**Step: Data Acquisition and Management with DVC and GitHub Actions**

1. Data Generation and DVC Setup:

The data generation process initiated with a custom script, resulting in a dataset stored in a CSV format. To maintain version control and facilitate collaborative work, we leveraged DVC, an open-source version control system designed explicitly for handling large files and datasets. DVC efficiently captures and tracks changes in data files without duplicating the entire dataset.

The DVC setup involved creating a `dvc.yaml` configuration file, where we defined stages for data processing and versioning. This configuration file outlines dependencies, outputs, and commands for each stage, ensuring a structured and reproducible data pipeline. This approach enhances collaboration and enables team members to reproduce the exact dataset at any given point in time.

2. Remote Storage on Google Drive:

For remote storage, we integrated DVC with Google Drive. The DVC remote storage configuration includes the use of a Google Drive service account, enabling secure and seamless access to the dataset. The service account credentials were stored in a JSON file and securely managed through Google Cloud, ensuring proper authentication without compromising security.

The dataset was uploaded to Google Drive with DVC, and a unique hash representing the dataset's content was stored. This hash acts as a version identifier, allowing for efficient tracking of changes and providing a reliable means of accessing specific dataset versions remotely.

3. Automation with GitHub Actions:

To streamline the entire data acquisition and management process, we integrated GitHub Actions into our workflow. GitHub Actions allows for the automation of routine tasks, reducing manual intervention and enhancing efficiency.

For automation, we created an `mlflow.yaml` configuration file that defines workflows to be executed by GitHub Actions. This includes setting up Python environments, installing dependencies, configuring DVC remotes, and executing DVC commands for pulling, processing, and pushing data. Additionally, MLflow was utilized for model tracking and experiment management, contributing to a comprehensive and automated machine learning pipeline.

**Step: Data Preprocessing and Model Training**

**Feature engineering**

The feature engineering done in this code appears purposeful and is aligned with common practices for handling time-series data, especially in predictive maintenance scenarios. Let's break down the key aspects:

Time-related Features: Extracting various time-related features from the 'Timestamp' column ('Hour', 'Minute', 'Second', 'DayofWeek', 'DayofMonth', 'Month', 'Year') is a common practice in time-series analysis. These features can help the model capture patterns that may vary at different times of the day, week, month, or year.

One-Hot Encoding: One-hot encoding is applied to the categorical variables 'Machine_ID' and 'Sensor_ID.' This transformation is necessary because machine learning models typically work with numerical inputs, and one-hot encoding is a way to represent categorical variables numerically. It allows the model to understand and learn from different categories.

Scaling: Standard scaling is applied to the features after one-hot encoding. Scaling ensures that all features have a similar scale, preventing certain features from dominating others during the model training process. This is particularly important for algorithms that rely on distance measures or gradient-based optimization.

Column Transformer: The use of ColumnTransformer in the preprocessing pipeline allows for different transformations to be applied to specific subsets of columns. This is a flexible and organized way to handle various types of features differently.

Remainder='passthrough': Including remainder='passthrough' in the ColumnTransformer ensures that any columns not explicitly transformed are passed through without changes. This is beneficial in maintaining the original features that do not require specific transformations.

In summary, the feature engineering approach is purposeful and aligns with best practices for handling time-series data.

1. **Model Training**

    **Data Preprocessing**
    The dataset, located in 'dummy_sensor_data.csv,' is preprocessed using the preprocess_data function. The preprocessing involves handling missing values, scaling, and encoding categorical features.

    **Data Splitting**
    The preprocessed data is split into training and validation sets with an 80-20 split ratio.

    **Model Selection**

A Random Forest Regressor is chosen as the machine learning algorithm for its suitability for time-series data.

**Hyperparameter Tuning**
Hyperparameter tuning is performed using GridSearchCV to find the optimal combination of hyperparameters. The grid search explores different values for the number of estimators and maximum depth of the trees.

**Model Training and Logging**
The best model obtained from hyperparameter tuning is logged using MLflow. Model parameters, preprocessing details, and training metrics (mean squared error on the training set) are logged for future reference.

**Model Registration**
The best model is registered in the MLflow Model Registry with the name 'PredictiveMaintenanceModel' and the stage 'Production.'

1. **Model Testing**

**Live Data Preprocessing**
The live data, assumed to be in 'live_data.csv,' is preprocessed using the same steps as during training. This ensures consistency in feature transformations.

**Model Loading**
The registered model named 'PredictiveMaintenanceModel' in the 'Production' stage is loaded from the MLflow Model Registry using the mlflow.sklearn.load_model function.

**Model Inference**
The loaded model is used to make predictions on the preprocessed live data. The predictions are stored in the variable 'live_predictions.'

**Evaluation of model perfomance**

Task is a regression problem (predicting a continuous target variable), then mean squared error (MSE) is a suitable metric for evaluating model performance. It measures the average squared difference between the actual and predicted values.

**Step: Deployment**

1. Package the Trained Model into a Docker Container

After successfully training the predictive maintenance model using the selected machine learning algorithm, the next step is to package the trained model into a Docker container for deployment. This involves creating a Dockerfile, specifying dependencies, and defining the entry point for running the Flask application.

2. Dockerfile

The Dockerfile starts with the official Python runtime image, sets the working directory, installs dependencies, copies project files, exposes port 8080, and specifies the command to run the Flask application.

3. Deploying the Docker Container

Build the image using docker build -t techtrio/project:lts . and run the container with docker run -p 8080:8080 techtrio/project:lts. This deploys the Flask application inside the Docker container, accessible on port 8080.

**Step: Drift Monitoring**

1. Implementing Concept Drift Monitoring

Concept drift monitoring is essential for observing the model's performance over time. In our approach, a subset of the training data acts as the "current validation" set for drift metric monitoring.

2. drift.py

This script monitors drift by calculating Mean Squared Error (MSE) as a metric. It checks for concept drift by comparing the current MSE with the baseline MSE. If the relative change exceeds the defined threshold, concept drift is detected, triggering model retraining.

3. Automated Retraining Triggers

The script checks for concept drift and, if detected, re-trains the model on the full training set. This ensures adaptation to changing patterns in the data, enhancing the model's resilience to evolving conditions. In addition, a check is also added in mlflow.yaml for this.

Incorporating concept drift monitoring makes the MLOps pipeline more robust, ensuring the predictive maintenance model remains effective in the face of evolving data distributions.