# Reasonable Deviations

a rational approach to complexity

© Robert Andrew Martin 2023. All rights reserved. Disclaimer.

# Man and machine: GPT for second brains

05 Feb 2023 · 10 min read [ productivity ]

In this post, I discuss how I used GPT embeddings to build a smart search tool for my second brain note-taking system.

0:00

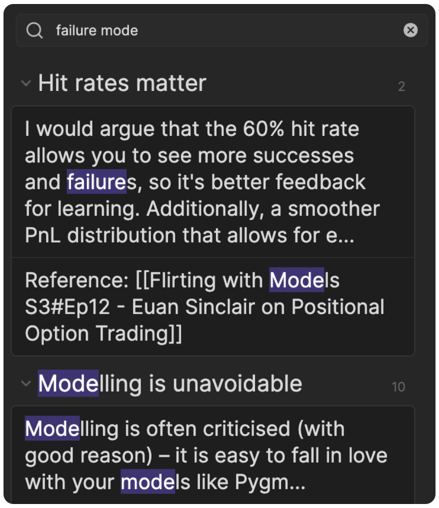
Try this if the video isn't working for you

# Introduction

I spend a lot of time thinking about how to improve my processes for learning and knowledge management; I've written extensively about my second-brain note-taking system, Molecular Notes.

I use Molecular Notes on a daily basis but am still constantly on the lookout for improvements. As my collection of notes grows, I have recently been running into a retrieval issue: sometimes I'm trying to find an idea I know I've written about but I can't remember the name of the concept or where I came across it.

For example, let's say I remember making some notes on failure modes in large organisations, but I can't remember any specific names or the source. I can try searching for relevant keywords in Obsidian search, but this doesn't help much. The problem is that Obsidian's default search is *lexical* – it tries to match words in the search query with words in the documents. It would be generous to call this "hit or miss": as you can see below, when I search for "failure mode", it matches anything with the string "mode", including words like "model" which are not relevant. I could do a literal search using double quotes, but that only returns exact matches.



failure mode

Hit rates matter 2

I would argue that the 60% hit rate allows you to see more successes and failures, so it's better feedback for learning. Additionally, a smoother PnL distribution that allows for e...

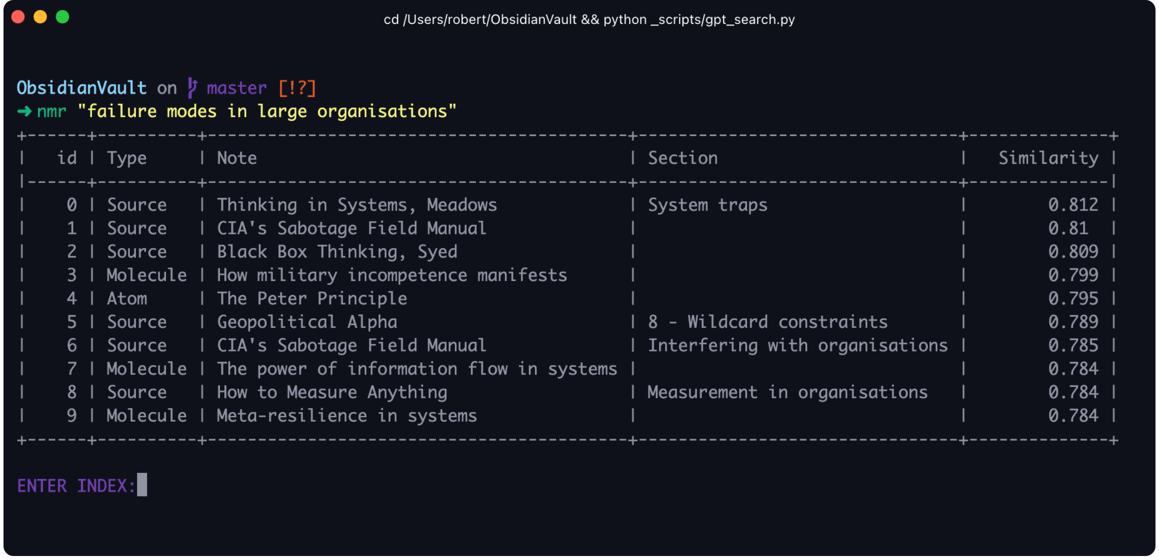
Reference: [[Flirting with Models S3#Ep12 - Euan Sinclair on Positional Option Trading]]

Modelling is unavoidable 10

Modelling is often criticised (with good reason) – it is easy to fall in love with your models like Pygm...

