.upsert(role_arn=role)`: This method is fundamental and must be used before using `start()`. It submits the pipeline definition (a JSON file

that describes the pipeline) to the Pipeline service and is used to create a pipeline if it does not exist or update an existing one. The role passed is used to create all the jobs the pipeline needs.

- `execution.wait()`: This method waits for the execution to finish. We used this to upload on S3 the log files generated during the execution.

Another aspect that we can talk about is parameters, in particular, what are the parameters that we decided to use and what they represent:

- pipeline ARN: This is the ARN of the pipeline that is going to be executed. We extract this parameter from the response of `pipeline.describe()` before starting the pipeline and is used later to configure automatic retraining.

- train image: This is the URI of the training image used by the *TrainingStep* to train the model. This parameter is fundamental if we want to launch a training with an old image, for transparency reasons.

- training instance type: This parameter represents the instance type used for training the model. The same instance is used across different steps to execute different jobs, as the machine used for training is most of the time safe to be used also for other jobs.

- inference instance type: This parameter represents the instance type used for inference. We do not use the same instance type used for training as for inference, a smaller machine that costs less is sufficient.

There are four more parameters, that we use to configure data quality and data bias steps, which are: `skip_check_data_quality`, `skip_check_data_bias`, `register_new_baseline_data_quality`, `register_new_baseline_data_bias`.

Section 3.6.5 will discuss the meaning and how we configured these parameters to obtain the desired behavior.

4.1.5 Use case

For the whole duration of the Pipeline development, the use case that we tested the pipeline on was chosen at the beginning of the project and it is **credit default risk**. The dataset for this use case is a very popular one and contains information about requests for loans with information about the person who requested it. This use case is a binary classification task, where the label of the dataset represents whether the person who asked for the loan could repay it or not. This dataset contains sensitive attributes, such as `CODE_GENDER`, that represents the sex of the person.

4.2 Automatic deploy

The pipeline can be automatically started in two ways:

- Triggers: As we saw in Section 3.5, we set up triggers on the service *CodeBuild* that are linked to our GitHub repository. The pipeline is started on pushes on the main branch and on any branch whose name starts with "feature/" where triggers match a certain regex.
- Retraining: Another way the pipeline is launched is if the model needs retraining.

  When drift on data is detected, the model's performances will likely degrade as the data the model was trained on differs from what the model is currently receiving from the endpoint. Since the model has never seen this kind of data, it could lack this degree of generalization and thus produce completely wrong predictions, potentially damaging the client or the users who employ those predictions to make critical decisions. We will discuss the implementation details of this mechanism more in-depth in Section 4.9.

When the pipeline is started after the retraining is triggered, we only use its ARN. The execution happens in the same conditions as the last pipeline execution (unless pipeline parameters are changed), so it keeps all previously used parameters and configurations.

On the other hand, when the pipeline is started through triggers, the environment is set up by installing libraries, exporting environment variables, and using `docker` cli commands to create our custom training image. This way, each time the pipeline is started, it always uses the latest training image unless we specify another as a pipeline parameter.

4.2.1 Containerization

As highlighted in 3.6.10, we use a custom training image for many reasons, so we need to perform containerization to obtain such an image.

In each image build process, Docker executes the commands we defined in the Dockerfile sequentially. The Dockerfile, the training script, configuration files, and other auxiliary files are in a folder called *trainer*. In the Dockerfile, we specify a Python image available on public ECR as a base image. Then, we install the libraries we will need for the training, such as scikit-learn, Pandas, etc. We then transfer the contents of the *trainer* folder into the local `/opt/ml/` directory and specify our training script as the entry point.

Inside the *trainer* folder, we have a bash script to perform the build and push of the image on ECR. Once uploaded to ECR, each image is versioned, allowing for the retrieval and inspection of the contents of each file from the time of its creation. This ensures transparency and traceability, permitting us to review the exact training script used for any specific model.

When we create our own custom images, they are not immediately compatible with SageMaker. However, we need to use two toolkits that implement the functionalities needed to adapt our containers to run scripts, train algorithms, and deploy models on SageMaker. For example, the training toolkit defines the locations for storing code and other resources, which we discussed before, and is `/opt/ml/`. The two toolkits are the *inference toolkit* and *training toolkit*.

To install the training toolkit, we include `sagemaker-training` in the list of libraries we install with pip in our Dockerfile. Using the toolkit also allows us to use the hyperparameters parameter in the *Estimator* object that we create for model training and read the hyperparameters from the training script using the library `argparse`. It also provides us with environment variables that we can read. For example, the variable `SM_CHANNEL_TRAINING` provides the training channel's paths.

On the other hand, the inference toolkit allows us to adapt our container to work with SageMaker hosting in case we want to host our model on an endpoint. To use the toolkit, we created a file called `inference.py` file, which we included in our Dockerfile and passed to the *Model* object (which we will see later) as an entrypoint. In the file, we have three main functions, which are:

- `model_fn(model_dir)`: In this function, we specify how SageMaker should load a model. In our case, since we saved the model as a pickle file, we read it using:

  ```
  with open(Path(model_dir) / "model.pkl", ’rb’) as f:
      loaded_model = pickle.load(f)
  ```

  we used the `Path` method from the library `pathlib`, which lets us concatenate paths better.

- `input_fn(input_data, content_type)`: In this function we pre-process input data. It allows us to deserialize input data so it can be passed to our model. It takes the input data and the content type as input and returns deserialized data. With the parameters, we can decide what kind of preprocessing to perform. For example, in our implementation we have:

  ```
  if content_type == ’text/csv’:
      df = pd.read_csv(
          StringIO(input_data),
  ```

```
                header=None,
                delimiter=";")
            return df.values
        elif content_type == 'application/x-npy':
            return np.load(
                BytesIO(input_data),
                allow_pickle=True)
```

  as we can see, we need to perform different preprocessing operations for different types of data.

- `predict_fn(input_object, model)`: This function is responsible for getting predictions from the model. It takes the model and the data returned from `input_fn` and returns the prediction. In our case, it is straightforward:

  ```
  predictions = model.predict(input_object)
  ```

After implementing these two toolkits, our custom image is ready to be used with SageMaker to train the model and deploy it on an endpoint. After the containerization process has finished, we can finally start the pipeline via the command line.

After this logical step, the pipeline execution starts with data ingestion.

4.3 Data ingestion

With data ingestion, the pipeline execution begins. We use a *ProcessingStep* that has the task of performing very basic preprocessing, i.e. retrieving the complete dataset from S3 and, if necessary, performing the feature selection process. Features are selected in a static way by compiling a list of features to keep. Ideally, this list is created by the data scientist who decides which features are important based on ML methods (for example, Random Forest), statistical methods (for example, correlation), domain knowledge, or by experimenting. In our case, the columns were selected by a data scientist when the project first started.

In this step, other than selecting features, we perform an additional type of preprocessing: map the label values that indicate a positive outcome to "1," while the others are set to "0." This step is necessary since evaluation methods based on calculating separate class metrics (e.g., recall, precision, etc) use "1" as the default value to indicate the positive class. Furthermore, the data scientist does not have to specify the label that represents the positive outcome in the configuration of the *ClarifyCheckStep* (in the next section), as it is already set to "1".

Finally, we split the dataset into training and testing following the configuration specified by the data scientist in the configuration file. The configuration that we used is `test_size=0.2`, which means that the test size will be 20% of the dataset size and `random_state: 42`, with which we set a seed that allows us to split the dataset consistently in the same way, so we avoid randomness across different pipeline runs.

This first step is mandatory in order to perform the monitoring of the data drift since the statistics must be computed (in the next step) on the same schema as the requests received by the model deployed on endpoints.

The *ScriptProcessor* is instantiated with the command "sh" since a bash script is first executed in this step. There was the need to use a bash script as the base image does not come with the library `PyYAML` installed, needed to load the `config.yaml` file. In the script, in fact, we install said library and then proceed to execute the Python script that contains the processing code.

The inputs for this step are:

- input dataset: the dataset related to the use case that we will preprocess.
- path of the python script: the path where the python script that will execute the preprocessing is located.
- path of the configuration file: the path where the yaml configuration file is located.
- path of the utility file: the path where the file that contains the utility script to load the configuration file is located.

As outputs, we obtain: the dataset that we will use for training and baselines computation, the dataset that we will use for testing, and a log file created during the execution of the step.

4.4 Data bias and data quality statistics

Once we performed the basic preprocessing, we used the dataset we obtained as output to compute the baselines. As we saw in Section 3.6.5, we will use two different types of steps: one that computes a baseline regarding data quality (and that will be used for monitoring), the other one that computes a baseline regarding biases in the input data; these steps are, respectively, *DataQualityCheckStep* and *ClarifyCheckStep*. Before creating these steps, we need to configure the *DataQualityCheckConfig* for the former, *DataConfig* and *BiasConfig* objects for the latter; in particular the most important configurations are:

- selection of the baseline dataset: This configuration tells the components what is the dataset on which baselines computation should be performed. The dataset is obviously the output of the previous step, so we retrieved it using the step properties `step_basic_preprocess_dataset.properties` and navigating using JsonPath notation until we reach the S3 URI of the folder containing the dataset. To access the dataset, we use SageMaker's builtin `Join` function, which allows us to perform string concatenation operations that are not otherwise allowed by SageMaker on properties or pipeline parameters. To access the dataset, which is called `train.csv`, we execute the following method:

```
Join(
    on='/',
    values=
    step_basic_preprocess_dataset.properties.\
    ProcessingOutputConfig.Outputs["train"].\
    S3Output.S3Uri,
    "train.csv"]
),
```

  Using the `properties` attribute creates a data dependency, so these two steps will always be executed after completing the previous one. Other than specifying the dataset, we also need to set the dataset format, for example `text/csv`.

