Interactive AI Demos & UI/UX

CS 203: Software Tools and Techniques for AI

Prof. Nipun Batra, IIT Gandhinagar

The "Demo" Gap

The Problem: You trained a great model. It lives in a Jupyter Notebook.
The Goal: Let stakeholders (non-coders) use it.

Options:

1. **Full Stack**: React + FastAPI + Docker + K8s. (Weeks of work).
2. **Low Code**: Streamlit / Gradio. (Hours of work).

Why Demos Matter:

- **Feedback**: Early user testing reveals edge cases.
- **Data Flywheel**: Users interacting = New training data.
- **Portability**: A URL is worth 1,000 GitHub stars.

Streamlit Architecture: The "Script" Model

Streamlit works differently from standard web apps.

Execution Flow:

1. User loads page -> Entire Python script runs top-to-bottom.
2. User clicks button -> **Entire script runs again from top!**
3. Frontend detects changes (diff) and updates DOM via WebSockets.

Implication:

- Variables are **reset** on every interaction.
- Need explicit **Session State** to remember things.
- Need **Caching** to prevent re-loading models.

Managing State

Without Session State:

```
count = 0
if st.button("Click"):
    count += 1
st.write(count) # Always 0 or 1. Resets every time!
```

With Session State:

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

if st.button("Click"):
    st.session_state.count += 1

st.write(st.session_state.count) # Persists!
```

Caching Strategies: Performance is UX

Re-loading a 5GB model on every button click = Unusable App.

1. `@st.cache_resource`: For Global Objects (Models, DB Connections).
    - Loaded once, shared across all users.
    - *Singleton pattern*.
2. `@st.cache_data`: For Computations (DataFrames, API calls).
    - Cached based on function inputs.
    - *Memoization pattern*.

```
@st.cache_resource
def load_llm():
    return AutoModelForCausalLM.from_pretrained("gpt2") # Run once

@st.cache_data
```

UX Patterns for GenAI

1. **Streaming**: Don't make users wait 10s for full text.
   - Show tokens as they arrive.
   - **Streamlit**: `st.write_stream(generator)`.
2. **Latency Masking**:
   - Use spinners (`st.spinner`).
   - Show intermediate steps ("Parsing PDF...", "Embedding...").
3. **Feedback Loops**:
   - Add
     👍
     /
     👎

Gradio: The Functional Approach

Gradio is optimized for "Input -> Model -> Output" workflows.

```
import gradio as gr

def predict(img):
    # Process image
    return "Cat"

# Auto-generates UI based on types
demo = gr.Interface(fn=predict, inputs="image", outputs="label")
demo.launch()
```

Streamlit vs Gradio:

- **Streamlit**: Full Data Dashboards, Multi-page apps, Custom Logic.
- **Gradio**: Quick Model APIs, Hugging Face Spaces integration.

Deployment

Hugging Face Spaces:

- Free hosting for ML demos.
- Git-based deployment.
- Supports Streamlit, Gradio, Docker.

Workflow:

1. Create `app.py`.
2. Create `requirements.txt`.
3. `git push` to HF Space.
4. Done.

Summary

1. **Architecture**: Streamlit reruns scripts; manage state carefully.
2. **Performance**: Cache resources (models) and data aggressively.
3. **UX**: Stream text, handle errors gracefully, provide feedback mechanisms.
4. **Deployment**: Use HF Spaces for zero-config hosting.

Lab: Build a Chat-with-PDF app using Streamlit, Session State, and Streaming.