Building a Personal Content Recommendation System, Part Two: Data Processing and Cleaning

2025-03-26 · 5 min · Saeed Esmaili

In part one of this blog series , I explored the motivation behind developing a personal recommendation system. The main goals are to learn how recommendation systems work and to build a tool that helps me find interesting blog posts and articles from feeds where only 1 in 20 posts might match my content interests.

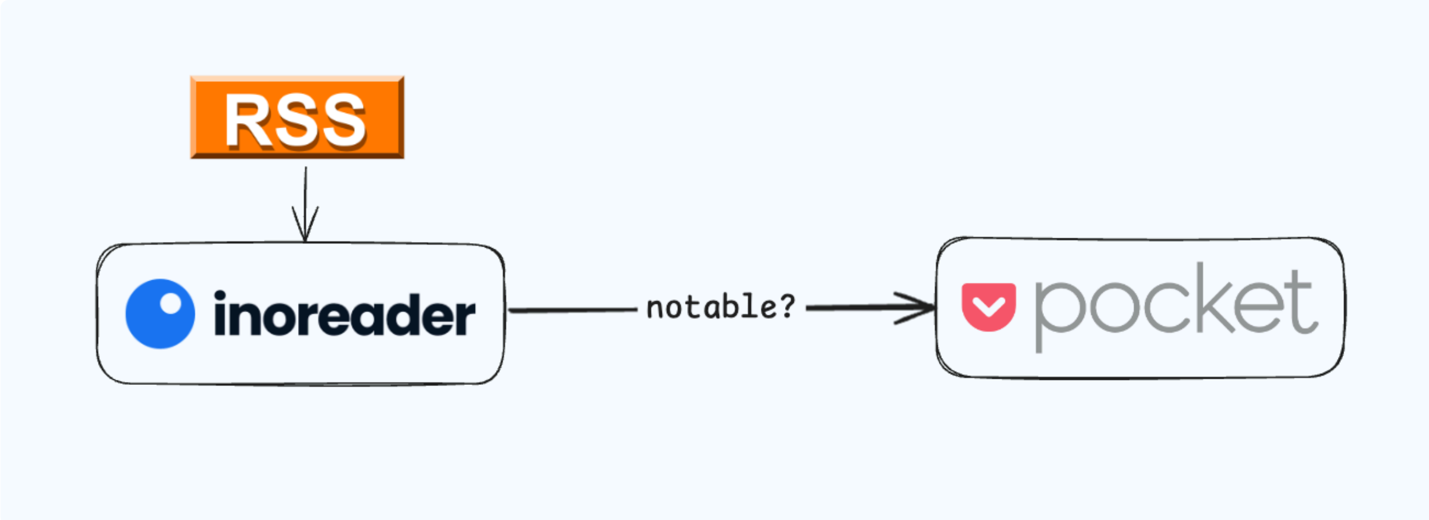
If you are interested in the technical implementation, the complete codebase is available in this github repository .

Creating an Articles Dataset

Step 1: Initial List of Liked Articles

Daily browsing of RSS feeds involves scanning through many articles, where sometimes engaging titles lead me to read their opening paragraphs. Through my established content workflow , I save interesting items to Pocket using Inoreader's built-in feature.

Initially, I planned to use a list of RSS items that I had clicked on. However, Inoreader doesn't provide an API to access this reading history, which led me to explore other options.



My RSS-based Content Consumption Workflow

Looking further into my workflow, I found a better solution: my archived items in Pocket. While I regularly read through my Pocket items, I never delete them - just archive them. This gave me a valuable collection of reading history.

Using the Pocket API , I retrieved around 4,000 URLs, dating back to December 2022. Though some archived items might be less relevant now, the dataset reflects my reading interests well. Later, I could add features like upvoting and downvoting to refine the content selection, but that's beyond the current scope.