(Molecular Notes, through its emphasis on linked notes, provides other means of finding content; in this case I could look at the local graph of notes connected to a relevant topic, like Management, and hopefully find what I'm looking for that way).

I was in dire need of *semantic* search, where the search tool understands the content of my second brain enough to return notes that are similar in meaning rather than just having the same words.



```
cd /Users/robert/ObsidianVault && python _scripts/gpt_search.py

ObsidianVault on  master [!?]
➜ nmr "failure modes in large organisations"
+------+----------+------------------------------------------+--------------------------------+--------------+
|   id | Type     | Note                                     | Section                        |   Similarity |
|------+----------+------------------------------------------+--------------------------------+--------------|
|    0 | Source   | Thinking in Systems, Meadows             | System traps                   |        0.812 |
|    1 | Source   | CIA's Sabotage Field Manual              |                                |        0.81  |
|    2 | Source   | Black Box Thinking, Syed                 |                                |        0.809 |
|    3 | Molecule | How military incompetence manifests      |                                |        0.799 |
|    4 | Atom     | The Peter Principle                      |                                |        0.795 |
|    5 | Source   | Geopolitical Alpha                       | 8 - Wildcard constraints       |        0.789 |
|    6 | Source   | CIA's Sabotage Field Manual              | Interfering with organisations |        0.785 |
|    7 | Molecule | The power of information flow in systems |                                |        0.784 |
|    8 | Source   | How to Measure Anything                  | Measurement in organisations   |        0.784 |
|    9 | Molecule | Meta-resilience in systems               |                                |        0.784 |
+------+----------+------------------------------------------+--------------------------------+--------------+

ENTER INDEX:
```

# Basics of GPT

I don't claim to have a good understanding of the internals of large language models. So here is my "black box" understanding of GPT, which emphasises the inputs and outputs rather than the internal function.

GPT (which stands for "Generative Pre-trained Transformer") is a Large Language Model (LLM) – a fancy deep learning model suited to a variety of language tasks. GPT-3 is the third iteration of GPT, with 10x the number of parameters of v2, and presumably some multiplier of the linguistic power.

The main task of GPT-3 is text completion – feed it some input text and it tries to return the most appropriate output text. This is why we've seen amusing screenshots of GPT saying things that are completely wrong – it has no inbuilt truth-seeking, it's only trying to make stuff sound right (in fairness, I don't think humans are very different on this point).

Here is an overview of the different tools OpenAI has built on top of GPT-3:

- ChatGPT: a variant of GPT-3 optimised for dialogue. It abstracts away the text completion aspect by making it feel like GPT is answering your questions, and wraps it in a nice UI.
- Text completions API: feed in text programmatically and get back a completion. For example, check out this demo where the completions API is used to fill in a spreadsheet based on the surrounding cells. You could use the text completions API to build your own version of ChatGPT.
- DALL-E: an image generator based on GPT-3, which spits out images instead of text completions in response to some text prompt
- Embeddings API: this is what I used in the project. It is less intuitive than the others, so I'll explain more below.

