Session 01 - Introduction to MLOps

- Due No due date
- Points 0

Welcome to EMLO4

Pre-Requisites

- One of the main pre-requisite of this course is you should know how to work with deep learning networks in PyTorch (training and inferencing), as our world will revolve around that.
- Also, it's good to have experience working with Linux systems, but if not you'll definitely learn a lot on the way.
- It is assumed you are comfortable with Git and GitHub.
- You must finish the capstone project to be eligible for the course certification.
- You need to maintain a 70% aggregate score to be eligible for taking part in the CAPSTONE project

Here is the complete Syllabus:

Session	Name	Description
Session 1	Introduction to MLOps	An overview of MLOps (Machine Learning Operations), covering the best practices and tools to manage, deploy, and maintain machine learning models in production.
Session 2	Docker - I	A hands-on session on creating Docker containers from scratch and an introduction to Docker, the containerization platform, and its core concepts.
Session 3	Docker - II	An introduction to Docker Compose, a tool for defining and running multi-container Docker applications, with a focus on

Session	Name	Description
		deploying machine learning applications.
Session 4	PyTorch Lightning - I	An overview of PyTorch Lightning, a PyTorch wrapper for high-performance training and deployment of deep learning models, and a project setup session using PyTorch Lightning.
Session 5	PyTorch Lightning - II	Learn to build sophisticated ML projects effortlessly using PyTorch Lightning and Hydra, combining streamlined development with advanced functionality for seamless model creation and deployment.
Session 6	Data Version Control (DVC)	Data Version Control (DVC), a tool for managing machine learning data and models, including versioning, data and model management, and collaboration features.
Session 7	Experiment Tracking & Hyperparameter Optimization	A session covering various experiment tracking tools such as Tensorboard, MLFlow and an overview of Hyperparameter Optimization techniques using Optuna and Bayesian Optimization.
Session 8	AWS Crash Course	A session on AWS, covering EC2, S3, ECS, ECR, and Fargate, with a focus on deploying machine learning models on AWS.
Session 9	Model Deployment w/ FastAPI	A hands-on session on deploying machine learning models using FastAPI, a modern, fast, web framework for building APIs.
Session 10	Model Deployment for Demos	Gradio, an open-source platform for creating and sharing der of machine learning models, and a session on Model Tracing.
Session 11	Model Deployment on Serverless	An overview of Serverless deployment of machine learning models, including an introduction to AWS Lambda
Session 12	Model Deployment w/ TorchServe	An introduction to TorchServe, a PyTorch model serving library, and a hands-on session on deploying machine learning models using TorchServe.
Session 13	Kubernetes - I	This session provides an introduction to Kubernetes, a popular container orchestration platform, and its key concepts and components.
Session 14	Kubernetes - II	In this session, participants will learn how to monitor and configure Kubernetes clusters for machine learning workloads.
Session 15	Kubernetes - III	This session will cover introduction to EKS, Kubernetes Service on AWS, Deploying a FastAPI - PyTorch Kuberentes Service on EKS
Session 16	Kubernetes - IV	This session covers EBS Volumes, ISTIO and KServe, learning to deploy pytorch models on KServe
Session 17	Canary Deployment & Monitoring	This session covers how to deploy models with Canary Rollout Strategy while monitoring it on Prometheus and Grafana
Session 18	Capstone	This session is a final project where participants will apply the knowledge gained throughout the course to develop and deploy

Session	Name	Description

an end-to-end MLOps pipeline.

