Streamlit Interface: Design Decisions & User Flow

Why Streamlit?

Streamlit is an open-source Python library that makes it easy to build and deploy interactive web applications—especially for machine learning and data science projects—without requiring deep knowledge of front-end development.

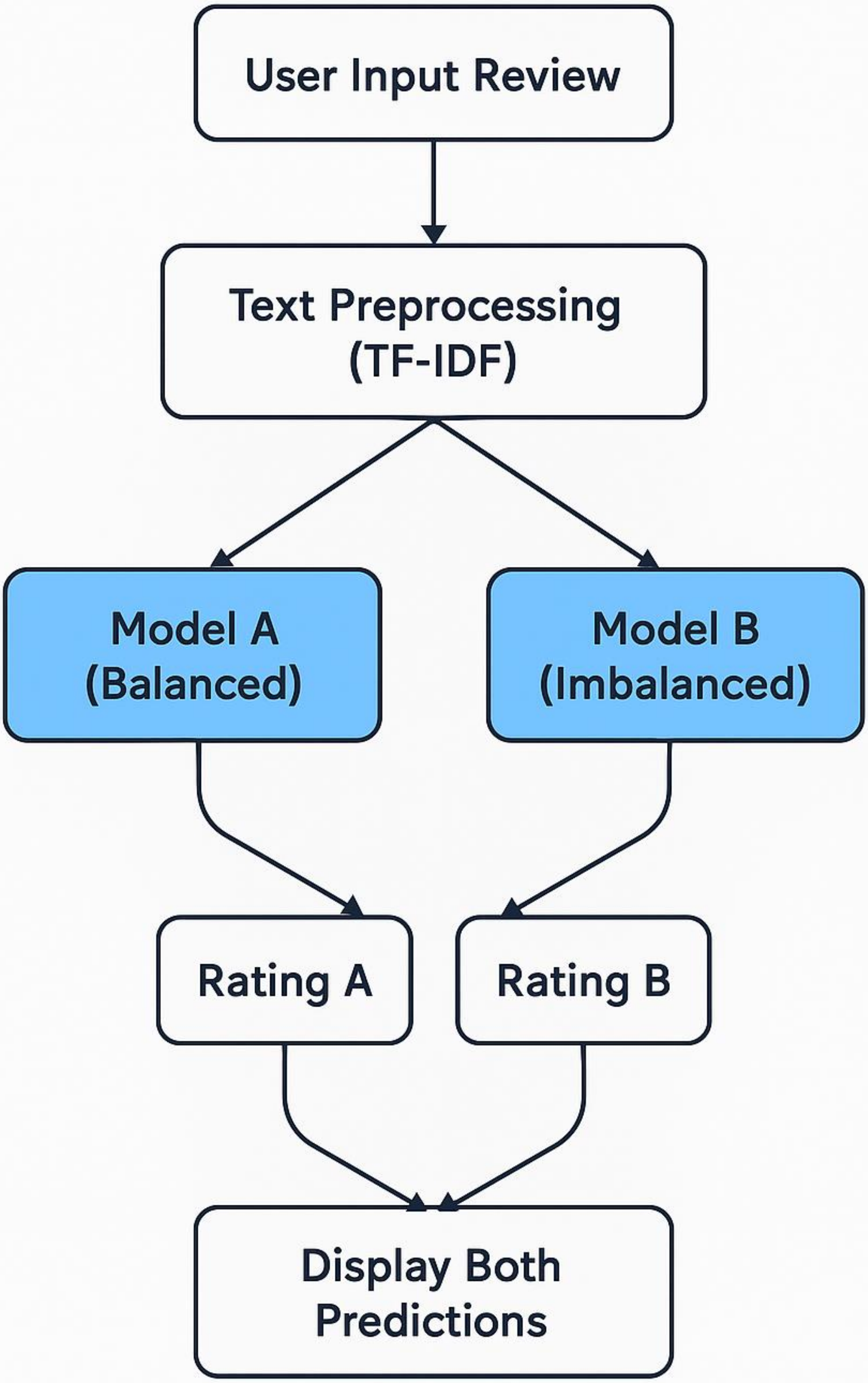
- Easy integration with Python models (`pickle`, `sklearn`, `pandas`, etc.)
- Instant feedback with interactive widgets like buttons and text inputs
- Clean UI out of the box—ideal for rapid prototyping
- Local and cloud deployment friendly (`streamlit share`, Heroku, etc.)

