FSDL 2021

Project Report

Thomas Paula and Jonathan Salfity

May, 2021

Introduction

This report describes the final project developed for the Full Stack Deep Learning 2021 course. It provides an overview of the motivation and objectives behind the project, the architecture and implementation details of what we built, the main lessons learned during the project, the limitations, and potential future work.

Motivation and Objectives

We joined this course to learn more about the end-to-end machine learning workflows, understanding what it takes to take a model from development to production. In order to exercise these concepts, we decided to develop a project that aimed to exercise end-to-end ML workflows in a simple classification scenario.

We want to have data preparation, training, and evaluation pipelines built on a tool like Airflow. The training runs/experiments will be logged in MLFlow, as well as the trained models. The models will be deployed through MLFlow and we would like to develop a web UI to interact with the model. If we are successful and time permits, we would like to implement a data flywheel by creating a user-guided trigger to respond to incorrect classifications by adding new labeled images to the dataset and restarting training, deployment and monitoring -- correcting the customers' served model.

More specifically, we would like to exercise the following concepts:

- Experiment tracking
- Data, training, and evaluation pipelines
- Model registry & deployment
- Integration of deployed model with Web Application

We want to leverage this project to learn these concepts and understand more the challenges of deploying ML solutions. We will use simpler models to be able to develop in our machines.

Implementation Details - Architecture

We decided to start using a simple classification model to be able to develop the end-to-end processes first and, then, slowly add complexity. A lot of the tools in this project were new to us, as well as setting up the required infrastructure for them to run properly. We decided to start with a simple Docker[1] container with Airflow[2] to learn more about how it works. Over time, we included additional components and increased complexity, tying it all together.

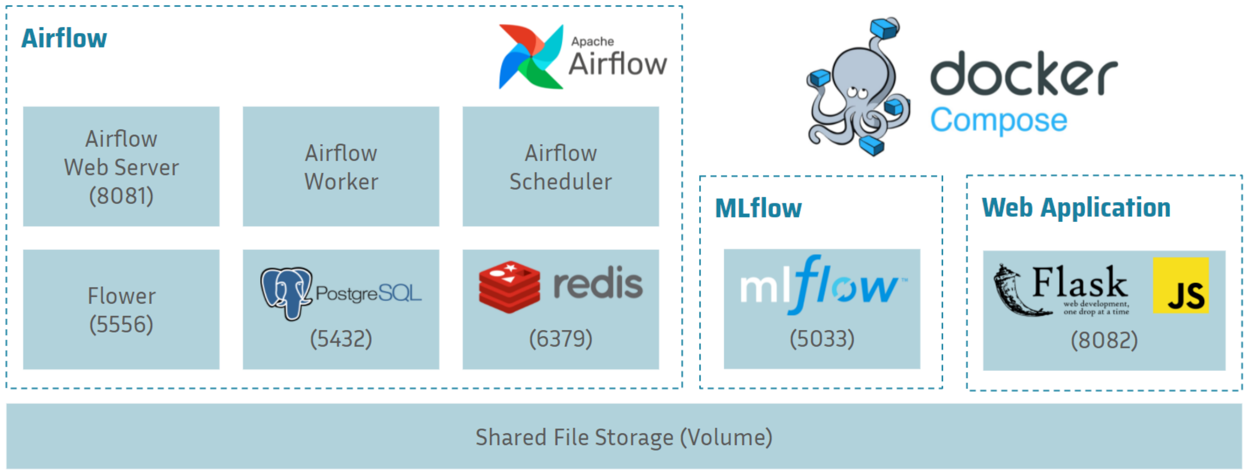


Figure 1. *High-level view of our architecture from the containers perspective.*

Figure 1 details our architecture from the containers perspective. We have eight different containers, represented by the eight boxes in the image. Six containers are part of Airflow, responsible for orchestrating the DAGs related to data preparation and model training. The models are registered in MLFlow[3], whose tracking server is also running in a dedicated container. Finally, we developed a simple web application with Flask[4] and JavaScript to enable users to interact with the trained model from MLFlow. It also runs in its dedicated container. All containers are orchestrated with Docker Compose.

We will now detail each of these building blocks, the technical choices we made, and how we implemented them.

