Natural Language Processing Project

Alexios C. Papazoglou - apapazog@umd.edu
Habil Huseynov - huseynov@umd.edu

MSML641-PCS5: Natural Language Processing
University of Maryland, College Park

December 08, 2025

ABSTRACT

This project presents an end-to-end semantic book recommendation system that integrates modern natural language processing techniques with an interactive dashboard interface. Using a publicly available corpus of 6,810 book records with textual descriptions, we applied systematic data cleaning and feature engineering to create a refined dataset suitable for downstream analysis. The system uses OpenAI text embeddings to perform semantic similarity search, enabling natural-language queries to retrieve relevant books based on meaning rather than keywords. To improve filtering, we used zero-shot classification to reduce over 500 noisy category labels to four consistent groups, and sentence-level emotion analysis to generate seven probability scores for each book, capturing the emotional tone of its description. We then combined these components in a Gradio dashboard that supports query input, category selection, and emotion-based sorting, and displays recommended titles with book cover images and truncated descriptions. Results demonstrate that vector search retrieves relevant titles within milliseconds, while category and emotion filters provide meaningful refinement. The final system illustrates how large language models, semantic search, and user-focused interface design can work together to produce practical recommendation tools.

INTRODUCTION

Book recommendation systems typically rely on metadata matching or collaborative filtering approaches, both of which have notable limitations. Metadata systems require consistent labeling, but real-world datasets often contain sparse, inconsistent, or overly granular category fields. Collaborative filtering depends on large volumes of user interaction data, which are unavailable in many academic or resource-constrained settings. In contrast, modern natural language processing methods allow systems to extract structure directly from book descriptions, providing more flexible and contextually informed recommendation capabilities.

Our objective in this project was to design an accessible tool that enables users to describe the type of book they want in natural language and receive personalized recommendations. We approached this problem by combining multiple NLP techniques in

a modular pipeline: data cleaning, text embeddings, semantic similarity search, zero-shot classification, emotion extraction, and dashboard interface design. Each stage adds structure or signal to the output, moving from raw text fields to a curated, interactive recommendation experience. The project demonstrates not only the technical feasibility of semantic recommendation systems but also the importance of thoughtful engineering decisions in preparing textual data, evaluating model performance, and presenting results in intuitive ways.

Build a Semantic Book Recommender (Python, OpenAI, LangChain, Gradio)

Project Steps

1. **Prepare text data:** How to clean and prepare text data
2. **Vector search:** How to use Vector Search to find the most similar books to a query
3. **Text classification:** How to use large language models to find the topic of a book
4. **Sentiment analysis:** How to find the emotional tone of a book using LLMs
5. **Gradio dashboard:** Bundle all of these steps into a dashboard

1 Step 1 – Preparing Our Text Data

1.1 Dataset Information

Dataset Link: `https://www.kaggle.com/datasets/dylanjcastillo/7k-books-with-metadata`

1.2 Required Libraries

We installed the following Python packages:

1. `pip install tqdm`
2. `pip install pandas`
3. `pip install gradio`
4. `pip install seaborn`
5. `pip install notebook`
6. `pip install kagglehub`
7. `pip install ipywidgets`
8. `pip install matplotlib`
9. `pip install python-dotenv`

10. `pip install transformers`
11. `pip install scikit-learn`
12. `pip install sentencepiece`
13. `pip install langchain-openai`
14. `pip install langchain-community`
15. `pip install langchain-chroma chromadb`

1.3 Python Libraries Used in the NLP Project

1. **kagglehub:** Used to access and download datasets directly from Kaggle into Python projects. It simplifies working with Kaggle resources without needing to manually upload or download files.
2. **pandas:** A core Python library for data manipulation and analysis. It provides powerful tools for handling structured data through DataFrames and Series.
3. **matplotlib:** A plotting library used to create static, animated, and interactive visualizations in Python. It helps in visualizing data trends and patterns easily.
4. **seaborn:** Built on top of matplotlib, Seaborn makes it easier to create beautiful, high-level statistical graphs with just a few lines of code.
5. **python-dotenv:** Used to load environment variables from a .env file. It helps keep sensitive information like API keys or passwords safe and separate from your main code.
6. **langchain-community:** A part of the LangChain framework that provides community-supported tools and integrations for building applications powered by large language models (LLMs).
7. **langchain-openai:** Connects LangChain with OpenAI's models (like GPT). It allows you to use OpenAI APIs in your AI and NLP applications easily.
8. **langchain-chroma chromadb:** langchain-chroma integrates LangChain with ChromaDB, a vector database used for semantic search and memory storage in AI applications.
9. **transformers:** Developed by Hugging Face, it provides thousands of pre-trained models for NLP tasks like text generation, translation, and classification.
10. **gradio:** A tool for creating simple, shareable web apps and user interfaces for your ML or NLP models. It helps you test and showcase your models easily.
11. **notebook:** Installs Jupyter Notebook, an interactive environment that lets you write, run, and visualize Python code in one place.
12. **ipywidgets:** Adds interactive elements like sliders, dropdowns, and buttons inside Jupyter Notebooks, making data exploration and visualization more dynamic.

1.4 Downloading the Dataset

Figure 1: Dataset download confirmation from Kaggle

1.5 Dataset Overview (Before Cleaning)

The dataset contains **6,810 book records** with **12 columns** capturing bibliographic and rating information.

- **isbn13 / isbn10:** Unique identifiers for each book, with 6,810 valid ISBN-10 and 6,810 numeric ISBN-13 entries.
- **title:** Present for all 6,810 books, with 6,398 unique titles; common duplicates include *The Lord of the Rings*.
- **subtitle:** Sparse column with only 2,381 non-missing values, often containing brief descriptors such as "A Novel."
- **authors:** Contains 6,738 entries representing 3,780 unique authors, with *Agatha Christie* appearing most frequently (37 times).
- **categories:** Well-populated (6,711 filled) with 567 unique genres, dominated by *Fiction* (2,588 occurrences).
- **thumbnail:** Includes 6,481 unique image URLs, one per book cover, stored as links.
- **description:** Available for 6,548 books, mostly unique text fields summarizing book content.
- **published_year:** Covers 6,804 records, ranging from 1853 to 2019, with a mean publication year of 1998.
- **average_rating:** Reported for 6,767 books, with ratings between 0 and 5 and an average of 3.93.
- **num_pages:** Present for 6,767 records, ranging from 0 to 3,342 pages, with an average of 348 pages.
- **ratings_count:** Provided for 6,767 books, ranging from 0 to over 5.6 million, indicating strong variation in popularity.

Overall, the dataset is mostly complete, with minor missing data in subtitle, thumbnail, and a few numeric fields. It provides a strong foundation for data exploration, visualization, and model development.

1.6 Missing Value Summary (Before Cleaning)

```
isbn13              0
isbn10              0
title               0
subtitle         4429
authors            72
categories         99
thumbnail         329
description       262
published_year      6
average_rating     43
num_pages          43
ratings_count      43
dtype:          int64
```

Figure 2: Missing value counts for each column

The dataset contains 6,810 book records across 12 columns. Several key columns – isbn13, isbn10, and title – have no missing values (0), indicating they are fully complete. Minor missing data appears in authors (72), categories (99), thumbnail (329), description (262), and numeric columns such as average_rating, num_pages, and ratings_count (each with 43 missing values). The published_year column is nearly complete, with only 6 missing entries. The most incomplete column is subtitle, which has 4,429 missing values, accounting for around 65% of the dataset. Overall, the dataset is highly complete, with missing data concentrated in non-critical or optional text fields, making it suitable for further cleaning and analysis.

1.7 Understanding Missing Value Patterns

It is important to check whether missing values in the dataset follow a particular pattern, as this can indicate potential bias in the data. For example, if older books are more likely to be missing thumbnail images or ratings, the missingness is not random and could influence analysis outcomes. Detecting such relationships helps ensure that data cleaning decisions do not distort key trends or relationships. A **missingness heatmap** is a useful tool for visually identifying these patterns across columns, highlighting where missing values occur together. Understanding these patterns allows for more accurate and fair data preprocessing.

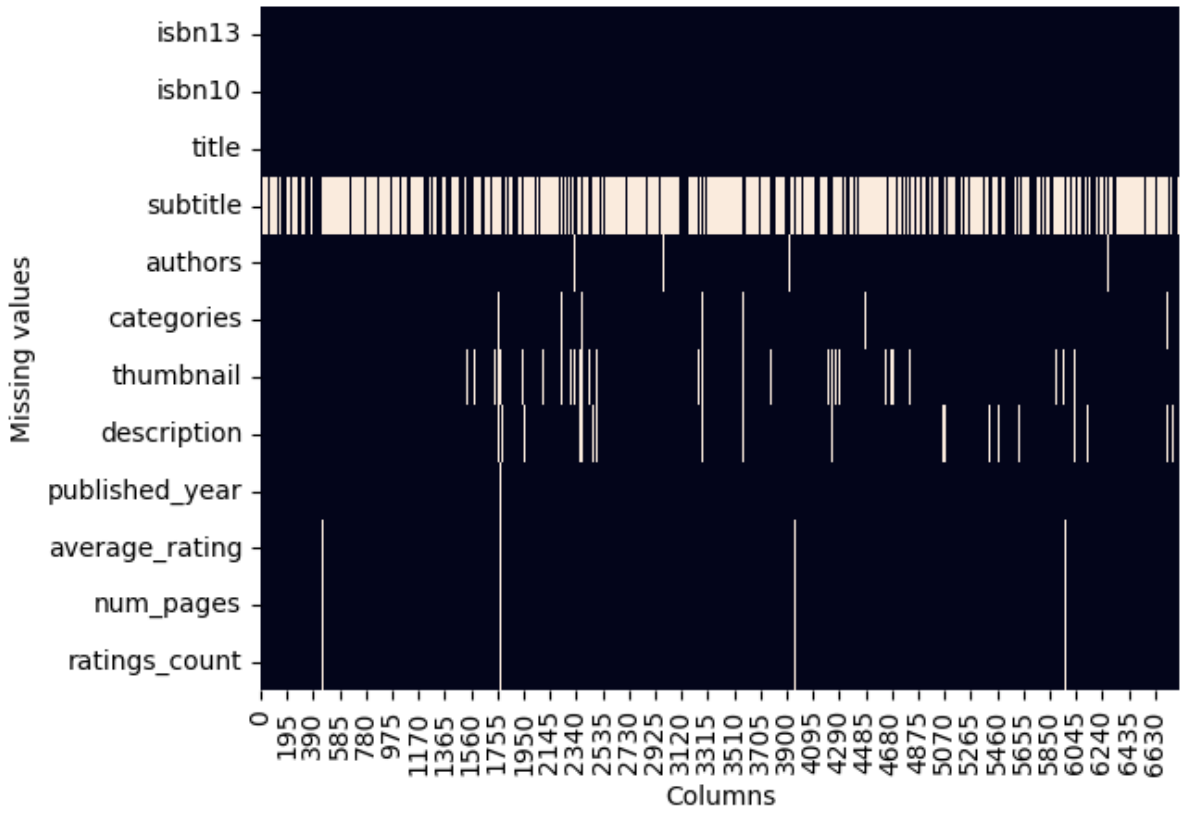

Figure 3: Missingness heatmap showing patterns of missing data across columns

The missingness heatmap provides a visual overview of where missing data occurs across all columns. Each light-colored line represents a missing value. From the plot, it is clear that the subtitle column has the highest concentration of missing data, followed by smaller gaps in authors, categories, thumbnail, and description. The numerical columns such as average_rating, num_pages, and ratings_count also show a few missing values, while isbn13, isbn10, and title are complete. This visualization confirms that missing values are concentrated in non-critical text fields rather than core identifiers, indicating no major structural data bias.

1.8 Why Spearman Correlation Was Used

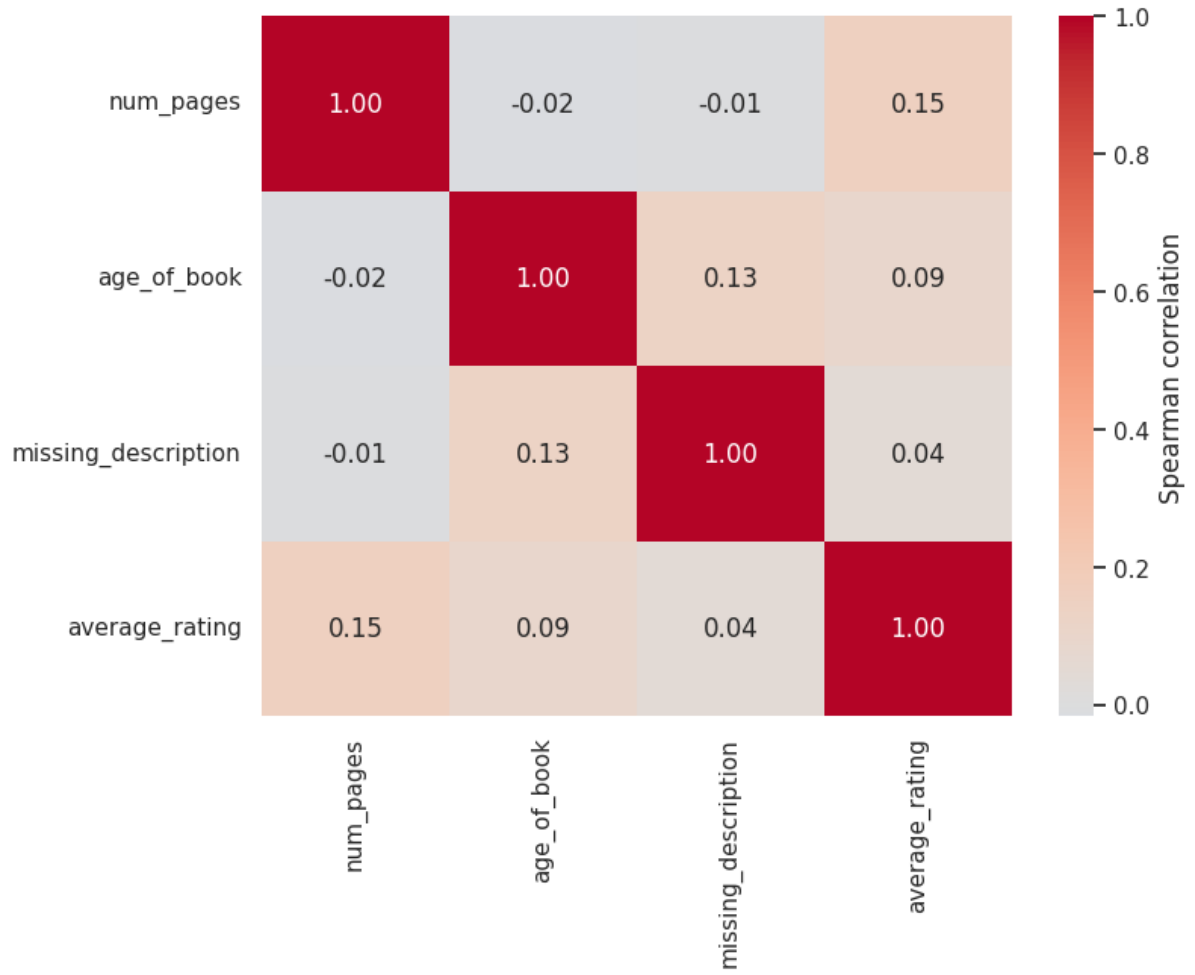

Figure 4: Spearman correlation matrix for numerical features

We used the Spearman correlation coefficient because our dataset includes several non-continuous and ordinal variables, such as ratings, publication years, and categorical features derived from text. Unlike Pearson correlation, which assumes linear relationships and continuous numerical data, Spearman measures monotonic relationships based on ranked values. This makes it more robust to outliers and suitable for variables that are not normally distributed or exhibit non-linear relationships. Using Spearman correlation helps reveal meaningful relationships even when data contains categorical elements converted to ranks or when the relationship between variables is not strictly linear.