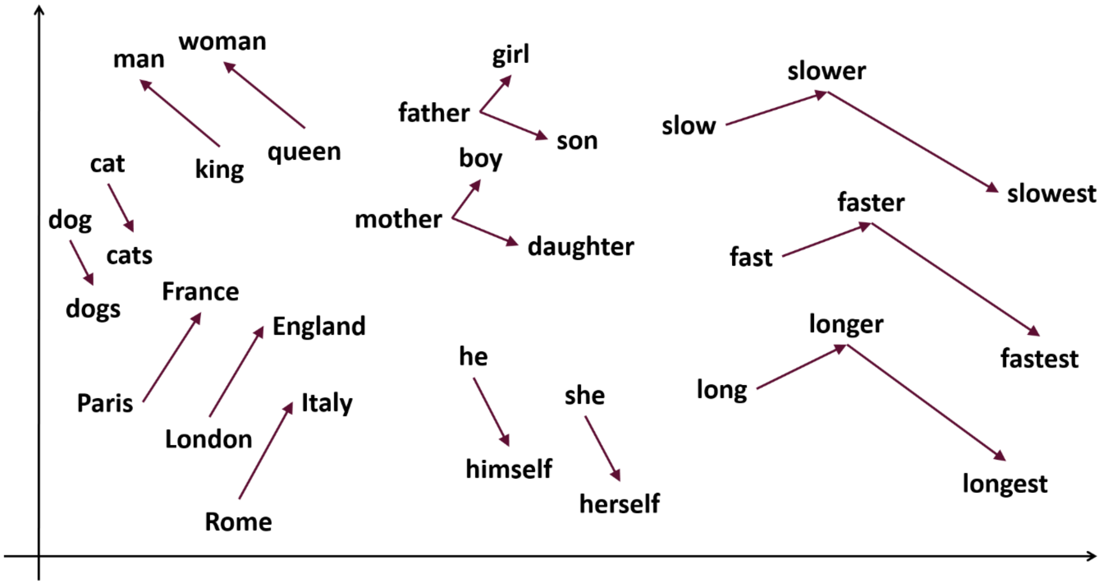
# Embeddings

The core idea of embeddings is to represent a block of text as a point in space (specifically, a vector in high dimensional space).

We mostly want to do this because once text has some spatial location, we can derive useful information based on its location. For example, we might be able to see that “dog” and “cat” are spatially closer than “dog” and “trumpet”.

Also, vectors are easier to mathematically manipulate than blocks of text: adding and subtracting vectors is just a matter of following arrows. So once we’ve represented words as vectors, we can more rigorously say things like:

- king = queen + man
- tiger = cat + stripes
- slower - slow + fast = faster



Samy Zafrany’s word2vec tutorial.

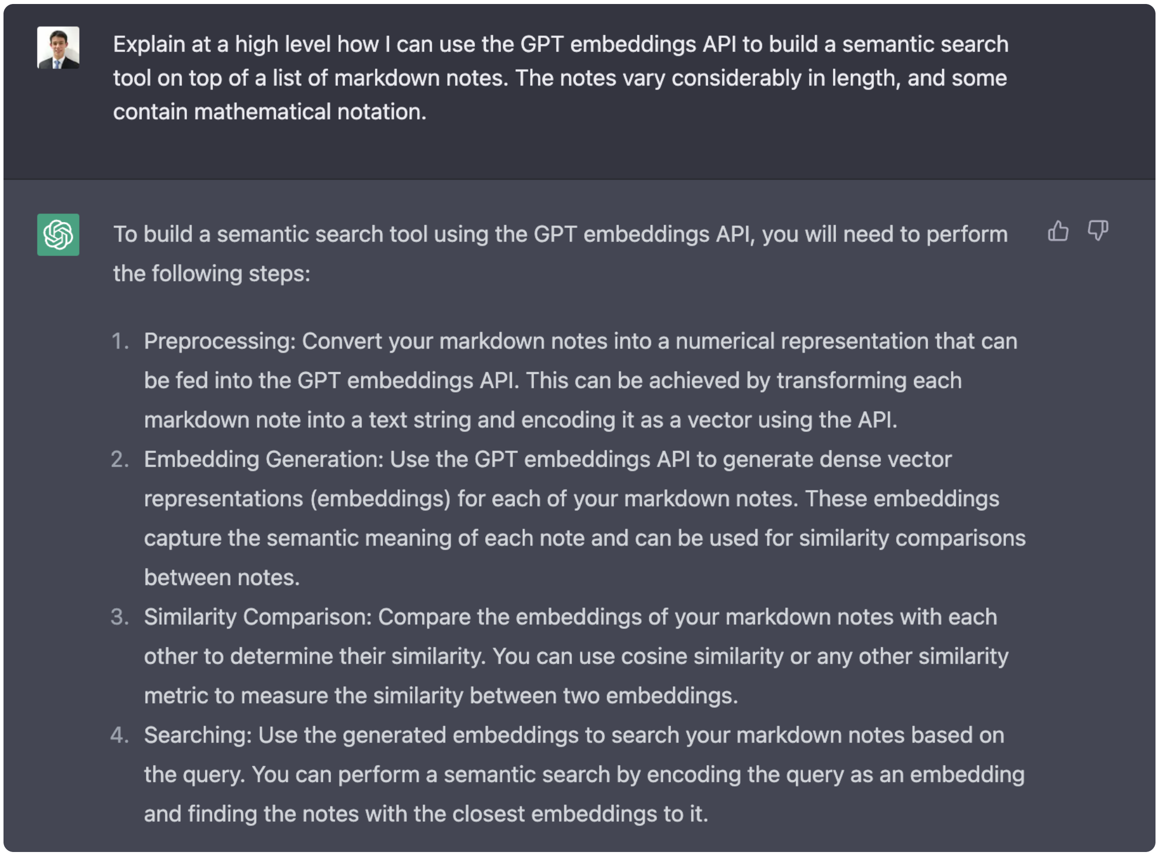
None of this is particularly new – word2vec was published in 2013. But the hope is that with LLMs like GPT-3, the embeddings can become even more expressive, encoding paragraphs or documents rather than just words and identifying abstract relationships between text blocks.

