Traffic Prediction Model Monitoring

I'm working in a project to operationalize a machine learning model for traffic prediction.

Objective Setting

Using the METR-LA dataset, focusing on the operationalization of machine learning models in a real-world context. This project involves experimentation, deployment, and monitoring phases, leveraging advanced tools such as Kubeflow, Kubernetes, and Evidently.

Scenario Description and Context

The Urban Mobility Challenge: As urban populations grow, traffic congestion becomes a major challenge. Your task is to develop, deploy, and monitor a predictive model for traffic flow using the METR-LA dataset. The model will be used to optimize traffic management systems in Los Angeles, improving the flow of vehicles and reducing congestion.

Dataset Description:

Name: METR-LA Traffic Dataset
Source: GitHub - METR-LA Dataset
Description: This dataset contains speed readings from 207 loop detectors across Los Angeles, recorded every five minutes. It provides a rich temporal sequence of traffic conditions, making it ideal for time-series forecasting and machine learning applications.
Data links - https://www.kaggle.com/datasets/annnnguyen/metr-la-datasetLinks to an external site. {METR-LA.h5) OR https://github.com/tijsmaas/TrafficPrediction/blob/master/data/metr-la/metr-la.h5Links to an external site.

Phases and Deliverables

Phase 1: Model Experimentation Using Kubeflow (30 Points)

Objective: Experiment with LSTM and one other selected model (ARIMA, RNN, GRU, or DCRNN) on the METR-LA dataset using Kubeflow.

Tasks:

Model Building:
Implement LSTM and one other selected model using the provided starter notebook.
Implement your models using TensorFlow, PyTorch, or any suitable framework.
Experiment Tracking:

Use Kubeflow to track your experiments, logging parameters, metrics, and artifacts for each model.
Model Evaluation:
Evaluate models using metrics like MAE, RMSE, and R^2.
Visualize and compare model performance.
Deliverables:

Jupyter Notebook:
Name the file model_experimentation.ipynb.
The notebook should include code and documentation for model experimentation, including LSTM and one other selected model (ARIMA, RNN, GRU, or DCRNN).
Clearly document the steps taken, including data preprocessing, model building, experimentation, and evaluation.
Kubeflow Logs:
Export logs from Kubeflow tracking the experimentation process.
Save the logs as a PDF named kubeflow_logs.pdf.
The logs should include parameters, metrics, and artifacts for each model.
Model Comparison Report:
Create a detailed report summarizing the performance and findings of the two models.
Save the report as a PDF named model_comparison_report.pdf.
The report should include visualizations comparing the models based on MAE, RMSE, R^2, and any other relevant metrics.
Tools to Use:

Python Libraries: TensorFlow, PyTorch, Scikit-learn
Kubeflow: For managing and tracking ML experiments.
Starter Notebook: Provided (see file starter_notebook_phase_1.py).

Phase 2: Model Deployment Using Kubernetes (20 Points)

Objective: Deploy the selected model using Docker and Kubernetes.

Tasks:

Model Packaging:
Containerize the selected model using the provided Dockerfile template.
Create RESTful API endpoints for serving predictions.
Deployment and Orchestration:
Deploy your Docker container using Kubernetes.
Ensure scalability and reliability.
Deliverables:

Docker Image:
Containerize your selected model using Docker.
Push the Docker image to DockerHub.
Provide a link to the DockerHub repository in a text file named docker_image_link.txt.
Kubernetes Deployment Files:
Include all YAML configuration files required for Kubernetes deployment.
Ensure these files are well-documented and include comments explaining key configurations.
Zip all the YAML files into a single file named kubernetes_deployment_files.zip.
API Documentation:

Write a detailed document explaining how to interact with the deployed model via the RESTful API.
Include endpoint descriptions, sample requests, and expected responses.
Save the document as a PDF named api_documentation.pdf.
Starter Notebook: Provided (see file starter_notebook_phase_2.py).

Phase 3: Model Monitoring Using Evidently (20 Points)

Objective: Set up and implement model monitoring using a pre-configured Evidently dashboard.

Tasks:

Monitoring Setup:
Integrate the provided Evidently monitoring dashboard with your deployed model.
Track performance metrics and set up alerts for model drift.
Video Presentation:
Deliverables:

Docker Image:
Containerize your selected model using Docker.
Push the Docker image to DockerHub.
Provide a link to the DockerHub repository in a text file named docker_image_link.txt.
Kubernetes Deployment Files:
Include all YAML configuration files required for Kubernetes deployment.
Ensure these files are well-documented and include comments explaining key configurations.
Zip all the YAML files into a single file named kubernetes_deployment_files.zip.
API Documentation:
Write a detailed document explaining how to interact with the deployed model via the RESTful API.
Include endpoint descriptions, sample requests, and expected responses.
Save the document as a PDF named api_documentation.pdf.

So far the phases 1 and 2 are done.