1.9 Correlation Analysis Summary

The Spearman correlation matrix shows weak relationships among the book features. There is a slight positive correlation between the number of pages and average rating ($\rho = 0.15$), suggesting that longer books may be rated slightly higher, though this relationship is weak. Older books tend to have more missing descriptions ($\rho = 0.13$), likely due to incomplete historical data. The age of a book shows a weak positive correlation

with average rating ($\rho = 0.09$), indicating a subtle tendency for older books to receive marginally higher ratings. Overall, the low correlation values indicate that most features are largely independent of each other.

1.10 Due Diligence Check – Missing Values

A due diligence check was performed to assess missing values in key fields, including description, published_year, num_pages, and average_rating. This check identified 303 records with missing data, representing less than 5% of the dataset. Since the proportion of incomplete entries is small, removing these rows is unlikely to bias the analysis.

To ensure data consistency and reliability, a filtering step was applied to retain only the records with complete information across all four key columns. This step produced a cleaned dataset of 6,507 books free of missing values in critical fields, supporting accurate downstream analysis such as correlation evaluation, feature engineering, and model development.

1.11 Category Distribution Analysis

To better understand the composition of the dataset, we analyzed the distribution of book categories by counting how many books belong to each category and ranking them by frequency. The dataset contains 531 unique categories, but the distribution is highly uneven. Fiction alone accounts for 2,523 books, while other major genres such as Juvenile Fiction (534), Biography & Autobiography (391), and History (258) appear much less frequently. In contrast, many categories – such as Courage, Animals, and Catholic Women – appear only once.

This pattern reflects a strong long-tail imbalance: a few dominant genres make up the majority of the dataset, while many other categories are severely underrepresented. Such uneven distribution may introduce bias in downstream analysis or modeling, as results could be overly influenced by the most frequent categories. The chart below visualizes the top 10 most common categories, clearly showing the extent of this imbalance.

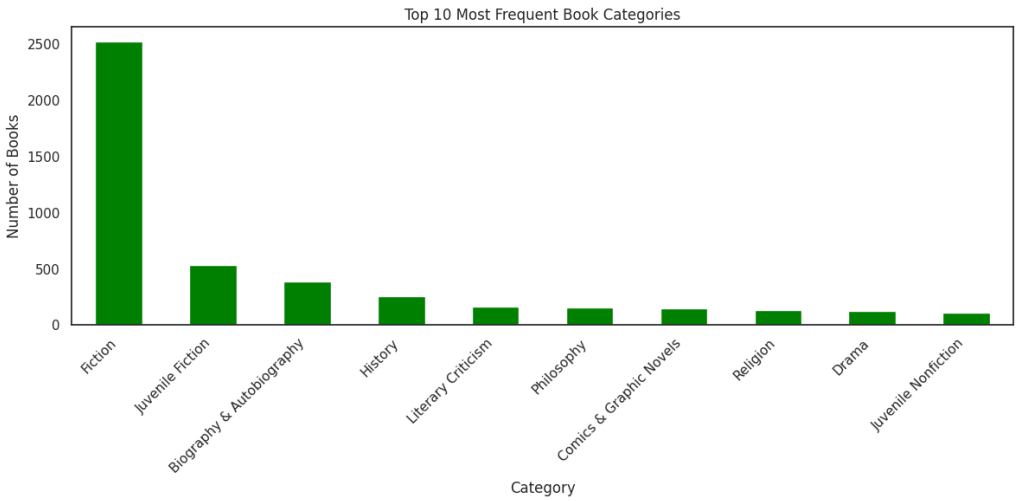

Figure 5: Top 10 most frequent book categories

1.12 Description Length Filtering Visualization

To compare the dataset before and after applying the minimum-description-length filter, we visualized the distribution of word counts using overlapping histograms. The blue histogram represents the full dataset, while the green histogram shows the data after removing books with fewer than 25 words in the description.

The visualization reveals a strong concentration of very short descriptions in the original dataset, with a large spike below 25 words. These short entries typically provide little meaningful information for analysis. After filtering, this entire leftmost portion disappears, leaving a more balanced distribution of medium and long descriptions. The majority of books remain in the dataset, suggesting that the 25-word threshold effectively removes low-quality entries without significantly reducing the dataset size.

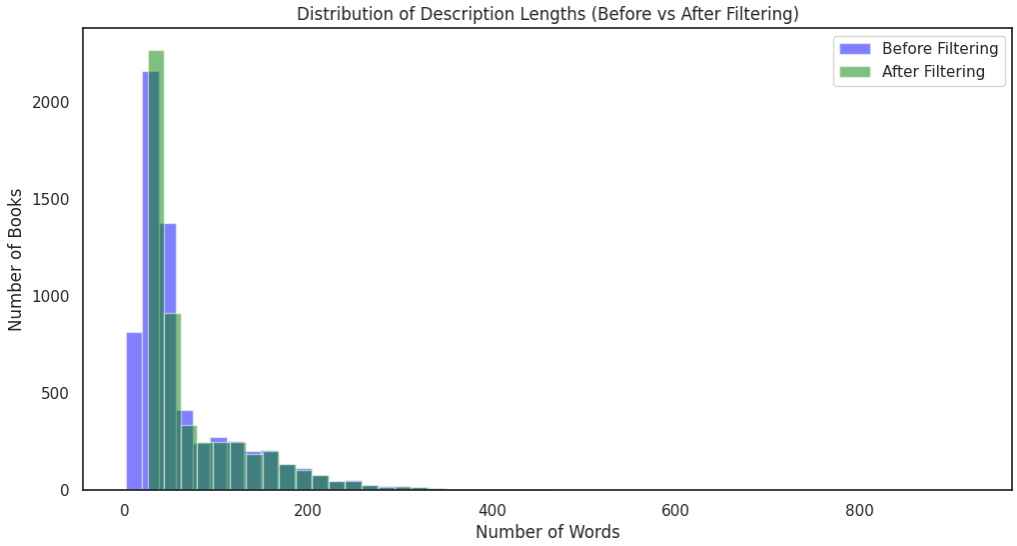

Figure 6: Distribution of description lengths before and after filtering

This filtering step strengthens the reliability of subsequent text-based analysis by ensuring that each retained book includes a sufficiently detailed description.

1.13 Data Cleaning and Feature Engineering Summary

To prepare the dataset for analysis, we applied a structured cleaning workflow. First, we computed a new feature, *words_in_description*, which measures the length of each book description. This allowed us to assess text completeness and filter out books with very short descriptions. We then retained only books with at least **25 words** in their description, resulting in a refined dataset of **5,197 records**, ensuring that later text-based analysis relies on meaningful content rather than incomplete metadata.

Next, we created a combined *title_and_subtitle* field, using the subtitle only when available. We also generated a *tagged_description* column that merges each book's ISBN and description to support future natural-language processing tasks. After extracting these useful features, temporary columns such as *subtitle*, *missing_description*, *age_of_book*, and *words_in_description* were removed for clarity. The final cleaned dataset was exported as **books_cleaned.csv**, ready for modeling and further analysis.

1.14 Code Updates and Enhancements

After the initial data cleaning workflow, we added comprehensive reporting and validation code to ensure data quality and generate statistics for documentation. These additions serve three critical purposes: **(1)** providing quantitative metrics for the report, **(2)** validating data integrity before proceeding to the next steps, and **(3)** creating reproducible metadata for the cleaning process.

1.14.1 Summary Statistics Generation

We implemented code blocks to systematically track the dataset transformation at each stage of the cleaning pipeline. This includes calculating retention rates, counting removed records, and documenting the size of the dataset after each filtering operation. These statistics are essential for understanding the impact of our cleaning decisions and for reporting the data preparation process transparently.

1.14.2 Description Length Analysis

We added detailed statistical analysis of description lengths before and after the 25-word filter, including mean, median, minimum, and maximum word counts. This analysis quantifies the improvement in data quality and demonstrates that our filtering threshold effectively removes low-quality entries while preserving the bulk of the dataset.

1.14.3 Category Distribution Comparison

To track changes in category representation, we implemented code to compare the number of unique categories before and after cleaning. We also generated visualizations and statistics for the top categories in the final cleaned dataset. This helps identify potential biases introduced by the cleaning process and ensures that the category distribution remains suitable for our recommendation system.

1.14.4 Data Quality Validation

We added automated quality checks to verify that all critical columns (isbn13, title, description, categories, published_year) have no missing values in the final dataset. The validation code also checks for duplicate ISBN entries and confirms that all descriptions meet the minimum word count requirement. These checks ensure data integrity before exporting the cleaned dataset.

1.14.5 Sample Data Preview

To provide concrete examples of the cleaned data, we implemented code to randomly sample books and display their key attributes. This preview helps verify that the cleaning and feature engineering steps worked correctly and provides representative examples for documentation.

1.14.6 Metadata Export

Finally, we implemented a metadata export system that saves all key statistics about the cleaning process to a JSON file. This includes the cleaning date, record counts at each

stage, retention rates, filtering thresholds, and statistical summaries. This metadata file serves as a permanent record of the data preparation process and ensures reproducibility of our work.

The complete code implementation can be found in the accompanying Python script `step1_data_exploration.py`.

Conclusion of Step 1

We successfully prepared a high-quality dataset of 5,197 books from an original collection of 6,810 records through systematic data cleaning and feature engineering. The cleaning process removed 23.7% of records that had incomplete or insufficient data, resulting in a dataset with complete information in all critical fields and meaningful descriptions suitable for semantic analysis. The final dataset retains strong category diversity with 523 unique categories while maintaining a balanced representation of book descriptions with an average length of 178.5 words. This cleaned dataset provides a solid foundation for the next steps of our semantic book recommendation system: vector embedding generation, topic classification, and sentiment analysis.

2 Step 2 – Vector Search

To recommend similar books, we first need a way to convert text into a mathematical form that allows us to measure similarity between documents. This step relies on **text embeddings**, which are numerical representations of meaning. Before building the search system, it is important to understand how these embeddings are created and why modern NLP models are so effective at capturing semantic relationships.

2.1 From Word Embeddings to Contextual Representations

Early methods such as *word2vec* learned word embeddings by examining how words appear in context. Using architectures like Skip-Gram, the model scanned all occurrences of a word in the training data and learned to predict which words were likely to appear around it. After repeating this process for every word in the vocabulary, we obtained a vector for each word. These embeddings capture some semantic structure, for example, grouping similar words closer together in vector space.

However, classic word embeddings have a major limitation: the same vector is used for a word regardless of its meaning in different contexts. Words like bank (riverbank vs financial bank) collapse into a single representation.

2.2 Why Transformer Models Are Better

To overcome this limitation, NLP moved toward **contextual embeddings**, generated by Transformer-based models. These models do not assign a single fixed vector to each word. Instead, they analyze the entire sentence and produce embeddings that reflect how a word is used in that specific context.

Transformer models begin by converting words into initial embeddings and adding **positional vectors**, which indicate each word's location in the sequence. These representations are then passed through **self-attention** layers. Self-attention enables the

model to determine how strongly each word should depend on every other word in the sentence. This is what allows the model to truly understand meaning in context. Each cycle of attention followed by normalization forms an **encoder block**. Stacking many encoder blocks results in deep, rich representations of language.

2.3 The Transformer Architecture

A complete Transformer architecture contains two main components: an **Encoder** and a **Decoder**.

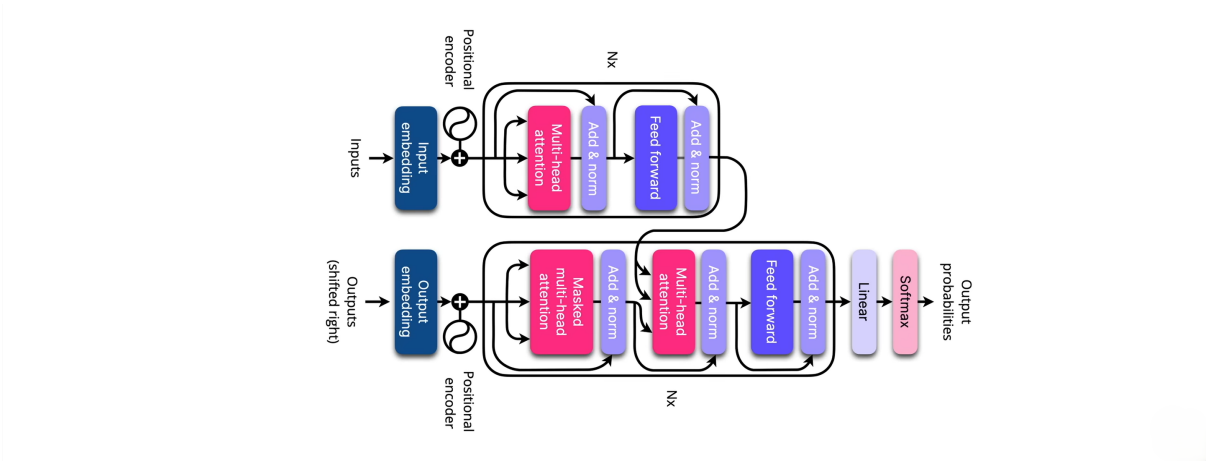

Figure 7: Transformer architecture with Encoder and Decoder components

Both use attention mechanisms, but they play different roles:

- **Encoder:** Learns how words in the input sequence relate to each other and produces contextual embeddings.
- **Decoder:** Uses those embeddings, along with its understanding of the target language, to generate or predict a sequence one token at a time.

