Machine Learning adventures with MLFlow

Building ML workflow on your local system

Image by Author

Hello There!

This article is for anyone who wants to get started with MLFlow. We will explore concepts of MLFlow, implementing a simple end-to-end ML workflow using MLFlow—from creating a model in a notebook to packaging and deploying the model. After that I will explain how this simple workflow can be translated into production level model management.

We will incorporate a few MLOps concepts in the workflow like *model training, model evaluation, packaging and model deployment.* To understand MLOps and related concepts, you can read my other article

Let's begin!

What is MLFlow?

MLFlow is an open-source end-to-end platform for managing a machine learning lifecycle provided by Databricks. There is also a Managed MLFlow version for enterprise usage.

In this article, I use the open-source version.

OK! Why do I need MLFlow?

Most data scientists and ML engineers are able to create a model on their laptops and desktops. They may use Anaconda, Jupyter or some other IDE to code their ML models. The following problems arise when they have to improve their model performance over time and when multiple members of a team are working on the same model

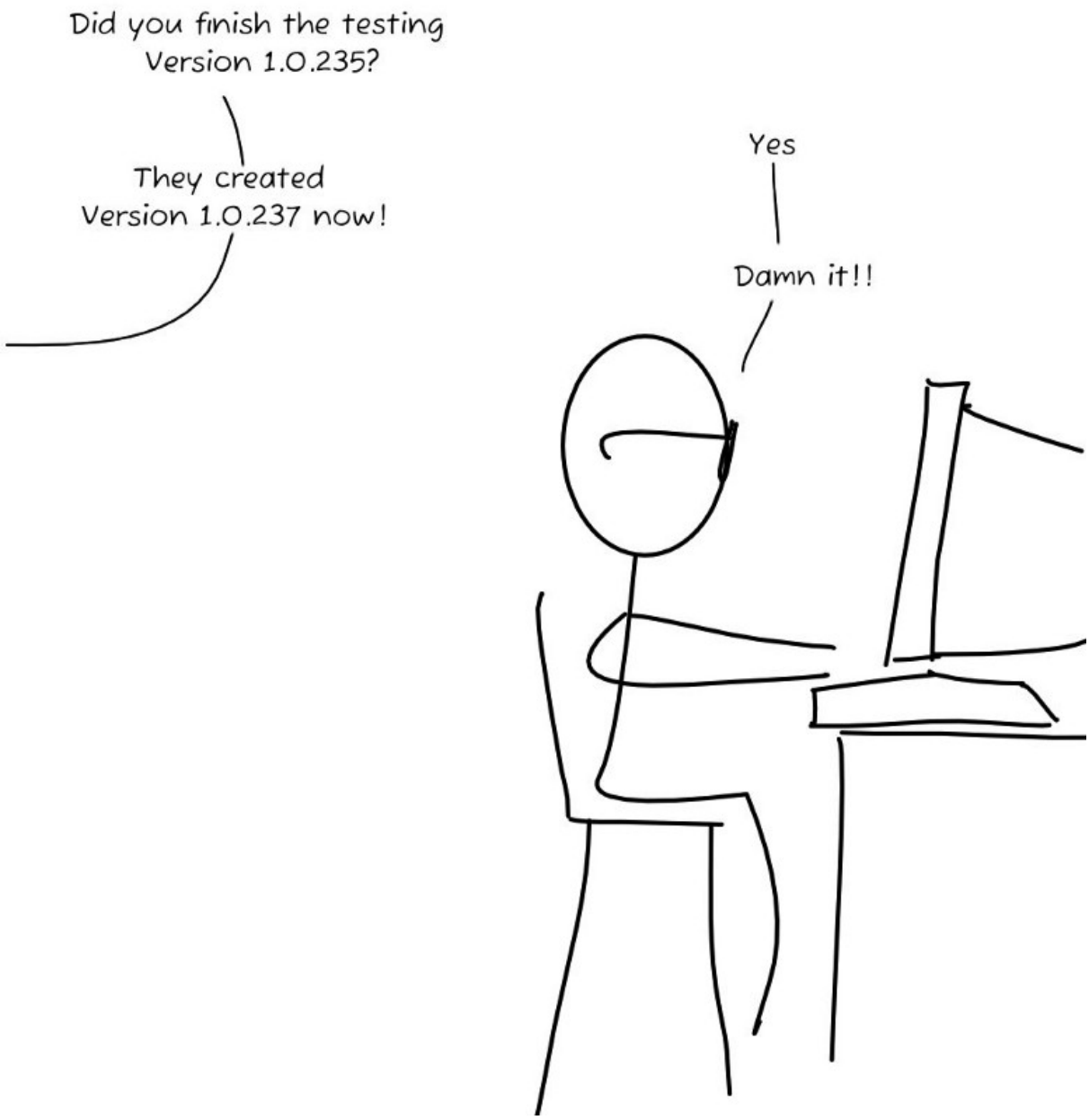
- Keep track of all the parameters tuned and tweaked in the model.
- Keep track of the outputs, accuracy and error scores.
- Maintain the record of models and their related data objects (scalers, imputers, encoders etc.)
- Version their models.
- Share the model with team members—what are the prerequisites/setup needed in place for other members to run the model on their system.
- Wrap their models with API and deploying it, will require extra coding and tech stack knowledge.

This is where the MLFlow comes into the picture, relieves the pain of learning an entirely new tech stack to maintain, track and deploy models. Provide simple APIs which you can integrate into the model code and you are on your way to deploying your model to production.

All right! What concepts do I need to know?

Before we jump headlong into the coding part, we need to know a few concepts regarding MLFlow.

