Building Interactive AI Demos

Week 9 · CS 203: Software Tools and Techniques for AI

Prof. Nipun Batra

IIT Gandhinagar

The Demo Problem

Your situation:

- You built an amazing Netflix movie predictor
- It works perfectly in your Jupyter notebook
- Your professor asks: "Can I try it?"

Options:

1. "Uh... install Python, pandas, sklearn, then run this notebook..."
2. "Here's a link! Just click and use it."

Which sounds better?

Why Build Demos?

For feedback:

- Non-coders can test your model
- Find edge cases you missed
- Iterate based on real usage

For sharing:

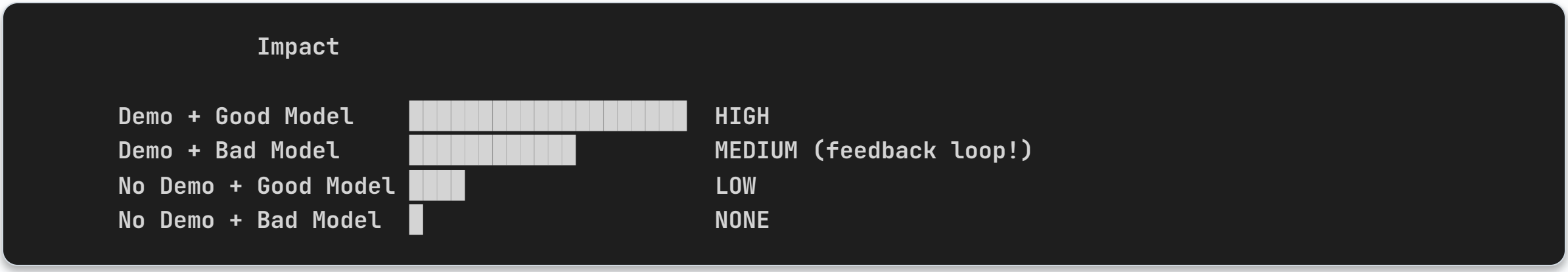
- A URL is worth 1000 GitHub stars
- Stakeholders can see your work
- Portfolio projects that impress

For learning:

- Understand your model better
- See how users interact with it

The Demo Paradox

The best models often die in notebooks. You can have 99% accuracy, but if no one can try it, does it matter? A mediocre model with a great demo gets feedback, funding, and users. A perfect model in a notebook gets forgotten.



The demo is what makes the model useful.

The Demo Spectrum

Approach	Time	When to Use
Jupyter Notebook	0 hrs	Personal use
Streamlit/Gradio	2 hrs	Demos, prototypes
FastAPI + Frontend	1 - 2 days	Internal tools
Full Web App	2+ weeks	Production

Today's focus: The 2-hour solution (Streamlit & Gradio)

Connection to Our Netflix Project

```
Week 1-7: Built the movie predictor model
Week 8:   Made it reproducible
          ↓
Week 9:   Let ANYONE use it!
          - No Python required
          - Just a web browser
          - Interactive interface
```

Part 1: Streamlit Basics

From notebook to web app in minutes

What is Streamlit?

Streamlit turns Python scripts into web apps.

You write:

```python
import streamlit as st

st.title("Hello World!")
name = st.text_input("What's your name?")
if name:
    st.write(f"Hello, {name}!")
```

You get:

A fully functional web page with a title, text input, and dynamic output.

No HTML, CSS, or JavaScript required!