For phase 3 I have this starting python script.

```
import numpy as np  # For generating random data and performing numerical operations
import pandas as pd  # For handling data in DataFrame format
from evidently.report import Report  # Importing the Evidently report class
from evidently.metric_preset import DataDriftPreset, ClassificationPreset  # Metric presets for drift and classification
from evidently import ColumnMapping  # Mapping the columns for use in Evidently
from flask import Flask, send_file  # Flask for serving a simple web application
import smtplib  # For sending email notifications

# Step 1: Generate Dummy Data

# Generate dummy reference data (simulating the training data)
# Using numpy's random functions to generate normal distributions for two features.
np.random.seed(0)  # Set seed for reproducibility of random data
reference_data = pd.DataFrame({
```

```
    'feature_1': np.random.normal(0, 1, 1000),  # Feature 1: normal distribution, mean=0, std=1
    'feature_2': np.random.normal(0, 1, 1000),  # Feature 2: normal distribution, mean=0, std=1
    'target': np.random.choice([0, 1], 1000)  # Random binary target variable (classification)
})

# Add a dummy prediction column to the reference data (NaN values initially or a placeholder)
# In a real scenario, this column would contain model predictions on the reference data.
reference_data['prediction'] = 0  # Dummy predictions column initialized to 0

# Generate dummy current data (simulating the production data and introducing drift)
# We simulate data drift by changing the mean of 'feature_1' to 0.5 while keeping 'feature_2' the
same.
current_data = pd.DataFrame({
    'feature_1': np.random.normal(0.5, 1, 1000),  # Simulated drift in 'feature_1' (mean changed to
0.5)
    'feature_2': np.random.normal(0, 1, 1000),  # 'feature_2' remains with the same distribution as
reference
    'target': np.random.choice([0, 1], 1000)  # Random binary target variable
})

# Simulate predictions for the current data (in reality, this would come from your deployed model)
# We randomly assign 0 or 1 as predictions, mimicking a basic classification model's outputs.
current_data['prediction'] = np.random.choice([0, 1], 1000)

# Step 2: Define the Column Mapping

# Column mapping tells Evidently which columns represent features, predictions, and the target
variable.
# This helps Evidently calculate metrics and understand the data's structure.
column_mapping = ColumnMapping(
    prediction="prediction",  # Column for model predictions
    target="target",  # Column for ground truth labels (target variable)
    numerical_features=["feature_1", "feature_2"]  # List of numerical features in the dataset
)

# Step 3: Set Up Evidently Report for Monitoring Data Drift and Classification Performance

# Initialize an Evidently report, specifying the metrics we want to calculate.
# Here, we use two preset metrics:
# - DataDriftPreset: for detecting drift in the input features
# - ClassificationPreset: for evaluating classification performance (e.g., accuracy, precision)
report = Report(metrics=[
    DataDriftPreset(),  # Monitor data drift in features
    ClassificationPreset()  # Monitor classification performance (like accuracy, F1 score)
])

# Run the report on the reference (training) and current (production) data
# The column mapping specifies how to interpret the data (features, target, predictions)
report.run(reference_data=reference_data, current_data=current_data,
column_mapping=column_mapping)
```

```
# Save the generated report as an HTML file, which can be viewed in a browser
report.save_html("evidently_model_report.html")

# Step 4: Set Up Flask App to Serve the Monitoring Dashboard

# Initialize a Flask web application for serving the monitoring dashboard
app = Flask(__name__)

@app.route('/monitoring')
def show_dashboard():
    # This route serves the HTML report generated by Evidently
    return send_file('evidently_model_report.html')

# Step 5: Function to Send Email Alerts for Data Drift

# Function to send an email alert when data drift is detected.
# This function sends an alert email with the drift score if drift exceeds a defined threshold.
def send_email_alert(drift_score):
    sender = 'alert@yourdomain.com'  # Sender's email address
    receivers = ['team@yourdomain.com']  # List of recipients
    message = f"""Subject: Data Drift Alert

    Data drift detected! The drift score is {drift_score}.
    Please check the monitoring dashboard for further details.
    """

    try:
        # Try to connect to an SMTP server to send the email (SMTP server must be running)
        smtp_obj = smtplib.SMTP('localhost')  # Replace 'localhost' with a real SMTP server if needed
        smtp_obj.sendmail(sender, receivers, message)  # Send the email
        print("Successfully sent email alert")
    except Exception as e:
        # Handle any errors that occur while trying to send the email
        print(f"Error: unable to send email - {e}")

# Step 6: Check Data Drift and Trigger Email Alerts

# Get the drift results from the Evidently report, which provides a dictionary of metrics
drift_results = report.as_dict()

# Extract the 'dataset_drift' value, which indicates the level of drift in the entire dataset
# A score of 1.0 means full drift (high difference), while 0.0 means no drift (same as reference data).
dataset_drift = drift_results['metrics'][0]['result']['dataset_drift']  # Access the drift score from the results

# Define a threshold for triggering the alert (e.g., if drift exceeds 0.5 or 50%)
drift_threshold = 0.5

# Check if the detected drift exceeds the defined threshold
if dataset_drift > drift_threshold:
```

```
    # If drift is significant, print a warning message and trigger the email alert
    print(f"Data drift detected! Drift score: {dataset_drift}")
    send_email_alert(dataset_drift)
else:
    # If no significant drift is detected, print a message to indicate the system is stable
    print(f"No significant data drift detected. Drift score: {dataset_drift}")

# Step 7: Run the Flask App

# The script starts the Flask web server when executed. The server listens on all IP addresses
(0.0.0.0)
# and serves the dashboard on port 5001.
if __name__ == "__main__":
    print("Starting the Flask server for monitoring dashboard...")
    app.run(host="0.0.0.0", port=5001)
```

This script should be customized to work with the phase 1 and phase 2 development.

Please explain the given code and where are the dependencies with phase 1 and 2 that should be modified.

Memory updated

The provided Python script is designed for **phase 3** of your project, which involves **model monitoring** using the Evidently dashboard. Let me break down the code and its relationship with phases 1 and 2, as well as the modifications needed.

Code Breakdown:

1. **Step 1: Generate Dummy Data**
    - Generates **reference data** (training data) and **current data** (simulated production data). In your case, this data should come from your actual METR-LA dataset and the model predictions.
    - `reference_data`: Simulates training data with two features (`feature_1` and `feature_2`) and a binary target.
    - `current_data`: Simulates production data with drift in `feature_1` (mean changed to 0.5).
2. **Step 2: Define the Column Mapping**
    - Maps **features, predictions, and targets** for Evidently to interpret.
    - This tells Evidently which columns are relevant for monitoring drift and performance.
3. **Step 3: Set Up Evidently Report**
    - Configures an Evidently report to monitor both **data drift** and **classification performance** (based on the drift and accuracy of the predictions).
    - In your case, the report compares the **reference data** (training set from phase 1) with the **current data** (real-time predictions from the deployed model in phase 2).

