Member-only story

Build Machine Learning Pipelines with Airflow and Mlflow: Reservation Cancellation Forecasting

Learn how to create reproducible and ready-for-production Machine Learning pipelines through a Senior Machine Learning assignment

Jeremy Arancio · Following
Published in Towards Data Science
21 min read · Jan 12, 2024

These days, companies require engineering skills for the development and deployment of Machine Learning models into production.

A **Machine Learning Engineer** should now possess the capability to handle the entire model life cycle, covering aspects like *experiment tracking*, *orchestrator*, *CI/CD*, *version control*, and *model registry* — from data extraction to productionization.

In this article, **we'll dive into a machine-learning assignment I conducted for a leader in the short-term rental industry,** using two widely used tools in the industry: **Airflow** and **Mlflow**.

This article not only guides readers through the resolution of a real-case Machine Learning project but also provides public access to the entire repository for later use in machine learning projects.

GitHub - jeremyarancio/reservation_cancellation_prediction:

Predict if a reservation will be canceled using robust Machine Learning pipelines with Airflow and Mlflow - GitHub ...

github.com

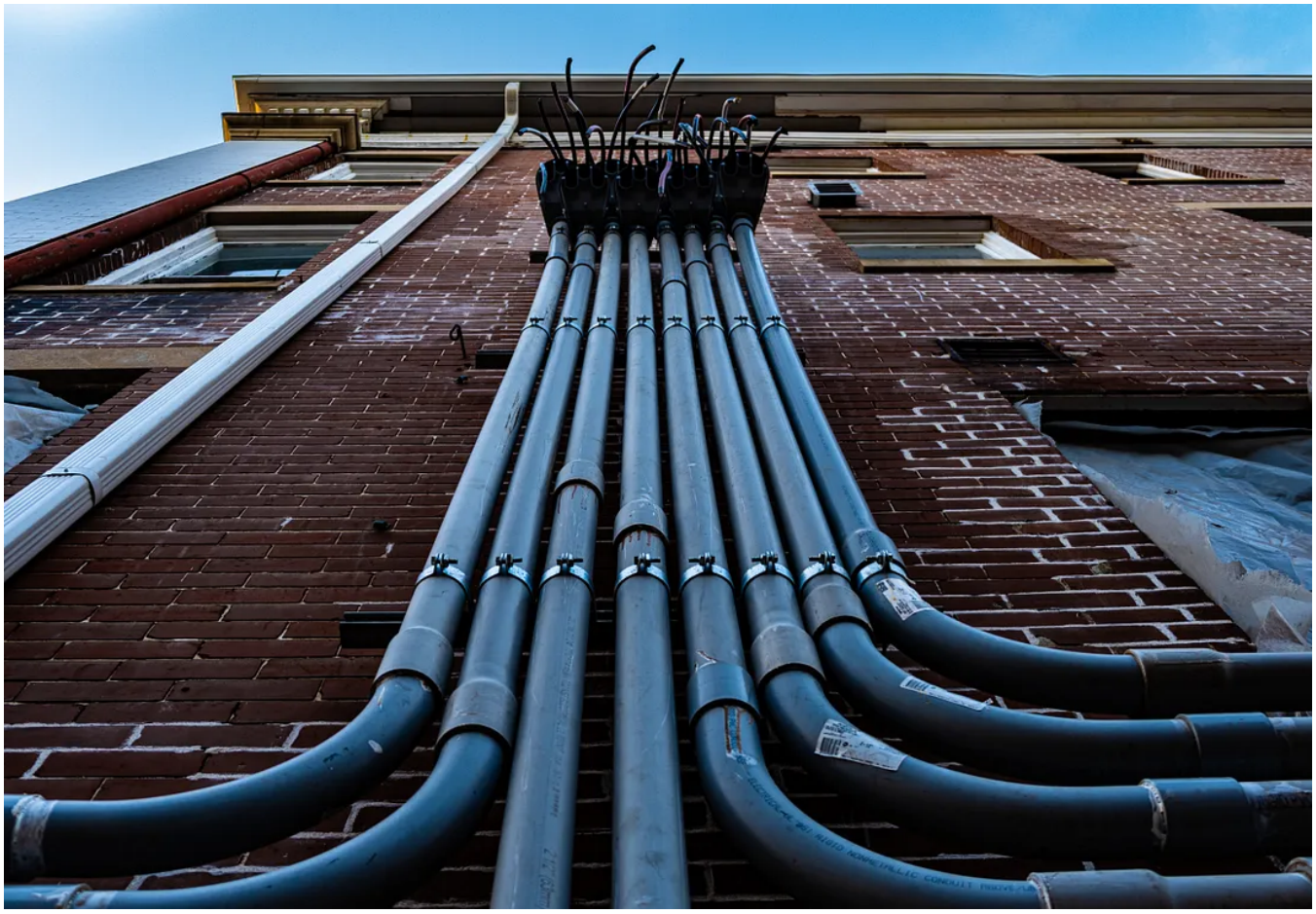


Photo by EJ Strat on Unsplash

Overview

1. **The Project**
2. **Exploratory Data Analysis**
3. **Configurations**
4. **Steps**
5. **Build DAGs with Airflow**
6. **Conclusions**

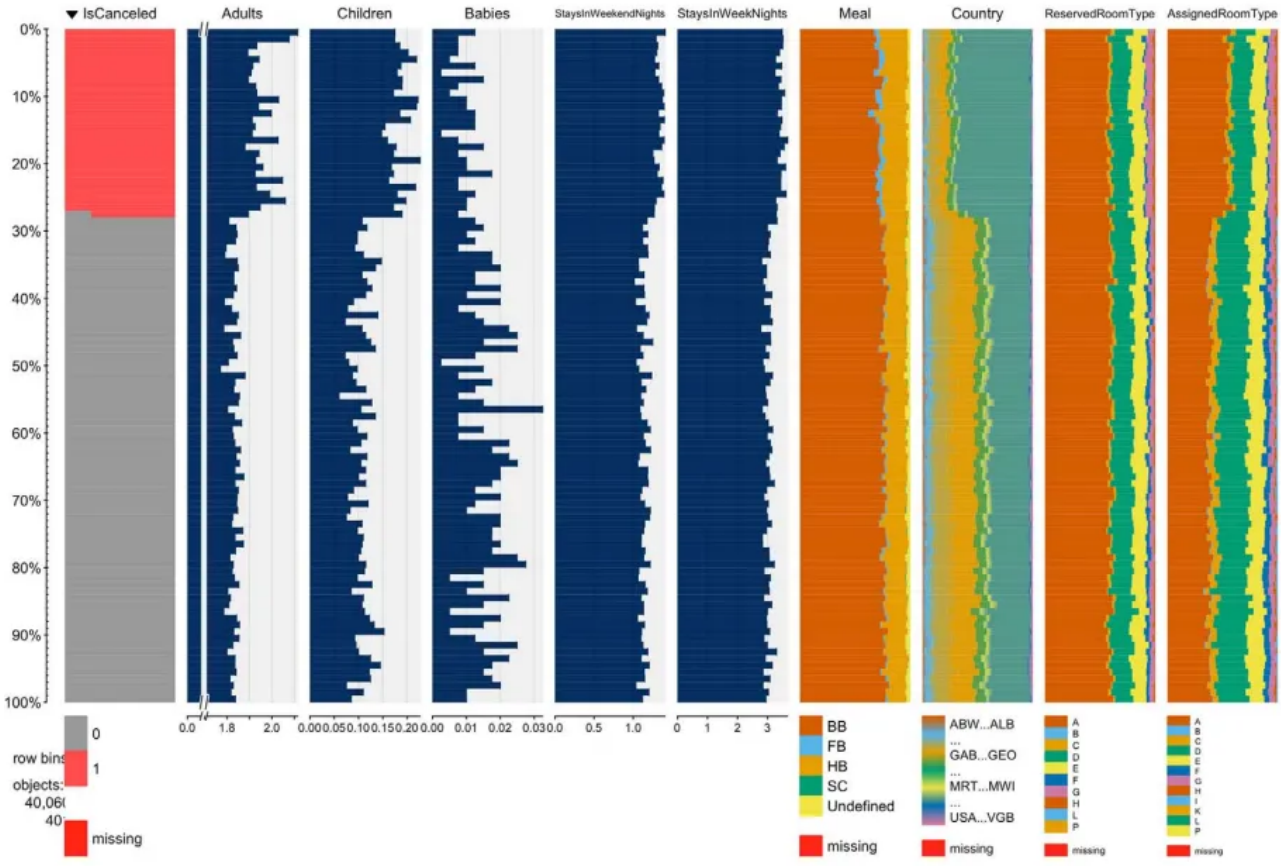
The Project

The company is a leader in the short-term homestay management industry. To find the next Senior Machine Learning engineer, they provided an assignment to evaluate the candidate's ability to build robust machine learning infrastructures.

A publicly available dataset was provided, containing **hotel reservations** separated into two categories: *bookings that effectively arrived; and those that were canceled.*

The objective of the assignment was the following:

> Based on reservation characteristics, predict if it is likely to be canceled.



Dataset partial visualization of all observations. From paper [1]

Now the assignment is clear, it's time to perform exploratory data analysis.

Exploratory Data Analysis

To perform the EDA, we'll use **Jupyter Notebook** with **Pandas.**

No data transformation will be performed in a notebook, the reason being the same preprocessing and feature engineering steps will be reproduced during the training phase, but also inferences.

Even if we haven't even started the project, we already think about the production architecture and how the model will be used once deployed. Remember that a model has no value if it remains in a notebook.

The dataset is composed of nearly 120,000 reservations for 2 different hotels: a resort hotel (79,330) and a city hotel (40,060). Our exploration showed that the dataset was already quite clean, but some elements are worth being pointed out.

Missing values

We start by replacing missing values for each feature.

```
count_na = df.isna().sum(axis=0)
count_na[count_na != 0]

#children         4
#country        488
#agent         16340
#company      112593
```

Agent and *Company* features indicate whether the reservation was made through a travel agency or a company, and they are represented by unique IDs. Given that there is no ID assigned as 0, it is safe to replace any NaN values in these features with 0.

Concerning the *children* feature, regarding the low number of missing values, we'll simply consider them equal to 0.

Finally, missing countries will be replaced by the categorical value *Unknown*.

Data leakage

After exploring the features, we noticed that some were strongly associated with the target *is_canceled,* leading to a risk of **data leakage.**

Data leakage happens when the training dataset contains information about the target that is no longer present during predictions once the model is deployed. The following article explains well the concept of data leakage.

In our example, two features, namely *deposit_type* and *reservation_status*, give information about the target but are less likely to be present in production.

	is_canceled	deposit_type	reservation_status
24735	0	No Deposit	Check-Out
51670	1	Non Refund	Canceled
2716	0	No Deposit	Check-Out
38975	0	No Deposit	Check-Out
97199	0	No Deposit	Check-Out
56881	1	Non Refund	Canceled
19747	0	No Deposit	Check-Out
63485	1	Non Refund	Canceled
46072	1	Non Refund	Canceled
78489	0	No Deposit	Check-Out

Sample of the dataset containing data leakage. Image by author.

