Documentation - Pipeline AirFlow

Ernst Tim, Gallandat Théo, Guidetti Laetitia, Küenzi Jean-Daniel, Perez Yohann

1 Introduction

This project aims to set up an AirFlow pipeline to perform all the steps of a Machine Learning project. The pipeline should allow to download the data, preprocess it, train a model and make prediction. This approach allows to automate the process, make it reproducible and understand the advantages and disadvantages of using an AirFlow pipeline.

1.1 Airflow

Apache Airflow is a workflow management platform that allows to schedule, monitor, automate, and manage workflows. It is open source, Python native, and was developed by Airbnb in 2015. His principles are scalable, dynamic pipeline generation, easily extensible and explicit.

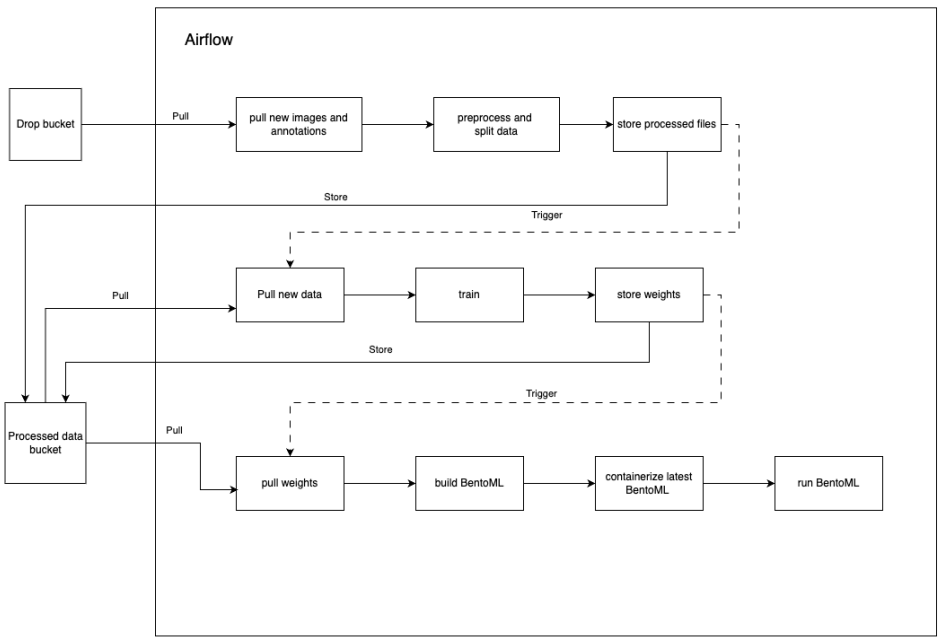
Workflows are represented as Directed Acyclic Graphs (DAGs). A DAG is composed of tasks that are individual units of work. Tasks can be of different types, such as Python functions, SQL queries, or Bash scripts. Tasks can be chained together, and dependencies can be defined between them. It is possible to make decisions based on the result of a task. There is a wide range of providers for AirFlow, which allow to connect to different services such as AWS, Google Cloud, Azure, etc.

1.2 Use case

The use case chosen for this project is the training of a YOLOv8 model to detect road signs in images. The goal is to train a model that can detect the road signs in the images and draw bounding boxes around them.

This is a common problem in the field of road safety and autonomous driving. It is an important task for the development of navigation and automatic driving systems. It allows to identify road signs on the road and to take the necessary measures according to the information they provide.

For this, the objective is to realize the following configuration:



This requires breaking down the process into several tasks that will be executed by the AirFlow pipeline. All tasks allow the completion of a complete Machine Learning project, from data retrieval to model evaluation through training. The pipeline is divided into 3 main DAGs: Preprocessing, Training, and Deployment.

1.3 Dataset

The dataset comes from Mapillary, a platform containing street images and map data from around the world. Mapillary Traffic Sign Dataset is a dataset of annotated road sign images in the form of bounding boxes. It contains 100,000 images annotated with 400 classes of road signs on 6 continents with a wide variety of weather conditions and brightness.

