BIRMINGHAM CITY University

CMP5366 DATA MANAGEMENT AND MACHINE LEARNING OPERATIONS

Anush Pandey
Student ID: 24128423

Table of Contents

Table of figures

Table of Tables

Abstract

This report describes the creation, execution, and maintenance of an MLOps pipeline dedicated to predicting student depression risk using survey data. This pipeline features cutting-edge technological tools such as Apache Airflow for orchestration, MariaDB for storage, Redis for caching, Great Expectations for validation, MLflow for experiment tracking, and FastAPI for deployment, which modernize the processes of data ingestion, preprocessing, model training, and deployment automation. Key innovations are described in depth concerning data validation at important checkpoints, ethically sensitive mental health data de-identification through k-anonymity, and model training acceleration through Redis. A REST API provided access to a Random Forest classifier that maintained prediction accuracy despite challenges with missing values, outliers, and privacy issues. The implementation of this solution serves as an example of responsibly scalable, reproducible, and ethically aligned machine learning operations for social-good endeavors.

Keywords: MLOps Pipeline, Depression Prediction, Data Validation, Privacy-Preserving Analytics, Model Deployment

Report Introduction

In today's data-driven landscape, the ability to efficiently manage, deploy, and monitor machine learning (ML) models has become a crucial competency for digital organizations. Through real-world datasets and modern tools, the report addresses the complete lifecycle from data ingestion and preprocessing to model deployment and monitoring within an enterprise context. It also evaluates critical concerns surrounding data security, privacy, and compliance. With businesses increasingly depending on automated systems for decision-making, integrating robust MLOps pipelines ensures that insights are not only actionable but also ethically grounded (Allam et al., 2021). The report aims to reflect both technical proficiency and critical reflection, aligning with industry and academic expectations.

Identification of Three Candidate Source Data Sets

To create a solid supervised learning model, we need to pick a dataset that has labeled records for both training and evaluation (IBM, n.d.). Below, you'll find three potential datasets from Kaggle, evaluated based on their structure, features, and how well they fit into machine learning. This careful selection process helps us find the best dataset for effective model training, preprocessing, and analysis.

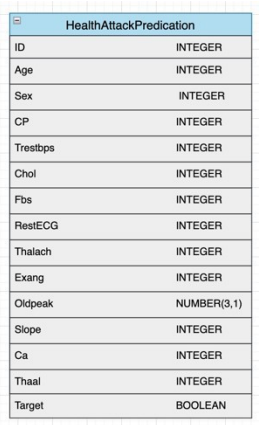
Candidate Source Data Set 1: Heart Attack Prediction

This dataset compiles anonymized patient records from various heart institutes around the world. It includes about 1,300 samples with details like age, resting blood pressure, cholesterol levels, chest pain type, and a binary outcome that shows whether heart disease is present. The data is organized in a flat file (CSV) format with 14 columns, featuring both categorical and continuous attributes. While this dataset is crucial from a medical standpoint, its relatively small size and specialized terminology (like thalassemia and angina) might make it challenging to apply directly without expert input or clinical insights (Juledz, 2022).

Data structure: Flat table Target variable: Presence of heart disease (binary) Limitation: Needs clinical expertise; smaller sample size

# age Numeric (e.g., 52)	# sex Categorical (0: Female, 1: Male)	# cp Categorical (0: Typical Angina, 1: Atypical Angina, 2: Non-anginal Pain, 3: Asymptomatic)	# trestbps Numeric (e.g., 125)	# chol Numeric in mg/dL (e.g., 212)	# fbs Categorical (0: <= 120 mg/dL, 1: > 120 mg/dL)	# restecg Categorical (0: Normal, 1: Abnormality, 2: Hypertrophy)	# thalach Numeric (e.g., 168)	# exang Categorical (0: No, 1: Yes)	# oldpeak Numeric (e.g., 1.0)
29 77	0 1	0 3	94 200	126 564	0 1	0 2	71 202	0 1	0 6.2
52	1	0	125	212	0	1	168	0	1.0
53	1	0	140	203	1	0	155	1	3.1
70	1	0	145	174	0	1	125	1	2.6
61	1	0	148	203	0	1	161	0	0.0
62	0	0	138	294	1	1	106	0	1.9
58	0	0	100	248	0	0	122	0	1.0
58	1	0	114	318	0	2	140	0	4.4
55	1	0	160	289	0	0	145	1	0.8
46	1	0	120	249	0	0	144	0	0.8
54	1	0	122	286	0	0	116	1	3.2
71	0	0	112	149	0	1	125	0	1.6
43	0	0	132	341	1	0	136	1	3.0
34	0	1	118	210	0	1	192	0	0.7
51	1	0	140	298	0	1	122	1	4.2
52	1	0	128	204	1	1	156	1	1.0
34	0	1	118	210	0	1	192	0	0.7
51	0	2	140	308	0	0	142	0	1.5

Figure 1: Heart attack prediction dataset



HealthAttackPredication	
ID	INTEGER
Age	INTEGER
Sex	INTEGER
CP	INTEGER
Trestbps	INTEGER
Chol	INTEGER
Fbs	INTEGER
RestECG	INTEGER
Thalach	INTEGER
Exang	INTEGER
Oldpeak	NUMBER(3,1)
Slope	INTEGER
Ca	INTEGER
Thaal	INTEGER
Target	BOOLEAN

Figure 2: ERD of heart attack prediction dataset

Candidate Source Data Set 2: Flight Price Prediction

This dataset boasts over 300,000 records and 34 columns, capturing details about flights throughout India, including the airline, departure times, number of stops, duration, and ticket prices. It's perfect for regression tasks aimed at predicting airfare. Structured as a flat CSV, it offers a wealth of features for model training. However, many of these features are quite specific to commercial travel, and preprocessing can often involve intricate text-to-category conversions (like parsing "5h 50m" for flight duration), which might divert attention from the main model development (Bathwal, 2021).

Data structure: Flat table Target variable: Flight ticket price (continuous) Limitation: Significant preprocessing needed for text conversion; regression-focused

# Serial Number	airline Airline Company	flight Flight Code	source_city Souce City	departure_time Departure Time	stops Number of Stops	arrival_time Arrival Time	destination_city Destination City	class Ticket Class	duration Flight Duration
0 300k	Vistara 43% Air_India 27% Other (91402) 30%	UK-706 1% UK-772 1% Other (294177) 98%	Delhi 20% Mumbai 20% Other (177914) 59%	Morning 24% Early_Morning 22% Other (162217) 54%	one 84% zero 12% Other (13286) 4%	Night 30% Evening 26% Other (130292) 43%	Mumbai 20% Delhi 19% Other (183696) 61%	Economy 69% Business 31%	0.83 49.8
0	SpiceJet	SG-8709	Delhi	Evening	zero	Night	Mumbai	Economy	2.17
1	SpiceJet	SG-8157	Delhi	Early_Morning	zero	Morning	Mumbai	Economy	2.33
2	AirAsia	I5-764	Delhi	Early_Morning	zero	Early_Morning	Mumbai	Economy	2.17
3	Vistara	UK-995	Delhi	Morning	zero	Afternoon	Mumbai	Economy	2.25
4	Vistara	UK-963	Delhi	Morning	zero	Morning	Mumbai	Economy	2.33
5	Vistara	UK-945	Delhi	Morning	zero	Afternoon	Mumbai	Economy	2.33
6	Vistara	UK-927	Delhi	Morning	zero	Morning	Mumbai	Economy	2.08
7	Vistara	UK-951	Delhi	Afternoon	zero	Evening	Mumbai	Economy	2.17
8	GO_FIRST	G8-334	Delhi	Early_Morning	zero	Morning	Mumbai	Economy	2.17
9	GO_FIRST	G8-336	Delhi	Afternoon	zero	Evening	Mumbai	Economy	2.25
10	GO_FIRST	G8-392	Delhi	Afternoon	zero	Evening	Mumbai	Economy	2.25
11	GO_FIRST	G8-338	Delhi	Morning	zero	Afternoon	Mumbai	Economy	2.33
12	Indigo	6E-5001	Delhi	Early_Morning	zero	Morning	Mumbai	Economy	2.17
13	Indigo	6E-6202	Delhi	Morning	zero	Afternoon	Mumbai	Economy	2.17
14	Indigo	6E-549	Delhi	Afternoon	zero	Evening	Mumbai	Economy	2.25
15	Indigo	6E-6278	Delhi	Morning	zero	Morning	Mumbai	Economy	2.33
16	Air_India	AI-887	Delhi	Early_Morning	zero	Morning	Mumbai	Economy	2.08
17	Air_India	AI-665	Delhi	Early_Morning	zero	Morning	Mumbai	Economy	2.17
18	AirAsia	I5-747	Delhi	Evening	one	Early_Morning	Mumbai	Economy	12.25

Figure 3: Flight ticket price prediction dataset

Flight Ticket Price Predication	
ID	INTEGER
Airline	VARCHAR
Flight_Code	VARCHAR
Source_City	VARCHAR
Destination_City	VARCHAR
Departure_Time	VARCHAR
Arrival_Time	VARCHAR
Stops	VARCHAR
Class	VARCHAR
Duration	NUMERIC(4,2)
Days_Left	INTERGER
Duration	INTERGER

Figure 4: ERD of flight price prediction

Candidate Source Data Set 3: Student Depression

The Student Depression dataset consists of labeled survey responses from Indian university students collected via Qualtrics. It includes a diverse mix of demographic, academic, and psychological indicators such as age, gender, CGPA, financial stress, suicidal thoughts, dietary habits, and sleep duration. The binary target label, Depression_Status, identifies whether a student is at risk of depression (Shamim, 2022).

The dataset was originally clean and well-structured. However, during exploratory analysis, it was discovered that two columns Work Pressure and Job Satisfaction contained only zero values across all records. These features lacked variance and analytical value, possibly due to flawed survey logic or skipped responses. In real-world data, such issues are common and require detection and removal to avoid skewing model training.

To simulate realistic data engineering challenges, the dataset was further modified. Float values were introduced in the Age column (e.g., 74.8443), and over 200 missing values were inserted across critical fields such as Depression_Status, Academic Pressure, and CGPA. These changes significantly increased the complexity of preprocessing, requiring data validation, imputation, and column removal to restore dataset integrity and ensure robust model training.

id A unique identifier assigned to each student record in the dataset.	Gender The gender of the student (e.g., Male, Female, Other). This helps in analyzing gender-specific trends in	Age The age of the student in years.	City The city or region where the student resides, providing geographical context for the analysis.	Profession The field of work or study of the student, which may offer insights into occupational or academic stress factors.	Academic Pressure A measure indicating the level of pressure the student faces in academic settings. This could include stress from	Work Pressure A measure of the pressure related to work or job responsibilities, relevant for students who are employed alongside	CGPA The cumulative grade point average of the student, reflecting overall academic performance.	Study Satisfaction An indicator of how satisfied the student is with their studies, which can correlate with mental well-being.	Job Satisfaction A measure of the student's satisfaction with their job or work environment, if applicable.
2 141k	Male 56% Female 44%	18 59	Kalyan 6% Srinagar 5% Other (24959) 89%	Student 100% Architect 0% Other (23) 0%	0 5	0 5	0 10	1.00 - 1.25 Count: 5,451 0 5	0 4
2	Male	33.0	Visakhapatnam	Student	5.0	0.0	8.97	2.0	0.0
8	Female	24.0	Bangalore	Student	2.0	0.0	5.9	5.0	0.0
26	Male	31.0	Srinagar	Student	3.0	0.0	7.03	5.0	0.0
30	Female	28.0	Varanasi	Student	3.0	0.0	5.59	2.0	0.0
32	Female	25.0	Jaipur	Student	4.0	0.0	8.13	3.0	0.0
33	Male	29.0	Pune	Student	2.0	0.0	5.7	3.0	0.0
52	Male	30.0	Thane	Student	3.0	0.0	9.54	4.0	0.0
56	Female	30.0	Chennai	Student	2.0	0.0	8.04	4.0	0.0
59	Male	28.0	Nagpur	Student	3.0	0.0	9.79	1.0	0.0
62	Male	31.0	Nashik	Student	2.0	0.0	8.38	3.0	0.0
83	Male	24.0	Nagpur	Student	3.0	0.0	6.1	3.0	0.0
91	Male	33.0	Vadodara	Student	3.0	0.0	7.03	4.0	0.0
94	Male	27.0	Kalyan	Student	5.0	0.0	7.04	1.0	0.0
100	Female	19.0	Rajkot	Student	2.0	0.0	8.52	4.0	0.0
103	Female	19.0	Kalyan	Student	5.0	0.0	5.64	5.0	0.0
106	Male	29.0	Srinagar	Student	3.0	0.0	8.58	3.0	0.0

Figure 5: Student depression dataset

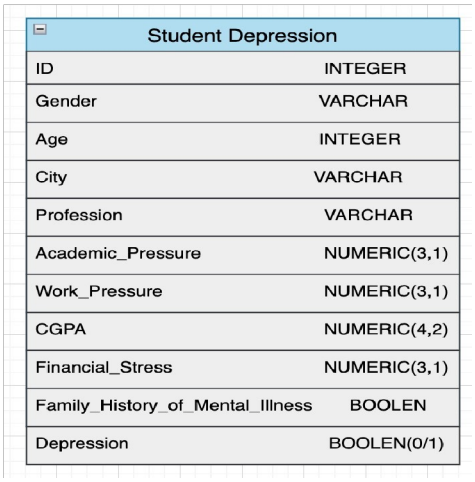


Figure 6: ERD of student depression

id	Gender	Age	City	Professio	Academic	Work Pre	CGPA	Study Sa	Job Satisf	Sleep Du	Dietary H	Degree	Have you	Work/Stu	Financial	Family Hi	Depression
2	Male	33	Visakhap	Student	5	0	8.97	2	0	'5-6 hours'		B.Pharm	Yes	3	1	No	1
8	Female	24	Bangalor	Student	2	0	5.9	5	0	'5-6 hour	Moderate	BSc	No	3	2	Yes	0
26	Male	31	Srinagar	Student	3	0	7.03		0	'Less tha	Healthy	BA	No	9		Yes	0
30	Female		Varanasi	Student	3	0	5.59	2	0		Moderate	BCA	Yes	4	5	Yes	1
32	Female	25	Jaipur	Student	4	0		3	0	'5-6 hour	Moderate	M.Tech	Yes	1	1	No	0
33		74.844	Pune	Student	2	0	5.7	3	0		Healthy	PhD	No	4	1	No	0
52	Male	30	Thane	Student	3	0		4	0	'7-8 hour	Healthy	BSc	No	1	2	No	0
56	Female	30	Chennai	Student	2	0	8.04	16.56	0	'Less tha	Unhealth	'Class 12'		0	1	Yes	5.5056
59	Male	28		Student	3	0	9.79	1	0	'7-8 hour	Moderate	B.Ed	Yes	12	3	No	1
62	Male	31	Nashik	Student	17.001	0	8.38	3	0	'Less tha	Moderate	LLB		2	5	No	1
83	Male		Nagpur	Student	3	0		3	0	'5-6 hour	Moderate	'Class 12	Yes	11	1	Yes	1
91	Male	33		Student	3	0.7199	7.03	4		'Less tha	Healthy	BE	Yes	10	2	Yes	0
94	Male	27	Kalyan	Student	5	0	7.04	1	0	'Less tha	Moderate	M.Tech	No	10	1	Yes	1
100	Female	19	Rajkot	Student	2	0	8.52	4	0	'Less tha	Unhealth	'Class 12	No	6	2	Yes	0
103	Female	74.844	Kalyan	Student	5	0	5.64	5	0	'Less tha	Moderate	'Class 12	Yes	4	5	Yes	1
106	Male	29	Srinagar	Student	3	0	8.58	3	0	'More tha	Moderate	M.Tech		10	2	Yes	1
120	Male	25	Nashik	Student	5	0	6.51	2	0	'Less tha	Unhealth	M.Ed	Yes	2	5	Yes	1
132	Female	20	Ahmedat	Student	5	0	7.25		0	'5-6 hour	Healthy	'Class 12	Yes	10	3	No	1
139	Male	19	Chennai	Student	2	0		2	0		Unhealth	'Class 12	No	6	3	No	0
145	Male	25	Kalyan	Student	3		9.93	3	0	'5-6 hour	Moderate	B.Ed		8	3		1
161	Male	29	Kolkata	Student	3	0	8.74	4	0	'5-6 hour	Moderate	B.Ed	Yes		1	No	0
162	Male	29	Kolkata	Student	3	0	6.73	3	0	'7-8 hour	Moderate	M.Tech	No	0	1	No	0
166	Female	25	Ahmedat	Student	3	0	5.57	3	0	'More tha	Unhealth	MSc	Yes	10	5	No	1
172	Male	23	Thane	Student	1	0	8.59	4	0	'7-8 hour	Healthy	BHM	No	11	3	No	0
173	Male		Bangalor	Student	4	0	7.1	3	0		Unhealth	'Class 12	Yes	11	5	Yes	1
176	Female	20	Mumbai	Student	5	0	8.58	16.56	0	'7-8 hour	Moderate	'Class 12	No	2	2	Yes	1
186	Male	31	Ahmedat	Student	2	0	6.08	5		'7-8 hour	Moderate	LLB	Yes	3	3	Yes	1
193	Male		Lucknow	Student	3	0	7.25	3	0	'More tha	Unhealth	M.Ed	Yes	10	5	No	1
208	Male	33	Indore	Student	5	0	5.74	2	0	'Less tha	Moderate	M.Pharm	No	8	3	Yes	0
214	Male	28	Kalyan	Student	3			3	0	'7-8 hour	Unhealth	M.Pharm	Yes	11	2	No	1
222	Male	18	Surat	Student	4	0	6.7	5	0	'Less than	5 hours	'Class 12	Yes		4		1
232	Male	18	Visakhap	Student	17.001	0	6.21	3	0	'5-6 hour	Unhealth	'Class 12	Yes	4	2	No	1
239	Male	21	Jaipur	Student	1	0	7.25	1	0		Healthy	MCA	Yes	7	2	No	0
240	Female	31	Kalyan	Student	1	0	5.87	3	0	'7-8 hour	Healthy	PhD	No	8	4	Yes	0
242	Male	21	Surat	Student	1	0	8.04	3	0	'More tha	Healthy	MA		0	3	Yes	0

Figure 7: Converted to Raw dataset

Why this dataset was chosen:

This dataset was selected for its social impact, technical depth, and practical preprocessing challenges. It addresses a real-world problem detecting depression in students making the project both meaningful and applicable to academic institutions.

