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

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

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

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

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

**Task 1 -**

**1. Model Persistence Techniques**

**a) Three Model Persistence Techniques**

1. **Pickle (Python’s pickle)**
   * A Python-native binary serialization format.
   * Store an entire object graph (the model, its parameters, and Python state) as bytes.
2. **Joblib (from scikit-learn)**
   * Part of scikit-learn ecosystem, optimized for NumPy arrays and scikit-learn models.
   * Faster than Pickle for large numerical data (large numpy arrys)
   * Uses .joblib or .pkl extensions
   * Better for scikit-learn pipelines
3. **ONNX (Open Neural Network Exchange)**
   * A cross-platform format for deep learning models /
   * Supports interoperability between frameworks (PyTorch → TensorFlow → ONNX Runtime).

**Key Differences**:

* *Pickle*: Universal but Python-only
* *Joblib*: Best for scikit-learn with large numpy arrays
* *ONNX*: Framework-agnostic for production deployments

Each technique serves different needs in the ML lifecycle from experimentation to production deployment.

**b) Use Cases, Supported Models, and Pitfalls**

| **Technique** | **Use Cases** | **Supported Models** | **Pitfalls/Compatibility Issues** |
| --- | --- | --- | --- |
| **Pickle** | General Python object serialization, small models | Any Python object (scikit-learn, custom classes) | - Security risks (malicious payloads)  - Python version dependency  - Not cross-language |
| **Joblib** | Large NumPy-based models (scikit-learn) | scikit-learn, NumPy-heavy models | - Slower for non-NumPy objects  - Python-only  - Version mismatches can break loading |
| **ONNX** | Cross-framework deep learning models | PyTorch, TensorFlow, XGBoost (limited scikit-learn) | - Limited support for non-DL models  - Pre/post-processing not always included  - Framework-specific ops may not convert |

**c) Comparison: Speed, Cross-Platform, Readability**

| **Technique** | **Serialization Speed** | **Cross-Platform Compatibility** | **Human Readability** |
| --- | --- | --- | --- |
| **Pickle** | Fast for small objects | ❌ (Python-only) | ❌ (Binary format) |
| **Joblib** | Very fast for NumPy | ❌ (Python-only) | ❌ (Binary format) |
| **ONNX** | Moderate (conversion step) | ✅ (Multi-language/framework) | ❌ (Binary/protobuf) |

**d) When to Use Each Technique**

| **Technique** | **When to Use** | **When to Avoid** |
| --- | --- | --- |
| **Pickle** | - Quick prototyping  - Non-NumPy Python objects | - Production deployments  - Cross-language systems |
| **Joblib** | - scikit-learn models with large NumPy arrays  - Faster than Pickle for ML workloads | - Non-Python environments  - Non-NumPy data |
| **ONNX** | - Deploying DL models across frameworks  - Edge/embedded systems | - Non-deep-learning models  - Complex pre/post-processing |

**Key Takeaways**

* **For scikit-learn**: Prefer **Joblib** (speed) or **Pickle** (flexibility).
* **For deep learning**: **ONNX** ensures framework interoperability.
* **For production**: Avoid raw Pickle/Joblib; use **MLflow** or **containerized APIs** for versioning and security.

This structured approach ensures optimal model persistence based on use case, performance, and deployment needs.

**Task 2**

**2. Model Serving with RESTful APIs**

**a) Role of Flask + Connexion in Serving ML Models**

Flask is a lightweight Python web framework commonly used to serve ML models via REST APIs. **Connexion** is a Flask extension that adds OpenAPI/Swagger support, enhancing model serving by:

* **Automating API Endpoints**: Converts OpenAPI specifications (YAML/JSON) into working Flask routes, reducing boilerplate code.
* **Input Validation**: Enforces schema rules (data types, required fields) before requests reach the model.
* **Interactive Documentation**: Generates Swagger UI for easy API testing and client integration.
* **Error Handling**: Standardizes error responses (e.g., 400 for invalid input).

**Example Workflow**:

1. Define an OpenAPI spec (e.g., swagger.yml) describing the /predict endpoint.
2. Connexion reads the spec and auto-creates the Flask route.
3. Flask serves predictions with built-in validation The ML model loads at startup, and predictions are served via validated API calls.

