Face Sentiment Detection

Yassine Maatougui

1. Project Inception

1.1. Framing the Business Idea as an ML Problem (milestone 1, 5%)

- **Business case description:**
  The project aims to develop a Face Sentiment Detection system capable of identifying and analyzing facial expressions to detect emotions. This technology can be applied in various sectors such as customer service, security, and healthcare to enhance user interactions and emotional assessments.

- **Business value of using ML:**
  By employing machine learning, we can automate the recognition of facial emotions with high accuracy, which is crucial for real-time applications. This reduces the need for manual interpretation and allows for quicker and more efficient decision-making processes in professional settings.

- **Data overview:**
  The project utilizes a comprehensive dataset consisting of images labeled with eight distinct emotions. This dataset helps in training the model to accurately recognize and classify different facial sentiments.

- **Project archetype:**

This is a classification problem where the model predicts the type of emotion based on facial features extracted from images.

1.2. Feasibility analysis

- **Literature review:**
  Reviewed various models from the TensorFlow 2 Detection Model Zoo and the yolov8-emotions model, which are renowned for their efficiency in object detection tasks. The selection was based on their performance metrics in similar tasks outlined in the models TensorFlow 2 Detection Model Zoo and YOLOV8.

- **Model choice/ specification of a baseline:**

Chose a robust model from the TensorFlow 2 Detection Model Zoo for initial benchmarking due to its pre-trained capabilities on similar image data, providing a solid baseline for performance comparison.

- **Metrics for business goal evaluation:**
  The primary metrics for evaluating the business goals are accuracy, processing speed, and the model's adaptability across various environments and demographic groups.

2. ML Pipeline Development - From a Monolith to a Pipeline

2.1. Ensuring ML Pipeline Reproducibility (milestone 2, 15%)

- **Project structure definition and modularity:**
  The ML pipeline is structured for modularity using TFX components to separate data handling, model training, and evaluation processes.

  For installing TFX, I have used Anaconda:
  - Installation of TFX using Miniconda:
    1. Firstly, I have installed miniconda3 from their official website.
    2. Then, I created an environment for my project using the command:
       conda create - - name tfx python=3.9
       conda activate tfx
    3. After that I installed tfx: pip install tfx==1.15.0 tensorflow==2.15.1
    4. Now, I was able to use tfx library and it gives me this result:
    5. For further installation of the libraries, I used conda install … Therefore, conda will not deprecate tfx and tensorflow versions, and it installs the compatible libraries needed.

  - However, I still had an issue when I wanted to use the Object Detection library. Therefore, I went to the tfx library, which I installed it using the previous commands and I changed the function ‘preprocessing_fn’. Under that function there is a function named ‘scale_input(inputs)’. On this function I have added

these lines of code:

```
image = tf.image.decode_jpeg(inputs, channels=3)
image = tf.image.resize(image, [600, 600])
return image
```

For creating the Pipeline:

Command used: "`tfx pipeline create --pipeline-name=my_face_sentiment_detection_pipeline --pipeline-root={PIPELINE_ROOT} --endpoint={ENDPOINT}`"

- `This is the notebook that contains the whole pipeline that I have built` ➔ https://github.com/yassineMtg/ObjectDetection/blob/main/tfxfu.ipynb

- **Code versioning:**
  Using Git hosted on GitHub for version control allows meticulous tracking of all changes throughout the development cycle, enhancing transparency and collaboration.

  Git commands for versioning:

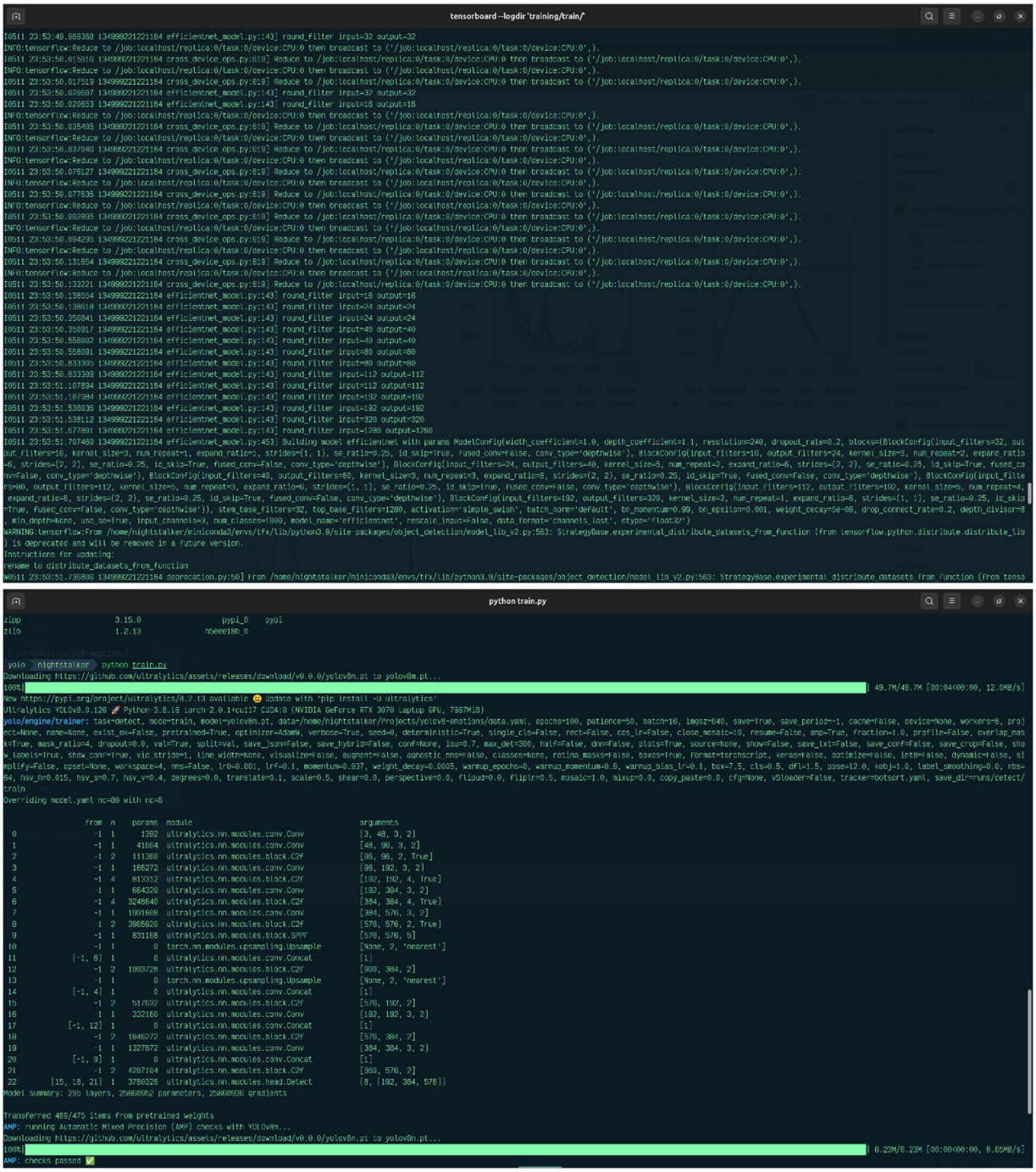
```
git init
git add .
git commit -m "Initial commit"
```

- **Data versioning:**
  Data Version Control (DVC) manages dataset changes systematically, maintaining data integrity and ensuring reproducibility across different pipeline stages.

  DVC commands used to track data versions:

```
dvc init
dvc add data/dataset
git add data.dvc .gitignore
git commit -m "Add dataset with DVC"
```

- **Experiment tracking and model versioning:**
  The Metadata Store in TFX is used for tracking experiments and managing model versions, providing detailed insights into model performance and version history.



- **Setting up a meta store for metadata:**
  TFX's ML Metadata is used to store metadata about experiments, model training, and evaluation processes.

- **Setting up the machine learning pipeline under an MLOps platform:**

The entire ML pipeline is configured and managed using TFX, facilitating continuous integration and deployment practices.

2.2. Pipeline Components (Milestone 3 and 4, 20%)

1. **Setup of data pipeline within the larger ML pipeline/ MLOps Platform**
    - **Data Validation and Verification:**

      TFX Example Validator ensures data quality by comparing data statistics against a schema that describes expectations about data.
      https://github.com/yassineMtg/ObjectDetection/blob/main/src/pipeline/examplegen.py

SchemaGen

```
schema_gen = tfx.components.SchemaGen(
    statistics=statistics_gen.outputs['statistics'],
    infer_feature_shape=False)
context.run(schema_gen, enable_cache=True)
```

▼**ExecutionResult** at 0x7a1bdc5b7550

.execution_id	3
.component	▶**SchemaGen** at 0x7a1be5608ee0
.component.inputs	**['statistics']** ▶**Channel** of type **'ExampleStatistics'** (1 artifact) at 0x7a1be55f3d00
.component.outputs	**['schema']** ▶**Channel** of type **'Schema'** (1 artifact) at 0x7a1bbc3aa970

https://github.com/yassineMtg/ObjectDetection/blob/main/src/pipeline/schemagen.py

```
context.show(schema_gen.outputs['schema'])
```

Artifact at /tmp/tfx-interactive-2024-05-12T12_47_30.457696-ip1zycrf/SchemaGen/schema/3

