APIs & Model Demos

Week 12 : CS 203 - Software Tools and Techniques for AI

Prof. Nipun Batra

IIT Gandhinagar

Your Model is Trapped

You built an amazing sentiment classifier. Your friend says **"Can I try it?"**

You reply:

> "Sure! Install Python 3.10, create a conda environment, pip install these 14 packages, download the weights from Google Drive, open the notebook, and run cell #4. Oh, and don't run cell #3 — that retrains."

Your friend:
😐

The Rescue Plan

Week	What We Built	What It Solves
Week 9	Git	"Which version is the right one?"
Week 10	Environments & Docker	"It doesn't work on my machine"
Week 11	CI/CD	"Who broke the tests?"
Week 12	APIs & Demos	"How do I actually USE this model?"

Today we climb three steps:

```
API          →  machines can talk to your model
   GUI       →  humans can use your model
      Cloud  →  the whole world can reach your model
```

Part 1: APIs

Teaching machines to talk to your model

What Is an API?

Analogy: The Drive-Thru Window

You don't walk into the kitchen. You:

1. Read the **menu** (documentation)
2. Place an **order** at the window (request)
3. Get **food in a bag** (response)

You don't care HOW they cook it. You just get what you ordered.

API = Application Programming Interface. A contract between two programs.

HTTP in 30 Seconds

For ML, you only need **two** HTTP methods:

Method	What It Does	Example
GET	Read / retrieve info	`GET /` — "Is the server alive?"
POST	Send data, get result	`POST /predict` — "Here are features, give me a prediction"

And **four** status codes:

Code	Meaning	Plain English
200	OK	"Here's your prediction"
400	Bad Request	"I don't understand what you sent"
404	Not Found	"That endpoint doesn't exist"
500	Server Error	"Something crashed on my side"

That's it. You now know enough HTTP.

JSON — The Universal Translator

JSON is just text that any programming language can read.

Python dict:

```
features = {
    "budget": 50.0,
    "runtime": 120,
    "genres": ["action", "drama"]
}
```

JSON (what gets sent over the wire):

```
{
    "budget": 50.0,
    "runtime": 120,
    "genres": ["action", "drama"]
}
```

They look the same! Python dicts map directly to JSON.

`json.dumps(dict)` → JSON string | `json.loads(string)` → Python dict

Meet FastAPI — Hello World

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def health():
    return {"status": "alive", "model": "movie-predictor-v1"}
```

Run it:

```
pip install fastapi uvicorn
uvicorn app:app --reload
```

Now visit:

- `http://localhost:8000` → your health check
- `http://localhost:8000/docs` → **auto-generated interactive docs** (try it!)

5 lines of code. A working API. Free documentation.

Pydantic — The Bouncer at the Door

Analogy: A bouncer checking IDs. Wrong ID? You don't get in.

```python
from pydantic import BaseModel, Field

class MovieFeatures(BaseModel):
    budget: float = Field(..., gt=0, description="Budget in millions")
    runtime: float = Field(..., gt=0, le=300, description="Runtime in minutes")
    genre_action: int = Field(..., ge=0, le=1)
    genre_comedy: int = Field(..., ge=0, le=1)
```

Send `{"budget": -5, "runtime": 120, ...}` and you get back:

```
{
  "detail": [{"loc": ["body", "budget"],
              "msg": "ensure this value is greater than 0"}]
}
```

Automatic 422 error. No manual if-checks needed. The bouncer does the work.

Serving a Model — The Money Slide

```python
from fastapi import FastAPI
import joblib, numpy as np

# Load model ONCE at startup — not on every request!
model = joblib.load("model.pkl")
app = FastAPI(title="Movie Predictor API")

class MovieFeatures(BaseModel):
    budget: float = Field(..., gt=0)
    runtime: float = Field(..., gt=0, le=300)
    genre_action: int = Field(..., ge=0, le=1)
    genre_comedy: int = Field(..., ge=0, le=1)

@app.post("/predict")
def predict(features: MovieFeatures):
    X = np.array([[features.budget, features.runtime,
```

That's it. Your model is no longer trapped.

Part 2: Demo UIs

Teaching humans to use your model

APIs Are Ugly for Humans

Your API works. But here's what using it looks like:

```
curl -X POST http://localhost:8000/predict \
  -H "Content-Type: application/json" \
  -d '{"budget": 50, "runtime": 120, "genre_action": 1, "genre_comedy": 0}'
```

Your mom doesn't want to type that. Your project evaluator doesn't either.

We need a face for our model.

Gradio — The Pop-Up Kiosk

Analogy: A quick cardboard stand at a college fest. Takes 5 minutes to set up, looks decent, does the job.