From a data science perspective, it offers a rich variety of data types (categorical, ordinal, continuous), enabling the application of techniques such as encoding, scaling, and SMOTE. The need to drop zero-only columns, handle missing values, and clean noisy data provided a realistic hands-on experience in building a robust data pipeline.

Moreover, the dataset supports a binary classification task, making it ideal for developing, evaluating, and deploying machine learning models. Its combination of ethical relevance and real-world data issues makes it the most compelling and educational choice among the candidates.

- Data structure: Flat table (processed into a big table)
- Target variable: Depression_Status (binary)
- Strengths: Real-world relevance, mixed data types, realistic preprocessing experience
- Limitation: Some survey fields lacked validity and were removed

In contrast, the Heart Attack dataset is small and clinically oriented, requiring domain expertise to interpret some of its features. While it is suitable for a binary classification task, it lacks the variety and contextual depth that make a project more engaging and applicable to broader audiences. Similarly, the Flight Price Prediction dataset is useful for regression modeling but focuses on commercial trends rather than social issues. While the dataset is rich in features, the preprocessing mainly revolves around text parsing and time conversion, which are technically useful but less insightful from a societal perspective.

For these reasons, the Student Depression dataset was deemed the most educational, practical, and impactful option. It allows for a complete pipeline that includes not only ingestion and storage design, but also complex preprocessing, modeling, deployment, and monitoring all while addressing a problem that matters.

Data Analytics Pipeline: Planning and Design

The data analytics pipeline was strategically planned to ensure efficient data flow from ingestion to model deployment. At the start, CSV data was loaded straight into the pipeline, but after noticing inconsistencies and empty values, Great Expectation was used before and after preprocessing to check the data. Next, MariaDB was setup for neat data inputs and Redis was added to help reduce time for training. Airflow DAGs were changed and improved over time to guarantee correct connections. FastAPI was used for APIs, MLflow introduced an easier way to reproduce experiments and data transformations were adjusted to better handle outliers.

CSV File → Ingestion (Airflow Task) → MariaDB Storage

↓

Great Expectations Validation (Before)

↓

Preprocessing (Python: Cleaning, Imputation, Feature Eng.)

↓

Great Expectations Validation (After)

↓

Redis Storage (CSV + Parquet)

↓

Model Training (MLflow + Scikit-learn + Random Forest)

↓

FastAPI Deployment → Real-time Predictions

Planning the Data Analytics Pipeline (Machine Learning Operations Pipeline)

A machine learning (ML) pipeline is a structured sequence of processes that transforms raw data into deployable predictive models. Each stage in the pipeline builds upon the previous one, and together, they form a complete, end-to-end system. This is where Machine Learning Operations (MLOps) comes into play. MLOps combines principles from DevOps and data engineering to automate and monitor the entire ML lifecycle. It enables automated retraining when new data arrives, integrates data validation and model tracking, and supports continuous deployment.
(Schelter et al., 2020).

A typical MLOps pipeline includes the following stages:

- Data Ingestion: Raw survey data is collected from different sources and stored in a centralized data lake or analytics database. This ensures consistency and traceability across the pipeline.

- Data Preprocessing: The ingested data is cleaned by removing missing or non-informative columns, normalizing numerical features, encoding categorical data, and creating engineered features suitable for training.

- Model Development: Preprocessed data is used to train multiple machine learning models. Tools like MLflow are used to track experiments, parameters, and performance metrics to identify the best model.

- Model Deployment: The approved model is containerized (e.g., using Docker) and deployed via an API, allowing integration into real-time systems for student depression prediction.

- Model Monitoring: The live model is continuously monitored for accuracy, drift, latency, and input quality. Alerts or triggers initiate retraining if model performance drops below threshold.

Tools for Python libraries management, containerization, pipeline creation and management

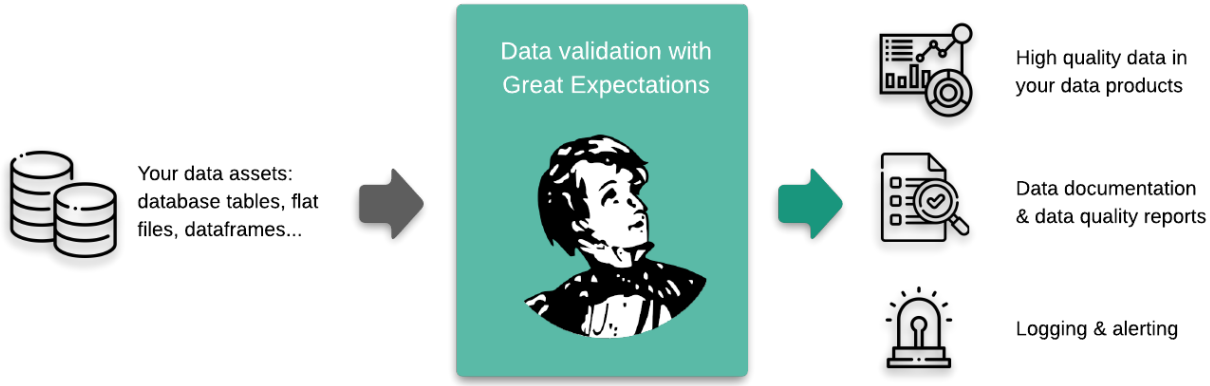
Several python packages and libraries were used for this coursework. Below is the description of them:

SN.	LIBRARIES	DESCRIPTION
1.	Docker	Docker is a widely used containerization tool that provides efficient, high-performance virtualization by running applications in private areas directly on the host computer's kernel (Loukidis- Andreou et al., 2018). In this coursework, Docker is used to run MariaDB, Redis, MLflow, and FastAPI in isolated containers, ensuring consistent environments, easy setup, and smooth deployment of the entire MLOps pipeline.

2.	Anaconda	Tools like conda-env-mod streamline this process by automating the creation and management of isolated Conda environments, which are essential for running different stages of a pipeline without package conflicts or version mismatches. By isolating packages per environment and supporting modular activation, conda-env-mod helps users maintain clean, reproducible pipelines, enabling the efficient development and deployment of data science and machine learning applications in HPC systems (Maji et al., 2020).
3.	Airflow	With Apache Airflow, we write, schedule and monitor workflows which gives us great control over the pipeline for handling data. Since scalability and modularity were built into Airflow, we created workflows using Python that are stored as Directed Acyclic Graphs (DAGs), making them more flexible for reuse.
4.	Redis	Redis stores all data in memory, thus providing the fastest possible time in reading or writing any data. It also has replication capabilities that can physically place the data nearer to the user for the lowest possible latency (IBM, 2021).
5.	Great Expectation	Great Expectation is an open-source tool, with its foundation in Python. Some of the most important features are data validation, profiling, and documentation of the entire project. We used this for automation and proper logging, along with alerts on top of data quality solutions. Great Expectations (GE) provides a handful of integrations (Airflow, Slack, Github Actions, etc.) that aid and enable these tasks (Tomáš Sobotík, 2021).

6.	MLFlow	MLflow is an open-source platform for machine learning capstone workflows. There are four primary components to it: MLflow Tracking, MLflow Models, MLflow projects and MLflow model registry. It influences how runs are organised, accessed and maintained. We did an experiment which has multiple runs and it helped to efficiently go through those runs and perform activities, for example, visualisation search and comparisons. Plus, experiments allow you to run artifacts and metadata for analysis by other tools (Munteanu, 2018).
7.	FastAPI	Built with Python and standard type hints, FastAPI is a web framework for quickly (with high performance) creating APIs. With it, projects move forward at an astounding 300% speed increase and with fewer (around 40%) errors committed by developers. Developed for ease of use, it has a good editor, doesn't require copying much code and includes interactive documentation (FastAPI, 2023).

What is Great Expectation about?



A brief about the usage of the tools:

The architecture was mainly oriented towards modularity, a feature of MLOps, the reproducibility, and scalability. Since the data on student depression is very sensitive and has a structured format, Python-based transformations were chosen to obtain more detailed processing of missing values,

outliers, and encoded data types than it was possible with SQL-based transformations. Docker containers have been employed to implement the idea of platform-independent deployment and make it easier for MariaDB, Redis, and MLflow to be deployed, therefore reducing environmental inconsistencies. In the case of systems where speed of accessing data was significantly crucial (e.g., ML training), the employment of Redis was made to help solve the problem. MLflow along with Airflow were considered for conducting continuous experiments and doing the automation. The tools being employed here are directly in line with the goals of the project of being ethical, fast, and reliable depression prediction.

Initial Pipeline Overview

The below diagram illustrates the MLOps pipeline from data ingestion to model deployment. A CSV dataset is first loaded into MariaDB ColumnStore running in a Docker container. The data is pre-processed, then serialized and stored in Redis. Finally, the trained model, along with parameters, metrics, and artifacts, is logged and stored in MLflow for future production use.

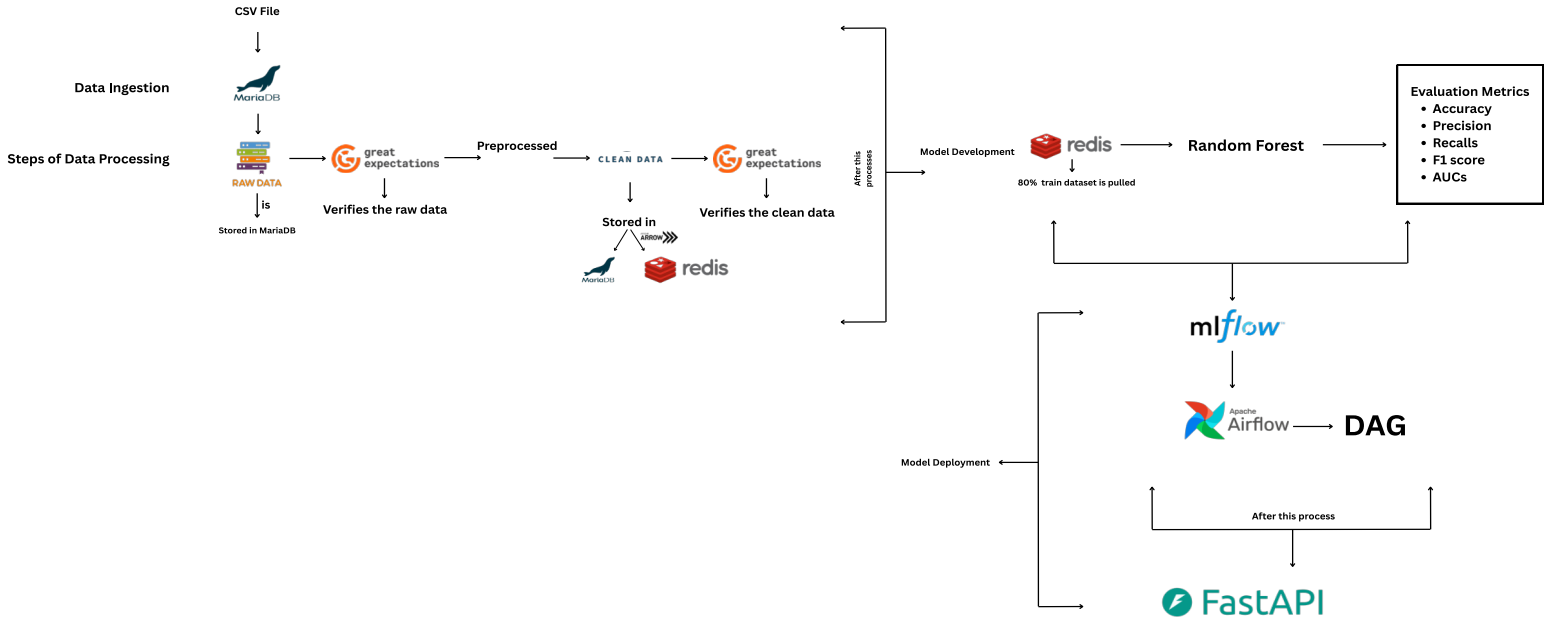


Figure 8: Main MLOPs Pipeline steps

Data Ingestion Stage

Tools Used in Code

Tool/Library	Logo	Purpose in Code
pandas		Used to read the CSV file into a DataFrame and to manipulate tabular data.
os		Checks whether the specified CSV file exists using os.path.exists().
SQLAlchemy		Creates an engine for the MariaDB connection and interacts with the database
pymysql		Used as a database driver in the SQLAlchemy connection string for MySQL/MariaDB

Table 1: Tools used in data ingestion stage

Retrieval Method	Data Source Location
Access a CSV file directly from the local system	Stored locally on the file system
Use wget to download the file via HTTP/HTTPS from a web server, then load with Pandas	Remote server
Fetch data from an API endpoint (likely JSON fornat) and load using Pandas	Remote source
Connect to a relational database or data lake to extract data	Remote (typically on another server or accessed over a network)

Table 2: Data storage

```python
def ingest_data():
    """
    Ingest CSV data into MariaDB database
    """
    print("🚀 Starting data ingestion...")

    # Step 1: Load CSV file
    csv_path = "/home/anush-pandey-24128423/Final_Dirty_Anush_Dataset.csv"

    if not os.path.exists(csv_path):
        raise FileNotFoundError(f"CSV file not found at {csv_path}")

    df = pd.read_csv(csv_path)
    print("📊 Preview of data:")
    print(df.head())
    print(f"Dataset shape: {df.shape}")

    # Step 2: Define MariaDB connection
    user = "anush"
    password = "Anush123"
    host = "localhost"
    port = 3306
    database = "student_depression"

    connection_string = f"mysql+pymysql://{user}:{password}@{host}:{port}/{database}"

    # Create SQLAlchemy engine
    engine = create_engine(connection_string)

    # Step 3: Ingest data into MariaDB
    table_name = "student_depression_table"

    # Replace the table if it exists
    df.to_sql(name=table_name, con=engine, if_exists="replace", index=False)

    print(f"✅ Data ingestion complete! Table '{table_name}' created in database '{database}'.")

    # Step 4: Confirm data ingestion and preview
    with engine.connect() as conn:
        result = conn.execute(text(f"SELECT COUNT(*) FROM {table_name}"))
        row_count = result.scalar()
        print(f"✅ Confirmed: {row_count} rows exist in '{table_name}' table.")

    df_check = pd.read_sql(f"SELECT * FROM {table_name} LIMIT 5", con=engine)
    print(f"🖱️ Sample rows from '{table_name}':")
    print(df_check)

    return f"Successfully ingested {row_count} rows into {table_name}"

```

Figure 8: Data ingestion stage

Automatically importing data from a CSV file and writing it to MariaDB takes just one function call with ingest_data(). It requires verifying the file, connecting to a database, adding data to a table and ensuring the file was ingested. Below is the step-by-step explanation:

1. Step 1: Load CSV File
    - Specifies the file path of the CSV file.
    - Uses os.path.exists() to check if the file exists. If not, it raises a FileNotFoundError.
    - Reads the CSV file into a DataFrame using Pandas.
    - Displays a preview of the data and its shape.

2. Step 2: Define MariaDB Connection
    - Sets the database connection credentials.
    - Constructs the connection string required for SQLAlchemy to connect to MariaDB using the PyMySQL driver.

3. Step 3: Ingest Data into MariaDB
    - Creates a SQLAlchemy engine that acts as the connection interface.
    - Uploads the DataFrame to the MariaDB database.
    - If a table with the same name already exists, it replaces it.

4. Step 4: Confirm Data Ingestion and Preview
    - Connects to the database and counts the number of rows in the new table to confirm successful ingestion.
    - Retrieves a sample of 5 rows from the ingested table to preview the data.
    - Returns a confirmation message with the number of rows ingested.

Input(s) and output(s)

For the data ingestion phase, we depend mainly on the raw dataset file for input. I have the Final_Dirty_Anush_Dataset.csv file on my local filesystem. Data must be arranged so that it can be used in one of the following ways: We can put the file directly into MariaDB ColumnStore as a table. Alternatively, upload the data into a Pandas DataFrame to clean it, check it and modify it and then transfer the whole DataFrame to MariaDB. Pandas handles missing values in the data, verifies column types and checks the consistency of the information as it loads the dataset. When the ingestion stage is successful, then the student data is placed into the MariaDB ColumnStore table named student_depression_table. As a result, the dataset is ready for use in preprocessing, validation, model building and analysis, helping the stakeholder see valuable insights in the data.

Figure 9: Data Ingestion Stage process

Before executing the Airflow DAG for the Student Depression Analysis pipeline, we ensure that the student_depression database exists in the MariaDB ColumnStore running inside the Docker container. If not, then we manually create it and confirm that the user has the necessary access

rights. This step is essential to avoid connection errors and ensure smooth execution of all database dependent tasks in the pipeline.

