CS5830: Big Data Laboratory

Final Project

Report

Course Instructor: Balaraman Ravindran

Submitted By: Group 2 - Vishal V, Akranth, Sai Gautam
Roll Number: ME20B204, ME20B100, ED19B063

Date: 17/05/2024

Indian Institute of Technology Madras

Chennai 600036, India

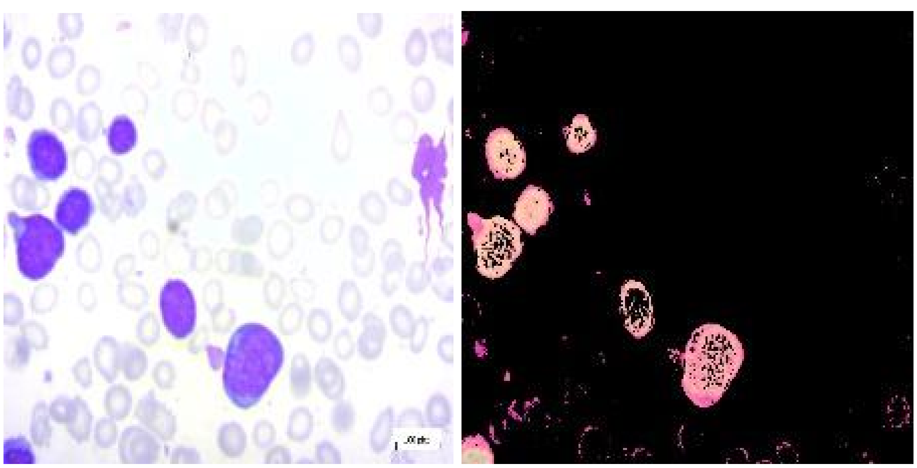
Repo Link: https://github.com/Akranth3/Final_project

Problem Statement & Workflow

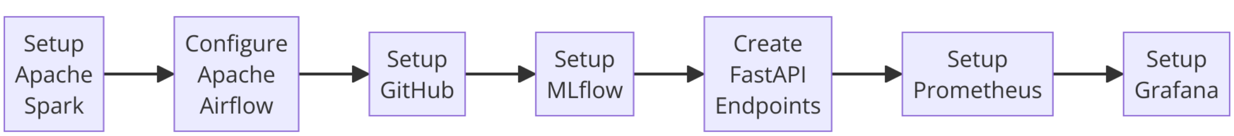
Acute Lymphoblastic Leukemia (ALL) is an aggressive form of cancer that predominantly affects children. Early detection and accurate classification of ALL are crucial for effective treatment and improving patient outcomes. However, diagnosing ALL through microscopic blood smear images is challenging due to visual similarities with other conditions and the need for specialized expertise.

This project aims to leverage an MLOps approach to build an end-to-end machine learning solution for the detection and classification of ALL from microscopic blood smear images. The solution will include a

- data preprocessing pipeline using Apache Airflow,
- machine learning model tracking via MLflow, and
- a scalable REST API for model deployment using FastAPI.
- the entire solution will be containerized for seamless deployment and monitored using Prometheus and Grafana.



Original & Segmented Image



Workflow Diagram

Apache Spark

Preprocessing function using Apache Spark.

```
def preprocessing(car_path, mask_path):
    car_img = tf.io.read_file(car_path)
    car_img = tf.image.decode_jpeg(car_img, channels=3)
    car_img = tf.image.resize(car_img, img_size)
    car_img = tf.cast(car_img, tf.float32) / 255.0

    mask_img = tf.io.read_file(mask_path)
    mask_img = tf.image.decode_jpeg(mask_img, channels=3)
    mask_img = tf.image.resize(mask_img, img_size)
    mask_img = mask_img[:,:,:1]
    mask_img = tf.math.sign(mask_img)

    return car_img, mask_img
```

Preprocessing function using Apache Spark.