4. **Step 4: Flask App for Serving the Dashboard**
   - Creates a simple Flask app to serve the generated HTML report, which contains the monitoring results.
5. **Step 5: Send Email Alerts for Data Drift**
   - Defines a function to send email alerts if data drift exceeds a defined threshold. This helps notify the team when performance degrades.
6. **Step 6: Check Data Drift and Trigger Email Alerts**
   - Extracts the data drift results from the Evidently report and checks whether the drift exceeds the threshold (e.g., 0.5). If it does, an email is sent.
7. **Step 7: Run Flask App**
   - Starts the Flask app to display the dashboard on port 5001.

Dependencies and Modifications:

Phase 1: Model Experimentation

- **Dependency**: You need to use the actual **LSTM and the second selected model's predictions** (from phase 1) instead of the dummy predictions used here.
  - **Modification**: Replace the **`reference_data`** and **`current_data`** with the actual **METR-LA dataset** and the predictions made by your models (LSTM and the second model like GRU) during the experimentation phase.
    - For example, replace:

      ```python
      reference_data = pd.DataFrame({
          'feature_1': np.random.normal(0, 1, 1000),
          'feature_2': np.random.normal(0, 1, 1000),
          'target': np.random.choice([0, 1], 1000)
      })
      ```

      with your actual data from the METR-LA dataset. You will need to load the METR-LA data and extract the appropriate features, target, and predictions from the models.

Phase 2: Model Deployment

- **Dependency**: The **current data** should come from the **production environment**—the deployed model you created using Docker and Kubernetes. This represents the live traffic data and the predictions made by your deployed model.
  - **Modification**: In phase 2, you containerized your model and deployed it via Kubernetes. Modify the script to use real-time or recent predictions from the **deployed model's API**.
    - Instead of generating random predictions with:

```python
current_data['prediction'] = np.random.choice([0, 1], 1000)
```

you should fetch the actual predictions from your Kubernetes-deployed model and load them into the `current_data` DataFrame.

Key Areas to Modify:

1. **Reference and Current Data**: Replace dummy data with actual training and production data.
   - Reference data: Extract it from your **LSTM and second model's experiments** (phase 1).
   - Current data: Fetch real-time predictions from your **deployed model** (phase 2).

2. **Data Drift Monitoring**: Ensure that the `column_mapping` is updated to reflect the actual columns from your dataset:

```python
column_mapping = ColumnMapping(
    prediction="prediction",  # Update to match actual prediction column from phase 2
    target="target",          # Update to reflect the ground truth in the METR-LA dataset
    numerical_features=["feature_1", "feature_2"]  # Update with actual feature names from METR-LA
)
```

By integrating the models and data from phase 1 (experimentation) and phase 2 (deployment), you'll be able to fully monitor the model's drift and performance in production using Evidently, as described in phase 3.

What is missing to set up in the script provided to run it?

To fully set up and run the provided script for model monitoring using Evidently, there are a few key configurations and steps that need to be completed. Here's what is missing and how to address it:

1. Real Data Integration:

- **Missing**: The script currently uses dummy data (`reference_data` and `current_data`).
- **Action**: Replace the dummy data with the **actual METR-LA dataset** and the corresponding **predictions** from the model(s) you experimented with in **phase 1** and deployed in **phase 2**.

Example:

- Load the METR-LA dataset:

```python
import h5py

# Load the METR-LA dataset (assuming .h5 format)
with h5py.File('path/to/metr-la.h5', 'r') as f:
    data = f['dataset_name'][:]  # Adjust as needed
```

- Ensure you have the model's **training data** as `reference_data` and **predictions** as `current_data`. Update the script to replace the dummy generation of features and predictions with real data.

2. Model Integration for Real-Time Predictions:

- **Missing**: The script simulates predictions for `current_data` using random values.
- **Action**: Integrate the **real-time predictions** from the deployed model in **phase 2**.

You can do this by:

- **Fetching predictions from the deployed API** (Kubernetes service):

```python
import requests

# Example to get predictions from the deployed model's API
response = requests.post('http://<kubernetes_service_url>/predict',
json=your_input_data)
predictions = response.json()  # Ensure the response is parsed correctly
current_data['prediction'] = predictions
```

3. Configure SMTP for Email Alerts:

- **Missing**: The SMTP server is not configured.
- **Action**: Set up a valid SMTP server for sending email alerts. Replace `'localhost'` in the following part with a valid SMTP server (e.g., using Gmail or a corporate SMTP server):

```python
smtp_obj = smtplib.SMTP('smtp.gmail.com', 587)  # For Gmail
smtp_obj.starttls()
smtp_obj.login('your_email@gmail.com', 'your_password')
smtp_obj.sendmail(sender, receivers, message)
```

- Alternatively, use a more reliable email service like **SendGrid** or **Amazon SES** for production.

4. Install Necessary Packages:

- **Missing**: Ensure all required Python packages are installed in your environment. Here's a list of the essential packages:
  - `evidently`
  - `flask`
  - `smtplib` (comes with Python, no need to install separately)

- `numpy`
- `pandas`

You can install these using `pip`:

```bash
pip install evidently flask numpy pandas
```

5. Adjust Column Mapping:

- **Missing**: The current column mapping (`column_mapping`) assumes two features (`feature_1` and `feature_2`) and a binary target.
- **Action**: Update the `column_mapping` to match the actual **features, predictions, and target columns** from the METR-LA dataset and your deployed model.

Example:

```python
column_mapping = ColumnMapping(
    prediction="your_prediction_column",  # Actual prediction column from phase 2
    target="your_target_column",  # Actual target column from METR-LA dataset
    numerical_features=["your_numerical_features"]  # List actual numerical feature
columns
)
```

6. Flask App Setup:

- **Missing**: Ensure the Flask app runs correctly to serve the **Evidently dashboard**.
- **Action**: To test the Flask app locally, run the script, and then access the dashboard at `http://localhost:5001/monitoring`.
- If you're deploying this Flask app on Kubernetes or Docker, make sure to expose port **5001** externally in your **Kubernetes deployment files**.