[1] Docker - https://www.docker.com/
[2] Airflow - https://airflow.apache.org/
[3] MLFlow - https://mlflow.org/
[4] Flask - https://flask.palletsprojects.com/en/1.1.x/

Airflow

In this project we wanted to learn about Airflow and evaluate its suitability for orchestrating ML workflows. Airflow is an open source platform that allows one to programmatically create, schedule, and monitor workflows. It allows one to create workflows as directed acyclic graphs (DAGs), programmatically, with pure Python code. We would like to understand to what extent it could be used to our simple classification scenario.

One of the key challenges we faced was setting up a Docker container with Airflow. There is one implementation that is vastly referenced by different tutorials and repositories across the web[5]. We started using this version to exercise how to use Airflow and its main concepts. We created a simple DAG based on the Wine Classification problem, using an Elastic Net with default hyperparameters. After many iterations and experiments, we decided to upgrade the Airflow version to the latest one. For that, we had to create our own version of the open source repository, creating a custom Docker Compose file. That was an important use of our time, since it required understanding better how Airflow and its different containers are organized.

We created a custom Docker Compose file based on the file provided by their official documentation[6]. Our implementation has six main docker containers.

- Airflow Web Server: the web interface from Airflow. Can be used to browse existing DAGs, trigger them, monitor their progress, and so on.
- Airflow Scheduler: responsible for orchestrating the DAGs and their tasks, and making sure they work properly. Manages dependencies between DAGs and their tasks.
- Airflow Worker: responsible for actually running the DAGs. The Executor is responsible for allocating required resources and putting tasks to be run in a queue, which is run by the worker. We used Celery[7] as our Executor and Redis[8] as the backend queue mechanism.
- Postgres: SQL database to store metadata about the pipelines being run in Airflow.
- Redis: in-memory data structure store, which is used as a database, cache, and message broker. In this case, it works as a backend for Celery.
- Flower: web-based tool for monitoring and managing Celery clusters.

All the Airflow containers (Web Server, Scheduler, Worker, and Flower) are based on a custom Dockerfile we created. Such a Dockerfile is based on Airflow 2.0.1 and installed Python libraries we used in our project.

[5] https://github.com/puckel/docker-airflow
[6] https://airflow.apache.org/docs/apache-airflow/stable/start/docker.html
[7] https://docs.celeryproject.org/en/latest/index.html
[8] https://redis.io/

MLflow

MLflow is an open source platform that aims to manage the entire machine learning lifecycle. It is comprised of four main components: MLflow tracking, which aims to record experiments and its artifacts; MLflow Projects, that provide an standard format for packaging ML code; MLflow Models, which provide a default format for deploying ML models in different serving platforms; and Model Registry, which allows one to store and manage models in a central place.

For our project, we decided to use the MLflow Tracking Server and the Model Registry. The Tracking Server allows us to track results of model training and related metadata, while the Model Registry is a place where the trained models are stored. The registry also allows us to transition the models from development to production and do model versioning.

We set up MLflow as a separate container in our docker compose file as well. Our setup stores the artifacts in the Shared Stored Volume and uses SQLite[9] as the backend store. The artifacts include files and general information from the runs stored in the disk. The SQLite database stores entities (runs, hyperparameters, metrics, etc.). More information on the different options to run MLflow are available in their documentation[10].

We created a custom MLflow Dockerfile that is used in our project. It is based on a Minoconda[11] base image and installs needed custom libraries. One important point is that we had to install git in this container, since MLflow uses it under the hood for versioning purposes. It was causing a silent failure that was hard to catch.