Installing and Running Streamlit

Install:

```
pip install streamlit
```

Create a file `app.py`:

```
import streamlit as st
st.title("My First App")
st.write("Hello, Streamlit!")
```

Run:

```
streamlit run app.py
```

Opens in browser at: http://localhost:8501

Streamlit Input Widgets

Text inputs:

```
name = st.text_input("Enter your name")
bio = st.text_area("Tell us about yourself")
```

Numbers:

```
age = st.number_input("Your age", min_value=0, max_value=120)
rating = st.slider("Rate this movie", 1, 10, 5)
```

Choices:

```
genre = st.selectbox("Pick a genre", ["Action", "Comedy", "Drama"])
genres = st.multiselect("Pick genres", ["Action", "Comedy", "Drama"])
```

Files:

```
uploaded_file = st.file_uploader("Upload a CSV")
```

Streamlit Display Elements

Text:

```
st.title("Big Title")
st.header("Section Header")
st.write("Regular text or markdown")
st.markdown("**Bold** and *italic*")
```

Data:

```
st.dataframe(df)        # Interactive table
st.table(df)            # Static table
st.json({"key": "value"})
```

Media:

```
st.image("photo.jpg", caption="My photo")
st.audio("song.mp3")
st.video("movie.mp4")
```

Building a Netflix Demo: Step 1

Basic structure:

```
import streamlit as st
import pandas as pd
import pickle

# Title
st.title("Netflix Movie Success Predictor")
st.write("Will your movie be a hit? Let's find out!")

# Load the model (we trained this earlier)
@st.cache_resource
def load_model():
    with open("model.pkl", "rb") as f:
        return pickle.load(f)

model = load_model()
```

Building a Netflix Demo: Step 2

Add user inputs:

```
st.header("Enter Movie Details")

genre = st.selectbox("Genre", ["Action", "Comedy", "Drama", "Horror"])
budget = st.slider("Budget (millions $)", 1, 300, 50)
runtime = st.slider("Runtime (minutes)", 60, 240, 120)
is_sequel = st.checkbox("Is this a sequel?")

# Create feature vector
features = pd.DataFrame({
    "genre": [genre],
    "budget": [budget],
    "runtime": [runtime],
    "is_sequel": [int(is_sequel)]
})
```

Building a Netflix Demo: Step 3

Make prediction and show results:

```
if st.button("Predict Success"):
    # Get prediction
    prediction = model.predict(features)[0]
    probability = model.predict_proba(features)[0]

    # Display result
    st.header("Prediction")

    if prediction == 1:
        st.success(f"This movie will likely SUCCEED!")
        st.balloons()
    else:
        st.error(f"This movie might struggle...")

    # Show confidence
    st.write(f"Confidence: {max(probability)*100:.1f}%")
```

The @st.cache Decorators

Problem: Streamlit reruns your entire script on every interaction.

Loading a model every time = Slow!

Solution: Cache it!

```
# Cache the model (load once, reuse forever)
@st.cache_resource
def load_model():
    return pickle.load(open("model.pkl", "rb"))

# Cache data computations
@st.cache_data
def load_data(url):
    return pd.read_csv(url)
```

`@st.cache_resource` : For models, database connections
`@st.cache_data` : For data that depends on inputs

Why Caching Matters: The Rerun Model

Streamlit's mental model: Every widget interaction reruns your ENTIRE script from top to bottom. This is simple but naive - without caching, you'd reload your 500MB model every time a user moves a slider!

Cache = "Remember this so we don't do it again."

Adding Loading Feedback

Users hate waiting without feedback!

```
# Spinner for loading
with st.spinner("Loading model..."):
    model = load_model()
st.success("Model loaded!")

# Progress bar for long operations
progress = st.progress(0)
for i in range(100):
    progress.progress(i + 1)
    time.sleep(0.01)

# Status messages
st.info("Processing your request...")
st.warning("This might take a moment...")
st.error("Something went wrong!")
st.success("Done!")
```

Layout with Columns

Side-by-side content:

```
col1, col2 = st.columns(2)

with col1:
    st.header("Input")
    genre = st.selectbox("Genre", ["Action", "Comedy"])
    budget = st.slider("Budget", 1, 300)

with col2:
    st.header("Output")
    st.write(f"You selected: {genre}")
    st.write(f"Budget: ${budget}M")
```

Useful for: Input on left, output on right.