While it is obvious the *reservation_status* is linked with the target feature, it also seems **the deposit wasn't refunded in case of cancelation**. To confirm, we calculate the proportion of canceled reservations when the deposit wasn't refunded, and it exceeds **99%**, pointing towards a clear causal relationship.

Those features will have to be removed during the feature engineering step.

Categorical values

Half of the features are categorical, meaning they need to be transformed into numerical values to be considered by the model. A simple way to encode those features is to use the **Ordinal Encoder** from Sklearn.

It converts each unique category into a unique ID. However, what happens during an inference if the encoder prepared during training encounters a category it has never encountered before?

Lucky for us, Sklearn came with an option to **handle unknown values**, replacing them with a default one.

```
from sklearn.preprocessing import rdinalEncoder
ordinal_encoder = OrdinalEncoder(
            handle_unknown="use_encoded_value",
            unknown_value=-1
        )
```

Yet, certain categorical values exhibit **high cardinality**, involving numerous unique categories, which can result in what is commonly referred as the *Curse of Dimensionality*.

To solve this issue, we use the **Target Encoder** from Sklearn, which encodes categories based on their influence on the target.

$$encoding_j = countTarget1_j / totalOccurence_j$$

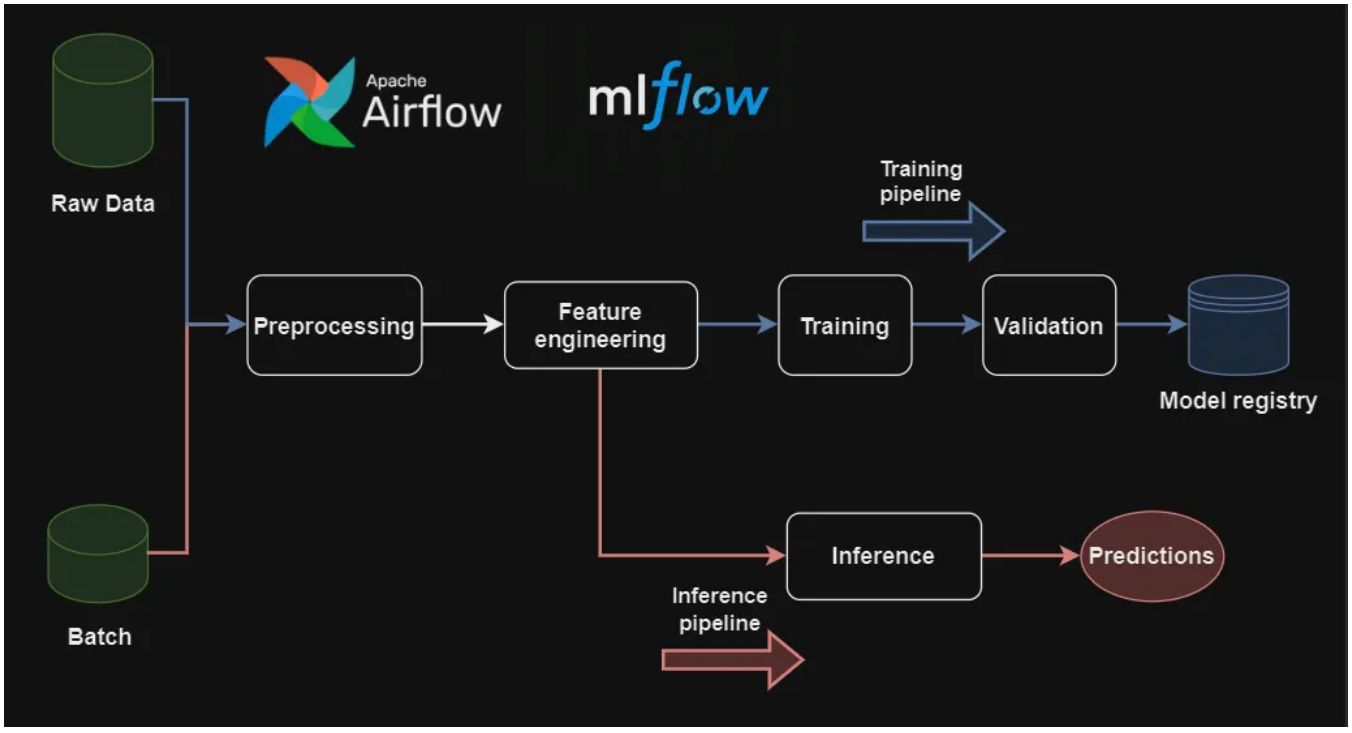
Target encoding algorithm, j representing a category. Image by author.

> ***Important note:*** *these encoders will be prepared using the* ***training dataset****, exported as artifacts, and subsequently employed during model evaluations and inferences. Therefore, it is imperative that the feature engineering step takes place after the data is split into train and test sets.*

We have briefly talked about the EDA. You'll find more analysis in the exploratory notebook located at `notebooks/0_exploratory_data_analysis` in the GitHub repository.

Now let's dig into building machine-learning pipelines that will:

- **Preprocess the data,**
- **Perform feature engineering,**
- **Train and evaluate the model,**
- **Register a new model version if it performs better than the deployed one,**
- **Perform inferences using the trained model and encoders.**



Machine learning pipelines. Image by author.

We'll use **Airflow** coupled with **Mlflow** to build those pipelines, two tools widely used in the industry today in Machine Learning.

Configurations

In this section, we'll walk through the configuration of Airflow and Mlflow for this project. To keep it simple, we'll set up these tools locally.

You can skip this section if you're not interested in reproducing this work.

Mlflow

Mlflow is an experiments tracking and model registry tool widely used in the industry to develop and deploy Machine Learning models in production.

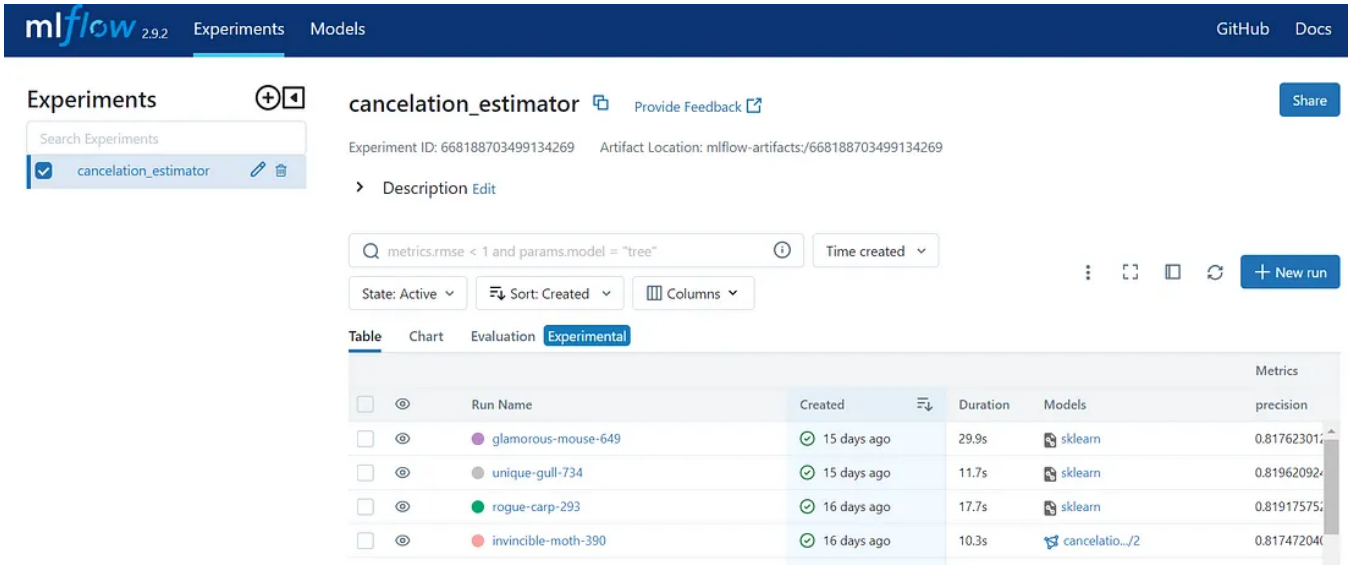
> **Quickstart: Install MLflow, instrument code & view results in minutes**
>
> In less than 15 minutes, you will: Install MLflow Add MLflow tracking to your code View runs and experiments in the...
>
> mlflow.org

To run it locally, you first need to install Mlflow in your environment using the command `pip install mlflow .`

We then start a new tracking server locally by configuring the path where experiments and registered models will be stored. To do so, you can type the following command in the shell.

```
mlflow server --backend-store-uri mlflow/ --artifacts-destination mlflow/ --por
```

This will start the Mlflow UI under the port `8000`, giving you a visual of your ML experimentations.



Mlflow UI relative to the project. Image by author.

Mlflow is now set up.

Airflow

Airflow is an industry-first-choice orchestrator. It was initially developed to build data engineering pipelines but with the expansion of Machine Learning in business, it started to be used to manage ML workflows as well.

We'll install the standalone version that will run in your repository.

> **Quick Start - Airflow Documentation**
>
> This quick start guide will help you bootstrap an Airflow standalone instance on your local machine. Note Successful...
>
> airflow.apache.org

To install the library, you first need to indicate the airflow path containing the database and the **dags**, referring to the pipelines as **Directed Acyclic Graph.**

```
# Add the airflow directory into your .bashrc file
export AIRFLOW_HOME=<path/to/airflow/directory/>
```

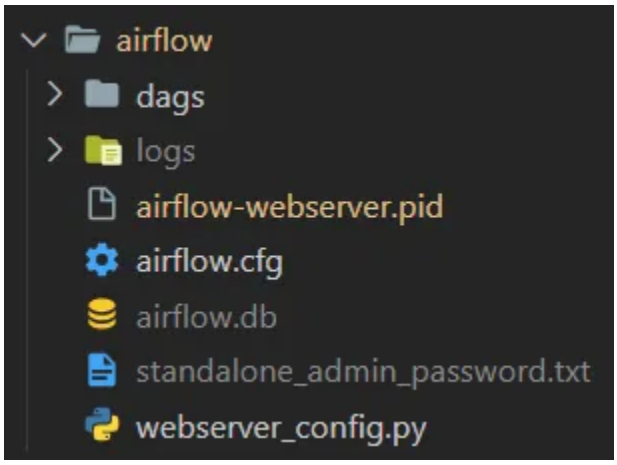
We then install Airflow following the instructions from the official documentation. Airflow uses constraint files to enable reproducible installation, using `pip` .

```
AIRFLOW_VERSION=2.8.0

# Extract the version of Python you have installed. If you're currently using a
# See above for supported versions.
PYTHON_VERSION="$(python --version | cut -d " " -f 2 | cut -d "." -f 1-2)"

CONSTRAINT_URL="https://raw.githubusercontent.com/apache/airflow/constraints-${
# For example this would install 2.8.0 with python 3.8: https://raw.githubuserc

pip install "apache-airflow==${AIRFLOW_VERSION}" --constraint "${CONSTRAINT_URL
```

You can then open a new terminal and run the airflow UI as a standalone application.

```
airflow standalone
```

On the first installation, this will create the `airflow` directory with config files and a local database.