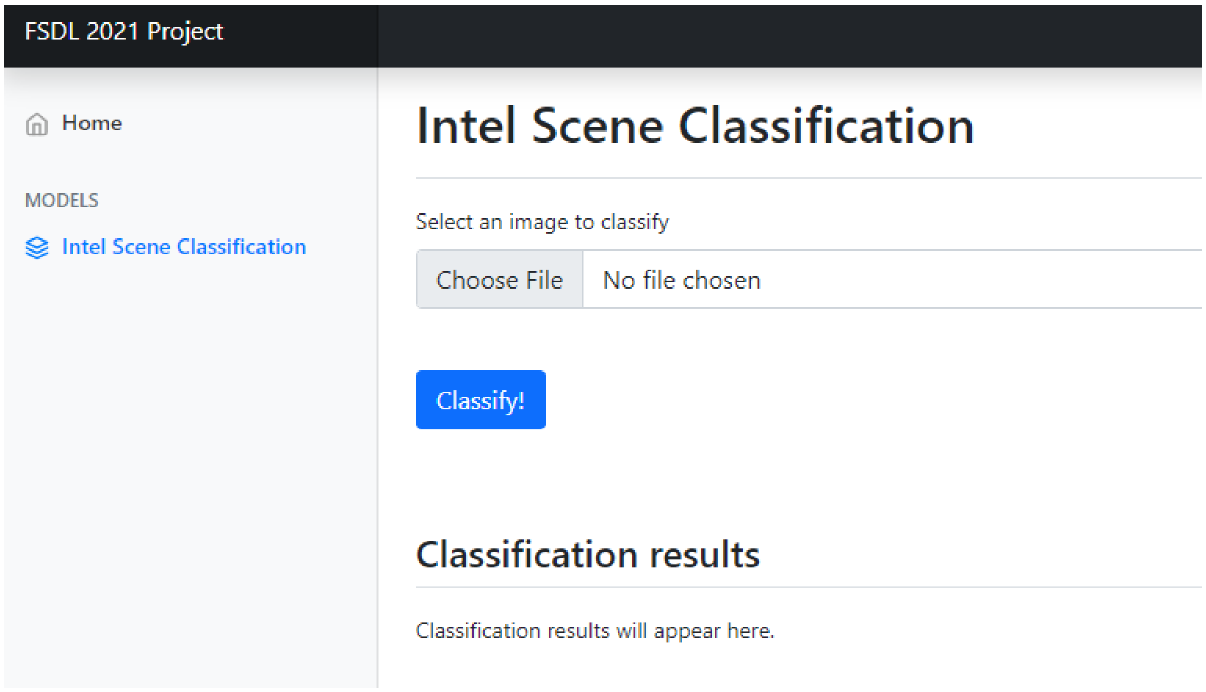
Web Application

To create the data flywheel, we developed a web application to interact with the developed models. Our web application was developed using Flask, JavaScript, and Bootstrap. Flask is a Python micro web framework, commonly used during the development of web applications with Python. Bootstrap[12] is a web framework that makes it simpler to develop responsive web pages. It contains components like custom buttons, input fields, and animations, as well as layout components that simplify the creation of web pages.

We developed backend REST APIs that are responsible for handling the frontend requests properly. The user can upload an image and predict that with the trained model (**Figure 2**). Such predictions can then be labeled as correct or incorrect, which automatically stores the new

[9] https://sqlite.org/docs.html
[10] https://www.mlflow.org/docs/latest/tracking.html#how-runs-and-artifacts-are-recorded
[11] https://docs.conda.io/en/latest/miniconda.html
[12] https://getbootstrap.com/

uploaded image into a feedback folder. The frontend logic is entirely written in JavaScript while the backend is Python, using custom endpoints in Flask. We also used werkzeug web application library, which allowed us to store files securely in the backend. More details on the workflows will be explained in the next section.



Intel Scene Classification

Select an image to classify

Classification results

Classification results will appear here.

Figure 2. *UI of the developed Web Application. Allows users to upload and image and classify into the classes supported by the Intel Scene Classification dataset.*

We created a custom Dockerfile for the web application. It is based on Minoconda base image as well, and installs all the required libraries.

Custom Python Library - fsdl_lib

We have code that is shared among different containers in our project. For example, our model leverages a pre-trained feature extractor based on ResNet50, which is used in the training DAG in Airflow as well as in the prediction code in the web application. To make sure we're using the same methods and enable reusability, we created a small Python package called "fsdl_lib". Its main purpose is to be used in all containers that share common code in order to enable leverage.

The custom Python library is installed in Airflow containers and in the Web Application container. It is installed during the build process of the image of the containers.

Shared Storage Volume

In order to simplify sharing specific data and folders within our project, we mapped the main paths as Docker volumes in the Docker Compose file.

We have a root folder called “data”, which is organized into three subfolders: “raw”, “processed”, and “feedback”. Raw contains the datasets used in our experiments. Processed stores any intermediary representation of the data that can be used in the DAGs. For example, features extracted from the images in the raw folder are stored into the “processed” folder. Finally, feedback stores the images that the user provided feedback via the Web Application. Such images are then incorporated into the dataset to enable retraining.