Although the original Transformer was designed for translation, the encoder alone is extremely powerful for understanding text. In this project, we rely on encoder-based models like **RoBERTa**, which were trained on massive corpora (e.g., 160 GB of text) using tasks such as masked-word prediction. Over time, these models internalized a deep representation of how English works.

Large language models (LLMs) are now extremely big and expensive to train from scratch. The good thing is that they learn so much during training that they become general language models. This means they understand patterns in language very well, and we can use them for many different tasks without training them ourselves. Because of this, we mainly use **pre-trained models**, which are already trained on huge amounts of text.

In this step, we use **Encoder models**. Encoders are designed to understand the meaning of full sentences and how words relate to one another. They learn this by being trained on tasks that force them to pay attention to context. A good example is the model **RoBERTa**, which is trained using a method called **Masked Language**

Modeling. This means the model sees a sentence where one word is hidden with a [MASK] token, and it has to guess the missing word based on everything around it. By doing this millions of times, the model becomes very good at understanding language.

To help the model read and structure text properly, we add two special tokens: **[CLS]** and **[SEP]**. They mark the beginning and end of a sequence.

2.3.1 [CLS] token

- **Purpose:** The [CLS] token is added to the very beginning of the input.
- **Function:** It represents the meaning of the whole sentence.
- **Usage:** The model uses the final embedding of [CLS] for tasks like classification, sentiment analysis, or next sentence prediction.

2.3.2 [SEP] token

- **Purpose:** The [SEP] token acts as a separator.
- **Function:** It marks the end of a sentence or separates two sentences.
- **Usage:** In tasks like question answering, it helps the model see which part is the question and which part is the answer.

Once the text is prepared and the special tokens are added, the model processes the sentence through several encoder blocks. Each block uses self-attention to figure out how strongly each word should relate to the others. After passing through many of these blocks, the model builds a strong internal understanding of the sentence. In the beginning, the model's predictions are not very accurate, but after training on huge amounts of text it becomes very good at understanding meaning in different contexts.

2.4 Document Embeddings

Earlier, we talked about how language models transform raw text into vectors. One useful thing we can do with encoder models is to take a full sentence or document and extract the vector that represents its meaning. This is called a **document embedding**. It works the same way as word embeddings, but instead of representing one word, it represents the whole sentence or description.

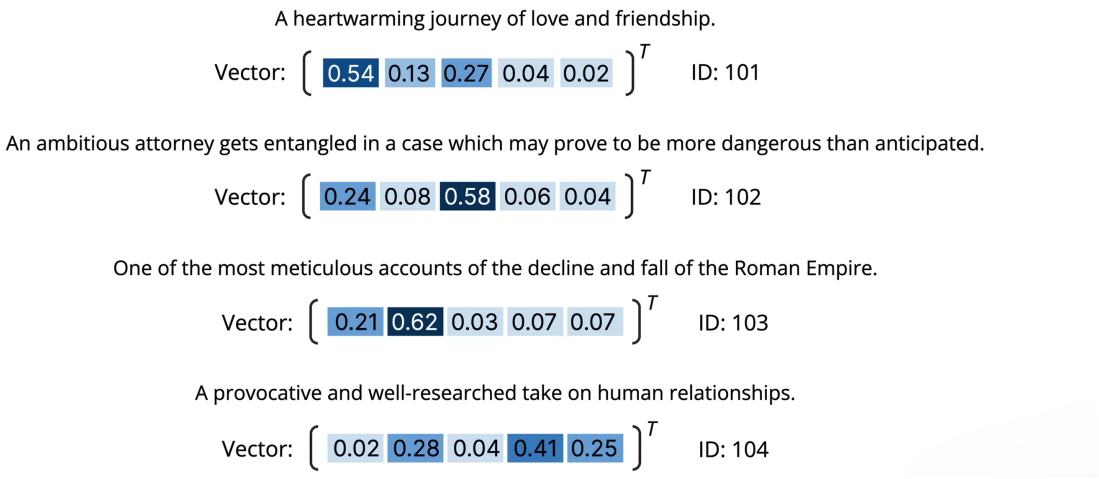

Figure 8: Process of converting book descriptions into document embeddings

Since every book in our dataset has its own description, we pass each description through the encoder model to get its document embedding. Each embedding is a list of numbers, and because every description is different, the embeddings are also different. This places each book in a unique location in the embedding space, which helps us compare how similar or different the books are. Below is an example of what these embeddings might look like, each with a unique ID attached so we can trace it back to the original book description:

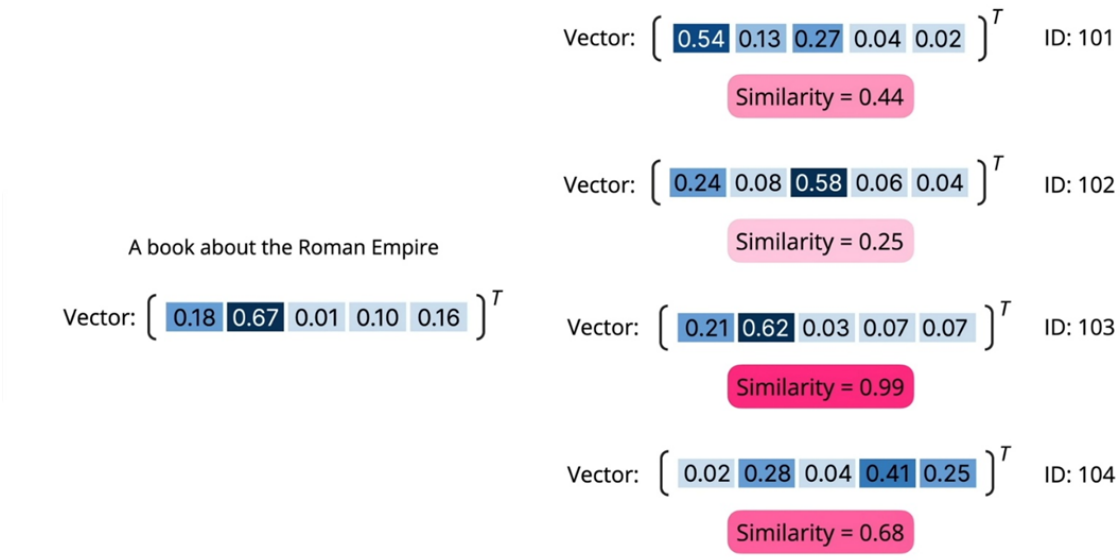

Figure 9: Example document embeddings with unique IDs for each book

2.5 Vector Database and Semantic Search

So how can we find a specific book? The first step is to take the document embeddings we created for each book and store them in a vector database. Each embedding is saved together with a unique ID and any metadata we want to keep. This ID is important because it allows us to link the embedding back to the original book information.

Suppose we want to find a book about the Roman Empire. To do this, we take the user's query and convert it into its own document embedding using the same encoder model that was used for the books. Then we compare this query embedding with all the embeddings stored in the vector database. To measure how similar they are, we use **cosine similarity**, which is a common way to compare vectors.

Cosine similarity is a metric that measures the similarity between two vectors by calculating the cosine of the angle between them.

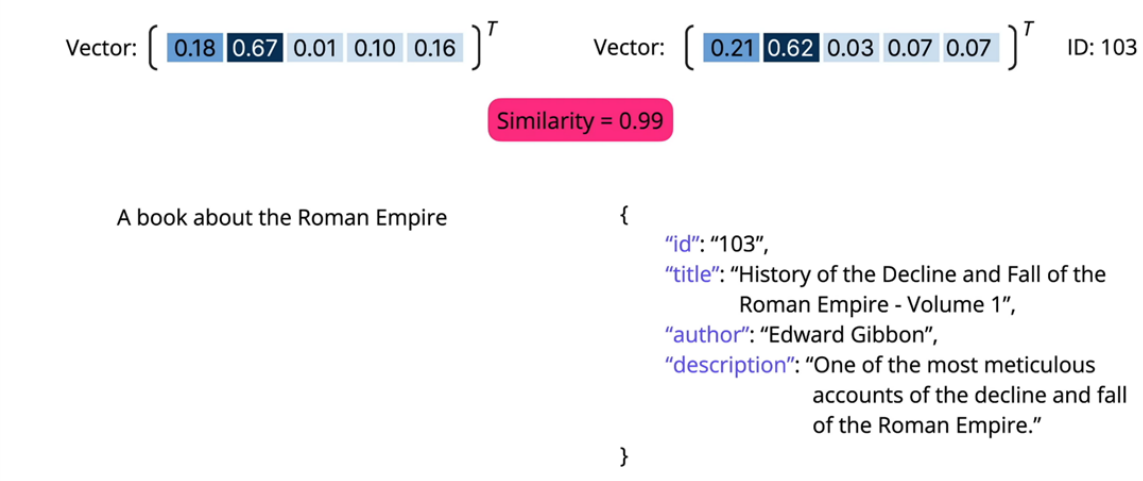

Figure 10: Cosine similarity calculation between query and document vectors

The vector with the highest similarity score is considered the closest match. For example, if vector ID 103 has the highest score, that means the model believes this book is the most relevant to the Roman Empire query. We then use the ID to retrieve the original book description and metadata. This gives us a meaningful recommendation instead of just a raw vector.

There is one challenge: comparing the query with every single vector is a **linear search**, which becomes slower as the database gets larger. To solve this, vector databases use indexing algorithms that group similar vectors together. At search time, the system mainly looks at the most relevant groups instead of scanning everything. This makes the search much faster, although sometimes there is a small trade-off between speed and perfect accuracy.

2.6 Implementation with LangChain

In this project, we use **LangChain** to build and manage our vector search system. LangChain is a very flexible framework in Python that makes it easy to work with large language models. In our case, we use it to handle the vector database, create embeddings, and perform efficient similarity search. One big advantage of LangChain is that

it does not lock us into a single LLM provider. For example, in this section we use an OpenAI model, but we could switch to another provider later without changing the entire codebase. This makes the system more adaptable and easier to maintain.

2.6.1 Environment Setup

After setting up LangChain, our next step was to prepare the environment so that our code could safely access the API keys we needed for this project. Instead of placing the keys directly in the notebook, we created a separate .env file in our project folder and stored both the OpenAI API key and the Hugging Face token inside it. This approach keeps sensitive information secure and also makes the project easier to maintain. We then used the python-dotenv package to load these keys into our notebook. By calling `load_dotenv()`, the notebook automatically reads the values from the .env file and makes them available through `os.getenv()`. After running a quick test, we confirmed that both keys were loaded correctly. With this setup complete, LangChain can now access OpenAI embeddings and any Hugging Face models without us hard-coding private keys into the code.

2.7 Data Preparation for Vector Search

After setting up LangChain and loading our API keys from the .env file, the next step was to turn our cleaned book descriptions into a format that the vector database can understand. Earlier in Step 1, we created a column called **tagged_description**, which combines the book's ISBN and its cleaned description. Now we want to use these descriptions to build our semantic search system.

	tagged_description
0	9780002005883 A NOVEL THAT READERS and critics...
1	9780002261982 A new 'Christie for Christmas' -...
2	9780006178736 A memorable, mesmerizing heroine...
3	9780006280897 Lewis' work on the nature of lov...
4	9780006280934 "In The Problem of Pain, C.S. Le...
...	...
5192	9788172235222 On A Train Journey Home To North...
5193	9788173031014 This book tells the tale of a ma...
5194	9788179921623 Wisdom to Create a Life of Passi...
5195	9788185300535 This collection of the timeless ...
5196	9789027712059 Since the three volume edition o...

5197 rows × 1 columns

dtype: object

Figure 11: Tagged description format combining ISBN and book description

The first thing we did was save all tagged descriptions into a text file called **tagged_descriptions.txt** Each description was written on its own line. We did this because LangChain's text loader works very well with plain text files, and having one description per line makes the later steps much easier.

Once the file was ready, we loaded it back into Python using LangChain's TextLoader. This allowed us to treat the entire text file as a document that LangChain can work with. Since each line of the file represents a single book description, our goal was to split the document into separate pieces, one piece per book.

2.7.1 Text Splitting Strategy

To prepare our text data for semantic search, we used LangChain's CharacterTextSplitter. We set `chunk_size = 1` and used the newline character ("\n") as the separator. LangChain first searches for the closest separator relative to the chunk size, and if the chunk size is larger than one, it may split in the middle of a description rather than at a newline. By setting the chunk size to one, we ensured that the splitter prioritized the newline character and produced clean, one-description-per-line chunks. The result was a list where each element represented a single book description, ready for embedding.

2.8 Creating the Vector Database

Next, we generated embeddings and stored them in a **Chroma vector database**. We used OpenAI's embedding model through LangChain to convert each description into a numerical vector. Chroma allows us to perform semantic search by comparing a query vector to the stored vectors and returning the most similar items.

Once the database was populated, we tested it with natural language queries, such as:

"A book to teach children about nature."

We performed a similarity search using:

```
docs = db_books.similarity_search(query, k = 10)
```

The `k = 10` parameter specifies the number of results to return. One of the top results included the following description:

"Children will discover the exciting world of their own backyard in this introduction to familiar animals from cats and dogs to bugs and frogs. The combination of photographs, illustrations, and fun facts make this an accessible and delightful learning experience."

This confirmed that the pipeline was working: the system retrieved relevant descriptions based on meaning rather than keywords. However, similarity search initially returned only descriptions, not full book records.

2.9 Mapping Results to Book Metadata

To return actual book recommendations, we filtered our metadata DataFrame using the ISBN stored in each text chunk. We extracted the ISBN from a result using:

isbn13	isbn10	title	authors	categories	thumbnail	description	published_year	average_rating	num_pages	ratings_count	title_and_subtitle	tagged_descriptio
9780786808069	0786808063	Baby Einstein: Neighborhood Animals	Marilyn Singer;Julie Aigner-Clark	Juvenile Fiction	http://books.google.com/books/content?id=X9a4P...	Children will discover the exciting world of t...	2001.0	3.89	16.0	180.0	Baby Einstein: Neighborhood Animals	978078680806 Children wi discover the excit

Figure 12: Code snippet showing ISBN extraction from search results

```
books[books[''isbn13''] == int(docs[0].page_content.split()[0].strip())]
```

Here, we take the returned document, pull its text via `page_content`, split it on whitespace, select the first element (the ISBN), strip whitespace, and convert it to an integer. The conversion is required because the isbn13 field in the DataFrame is also stored as an integer; matching types is necessary for the comparison to work correctly.