For each image, there is a JSON file containing the annotations of the road signs.

Image example:

2 Pipeline description

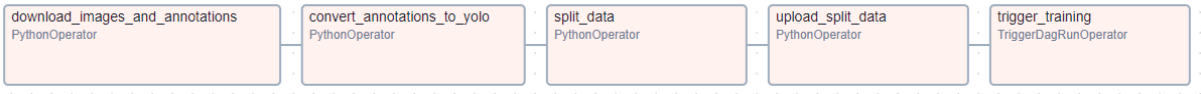
The pipeline is separated into 3 main DAGs:

- Preprocessing
- Training
- Deployment

This separation allows to better organize the tasks and improve the readability of the code. Each DAG is composed of several tasks that are executed in a specific order. In our case, the tasks are executed sequentially, with dependencies between them.

2.1 DAG – Preprocessing

This DAG is responsible for preprocessing the data. It must retrieve the raw data, preprocess it and save the results obtained. The tasks are as follows:



2.1.1 Task - Download data

The `download_images_and_annotations` task is responsible for downloading the raw data (images and annotations in json format) from a cloud storage (here we are using a Google Cloud Storage bucket). It will then save the data locally in a temporary directory.

The goal here was initially to have a trigger based on the arrival of new data in the cloud storage, but we did not manage to implement it. The data added to this "drop" bucket will be then used to retrain the model with new data.

2.1.2 Task - Annotations conversion

The `convert_annotations_to_yolo` task is responsible to take the newly downloaded annotations and transform them into a format that can be used by the YOLO model.

To train a YOLO model, it is necessary to have annotations in a specific format. Yolo uses the following format in a txt file:

`<object-class> <x> <y> <width> <height>`

Where:

- `<object-class>` is the index of the object's class
- `<x> <y> <width> <height>` are the coordinates of the center of the bounding box and its width and height.
- The coordinates are normalized with respect to the image size.

The annotations of the road signs are provided as bounding boxes in a JSON file. The bounding boxes are defined by the minimum and maximum value of the x and y coordinates. These values are not normalized, and a lot of unnecessary information is present. It is therefore necessary to convert these annotations into a format usable by YOLO.

2.1.3 Task - Split data

The `split_data` task is responsible for splitting the data into training, validation, and test sets. The data is divided into 80% training, 10% validation, and 10% test sets. The data is shuffled before being split and then saved locally waiting for the upload.

2.1.4 Task - Upload data

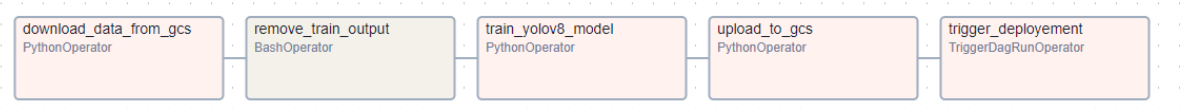
The `upload_split_data` task is responsible for uploading the processed annotations and images data to a post-processed bucket. The data is stored in a specific directory for each set (training, validation, test). This will be used by the training DAG to train the model.

2.1.5 Task - Trigger training

The last `trigger_training` task is responsible for triggering the training DAG. This task is executed after the upload of the processed data. It allows to start the training of the model as soon as the data is ready.

2.2 DAG – Training

This DAG is responsible for training the YOLOv8 model. It must retrieve the preprocessed data, train the model and save the results obtained. The tasks are as follows:



2.2.1 Task - Download data

The `download_data_from_gcs` task downloads the preprocessed dataset from the cloud. The dataset is separated into training, validation, and test sets.

2.2.2 Task - Remove old metrics

The `remove_train_output` task removes the old metrics from the cloud to avoid conflicts with the new metrics.

2.2.3 Task - Train YOLOv8

The `train_yolov8_model` task trains the YOLOv8 model on the preprocessed dataset. The YOLOv8 model is a real-time object detection algorithm based on a convolutional neural network. It is developed by Ultralytics. We use the pre-trained model and retrain it with the preprocessed dataset. Inside this task the YOLOv8 outputs a variety of files including the weights, metrics and some visualizations.