- S3 output paths: where the outputs of these two steps will be stored on S3, namely the `statistics.json` and `constraints.json` files for the *DataQualityCheckStep* and the `analysis.json` and the reports for the *ClarifyCheckStep*.

- configurations for bias checks: This configuration is used to configure the jobs that will compute the bias metrics. These configurations include the column name of the label, what is the label that represents a positive outcome, the column name of the protected attribute, and the value that is disadvantaged. In our case, we chose as the protected attribute the column `COLUMN_GENDER`, which represents the gender of the person who requested the loan, and as a disadvantaged value, we chose `'F'`, which represents females.

There are also other settings that need to be configured. After these configurations are done, the steps can be created.

When the *ClarifyCheckStep* finishes, it is followed by a significant addition regarding fairness and the AI Act, which is a *ConditionStep* to check if a certain bias measure is within a specific boundary or not; the data scientist chooses the measure and the threshold. This is fundamental because if the dataset turns out to be biased, the model will learn from biased data, and the predictions will be biased. With this step, we introduce an innovation as the pipeline is automatically stopped if fairness in the dataset is not met. If the condition is not satisfied, the pipeline is redirected to execute a *FailStep*, with a message informing the data scientist that the pipeline could not be executed further as there are fairness problems. The pipeline moves to the *TrainingStep* if the condition is satisfied.

Another step is executed before the condition is evaluated: a *ProcessingStep*. This step is used to rearrange the output of the *ClarifyCheckStep* (a JSON file) as it is unsuitable to search for the chosen fairness measure to check via JsonPath. This step takes only one input, which is the output of the *ClarifyCheckStep* through its `properties` attribute. This creates a data dependency, and this step is only executed after the *ClarifyCheckStep* has finished computing fairness metrics. The outputs of this step are a log file created during the execution of the step and the rearranged JSON file that is saved as a *property file* that can be accessed by the subsequent *ConditionStep*. The new file layout can now be easily explored using JsonPath, like so:

```
metrics.<target_measure>
```

Since some measures are only positive, like the KL divergence, while others are between -1 and 1, like Class Imbalance, another operation that the *ProcessingStep* performs is to compute the absolute value of the measure before saving it into the output file. This is actually a convenient operation since positive measures are not affected, while measures that can take negative values are symmetric (discussed later). This lets the data scientist specify a threshold representing an upper bound, but for symmetric measures, it is really an interval that the measure should be within. The code this step should execute is inside a Python script, and the command to execute it is `python`, as we do not need a shell script in this case to perform prior operations.

For the *QualityCheckStep*, instead, we execute the step and use its output further in the pipeline when we will register the model on Model Registry.

We will now see a more in-depth view of the metrics computed by the *Clarify-CheckStep*.

4.4.1 Bias metrics

The metrics computed by the *ClarifyCheckStep* are several, describing different aspects of the data. Since we are computing these metrics before the model training, they are called pre-training bias metrics because we analyze the dataset for biases and mitigate them before the data is used for model training. These kinds of metrics answer the question: Do all facet values have equal or similar representation in the data?

These metrics can be further split into macro-categories. The examples that we will present are referred to the dataset that we chose, where the facet is the column that represents gender. We will use d to represent the facet value that the data scientist chooses as disadvantaged, while a to represent the advantaged facet value; in our case, d is represented by females, while a is represented by males.

- Facet value representation irrespective of labels: This category contains only one metric and is computed on the facet without taking into consideration the label.
  - Class Imbalance (CI): Measures how balanced is the representation of facet values in the data. For example, in our dataset, males represent the 34.2% of data, females 65.8%. Class Imbalance is computed as:

    $$CI = 0.342 - 0.658 = -0.316$$

    A Positive CI means that the facet a has more training samples in the dataset. A high class imbalance could lead to worse predictive performance for the facet value with a smaller representation. This is a symmetric measure and has values between -1 and 1.
- Facet value representation at the level of positive labels only: This category contains two metrics representing whether all facet values contain a similar fraction of samples with positive observed labels. This could reveal that some groups are more advantaged than others. The metrics are:
  - Label Imbalance (DPL): computes the difference between the proportion of observed outcomes with positive labels for facet d with the proportion of observed outcomes with positive labels of facet a in the training dataset. For example, in our dataset, it is 0.031, computed as:

    $$DPL = \frac{n_a^1}{n_a} - \frac{n_d^1}{n_d} = 0.899 - 0.930 = -0.031$$

    A Positive DPL means that facet a has a higher proportion of positive outcomes. If it is close enough to 0, we can say that demographic parity has been achieved.
  - Conditional Demographic Disparity (CDD): Demographic disparity helps us understand if a facet has a larger proportion of the rejected outcomes in the dataset than the accepted outcomes.

    Conditional DD, on the other hand, conditions DD on attributes that define a strata of subgroups on the dataset. The regrouping can provide insights into the cause of apparent demographic disparities for less favored facets. For example, for a famous sample dataset, DD indicated that women had a lower acceptance rate than men. However, with CDD, we can discover that women applied to departments with a lower acceptance rate than men, and women were accepted more than men for these departments.

This measure was not computed for our dataset as we could not identify a column to regroup on.

- Facet value representation at the level of each label separately: With these measures, we try to understand representation equality for each label, not just the positive label. These metrics are more naturally applicable to non-binary labels since imbalance in the positive label can be used to compute imbalance in the negative label. This means that these metrics provide the same insights as DPL. To compute them, we need to first introduce the concept of probability distributions of labels for each facet. For binary labels, we can compute these probabilities as:

$$P_a(y^1) = \frac{n_a^1}{n_a}, \quad P_a(y^0) = \frac{n_a^0}{n_a}$$
$$P_d(y^1) = \frac{n_d^1}{n_d}, \quad P_d(y^0) = \frac{n_d^0}{n_d}$$

  - Kullback-Leibler Divergence (KL): Measures how much the outcome distributions of different facets diverge from each other. The formula is:

$$KL(P_a||P_d) = \sum_y P_a(y) \cdot \log[P_a(y)/P_d(y)]$$

    In our case, we only have two possible outcomes, 1 and 0; for our dataset the value is:

$$KL = 0.06$$

    Values near zero mean the outcomes are similarly distributed for the different facets and the greater the value the higher the divergence.

  - Jensen-Shannon Divergence (JS): measures how much the label distributions of different facets diverge from each other. It is based on the Kullback-Leibler divergence, but it is symmetric. The formula is:

$$JS() = \frac{1}{2}KL(P_a||P) + \frac{1}{2}KL(P_d||P)$$

    Where $P = \frac{1}{2}(P_a + P_b)$.

Values near zero mean the labels are similarly distributed and the greater the value the higher the divergence. This metric is particularly useful to understand if a significant divergence exists in one of the labels across facets.

- L_p-norm (LP): The L_p-norm (LP) measures the p-norm distance between the facet distributions of the observed labels. The formula is:

$$L_p(P_a, P_d) = (\sum_y ||P_a - P_d||_p]^{\frac{1}{p}}$$

The 2-norm is the Euclidean norm.

The range of this measure is $[0, \sqrt{2})$. Values near zero mean the labels are similarly distributed and the greater the value the higher the divergence.

- Total Variation Distance (TVD): Measures half of the L1-norm and represents the largest possible difference between the probability distributions for label outcomes of facets a and d. It quantifies how many outcomes in facet a would have to be changed to match the outcomes in facet d.

The range of this measure is $[0, 1)$. Values near zero mean the labels are similarly distributed and the greater the value the higher the divergence.

- Kolmogorov-Smirnov (KS): This measure is equal to the maximum divergence between labels in the distributions for facets a and d. It helps us find the most imbalanced label. The formula is:

$$KS = max(|Pa_(y) - P_d(y)|)$$

In our case, both labels provide the same value, 0.031, so there is not one label that is more unbalanced than the other.

The range of this measure is $[0, +1]$. Values near zero mean the labels are similarly distributed between facets in all outcome categories. Values near one indicate the labels for one outcome were all in one facet, while intermediate values indicate relative degrees of maximum label imbalance.

The data scientist should choose the right metric which is conceptually appropriate for the application and the situation.

4.5 Model training

If the condition on the fairness measure is satisfied, the Pipeline can move on to the next phase, which is training the model. To train the model, we create the *Estimator* object to which we pass a *training image*. As discussed before, we can use a pre-built SageMaker image or use a custom image following the practice

"Bring your own container" discussed in Section 3.6.10. The latter is the strategy that we use; in fact, to train the model, we pass the custom training image that was created during the automatic deployment, whose process is described in Section 4.2.1. We used an algorithm that is not provided by SageMaker, Random Forest. Another important parameter, besides the training image, is `sagemaker_session`, which should be assigned a `PipelineSession` object, for the reasons we discussed before; another parameter that we set is the S3 output path, which is where the output of this step will be saved. In fact, after creating the Estimator, we need to perform the fit of the model whose results are parameters that we will pass to the *TrainingStep*.