We also created another root folder called “storage”, which contains mostly MLflow data. The “artifacts” folder stores all artifacts saved by MLflow. The “database” folder stores the SQLite database used by MLflow in the backend. We decided to keep storage separated from data, but they could mostly be the same folder.

Key Lessons Learned

During the implementation of this infrastructure, we faced several challenges and learned important lessons. A lot of challenges were related to setting up environment variables, required libraries, and mapping paths correctly. Here we summarized some of the main lessons divided into the specific components, but they are not an exhaustive list.

Airflow

- Permissions: Airflow image defines a system user called “Airflow”, with restricted permissions. In order to install system libraries through apt-get, for example, we needed to define “USER root” in part of the Dockerfile. In addition, to be able to download Pytorch pretrained Resnet50, we needed to specify a specific path to download it into.
- Pass data between tasks of a DAG: you can use XCOM to pass data serialized as JSON between tasks of DAG. If you want to pass Python objects, you need to set up a flag in the container “AIRFLOW__CORE__ENABLE_XCOM_PICKLING: 'true'”. However, it should not be used to pass large objects (such as datasets or large Numpy arrays). Therefore, for larger objects we stored them in the disk and loaded again in the next task of the DAG.

MLflow

- Silent failure to log models: we had an issue where the models were not being logged into MLflow, not appearing in the artifacts folder in the MLflow UI. After a good time debugging, we discovered the container should have Git installed to be able to version and store the artifacts correctly.

- The use of MLflow with Airflow presented some challenges, mostly because of the way MLflow handles experiments and runs. That led us to control the creation and management of experiments and sessions more manually as to be able to keep track of the same experiments within different operators/tasks of a DAG.

Implementation Details - Workflow

In this project, we started our work with a simple workflow to understand the mechanics of Airflow, learn how to integrate it properly with MLflow, and define a suitable way to define an ML training flow as an Airflow DAG. Next, we moved to a more complex workflow for image classification, integrating it with a web application to provide feedback. Within such workflow, we also explored the idea of training multiple models in parallel and within the same operator ini the DAG. In the next sessions we will detail our work.

Simple Workflow - Wine Classification

For the simple workflow, we defined a training DAG that loads the data, preprocesses it, trains a simple machine learning model, evaluates it, and registers it into the MLflow model registry. It was useful for us to understand the mechanics of Airflow, as well some of the main challenges (e.g., how to pass data between tasks in the DAGs). We also started with an older version of Airflow and migrated afterwards.

Dataset

We used the dataset Red Wine Dataset from the Wine Quality Dataset, available in the UCI Machine Learning repository[13]. The dataset is composed of 11 variables related to physicochemical properties of the wine: fixed acidity, volatile acidity, citric acid, residual sugar, chlorides, free sulfur dioxide, total sulfur dioxide, density, pH, sulphates, and alcohol. The target variable is the wine quality, which is a score between 0 and 10.

DAG

We created a single DAG for this problem, which is shown in **Figure 3**. The workflow is based on four main building blocks. The first one is responsible for loading the data from a CSV file, doing an initial preprocessing, and generating the train test split for the data. The processed data, represented by four numpy arrays (X_train, X_test, y_train, and y_test) are stored in the "processed" folder. Train model basically loads the processed data from disk and trains an Elastic Net model with default parameters. We did not work to improve this model since our objective was not exactly model performance, but exercising the pipeline.

[13] http://archive.ics.uci.edu/ml/datasets/Wine+Quality

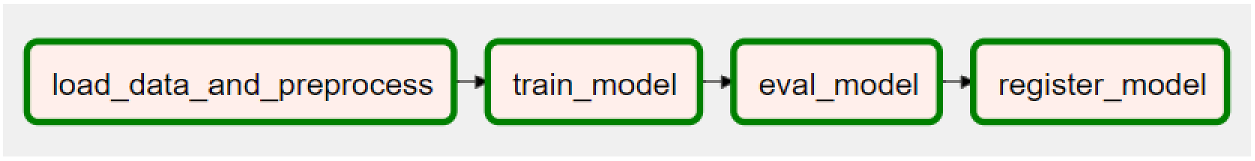


Figure 3 *- DAG training*

Next, the evaluation building blocks takes the trained model, loads the test data, does the prediction, and computes three metrics (RMSE, R2, and MAE), which are all logged in MLflow. Finally, the last building block registers the model in MLflow as to be used by any application. The training and eval steps run under a MLflow experiment with the autolog feature for Scikit Learn, which automatically logs all the relevant information such hyperparameters and other model information.

