**1. Model Persistence Techniques. (4 Marks)**

a) Identify and describe three different model persistence techniques used in machine learning.

* Pickle: Python’s built-in tool for saving objects. It turns a trained machine learning model into a byte stream that I can store and reload later. It works for almost any Python object.
* Joblib: Similar to Pickle but is more efficient when working with large numerical arrays, which makes it good for NumPy arrays (like scikit-learn). It compresses the data and loads it faster compared to Pickle.
* TensorFlow SavedModel: This is the standard way to save models built in TensorFlow. It stores both architecture and weights. Easier to deploy models across different platforms.

b) Explain their use cases, supported model types and common pitfalls or compatibility issues when used across environments.

* Pickle
  + Use cases:   
    Useful for quick experiments or saving a model when I know I won’t be changing the Python setup. Often used in prototyping.
  + Supported models:  
    Any python-baased ML object
  + Pitfalls:   
    Can easily break if the Python version or library versions change.
* Joblib
  + Use cases:  
    Useful for scikit-learn models and other models that are heavy on NumPy arrays. Commonly used in production pipelines where efficiency matters.
  + Supported models:  
    Scikit-learn models, NumPy-heavy Python objects
  + Pitfalls:  
    Might run into compatibility problems if the sickit-learn version changes between training and serving.
* TensorFlow SavedModel
  + Use cases:  
    Best for deploying deep learning models across multiple environments. Can use it with TensorFlow Serving, TensorFlow Lite for mobile, or TensorFlow.js for the browser.
  + Supported models:   
    TensorFlow and Keras models
  + Pitfalls:  
    It’s more tied to TensorFlow, so it won’t help if the model was trained in scikit-learn. Sharing models can also be a bit heavy due to large file sizes.

c) Compare these techniques in terms of serialisation speed, Cross-platform compatibility and Human readability.

* Pickle
  + Serialization speed:   
    Moderate. Works well for small or medium objects, but can be slow with large NumPy arrays.
  + Cross-platform compatibility:  
    Low portability
  + Human readability:  
    Not human-readable. Data stored as a binary byte stream.
* Joblib
  + Serialization speed: Faster than Pickle for numerical data because it is optimized for handling NumPy arrays.
  + Cross-platform compatibility:  
    Low portability
  + Human readability:  
    Not human-readable. Stores compressed binary files.
* TensorFlow SavedModel
  + Serialization speed: Depends on model size.
  + Cross-platform compatibility:  
    High portability
  + Human readability:  
    Partially human-readable. Stores model metadata and graph in a directory structure with some files in readable text/JSON format.

d) Present your findings in a comparison table and describe when to use each.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Technique** | **Serialization Speed** | **Cross-Platform Compatibility** | **Human Readability** | **Best Use Cases** | **Pitfalls** |
| Pickle | Moderate | Low | No | Quick prototyping | Often fail to load if Python version or package version differs |
| Joblib | Faster than Pickle | Low | No | Scikit-learn models and pipelines in production where efficiency matters | Version mismatch in scikit-learn can lead to incompatibility |
| TensorFlow SavedModel | Depends on model size | High | Partial | Deploying deep learning models across platforms | Large file sizes. Restricted to TensorFlow and Keras models. |

* Pickle is best used when the goal is speed and convenience within the same development environment. It’s a good choice for quick experiments, or proof-of-concepts, where the model does not need to move across environments. Generally not recommended for production deployment.
* Joblib is the preferred option for production pipelines with scikit-learn or NumPy-heavy models, since it is much faster and more space-efficient than Pickle when handling numerical arrays. But, the pitfall is that it lacks cross-platform flexibility.
* TensorFlow SavedModel is designed for scalable deployment and cross-platform use. This format preserves both the model weight and architecture, which allows easy deployment across TensorFlow Serving, TensorFlow Lite (for mobile), and TensorFlow.js (for browsers). It is the most portable option among the 3 options. However, it requires large storage and is locked to TensorFlow and Keras models.

**2. Model Serving with RESTful APIs. (4 Marks)**

a) Explain the role of **Flask + Connexion** in serving models via REST APIs.

