# AI In Healthcare, Homework 5: Self-Learning Tutorial

**Evan Jones**, UT ID: `ej8387`

# 1: Project Overview & Motivation

This guide is in five parts:

# Side Effects Can Ruin Your Life

In 2002, my friend David MacLean woke up in a train station in Hydrabad, India.

He didn't know why he was there.

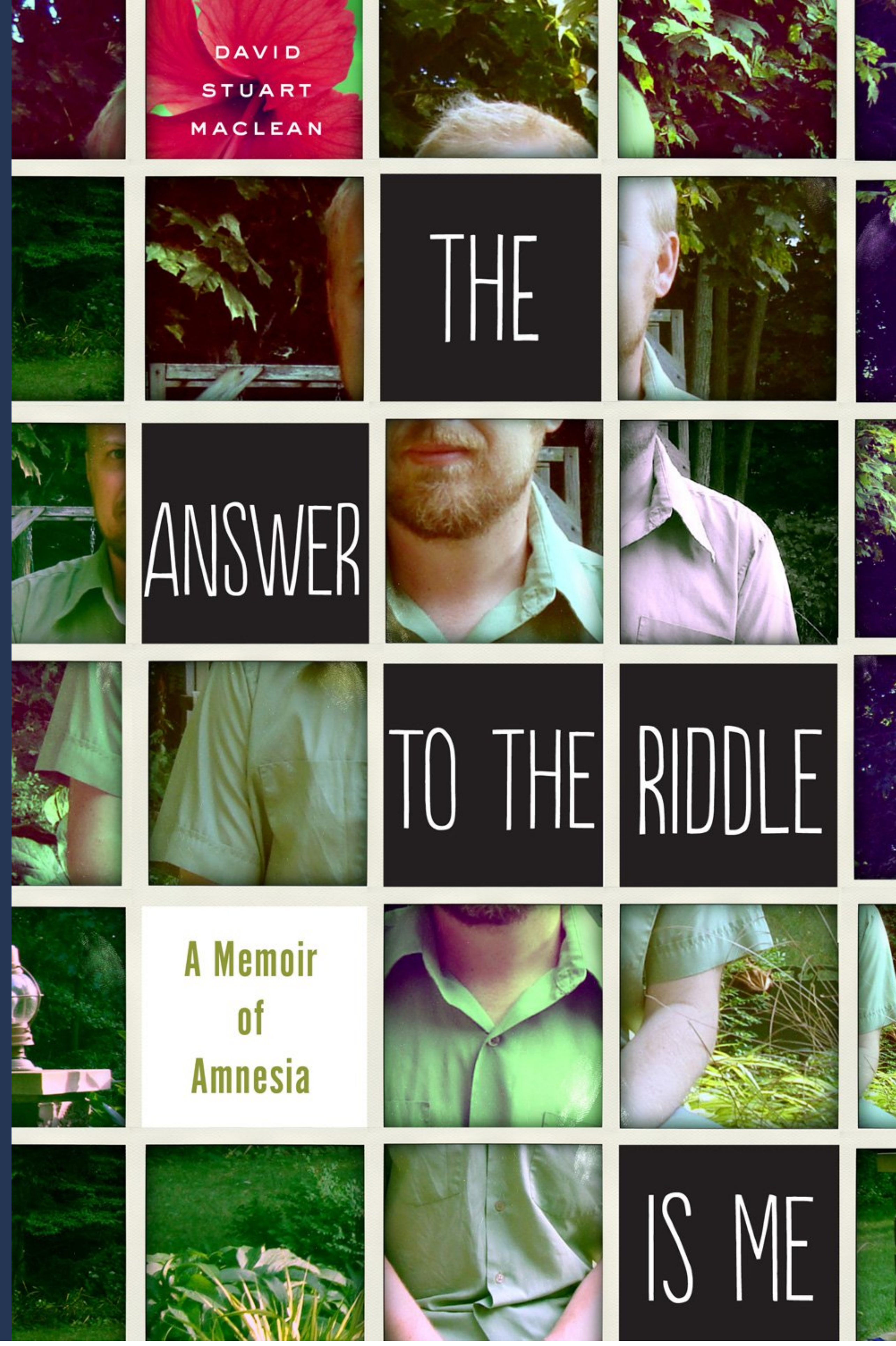
He didn't know his own name.