As input to this step, we pass the S3 path of the dataset obtained as the output of the basic preprocessing step. To access the dataset, we perform the same operation that we did in Section 4.4, so we use the Join operation to concatenate the name of the file with the folder path that we obtain as a property of the step. This creates a data dependency, thus executing the *TrainingStep* necessarily after the basic preprocessing. This happens even if we do not have this data dependency, as the *TrainingStep* is always executed after the *ConditionStep* on the fairness measure. As output, we obtain a file called `model.tar.gz`, which contains everything we decide to save within the training script, i.e. the trained model saved as a `.pkl` (pickle) file. Pickle is a generic object serialization module that can be used for serializing and deserializing objects, but is commonly used to store and reload trained machine learning models. We save the model in the following way:

```
with open(model_path, "wb") as out:
    pickle.dump(model, out)
```

This means that we open the file where we want to save the model in binary mode (wb), and then we perform `pickle.dump()` to save the model on the specified path.

4.6 Model evaluation

After the model has been trained, we must evaluate it against a test dataset to compute metrics that let us decide whether the model should be deployed or is too weak and needs more polishing. To perform model evaluation we instantiated an object of type *ScriptProcessor* with a base scikit-learn image that is provided by SageMaker, the code that should be executed, and `'python'` as a command to execute the evaluation Python script. After, we create the *ProcessingStep* to which we pass the *ScriptProcessor* that we just created, inputs and outputs.

The evaluation we perform is to compute different classification metrics, allowing the data scientist to choose which one he wants to evaluate his model on. The metrics that we compute are accuracy, precision, recall, f1, and the confusion matrix; they can be visualized in the Model Registry within SageMaker Studio, alongside the other metrics.

For this task, we pass the test set and the model as input. The test set was created during the basic preprocessing step, and we obtained it using its properties. This also creates a data dependency between the two steps. Another data dependency is created with the second input, which is the model that we will actually test.

As outputs, we obtain a log file that is created during the step execution and a property file with the evaluation report. We created this property file so it can be accessed by the next *ConditionStep*, with which we will decide whether to continue with the pipeline.

4.7 Model registration

After we evaluated the model, we now need to register the model on the Model Registry. The registration is not performed every time; it can happen that we complete a training process that does not bring the results we hoped. For this reason, it would not make sense to save a model that does not reach certain minimum standards in terms of metrics, so we use a *ConditionStep* to check the results of the evaluation before proceeding further with the pipeline.

4.7.1 ConditionStep

Using the *ConditionStep*, we can decide to continue with the pipeline, so register the model and deploy it on an endpoint. The condition we evaluate is based on the binary classification metrics we computed inside the evaluation step; they can now be accessed through the generated property file called `evaluation.json`.

For the pipeline to continue, we want a certain metric, decided by the data scientist, to be greater than or equal to a certain threshold, also defined by the data scientist. To achieve this, we use the condition `ConditionGreaterThanOrEqualTo`. In particular, the configuration is the following:

```
ConditionGreaterThanOrEqualTo(
    left=JsonGet(
        step_name=step_eval.name,
        property_file=evaluation_report,
        json_path=
        f"binary_classification_metrics.{target_measure}.value"
    ),
    right=threshold_register,
)
```

Where `target_measure` and `threshold_register` are obtained from the configuration file.

After we define the condition, we can create the *ConditionStep* where we include as `if_steps` the model registration step and the model deployment step. As `else_steps` we instead have a *FailStep*, which blocks the execution of the pipeline if the threshold is not met.

4.7.2 ModelStep

After the *ConditionStep* evaluates the condition to be true, we can register the model on the Model Registry. First, we create an object of type *Model*, which we configure by passing as the `image_uri` our custom training image, as `model_data` we pass the trained model that we obtained as an output of the training step using its properties attribute. This creates a data dependency. Then we also need to remember to pass as `sagemaker_session` the object of type `PipelineSession`, since we need to execute the `model.register()` method. For this method, other than the parameters that we saw before, we also need to configure `approval_status="Approved"` since we will need the model in this status to be able to create the monitoring schedule in the next step automatically; we also remember to pass `inference_instance_type` to the `inference_instance` attribute.

Two more parameters need to be configured; in fact, we need to register the metrics we computed before on the registry (data quality and bias). In order to do so, we need to configure two more parameters, which are:

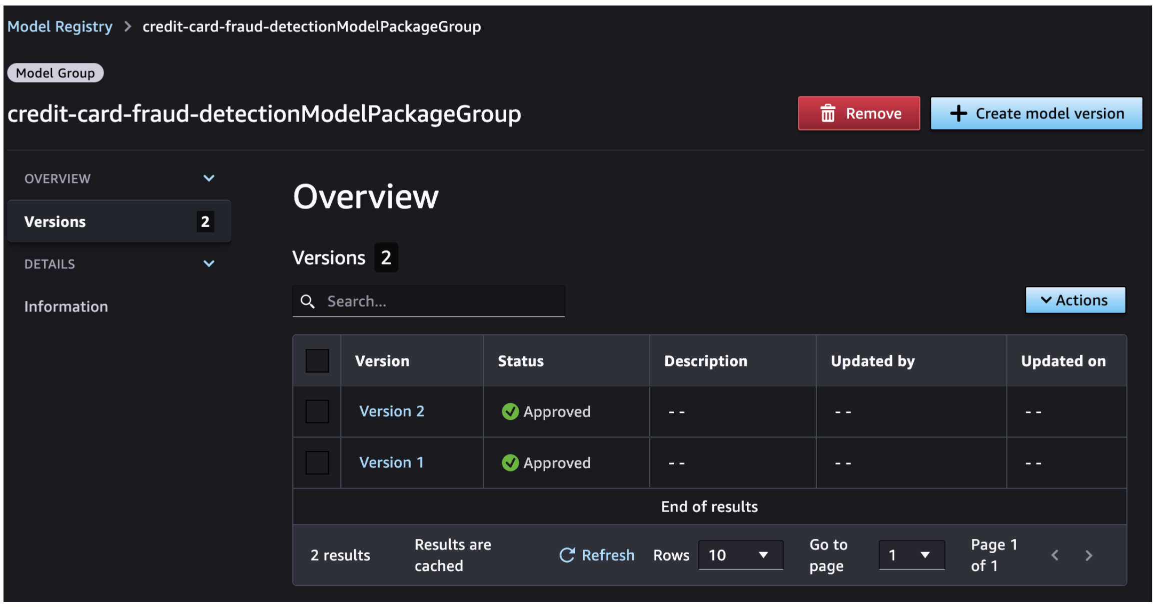
```
model_metrics=model_metrics,
drift_check_baselines=drift_check_baselines,
```

With `model_metrics`, we register the data quality statistics that we computed with the *DataQualityCheckStep*, so they can be used later during model monitoring to compare the baseline computed on the training dataset with the inputs captured by the endpoint. In `model_metrics`, we also store the metrics we computed with the evaluation step, which will appear in the Model Registry, as shown in Figure 4.2. To pass these metrics, we create an object of type *ModelMetrics*, which we initialize with objects of type *MetricsSource*. We initialize the latter with the S3 URI of the file we want to store and its content type.

On the other hand, with `drift_check_baselines`, we register data bias statistics that we computed with the *ClarifyCheckStep*. These would be used if we also monitored data bias. These metrics also appear in the Model Registry, as shown in Figure 4.3. To pass these metrics, we create an object of type *DriftCheckBaselines*, which we initialize with objects of the same type that we saw before: *MetricsSource*. We can see an example in the following code snippet:

```
model_metrics = ModelMetrics(
        model_data_statistics=MetricsSource(
            s3_uri=
            step_data_quality_check.\
            properties.CalculatedBaselineStatistics,
            content_type="application/json",
        ),
        model_data_constraints=MetricsSource(...)
    )

    drift_check_baselines = DriftCheckBaselines(
        bias_pre_training_constraints=MetricsSource(
        s3_uri=
        step_data_bias_check.\
        properties.\
        BaselineUsedForDriftCheckConstraints,
            content_type="application/json",
        )
    )
```



Model Registry > credit-card-fraud-detectionModelPackageGroup

Model Group

credit-card-fraud-detectionModelPackageGroup

Overview

Versions 2

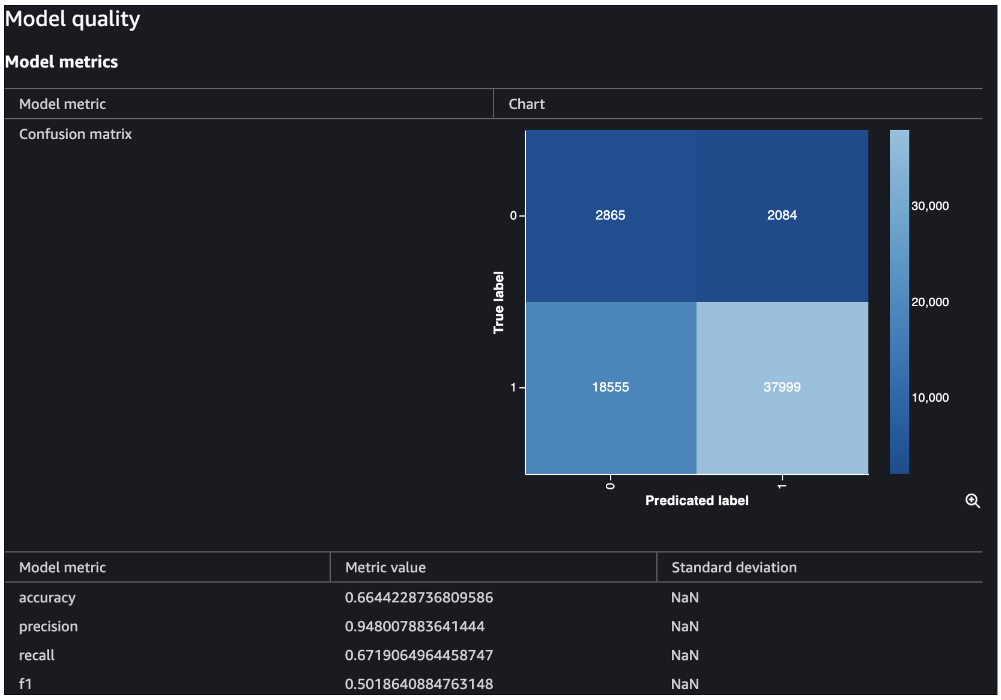
	Version	Status	Description	Updated by	Updated on
	Version 2	✓ Approved	- -	- -	- -
	Version 1	✓ Approved	- -	- -	- -

End of results

2 results Results are cached Refresh Rows 10 Go to page 1 Page 1 of 1

Figure 4.1: The Model Registry with some model packages

The result of the `register()` method is used to finally create the *ModelStep*, which will register our model when executed. We can see an example of the Model Registry with multiple versions of the model (model packages) in Figure 4.1.