Model Tracking



Did we update? (Image by Author)

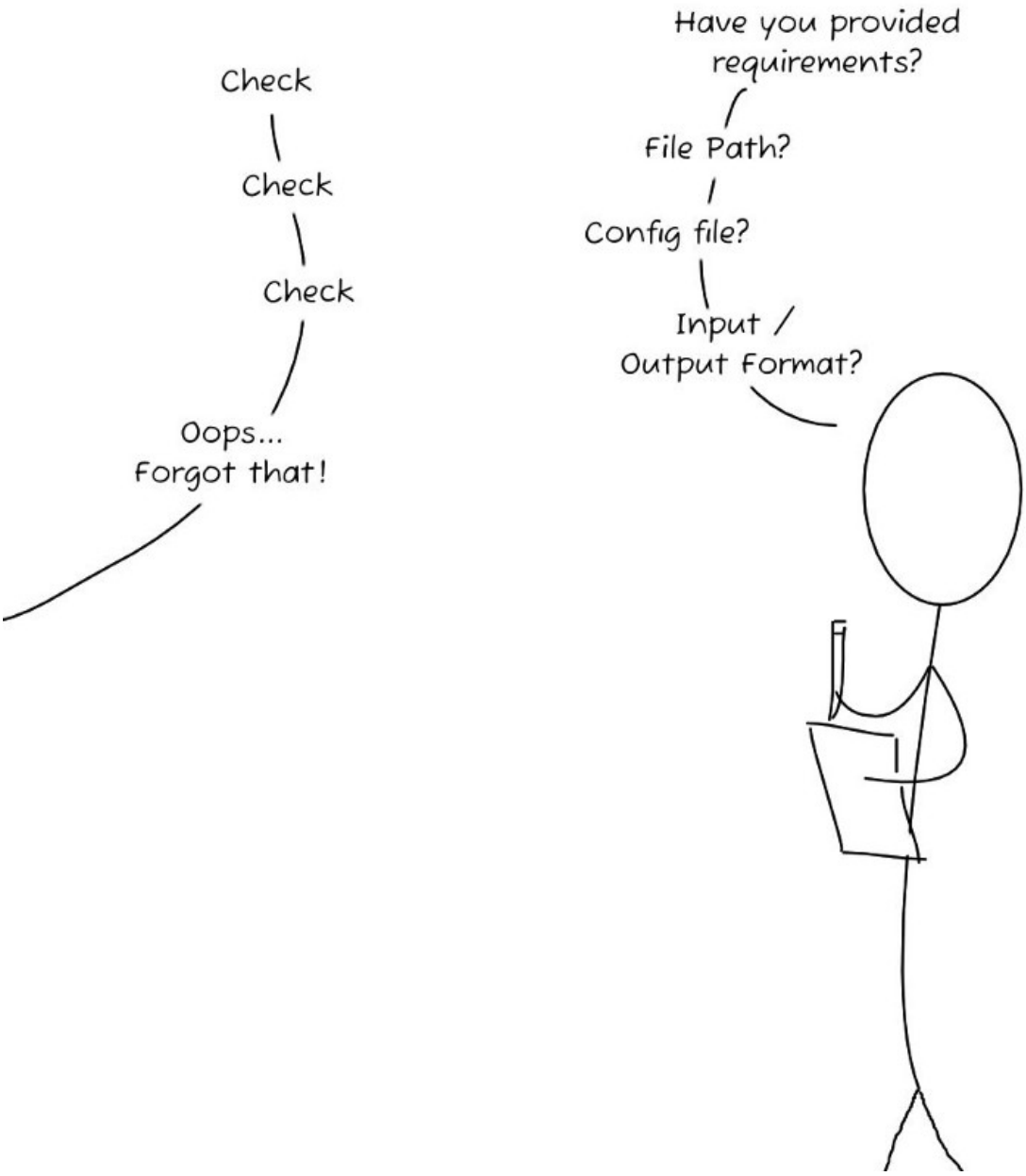
As a data scientist or ML engineer, you will spend a lot of time improving the models that you have created. It becomes tedious to keep track of what parameters you have tweaked, which particular combination gives you better performance and which error metric you need to compare. Model Tracking is an important concept to understand. It's easier to track model performance and compare the models when you have all the metrics and measures tabulated and readily available.

All this extra bookkeeping is taken care of by the MLFlow Tracking APIs.

ML Projects

All software developed generally follow a design or architecture pattern, this is not the case when it comes to machine learning. ML models do not have a specified structure or architecture pattern, each problem involves a different set of requirements and design. It becomes difficult to share the code and setup model across systems in a team or organization.

MLFlow creates a standardized pattern and structure which covers most ML use cases and makes the code easily portable. You can provide the directory structure, details of the environment where you want the model to run and the entry points for your model i.e what input your model requires. Think of it as a brief summary for an ML model, it will inform you what this model is, what it requires to run, what input you need to provide and what output you will get.



Check the manifest! (Image by Author)