His entire past had disappeared.

## Loss and recovery

A kind policeman helped him find his home. He discovered that he was American, on a Fulbright scholarship to study in India. And he discovered that he'd been taking Lariam (Mefloquine), an antimalarial medication recommended by the U.S. Government.

Eventually, he discovered that Lariam was linked to murders, suicides, and amnesia like his own. He told his story on the national radio program, This American Life, and wrote a book, The Answer to the Riddle is Me. He slowly pieced his life together.




# David's This American Life segment won awards. It's really worth listening to!

# Identifying Long-tail Side Effects with LLMs

Side effects like David's are rare.
But when many people take a medication, even "rare" side effects happen quite often.

Small text warnings on a drug label can't prepare you for the impact a medicine might have on your life.

This project uses Large Language Models (LLMs) to summarize patient reviews for psychoactive medications and highlight outlier experiences, good or bad.

# 2: Project Setup

- Install UV

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

- Clone project

```
git clone https://github.com/etjones/aihc_hw5_self_learning_tutorial_ej8387.git
```

- Request an API Key for your LLM
- [Optional] Configure `llm` for alternate models
- Run program

```
export OPENAI_API_KEY=<YOUR_OPENAI_API_KEY>; uv run summarize_webmd_drug_reviews.py
```

Astral's uv package manager is the best thing to happen to Python packaging in years. Normally we'd have to install Python, set up a virtual environment, install dependencies, and activate the environment to use this code. Among many
other things, uv just handles this.

# Request an API Key for your LLM

DeepSeek, OpenAI ChatGPT, and Claude are capable, affordable, LLMs. You may need to set up accounts and credit cards for billing first.

DeepSeek API Keys Page
OpenAI API Keys Page
Anthropic API Keys Page

## LLM API Costs, 2025-03-15

| LLM | Input cost, 1M tokens | Output cost, 1M tokens | Max Context Window, tokens |
|---|---|---|---|
| DeepSeek-Chat | $0.07 | $1.10 | 64K |
| OpenAI GPT-4o Mini | $0.15 | $0.60 | 128K |
| Anthropic Haiku 3.5 | $0.80 | $4.00 | 200K |

The code in this repository defaults to OpenAI `gpt-4o-mini`, so you'd need to change the model_name in the `main()` function and configure API keys for other models, but these are minimal changes. Knock yourself out, but for simplicity I've just covered the OpenAI model in this tutorial.

# [Optional] Configure `llm` for alternate models

`llm` has several mechanisms to manage API keys. You can set and forget by running:

```
# (Installs an llm plugin to use Anthropic models)
llm install llm-anthropic
# (Prompts for your API key and stores it persistently)
llm set keys anthropic
```

