0. Topics

Sunday, May 25, 2025 11:14 AM

1. **Model Serialization**
2. **Pickle vs Joblib**
3. **Designing Model I/O Schemas**
4. **Serving ML Models with FastAPI**
5. **Handling Batch Predictions**

1. Model Serialization

Sunday, May 25, 2025 11:23 AM

What is Model Serialization?

- Serialization is the process of converting a trained machine learning model into a byte stream that can be saved to a file or database
- This can later be deserialized (loaded) to recreate the model in memory without retraining it from scratch

Common Libraries:

- Pickle
- Joblib
- Keras (**.h5, .keras**)
- Tensorflow (**SavedModel**)
- Pytorch (**.pt**, **.pth**)

Common Formats:

- JSON
- Binary
- HDF5

Why Model Serialization is Important:

1. **Saves Time and Computational Resources:**
   - Training ML models, especially deep learning models, can take minutes to hours—or even days
   - Serialization allows you to store the trained model once and use it repeatedly without incurring the cost of retraining
   - This is especially critical in:
     - Production deployments
     - Iterative testing
     - Rapid prototyping
     - Resource-constrained environments (like edge devices)
2. **Portability Across Platforms and Environments:**
   - A serialized model can be moved across different machines, operating systems, or cloud services
   - It facilitates collaboration across teams—one team trains the model, another team deploys it
   - Serialization also allows ML models to be integrated into mobile apps, IoT devices, or cloud containers (ex. **Docker**)
3. **Reproducibility and Consistency:**
   - Serialization preserves the trained state exactly—including model parameters, weights, and internal

configurations

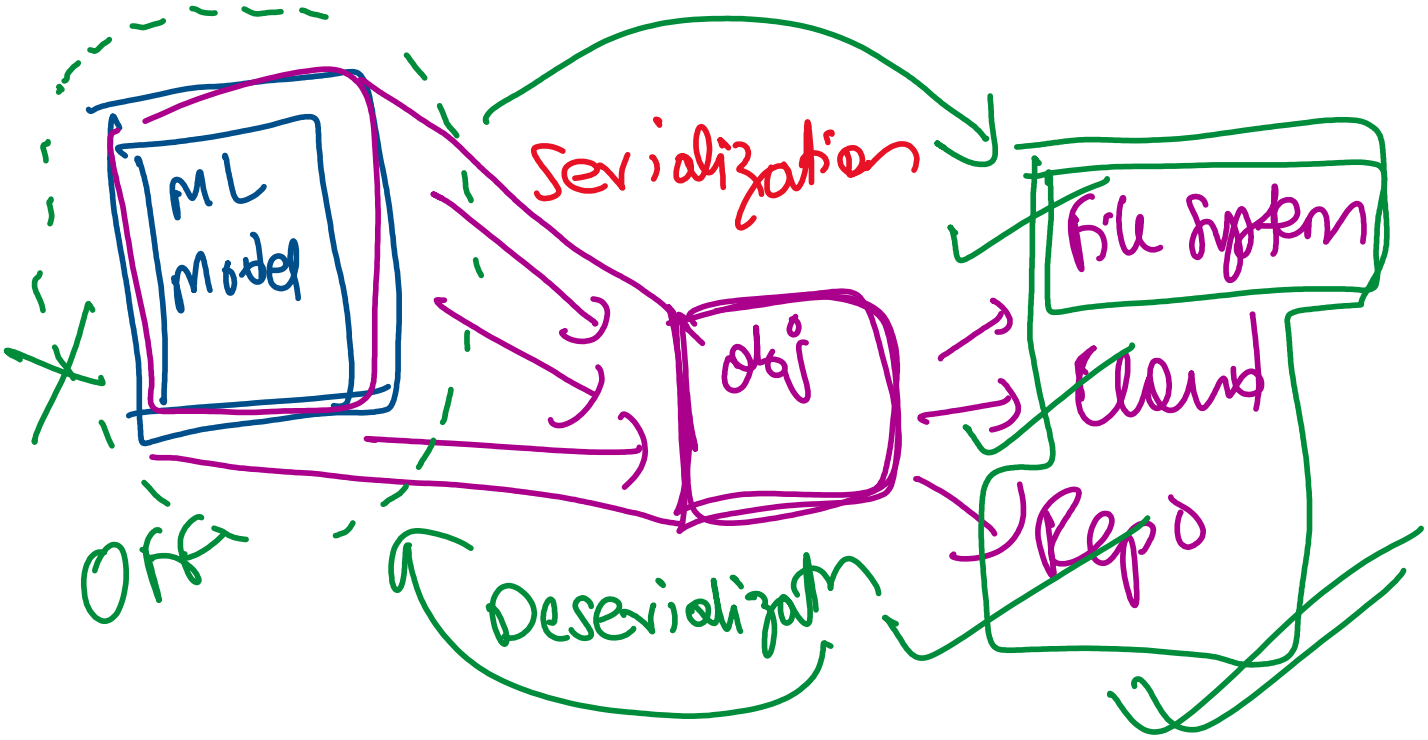
- Ensures consistent predictions across different runs, which is crucial for:
  - Model validation
  - A/B testing
  - Regulatory or compliance audits

