IT University of Copenhagen
Big Data Management

Assignment 3: ML Lifecycle

Tracking, reproducibility and deployment using MLflow

November, 2022

1 Introduction

In this assignment you will revisit the wind power forecasting system you made in Assignment 1, and use this as a case to go through some of the steps in the data analytics lifecycle (Figure 1.1). You will do organized experiment tracking, packaging, deployment, and governance - all using MLflow.

We increase the complexity of the system you will build, asking you to create several components that interoperate and communicate over the network. We're getting you further out of the comfort-zone of your own local laptop environments. This will add some friction, but the purpose of this assignment is two-fold: (1) Familiarize you with a ML lifecycle tool, and (2) expose you to the complexity/messiness of building scalable systems (motivating the myriad of tools for this).

This assignment also requires you to use tools or systems that you might not be familiar with from previous courses: Setting up a virtual machine (VM) with a public IP address, installing/-configuring software in a linux environment, managing long-running processes, automating recurrent tasks, doing basic networking configuration, version control using git, etc. Most of these sound more complicated than they are, but it can be a daunting task if you're new to it. We will try to cover these topics as needed in exercises, but a healthy dose of self-study is probably needed.

Again, we highly recommend the student-run MIT course The Missing Semester of Your CS Education, which cover many of these aspects. For this assignment it is especially the first two lectures and the first part of lecture 5.

2 MLflow

We will use MLflow, a set of open source tools for the machine learning lifecycle. It is a fairly new project, and in rapid development.

MLflow covers three main areas: Experiment tracking, reproducibility, and model deployment.

Prior to working on this assignment, we highly recommend you go through the MLflow Tutorial in some detail.

2.1 Experiment tracking

When developing ML models, you go through a lot of experiments, trying many different combinations of models, hyper-parameters and feature extraction. This means you need to keep track of a lot of metrics, and how you created each metric. This is what MLflow Tracking is trying to solve.

In MLflow each *experiment* can consist of many *runs*, and in each run you can log *parameters*, *metrics*, *tags* and *artifacts*. To get a better explanation of this, visit the MLflow Tracking introduction

A simple experiment could look like this:

```
import mlflow
with mlflow.start_run():
    mlflow.log_param("Param one", 1)
    mlflow.log_metric("Accuracy", 0.5)
```

By default, MLflow will log your runs to the default experiment, and will save all your runs to your local file system in the folder `mlruns`. After running the example above, the directory structure could look like this:

```
mlruns/
    0/
      1f880fec49d64bffa1fdd4e7600f7c5b/
          artifacts/
          meta.yaml
          metrics/
              Accuracy
          params/
              Param one
          tags/
              mlflow.source.git.commit
              mlflow.source.name
              mlflow.source.type
              mlflow.user
      meta.yaml
```

The experiment id is `0`, the run id is `1f880fec49d64bffa1fdd4e7600f7c5b` and everything is just saved as text files. Try opening some of these files in your text editor. When running

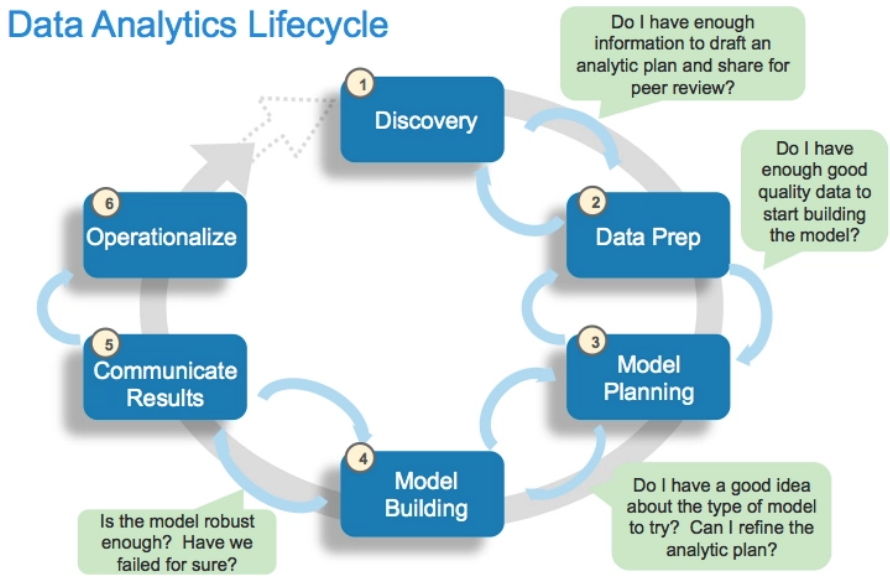

Figure 1.1: The data analytics life cycle. MLflow addresses tracking (2,3,4), reproducibility (4,5), and deployment/governance (6)

MLflow from within a git repository, the current commit is also saved as a tag, making it easier to recreate experiments.