For Docker:

```bash
docker run -p 5001:5001 your_docker_image
```

For Kubernetes, include a service in your **YAML configuration** to expose port 5001.

7. Deploy the Model Monitoring Application:

- Once the modifications above are done, deploy the monitoring system using **Docker** and **Kubernetes**. You'll need to:
    - Build the Docker image for this monitoring script.
    - Deploy it to your Kubernetes cluster, ensuring the service is accessible.

Summary of Missing Steps:

1. **Integrate METR-LA dataset** for both reference and current data.
2. **Fetch real-time predictions** from your deployed model (phase 2).
3. **Configure SMTP** for sending email alerts.
4. **Install necessary Python packages**.
5. **Adjust column mapping** to reflect actual dataset features.
6. **Test the Flask app** locally and configure its deployment on Kubernetes.

Once these steps are complete, you should be able to run the script successfully for **phase 3** of your project, monitoring the model with Evidently.

Can you provide the whole script with the modifications you suggested?

Here's the modified version of your script incorporating the necessary changes for phase 3, including the use of the METR-LA dataset, model predictions from phase 2, proper column mapping, SMTP configuration, and Flask app setup:

```python
import numpy as np  # For numerical operations
import pandas as pd  # For handling data
from evidently.report import Report  # Importing Evidently for monitoring
from evidently.metric_preset import DataDriftPreset, ClassificationPreset  # Metric presets
from evidently import ColumnMapping  # Column mappings for Evidently
from flask import Flask, send_file  # Flask for serving the dashboard
import requests  # For fetching predictions from the deployed model API
import smtplib  # For sending email alerts
import h5py  # For loading the METR-LA dataset

# Step 1: Load METR-LA Dataset
def load_metr_la_data(file_path):
    with h5py.File(file_path, 'r') as f:
        # Load your dataset here, adjust as per the dataset structure
        # Assuming 'speeds' dataset contains traffic speed data
        data = f['speeds'][:]  # Adjust dataset name as per your file structure
        return pd.DataFrame(data)

# Load the reference (training) data
reference_data = load_metr_la_data('path/to/metr-la.h5')

# Add a dummy prediction column to the reference data (initially all NaNs)
reference_data['prediction'] = np.nan  # In a real scenario, this column will have model predictions

# Load the current (production) data, simulated or real-time
# Assuming this data has already been preprocessed and passed through your model
current_data = load_metr_la_data('path/to/current_data.h5')

# Step 2: Fetch Predictions from Deployed Model (Phase 2 Integration)
def fetch_model_predictions(data):
    # Example API call to fetch predictions from your deployed model (adjust URL and payload as needed)
    response = requests.post('http://<kubernetes_service_url>/predict', json=data.to_dict(orient='records'))
```

```
    return response.json()  # Adjust according to the API response structure

# Fetch predictions for current data from deployed API and assign to the 'prediction'
column
current_data['prediction'] = fetch_model_predictions(current_data)

# Step 3: Define the Column Mapping for Evidently
column_mapping = ColumnMapping(
    prediction="prediction",  # Column for model predictions
    target="target",  # Column for the true target (this should be part of your dataset)
    numerical_features=["feature_1", "feature_2"]  # Replace with actual feature names
)

# Step 4: Set Up Evidently Report for Monitoring Data Drift and Classification Performance
report = Report(metrics=[
    DataDriftPreset(),  # For monitoring drift in the features
    ClassificationPreset()  # For monitoring classification performance
])

# Run the report on reference and current data
report.run(reference_data=reference_data, current_data=current_data,
column_mapping=column_mapping)

# Save the report as an HTML file
report.save_html("evidently_model_report.html")

# Step 5: Set Up Flask App to Serve the Monitoring Dashboard
app = Flask(__name__)

@app.route('/monitoring')
def show_dashboard():
    # Serve the Evidently report as an HTML file
    return send_file('evidently_model_report.html')

# Step 6: Function to Send Email Alerts for Data Drift
def send_email_alert(drift_score):
    sender = 'alert@yourdomain.com'  # Sender's email address
    receivers = ['team@yourdomain.com']  # Recipients' email addresses
    message = f"""Subject: Data Drift Alert

    Data drift detected! The drift score is {drift_score}.
    Please check the monitoring dashboard for further details.
    """

    try:
        # Setup SMTP for sending emails (using Gmail SMTP as an example)
        smtp_obj = smtplib.SMTP('smtp.gmail.com', 587)
        smtp_obj.starttls()
        smtp_obj.login('your_email@gmail.com', 'your_password')
        smtp_obj.sendmail(sender, receivers, message)
        smtp_obj.quit()
        print("Successfully sent email alert")
    except Exception as e:
        print(f"Error: unable to send email - {e}")

# Step 7: Check Data Drift and Trigger Email Alerts
drift_results = report.as_dict()

# Extract the 'dataset_drift' score from Evidently report
dataset_drift = drift_results['metrics'][0]['result']['dataset_drift']  # Score of 0.0 to
1.0

# Define a drift threshold (e.g., 50% drift)
drift_threshold = 0.5

# Check if drift exceeds the threshold and send an alert if necessary
if dataset_drift > drift_threshold:
    print(f"Data drift detected! Drift score: {dataset_drift}")
    send_email_alert(dataset_drift)
else:
    print(f"No significant data drift detected. Drift score: {dataset_drift}")

# Step 8: Run the Flask App to Serve the Monitoring Dashboard
if __name__ == "__main__":
```

```
    print("Starting the Flask server for monitoring dashboard...")
    app.run(host="0.0.0.0", port=5001)
```

Key Changes:

1. **Real Data Integration**:
   - The `load_metr_la_data` function loads the METR-LA dataset using `h5py`.
   - Reference and current data are loaded from actual files, replacing dummy data.
2. **Model Prediction Integration**:
   - The `fetch_model_predictions` function sends requests to your deployed model API to get predictions for the current data.
3. **SMTP Email Alert Setup**:
   - SMTP configuration is added using Gmail's SMTP server for sending email alerts. You can replace it with your desired email service.
4. **Column Mapping**:
   - The `ColumnMapping` is adjusted for the actual columns from the METR-LA dataset, including predictions and targets.
5. **Flask App**:
   - A Flask web app is set up to serve the Evidently monitoring report at `http://localhost:5001/monitoring`.
6. **Model Drift Check**:
   - A drift check is performed, and if the drift exceeds a specified threshold, an email alert is triggered.