UI Design Decisions

Element	Decision
Page Title	`st.title(" Review Rating Predictor")` — communicates the purpose immediately.
Review Input	`st.text_area()` used to accept multi-line user input.
Predict Button	`st.button()` placed directly below input to ensure natural flow.
Feedback Handling	`st.warning()` used when no input is provided.
Results Display	- `st.success()` for Model A (Balanced) - `st.info()` for Model B (Imbalanced)

User Flow

1. **App Launch**
   The user opens the app and sees a welcoming title, " Review Rating Predictor".
2. **Review Entry**
   A multi-line text box allows the user to enter a product review.
3. **Button Click: "Predict Ratings"**
   - If no review is entered, the app prompts: *Please enter some text*
   - If review is entered:
     - Input is transformed using **TF-IDF vectorizer** for both models.
     - Models **Model_A** (trained on balanced data) and **Model_B** (trained on imbalanced data) predict review ratings.
4. **Predictions Displayed**
   - Model A's result appears using a green success message: *Balanced Model Prediction*
   - Model B's result appears using a blue info message: *Imbalanced Model Prediction*


User Input Review
Text Preprocessing (TF-IDF)
Model A (Balanced)
Model B (Imbalanced)
Rating A
Rating B
Display Both Predictions

Detailed Explanation (Step-by-Step)

Step	Action
1. User Input	The user types a review in the text area.
2. Preprocessing	The review is transformed separately using two TF-IDF vectorizers — one for each model.
3. Model A Prediction	The review (transformed) is passed to Model A, trained on **balanced data**, and outputs a predicted rating.
4. Model B Prediction	Simultaneously, the transformed review is passed to Model B, trained on **imbalanced data**, and outputs another rating.
5. Output	The two ratings are displayed clearly — but **not compared, ranked, or analyzed** — just shown side-by-side for user awareness.

Additional Notes

- **Caching with `@st.cache_resource`**:
  The model and vectorizer are cached to avoid reloading on every interaction, ensuring faster predictions.
- **Folder Structure Flexibility**:
  Absolute or relative paths like `'Models/model_A.pkl'` ensure modularity and easier portability of the app.
- **Scalability**:
  The UI design is minimal and responsive, making it easy to expand—e.g., showing probability scores, adding charts, model confidence, etc.

Code for the streamlit app.py

```
import streamlit as st
import pickle

# Load model and vectorizer
@st.cache_resource
def load_model_and_vectorizer(path_model, path_vectorizer):
    model = pickle.load(open(path_model, 'rb'))
    vectorizer = pickle.load(open(path_vectorizer, 'rb'))
    return model, vectorizer
# Absolute or relative paths to your files
model_A, tfidf_A = load_model_and_vectorizer(
    'Models/model_A.pkl', 'Models/TfidfVectorizer_A.pkl')
model_B, tfidf_B = load_model_and_vectorizer(
    'Models/Model_B.pkl', 'Models/TfidfVectorizer_b.pkl')
```

```
# Streamlit UI
st.title("Review Rating Predictor")

user_input = st.text_area(" Enter your product review here:")

if st.button("Predict Ratings"):
    if user_input.strip() == "":
        st.warning("Please enter some text.")
    else:
        X_input_A = tfidf_A.transform([user_input])
        X_input_B = tfidf_B.transform([user_input])

        pred_A = model_A.predict(X_input_A)[0]
        pred_B = model_B.predict(X_input_B)[0]

        st.success(f"Model A (Balanced) Prediction: ⭐ {pred_A}")
        st.info(f"Model B (Imbalanced) Prediction: ⭐ {pred_B}")
```

What is Deep Learning?

Deep Learning is a subset of **machine learning** that uses algorithms called **artificial neural networks** inspired by the structure and function of the human brain. These models learn patterns from large amounts of data by automatically extracting high-level features, often requiring minimal manual intervention or feature engineering.