# GPT Embeddings for semantic search

The core workflow for semantic search is very straightforward – it’s basically the “hello world” of GPT projects.

1. Take a bunch of text blocks and feed them to the OpenAI embeddings API. This returns a 1536-dimensional vector for each text block.
2. Likewise, embed the search query using the same API to get a query vector.
3. Find the most similar vectors among my notes. There are several options to compute the similarity between vectors, but most people default to cosine distance, which measures how closely two vectors align.
4. Return the top n notes by similarity.

Here's ChatGPT's explanation, which is a bit more detailed.



Explain at a high level how I can use the GPT embeddings API to build a semantic search tool on top of a list of markdown notes. The notes vary considerably in length, and some contain mathematical notation.

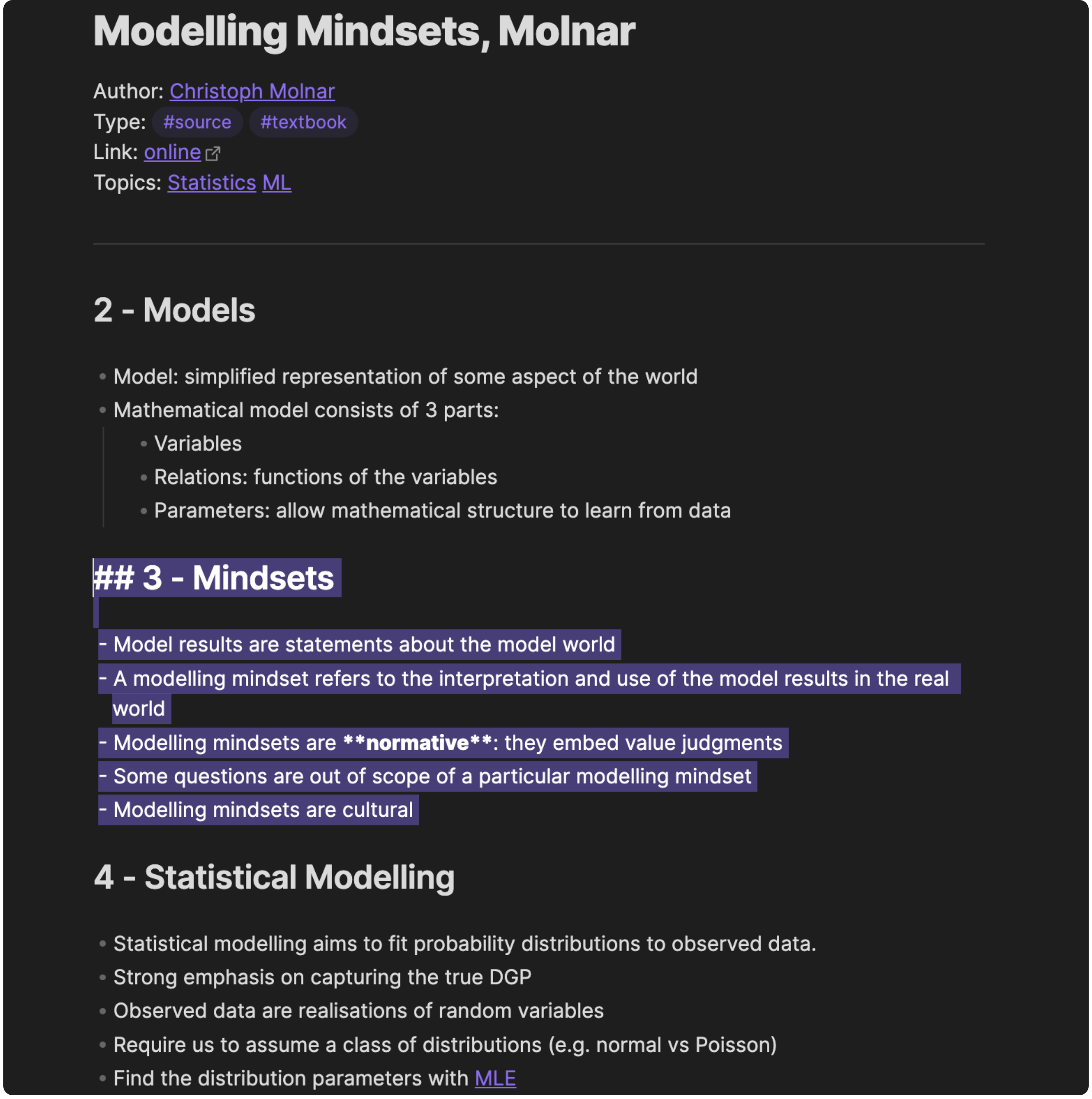
To build a semantic search tool using the GPT embeddings API, you will need to perform the following steps:

1. Preprocessing: Convert your markdown notes into a numerical representation that can be fed into the GPT embeddings API. This can be achieved by transforming each markdown note into a text string and encoding it as a vector using the API.
2. Embedding Generation: Use the GPT embeddings API to generate dense vector representations (embeddings) for each of your markdown notes. These embeddings capture the semantic meaning of each note and can be used for similarity comparisons between notes.
3. Similarity Comparison: Compare the embeddings of your markdown notes with each other to determine their similarity. You can use cosine similarity or any other similarity metric to measure the similarity between two embeddings.
4. Searching: Use the generated embeddings to search your markdown notes based on the query. You can perform a semantic search by encoding the query as an embedding and finding the notes with the closest embeddings to it.

The main nuance is deciding what constitutes a text block. There were two main considerations here:

- The OpenAI embeddings API limits the token size to 8191 (~6000 words). So each text block has to be smaller than that.
- What output do I want to be returned? It's not super useful if the search points me to a large note as I'd still have to scroll through that to find what I'm after.

Fortunately, my long notes are written in a fairly structured way, separated by headings. So I just fed in each section as a text block.



# Modelling Mindsets, Molnar

Author: Christoph Molnar
Type: #source #textbook
Link: online
Topics: Statistics ML

## 2 - Models

- Model: simplified representation of some aspect of the world
- Mathematical model consists of 3 parts:
  - Variables
  - Relations: functions of the variables
  - Parameters: allow mathematical structure to learn from data

## 3 - Mindsets

- Model results are statements about the model world
- A modelling mindset refers to the interpretation and use of the model results in the real world
- Modelling mindsets are **normative**: they embed value judgments
- Some questions are out of scope of a particular modelling mindset
- Modelling mindsets are cultural