Feature name	Type	Presence	Valency	Domain
'image/encoded'	BYTES	required	single	-
'image/filename'	BYTES	required	single	-
'image/format'	STRING	required	single	'image/format'
'image/height'	INT	required	single	-
'image/object/bbox/xmax'	FLOAT	required	single	-
'image/object/bbox/xmin'	FLOAT	required	single	-
'image/object/bbox/ymax'	FLOAT	required	single	-
'image/object/bbox/ymin'	FLOAT	required	single	-
'image/object/class/label'	INT	required	single	-
'image/object/class/text'	STRING	required	single	'image/object/class/text'
'image/source_id'	BYTES	required	single	-
'image/width'	INT	required	single	-

Domain	Values
'image/format'	'jpeg'
'image/object/class/text'	'anger', 'contempt', 'disgust', 'fear', 'happy', 'neutral', 'sad', 'surprise'

```
example_validator = tfx.components.ExampleValidator(
    statistics=statistics_gen.outputs['statistics'],
    schema=schema_gen.outputs['schema'])
context.run(example_validator, enable_cache=True)
```

```
▼ExecutionResult at 0x7a1be565ad00
    .execution_id  4
    .component  ▶ExampleValidator at 0x7a1bdc6df8b0
    .component.inputs
        ['statistics']  ▶Channel of type 'ExampleStatistics' (1 artifact) at 0x7a1be55f3d00
        ['schema']  ▶Channel of type 'Schema' (1 artifact) at 0x7a1bbc3aa970
    .component.outputs
        ['anomalies']  ▶Channel of type 'ExampleAnomalies' (1 artifact) at 0x7a1bdc6df640
```

```
from cassandra.cluster import Cluster
cluster = Cluster(['127.0.0.1'])
session = cluster.connect('emotion_detection')
rows = session.execute('SELECT * FROM image_labels')
```

https://github.com/yassineMtg/ObjectDetection/blob/main/src/pipeline/examplevalidator.py

ExampleValidator

```
context.show(example_validator.outputs['anomalies'])
```

Artifact at /tmp/tfx-interactive-2024-05-12T12_47_30.457696-ip1zycrf/ExampleValidator/anomalies/4

'train' split:

No anomalies found.

'eval' split:

No anomalies found.

- **Preprocessing and Feature Engineering:**
  TFX Transform component is used for data preprocessing and feature engineering, ensuring that data is in the correct format for training.
  https://github.com/yassineMtg/ObjectDetection/blob/main/src/pipeline/transform.py

```python
def create_tf_example(image_path, annotations, class_int):
    # Load the image
    image = Image.open(image_path)
    image = image.resize((640, 640))  # Resize if not already 640x640

    # Encode the image to JPEG
    img_byte_arr = io.BytesIO()
    image.save(img_byte_arr, format='JPEG')
    image_bytes = img_byte_arr.getvalue()

    # Normalize bounding box coordinates
    xmin = [anno['xmin'] / image.width for anno in annotations]
    xmax = [anno['xmax'] / image.width for anno in annotations]
    ymin = [anno['ymin'] / image.height for anno in annotations]
    ymax = [anno['ymax'] / image.height for anno in annotations]
    filename = image_path.split('/')[-1].encode('utf-8')
    classes_text = [anno['class'].encode('utf-8') for anno in annotations]
    classes = [class_id for class_id in class_int]

    data = {
        'image/height': dataset_util.int64_feature(image.height),
        'image/width': dataset_util.int64_feature(image.width),
        'image/filename': dataset_util.bytes_feature(filename),
        'image/source_id': dataset_util.bytes_feature(filename),
        'image/encoded': dataset_util.bytes_feature(image_bytes),
        'image/format': dataset_util.bytes_feature(b'jpeg'),
        'image/object/bbox/xmin': dataset_util.float_list_feature(xmin),
        'image/object/bbox/xmax': dataset_util.float_list_feature(xmax),
        'image/object/bbox/ymin': dataset_util.float_list_feature(ymin),
        'image/object/bbox/ymax': dataset_util.float_list_feature(ymax),
        'image/object/class/text': dataset_util.bytes_list_feature(classes_text),
        'image/object/class/label': dataset_util.int64_list_feature(classes),
    }

    return tf.train.Example(features=tf.train.Features(feature=data))
```

```python
_transform_module_file = '_transform_component.py'
```

```python
%%writefile {_transform_module_file}
import tensorflow_transform as tft
import tensorflow as tf

def preprocessing_fn(inputs):
    """Preprocess input columns into transformed columns."""
    outputs = inputs

    return outputs
```

```
Overwriting _transform_component.py
```

```python
transform = tfx.components.Transform(
    examples=example_gen.outputs['examples'],
    schema=schema_gen.outputs['schema'],
    module_file=os.path.abspath(_transform_module_file))
context.run(transform, enable_cache=True)
```

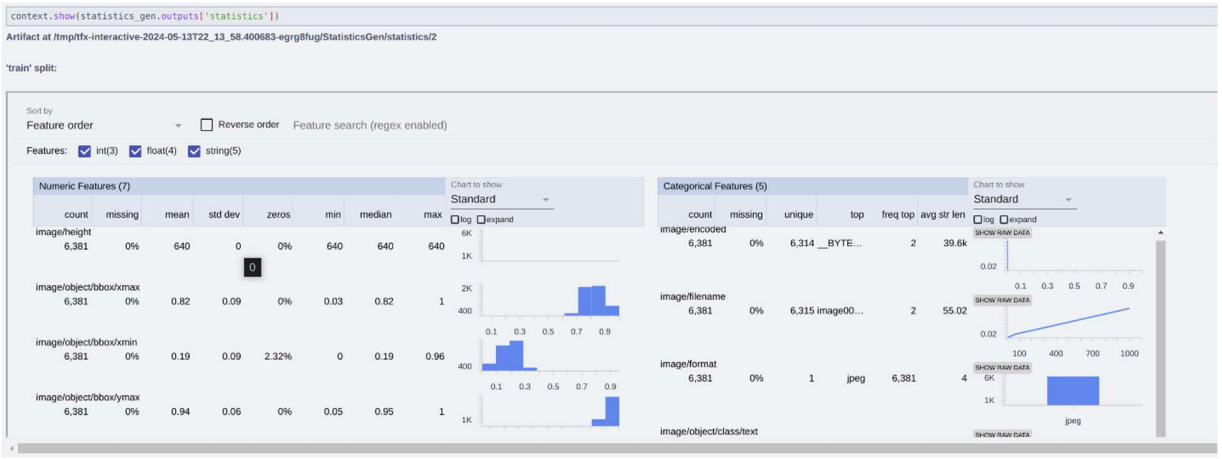
https://github.com/yassineMtg/ObjectDetection/blob/main/src/pipeline/statisticsgen.py

StatisticsGen

```
statistics_gen = tfx.components.StatisticsGen(
    examples=example_gen.outputs['examples'])
context.run(statistics_gen, enable_cache=True)
```

▼ **ExecutionResult** at 0x7a0f8c236dc0

.execution_id	2
.component	▶**StatisticsGen** at 0x7a0eb5d830a0
.component.inputs	['examples'] ▶**Channel** of type **'Examples'** (1 artifact) at 0x7a0ebdab5a90
.component.outputs	['statistics'] ▶**Channel** of type **'ExampleStatistics'** (1 artifact) at 0x7a0eb5d83160



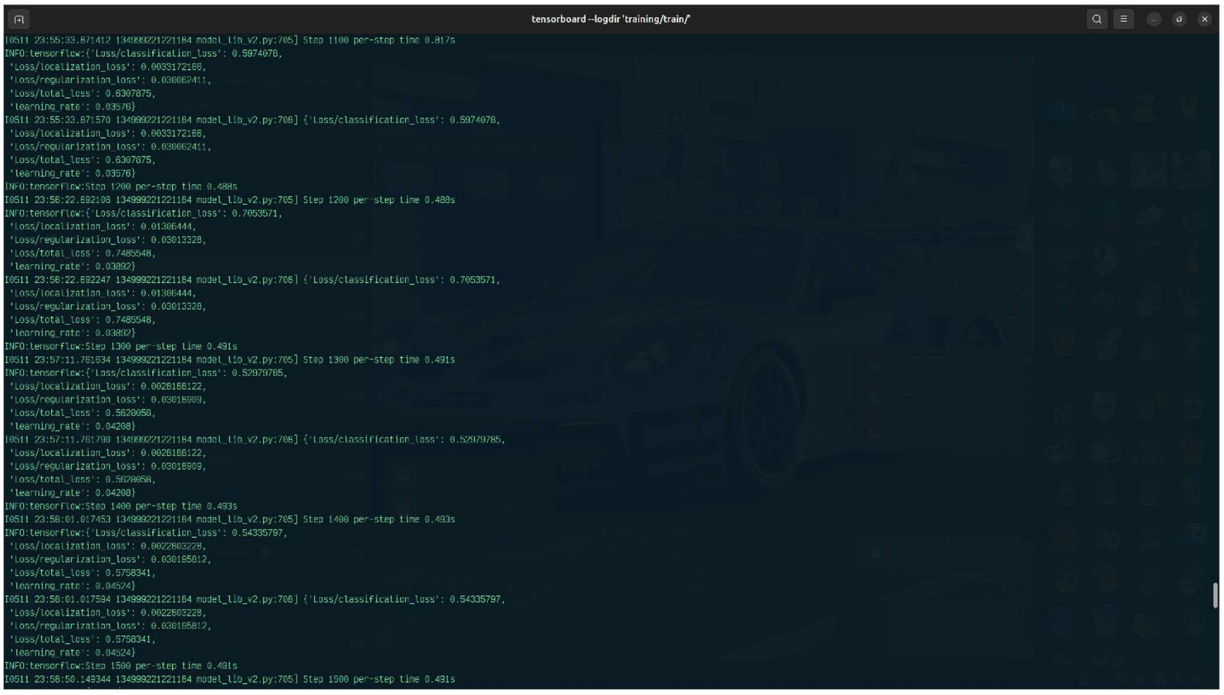
2. **Integration of model training and offline evaluation into the ML pipeline / MLOps Platform**