Complex Workflow - Image Scene Classification

In the complex workflow, we developed a training routine for the dataset Intel Scene Classification Challenge, from Kaggle. We decided to take an approach of multiple steps to exercise a more interesting scenario. Therefore, we defined a training workflow where we employ a pre-trained convolutional neural network to extract the features. Next, such features are used to train a simple classifier. Such a classifier registered and moved to production in MLflow, and then integrated on a Flask web server for classification of users' images. When images are incorrectly classified, the user correctly labels their uploaded image, which is eventually fed back for re-training.

Dataset

The dataset from the Intel Image Scene Classification[14] contained around 25k RGB images of size 150x150 distributed into 6 categories - 'buildings', 'forest', 'glacier', 'mountain', 'sea', 'street'. The dataset comes pre-split into train with 14k images, test with 3k images, and Prediction with 7k images. We used a subset of the full dataset, consisting of 1000 of images during training, to anticipate for the relatively small number of feedback images influencing the training process. The subset we used was created by randomly getting images from the full dataset.

DAGs

We decompose the different building blocks of an ML pipeline -- data preparation, training, evaluation, model registry, and feedback data syncing -- into Airflow DAGs, which are composed

[14] https://www.kaggle.com/puneet6060/intel-image-classification

of individual Airflow Operators. We detail the DAGs we build for Intel Scene Image Classification below.

DAG for training machine learning models with ResNet50 features and registering models in MLflow