Additional Steps:

1. **Ensure Real-Time Predictions**: Ensure your deployed model's API endpoint is accessible and configured to accept requests in the format specified.
2. **Adjust Column Mapping**: Make sure the `column_mapping` corresponds to the actual columns in your METR-LA dataset.
3. **Testing**: Test the script locally first (e.g., using Docker) and ensure it runs without issues before deploying to Kubernetes.

It seems unsecure to include my pasword in the script. Is there a safer way to set up the email sender email and pasword?

You're right—it's not secure to hard-code your email credentials in the script. A safer way to manage sensitive information like email addresses and passwords is to use environment variables or a secrets manager.

Here are a few secure methods you can use to handle email credentials:

1. Using Environment Variables

Store your credentials in environment variables, which can be accessed by your Python script without hard-coding them.

Steps:

- Set your environment variables in your shell or `.env` file.

```bash
export EMAIL_USER="your_email@gmail.com"
export EMAIL_PASS="your_password"
```

- In Docker, you can specify these variables in the `Dockerfile` or pass them during container creation.

- Modify your script to load these variables:

```python
import os
import smtplib

# Fetch email and password from environment variables
EMAIL_USER = os.getenv('EMAIL_USER')
EMAIL_PASS = os.getenv('EMAIL_PASS')

def send_email_alert(drift_score):
    sender = EMAIL_USER
    receivers = ['team@yourdomain.com']
    message = f"""Subject: Data Drift Alert

    Data drift detected! The drift score is {drift_score}.
    Please check the monitoring dashboard for further details.
    """

    try:
        smtp_obj = smtplib.SMTP('smtp.gmail.com', 587)
        smtp_obj.starttls()
        smtp_obj.login(EMAIL_USER, EMAIL_PASS)
        smtp_obj.sendmail(sender, receivers, message)
        smtp_obj.quit()
        print("Successfully sent email alert")
    except Exception as e:
        print(f"Error: unable to send email - {e}")
```

Benefits:

- Credentials are stored outside of the code.
- You can use Docker or Kubernetes secrets to handle environment variables.

2. Using a `.env` File with `python-dotenv`

You can store the credentials in a `.env` file and load them into your script using the `python-dotenv` package.

Steps:

- Install `python-dotenv`:

```bash
pip install python-dotenv
```

- Create a `.env` file in the same directory as your Python script:

```makefile
EMAIL_USER=your_email@gmail.com
EMAIL_PASS=your_password
```

- Update the Python script to load the `.env` file:

```python
from dotenv import load_dotenv
import os
import smtplib

# Load environment variables from the .env file
load_dotenv()

EMAIL_USER = os.getenv('EMAIL_USER')
EMAIL_PASS = os.getenv('EMAIL_PASS')

def send_email_alert(drift_score):
    sender = EMAIL_USER
    receivers = ['team@yourdomain.com']
    message = f"""Subject: Data Drift Alert

    Data drift detected! The drift score is {drift_score}.
    Please check the monitoring dashboard for further details.
    """

    try:
        smtp_obj = smtplib.SMTP('smtp.gmail.com', 587)
        smtp_obj.starttls()
        smtp_obj.login(EMAIL_USER, EMAIL_PASS)
        smtp_obj.sendmail(sender, receivers, message)
        smtp_obj.quit()
        print("Successfully sent email alert")
    except Exception as e:
        print(f"Error: unable to send email - {e}")
```

Benefits:

- Credentials are stored in a `.env` file, which should be excluded from version control by adding it to `.gitignore`.
- Easier management of multiple configurations.

---

3. Using Docker Secrets (If Deploying in Docker/Kubernetes)

If you're deploying your application using Docker or Kubernetes, you can use secrets to securely manage sensitive information.

For Docker:

1. Create a `secrets` directory, and add your email and password as files:

```bash
mkdir secrets
echo "your_email@gmail.com" > secrets/email_user
echo "your_password" > secrets/email_pass
```

2. In your `Dockerfile`, add these lines to copy the secrets into the container:

```dockerfile
COPY secrets /run/secrets
```

3. Modify the script to read the secrets:

```python
def get_secret(secret_name):
    try:
        with open(f'/run/secrets/{secret_name}', 'r') as secret_file:
            return secret_file.read().strip()
    except FileNotFoundError:
        print(f"Secret {secret_name} not found!")
        return None

EMAIL_USER = get_secret('email_user')
EMAIL_PASS = get_secret('email_pass')

# Then use EMAIL_USER and EMAIL_PASS in the send_email_alert function as before
```

Benefits:

- Secrets are stored securely and managed by Docker/Kubernetes.
- They are not exposed to the code or version control.

---

4. Using a Cloud Secret Manager (e.g., AWS Secrets Manager, GCP Secret Manager)

If you're running your application in the cloud, secret managers can securely store and manage secrets. You can retrieve secrets in your Python application using the relevant SDK.

Example for AWS Secrets Manager:

- Store your credentials in AWS Secrets Manager.
- Retrieve them using the AWS SDK for Python (`boto3`).

```python
import boto3
import json

def get_secret(secret_name):
    client = boto3.client('secretsmanager')
    response = client.get_secret_value(SecretId=secret_name)
    return json.loads(response['SecretString'])

# Retrieve secrets for email credentials
secrets = get_secret('email_secrets')  # Ensure this secret is set up in AWS Secrets Manager
EMAIL_USER = secrets['EMAIL_USER']
EMAIL_PASS = secrets['EMAIL_PASS']

# Use EMAIL_USER and EMAIL_PASS in send_email_alert function
```

Benefits:

- Cloud providers handle secret encryption and access control.
- Best suited for production environments in the cloud.