Airflow local directory. Image by author.

The first airflow UI launch usually comes with a bunch of examples. We don't want them. In the `airflow.cfg` file, set `load_examples` to `False` .

Also, be sure to indicate the path to the dags folder:

```
dags_folder = /path/to/project/airflow/dags
```

You're now set up with Mlflow and Airflow. Let's build some pipelines now!

Steps

We'll build the pipeline module by module, which we call **step**. Preprocessing, training, inference, and so on. The primary advantage lies in the modularity it offers: **independence, portability, and testability**.

Each step is a Python class containing two main methods: `__init__` and `__call__` . Here's an example:

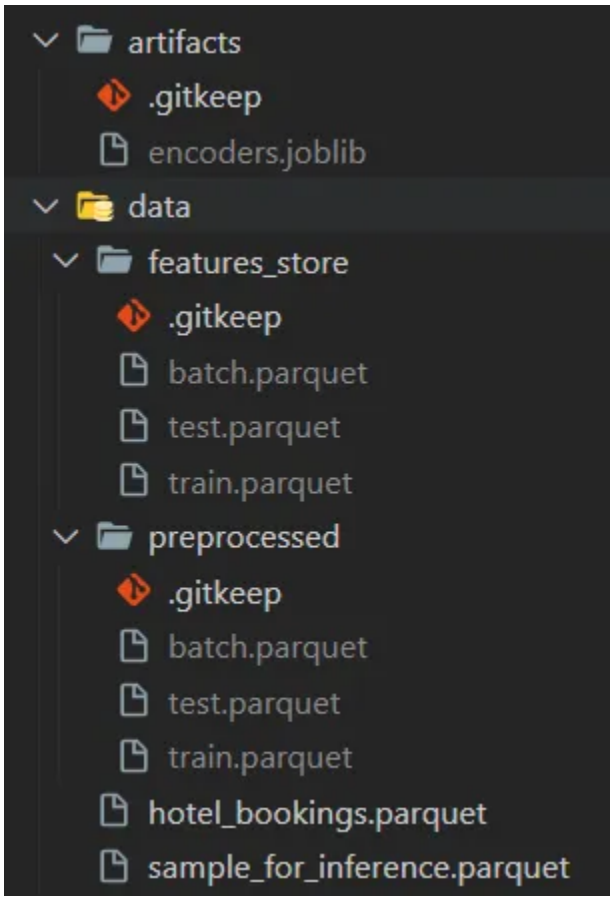
```
class PreprocessStep:
    """Preprocessing based on Exploratory Data Analysis done in `notebooks/0_ex

    Args:
        data_path (str): data.parquet file to preprocess. If `training_mode==Fa
        preprocessed_data_path (Optional[Path], optional): Output path of the p
        training_mode (Optional[bool], optional): Switch to training or inferen
    """

    def __init__(
        self,
        inference_mode: bool,
        preprocessing_data: PreprocessingData
    ) -> None:
        self.inference_mode = inference_mode
```

```
        self.preprocessing_data = preprocessing_data

    def __call__(self, data_path: Path) -> None:
        ...
```

The `__init__` method accepts as input all necessary data paths, including *output paths*, to execute the step. Additionally, specific options like *training_mode* can be incorporated, indicating whether this step is intended for the training or inference phase.

The `__call__` executes the step and accepts as inputs the incoming data paths and additional data coming from the previous step, like **ids**.

Since the size of the data that flows between steps is limited depending on the database *(64kB for mySQL for example!)*, each step accepts and returns *string values*, such as **data path** or **ids**, but not data!



Data and artifact paths. Image by author.

Data preprocessing

During the **data preprocessing** step, the data is cleaned and missing values are replaced following our initial exploratory data analysis. The data is then split into a **train** and a **test** set.

But why split the data now and not during the training phase only?

As we described previously in this article, some features are categorical values. To transform these features into numerical values, we'll use encoders: Ordinal and Target encoders from Sklearn. These encoders will be `fit()` to the data during the **feature engineering** step.

However, if these encoders are fitted on the entire dataset, it will insert bias during the evaluation stage. Therefore, we split the data during the preprocessing before the **feature engineering** step could see the data.

```
from pathlib import Path

import pandas as pd

from steps.utils.data_classes import PreprocessingData

class PreprocessStep:
    """
    Preprocessing based on Exploratory Data Analysis done in `notebooks/0_explo

    Args:
      inference_mode (bool): Training or inference mode.
      preprocessing_data (PreprocessingData): PreprocessingStep output paths.
    """
    def __init__(
        self,
        inference_mode: bool,
        preprocessing_data: PreprocessingData
    ) -> None:
        self.inference_mode = inference_mode
        self.preprocessing_data = preprocessing_data

    def __call__(self, data_path: Path) -> None:
        """
```

```
        Data is preprocessed then, regarding if inference=True or False:
            * False: Split data into train and test.
            * True: Data preprocessed then returned simply

        Args:
            data_path (Path): Input
        """

        preprocessed_df = pd.read_parquet(data_path)
        preprocessed_df = self._preprocess(preprocessed_df)

        if not self.inference_mode:
            train_df = preprocessed_df.sample(
                frac=TrainerConfig.train_size, random_state=TrainerConfig.rando
            )
            test_df = preprocessed_df.drop(train_df.index)
            train_df.to_parquet(self.preprocessing_data.train_path, index=False
            test_df.to_parquet(self.preprocessing_data.test_path, index=False)

        if self.inference_mode:
            preprocessed_df.to_parquet(self.preprocessing_data.batch_path, inde

    @staticmethod
    def _preprocess(df: pd.DataFrame) -> pd.DataFrame:
        """Preprocessing."""
        df["children"].fillna(0, inplace=True)
        df["country"].fillna("Unknown", inplace=True)
        df["agent"].fillna(0, inplace=True)
        df["company"].fillna(0, inplace=True)
        return df
```

As you might have noticed, the PreprocessStep class accepts a `PreprocessingData` object as input, which is a dataclass we designed to encapsulate all the requirements for this step.

```
from dataclasses import dataclass
from pathlib import Path
from typing import Optional

@dataclass
class PreprocessingData:
    """Data paths generated by PreprocessingStep.
    Regarding if it's the training or inference pipeline, the correct paths nee
```

```
    (respectively train_path, test_path / batch_path)
    """
    train_path: Optional[Path] = None
    test_path: Optional[Path] = None
    batch_path: Optional[Path] = None
```

Once the data is cleaned and split, the **feature engineering** step takes place.

Feature engineering

We describe feature engineering as the stage where features (columns in this case) are modified, fused, or deleted for the model training or inference.

The feature engineering step is divided into two parts.

We first remove the features having a causality relationship with the target.

We then transform the categorical values into numerical values using the Ordinal and Target Encoders from Sklearn.

However, this step poses several challenges since it carries **three distinct processes:**

- The **training phase** in which encoders are tuned on the training data, then saved as artifacts.
- The **evaluation phase** in which encoders are loaded and then applied to the test dataset.
- The **inference phase** mirrors the evaluation phase, with the only distinction being the absence of a *target*.

To save and load the tuned encoders, we create a *dataclasse* object containing both encoders, their respective features, and the target features required by the Target encoder.

```
from dataclasses import dataclass
from pathlib import Path
from typing import Iterable
```

```
import joblib

from sklearn.preprocessing import OrdinalEncoder, TargetEncoder

@dataclass
class FeaturesEncoder:
    """Encoders artifact dumped and loaded during the feature_engineering step.

    ordinal_encoder: OrdinalEncoder
    target_encoder: TargetEncoder
    base_features: Iterable[str]
    ordinal_features: Iterable[str]
    target_features: Iterable[str]
    target: str

    def to_joblib(self, path: Path) -> None:
        """Dump artifact as a joblib file.

        Args:
            path (Path): Encoders path
        """
        joblib.dump(self, path)
```

The `FeatureEngineeringStep` class is built as follows.

The class contains additional methods we'll not explain in detail in this article: `fit_transform()` or `fit()` , but feel free to check the script in the Github repository.

```
from pathlib import Path
from typing import Optional

import pandas as pd

from steps.utils.data_classes import FeaturesEngineeringData

class FeatureEngineeringStep:
    """Feature engineering: transform features for model training and inference

    def __init__(
        self,
        inference_mode: bool,
        feature_engineering_data: FeaturesEngineeringData
    ) -> None:
```

```
        """
        Args:
            inference_mode (bool): Whether the step is used in the training or
            feature_engineering_data (FeaturesEngineeringEData): Paths relative
        """
        self.inference_mode = inference_mode
        self.feature_engineering_data = feature_engineering_data


    def __call__(
        self,
        train_path: Optional[Path] = None,
        test_path: Optional[Path] = None,
        batch_path: Optional[Path] = None,
    ) -> None:
        """
        Input data paths depending on whether it's training (train, test) or in

        Args:
            train_path (Optional[Path], optional): Input train path. Defaults t
            test_path (Optional[Path], optional): Input test path. Defaults to
            batch_path (Optional[Path], optional): input batch path. Defaults t
        """
        if not self.inference_mode:
            train_df = pd.read_parquet(train_path)
            test_df = pd.read_parquet(test_path)
            self.fit_transform(
                df=train_df,
                output_path=self.feature_engineering_data.train_path
            )
            self.transform(
                df=test_df,
                output_path=self.feature_engineering_data.test_path
            )

        if self.inference_mode:
            batch_df = pd.read_parquet(batch_path)
            self.transform(
                batch_df,
                output_path=self.feature_engineering_data.batch_path
            )

    def fit_transform(
          self,
          df: pd.DataFrame,
          output_path: Path
      ) -> None:
          """Fit encoders on data and store the encoder into the features store
```

```
            The processed data is then stored.

            Args:
                df (pd.DataFrame): Data to train encoders and to transform.
                output_path (Path): Data path after encoding.
            """
            LOGGER.info("Start features engineering 'fit_transform'.")
            ...


    def transform(
            self,
            df: pd.DataFrame,
            output_path: Path
        ) -> None:
            """Transform data based on trained encoders.

            Args:
                df (pd.DataFrame): Data to transform.
                output_path (Path): Transformed data path.
            """
            LOGGER.info("Start features engineering 'transform'.")
            ...
```

Once the dataset is prepared, we can then go to the **training** stage (or **inference** in case of inference pipeline).