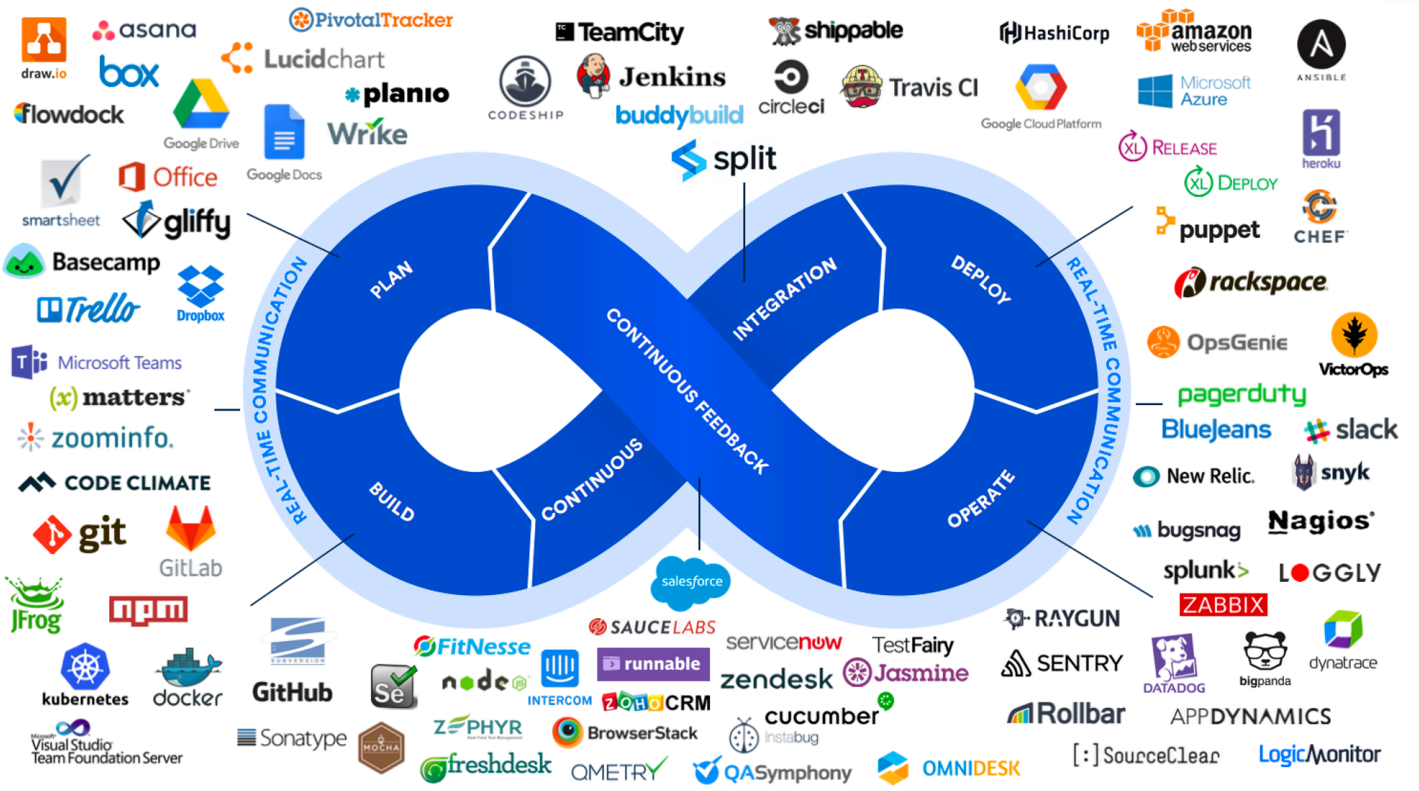
Introduction to MLOps

In the world of traditional software development, a set of practices known as DevOps have made it possible to ship software to production in minutes and keep it running reliably. DevOps relies on tools, automation, and workflows to abstract away the accidental complexity and lets developers focus on the actual problems that need to be solved. Companies and developers are already adept at it, but then why is it difficult to simply keep doing the same things for ML?

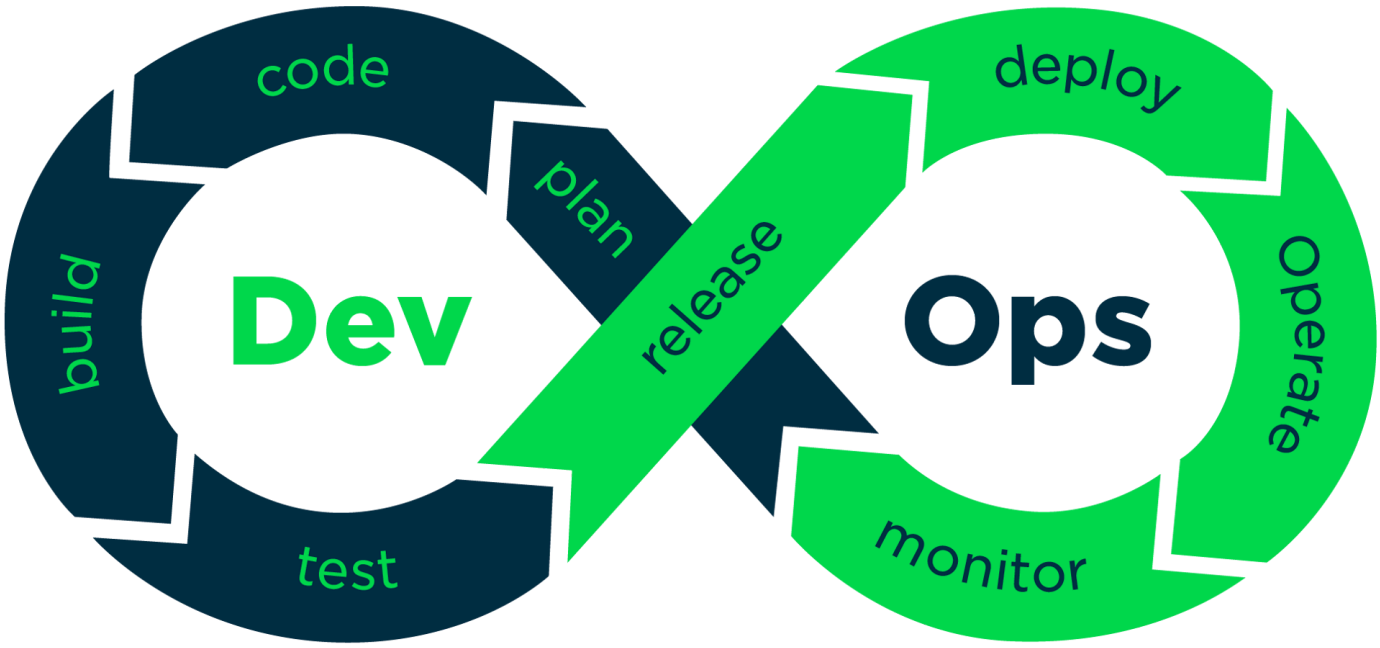
The root cause is that there's a fundamental difference between ML and traditional software: **ML is not just code, it's code plus data**.