Data Preprocessing Pipeline

This preprocessing pipeline is a critical part of preparing raw data for machine learning applications. Without it, raw CSV data from the student_depression_table would be noisy, inconsistent, and unsuitable for modelling. The process begins by establishing a secure connection to a MariaDB database, using PyArrow for faster data loading. PyArrow accelerates performance by using a columnar memory format, ideal for large datasets and analytical operations (Statology, 2025).

Next, step is data cleaning, it removes duplicates, standardizes column names, and handles inconsistencies in key columns like Age and Sleep_Duration. Numeric columns are coerced to valid formats, and problematic features such as Work_Pressure are dropped. Missing values are handled thoughtfully like categorical fields are filled using the mode, and numerical columns with the mean. Then comes feature engineering, where new, insightful columns such as Age_Group, CGPA_Category, and binary risk factors are created. These enhance the model's ability to detect patterns and improve prediction accuracy (GeeksforGeeks, 2025a).

To make categorical data usable for algorithms, One-Hot Encoding is applied to multi-category features like City, Profession, and Degree. This prevents the model from assuming false ordinal relationships and ensures fairness (GeeksforGeeks, 2025b). Afterward, feature scaling is performed using MinMaxScaler, which normalizes the range of numerical variables. This step is vital because unscaled features could dominate others, resulting in biased learning and poor convergence (GeeksforGeeks, 2025c). Processed data is saved back to the database and then split into training (80%) and test sets (20%) using stratified sampling. The pipeline also integrates Redis to cache both CSV and Parquet formats of the datasets. This reduces I/O latency and supports real-time access, especially for model deployment or inference systems (Redis, 2025).

Software and Python Packages Required in Data Preprocessing Stage

The following tools and packages are essential for executing the pipeline:

- Python – 3.8+
- Pandas – data manipulation and cleaning
- NumPy – numerical operations
- SQLAlchemy – database connectivity

- PyMySQL – MySQL/MariaDB connector
- PyArrow – optimized data processing
- Scikit-learn – for preprocessing, train/test split, and scaling
- Redis – in-memory data caching
- LabelEncoder & MinMaxScaler – feature encoding and scaling utilities

```python

def preprocess_data():
    """
    Complete data preprocessing pipeline with PyArrow optimization
    """
    print("🚀 Starting data preprocessing...")

    # Database connection
    user = "anush"
    password = "Anush123"
    host = "localhost"
    port = 3306
    database = "student_depression"
    connection_string = f"mysql+pymysql://{user}:{password}@{host}:{port}/{database}"

    # Load data from database with PyArrow backend for better performance
    engine = create_engine(connection_string)
    query = "SELECT * FROM student_depression_table"

    try:
        # Try to use PyArrow backend for faster loading
        df = pd.read_sql(query, engine, dtype_backend='pyarrow')
        print("✅ Loaded data with PyArrow backend for optimal performance")
    except Exception as e:
        print(f"⚠ PyArrow backend failed ({e}), using default backend")
        df = pd.read_sql(query, engine)

    print(f"✅ Loaded {len(df)} rows from database")

    # Data exploration
    print("\n" + "="*50)
    print("📊 DATA EXPLORATION")
    print("="*50)
    print(f"Dataset shape: {df.shape}")
    print(f"Missing values:\n{df.isnull().sum()}")

    # Data cleaning
    print("\n" + "="*50)
    print("🧹 DATA CLEANING")
    print("="*50)

    df_clean = df.copy()

    # Remove duplicates
    initial_rows = len(df_clean)
    df_clean = df_clean.drop_duplicates()
    print(f"✅ Removed {initial_rows - len(df_clean)} duplicate rows")

    # Clean column names
    df_clean.columns = df_clean.columns.str.strip().str.replace(' ', '_').str.replace('?', '').str.replace('/', '_')

    # Clean Age column
    if 'Age' in df_clean.columns:
        df_clean['Age'] = pd.to_numeric(df_clean['Age'], errors='coerce')
        df_clean.loc[(df_clean['Age'] < 15) | (df_clean['Age'] > 80), 'Age'] = np.nan
        print("✅ Cleaned Age column")

    # Clean Sleep Duration
    if 'Sleep_Duration' in df_clean.columns:
        df_clean['Sleep_Duration'] = df_clean['Sleep_Duration'].astype(str).str.replace("'", "")
        df_clean['Sleep_Duration'] = df_clean['Sleep_Duration'].replace('nan', np.nan)
        print("✅ Cleaned Sleep Duration column")

    # Clean numeric columns
    numeric_cols = ['Academic_Pressure', 'Work_Pressure', 'CGPA', 'Study_Satisfaction',
                    'Job_Satisfaction', 'Work_Study_Hours', 'Financial_Stress']
    for col in numeric_cols:
        if col in df_clean.columns:
            df_clean[col] = pd.to_numeric(df_clean[col], errors='coerce')

```

```
458
459     # Drop problematic columns
460     cols_to_drop = []
461     if 'Work_Pressure' in df_clean.columns:
462         cols_to_drop.append('Work_Pressure')
463     if 'Job_Satisfaction' in df_clean.columns:
464         cols_to_drop.append('Job_Satisfaction')
465
466     if cols_to_drop:
467         df_clean = df_clean.drop(columns=cols_to_drop)
468         print(f"⚠ Dropped columns: {cols_to_drop}")
469
470     # Clean target variable
471     if 'Depression' in df_clean.columns:
472         df_clean['Depression'] = pd.to_numeric(df_clean['Depression'], errors='coerce')
473         df_clean['Depression'] = df_clean['Depression'].apply(
474     lambda x: 1 if pd.notna(x) and x > 0.5 else 0 if pd.notna(x) else np.nan
475 )
476
477         print("✅ Cleaned target variable (Depression)")
478
479     # Handle missing values
480     print("\n" + "="*50)
481     print("🔧 HANDLING MISSING VALUES")
482     print("="*50)
483
484     # Fill categorical columns with mode
485     categorical_cols = ['Gender', 'City', 'Profession', 'Sleep_Duration',
486                         'Dietary_Habits', 'Degree', 'Have_you_ever_had_suicidal_thoughts_',
487                         'Family_History_of_Mental_Illness']
488     for col in categorical_cols:
489         if col in df_clean.columns:
490             mode_val = df_clean[col].mode()
491             if not mode_val.empty:
492                 df_clean[col] = df_clean[col].fillna(mode_val[0])
493                 print(f"✅ Imputed {col} with mode: {mode_val[0]}")
494
```

```
495     # Fill numeric columns with mean
496     remaining_numeric_cols = ['Age', 'Academic_Pressure', 'CGPA',
497                               'Study_Satisfaction', 'Work_Study_Hours', 'Financial_Stress']
498     for col in remaining_numeric_cols:
499         if col in df_clean.columns:
500             mean_val = df_clean[col].mean()
501             if pd.notna(mean_val):
502                 df_clean[col] = df_clean[col].fillna(mean_val)
503                 print(f"✅ Imputed {col} with mean: {mean_val:.2f}")
504
```

```
504
505     # Feature engineering
506     print("\n" + "="*50)
507     print("⚙ FEATURE ENGINEERING")
508     print("="*50)
509
510     # Create age groups
511     if 'Age' in df_clean.columns:
512         df_clean['Age_Group'] = pd.cut(df_clean['Age'],
513                                        bins=[0, 20, 25, 30, 35, 100],
514                                        labels=['<20', '20-25', '25-30', '30-35', '35+'])
515         print("✅ Created Age_Group feature")
516
517     # Create CGPA categories
518     if 'CGPA' in df_clean.columns:
519         df_clean['CGPA_Category'] = pd.cut(df_clean['CGPA'],
520                                            bins=[0, 6, 7, 8, 10],
521                                            labels=['Low', 'Average', 'Good', 'Excellent'])
522         print("✅ Created CGPA_Category feature")
523
524     # Create binary risk factors
525     if 'Have_you_ever_had_suicidal_thoughts_' in df_clean.columns:
526         df_clean['Suicidal_Thoughts_Binary'] = (df_clean['Have_you_ever_had_suicidal_thoughts_'] == 'Yes').astype(int)
527         print("✅ Created Suicidal_Thoughts_Binary feature")
528
529     if 'Family_History_of_Mental_Illness' in df_clean.columns:
530         df_clean['Family_History_Binary'] = (df_clean['Family_History_of_Mental_Illness'] == 'Yes').astype(int)
531         print("✅ Created Family_History_Binary feature")
532
533     # Encode categorical variables
534     print("\n" + "="*50)
535     print("🔤 ENCODING CATEGORICAL VARIABLES")
536     print("="*50)
537
538     # Label encoding for binary categories
539     binary_cats = ['Gender', 'Have_you_ever_had_suicidal_thoughts_', 'Family_History_of_Mental_Illness']
540     for col in binary_cats:
541         if col in df_clean.columns:
542             le = LabelEncoder()
543             df_clean[col + '_encoded'] = le.fit_transform(df_clean[col].astype(str))
544             print(f"✅ Label encoded {col}")
545
```

```
545
546     # One-hot encoding for multi-category variables
547     onehot_cats = ['City', 'Profession', 'Degree', 'Dietary_Habits', 'Age_Group', 'CGPA_Category']
548     for col in onehot_cats:
549         if col in df_clean.columns:
550             dummies = pd.get_dummies(df_clean[col], prefix=col, drop_first=False).astype(int)
551             df_clean = pd.concat([df_clean, dummies], axis=1)
552             print(f"✅ One-hot encoded {col} ({len(dummies.columns)} new columns)")
553
554     # Drop original categorical columns
555     cols_to_drop = binary_cats + onehot_cats + ['Sleep_Duration']
556     cols_to_drop = [col for col in cols_to_drop if col in df_clean.columns]
557     df_clean = df_clean.drop(columns=cols_to_drop)
558
559     # Remove rows with missing target
560     if 'Depression' in df_clean.columns:
561         initial_rows = len(df_clean)
562         df_clean = df_clean.dropna(subset=['Depression'])
563         print(f"✅ Removed {initial_rows - len(df_clean)} rows with missing target variable")
564
565     # Save cleaned dataset
566     df_clean.to_sql(name="student_depression_table_cleaned", con=engine, if_exists="replace", index=False)
567     print("✅ Saved cleaned dataset to database")
568
569     # Feature scaling
570     print("\n" + "="*50)
571     print("🔪 FEATURE SCALING")
572     print("="*50)
573
574     df_scaled = df_clean.copy()
575     numeric_cols = df_scaled.select_dtypes(include=[np.number]).columns.tolist()
576     cols_to_exclude = ['Depression']
577     numeric_cols = [col for col in numeric_cols if col not in cols_to_exclude]
578
579     if numeric_cols:
580         scaler = MinMaxScaler()
581         df_scaled[numeric_cols] = scaler.fit_transform(df_scaled[numeric_cols])
582         print(f"✅ Scaled {len(numeric_cols)} numerical features")
583
583
584     # Save processed dataset
585     df_scaled.to_sql(name="student_depression_table_processed", con=engine, if_exists="replace", index=False)
586     print("✅ Saved processed dataset to database")
587
588     # Prepare train-test split and store in Redis
589     X = df_scaled.drop('Depression', axis=1)
590     y = df_scaled['Depression']
591
592     X_train, X_test, y_train, y_test = train_test_split(
593         X, y, test_size=0.2, random_state=42, stratify=y
594     )
```

```
595
596     print(f"\n🎯 DATA READY FOR MODELING:")
597     print(f"Training set: {X_train.shape}")
598     print(f"Test set: {X_test.shape}")
599
600     # Store in Redis with enhanced error handling
601     try:
602         redis_client = redis.Redis(host='localhost', port=6379, db=0)
603
604         # Store training data
605         train_df = X_train.copy()
606         train_df['Depression'] = y_train.values
607         train_csv = train_df.to_csv(index=False)
608         redis_client.set('student_depression_train', train_csv)
609
610         # Store test data
611         test_df = X_test.copy()
612         test_df['Depression'] = y_test.values
613         test_csv = test_df.to_csv(index=False)
614         redis_client.set('student_depression_test', test_csv)
615
616         # Also store as Parquet in Redis for faster retrieval (if PyArrow works)
617         try:
618             import io
619             train_buffer = io.BytesIO()
620             test_buffer = io.BytesIO()
621
622             train_df.to_parquet(train_buffer, engine='pyarrow', compression='snappy')
623             test_df.to_parquet(test_buffer, engine='pyarrow', compression='snappy')
624
625             redis_client.set('student_depression_train_parquet', train_buffer.getvalue())
626             redis_client.set('student_depression_test_parquet', test_buffer.getvalue())
627             print("✅ Stored train and test datasets in Redis (CSV + Parquet)")
628         except Exception as e:
629             print(f"⚠ Parquet Redis storage failed: {e}")
630             print("✅ Stored train and test datasets in Redis (CSV only)")
631
632     except Exception as e:
633         print(f"⚠ Redis storage failed: {e}")
634         raise Exception("Redis storage is required for the pipeline to work properly")
635
636     return f"Successfully preprocessed data. Train: {X_train.shape}, Test: {X_test.shape}"
637
```

Figure 10: Data Preprocessing Stage

Describing the student depression preprocessing pipeline shown in the fig:12:

FEATURES	DESCRIPTION
Data Connection and Loading	It uses SQLAlchemy to load data from a MariaDB database through the pipeline. PyArrow is taken advantage to work more quickly.
Data Exploration and Cleaning	We examine the shape and missing entries in the dataset. Any rows that duplicate are removed and every
	column name is cleaned by dropping out any incorrect characters.

Cleaning Specific Columns	Age that doesn't pass the check is removed and Sleep_Duration is made consistent. All the numeric columns are changed into formats for computation before being combined.
Dropping Irrelevant Or Corrupt Columns	Issues related to Work_Pressure and Job_Satisfaction mean the columns aren't used and are discarded.
Target Variable Transformation	A lambda function is applied to clean the Depression column and change it into binary classification data.
Missing Value Imputation	Data for categorical types comes from mode and for numerical types, the mean is used.
Feature Engineering	Age_Group, CGPA_Category and column-based representations are created.
Categorical Encoding	For binary columns, I apply label encoding and for each multi-category field, I use one-hot encoding.
Column Cleanup	After completion of encoding, the original categorical features are removed, while rows with empty target values are excluded.
Saving Cleaned Data	Once the dataset has been cleaned, it is saved again to MariaDB, so it is always on hand.
Feature Scaling	Scaling all the numeric features (other than the target) is performed using MinMaxScaler.
Train/Test Split	The data is separated into training which uses 80% and test which uses 20%, with the same distribution of the target.

REDIS STORAGE (CSV AND PARQUET)	All training data for the models is stored as CSV and Parquet files in Redis's memory so it's quickly accessible when needed.

Table 3: Describing the student depression preprocessing pipeline

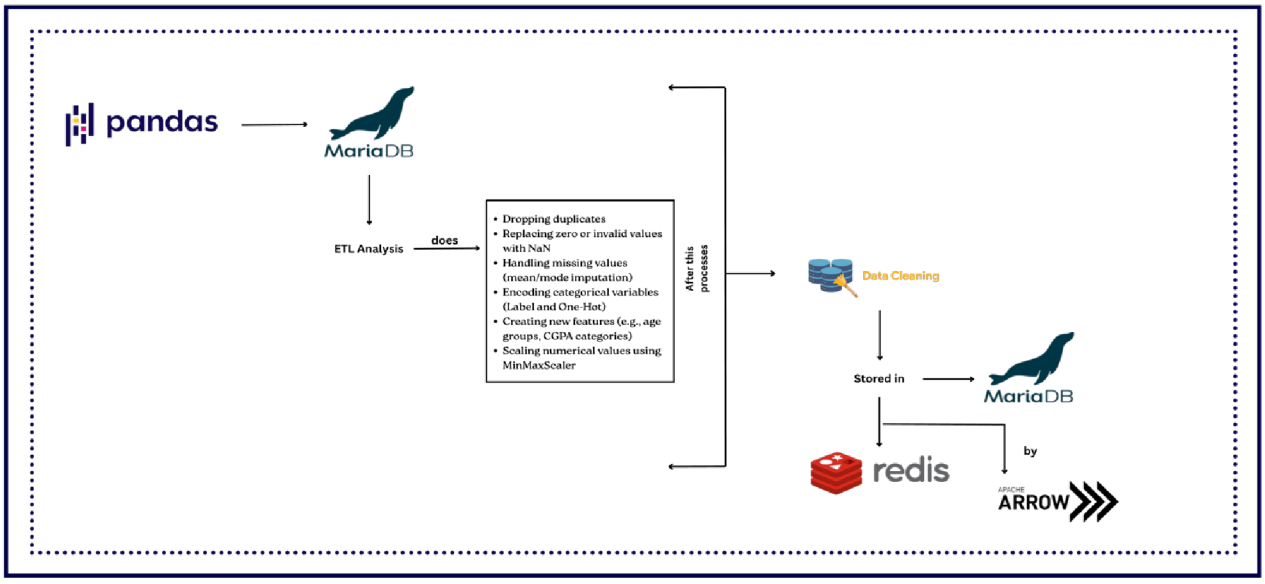


Figure 11: Data Preprocessing Stage

Input(s) and Output(s)

INPUT	OUTPUT
DATA SOURCE: STUDENT_DEPRESSION_TABLE FROM A MARIADB DATABASE	Cleaned dataset stored as: • student_depression_table_cleaned (in SQL) • student_depression_table_processed (scaled dataset in SQL)
FORMAT: RAW TABULAR DATA WITH POTENTIAL DUPLICATES, MISSING VALUES, AND MIXED DATA TYPES	Redis keys: • student_depression_train, student_depression_test (CSV format)

	• student_depression_train_parquet, student_depression_test_parquet (Parquet format)

Table 4: Input and output of data preprocessing

Model Building: Using Student Depression Dataset

This project uses the pre-processed set of student's answers about mental health, academic challenges and how they live daily. Because of its successful results and high-dimension handling, the RandomForestClassifier from Scikit-learn is selected for this task.