## 4 - Statistical Modelling

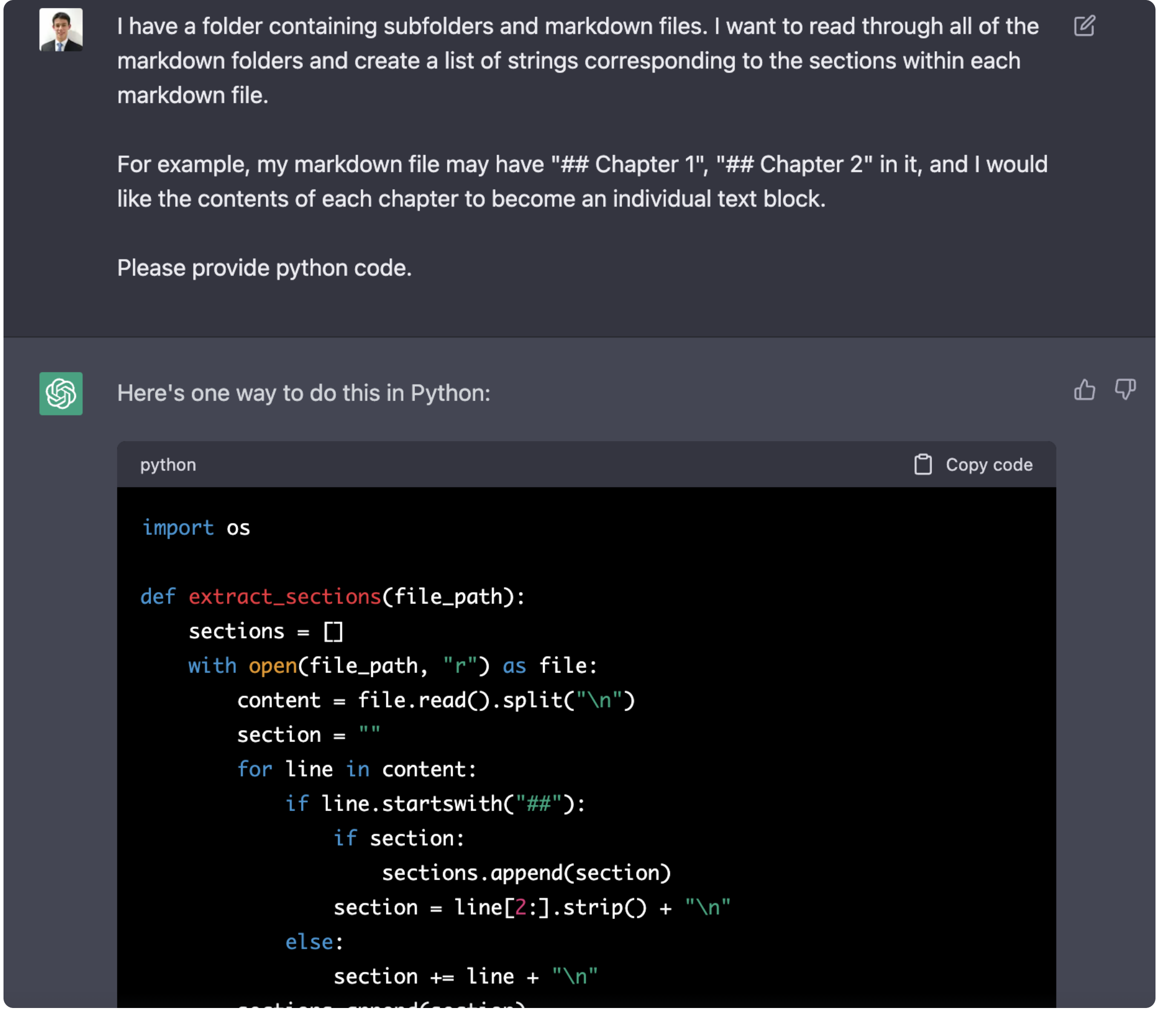
- Statistical modelling aims to fit probability distributions to observed data.
- Strong emphasis on capturing the true DGP
- Observed data are realisations of random variables
- Require us to assume a class of distributions (e.g. normal vs Poisson)
- Find the distribution parameters with MLE

# Technical implementation

The code is on GitHub, though the implementation is tied quite closely to my particular system and I don't intend to put in any effort to generalise it.

## Processing input

Because Obsidian notes are flat files, it's simple to write a python script to walk through the Obsidian vault to get all the notes into python. I asked ChatGPT to do it for me:



I have a folder containing subfolders and markdown files. I want to read through all of the markdown folders and create a list of strings corresponding to the sections within each markdown file.

For example, my markdown file may have "## Chapter 1", "## Chapter 2" in it, and I would like the contents of each chapter to become an individual text block.

Please provide python code.

Here's one way to do this in Python:

```python
import os

def extract_sections(file_path):
    sections = []
    with open(file_path, "r") as file:
        content = file.read().split("\n")
        section = ""
        for line in content:
            if line.startswith("##"):
                if section:
                    sections.append(section)
                section = line[2:].strip() + "\n"
            else:
                section += line + "\n"
```

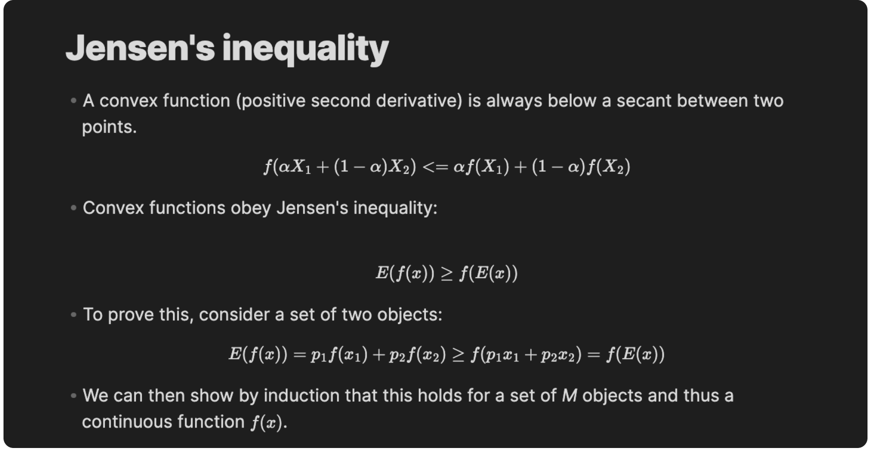
The result of this step is a dictionary representation of my entire Obsidian vault, mapping the note name and chapter to the note text.

```
notes = {('Some Podcast', 'Episode 1'): 'text',
         ('Some Podcast', 'Episode 2'): 'other text',
         ('Some book', 'Chapter 1'): 'my notes'}
```