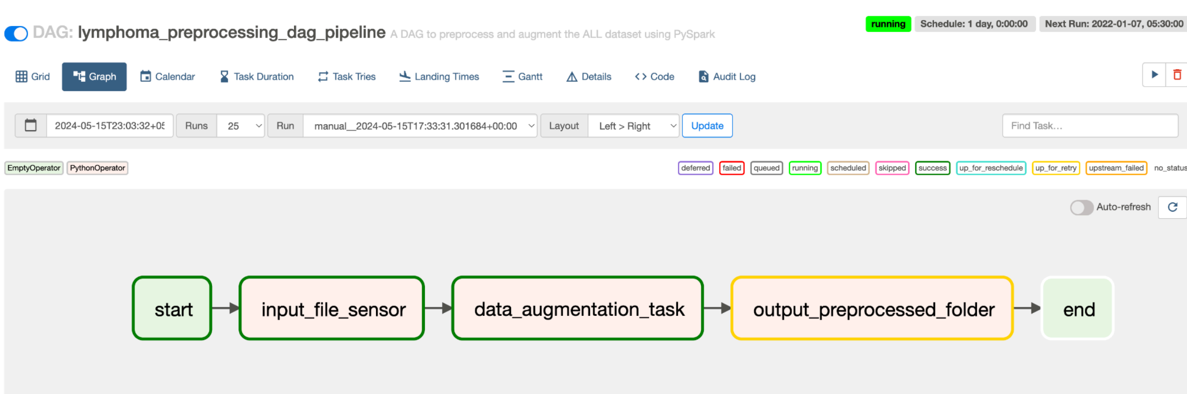
```
def preprocessing(car_path, mask_path):
    # Read car image and mask image
    car_img_data = tf.io.read_file(car_path)
    mask_img_data = tf.io.read_file(mask_path)

    # Define a function to process each image
    def process_image(img_data):
        img = tf.image.decode_jpeg(img_data, channels=3)
        img = tf.image.resize(img, img_size)
        img = tf.cast(img, tf.float32) / 255.0
        return img.numpy()

    # Process car image and mask image
    car_img_np = np.array([process_image(car_img_data)])
    mask_img_np = np.array([process_image(mask_img_data)[:,:,:1]

    return car_img_np, mask_img_np

# Define the image size
img_size = (256, 256)

# Call the preprocessing function
car_img_np, mask_img_np = preprocessing(car_path, mask_path)

# Create RDDs from the numpy arrays
car_img_rdd = spark.sparkContext.parallelize(car_img_np)
mask_img_rdd = spark.sparkContext.parallelize(mask_img_np)

# Collect RDDs into lists
car_img_list = car_img_rdd.collect()
mask_img_list = mask_img_rdd.collect()

# Close the SparkSession
spark.stop()

# Convert lists to numpy arrays
car_img_np_final = np.array(car_img_list)
mask_img_np_final = np.array(mask_img_list)
```