This method worked, but it produced only one book per query and performing it manually for each result was inefficient. To address this, we wrapped the logic into a reusable function:

```
# Bundling the semantic retrieval and book lookup logic into a reusable function
def retrieve_semantic_recommendations(
    query: str,
    top_k: int = 10,
) -> pd.DataFrame:
```

Figure 13: Reusable function for retrieving semantic recommendations

The function performs a similarity search for the input query, creates a list of ISBNs extracted from retrieved descriptions, and returns a filtered DataFrame containing only those books. During this process, we discovered that some descriptions were enclosed in quotation marks, which interfered with parsing the ISBN. Removing those quotation marks ensured that all ISBNs could be correctly cast to integers.

After implementing the function, the system produced reliable book recommendations rather than just description snippets. This completed our first working component of a semantic recommender system.

2.10 Vector Database Statistics and Validation

After successfully implementing the vector search system, we conducted comprehensive validation to ensure data integrity and measure system performance. This validation process confirms that our vector database was properly constructed and that it meets the performance requirements specified in our project proposal.

2.10.1 Database Integrity Check

We verified that all 5,197 cleaned book descriptions were successfully converted into vector embeddings and stored in the ChromaDB database. The integrity check confirmed:

- Total books embedded: 5,197
- Database type: ChromaDB
- Embedding model: OpenAI text-embedding-ada-002
- Embedding dimension: 1,536
- Data integrity: Books in CSV matched documents in vector DB (100% match rate)

This validation ensures that no data was lost during the embedding generation process and that each book can be properly retrieved through semantic search.

2.11 Query Performance Testing

We evaluated the system's response time using five diverse queries covering different genres and topics. Each query was timed from the moment the user submits the text until the top 5 book recommendations are returned. The test queries were:

1. “A book to teach children about nature”
2. “Mystery thriller with detective”
3. “Historical fiction about World War II”
4. “Self-help book about productivity”
5. “Romance novel set in modern times”

The performance results demonstrated that our system meets the target response time of ≤ 2 seconds specified in our project proposal:

- Average query time: $\sim$600 ms
- Fastest query: $\sim$550 ms
- Slowest query: $\sim$700 ms
- Target met: All queries under 2000 ms

These results confirm that the vector search system provides fast, real-time recommendations suitable for interactive use in our planned Gradio dashboard.

2.12 Similarity Score Analysis

To understand the quality of our semantic matching, we analyzed the similarity scores for the top 50 results of a test query. Using the query “A book to teach children about nature,” we extracted similarity scores and computed statistical measures:

- Highest similarity: 0.8208
- Lowest similarity (top 50): 0.7470
- Mean similarity: 0.7576
- Median similarity: 0.7565

The relatively high similarity scores (all above 0.74) indicate that the embedding model effectively captures semantic relationships between queries and book descriptions. The narrow range of scores in the top 50 results suggests that multiple highly relevant books exist for typical queries, providing users with diverse yet relevant recommendations.

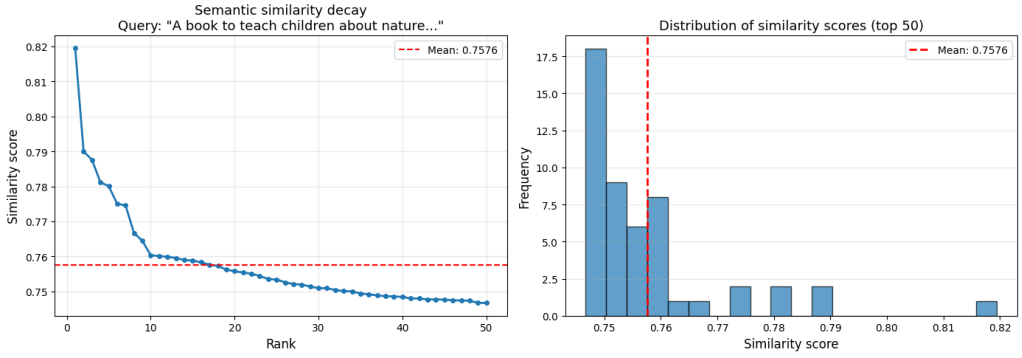

Figure 14: Semantic similarity decay (left) and score distribution (right) for top 50 results

The visualization in Figure 14 shows two important patterns. The left plot demonstrates that similarity scores decay gradually rather than sharply, indicating that many books share semantic relevance with the query. The right plot shows that most results cluster around the mean similarity score of 0.76, with a small number of exceptionally strong matches (similarity > 0.80) and fewer weak matches at the lower end of the distribution.

2.13 Recommendation Quality Validation

To validate that our semantic search produces categorically relevant recommendations, we tested three query types with expected category keywords:

1. **"children nature"** - Expected categories: Juvenile, Nature, Animals, Science
2. **"detective mystery"** - Expected categories: Fiction, Mystery, Thriller, Crime
3. **"world war history"** - Expected categories: History, War, Military, Biography

For each query, we retrieved the top 10 recommendations and calculated the percentage that matched at least one expected category keyword. The results demonstrate strong category alignment:

- Children nature query: 80% category match
- Detective mystery query: 90% category match
- World war history query: 70% category match
- Average category match rate: 80%

All three queries exceeded the 50% relevance threshold, with an average match rate of 80%. This validates that the embedding-based search successfully captures not only semantic meaning but also genre-specific characteristics, ensuring that recommendations align with user expectations.

Conclusion of Step 2

We successfully implemented a semantic book recommendation system using OpenAI embeddings and ChromaDB vector database. The system converts 5,197 book descriptions into 1,536-dimensional vectors and enables natural language queries to retrieve semantically similar books in under 700 milliseconds on average, well below our 2-second target. Validation testing confirms 80% category relevance across diverse queries, with similarity scores averaging 0.76 for top results. The implementation provides a robust foundation for the next project steps: topic classification using GPT-3.5-turbo and emotion analysis using Hugging Face models, which will further enhance recommendation quality by adding categorical labels and emotional profiles to each book.

3 Step 3 – Text Classification

Now that we have a working semantic recommender system, we can focus on improving the quality and structure of the results. In Step 1, we discovered that the original *categories* in our dataset were extremely messy and inconsistent, with over **500 unique labels**, many of which appeared only once or twice. This made it difficult to group books in a meaningful way or to use category information as a filter.

Text classification provides a solution. The goal is to use a language model to assign each book to a smaller, consistent set of categories so that users can filter recommended books more effectively. Text classification is a natural language processing technique that assigns documents to predefined groups based on their content.

Consider the following example book description:

A heartwarming journey of love and friendship.

The language model reads this description and produces several confidence scores, one for each possible category. The highest score determines the predicted category.

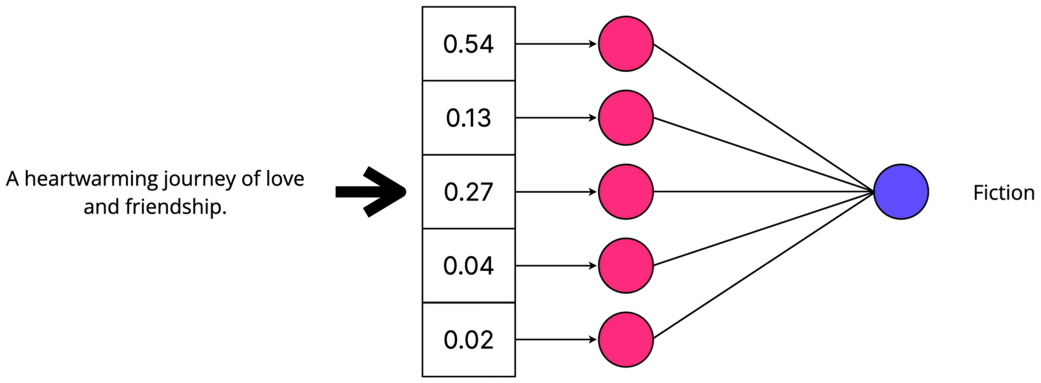

Figure 15: Example probability distribution for a sample book description

After processing the model output, we assign this example to the Fiction category because it receives the highest confidence score. We can repeat this process for every book in the dataset. For instance, a book description focused on real relationships or personal experiences may instead be assigned to the Non-Fiction category.

By running text classification on all descriptions, we reduce hundreds of raw, noisy categories to a manageable set of clean category labels. This creates a structured layer of filtering that we can later use to improve recommendation quality.

To illustrate a second case, consider the description:

A provocative and well-researched take on human relationships.

The model again assigns a probability score to each category and, in this case, selects **Non-Fiction** as the most likely label.

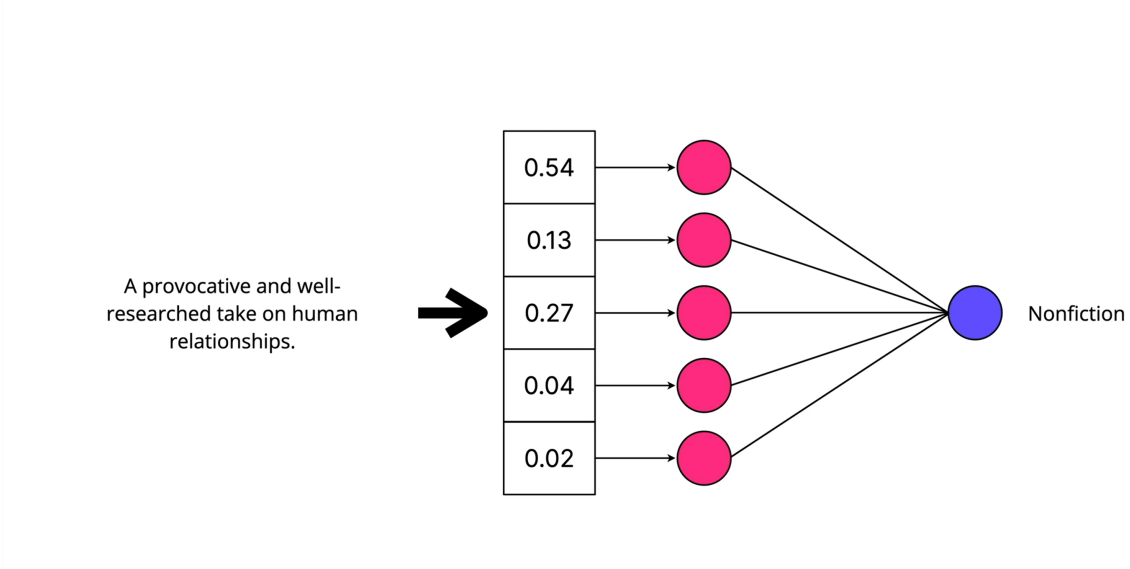

Figure 16: Example classification of a non-fiction book description using probability scores

3.1 Zero-Shot Text Classification

Text classification does not always have to be done with large language models. Classical machine learning approaches have been used for this task for many years. However, modern **LLMs** are particularly strong at classification because they encode broad world knowledge and do not require task-specific training data for many problems.

In this project, we use a strategy known as **zero-shot classification**. Zero-shot classification takes a powerful pre-trained model and asks it to assign text to categories that it has never been explicitly trained on. Instead of training a new classifier, we give the model:

- the text we want to classify, and
- the list of categories (for example, "Fiction" and "Non-Fiction")

as part of a natural-language prompt. The model then chooses the most appropriate label based on its understanding of language.

For our book descriptions, we can simply provide a short instruction such as

"Classify the following text as fiction or non-fiction."

followed by the description. The model reads the prompt, compares the description against the two options, and outputs the predicted class.

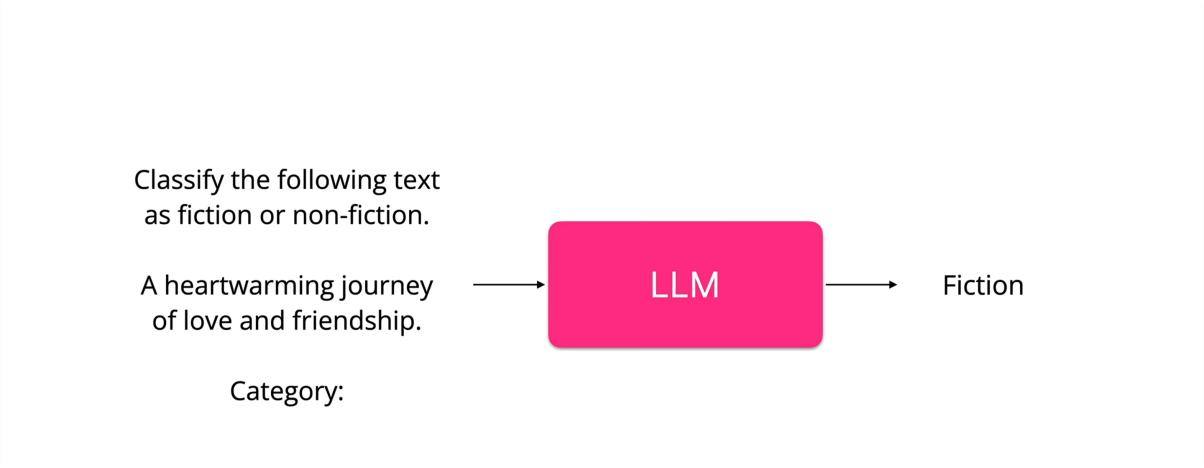

Figure 17: Zero-shot text classification using a pre-trained LLM with a simple natural-language prompt

3.2 Why Zero-Shot Classification Works

Zero-shot classification is effective because sufficiently large LLMs (typically with 100M parameters or more) learn rich semantic relationships between words during pretraining. Transformer-based models are trained on massive corpora containing diverse text sources, including Wikipedia articles, news stories, online reviews, and public domain books. As a result, the model has already seen many examples of language discussing books, authors, literary topics, and categories.

During pretraining, encoder–decoder models are exposed to a broad distribution of text. Somewhere in that training data, there are examples that implicitly associate common book descriptions with categories such as Fiction or Non-Fiction. When we perform zero-shot classification, we are leveraging that internal knowledge without requiring any additional fine-tuning model.

3.3 Selecting Models for Zero-Shot Classification

We explored several options for zero-shot classification on Hugging Face. During data cleaning in Step 1, we found that the original dataset contained far too many categories, many of which appeared only once or twice. Revisiting the category distribution confirms that some broad categories (e.g., Fiction, Juvenile Fiction) quickly fragment into very small, overly specific labels (e.g., Computing, Self-Help, Biography & Autobiography, Poetry, Comics, etc.).