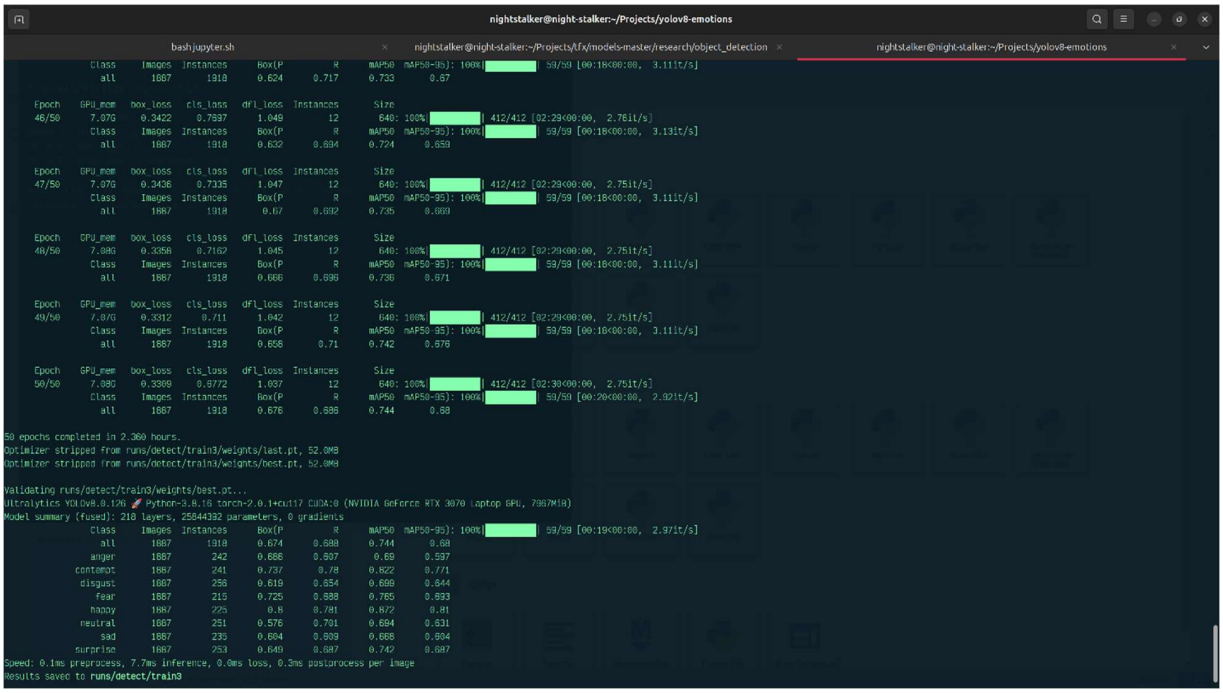
   TFX Trainer and Evaluator components are integrated for model training and performance evaluation, ensuring the model is trained and assessed rigorously before deployment.

   https://github.com/yassineMtg/ObjectDetection/blob/main/src/pipeline/trainer.py

```
trainer = tfx.components.Trainer(
    module_file=os.path.abspath(_trainer_module),
    examples=transform.outputs['transformed_examples'],
    transform_graph=transform.outputs['transform_graph'],
    schema=schema_gen.outputs['schema'],)
context.run(trainer, enable_cache=True)
```

Result:

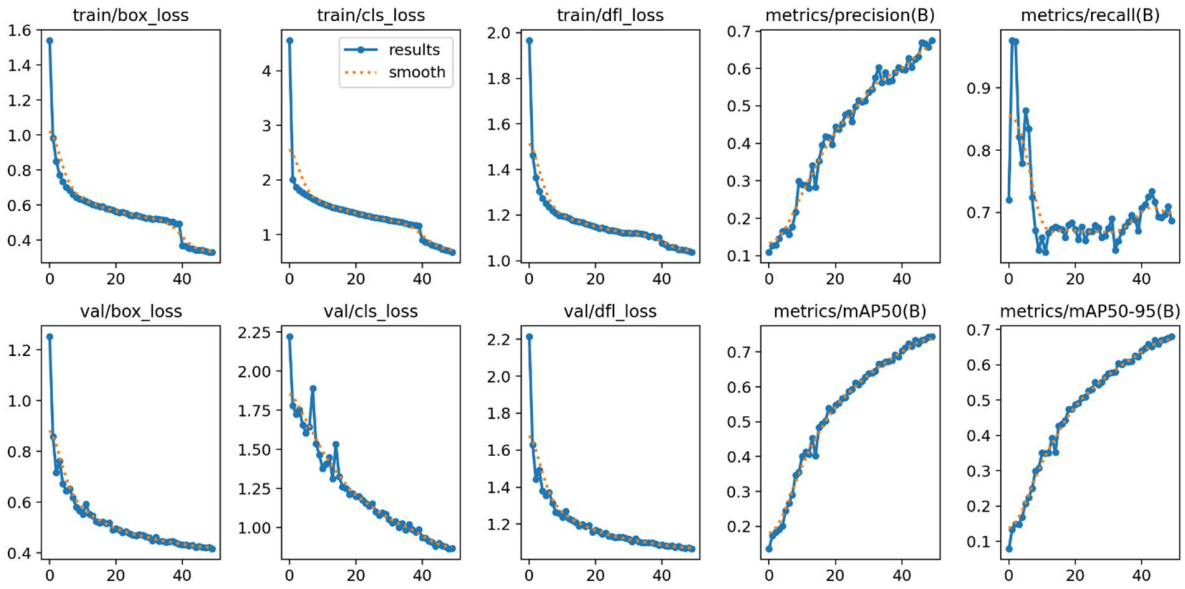


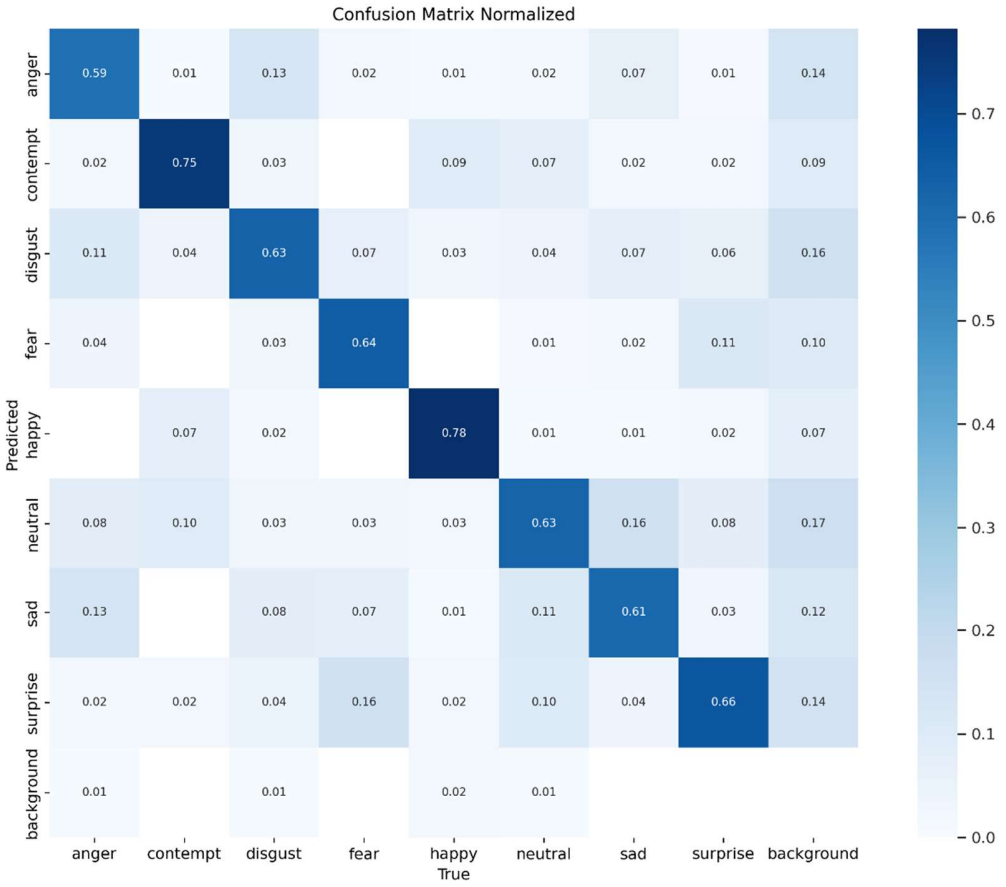


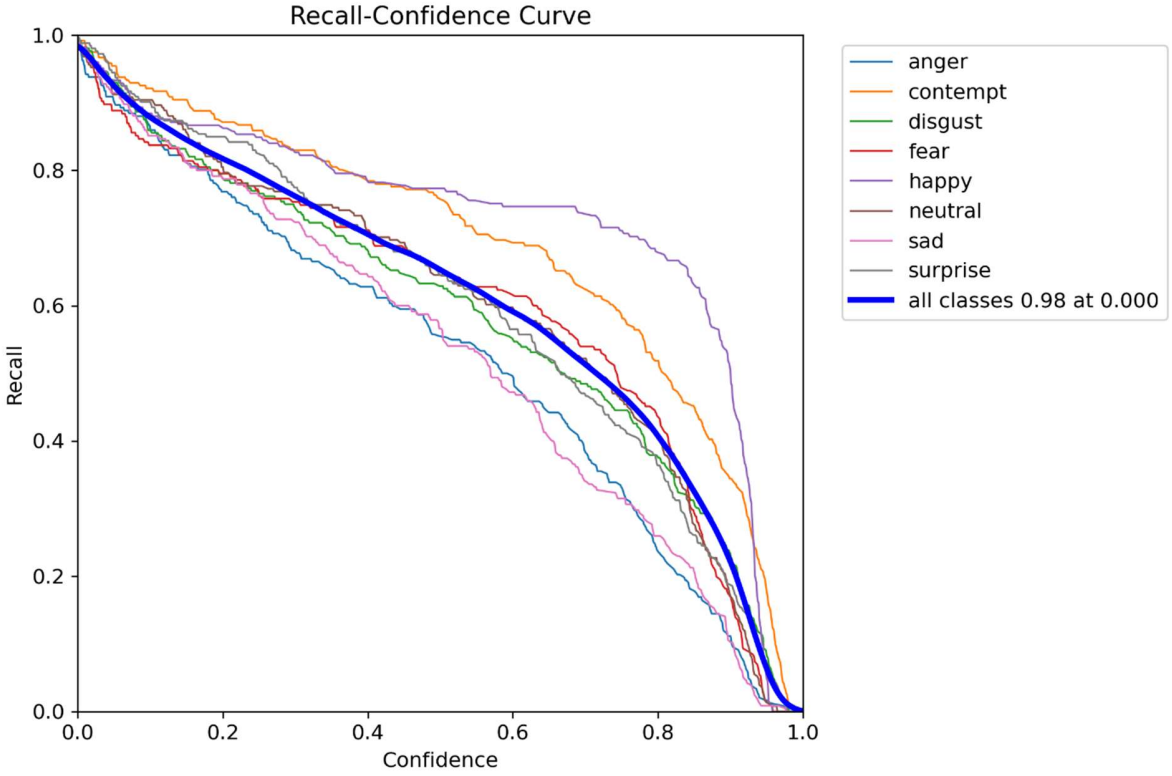
3. Development of model behavioral tests

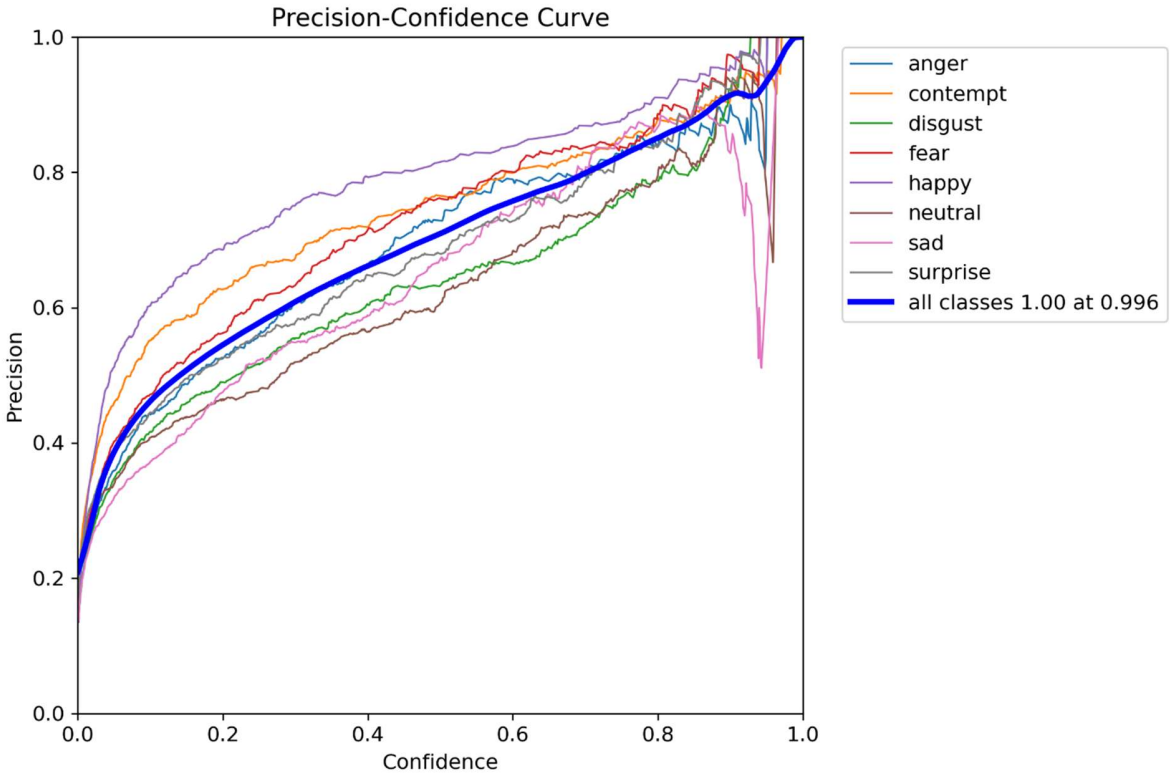
Behavioral tests are designed to verify the model's functionality and performance under various conditions, ensuring robustness and reliability. I've used the library pytest for this step.

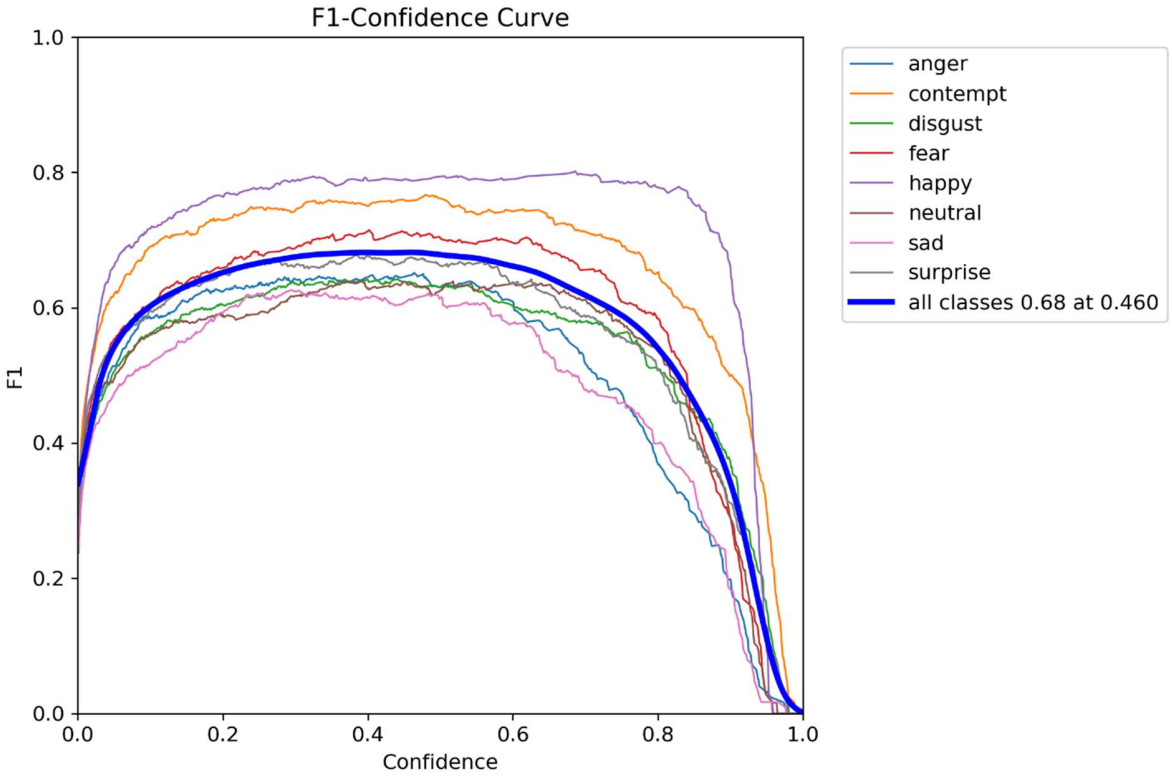
https://github.com/yassineMtg/ObjectDetection/blob/main/src/pipeline/testing.py

