

Interactive AI Demos

CS 203: Software Tools and Techniques for AI

Prof. Nipun Batra, IIT Gandhinagar

Why Build Demos?

"If it's not a demo, it doesn't exist."

1. **Communication:** Show stakeholders/users what the model *actually* does.
2. **Debugging:** Interactive exploration reveals edge cases static metrics miss.
3. **Data Collection:** Users interacting with the model generate new training data.
4. **Portfolio:** A live link is worth 1000 GitHub stars.

The "Old" Way:

- Build React/Vue frontend
- Build Flask/FastAPI backend
- Connect them
- Spend 2 weeks on CSS

The "New" Way:

- Streamlit / Gradio

Streamlit: The Data App Framework

Philosophy: "Scripting" data apps. Run from top to bottom on every interaction.

```
pip install streamlit
streamlit run app.py
```

Hello World:

```
import streamlit as st
import pandas as pd

st.title("My First AI App")

name = st.text_input("What is your name?")
if name:
    st.write(f"Hello, {name}!")

data = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6]})
st.line_chart(data)
```

Streamlit: Widgets & Layout

Input Widgets:

```
age = st.slider("Age", 0, 100, 25)
role = st.selectbox("Role", ["Student", "Teacher", "Engineer"])
text = st.text_area("Enter prompt", height=200)
image = st.file_uploader("Upload Image", type=['png', 'jpg'])
```

Layout:

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

with col1:
    st.header("Input")
    prompt = st.text_input("Prompt")

with col2:
    st.header("Output")
    if st.button("Generate"):
        st.write(f"Generated: {prompt}")
```

Streamlit: Sidebar & State

Sidebar:

```
st.sidebar.title("Configuration")
model_name = st.sidebar.selectbox("Model", ["GPT-4", "Llama-3"])
temp = st.sidebar.slider("Temperature", 0.0, 1.0, 0.7)
```

Session State:

Streamlit reruns the whole script on interaction. Variables reset unless stored in

`st.session_state`.

```
if 'counter' not in st.session_state:
    st.session_state.counter = 0

if st.button("Increment"):
    st.session_state.counter += 1

st.write(f"Count: {st.session_state.counter}")
```

Streamlit: Caching

Problem: Re-running expensive functions (loading models) on every click.

Solution: `@st.cache_resource` (for models/DB) or `@st.cache_data` (for dataframes).

```
@st.cache_resource
def load_model():
    print("Loading model ... (this happens once)")
    return LargeModel()

@st.cache_data
def process_data(df):
    print("Processing ... (happens only if df changes)")
    return df.groupby('category').sum()

model = load_model() # Fast on subsequent runs
```

Gradio: UI for ML Models

Philosophy: Function-centric. Define a function, define inputs/outputs, get a UI.

```
pip install gradio
```

The Interface Class:

```
import gradio as gr

def reverse_text(text):
    return text[::-1]

demo = gr.Interface(
    fn=reverse_text,
    inputs="text",
    outputs="text",
    title="Text Reverser"
)

demo.launch()
```

Gradio: Multimodal Inputs

Image to Label:

```
def classify_image(img):  
    # img is a numpy array  
    prediction = model.predict(img)  
    return {label: float(conf) for label, conf in prediction.items()}  
  
demo = gr.Interface(  
    fn=classify_image,  
    inputs=gr.Image(shape=(224, 224)),  
    outputs=gr.Label(num_top_classes=3)  
)
```

Supported Inputs: Audio, Video, 3D Models, DataFrames, Files.

Gradio Blocks: Complex Layouts

For more control than `Interface` (similar to Streamlit layout):

```
with gr.Blocks() as demo:
    gr.Markdown("# Chat with Documents")

    with gr.Row():
        with gr.Column():
            file_input = gr.File(label="Upload PDF")
            api_key = gr.Textbox(label="API Key", type="password")

        with gr.Column():
            chatbot = gr.Chatbot()
            msg = gr.Textbox(label="Message")
            clear = gr.Button("Clear")

    def respond(message, chat_history):
        # logic here
        chat_history.append((message, "Response"))
        return "", chat_history

    msg.submit(respond, [msg, chatbot], [msg, chatbot])
```

Understanding Reactive Programming

Traditional Web Apps:

```
User clicks → Event listener fires → Update specific DOM element
```

Streamlit/Gradio Paradigm:

```
User interacts → Re-run entire script → Framework diffs and updates UI
```

Why this matters:

- No manual DOM manipulation
- State persists via `st.session_state` (Streamlit) or component state (Gradio)
- Pure Python - no JavaScript required

The Streamlit Execution Model

On every interaction (button click, slider change, text input):

1. Script runs from top to bottom
2. Variables reset (unless in `st.session_state`)
3. Framework computes diff between old and new UI
4. Only changed elements re-render in browser

Example:

```
import streamlit as st

counter = 0 # ✗ Always 0 on rerun

if st.button("Increment"):
    counter += 1 # ✗ Won't persist

st.write(counter) # Always shows 0
```

State Management Patterns

Pattern 1: Initialize on First Run

```
if 'model' not in st.session_state:  
    st.session_state.model = load_expensive_model()
```

Pattern 2: Form State