Training

For the **training** stage, we train the cancelation predictor with the **Gradient Boosting Classifier** algorithm from the Sklearn library. It's usually a good choice when working with tabular data.

A requirement for this project is the parametrization of the pipeline: for example, any model parameterization should be done without modifying the training script. Therefore, we add `params` as a variable of the `__init__` method, containing the parameters required by the model.

```
from steps.config import TrainerConfig

class TrainStep:
    """Training step tracking experiments with MLFlow.
```

```
    In this case, GradientBoostingClassifier has been picked, and the chosen me
    * precision
    * recall
    * roc_auc

    Args:
        params (Dict[str, Any]): Parameters of the model. Have to match the mod
        model_name (str, optional): Additional information for experiments trac

    def __init__(
            self,
            params: Dict[str, Any],
            model_name: str = TrainerConfig.model_name
    ) -> None:
        self.params = params
        self.model_name = model_name
```

All parameters of the entire project are stored in a single `config` file to avoid floating variables in the code.

```
class TrainerConfig:
    model_name ="gradient-boosting"
    random_state = 42
    train_size = 0.2
    shuffle = True
    params = {
        "n_estimators": 100,
        "min_samples_split": 2,
        "min_samples_leaf": 1
    }
```

Since the `TrainStep` is invoked in the pipeline script, updating the pipeline to run a new model parametrization becomes a straightforward process, simplifying experimentation.

Regarding the **metrics**, we choose **Precision, Recall,** and **ROC-AUC** to evaluate our model performances, which is common in **classification tasks.**

To log these metrics and the model once trained, we use **Mlflow.**

Before starting to log, we need to indicate the Mlflow server we created on the port `8000` . We also need to indicate the name of the project, also called *Experiment*.

> *If not already existing, you need to create a new experiment. In this case, it is named "cancelation_estimator"*

```
import mlflow

mlflow.set_tracking_uri("http://0.0.0.0:8000")
mlflow.set_experiment("cancelation_estimator")
```

After training the model, the run will be saved, and a unique ID will be assigned to it. This run serves as the reference for retrieving metrics and artifacts generated during the process.

Consequently, if we intend to use any components of this run, such as registering the model or obtaining the best run, we need to return the `run_id` within the pipeline, using **Xcoms.**

> *In Airflow, **Xcoms** (short for "cross-communications") are a mechanism that let **Steps** talk to each other, as by default, **Steps** are entirely isolated and may be running on entirely different machines.*

Let's talk now about the artifact.

An artifact, whether compressed or not, is a folder encompassing the model (or any objects like data or encoders) and all the necessities for its execution. This includes scripts, the `requirements.txt` file, and more... Mlflow provides different artifact templates to run the trained model.

Since we used Sklearn, let's use the `sklearn` artifact from Mlflow to log the model.

```
mlflow.sklearn.log_model(
```

```
    sk_model=model,                                    # Gradient Boosting model
    artifact_path="model-artifact",                    # Artifact path: runs:/<mlflo
)
```

Similarly, in certain scenarios, customization of the artifact may be necessary. It can be easily done using `mlflow.pyfunc.PythonModel` . Check an example of a custom artifact in the repository located at `steps/utils/_artifact.py` or check the official documentation.

Here's the complete script to train and evaluate the model, log metrics, and log the model into an artifact. The `run_id` is returned to be passed as an **Xcom** value to the next step.

```
# TrainStep class
def __call__(
    self,
    train_path: Path,
    test_path: Path,
    target: str
) -> None:

mlflow.set_tracking_uri(MlFlowConfig.uri)
mlflow.set_experiment(MlFlowConfig.experiment_name)

with mlflow.start_run():

    train_df = pd.read_parquet(train_path)
    test_df = pd.read_parquet(test_path)

    # Train
    gbc = GradientBoostingClassifier(
        random_state=TrainerConfig.random_state,
        verbose=True,
        **self.params
    )
    model = gbc.fit(
        train_df.drop(target, axis=1),
        train_df[target]
    )

    # Evaluate
```

```
        y_test = test_df[target]
        y_pred = model.predict(test_df.drop(target, axis=1))

        # Metrics
        precision = precision_score(y_test, y_pred)
        recall = recall_score(y_test, y_pred)
        roc_auc = roc_auc_score(y_test, y_pred)
        print(classification_report(y_test, y_pred))

        metrics = {
            "precision": precision,
            "recall": recall,
            "roc_auc": roc_auc
        }

        # Mlflow
        mlflow.log_params(self.params)
        mlflow.log_metrics(metrics)
        mlflow.set_tag(key="model", value=self.model_name)
        mlflow.sklearn.log_model(
            sk_model=model,
            artifact_path=MlFlowConfig.artifact_path,
        )

        return {"mlflow_run_id": mlflow.active_run().info.run_id} # The output of t
```

If we now look at the Mlflow UI, we can spot our run last run containing the logged metrics and artifact.

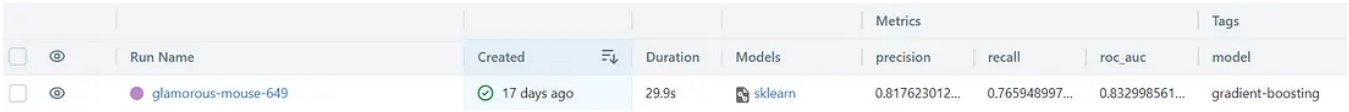


Image by author.

If we click on the run, we access the run card with further details. This run card is fully customizable adding scalability and customizability to your project.

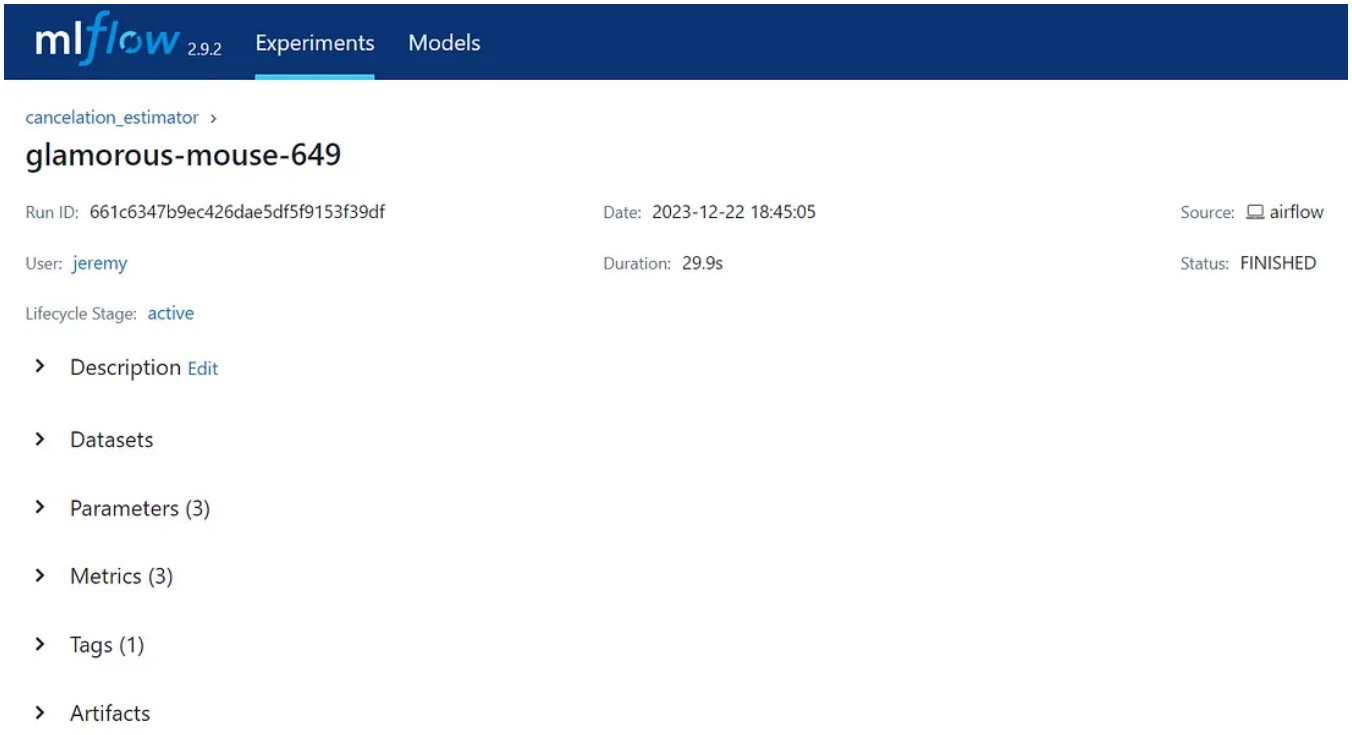


Image by author.

On the upper left, we notice the `run_id`, the unique run identifier. We will use this ID to retrieve the model and all the metrics from this run to register and load the model artifact during inference.

Register the model

During Machine Learning, runs are compared then the best one is selected to be saved in the **model registry**. The model registry is the last step before deploying a model in production. It tracks model versions, enabling continuous development in projects.

In this project, we will automatically register a new model if its performances exceed the one already in the model registry. If no model has been registered, we take care of creating a new model.

To decide if a model should be registered, we will compare a specific metric, such as **roc-auc**. We add customization to the pipeline by adding options to the `__init__` method, which will be called in the *dag* script.

We also add `criteria`, a coefficient to consider if the new metric is significantly

better than the one already registered. It simply follows this equation: `if metric > registered_metric * (1 + criteria)` .

```
from typing import Literal

class ConditionStep:
    """Condition to register the model.

    Args:
        criteria (float): Coefficient applied to the metric of the model regist
        metric (str): Metric as a reference. Can be `["precision", "recall", or
    """

    def __init__(
        self,
        criteria: float,
        metric: Literal["roc_auc", "precision", "recall"]
    ) -> None:
        self.criteria = criteria
        self.metric = metric
```

For this step, we need two things: the `active_run_id` provided after the **training** step, and the `last_registered_model_version_run_id` to get all the information on the existing model in the model registry.

Here's the `__call__` method of the `ConditionStep` class:

```
import mlflow

from steps.config import MlFlowConfig

def __call__(self, mlflow_run_id: str) -> None:
        """
        Compare the metric from the last run to the model in the registry.
        if `metric_run > registered_metric*(1 + self.criteria)`, then the model

        Args:
          mlflow_run_id (str): active run id
        """
```

```
        LOGGER.info(f"Run_id: {mlflow_run_id}")
        mlflow.set_tracking_uri(MlFlowConfig.uri)

        run = mlflow.get_run(run_id=mlflow_run_id)
        metric = run.data.metrics[self.metric]

        registered_models = mlflow.search_registered_models(
            filter_string=f"name = '{MlFlowConfig.registered_model_name}'"
        )

        if not registered_models:
            mlflow.register_model(
                model_uri=f"runs:/{mlflow_run_id}/{MlFlowConfig.artifact_path}"
                name=MlFlowConfig.registered_model_name,
            )
            LOGGER.info("New model registered.")

        latest_registered_model = registered_models[0]
        registered_model_run = mlflow.get_run(
            latest_registered_model.latest_versions[0].run_id
        )
        registered_metric = registered_model_run.data.metrics[self.metric]

        if metric > registered_metric * (1 + self.criteria):
            mlflow.register_model(
                model_uri=f"runs:/{mlflow_run_id}/{MlFlowConfig.artifact_path}"
                name=MlFlowConfig.registered_model_name,
            )
            LOGGER.info("Model registered as a new version.")
```

Once this step is done, if the model is validated, it is registered into the Mlflow models registry.