Models don’t make it into production, and if they do, they break because they fail to adapt to changes in the environment.

DevOps and MLOps

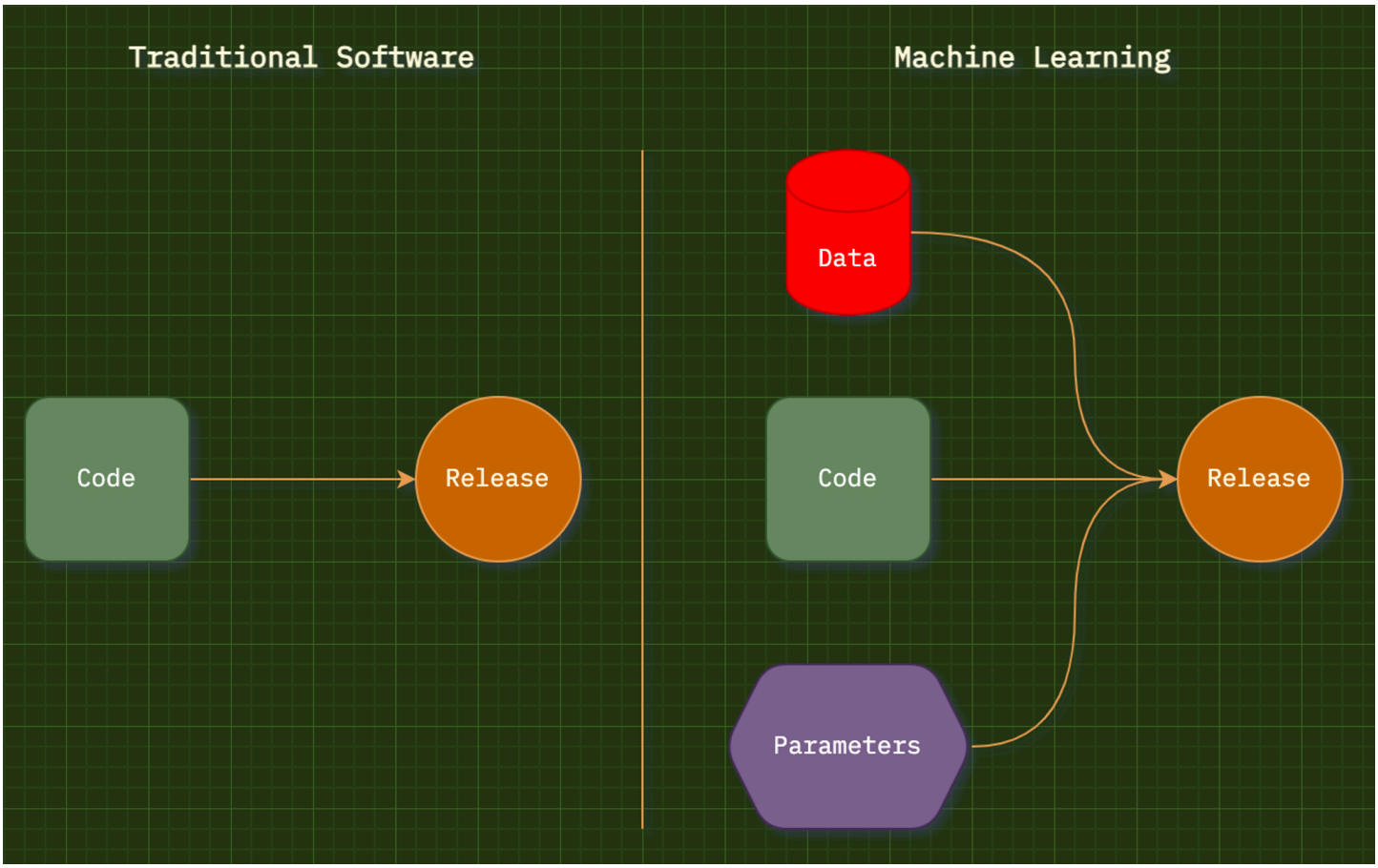


DevOps and MLOps: A Panic-less Introduction

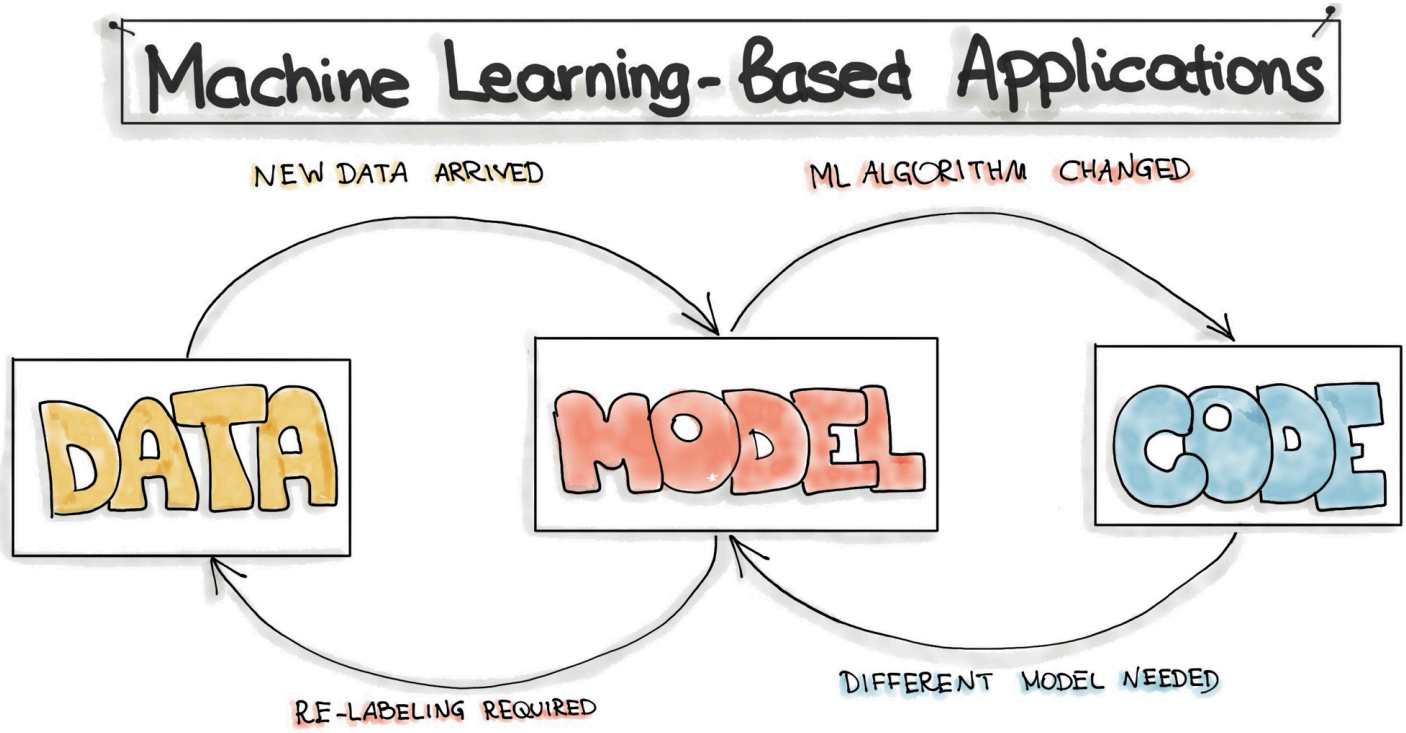


DevOps, or development operations, refers to streamlining the process of development, testing, and operational aspect of software development. With DevOps once your code is checked in, tests ru code is integrated automatically and code is deployed automatically.

What about MLOps? you are still doing your code push, the code has data preprocessing, model building, and training. but after that things look different. The automation process doesn't compile your code (well most of the DL frameworks dominantly prefer scripted languages), instead, it starts fetching the data, preprocessing it, and training a model on it. And once that model is trained it spits out some metrics like mAP, F1 Score, Precision, Recall, etc. And if this metric is better than the previous model, it is then deployed.



MLOps provides a set of standardized processes and technology capabilities for building, deploying and operationalizing ML Systems *rapidly* and *reliably*.



It builds on the concepts of DevOps and adds in the missing pieces, that will make managing the data and models along with the code. It's simply not enough to just version our code, we need to also version the models, data, and parameters along with it.

Example with Vision and Language

1. Collaboration: In a vision model example, data scientists collaborate with engineers and domain experts to develop an image recognition system for self-driving cars. They work together to define the model requirements, discuss labeling strategies, and identify potential challenges specific to autonomous vehicles.

   In a language model example, data scientists collaborate with linguists and subject matter experts to build a sentiment analysis model for customer feedback. They collaborate to define the sentiment categories, curate labeled datasets, and ensure the model captures nuances in language.