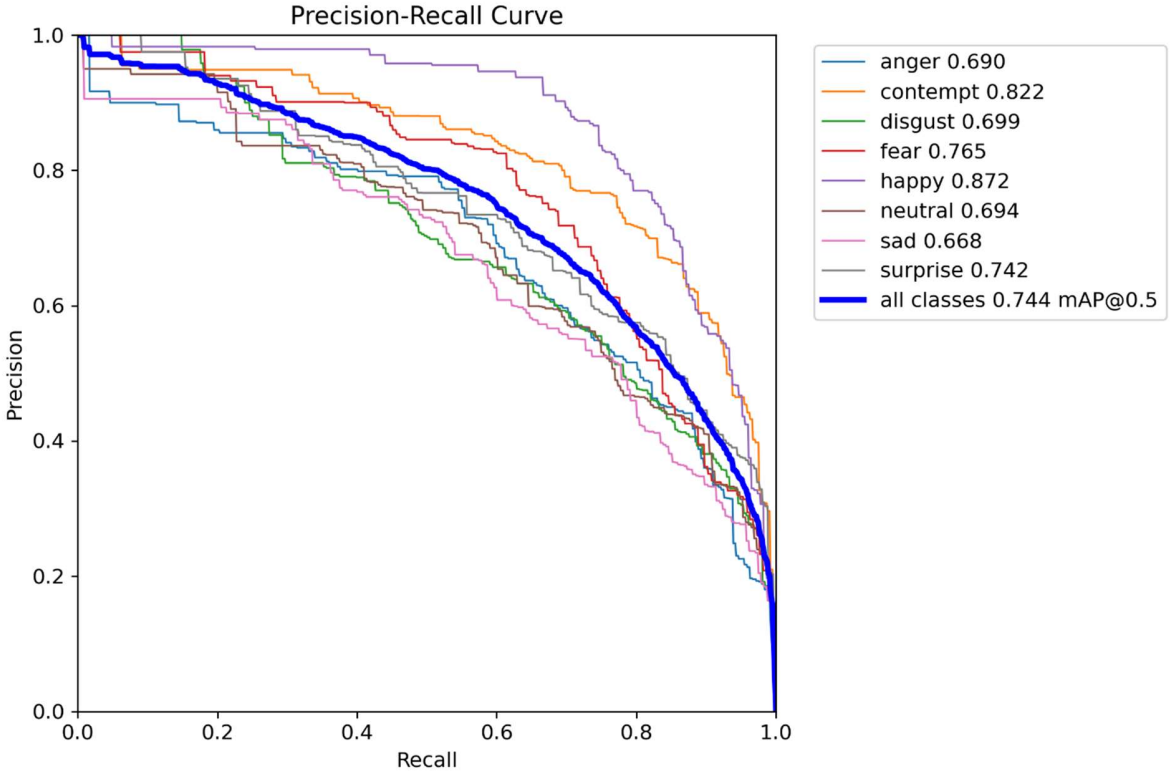



Confusion Matrix Normalized
Predicted
True
anger
contempt
disgust
fear
happy
neutral
sad
surprise
background
0.59 0.01 0.13 0.02 0.01 0.02 0.07 0.01 0.14
0.02 0.75 0.03 0.09 0.07 0.02 0.02 0.09
0.11 0.04 0.63 0.07 0.03 0.04 0.07 0.06 0.16
0.04 0.03 0.64 0.01 0.02 0.11 0.10
0.07 0.02 0.78 0.01 0.01 0.02 0.07
0.08 0.10 0.03 0.03 0.03 0.63 0.16 0.08 0.17
0.13 0.08 0.07 0.01 0.11 0.61 0.03 0.12
0.02 0.02 0.04 0.16 0.02 0.10 0.04 0.66 0.14
0.01 0.01 0.02 0.01
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0


Recall-Confidence Curve
Recall
Confidence
0.0
0.2
0.4
0.6
0.8
1.0
anger
contempt
disgust
fear
happy
neutral
sad
surprise
all classes 0.98 at 0.000

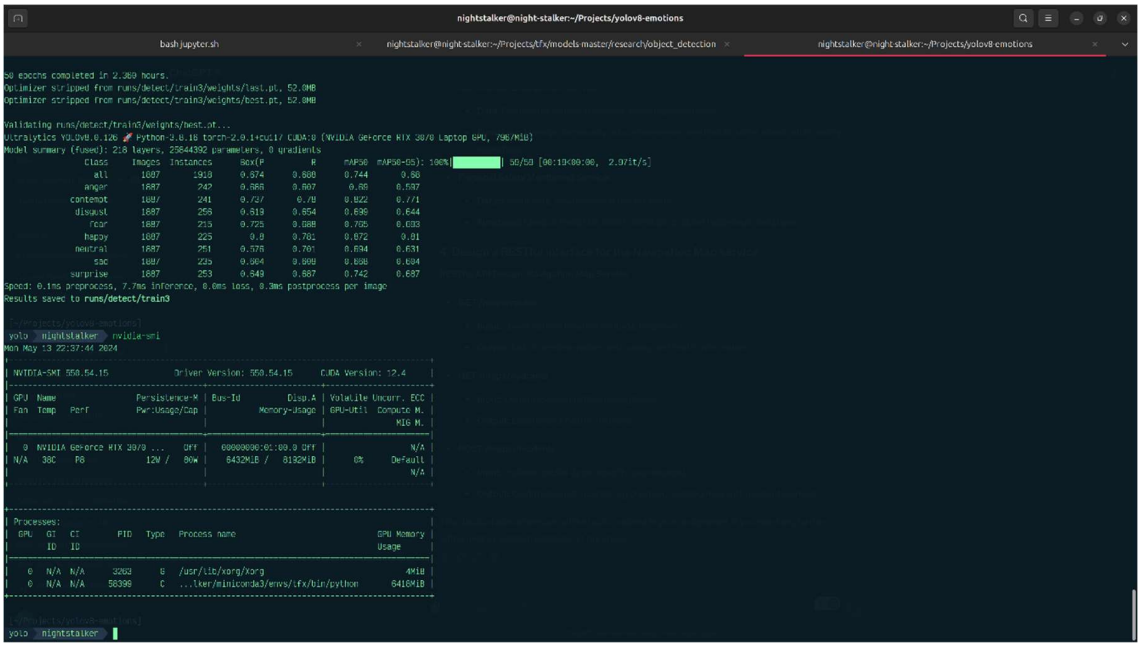

Precision-Confidence Curve
Precision
Confidence
0.0
0.2
0.4
0.6
0.8
1.0
anger
contempt
disgust
fear
happy
neutral
sad
surprise
all classes 1.00 at 0.996


F1-Confidence Curve
F1
Confidence
0.0
0.2
0.4
0.6
0.8
1.0
anger
contempt
disgust
fear
happy
neutral
sad
surprise
all classes 0.68 at 0.460


Precision-Recall Curve
Precision
Recall
0.0
0.2
0.4
0.6
0.8
1.0
anger 0.690
contempt 0.822
disgust 0.699
fear 0.765
happy 0.872
neutral 0.694
sad 0.668
surprise 0.742
all classes 0.744 mAP@0.5


sad
surprise
anger
anger
disgust
neutral
anger
disgust
disgust
disgust
anger
disgust
sad
sad
neutral
anger

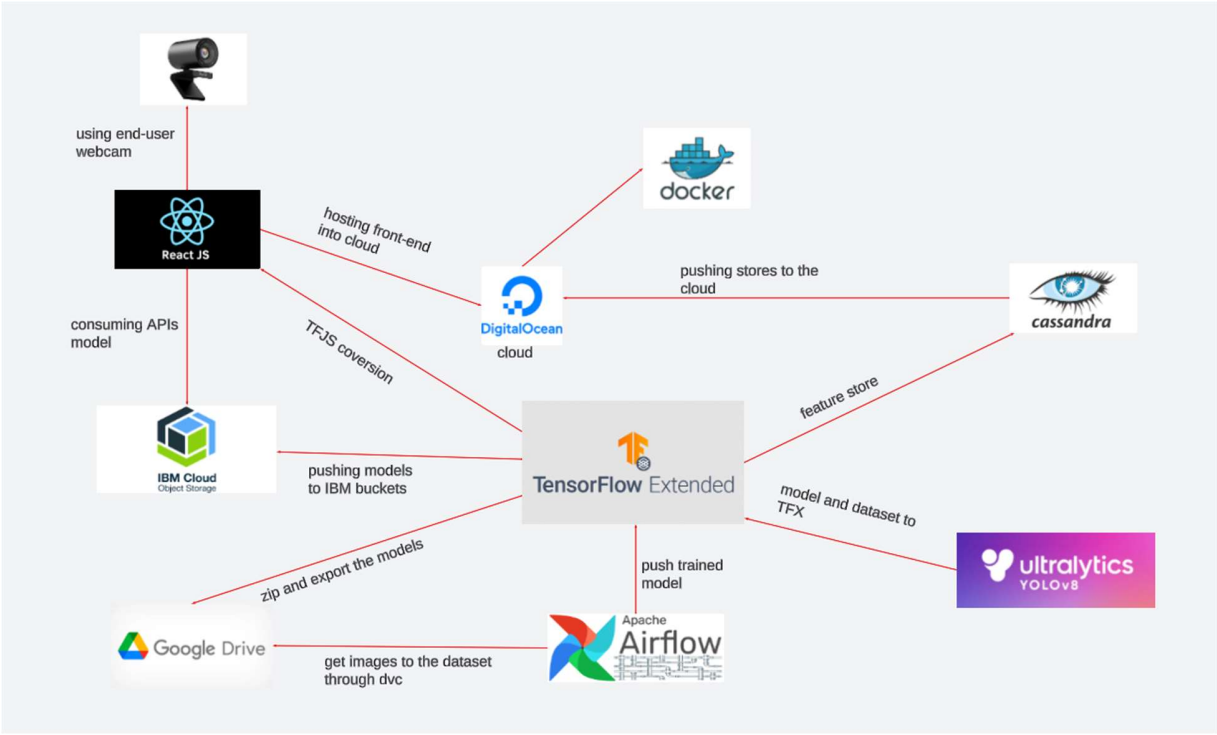
4. **Energy efficiency measurement**
   Monitors and measures the model's energy consumption during training and inference to assess and optimize the system's energy efficiency.