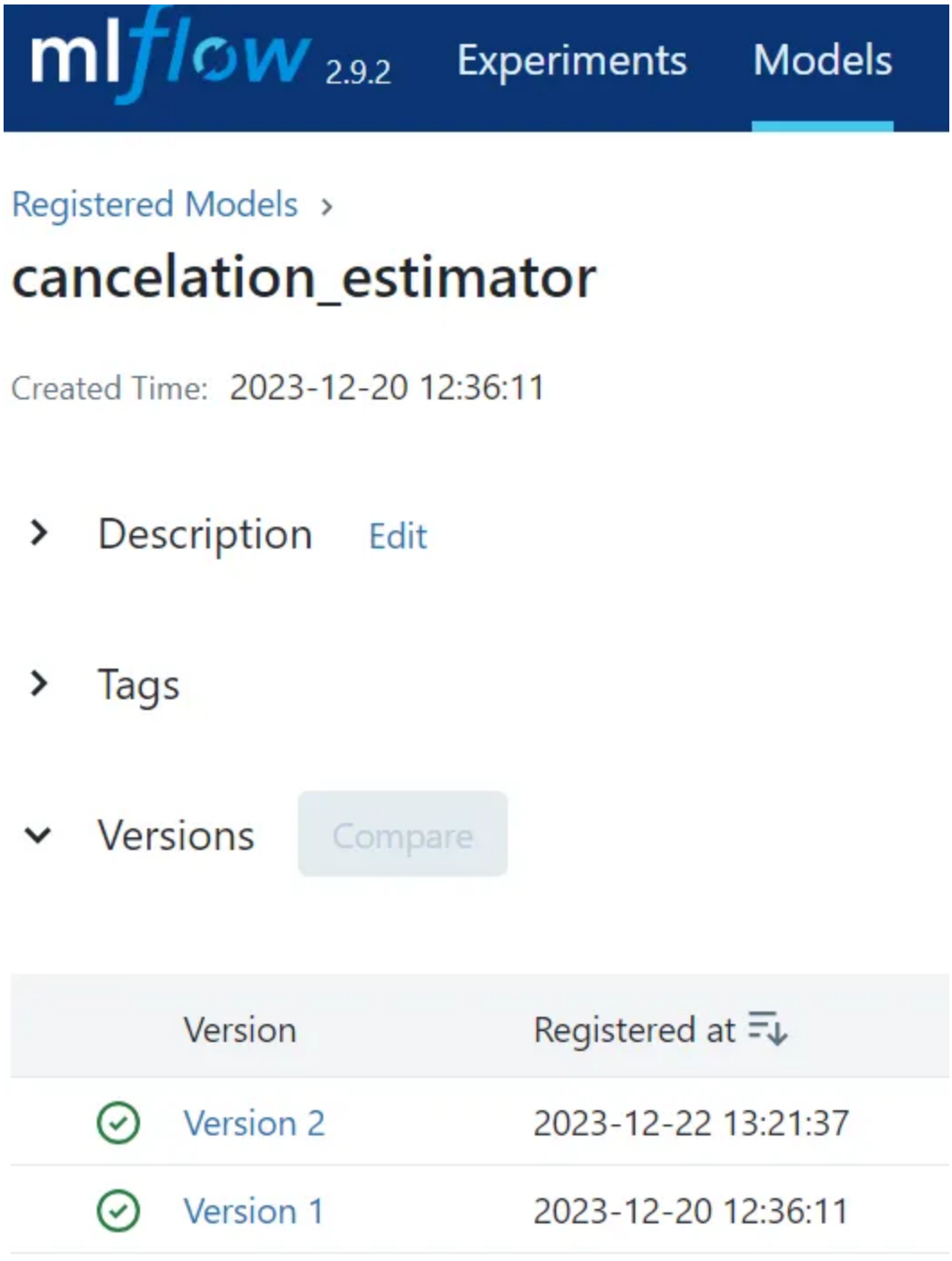


Image by author.

Inference

Finally, the last module of our pipelines: the **Inference** step.

This stage loads the model artifact from the model registry and runs batch predictions on **preprocessed data.**

> *In theory, this **step** should load the model from a deployed endpoint, but for the sake of simplicity, we haven't considered cloud deployment in this project.*

By using the Sklearn artifact provided by Mlflow, we can easily load and perform batch prediction using `model.predict(batch)` .

The prediction is then outputted as the dag's output using `return` . We can also imagine storing the data directly into a database.

```
import logging
from pathlib import Path
import json
from typing import List

import mlflow
import pandas as pd

from steps.config import MlFlowConfig

LOGGER = logging.getLogger(__name__)

class InferenceStep:
    "Get the model from the model registry and predict in batch"

    def __call__(self, batch_path: Path) -> List[int]:
        """Use the MLFlow artifact built-in predict.

        Args:
            batch_path (Path): Input batch_path

        Return (List[int]):
            Predictions
        """
        model = self._load_model(
            registered_model_name=MlFlowConfig.registered_model_name
        )
        batch = self._load_batch(batch_path)
        if model:
            # Transform np.ndarray into list for serialization
            prediction = model.predict(batch).tolist()
            LOGGER.info(f"Prediction: {prediction}")
            return json.dumps(prediction)
```

```
        else:
            LOGGER.warning(
                "No model used for prediction. Model registry probably empty."
            )

    @staticmethod
    def _load_model(registered_model_name: str):
        """Load model from model registry.

        Args:
            registered_model_name (str): Name

        Returns:
            Model artifact
        """
        mlflow.set_tracking_uri(MlFlowConfig.uri)
        models = mlflow.search_registered_models(
            filter_string=f"name = '{registered_model_name}'"
        )
        LOGGER.info(f"Models in the model registry: {models}")
        if models:
            latest_model_version = models[0].latest_versions[0].version
            LOGGER.info(
                f"Latest model version in the model registry used for predictio
            )
            model = mlflow.sklearn.load_model(
                model_uri=f"models:/{registered_model_name}/{latest_model_versi
            )
            return model
        else:
            LOGGER.warning(
                f"No model in the model registry under the name: {MlFlowConfig.
            )

    @staticmethod
    def _load_batch(batch_path: Path) -> pd.DataFrame:
        """Load dataframe from path"""
        batch = pd.read_parquet(batch_path)
        LOGGER.info(f"Batch columns: {batch.columns}")
        return batch
```

We are now done with each step that composes the training and the inference

pipeline. Let's connect them using **Airflow**.

Build DAGs with Airflow

The process of building dags is now straightforward.

Airflow dags are composed of **Operators**, objects designed to perform a task, such as `BashOperator` or `PythonOperator` . There are various types of operators handled by Airflow.

The library also provides hooks for the pipeline author to define their own parameters, macros, and templates, such as *Mlflow* or *Sagemaker*.

This is why using an orchestrator in your project is so handy: from one place, you control every step of your pipeline, each of them requiring its own requirements or computation resource.

To build a dag with Airflow, we need to instantiate the `DAG` class object using `with` . For our purpose, we don't need to schedule the pipeline, meaning that the pipeline will run only if triggered manually.

```
from airflow import DAG

default_args = {
    "owner": "user",                        # user's name
    "depends_on_past": False,               # keeps a task from getting triggered
    "retries": 0,                           # Number of retries for a dag
    "catchup": False,                       # Run the dag from the start_date to t
}

with DAG(
    "training-pipeline",                    # Dag name
    default_args=default_args,              # Default dag's arguments that can be
    start_date=datetime(2023, 12, 19),      # Reference date for the scheduler (ma
    tags=["training"],                      # tags
    schedule=None,                          # No repetition
) as dag:
```

Don't hesitate to check the official documentation to dig more into the tool. The

possibilities Airflow provides are quite wide.

```
# training_pipeline.py
from datetime import datetime

from airflow import DAG
from airflow.operators.python import PythonOperator

from steps.preprocess_step import PreprocessStep
from steps.train_step import TrainStep
from steps.condition_step import ConditionStep
from steps.feature_engineering_step import FeatureEngineeringStep
from steps.utils.data_classes import PreprocessingData, FeaturesEngineeringData
from steps.config import (
    TRAINING_DATA_PATH,
    TrainerConfig,
    ConditionConfig,
    PreprocessConfig,
    FeatureEngineeringConfig,
)

# Preparation
inference_mode = False
preprocessing_data = PreprocessingData(
    train_path=PreprocessConfig.train_path,
    test_path=PreprocessConfig.test_path
)
feature_engineering_data = FeaturesEngineeringData(
    train_path=FeatureEngineeringConfig.train_path,
    test_path=FeatureEngineeringConfig.test_path,
    encoders_path=FeatureEngineeringConfig.encoders_path,
)
target = FeatureEngineeringConfig.target

# Steps
preprocess_step = PreprocessStep(
    inference_mode=inference_mode,
    preprocessing_data=preprocessing_data
)
feature_engineering_step = FeatureEngineeringStep(
    inference_mode=inference_mode,
    feature_engineering_data=feature_engineering_data
)
train_step = TrainStep(
    params=TrainerConfig.params
)
```

```
condition_step = ConditionStep(
    criteria=ConditionConfig.criteria,
    metric=ConditionConfig.metric
)

default_args = {
    "owner": "user",                        # user's name
    "depends_on_past": False,               # keeps a task from getting triggered
    "retries": 0,                           # Number of retries for a dag
    "catchup": False,                       # Run the dag from the start_date to t
}

with DAG(
    "training-pipeline",                    # Dag name
    default_args=default_args,              # Default dag's arguments that can be
    start_date=datetime(2023, 12, 19),      # Reference date for the scheduler (ma
    tags=["training"],                      # Tags
    schedule=None,                          # No repetition
) as dag:
    preprocessing_task = PythonOperator(
        task_id="preprocessing",
        python_callable=preprocess_step,
        op_kwargs={"data_path": TRAINING_DATA_PATH},
    )
    feature_engineering_task = PythonOperator(
        task_id="feature_engineering",
        python_callable=feature_engineering_step,
        op_kwargs={
            "train_path": preprocessing_data.train_path,
            "test_path": preprocessing_data.test_path,
        },
    )
    training_task = PythonOperator(
        task_id="training",
        python_callable=train_step,
        op_kwargs={
            "train_path": feature_engineering_data.train_path,
            "test_path": feature_engineering_data.test_path,
            "target": target
        },
    )
    validation_task = PythonOperator(
        task_id="validation",
        python_callable=condition_step,
        op_kwargs=training_task.output,
    )

    preprocessing_task >> feature_engineering_task >> training_task >> validati
```

