From Script to Web App

Week 10: CS 203 - Software Tools and Techniques for AI

Prof. Nipun Batra

IIT Gandhinagar

The Problem

You trained a great model:

```
model.predict(["WINNER! Free iPhone! Click NOW!"])
# → "Spam"
```

But right now:

- Only **you** can run it (on your laptop, in a terminal)
- Your teammate can't try it
- Your professor can't evaluate it
- The world can't see it

Today: 3 ways to put a UI on your model, then ship it to the world.

Today's Plan

We'll deploy the **same spam classifier** three different ways:

#	Framework	Philosophy	Lines of code
1	**Flask**	Write your own HTML + Python backend	~40
2	**Streamlit**	Pure Python, no HTML needed	~15
3	**Gradio**	Just define inputs & outputs	~10

Then we'll **deploy Gradio to HuggingFace Spaces** — free, public URL in 2 minutes.

Clone the repo and follow along:

```
cd lecture-demos/week10/web-apps
```

Step 0: The Model

Before any web app, we need a trained model saved to disk.

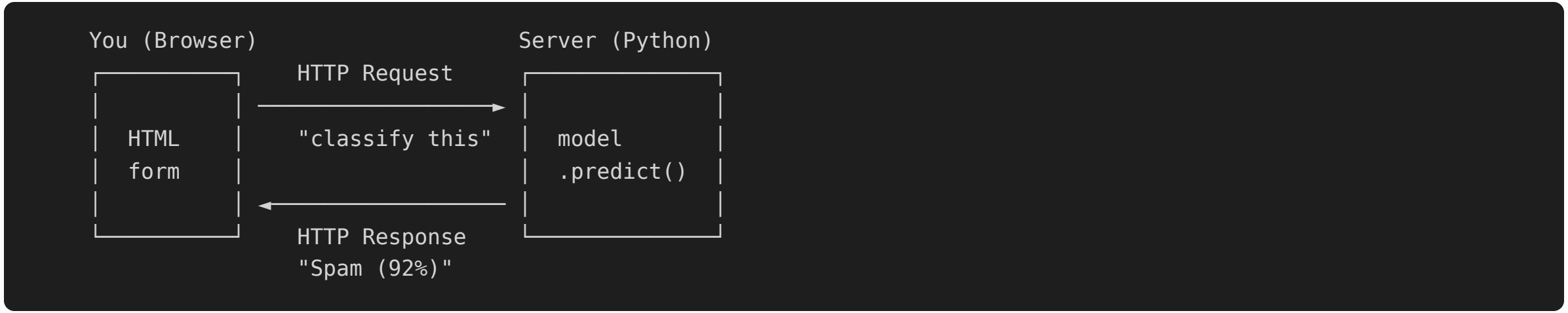
```
from sklearn.pipeline import make_pipeline
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
import joblib

pipeline = make_pipeline(
    TfidfVectorizer(max_features=5000, stop_words="english"),
    MultinomialNB()
)
pipeline.fit(X_train, y_train)
joblib.dump(pipeline, "spam_model.pkl")
```

```
cd 0-model && python train_model.py
```

This produces `spam_model.pkl` — a single file containing the entire pipeline.

What is a Web App?



Frontend = what the user sees (HTML, buttons, text boxes)
Backend = where the ML runs (Python, model loading, prediction)

Flask: you write both. Streamlit/Gradio: they generate the frontend for you.

1. Flask

You write the HTML. You write the Python. Full control.

Flask: The Backend (app.py)

```
from flask import Flask, request, render_template
import joblib

app = Flask(__name__)
model = joblib.load("../0-model/spam_model.pkl")  # load ONCE

@app.route("/", methods=["GET", "POST"])
def home():
    prediction = None
    if request.method == "POST":
        text = request.form["text"]
        proba = model.predict_proba([text])[0]
        pred_class = model.predict([text])[0]
        prediction = "Spam" if pred_class == 1 else "Not Spam"
        confidence = f"{max(proba) * 100:.1f}%"
    return render_template("index.html",
```

Flask: The Frontend (templates/index.html)

```
<h1>Spam Classifier</h1>
<form method="POST">
    <textarea name="text" placeholder="Type a message..."></textarea>
    <button type="submit">Classify</button>
</form>

{% if prediction %}
    <p>Prediction: {{ prediction }}</p>
    <p>Confidence: {{ confidence }}</p>
{% endif %}
```

Two files, two languages (Python + HTML). You have full control over the look and feel.

```
cd 1-flask && python app.py    # → http://localhost:5000
```

Flask: The Pain Point

For a simple text box and button, we needed:

- `app.py` — 20 lines of Python (routes, form parsing, template rendering)
- `templates/index.html` — 20 lines of HTML + CSS
- Knowledge of HTTP methods (GET vs POST)
- Jinja2 templating (`{{ prediction }}`, `{% if %}`)

What if you just want a quick demo and don't care about HTML?