ML Models

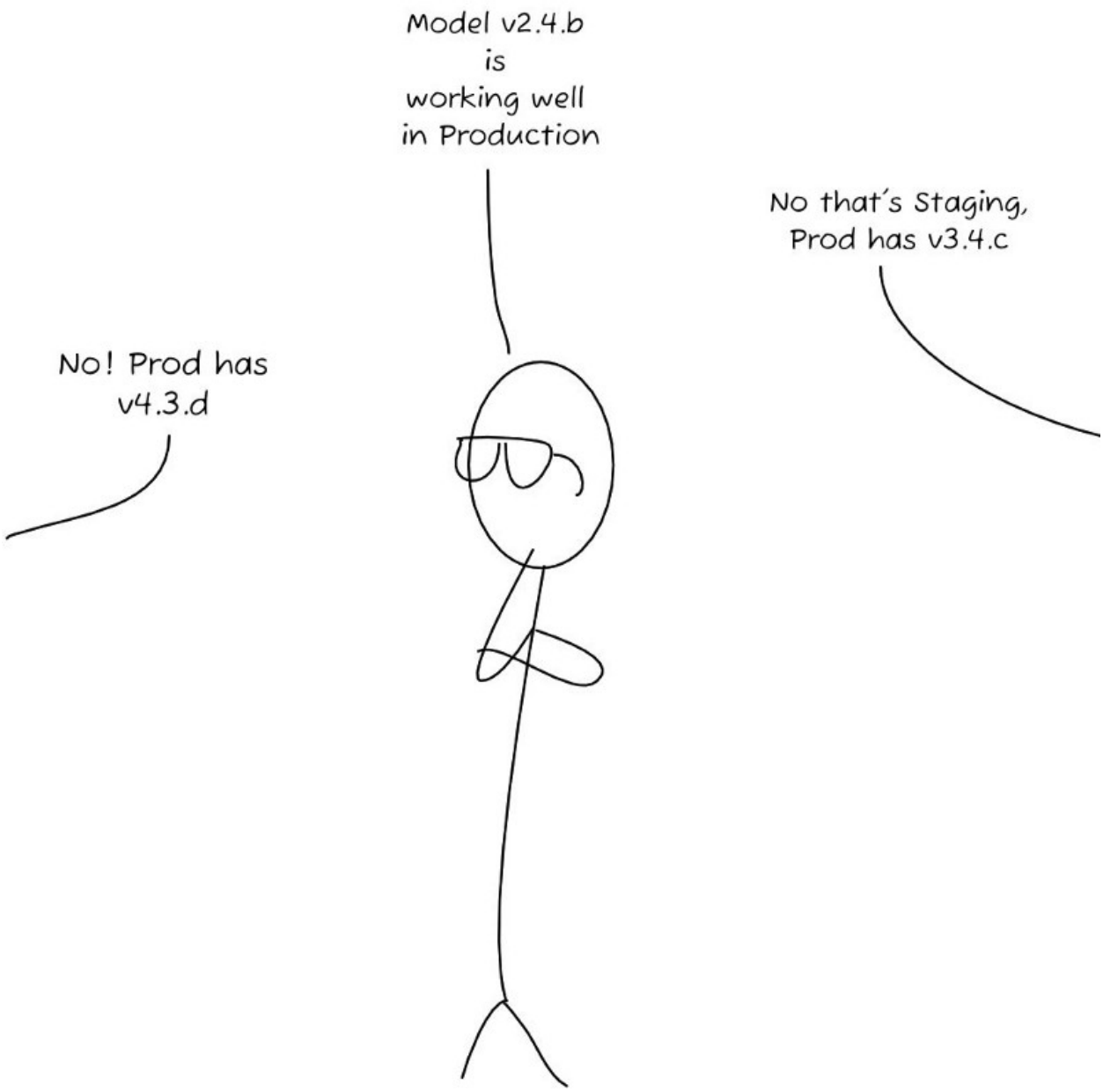
Once your model is ready, you want your model to be deployed to a cloud or an edge device. You need to package your model to deploy it. Packaging a model involves specifying what environment your model requires, what is the signature of your model *i.e what is the input your model needs and in what format should the input be provided?*

Once you have packaged the model, you can store it in a pickle format and can be deployed.

MLFlow provides various inbuilt formats for packaging the model based on the libraries or flavours. Ex: Tensorflow, Sklearn, Keras, H20, XGBoost etc.

Model Registry

As you keep improving the models with new data, you will have different versions of models with different accuracy, error level and other metrics. You might have a different model in production than you do in the development stage. MLflow Model Registry helps you keep track of which model is present in which stage. You can even archive some of the older models when not in need.



Ah! Confusion (Image by Author)

Now we have understood the concepts in MLFlow, it's time to apply them in practice. I will show you how you can introduce simple machine learning pipelines in your local system, from there we will see how it scales up to production systems.

Example—Red/Blue Classifier

We have a data set of points in space. Our job is to classify if the given point belongs to Team Red or Blue. We train a classifier to help us in the task. I have provided a full set