Here's how the training pipeline script is built:

1. We import every **step** class object and initialize them using their respective `__init__` method.
2. We initialize the dag using `with DAG()` and compose it with `PythonOperator` .
3. Since we used the `__call__` method, the instantiate **step** class object becomes a callable. We input the required parameters using `op_kwargs` , which accepts a `dict` object.
4. You notice that the argument input of the `validation_task` is actually the output of the last step. If you come back the `TrainingStep` class, we outputted the `mlflow_run_id` required by the `ConditionStep` class. The output value is stored as an **Xcom.** *See the image below.*
5. Finally, we organize the task using the `>>` operator

XCom

Key	Value
return_value	{'mlflow_run_id': 'fd279e7e6648411fbe6a55ffd31e35bc'}

Xcom value

The pipeline looks like this:

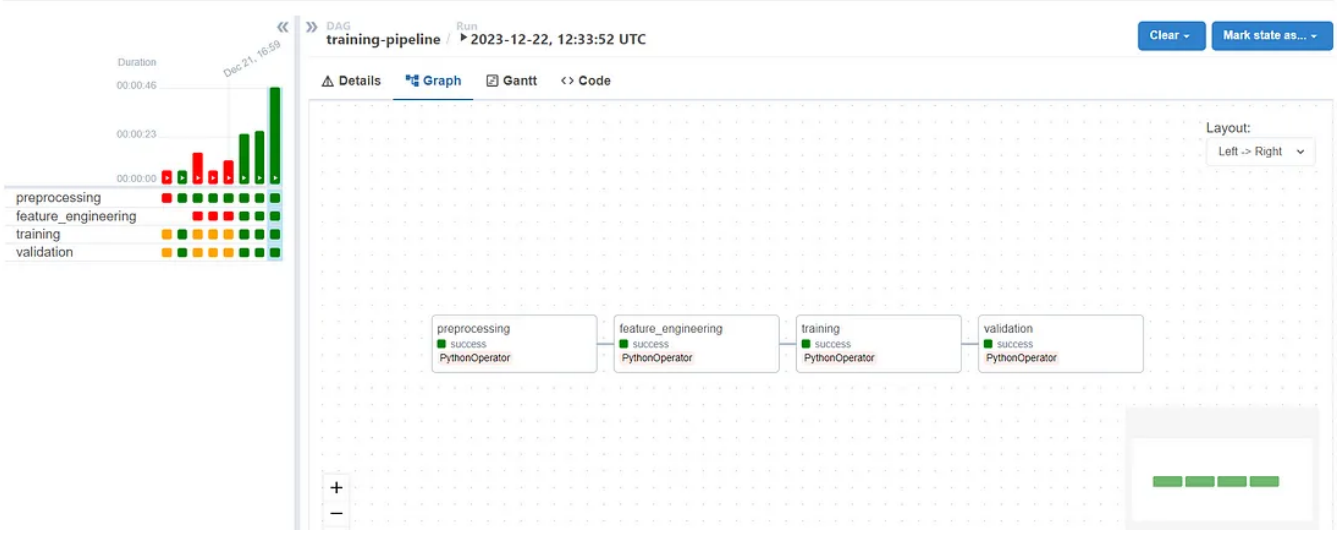
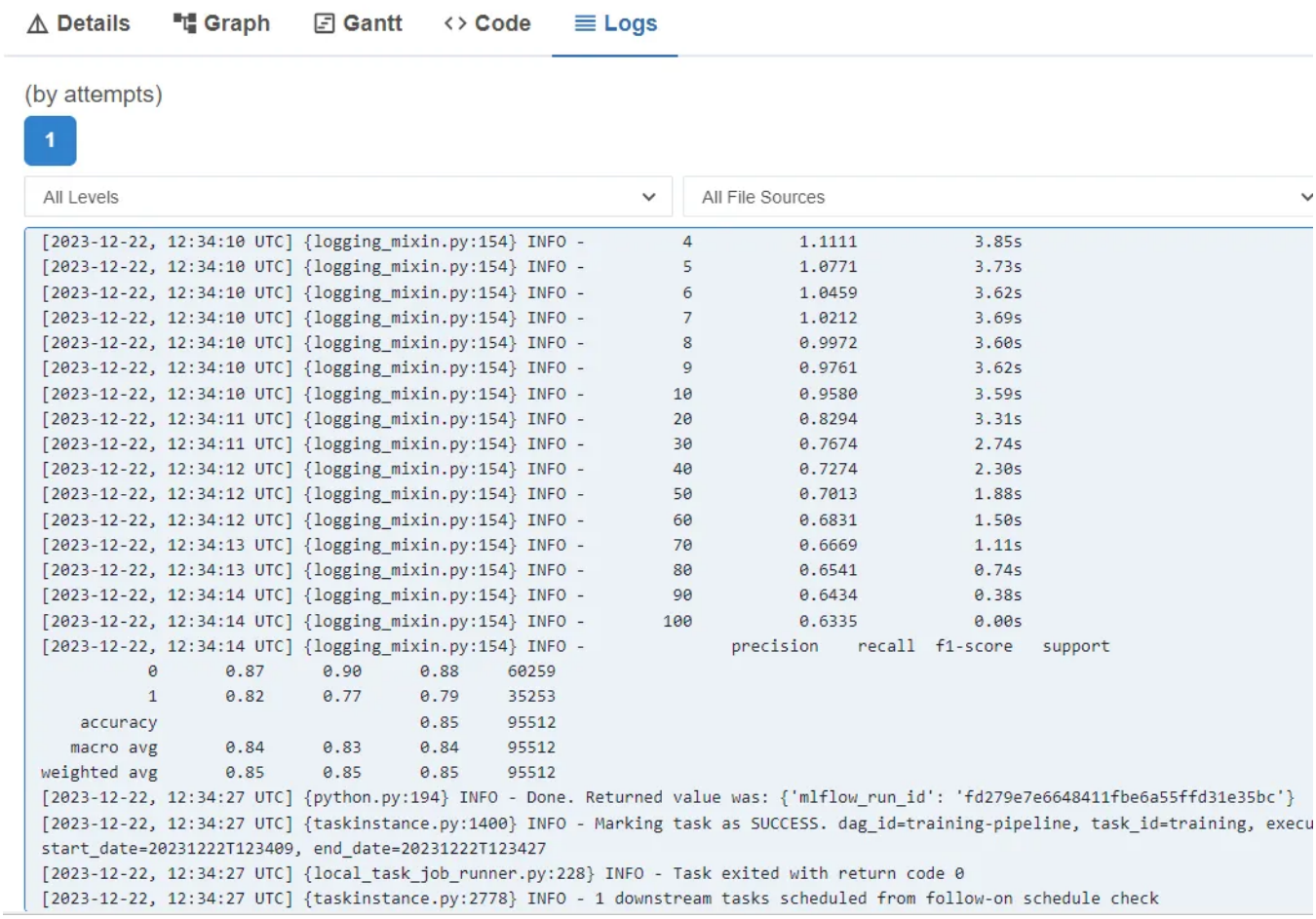


Image by author.

> ***Notes:*** *Debugging Arflow dags is super easy with logs. Don't hesitate to add logging to your code as much as possible to be able to debug quickly whatever happens in production once deployed.*



Training step logs in Airflow. Image by author.

We do the same with the **inference pipeline:**

```
from datetime import datetime

from airflow import DAG
from airflow.operators.python import PythonOperator

from steps.preprocess_step import PreprocessStep
from steps.inference_step import InferenceStep
from steps.feature_engineering_step import FeatureEngineeringStep
from steps.utils.data_classes import PreprocessingData, FeaturesEngineeringData
from steps.config import (
    FeatureEngineeringConfig,
    INFERENCE_DATA_PATH,
    PreprocessConfig,
)


# Preparation
inference_mode = True
preprocessing_data = PreprocessingData(
    batch_path=PreprocessConfig.batch_path
)
features_engineering_data = FeaturesEngineeringData(
    batch_path=FeatureEngineeringConfig.batch_path,
    encoders_path=FeatureEngineeringConfig.encoders_path,
)

# Steps
preprocess_step = PreprocessStep(
    inference_mode=inference_mode,
    preprocessing_data=preprocessing_data
)
feature_engineering_step = FeatureEngineeringStep(
    inference_mode=inference_mode,
    feature_engineering_data=features_engineering_data
)
inference_step = InferenceStep()


default_args = {
    "owner": "user",
    "depends_on_past": False,
```

```
    "retries": 0,
    "catchup": False,
}

with DAG(
    "inference-pipeline",
    default_args=default_args,
    start_date=datetime(2023, 12, 20),
    tags=["inference"],
    schedule=None,
) as dag:
    preprocessing_task = PythonOperator(
        task_id="preprocessing",
        python_callable=preprocess_step,
        op_kwargs={
            "data_path": INFERENCE_DATA_PATH
        }
    )
    feature_engineering_task = PythonOperator(
        task_id="feature_engineering",
        python_callable=feature_engineering_step,
        op_kwargs={
            "batch_path": preprocessing_data.batch_path
        }
    )
    inference_task = PythonOperator(
        task_id="inference",
        python_callable=inference_step,
        op_kwargs={
            "batch_path": features_engineering_data.batch_path
        }
    )

    preprocessing_task >> feature_engineering_task >> inference_task
```

If we run this pipeline, we get the prediction of the provided batch as an Xcom value:

XCom

Key	Value
return_value	[0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1]

Inference pipeline output. Image by author.

Conclusions

This concludes this article on building a machine learning pipeline using Airflow coupled with Mlflow.

Not only did we train a model to predict at **82% Precision** if a reservation will be canceled, but we also **built a complete and robust pipeline for the preprocessing, training, and inference stages**. This enables the project to be reproducible, makes model experimentations fast, deployment easier thanks to the model registry, and reduces the risk of data inadequacy by making the **preprocessing** and **feature engineering** steps modular and used in both pipelines.

This project served as an opportunity to experiment with **Airflow** and **Mlflow** in a real-world scenario. The primary challenge in Machine Learning is not so much about the models or their performances, **but rather about how to develop and deploy these models robustly.**

I hope this article was informative.

I would like to add that the architectural design outlined in this article is highly contingent on the project requirements and can vary based on business or development needs. For instance, the inference pipeline was constructed with the **preprocessing** and **feature engineering** step directly integrated to ensure the data is correctly processed for model predictions. However, in scenarios where a feature store is employed as an intermediate stage before the model "sees" the data, the need for these preprocessing steps in the inference pipeline may be eliminated, requiring only the inference step. Additionally, how predictions are displayed can also differ, whether it's through an API or directly stored in a database.

For these reasons, coding each step as a modular and independent class object makes the pipeline manageable and adaptable to almost any scenario.