4. **Model Serving and Integration:**

- Serialization enables seamless deployment of ML models into real-world systems like REST APIs, web applications, dashboards, or edge devices
- Allows decoupling of training and inference phases:
  - Training happens offline (e.g., Jupyter notebook)
  - Inference happens online (e.g., real-time request to a FastAPI or Flask server)

5. **Foundation for Model Versioning and CI/CD Pipelines:**

- Serialization plays a vital role in MLOps:
  - Store models in versioned model registries (e.g., **MLflow**, **DVC**, **AWS SageMaker**)
  - Automate model deployment and rollback
- Helps teams track changes and compare performance across versions



2. Pickle vs Joblib

Sunday, May 25, 2025 12:38 PM

FEATURE	pickle	joblib
Purpose	General-purpose serialization	Optimized for objects with large NumPy arrays (common in ML)
Library	Built-in (import pickle)	External (pip install joblib)
Serialization Format	Binary	Binary (optimized with efficient NumPy storage)
Performance	Slower for large NumPy arrays	Faster for large NumPy arrays due to efficient memory mapping
Use Case	Any Python object	Large data like ML models, NumPy arrays, and SciPy objects
Compression Support	Manual (you must use gzip, bz2, etc. separately)	Built-in (compress=True)
Parallel I/O	Not supported	Supported (internally uses multiple I/O operations)
File Size	Larger with numerical arrays	Smaller for numerical data due to compression
Backward Compatibility	Good	Similar to pickle, but better compatibility with NumPy
Common ML Usage	Small models (e.g., decision trees, dicts)	Large models (e.g., scikit-learn pipelines with arrays)

3. Designing Model I/O Schemas

Sunday, May 25, 2025 12:24 PM

Why Define Input and Output Schemas?

1. **Data Validation and Type Safety:**
   - Without schema validation, you're blindly trusting that the incoming data is well-formed—which is risky
   - Can implement automatic type checking using Pydantic and rules using Field

2. **Clear API Contracts:**
   - Schemas define a contract for how your API should be used
     - **InputSchema** = what users should send
     - **OutputSchema** = what your app will return

3. **Improved Developer Experience:**
   - FastAPI auto-generates beautiful interactive docs (Swagger UI)
   - Built-in validation error messages help frontend/backend developers debug easily
   - **Less guesswork = faster development and fewer bugs**