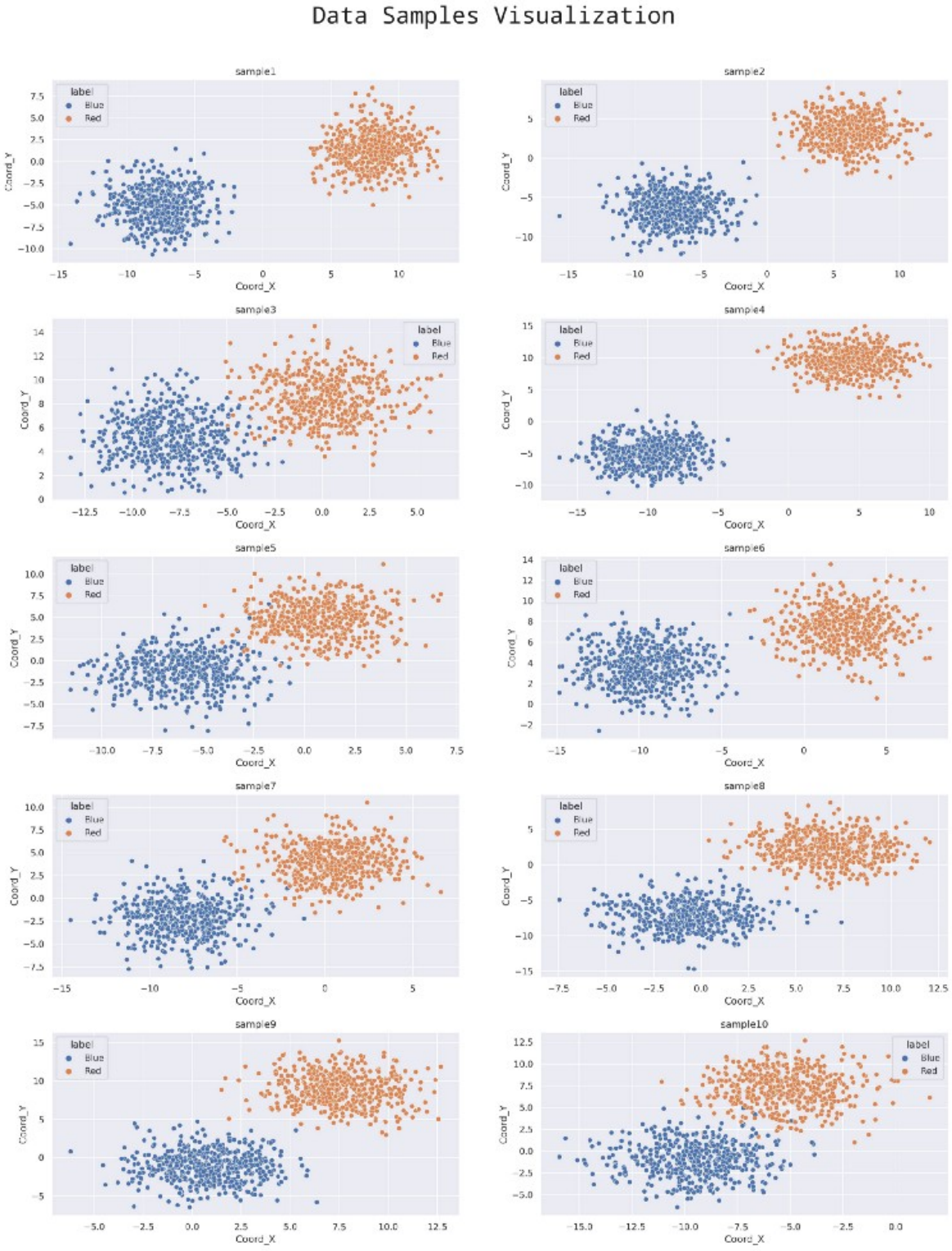
of sample data and jupyter notebook in this github-repo so that you can follow along.

> Note:

This example is created on a Linux system (Ubuntu). Should work on Windows as long as you have Python and Docker installed.

Let’s start with Data

You can see the data samples in the *data* folder in the repo. Here’s a visualization of the samples



Data Samples Visualization (Image by Author)

As you can see from the data, the red points are mostly on the right and top-right region; blue points are on the left and bottom-left region. Also you notice how the x-axis and y-axis scale keeps changing. **So we need to design a binary classifier that is capable of training with changing scales.**

Start MLFlow server locally

Before we race off to build the greatest classifier ever seen, we need to start the MLFlow server on our local system. Download the mlflow library using python

```
$ pip3 install mlflow
```

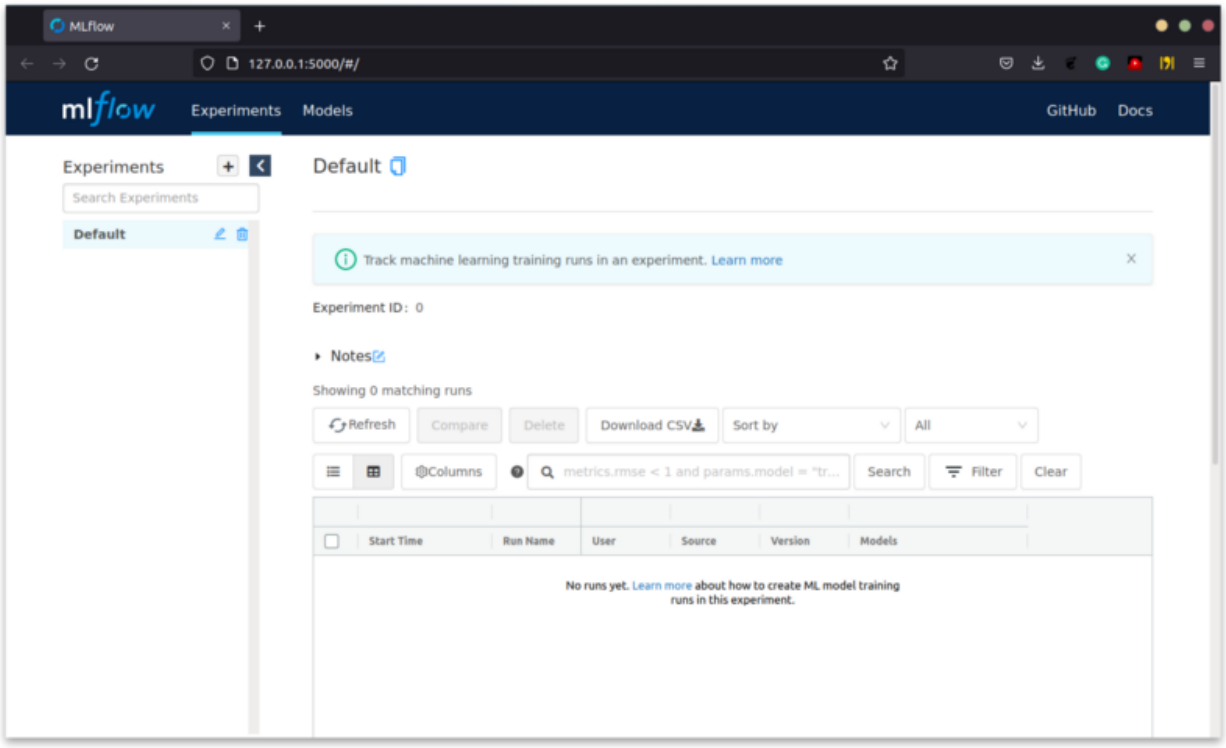
Clone the github repo that I have provided on your local system, open the folder in a terminal and run the following command to start the mlflow server

```
$ mlflow server --backend-store-uri sqlite://mlruns.db --default-artifact-root file://$PW
D/mlruns
```

This command does 3 things

- Creates a sqlite file called *mlruns.db* to record all experiments and associated data
- Creates a local *mlruns* folder to store all models and their associated data
- Starts the MLFlow server on http://127.0.0.1:5000/ on your system

You can now browse to the link in your favourite browser and see the dashboard.