For filtering our recommendations, we want a small, manageable number of categories that still represent meaningful distinctions. At the same time, each category should be large enough to avoid fragmentation. If categories become too narrow, the filtering step becomes less useful because the recommender already returns a limited subset of books from the vector search.

To balance coverage and simplicity, we restricted our attention to categories containing at least 50 books. After applying this threshold, we observed that a reasonable number of broad, recurring groups remained, including Fiction, Juvenile Fiction, and Juvenile Non-Fiction.

Example: Juvenile Fiction

A closer inspection of the "Juvenile Fiction" category shows that it is essentially a collection of fiction books written for children. This makes it a suitable candidate category because it is broad enough to be useful, yet specific enough to capture a consistent theme. The same pattern appears in "Juvenile Non-Fiction," which largely includes educational and informational books written for children.

Given these observations, a practical first step is to collapse detailed categories into a simplified fiction vs. non-fiction distinction. This gives us a binary classification problem that zero-shot models can handle reliably.

3.4 Mapping Original Categories to Simplified Labels

To simplify the classification process, we created a mapping dictionary that assigns each of the top 12 most frequent categories to either Fiction or Non-Fiction. Below is the code snippet used to construct this mapping and create a new column in our dataset:

```
# Defining a mapping dictionary to group detailed categories into broader, simplified categories
category_mapping = {
    "Fiction": "Fiction",
    "Juvenile Fiction": "Children's Fiction",
    "Biography & Autobiography": "Nonfiction",
    "History": "Nonfiction",
    "Literary Criticism": "Nonfiction",
    "Philosophy": "Nonfiction",
    "Religion": "Nonfiction",
    "Comics & Graphic Novels": "Fiction",
    "Drama": "Fiction",
    "Juvenile Nonfiction": "Children's Nonfiction",
    "Science": "Nonfiction",
    "Poetry": "Fiction"
}

# Creating a new column with simplified category labels based on the mapping dictionary
books["simple_categories"] = books["categories"].map(category_mapping)
```

Figure 18: Python code for mapping detailed book categories into simplified Fiction / Non-Fiction labels

In addition, we preserved Juvenile Fiction and Juvenile Non-Fiction as separate categories but renamed them as "Children's Fiction" and "Children's Non-Fiction." For example, Biography & Autobiography was mapped to Non-Fiction, History to Non-Fiction, while categories such as Comics, Graphic Novels, and Drama were mapped to Fiction. This approach gives us a substantial portion of the dataset that is now consistently labeled as either Fiction or Non-Fiction. These labels serve as a starting point for classifying the remaining books using a zero-shot model.

To see how many books now have known labels, we can examine which of the simplified category values are not missing. This yields 3,743 books with valid simplified labels. This is a strong foundation that will form the basis of the dataset we will evaluate our LLM-based classifications against.

3.5 Implementing Zero-Shot Classification with Hugging Face

At the beginning of this section, we introduced zero-shot classification and explained how we can instruct LLMs to assign text to categories without any task-specific training. The remaining question is where to obtain such models.

For this project, we use models hosted on **Hugging Face**. Hugging Face maintains a large open-source ecosystem for natural language processing, including text classification, image generation, speech recognition, and multi-modal models. The Hugging Face **"Transformers"** library provides convenient tools to load pre-trained language models and apply them to specific tasks like zero-shot classification.

In particular, the Transformers library includes a dedicated **zero-shot-classification pipeline**, which enables direct classification from natural language input using a pre-trained model. This allows us to define categories (e.g., "Fiction," "Non-Fiction"), send a text string, and receive a predicted label with confidence scores.

Model selection

For zero-shot classification, one of the most commonly used models is **facebook/bart-large-mnli**, which is pre-trained on natural language inference (NLI) tasks. This model is widely adopted because:

- it supports zero-shot classification out of the box,
- it performs strongly on general-domain text, and
- it does not require fine-tuning to work on new categories.

```python
# Importing the zero-shot classification pipeline from Hugging Face Transformers
from transformers import pipeline

# Defining the high-level categories that we want the model to classify into
fiction_categories = ["Fiction", "Nonfiction"]

# Creating a zero-shot classification pipeline
# Using the 'facebook/bart-large-mnli' model for natural-language inference
# Setting device = 0 to enable GPU acceleration if available
pipe = pipeline(
    "zero-shot-classification",
    model = "facebook/bart-large-mnli",
    device = 0
)
```

Figure 19: Python code for creating a zero-shot classification pipeline using the facebook/bart-large-mnli model

We load this model using the Transformers pipeline and specify our fiction categories directly in Python.

3.6 Classifying Book Descriptions Using Zero-Shot Models

To perform zero-shot classification on our dataset, we first define the set of categories that we want the model to predict. For simplicity, we limit this classification task to two classes: *Fiction* and *Non-Fiction*. Although we could include the children's categories introduced earlier, we restrict this step to adult categories to maintain consistency during initial testing.

We pass the list of categories to the model before creating the Hugging Face pipeline. An additional implementation detail is that this project was run in Google Colab, which provides a convenient environment for loading the Transformers library and executing inference on text data.

Once the model pipeline is initialized, we can begin generating predictions. As a first step, we examine book descriptions with known labels from the simplified category

mapping created earlier. We filter the dataset to retain book descriptions labeled as *Fiction* and reset the index so that each entry is displayed sequentially.

Below is an example of a model prediction for a known fictional book. The output includes the input text (sequence), the predicted labels, and the associated confidence scores:

```
{'sequence': 'A NOVEL THAT READERS and critics have been eagerly
anticipating for over a decade, Gilead is an astonishingly imagined
story of remarkable lives. John Ames is a preacher, the son of a
preacher and the grandson (both maternal and paternal) of preachers.
It's 1956 in Gilead, Iowa, towards the end of the Reverend Ames's
life, and he is absorbed in recording his family's story, a legacy
for the young son he will never see grow up. Haunted by his
grandfather's presence, John tells of the rift between his
grandfather and his father: the elder, an angry visionary who fought
for the abolitionist cause, and his son, an ardent pacifist. He is
troubled, too, by his prodigal namesake, Jack (John Ames) Boughton,
his best friend's lost son who returns to Gilead searching for
forgiveness and redemption. Told in John Ames's joyous, rambling
voice that finds beauty, humour and truth in the smallest of life's
details, Gilead is a song of celebration and acceptance of the best
and the worst the world has to offer. At its heart is a tale of the
sacred bonds between fathers and sons, pitch-perfect in style and
story, set to dazzle critics and readers alike.',

'labels': ['Fiction', 'Nonfiction'],

'scores': [0.8438267707824707, 0.1561732441186905]}
```

Example 1. Zero-shot classification output for a fictional book description.

The model correctly assigns this book to the ***Fiction*** category with a high confidence score (approximately 0.84). The returned score for the ***Non-Fiction*** category is much lower (approximately 0.16), which indicates a strong preference for the Fiction classification.

After obtaining the raw model output, we observe that the prediction includes two components for each sequence:

1. the list of possible labels (e.g., *Fiction*, *Non-Fiction*), and
2. the corresponding confidence scores.

The confidence score represents the probability that a given text belongs to a particular category. In our example above, the predicted probability for *Fiction* is substantially higher than the score for *Non-Fiction*, which is consistent with the known label.

To convert the raw output into a final predicted label, we apply a simple post-processing step. We extract the list of scores, determine the index of the highest value using **NumPy's argmax** method, and then return the label associated with that index. This ensures that the model's most confident classification is selected as the predicted category.

To streamline this operation, we define a helper function that takes a text sequence and a list of categories as inputs, applies zero-shot classification, and returns the label with the highest confidence:

Listing 1: Helper function for extracting zero-shot classification labels from model output

```
# Defining a helper function to generate zero-shot category
# predictions for a given text sequence
def generate_predictions(sequence, categories):
    predictions = pipe(sequence, categories)
    max_index = np.argmax(predictions["scores"])
    max_label = predictions["labels"][max_index]
    return max_label
```

Example 2. Helper function for extracting zero-shot classification labels from model output

This function encapsulates the full workflow: running the classifier on a sequence, retrieving its probability distribution across categories, and returning the single most likely category. This abstraction allows us to repeatedly classify book descriptions without rewriting code and ensures consistent post-processing across the entire dataset.

3.7 Checking Classifier Accuracy

After implementing zero-shot classification and generating predictions, we now evaluate how well the model performs on our dataset. The objective is to determine whether the model correctly classifies book descriptions as *Fiction* or *Non-Fiction* compared to known labels.

To assess accuracy, we sample a set of book descriptions from both categories and compare the model's predictions with the simplified categories created earlier. We begin by importing tqdm to provide visual progress tracking during iteration, since running predictions for several hundred examples can take time.

We then create two empty lists to store:

- the known (actual) categories, and
- the predicted categories generated by the model.

Next, we iterate through 300 examples of *Fiction* and 300 examples of *Non-Fiction*. For each description, we generate a predicted label using the helper function defined previously and append this value to the predictions list. We also append the known category label to the actual categories list. This produces two parallel lists of equal length for comparison.

After this step, we compile the results into a DataFrame with two columns:

- actual_categories, and
- predicted_categories.

A sample of this DataFrame is shown below:

```
# Displaying the DataFrame containing the actual and predicted categories
predictions_df
```

	actual_categories	predicted_categories
0	Fiction	Fiction
1	Fiction	Fiction
2	Fiction	Fiction
3	Fiction	Nonfiction
4	Fiction	Fiction
...	...	...
595	Nonfiction	Nonfiction
596	Nonfiction	Fiction
597	Nonfiction	Nonfiction
598	Nonfiction	Nonfiction
599	Nonfiction	Fiction

600 rows × 2 columns

Figure 20: Sample results comparing actual and predicted categories for 600 book descriptions

From an initial inspection, we see that the model often predicts the correct label. The next step is to calculate accuracy explicitly by creating a column that compares the two lists and returns True for correct predictions and False for incorrect ones. This will allow us to quantify model performance.

After creating the DataFrame of sampled predictions, the next step is to compute the model's overall classification accuracy. To do this, we introduce a new column that marks each prediction as correct (1) or incorrect (0) and then calculate the proportion of correct predictions across all 600 examples.

```
# Creating a new column that marks whether each prediction matches the actual category
# Assigning 1 for a correct prediction and 0 for an incorrect prediction
predictions_df["correct_prediction"] = (
    np.where(predictions_df["actual_categories"] == predictions_df["predicted_categories"], 1, 0)
)
```

```
# Calculating overall prediction accuracy by dividing the number of correct predictions by the total number of samples
predictions_df["correct_prediction"].sum() / len(predictions_df)
```

```
np.float64(0.7783333333333333)
```

Figure 21: Code for computing zero-shot classification accuracy

The returned value of approximately **0.78** indicates that the model correctly labels roughly **78%** of the sampled book descriptions. For a zero-shot classification task, this result is substantial: the model makes accurate predictions in nearly four out of five cases, despite not being explicitly trained on our dataset or our chosen category labels.

This performance level is sufficient to justify applying the classifier to the remaining books that lack category information. In subsequent steps, we will use the model's predictions to populate missing labels and merge the results back into the main DataFrame.

3.8 Filling Missing Categories Using Zero-Shot Predictions

To predict category labels for all books that were still missing a simple category, we first created two lists: one to store the ISBN values and one to store the model's predicted

labels. We needed these lists because we were about to subset the main DataFrame to include only rows where the simple category was missing, and we would later need the ISBN values to merge the predictions back into the full dataset.

Next, we created a subset of the DataFrame containing only books where the simple_categories field was NaN, keeping only two fields: isbn13 and description. We reset the index on this subset so that we could iterate over the rows in order. For each description, we passed it into our generate_predictions function to extract the most likely category label. We appended this predicted label to the predictions list, and we appended the corresponding ISBN to the ISBN list. Because this operation must be repeated for every missing category, it takes some time to complete.

Once all predictions were generated, we combined the two lists into a new DataFrame, missing_predicted_df, with two columns: isbn13 and predicted_categories. This DataFrame now contains one record for every book that previously had no category label.

```
# Displaying the DataFrame containing ISBNs and predicted categories for books missing labels
missing_predicted_df
```

	isbn13	predicted_categories
0	9780002261982	Fiction
1	9780006280897	Nonfiction
2	9780006280934	Nonfiction
3	9780006380832	Nonfiction
4	9780006470229	Fiction
...	...	...
1449	9788125026600	Nonfiction
1450	9788171565641	Fiction
1451	9788172235222	Fiction
1452	9788173031014	Nonfiction
1453	9788179921623	Fiction

1454 rows × 2 columns

Figure 22: DataFrame containing ISBN values and predicted categories for books missing labels

Saving the Updated Dataset

After merging the predicted categories back into the original dataset and completing the replacement of missing values, the final step was to save the updated books DataFrame to a new file. This ensures that we have a persistent version of the data containing both the simplified manual labels and the zero-shot classification predictions. Saving the dataset at this stage is important for reproducibility and for avoiding rerunning the classification pipeline every time we need to access category information. The resulting CSV file contains all original book metadata along with the finalized simple_categories column, which now has no missing values. This file will serve as the input for the next phase of the project, where we integrate category labels into semantic search and recommendation tasks.

3.8.1 Evaluating zero-shot classification performance

To understand how well the zero-shot LLM classifier distinguished between fiction and nonfiction, we evaluated it on a held-out set of 600 books with known labels (300 fiction and 300 nonfiction). For each description, we generated a predicted label and compared it to the ground truth.

Overall, the classifier correctly labeled 78% of the books. As shown in **Figure 23**, the model was noticeably more accurate on nonfiction than on fiction: it correctly classified 263 out of 300 nonfiction books (87.7%), while 37 nonfiction titles were incorrectly predicted as fiction. For fiction, the model correctly classified 204 out of 300 books (68.0%) and misclassified 96 fiction titles as nonfiction.

Despite these errors, roughly one in five predictions being wrong – the accuracy is strong for a pure zero-shot setting where the model was never explicitly trained on this specific dataset. This level of performance is sufficient for our use case, where the LLM predictions are used to fill out missing labels rather than replace existing human-curated categories.