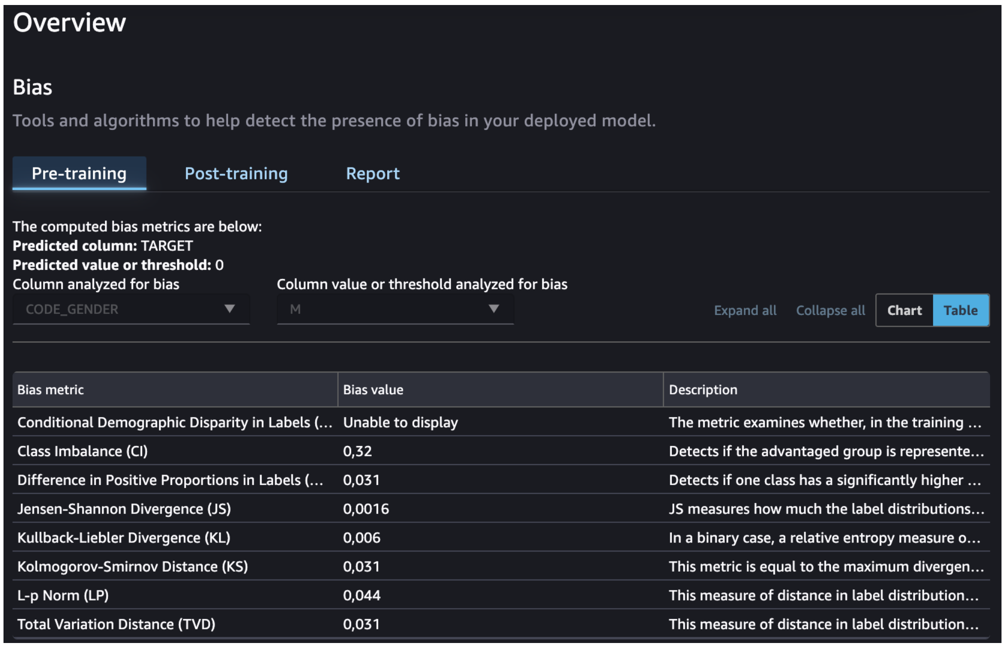
Model quality

Model metrics

Model metric	Chart
Confusion matrix	

Model metric	Metric value	Standard deviation
accuracy	0.6644228736809586	NaN
precision	0.948007883641444	NaN
recall	0.6719064964458747	NaN
f1	0.5018640884763148	NaN

Figure 4.2: Evaluation metrics on the model registry



Overview

Bias

Tools and algorithms to help detect the presence of bias in your deployed model.

The computed bias metrics are below:
Predicted column: TARGET
Predicted value or threshold: 0
Column analyzed for bias: CODE_GENDER
Column value or threshold analyzed for bias: M

Bias metric	Bias value	Description
Conditional Demographic Disparity in Labels (...	Unable to display	The metric examines whether, in the training ...
Class Imbalance (CI)	0,32	Detects if the advantaged group is represente...
Difference in Positive Proportions in Labels (...	0,031	Detects if one class has a significantly higher ...
Jensen-Shannon Divergence (JS)	0,0016	JS measures how much the label distributions...
Kullback-Liebler Divergence (KL)	0,006	In a binary case, a relative entropy measure o...
Kolmogorov-Smirnov Distance (KS)	0,031	This metric is equal to the maximum divergen...
L-p Norm (LP)	0,044	This measure of distance in label distribution...
Total Variation Distance (TVD)	0,031	This measure of distance in label distribution...

Figure 4.3: Bias metrics on the model registry

4.8 Model deployment

After registering the model, the next and last step of the pipeline is a *LambdaStep*, with which we perform many operations linked to model deployment, which are, in order:

1. Deploy the model on an endpoint
2. Activate model monitoring
3. Enable notifications when drift is detected
4. Enable retraining when drift is detected

We will talk about each of these points in the following sections. Before moving to the implementation, we will talk about the highest level of this stack, which is the *LambdaStep* itself. We discussed this step in Section 3.6.9; now we will describe its configuration, inputs, and outputs and then talk about the code the Lambda function will execute.

Before creating the *LambdaStep*, we have to instantiate the object of type *Lambda*, which we configure in the following way:

- `execution_role`: For the Lambda to be executed, we created an ad-hoc role to be used only within this context.
- `script`: the path of the script that will be executed by the *Lambda* is configurable via the `config.yaml` file.
- `handler`: This is created before instantiating the *Lambda* object and is a concatenation of the name of the file that contains the script (without extension) and the name of the function that will be executed inside the script, also configurable via `config.yaml`.

After configuring the *Lambda* object, we can create the *LambdaStep*, whose inputs are:

- `bucket_name, s3_logs_path`: we need these parameters because inside the code we have to save the log file we produce during the execution on S3.
- `use_case`: This is the use case configured at the beginning of the pipeline. It is used to generate the names for the endpoint, configurations, monitoring schedule, etc.
- `model_package_group_name`: This is used to retrieve the model from the Model Registry so it can be created and deployed on an endpoint.

- `execution_role, events_execution_role`: These two parameters are used to execute SageMaker functions inside the code. The former is the role that we also use in the pipeline; the latter is the role that we created ad-hoc to be able to execute the automatic retraining of the model.
- `train_image`: This parameter is the training image used to train the current model and will be passed again to the pipeline when it is retrained.
- `monitoring_report_path`: This parameter represents the path on S3 where the monitoring report will be saved if violations are detected (drift).
- `features_to_monitor`: This parameter represents the features we should be alarmed if they drift.
- `pipeline_arn`: This parameter represents the ARN of the pipeline that will be executed again when drift is detected. We use the ARN of the current pipeline.

We can now discuss the operations that the Lambda function executes. Throughout the execution, we will use methods that belong to the AWS Python SDK, which is called Boto3, and is used to create, configure, and manage AWS services. We will create a client for each service we need and use its methods to perform the required operations.

4.8.1 Deploy the model on an endpoint

To deploy the model on an endpoint, the first operation is to obtain the latest model from the model package. Before that, we create the Boto3 client that we need for these operations with: `sm_client=boto3.client('sagemaker')`. To obtain the latest model package, we perform `sm_client.list_model_packages`, which we configure with the name that we passed from the inputs, sort the results descending by creation time and say that we want only one object. The result that we obtain gives us the ARN of the model version on the Model Registry, and we use it as a parameter of the method `sm_client.describe_model_package`, with which we obtain the inference instance type that we configured when we registered the model. Now, we create the SageMaker model with the ARN we obtained before and name it using the following pattern: `model-{use_case}-{datetime}`, for versioning reasons. To create the model, we use `sm_client.create_model`. The model can also be created because we registered it on the Model Registry with approval status "Approved", so we do not need to do it by hand.

Once the model has been created, we can move on to the endpoint creation. To create the endpoint, we first need to create its configuration, but if it already exists, we remove it to update it to a new one. To check if a configuration already exists, we use the method `sm_client.list_endpoint_configs`, which, as with the model packages, we configure to obtain the latest endpoint configuration with a certain name. The name of the endpoint configuration follows a specific pattern, which is `{use_case}-endpoint-config`. If at least one endpoint configuration is found, we delete it with `sm_client.delete_endpoint_config` and create it again using `sm_client.create_endpoint_config`. To create an endpoint configuration, we specify:

- instance type: We use the instance type we retrieved from the model package.
- model name: We use the model name that we created before.
- configuration for data capture: Data capture is fundamental for monitoring, as we specify that we want to capture the data that is fed into the endpoint to make predictions. The model monitor will use this data to compare with the baseline computed at the beginning of the pipeline and detect drift. To configure data capture, we pass a dictionary which, in our case, contains the following entries:
  - enable data capture: Whether data capture should be enabled.
  - sampling percentage: What is the fraction of data that will be stored, we specified 100%, to store all the requests that arrive at the endpoint.
  - destination S3 URI: Where the captured data should be stored.
  - content type: What is the format of this data, we specified to accept data in both CSV and numpy formats.
  - capture options: Whether to store input data, model predictions, or both. We set both.