MLFlow Dashboard (Image by Author)

Now we have our server up and running, we can start building the classifier.

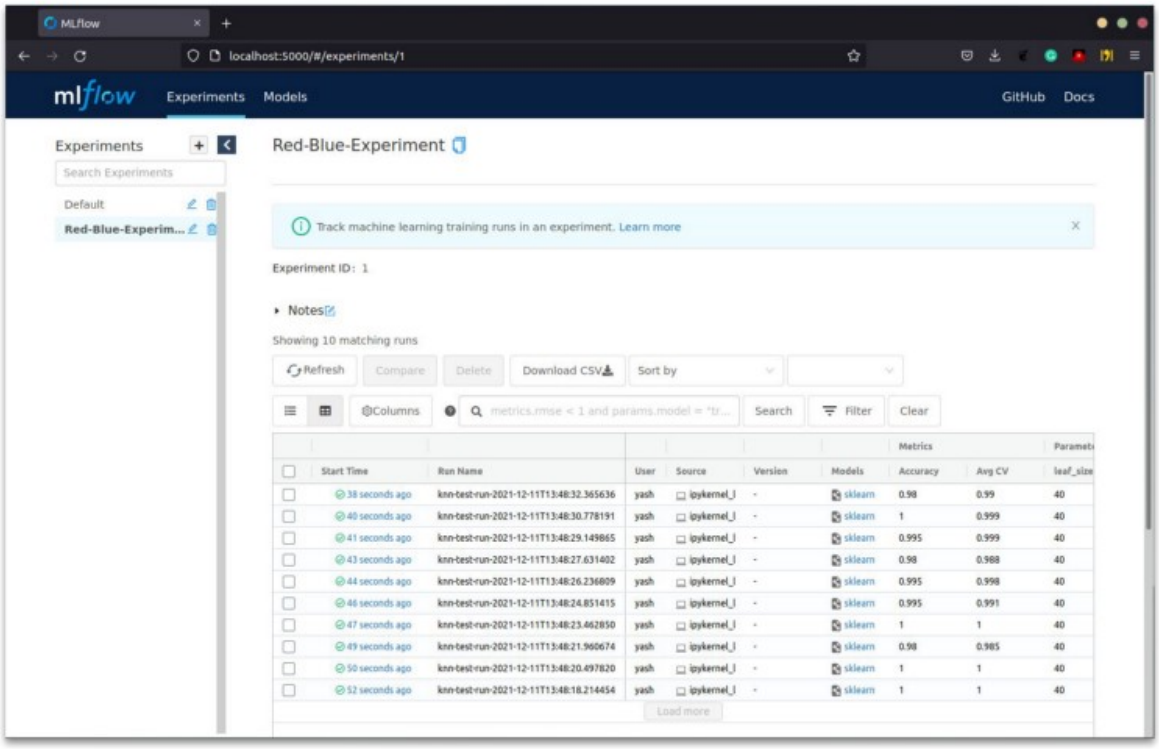
Training the Model

You can run the code cells in the Jupyter Notebook in sequential order. We have 3 functions—**transform_data, train_model and scale_data.** These functions help us to train our classifier.

We train two types of classifiers here:

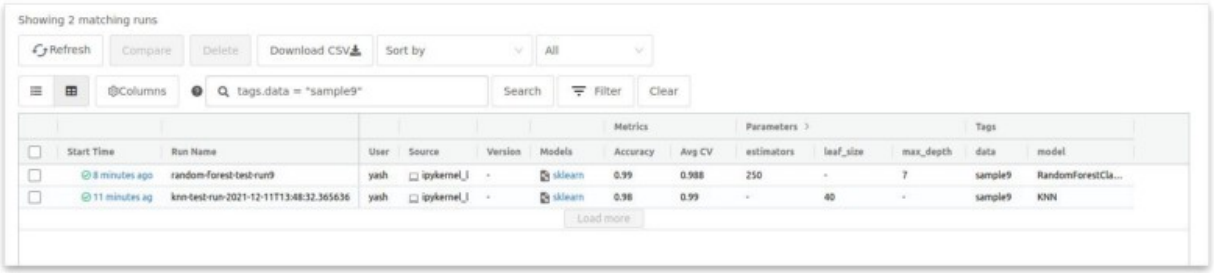
- K Nearest Neighbour Classifier
- Random Forest Classifier

You can run the cells under the **Training and Tracking** section. Once the cells are run, you will get the entries in your MLFlow Dashboard.



MLFlow Tracking (Image by Author)

Here we notice how the experiment runs are neatly tabled for us. The dashboard filter allows us to even search and filter a specific run.