2. Automation: For the vision model, data preprocessing tasks such as resizing images, normalizing pixel values, and applying data augmentation techniques are automated using scripts or libraries. This ensures efficient and consistent data preparation without manual intervention.

   For the language model, text preprocessing tasks like tokenization, removing stop words, and stemming are automated using natural language processing libraries. This enables consistent and scalable text data processing for the sentiment analysis model.

3. Version Control: The code for the vision model, including data preprocessing, model architecture, and training, is stored in a version control system like Git. This allows tracking changes, collaborating with team members, and reverting to previous versions if needed.

   Similarly, the code for the language model, including text preprocessing, feature extraction, and model training, is also stored in a version control system. This ensures transparency and reproducibility of experiments and facilitates collaboration among team members.

4. Continuous Integration and Deployment (CI/CD): For the vision model, a CI/CD pipeline is established to automatically integrate code changes, run model training and evaluation, and deploy updated versions of the model. This enables frequent model updates, rapid experimentation, and streamlined deployment in production environments.

   For the language model, a similar CI/CD pipeline is set up to automate code integration, feature extraction, model training, and deployment. This ensures seamless model updates and efficient deployment for sentiment analysis tasks.

5. Monitoring and Governance: The deployed vision model is monitored for its performance, including metrics like accuracy, precision, and recall. Data quality is assessed, and any concept drift or distributional changes in input images are detected. Regular monitoring ensures that the model remains effective and reliable over time.