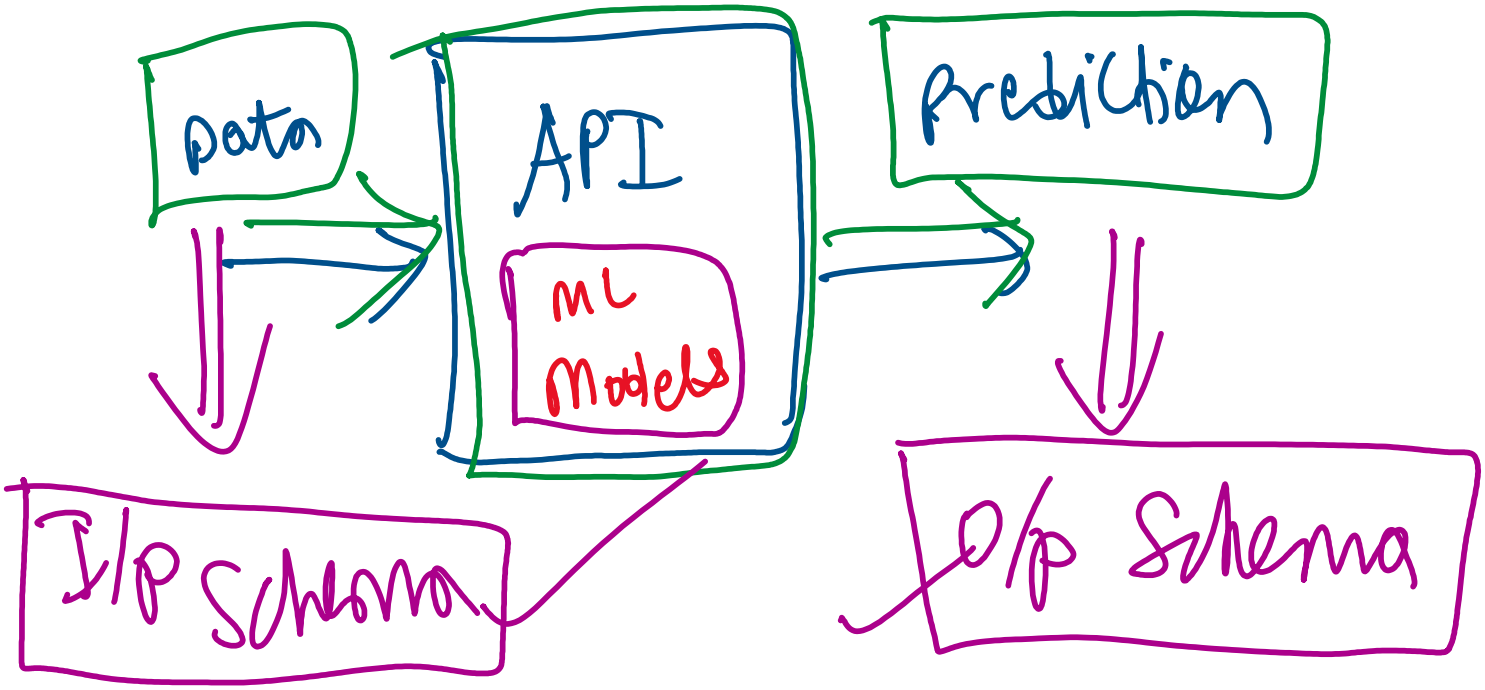
4. **Cleaner Code and Reusability**

5. **Secure and Robust APIs**

6. **Managing Nested & Complex Structures:**
   - ML applications often involve:
     - Nested JSON structures
     - Optional fields
     - Lists of structured items
   - Schemas make these easy to define and validate using **BaseModel**, **Optional**, **List**, etc.

7. **Logging, Auditing, and Testing:**
   - Well-defined schemas simplify structured logging
   - Makes it easier to write tests with known input/output formats

- Help trace issues back to specific schema validation failures



4. Serving ML Models

Sunday, May 25, 2025 12:39 PM

Why Serve ML Models via FastAPI?