MLFlow Search (Image by Author)

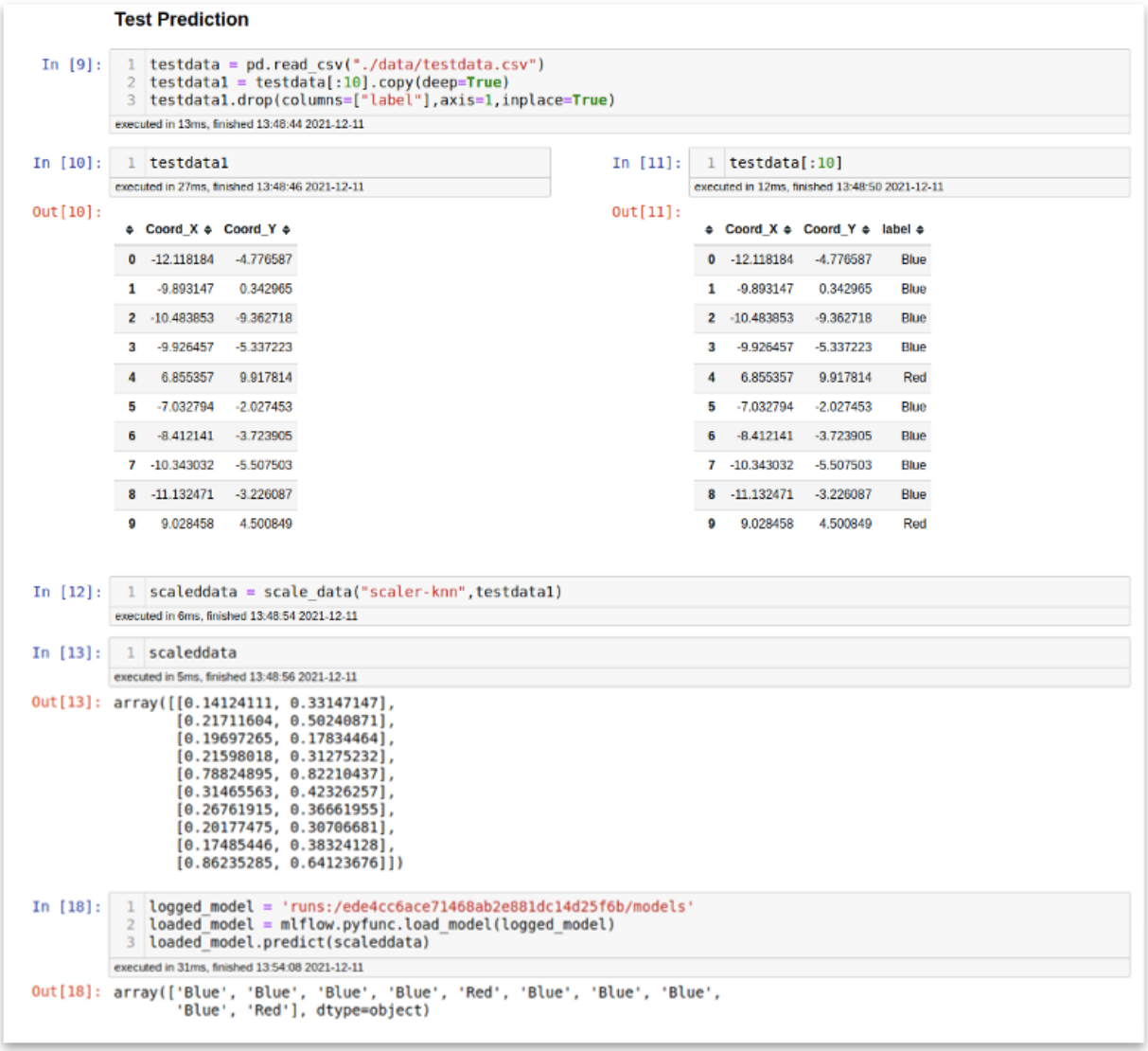
You might ask—**Why didn't we use the entire data set by merging the samples?**

We could have run the training over the entire data set, but my objective is to explain how MLFlow can be utilized in real scenarios.

> Data in the real world isn't clean, data isn't properly formatted, data isn't 100% accurate and data is never completely available.

By splitting the data into samples, we can replicate how our model will encounter new data over a period of time and train our model with new data.

Now that we have trained our model, we can test it with separate data it has never seen before (real-world scenario). Run the cells under the **Testing Predictions** section. Here we load the data, load the Scaler we saved, load a model—you can choose any model, just substitute the *runId* from one of the experiment runs you have in the dashboard.



Test Prediction

In [9]:
```
testdata = pd.read_csv("./data/testdata.csv")
testdata1 = testdata[:10].copy(deep=True)
testdata1.drop(columns=["label"],axis=1,inplace=True)
```
executed in 13ms, finished 13:48:44 2021-12-11

In [10]:
```
testdata1
```
executed in 27ms, finished 13:48:46 2021-12-11

Out[10]:

	Coord_X	Coord_Y
0	-12.118184	-4.776587
1	-9.893147	0.342965
2	-10.483853	-9.362718
3	-9.926457	-5.337223
4	6.855357	9.917814
5	-7.032794	-2.027453
6	-8.412141	-3.723905
7	-10.343032	-5.507503
8	-11.132471	-3.226087
9	9.028458	4.500849

In [11]:
```
testdata[:10]
```
executed in 12ms, finished 13:48:50 2021-12-11

Out[11]:

	Coord_X	Coord_Y	label
0	-12.118184	-4.776587	Blue
1	-9.893147	0.342965	Blue
2	-10.483853	-9.362718	Blue
3	-9.926457	-5.337223	Blue
4	6.855357	9.917814	Red
5	-7.032794	-2.027453	Blue
6	-8.412141	-3.723905	Blue
7	-10.343032	-5.507503	Blue
8	-11.132471	-3.226087	Blue
9	9.028458	4.500849	Red

In [12]:
```
scaleddata = scale_data("scaler-knn",testdata1)
```
executed in 6ms, finished 13:48:54 2021-12-11