Airflow

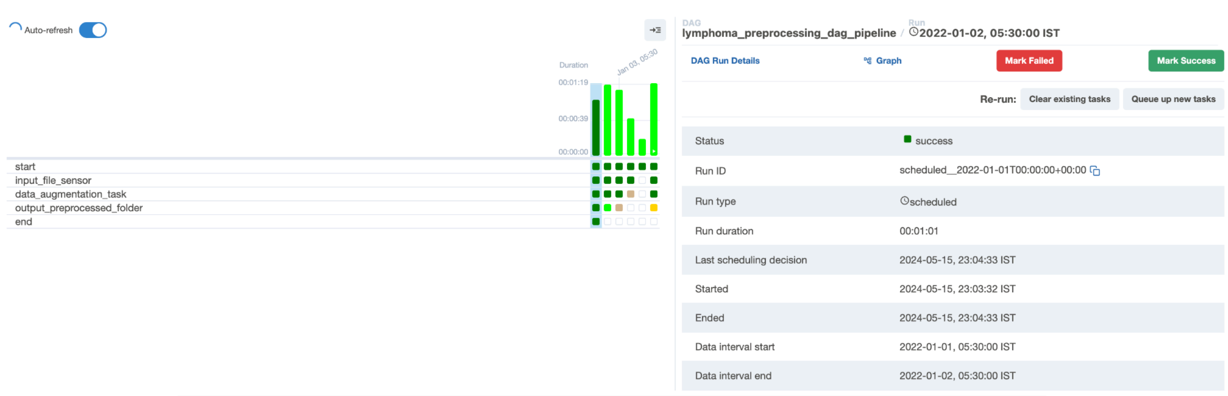
Run airflow standalone

In the browser at http://localhost:8080/, the airflow dag can be run (dag named lymphoma_preprocessing_dag_pipeline)

DAG visualization - Graph



Grid

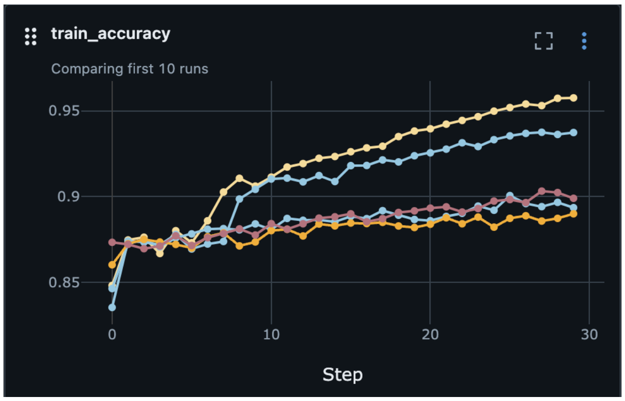


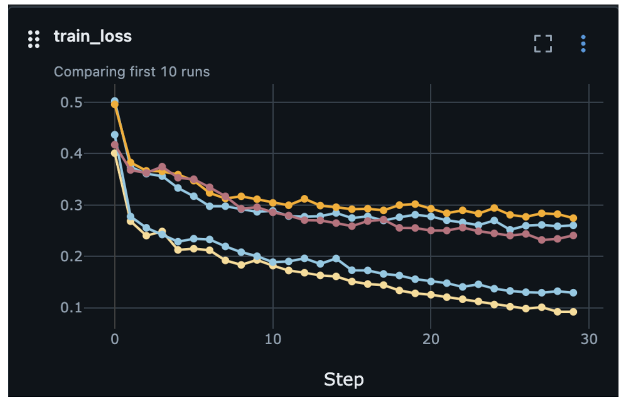
- Automation: Reduces manual effort and ensures consistent preprocessing and augmentation of images.
- Reproducibility: Provides a repeatable process that can be easily triggered and monitored.
- Flexibility: Can be adapted for different datasets and preprocessing requirements.

MLflow

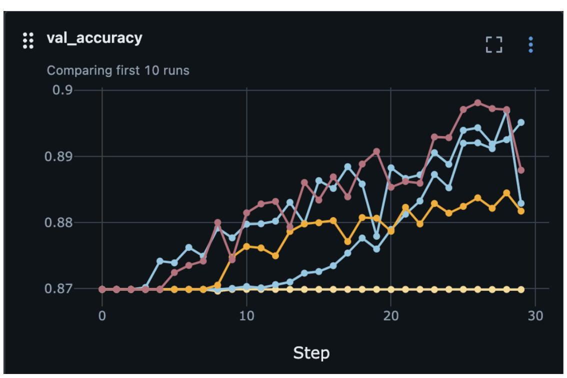
Run mlflow server

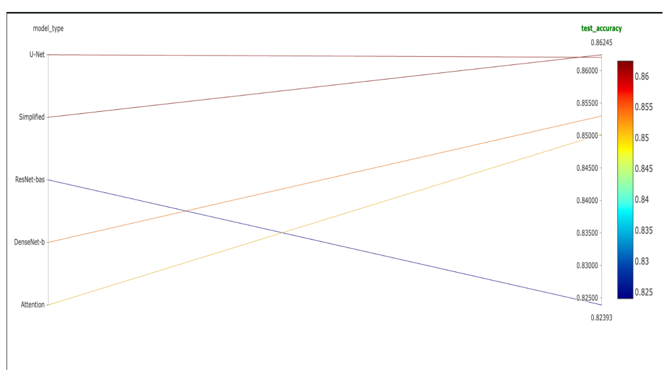
In the browser at http://localhost:5000/, the experiments conducted and metrics tracked can be visualized and recorded.