Going further

Some parts of the work can be improved :

- The data analysis can be pushed further and the **Shap** library could be used to

understand the model predictions and estimate the importance of its features.

- **Exceptions**, along with **data validation** using **Pydantic**, can be configured to manage errors originating from outside the pipeline, such as issues with the ingested data. This challenge is quite common in production environments.
- We can incorporate **Data Version Control (DVC)** and/or leverage a **Feature Store** to monitor the data used for training models and its lineage. This can be seamlessly implemented thanks to the Airflow orchestrator.
- Since the code is highly modular, it provides an ideal setup for unit tests and integration tests as part of a CI/CD process.

Code repository

You can find the entire code structure on the following link:

GitHub - jeremyarancio/reservation_cancellation_prediction: Predict if a reservation will be...

Predict if a reservation will be canceled using robust Machine Learning pipelines with Airflow and Mlflow - GitHub ...

github.com

If you liked this article, **feel free to join my newsletter. I share my content about Machine Learning, MLOps, and entrepreneurship.**

You can reach out to me on Linkedin, or check my Github.

If you're a business and want to implement Machine Learning into your product, you can also **book a call**.

See you around and happy coding!

References

- [1] — Nuno Antonio, Ana de Almeida, Luis Nunes, **Hotel booking demand datasets**, Data in Brief, Volume 22, 2019, Pages 41–49, ISSN 2352–3409, https://doi.org/10.1016/j.dib.2018.11.126, (license: CC BY 4.0 DEED).

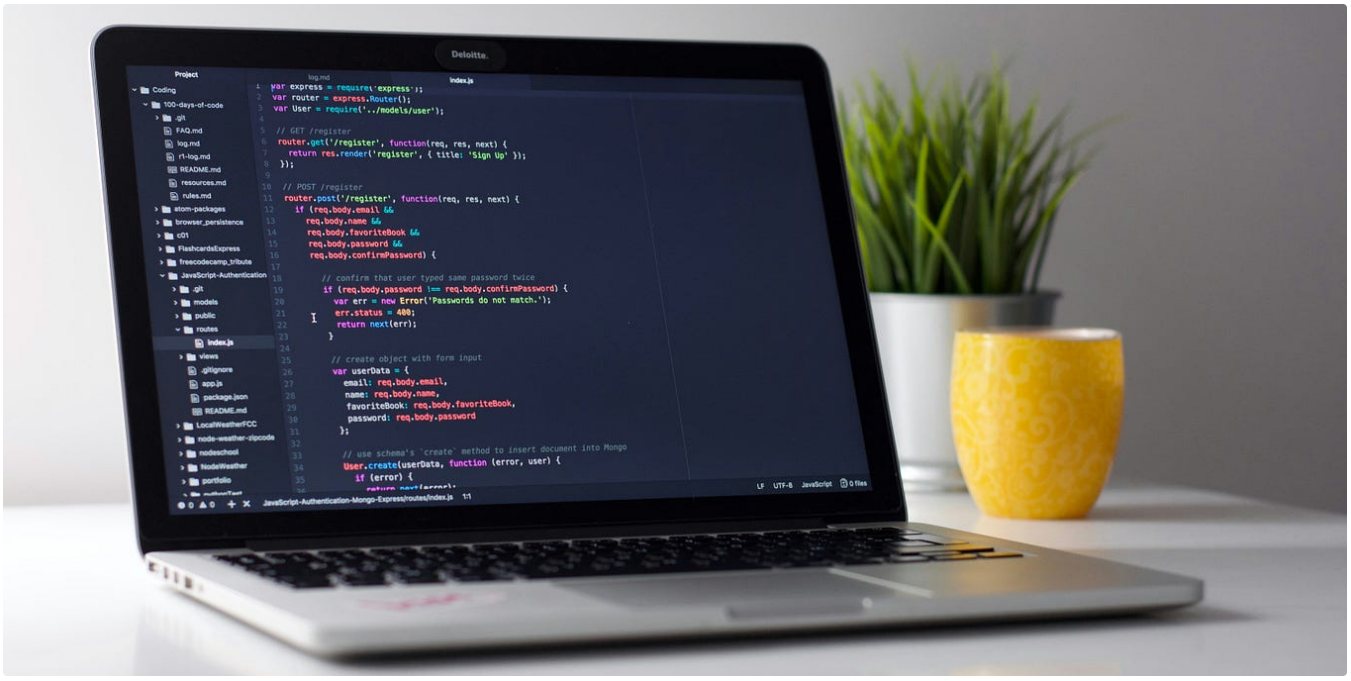
Machine Learning | Airflow | Mlflow | Mlops | Hands On Tutorials

Written by Jeremy Arancio

1.7K Followers · Writer for Towards Data Science

NLP Engineer & AI-ndependant - I help companies leveraging texts using Machine Learning! - Website: https://linktr.ee/jeremyarancio

More from Jeremy Arancio and Towards Data Science



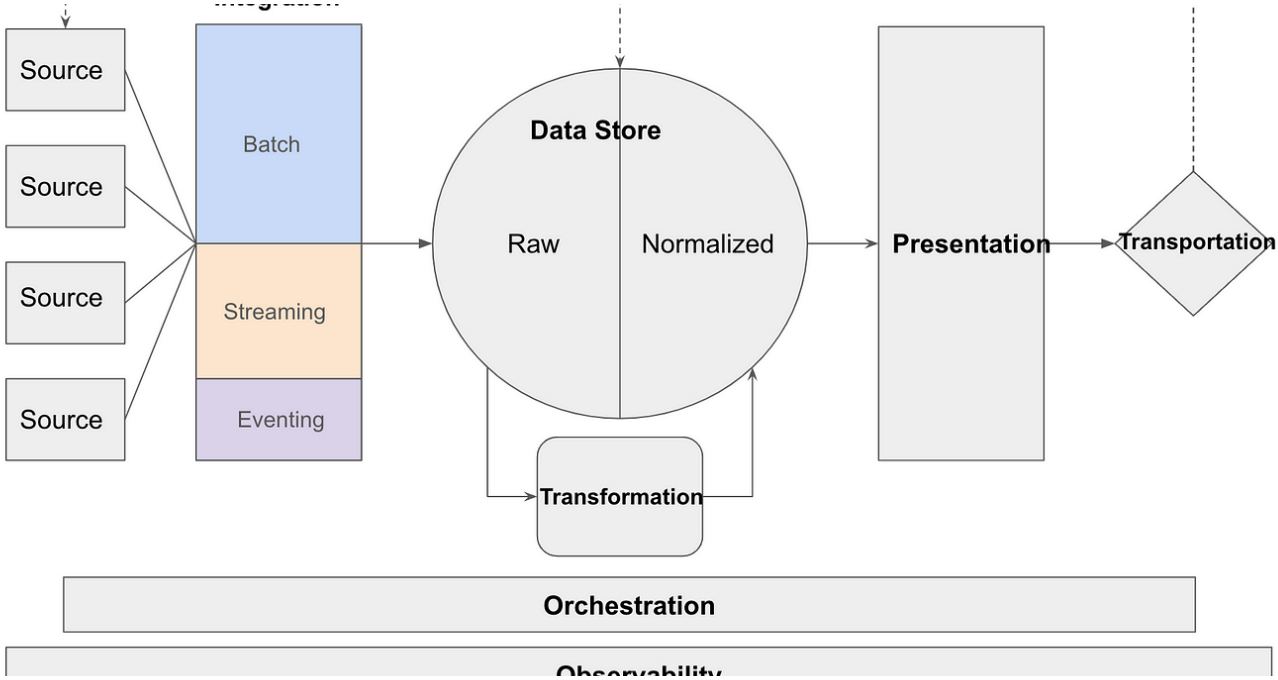
Jeremy Arancio in Towards Data Science

Scale your Machine Learning Projects with SOLID principles

How to write code that scales and accelerates your work as a data scientist or machine learning engineer.

✨ · 13 min read · 5 days ago

349 6



Dave Melillo in Towards Data Science

Building a Data Platform in 2024

How to build a modern, scalable data platform to power your analytics and data science projects (updated)

9 min read · Feb 6, 2024

 2.2K  32 

Patrick Brus in Towards Data Science

How to Write Clean Code in Python

Top takeaways from the book Clean Code

· 21 min read · Feb 24, 2024

Jeremy Arancio in Towards AI

Why are AI Products Doomed to Fail?

After one year of implementing AI features for various businesses, I share my perspective on the mistakes I see companies making with LLMs...

✨ · 17 min read · Nov 17, 2023

2.1K 41

See all from Jeremy Arancio

See all from Towards Data Science

Recommended from Medium

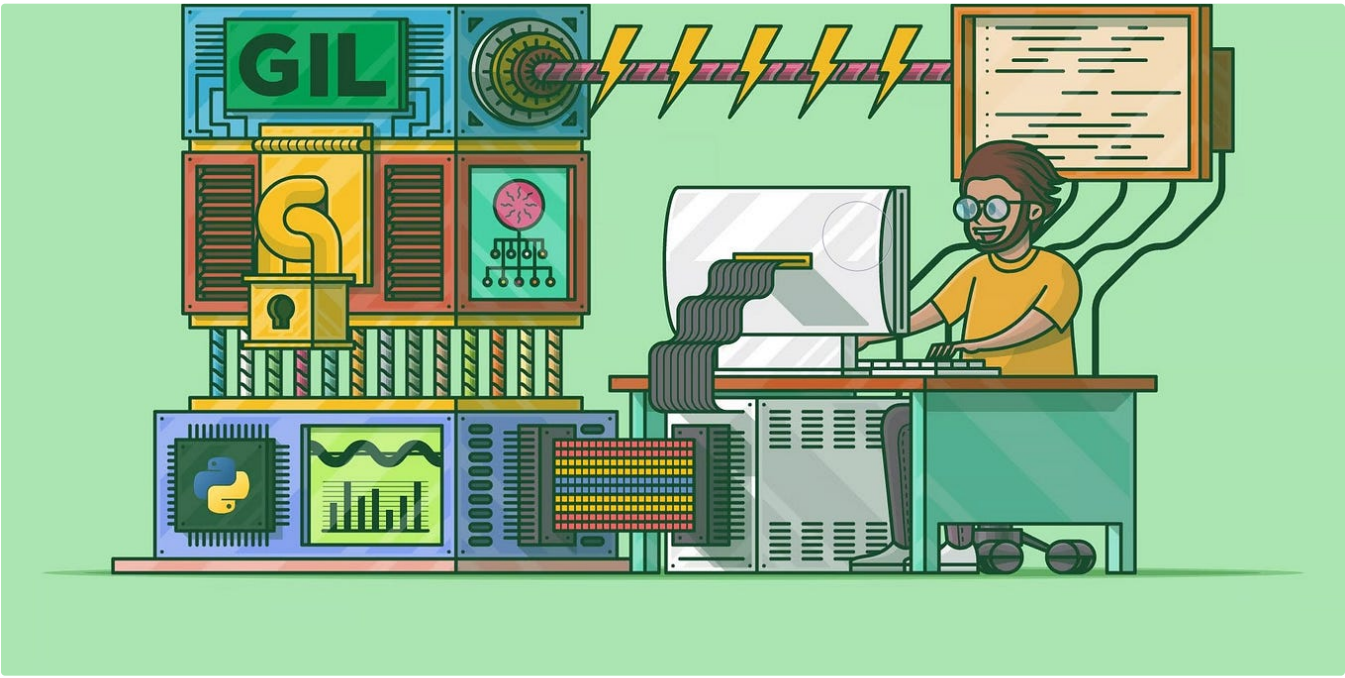
Khouloud El Alami in Towards Data Science

I Spent $96k To Become a Data Scientist. Here Are 5 Crucial Lessons All Beginners Must Know

or how to manage your data science education when you know nothing about data science

· 14 min read · 4 days ago

746 11



Dylan Cooper in Python in Plain English

Python is Ushering in Real Multi-Threading

The Global Interpreter Lock(GIL) can be removed. From then on, Python will no longer be what people call pseudo-multithreading.