3. Model Deployment (Milestones 5-6, 35%)

3.1 ML System Architecture

- **Drawing with architecture highlights:**



3.2 Application development

- **Model service development:**

```
docker pull tensorflow/serving
docker run -p 8501:8501 --name=my_model_serving -v "/models/model/yolo-v8" -e MODEL_NAME=model tensorflow/serving
```

- **Front-end client development:**

```
function App() {
  const webcamRef = useRef(null);
  const canvasRef = useRef(null);

  // Main function
  const runCoco = async () => {
    const net = await tf.loadGraphModel('https://directionstfod.s3.au-syd.cloud-object-storage.appdomain.cloud/model.json')

    // Loop and detect hands
    setInterval(() => {
      detect(net);
    }, 16.7);
  };
```

3.3 Integration and Deployment

- **Packaging and containerization:**

```
docker build -t my-model-service .
docker run -d -p 8501:8501 my-model-service
```

- **Integration with a CI/CD Pipeline:**

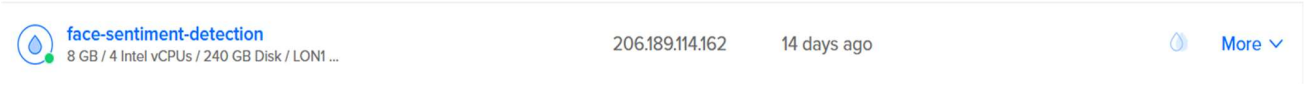
The CI/CD framework, orchestrated with GitHub Actions, is in the deepest of this structure. Running at every single update in data or change in the code base, this automated pipeline will trigger all the related workflows. For orchestration of the various tasks, rebuilding of Docker containers automatically, in case of changes in dependencies, pushing in a registry of the said containers, and deployment in production, the GitHub Actions will be utilized. This integration is key to ensure that the least possible downtime is incurred and that the model benefits from continuous updates with as little human intervention as possible.

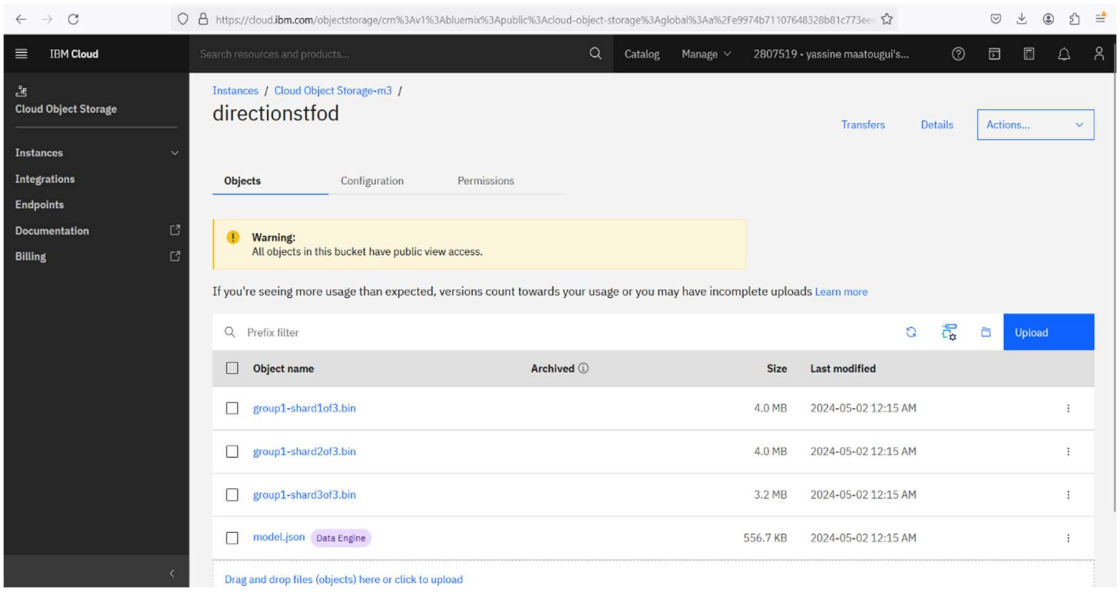
TFX is the project's machine learning backbone, from data ingestion with ExampleGen down to model validation and serving with components such as Evaluator and Pusher. Each of the TFX components is designed within data transformation, model training, and deployment in a clear, reproducible way across environments, which is enforced by containerization in Docker.

In addition, Docker makes TFX even more robust as it wraps each component in a separate container, hence managing dependencies and configurations with simplicity. Furthermore, this scaling in deployment will also lead to isolation of processes, hence increasing safety and stability in the machine-learning pipeline. In essence, orchestration between TFX, Docker, and GitHub Actions is a best practice in MLOps that presents the way forward for the realization of a resilient, scalable, and sustainable infrastructure toward pushing machine learning models. It's a strategic way to achieve model adaptation and improvements in productions while keeping up with the principles of agility and excellence in machine learning deployments.

- **Hosting the application:**

I have hosted the application on Digital Ocean this is the link to my application Object Detection





3.4. Model Serving and online testing

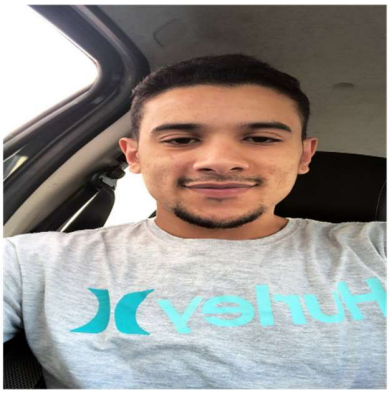
- **Model serving runtime:**
  The model is served using TensorFlow Serving, which is optimized for both high performance and high availability. TensorFlow Serving provides out-of-the-box support for serving ML models, handling requests in real-time with low latency. It's set up to automatically load new versions of the model as they are trained and pushed to the serving directory, ensuring that the most accurate and up-to-date model is always in use.
  https://github.com/yassineMtg/ObjectDetection/blob/main/src/pipeline/pusher.py

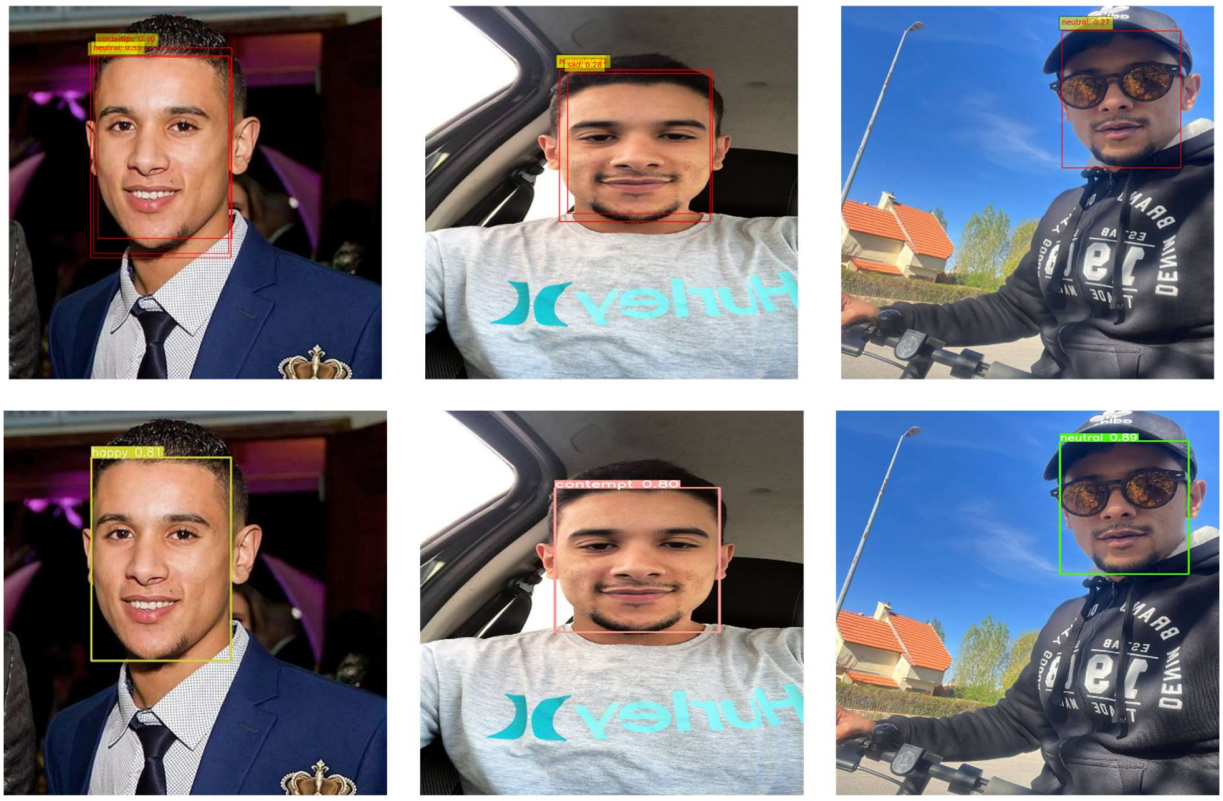
```
pusher = tfx.components.Pusher(
    model=trainer.outputs['model'],
    model_blessing=evaluator.outputs['blessing'],
    push_destination=tfx.proto.PushDestination(
        filesystem=tfx.proto.PushDestination.Filesystem(
            base_directory=_serving_model_dir)))
context.run(pusher, enable_cache=True)
```

- **Serving mode (batch, on demand to a human, on demand to a machine):**

```
def serve_real_time(image):
    """Handle real-time requests."""
    response = requests.post("http://localhost:8501/v1/models/model:predict", json={
    return response.json()
```

- **Online testing (A/B Testing, Bandit):**
  https://github.com/yassineMtg/ObjectDetection/blob/main/src/pipeline/abtesting.py

1. **Detection Quality**:

   - **TensorFlow Object Detection API**: While it provides good detection within its targeted areas (upper face), it may generate multiple bounding boxes that can complicate the output, especially in scenarios needing streamlined and minimal error processing. This could impact its performance negatively if the goal is clear and singular detection per face.
   - **YOLOv8 Detection**: It consistently delivers high-quality detections with a single bounding box per face, closely matching the ground truth. This reliability makes it preferable in applications where accurate full-face detection is critical, such as in security systems or comprehensive emotion analysis.