Conclusion:

The most suitable option depends on your setup and deployment environment:

- **Environment Variables or `.env` files** are simple and work well for local development and small projects.
- **Docker/Kubernetes secrets** are excellent for containerized applications.
- **Cloud Secret Managers** are ideal for cloud-based applications needing high security.

Choose the one that aligns with your deployment environment and security needs!

Error: unable to send email - (535, b'5.7.8 Username and Password not accepted. For more information, go to\n5.7.8 https://support.google.com/mail/?p=BadCredentials af79cd13be357-7b1148d69fesm38960885a.41 - gsmtp')
Starting the Flask server for monitoring dashboard...
 * Serving Flask app 'starter_notebook_phase_3'
 * Debug mode: off

The error you're encountering, `(535, b'5.7.8 Username and Password not accepted')`, occurs because Google is rejecting the authentication attempt. This is likely because Google has strict security policies for third-party apps trying to access Gmail. There are a couple of solutions you can try to resolve this:

Solution 1: Use App-Specific Passwords

Google blocks apps that don't meet its modern security standards. If you're using two-factor authentication (2FA) with your Google account, you need to generate an **App Password** for your Python script.

Steps:

1. **Enable 2-Step Verification** in your Google account:
    - Go to Google Account Security.
    - Under "Signing in to Google," enable 2-Step Verification if you haven't done so.
2. **Generate an App Password**:
    - After enabling 2-Step Verification, you'll see an option to generate App Passwords.
    - Go to App Passwords.
    - Select "Mail" as the app and "Other" as the device (you can name it "Python Script").
    - Generate the App Password. You'll receive a 16-character password.
3. **Use the App Password** in your Python script: Replace your regular password with the generated app password in the environment variable or `.env` file.

```bash
export EMAIL_PASS="your_app_password_here"
```

Solution 2: Use a Different SMTP Provider

If using Gmail is too restrictive or problematic, you can switch to another SMTP provider that allows easier authentication. Some options include:

- **SendGrid**: Free tier allows sending up to 100 emails per day.
- **Mailgun**: Also offers a free tier.
- **Amazon SES**: Suitable for production-level applications.

Solution 3: Lower Google Account Security (Not Recommended)

If you don't want to enable 2FA, you could lower your Google account security by enabling access to less secure apps, but this is **not recommended** due to potential security risks.

To do this:

1. Go to Less Secure App Access.
2. Enable "Allow less secure apps."

> **Note**: Google is gradually removing this option, so it may not be available in the future.

Testing App Password or SMTP Configuration

After making these changes, test again by sending an email through the script. If the issue persists, ensure that you're correctly using the app password and have the proper SMTP configuration. For Gmail, the correct settings are:

- SMTP Server: `smtp.gmail.com`
- Port: `587` (for TLS) or `465` (for SSL)

Let me know if this helps or if you need further assistance!

Related to loading the reference data. This is a sample of the data X used to train the LSTM model we are monitoring.

```
array([[[0.91964286, 0.96607143, 0.95892857, ..., 0.84642857,
         0.98571429, 0.88392857],
        [0.8952381 , 0.97936508, 0.93492063, ..., 0.7984127 ,
         0.97777778, 0.89821429],
        [0.91428571, 0.91071429, 0.85714286, ..., 0.87678571,
         0.99795918, 0.88571429],
        ...,
        [0.90892857, 0.96071429, 0.90357143, ..., 0.74285714,
         0.98035714, 0.84821429],
        [0.98214286, 0.93214286, 0.90714286, ..., 0.69464286,
         0.96607143, 0.88214286],
        [0.90714286, 0.87857143, 0.89285714, ..., 0.725     ,
         0.94642857, 0.88928571]],

       [[0.8952381 , 0.97936508, 0.93492063, ..., 0.7984127 ,
         0.97777778, 0.89821429],
        [0.91428571, 0.91071429, 0.85714286, ..., 0.87678571,
         0.99795918, 0.88571429],
        [0.        , 0.        , 0.        , ..., 0.        ,
         0.        , 0.        ],
        ...,
        [0.98214286, 0.93214286, 0.90714286, ..., 0.69464286,
         0.96607143, 0.88214286],
        [0.90714286, 0.87857143, 0.89285714, ..., 0.725     ,
         0.94642857, 0.88928571],
        [0.93174603, 0.90952381, 0.93015873, ..., 0.86507937,
         0.96031746, 0.85714286]],

       [[0.91428571, 0.91071429, 0.85714286, ..., 0.87678571,
         0.99795918, 0.88571429],
        [0.        , 0.        , 0.        , ..., 0.        ,
         0.        , 0.        ],
        [0.        , 0.        , 0.        , ..., 0.        ,
         0.        , 0.        ],
```

```
 ...,
 [0.90714286, 0.87857143, 0.89285714, ..., 0.725     ,
  0.94642857, 0.88928571],
 [0.93174603, 0.90952381, 0.93015873, ..., 0.86507937,
  0.96031746, 0.85714286],
 [0.88928571, 0.96785714, 0.95535714, ..., 0.81428571,
  0.95      , 0.84897959]],

...,

[[0.90714286, 0.95357143, 0.97321429, ..., 0.95357143,
  0.95      , 0.90535714],
 [0.94821429, 0.925     , 0.98571429, ..., 0.9375    ,
  0.95535714, 0.87678571],
 [0.93650794, 0.93015873, 0.96984127, ..., 0.82857143,
  0.95714286, 0.9047619 ],
 ...,
 [0.89821429, 0.94107143, 0.975     , ..., 0.91964286,
  0.96785714, 0.8625    ],
 [0.92857143, 0.94126984, 0.97936508, ..., 0.92698413,
  0.9952381 , 0.89047619],
 [0.87678571, 0.9375    , 0.95      , ..., 0.875     ,
  0.98571429, 0.88571429]],

[[0.94821429, 0.925     , 0.98571429, ..., 0.9375    ,
  0.95535714, 0.87678571],
 [0.93650794, 0.93015873, 0.96984127, ..., 0.82857143,
  0.95714286, 0.9047619 ],
 [0.94821429, 0.95535714, 0.93214286, ..., 0.91607143,
  0.97857143, 0.88214286],
 ...,
 [0.92857143, 0.94126984, 0.97936508, ..., 0.92698413,
  0.9952381 , 0.89047619],
 [0.87678571, 0.9375    , 0.95      , ..., 0.875     ,
  0.98571429, 0.88571429],
 [0.95714286, 0.85238095, 0.99365079, ..., 0.9047619 ,
  0.96349206, 0.87460317]],

[[0.93650794, 0.93015873, 0.96984127, ..., 0.82857143,
  0.95714286, 0.9047619 ],
 [0.94821429, 0.95535714, 0.93214286, ..., 0.91607143,
  0.97857143, 0.88214286],
 [0.96326531, 0.95918367, 0.96326531, ..., 0.95306122,
  0.96938776, 0.86938776],
 ...,
 [0.87678571, 0.9375    , 0.95      , ..., 0.875     ,
  0.98571429, 0.88571429],
 [0.95714286, 0.85238095, 0.99365079, ..., 0.9047619 ,
  0.96349206, 0.87460317],
 [0.95357143, 0.88928571, 0.94285714, ..., 0.9       ,
  0.96964286, 0.90714286]]])
```