Each item in the dataset includes this basic information:

```
{
  "pocket_item_id": 5598506,
  "given_url": "https://nat.org/",
  "resolved_url": "http://nat.org/",
  "title": "Nat Friedman",
  "time_added": 1736940058,
  "word_count": 451,
  "domain": "nat.org"
}
```

Step 2: Text Content of Articles

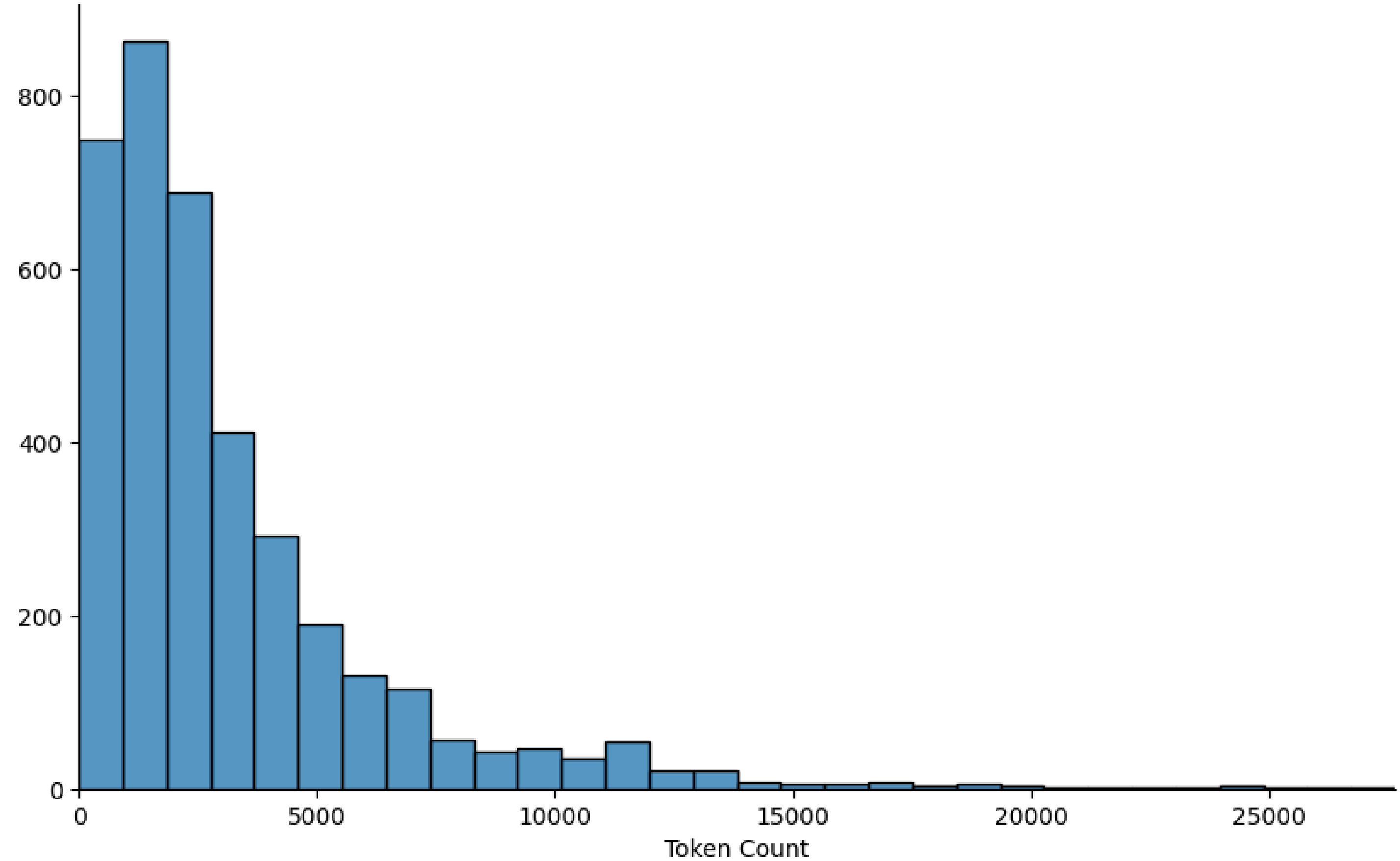
To build an effective recommendation model, we need the actual content of each URL, not just its metadata, but unfortunately Pocket doesn't provide that. While I love writing scrapers, for this project I want to focus on developing the recommendation model itself.