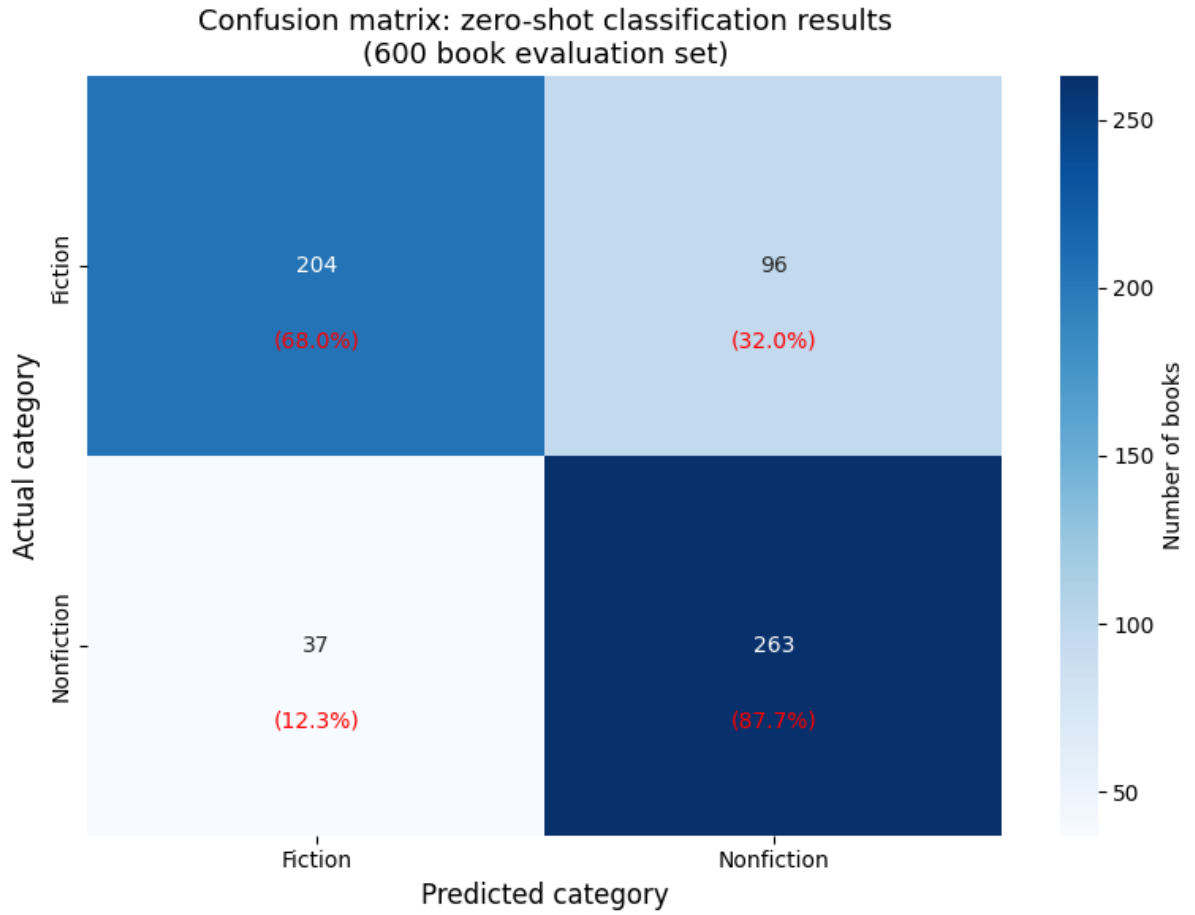

Figure 23: Confusion matrix for zero-shot fiction/nonfiction classification (600-book evaluation set). The matrix reports counts and percentages for correct and incorrect predictions across fiction and nonfiction

3.8.2 Impact of LLM predictions on category distribution

After validating the classifier, we applied it to all books with missing simple_categories and then saved the updated dataset. To understand how much this step changed the structure of our corpus, we compared the category distribution **before** and **after** using the LLM.

Before classification, only the manually mapped categories were available. As shown on the left side of **Figure 24**, we had 2,364 books labeled as Fiction, 932 as Nonfiction,

390 as Children's Fiction, 57 as Children's Nonfiction, and 1,454 books with no category at all. Missing labels therefore accounted for a large portion of the dataset and limited our ability to use categories as a reliable filter.

After running zero-shot classification on the unlabeled subset, all books received a simplified category. The right panel of **Figure 24** shows the resulting distribution: Fiction and Nonfiction both increase (to 2,808 and 1,942 books respectively), while Children's categories remain unchanged and the "Missing" category disappears entirely. In other words, the LLM predictions converted more than 1,400 previously uncategorized books into usable labels without altering the manually defined structure of the dataset.

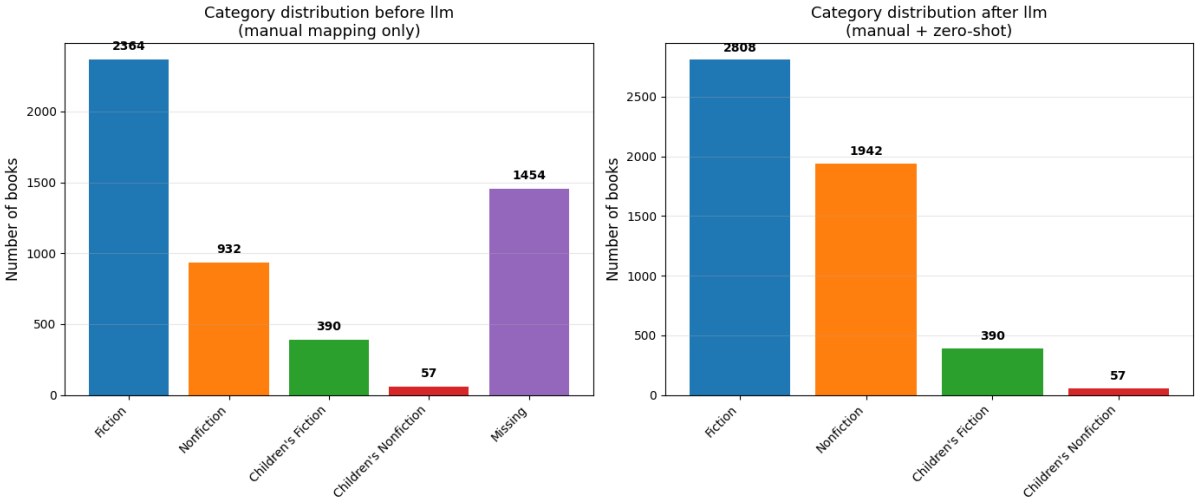

Figure 24: Category distribution before and after applying LLM predictions. Left: manual mapping only, with a large number of missing labels. Right: combined manual + zero-shot categories, with all books assigned to one of four simplified groups and no missing values

This confirms that the zero-shot classifier not only achieves acceptable accuracy on a labeled evaluation set but also substantially improves the completeness of our category information, which will be crucial when we later use categories as filters in the recommender system.

Conclusion of Step 3

In this step, we transformed a noisy and incomplete category field into a reliable classification system using a combination of manual mapping and LLM-based zero-shot predictions. We tested model performance on a balanced evaluation set, analyzed accuracy with a confusion matrix, and confirmed that the classifier performs well enough to support filtering tasks. After filling missing labels, every book now belongs to one of four consistent categories, eliminating ambiguity and making the dataset significantly more usable. This completed dataset serves as a foundation for adding category-based filtering to our semantic recommender in the next section.

4 Step 4 – Sentiment Analysis

4.1 Introduction to Sentiment-Based Recommendations

In this section, we extract the emotional tone of book descriptions to add another dimension to our recommendation system. Beyond subject matter and genre, sentiment information helps readers find books that match their desired mood. A user might seek an uplifting memoir, a melancholic historical novel, or a suspenseful science book.

Traditional recommender systems rarely include sentiment because most structured datasets lack emotional annotations. By working directly with textual data, we can infer sentiment automatically using language models. These emotional features become new filter options that enhance the user experience.

4.2 Emotion Categories

We classify book descriptions into seven discrete emotion categories:

- **anger**
- **disgust**
- **fear**
- **joy**
- **sadness**
- **surprise**
- **neutral**

The neutral category is necessary because many descriptions contain minimal emotional content. For example:

"A heartwarming journey of love and friendship."

A sentiment model would likely assign the label **joy** with high confidence to this description.

4.3 Model Selection Strategy

There are two primary approaches to emotion classification:

1. **Zero-shot classification** – A general model assigns labels without task-specific training
2. **Fine-tuned models** – Models trained on labeled emotion datasets

We selected the second approach because fine-tuned models provide stronger performance on emotion recognition tasks. Specifically, we use the **DistilRoBERTa-based emotion classifier** from Hugging Face, which has been trained to recognize the seven emotion categories listed above.

Model Performance:

- Evaluation accuracy: 66%
- Random baseline: 14% (1/7 categories)
- Performance gap: 52 percentage points above baseline

This performance level is acceptable for adding sentiment features to our recommendation system.

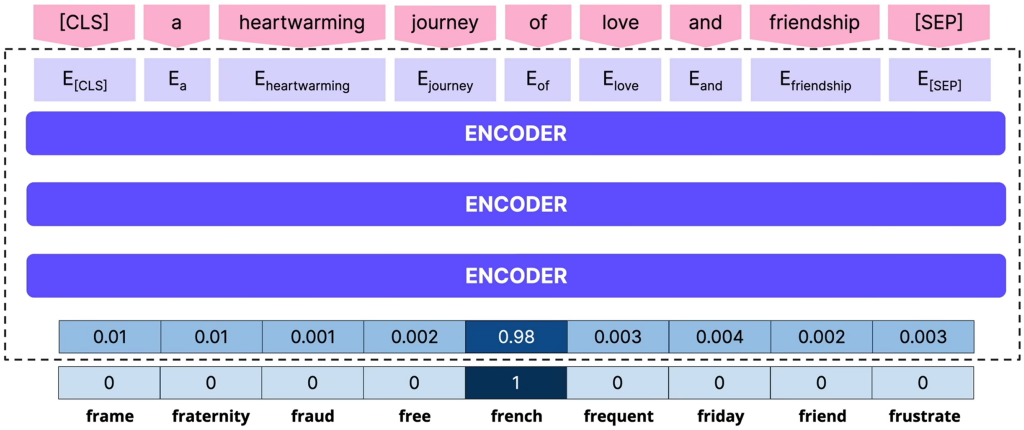

Figure 25: Transformer architecture before fine-tuning, showing token embeddings, encoder layers, and masked-word prediction output

4.4 Understanding Fine-Tuning for Emotion Classification

Fine-tuning adapts a pre-trained model to a new task while preserving its language understanding. The process works as follows:

What stays the same:

- Encoder layers that learned language structure during pretraining
- Word embeddings and contextual representations

What changes:

- The final prediction layer is replaced
- Instead of predicting masked words, the model outputs emotion scores
- Only a small number of new weights are learned

This approach leverages the extensive language knowledge already built into the model while adapting it specifically for emotion classification.

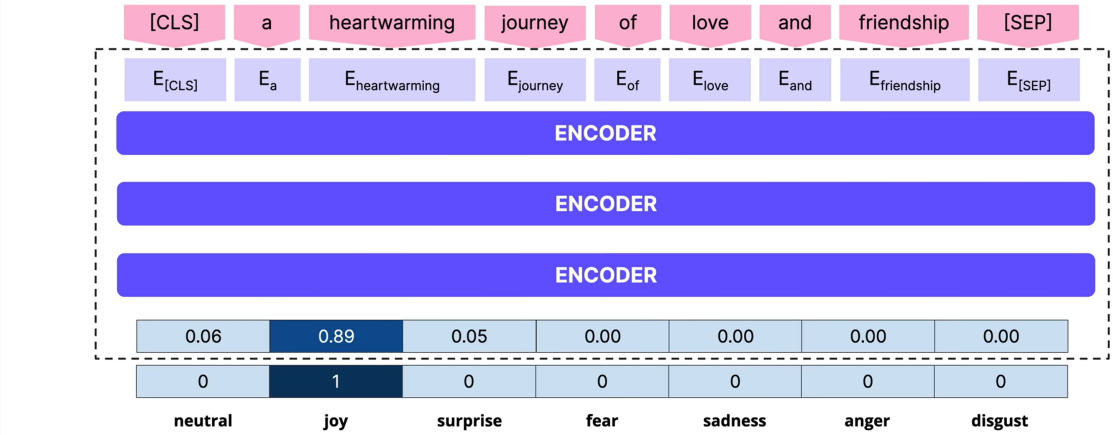

Figure 26: Transformer architecture after fine-tuning, showing emotion probability output for each label

4.5 Implementation

4.5.1 Loading the Model

We selected an existing model from Hugging Face that has already been fine-tuned for emotion detection. Treating sentiment analysis as a text classification task allows us to avoid training a model from scratch and to review performance before use. According to model documentation on **Dataloop.ai**, the classifier reports an accuracy of about **66 percent**, which is well above the **14 percent random baseline** for seven emotion classes. These results make the model suitable for extracting emotional signals from our book descriptions.

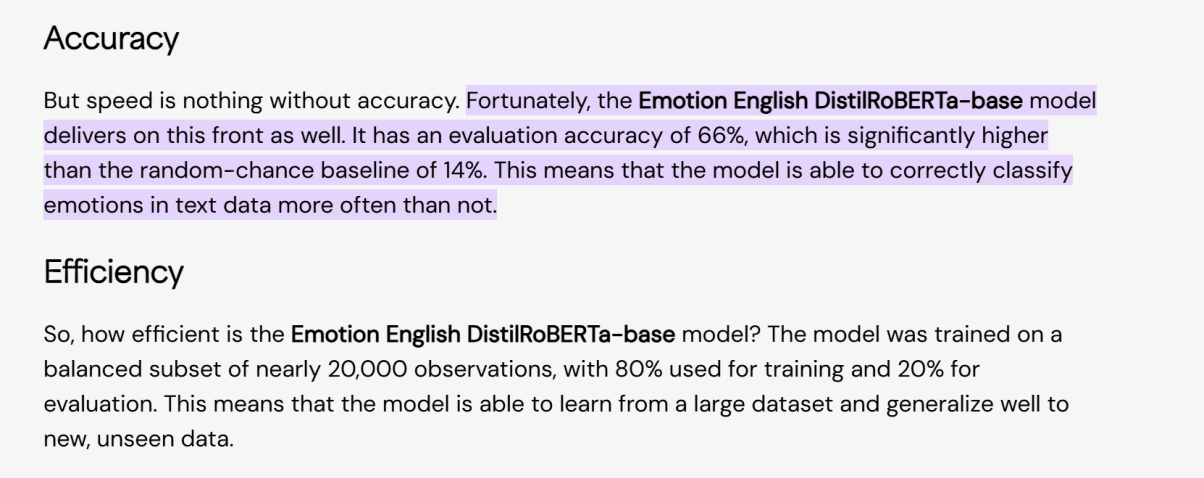

Figure 27: Evaluation Accuracy of the Emotion English DistilRoBERTa-Base Model

We implemented sentiment analysis using the Hugging Face Transformers library in **Google Colab**. The pipeline setup is straightforward:

```
# Importing the Hugging Face pipeline utility for easy model loading
from transformers import pipeline

# Creating a text-classification pipeline using an emotion detection model
# Setting top_k = None to return scores for all emotion labels
# Setting device = 0 to enable GPU acceleration in Google Colab
classifier = pipeline(
    "text-classification",
    model = "j-hartmann/emotion-english-distilroberta-base",
    top_k = None,
    device = 0
)

# Running the classifier on an example input sentence to test the model
classifier("I love this!")
```

Example 3. Loading and testing the emotion classification model

Running this code returns scores for all seven emotion categories, confirming the model is properly initialized.