2. Streamlit

Pure Python. No HTML. Hot-reload.

Streamlit: The Entire App

```
import streamlit as st
import joblib

@st.cache_resource
def load_model():
    return joblib.load("../0-model/spam_model.pkl")

model = load_model()

st.title("Spam Classifier")
text = st.text_area("Message", placeholder="Type a message...")

if st.button("Classify"):
    proba = model.predict_proba([text])[0]
    pred_class = model.predict([text])[0]
    label = "Spam" if pred_class == 1 else "Not Spam"
    confidence = max(proba) * 100
```

Streamlit: What Just Happened?

No HTML. No CSS. No templates folder. No routes.

Flask	Streamlit
`<textarea name="text">`	`st.text_area("Message")`
`<button type="submit">`	`st.button("Classify")`
`render_template(...)`	`st.write(...)`
Restart server to see changes	**Hot-reload on save**

```
cd 2-streamlit && streamlit run app.py    # → http://localhost:8501
```

Try changing the title, save the file, and watch the browser update instantly.

Streamlit: `@st.cache_resource`

```
@st.cache_resource
def load_model():
    return joblib.load("spam_model.pkl")
```

Why? Streamlit re-runs the **entire script** top-to-bottom on every interaction (button click, text change). Without caching, the model would reload from disk every single time.

`@st.cache_resource` = "run this function once, remember the result forever."

3. Gradio

Define inputs + outputs. That's it.

Gradio: The Entire App

```
import gradio as gr
import joblib

model = joblib.load("../0-model/spam_model.pkl")

def classify(text):
    proba = model.predict_proba([text])[0]
    return {"Not Spam": float(proba[0]), "Spam": float(proba[1])}

demo = gr.Interface(
    fn=classify,
    inputs=gr.Textbox(label="Message", lines=4),
    outputs=gr.Label(label="Prediction"),
    title="Spam Classifier",
    examples=[
        ["WINNER!! Free iPhone! Click NOW!"],
```

```
cd 3-gradio && python app.py    # → http://localhost:7860
```

Gradio: Built-in Superpowers

Things you get **for free** that Flask/Streamlit don't give you:

Feature	How
Examples	Click-to-fill sample inputs
Flagging	Users can flag bad predictions
API endpoint	Click "Use via API" at the bottom
Share link	`demo.launch(share=True)` → public URL

The API endpoint is huge — it means **other code** can call your model:

```
from gradio_client import Client
client = Client("http://localhost:7860")
result = client.predict("Free iPhone!", api_name="/predict")
```

When to Use What?

Framework	Best For	Trade-off
Flask	Full custom web apps, APIs, production sites	You write everything
Streamlit	Data dashboards, internal tools, EDA apps	Less control over layout
Gradio	ML model demos, quick sharing, HF ecosystem	Limited to input→output pattern

Rule of thumb:

- Need a custom UI or multi-page app? → **Flask**
- Need charts, tables, data exploration? → **Streamlit**
- Need to demo an ML model quickly? → **Gradio**

4. FastAPI

For when machines talk to machines

Flask/Streamlit/Gradio = Humans Use Your Model

But what if **another program** needs to call your model?

- A mobile app sending text to your spam classifier
- A frontend (React/Vue) calling your prediction backend
- Another ML pipeline consuming your model's output

You need an **API** — no HTML, no buttons. Just JSON in, JSON out.

```
POST /predict
{"text": "WINNER! Free iPhone!"}

→ {"label": "Spam", "confidence": 92.3}
```

FastAPI = modern Python framework for building production APIs.

FastAPI: The Entire App

```
from fastapi import FastAPI
from pydantic import BaseModel
import joblib

app = FastAPI(title="Spam Classifier API")
model = joblib.load("../0-model/spam_model.pkl")

class Message(BaseModel):
    text: str

@app.post("/predict")
def predict(msg: Message):
    proba = model.predict_proba([msg.text])[0]
    pred_class = model.predict([msg.text])[0]
    return {
        "label": "Spam" if pred_class == 1 else "Not Spam",
```

```
cd 4-fastapi && uvicorn app:app --reload --port 8000
```

FastAPI: Auto-Generated Docs

Open `http://localhost:8000/docs` — you get **Swagger UI for free**:

- Interactive API documentation
- "Try it out" button to test your endpoint
- Request/response schema auto-generated from your Pydantic models

Test from the terminal:

```
curl -X POST http://localhost:8000/predict \
     -H "Content-Type: application/json" \
     -d '{"text": "WINNER! Free iPhone!"}'
```

```
{"label": "Spam", "confidence": 92.3}
```

No HTML. No UI. Just a clean JSON API.