```
import gradio as gr
import joblib

model = joblib.load("model.pkl")

def predict(budget, runtime, genre):
    features = [budget, runtime, int(genre == "Action"), int(genre == "Comedy")]
    pred = model.predict([features])[0]
    return "Hit!" if pred else "Flop"

demo = gr.Interface(
    fn=predict,
    inputs=[gr.Number(label="Budget ($M)"),
            gr.Slider(60, 240, label="Runtime (min)"),
            gr.Dropdown(["Action", "Comedy", "Drama"], label="Genre")],
    outputs=gr.Text(label="Prediction"),
```

15 lines. A real web UI with form validation.

Gradio — Not Just Text Boxes

Gradio handles all kinds of ML inputs out of the box:

```
# Image classification
gr.Interface(fn=classify, inputs=gr.Image(type="numpy"),
             outputs=gr.Label(num_top_classes=5))

# Audio from microphone
gr.Interface(fn=transcribe, inputs=gr.Audio(sources=["microphone"]),
             outputs=gr.Textbox())

# Sketch pad for digit recognition
gr.Interface(fn=recognize, inputs=gr.Sketchpad(),
             outputs=gr.Label())
```

Image upload, audio recording, webcam, drawing pad — all built in.

No need to write file-upload logic yourself.

Gradio — Share With the World

```
# Local only (default)
demo.launch()
# → http://127.0.0.1:7860

# Public URL in 1 second
demo.launch(share=True)
# → https://abc123.gradio.live  (free, expires in 72h)

# Permanent free hosting on Hugging Face Spaces
# 1. Create a Space at huggingface.co/spaces
# 2. git push your code + model file
# 3. Done — free hosting, free GPU available!
```

`share=True` creates a tunnel to your laptop. Anyone with the link can use your model.

For class projects: push to **Hugging Face Spaces** for a permanent link.

Streamlit — The Mission Control Dashboard

When you need charts, tables, sidebars, and multiple pages.

```
import streamlit as st
import joblib

model = joblib.load("model.pkl")
st.title("Movie Success Predictor")

col1, col2 = st.columns(2)
with col1:
    budget = st.number_input("Budget ($M)", min_value=0.1, value=50.0)
    runtime = st.slider("Runtime (min)", 60, 240, 120)
with col2:
    genre = st.selectbox("Genre", ["Action", "Comedy", "Drama"])

if st.button("Predict"):
    features = [budget, runtime, int(genre == "Action"), int(genre == "Comedy")]
    pred = model.predict([features])[0]
```

```
streamlit run app.py
```

Gradio vs Streamlit

	Gradio	Streamlit
Best for	Quick ML demo	Data dashboard
Feels like	A pop-up kiosk	Mission control
Setup	Define function → UI appears	Write script top-to-bottom
Rich ML inputs	Image, audio, video, sketch	Standard widgets
Sharing	`share=True` → instant URL	Streamlit Community Cloud
Hugging Face	First-class support	Also supported

Need to show your model to someone in 5 minutes? → Gradio
Building an internal dashboard with charts? → Streamlit

For this course, either works. Gradio is usually faster for ML projects.

Part 3: Deployment

From your laptop to the world

Online vs Batch Inference

	Online (Real-time)	Batch
Flow	User clicks → prediction in ~100ms	Upload 10K rows → results in an hour
When	User is waiting	Nobody is waiting
Example	"Is this transaction fraud?"	"Score all customers for churn"
Tech	FastAPI + server	Script + cron job

Today we build online inference.

Batch inference is just a Python script with a for-loop — nothing fancy.

Model Serialization — Save It to Serve It

Before serving, you need to save your trained model to a file.

Format	Library	Save	Load
joblib	scikit-learn	`joblib.dump(model, "model.pkl")`	`joblib.load("model.pkl")`
state_dict	PyTorch	`torch.save(model.state_dict(), "model.pth")`	`model.load_state_dict(torch.load(...))`
ONNX	Any → portable	`torch.onnx.export(model, ...)`	`onnxruntime.InferenceSession(...)`

For this course: `joblib` for sklearn, `torch.save` for PyTorch.

ONNX is the "PDF of models" — portable across languages and frameworks.

Where Does It Go?

Platform	Cost	Effort	Best For
Hugging Face Spaces	Free	Very low	Gradio/Streamlit demos
Railway / Render	Free tier	Low	Simple FastAPI apps
Google Cloud Run	Pay-per-use	Medium	Production APIs

For class projects: Hugging Face Spaces. Free, easy, supports GPU.

Steps:

1. Create a Space at `huggingface.co/spaces`
2. Choose Gradio or Streamlit SDK
3. Push your code + model file
4. Your app is live at `huggingface.co/spaces/yourname/yourapp`

Docker for APIs — Quick Recap

Remember Docker from Week 10? Same idea, now with `uvicorn`:

```
FROM python:3.10-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY model.pkl app.py .
CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "8000"]
```

```
docker build -t movie-api .
docker run -p 8000:8000 movie-api
```

Same container runs on your laptop, your teammate's laptop, and the cloud.

Tip: ML APIs keep services **stateless** — same input always gives same output, so you can run multiple copies behind a load balancer.

The Full Journey

You went from "run cell #4" to a link anyone can click.

Key Takeaways

1. **APIs** let any program use your model
   - FastAPI: 5 lines for a working API, free docs at `/docs`
   - Pydantic: automatic input validation (the bouncer)
   - Load your model once at startup, not per request
2. **Demo UIs** let any human use your model
   - Gradio: quick ML demo, `share=True` for instant public link
   - Streamlit: richer dashboards with charts and tables
3. **Deployment** lets the whole world use your model
   - Hugging Face Spaces: free, easy, perfect for class projects
   - Docker: same container works everywhere

Next week: Make it fast and small — profiling & quantization