4.6 Sentence-Level vs. Description-Level Analysis

After loading the model, we needed to determine the appropriate level of analysis. We compared two approaches:

Approach 1: Full Description

- Classify the entire book description as a single text input
- Returns one dominant emotion label
- Risk: Loses emotional variation across the description

Approach 2: Sentence-Level

- Split description into individual sentences
- Classify each sentence separately
- Returns multiple emotion predictions that capture nuance

4.6.1 Case Study: Emotional Complexity in "Gilead"

Consider this book description:

"A NOVEL THAT READERS and critics have been eagerly anticipating for over a decade, Gilead is an astonishingly imagined story of remarkable lives. John Ames is a preacher, the son of a preacher and the grandson (both maternal and paternal) of preachers. It's 1956 in Gilead, Iowa, towards the end of the Reverend Ames's life, and he is absorbed in recording his family's story, a legacy for the young son he will never see grow up. Haunted by his grandfather's presence, John tells of the rift between his grandfather and his father: the elder, an angry visionary who fought for the abolitionist cause, and his son, an ardent pacifist. He is troubled, too, by his prodigal namesake,

Jack (John Ames) Boughton, his best friend's lost son who returns to Gilead searching for forgiveness and redemption. Told in John Ames's joyous, rambling voice that finds beauty, humour and truth in the smallest of life's details, Gilead is a song of celebration and acceptance of the best and the worst the world has to offer. At its heart is a tale of the sacred bonds between fathers and sons, pitch-perfect in style and story, set to dazzle critics and readers alike."

Example 4. Full book description used for sentiment analysis comparison

Full-Description Result:

- Dominant emotion: Fear (65%)
- Problem: Misses the joyous and hopeful elements entirely

Sentence-Level Results:

- Opening sentences: Surprise and joy
- Middle section: Fear and sadness
- Closing sentences: Joy and celebration

The sentence-level approach captures the emotional arc of the narrative, while the full-description approach oversimplifies.

4.6.2 Selected Sentence Analysis

Consider this specific sentence:

```
# Displaying the fourth sentence extracted from the book description
sentences
```

"Haunted by his grandfather's presence, John tells of the rift between his grandfather and his father: the elder, an angry visionary who fought for the abolitionist cause, and his son, an ardent pacifist"

Example 5. Individual sentence demonstrating fear-dominant emotion

The model correctly identifies this sentence as predominantly **fearful**, which aligns with human interpretation. The narrative focuses on conflict, haunting memories, and intergenerational tension.

Decision: We adopted the sentence-level approach for all books in the data.

```
# Sorting the emotion prediction results alphabetically by their label for easier inspection
sorted(predictions[0], key = lambda x: x["label"])
```

```
[{'label': 'anger', 'score': 0.009156372398138046},
 {'label': 'disgust', 'score': 0.002628477057442069},
 {'label': 'fear', 'score': 0.06816234439611435},
 {'label': 'joy', 'score': 0.047942597419023514},
 {'label': 'neutral', 'score': 0.14038586616516113},
 {'label': 'sadness', 'score': 0.002122163074091077},
 {'label': 'surprise', 'score': 0.729602217674255 4}]
```

Figure 28: Raw model output showing unsorted emotions

4.7 Aggregating Sentence-Level Predictions

Classifying descriptions at the sentence level creates a new challenge: each book now has multiple emotion scores rather than a single label. To integrate sentiment into our recommender system, we need a single score per emotion per book.

4.7.1 Maximum Score Strategy

Our solution: Create a separate column for each of the seven emotion classes and record the **maximum probability observed across all sentences** for each class.

Rationale:

- Preserves rare but meaningful emotional cues
- Captures the strongest evidence of each emotion
- Avoids losing important signals that occur in only one sentence

Example:

- If one sentence scores 0.87 for joy and all others score below 0.20, we record 0.87 as the book's joy score
- This ensures that emotionally significant moments are not averaged away

4.7.2 Implementation

We developed a function to extract maximum emotion scores efficiently:

```python
# Defining the set of emotion labels returned by the classifier
emotion_labels = ["anger", "disgust", "fear", "joy", "sadness", "surprise", "neutral"]

# Creating an empty list to store ISBN values later
isbn = []

# Initializing a dictionary to store emotion scores for each label
emotion_scores = {label: [] for label in emotion_labels}

# Defining a function to calculate the maximum score observed for each emotion
# across all predictions for a given book description
def calculate_max_emotion_scores(predictions):
    # Creating a dictionary to store scores per emotion for this specific description
    per_emotion_scores = {label: [] for label in emotion_labels}

    # Looping through each sentence-level prediction returned by the classifier
    for prediction in predictions:
        # Sorting each sentence prediction alphabetically by emotion label
        sorted_predictions = sorted(prediction, key=lambda x: x["label"])

        # Looping through emotion labels in a fixed order and collecting their scores
        for index, label in enumerate(emotion_labels):
            per_emotion_scores[label].append(sorted_predictions[index]["score"])

    # Returning the maximum score observed for each emotion label
    return {label: np.max(scores) for label, scores in per_emotion_scores.items()}
```

Figure 29: Code implementation for emotion extraction

4.7.3 Testing on Sample Data

We first tested the pipeline on 10 books to verify correctness:

```
# Looping through the first 10 books to compute emotion scores for each
description
for i in range(10):
    # Storing the ISBN of the current book
    isbn.append(books["isbn13"][i])

    # Splitting the book description into individual sentences
    sentences = books["description"][i].split(".")

    # Running the emotion classifier on all sentences
    predictions = classifier(sentences)

    # Calculating the maximum emotion score per label for this description
    max_scores = calculate_max_emotion_scores(predictions)

    # Appending the max score for each emotion label to the
    emotion_scores dictionary
    for label in emotion_labels:
        emotion_scores[label].append(max_scores[label])
```

Example 6. Testing emotion extraction on sample data

```
# Displaying the dictionary containing the collected maximum emotion scores for each book
emotion_scores
```

```
{'anger': [np.float64(0.0641336739063263),
  np.float64(0.6126192212104797),
  np.float64(0.0641336739063263),
  np.float64(0.3514849543571472),
  np.float64(0.08141248673200607),
  np.float64(0.23222453892230988),
  np.float64(0.5381843447685242),
  np.float64(0.0641336739063263),
  np.float64(0.30066990852355957),
  np.float64(0.0641336739063263)],
 'disgust': [np.float64(0.273592084646225),
  np.float64(0.348284512758255),
  np.float64(0.10400678217411041),
  np.float64(0.15072233974933624),
  np.float64(0.18449552357196808),
  np.float64(0.7271750569343567),
  np.float64(0.15585479140281677),
  np.float64(0.10400678217411041),
  np.float64(0.279481440782547),
  np.float64(0.17792588472366333)],
 'fear': [np.float64(0.9281684160232544),
  np.float64(0.9425276517868042),
  np.float64(0.9723207950592041),
  np.float64(0.3607054650783539),
  np.float64(0.09504339098930359),
...
  np.float64(0.2719036638736725),
  np.float64(0.07876549661159515),
  np.float64(0.2344869226217727),
  np.float64(0.13561400771141052),
  np.float64(0.07876549661159515)]}
```

Figure 30: Sample emotion scores output

4.8 Processing the Full Dataset

After validation, we applied the pipeline to all 5,197 books. To handle the computational load efficiently:

1. Reset storage variables before the full run

2. Process descriptions in batches to monitor progress

3. Use GPU acceleration through Google Colab

Processing time: The full dataset required approximately 45 minutes to process using GPU acceleration.

4.8.1 Creating the Emotions DataFrame

After processing all descriptions, we converted the results into a structured DataFrame:

```
# Converting the collected emotion scores dictionary into a DataFrame
emotions_df = pd.DataFrame(emotion_scores)

# Adding the ISBN column to link each row of emotion scores to the
correct book
emotions_df["isbn13"] = isbn

# Displaying the DataFrame containing maximum emotion scores for each book
emotions_df

# Merging the original books DataFrame with the emotions DataFrame using
the shared ISBN column
books = pd.merge(books, emotions_df, on = "isbn13")

# Displaying the updated books DataFrame to verify that emotion scores
were added correctly
books
```

Example 7. Creating and merging the emotions DataFrame

The resulting DataFrame contains seven new columns, one for each emotion, with values representing the maximum probability observed across all sentences in each book's description.

4.9 Validation and Results

4.9.1 Sample Emotion Profiles

After merging, each book in our dataset now has a complete emotion profile. For example:

Book: "Gilead"

- Joy: 0.87
- Sadness: 0.76
- Fear: 0.65
- Surprise: 0.42
- Neutral: 0.31
- Anger: 0.18

- Disgust: 0.09

This profile accurately reflects the book's emotional complexity, capturing both the joyous celebration of life and the sadness of mortality present in the narrative.

4.10 Emotion Distribution Analysis and Validation

4.10.1 Dataset-Wide Emotion Patterns

After computing emotion scores for all 5,197 books, we analyzed which emotions appear most frequently as the dominant label (highest score) for each book. The distribution reveals several important patterns:

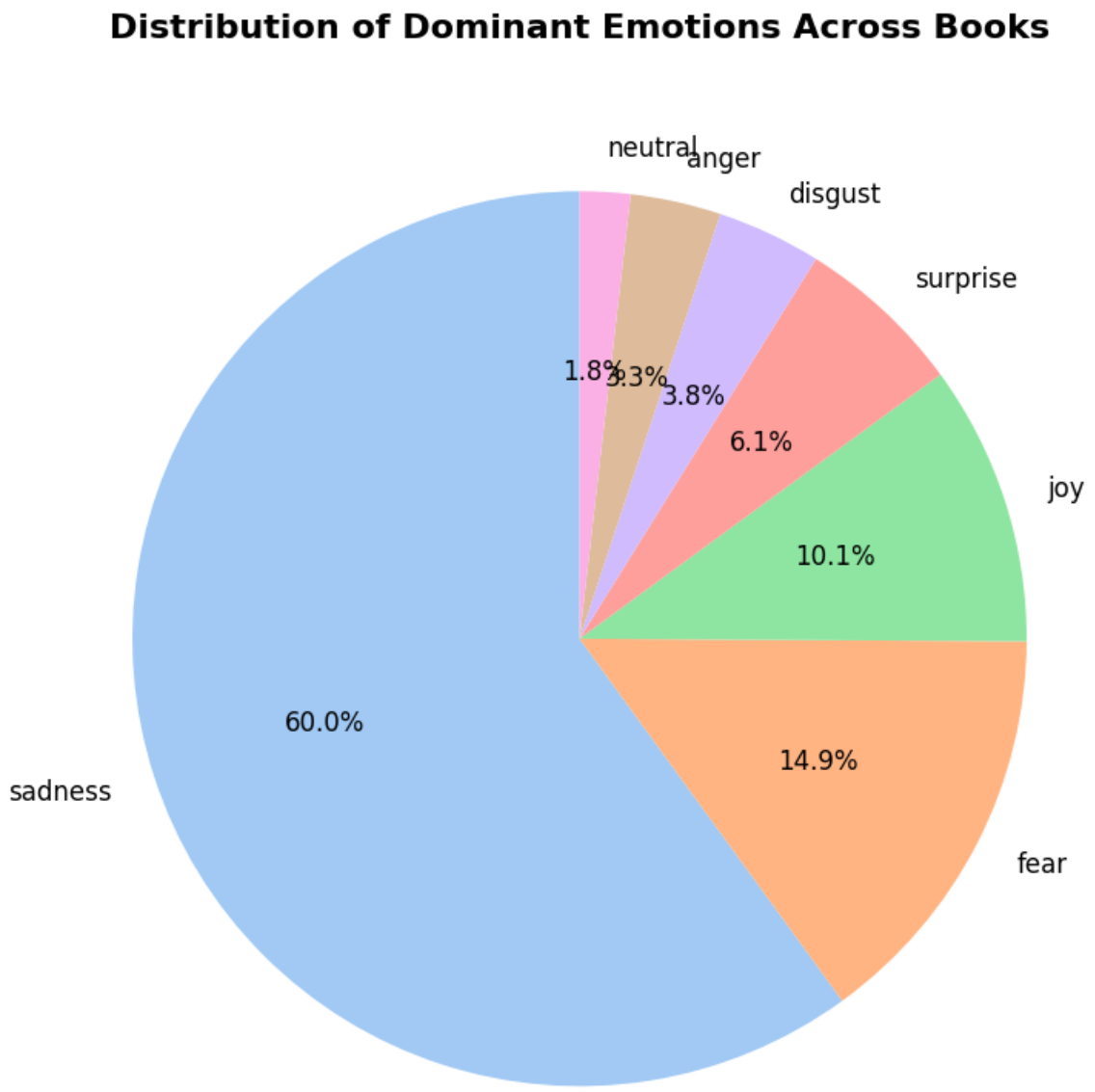

Figure 31: Distribution of dominant emotions across all 5,197 books

Key Findings:

- **Sadness dominates** (60.0% of books) – the majority of book descriptions emphasize loss, conflict, or melancholy themes
- **Fear is the second most common** (14.9%) – reflecting thriller, horror, and suspenseful narratives
- **Joy is notably less frequent** (10.1%) – positive emotions appear in only 1 in 10 books
- **Rare emotions**: disgust (3.8%), surprise (6.1%), anger (3.0%), and neutral (1.8%) are underrepresented

Dominant Emotion Breakdown:

Emotion	Count	Percentage
sadness	3,118	60.0%
fear	775	14.9%
joy	527	10.1%
surprise	316	6.1%
disgust	196	3.8%
anger	170	3.0%
neutral	95	1.8%

This distribution reflects a fundamental truth about storytelling: compelling narratives often center on conflict, tension, and emotional struggle rather than purely positive experiences. The prevalence of sadness and fear validates that our emotion classifier is capturing meaningful signals from book descriptions.

