Department of Artificial Intelligence and Data Science

LABORATORY MANUAL

OF

MLOPS

FOR

T.Y. Semester – V

Name of Student:- .....

Class:-..... DIV:-.....

Roll No:-..... PRN No:-.....

SEM:- ODD Academic Year:- .....

Vision of the Department Developing AI & DS applications that addresses regional challenges and empowering innovation and challenges, to become a leading rural AI & DS hub.	**Mission of the Department** **M1:** Emphasizing responsible AI development that considers social and environmental factors specific to the region. **M2:** Educating the public about the potential benefits and responsible use of AI & DS technologies. **M3:** Facilitating internships and research projects with AI & DS companies to bridge the gap between theory and practice.

GENERAL LABORATORY INSTRUCTIONS

1. Students are advised to come to the laboratory at least 5 minutes before (to the starting time), those who come after 5 minutes will not be allowed into the lab.
2. Plan your task properly much before to the commencement, come prepared to the lab withthe synopsis / program / experiment details.
3. Student should enter into the laboratory with:
    a. Laboratory observation notes with all the details (Problem statement, Aim, Algorithm, Procedure, Program, Expected Output, etc.,) filled in for the lab session.
    b. Laboratory Record updated up to the last session experiments and other utensils (if any) needed in the lab.
    c. Proper Dress code and Identity card.
4. Sign in the laboratory login register, write the TIME-IN, and occupy the computer system allotted to you by the faculty.
5. Execute your task in the laboratory, and record the results / output in the MS WORD format, and get certified by the concerned faculty.
6. All the students should be polite and cooperative with the laboratory staff, must maintain the discipline and decency in the laboratory.
7. Computer labs are established with sophisticated and high-end branded systems, which should be utilized properly.
8. Students / Faculty must keep their mobile phones in SWITCHED OFF mode during the lab sessions. Misuse of the equipment, misbehaviors with the staff and systems etc., will attract severe punishment.

Vision of the Department Developing AI & DS applications that addresses regional challenges and empowering innovation and challenges, to become a leading rural AI & DS hub.	**Mission of the Department** **M1:** Emphasizing responsible AI development that considers social and environmental factors specific to the region. **M2:** Educating the public about the potential benefits and responsible use of AI & DS technologies. **M3:** Facilitating internships and research projects with AI & DS companies to bridge the gap between theory and practice.

9. Students must take the permission of the faculty in case of any urgency to go out; if anybody found loitering outside the lab / class without permission during working hours will be treated seriously and punished appropriately.
10. Students should LOG OFF/ SHUT DOWN the computer system before he/she leaves the lab after completing the task (experiment) in all aspects. He/she must ensure the system / seat is kept properly.

Vision of the Department Developing AI & DS applications that addresses regional challenges and empowering innovation and challenges, to become a leading rural AI & DS hub.	**Mission of the Department** **M1:** Emphasizing responsible AI development that considers social and environmental factors specific to the region. **M2:** Educating the public about the potential benefits and responsible use of AI & DS technologies. **M3:** Facilitating internships and research projects with AI & DS companies to bridge the gap between theory and practice.

INDEX

Expt. No.	Title of Experiment	Date of Performance	Date of Submission	Sign of Faculty
1	Introduction to MLOps: comparison of Software Development, DevOps and MLOps, MLOps lifecycle, key components, challenges in production ML systems, survey of MLOps tools, and preparation of workflow diagram and report.			
2	Git installation and configuration, repository creation, branching and merging, managing ML project source code, collaboration using GitHub/GitLab, version history and merge conflict resolution.			
3	Installation and configuration of DVC, dataset tracking and version management, storage of data artifacts, comparison of dataset versions, integration of DVC with Git, and reproducibility of ML experiments.			
4	Data collection, preprocessing, feature engineering, data validation, pipeline design, automation of the data processing workflow, and execution of an end-to-end data pipeline.			
5	Introduction to Feature Stores: feature repository creation, feature definition and registration, data ingestion, retrieval of online/offline features, and analysis of feature-store benefits in MLOps.			
6	Installation and configuration of MLflow: training ML models, logging parameters/metrics, artifact tracking, experiment comparison, model registration, and selection of the best model.			
7	Development of a baseline ML model, hyperparameter selection using Grid Search and Random Search, performance evaluation, and			

Expt. No.	Title of Experiment	Date of Performance	Date of Submission	Sign of Faculty
	experiment tracking with MLflow.			
8	Unit testing of data processing functions, testing of training pipelines, validation of model outputs, creation of automated test cases, and execution of tests using CI tools.			

This is to certify that ..... and Roll No. ..... of T.Y. AI & DS has completed all the experiment in the subject MLOps (Machine Learning Operations Lab) during academic Year 2026-27.

Sign of Student **Sign of Faculty** **HOD- AI & DS**

0. Windows Command Prompt / PowerShell Basics

Before starting with Git and MLOps tools, it is useful to know basic Windows navigation commands (used in Command Prompt / PowerShell), since all installations below are demonstrated for Windows.

Command / Line of Code	Meaning / Explanation
cd	Prints the current working directory (folder you are in).
cd foldername	Change directory INTO a sub-folder named 'foldername'.
cd ..	Move UP one directory level (go to parent folder).
cd \	Go to the root of the current drive (e.g., C:\).
dir	List all files and folders in the current directory (like 'ls' in Linux).
mkdir project1	Create ('make directory') a new folder named 'project1'.
rmdir project1	Remove an empty folder named 'project1'.
del file.txt	Delete a file named file.txt.
copy a.txt b.txt	Copy a.txt to a new file b.txt.
move a.txt D:\backup	Move a.txt to another folder/drive.
cls	Clear the terminal screen.
echo %cd%	Print the current directory path (CMD).
where python	Show the installation path of the python executable.
python --version	Show the installed Python version.
exit	Close the terminal window.

1. Assignment 1 (Environment Setup) — Reference

Although not part of assignments 2-8, every subsequent assignment assumes the following base setup on Windows:

```
REM Install Python from python.org (tick 'Add Python to PATH')
python –version
```

```
REM Create and activate a virtual environment
python -m venv mlops_env

Activates the environment
mlops_env\Scripts\activate

REM Upgrade pip and install common packages
python -m pip install --upgrade pip
pip install numpy pandas scikit-learn matplotlib jupyter
```

Command / Line of Code	**Meaning / Explanation**
python -m venv mlops_env	Creates a new isolated Python virtual environment in a folder called 'mlops_env'.
mlops_env\Scripts\activate	Activates the virtual environment on Windows so installed packages don't affect the global Python.
python -m pip install --upgrade pip	Upgrades pip (Python's package manager) to the latest version inside the venv.
pip install numpy pandas scikit-learn matplotlib jupyter	Installs the core data-science libraries needed for the rest of the assignments.

Date of Performance: …… **Date of Submission:** ……

Assignment 1: Introduction to MLOps

Objective: To study the fundamentals of MLOps, compare Software Development, DevOps and MLOps, understand the MLOps lifecycle and its key components, identify challenges in production Machine Learning systems, survey commonly used MLOps tools and technologies, and prepare an MLOps workflow diagram and report.

Course Outcomes (COs) Addressed:

- Understand the fundamental concepts, lifecycle, and components of MLOps, and differentiate it from traditional Software Development and DevOps practices.

Software Required:

- Web browser and internet access
- MS Word
- Reference documentation of MLOps tools (Git, DVC, MLflow, Docker, Kubernetes, Airflow, Feast)

1.1 Introduction to MLOps

MLOps (Machine Learning Operations) is a set of practices that combines Machine Learning, DevOps and Data Engineering to reliably and efficiently deploy and maintain ML models in production. While a data scientist can build an accurate model in a notebook, taking that model to production and keeping it performing well over time requires versioning of data, code and models, automated testing, continuous integration/deployment, and continuous monitoring — the practices that MLOps standardizes.