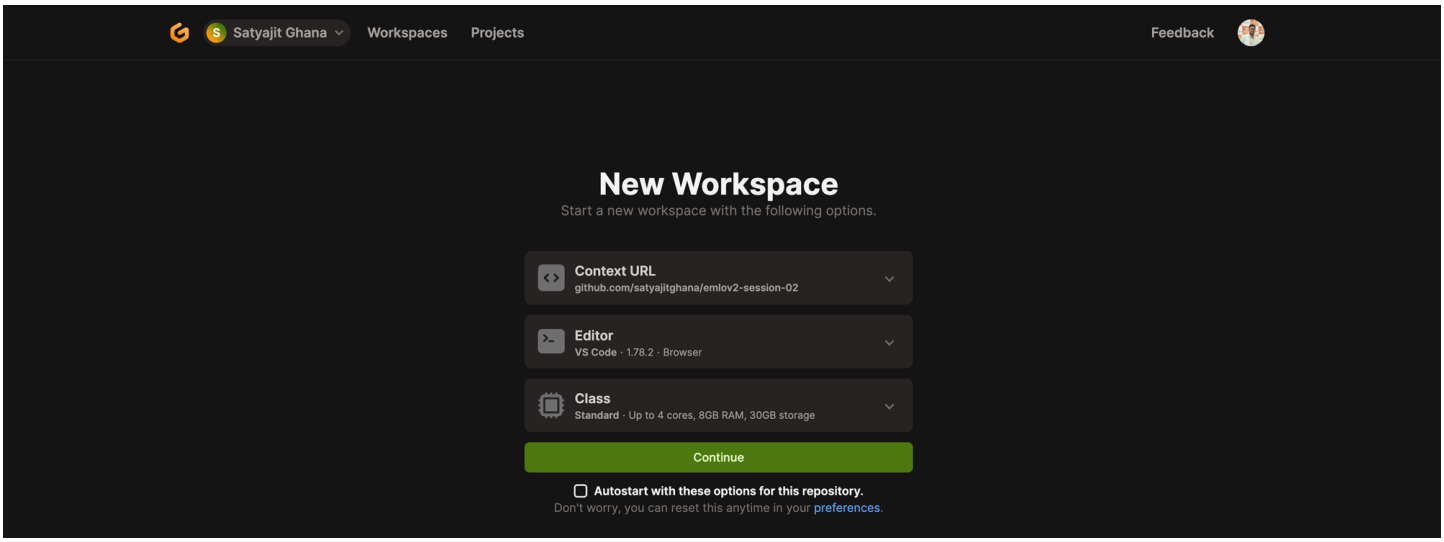
Similarly, the deployed language model for sentiment analysis is monitored to track performance metrics such as precision, recall, and F1 score. Monitoring also includes checking for data biases and ethical considerations related to sentiment analysis in various contexts.

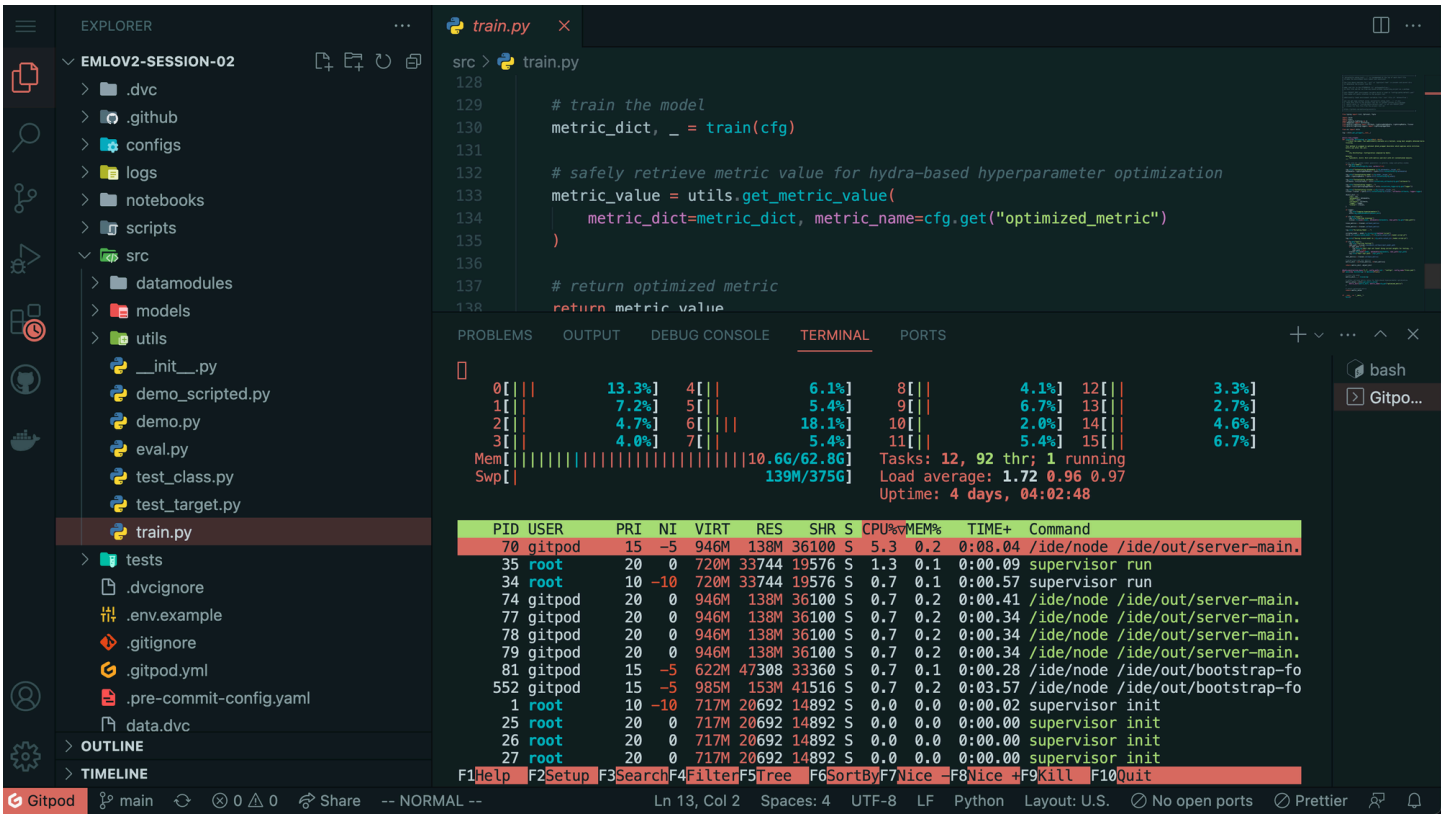
6. Reproducibility and Replicability: For the vision model, all code, model architectures, hyperparameters, and dataset versions used for training are documented and saved. This allows for easy replication of experiments, comparisons between different approaches, and sharing results with colleagues or the wider community.