In [13]:
```
scaleddata
```
executed in 5ms, finished 13:48:56 2021-12-11

Out[13]:
```
array([[0.14124111, 0.33147147],
       [0.21711604, 0.50240871],
       [0.19697265, 0.17834464],
       [0.21598018, 0.31275232],
       [0.78824895, 0.82210437],
       [0.31465563, 0.42326257],
       [0.26761915, 0.36661955],
       [0.20177475, 0.30706681],
       [0.17485446, 0.38324128],
       [0.86235285, 0.64123676]])
```

In [18]:
```
logged_model = 'runs:/ede4cc6ace71468ab2e881dc14d25f6b/models'
loaded_model = mlflow.pyfunc.load_model(logged_model)
loaded_model.predict(scaleddata)
```
executed in 31ms, finished 13:54:08 2021-12-11

Out[18]:
```
array(['Blue', 'Blue', 'Blue', 'Blue', 'Red', 'Blue', 'Blue', 'Blue',
       'Blue', 'Red'], dtype=object)
```

Prediction Result (Image by Author)

And behold the magic of our classifier.

Package and Deploy the model

Now that we have an awesome classifier on our hands, you want to share this with your friends and teammates. There are two ways you can deploy your model

1. MLFlow serve (no-code approach)
2. Custom wrapper for model

MLFlow Serve

You can directly run the model on your local system using the command

```
$ mlflow models serve --model-uri runs:/ede4cc6ace71468ab2e881dc14d25f6b/models --port 505
0
```

To deploy it across servers or on other systems, we need to package the model. I have written a sample Dockerfile that will help us package the model.

Dockerfile Gist

> Note: You can change the runId and port number as per your choice, both in the command and Dockerfile

Build the docker image using

```
$ docker build -t my-awesome-model .
```

You can the run the image using

```
$ docker run -d -p 5050:5000 my-awesome-model:latest
```

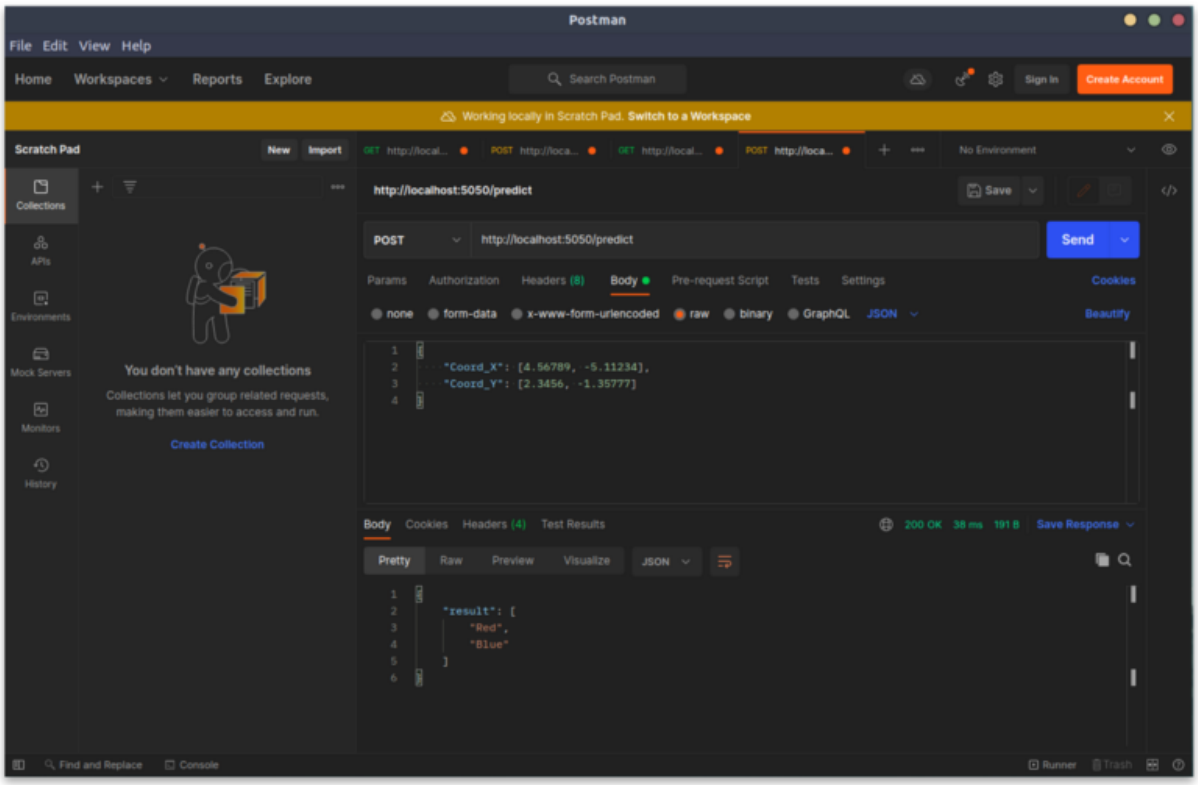
You can also publish it to Dockerhub or a docker registry of your choice.

Custom Wrapper

You have to understand that the model you deployed with *mlflow serve* assumes that you have given the input which is already scaled, this might not always be the case. Sometimes we need some extra APIs in our model, say `/health` API or `/visitors` API to check how many times the model was used.

To include extra configurations we code custom wrappers, usually using Flask or FastAPI for this. You can find the example in the `app_wrapper.py` file and Dockerfile in the repo.

You can test the model using the Postman app, send the POST request as shown `/invocations` if using the mlflow serve and `/predict` if using the custom wrapper.