2.2.4 Task - Upload model

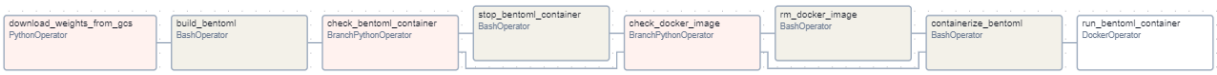
The `upload_to_gcs` task uploads the trained model to the cloud. The model weights are stored in a file that can be used for deployment. All the files generated by the YOLOv8 model are uploaded to the cloud.

2.2.5 Task - Trigger deployment

The `trigger_deployment` task triggers the deployment DAG to deploy the trained model.

2.3 DAG – Deployment

This DAG is responsible for deploying the trained model. It must download the model weights, build a BentoML archive, containerize the model, and deploy it. The tasks are as follows:



2.3.1 Task - Download weights

The `download_weights_from_gcs` task downloads the latest and best-performing model weights from Google Cloud Storage to a local directory. These weights will be used to create a BentoML archive in subsequent steps.

2.3.2 Task - Build bentoml

The `build_bentoml` task creates a new BentoML archive using the previously downloaded model weights. The archive packages the model and metadata necessary for deployment.

2.3.3 Task - Check bentoml container

The `check_bentoml_container` task verifies if a BentoML container is already running on the host system. If a container is already running, it triggers the stop bentoml container task to avoid deployment conflicts. Otherwise, the stop task is skipped.

2.3.4 Task - Stop bentoml container

The `stop_bentoml_container` task stop the running BentoML container deployed on the host. This ensures there are no conflicting deployments before proceeding to deploy the updated model.

2.3.5 Task - Check docker image

The `check_docker_image` task checks the host system for an existing Docker image of the BentoML deployment. If an image is found, it triggers the remove docker image task to ensure the latest version is used. Otherwise, the remove task is skipped.

2.3.6 Task - Remove docker image

The `rm_docker_image` task removes the existing Docker image of the BentoML deployment from the host system. This ensures the latest version is used for deployment and to save storage space.

2.3.7 Task - Containerize bentoml

The `containerize_bentoml` task builds a new Docker image using the latest BentoML archive. The resulting containerized application is ready for deployment and stored on the host.