MLflow also have (more-or-less experimental) autologging features, which you are welcome to try out.

2.1.1 Public tracking server

In addition to tracking your experiments on your local computer, you can also use a public tracking server. We have set up such a server at `http://training.itu.dk:5000/`. This tracking server stores runs in a PostgreSQL database instead of the local file system.

MLflow is not suitable as a competition framework, and is not really intended for many users. There is no authentication and results are not validated. This means that other users can delete your runs/experiments (possibly by mistake) and it is trivial to log fake metrics. Do not use the public tracking server for results that you cannot afford to lose. Take it with a grain of salt.

Alternatively, Microsoft Azure (See Azure section for how to get access) provides a similar interface to the MLFlow UI which can be used as a tracking server. To use that, you can:

1. Create Azure Machine Learning workspace
2. Go in the ML workspace you just created, download config.json life and store it locally together with your code.
3. Launch studio
4. Run your experiments locally (or on Azure VM, see below sections to set up the VM to be able to run mlflow there) and you should be able to see the results in the Experiments tab on Microsoft Azure Machine Learning Studio.

2.2 Reproducibility

MLflow Projects is a standardized way to encapsulate an experiment in a reproducible manner. This is done by specifying all the dependencies as either a `conda` or `docker` environment. Here we will only talk about the `conda` approach.

An MLflow project consists of:

- The code you want to package, including the data for your experiments
- An environment file specifying the dependencies for the code. In this case a YAML file with the conda environment.
- An `MLproject` file specifying which environment file to use, and the entry points to the code.

We've made a small project that shows an example of polynomial regression, which we will use to show how MLflow projects work: `https://github.com/NielsOerbaek/PolyRegExample`

The main experiment, in which we simulate some data[1] and model it using different degrees of polynomial regression, is defined in `experiment.py`:

```
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import Pipeline
from matplotlib import pyplot as plt
import numpy as np

import sys
num_samples = int(sys.argv[1]) if len(sys.argv) > 1 else 100

def get_ys(xs):
    signal = -0.1*xs**3 + xs**2 - 5*xs - 5
    noise = np.random.normal(0,100,(len(xs),1))
    return signal + noise
```

[1] The data is simulated using the polynomial function $f(x) = -0.1x^3 + x^2 - 5x + 5 + \epsilon$, where ϵ is Gaussian noise.

```

X = np.random.uniform(-20,20,num_samples).reshape((num_samples,1))
y = get_ys(X)

plt.scatter(X,y,label="data")

for degree in range(1,4):
    model = Pipeline([
        ("Poly", PolynomialFeatures(degree=degree)),
        ("LenReg", LinearRegression())
    ])
    model.fit(X,y)
    plotting_x = np.linspace(-20,20,num=50).reshape((50,1))
    preds = model.predict(plotting_x)
    plt.plot(plotting_x, preds, label=f"degree={degree}")

plt.legend()
plt.show()
```

The Conda environment for this experiment is specified in `PolyReg.yaml`:

```
name: PolyReg
channels:
  - defaults
dependencies:
  - scikit-learn>0.23
  - numpy>1.19
  - pip>20
  - python=3.8
  - pip:
    - mlflow
    - matplotlib>3
```

Finally, the `MLproject`-file specifies how to run the project:

```
name: PolyReg

conda_env: PolyReg.yaml

entry_points:
  main:
    parameters:
      num_samples: {type: int, default: 100}
    command: "python experiment.py {num_samples}"
```

With these three things in place, you can run the experiment using `mlflow run <path to project>`. So if the project is located on your computer, you can navigate to the directory and do:

```
mlflow run .
```

Because this project is hosted as a git repository, you can simply do:

```
mlflow run https://github.com/NielsOerbaek/PolyRegExample.git
```

This will fetch the project, resolve the environment, and run the main entry point with the default parameters.