or

```
llm install llm-deepseek
llm set keys deepseek
```

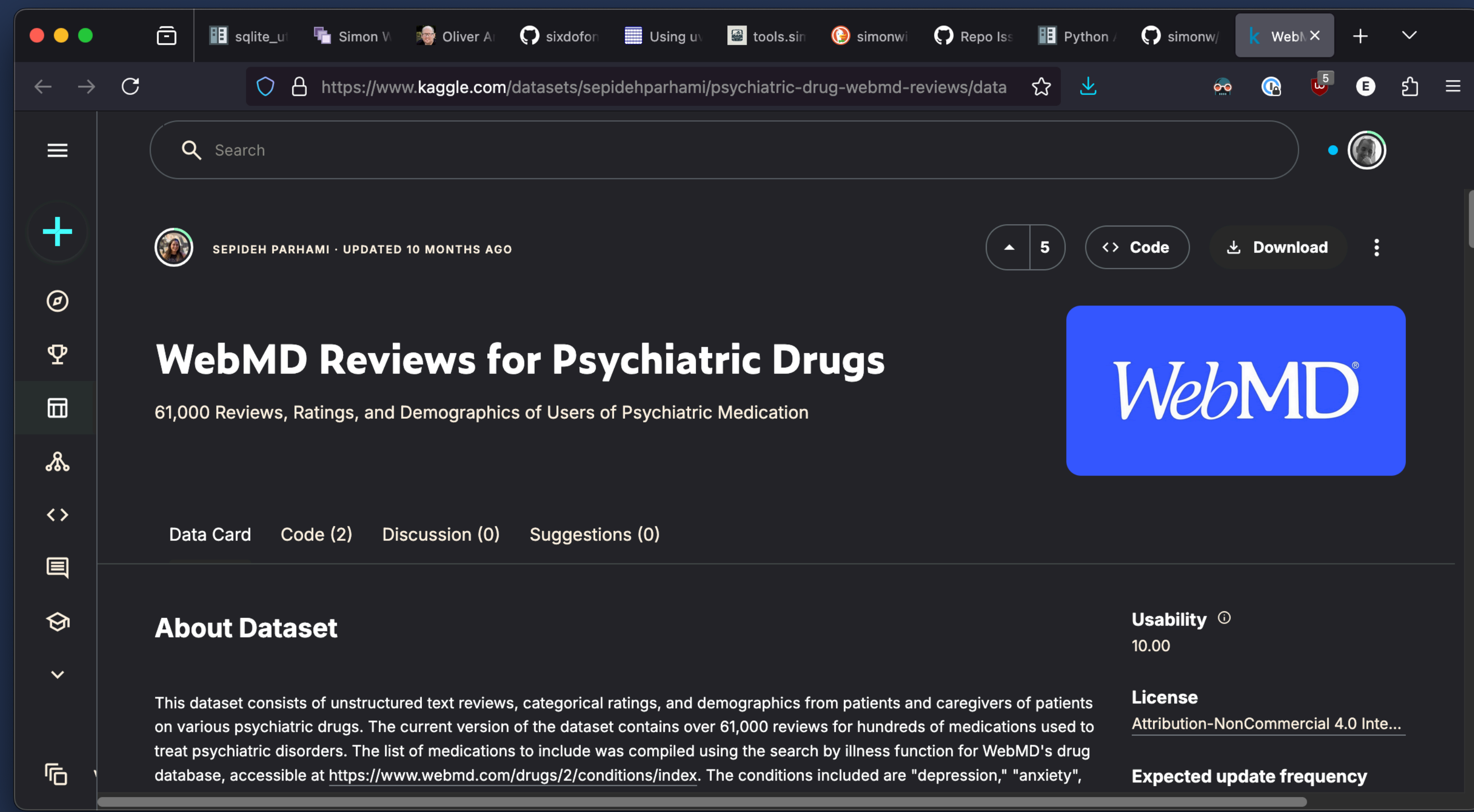
# 3. Dataset Ingestion & Database Creation

# Dataset Source

Shruti Narayanan's earlier project highlighted a dataset of 61,000 reviews of psychiatric drugs from WebMD.com

I've included the zipped dataset, but more information is available at the source:

https://www.kaggle.com/datasets/sepidehparhami/psychiatric-drug-webmd-reviews/data



# From Dataset To Database

Simon Willison's `sqlite_utils` makes using local SQLite databases a breeze.

pandas ingests the dataset, and `sqlite_utils` makes a working, persistent database in just a few lines

```python
def import_csv_to_sqlite_db(csv_path: str, table_name) -> Database:
    # Load/create a database file
    DB = sqlite_utils.Database("psych_med_reviews.sqlite")

    # Add to DB if it's not already there
    if table_name not in DB.table_names():
        df = pd.read_csv(csv_path)
        print(f"Dataset has {len(df)} rows, with columns: {list(df.columns)}")
        DB[table_name].insert_all(df.to_dict(orient="records"))

    print(f"Database table {table_name} has {DB[table_name].count} rows")
    return DB
```

I can't really say enough good things about `sqlite_utils`. You end up with the full power and reliability of a SQL database, but creating tables or making simple queries is really trivial.

Like the best software, it makes "Easy Things Easy and Hard Things Possible"

# 4. LLM Summarization

People are bad at reading text and noticing the important parts.

Fortunately, LLMs are really good at this!

(Much better than traditional NLP approaches-- and easier too)

It's hard to read through hundreds of thousands of reviews and pick out all the important parts without overlooking anything that could make a difference for someone.

Fortunately LLMs are really good at this. They're simpler to use than traditional NLP
approaches, and much better at noticing things we might care about.

For example, "Sentiment analysis" requires a certain amount of work to set up,
and at the end can mostly tell you whether a message is generally positive or negative.
LLMs will notice important pieces (like life-impacting side effects!) and surface
them for us.