A 5-fold cross-validation is applied during training, so basic metrics such as the mean accuracy and standard deviation, can be measured. Following training, the model is given new data to see if it can handle problems, it has not trained on. Accuracy and predictions are estimated, and a history of the model's achievements is recorded. With MLflow, we can capture information about experiments. It includes the model, the resulting evaluation scores and the set hyperparameters (n_estimators, random_state). Reproducing results, versioning them and making comparisons in the future are made possible because of this.

Software and Python Packages Required added

MLflow: Logging, tracking, and storing models for reproducibility

Airflow: for automation

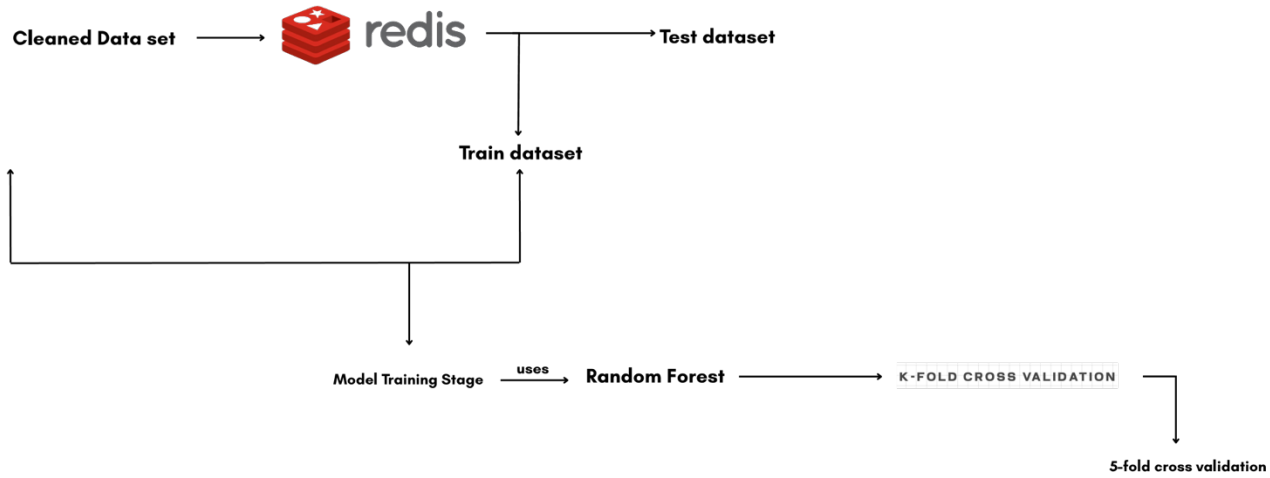


Figure 12: Model Training Diagram

```python

def train_model():
    """
    Train RandomForest model and log to MLflow with PyArrow optimization
    """
    print("🚀 Starting model training...")

    # MLflow setup
    mlruns_dir = "/home/anush-pandey-24128423/codeforstudentdepression/mlruns"
    os.makedirs(mlruns_dir, exist_ok=True)
    os.makedirs(os.path.join(mlruns_dir, ".trash"), exist_ok=True)
    mlflow.set_tracking_uri(f"file://{mlruns_dir}")

    # Try to load data from Redis first, fallback to database
    try:
        redis_client = redis.Redis(host='localhost', port=6379, db=0)

        # Try Parquet first for faster loading
        try:
            train_parquet = redis_client.get('student_depression_train_parquet')
            test_parquet = redis_client.get('student_depression_test_parquet')

            if train_parquet is not None and test_parquet is not None:
                train_df = pd.read_parquet(StringIO(train_parquet.decode('utf-8')), engine='pyarrow')
                test_df = pd.read_parquet(StringIO(test_parquet.decode('utf-8')), engine='pyarrow')
                print("✅ Loaded data from Redis (Parquet format)")
            else:
                raise ValueError("Parquet data not found in Redis")
        except Exception as e:
            print(f"⚠ Parquet loading failed: {e}")
            # Fallback to CSV
            train_csv = redis_client.get('student_depression_train')
            test_csv = redis_client.get('student_depression_test')
        y = df['Depression']

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

    print(f"Loaded training data: {X_train.shape}")
    print(f"Loaded test data: {X_test.shape}")

    # Initialize model
    model = RandomForestClassifier(random_state=42, n_estimators=100)

```

```
    # Cross-validation
    kf = KFold(n_splits=5, shuffle=True, random_state=42)
    cv_scores = cross_val_score(model, X_train, y_train, cv=kf, scoring='accuracy')

    print(f"Cross-Validation Accuracy Scores: {cv_scores}")
    print(f"Mean CV Accuracy: {cv_scores.mean():.4f} (+/- {cv_scores.std():.4f})")

    # Train final model
    model.fit(X_train, y_train)

    # Evaluate on test set
    y_pred = model.predict(X_test)
    test_accuracy = accuracy_score(y_test, y_pred)
    print(f"Test Set Accuracy: {test_accuracy:.4f}")

    # MLflow logging
    try:
        experiment_name = "Student_Depression_Classification"
        experiment = mlflow.get_experiment_by_name(experiment_name)
        if experiment is None:
            experiment_id = mlflow.create_experiment(experiment_name)
        else:
            experiment_id = experiment.experiment_id

        # End any active run
        if mlflow.active_run() is not None:
            mlflow.end_run()

        # Log to MLflow
        with mlflow.start_run(run_name="RandomForest_Depression_Model", experiment_id=experiment_id) as run:
            mlflow.sklearn.log_model(model, artifact_path="model")
            mlflow.log_metric("mean_cv_accuracy", cv_scores.mean())
            mlflow.log_metric("cv_accuracy_std", cv_scores.std())
            mlflow.log_metric("test_accuracy", test_accuracy)
            mlflow.log_param("n_estimators", 100)
            mlflow.log_param("random_state", 42)
            mlflow.log_param("pyarrow_enabled", True)

            # Log feature importance if possible
            if hasattr(model, 'feature_importances_'):
                feature_names = X_train.columns.tolist()
                feature_importance = pd.DataFrame({
                    'feature': feature_names,
                    'importance': model.feature_importances_
                }).sort_values('importance', ascending=False)

                # Save feature importance as artifact
                importance_path = "/tmp/feature_importance.csv"
                feature_importance.to_csv(importance_path, index=False)
                mlflow.log_artifact(importance_path, artifact_path="feature_analysis")
                print("✅ Logged feature importance analysis")

            print(f"✅ Model and metrics logged to MLflow, run ID: {run.info.run_id}")

    except Exception as e:
        print(f"⚠ MLflow logging failed: {e}")
        print("Model training completed without MLflow logging")

    return f"Model training completed. Test accuracy: {test_accuracy:.4f}"


# Try to import Great Expectations, fallback to basic validation if not available
try:
    import great_expectations as gx
    GX_AVAILABLE = True
    print("Great Expectations imported successfully")
except (ImportError, SyntaxError) as e:
    GX_AVAILABLE = False
    print(f"Great Expectations not available: {e}")

# DAG default arguments
default_args = {
    "owner": "Anush Pandey",
    "depends_on_past": False,
    "email": ["anush.pandey@example.com"],
    "email_on_failure": False,
    "email_on_retry": False,
    "retries": 1,
    "retry_delay": timedelta(minutes=5)
}
```

Figure 13: Model Training Stage

Below is a section-wise explanation of the model development and logging pipeline:

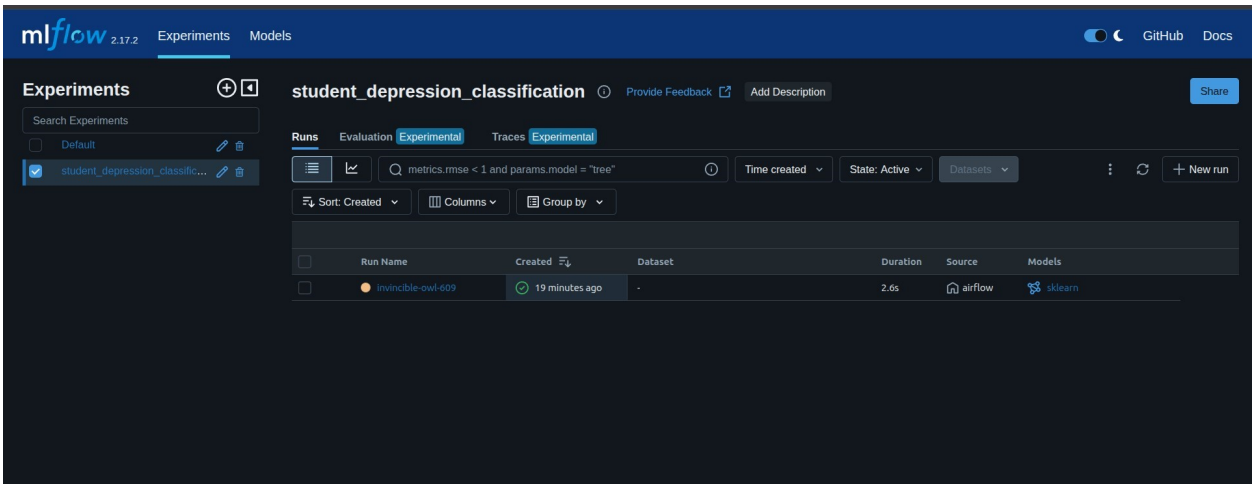
FEATURES	DESCRIPTIONS
MLFLOW SETUP	A directory for experiments is set up and managed locally by MLflow. As a result, all metrics, parameters and models are saved for others to reproduce the work later.
REDIS-BASED DATA LOADING (WITH FALLBACKS)	It makes a point to store Redis data in Parquet for superior speed. When not found, it will use CSV and when that fails, try MariaD.
TRAIN-TEST SPLIT	Both the feature set (X) and the label set (y) are divided into training and testing groups that help us measure the performance.
MODEL INITIALIZATION AND CROSS-VALIDATION	A Random Forest Classifier is built and accuracy statistics are obtained using 5-fold cross-validation.
FINAL MODEL TRAINING AND EVALUATION	The model goes through training with all data points and is tested on a smaller test data set.
MLFLOW EXPERIMENT LOGGING	All information about training can be found under the experiment named Student_Depression_Classification.
FEATURE IMPORTANCE LOGGING	The importance of each feature is calculated and written to a CSV file which is logged as an MLflow artifact.

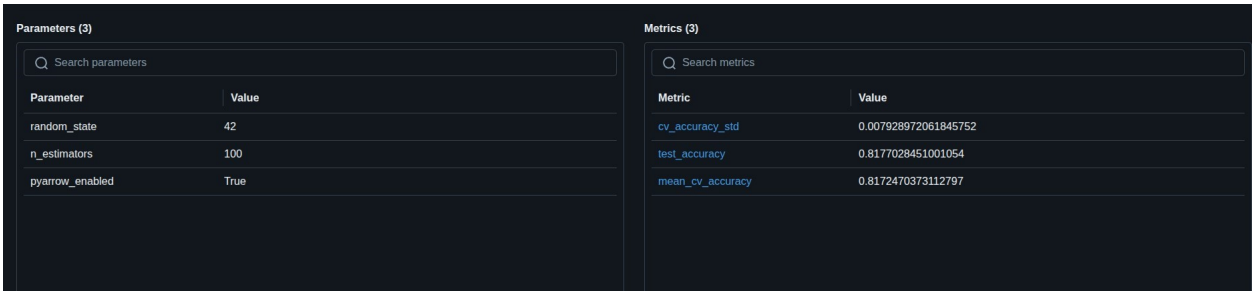
Table 5: model development and logging pipeline

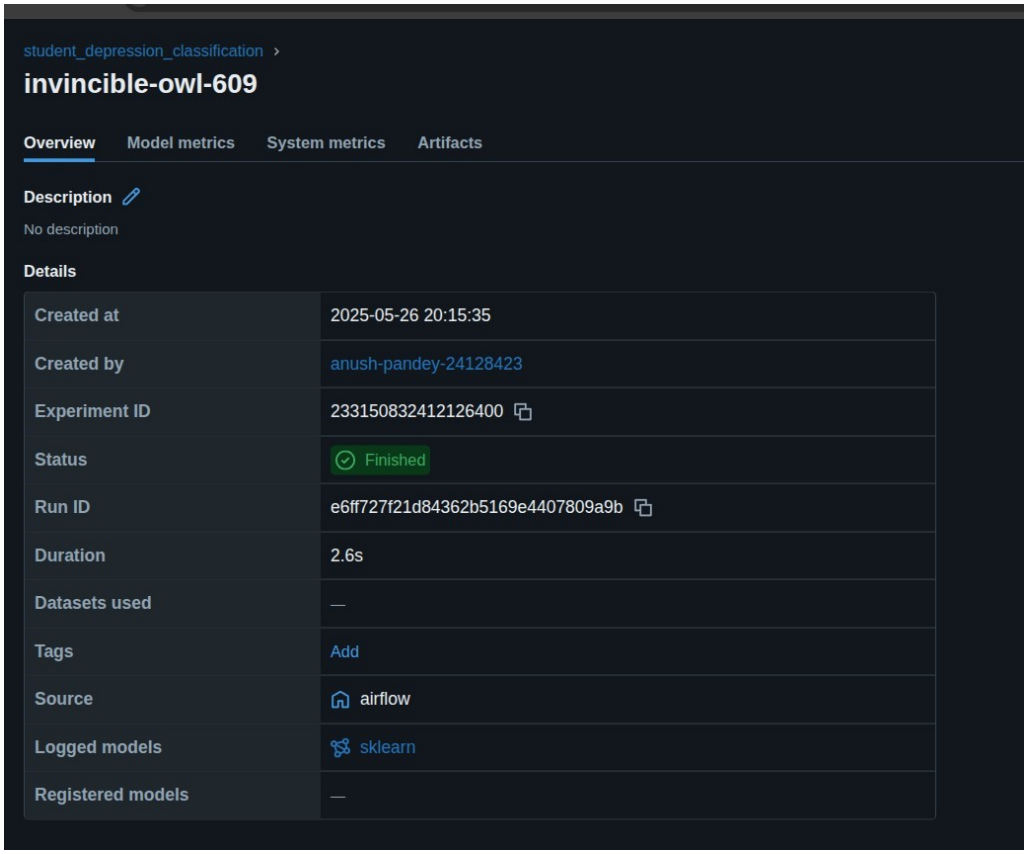
Input(s) and Output(s)

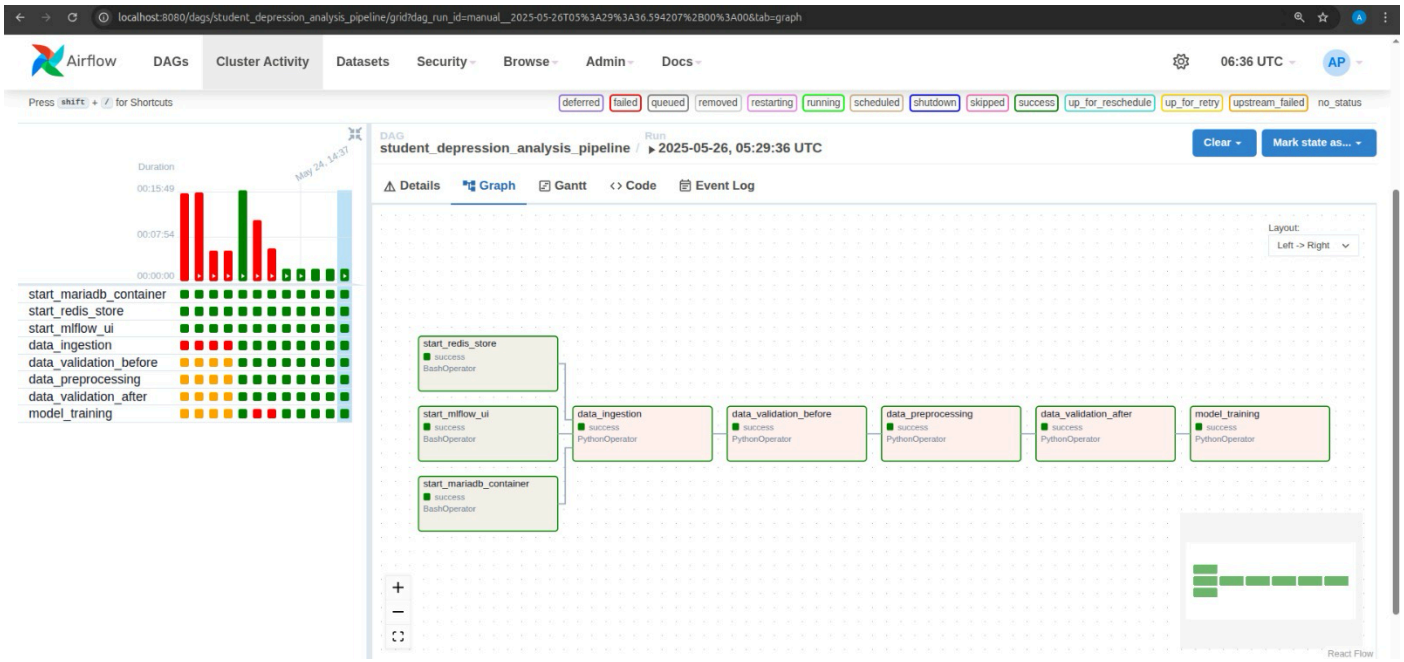
INPUT	OUTPUT
Pre-processed training and testing datasets redis keys: • Student_depression_train, student_depression_test . • If redis fails, fallback student_depression_table_processed mariadb.	Trained RandomForestClassifier model
	Logged model and metrics in MLflow under experiment Student_Depression_Classification
	Evaluation scores: cross-validation accuracy
	Parameters and metadata associated with the model

Table 6: input and output of model training


mlflow 2.17.2
Experiments
Models
GitHub
Docs
Experiments
Search Experiments
Default
student_depression_classific...
student_depression_classification
Provide Feedback
Add Description
Share
Runs
Evaluation Experimental
Traces Experimental
metrics.rmse < 1 and params.model = "tree"
Time created
State: Active
Datasets
New run
Sort: Created
Columns
Group by
Run Name
Created
Dataset
Duration
Source
Models
invincible-owl-609
19 minutes ago
-
2.6s
airflow
sklearn


Parameters (3)
Search parameters
Parameter
Value
random_state
42
n_estimators
100
pyarrow_enabled
True
Metrics (3)
Search metrics
Metric
Value
cv_accuracy_std
0.007928972061845752
test_accuracy
0.8177028451001054
mean_cv_accuracy
0.8172470373112797


student_depression_classification
invincible-owl-609
Overview
Model metrics
System metrics
Artifacts
Description
No description
Details
Created at
2025-05-26 20:15:35
Created by
anush-pandey-24128423
Experiment ID
233150832412126400
Status
Finished
Run ID
e6ff727f21d84362b5169e4407809a9b
Duration
2.6s
Datasets used
—
Tags
Add
Source
airflow
Logged models
sklearn
Registered models
—



Model Deployment

When the machine learning model is developed and validated, ensuring it is deployed into a production environment is the most important step next. To fulfil its task, the model must first be used in an actual system through deployment. For our project, we use FastAPI, a top web framework, to deploy the selected model as an asynchronous API tracked and stored with MLflow (FastAPI, n.d.). A client and server talk to each other in a REST API by using GET, POST, PUT or DELETE HTTP methods. All sorts of clients such as apps and web resources, are able to use the same RESTful API with IBM Cloud Functions (IBM, n.d.). To serve predictions, Uvicorn, an ASGI server, is used to run the FastAPI application (Uvicorn, n.d.). The API interacts with MLflow to retrieve the trained model and associated artifacts necessary for input data transformation.