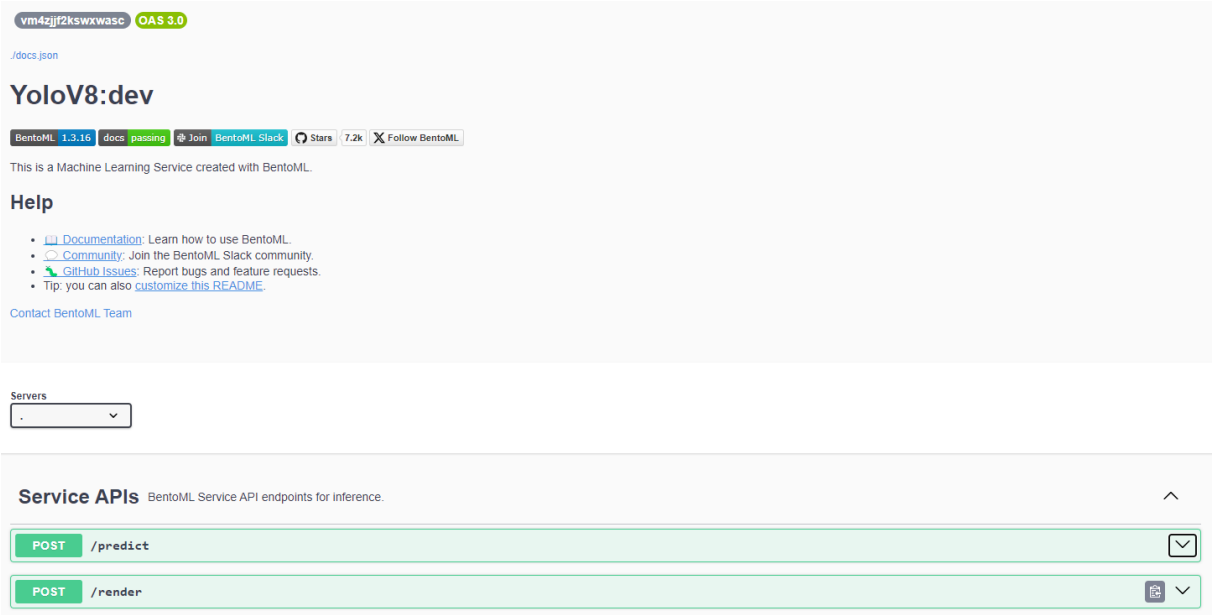
2.3.8 Task - Run bentoml container

The `run_bentoml_container` task deploys the newly built Docker image on the host system. By running the container independently, the deployment remains operational even if the Airflow service is unavailable.

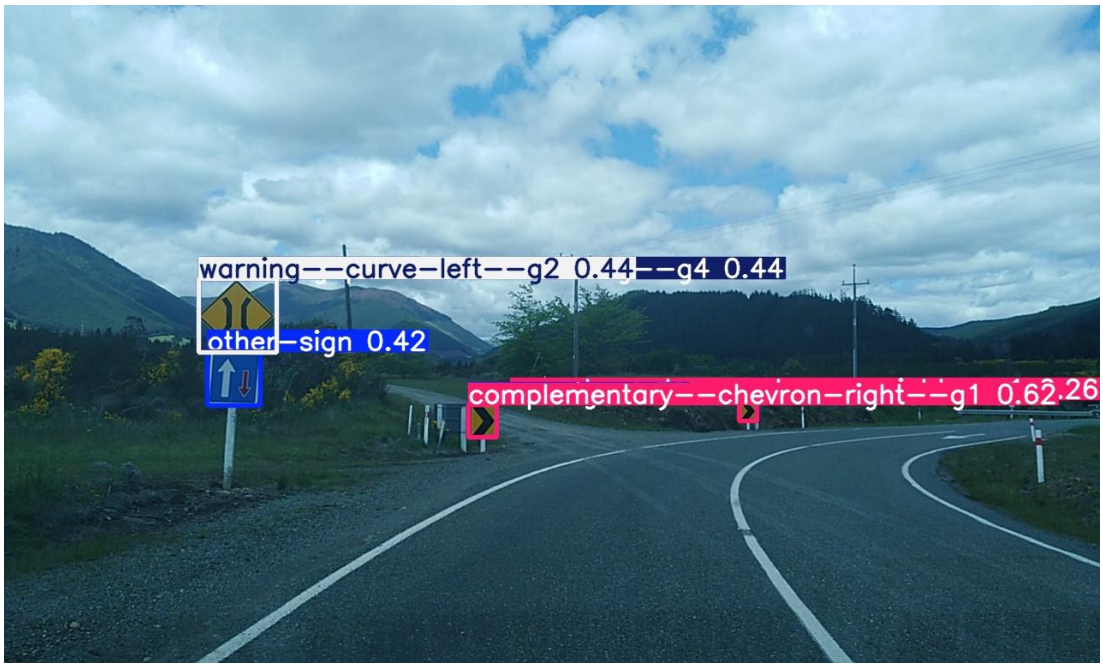
2.4 Results

At the end of the pipeline execution, we obtain a trained and deployed model. This model can be used to predict road signs in images. It is possible to test the model by giving it an image as input and visualizing the predictions.

Example of the API that can be used to test the model:



Prediction result (bounding box around the road signs):



3 Added value of using an AirFlow pipeline

This use case is an example of using a pipeline to automate the process of training a model. It allows to see the different advantages of using this type of tool.