After creating the endpoint configuration, we check if an endpoint exists using the same procedure we used before, with `sm_client.list_endpoints`, searching for an endpoint name that follows the pattern: `{use_case}-endpoint`. If an endpoint already exists, we update its configuration with `sm_client.update_endpoint`. Otherwise, we create it by passing the configuration with `sm_client.create_endpoint`.

4.8.2 Activate model monitoring

After creating the endpoint, we can create the monitoring schedule, which will be configured to run at a specific interval. Before creating the schedule, we need to notice that as the endpoint is created or updated, it will be in status "Updating" or "Creating", which does not allow us to set the schedule on the endpoint. In order to solve this problem, I used the method `sm_client.describe_endpoint` to obtain the status of the endpoint and periodically checked (every 60 seconds) if the status changed or not by calling the same method again. As soon as the status changes to "In service" we can continue with the creation of the schedule. We allow only one monitoring schedule per endpoint, with a name that follows the pattern: `{use_case}-monitoring-schedule`. Before creating a new one, we first delete the existing one. If a schedule exists, it will be deleted; otherwise, the method `sm_client.delete_monitoring_schedule` throws an exception. In either case, we finally create our monitoring schedule with the method `sm_client.create_monitoring_schedule`. To configure the schedule, we use the parameter `MonitoringScheduleConfig`, to which we pass a dictionary with the following entries:

- `ScheduleConfig`: We use this entry to set the `ScheduleExpression`; in our case, we want to run the monitoring job hourly, so we pass `cron(0 * ? * * *)`.
- `MonitoringJobDefinition`: With this entry, we define what baselines the job should use to compare new data. We take the baselines from the model we obtained from the Model Registry since we stored this information when we registered the model.
- `MonitoringInputs`: This entry defines where the monitoring job will take its inputs. We specified the endpoint that we just created.
- `MonitoringOutputConfig`: With this entry, we define the path on S3 where the violations report will be saved if any are found during the monitoring process.
- `MonitoringResources`: We define the technical configuration of the underlying machines, such as the number of instances or instance type.
- `MonitoringAppSpecification`: With this entry we specify two important configurations:
  - What is the image URI that the job will execute. In our case, we pass the default image, but we could decide to create our own monitoring image to execute personalized drift methods.
  - a preprocessor, if necessary, that will preprocess input data before it is passed on to the monitoring job. We used a preprocessor since there are problems with the CSV format, so we need to reshape the input.

4.8.3 Enable notifications when drift is detected

After creating the monitoring schedule, we can allow data scientists or clients to receive notifications when drift is detected. In order to do so, we will use

CloudWatch Alarms and Amazon Simple Notification Service (SNS). To perform actions when drift is detected, we have to configure one or more alarms, each based on a feature we want to monitor drift on. The basic characteristic of an alarm is the metric we monitor (the feature in our case), the frequency with which the metric is evaluated, a comparison operator, and the threshold; we will see more specific parameters later. We can also specify actions that happen on state changes. There are three possible states: insufficient data, OK, and alarm. If we do not send data to the endpoint, the metric cannot be evaluated as there is no data, so we are in the first state. When data is sent, the monitoring job computes baseline drift metrics, which are, for example, how distant the two distributions are; this metric is what is evaluated by Cloudwatch alarms, and if the condition we specified through the threshold and the comparison operator is true, the status is "In alarm". If the condition is not satisfied, e.g. the metric is lower than the threshold, the status is "OK". The action is executed only once, on state change; if the alarm persists in "Alarm" status, the action is not executed again.

We can configure an action to notify us when drift is detected. Now, Amazon SNS comes into play. The first thing we need to do is to create a topic and then subscribe to it. In SNS, we have Publishers who send messages to a topic; the clients subscribe to the topic by defining how to receive notifications (e.g. email, SMS, etc) and where (e.g. email address, phone number, etc). Once a Publisher sends a message to the topic, it is broadcasted to the subscribers through the method that they specified when they subscribed to the topic. To enable these notifications, we specify the topic among the actions, for example, the "In alarm" actions.

To use these services, we must first create their respective Boto3 clients with `cloudwatch=boto3.client('cloudwatch'), sns=boto3.client('sns')`. After that, we first check if a topic exists as we did before, by listing the topics using the method `sns.list_topics` and checking if our topic exists. If it does not exist, we create a new one using `sns.create_topic`, in which we specify its name that follows the pattern `data-it-ena-mlops-{use_case}-drift-topic`. The subscription to said topic is done manually in the AWS console. After creating the topic, we can finally create one alarm for each metric, with `cloudwatch.put_metric_alarm` and configure it with the following parameters:

- alarm name: The alarm is named after the pattern

```
{use_case}-drift-alarm-{metric_name}
```

- Alarm actions: We insert the ARN of the topic we have just created. We retrieve the ARN from the response of the method `crete_topic`. We did not specify any action in Ok actions or InsufficientData actions.

- Information on how to aggregate data points: This information is important since when the monitoring job is executed, it computes baseline drift metrics, which are saved as data points inside CloudWatch. We can then decide the length used each time the specified metric is evaluated (Period), the number of periods over which data is compared to the specified threshold (EvaluationPeriods), the number of data points within the Evaluation Periods that must be breaching to trigger the alarm (DatapointsToAlarm) and the metric data aggregations over specified periods (Statistic). We checked the metric each hour with an evaluation period of 1, with one data point to alarm and average as statistic.

- Comparison operator and threshold: We specified as threshold 0.1 and as comparison operator "GreaterThanThreshold". This means that if the aggregated data points exceed the threshold, the alarm goes into "Alarm" status.

4.8.4 Enable retraining when drift is detected

After enabling the notifications service for the user, we can finally enable the retraining. For this purpose, we need the alarms that we created earlier, and we will use an AWS service called *EventBridge*. EventBridge is a serverless service that allows us to interconnect different applications through events. For example, we can start the pipeline training when a new file is uploaded on S3, interconnecting SageMaker pipelines with AWS S3. In our case, we will use EventBus, a router that receives events from one or more sources and delivers them to one or more targets. To an EventBus, we associate rules that evaluate events as soon as they are received. Each rule checks if the event matches a pattern specified by the rule itself and sends the event to the target. Since we are using a new AWS service, we must create its Boto3 client, which is `event_bridge = boto3. client('events')`.

In order to create the rule, we need to define an object beforehand, the *event pattern*, which defines what events EventBridge will select to send to the targets. An event pattern has the following structure:

```
event_pattern = {
  "source": [
    <source>
  ],
  "detail-type": [
    <detail-type>
  ],
  "resources": <resources>
}
```

We specify the `source`, which is the service that produced the event we want to capture; the `detail-type` specifies the event name generated by the source; `resources` specifies the ARNs of the resources involved in the event.

The previous snippet is what we used in our code, and we configured it so the `source` parameters contains the value "aws.cloudwatch", the `detail-type` contains "CloudWatch Alarm State Change" and finally `resources` contain the variable `alarm_arns`. This variable is created even before the event pattern and contains the ARNs of the alarms. In order to retrieve these ARNs, we use the method `cloudwatch.describe_alarms_for_metric`, which returns us a description of the alarm by searching for a metric; we search for the same metrics on which we created the alarm before.

To sum up the event pattern, we configured it such that we want to capture events sent by CloudWatch that represent a state change into "Alarm" status and that are generated by the alarms created previously.

As we can see, the value of each of these fields is an array. The event pattern matches the received event if any value in the array matches the value in the event, so it resembles an OR condition.

We can now create the rule with the method `event_bridge.put_rule`, naming the rule with the pattern: `data-it-ena-mlops-{use_case}-retrain-pipeline`.

We are not finished yet, since we created the rule but we still need to attach targets, which will receive the event after it has been generated and captured. To attach targets, we use the method `event_bridge.put_targets`, which we configure by specifying the name of the rule that we have just created and an array of targets. Each target is a dictionary identified by an id, which in our case will be `{use_case}-retrain-pipeline`. We also need to specify a role that will be used for this target when the rule is triggered and the ARN of the target, in our case, the pipeline itself; the role instead was created ad-hoc to be able to start the pipeline. Since we are specifying a pipeline as a target, we can also pass pipeline parameters so we can change the behavior of the pipeline when it is retrained if we want to. In our case, we decided to pass the same parameters that were passed when the pipeline was started with the `start()` method, as we want the pipeline to repeat the same execution, just on new and updated data.

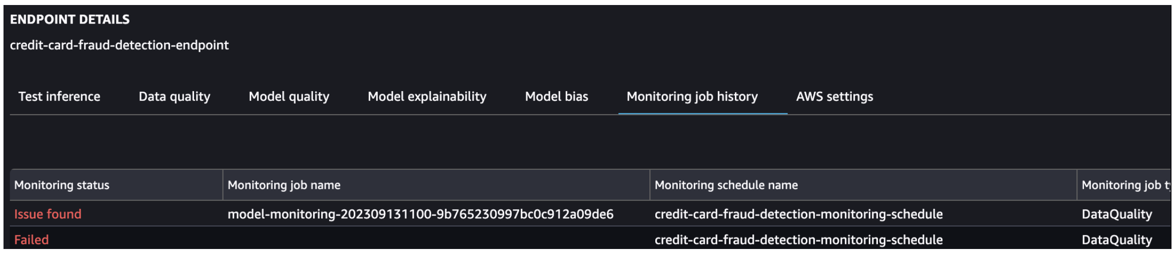
As of now, the retraining is performed on the same data, so it is exposed to the same drift. We did not test it on a real case scenario, as we could not modify the underlying data before the pipeline restarted. Ideally, the data we capture from the endpoint should be labeled so it is available for new training.