1.2 Software Development vs DevOps vs MLOps

Traditional software development focuses on writing and shipping deterministic code. DevOps extends this with automation, CI/CD and infrastructure practices to deliver software faster and more reliably. MLOps builds further on DevOps by adding the extra dimensions unique to ML systems: data, and the trained model itself, both of which change over time and must be versioned, tested and monitored alongside the code.

Aspect	Software Development	DevOps	MLOps
Primary Artifact	Source code	Source code + infrastructure	Code + data + model
Versioning	Code (Git)	Code + config (Git, IaC)	Code + data (DVC) + model (MLflow)
Testing	Unit / integration tests	Automated build & deployment tests	Data validation, model validation, unit

Aspect	Software Development	DevOps	MLOps
			tests
Deployment Trigger	Manual / CI on code change	CI/CD on code change	CI/CD on code, data or model change
Performance Drift	Not applicable	Not applicable	Data drift / model decay over time
Monitoring Focus	Application logs	System health, uptime	Model accuracy, data quality, prediction drift

1.3 MLOps Lifecycle

The MLOps lifecycle is iterative and cyclical rather than a one-time linear process, since models must be continuously retrained and redeployed as new data becomes available.

- Data Collection & Versioning - gathering raw data and tracking it with tools such as DVC.
- Data Preparation & Feature Engineering - cleaning, transforming, and generating features, often via a Feature Store.
- Model Development & Training - experimenting with algorithms and hyperparameters, tracked using MLflow.
- Model Evaluation & Validation - testing model performance against validation/test data and business metrics.
- Model Packaging & Registration - versioning the trained model in a model registry.
- CI/CD - automated build, test, and deployment pipelines for code, data and models.
- Deployment & Serving - exposing the model via a REST API or batch job, often containerized with Docker.
- Monitoring & Retraining - tracking live performance, detecting data/model drift, and triggering retraining.

1.4 Key Components of MLOps

- Version Control - Git for code, DVC for data and datasets.
- Experiment Tracking - MLflow, Weights & Biases for logging parameters, metrics and artifacts.
- Feature Store - Feast, Tecton for consistent, reusable features across training and serving.
- Pipeline Orchestration - Airflow, Kubeflow Pipelines, Prefect for automating multi-stage workflows.
- CI/CD - GitHub Actions, GitLab CI, Jenkins for automated build, test and deployment.

- Containerization & Orchestration - Docker for packaging, Kubernetes for scaling and managing deployments.
- Model Serving - Flask, FastAPI, TensorFlow Serving, TorchServe for exposing models as APIs.
- Monitoring - Prometheus, Grafana, Evidently AI for tracking data drift and model performance in production.

1.5 Challenges in Production Machine Learning Systems

- Data drift and concept drift - the statistical properties of incoming data change over time, degrading model accuracy.
- Reproducibility - difficulty in recreating exact results without properly versioned data, code and environment.
- Technical debt - ML systems accumulate hidden dependencies between data, features, and models ("CACE": Changing Anything Changes Everything).
- Scalability - serving predictions at low latency for a large number of concurrent requests.
- Model monitoring - detecting silent failures, since a model can keep returning predictions while its accuracy silently degrades.
- Collaboration - coordinating between data scientists, ML engineers and DevOps teams who use different tools and workflows.
- Governance & compliance - maintaining audit trails, explainability, and fairness in regulated domains.

1.6 Survey of MLOps Tools and Technologies

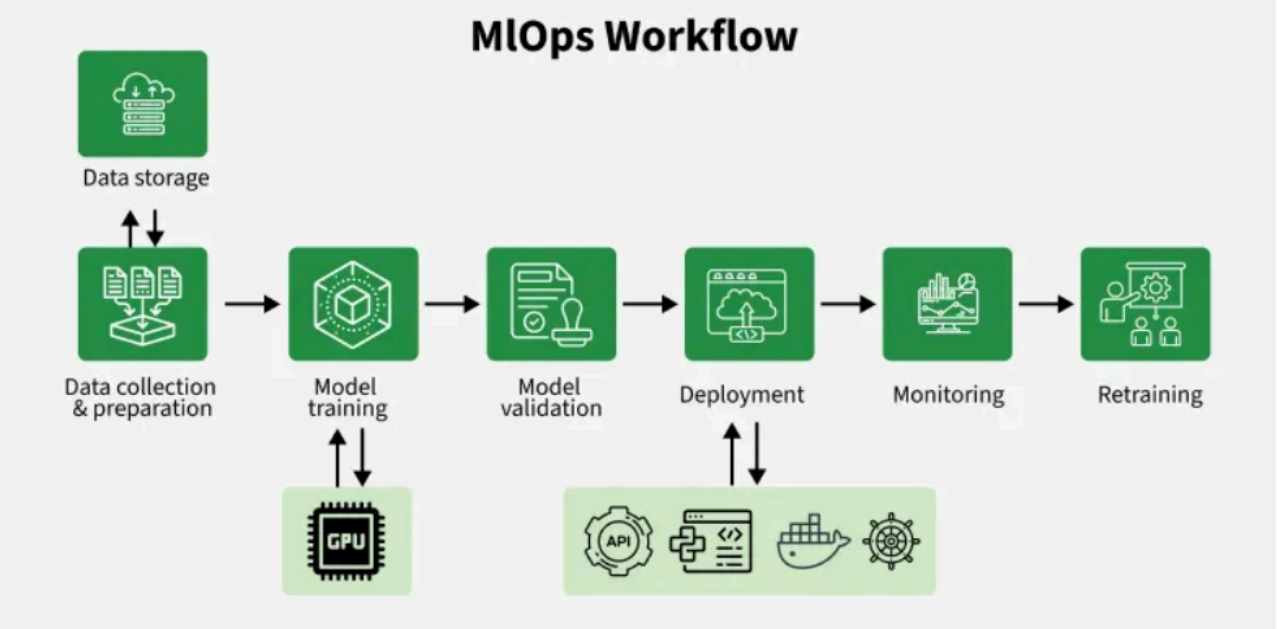
Stage	Representative Tools
Version Control	Git, GitHub, GitLab, Bitbucket
Data Versioning	DVC, LakeFS, Delta Lake
Feature Store	Feast, Tecton, Hopsworks
Experiment Tracking	MLflow, Weights & Biases, Neptune.ai
Pipeline Orchestration	Apache Airflow, Kubeflow Pipelines, Prefect, ZenML
CI/CD	GitHub Actions, GitLab CI/CD, Jenkins, CircleCI
Containerization	Docker, Podman
Container Orchestration	Kubernetes, Docker Swarm
Model Serving	Flask, FastAPI, TensorFlow Serving, TorchServe, Seldon Core

Stage	Representative Tools
Monitoring	Prometheus, Grafana, Evidently AI, WhyLabs

7. MLOps Workflow Diagram

The block diagram below represents a typical end-to-end MLOps workflow, showing how the components interact in a continuous loop from data to deployment and back to retraining.

MLOps Workflow Diagram :



1.7 Stages of MLOps Workflow:

1. Data Collection & Preprocessing

- Gather structured and unstructured data from multiple sources.
- Clean, normalize and transform data to ensure quality for training.
- Manage data versioning for reproducibility.

2. Model Development

- Build ML models using supervised, unsupervised or reinforcement learning.
- Experiment with algorithms, architectures and hyperparameters.
- Track experiments for reproducibility using tools like MLflow or Weights & Biases.