- The experiment tracking with MLflow provides valuable insights into the performance of different models
- DenseNet-based Model: Offers a balanced performance with a high training accuracy (93.73%) and a strong validation accuracy (88.29%). Its consistent metrics make it a reliable choice for further use.





Code:

```
def log_metrics(history, metrics):
    for epoch in range(len(history.history['loss'])):
```

```
        mlflow.log_metric("train_loss",
history.history['loss'][epoch], step=epoch)
        mlflow.log_metric("val_loss",
history.history['val_loss'][epoch], step=epoch)
        for metric in metrics:
            mlflow.log_metric(f"train_{metric}",
history.history[metric][epoch], step=epoch)
            mlflow.log_metric(f"val_{metric}",
history.history[f"val_{metric}"][epoch], step=epoch)

# MLflow tracking for experiments with different model architectures
with mlflow.start_run(run_name='ALL Experiments only cross entropy
2'):

    # Nested run for the U-Net model
    with mlflow.start_run(nested=True, run_name='U-Net Model'):
        unet_model = build_unet_model(1)
        unet_model.compile(optimizer='adam',
loss='binary_crossentropy', metrics=['accuracy'])
        history = unet_model.fit(train_dataset, epochs=EPOCHS,
steps_per_epoch=steps_per_epoch,
                       validation_data=valid_dataset,
validation_steps=validation_steps,
                       callbacks=[early_stop])
        mlflow.log_param("model_type", "U-Net")
        mlflow.log_param("optimizer", "adam")
        mlflow.log_param("loss_function", "binary_crossentropy")
        mlflow.log_param("metrics", ["accuracy"])
        log_metrics(history, ["accuracy"])

        # Evaluate on the test set
        test_loss, test_accuracy = unet_model.evaluate(test_dataset)
        mlflow.log_metric("test_loss", test_loss)
        mlflow.log_metric("test_accuracy", test_accuracy)

        mlflow.keras.log_model(unet_model, 'unet_model')

    # Nested run for the Simplified CNN model
    with mlflow.start_run(nested=True, run_name='Simplified CNN
```

```
Model'):
        cnn_model = build_simplified_cnn()
        cnn_model.compile(optimizer='adam',
loss='binary_crossentropy', metrics=['accuracy'])
        history = cnn_model.fit(train_dataset, epochs=EPOCHS,
steps_per_epoch=steps_per_epoch,
                              validation_data=valid_dataset,
validation_steps=validation_steps)
        mlflow.log_param("model_type", "Simplified CNN")
        mlflow.log_param("optimizer", "adam")
        mlflow.log_param("loss_function", "binary_crossentropy")
        mlflow.log_param("metrics", ["accuracy"])
        log_metrics(history, ["accuracy"])

        # Evaluate on the test set
        test_loss, test_accuracy = cnn_model.evaluate(test_dataset)
        mlflow.log_metric("test_loss", test_loss)
        mlflow.log_metric("test_accuracy", test_accuracy)

        mlflow.keras.log_model(cnn_model, 'simplified_cnn_model')

    # Nested run for the DenseNet-based model
    with mlflow.start_run(nested=True, run_name='DenseNet-based
Model'):
        densenet_model = build_densenet_based_model()