Sidebar for Controls

Keep controls separate:

```
# Sidebar inputs
st.sidebar.header("Settings")
model_type = st.sidebar.selectbox("Model", ["Random Forest", "XGBoost"])
threshold = st.sidebar.slider("Confidence threshold", 0.0, 1.0, 0.5)

# Main content
st.title("Movie Predictor")
st.write(f"Using {model_type} with threshold {threshold}")
```

Sidebar stays visible while scrolling!

Part 2: Gradio

Even simpler for ML demos

What is Gradio?

Gradio is optimized for "input → model → output" demos.

```
import gradio as gr

def predict(text):
    return f"You said: {text}"

demo = gr.Interface(
    fn=predict,
    inputs="text",
    outputs="text"
)
demo.launch()
```

That's it! A working web interface.

Gradio Input/Output Types

Inputs:

```
"text"      # Single line text
"textbox"   # Multi-line text
"image"     # Image upload
"audio"     # Audio file
"file"      # Any file
"slider"    # Numeric slider
"checkbox"  # Boolean
```

Outputs:

```
"text"      # Text display
"image"     # Show image
"label"     # Classification result
"json"      # JSON display
"dataframe" # Pandas table
```

Gradio for Image Classification

```
import gradio as gr
from PIL import Image

def classify(image):
    # Your model prediction here
    prediction = model.predict(image)
    return {
        "Cat": prediction[0],
        "Dog": prediction[1],
        "Bird": prediction[2]
    }

demo = gr.Interface(
    fn=classify,
    inputs="image",
    outputs="label",
    title="Animal Classifier"
)
demo.launch()
```

Streamlit vs Gradio

Feature	Streamlit	Gradio
Best for	Dashboards, multi-page apps	Quick model demos
Code style	Imperative (step by step)	Declarative (define interface)
Flexibility	High	Medium
Learning curve	Medium	Low
Hugging Face Spaces	Yes	Native support

Rule of thumb:

- Simple model demo → Gradio
- Dashboard or complex app → Streamlit

The Mental Model Difference

Streamlit thinks in pages. Gradio thinks in functions. Streamlit is "I want to build a web app that happens to have ML". Gradio is "I have a function, make it web-accessible". Choose based on how you think about your project.

```
# Gradio: Define the function, wrap it
def predict(x): return model(x)
gr.Interface(fn=predict, inputs="text", outputs="label")

# Streamlit: Build the page, call the function
st.title("My App")
x = st.text_input("Input")
if st.button("Predict"):
    st.write(predict(x))
```

Neither is better - they're different mental models.

Part 3: Deployment

Share your demo with the world

Deployment Options

Platform	Cost	Best For
Hugging Face Spaces	Free	Public demos
Streamlit Cloud	Free	Streamlit apps
Render	Free tier	General apps
Railway	Pay-as-go	More control
Heroku	Pay-as-go	Established platform

For this course: Hugging Face Spaces (free, easy)

Deploying to Hugging Face Spaces

Step 1: Create account at huggingface.co

Step 2: Create new Space

- Choose Streamlit or Gradio
- Give it a name

Step 3: Add files

- `app.py` - your application code
- `requirements.txt` - dependencies

Step 4: Push to Space (like Git)

```
git clone https://huggingface.co/spaces/username/myapp
# Add files
git add .
git commit -m "Initial commit"
```

requirements.txt for Deployment

```
streamlit==1.28.0
pandas==2.0.0
scikit-learn==1.3.0
numpy==1.24.0
```

Important:

- Pin versions (avoid `numpy` without version)
- Include ALL dependencies
- Test locally first

Secrets Management

Never put API keys in code!

Streamlit Cloud:

1. Go to app settings
2. Add secrets in TOML format:

```
OPENAI_API_KEY = "sk-..."
```

In code:

```
import streamlit as st
```

Hugging Face Spaces:

- Use Settings → Repository secrets

Part 4: UX Best Practices

Making demos people actually want to use

The Psychology of Good Demos

Users form opinions in 3 seconds. If your demo looks broken, confusing, or slow when they first load it, they'll leave. First impressions aren't just important - they're everything.