· 5 min read · 5 days ago

837 5

Lists

Predictive Modeling w/ Python

20 stories · 1004 saves

Practical Guides to Machine Learning

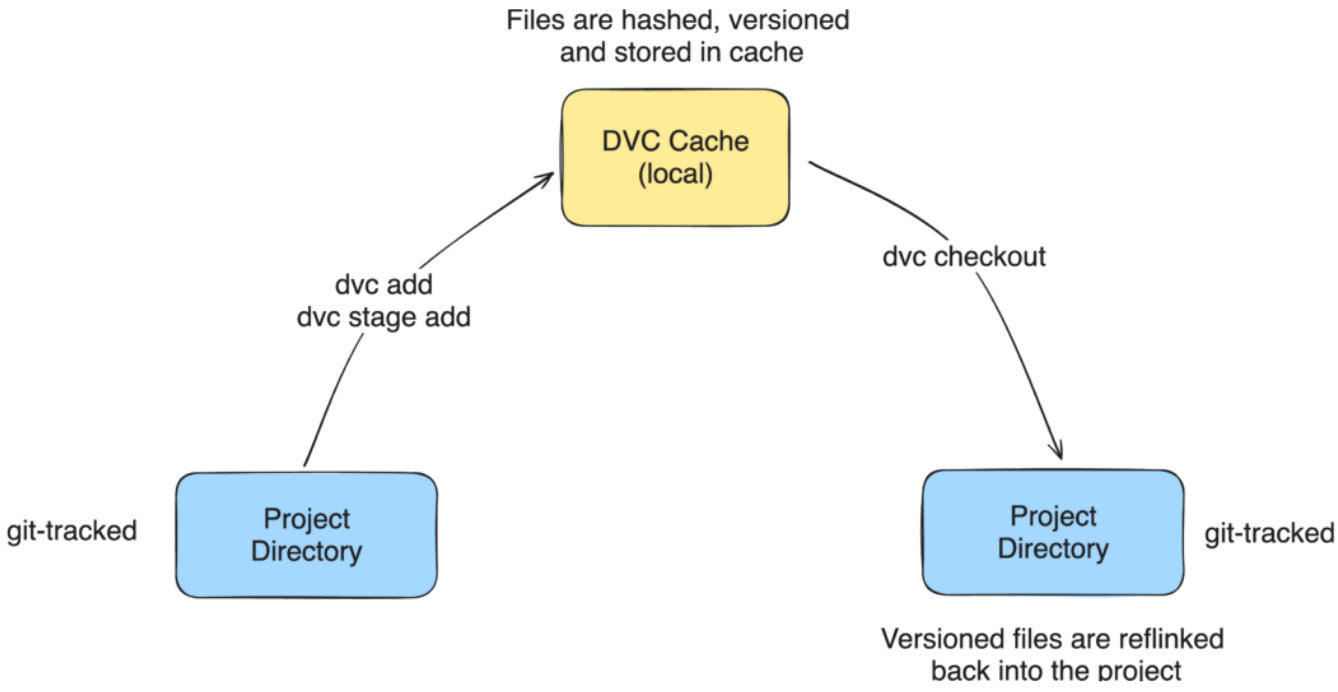
10 stories · 1202 saves

Natural Language Processing

1284 stories · 778 saves

The New Chatbots: ChatGPT, Bard, and Beyond

12 stories · 334 saves



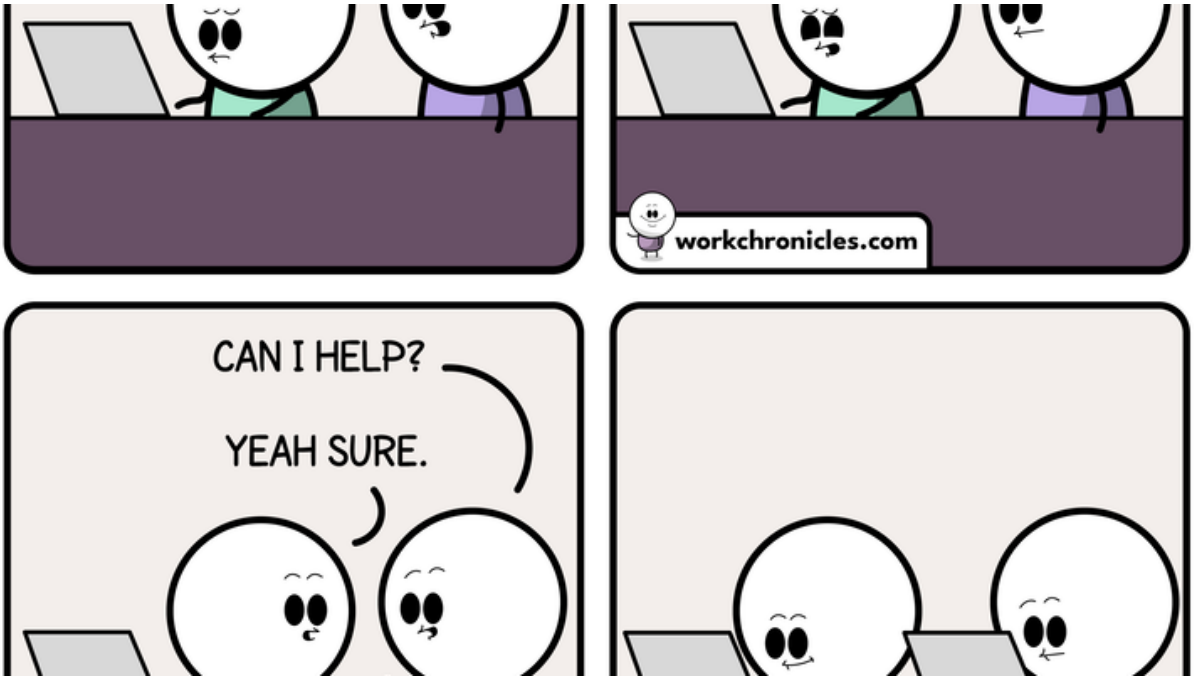
Semaphore

MLOps: From Jupyter to Production

Jupyter notebooks are great for learning and running experiments on Machine Learning. They, however, fall short when it comes to...

6 min read · Feb 1, 2024

277 2



Jacob Bennett in The Atomic Engineer

Nobody wants to work with our best engineer

Kindness is underrated.

· 2 min read · Mar 8, 2024

857 43

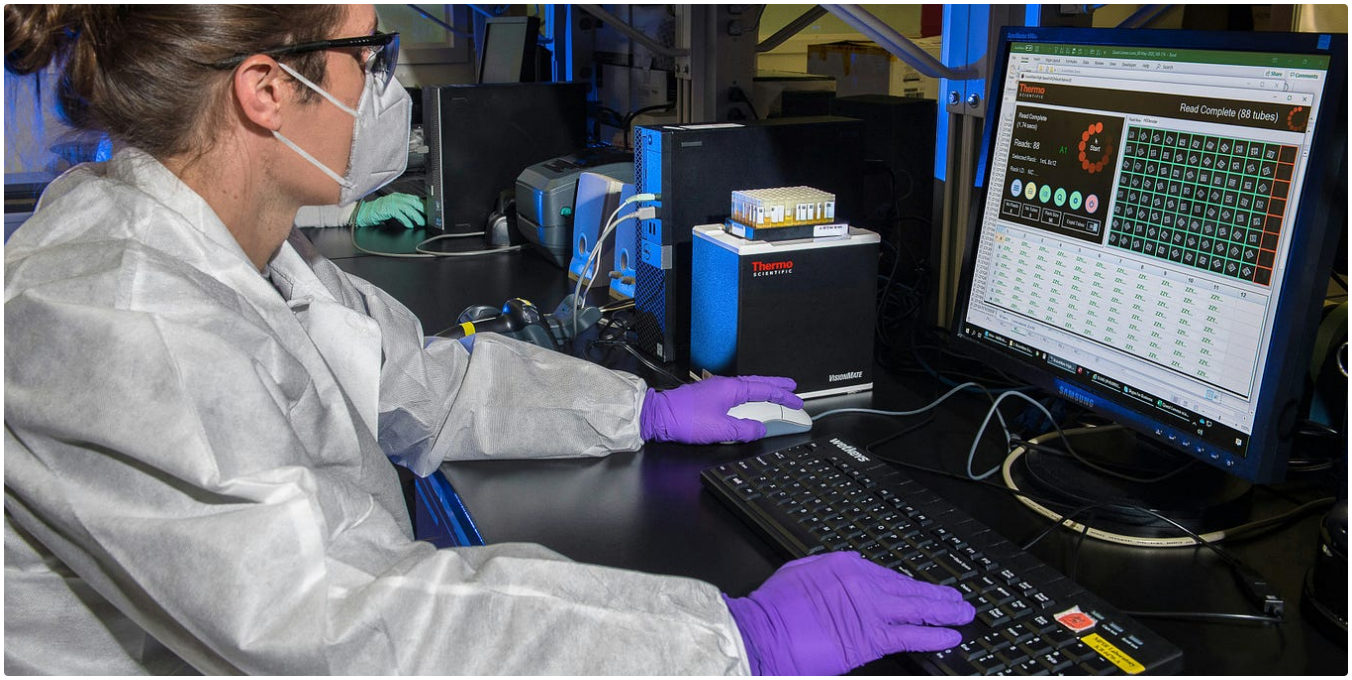
Nathan Rosidi

Data Science in 2024—What Has Changed

What has changed in the data science landscape, and what are the challenges of the 2024 data science job market?

4 min read · Jan 29, 2024

938 16



Kelvin Lu in Towards AI

Full-Stack Data Scientist?

Research of the emerging full-stack data scientists, data science engineering management, and suggestions of career development.

9 min read · Feb 10, 2024

1.4K 12

See more recommendations