        densenet_model.compile(optimizer='adam',
loss='binary_crossentropy', metrics=['accuracy'])
        history = densenet_model.fit(train_dataset, epochs=EPOCHS,
steps_per_epoch=steps_per_epoch,
                                   validation_data=valid_dataset,
validation_steps=validation_steps)
        mlflow.log_param("model_type", "DenseNet-based")
        mlflow.log_param("optimizer", "adam")
        mlflow.log_param("loss_function", "binary_crossentropy")
        mlflow.log_param("metrics", ["accuracy"])
        log_metrics(history, ["accuracy"])

        # Evaluate on the test set
        test_loss, test_accuracy =
```

```
densenet_model.evaluate(test_dataset)
        mlflow.log_metric("test_loss", test_loss)
        mlflow.log_metric("test_accuracy", test_accuracy)

        mlflow.keras.log_model(densenet_model,
'densenet_based_model')

        # Nested run for the Attention U-Net model
    # Nested run for the Attention U-Net model
    with mlflow.start_run(nested=True, run_name='Attention U-Net
Model'):
        print('Attention U-Net Model Training...')
        attention_unet_model = build_attention_unet(1)
        attention_unet_model.compile(optimizer='adam',
loss='binary_crossentropy', metrics=['accuracy'])
        history = attention_unet_model.fit(train_dataset,
epochs=EPOCHS, steps_per_epoch=steps_per_epoch,

validation_data=valid_dataset, validation_steps=validation_steps)
        print('Logging parameters and metrics...')
        mlflow.log_param("model_type", "Attention U-Net")
        mlflow.log_param("optimizer", "adam")
        mlflow.log_param("loss_function", "binary_crossentropy")
        mlflow.log_param("metrics", ["accuracy"])
        log_metrics(history, ["accuracy"])

        # Evaluate on the test set
        test_loss, test_accuracy =
attention_unet_model.evaluate(test_dataset)
        mlflow.log_metric("test_loss", test_loss)
        mlflow.log_metric("test_accuracy", test_accuracy)

        mlflow.keras.log_model(attention_unet_model,
'attention_unet_model')

    # Nested run for the ResNet-based U-Net model
    with mlflow.start_run(nested=True, run_name='ResNet-based U-Net
Model'):
        print('ResNet-based U-Net Model Training...')
```

```
        resnet_unet_model = build_resnet_unet(1)
        resnet_unet_model.compile(optimizer='adam',
loss='binary_crossentropy', metrics=['accuracy'])
        history = resnet_unet_model.fit(train_dataset, epochs=EPOCHS,
steps_per_epoch=steps_per_epoch,

validation_data=valid_dataset, validation_steps=validation_steps)
        print('Logging parameters and metrics...')
        mlflow.log_param("model_type", "ResNet-based U-Net")
        mlflow.log_param("optimizer", "adam")
        mlflow.log_param("loss_function", "binary_crossentropy")
        mlflow.log_param("metrics", ["accuracy"])
        log_metrics(history, ["accuracy"])

        # Evaluate on the test set
        test_loss, test_accuracy =
resnet_unet_model.evaluate(test_dataset)
        mlflow.log_metric("test_loss", test_loss)
        mlflow.log_metric("test_accuracy", test_accuracy)

        mlflow.keras.log_model(resnet_unet_model,
'resnet_unet_model')
```