# `llm` For The Win

Simon Willison's `llm` package simplifies interacting with LLMs from Python or the command line.

Basic use is as simple as:

```
import llm
model = llm.get_model() # defaults to OpenAI's gpt-4o-mini
response_text = model.prompt("Summarize these reviews: ...")
```

# The Crucial Prompt

LLMs are good at paying attention to things.

But we have to tell them what to prioritize, and what format we'd like returned.

Extra effort in specifying a prompt is usually rewarded.

```python
def summarization_prompt(drug_name: DrugName) -> str:
    review_prompt = f"""I'm appending a number of anonymized patient reviews for
    the  medication {drug_name} below. Please read the reviews and give me a
    short summary of  patients' feelings about the drug. You might list people's
    favorite good points, and least-liked elements about the experience. Pay
    attention to people who have  very strong positive or negative experiences,
    and include excerpts from those  reviews occasionally if they're very
    unusual. As an example, please don't include any excerpts about common mild
    symptoms, but if one user reports life-changing  headaches or amnesia,
    that's probably worth highlighting. Try to limit your summary to about two
    hundred words.

    Return a summary in Markdown format with sections ## Positive, ## Negative,
    ## Noteworthy (optional, include only for outlier experiences), and
    ## Conclusion

    Please don't introduce the summary, just begin right away.

    Reviews follow, and each is bracketed by <review> to start
    and </review> to end.


    """
    return review_prompt
```

# Plumbing and minding details

We submit reviews for 200+ drugs.

For some drugs, we have more reviews than fit into a model's context window.

And we want to cache responses to avoid repeated prompting.

```python
def fetch_llm_summaries(
    drug_info: dict[DrugName, list[DrugReview]],
    model_name: str,
    db: Database | None = None,
) -> dict[DrugName, DrugSummary]:
    """
    Fetch summaries for all drugs & reviews, respecting max context windows
    If db is supplied, fetch/store from the 'drug_summaries' table
    """
    llm_context_windows = {
        "deepseek-chat": 64_000,
        "gpt-4o-mini": 128_000,
        "anthropic/claude-3-5-haiku-latest": 200_000,
    }
    review_summaries: dict[DrugName, DrugSummary] = {}

    table_name = "drug_summaries"

    # Because words don't map perfectly to tokens, take a conservative
    # estimate of context window in terms of words
    est_max_words = llm_context_windows[model_name] * 0.6
    for drug, notes in drug_info.items():
        # Use a cached response for this drug/model if it's there
        summary = fetch_stored_summary(table_name, drug, model_name, db)

        # If there's no cached response, ask the LLM for a summary,
        # breaking into several prompts if needed
        if not summary:
            summary = submit_drug_review_and_join(
                model_name, drug, notes, est_max_words
            )
            # If we just fetched a summary, store it (No-op if no DB supplied)
            store_summary(table_name, drug, model_name, summary, db)

        review_summaries[drug] = summary

    return review_summaries
```

There's a certain amount of complexity in making robust requests, but this function is the top-level overview of how we do our fetching.

We fetch summaries drug by drug. We cache results when we get them, so that we don't have to make the same requests again and again while debugging or editing.

And for drugs that have so many reviews that they won't fit in a model's context window, we break up the reviews into appropriate-sized groups, and then ask the LLM to combine the final summaries.

See `submit_drug_review_and_join()` for the details there.