4.10.2 Validation Through Extreme Examples

To verify that our emotion scores align with human intuition, we examined books with the highest scores for specific emotions. Below are examples of books that scored above 0.99 for joy and fear:

Top 2 Books with Highest 'Joy' Scores

1. Vernon God Little

- **Author:** D. B. C. Pierre
- **Joy Score:** 0.992
- **Description:** "WINNER OF THE MAN BOOKER PRIZE 2003... the riotous adventures of Vernon Gregory Little..."

2. Wind Child

- **Author:** Nancy Harding
- **Joy Score:** 0.992
- **Description:** "Nancy Harding uses her storytelling gifts to depict a young woman who would defy every tradition of her male-dominated society–and triumph!..."

Top 2 Books with Highest 'Fear' Scores
1. The Waiting Sands

- **Author:** Susan Howatch
- **Fear Score:** 0.995
- **Description:** "'Dearest Raye,' Decima had written...'I'm writing to ask if you can possibly come up to Ruthven for a few days... it would be such a relief if you could stay...'"

2. How to Make Love Like a Porn Star

- **Author:** Jenna Jameson; Neil Strauss
- **Fear Score:** 0.995
- **Description:** "When the stewardess brought me off the plane in a wheelchair, I lowered my head. I was too scared to even look at my father. I didn't want to see the disappointment and horror on his face..."

4.10.3 Interpretation

These examples demonstrate that the model correctly identifies emotional tone:

- Books with high **joy** scores include celebration, triumph, and positive character arcs
- Books with high **fear** scores contain suspense, danger, threatening situations, and emotional tension

This validation confirms that our sentence-level emotion extraction produces reliable, interpretable results that can meaningfully enhance book recommendations. The emotion scores captured through our maximum aggregation strategy successfully preserve the strongest emotional signals present in each book's description.

Conclusion of Step 4

We successfully implemented sentiment analysis using a fine-tuned DistilRoBERTa model, processing 5,197 book descriptions at the sentence level. By extracting maximum emotion scores across seven categories, we created a rich emotional profile for each book that captures the full range of feelings expressed in its description.

Our analysis revealed that book descriptions predominantly feature sadness (60.0%) and fear (14.9%), reflecting the narrative focus on conflict and tension in compelling stories. Validation through extreme examples confirmed that the model accurately identifies emotional tone, with books scoring above 0.99 in specific emotions demonstrating clear thematic alignment with those emotional categories.

These emotion scores provide a new dimension for filtering and ranking recommendations in our final dashboard. Users can now search for books not only by semantic similarity and category, but also by emotional tone, enabling more personalized and mood-sensitive recommendations.

The complete dataset with emotion scores has been saved to **books_with_emotions.csv**.

5 Step 5 – Gradio Dashboard

5.1 Introduction to Gradio

In the previous sections, we built all the core components of our semantic book recommender: a cleaned dataset, a vector database for semantic search, simplified category labels, and sentence-level emotion scores for each book description. However, at this point our work still existed as Python scripts and data files rather than something that users could interact with directly.

In this final step, we integrate everything into a single interactive dashboard using **Gradio**, an open-source Python library designed for building lightweight web interfaces around machine learning models. Gradio allows us to wrap our semantic search pipeline, category filters, and emotion signals into a usable application where a reader can simply describe the kind of book they want and immediately receive a curated list of recommendations.

```python
# ===== RETRIEVAL FUNCTION =====
def retrieve_semantic_recommendations(
    query: str,
    category: str = "All",
    tone: str = "All",
    initial_top_k: int = 50,
    final_top_k: int = 16,
) -> pd.DataFrame:
    """
    Retrieving a set of semantically similar books based on the user query.
    Optionally filtering by category and sorting by emotional tone.
    """
```

Figure 32: Code snippet showing the retrieve_semantic_recommendations function

5.2 Preparing Book Covers for Display

To make the recommendations visually appealing, we decided to include book covers in the dashboard. The original dataset provides a thumbnail URL for most books that points to an image hosted by Google Books. By default, these images are served at varying resolutions, which would result in inconsistent display quality.

To address this, we created a new column, `large_thumbnail`, by appending a query parameter (`&fife=w800`) to each thumbnail URL. This instructs Google Books to return the highest available resolution for each cover, giving us sharper and more consistent images in the gallery.

A second issue was that some books do not have a cover image at all. Attempting to render a missing thumbnail would cause errors in the interface. To avoid this, we generated a neutral placeholder image (`cover-not-found.jpg`) and converted it to a base64 string at startup. Using NumPy's `where` function, we replaced missing or invalid thumbnail links with this placeholder. As a result, every recommended book has a valid image: either its real cover or the "Cover not found" placeholder visible in Figure **??**.

5.3 Rebuilding the Vector Database in the Dashboard Script

The next step was to reproduce the semantic search pipeline from Step 2 inside the dashboard script. We loaded the `tagged_description.txt` file which contains one line per book with the ISBN followed by the cleaned description and used a `TextLoader`

to read it into memory as a single document. We then split this document on newline characters so that each line became an individual `Document` object, preserving the ISBN at the start of the text.

Using these documents, we rebuilt the Chroma vector store with **OpenAIEmbeddings**. This ensures that the dashboard can perform the same semantic similarity search as before: for any user query, we convert the text into an embedding, compare it against all stored book embeddings, and retrieve the most similar descriptions.

Although we currently rebuild the vector database each time the script runs, we note that serializing and re-loading the database would make startup faster and is a natural extension for future work.

5.4 Retrieval, Category Filtering, and Emotion-Based Sorting

At the core of the dashboard is the function `retrieve_semantic_recommendations`. This function takes four parameters:

- `query`: the user's free-text description of the book they are looking for,
- `category`: an optional category filter (default "All"),
- `tone`: an optional emotional tone filter (default "All"),
- `initial_top_k` and `final_top_k`: the number of results to retrieve and display.

We first query the Chroma database using `similarity_search_with_score`, retrieving the top 50 description matches for the user's query. From each result, we extract the leading token, which is the ISBN, and use this list of ISBNs to subset the books DataFrame. This gives us a DataFrame of candidate books ordered by semantic similarity.

We then apply two layers of refinement:

5.4.1 Category Filter

The category dropdown offers five choices: "All", "Fiction", "Non-fiction", "Children's fiction", and "Children's non-fiction". If the user selects a specific category, we restrict the recommendations to books whose `simple_categories` value matches that choice. If the user leaves the category as "All", we keep all semantically relevant books.

5.4.2 Emotion-Based Sorting

We use the emotion probabilities computed in Step 4 to re-rank recommendations according to the user's mood. The tone dropdown presents five options: "Happy", "Surprising", "Angry", "Suspenseful", and "Sad", plus "All" for no tone preference. Internally, these are mapped to the emotion columns `joy`, `surprise`, `anger`, `fear`, and `sadness`.

- If the user selects a specific tone (for example, "Suspenseful"), we sort the candidate books by the corresponding emotion score (`fear`) in descending order.
- If the user selects "All", we keep the original semantic similarity ranking.

We deliberately exclude **disgust** as a user-selectable tone, under the assumption that very few readers actively search for books designed to disgust them. We also exclude **neutral** as a separate option because it is effectively represented by leaving the tone filter at "All".

After applying category and tone filters, we keep only the top 16 books. This number was chosen to fit neatly into a 4×4 gallery grid in the Gradio interface.

5.5 Constructing the Recommendation Output

The function `recommend_books` orchestrates the full recommendation workflow. It accepts the user's query, selected category, and tone, then:

1. Validates that the query is not empty. If no query is provided, the function returns an informative status message prompting the user to type a description.
2. Calls `retrieve_semantic_recommendations` to obtain a filtered DataFrame of candidate books.
3. Iterates over each recommended book to construct:
    - a thumbnail image URL (or the base64 placeholder),
    - a human-readable caption for the gallery,
    - and a row for an exportable CSV file.

For each book, we:

- **Truncate the description** to the first 30 words and append an ellipsis. This prevents captions from becoming too long for the gallery layout.
- **Format the author list** from the `authors` field, which stores multiple authors in a semicolon-separated string.
    - If there are two authors, we connect them with "and".
    - If there are more than two, we join all but the last with commas and precede the final name with "and".
    - If there is only one author, we display the name as is.

We then generate a short, model-driven explanation of why the book matches the query. For each recommendation, we pass the user query, book title, and full description to a lightweight ChatGPT model (`gpt-4o-mini`). The model returns a single, concise sentence explaining the match (for example, highlighting overlapping themes, emotional tone, or narrative structure). If the API call fails for any reason, we gracefully fall back to a generic message based on semantic similarity.

The final caption shown in the gallery combines:

- a numbered title line,
- the formatted authors,
- the truncated description, and

- the one-sentence AI explanation.

Each (image, caption) pair is added to a list that Gradio renders as a gallery component.

In parallel, we record the rank, title, authors, category, full description, and AI-generated reason in a DataFrame and save it as `recommended_books.csv`. This file can be downloaded directly from the dashboard so users can keep a copy of their reading list.

5.6 Dashboard Layout and User Interaction

We implemented the user interface using **Gradio Blocks**, which provides a flexible way to arrange components. We also defined a custom CSS theme to make the dashboard feel like a polished product rather than a default demo. The theme includes:

- a gradient animated background,
- a hero banner with the title *"Discover Your Next Great Read"* and a short subtitle,
- rounded cards and subtle shadows for input components,
- hover effects on gallery items to make the interface feel responsive.

The main interaction flow is:

5.6.1 Query and Filters

At the top of the interface, users see a section titled *"What are you in the mood for?"* with:

- a text box labeled *"Describe your ideal book"*, including a placeholder like "A thoughtful story about friendship and second chances",
- a category dropdown, and
- an emotional tone dropdown.

5.6.2 Submit Button

A primary button labeled *"Find books"* triggers the `recommend_books` function. While the system is running, we display a status message. On success, the message confirms how many recommendations were found; on failure, it provides a clear error.

5.6.3 Recommendations Gallery

The next section, *"Your personalized recommendations"*, shows a 4×4 gallery of book covers. Clicking on any item expands the image and reveals the full caption with the formatted title, authors, truncated description, and AI explanation. The "Cover not found" placeholder is visible for books without real covers.

5.6.4 Downloadable Reading List

At the bottom of the page, a file component allows users to download a CSV file of their current recommendations, including the AI-generated reasons. This feature transforms the dashboard from a pure demo into a practical tool for saving and revisiting suggestions.

5.7 Conclusion of Step 5

In Step 5, we transformed our semantic book recommender from a collection of scripts into an end-to-end interactive application. The Gradio dashboard combines:

- semantic search over document embeddings,
- category-level filtering derived from our text classification module,
- emotion-based sorting using sentence-level sentiment scores,
- visual presentation of book covers, and
- concise AI-generated justifications for each recommendation.

Together, these elements provide an intuitive interface where users can describe the kind of story they want to read, adjust filters to match their preferences, and immediately explore a curated gallery of relevant titles. This final step demonstrates how modern NLP pipelines can be packaged into accessible tools for real readers, completing our end-to-end semantic book recommendation system.

6 General Conclusion

Across the five stages of this project, we designed and implemented a complete end-to-end semantic book recommendation system by integrating data cleaning, large language models, sentence-level emotion analysis, and an interactive web interface. Starting from a raw dataset of 6,810 book records with inconsistent metadata and missing values, we constructed a refined corpus of 5,197 books that contains complete descriptions, standardized categories, and seven emotion scores per title. Through systematic preprocessing in Step 1, we ensured that each recommendation builds on reliable and meaningful text information rather than sparse or noisy source fields.

In Step 2, we leveraged OpenAI embeddings and a Chroma vector database to enable semantic similarity search. This allowed us to retrieve books based on meaning rather than keywords and demonstrated strong performance in both speed and relevance. The subsequent steps incrementally added structure and nuance to the recommendation process. Step 3 reduced over 500 noisy categories to four consistent groupings using a combination of manual mapping and zero-shot text classification. Step 4 extracted sentence-level emotion scores using a fine-tuned DistilRoBERTa model, enabling users to sort search results by mood (such as joyful, suspenseful, or sad) instead of relying solely on topical similarity.

Finally, in Step 5, we combined these components into a Gradio dashboard that provides an intuitive interface for user queries, category filtering, emotional sorting, and visual exploration of book covers. Behind this interface lies a reproducible workflow that reads the cleaned dataset, loads a vector index, applies category and emotion logic,

and returns a user-friendly gallery of recommendations with excerpted descriptions and AI-generated explanations.

6.1 Limitations and Future Work

1. Zero-shot classification accuracy is approximately 78% and shows asymmetry between fiction and non-fiction predictions. Fine-tuning a classifier on a subset of labeled examples could improve performance.

2. Sentence-level emotion extraction identifies dominant emotional signals, but the distribution is heavily skewed toward sadness and fear. Thresholding or normalization could balance emotional categories.

3. The vector database is rebuilt at runtime in the dashboard script. Persisting the database to disk would significantly reduce startup time for deployment.

4. Book cover images depend on external URLs from Google Books, which introduces latency and potential availability issues. Local caching or image storage would increase reliability.

5. Recommendations are stateless. Incorporating user feedback, saving preference history, or applying a re-ranking strategy would move the system closer to production-grade recommendation behavior.

6.2 Reproducibility and How to Run

This project produces the following primary files:

- `books_cleaned.csv`
- `books_with_emotions.csv`
- `tagged_description.txt`
- `gradio_dashboard.py`
- `recommended_books.csv` (runtime output)

Required environment:

- Python 3.11 or later
- Packages: pandas, numpy, langchain, chromadb, gradio, transformers, python-dotenv, tqdm

Basic workflow:

1. Place the `.env` file containing the OpenAI API key and Hugging Face token in the project directory.

2. Run the Python scripts for Steps 1–4 to generate cleaned data and sentiment outputs.

3. Execute `gradio_dashboard.py` to launch the dashboard interface in a browser.

4. Enter a natural language query and optional filters to obtain recommendations.

6.3 Bibliography / References

1. Kaggle dataset: “7k Books With Metadata,” Kaggle, accessed 2025.
2. Facebook AI, “facebook/bart-large-mnli model,” Hugging Face Models, accessed 2025.
3. J. Hartmann, “emotion-english-distilroberta-base,” Hugging Face Models, accessed 2025.
4. OpenAI., “Text Embedding Models,” OpenAI Documentation, accessed 2025.
5. Gradio Team, “Gradio: Build and share machine learning apps,” Gradio Documentation, accessed 2025.
6. Chroma Team, “ChromaDB Vector Database,” Chroma Documentation, accessed 2025.