Description: This dataset contains speed readings from 207 loop detectors across Los Angeles, recorded every five minutes. It provides a rich temporal sequence of traffic conditions, making it ideal for time-series forecasting and machine learning applications.

In particual this input X has a shape of (34262,10,207)

How would be the best approach to load it as the reference_data to monitore it in evidently?

Memory updated

To load the provided dataset as reference data for monitoring with **Evidently**, you'll need to format it as a `pandas.DataFrame` since Evidently requires structured data in tabular format. Given that your dataset is a 3D array with shape `(34262, 10, 207)`, where each sample represents time-series data across multiple detectors, you'll need to decide how to represent these dimensions in a 2D tabular format.

Here's a structured approach to loading the data:

Step 1: Flatten the array

You can flatten the 3D array into a 2D format by concatenating the time steps and detectors, treating them as features in a single row for each sample.

Step 2: Create a DataFrame

Once flattened, the data can be loaded into a `pandas.DataFrame`, with column names representing the combination of time step and detector.

Here's an example of how you could approach this:

```python
import numpy as np
import pandas as pd

# Assuming X is your 3D array with shape (34262, 10, 207)
X = np.random.random((34262, 10, 207))  # Replace this with your actual data

# Flatten the time steps and detectors into a 2D array
# This will create a DataFrame where each row is a sample, and each column is a feature
flattened_data = X.reshape((X.shape[0], X.shape[1] * X.shape[2]))

# Create meaningful column names combining time step and detector index
column_names = [f'time_{t}_detector_{d}' for t in range(X.shape[1]) for d in
range(X.shape[2])]

# Create the DataFrame
reference_data = pd.DataFrame(flattened_data, columns=column_names)

# Print the shape to ensure it's correct
print(reference_data.shape)
```

Step 3: Use it in Evidently

Now that the data is in a `pandas.DataFrame`, you can pass it as `reference_data` to Evidently:

```python
from evidently.dashboard import Dashboard
from evidently.tabs import DataDriftTab

# Initialize Evidently dashboard with the required tabs
dashboard = Dashboard(tabs=[DataDriftTab()])

# Load reference data and current data (assuming current_data is your new data)
dashboard.calculate(reference_data=reference_data, current_data=current_data)

# Generate the dashboard HTML
dashboard.save("evidently_report.html")
```

Key Considerations:

- **Reshaping**: Ensure the time steps and detectors are flattened in the right order.
- **Column Naming**: You might adjust the column naming logic depending on how you want to reference time and detectors in your analysis.

This approach gives you the flexibility to use Evidently's monitoring tools on your traffic dataset.

Explain why is it neccesary in this case for timeseries forcasting using an LSTM model to create sequences using this function.
def create_sequences(data, time_steps=10):
    X, y = [], []
    for i in range(len(data) - time_steps):
        X.append(data[i:i + time_steps]) # Sequence of features
        y.append(data[i + time_steps, 0]) # Predict the next value for the first feature
    return np.array(X), np.array(y)

In time series forecasting, especially with models like LSTMs (Long Short-Term Memory networks), it's important to capture temporal dependencies in the data. The function `create_sequences` is crucial in preparing the data for such models. Here’s why:

1. Sequential Data Representation

Time series forecasting relies on understanding how past values affect future ones. LSTM models are specifically designed to handle sequences of data because they have memory cells that allow them to remember information over time.

The function `create_sequences` converts the raw time series into sequences of data, where:

- Each **input sequence** `X` contains multiple consecutive time steps from the past (e.g., the last 10 readings), which the model can use to predict future values.
- The corresponding **target value** `y` is the value at the next time step (e.g., the traffic speed at the next time point) that the model is trained to predict.

This structure is important because it enables the LSTM to learn the temporal relationships within the data.

2. Capturing Temporal Dependencies

The goal of LSTMs is to capture the long-term dependencies in the sequence. By using sequences of time steps, the function gives the model context about the previous values so that it can understand how past patterns influence the future.

In the provided function:

- `X.append(data[i:i + time_steps])`: This line creates a window of past data for the model to use as input. For example, if `time_steps = 10`, the function grabs 10 consecutive time steps of data as one input sample.
- `y.append(data[i + time_steps, 0])`: This line selects the value at the next time step (the target value) that corresponds to the sequence of 10 previous steps. The `0` indicates it's using the first feature in the data for prediction (e.g., the traffic speed).