For optimal embedding performance, I cleaned some of the text:

- Removed some of the metadata I use in each note (e.g. author tags, metadata).
- Remove newlines and special characters
- Remove URLs and just keep the name of the link
- Removed Obsidian backlinks (double square brackets).

I left mathematical notation untouched, because ideally I want the search engine to understand maths. It seems to have some understanding – when I query “expectation of a function versus the function of an expectation”, it does suggest Jensen’s inequality, even though my note on Jensen’s inequality doesn’t mention the word “expectation” except in mathematical terms. But I can’t be sure, perhaps it’s just picking up on the word “function”.



## Jensen's inequality

- A convex function (positive second derivative) is always below a secant between two points.

$$f(\alpha X_1 + (1-\alpha)X_2) <= \alpha f(X_1) + (1-\alpha)f(X_2)$$

- Convex functions obey Jensen's inequality:

$$E(f(x)) \geq f(E(x))$$

- To prove this, consider a set of two objects:

$$E(f(x)) = p_1 f(x_1) + p_2 f(x_2) \geq f(p_1 x_1 + p_2 x_2) = f(E(x))$$

- We can then show by induction that this holds for a set of *M* objects and thus a continuous function $f(x)$.

# Embeddings

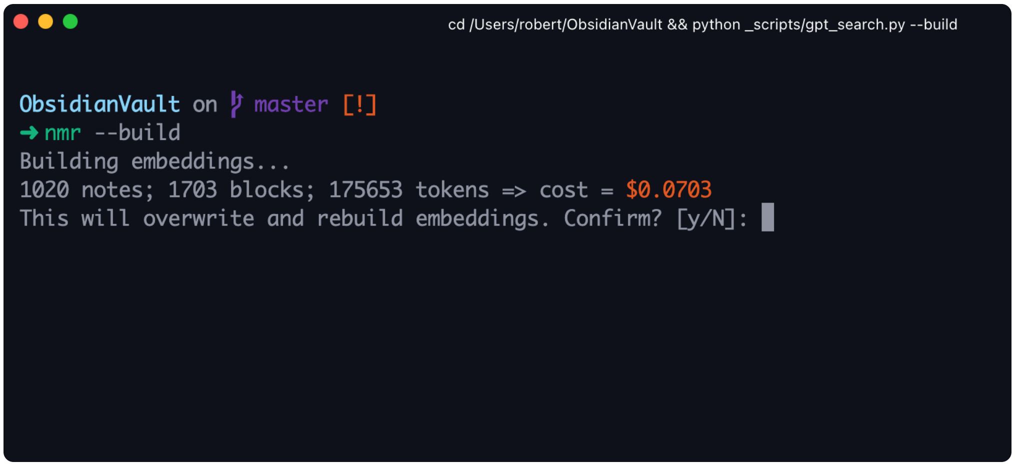
This is the easiest part of the project: OpenAI does the heavy lifting. The embedding function (which I took from one of their cookbooks) takes in a text block and outputs a vector.

```
def get_embedding(block: str) -> list[float]:
    return openai.Embedding.create(
                    input=block, model="text-embeding-ada-002"
                )["data"][0]["embedding"]
```