Key Characteristics of Deep Learning:

Feature	Description
Hierarchical Learning	Learns features layer by layer—from low-level (edges, shapes) to high-level (objects, semantics).
End-to-End Training	Raw data is directly mapped to output without needing handcrafted features.
Large Data Requirement	Performs best with large datasets and high computational power (e.g., GPUs).
Autonomous Feature Extraction	Automatically identifies the most relevant features from input data.
Scalability	Can handle very complex and high-dimensional data like text, images, audio, and video.

Deep Learning Models for Text Classification:

1. Recurrent Neural Network (RNN)

Summary:

- Designed to handle **sequential data**.
- Processes text one token at a time, **maintaining memory** of past tokens via hidden states.

How It Works:

At each time step:

```
h_t = tanh(W_x * x_t + W_h * h_{t-1} + b)
```

Where:

- `x_t` = current word vector
- `h_t` = hidden state at time t
- `W_x`, `W_h` = learnable weights

Pros:

- Simple and effective for **short sequences**.
- Easy to implement.

Cons:

- Suffers from **vanishing gradient** → hard to remember long-term context.
- Can't parallelize well (slower training).

Example Use:

Classifying short tweets or product tags.

2. Long Short-Term Memory (LSTM)

Summary:

- A special kind of RNN with **memory cells** and **gates**.
- Designed to **remember long-term dependencies**.

How It Works:

Each unit contains:

- **Forget gate** `f_t`: Decides what to discard.

- **Input gate** `i_t`: Decides what to store.
- **Output gate** `o_t`: Decides what to output.

Memory is updated using:

```
c_t = f_t * c_{t-1} + i_t * ĉ_t
h_t = o_t * tanh(c_t)
```

Pros:

- Works well for **long reviews**.
- Avoids vanishing gradients.
- Captures context and sequence dynamics.

Cons:

- **Slower training** due to complexity.
- More parameters → requires more data.

Example Use:

Amazon product review classification, speech-to-text models.

3. Gated Recurrent Unit (GRU)

Summary:

- A simplified version of LSTM.
- Combines forget + input gates into a **single "update gate"**.

Equations:

```
z_t = sigmoid(W_z * [h_{t-1}, x_t])        (update gate)
r_t = sigmoid(W_r * [h_{t-1}, x_t])        (reset gate)
h_t = (1 - z_t) * h_{t-1} + z_t * h̃_t      (final state)
```

Pros:

- **Faster than LSTM** with comparable performance.
- Fewer parameters → good for smaller datasets.

Cons:

- May underperform on very complex text.

Example Use:

Real-time text prediction where speed is important.

4. Bidirectional LSTM (BiLSTM)

Summary:

- Processes text in **both directions**: forward and backward.
- Gives context from **before and after** each word.

How It Works:

For a sequence:

- Forward LSTM reads left to right.
- Backward LSTM reads right to left.
- Final output is the **concatenation** of both directions.

Pros:

- Better **understanding of full context**.
- Improves accuracy for **complex sentence structures**.

Cons:

- Double the parameters → more compute.
- Slower than unidirectional LSTM.

Example Use:

Intent classification, question answering, chatbot intent understanding.

5. Transformers (e.g., BERT, RoBERTa)

What It Is:

Transformers are the current **state-of-the-art** in NLP. They use **self-attention** to understand relationships between all words **simultaneously**, not sequentially.

Uses:

- **Multi-head self-attention**
- **Positional encoding**
- **Layer normalization and feedforward layers**

Self-attention core formula:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Where Q, K, V = query, key, value matrices derived from embeddings.

Pros:

- Understands **global context**.
- Pretrained models like **BERT** can be fine-tuned with little data.
- Extremely high accuracy.

Cons:

- **Heavy** (large size, requires GPU).
- Slower inference unless distilled or optimized.

Example Use:

- Review rating prediction
- Sentiment analysis
- Summarization
- Named entity recognition

Summary Table (Enhanced)

Model	Strengths	Weaknesses	Best Use Case
RNN	Simple, learns sequences	Short memory	Short sentences, simple tasks
LSTM	Long-term memory, stable	Slower training	Long reviews, moderate complexity
GRU	Faster, less complex than LSTM	Slightly less expressive	Faster deployment, small data
BiLSTM	Full sentence context	Slower, more memory	Sentence-level understanding
BERT	State-of-the-art accuracy	Requires compute	All NLP tasks with high performance needs