The last step executed by the Lambda function is to upload the log file on S3, using the information we passed as inputs to the Lambda.

4.9 Model monitoring

The pipeline description can be considered over, as there are no more steps left, and with the last one, we enabled anything necessary to perform monitoring, notifications, and retraining. For this reason, we can talk more in-depth about model monitoring, how it is performed, and how it detects drift.

When the pipeline terminates its execution without errors, the monitoring job is scheduled to follow the schedule we configured. The monitoring job will be always executed, whether the endpoint receives inputs or not. In SageMaker Studio, we can see the monitoring jobs history in the Endpoint details, where we can see the result of the monitoring job and when it was executed. We can see an example in Figure 4.4.



Monitoring status	Monitoring job name	Monitoring schedule name	Monitoring job t
Issue found	model-monitoring-202309131100-9b765230997bc0c912a09de6	credit-card-fraud-detection-monitoring-schedule	DataQuality
Failed		credit-card-fraud-detection-monitoring-schedule	DataQuality

Figure 4.4: An example of monitoring history

The monitoring job can produce three results, which are:

- Failed: The job has failed its execution; the most common reason is that it did not find any data to analyze because the endpoint did not receive inputs;
- No issues: The monitoring job was executed successfully; after analyzing the input data captured from the endpoint, it did not find any issue;
- Issue found: The monitoring job ran successfully, but the input data captured by the endpoint had discrepancies with the baseline computed in the first steps of the pipeline.

When issues are found, we can inspect the job execution and find out what are the violations found and what are the features affected. Inside this detail page, there is also a link to a notebook, which, by inserting the ARN of the monitoring job, allows us to inspect the situation with graphs and tables. The violations are not only about drift, but can also be about different aspects and are stored in a file called `constraints_violations.json`. The path on S3 is configured when we create the monitoring schedule; in our case, we save it in the directory "report" of the current use case. This folder is further divided by year, month, day, and hour. The model monitoring job provides the following checks:

- `data_type_check`: We have a violation if the data types of the inputs are not the same found in the baseline dataset.
- `completeness_check`: We have a violation if the percentage of non-null items exceeds a certain threshold computed on the baseline dataset.
- `baseline_drift_check`: This is the violation that we are most interested in. We have a violation if the distribution distance between the inputs and the baseline dataset exceeds a certain threshold.
- `missing_column_check`: We have a violation if the number of columns in the input is less than the number of columns in the baseline dataset.
- `extra_column_check`: We have a violation if the number of columns in the input is more than the number of columns in the baseline dataset.
- `categorical_values_check`: We have a violation if there are more unknown values in the input than in the baseline dataset.

As we said before, to perform model monitoring, we use the default image URI that provides SageMaker, which we can find using the SageMaker Python SDK function `image_uris.retrieve`, passing to the parameter `framework` the value `model-monitor` and as a region our current region, which is `eu-west-1` (Ireland). The model monitor also emits CloudWatch metrics for each feature/column observed in the dataset. For numerical features, the pre-built container emits the metric `feature_data_{feature_name}`, which records the value that the feature assumed in the baseline dataset. For both numerical and string fields, instead, it emits two metrics, which are `feature_baseline_drift_{feature_name}` and `feature_non_null_{feature_name}`. The former represents the distance between the distributions of input and baseline datasets; the latter represents the number of non-null items. In particular, `feature_baseline_drift_{feature_name}` is the metric we will create the alarm on because it directly returns us the distance between the distributions, which we can quickly check with a threshold.

Regarding drift detection, the default model monitor computes the distance between two data distributions. We will now analyze the details of the method implementation. The method we will see is based on the statistical properties of data, comparing data distributions at different points in time. The method differs whether features are numerical or categorical, starting from the histogram computed to represent the data distribution:

- numerical features: The histogram is computed using ten bins, so all bins have the same width, namely:

$$\text{binWidth} = \frac{\max - \min}{10}$$

where $\max, \min$ represent the maximum and minimum values the feature assumes, respectively. The first bin is $\min + \text{binWidth}$ and so on.

- categorical features: The histogram is computed as the number of instances for each distinct value assumed by the feature.

To detect drift for numerical features, the default model monitor uses the Two-sample Kolmogorov–Smirnov test and the L-infinity test for categorical features.

4.9.1 Numerical features

For numerical features, the drift is detected using a method based on the Two-sample Kolmogorov–Smirnov test [24]. The KS test is a nonparametric [1] test of the equality of continuous one dimensional probability distributions [2]. Since in our case it is a two-sample KS test, we compare two samples, not one sample and one reference probability distribution (one sample KS). The two samples are the baseline dataset and the input dataset. It is used to quantify the distance between the empirical distribution functions of two samples. The null hypothesis is that the samples are drawn from the same distribution. The KS statistics is

$$D_{n,m} = \sup_x |F_{1,n}(x) - F_{2,m}(x)|$$

Where n, m are respectively the sizes of the first and second samples.

The test statistic is finally compared with the threshold that is set inside the `constraints.json` file.

4.9.2 Categorical features

For categorical features, we use a similar approach called L-infinity distance, also referred to as Chebyshev distance [25]. L-infinity distance is the infinity-norm between pair of vectors, which in our case represent frequency distributions of each feature. This distance is computed as the maximum between the absolute distances for each categorical value. The Chebyshev distance can be written as:

$$D_c(x, y) = \max_i(|x_i, y_i|)$$

Where x and y represent the two samples, i.e., the input and baseline distribution. The index i represents instead each categorical value.

[1]we make no assumptions on the population from which the sample is drawn, since we do not assume a particular distribution or if we do, we do not know its parameters

[2]one dimensional since the sample space is one dimensional (only one feature at a time)

4.10 Drift simulation

To check that model monitoring actually works, we simulated traffic on the endpoint by using a notebook instance that we launched through the SageMaker console. We use an object of type *Predictor* from the SageMaker Python SDK to perform predictions. The object is instantiated with the endpoint name and, most importantly, with a serializer and deserializer, so we can communicate with the endpoint by encoding the data that we feed to the endpoint as input and deserializing result data. As serializer, we use an object of type *CSVSerializer*; as deserializer, we use an object of type *CSVDeserializer*. To make a prediction, we use the method `predictor.predict(data=<payload>)`.

After instantiating the object we are ready to make predictions, so we simulate drift by changing the data distribution of two features that we chose for no particular reasons, one categorical and one numerical:

- The categorical feature is `FLAG_OWN_CAR`, which represents the fact that a person owns a car and can take values of `Y` or `N`. We change this feature so all the new instances that we send to the endpoint take value `Y`.
- The numerical feature is `AMT_ANNUITY`, which represents the loan amount, which has a mean around 27 715$. We changed this feature so all the new instances have the mean around 147 715$.

We can see a comparison between the old and new feature distribution for 100 samples in Figure 4.6 for the numerical feature, in Figure 4.5 for the categorical feature. After creating artificial drifted traffic, we wait for the monitoring job to be executed according to the schedule. When the job finishes, we can see that it found issues; by checking the details of the job, we can look at the features that created problems and the type of issue. An example is in Figure 4.7. In our case, the constraint violated is `feature_baseline_drift`; we can also see details of the violation. For example, for the feature `AMT_ANNUITY`, we notice that we have a problem since the baseline drift distance (0.9) exceeds the threshold of 0.1. In this details page, we also have a link to a notebook that we can access within Studio, which generates plots and tables to inspect problems more in-depth. An example of generated plot is in Figure 4.8, in which we can also observe the drift on the `AMT_ANNUITY` feature. Since we set up a CloudWatch Alarm, we can see in Figure 4.9 the alarm in the SageMaker console. Furthermore, we received an email alerting us of drift, and the pipeline started the retraining.