1. **Decoupling Model from Application Logic:**
   - Keeps ML logic separate from UI or business logic
   - Enables modular, reusable models that can be consumed by multiple applications (web apps, mobile apps, internal tools, etc.) via HTTP requests

2. **Real-Time Predictions:**
   - Clients can send input and receive predictions instantly via a /predict endpoint
   - Essential for use-cases like **fraud detection**, **recommendation engines**, or **medical triaging** where immediate inference is needed

3. **Platform-Agnostic Integration:**
   - The model is accessible through a standard REST API
   - Any frontend, mobile app, or backend service, regardless of language (JS, Java, etc.), can access the model using HTTP

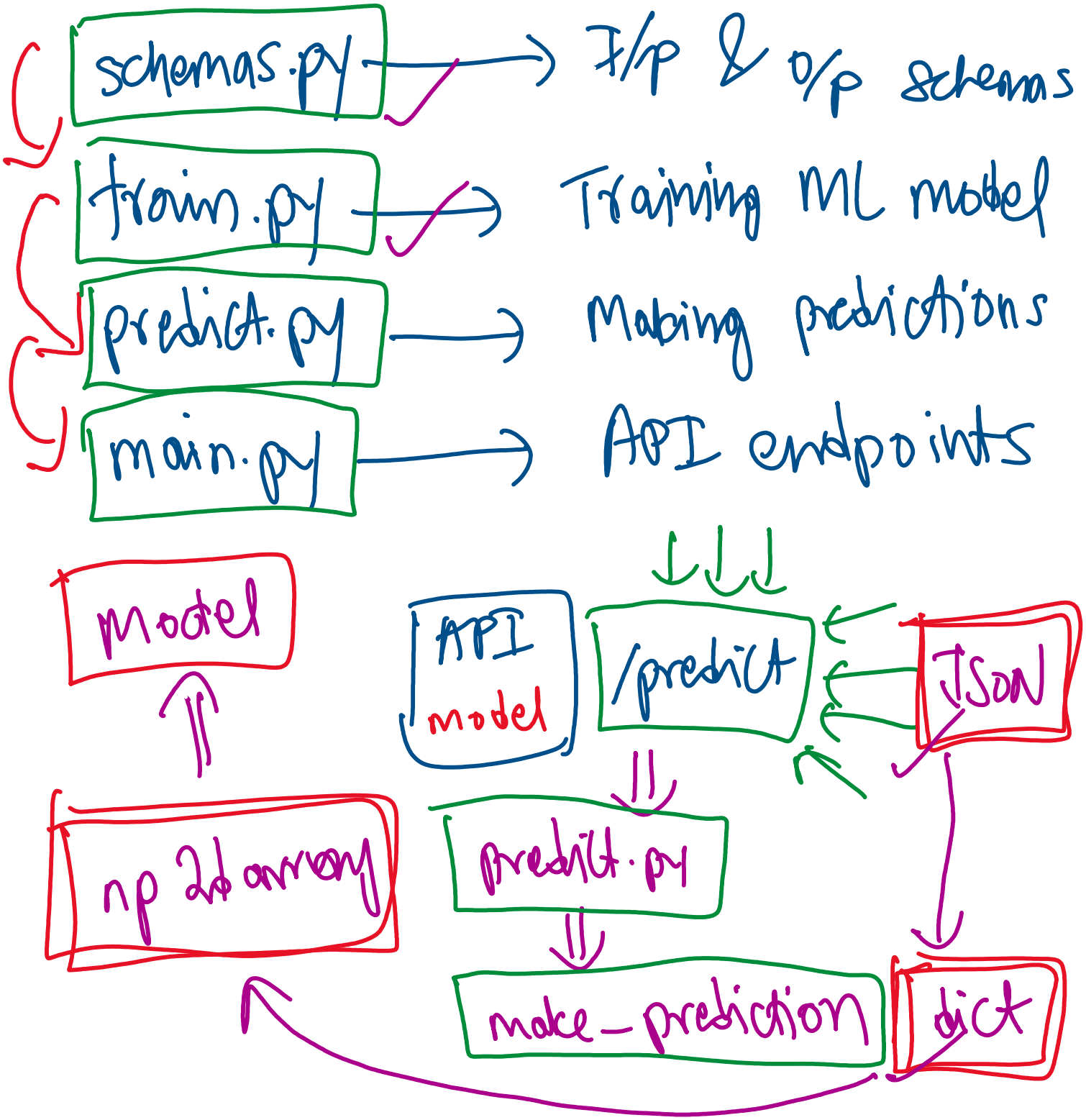
4. **Production-Readiness:**
   - FastAPI supports ASGI, async I/O, and is built for performance
   - FastAPI is faster than Flask, with excellent concurrency, making it scalable for real-world traffic