3. Model Training & Validation

- Train models on preprocessed datasets.
- Validate performance using metrics such as accuracy, F1-score or RMSE.
- Address overfitting/underfitting using techniques like cross-validation.

4. Model Deployment

- Deploy models to production using cloud, on-premise or edge infrastructure.
- Use CI/CD pipelines for seamless integration.
- Ensure containerization in Docker or Kubernetes for portability.

5. Monitoring & Maintenance

- Continuously monitor model performance, latency and accuracy.
- Detect data drift or concept drift and trigger retraining if needed.

Conclusion:

In this practical, we studied the fundamentals of MLOps, compared it with traditional Software Development and DevOps, explored the MLOps lifecycle and its key components, identified common challenges faced in production ML systems, surveyed widely used MLOps tools, and prepared a workflow diagram summarizing the complete MLOps pipeline.

Date of Performance: …… **Date of Submission:** ……

Assignment 2: Git — Installation, Configuration, Branching, Merging, Collaboration

2.1 Objective

To install and configure Git on Windows, create a repository, manage ML project source code using branches, collaborate using GitHub, and resolve merge conflicts.

2.2 Installation of Git on Windows

- Download the Git installer from https://git-scm.com/download/win
- Run the .exe installer, keep default options (Git Bash, Git GUI, adding Git to PATH) and click Install.
- Verify installation by opening Command Prompt / Git Bash.

```
git --version
```

Command / Line of Code	Meaning / Explanation
git --version	Displays the installed Git version, confirming Git was installed correctly.

2.3 Initial Configuration

```
git config --global user.name "Yajurv"
git config --global user.email "yajurv@example.com"
git config --global core.editor "notepad"
git config --list
```

Command / Line of Code	Meaning / Explanation
git config --global user.name "Yajurv"	Sets the name that will be attached to every commit you make, globally (for all repos on this machine).
git config --global user.email "yajurv@example.com"	Sets the email attached to commits — should match your GitHub account email.
git config --global core.editor "notepad"	Sets Notepad as the default text editor Git opens for commit messages on Windows.
git config --list	Prints all current Git configuration settings (name, email, editor, etc.).

2.4 Repository Creation

```
mkdir ml-project
cd ml-project
git init
echo "# ML Project" > README.md
git add README.md
git commit -m "Initial commit: added README"
```

Command / Line of Code	Meaning / Explanation

mkdir ml-project	Creates a new project folder named 'ml-project'.
cd ml-project	Moves into that folder.
git init	Initializes a new, empty Git repository (creates a hidden .git folder that tracks history).
echo "# ML Project" > README.md	Creates a README.md file and take the text "# ML Project" and write it into a it.
git add README.md	Stages the README.md file — marks it to be included in the next commit.
git commit -m "Initial commit: added README"	Saves a permanent snapshot (commit) of the staged changes with a descriptive message.

2.5 Connecting to GitHub (Remote Repository)

```
git remote add origin https://github.com/username/ml-project.git
git branch -M main
git push -u origin main
```

Command / Line of Code	**Meaning / Explanation**
git remote add origin <url>	Links the local repository to a remote repository on GitHub, naming that remote 'origin'.
git branch -M main	Renames the current branch to 'main' (GitHub's default branch name).
git push -u origin main	Uploads local commits to GitHub's 'main' branch and sets it as the default upstream for future 'git push'/'git pull'.

2.6 Branching — Managing ML Project Source Code

In Git, **branching** means creating a separate line of development so you can work on something without disturbing the main code.

```
            feature-login
           /
A---B---C------D
    \
     main
```

More accurately, main continues along one path while your feature branch develops independently:

```
A---B---C---E---F       ← main
     \
      D---G---H     ← feature-login
```

```
git branch feature-preprocessing
git checkout feature-preprocessing
REM or in one step:
git checkout -b feature-training
git status

cat > train.py <<'EOF'
print("Starting ML training...")

model_name = "Linear Regression"
accuracy = 0.92

print("Model:", model_name)
print("Accuracy:", accuracy)
EOF

git add train.py
git commit -m "Added baseline training script"
git push origin feature-training
```

Command / Line of Code	**Meaning / Explanation**
git branch feature-preprocessing	Creates a new branch called 'feature-preprocessing' without switching to it.
git checkout feature-preprocessing	Switches the working directory to the specified branch.
git checkout -b feature-training	Creates AND switches to a new branch 'feature-training' in a single command.
git status	Shows which files are modified, staged, or untracked in the current branch.
cat > train.py <<'EOF' print("Starting ML training...") model_name = "Linear Regression" accuracy = 0.92	This will create train.py with those lines of Python code.

print("Model:", model_name) print("Accuracy:", accuracy) EOF	
git add train.py	Stages the file train.py for the next commit.
git commit -m "..."	Commits the staged changes on this feature branch.
git push origin feature-training	Pushes the new branch and its commits to GitHub.

2.7 Merging Branches

```
git checkout main
git pull origin main
git merge feature-training
git push origin main
```

Command / Line of Code	Meaning / Explanation
git checkout main	Switches back to the main branch, the target of the merge.
git pull origin main	Fetches and merges the latest changes from GitHub into local main (keeps it up to date first).
git merge feature-training	Combines the commits from 'feature-training' into 'main'.
git push origin main	Publishes the merged main branch back to GitHub.

2.8 Merge Conflict Resolution

A conflict occurs when the same lines of a file are changed differently on two branches. Example resolution workflow:

```
REM --- Create the conflict ---
git checkout main

echo 'print("Version from MAIN")' > train.py
git add train.py
git commit -m "Update train.py on main"

git checkout feature-training
echo ‘print("Version from FEATURE branch")’ > train.py
git add train.py
git commit -m "Update train.py on feature-training"
```

```
REM --- Now merge and trigger the conflict ---
git checkout main
git merge feature-training
REM Output: CONFLICT (content): Merge conflict in train.py

REM Open train.py -> resolve manually between <<<<<<<, =======, >>>>>>> markers
git add train.py
git commit -m "Resolved merge conflict in train.py"
```

Command / Line of Code	Meaning / Explanation
CONFLICT (content) message	Git tells you it could not automatically merge the file and marks the conflicting sections.
<<<<<<< HEAD ... ======= ... >>>>>>> branch	Conflict markers Git inserts into the file: the section above ======= is your current branch's (main) version, the section below is the incoming branch's (feature-training) version. You manually edit to keep the correct code and delete the markers.
git add train.py	After manually fixing the file, staging it tells Git the conflict is resolved.
git commit -m "..."	Finalizes the merge commit with the resolved content.

2.9 Version History

```
git log
git log --oneline --graph --all
git diff HEAD~1 HEAD
git show <commit-hash>
```

Command / Line of Code	Meaning / Explanation
git log	Shows the full commit history (author, date, message, hash) in reverse chronological order.
git log --oneline --graph --all	Compact one-line-per-commit view with an ASCII graph of all branches — great for visualizing merges.
git diff HEAD~1 HEAD	Shows the exact line changes between the previous commit and the current one.
git show <commit-hash>	Displays full details (metadata + diff) of one specific commit.