If you want to run the experiment with 500 samples, instead of the default 100, you can do:

```
mlflow run https://github.com/NielsOerbaek/PolyRegExample.git -P num_samples=500
```

2.3 Deployment

Once you have a model you are satisfied with, you can log and deploy it using MLflow Models model format. There's detailed descriptions of the deployment options in the documentation.

As an example, have a look at this repo: https://github.com/NielsOerbaek/caiso-mlflow. This is an MLflow Project that trains a CAISO model for electricity load forecasting and saves it to the local filesystem. The saved model can then be distributed and deployed using MLflow Models.

To use the saved artifacts, we need to clone the repo to your filesystem before running (otherwise the model will just be saved in a temporary folder).

```
git clone https://github.com/NielsOerbaek/caiso-mlflow
cd caiso-mlflow
mlflow run .
mlflow models serve -m model
```

The model will now be served on your local machine. You can then query your model by opening another terminal and running:

```
curl http://127.0.0.1:5000/invocations -H 'Content-Type: application/json' -d '{
  "columns": ["Time"],
  "data": [["2021-04-15T20:00:00"]]
}'
```

You will then get an answer like `[21.49216666666666]`, meaning that your model thinks that the electricity demand in Orkney at 8 PM on Saturday the 1st of May 2021 will be 21.49 MW.

2.3.1 To the cloud!

Deploying to your own machine might seem a bit roundabout. The interesting thing comes when deploying to a server and can be queried by many users.

You can look into deploying your model to a cloud-based virtual machine with a public IP address. One such option is to use Microsoft Azure. MLflow has methods for deploying directly to Azure, but we've had mixed experiences with it. Again, have a look at the documentation.

If you already have a VM on Azure (if not, follow section Azure steps 1-2) you can ssh into it and serve your model:

1. Push your MLflow project to Github if you develop your project locally. In case you did your MLflow stuff (running experiments/training model) on Azure VM, you can skip steps 3-6.
2. ssh into your VM on Azure `ssh <username>@<Public IP address>`. You can find VM Public IP address in the Overview.
3. Get miniconda on the VM:

```
wget https://repo.anaconda.com/miniconda/Miniconda3-py39_4.10.3-Linux-x86_64.sh
bash Miniconda3-py39_4.10.3-Linux-x86_64.sh
```

4. Exit VM and ssh again so that conda enviroment is activated.
5. Get MLflow:

```
conda install -c conda-forge mlflow
```

6. Get your model from a Github repository to the VM (if you are using our example MLflow project Caiso you need to rename MLProject file to MLproject):

```
git clone https://github.com/NielsOerbaek/caiso-mlflow
cd caiso-mlflow

#rename MLproject file if it is named MLProject
mv MLProject MLproject
```

7. Open port 5000[2]. Skip Create a network security group, go to your Azure Resource group and select existing Network security group.
8. Serve the model:

[2] Port 5000 is default for MLflow

```
mlflow run .
mlflow models serve -m model -h 0.0.0.0 -p 5000
```

9. Query served model from your local machine:

```
curl http://<Public IP address>:5000/invocations -H 'Content-Type: application/json'
     -d '{
  "columns": ["Time"],
  "data": [["2020-11-14T20:00:00"]]
}'
```

   You will then get an answer like `[16.07111111111111]`, meaning that your model thinks that the electricity demand in Orkney at 8 PM on Saturday the 14th of November 2020 will be 16.07 MW.

You might also want to host a local tracking server on the same VM, and possibly a database or a S3-compatible object store. To start long-running processes that don't stop when you terminate your ssh connection, have a look at the `nohup` command. To install new services (like MinIO), you might want to use docker. Think about how these steps could be automated.

2.3.2 Azure

One option for getting a VM with a public IP is to use Microsoft Azure. Azure is enterprise-oriented and can be quite complicated, so you need to go through some steps that might feel like a bit over the top for this task. But on the plus-side, you should all be eligible for free student credits, giving you some credits to experiment with.

You can follow these guides to get you started:

1. Setup your student account
2. Setup a VM with a public IP:
   - You should pick a region a little closer to home. Like Germany or Sweden.
   - In the Image choose Ubuntu Server 20.04.
   - You should pick a cheaper image. Like "B1s", which is only 60kr/month.
   - You can choose to use your existing public key instead of generating a new pair, if you wish.
   - Create a network security group and open the ports you need[3]
     NOTE: This guide uses an old version of the Azure user interface, so you wont need the "advanced" button, but you should pick a "Custom" service with port 5000.