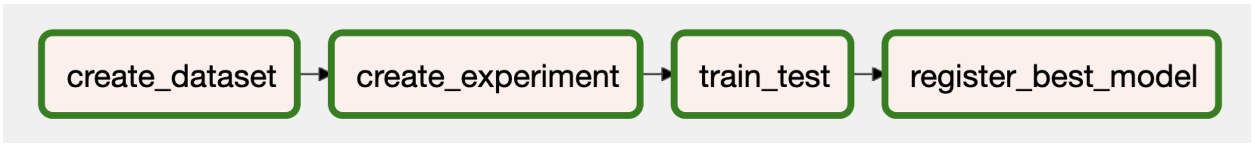


Figure 4 *- DAG training models in sequence*

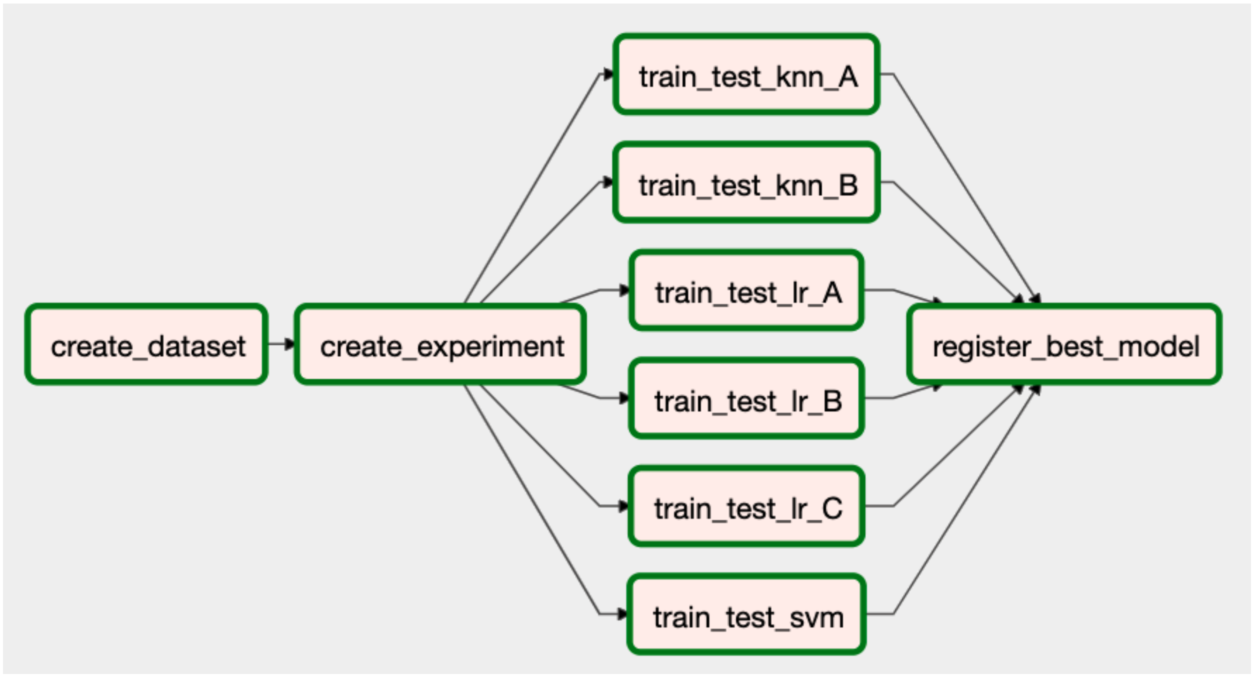


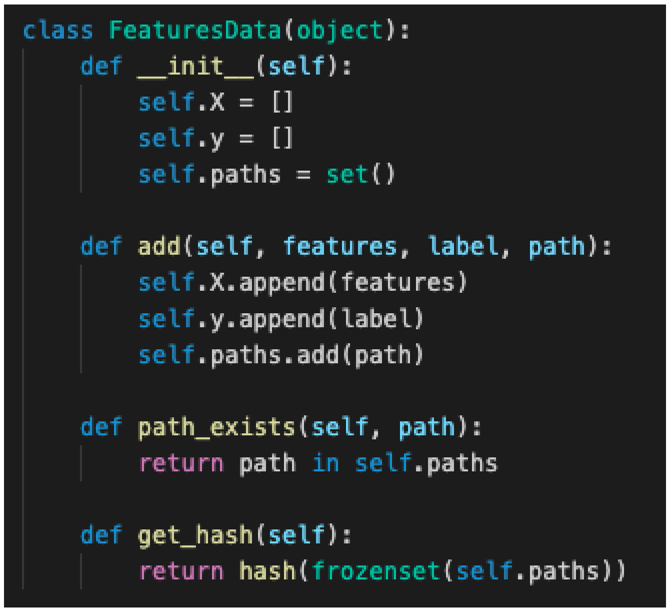
Figure 5 *- DAG training models in parallel*

Create Dataset

In this Operator, a Resnet-50 model pretrained on ImageNet is downloaded from PyTorch Model Zoo and utilized for feature extraction by using the final layer before classification. The images are preprocessed and then passed through the feature extractor and paired with the respective image class.

To make it easier to transfer features between different operators and avoid redundant feature extractions, we implemented a simple FeaturesData() class, which stores seen image files, allowing feature extraction to be performed only on images that have not been previously processed. Bypassing previously processed images saves computation time and resources. In addition to processed images, FeaturesData() contains X, model's input of a feature vector, and y, the model's output of a class, as numpy arrays and integers, respectively.

To save the different train and test data over the course of the data preparation, train, test, register, and feedback process, we sought to version control our datasets. Exploring different open source data version control software tools, like DVC[15], we found some difficulties in implementation, which are addressed in the "Lessons Learned" section. Ultimately, we implemented a simple versioning method in our FeaturesData() class. As a FeaturesData() class is created by processing images to features, the FeaturesData() class stores the set of processed image paths. By hashing the set of processed image paths, the FeaturesData() class creates a unique identifier for each dataset. The hash output is then passed and eventually logged with the associated model in MLflow.



```python
class FeaturesData(object):
    def __init__(self):
        self.X = []
        self.y = []
        self.paths = set()

    def add(self, features, label, path):
        self.X.append(features)
        self.y.append(label)
        self.paths.add(path)

    def path_exists(self, path):
        return path in self.paths

    def get_hash(self):
        return hash(frozenset(self.paths))
```

Figure 6 *- FeaturesData() class in our fsdl_lib*

Create Experiment

An MLflow experiment is created to group the upcoming MLflow processes. An experiment-id is generated and passed through to the next operator. If the experiment already exists, this methods make sure it is leveraged and a new run under the same experiment is created.