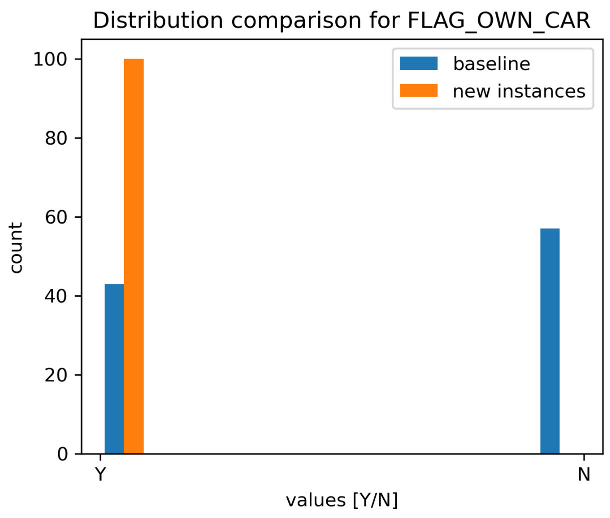


Figure 4.5: Drift distribution comparison for FLAG_OWN_CAR

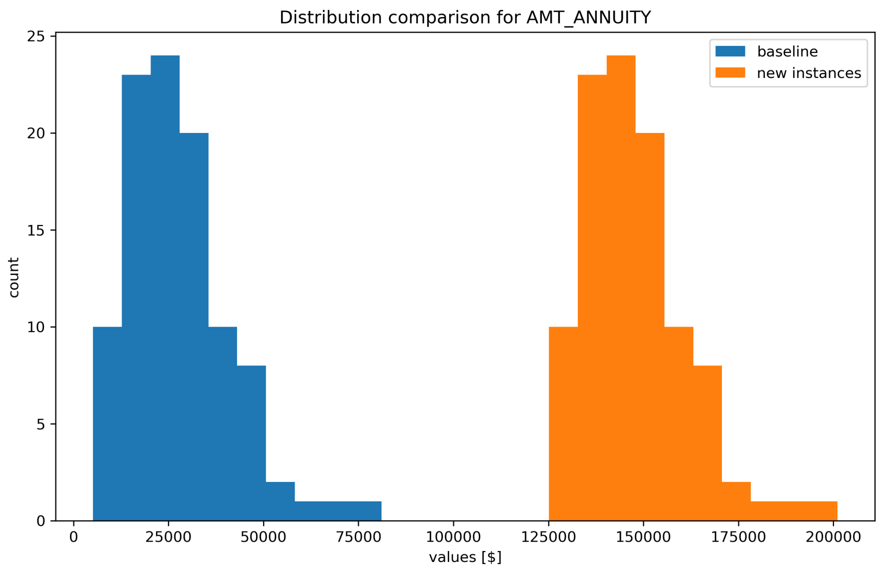
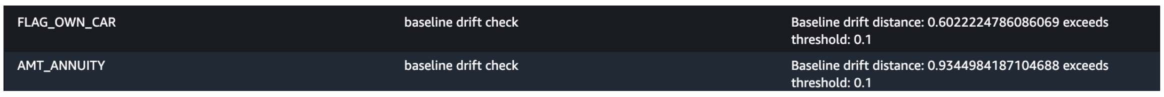


Figure 4.6: Drift distribution comparison for AMT_ANNUITY



FLAG_OWN_CAR	baseline drift check	Baseline drift distance: 0.6022224786086069 exceeds threshold: 0.1
AMT_ANNUITY	baseline drift check	Baseline drift distance: 0.9344984187104688 exceeds threshold: 0.1

Figure 4.7: Violation details in SageMaker studio

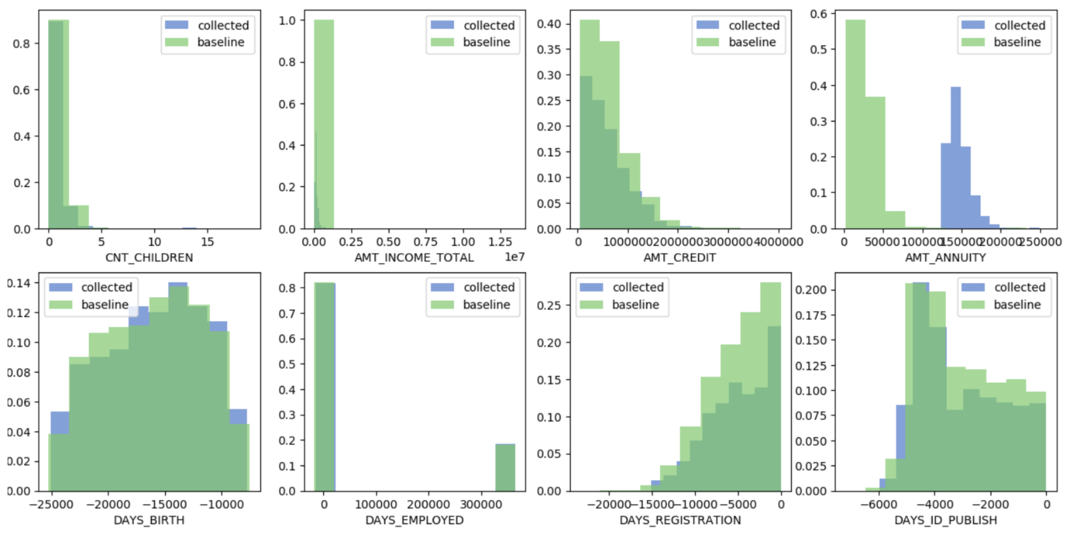
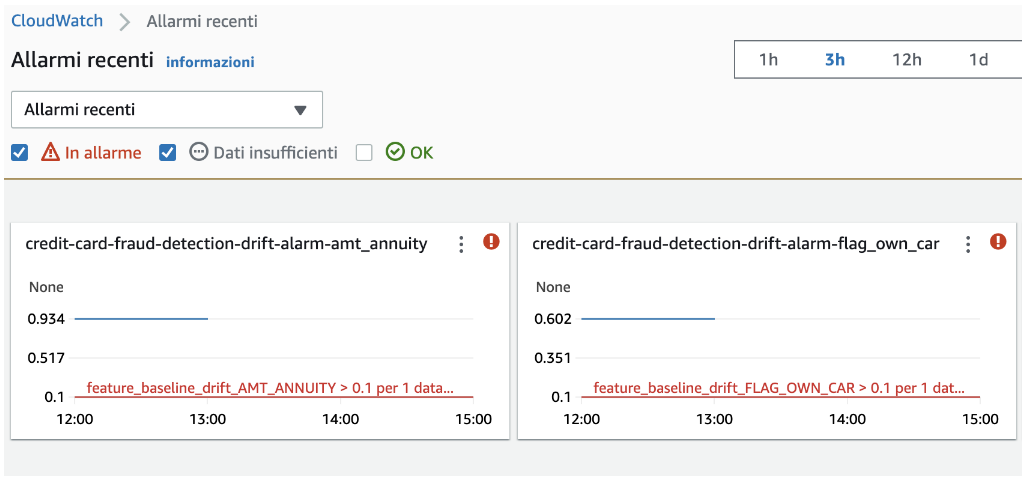


Figure 4.8: Plot of collected and baseline distributions for each feature


CloudWatch > Allarmi recenti

Allarmi recenti informazioni

1h 3h 12h 1d

Allarmi recenti

In allarme Dati insufficienti OK

credit-card-fraud-detection-drift-alarm-amt_annuity

None

0.934

0.517

0.1 feature_baseline_drift_AMT_ANNUITY > 0.1 per 1 data...

12:00 13:00 14:00 15:00

credit-card-fraud-detection-drift-alarm-flag_own_car

None

0.602

0.351

0.1 feature_baseline_drift_FLAG_OWN_CAR > 0.1 per 1 dat...

12:00 13:00 14:00 15:00

Figure 4.9: CloudWatch alarm

Chapter 5

Evaluation

5.1 Value provided to potential clients

We can now evaluate our proposed solution considering its value to potential clients. In particular, we can find many strengths, which are:

- Process standardization: By executing the pipeline, each data scientist follows the same process from the beginning to the end of the ML life cycle by writing and training models in a definite and "standard" way, which is also repeatable.
- High traceability and fast reproducibility: These two characteristics are fundamental for debugging and creating baselines from models that were trained in the past. Furthermore, we can easily compare model versions, even if they are very distant in time, e.g. years.
- Ease of retraining for different model versions: the pipeline allows, other than retraining a model, for example, with an old training image, also to execute a whole set of operations that may be considered "boring" by data scientists. These operations include registering on the model registry, tracking metrics, deploying on an endpoint, monitoring, and such. These operations are, in fact, usually seen as "extra" work and handled manually with scripts or notebooks launched when needed. This also allows us to easily compare different model versions since we can train two different model versions on the fly and compare them in terms of evaluation metrics, predictions, and so on.
- Possibility to define strategies to handle drift: This is a significant point since drift should not be handled in the same way for all use cases, but each brings its peculiarities and challenges that have to be addressed with their strategy, for example, we should decide what are the features to monitor, what the drift threshold, whether to automatically retrain the pipeline and other operations.

Furthermore, another critical characteristic of our solution is that drift is handled by design; by doing this, we can reduce the chances of deteriorating the performances.

- Possibility of reuse with other use cases by having only one "change point", the file `config.yaml`; we also call this *modularity*. Other than being modular, the configuration needed to adapt the pipeline to a new use case is not heavy, but there are only a few entries that need to be changed, in particular:
  - for the basic preprocessing step, we only have to change the name of the dataset file, specify the columns to keep, the name of the label column, and the value that represents the positive outcome;
  - for the data bias check, we only have to change two entries: the name of the sensitive attribute column and the disadvantaged value;
  - for the bias and evaluation condition, we could change the metrics and respective thresholds;
  - for the model deployment, we have to specify what are the features to monitor.

5.2 Fairness and Legal implications

Now that we have a clear view of the whole pipeline, we can discuss its implications regarding fairness and the upcoming AI Act, which will be approved by the end of the year.

The first point on which we can evaluate our solution in terms of fairness is that it is effective in preemptively blocking the training of the model if the training dataset does not respect fairness measures decided by the data scientist, given that any bias in a model's predictions derive from a biased dataset. In this way, understanding the problems with a dataset lets us correct any bias so the resulting model is fairer than what it could have been.

Another important aspect is that the model monitor automatically creates a bias report, stating the bias metrics computed on the dataset and their values. It also provides graphs presenting a high-level view of the dataset based on the sensitive attribute, allowing us, for example, to see any imbalance in the dataset. This report can be used as a self assessment or as a starting point to produce one since this document is mandatory for high risk applications, given that companies must be able to demonstrate that their application does not discriminate.