```
with st.form("my_form"):  
    name = st.text_input("Name")  
    age = st.slider("Age", 0, 100)  
    submitted = st.form_submit_button("Submit")  
  
if submitted:  
    st.session_state.user_data = {"name": name, "age": age}
```

Pattern 3: Callback Functions

```
def on_slider_change():  
    st.session_state.slider_moved = True
```

Handling Long-Running Tasks

Problem: Model inference takes 5 seconds. UI freezes.

Solution 1: Spinners

```
with st.spinner("Generating ... "):  
    result = slow_model(input_text)  
st.success("Done!")  
st.write(result)
```

Solution 2: Progress Bars

```
progress_bar = st.progress(0)  
for i in range(100):  
    process_chunk(i)  
    progress_bar.progress(i + 1)
```

Solution 3: Status Updates

```
with st.status("Downloading data ...") as status:
```

Error Handling Best Practices

Don't show stack traces to users:

```
# ❌ Bad
result = model.predict(user_input) # Crashes with ugly error

# ✅ Good
try:
    result = model.predict(user_input)
except Exception as e:
    st.error(f"Prediction failed: {str(e)}")
    st.stop() # Halt execution gracefully
```

Validate inputs:

```
uploaded_file = st.file_uploader("Upload Image", type=['png', 'jpg'])

if uploaded_file is not None:
    if uploaded_file.size > 5_000_000: # 5MB
        st.error("File too large. Max 5MB.")
    else:
```

Comparison: Streamlit vs Gradio

Feature	Streamlit	Gradio
Paradigm	Scripting (Top-down)	Functional / Event-driven
Best For	Data Dashboards, Multi-page apps	Model Demos, Hugging Face Spaces
Customization	Moderate (CSS hacks)	Moderate (Themes)
State	Session State (Explicit)	State component (Implicit)
Hosting	Streamlit Cloud	Hugging Face Spaces
Execution Model	Full script rerun	Function-level reruns

Rule of Thumb:

- Need a full dashboard/app? **Streamlit**
- Need to quickly show off a model function? **Gradio**

Deploying Demos

Hugging Face Spaces:

- Free hosting for Gradio/Streamlit apps
- git-based workflow
- Great for portfolio







```
git clone https://huggingface.co/spaces/user/my-space
cd my-space
# Add app.py and requirements.txt
git add .
git commit -m "Init"
git push
```

Streamlit Cloud:

- Connects to GitHub repo
- Automatic redeploy on push

Deployment Checklist

Before Deployment:

1.  Test locally with `streamlit run app.py`
2.  Create `requirements.txt` with pinned versions
3.  Add `.gitignore` for large files/secrets
4.  Use environment variables for API keys (not hardcoded!)
5.  Add loading states and error handling
6.  Test with slow internet (users have varying speeds)

Environment Variables (Streamlit):

```
import os
api_key = os.getenv("GEMINI_API_KEY") # Set in Streamlit Cloud settings
```

Secrets (Streamlit Cloud):

Multi-Page Apps (Streamlit)

Structure:

```
my_app/
├── Home.py           # Main page (name matters!)
├── pages/
│   ├── 1_📊_Data.py
│   ├── 2_🤖_Model.py
│   └── 3_📈_Results.py
└── requirements.txt
```

Navigation is automatic! Streamlit detects files in `pages/`.

Sharing state across pages:

```
# In Home.py
st.session_state.user_name = "Alice"

# In pages/1_📊_Data.py
st.write(f"Welcome, {st.session_state.user_name}")
```

Building a Chatbot (Streamlit)

```
import streamlit as st
from langchain.chat_models import ChatOpenAI

st.title("GPT-4 Wrapper")

if "messages" not in st.session_state:
    st.session_state.messages = []

# Display history
for msg in st.session_state.messages:
    with st.chat_message(msg["role"]):
        st.markdown(msg["content"])

# User input
if prompt := st.chat_input("Say something ... "):
    with st.chat_message("user"):
        st.markdown(prompt)
    st.session_state.messages.append({"role": "user", "content": prompt})

# AI response
with st.chat_message("assistant"):
    response = "This is a dummy response."
    st.markdown(response)
st.session_state.messages.append({"role": "assistant", "content": response})
```

Best Practices for Demos

1. Handling Long-Running Tasks:

- Use spinners: `with st.spinner('Generating ... '):`
- Show progress bars.

2. Error Handling:

- Don't show stack traces to users.
- `try: ... except: st.error("Something went wrong")`

3. Examples:

- Provide "Clickable Examples" so users don't have to type/upload.
- Gradio: `examples=[["cat.jpg"], ["dog.jpg"]]`

4. Instructions:

- Clear markdown explanation of what the model does and its limitations.

Lab Preview: Multiple Apps

You will build **THREE** apps to master Streamlit:

1. Sentiment Analysis Dashboard

- Text input → real-time sentiment prediction
- Confidence visualization
- History tracking with session state

2. Image Classification App

- Upload images → classification results
- Multiple model selection
- Batch processing

3. YouTube Video Summarizer

- URL input → transcript extraction → AI summary
- Chat interface for follow-up questions

Resources

- **Streamlit Gallery:** streamlit.io/gallery
- **Gradio Guides:** gradio.app/guides
- **Hugging Face Spaces:** huggingface.co/spaces
- **LangChain + Streamlit:**
python.langchain.com/docs/integrations/providers/streamlit

Questions?