2.10 Important Git Commands Reference Table

Command / Line of Code	Meaning / Explanation
git clone <url>	Downloads (copies) an existing remote repository to your local machine.
git status	Shows the state of the working directory and staging area.
git add .	Stages ALL changed/new files in the current directory.
git commit -am "msg"	Stages all TRACKED modified files and commits them in one step.
git pull	Fetches remote changes and merges them into the current branch.
git fetch	Downloads remote commits/branches WITHOUT merging them (safe preview).
git stash	Temporarily saves uncommitted changes so you can switch branches cleanly.
git stash pop	Restores the most recently stashed changes.
git reset --hard HEAD	Discards all uncommitted changes, reverting to the last commit.
git revert <commit>	Creates a NEW commit that undoes the changes of a previous commit (safe for shared history).
git branch -d branch-name	Deletes a branch that has already been merged.
git tag v1.0	Marks a specific commit as a release point, e.g., v1.0.
git rm --cached file	Stops tracking a file without deleting it from disk.
git ignore (.gitignore file)	A file listing patterns (e.g., *.pyc, venv/) that Git should never track.

2.11 Collaboration Using GitHub — Typical Workflow

Command / Line of Code	Meaning / Explanation
Fork	Creates your own copy of someone else's repository on GitHub.
Clone	git clone <your-fork-url> — downloads your fork locally to work on it.

Command / Line of Code	Meaning / Explanation
Create feature branch	git checkout -b fix-bug-123 — isolates your change.
Commit & Push	git add, git commit, git push origin fix-bug-123.
Pull Request (PR)	On GitHub, open a PR from your branch into the original repo's main branch for review.
Code Review	Collaborators comment/approve changes before merging.
Merge PR	Once approved, the PR is merged into main, integrating the change.

Note: GitLab uses the equivalent term 'Merge Request (MR)' instead of 'Pull Request'; the underlying git commands are identical.

Conclusion:

In this practical, we learned how to install and configure Git for version control. We created a repository, performed branching and merging operations, and understood how Git helps manage source code changes in ML projects. We also simulated a merge conflict and successfully resolved it, and explored collaboration workflows using GitHub. This experiment helped us understand the importance of version control in maintaining code history and enabling smooth collaboration among team members.

Date of Performance: …… **Date of Submission:** ……

Assignment 3: DVC — Data Version Control

3.1 Objective

Install and configure DVC, track dataset versions, store data artifacts, compare dataset versions, integrate DVC with Git, and ensure reproducibility of ML experiments.

3.2 Installation on Windows

```
pip install dvc
REM Optional: with cloud remote support
pip install "dvc[gdrive]"
dvc --version
```

Command / Line of Code	Meaning / Explanation
pip install dvc	Installs the core DVC (Data Version Control) tool via pip.
pip install "dvc[gdrive]"	Install DVC and everything it needs to use Google Drive as a remote storage location.
dvc --version	Confirms DVC installed correctly and shows its version.

3.3 Initializing DVC in a Git Project

```
cd ml-project
dvc init
git status
git commit -m "Initialize DVC"
```

Command / Line of Code	Meaning / Explanation
dvc init	Sets up DVC inside an existing Git repo — creates a .dvc/ folder for internal config/cache.
git status	Shows the new .dvc files that DVC created, ready to be committed to Git.
git commit -m "Initialize DVC"	Commits DVC's configuration files to Git so the project history records that DVC is enabled.

3.4 Dataset Tracking and Version Management

Steps to follow:
1. Make a new folder "data".
2. Inside the data folder make a simple csv ("raw_dataset") or any kind of file.

```
dvc add data/raw_dataset.csv
git add data/raw_dataset.csv.dvc data/.gitignore
git commit -m "Track raw dataset v1 with DVC"
```

Command / Line of Code	Meaning / Explanation
dvc add data/raw_dataset.csv	Tells DVC to start tracking this dataset: it moves the actual data into DVC's cache and creates a small pointer file 'raw_dataset.csv.dvc'.
git add data/raw_dataset.csv.dvc data/.gitignore	Stages the lightweight .dvc pointer file and the auto-generated .gitignore (which excludes the actual large data file from Git).
git commit -m "Track raw dataset v1 with DVC"	Commits the pointer file to Git — this is how Git 'knows about' a large dataset without storing it directly.

3.5 Storage of Data Artifacts (Remote Storage)

```
dvc remote add -d myremote D:\dvc_storage
REM Or a cloud remote, e.g. Google Drive:
dvc remote add -d gdrive_remote gdrive://<folder-id>
git commit .dvc\config -m "Configure DVC remote storage"
dvc push
```

Command / Line of Code	Meaning / Explanation
dvc remote add -d myremote D:\dvc_storage	Registers a local folder (or cloud bucket) as DVC's default (-d) remote storage location for large files.
git commit .dvc\config -m "..."	Commits the remote configuration so teammates can pull from the same storage location.
dvc push	Uploads the actual data files (tracked via .dvc pointers) to the configured remote storage.

3.6 Comparison of Dataset Versions

```
REM After modifying the dataset and re-adding it:
dvc add data\raw_dataset.csv
git add data\raw_dataset.csv.dvc
git commit -m "Update dataset to v2"
dvc push
REM Compare versions:
git log data\raw_dataset.csv.dvc
git diff HEAD~1 HEAD -- data\raw_dataset.csv.dvc
dvc diff HEAD~1 HEAD
```

Command / Line of Code	Meaning / Explanation
git log data\raw_dataset.csv.dvc	Lists every commit that changed the dataset's pointer file — i.e., the full version history of the dataset.
git diff HEAD~1 HEAD -- <file>.dvc	Shows the change in the pointer file's md5 hash between two commits, proving the underlying data changed.
dvc diff HEAD~1 HEAD	DVC-native command that reports which tracked data files changed (added/deleted/modified) between two commits/revisions.

3.7 Retrieving a Previous Dataset Version

```
git checkout <commit-hash> -- data\raw_dataset.csv.dvc
dvc checkout
```

Command / Line of Code	Meaning / Explanation
git checkout <hash> -- <file>.dvc	Restores the OLD pointer file from a previous commit into the working directory.
dvc checkout	Uses the restored pointer file to pull the matching version of the actual data from the DVC cache/remote — syncing the real dataset file to that historical version.

3.8 Integration of DVC with Git — Pipelines & Reproducibility

```
dvc stage add -n preprocess -d src\preprocess.py -d data\raw_dataset.csv -o data\processed.csv
python src\preprocess.py
dvc stage add -n train -d src\train.py -d data\processed.csv -o models\model.pkl python src\train.py
```

```
dvc repro
git add dvc.yaml dvc.lock
git commit -m "Add DVC pipeline for preprocessing and training"
```

Command / Line of Code	Meaning / Explanation
dvc stage add -n preprocess ... python src\preprocess.py	Defines a named pipeline stage 'preprocess': its dependencies (-d), its output (-o), and the command to run — saved into dvc.yaml.
dvc stage add -n train ...	Defines a second stage 'train' that depends on the preprocessing output, forming a DAG (directed acyclic graph).
dvc repro	Reproduces the pipeline: re-runs only the stages whose dependencies changed, guaranteeing consistent, reproducible results.
dvc.lock	Auto-generated file recording exact hashes of inputs/outputs for the last successful run — the key to reproducibility.
git add dvc.yaml dvc.lock	Commits the pipeline definition and lock file to Git so the whole pipeline is versioned alongside code.

3.9 Important DVC Commands Reference

Command / Line of Code	Meaning / Explanation
dvc status	Shows whether tracked data/pipeline outputs are out of sync with their source.
dvc pull	Downloads data files from remote storage into the local workspace/cache.
dvc list <repo-url>	Lists DVC-tracked files in a remote Git+DVC repository without cloning.
dvc dag	Prints/visualizes the pipeline dependency graph (DAG) of stages.
dvc metrics show	Displays metrics logged by pipeline stages (e.g., accuracy, loss).
dvc params diff	Compares hyperparameters between two experiment versions.
dvc exp run	Runs an experiment variant without polluting the main pipeline history.