   For the language model, the code, feature extraction techniques, model architectures, and hyperparameters used for sentiment analysis are documented and saved. This enables other researchers to reproduce the experiments and verify the results using the same settings and data.

In both examples, the core principles of MLOps are applied to ensure effective collaboration, automation, version control, continuous integration and deployment, monitoring, and reproducibility. These principles help in the development, management, and deployment of vision and language deep learning models in a systematic and efficient manner.

GitPod





GitHub Actions

https://github.com/features/actions (https://github.com/features/actions)

GitHub Classroom Assignment Flow

You're ready to go!

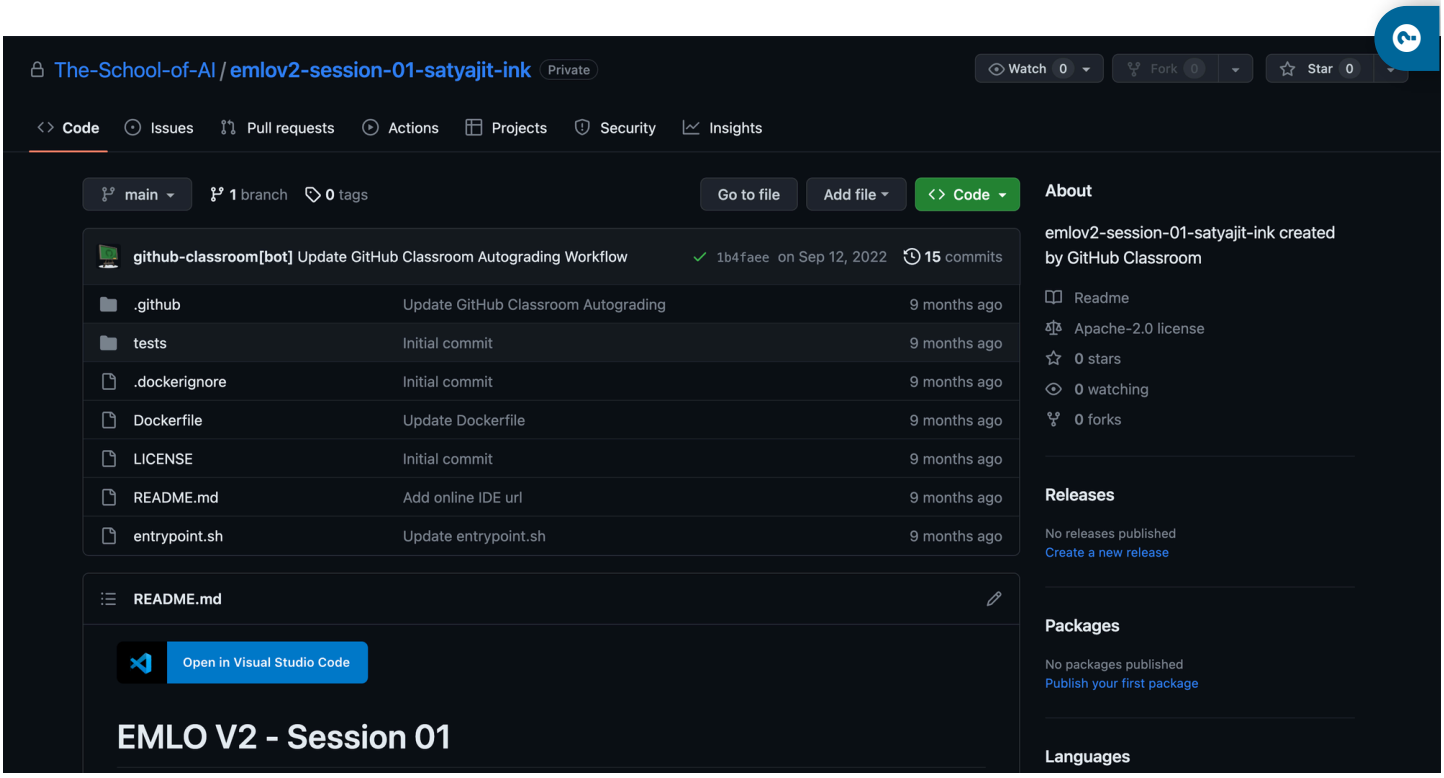
You accepted the assignment, **EMLOV2 Session 01**.