**b) Advantages of OpenAPI Specification**

Using OpenAPI with Connexion provides:

1. **Standardized API Contracts**
   * Clear definitions of inputs/outputs (e.g., feature names, data types).
   * Ensures consistency across development and production.
2. **Automated Validation**
   * Rejects invalid requests (e.g., missing fields, incorrect data formats).
3. **Self-Documenting APIs**
   * Swagger UI allows developers to test endpoints interactively.
4. **Client SDK Generation**
   * Tools like swagger-codegen can auto-create client libraries (e.g., for React, Java).
5. **Team Collaboration**
   * OpenAPI specs serve as a single source of truth for frontend/backend teams.

**c) REST API vs. MLflow Serving**

| **Feature** | **Flask/FastAPI** | **MLflow** |
| --- | --- | --- |
| **Setup** | Manual endpoint creation | Pre-built /invocations route |
| **Validation** | OpenAPI or custom logic | Model signatures only |
| **Versioning** | Custom implementation | Built-in (Model Registry) |
| **Scalability** | Requires Docker/K8s setup | Native Docker support |
| **Best For** | Custom pipelines | Standardized model deployments |

**When to Use Which**:

* **Flask/FastAPI**: Best for custom logic (e.g., complex preprocessing, multi-model ensembles).
* **MLflow**: Ideal for rapid deployment with built-in MLOps (e.g., A/B testing, logging).

**d) Example Tools for Model Deployment**

| **Tool** | **Use Case** | **Key Feature** |
| --- | --- | --- |
| **Flask** | Custom APIs with OpenAPI | Lightweight, flexible |
| **FastAPI** | High-performance async APIs | Auto-docs, Pydantic validation |
| **MLflow** | End-to-end MLOps | Model registry, Docker support |
| **TF Serving** | Production TensorFlow models | gRPC/REST, versioning |

**Task 3**

**3. Consuming ML APIs in Full Stack Applications**

**a) API Request Structure**

To consume the /predict endpoint:

1. **HTTP Method**: POST (since we’re sending input data).
2. **Headers**:
   * Content-Type: application/json (for JSON data).
   * Authorization: Bearer <token> if the API requires authentication.

Below the **feature values** send to the /predict API—one row per sample.

**"features": [**

**[5.9, 3.0, 5.1, 1.8],**

**[5.4, 3.9, 1.7, 0.4],**

**[6.3, 2.5, 4.9, 1.5]**

**]**

 **Response Handling**:

* Expect JSON response with prediction results
* Handle status codes (200 for success, 400/500 for errors)

**Key Considerations**:

* Validate input data before sending (e.g., check feature types/nulls).
* Handle API errors (e.g., 400 Bad Request for invalid input).

**b) Handling Asynchronous Responses**

#### ****1. Core Concept -**** Asynchronous requests let your app: ✅ Stay responsive while waiting for predictions ✅ Handle multiple API calls efficiently ✅ Gracefully manage delays/errors

### ****2. Key Code Snippets****

#### ****Frontend (JavaScript)****

// Using async/await (modern approach)

async function fetchPrediction(data) {

const response = await fetch('/predict', {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify(data)

});

return await response.json(); // Auto-parses JSON

}

#### ****Error Handling****

try {

const result = await fetchPrediction({ temp: 30 });

console.log("Prediction:", result);

} catch (error) {

console.error("API Error:", error); // Network/server errors

}

#### ****Backend (Python)****

# Async request with aiohttp

async def predict(data):

async with aiohttp.ClientSession() as session:

async with session.post('/predict', json=data) as resp:

return await resp.json()

### ****3. Critical Practices (Javascript)****

**Timeouts**

 // Cancel request after 5 seconds

const controller = new AbortController();

setTimeout(() => controller.abort(), 5000);

fetch('/predict', { signal: controller.signal });

 **Loading States**

 const [loading, setLoading] = useState(false);

const getPrediction = async () => {

setLoading(true);

const result = await fetchPrediction(data);

setLoading(false);

};

 **Parallel Requests**

const [user, prediction] = await Promise.all([

fetch('/user'),

fetchPrediction(data)

]);

**c) CORS and Security Implications**

For browser apps, call /predict i through own backend (BFF/proxy): the browser talks to server (same origin), and server calls the ML API—this hides secrets, enforces validation/rate limits, and avoids CORS headaches.