3. Sliding Window Approach

The function uses a **sliding window approach**:

- For each possible position in the dataset (i.e., `i`), it grabs the next `time_steps` data points for the input `X` and the immediate next data point for the output `y`.
- This sliding window moves forward through the data, generating training samples and corresponding targets for the model.

4. Handling Temporal Context for Forecasting

The function prepares the data in a way that allows the LSTM to make predictions based on patterns observed in previous time steps. Since LSTMs are good at remembering past information, providing sequences of data allows the model to capture these temporal relationships.

Example:

Suppose you have traffic speed readings at 10-minute intervals, and you want to predict the speed at the next time interval based on the past 10 intervals. If your data looks like this:

Time Step	Speed (Feature 1)
1	30 km/h
2	32 km/h
3	35 km/h
...	...
12	40 km/h

With `time_steps = 10`, the function would generate the following:

- **Input** `X`: Time steps 1 to 10 (values 30 km/h, 32 km/h, ...).
- **Target** `y`: Speed at time step 11 (e.g., 38 km/h).

This process repeats for the entire dataset, generating numerous (input, target) pairs that the LSTM can learn from.

Conclusion:

The function `create_sequences` is necessary because it transforms a raw time series into a format that LSTMs can use to learn patterns over time. It creates a structured input of sequential data points and corresponding targets, which is essential for time series forecasting tasks, allowing the model to predict future values based on historical sequences.

Considering that my dataset contains speed readings from 207 loop detectors across Los Angeles, recorded every five minutes. It provides a rich temporal sequence of traffic conditions, making it ideal for time-series forecasting and machine learning applications.

Which would be the features and the target value used by the LSTM model to forcast traffic?

In your dataset, which contains speed readings from 207 loop detectors across Los Angeles recorded every five minutes, the **features** and **target values** used by the LSTM model for traffic forecasting can be understood as follows:

Features:

The **features** are the input variables that the LSTM uses to make predictions. In your case:

- Each data point in the sequence represents the traffic speeds recorded by 207 loop detectors at a particular time step.
- Therefore, at each time step, the **features** are the speed readings from all 207 loop detectors. This creates a feature vector with 207 values for each time step.

For example, if your data at a single time step looks like this:

```python
[0.91964286, 0.96607143, 0.95892857, ..., 0.84642857, 0.98571429, 0.88392857]
```

This vector represents the speed readings from 207 detectors at a given point in time.

Thus, the input to your LSTM at each time step will be a feature vector of shape `(207,)`, representing the speed readings from all the loop detectors at that time.

Target Value:

The **target value** is the variable that the model is trying to predict. In a typical traffic forecasting problem, the target is usually the speed at a future time step for one or more of the detectors.

Given that the LSTM works by looking at sequences of past data to predict the future, the target value for your model could be:

- **The speed at the next time step (for all or a specific detector)**: You might want the model to predict the speed at all 207 loop detectors for the next time step (i.e., 5 minutes into the future).

  In this case, the target would be another vector of 207 values corresponding to the future speeds at the same detectors:

  ```python
  [speed1_next, speed2_next, ..., speed207_next]
  ```

- **The speed at a particular future time step (for all detectors)**: You might want to predict the speed at some point further in the future (e.g., 30 minutes or an hour ahead), in which case the target value would be the speed readings at that time step.

- **Speed at a specific detector (at the next time step or in the future)**: In some cases, you might focus on forecasting the speed at one specific loop detector rather than all 207, using all the detectors as inputs but predicting the target for just one detector.

Input and Output Shape:

- The **input to the LSTM model** would be a sequence of traffic speeds over a specified number of time steps (e.g., the last 10 intervals) for all 207 loop detectors. If you're using 10 time steps (which means using the traffic speeds recorded over the last 50 minutes), the input shape would be **`(number_of_samples, 10, 207)`**:
  - **`number_of_samples`**: The number of sliding windows you extract from the data.
  - **`10`**: The length of the sequence (number of past time steps).
  - **`207`**: The number of loop detectors (features).

- The **output of the LSTM model** would depend on what you're forecasting:
  - If you're predicting the next time step for **all 207 detectors**, the output shape would be **`(number_of_samples, 207)`**.
  - If you're predicting the future speed at just one detector, the output shape would be **`(number_of_samples, 1)`**.

Example:

If you want to predict traffic speed 5 minutes into the future (the next time step) for all 207 loop detectors, the target value **`y`** at each time step would be the speed readings of the detectors at the next time step:

```python
y.append(data[i + time_steps])  # Predict next step for all 207 detectors
```

In this case, the LSTM learns to predict the speed readings of all detectors based on the historical traffic patterns of the previous `time_steps` (e.g., 10).

Summary:

- **Features (X)**: The speed readings from all 207 loop detectors over a sequence of past time steps (e.g., the last 10 intervals).
- **Target (y)**: The speed readings at the next time step (5 minutes into the future) for all 207 detectors, or for a specific future time step or detector, depending on your use case.

I'm predicting the nex value for the first feature as this:

y.append(data[i + time_steps, 0])

Memory updated

In your case, you're predicting the next traffic speed reading for the first feature (the first loop detector). The approach you're using is appropriate for traffic forecasting, where:

- **Features (X)**: These consist of the traffic speed readings from all 207 loop detectors over a sequence of `time_steps`. This captures the spatial (detectors) and temporal (previous speed values) dependencies in the data. For each training example, the model receives a window of traffic data of size `(time_steps, 207)` to learn from the pattern.

- **Target (y)**: You're predicting the next value for the first loop detector after the `time_steps` window. The specific target is the traffic speed for the first detector (`data[i + time_steps, 0]`), which makes this a univariate prediction task, but conditioned on multivariate (207 detectors) data.

This design allows the LSTM model to learn from the temporal dynamics of traffic across multiple detectors and then make predictions for the most important one (the first).