# 5. Final Presentation

Once we've created all our LLM summaries, we're left with a dictionary of

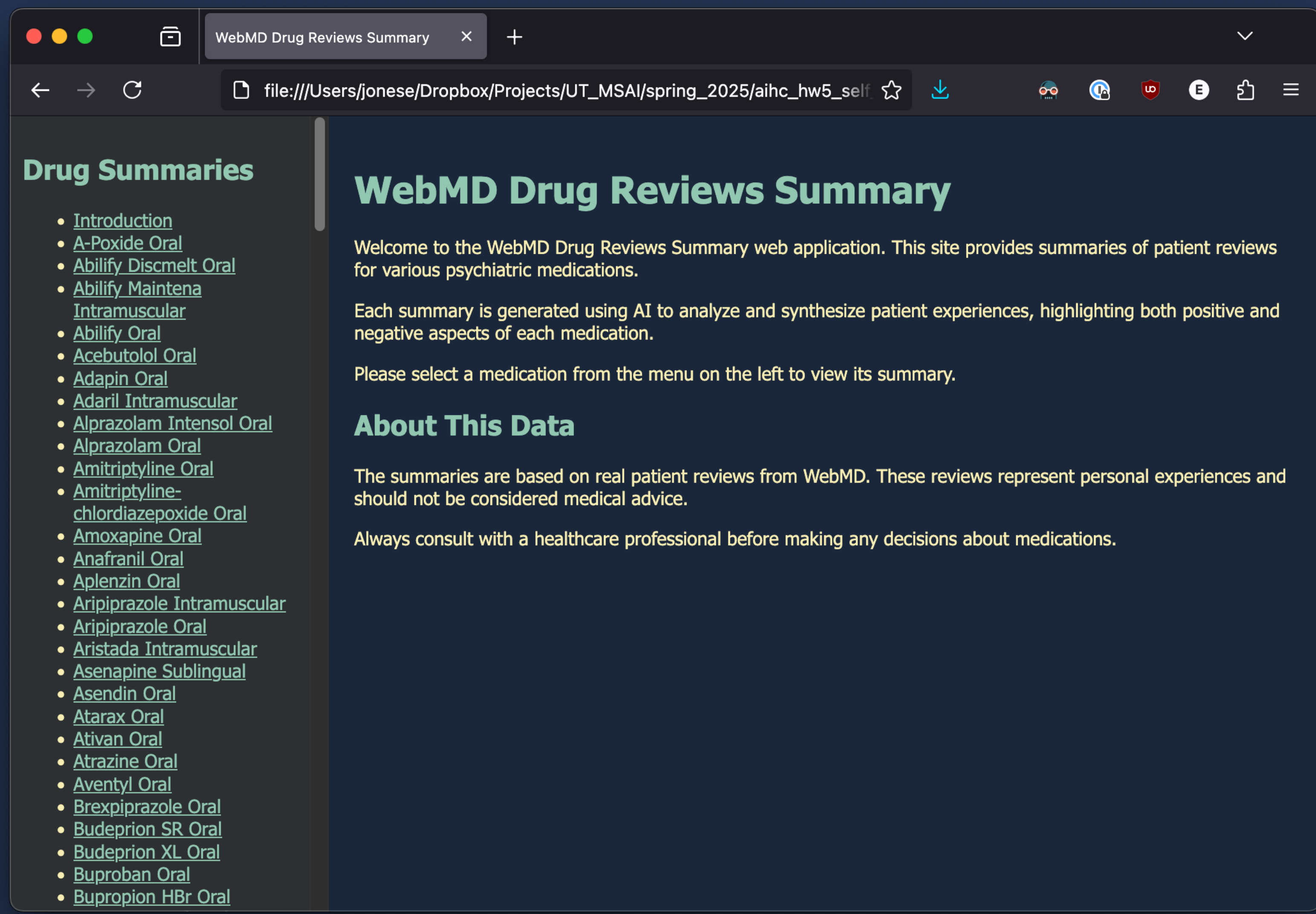
`{drug_name: drug_summary_markdown}` pairs.

Let's improve the presentation!

# Create a minimal Web app

I'm not much of a web programmer, but Claude is. I used the following prompt to make a function that makes a passable-looking, vanilla HTML web app:

```
Make a function that accepts a dict of {drug_name: markdown_summary} pairs and
writes a directory called summary_webapp that contains a plain vanilla html page
for each drug_name, renders the markdown_summary (which is in markdown) to HTML.
There should also be an index.html page that contains an introduction, and all
pages should have a menu column on the left side linking to each drug_name page.
Let's not duplicate the menu on each page-- we should be able to use one menu
list throughout the entire app. We should also include CSS that applies to the
entire web app and defaults to sans-serif yellow-ish text on a darker blue
background-- but make sure the color palettes are appealing and use web standards
```




The result was about 200 lines of code, `create_summary_webapp()` and worked on the first try.