2. **Model Robustness**:

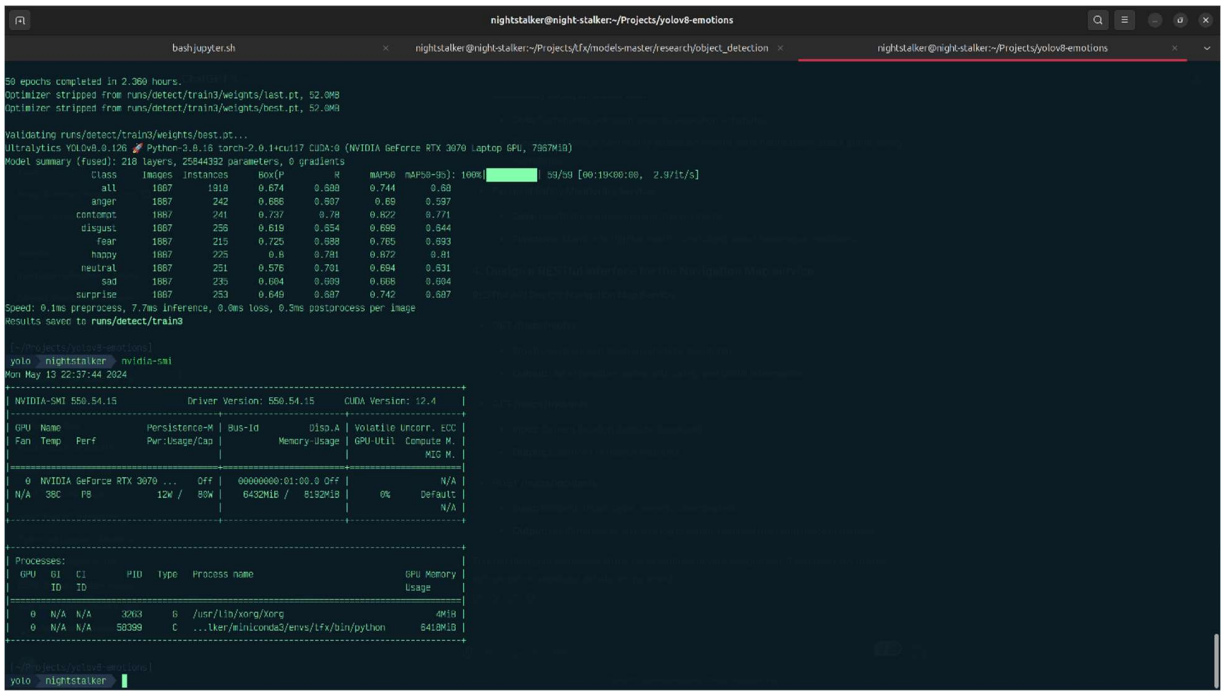
   - Since YOLOv8 provides close-to-ground truth detection without multiple bounding box issues, it suggests greater robustness in varied real-world conditions compared to TensorFlow, which might struggle with overlapping or closely situated faces.

3. **Potential for False Positives/Negatives**:

- The TensorFlow model, due to its tendency to create multiple bounding boxes, might have a higher rate of false positives. This is important in scenarios where precision is more crucial than recall.
- YOLOv8, with its single, accurate bounding box approach, likely exhibits fewer false positives and negatives, providing more reliable results overall.

Monitoring and Continual Learning (milestone 7, 25%)

4.1. (2 pts) Resource Monitoring



- To calculate the energy consumed:

Since power is typically calculated in watts (Joules per second), we need to convert the time from hours to seconds. Then, we multiply the power usage by the time in seconds to get the total energy in Joules.

Given:

$$Power\ (P)\ =\ 80\ watts$$

$$Time\ (T)\ =\ 2.36\ hours$$

Conversion:

$$Time\ in\ seconds\ =\ (\ 2.36\ \times 3600\ )\ seconds/hour\ =\ 8496\ seconds$$

Energy Consumption:

$$Totalenergy(E) = Power \times Time = (80\,\text{watts} \times 8496\,\text{seconds})$$

Calculate the Total Energy:

$$E = 80\,\text{W} \times 8496\,\text{s} = 679{,}680\,\text{Joules}$$

This is the energy consumed in Joules. If we wish to convert this into more common energy billing units like kilowatt-hours (kWh), you can use the $conversion(1\,\text{kWh} = 3{,}600{,}000\,\text{J})$.

Conversion to kWh:

$$\text{kWh} = \frac{679{,}680\,\text{J}}{3{,}600{,}000\,\text{J/kWh}} \approx 0.189\,\text{kWh}$$

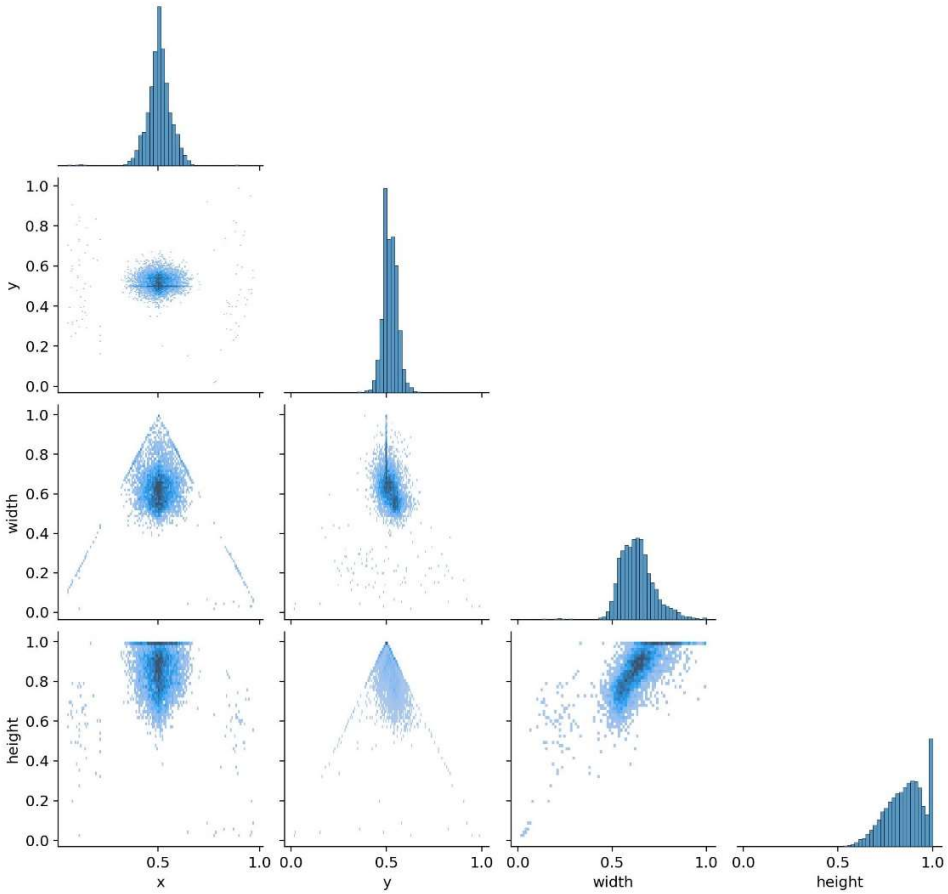
Thus, we consumed approximately 0.189 kWh of energy during the training session.