Command / Line of Code	Meaning / Explanation
dvc gc	Garbage-collects unused cached data files to save disk space.

Conclusion:

In this practical, we installed and configured DVC (Data Version Control) and integrated it with Git to manage dataset versions effectively. We tracked datasets, stored data artifacts in remote storage, and compared different versions of the dataset. This experiment helped us understand how DVC ensures reproducibility of ML experiments by keeping data and code versions in sync, which is essential for maintaining consistency across ML pipeline runs.

Date of Performance: …… **Date of Submission:** ……

Assignment 4: Data Collection, Preprocessing, Feature Engineering & Pipeline Automation

4.1 Objective

To collect data, preprocess it, engineer features, validate data quality, design a pipeline, and automate/execute an end-to-end data processing workflow.

4.2 Setup

```
pip install pandas numpy scikit-learn great_expectations
```

Command / Line of Code	Meaning / Explanation
pip install pandas numpy scikit-learn great_expectations	Installs libraries for data handling (pandas/numpy), preprocessing/feature engineering (scikit-learn) and data validation (Great Expectations).

4.3 Step 1 — Data Collection

```
import pandas as pd

df = pd.read_csv("data/raw_dataset.csv")
print(df.shape)
print(df.head())
```

Command / Line of Code	Meaning / Explanation
import pandas as pd	Imports the pandas library, aliased as 'pd', used for tabular data handling.
df = pd.read_csv("data/raw_dataset.csv")	Reads a CSV file from disk into a pandas DataFrame called df — this is the 'data collection' step.
print(df.shape)	Prints (rows, columns) — a quick sanity check on the dataset's size.
print(df.head())	Displays the first 5 rows so we can visually inspect the raw data.

4.4 Step 2 — Data Preprocessing

```
# Handle missing values
df = df.dropna(subset=["target"])
```

```
df.fillna(df.mean(numeric_only=True), inplace=True)

# Remove duplicates
df = df.drop_duplicates()

# Encode categorical variables
df = pd.get_dummies(df, columns=["category_col"], drop_first=True)
```

Command / Line of Code	Meaning / Explanation
df.dropna(subset=["target"])	Removes rows where the target/label column is missing, since such rows are unusable for supervised learning.
df.fillna(df.mean(numeric_only=True), inplace=True)	Fills remaining missing numeric values with the column mean — a simple imputation strategy.
df.drop_duplicates()	Removes exact duplicate rows to avoid biasing the model.
pd.get_dummies(df, columns=["category_col"], drop_first=True)	One-hot encodes a categorical column into binary indicator columns; drop_first avoids the 'dummy variable trap' (multicollinearity).

4.5 Step 3 — Feature Engineering

```
from sklearn.preprocessing import StandardScaler

df["income_per_age"] = df["income"] / df["age"]

scaler = StandardScaler()
num_cols = ["age", "income", "income_per_age"]
df[num_cols] = scaler.fit_transform(df[num_cols])
```

Command / Line of Code	Meaning / Explanation
df["income_per_age"] = df["income"] / df["age"]	Creates a new engineered feature (ratio) that may carry more predictive signal than the raw columns alone.
scaler = StandardScaler()	Instantiates a scaler that will standardize features to zero mean and unit variance.
scaler.fit_transform(df[num_cols])	Learns the mean/std of each numeric column and transforms them in one step — required by many ML algorithms (e.g., SVM, KNN, logistic regression).

4.6 Step 4 — Data Validation

```
assert df.isnull().sum().sum() == 0, "Missing values still present!"
assert (df["age"] >= 0).all(), "Invalid negative age found!"
assert df.duplicated().sum() == 0, "Duplicate rows found!"
print("Data validation passed.")
```

Command / Line of Code	Meaning / Explanation
assert df.isnull().sum().sum() == 0	Programmatically checks that no missing values remain; raises an error and stops the pipeline if the condition fails.
assert (df["age"] >= 0).all()	Checks a business rule: age cannot be negative — catches corrupted/invalid data.
assert df.duplicated().sum() == 0	Confirms no duplicate rows remain after preprocessing.
print("Data validation passed.")	Confirms to the user/logs that all checks succeeded before the pipeline proceeds.

4.7 Step 5 — Automated End-to-End Pipeline (Python script)

Save as data_pipeline.py:

```
import pandas as pd
from sklearn.preprocessing import StandardScaler

def collect_data(path):
    return pd.read_csv(path)

def preprocess(df):
    df = df.dropna(subset=["target"])
    df.fillna(df.mean(numeric_only=True), inplace=True)
    df = df.drop_duplicates()
    return df

def engineer_features(df):
    df["income_per_age"] = df["income"] / df["age"]
    scaler = StandardScaler()
    cols = ["age", "income", "income_per_age"]
    df[cols] = scaler.fit_transform(df[cols])
    return df

def validate(df):
```

```
    assert df.isnull().sum().sum() == 0
    assert df.duplicated().sum() == 0
    return True

def run_pipeline(input_path, output_path):
    df = collect_data(input_path)
    df = preprocess(df)
    df = engineer_features(df)
    validate(df)
    df.to_csv(output_path, index=False)
    print(f"Pipeline complete. Output saved to {output_path}")

if __name__ == "__main__":
    run_pipeline("data/raw_dataset.csv", "data/processed_dataset.csv")
```

Command / Line of Code	Meaning / Explanation
def collect_data(path): ...	Wraps data collection in a reusable function so the pipeline stage is modular and testable.
def preprocess(df): ...	Encapsulates all cleaning steps as one function/stage.
def engineer_features(df): ...	Encapsulates feature-creation and scaling as one function/stage.
def validate(df): ...	A dedicated validation stage that halts execution (via assert) if data quality checks fail.
def run_pipeline(input_path, output_path): ...	Orchestrates all stages in sequence — this IS the 'end-to-end pipeline'.
if __name__ == "__main__": run_pipeline(...)	Ensures the pipeline only auto-runs when the script is executed directly (not when imported as a module).

Run it on Windows:

```
python data_pipeline.py
```

Command / Line of Code	Meaning / Explanation
python data_pipeline.py	Executes the full automated pipeline end-to-end: collection -> preprocessing -> feature engineering -> validation -> save output.

Conclusion:

In this practical, we performed data collection, preprocessing, and feature engineering on a dataset,

followed by data validation to ensure data quality. We designed and automated an end-to-end data pipeline that executes all preprocessing steps sequentially. This experiment helped us understand the importance of building clean, validated, and automated data pipelines, which form the foundation of reliable machine learning systems.

Date of Performance: …… **Date of Submission:** ……

Assignment 5: Introduction to Feature Stores (Feast)

5.1 Objective

Create a feature repository, define and register features, ingest data, retrieve online/offline features, and analyze the benefits of a feature store in MLOps.

5.2 Installation

```
pip install feast
feast version
```

Command / Line of Code	Meaning / Explanation
pip install feast	Installs Feast, an open-source feature store framework.
feast version	Confirms installation and shows the installed Feast version.

5.3 Feature Repository Creation

```
feast init customer_repo
cd customer_repo\feature_repo
dir
```

Command / Line of Code	Meaning / Explanation
feast init customer_repo	Scaffolds a new Feast feature repository with example files (feature_store.yaml, example.py, data/).
cd customer_repo\feature_repo	Enters the generated feature repository folder.
dir	Lists generated files: feature_store.yaml (config), example_repo.py (feature definitions), data/ (sample parquet files).