The Jina Reader API offers a straightforward solution, converting webpage content into markdown format. Here's an example of what we get (shortened version, full content available here):

```
Title: Nat Friedman

URL Source: https://nat.org/

Markdown Content:
I'm an investor, entrepreneur, developer.

Some things about me:

*   Grew up in Charlottesville, VA
*   On the Internet since 1991, which is my actual "home town"
*   Went to MIT because I loved the Richard Feynman [autobiographies](https://www.am
*   Started [two](https://en.wikipedia.org/wiki/Ximian) [companies](https://en.wikip
*   CEO of [GitHub](https://github.com/) from 2018 through 2021
*   Live in California
*   Working on reading the [Herculaneum Papyri](https://scrollprize.org/)
*   Tested 300 Bay Area foods for [plastic chemicals](https://plasticlist.org/)

Some things I believe:
...
```

The converted content is excellent, but it has two main challenges:

1. Markdown formatting adds unnecessary complexity for our use case. Plain text would work better.
2. Articles vary greatly in length - from short paragraphs to long essays - which could make comparing them via semantic similarity difficult later.



Distribution of Documents Lengths (in Tokens)

Step 3: Summarizing the Markdown Content

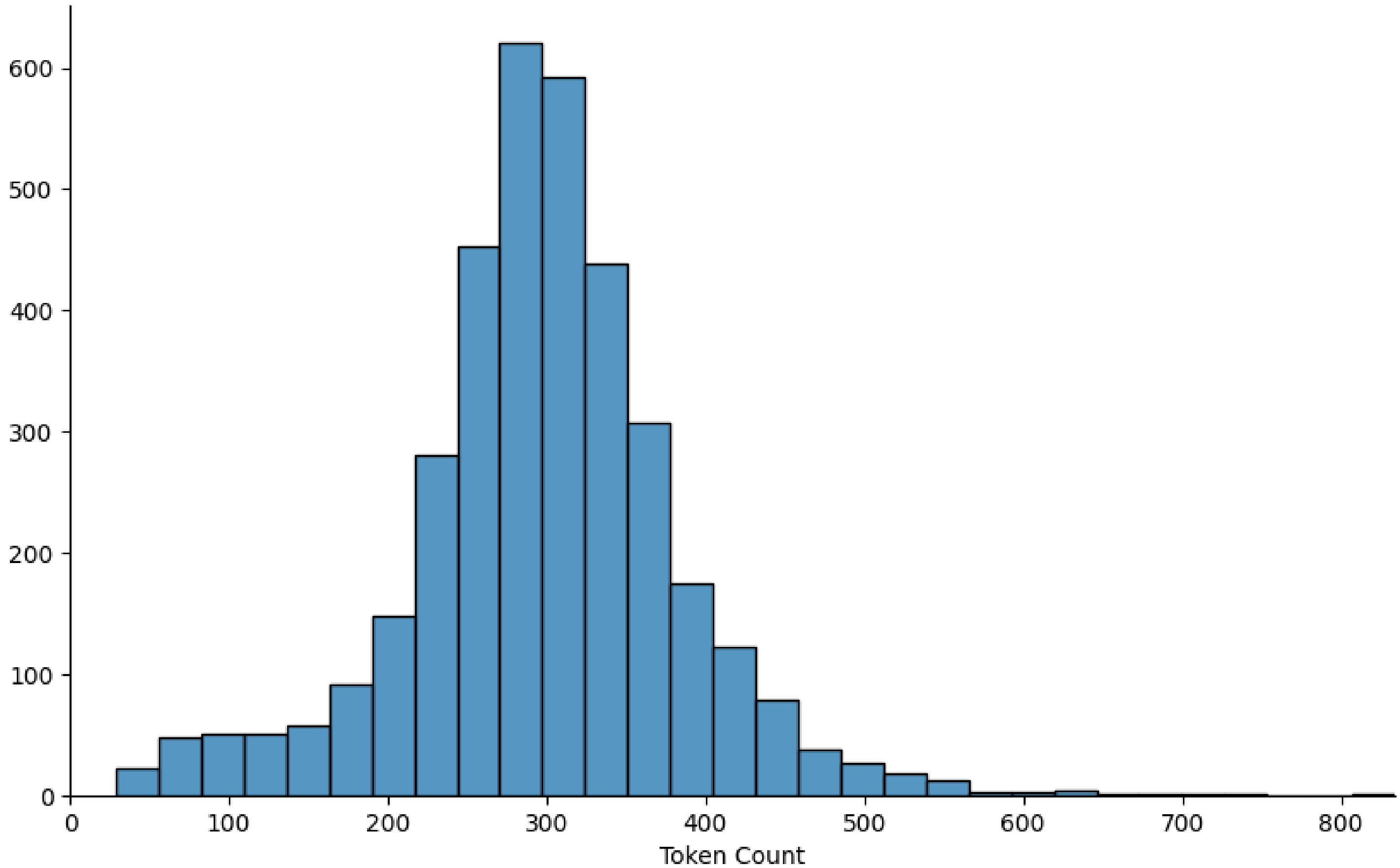
To solve both issues at once, I used the `gemini-2.0-flash` model to create consistent-length summaries of each document. Here's an example summary for `https://nat.org/` :

```
I'm an investor, entrepreneur, and developer who's been online since 1991, consideri

Fundamentally, I believe we have a right, perhaps a duty, to shape the universe to o

I also challenge the efficient market hypothesis, viewing it as a flawed heuristic,
```