Moreover, there are many links with the previous section about the value provided to clients, in particular with:

- Traceability and reproducibility: These two aspects are essential for the AI Act since they are two actions that improve transparency and accountability, which are requisites of the AI Act.
- Drift detection: Drifting in data can introduce new biases (which we can then detect) or degrade the model's performance. If the model is used in high risk applications, a degradation in performance could mean that decisions based on wrong predictions harm people affected.

Chapter 6

Conclusions and future works

MLOps has gained much interest in the ML field, especially in the industrial context, since it helps reduce technical debt and standardize processes. Two related aspects, concept drift and fairness, nicely link with MLOps to produce a solution encompassing all three.

This thesis focuses on developing a pipeline on Amazon AWS that can incorporate the practices of MLOps while also focusing on concept drift and fairness in the scope of the AI Act. Concept drift is addressed by monitoring the deployed model, notifications and retraining. At the same time, the AI Act is considered in the bias detection methods and following evaluation based on fairness measures.

The main accomplishments of this project are:

- The successful development of a pipeline to manage the end-to-end life cycle of an ML application, from dataset to deployed model on an endpoint.

- The solution is production-ready, which means that it can already be sold as-is to a potential client who may need it, since it includes the critical stages of any ML application development.

- The solution brings value to clients while foreseeing a correct usage when the AI Act is enforced by law.

The main limitations of this project are:

- The solution is strongly linked with the platform it was developed on. If a client uses another platform, the pipeline must be rewritten, or the client should switch platforms.

- The solution assumes that before starting the pipeline process, the data scientist already knows the columns that must be selected from the dataset or that the dataset has already been preprocessed for feature engineering.
- The data scientist should understand the fairness measures that are computed and should be able to clearly define thresholds based on the dataset used and the objective of the application. Furthermore, the data scientist should also select the columns of the dataset that are monitored (to receive notifications or make the pipeline restart) based on the objectives of the application and the use case.

In the future, we can foresee some interesting developments that can enhance the solution and make it more powerful and valuable:

- Implement a more advanced method of detecting drift: Since SageMaker allows us to create a custom Docker image with monitoring capabilities, the client could think to implement its own drift detection methods based on other statistical tests that might be more fit for a specific use case.
- As of now, the retraining is performed on the same dataset used for the model training. We need a more advanced mechanism to update the dataset with the new instances (already captured by the endpoint) and label them. This way, the retraining has a meaning, and the model is updated with new data. The retraining proposed by our solution is a proof of concept to show that it is a doable possibility.
- We can also improve our solution by enabling scheduled retrainings, which update the model whether there is drift or not. It is assumed that the training dataset is updated with new labelled instances.
- As of now, the bias monitor cannot be customized, and we are not able to create our own custom Docker image like we can for the drift monitor, so we cannot compute custom bias metrics. Instead, We can think of developing a processing job that computes custom fairness measures and outputs a property file that we can evaluate. This would need a fair amount of work to integrate with the current solution.

Bibliography

[1] IBM. *IBM Global AI Adoption Index 2022*. Tech. rep. 2022 (cit. on p. 4).

[2] Robert J. Glushko. «Seven ways to make a data science project fail». In: *Data and Information Management* 7.1 (2023). Special Issue on Data Science and Information Science., p. 100029. ISSN: 2543-9251. DOI: `https://doi.org/10.1016/j.dim.2023.100029`. URL: `https://www.sciencedirect.com/science/article/pii/S2543925123000037` (cit. on p. 4).

[3] David Sculley et al. «Hidden technical debt in machine learning systems». In: *Advances in neural information processing systems* 28 (2015) (cit. on p. 4).

[4] Georgios Symeonidis, Evangelos Nerantzis, Apostolos Kazakis, and George A Papakostas. «Mlops-definitions, tools and challenges». In: *2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE. 2022, pp. 0453–0460 (cit. on p. 5).

[5] Sanjeev Sharma. *The DevOps adoption playbook :* Includes index. Indianapolis, IN : Wiley, 2017. URL: `https://doi.org/10.1002/9781119310778` (cit. on p. 5).

[6] Monika Steidl, Michael Felderer, and Rudolf Ramler. «The pipeline for the continuous development of artificial intelligence models—Current state of research and practice». In: *Journal of Systems and Software* 199 (2023), p. 111615. ISSN: 0164-1212. DOI: `https://doi.org/10.1016/j.jss.2023.111615`. URL: `https://www.sciencedirect.com/science/article/pii/S0164121223000109` (cit. on p. 5).

[7] Sridhar Alla and Suman Adari. *Beginning MLOps with MLFlow: Deploy Models in AWS SageMaker, Google Cloud, and Microsoft Azure*. Jan. 2021. ISBN: 978-1-4842-6548-2. DOI: `10.1007/978-1-4842-6549-9` (cit. on p. 6).

[8] Kristian Kersting and Ulrich Meyer. «From Big Data to Big Artificial Intelligence?» In: *KI - Künstliche Intelligenz* 32 (2018), pp. 3–8. URL: `https://api.semanticscholar.org/CorpusID:256072197` (cit. on p. 6).

[9] Douglas Laney. *3D Data Management: Controlling Data Volume, Velocity, and Variety.* Tech. rep. META Group, Feb. 2001. URL: http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf (cit. on p. 6).

[10] Wei Fan and Albert Bifet. «Bifet, A.: Mining Big Data: Current Status, and Forecast to the Future. SIGKDD Explorations 14(2), 1-5». In: *ACM SIGKDD Explorations Newsletter* 14 (Apr. 2013), pp. 1–5. DOI: 10.1145/2481244.2481246 (cit. on p. 6).

[11] Gregory Ditzler, Manuel Roveri, Cesare Alippi, and Robi Polikar. «Learning in Nonstationary Environments: A Survey». In: *Computational Intelligence Magazine, IEEE* 10 (Nov. 2015), pp. 12–25. DOI: 10.1109/MCI.2015.2471196 (cit. on pp. 6, 8).

[12] Jie Lu, Anjin Liu, Fan Dong, Feng Gu, João Gama, and Guangquan Zhang. «Learning under Concept Drift: A Review». In: *IEEE Transactions on Knowledge and Data Engineering* 31.12 (2019), pp. 2346–2363. DOI: 10.1109/TKDE.2018.2876857 (cit. on pp. 6, 7).

[13] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Hamid Bouchachia. «A Survey on Concept Drift Adaptation». In: *ACM Computing Surveys (CSUR)* 46 (Apr. 2014). DOI: 10.1145/2523813 (cit. on pp. 6, 7).

[14] Alexey Tsymbal. «The Problem of Concept Drift: Definitions and Related Work». In: (May 2004) (cit. on p. 8).

[15] Hang Yu, Tianyu Liu, Jie Lu, and Guangquan Zhang. *Automatic Learning to Detect Concept Drift.* 2021. arXiv: 2105.01419 [cs.AI] (cit. on p. 8).

[16] Ben Green. «The flaws of policies requiring human oversight of government algorithms». In: *Computer Law &amp Security Review* 45 (July 2022), p. 105681. DOI: 10.1016/j.clsr.2022.105681. URL: https://doi.org/10.1016%2Fj.clsr.2022.105681 (cit. on p. 9).

[17] Wikipedia. *COMPAS (software) — Wikipedia, The Free Encyclopedia.* http://en.wikipedia.org/w/index.php?title=COMPAS%20(software)&oldid=1169881982. [Online; accessed 31-August-2023]. 2023 (cit. on p. 9).

[18] Ellora Israni. «When an Algorithm Helps Send You to Prison». In: *New York Times* (Oct. 2017). URL: https://www.nytimes.com/2017/10/26/opinion/algorithm-compas-sentencing-bias.html (cit. on p. 9).

[19] European Parliament News Press. *EU AI act: First regulation on artificial intelligence: News: European parliament.* June 2023. URL: https://www.europarl.europa.eu/news/en/headlines/society/20230601STO93804/eu-ai-act-first-regulation-on-artificial-intelligence (cit. on p. 9).

[20] *Kubeflow*. [Online; accessed 02-September-2023]. URL: `https://www.kubeflow.org` (cit. on p. 16).

[21] Shashank Prasanna and Alex Chung. *Introducing Amazon SageMaker Components for Kubeflow Pipelines*. Accessed on 12 July, 2023. 2020. URL: `https://aws.amazon.com/it/blogs/machine-learning/introducing-amazon-sagemaker-components-for-kubeflow-pipelines/` (cit. on p. 17).

[22] Zohar Karnin, Kevin Lang, and Edo Liberty. *Optimal Quantile Approximation in Streams*. 2016. arXiv: `1603.05346 [cs.DS]` (cit. on p. 26).

[23] AWS. *When should I extend a SageMaker container?* Accessed on 07 September, 2023. 2023. URL: `https://sagemaker-examples.readthedocs.io/en/latest/advanced_functionality/scikit_bring_your_own/scikit_bring_your_own.html#When-should-I-build-my-own-algorithm-container%3F` (cit. on p. 34).

[24] Wikipedia. *Kolmogorov–Smirnov test*. Accessed on 13 September, 2023. URL: `https://en.wikipedia.org/wiki/Kolmogorov%E2%80%93Smirnov_test` (cit. on p. 61).

[25] Cyrus Cantrell. «Modern Mathematical Methods for Physicists and Engineers». In: *Measurement Science and Technology* 12 (Nov. 2001), p. 2211. DOI: `10.1088/0957-0233/12/12/702` (cit. on p. 61).