[15] https://dvc.org/

Train and Evaluation

After data preparation, the train FeaturesData() and test FeaturesData() are loaded from the 'data/processed' folder. LogisticRegression, SVM, and KNN models, each with different hyperparameters, are trained. We implemented the model training to be in sequence or in parallel, shown in Figure 4 and Figure 5. Additionally, within the DAG, we created modularity for the type of model and model hyperparameters.

After the set of models are trained, each model is tested against the test set for metrics of accuracy, F1 score, precision, and recall. We chose the most important evaluation metric to be the F1 score, which considers both the False Positive and False Negative classifications on the test set.

Register Best Model

After the models are evaluated for performance metrics, this Operator registers the model with the highest F1 score to MLflow. Along with registering and storing the model, the model artifacts, parameters of the training session, performance metrics, and dataset version are logged to MLflow. For now, the model is manually switched from Staging to Production in the MLflow web browser and is ready to be called by the Flask web application.

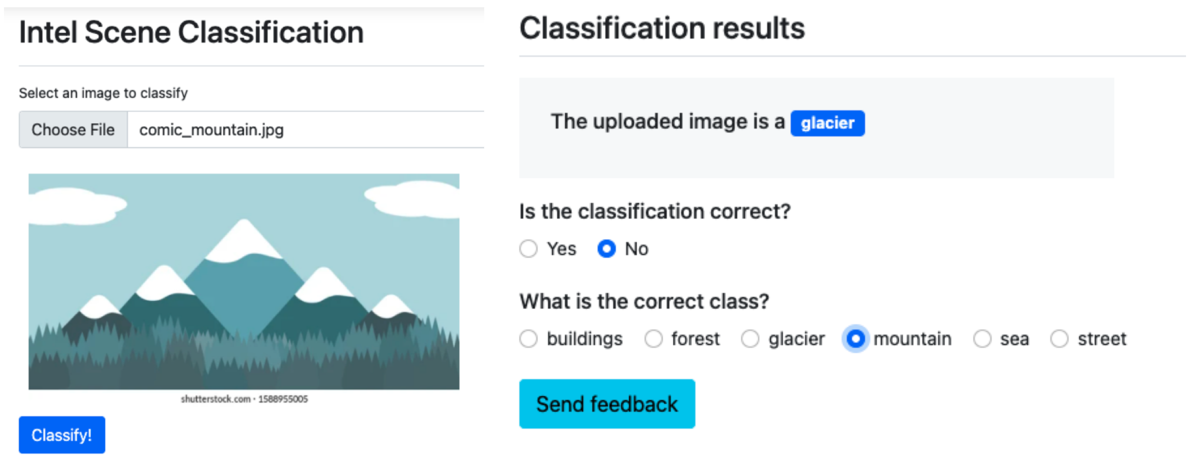
DAG to Sync Feedback

As described, all data is stored locally in the project '/data' folder. As images are incorrectly classified, the web application is continuously adding users' uploaded images to the 'data/feedback/<class>' folder. This DAG sync's feedback images for use in the training pipeline by calling a shell script to manually move the images from 'data/feedback/<class>' to 'data/raw/<class>'.

Front End

After a model is registered in MLflow and set as production, the front end web page is ready to make predictions. We used Flask to develop the backend of our services and the interface is mostly HTML, CSS, and JavaScript. We created custom JavaScript code to handle the calls to the REST APIs in the backend and trigger changes in the UI.

The backend loads the feature extractor using the fsdl_lib. The latest production model available in MLflow's model registry is loaded when the first request comes. As we use fsdl_lib we share the same preprocessing steps applied during the training phase in Airflow are used during the prediction time. More details of how the webpage calls the model are in the "Implementation" section.


Intel Scene Classification

Select an image to classify

Choose File comic_mountain.jpg

Classify!

Classification results

The uploaded image is a glacier

Is the classification correct?

Yes [✓ No]

What is the correct class?

buildings forest glacier [✓ mountain] sea street

Send feedback

Figure 7 *- An incorrect classification gets corrected by the user*

Upon incorrect classifications, the user has the ability to send their uploaded image and the correct label, which is added to the correct 'data/feedback/<class>' folder and ready to be integrated into a new version of training and test data. If the image was correctly classified, the user can also send feedback and incorporate the new image to the training, further enhancing the model.