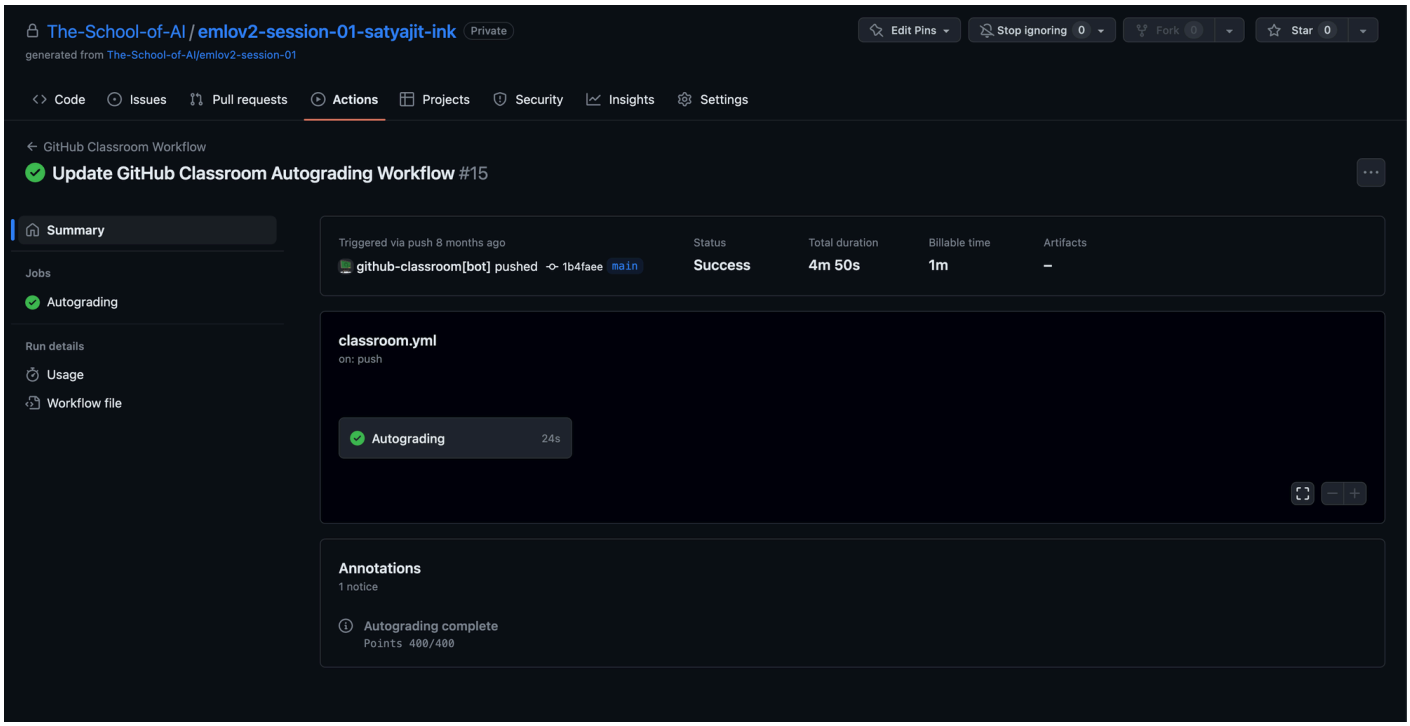
Your assignment repository has been created:

https://github.com/The-School-of-AI/emlov2-session-01-satyajit-ink

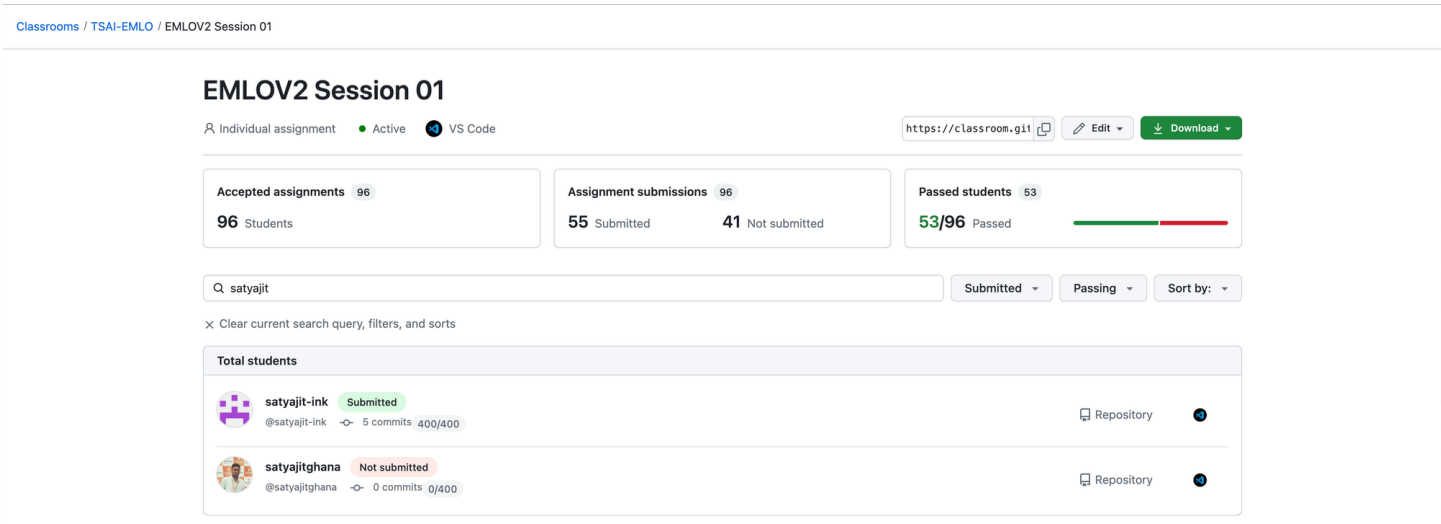
We've configured the repository associated with this assignment (update).