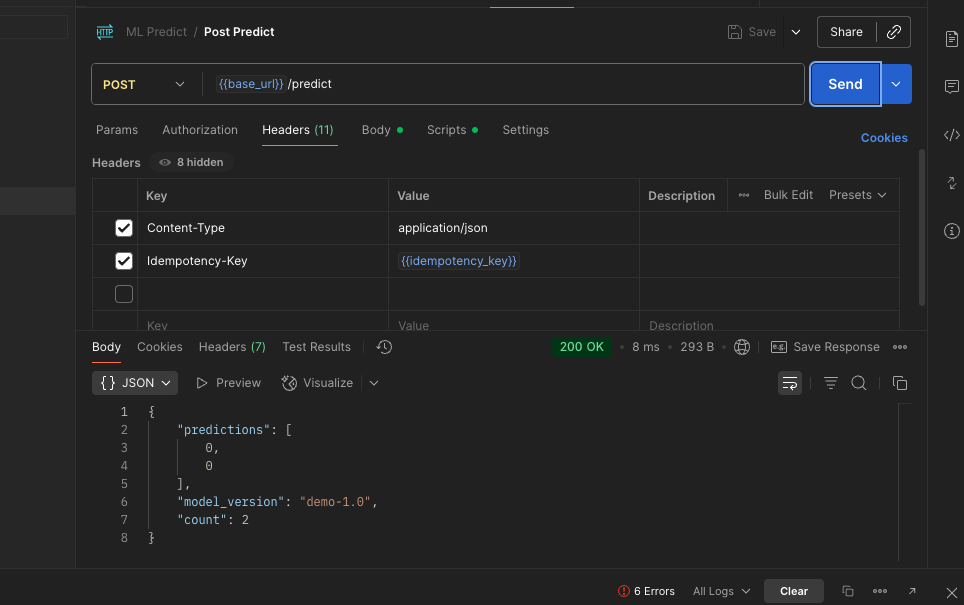
If must call the ML API directly, enable proper CORS on the API: respond to OPTIONS preflights, set a specific Access-Control-Allow-Origin (not \* with credentials), allow POST and headers like Content-Type, Authorization, Idempotency-Key, and expose Location/Retry-After for async flows.

Security-wise, use HTTPS everywhere, short-lived tokens CSRF protection if cookies are used, strict input validation, rate limits, request size caps, minimal error bodies, and Cache-Control: no-store for prediction responses.

**Points:**

* **Architecture:** a backend proxy/BFF → hides secrets, same-origin = no CORS issues.
* **CORS (if direct from browser):**
  + Handle OPTIONS preflight; Access-Control-Allow-Methods: POST, OPTIONS.
  + Access-Control-Allow-Origin: https://app.example.com (whitelist, not \* if using creds).
  + Access-Control-Allow-Headers: Content-Type, Authorization, Idempotency-Key.
  + Access-Control-Expose-Headers: Location, Retry-After, X-Request-ID.
  + Access-Control-Allow-Credentials: true only if you must use cookies; add Vary: Origin.
* **Auth & secrets:** Short-lived JWTs/API keys stored server-side; rotate keys; scope permissions.
* **CSRF/XSS:** If cookies, use SameSite + CSRF tokens; sanitize any UI-rendered messages.
* **Transport:** Enforce HTTPS; consider HSTS.
* **Abuse control:** Rate limiting, request body size limits, WAF/API gateway; retries use **Idempotency-Key** + backoff.
* **Observability:** Add X-Request-ID; structured logs without PII; clear 4xx vs 5xx.
* **Privacy & caching:** Minimize inputs, redact logs; Cache-Control: no-store on predictions.
* **Async specifics:** Expose/read Location + Retry-After; handle 202 Accepted polling cleanly.

**d) Implementation Example**

**Outpout on Postman**

**Task 4**

**a) What is MLflow? How it differs from custom Flask APIs**

MLflow is a free, open-source tool that helps manage the whole machine learning workflow. It can track experiments, save and package models, register different versions, and also serve models for predictions.

**How it’s different from writing own Flask API:**

* **No manual server code:** You don’t need to build routes yourself. Just run mlflow models serve and it creates a REST API automatically.
* **Standard way to save models:** Models are stored in a common format (called “flavors” like sklearn or pyfunc), so they can be loaded and served the same way, no matter which library was used to train them.
* **Reproducible:** It automatically saves model files, parameters, metrics, and environment info, so you can re-run the work later.
* **Input/output schema:** You can add a schema that checks requests and responses, reducing errors.
* **Model registry:** You can manage different versions of a model and easily move them from *Staging* to *Production* without rewriting code.

**4b and c) Refer to attached files**

**4d) See below screenshot which shows working cURL/Postman request and MLflow response.**