5.4 Feature Definition and Registration

Edit example_repo.py:

```
from datetime import timedelta
from feast import Entity, FeatureView, Field, FileSource
from feast.types import Float32, Int64

customer = Entity(name="customer_id", join_keys=["customer_id"])

customer_stats_source = FileSource(
    path="data/customer_stats.parquet",
    timestamp_field="event_timestamp",
)

customer_stats_fv = FeatureView(
    name="customer_stats",
    entities=[customer],
    ttl=timedelta(days=1),
    schema=[
```

```
    Field(name="total_spend", dtype=Float32),
    Field(name="num_transactions", dtype=Int64),
  ],
  source=customer_stats_source,
)
```

Command / Line of Code	Meaning / Explanation
Entity(name="customer_id", join_keys=["customer_id"])	Defines the primary key/entity that features are associated with — here, each customer.
FileSource(path=..., timestamp_field="event_timestamp")	Points Feast to the raw data file and tells it which column holds event timestamps (needed for point-in-time correctness).
FeatureView(name="customer_stats", entities=[customer], ...)	Groups a set of related features together, tied to the 'customer' entity, ready to be served.
ttl=timedelta(days=1)	Time-to-live: how 'fresh' a feature value must be to be considered valid during retrieval.
schema=[Field(name="total_spend", dtype=Float32), ...]	Explicitly declares each feature's name and data type — this is the feature 'definition'.

feast apply

Command / Line of Code	Meaning / Explanation
feast apply	Scans the repo's Python files, validates the feature definitions, and REGISTERS them into Feast's metadata registry (registry.db).

5.5 Data Ingestion (Materialization)

```
feast materialize-incremental $(date +%Y-%m-%d)
REM On Windows PowerShell, use:
feast materialize-incremental (Get-Date -Format "yyyy-MM-dd")
```

Command / Line of Code	Meaning / Explanation
feast materialize-incremental <date>	Loads (ingests) feature values from the offline source (parquet/warehouse) into the online store, up to the given timestamp — making them fast to serve in real time.

5.6 Retrieval of Offline Features (for Training)

```
from feast import FeatureStore
```

```
import pandas as pd

store = FeatureStore(repo_path=".")

entity_df = pd.DataFrame({
    "customer_id": [1001, 1002],
    "event_timestamp": [pd.Timestamp.now()] * 2,
})

training_df = store.get_historical_features(
    entity_df=entity_df,
    features=["customer_stats:total_spend", "customer_stats:num_transactions"],
).to_df()
print(training_df)
```

Command / Line of Code	Meaning / Explanation
store = FeatureStore(repo_path=".")	Connects to the local feature repository defined by feature_store.yaml.
entity_df = pd.DataFrame({...})	A DataFrame of entity keys + timestamps for which we want historical feature values — mimics a training label set.
store.get_historical_features(...)	Performs a point-in-time-correct JOIN between entity_df and the feature definitions, preventing data leakage from future values.
.to_df()	Converts the retrieval job's result into a pandas DataFrame usable directly for model training.

5.7 Retrieval of Online Features (for Real-time Serving)

```
online_features = store.get_online_features(
    features=["customer_stats:total_spend", "customer_stats:num_transactions"],
    entity_rows=[{"customer_id": 1001}],
).to_dict()
print(online_features)
```

Command / Line of Code	Meaning / Explanation
store.get_online_features(...)	Fetches the LATEST feature values for given entities with low latency — used at prediction/inference time in production.
entity_rows=[{"customer_id": 1001}]	Specifies which entity's features to look up (a single

	customer here).
.to_dict()	Converts the response into a Python dictionary, easy to feed directly into a deployed model's API.

5.8 Benefits of a Feature Store in MLOps

- Consistency: Same feature computation logic used in training (offline) and serving (online), eliminating training-serving skew.
- Reusability: Features defined once can be reused across multiple models/teams, reducing duplicate engineering effort.
- Point-in-time correctness: Prevents data leakage by ensuring only historically valid feature values are used for training.
- Low-latency serving: Online store enables real-time predictions with millisecond feature lookups.
- Governance & Discovery: Centralized registry makes it easy to discover, document, and audit existing features.
- Versioning: Feature definitions can be versioned alongside code (Git), supporting reproducibility.

Conclusion:

In this experiment, we learned about Feature Stores and their role in MLOps. We created a feature repository, defined and registered features, and performed data ingestion into the store. We also retrieved both online and offline features and analyzed the benefits of using a Feature Store, such as consistency between training and serving data and reusability of features across multiple models. This experiment helped us understand how Feature Stores reduce training-serving skew and improve collaboration in ML projects.

Date of Performance: …… **Date of Submission:** ……

Assignment 6: MLflow — Experiment Tracking and Model Registry

6.1 Objective

Install and configure MLflow, train ML models, log parameters/metrics/artifacts, compare experiments, register models, and select the best model.

6.2 Installation

```
pip install mlflow scikit-learn pandas
mlflow --version
```

Command / Line of Code	Meaning / Explanation
pip install mlflow scikit-learn pandas	Installs MLflow along with scikit-learn (for the model) and pandas (for data handling).
mlflow --version	Confirms MLflow installed correctly.

6.3 Starting the MLflow Tracking Server (local UI)

```
mlflow ui
REM By default this serves the UI at:
REM http://127.0.0.1:5000
```

Command / Line of Code	Meaning / Explanation
mlflow ui	Launches a local web server that reads the 'mlruns' folder and displays all logged experiments/runs in a browser dashboard.

6.4 Training a Model with Logging (train_mlflow.py)

```
import mlflow
import mlflow.sklearn
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, f1_score

mlflow.set_experiment("iris-classification")

X, y = load_iris(return_X_y=True)
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

with mlflow.start_run(run_name="rf_baseline"):
    n_estimators = 100
    max_depth = 5

    mlflow.log_param("n_estimators", n_estimators)
    mlflow.log_param("max_depth", max_depth)

    model = RandomForestClassifier(n_estimators=n_estimators, max_depth=max_depth,
random_state=42)
    model.fit(X_train, y_train)
    preds = model.predict(X_test)

    acc = accuracy_score(y_test, preds)
    f1 = f1_score(y_test, preds, average="macro")

    mlflow.log_metric("accuracy", acc)
    mlflow.log_metric("f1_score", f1)

    mlflow.sklearn.log_model(model, "model")
    print(f"Accuracy: {acc:.4f}, F1: {f1:.4f}")
```

Command / Line of Code	Meaning / Explanation
mlflow.set_experiment("iris-classification")	Creates (or selects) an experiment named 'iris-classification' — a container that groups related runs together.
with mlflow.start_run(run_name="rf_baseline"):	Opens a new tracked 'run' — everything logged inside this block belongs to this single run and is closed automatically at block end.
mlflow.log_param("n_estimators", n_estimators)	Logs a hyperparameter (input configuration) so it's recorded and comparable across runs.
model.fit(X_train, y_train)	Trains the RandomForestClassifier on the training split.
mlflow.log_metric("accuracy", acc)	Logs a numeric performance metric tied to this run, viewable/sortable in the MLflow UI.
mlflow.sklearn.log_model(model, "model")	Serializes and saves the trained model itself as an MLflow 'artifact', including environment info needed to reload it later.

Run on Windows:

```
python train_mlflow.py
```

6.5 Experiment Comparison

```
REM In the MLflow UI (http://127.0.0.1:5000):
REM 1. Open the 'iris-classification' experiment
REM 2. Select multiple runs using checkboxes
REM 3. Click 'Compare' to see side-by-side params/metrics and parallel-coordinate plots

REM Programmatically:
import mlflow
runs = mlflow.search_runs(experiment_names=["iris-classification"])
print(runs[["run_id", "params.n_estimators", "metrics.accuracy", "metrics.f1_score"]])
```

Command / Line of Code	Meaning / Explanation
mlflow.search_runs(experiment_names=[...])	Queries MLflow's tracking store and returns all runs of an experiment as a pandas DataFrame, including all logged params/metrics as columns.
runs[["run_id", "params...", "metrics..."]]	Selects just the relevant columns to compare runs side by side programmatically.