Open in Visual Studio Code





What I see



Basics of Linux

SSH (Secure Shell)

SSH is a protocol used to securely log onto remote systems. It's used most often in the command-line interface for Unix-like systems such as Linux. SSH provides a secure channel over an unsecured network in a client-server architecture. For instance, if you're at a computer in New York and need to manage a server located in Tokyo, you could use SSH to log into the server and run commands as if you were there in person.

Example:

```
ssh username@remotehost.com
```

The above command logs you into `remotehost.com` with the username `username`.

VIM

Vim is a highly configurable text editor built to enable efficient text editing. It's an improved version of the vi editor distributed with most UNIX systems. It has many commands and is known for its keyboard-centered operation, which lets users perform tasks efficiently without the use of a mouse.

Example:

```
vim filename.txt
```

This command opens the file `filename.txt` in Vim. If the file does not exist, it will be created.

htop

htop is an interactive system-monitoring tool for Unix-based systems. It provides a real-time, dynamic view of the processes running on a system. htop is much more interactive than the standard top command; it allows for scrolling vertically and horizontally, so you can see all processes and complete command lines.

Example:

```
htop
```

This command will run htop, displaying a real-time overview of system processes.

ps

The `ps` command in Unix and Linux is used to display information about active processes. It provides a snapshot of the current processes along with detailed information like user ID, CPU usage, memory usage, command name, etc.

Example:

```
ps -aux
```

The above command displays all processes from all users in a detailed format.

scp (Secure Copy)

The `scp` command is used to securely copy files and directories between two locations. Like SSH, scp uses the SSH protocol for data transfer.

Example:

```
scp /path/to/local/file username@remotehost:/path/to/remote/directory
```

The above command copies a local file to a remote directory.

rsync (Remote Sync)

`rsync` is a fast and versatile tool used for copying and synchronizing files both locally and remotely. rsync is great for maintaining backups and mirroring data because it only transfers the difference between source and destination.

Example:

```
rsync -avz /path/to/source/directory/ username@remotehost:/path/to/destination/directory/
```

This command syncs the source directory to the destination directory. The `-avz` option stands for archive mode, verbose and compress data during the transfer.

tmux (Terminal Multiplexer)

tmux is a terminal multiplexer. It allows you to switch easily between several programs in one terminal, detach them (they keep running in the background) and reattach them to a different terminal.

Example:

```
tmux new -s train
```

This command starts a new tmux session. Within this session, you can create new windows (`Ctrl+b c`), switch between windows (`Ctrl+b n` for the next window), split windows into panes, and more.

serve.py

```
from fastapi import FastAPI
from fastapi.responses import StreamingResponse
from fastapi.middleware.cors import CORSMiddleware
from pydantic import BaseModel
from typing import List, Generator
from transformers import (
    AutoModelForCausalLM,
    AutoTokenizer,
```

```
    TextIteratorStreamer,
    pipeline,
)
from threading import Thread
import torch

app = FastAPI()

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

class Message(BaseModel):
    role: str
    content: str


class Messages(BaseModel):
    messages: List[Message]


device = "cuda" if torch.cuda.is_available() else "cpu"
model = AutoModelForCausalLM.from_pretrained(
    "microsoft/Phi-3.5-mini-instruct",
    device_map="auto",
    torch_dtype=torch.float16 if device == "cuda" else torch.float32,
    trust_remote_code=True,
)

tokenizer = AutoTokenizer.from_pretrained("microsoft/Phi-3.5-mini-instruct")

pipe = pipeline(
    "text-generation",
    model=model,
    tokenizer=tokenizer,
)

def text_generator(streamer: TextIteratorStreamer) -> Generator[str, None, None]:
    for new_text in streamer:
        yield new_text


@app.post("/chat")
```

```
async def generate_text(messages: Messages):
    message_list = [{"role": msg.role, "content": msg.content} for msg in messages.messages]

    streamer = TextIteratorStreamer(
        tokenizer=tokenizer,
        skip_prompt=True,
        decode_kwargs={"skip_special_tokens": True},
    )

    generation_args = {
        "max_new_tokens": 500,
        "return_full_text": False,
        "temperature": 0.7,
        "do_sample": True,
        "streamer": streamer,
    }

    thread = Thread(target=pipe, kwargs={"text_inputs": message_list, **generation_args})
    thread.start()

    return StreamingResponse(text_generator(streamer), media_type="text/plain")


if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

Recordings

Studio

EMLO4 - Session 01 - Studio

Google Meet

EMLO4 - Session 1 - Google Meet