# Run It Yourself

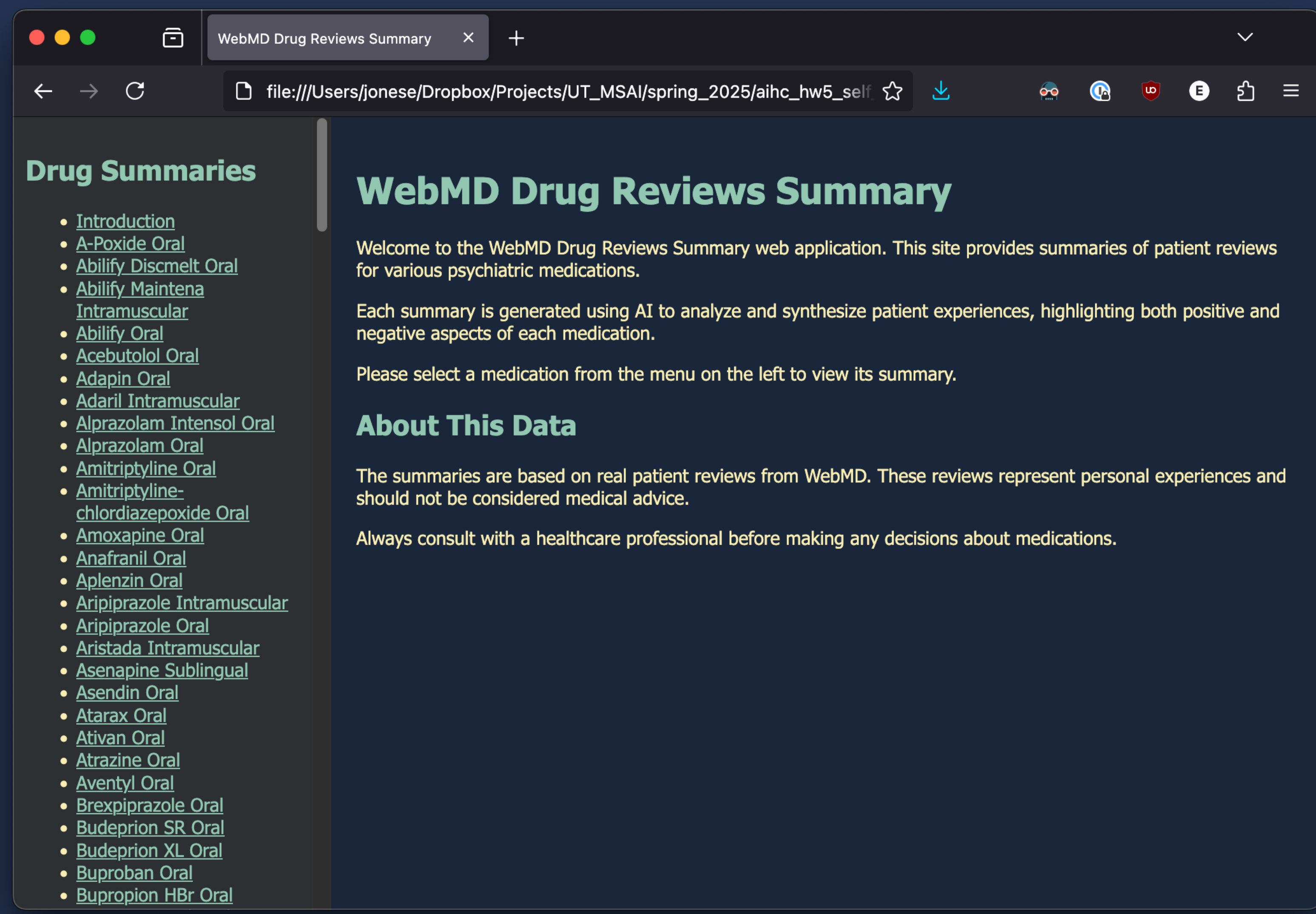
If you've cloned the repository, installed uv, and set up an LLM account, you're ready to go.

Run this program to summarize our dataset reviews:

```
export OPENAI_API_KEY=<YOUR_OPENAI_API_KEY>; uv run ./summarize_webmd_drug_reviews.py
```

View the web app at `summary_webapp/index.html`:

```
# (MacOS)
open summary_webapp.index.html
```




Note that if you use another LLM model, you'd need to change the API Key to an appropriate value, like `ANTHROPIC_API_KEY` or `DEEPSEEK_API_KEY`. (And change the `model_name` variable in the `main()` function)