User Psychology	Design Response
Short attention span	Show results quickly
Fear of breaking things	Clear, forgiving inputs
Uncertainty about what to do	Examples and defaults
Frustration with waiting	Progress indicators

Progressive Disclosure

Don't overwhelm users!

```
# Simple mode by default
mode = st.radio("Mode", ["Simple", "Advanced"])

if mode == "Simple":
    # Just the basics
    text = st.text_input("Enter movie title")
else:
    # All the options
    text = st.text_area("Enter movie description")
    genre = st.selectbox("Genre", genres)
    year = st.number_input("Year", 1900, 2025)
    budget = st.slider("Budget", 0, 500)
```

Start simple, reveal complexity when needed.

Error Handling

Don't crash on bad input!

```
try:
    result = model.predict(user_input)
    st.success(f"Prediction: {result}")
except ValueError as e:
    st.error(f"Invalid input: {e}")
    st.info("Try entering a different value")
except Exception as e:
    st.error("Something went wrong. Please try again.")
```

Always have a fallback!

Handling Long Operations

Users need to know something is happening:

```
# For unknown duration
with st.spinner("Analyzing your movie..."):
    result = model.predict(features)

# For known steps
st.write("Processing...")
progress = st.progress(0)
for i, step in enumerate(steps):
    process(step)
    progress.progress((i + 1) / len(steps))

st.success("Complete!")
```

Collecting Feedback

Learn from your users:

```
# After showing prediction
st.write(f"Prediction: {result}")

col1, col2 = st.columns(2)
with col1:
    if st.button("
👍
 Correct"):
        log_feedback(result, "positive")
        st.success("Thanks for the feedback!")
with col2:
    if st.button("
👎
 Wrong"):
        log_feedback(result, "negative")
        st.info("We'll improve!")
```

This data can improve your model!

Streaming for LLMs

Don't make users wait for entire response:

```
import streamlit as st

# Create a placeholder
output = st.empty()

# Stream tokens as they arrive
full_response = ""
for token in llm.stream(prompt):
    full_response += token
    output.write(full_response)
```

Users can start reading immediately!

Complete Netflix Demo Example

```
import streamlit as st
import pandas as pd
import pickle

st.title("
🎬
 Netflix Movie Success Predictor")

# Sidebar for model info
st.sidebar.header("About")
st.sidebar.write("This model predicts movie success based on features.")

# Main inputs
st.header("Movie Details")
col1, col2 = st.columns(2)

with col1:
    genre = st.selectbox("Genre", ["Action", "Comedy", "Drama", "Horror"])
    budget = st.slider("Budget ($M)", 1, 300, 50)

with col2:
    runtime = st.slider("Runtime (min)", 60, 240, 120)
    is_sequel = st.checkbox("Sequel?")

# Predict button
if st.button("
🎯
 Predict"):
    with st.spinner("Analyzing..."):
        # Make prediction (replace with your model)
        prediction = "Success" if budget > 100 else "Risky"

    st.header("Result")
    if prediction == "Success":
        st.success("
🎉
 This movie looks promising!")
    else:
        st.warning("
⚠️
 This movie might be risky.")
```

Key Takeaways

1. **Streamlit** turns Python scripts into web apps
   - Write Python, get web pages
   - Perfect for dashboards and demos
2. **Gradio** is even simpler for model demos
   - Define inputs, outputs, function
   - Great for quick prototypes
3. **Deploy to Hugging Face Spaces** for free
   - Just push your code
   - Get a shareable URL
4. **Good UX matters**
   - Progressive disclosure
   - Error handling
   - Loading feedback

Lab Preview

This week's hands-on:

1. Build a Streamlit app for your Netflix model
2. Add user inputs (sliders, dropdowns)
3. Display predictions with nice formatting
4. Deploy to Hugging Face Spaces
5. (Bonus) Build a Gradio version

By the end: Your own live ML demo with a public URL!

Questions?

Today's key concepts:

- Streamlit and Gradio
- Caching and performance
- Deployment to Hugging Face
- UX best practices

Remember: A working demo is worth 1000 notebooks!