Deployment

FastAPI: FastAPI is a modern, fast (high-performance), web framework for building APIs with Python 3.7+ based on standard Python type hints. It is designed to be easy to use and highly performant, making it ideal for creating APIs. In this project, FastAPI was used to create an API that processes image uploads and predicts digits using a pre-trained machine learning model. The API endpoints include:

- `GET /`: A simple endpoint to check if the server is running.
- `POST /predict-task-2/`: An endpoint to upload an image, process it, and return the predicted digit.
- `GET /metrics`: An endpoint to expose Prometheus metrics.

Prometheus: Prometheus is an open-source monitoring and alerting toolkit designed for reliability and scalability. It collects and stores metrics as time series data. In this project, Prometheus was integrated with FastAPI to track API usage and performance metrics. Specifically:

- **Counters**: Track the number of API requests from different client IP addresses.
- **Gauges**: Monitor the processing time of the API concerning the input text length, measuring the effective processing time in microseconds per character.

Grafana: Grafana is an open-source platform for monitoring and observability. It allows you to query, visualize, alert on, and understand your metrics no matter where they are stored. In this project, Grafana was used to visualize the metrics collected by Prometheus. By creating dashboards, users can monitor API performance and usage in real-time, identifying trends and potential issues.

Docker: Docker is a platform that allows developers to automate the deployment of applications inside lightweight, portable containers. Docker ensures that applications run consistently across different environments. In this project, Docker was used to containerize the FastAPI application, along with Prometheus and Grafana, making it easier to deploy and manage these services together. The Docker setup typically involves:

- **Dockerfile**: Defines the FastAPI application's environment and dependencies.
- **docker-compose.yml**: Manages multi-container Docker applications, setting up FastAPI, Prometheus, and Grafana to work together seamlessly.

Integration Details:

1. **FastAPI Integration**:
    - Created an API with endpoints for predicting digits (`/predict-task-2/`) and exposing metrics (`/metrics`).

2. **Prometheus Integration**:
    - Integrated Prometheus metrics by adding Counters and Gauges in the FastAPI application.
    - Exposed Prometheus metrics through the `/metrics` endpoint.
3. **Grafana Integration**:
    - Configured Grafana to connect to Prometheus as a data source.
    - Created dashboards in Grafana to visualize metrics such as API request counts and processing times.
4. **Docker Integration**:
    - Used Docker to containerize the FastAPI application, ensuring consistent deployment.
    - Created a `docker-compose.yml` file to set up FastAPI, Prometheus, and Grafana containers.

Endpoints:

- **`GET /`**: Health check endpoint.
- **`POST /predict-task-2/`**: Endpoint to upload an image and receive a predicted digit.
- **`GET /metrics`**: Endpoint to expose Prometheus metrics for scraping.

Docker file

```
#
FROM tiangolo/uvicorn-gunicorn-fastapi:python3.9

#
WORKDIR /code

#
COPY ./requirements.txt /code/requirements.txt

#
RUN pip install --no-cache-dir --upgrade -r /code/requirements.txt

#
COPY ./src /code/src
COPY ./utils /code/utils
COPY ./weights /code/weights

#
CMD ["fastapi", "run", "src/fast_api.py", "--port", "8080"]
```

Docker-compose file

```yaml
version: "3.8"

services:
 detect-api:
   container_name: detect-api
   build:
     context: .
     dockerfile: Dockerfile
   restart: 'on-failure'
   ports:
     - "8080:8080"

 prometheus:
   image: prom/prometheus
   restart: 'always'
   volumes:
     - ./prometheus.yml:/etc/prometheus/prometheus.yml
   command:
     - '--config.file=/etc/prometheus/prometheus.yml'
   ports:
     - "9090:9090"

 grafana:
   image: grafana/grafana
   restart: 'always'
   ports:
     - "3000:3000"
   environment:
     - GF_SECURITY_ADMIN_PASSWORD=admin
     - GF_USERS_ALLOW_SIGN_UP=false
     - GF_USERS_ALLOW_ORG_CREATE=false
     - GF_USERS_AUTO_ASSIGN_ORG=true
     - GF_USERS_AUTO_ASSIGN_ORG_ROLE=Editor
     - GF_AUTH_ANONYMOUS_ENABLED=true
     - GF_AUTH_ANONYMOUS_ORG_NAME=Main Org.
     - GF_AUTH_ANONYMOUS_ORG_ROLE=Viewer
   depends_on:
     - prometheus
```

[redacted]

To run the program

- Docker-compose up –build

<u>Contributions:</u>

Version control: **Git** and **Git-lfs**

- Vishal V (ME20B204):

  Apache Airflow Preprocessing Pipeline, Experiments Tracking using **MLFlow**, Monitoring Dashboard using **Grafana**

- Akranth (ME20B100):

Deployed APIs with **FastAPI**, tracking the API usage with **prometheus** and visualization with **grafana**, containerized the whole thing using **docker**. Tracking the project using git.

- Sai Gowtham Tamminaina (ED19B063):

  Apache Spark and Airflow for the Preprocessing Pipeline