By having the model to wrap summaries in XML tags (`<summary> ... </summary>`), I achieved two things:

- Clean extraction of just the summary text, in case LLM generates other texts before or after the summary (e.g. if it starts with `Sure, I can help you with summarizing the content ...`).
- Easy identification of broken links and error pages. This helped remove 141 invalid entries, leaving me with 3,642 quality documents.

The summaries created a more balanced distribution of text lengths:



Distribution of Summary Lengths (in Tokens)

Next Steps

The final dataset now contains well-organized entries with clean metadata and text summaries:


```
{
  "pocket_item_id": 5598506,
  "given_url": "https://nat.org/",
  "resolved_url": "http://nat.org/",
  "title": "Nat Friedman",
  "time_added": 1736940058,
  "word_count": 451,
  "domain": "nat.org",
  "text": "Title: Nat Friedman\n\nURL Source: https://nat.org/\n\nMarkdown Content:\
  "summary": "I'm an investor, entrepreneur, and developer who's been online since 1
}
```


Next week, I'll work on creating a user profile by concatenating these text summaries and metadata and converting them into vectors using Embedding models . I'm also researching how modern recommendation systems work with transformers and LLMs, which will help guide this project. After all, learning is the main goal here.

I welcome any thoughts or questions about this series - feel free to reach out!

Comment? Reply via Email, Bluesky or Twitter.

Recommendation Systems | Data Processing | Data Cleaning | LLM | Text Summarization | Python | Gemini | Projects

Related Posts

- Building a Personal Content Recommendation System, Part One: Introduction
- Released a new tool: llm-url-markdown
- Access Google Gemini LLM via OpenAI Python Library
- Never Been Easier to Learn
- Text Chunking and Headings Grouping: A Guide to Parsing Documents with Pandoc and Python