Flask is a Python web framework that makes it easy to expose a machine learning model through endpoints like /predict. However, Flask on its own doesn’t enforce structure. That’s what Connexion is used for. It sits on top of Flask and uses an OpenAPI specification (YAML/JSON) to automatically generate routes, validate requests, and ensure the API behaves consistently. In conclusion, Flask handles the web server logic, while Connexion ensures the API follows a clear contract.

b) Discuss the advantages of using an OpenAPI specification in this context.

* It automatically checks incoming requests and responses against the defined schema, reducing errors.
* OpenAPI generates interactive API docs (like Swagger UI) so developers and consumers can test endpoints without digging through code.
* Everyone works from the same “blueprint” making collaboration smoother and reducing miscommunication between frontend and backend teams.

c) Compare REST API model serving with MLflow-based serving.

* For REST APImodel serving, custom endpoints are built and maintained. This gives flexibility in how requests are handled, pre-processing and post-processing steps, and integration with other app logic. However, more work is needed for versioning and logging.
* For MLflow-based serving, instead of writing custom code, the MLflow can spin up a REST API directly from a saved model. It comes with built-in model versioning, experiment tracking, and deployment utilities, which makes it easier to manage models at scale. The trade-off is less flexibility for custom workflows compared to a hand-written Flask API.

d) Include example tools for model deployment (e.g., Flask, FastAPI, MLflow).

* Flask: Good for lightweight, custom REST APIs. Often used when full control is needed over request handling.
* FastAPI: A modern alternative to Flask. It’s faster, supports async calls out-of-the-box, and automatically generates OpenAPI documentation.
* MLflow: Best for end-to-end ML lifecycle management, including model tracking, versioning, and serving with minimal custom code.

**3. Consuming ML APIs in Full Stack Applications (6 Marks)**

You are provided with a deployed machine learning API with a /predict endpoint. The endpoint expects a POST request:

a) Describe how you would consume the /predict endpoint, considering API request structure

I would use HTTP POST request since predictions require me to send input data to the model. The request must include the URL, Headers, Body (Payload), and Response.  
  
Examples:

* URL: <http://api.example.com/predict>
* Headers: Often include Content-Type: ”application/json” and authentication tokens
* Body (Payload): JSON object containing input features needed for the model
* Response: JSON object containing prediction results

b)  Describe how you would consume the /predict endpoint, considering handling asynchronous responses

I would use async/await (JavaScript/ Python) to ensure that the application does not block while waiting for a response. In JavaScript, I could either use promises with the .then() syntax or I could use async/await for cleaner handling.

c) Describe how you would consume the /predict endpoint, considering CORS and security implications

When considering CORS and security implications, I would first check whether the frontend and backend are hosted on different domains. If it is, the server needs to explicitly allow cross-origin requests by setting appropriate headers like Access-Control-Allow-Origin. From security point of view, API should never be left open to the public without protection. Options for safeguards is to include the requirement of having an API key, using HTTPS to encrypt traffic, and adding authentication tokens. These are safeguards to prevent unauthorized access.

d) Implement a basic request to the API using your chosen method (JavaScript, Python, Postman, etc.). Include code and sample input/output.



**4. MLflow Model Serving (6 Marks)**

a) Briefly describe what MLflow is and how it differs from custom Flask APIs.

MLflow is an open-source platform for managing the end-to-end machine learning lifecycle, including experiment tracking, model packaging, and deployment. Unlike custom Flask APIs, which require manual setup for model loading, input validation, and serving, MLflow provides standardized tools to log, version, and serve models with minimal code.

MLflow model serving automatically handles the input/output schema, model versioning, and REST API creation, reducing the need for boilerplate code.

b) Use MLflow to serve a saved model locally or via mlflow.models.serve.

To serve a saved model locally using MLflow, I would open the Bash terminal in VS Code and enter:  
mlflow models serve -m <model\_path> -p 5000

I will need to replace <model\_path> with the actual model directory.

c) Include the model signature and input/output schema.

Let’s say a model takes age (integer), income (float), and education (string) as model features, the signature would be:  
inputs = [age: int, income: float, education: string], outputs = [prediction: string, confidence: float]

d) Paste a working cURL/Postman request and MLflow response.

cURL request:  
curl -X POST http://127.0.0.1:5000/invocations -H "Content-Type: application/json" -d '{"age": 30, "income": 50000, "education": "Bachelors"}'

Example MLflow response:

{  
"prediction": ["Approved"],  
"confidence": [0.92]  
}