   After these steps you should be able to ssh into your VM and serve your model or MLflow UI.
   MLflow also has methods for deploying directly to Azure, but I've had mixed experiences with it in the past. Again, have a look at the documentation.
   If you for some reason do not have student credits, contact us and we will figure something out.

2.3.3 Trying out your deployed model

We've made a little webpage where you can try to use your deployed model in something that resemples a real usecase a little more.
On `https://orkneycloud.itu.dk/mlflow/` you can insert the url for your invocation endpoint and it will fetch the newest weather forecasts, use your model to make predictions, and draw a little graph.

[3] Port 5000 is default for MLflow

3 Data

For this assignment we have frozen the dataset to make comparisons easier. This means that the data will not be fetched from the influx database, but simply loaded from the attached json-file. The json file can be loaded into a pandas dataframe by using:

```
df = pd.read_json("path/to/file.json", orient="split")
```

The data format is the same as for Assignment 1; the only difference being that the `generation` and `wind` dataframe have been joined by an outer join. This you only need to load a single dataframe as your dataset.
The attached dataset covers 180 days of data. Below is the output of `df.info()`

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 254967 entries, 2020-05-15 12:55:00 to 2020-11-11 12:54:00
Data columns (total 7 columns):
 #   Column       Non-Null Count   Dtype
---  ------       --------------   -----
 0   ANM          254967 non-null  float64
 1   Non-ANM      254967 non-null  float64
 2   Total        254967 non-null  float64
 3   Direction    1379 non-null    object
 4   Lead_hours   1379 non-null    float64
 5   Source_time  1379 non-null    datetime64[ns]
 6   Speed        1379 non-null    float64
dtypes: datetime64[ns](1), float64(5), object(1)
memory usage: 15.6+ MB
```

4 Requirements and Hand-in

This section describes the (fairly loose) functional requirements of your system. The assignment is split into 4 parts: (1) Tracking, (2) Packaging and Reproducibility, (3) Deployment and (4) Model Governance.
We ask you (as per usual) to describe your design choices and their trade-offs (even if some of those decisions are defined by the assignment). In addition, each part contains one or more discussion points that we ask you to reflect on in your report.
The maximum length for the report handed in is five pages.

4.1 Tracking

- Redo some of your experiments from Assignment 1, tracking the hyper-parameters and metrics using MLflow tracking. *Explain what you are logging and why.*
- *Imagine that you were developing a model wind prediction models for each grid connected turbine in Denmark (There are 6-7000 of them). The turbines have independent datasets and diverse scales and conditions. How would you setup this experiment in a distributed manner using MLflow?*

4.2 Packaging and Reproducibility

- Alter your retraining pipeline so it:
  - Logs *the metrics* for the saved and newly trained model to a remote MLflow tracking server[4]
  - Saves the final model in the MLflow packaging format.
- Package your script as an MLflow project and push it to a public git repo.
- *Discuss how each file in your repo affects the reproducibility of your project.*
- *Which steps does MLflow go through in order to run your project?*

[4] You can set up your own (on the VM you will use), or use our public one: http://training.itu.dk:5000/

4.3 Deployment

- Procure a virtual machine with a public IP address[5].
- Setup miniconda and mlflow
- Setup a recurring job (e.g. using cron) that:
  - Runs your retraining project from the git repo
  - Serves the saved model[6]
- *How does the contious logging of metrics of your retraining script help to identify data drift?*
- *Assume that your system had an extreme increase in usage. How would you scale your system to handle the large number of requests?*

4.4 Model governance (technical part optional)

- Instead of saving your model to the working directory, use the model registry of MLflow to log your models and track the version numbers and production version.
  - Initialize an S3-compatible object store service on your VM[7]
  - Change your retraining script to log the model, register it and promote it to production.
  - Serve the model from the model registry.
- *How does this approach alter the scalability of your wind power forecasting service? Which parts do we need to scale out?*
- *Assume that you find your model to be producing low quality forecasts. What steps would you need to take to roll back the model to a previous version? How about if you did not use the model registry?*

[5] We recommend using Azure, see subsubsection 2.3.2. If you run out of credits on Azure, contact us and we will figure something out.

[6] You might need to kill the previous job. Have a look at `killall` (e.g. `killall -g mlflow`)

[7] One way of doing so is to deploy a MinIO service locally using docker. Read the guide here.