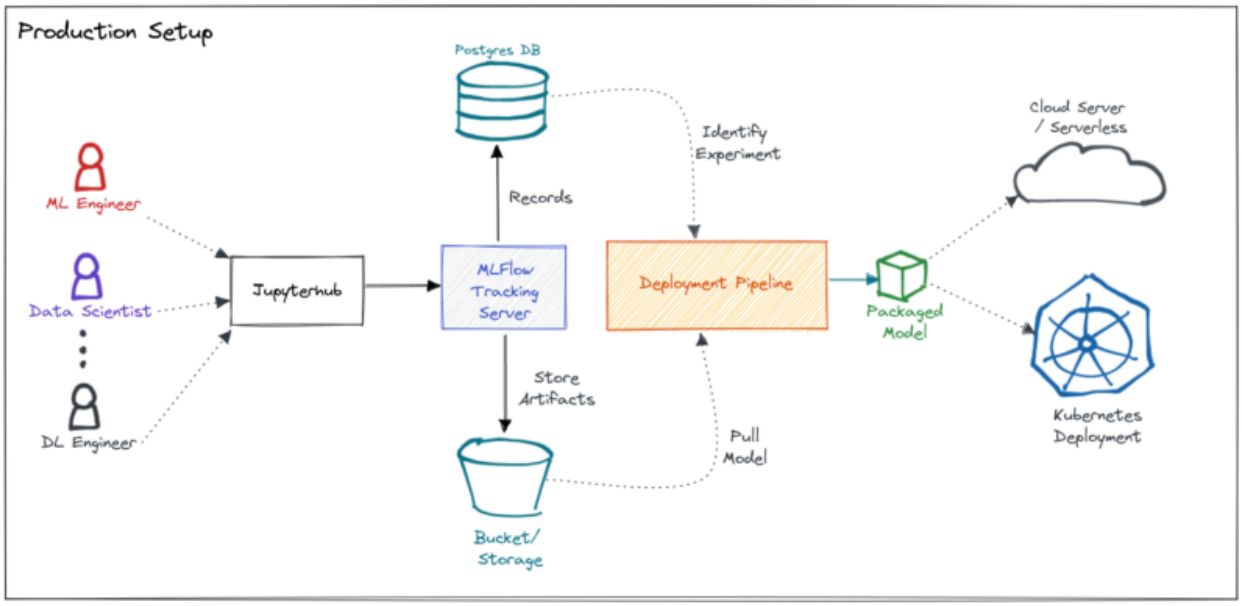


API Test in Postman (Image by Author)

We imported the data, built a classifier, trained it on multiple samples of data and deployed it locally. Congratulations! You have created your first ML pipeline using MLFlow. 👋

Production Workflow

Now that you have grasped how the workflow is implemented in your local system, we will learn how the workflow is implemented in production.



Sample Production Setup (Image by Author)

In the production setup, we generally include a JupyterHub server with user access management for coding the models. This JupyterHub server is connected to the MLFlow Tracking server for logging experiments. The experiment data is recorded on Postgres DB, this is a powerful database with high reliability and performance.

The models generated are stored on an S3 bucket or a remote FTP server.

Once the models and experiment records are logged, ML Pipeline will be triggered for deployment. A CI/CD server like Jenkins or GoCD can be used, this pipeline does the work of wrapping, building and packaging the model as we saw above. When the model is packed and ready, it is deployed to a desired public or private cloud.

Production Tips

Here are some tips for teams managing large scale model development workflow

- Use the **tags for identifying the models, easier for cataloguing and searching** as the experiment list grows.
- **Log all the metrics that are relevant for the model**, this helps us to trace model lineage and evaluate model performance over time.

- Add tags for the data used in training, helps to figure out how data changes over time. Use GitLFS or DVC for tagging and version control.
- Create new a experiment for each different problem set, **do not cram all experiment runs in one**, make sure the experiment names are unique.
- If a model is good enough to be tested and deployed, register the model in MLFlow, this helps us track the model in different stages of its life.
- Version your models, simple versioning style could be 3-dot versioning. i.e `awesome-model:1.2.3` would be `model-target.performance.pipeline/data` **model-target** would be a major change if the model is completely revamped or target data is completely changed, **performance** would be if the model has improved in performance over time and the **last number** is for either different data version or pipeline build number.
- **Try to keep a clear and consistent input signature and format**, if a model needs input as dataframe or tensor, specify it. As most requests sent are in JSON format, MLFlow does convert it implicitly, but if you are writing a custom wrapper for the model, the developer needs to know about the data or scale conversion.
- MLFlow can deploy model directly to Sagemaker and Azure, makes sure you select the right kind of instance for deployment, **do not select the default options blindly** (i.e using a c5 large instance on AWS for a model of size 30Mb would be a total wastage of computing power).
- If using a wrapper for model deployment, **make sure to use production ready wrappers like FastAPI**. If Flask / Django is being used add, Standalone WSGI Containers.
- Use a S3 bucket or S3-based storage (Minio) for storing the models. This helps in maintaining and transferring the models across deployed instances.
- Access restrictions can be added at S3 storage and Jupyterhub when using open source MLFlow as of now. Managed MLFlow offer authentication and ACL rules with notebook integrations, for enterprise usage.

These observations are from my personal experience in tracking and managing models. Hope it helps you out as well. 😄

Tracking, managing and optimizing models is a slow process. Keep experimenting with MLFlow, over the course of time, you will get better at model management.

That’s all for now. Until next time! Take care.

Bye! 👋

References

MLFlow—https://www.mlflow.org/docs/latest/index.html

KNN Classifier—https://scikit-learn.org/stable/modules/neighbors.html#classification

Example Github Repo—https://github.com/yashaswi-nayak/mlflow-adventure

All the illustrations/artwork in the article are created by me using Krita.