The advantages of using such pipelines are as follows:

- **Automation of the process:** all steps are automated and traceable. It is not necessary to manually run scripts or execute commands for each step. There is therefore no risk of human error.
- **Repeatability:** the pipeline can be restarted at any time to reproduce the results. This allows to test different configurations and compare the results.
- **Monitoring:** the pipeline allows to monitor the progress of the process. It is possible to see when a task was executed and if it failed. If a problem occurs, it is easy to identify and therefore to correct. This is not possible when the steps are executed manually.
- **Collaboration:** the pipeline allows to easily share the process with other people. It is possible to see the results of the different steps and compare them. There is no risk of confusion about the steps to follow or compatibility issues between the versions of the software used.
- **Modularity:** the pipeline is composed of different tasks that can be reused for other projects. It is also possible to modify a task without affecting the others to test different configurations.
- **Scalability:** the pipeline can be scaled to handle large datasets and complex workflows. It can be run on a single machine or distributed across multiple machines to speed up the process.

However, we also noticed some disadvantages:

- **Complexity:** setting up the pipeline can be complex. It is necessary to understand how AirFlow works, how to create a pipeline and how to configure the different tasks. This can take time and requires technical skills.
- **Learning curve:** using AirFlow requires knowledge of AirFlow and Directed Acyclic Graphs (DAGs). It is necessary to understand how to create a DAG, how to define tasks and dependencies between them. This can be difficult for people who are not familiar with these concepts.

- **Not useful for small projects:** using such a pipeline can be oversize and complicated to set up for a small project. It would take more time to configure the pipeline than to complete the project manually.

There is therefore a trade-off between automating the process and the complexity of setting up the pipeline. However, for a large-scale project, using a pipeline is a wise choice to save time and avoid human errors.

4 Difficulties encountered

Among the difficulties encountered, we can mention the distribution of tasks among the 5 members of the group. Many tasks required coordination between the members to avoid conflicts between the different parts of the project. In addition, understanding how AirFlow works was difficult. The learning process is quite lengthy, requiring significant time to grasp how to implement various tasks. Many aspects are not intuitive, and understanding how to configure them properly takes considerable effort.

There were also several problems with the Airflow documentation. Many points are difficult to find and when they are found, the examples provided are not always clear. For example, to do branching in a DAG, the example provided is very basic and does not allow to understand how to do more complex branching. It was necessary to browse the documentation for a long time to understand that the default value of a parameter had to be changed for the DAG to work correctly. A similar problem was encountered with the management of permissions for the different services used. It is done in an unusual and unintuitive way in AirFlow, and the documentation is also unclear on this point. As a result, we took time to configure the permissions correctly.

The difficulty above also led us to not deploy our pipeline in the cloud as initially planned.

5 Further improvements

Initially we wanted to showcase a continuous learning approach in our pipeline. The goal was to have a drop bucket that would trigger the pipeline upon data addition. This would allow to have a more flexible and dynamic pipeline that would not require manual intervention.

Continuous learning also meant that we would require a way to detect newly added data to the processed bucket, allowing to retrain only on new data. We were planning to use DVC to manage this versioning of the data, but we did not manage to implement it.

In a technical aspect, we could improve the code by factorizing some part related to Google Cloud Storage. Also, it's not a good practice to store the credentials in the code as we did to simplify. We should use a secret manager or some environment variables to store them.

An improvement to the deployment pipeline could be to allow deploying a specific tag of a BentoML archive. Currently, we always containerize using the latest bentoml archive (tag:latest). By specifying and deploying a precise tag, we could ensure better traceability and control over the versions being used in production.

Last, we could improve the reproducibility of the pipeline by having some deployments scripts that could be triggered based on some git commit or merge for example. This would allow to have a complete CI/CD pipeline.

6 Conclusion

In conclusion, the use of an AirFlow pipeline for a Machine Learning project allows to automate the process, make it reproducible and scalable. It is a powerful tool that can be used for large-scale projects

to save time and avoid human errors. However, it requires a good understanding of how AirFlow works and how to configure the different tasks.

The use case chosen for this project is the training of a YOLOv8 model to detect road signs in images. This project was rather complex, which required the creation of a pipeline with many tasks. The choice of this use case was therefore relevant to test the capabilities of AirFlow. This allowed us to see the advantages and disadvantages of using this technology for a Machine Learning project.