Software and Python Packages Required

The following tools were used to implement model deployment:

- FastAPI – For creating the RESTful web service to handle prediction requests (FastAPI, n.d.).
- Uvicorn – Lightweight server for running asynchronous FastAPI apps

- MLflow – To retrieve the stored model and artifacts required for processing input (MLflow, n.d.).
- Pydantic – Used by FastAPI to validate and serialize input data structures (Pydantic, n.d.).
- Scikit-learn & Pandas – To manage model predictions and preprocessing operations

```python
from fastapi import FastAPI, BackgroundTasks
from pydantic import BaseModel
from typing import Optional
from pipeline import (
    ingest_data,
    run_gx_validation_before,
    preprocess_data,
    run_gx_validation_after,
    train_model
)
import mlflow.sklearn
import numpy as np
from mlflow.tracking import MlflowClient

app = FastAPI(title="Student Depression ML Pipeline API")

@app.get("/")
def root():
    return {"message": "Student Depression ML Pipeline API is running."}

@app.post("/ingest")
def trigger_ingest(background_tasks: BackgroundTasks):
    background_tasks.add_task(ingest_data)
    return {"status": "Data ingestion started in background."}

@app.post("/validate/before")
def trigger_validation_before(background_tasks: BackgroundTasks):
    background_tasks.add_task(run_gx_validation_before)
    return {"status": "Data validation (before preprocessing) started in background."}

@app.post("/preprocess")
def trigger_preprocess(background_tasks: BackgroundTasks):
    background_tasks.add_task(preprocess_data)
    return {"status": "Data preprocessing started in background."}

@app.post("/validate/after")
def trigger_validation_after(background_tasks: BackgroundTasks):
    background_tasks.add_task(run_gx_validation_after)
    return {"status": "Data validation (after preprocessing) started in background."}

@app.post("/train")
def trigger_train(background_tasks: BackgroundTasks):
    background_tasks.add_task(train_model)
    return {"status": "Model training started in background."}

# Define your input features here. Add/remove fields as per your model.
class PredictionInput(BaseModel):
    Gender: str
    Age: int
    City: str
    Profession: str
    Academic_Pressure: int
    CGPA: float
    Study_Satisfaction: int
    Sleep_Duration: str
    Dietary_Habits: str
    Degree: str
    Have_you_ever_had_suicidal_thoughts: str
    Work_Study_Hours: int
    Financial_Stress: int
    Family_History_of_Mental_Illness: str
```

```python
@app.post("/predict")
def predict(input_data: PredictionInput):
    # Example: simple label encoding for demonstration
    # In production, use the same preprocessing as in your training pipeline!
    gender_map = {"Male": 0, "Female": 1, "Other": 2}
    city_map = {"CityA": 0, "CityB": 1, "CityC": 2}  # Replace with your actual mapping
    profession_map = {"Student": 0, "Employed": 1, "Other": 2}  # Replace as needed
    degree_map = {"Bachelors": 0, "Masters": 1, "PhD": 2}  # Replace as needed
    diet_map = {"Veg": 0, "Non-Veg": 1, "Other": 2}  # Replace as needed
    suicidal_map = {"Yes": 1, "No": 0}
    family_history_map = {"Yes": 1, "No": 0}

    # Apply encoding (update mappings as per your training pipeline)
    gender = gender_map.get(input_data.Gender, 0)
    city = city_map.get(input_data.City, 0)
    profession = profession_map.get(input_data.Profession, 0)
    degree = degree_map.get(input_data.Degree, 0)
    diet = diet_map.get(input_data.Dietary_Habits, 0)
    suicidal = suicidal_map.get(input_data.Have_you_ever_had_suicidal_thoughts, 0)
    family_history = family_history_map.get(input_data.Family_History_of_Mental_Illness, 0)

    # Prepare input for prediction (order must match training)
    input_array = np.array([[
        gender,
        input_data.Age,
        city,
        profession,
        input_data.Academic_Pressure,
        input_data.CGPA,
        input_data.Study_Satisfaction,
        degree,
        diet,
        suicidal,
        input_data.Work_Study_Hours,
        input_data.Financial_Stress,
        family_history
    ]])

    # Load the latest model from the latest MLflow run (not Model Registry)
    mlflow.set_tracking_uri("http://localhost:5000")
    client = MlflowClient()
    experiment = client.get_experiment_by_name("student_depression_classification")
    if experiment is None:
        raise RuntimeError("No MLflow experiment found with name 'student_depression_classification'. Train a model first.")

    runs = client.search_runs(experiment.experiment_id, order_by=["attributes.start_time DESC"])
    if not runs:
        raise RuntimeError("No runs found in MLflow experiment 'student_depression_classification'. Train a model first.")

    latest_run_id = runs[0].info.run_id
    model_uri = f"runs:/{latest_run_id}/model"
    model = mlflow.sklearn.load_model(model_uri)

    # Predict
    prediction = model.predict(input_array)[0]
    return {"prediction": int(prediction)}
```

Figure 14: Model Deployment stage

Input(s) and Output(s)

INPUT	OUTPUT
A trained model retrieved from mlflow	A live REST API endpoint (e.g., /predict/student) accepting POST requests

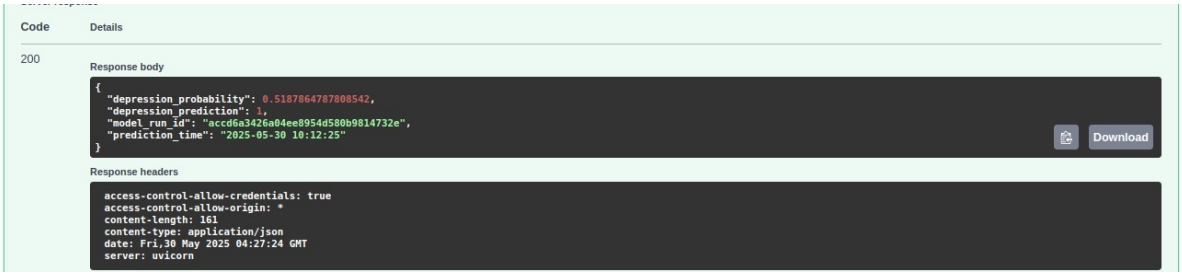
Artifacts (e.g., encoders or scalers) used during training to transform new data	A JSON response containing the original input features and the predicted outcome (depression prediction)
Json-formatted input data, matching the features used during model development	

Table 7: input and output of model deployment

API Design and Flow:

The REST API was structured with the following steps:

- Connect to MLflow and load the trained Random Forest model.
- Define a StudentInput data model using Pydantic, matching the feature schema.
- Create a StudentPrediction class for the API's output structure, including input fields and predicted result.
- Implement preprocessing functions that mirror those used during training to ensure input consistency.
- Create a POST endpoint using FastAPI at /predict/student, which receives input, processes it, runs the prediction, and returns a structured response.
- This approach ensures that the machine learning model is effectively productionized, accessible, and scalable across platforms.



Model Monitoring

Once the machine learning model is deployed within a production environment, it is crucial to continue monitoring it to guarantee accurate and consistent performance over time. This involves verifying the accuracy of predictions using metrics such as accuracy, precision, and F1 score, as well as system performance using latency and response times for API calls. In real-world application scenarios, a model's prediction might deteriorate after deployment. This is brought about by several reasons, such as emergence of new pattern data that do not closely match the training data, or insufficient generalization due to overfitting in model creation. Over time, the model will experience performance degradation, also known as model drift.

Refinement of the Pipeline

The initial pipeline focused mainly on data flow and stage inputs/outputs at a high level but lacked software engineering robustness. To improve, more emphasis will be placed on validating the data passing through each stage, ensuring integrity and reliability for a scalable, maintainable system.

Data Validation During Ingestion

Validation of data upon ingestion is a critical process in ensuring the quality and reliability of data being fed into the MLOps pipeline. Without this, there is a risk of feeding corrupted, incomplete, or inconsistent data into the system, which would lead to faulty analyses, incorrect model training, and eventually unpredictable conclusions.

In early releases of most pipelines, data ingestion typically proceeds without rigorous validation, simply relying on superficial checks like column counts or file format. This approach can carry faults farther. Having thorough validation during ingestion catches anomalies, missing fields, or schema mismatches in the beginning and prevents bad data from contaminating databases and ML workflows. Thus, robust ingestion validation guarantees the integrity of the entire analytics and modeling process, boosting system trustability and performance.

Incorporating Data Validation using Great Expectation into the Existing Pipeline

Tools like Great Expectations can be embedded to run validation checks, produce detailed reports, and halt pipeline execution if data anomalies are detected, thereby preventing errors from cascading downstream. This integration enhances pipeline robustness, reliability, and trustworthiness, enabling smoother model development and deployment. The data validation process will be completed entirely prior to the execution of the existing ingestion task and can be done after preprocessing too.

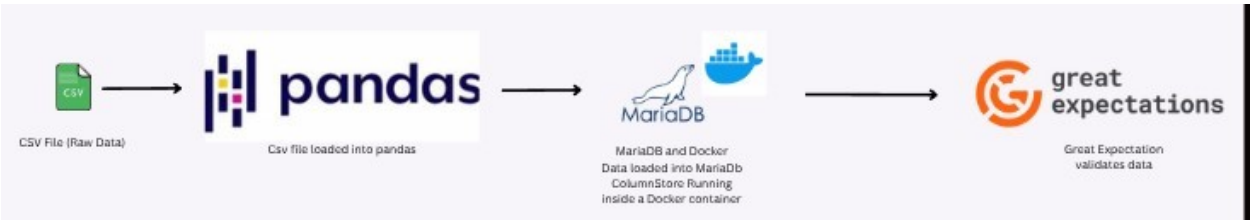


Figure 15: Incorporating Data Validation into the Existing Pipeline

Great Expectations of the Raw data

```python
def run_gx_validation_before():
    """
    Run Great Expectations validation on raw data before preprocessing
    """
    print("🔍 Starting data validation (before preprocessing)...")

    if not GX_AVAILABLE:
        print("⚠ Great Expectations not available, running basic validation...")
        # Load data for basic validation
        connection_string = "mysql+pymysql://anush:Anush123@localhost:3306/student_depression"
        engine = create_engine(connection_string)
        df = pd.read_sql("SELECT * FROM student_depression_table", engine)
        return basic_data_validation(df, "before")

    try:
        # Define MariaDB connection
        connection_string = "mysql+pymysql://anush:Anush123@localhost:3306/student_depression"
        # Batch Request
        batch_request = gx.core.batch.RuntimeBatchRequest(
            datasource_name="mariadb_datasource_before",
            data_connector_name="default_runtime_connector",
            data_asset_name="student_depression_data_asset",
            runtime_parameters={"query": f"SELECT * FROM {table_name}"},
            batch_identifiers={"default_identifier": "mariadb_batch_1"}
        )

        validator = context.get_validator(
            batch_request=batch_request,
            expectation_suite_name=suite_name
        )

        # Add basic expectations
        validator.expect_column_values_to_not_be_null("Gender")
        validator.expect_column_values_to_not_be_null("Age")
        validator.expect_column_values_to_not_be_null("Depression")
        validator.expect_column_values_to_be_between("Age", min_value=15, max_value=80)
        validator.expect_column_values_to_be_between("Depression", min_value=0, max_value=6)

        # Save expectations and validate
        validator.save_expectation_suite(discard_failed_expectations=False)
        validation_result = validator.validate()

        # Summary
        print("Raw Data Validation Summary")
        print("==========================")
        print(f"Success: {validation_result.success}")
        print(f"Evaluated: {validation_result.statistics['evaluated_expectations']}")
        print(f"Successful: {validation_result.statistics['successful_expectations']}")
        print(f"Failed: {validation_result.statistics['unsuccessful_expectations']}")

        return validation_result.success

    except Exception as e:
        print(f"⚠ Great Expectations validation failed: {e}")
        print("Falling back to basic validation...")
        # Load data for basic validation
        engine = create_engine(connection_string)
        df = pd.read_sql("SELECT * FROM student_depression_table", engine)
        return basic_data_validation(df, "before")
```

Figure 16: Great Expectation of Raw data

Below is the explanation of the raw data validation using Great Expectations as seen in the above figure:

FEATURES	DESCRIPTION
FUNCTION PURPOSE	run_gx_validation_before() checks that the data is correct before starting any Great Expectations preprocessing. When GX isn't present, validation switches to a basic method using Pandas.
FALLBACK FOR MISSING GX INSTALLATION	If GX be absent, the function obtains data straight from MariaDB and uses a simple validator.
CREATING A GX CONTEXT AND DATASOURCE	A context is made when GX is available. A MariaDB SQL data source is created by using a RuntimeBatchConnector within the GX framework.
CREATING THE EXPECTATION SUITE	The expectation suite called student_depression_raw_expectation_suite is either created or refreshed. It holds all the rules that govern validating data.
BATCH REQUEST SETUP	A runtime batch request is created to retrieve the raw data from MariaDB directly using SQL. This means validating data through flexible query execution.
DEFINING VALIDATION RULES	Basic expectations are added using the validator object: • Ensure columns like Gender, Age, and Depression are not null. • Confirm Age lies between 15 and 80. • Confirm Depression lies between 0 and 6.
EXECUTING AND LOGGING VALIDATION	All expectations are recorded and the system validates the data. Details on how many expectations were examined and which never succeeded are included in the summary.

FALLBACK TO BASIC VALIDATION	In case GX fails while doing its job, Pandas-based checks are run to confirm the dataset.

Table 8: raw data validation using Great Expectations

Great Expectation of Cleaned Dataset

```python
def run_gx_validation_after():
    """
    Run Great Expectations validation on processed data after preprocessing
    """
    print("🔍 Starting data validation (after preprocessing)...")

    connection_string = "mysql+pymysql://anush:Anush123@localhost:3306/student_depression"
    engine = create_engine(connection_string)

    try:
        df = pd.read_sql("SELECT * FROM student_depression_table_processed", engine)
    except Exception as e:
        print(f"⚠ Could not load processed data: {e}")
        return False

    if not GX_AVAILABLE:
        print("⚠ Great Expectations not available, running basic validation...")
        return basic_data_validation(df, "after")

    try:
        # Create Great Expectations context
        context = gx.get_context()

        # Add pandas datasource
        try:
            context.add_datasource(
                name="pandas_datasource_after",
                class_name="Datasource",
                execution_engine={"class_name": "PandasExecutionEngine"},
                data_connectors={
                    "default_runtime_data_connector": {
                        "class_name": "RuntimeDataConnector",
                        "batch_identifiers": ["default_identifier"],
                    }
                },
            )
        except Exception as e:
            print("Datasource might already exist:", e)

        # Create expectation suite
        suite_name = "student_depression_processed_expectation_suite"
        context.add_or_update_expectation_suite(expectation_suite_name=suite_name)

        # Prepare RuntimeBatchRequest
        batch_request = gx.core.batch.RuntimeBatchRequest(
            datasource_name="pandas_datasource_after",
            data_connector_name="default_runtime_data_connector",
            data_asset_name="student_depression_processed_asset",
            runtime_parameters={"batch_data": df},
            batch_identifiers={"default_identifier": "pandas_batch_processed"},
        )

        # Get validator
        validator = context.get_validator(
            batch_request=batch_request,
            expectation_suite_name=suite_name,
        )

        # Add basic expectations for processed data
        if 'Depression' in df.columns:
            validator.expect_column_values_to_not_be_null("Depression")
            validator.expect_column_values_to_be_in_set("Depression", [0, 1])
```

```
704
705         # Print results
706         print("\nProcessed Data Validation Summary")
707         print("==================================")
708         print(f"Success: {validation_result.success}")
709         print(f"Evaluated: {validation_result.statistics['evaluated_expectations']}")
710         print(f"Successful: {validation_result.statistics['successful_expectations']}")
711         print(f"Failed: {validation_result.statistics['unsuccessful_expectations']}")
712
713         return validation_result.success
714
715     except Exception as e:
716         print(f"⚠ Great Expectations validation failed: {e}")
717         print("Falling back to basic validation...")
718         return basic_data_validation(df, "after")
719
```

Figure 17: Great Expectation of Cleaned Dataset

FEATURES	DESCRIPTION
FUNCTION PURPOSE	The function run_gx_validation_after() is designed to validate the processed dataset after preprocessing is complete. It ensures that transformations did not introduce any integrity issues and that the target values confirm to expected types.
EXTRACT PROCESSED DATA	The function first loads the student_depression_table_processed dataset from MariaDB. If this fails, it returns False to indicate validation cannot proceed.
FALLBACK HANDLING	If the Great Expectations module is not installed, the function will use a simplified validation routine.
CREATE GREAT EXPECTATIONS CONTEXT	A GX context is initialized to manage the validation workflow and metadata such as suites and data sources.
DEFINE DATASOURCE FOR PANDAS EXECUTION	A PandasExecutionEngine is used instead of SQL for validating the in-memory processed DataFrame. The RuntimeDataConnector allows for dynamic data input at runtime.

