****

**Machine Learning Project Documentation**

Real-Time Air Quality Prediction and Classification System

**Group – 1 Members**

1. **Mohammad Wasil Jalali**
2. **Bahir Saidi**
3. **Mohammad Aref Rezvan Panah**
4. **Habibullah Farahmand**

**Deployment**

**1. Overview**

The deployment phase ensures that the machine learning models (RandomForestClassifier, TS-Mixer for regression, and PPO for reinforcement learning) are made accessible in a real-world production environment. To achieve this, the project involves several key steps: serializing the trained models, hosting the models on a suitable platform, integrating them into APIs for easy interaction, securing the deployment, and implementing monitoring mechanisms. The ultimate goal is to provide a seamless and scalable service where the models can handle real-time predictions efficiently and securely.

**2. Model Serialization**

To deploy the models, the first step is serializing them into formats suitable for loading during inference.

* **RandomForestClassifier**: This model is serialized using joblib for lightweight storage while maintaining scikit-learn compatibility.
* **TS-Mixer**: The trained Tensorflow model is serialized into .keras format. This format ensures the model’s state\_dict can be efficiently reloaded for inference.
* **PPO Model**: The reinforcement learning model, trained using stable-baselines3, is saved in a .pt format to retain all configurations and parameters.

For efficient storage and transfer, all serialized models are versioned properly and compressed where necessary. Additionally, ONNX may be explored for the deep learning model to optimize inference and ensure compatibility across different platforms.

**3. Model Serving**

The serialized models will be deployed using a scalable hosting platform that aligns with the project requirements. A cloud-based solution such as **AWS EC2**, **Render**, or **Railway** will be used to host both the backend (Django) and the models. The steps include:

1. Deploying a Django server where the models will be loaded into memory upon application startup.
2. Ensuring resource efficiency by containerizing the application using Docker.
3. Configuring the platform to autoscale during high-demand periods.

Alternatively, for local testing or private setups, the models can be served on an on-premises server using Docker and Kubernetes for orchestration. This approach ensures flexibility and control over resources.

**4. API Integration**

The models are integrated into a Django backend to provide API endpoints for predictions. This ensures the models are accessible programmatically. The Django REST Framework is used to define endpoints for:

* **Prediction**: Endpoints are provided for regression, classification, and reinforcement tasks. Each endpoint receives input in JSON format and returns the predictions in the same format.
* **Health Checks**: A /healthcheck endpoint is included to confirm the availability of the server and models.

Input data preprocessing, such as scaling or encoding, is handled within the backend to ensure seamless interaction with the models. The API design prioritizes clarity and consistency for user interactions, making it easy to integrate with external applications.

**5. Security Considerations**

Security is critical in this deployment to safeguard the models, data, and user interactions.

* **Authentication and Authorization**: API keys are implemented to restrict access, and role-based access control ensures users access only the endpoints they are permitted to use.
* **Data Encryption**: All communications between the client and server are encrypted using HTTPS, and sensitive data (e.g., user inputs or model files) is encrypted at rest.
* **Model Protection**: The models are deployed behind an API to prevent direct access. Obfuscation techniques may also be used to prevent reverse engineering of the models.
* **Rate Limiting**: To prevent abuse, rate limiting is implemented, ensuring fair usage across users.

**6. Monitoring and Logging**

To ensure the deployment remains stable and performs as expected, monitoring and logging mechanisms are set up:

* **Performance Metrics**: Key metrics such as latency, throughput, and resource usage (CPU, memory) are tracked using tools like Prometheus and Grafana.
* **Model-Specific Monitoring**: Accuracy drift and data distribution changes are monitored using tools like Evidently AI, which can flag discrepancies over time.
* **Logging**: Request and response logs are maintained using Django’s logging system or external tools like ELK Stack (Elasticsearch, Logstash, Kibana). These logs help debug issues and track usage patterns.
* **Alerting**: Alerts are configured to notify the development team of critical events, such as high latency, excessive errors, or API downtime, allowing for quick issue resolution.

This setup ensures the models perform reliably, with a robust mechanism in place to identify and address issues proactively.