5. **Built-in Validation with Pydantic:**
   - Ensures clean, validated data before it reaches the model (**InputSchema**)
   - The response is returned in a structured format (**OutputSchema**)
   - Prevents runtime errors due to bad input, reduces bugs, and improves model reliability

6. **Docker & Cloud Friendly:**
   - FastAPI apps can be containerized and deployed on AWS, Azure, GCP, Hugging Face, Render, etc.
   - Perfect fit for **CI/CD pipelines**, **Kubernetes**, and serverless deployment models

7. **Scalable Infrastructure:**

- Works with ASGI servers like Uvicorn for high-concurrency
- Allows to serve thousands of predictions per second with proper load balancing



5. Handling Batch Predictions

Sunday, May 25, 2025 2:03 PM

Vectorized Predictions for Speed:

- When making predictions using a machine learning model, especially for multiple data points, it's inefficient to loop over each input and call the **.predict()** method one-by-one
- Instead, we should leverage **vectorized predictions**, i.e., send a whole batch into the model at once
- This drastically improves performance due to:
  - Optimized linear algebra libraries under the hood
  - Reduced I/O overhead
  - Parallelized CPU/GPU execution

```
predictions = [model.predict([x]) for x in input_list]
```

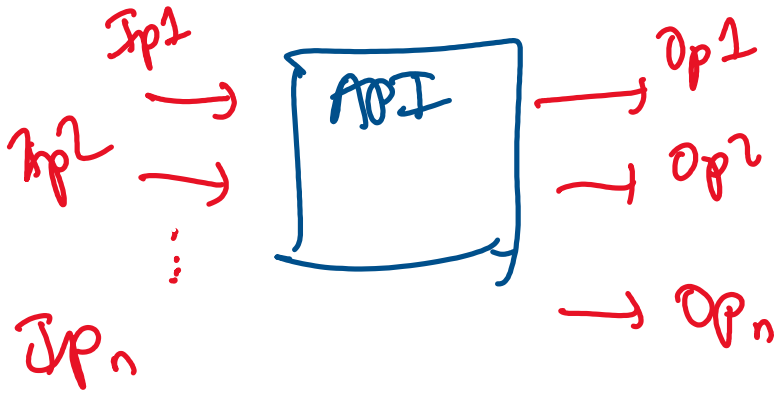
```
predictions = model.predict(input_array)
```

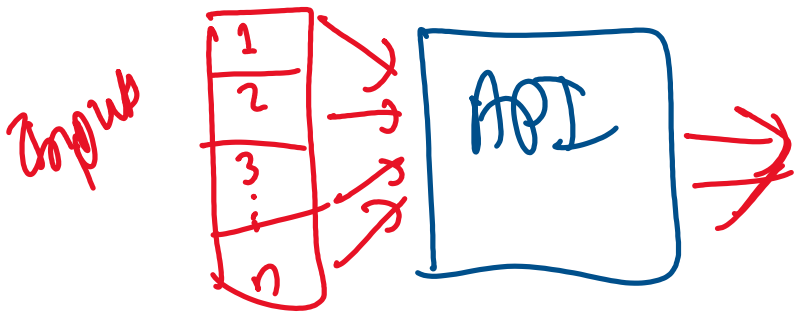
Accepting List of Inputs:

- To handle batch predictions via API, the endpoint should accept a list of input objects (typically JSON dictionaries)
- This means instead of a single input, the user will **POST** a list of inputs to the endpoint

Benefits:

FEATURE	BENEFIT
Vectorized input	Faster computation
List validation via Pydantic	Safer and cleaner error handling
JSON input/output	Easy integration with frontend/clients
Response model	Ensures consistent and documented API output




Queue
1
2
3
n
API