CREATE EXPECTATION SUITE	An expectation suite named student_depression_processed_expectation_suite is created or updated. This suite will contain validation rules for the post-cleaned data.
DEFINE RUNTIMEBATCHREQUEST	A RuntimeBatchRequest is created to pass the cleaned df as input directly for validation under the registered datasource.
DEFINE POST-PROCESSING EXPECTATIONS	The function ensures that: • Depression column must not contain null values • Depression must only contain binary values (0 or 1)
VALIDATE AND OUTPUT RESULTS	The validation is executed, and a summary is printed that includes: • Whether the validation passed • How many expectations were evaluated • How many were successful or failed
EXCEPTION HANDLING	If any error occurs during GX validation, it falls back to simple validation.

Table 9: Clean data validation using Great Expectations

```
...
    Validation Summary
    ==================
    Success                 : True
    Evaluated Expectations  : 84
    Successful Expectations : 84
    Failed Expectations     : 0

    Detailed Expectation Results
    ============================

    Expectation 1: expect_column_values_to_not_be_null
    ------------------------------------------------------
      column                    : Gender_encoded
      batch_id                  : a38422c4706ca1a67365969f7a5fc420
      Success           : True


    Expectation 2: expect_column_values_to_be_in_set
    ------------------------------------------------------
      column                    : Gender_encoded
      value_set                 : [0, 1, 2]
      batch_id                  : a38422c4706ca1a67365969f7a5fc420
      Success           : True
    ...
      batch_id                  : a38422c4706ca1a67365969f7a5fc420
      Success           : True
```

Storage of the Data Within the Analytics System

Again, we are at the analytics database and the system for putting the student depression dataset into MariaDB ColumnStore. The goal is to structure the data optimally for consumption by the automated MLOps pipeline as well as consumers such as Data Scientists and Business Analysts.

The analytics database, or Data Warehouse (DW), is primarily a read-optimized environment to store business intelligence queries and to furnish data for auto-generated model building in the MLOps pipeline. MariaDB ColumnStore offers substantial advantages over native CSV files, including efficient storage structure, structured data access, and data integrity constraints.
For this data set, it was decided to take the One Big Table (OBT) approach, where all the relevant information is stored within one, fully denormalized table. The approach is specifically well-suited to the student depression data set for the following reasons:

- Dataset Size and Simplicity: The data set consists of only nine columns, which naturally limits duplicating data. The relatively small volume of data further supports the OBT approach without significant storage waste.
- Query Performance: OBT facilitates faster read operations by minimizing the use of expensive joins because all the data is in one table. This aligns with the pipeline's requirement for fast data access when training models and doing analytics.
- Ease of Maintenance: The fact that there is one table simplifies database management and reduces the complexity of ETL, which is a benefit for a system intended for automated processes.

Even though other data models such as dimensional schemas or completely normalized relational models are better suited for larger and more complex sets of data, they introduce unnecessary complexity in this case. Multidimensional models are typically applied to multidimensional analysis in large-scale BI systems but introduce additional tables (fact and dimension tables) and consequently more join complexity. Normalized relational models reduce data duplication but can reduce query performance due to frequent joins and complicate pipeline data extraction.

Because of the minimal layout and compact size of the student depression data, the One Big Table solution gives the best performance, simplicity, and maintainability trade-off for automatic model construction and user data consumption.

Having decided to use a One Big Table structure for storing the student depression dataset, it is important to clarify the data extraction and loading approach. Two common paradigms exist: Extract, Transform, Load (ETL) and Extract, Load, Transform (ELT). ELT loads raw data quickly into the analytics system, enabling stakeholders to perform transformations and queries on demand beneficial when rapid availability is prioritized, and transformation needs vary. ETL, on the other hand, transforms data before loading to ensure that only clean, standardized, and business-ready data is stored. This method is preferred when mandatory transformations such as data cleansing or removal of sensitive information are required prior to storage. For this project, the ETL approach was chosen, with minimal transformations primarily involving attribute renaming to fit the database schema. This ensures consistency and integrity within the One Big Table, supporting efficient querying and model development.

Data Analysis and Insight

1. Statistical Data Types:

Data	Data Types
age	**Ratio**
Academic Pressure	**Interval**
Work Pressure	**Interval**

CGPA	**Ratio**
Study Satisfaction	**Interval**
Job Satisfaction	**Interval**
Work/Study Hours	**Ratio**
Depression	**Interval**
Gender	**Nominal**
City	**Nominal**
Profession	**Nominal**
Sleep Duration	**Ordinal**
Degree	**Nominal**
Have you ever had suicidal thoughts?	**Nominal**
Financial Stress	**Ordinal**
Family History of Mental Illness	**Nominal**

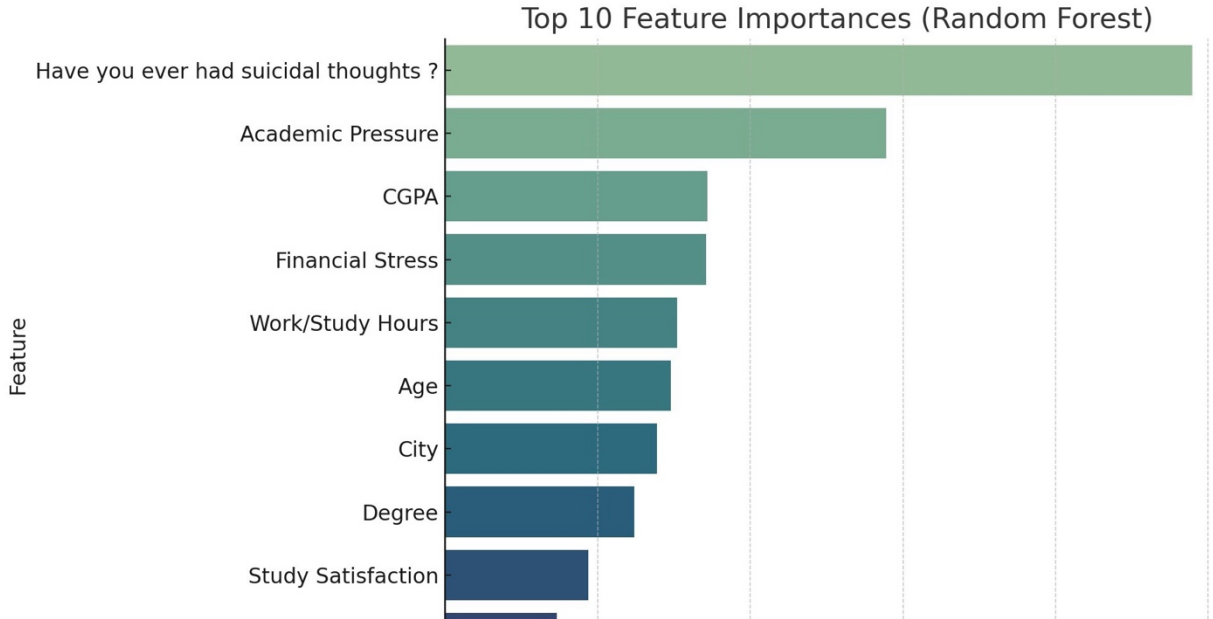
2. Research Questions:

- **How do the academic workload, concern over finances, and a family background of mental disorders impact the chances of depression for students?**
- **What personal and behavioral characteristics like age, sleep, and nutrition are most associated with depression for university students?**
- **Which combination of characteristics like suicidal ideation, CPGA, and city of residence best identify the highest risk of being depressed according to a Random Forest Classifier?**

3. Outliers & Missing Values:

   There are some obviously absent values in the columns of Age, Sleep Duration, CGPA, Financial Stress, and Depression. Data in the Sleep Duration column is given in string format, and the format is also inconsistent (e.g., different types of quotes, various ranges) hence it is necessary to make them standardized. The CGPA data has an extremely wide range and therefore it is beyond academic ranges (e.g., values higher than 10 normal):- outlier checks are required. Non-numeric categories in Sleep Duration need to be divided into categories or have their text fields encoded. A few of the Financial Stress values are captured as strings rather than as numeric.

4. Key Relationship: Following the initial observations, depression appears to be caused by higher academic pressure and a suicidal thought. This can be represented by feature importance visualization:



5. Distribution Observed: Heatmaps and visual plots were used to analyze the raw dataset and notice where important attributes appeared most. At this point, we check that all variables are clear for modeling and note down any necessary changes.

- Age: Graphically displayed the right skew with the modal age range being early twenties. There are a few outliers at the younger end (e.g., 17) suggesting some undergraduates might start quite early.

- CGPA: There seems to be a normal distribution albeit with some peaks in the middle range (7 – 8) and some outliers on the higher side (10+) most likely because of lenient marking schemes or clerical errors in inputting data.

- Academic Pressure and Work Pressure: Both show the same bimodal or multimodal pattern with most participants bundled around moderate and high-pressure levels which could be attributed to differences in academic year or workload.

- Sleep Duration: Categorical ordinal variable with ranges that are non-numeric (e.g., 4-6 hrs, 6-8 hrs). This trend showcases variety in lifestyle and may be associated with variables of mental wellness.

- Depression Score: Presents a distribution skewed to the left but has a longer tail on the right. There is an overall lack of value in the mid to lower range but greater value at the upper end which represents more severe symptoms, indicating that only a few students may be suffering from sizeable symptoms.

- Binary Variables (e.g., Suicidal Thoughts, Financial Stress): There is a strong class imbalance in the dataset as the majority of records skew to “No” which may impact the effectiveness of the model’s performance unless adjusted for this imbalance.

K-Anonymity is a privacy that guarantees a record in a dataset cannot be distinguished from a set of ‘k’ records. This is done through quasi-identifiers such as age, gender, or place. By generalization and suppression of these attributes, no single individual’s information can be pinpointed. In the deals with predicting student depression, sensitive attributes such as “City” and “Suicidal Thoughts” pose a threat of re-identification when integrated with outside demographic information. Applying k-anonymity, for example by super setting cities into larger regions or changing ages to ranges, protects identities without undermining analytical value.

K-anonymity on its own is effective but has disadvantages: it avoids attribute disclosure without prevention and many unique outliers will be problematic. In practice, it is paired with other methods such as l-diversity or differential privacy to ensure adequate protection.

A good example of using k-anonymity is the HIPAA Safe Harbor method which works by hiding or reducing certain personal information such as ZIP code and birthdate in patient records. As a result, sharing the datasets does not cause re-identification. Similar to that, changing “City” into regional groups and “Age” into ranges will not reveal student information and still make use of the data.

Evaluating the Process of the Project Pipeline:

- It’s important to handle the prediction of depression using suicidal thoughts carefully. Wrong use of assistive technology could result in students being treated differently.
- Even when the data is anonymized, showing the data in a real setting would need consent and to follow the DPA.
- Role-based access control, anonymization and audit logging are important improvements that are needed. It could be used with the results of a model to keep personal patterns protected. Also, explaining results with SHAP would eliminate ambiguous or insoluble issues linked to black-box models.

Privacy, Security, and Ethics

Privacy Concerns: Fields such as City, Degree, and Suicidal thoughts may be sensitive and require anonymization procedures to avoid re-identification by the other demographic variables. K-anonymity techniques or generalization to broader categories may be required.

Security Measures: Redis should store only the processed, non-identifiable records. Enforce limiting read/write permissions for MariaDB access. Sensitive fields (like Depression, Suicidal Thoughts) must be encrypted or hashed when exporting or sharing. API endpoints (FastAPI) should implement access tokens or OAuth.

Ethical Considerations:

- These interpretations bear real-life consequences. Misclassification could be stigmatizing or could otherwise worsen abandonment of students already at risk.

- Should this model be deployed into live systems, there has to be a provision for seeking consent from the participants.

- Whatever bias occurs through data collection (i.e., having more responses from the urban settlements or from certain genders), it has to be disclosed, and then must be dealt with by weighting measures or stratification sampling methods.

Some of the actionable recommendations are:

- Encryption: Make sure to encrypt critical fields such as “Depression,” “Suicidal Thoughts,” and “Financial Stress” while they’re stored in MariaDB. You can use AES functions or opt for field-level encryption to keep this data safe.

- Anonymization: To avoid any chance of indirect identification, swap out city names for region codes or anonymized IDs. You can also use techniques like aggregation or generalization, such as grouping ages into bands or clustering professions.

- Access Control:
    - Redis: Keep access secure by using strong password authentication (require pass) and setting up firewall rules (bind 127.0.0.1).
    - MariaDB: Implement role-based access control (RBAC) to ensure that users only have SELECT access for analysis purposes.
    - FastAPI: Incorporate token-based authentication methods like JWT or OAuth2, enforce HTTPS for secure connections.

Visualization and Explanation

The visualizations were created directly on the raw data extracted from MariaDB, without any advanced preprocessing or feature selection. This was a crucial step to understand the initial structure, distribution, and relationships of the dataset. Raw data visualization helps in identifying missing values, outliers, and significant trends at an early point, facilitating knowledgeable decisions during cleaning the data and modeling. It provides a clear picture of how various factors like age, academic pressure, and suicidal tendencies are related to depression, which acts as the foundation stone in building an efficient prediction model. For this all the visualization part data is pulled from MariaDB.

Heat Map Visualization

Correlation heatmap portrays the correlation between different features of the dataset using color values. It is useful in identifying which variables are negatively or positively correlated and to what extent each feature correlates with depression. Here, it showed that suicidal ideation, financial hardship, and family history of mental illness are moderately positively correlated with depression. This is important in the sense that it guides feature selection, avoids duplicity, and improves the efficiency and accuracy of the model.

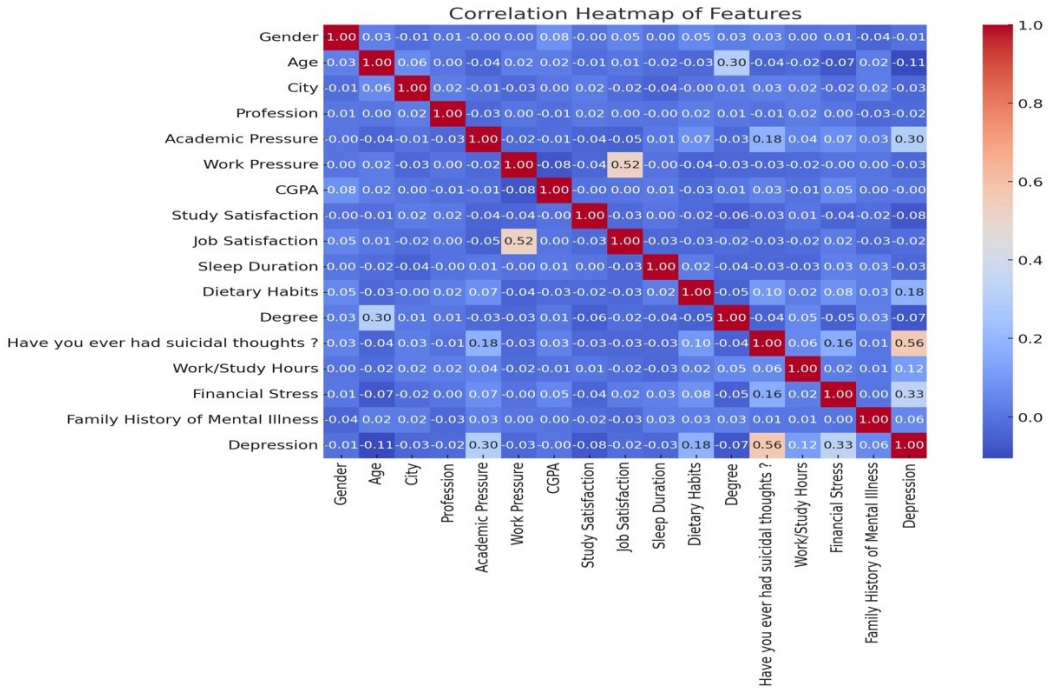


Figure 18: Correlation Heatmap

Distribution Of Age

The age distribution histogram shows the distribution of members of the dataset across different age brackets. It shows that most people fall into the 20 to 35 years bracket, which is consistent with the project's target group being students and young adults. The chart is important in that it confirms that the dataset aligns with the intended population as well as any peculiar outliers that might influence the model's performance.