Flywheel

Combining all aspects of our project, **Figure 8** summarizes the complete data flywheel.

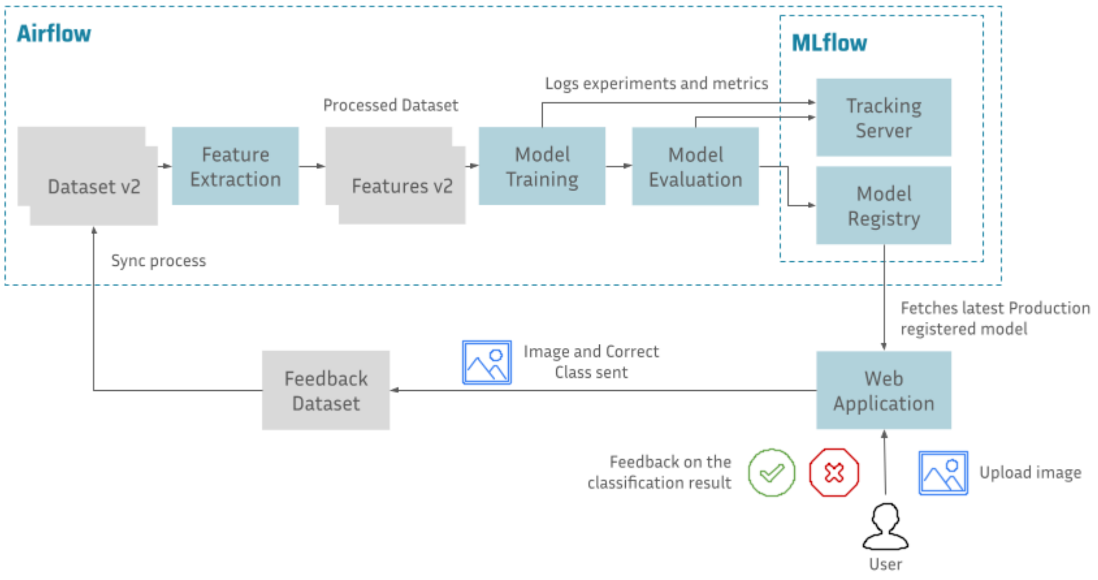


Figure 8 - Data Flywheel

The feedback signals from the live, runtime model is incredibly important for the deployed model's performance. As we learned in this class, there are different contributing factors to model degradation post-deployment, including data drift, model drift and a domain shift. We don't have an explicit monitoring system to measure distribution changes during deployment, instead we rely on users' input to improve the model's performance. Through this project, we developed a feedback process to mitigate data drift by adding more examples to the data expectation the model is built upon.

Key Lessons Learned

Data Versioning

- The underlying concept of versioning datasets is simple, however we found difficulty in implementation. Working with DVC, we learned the dependency on Git version control, which introduced credential issues with our project running in local Docker containers and storage complications for developers who plan to use our project.

Feedback Influence on Model Performance

- In this simple example, a single new mage introduced as feedback to the dataset is able to affect model performance upon retraining. We believe this is possible because the initial dataset and the number of classes are relatively small. In the future, we'd like to monitor deployment distance metrics and quantify the relationship of new data on retrained models.

Conclusion

Our objective in this project was to learn about end-to-end machine learning workflows by directly implementing and connecting different building blocks such as data storage, data preparation, model training, model registering, and a user interface to feedback data for improving model performance. Especially by constructing the entire workflow on a local machine, we were able to gain insights into the low-level complexities which arrive in ML Operations. When working with ML Ops tools in the future, we will be able to recognize the innovations, bottlenecks, and abstractions which enable ML in a production setting.

Throughout our project development and lecture content from class, we've learned many software tools arising to address different aspects of ML Ops. A huge learning for us is the ecosystem required to enable ML in production, where breakthrough neural network architectures and training schedules compose just a small slice for production ML. We could also see some challenges of applying existing tools and putting them together. For instance, it was tricky to handle the context/lifecycle of MLflow experiments within Airflow DAGs because we needed to manually manage the creation and execution of runs.

We hope this document can be a useful source of knowledge for people starting to learn more about this area, as well as for people with experience to gain insights on different approaches for the ML Ops puzzle.