Flask vs FastAPI

	Flask	FastAPI
Purpose	Web apps with HTML	JSON APIs
Validation	Manual (`request.form`)	Automatic (Pydantic models)
Docs	None (write your own)	Auto-generated Swagger UI
Async	No (by default)	Yes (built-in)
Speed	Good	Excellent
When to use	Full web apps with pages	Machine-to-machine APIs

For ML deployment: **Gradio** for demos, **FastAPI** for production APIs.

Deploying to Hugging Face Spaces

From localhost to a public URL in 2 minutes

HuggingFace Spaces: Why?

Right now your app runs on `localhost` — only you can see it.

HuggingFace Spaces = free hosting for Gradio and Streamlit apps.

What you need:

1. A `app.py` file
2. A `requirements.txt` file
3. A HuggingFace account

That's it. No server setup, no domain name, no cloud billing.

HuggingFace Spaces: How

Option A: Web upload

1. Go to huggingface.co/spaces
2. Click "Create new Space" → choose Gradio
3. Upload `app.py` and `requirements.txt`
4. Wait for build → get public URL

Option B: Git push

```
# Clone your space
git clone https://huggingface.co/spaces/YOUR_USERNAME/spam-classifier
cd spam-classifier

# Add your files
cp app.py requirements.txt .
git add . && git commit -m "Add spam classifier"
git push
```

HuggingFace Spaces: The Gotcha

The #1 beginner mistake:

Your local app loads the model from a relative path:

```
model = joblib.load("../0-model/spam_model.pkl")  # works locally
```

On HuggingFace, there is no `../0-model/` folder. You must:

1. **Include the model file** in your Space repo
2. **Update the path**: `model = joblib.load("spam_model.pkl")`

Your Space's `requirements.txt` must list **everything**:

```
scikit-learn
joblib
gradio
```

If you forget `scikit-learn`, the build will fail with `ModuleNotFoundError`.

HF Spaces with Docker

HF Spaces also supports **Docker** — perfect for FastAPI or custom setups.

Create a `Dockerfile` in your Space repo:

```
FROM python:3.10-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY app.py spam_model.pkl ./
ENV GRADIO_SERVER_NAME="0.0.0.0"
EXPOSE 7860
CMD ["python", "app.py"]
```

HuggingFace auto-detects the Dockerfile, builds the image, and deploys it.

This is how real ML apps get deployed — your Dockerfile becomes the deployment specification.

The Full Comparison

Same model. Same prediction. Five different developer experiences.

	Flask	Streamlit	Gradio	FastAPI
Files	app.py + HTML	app.py	app.py	app.py
Languages	Python + HTML	Python	Python	Python
Has UI	Yes (custom)	Yes (auto)	Yes (auto)	No (JSON API)
Has API	Manual	No	Auto	Yes
Auto-docs	No	No	No	Swagger UI
HF Spaces	No	Yes	Yes	Yes (Docker)
Best for	Custom web	Dashboards	ML demos	Production APIs

Common Mistakes

1. Loading the model inside the route/button handler:

```
# BAD — reloads from disk on EVERY request
@app.route("/predict")
def predict():
    model = joblib.load("model.pkl")  # 5 seconds every click!
```

2. Forgetting `requirements.txt` when deploying:

```
ModuleNotFoundError: No module named 'sklearn'
```

3. Hardcoding localhost paths:

```
model = joblib.load("/Users/nipun/Desktop/model.pkl")  # won't work anywhere else
```

Key Takeaways

1. `joblib.dump` / `joblib.load` — save your model to a `.pkl` file
2. **Flask** — full control, but you write HTML yourself
3. **Streamlit** — pure Python UI, great for dashboards, hot-reload
4. **Gradio** — ML-focused, built-in examples/API/sharing
5. **FastAPI** — production JSON APIs with auto-generated docs
6. **HuggingFace Spaces** — free deployment for Gradio/Streamlit/Docker apps
7. **Load the model once** at startup, not inside every request
8. `requirements.txt` — your app is useless without it