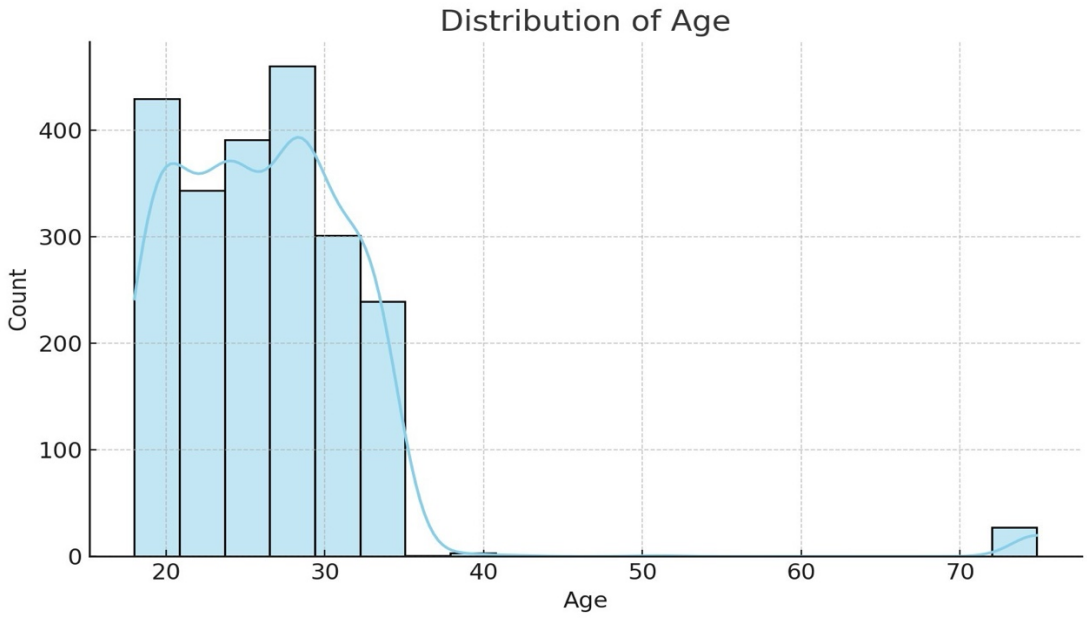


Figure 19: Visualization of distribution of age

Importance of Features (Random Forest)

The chart of feature importance illustrates the most impactful attributes in predicting depression with regard to the Random Forest model. Suicidal ideation, academic stress, and financial anxiety were the most dominant features. This visualization is beneficial for understanding why the model was built the way it was and assists in concentrating the interventions to the most important risk factors.

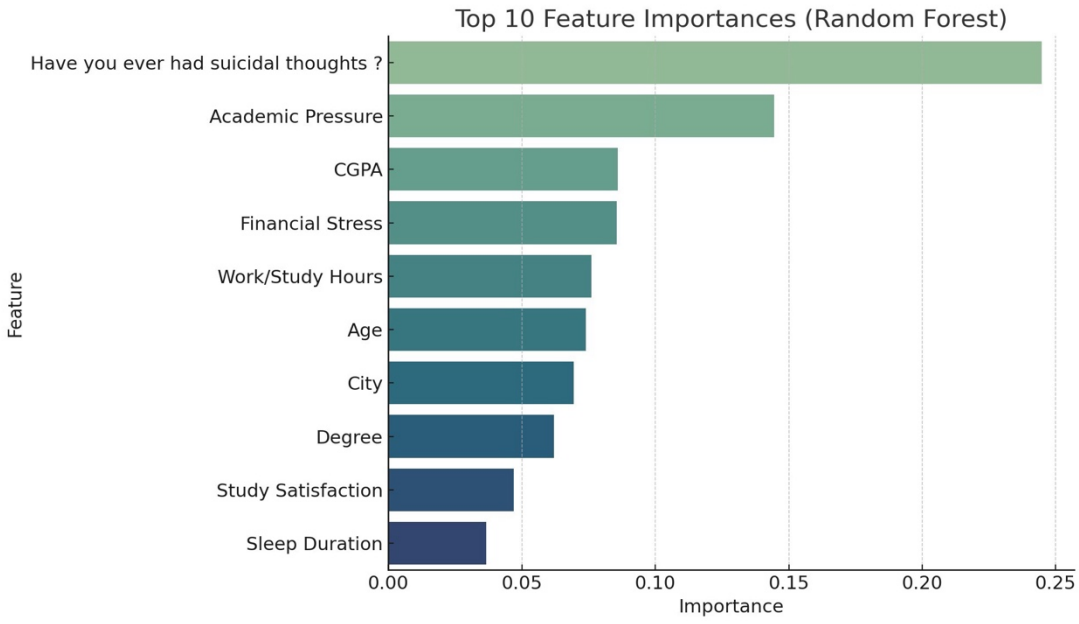


Figure 20: Importance of Features (Random Forest)

Appendix A

Setup and Configuration

The pipeline for data analytics is implemented on an Ubuntu 20.04 LTS due to its more recent data science support. Ubuntu was installed managed and control the environment for reproducibility purposes. Backups and cloning of the system are simplified and can be used to rapidly recover or replicate environments. This virtualization further separates the host and the pipeline which reduces conflicts and improves reliability during development and deployment.

Installation of Docker:

```
(student_depression) anush-pandey-24128423@AsusZenbook:~$ docker version
Client: Docker Engine - Community
 Version:           28.1.1
 API version:       1.49
 Go version:        go1.23.8
 Git commit:        4eba377
 Built:             Fri Apr 18 09:52:14 2025
 OS/Arch:           linux/amd64
 Context:           default

Server: Docker Engine - Community
 Engine:
  Version:          28.1.1
  API version:      1.49 (minimum version 1.24)
  Go version:       go1.23.8
  Git commit:       01f442b
  Built:            Fri Apr 18 09:52:14 2025
  OS/Arch:          linux/amd64
  Experimental:     false
 containerd:
  Version:          1.7.27
  GitCommit:        05044ec0a9a75232cad458027ca83437aae3f4da
 runc:
  Version:          1.2.5
  GitCommit:        v1.2.5-0-g59923ef
 docker-init:
  Version:          0.19.0
  GitCommit:        de40ad0
```

```
● docker.service - Docker Application Container Engine
     Loaded: loaded (/usr/lib/systemd/system/docker.service; enabled; preset: enabled)
     Active: active (running) since Sat 2025-05-24 14:59:37 +0545; 2 days ago
TriggeredBy: ● docker.socket
       Docs: https://docs.docker.com
   Main PID: 2969 (dockerd)
      Tasks: 18
     Memory: 37.8M (peak: 100.1M swap: 9.3M swap peak: 15.7M)
        CPU: 19.118s
     CGroup: /system.slice/docker.service
             └─2969 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock
```

MariaDB setup:

```
(student_depression) anush-pandey-24128423@AsusZenbook:~$ docker create --name
mcs_container -p 3306:3306 mariadb/columnstore
```

The container name "/mcs_container" is created and it is using container "59e5f07506b4785b99ca725fe2ac0e6371912224ff0b4bc6142e9a1336fcbf1d

```
(student_depression) anush-pandey-24128423@AsusZenbook:~$ docker ps -a | grep maria
59e5f07506b4   mariadb/columnstore   "/usr/bin/tini -- do…"   3 days ago   Exited (255) 2 days ago
 mcs_container
(student_depression) anush-pandey-24128423@AsusZenbook:~$
```

Inspecting the created container in MariaDB,

```
(student_depression) anush-pandey-24128423@AsusZenbook:~$ docker inspect mcs_container
[
    {
        "Id": "59e5f07506b4785b99ca725fe2ac0e6371912224ff0b4bc6142e9a1336fcbf1d",
        "Created": "2025-05-23T08:01:46.683227101Z",
        "Path": "/usr/bin/tini",
        "Args": [
            "--",
            "docker-entrypoint.sh",
            "start-services"
        ],
```

Logging into MariaDB,

```
(student_depression) anush-pandey-24128423@AsusZenbook:~$ mariadb -u root -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 81
Server version: 10.11.11-MariaDB-0ubuntu0.24.04.2 Ubuntu 24.04

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
```

Viewing the databases,

```
docker inspect mcs_container' at line 1
MariaDB [(none)]> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| student_depression |
| sys                |
+--------------------+
5 rows in set (0.013 sec)

MariaDB [(none)]>
```

```
MariaDB [(none)]> USE student_depression;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
MariaDB [student_depression]>
```

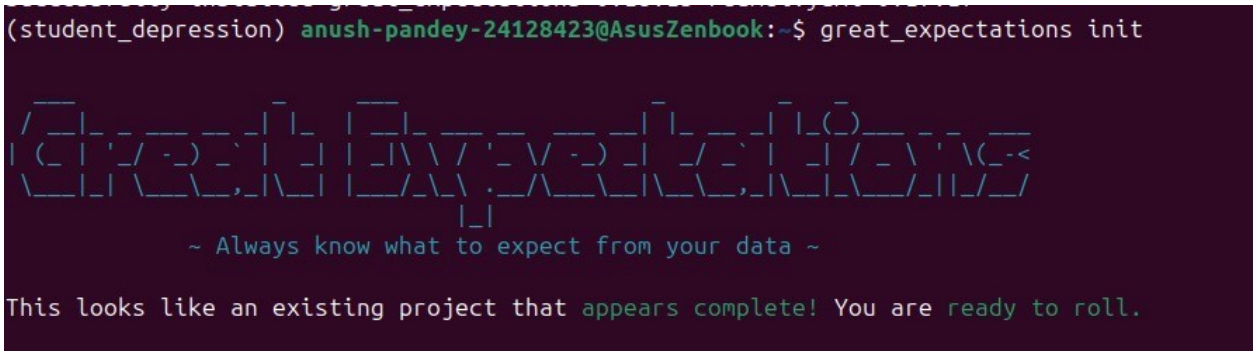
Redis setup:

```
(student_depression) anush-pandey-24128423@AsusZenbook:~$ sudo docker pull redis
[sudo] password for anush-pandey-24128423:
Using default tag: latest
latest: Pulling from library/redis
Digest: sha256:b3ad79880c88e302deb5e0fed6cee3e90c0031eb90cd936b01ef2f83ff5b3ff2
Status: Image is up to date for redis:latest
docker.io/library/redis:latest
(student_depression) anush-pandey-24128423@AsusZenbook:~$
```

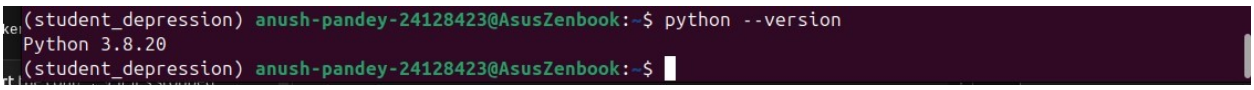
```
(student_depression) anush-pandey-24128423@AsusZenbook:~$ sudo docker create --name redis_store -p 6379:6379 r
edis 1dd4149f39477383e9b9a9367b773e0f7a60f26d8da61b136b2c02a1fcbf9d9c
```

```
be able to reuse that name.
(student_depression) anush-pandey-24128423@AsusZenbook:~$ docker ps -a | grep "redis"
1dd4149f3947   redis                "docker-entrypoint.s…"   3 days ago   Exited (255) 2 days ago
 redis_store
(student_depression) anush-pandey-24128423@AsusZenbook:~$
```

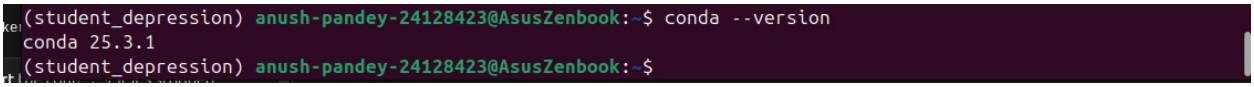
Great Expectation setup:



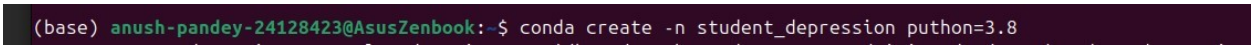
The installed python version,

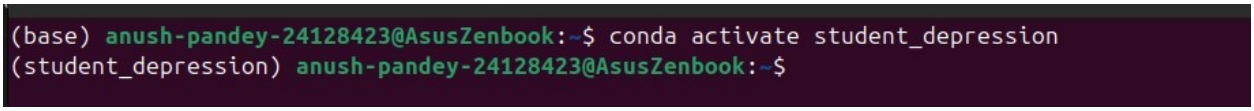


The installed conda version,



Creating a conda environment,



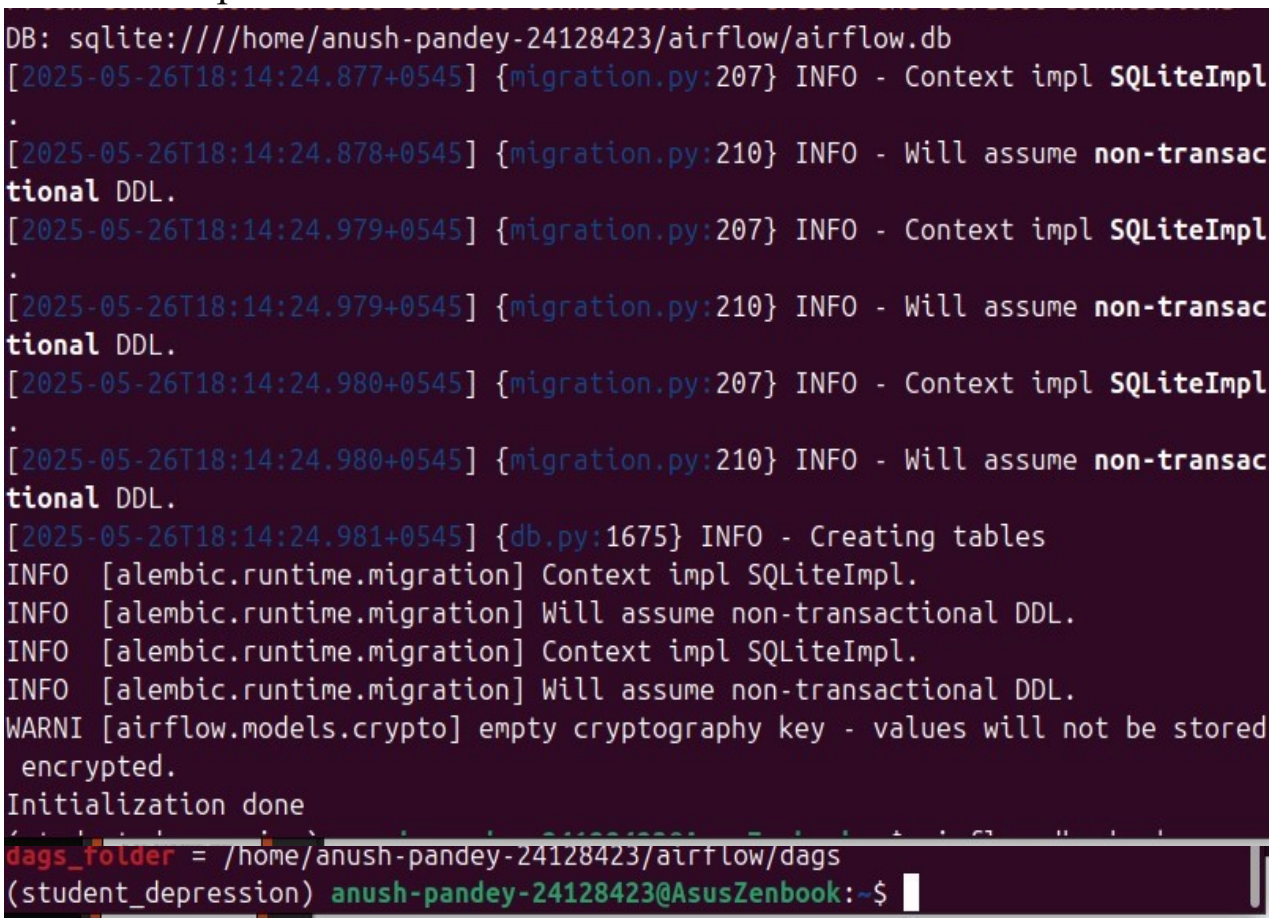


```
(student_depression) anush-pandey-24128423@AsusZenbook:~$ conda install -c anaconda pandas sqlalchemy pymysql
scikit-learn redis
Channels:
 - anaconda
 - conda-forge
 - https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free
 - https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main
 - defaults
Platform: linux-64
Collecting package metadata (repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: /home/anush-pandey-24128423/miniconda3/envs/student_depression

  added / updated specs:
    - pandas
    - pymysql
    - redis
    - scikit-learn
    - sqlalchemy
```

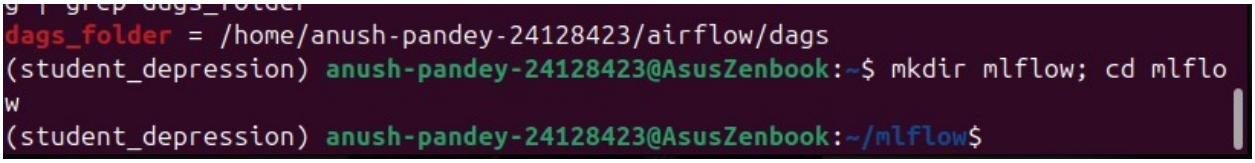
```
The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    redis-5.0.3                |       h7b6447c_0        10.6 MB  anaconda
    sqlalchemy-2.0.34          |   py38h00e1ef3_0         6.0 MB  anaconda
    ------------------------------------------------------------
                                           Total:        16.6 MB

The following NEW packages will be INSTALLED:

  redis              anaconda/linux-64::redis-5.0.3-h7b6447c_0
  sqlalchemy         anaconda/linux-64::sqlalchemy-2.0.34-py38h00e1ef3_0


Proceed ([y]/n)? y


Downloading and Extracting Packages:

Preparing transaction: done
Verifying transaction: done
Executing transaction: done
```