6.6 Model Registration

```
import mlflow

run_id = "<copy-the-run-id-from-UI-or-search_runs>"
model_uri = f"runs:/{run_id}/model"

mlflow.register_model(model_uri=model_uri, name="IrisClassifier")
```

Command / Line of Code	Meaning / Explanation
model_uri = f"runs:/{run_id}/model"	Builds a URI pointing to the specific model artifact logged in a specific run.
mlflow.register_model(model_uri, name="IrisClassifier")	Adds this model version to the MLflow Model Registry under the name 'IrisClassifier', enabling stage management (Staging/Production) and versioning.

6.7 Selecting the Best Model

```
best_run = runs.sort_values("metrics.accuracy", ascending=False).iloc[0]
print("Best run ID:", best_run.run_id)
print("Best accuracy:", best_run["metrics.accuracy"])
```

Command / Line of Code	Meaning / Explanation
runs.sort_values("metrics.accuracy", ascending=False)	Sorts all logged runs by accuracy, highest first.
.iloc[0]	Selects the top row — i.e., the single best-performing run — for promotion to the model registry / production.

```
from mlflow import MlflowClient
client = MlflowClient()
client.transition_model_version_stage(name="IrisClassifier", version=1, stage="Production")
```

Command / Line of Code	Meaning / Explanation
MlflowClient()	Creates a low-level client object for programmatic interaction with the tracking server and model registry.
client.transition_model_version_stage(...)	Promotes a specific registered model version to a lifecycle stage such as 'Staging' or 'Production', signalling it is ready for deployment.

Conclusion:

In this experiment, we installed and configured MLflow to track machine learning experiments. We trained ML models while logging parameters, metrics, and artifacts, and compared multiple experiment runs using the MLflow UI. We also registered the best-performing model in the Model Registry. This experiment helped us understand how MLflow simplifies experiment tracking, model versioning, and selection of the best model in an MLOps workflow.

Date of Performance: …… **Date of Submission:** ……

Assignment 7: Baseline Model, Grid Search / Random Search, and MLflow Tracking

7.1 Objective

Develop a baseline ML model, perform hyperparameter tuning using Grid Search and Random Search, evaluate performance, and track every experiment using MLflow.

7.2 Baseline Model

```
import mlflow
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

X, y = load_wine(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42)

mlflow.set_experiment("wine-hpo")

with mlflow.start_run(run_name="baseline"):
    baseline_model = RandomForestClassifier(random_state=42)
    baseline_model.fit(X_train, y_train)
    baseline_acc = accuracy_score(y_test, baseline_model.predict(X_test))
    mlflow.log_metric("accuracy", baseline_acc)
    print("Baseline accuracy:", baseline_acc)
```

Command / Line of Code	Meaning / Explanation
RandomForestClassifier(random_state=42)	Creates a baseline model using default hyperparameters — this is the reference point to beat via tuning.
mlflow.log_metric("accuracy", baseline_acc)	Logs the baseline's performance so it can later be compared against tuned models in the MLflow UI.

7.3 Hyperparameter Tuning — Grid Search

```
from sklearn.model_selection import GridSearchCV

param_grid = {
    "n_estimators": [50, 100, 200],
```

```
    "max_depth": [3, 5, 10, None],
    "min_samples_split": [2, 5],
}

with mlflow.start_run(run_name="grid_search"):
    grid_search = GridSearchCV(
        RandomForestClassifier(random_state=42),
        param_grid=param_grid,
        cv=5,
        scoring="accuracy",
        n_jobs=-1,
    )
    grid_search.fit(X_train, y_train)

    best_params = grid_search.best_params_
    mlflow.log_params(best_params)

    test_acc = accuracy_score(y_test, grid_search.best_estimator_.predict(X_test))
    mlflow.log_metric("accuracy", test_acc)
    mlflow.sklearn.log_model(grid_search.best_estimator_, "model")
    print("Grid Search best params:", best_params, "| Test acc:", test_acc)
```

Command / Line of Code	**Meaning / Explanation**
param_grid = {...}	Defines the exhaustive set of hyperparameter values Grid Search will try EVERY combination of.
GridSearchCV(..., cv=5, scoring="accuracy", n_jobs=-1)	Wraps the model in a search that performs 5-fold cross-validation for every combination in param_grid; n_jobs=-1 uses all CPU cores in parallel.
grid_search.fit(X_train, y_train)	Runs the full exhaustive search — trains and cross-validates a model for every hyperparameter combination.
grid_search.best_params_	Returns the hyperparameter combination that achieved the highest cross-validation score.
grid_search.best_estimator_	The final model already refit on the full training set using the best-found hyperparameters.
mlflow.log_params(best_params)	Logs the winning hyperparameters as an MLflow run so they're tracked and comparable to the baseline.

7.4 Hyperparameter Tuning — Random Search

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint

param_dist = {
   "n_estimators": randint(50, 300),
   "max_depth": randint(2, 20),
   "min_samples_split": randint(2, 10),
}

with mlflow.start_run(run_name="random_search"):
   random_search = RandomizedSearchCV(
      RandomForestClassifier(random_state=42),
      param_distributions=param_dist,
      n_iter=20,
      cv=5,
      scoring="accuracy",
      random_state=42,
      n_jobs=-1,
   )
   random_search.fit(X_train, y_train)

   mlflow.log_params(random_search.best_params_)
   test_acc = accuracy_score(y_test, random_search.best_estimator_.predict(X_test))
   mlflow.log_metric("accuracy", test_acc)
   mlflow.sklearn.log_model(random_search.best_estimator_, "model")
   print("Random Search best params:", random_search.best_params_, "| Test acc:", test_acc)
```

Command / Line of Code	Meaning / Explanation
param_dist = {"n_estimators": randint(50, 300), ...}	Defines PROBABILITY DISTRIBUTIONS (instead of fixed lists) to sample hyperparameters from — allows exploring a much larger/continuous search space.
RandomizedSearchCV(..., n_iter=20, ...)	Randomly samples only 20 combinations (n_iter) from param_dist instead of trying all of them — much faster than Grid Search for large spaces.
random_search.fit(X_train, y_train)	Executes the randomized search with cross-validation.
n_jobs=-1	Parallelizes the search across all available CPU cores to speed up training.

7.5 Performance Evaluation and Comparison

```
import mlflow
runs = mlflow.search_runs(experiment_names=["wine-hpo"])
print(runs[["tags.mlflow.runName", "metrics.accuracy"]].sort_values("metrics.accuracy",
ascending=False))
```

Command / Line of Code	Meaning / Explanation
mlflow.search_runs(experiment_names=["wine-hpo"])	Retrieves ALL runs (baseline, grid_search, random_search) from the experiment for comparison.
sort_values("metrics.accuracy", ascending=False)	Ranks the three approaches by test accuracy, making it immediately clear which tuning method produced the best model.

Note: Grid Search guarantees the best combination WITHIN the specified grid but scales exponentially with parameters. Random Search scales better and often finds near-optimal results faster, especially in high-dimensional spaces.

Conclusion:

In this experiment, we developed a baseline machine learning model and improved its performance using hyperparameter tuning techniques, namely Grid Search and Random Search. We evaluated model performance using appropriate metrics and tracked all experiment runs using MLflow. This experiment helped us understand the impact of hyperparameter tuning on model performance and the trade-offs between exhaustive (Grid Search) and randomized (Random Search) tuning approaches.

Date of Performance: …… **Date of Submission:** ……

Assignment 8: Unit Testing, Pipeline Testing, and CI

8.1 Objective

Write unit tests for data processing functions, test training pipelines, validate model outputs, automate test execution, and run tests using a CI tool (GitHub Actions).

8.2 Installation