I simply iterate through the dictionary of notes, applying the `get_embedding` function and storing the vectors in a pandas dataframe – this is almost certainly not the best structure but I'm familiar with them and the data is small enough where performance isn't an issue.

```
def embed(notes: dict[(str, str), str]) -> pd.DataFrame:
    # Embeds the notes into openAI and returns a dataframe containing
the vectors.
    res = {}
    showfunc = lambda n: f"{n[0][0]} {n[0][1]}" if n else ""
     with click.progressbar(notes.items(), item_show_func=showfunc) as
note_items:
        for (note, section), text in note_items:
            block = section + ". " + text
            n = num_tokens_from_string(block)
            # Truncate if too long
            if n > EMBEDDING_CTX_LENGTH:
                warnings.warn(f"{note} {section} exceeded token limit.
Truncating.")
                block = truncate_text_tokens(block)
            try:
                embedding = get_embedding(block)
            except Exception as e:
                print(f"Error for {note} {section}", e)
                continue
            res[(note, section)] = embedding
            time.sleep(0.1)
    df = pd.DataFrame(res)
    return df
```

I wrote a utility function to compute the cost of embedding a list of notes and requesting user confirmation before executing. Training on 1000+ notes only cost 7 cents!



```
cd /Users/robert/ObsidianVault && python _scripts/gpt_search.py --build

ObsidianVault on master [!]
➜ nmr --build
Building embeddings...
1020 notes; 1703 blocks; 175653 tokens => cost = $0.0703
This will overwrite and rebuild embeddings. Confirm? [y/N]:
```

I then used the same `get_embedding` function to process the query string, before computing the cosine distance against the note embeddings in the dataframe:

```
qvec = get_embedding(query_string)
cos_sim = np.apply_along_axis(lambda x: cosine_similarity(x, qvec), axis=0, arr=df)
results = pd.Series(cos_sim, index=df.columns).sort_values(ascending=False)
```

Finally, I sorted the resulting notes by similarity and returned them.

## User interface

I built a CLI (command-line interface) using python's click library. This allows for easy creation of user confirmation prompts, progress bars, and styling.

Best of all, with `click.launch(link)` one can actually launch Obsidian, though it took some effort to get the link in the right format.

Firstly, I installed the Advanced URI extension in Obsidian, which allows you to link to text blocks within notes rather than just the note itself (a URI is like a URL but can link to things other than web pages). I then reverse-engineered the format of the URI string, allowing me to convert my `(note_title, chapter_title)` pairs (which also keys the dataframe of embeddings vectors) into a proper URI. In the CLI, when I enter the note index, the URI is built and passed to `click.launch` which opens Obsidian.

# Next steps

There are several other ways I think GPT-3 can be used to augment my second brain experience. Here are some ideas, in descending order of importance.

## Link recommendation

I currently have an Obsidian extension that uses various distance metrics to list similar notes. This is based on the graph structure and backlinks: e.g. if note A links to note C, and note B links to note C, maybe A and B are similar.

But perhaps the GPT embeddings can capture more nuance in meaning. I plan to write a small script to sort all *unlinked* notes by GPT-similarity (for linked notes, I have already manually identified that there is a similarity so these should be excluded). I'm curious to see if there are any abstract relations I have missed: for example, maybe there is some idea from a particular event in Roman History that is a nice anecdote for a financial blowup in the 21st century.

## Summarisation

There are many projects out there using GPT-3's text completions for Obsidian, for example, obsidian-gpt allows users to generate text in Obsidian.

I feel quite strongly that this is a bad idea for second brains – writing content in my own words is a crucial aspect of Molecular Notes. Though it's possibly useful if you use Obsidian as part of your content creation workflow.

Instead, I'm intrigued by the idea of using GPT-3 as a kind of "Jarvis". Instead of entering queries to search, with GPT-3 I could make requests like:

- Show me my notes on investment frameworks
- Summarise the above notes
- Highlight any disagreements between the notes

# Conclusion

This was a surprisingly easy project and it's certainly not because of my coding skills. OpenAI has done all the heavy lifting, exposing a huge amount of functionality with some simple API calls.

I think there are some generalisable lessons: much of the effort involved getting my data into a suitable format for OpenAI to ingest and creating a user experience. I was lucky with the former because my data (small text blocks) is essentially the perfect input, whereas for some other projects people are trying to come up with clever ways of chunking up large pdf documents so they can be embedded. The UX side was also simple because I'm happy to use a plain CLI. I assume it's a lot more effort to build a consumer-facing web application.

This notwithstanding, I think the barrier to building a cool product has never been lower. I strongly encourage readers to examine their current processes to identify areas ripe for AI integration. I don't want to be replaced by AI, I want to be enhanced by it!

---

## Related Posts

Molecular Notes: Practice 12 Jun 2022

Molecular Notes: Principles 18 Apr 2022

How I Read Books 24 Jan 2022