**Machine Learning Project Documentation**

**Deployment**

**1. Overview**

The deployment phase of our machine learning model involves several key steps to make the model available for real-world use in a production environment. Encapsulate the model in a **scikit-learn** pipeline to unify the preprocessing and training steps. Serialize and save the model using **Pickle** for easy reuse in production. Deploy the model as an API using **FastAPI**, exposing endpoints for making predictions. Create a Docker image to containerize the **FastAPI** app, ensuring portability and easy deployment to various environments like cloud services or servers.

**2. Model Serialization**

The process of serializing a trained machine learning model for deployment is a critical step that allows you to save the model in a format that can later be loaded and used to make predictions without retraining it. Serialization ensures that the model's state, including learned parameters and pipeline configurations, is preserved and can be transferred between different environments.

The key steps in the serialization process are:

1. Model Training: First, you need a trained model.
2. Choose a Serialization Format: **Pickle** is a general-purpose serialization format in Python that works well for serializing simple models and objects.
3. Serialize and Save the Model: Once the model is trained, it can be serialized using **Pickle**. The serialized file can then be saved to disk and later loaded to make predictions without retraining.

**3. Model Serving**

Here are the steps to serve a serialized model using **FastAPI** and deploy it on an on-premises solution with **Docker**:

1. **Load the Serialized Mode**l: Use **FastAPI** to load the serialized model (example: model.pkl) when the app starts and define a prediction endpoint.
2. **Create a Dockerfile**: Write a **Dockerfile** to containerize the **FastAPI** app and install the necessary dependencies.
3. **Build the Docker Image**: Locally build the Docker image with the **FastAPI** app and serialized model.
4. **Run the Docker Container:** Start the Docker container on your on-premises server to serve the API.
5. **Access the API**: Use the **FastAPI** prediction endpoint to make predictions by sending **HTTP** requests with input data.

**4. API Integration**

After training and serializing our model with **FastAPI,** I integrated it into the API. I created two important endpoints: a **POST** endpoint **(/predict**) and a **GET** endpoint **(/students**). These endpoints work together to allow users to interact with the machine learning model and store and retrieve predictions in a **MySQL** database.

The **POST** endpoint (**/predict**) is designed to handle requests where a client sends student data in a JSON format. Once the request is received, the API processes the data by transforming it into a format suitable for the machine learning model.

The **GET** endpoint (**/students**) is responsible for retrieving and displaying all previously stored predictions and corresponding student data from the MySQL database. When a client makes a GET request to this endpoint, the API queries the database and collects all records containing past predictions. It then returns this information in a structured JSON format, allowing users to review previous predictions and student information.

**5. Security Considerations**

In order to enhance the security of the application, I utilized Pydantic and encryption methods from the cryptography.fernet module. Pydantic plays a crucial role in data validation and settings management. By defining the data models with Pydantic, the input data is ensured to adhere to the expected schema. This not only helps prevent potential injection attacks by validating types and formats, but also simplifies debugging by providing clear error messages when the data does not conform to the specified structure.

To further strengthen the security of the API, I implemented encryption using cryptography.fernet. Fernet is a symmetric encryption method that provides a straightforward way to encrypt and decrypt data, ensuring that sensitive information, such as user credentials or personal data, is stored securely.

**6. Monitoring and Logging**

The performance of the deployed model is monitored and logged using MLflow. This monitoring system verifies whether the model's predictions fall within an acceptable range. If a prediction exceeds 100 or is below 20, an error is logged in MLflow, and this error is also displayed in the frontend interface when a prediction is made. This dual reporting mechanism ensures that any discrepancies in model performance are promptly identified, allowing for timely intervention and adjustments to maintain the model's reliability and effectiveness.

**References:**

1 - <https://www.fastapitutorial.com/blog/introduction-pydantic-for-fastapi/>

2 - https://cryptography.io/en/latest/fernet/