```
pip install pytest pytest-cov
```

Command / Line of Code	Meaning / Explanation
pip install pytest pytest-cov	Installs pytest, Python's most popular testing framework, plus pytest-cov for measuring test coverage.

8.3 Project Structure

```
ml-project\
  src\
    data_pipeline.py
    train.py
  tests\
    test_data_pipeline.py
    test_train.py
  .github\workflows\ci.yml
```

8.4 Unit Testing of Data Processing Functions (tests/test_data_pipeline.py)

```
import pandas as pd
import pytest
from src.data_pipeline import preprocess, engineer_features, validate

@pytest.fixture
def sample_df():
    return pd.DataFrame({
        "age": [25, 30, None],
        "income": [50000, 60000, 55000],
        "target": [1, 0, 1],
    })
```

```
def test_preprocess_removes_nulls(sample_df):
    result = preprocess(sample_df)
    assert result.isnull().sum().sum() == 0

def test_preprocess_no_duplicates(sample_df):
    df_dup = pd.concat([sample_df, sample_df])
    result = preprocess(df_dup)
    assert result.duplicated().sum() == 0

def test_feature_engineering_adds_column(sample_df):
    df = preprocess(sample_df)
    result = engineer_features(df)
    assert "income_per_age" in result.columns

def test_validate_passes_on_clean_data(sample_df):
    df = engineer_features(preprocess(sample_df))
    assert validate(df) is True
```

Command / Line of Code	**Meaning / Explanation**
@pytest.fixture	Marks a function as a reusable setup routine — 'sample_df' will be freshly created and injected into every test function that requests it as an argument.
def test_preprocess_removes_nulls(sample_df):	A test function (pytest auto-discovers any function starting with 'test_'). Calls the real preprocess() function and checks its output.
assert result.isnull().sum().sum() == 0	The actual test assertion — the test PASSES only if there are zero missing values after preprocessing.
pd.concat([sample_df, sample_df])	Creates a DataFrame with intentional duplicate rows to specifically test duplicate-removal logic.
assert "income_per_age" in result.columns	Verifies the feature engineering function actually created the expected new column.
assert validate(df) is True	Confirms the validation function returns True (no exception raised) on clean data.

8.5 Testing of Training Pipelines (tests/test_train.py)

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
```

```
from sklearn.metrics import accuracy_score

def test_model_trains_without_error():
    X, y = load_iris(return_X_y=True)
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1)
    model = RandomForestClassifier(n_estimators=10, random_state=1)
    model.fit(X_train, y_train)
    assert hasattr(model, "estimators_")

def test_model_accuracy_above_threshold():
    X, y = load_iris(return_X_y=True)
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1)
    model = RandomForestClassifier(n_estimators=50, random_state=1)
    model.fit(X_train, y_train)
    acc = accuracy_score(y_test, model.predict(X_test))
    assert acc > 0.85, f"Model accuracy too low: {acc}"
```

Command / Line of Code	Meaning / Explanation
def test_model_trains_without_error():	Ensures the training code runs end-to-end without throwing exceptions — a 'smoke test'.
assert hasattr(model, "estimators_")	Checks a scikit-learn-specific internal attribute that only exists AFTER a RandomForest has actually been fitted — proves training happened.
def test_model_accuracy_above_threshold():	A regression-safety test: fails automatically if a future code change causes accuracy to silently drop.
assert acc > 0.85, f"..."	Asserts a minimum acceptable performance bar; the f-string provides a helpful failure message showing the actual value.

8.6 Validation of Model Outputs

```
import numpy as np

def test_predictions_are_valid_classes():
    X, y = load_iris(return_X_y=True)
    model = RandomForestClassifier(random_state=1).fit(X, y)
    preds = model.predict(X)
    assert set(np.unique(preds)).issubset(set(np.unique(y)))

def test_prediction_shape_matches_input():
    X, y = load_iris(return_X_y=True)
```

```
model = RandomForestClassifier(random_state=1).fit(X, y)
preds = model.predict(X)
assert preds.shape[0] == X.shape[0]
```

Command / Line of Code	Meaning / Explanation
set(np.unique(preds)).issubset(set(np.unique(y)))	Checks that every predicted class label is one of the classes that actually exist in the training labels — catches invalid/out-of-range outputs.
preds.shape[0] == X.shape[0]	Confirms the model returns exactly one prediction per input row — a basic sanity/shape check on model outputs.

8.7 Executing Tests (Windows)

```
pytest
pytest -v
pytest --cov=src tests\
pytest tests\test_data_pipeline.py::test_preprocess_removes_nulls
```

Command / Line of Code	Meaning / Explanation
pytest	Auto-discovers and runs every test function in the 'tests' folder.
pytest -v	Verbose mode: prints the name and pass/fail status of every individual test.
pytest --cov=src tests\	Runs tests AND reports what percentage of the 'src' code was actually executed (covered) by tests.
pytest tests\test_data_pipeline.py::test_preprocess_removes_nulls	Runs one single specific test function by name — useful for debugging.

8.8 Automated Testing with a CI Tool (GitHub Actions)

.github/workflows/ci.yml:

```
name: ML Pipeline CI

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
```

```
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: '3.10'

      - name: Install dependencies
        run: pip install -r requirements.txt

      - name: Run tests
        run: pytest --cov=src tests\ --cov-report=xml
```

Command / Line of Code	Meaning / Explanation
on: push / pull_request: branches: [main]	Triggers this workflow automatically whenever code is pushed OR a pull request is opened targeting the 'main' branch.
runs-on: ubuntu-latest	Tells GitHub Actions to execute the job on a fresh Ubuntu Linux virtual machine in the cloud.
uses: actions/checkout@v4	A pre-built GitHub Action that checks out (downloads) the repository's code into the runner.
uses: actions/setup-python@v5 with python-version: '3.10'	Installs and configures the specified Python version on the runner.
run: pip install -r requirements.txt	Installs all project dependencies listed in requirements.txt.
run: pytest --cov=src tests\ --cov-report=xml	Runs the full automated test suite with coverage reporting; if any test fails, the whole CI job fails and blocks the merge/push.

Note: Once this file is committed and pushed to GitHub, every future push/PR automatically triggers this pipeline — visible under the repository's 'Actions' tab — enforcing that broken code/tests can never silently reach 'main'.

8.9 Important Testing Commands Reference

Command / Line of Code	Meaning / Explanation
pytest -k "train"	Runs only tests whose name contains the keyword 'train'.
pytest -x	Stops at the FIRST failing test instead of running all of them.
pytest --maxfail=2	Stops after 2 failures.
pytest -m slow	Runs only tests marked with @pytest.mark.slow (custom markers).
pytest --html=report.html	Generates an HTML test report (requires pytest-html).
coverage report -m	Shows a text summary of code coverage with missing line numbers.

Conclusion:

In this experiment, we performed unit testing on data processing functions and validated the training pipeline and model outputs. We created automated test cases using pytest and understood how testing ensures the reliability and correctness of ML pipelines. This experiment helped us understand the importance of continuous testing in ML systems and how CI tools can be used to automate test execution during development.