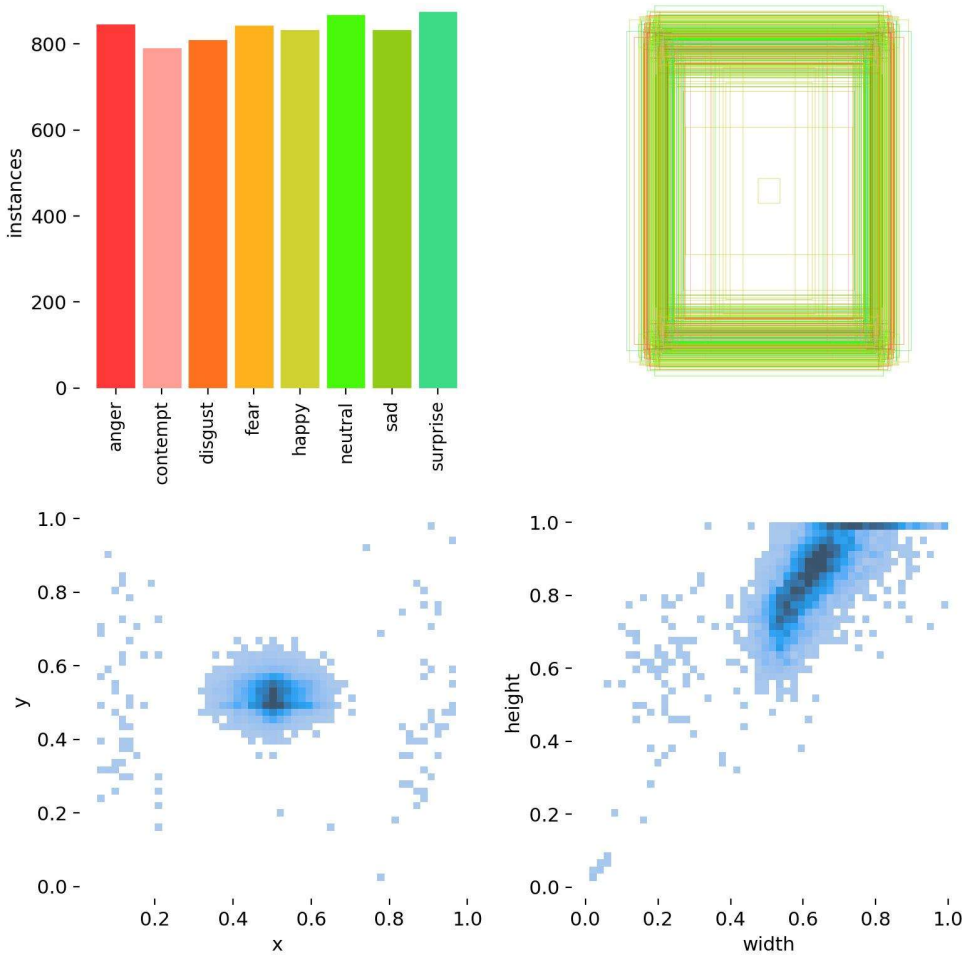
Cost Calculation:

If you want to know how much this cost, you'd multiply the kWh by the cost per kWh from your local utility provider. For example, if the cost is $0.12 per kWh:

$$\text{Cost} = 0.189\,\text{kWh} \times 0.12\,\text{per kWh} = 0.023$$

4.2. (10 pts) Model Performance Monitoring or data distribution drift monitoring





To ensure sustained model performance, continuous monitoring for model degradation and data drift is implemented. We utilize Evidently AI to detect significant shifts in data distribution and monitor predictive performance, helping to maintain the model's accuracy over time.

```python
# dashboard for monitoring data drift
from evidently.dashboard import Dashboard
from evidently.dashboard.tabs import DataDriftTab

data_drift_dashboard = Dashboard(tabs=[DataDriftTab()])
data_drift_dashboard.calculate(reference_data, production_data, column_mapping=None)
data_drift_dashboard.save('/save/data_drift/dashboard.html')
```

4.3. (10 pts) Continual Learning: CT/CD pipeline

The CT/CD pipeline ensures the model is continuously updated with new data, automatically retraining, and deploying without manual intervention. I used GitHub Actions

to automate the retraining process whenever new data is available or periodically based on a schedule.

ct.yaml

```yaml
name: CI/CD Pipeline for YOLOv8 Model Retraining

on:
  push:
    branches:
      - main
    paths:
      - "data/**"

jobs:
  train-yolov8:
    runs-on: ubuntu-latest
    container:
      image: ultralytics/yolov8:latest # Ensure you are using the correct Docker image

    steps:
      - name: Checkout repository
        uses: actions/checkout@v2

      - name: Set up Python environment
        uses: actions/setup-python@v2
        with:
          python-version: "3.9"

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

      - name: Sync data
        run: |
          echo "Syncing dataset..."
          rsync -avz your-data-storage:/path/to/data ./data

      - name: Train YOLOv8 model
        run: |
          echo "Running YOLOv8 training script..."
          python train.py --img 640 --batch 16 --epochs 50 --data ./data/dataset.yaml --weights best.pt --cache

      - name: Save trained model
        run: |
          echo "Saving trained model..."
          cp runs/train/exp/weights/best.pt ./models/

      - name: Deploy model
        if: ${{ github.ref == 'refs/heads/main' }}
        run: |
          echo "Deploying model..."
          scp ./models/best.pt deploy@nightstalker:/path/to/models/
          ssh deploy@nightstalker 'bash deploy_new_model.sh'
```

4.4. (3 pts) Pipeline orchestration

Apache Airflow orchestrates the workflow of the ML pipeline, managing tasks like data ingestion, preprocessing, model training, evaluation, and deployment. This coordination ensures that ML pipeline operates smoothly and efficiently.

```python
default_args = {
    'owner': 'ml_team',
    'depends_on_past': False,
    'start_date': datetime(2024, 05, 01),
    'email': ['y.maatougui@aui.ma'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}

dag = DAG('ml_workflow',
          default_args=default_args,
          description='Machine Learning workflow',
          schedule_interval=timedelta(days=1))

t1 = PythonOperator(task_id='train_model',
                    python_callable=train_model,
                    dag=dag)

t2 = PythonOperator(task_id='deploy_model',
                    python_callable=deploy_model,
                    dag=dag)

t1 >> t2
```

5. Responsible AI (milestone 8-optional, for later, 15% bonus)

5.1 Evaluation Beyond Accuracy

- **Audit Model for Bias:**
  To ensure the Face Sentiment Detection model operates fairly across different demographic groups, we audited the model for bias using the Aequitas toolkit.

  Steps:

  - Data Preparation: We prepared the evaluation data by including the model's predictions and the associated sensitive attributes such as gender and race.
  - Bias Evaluation with Aequitas:
    - Setup: Installed Aequitas using “conda install aequitas”

- Preprocessing: Preprocessed the data to fit the Aequitas input requirements.
- Bias Report Generation: Generated bias reports to evaluate potential disparities. Here's an example of how we used Aequitas:

```python
import pandas as pd
from aequitas.group import Group
from aequitas.preprocessing import preprocess_input_df

# Load evaluation data
eval_data = pd.read_csv('../data/processed/_annotations.csv')

# Add predictions to the data
eval_data['predictions'] = model.predict(eval_data[feature_columns])

# Preprocess data for Aequitas
input_data = preprocess_input_df(eval_data)

# Perform bias audit
g = Group()
xtab, _ = g.get_crosstabs(input_data)
bias_report = g.get_disparity_predefined_groups(
    xtab, original_df=input_data
)

# Save bias report
bias_report.to_csv('bias_report.csv')
```

By using Aequitas, we identified and addressed any biases in our model's predictions, ensuring it performs equitably across all user groups.

- **Model Explainability and Interpretability:**

  To ensure our model's decisions are transparent and interpretable, we utilized SHAP (SHAPLEY Additive EXPLANATIONS) and LIME (Local Interpretable Model-agnostic Explanations).

  Steps:
  - Setup: Installed SHAP using "conda install shap".
  - Explanation: Used SHAP to calculate and visualize the feature importance for individual predictions.

```python
import shap
import tensorflow as tf

# Load evaluation data
eval_data = ... # Load your evaluation data

# Load model
model = tf.keras.models.load_model('../models/yolo-v8')

# Create SHAP explainer
explainer = shap.KernelExplainer(model.predict, eval_data)

# Calculate SHAP values
shap_values = explainer.shap_values(eval_data)

# Plot SHAP values
shap.summary_plot(shap_values, eval_data)
```

6. Conclusion

- **Summary of Achievements:**
  Developed a sophisticated ML system for face sentiment detection, leveraging advanced models and a robust pipeline, with high accuracy in detecting emotional states.

- **Lessons Learned:**
  Emphasized the critical role of high-quality data and the impact of efficient data pipeline management in achieving high model performance.

- **Future Directions :**
  Plan to enhance the model's capability to recognize a broader range of emotions and subtle expressions, integrate newer and more efficient ML models, and expand the application areas of the system.

References

5. **TensorFlow Documentation**

   TensorFlow is extensively used for model training and serving in your project. The official TensorFlow documentation provides comprehensive insights into its APIs, functionalities, and best practices.

   - Website: TensorFlow

6. **TFX User Guide**

   TFX (TensorFlow Extended) is critical for implementing the full ML pipeline in your project. This guide covers all components of TFX, including setup, usage, and examples.

   - Website: TFX User Guide

7. **Prometheus Documentation**

   Used for monitoring system metrics within my project, Prometheus's documentation offers details on setup, configuration, and query execution.

   - Website: Prometheus

8. **Evidently AI**

   For monitoring model performance and data distribution drift, evidently AI's tools are used. Their documentation provides guidelines on how to integrate their tools with existing machine learning pipelines.

   - Website: Evidently AI

9. **Apache Airflow Documentation**

   As the orchestration tool in your project, understanding Airflow is crucial. This documentation provides information on how to define, execute, and monitor DAGs.

   - Website: Apache Airflow

10. **GitHub Actions Documentation**

    For implementing CI/CD pipelines that facilitate continual learning and deployment, GitHub Actions is a core component. This resource offers comprehensive guides on how to set up and manage workflows.

- Website: GitHub Actions

11. **Docker Documentation**

    Docker is used for creating, deploying, and running applications by using containers. The documentation provides a solid understanding of Docker commands and best practices.

    - Website: Docker

12. **Cassandra Documentation**

    Since Cassandra is used for storing features, the official documentation can help understand its data model, query language (CQL), and setup procedures.

    - Website: Apache Cassandra

13. **TensorFlow Serving with Docker**

    For deploying models into production, TensorFlow Serving with Docker provides a robust and scalable framework. This article offers a deep dive into configuration and optimization.

    - Website: Serving TensorFlow Models