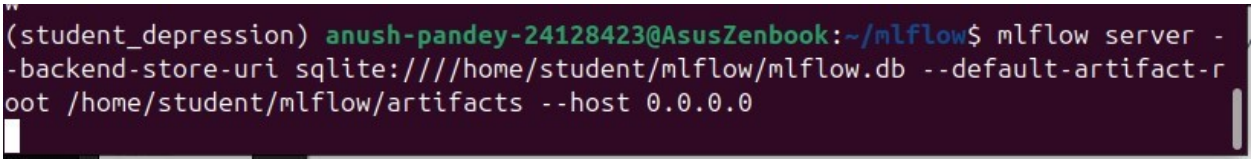
```
(student_depression) anush-pandey-24128423@AsusZenbook:~$ conda install -c conda-forge airflow mlflow fasapi u
vicorn
Channels:
 - conda-forge
 - https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free
 - https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main
 - defaults
Platform: linux-64
Collecting package metadata (repodata.json): done
```

Airflow setup:



Mlflow setup:





```
artifacts  mlflow.db  mlruns
(student_depression) anush-pandey-24128423@AsusZenbook:~/mlflow$ ls -l
total 232
drwxrwxr-x 2 anush-pandey-24128423 anush-pandey-24128423   4096 May 26 18:31 art
ifacts
-rw-r--r-- 1 anush-pandey-24128423 anush-pandey-24128423 225280 May 26 18:32 mlf
low.db
drwxrwxr-x 2 anush-pandey-24128423 anush-pandey-24128423   4096 May 26 18:31 mlr
uns
(student_depression) anush-pandey-24128423@AsusZenbook:~/mlflow$
```

Starting up docker containers

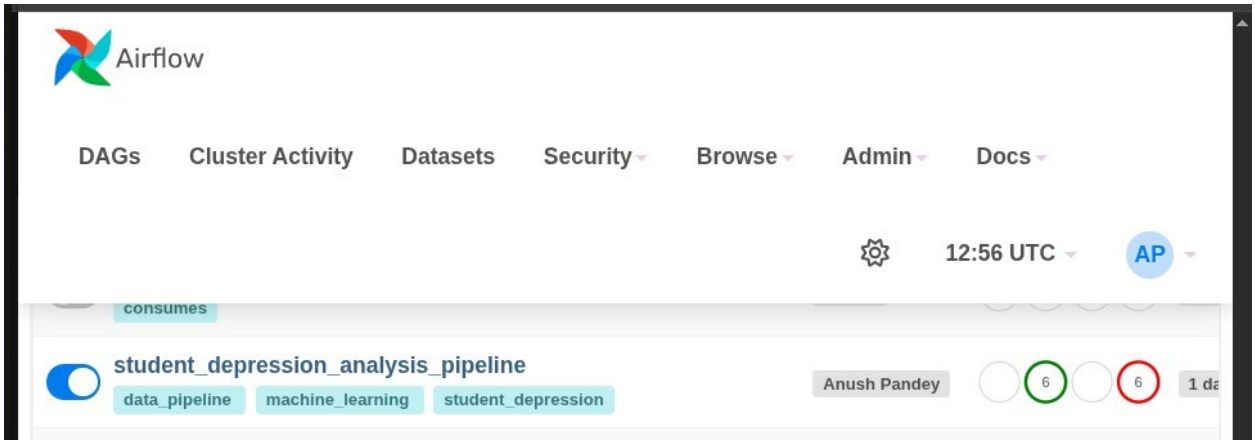
```
(student_depression) anush-pandey-24128423@AsusZenbook:~$ docker ps -a
CONTAINER ID   IMAGE                 COMMAND                  CREATED       STATU
S                    PORTS     NAMES
59e5f07506b4   mariadb/columnstore   "/usr/bin/tini -- do…"   3 days ago    Exite
d (255) 2 days ago             mcs_container
1dd4149f3947   redis                 "docker-entrypoint.s…"   3 days ago    Exite
d (255) 2 days ago             redis_store
(student_depression) anush-pandey-24128423@AsusZenbook:~$
```

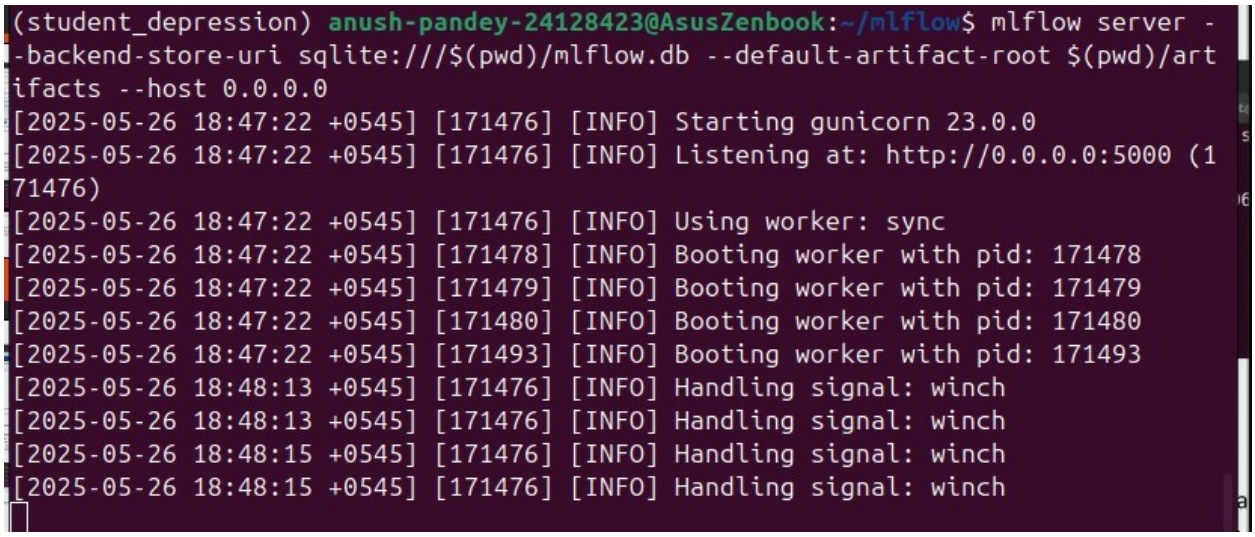
```
(student_depression) anush-pandey-24128423@AsusZenbook:~$ docker start mcs_conta
iner redis_store
```

Running Airflow

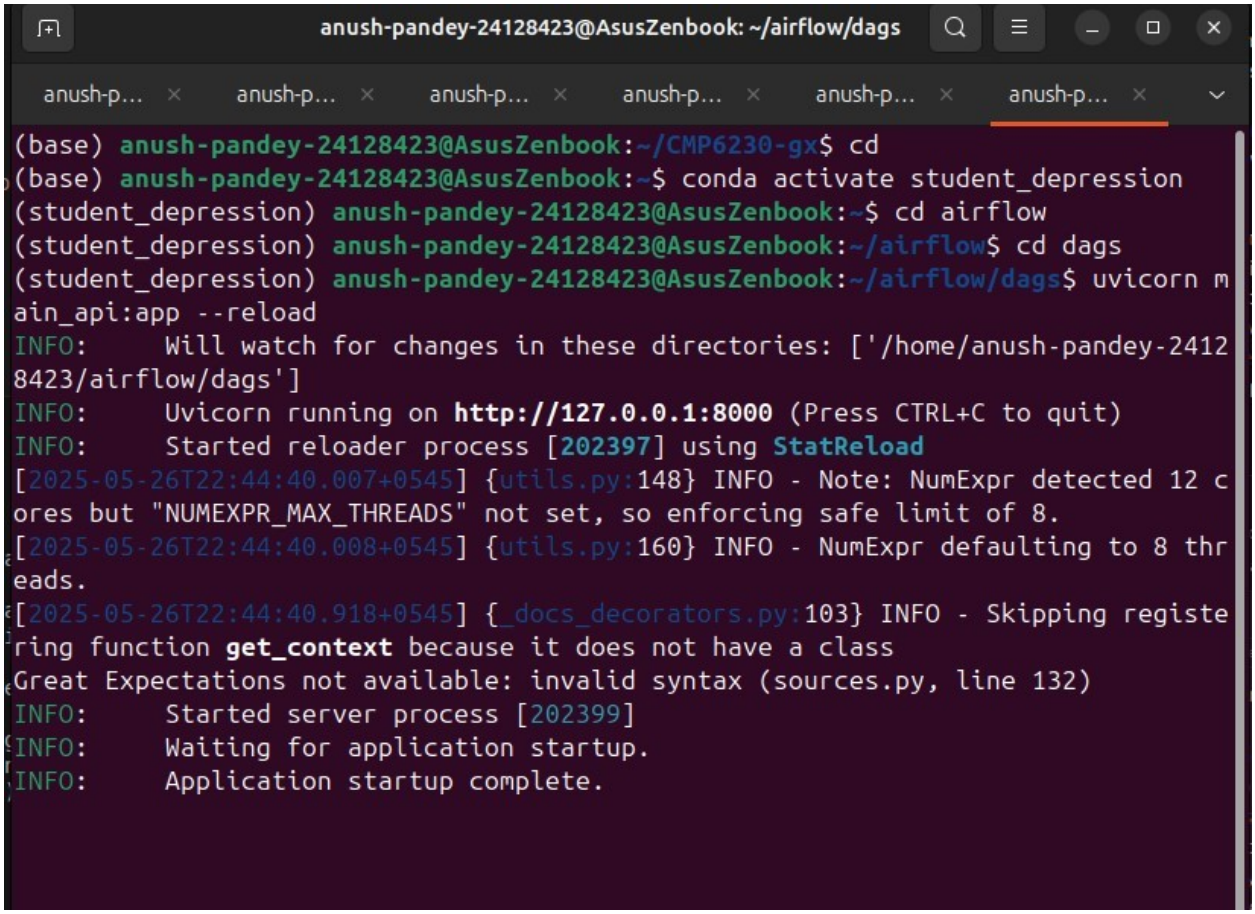
```
(base) anush-pandey-24128423@AsusZenbook:~$ conda activate student_depression
(student_depression) anush-pandey-24128423@AsusZenbook:~$ airflow webserver
  ____________       _____________
 ____    |__( )_________  __/__  /________      __
____  /| |_  /__  ___/_  /_ __  /_  __ \_ | /| / /
___  ___ |  / _  /   _  __/ _  / / /_/ /_ |/ |/ /
 _/_/  |_/_/  /_/    /_/    /_/  \____/____/|__/
Running the Gunicorn Server with:
Workers: 4 sync
Host: 0.0.0.0:8080
Timeout: 120
Logfiles: - -
Access Logformat:
==================================================================
```

```
(base) anush-pandey-24128423@AsusZenbook:~$ conda activate student_depression
(student_depression) anush-pandey-24128423@AsusZenbook:~$ airflow scheduler
  ____________       _____________
 ____    |__( )_________  __/__  /________      __
____  /| |_  /__  ___/_  /_ __  /_  __ \_ | /| / /
___  ___ |  / _  /   _  __/ _  / / /_/ /_ |/ |/ /
 _/_/  |_/_/  /_/    /_/    /_/  \____/____/|__/
[2025-05-26T18:40:15.335+0545] {utils.py:148} INFO - Note: NumExpr detected 12 c
ores but "NUMEXPR_MAX_THREADS" not set, so enforcing safe limit of 8.
[2025-05-26T18:40:15.336+0545] {utils.py:160} INFO - NumExpr defaulting to 8 thr
eads.
[2025-05-26T18:40:15.528+0545] {executor_loader.py:258} INFO - Loaded executor:
SequentialExecutor
[2025-05-26 18:40:15 +0545] [170778] [INFO] Starting gunicorn 23.0.0
[2025-05-26 18:40:15 +0545] [170778] [INFO] Listening at: http://[::]:8793 (1707
78)
[2025-05-26 18:40:15 +0545] [170778] [INFO] Using worker: sync
[2025-05-26T18:40:15.565+0545] {scheduler_job_runner.py:950} INFO - Starting the
 scheduler
[2025-05-26T18:40:15.566+0545] {scheduler_job_runner.py:957} INFO - Processing e
ach file at most -1 times
[2025-05-26 18:40:15 +0545] [170779] [INFO] Booting worker with pid: 170779
[2025-05-26T18:40:15.576+0545] {manager.py:174} INFO - Launched DagFileProcessor
Manager with pid: 170780
```





Fast API Setup



Reference

- Allam, Z., Jones, D.S. and Loukaitou-Sideris, A., 2021. *Data governance and ethical use of AI: opportunities and challenges*. AI & Society, 36(1), pp.185–193. Available at: https://www.researchgate.net/publication/387226449_Data_Ethics_in_AI_Addressing_Challenges_in_Machine_Learning_and_Data_Governance_for_Responsible_Data_Science [Accessed 24 May 2025].

- Bathwal, S. (2021) *Flight Price Prediction*. Kaggle. Available at: https://www.kaggle.com/datasets/shubhambathwal/flight-price-prediction (Accessed: 24 May 2025).

- Juledz. (2022) *Heart Attack Prediction*. Kaggle. Available at: https://www.kaggle.com/datasets/juledz/heart-attack-prediction (Accessed: 24 May 2025).

- Shamim, A. (2022) *Student Depression Dataset*. Kaggle. Available at: https://www.kaggle.com/datasets/adilshamim8/student-depression-dataset (Accessed: 24 May 2025).

- IBM. (n.d.) *Supervised Learning*. IBM Cloud Learn Hub. Available at: https://www.ibm.com/cloud/learn/supervised-learning (Accessed: 24 May 2025).

- Schelter, S., Lange, D., Bernecker, T., Klein, T. and Saake, G., 2020. Automating and operationalizing machine learning pipelines. *ACM Computing Surveys (CSUR)*, 53(6), pp.1–44. Available at: https://dl.acm.org/doi/10.1145/3363574

- Loukidis-Andreou, F. et al. (2018) 'Docker-SEC: A fully automated container security enhancement mechanism', 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS) [Preprint]. doi:10.1109/icdcs.2018.00169.

- Maji, A.K., Gorenstein, L. and Lentner, G. (2020) 'Demystifying python package installation with Conda-env-mod', 2020 IEEE/ACM International Workshop on HPC User Support Tools (HUST) and Workshop on Programming and Performance Visualization Tools (ProTools), pp. 27–37. doi:10.1109/hustprotools51951.2020.00011.

- Apache Airflow, n.d. *Apache Airflow Documentation*. [online] Available at: https://airflow.apache.org/docs/apache-airflow/stable/index.html [Accessed 24 May 2025].

- Meehan, J., Aslantas, C., Zdonik, S., Tatbul, N., & Du, J. (2017). Data Ingestion for the Connected World. CIDR 2017. https://www.cidrdb.org/cidr2017/papers/p48meehancidr17.pdf

- GeeksforGeeks. (2025a). What is Feature Engineering? Available at: https://www.geeksforgeeks.org/what-is-feature-engineering/ [Accessed 24 May 2025].

- GeeksforGeeks. (2025b). One Hot Encoding in Machine Learning. Available at: https://www.geeksforgeeks.org/ml-one-hot-encoding/ [Accessed 24 May 2025].

- GeeksforGeeks. (2025c). Feature Scaling Techniques. Available at: https://www.geeksforgeeks.org/ml-feature-scaling-part-2 [Accessed 24 May 2025].

- Statology. (2025). PyArrow Fundamentals for Statistical Data Processing. Available at: https://www.statology.org/pyarrow-fundamentals-statistical-data-processing/ [Accessed 24 May 2025].

- Redis. (2025). Redis for Machine Learning. Available at: https://redis.io/resources/redismachine-learning/ [Accessed 24 May 2025].

- IBM. (n.d.). What is a REST API? Available at: https://www.ibm.com/cloud/learn/restapis [Accessed 25 May 2025].

- FastAPI. (n.d.). FastAPI: The modern, fast (high-performance), web framework for building APIs with Python 3.7+. Available at: https://fastapi.tiangolo.com/ [Accessed 25 May 2025].

- Uvicorn. (n.d.). Uvicorn: The lightning-fast ASGI server implementation, using uvloop and httptools. Available at: https://www.uvicorn.org/ [Accessed 25 May 2025].

- MLflow. (n.d.). MLflow: Open source platform for the machine learning lifecycle. Available at: https://mlflow.org/ [Accessed 25 May 2025].

- Pydantic. (n.d.). Data validation and settings management using Python type annotations. Available at: https://docs.pydantic.dev/ [Accessed 25 May 2025].

- Breck, E., Cai, S., Nielsen, E., Salib, M. and Sculley, D., 2017. The ML test score: A rubric for ML production readiness and technical debt reduction. Available at:

https://storage.googleapis.com/pub-tools-public-publication-data/pdf/45742.pdf [Accessed 25 May 2025].

- IBM (2021). Redis. [online] Ibm.com. Available at: https://www.ibm.com/think/topics/redis.

- Munteanu, A. (2018). What is MLflow? | Ubuntu. [online] Ubuntu. Available at: https://ubuntu.com/blog/what-is-mlflow [Accessed 26 May 2025].