Université Paris Cité
Master 1 – VMI
Data Science

Information Retrieval Engine

Phase 1: Retrieval Basics

Team Members:

Member 1
Member 2
Member 3
Member 4

Instructor: Themis Palpanas
Teaching Assistant: Manos Chatzakis

March 2, 2026

Contents

1 Introduction

The goal of this project is to build an information retrieval (IR) engine: given a user query, the system must find and rank the most relevant documents from a large collection. This type of system is at the core of search engines and recommendation tools.

In this first phase, we implemented the basic retrieval pipeline. We explored the dataset, built three different retrieval methods (TF-IDF, BM25+, and text embeddings), evaluated them using standard metrics, and submitted our best results to the Kaggle competition.

1.1 AI Usage Disclaimer

We used Claude (LLM) as a coding assistant during this project, mainly for debugging errors, fixing library compatibility issues, and generating boilerplate code. All design decisions, analysis, and writing were done by the team.

2 Data Loading and Exploration

2.1 Dataset Overview

The dataset, provided through Kaggle, consists of four files:

- **docs.json** – The document corpus. Each document has an `id`, `title`, `text`, `tags` (list of keywords), and a `category` indicating its origin.
- **queries_train.json** – 327 training queries with the same fields as documents.
- **queries_test.json** – 141 test queries used for Kaggle submission.
- **qgts_train.json** – Ground truth: for each training query, the list of relevant document IDs.

2.2 Basic Statistics

Metric	**Value**
Total documents	216,041
Training queries	327
Test queries	141
Total relevance judgments	981
Avg. relevant docs/query	3.00

Table 1: Dataset summary.

An important observation is that each query has on average only 3 relevant documents out of 216,041. This means the retrieval task is quite challenging: the system needs to find a needle in a haystack.

2.3 Content Length Analysis

	Documents	Queries
Mean	137 words	11 words
Median	100 words	11 words
Min	9 words	3 words
Max	5,256 words	31 words

Table 2: Content length statistics (word count).

Queries are very short compared to documents. This is typical in IR: users express their needs in a few words, while documents contain much more detail. This gap can make it harder for keyword-based methods (TF-IDF, BM25+) to match queries with relevant documents.

2.4 Category Distribution

The documents are split across 5 categories, reflecting their origin:

Category	Documents
tex	68,184
unix	47,382
gaming	45,301
programmers	32,176
android	22,998

Table 3: Document distribution by category.

The distribution is imbalanced: *tex* has about 3 times more documents than *android*. This could bias retrieval towards categories with more documents.

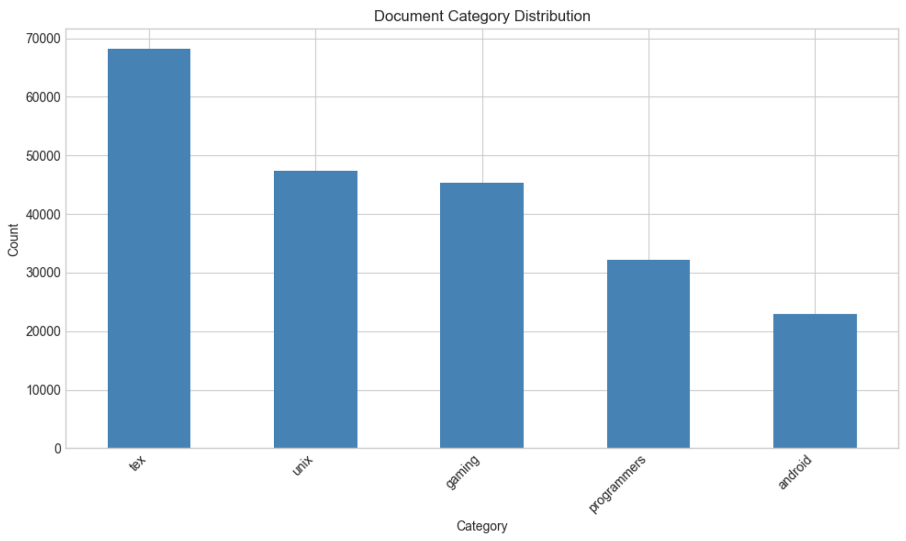

Figure 1: Category distribution across the document corpus.

3 Text Preprocessing

Before building our retrieval models, we created a unified `content` field for each document and query by concatenating the `title`, `text`, and `tags` fields. This gives each retrieval model a single, complete representation of the text.

For the keyword-based methods (TF-IDF and BM25+), we also applied a cleaning step: lowercasing and removing punctuation. This helps reduce noise and ensures that words like “Python” and “python” are treated as the same token.

For the embedding-based method, we used the original (uncleaned) content, since the pre-trained model was trained on natural text and benefits from proper casing and punctuation.

4 Retrieval Methods

We implemented three retrieval methods. For each query, the system returns the top-k most relevant documents (we used $k = 10$).

4.1 TF-IDF

TF-IDF (Term Frequency–Inverse Document Frequency) is a classic text representation method. The idea is simple: a word is important in a document if it appears frequently in that document (high TF) but rarely across the whole corpus (high IDF). Words like “the” or “is” get low scores because they appear everywhere, while specific terms like “LaTeX” or “Ubuntu” get higher scores.

We used scikit-learn’s `TfidfVectorizer` with English stop words removed and a vocabulary limited to 100,000 features. Each document and query is represented as a sparse vector, and we compute cosine similarity between query and document vectors to rank documents.

The resulting TF-IDF matrix has shape (216,041 $\times$ 100,000).

4.2 BM25+

BM25+ is an improvement over TF-IDF commonly used in search engines. It builds on the same idea of term frequency and document frequency, but adds two refinements:

- **Term frequency saturation**: having a word appear 10 times vs. 100 times in a document doesn’t make it 10x more relevant. BM25+ applies a diminishing return to term frequency.
- **Document length normalization**: longer documents naturally contain more words. BM25+ adjusts scores so that shorter, more focused documents aren’t penalized.

We used the `rank_bm25` library with the BM25+ variant, which also adds a small constant to ensure no term gets a zero score.

4.3 Text Embeddings (SentenceTransformers)

Unlike TF-IDF and BM25+ which rely on exact word matching, embedding-based retrieval captures the *meaning* of text. The model converts each piece of text into a dense vector (384 dimensions in our case) where semantically similar texts are close together in the vector space.

This means the model can match a query like "fix broken phone screen" with a document about "repairing smartphone displays" even though they share very few words.

We used the `all-MiniLM-L6-v2` model from the SentenceTransformers library, which provides a good balance between speed and quality. Document embeddings were generated on GPU to speed up the process (about 5 minutes for the full corpus). The resulting embedding matrix has shape (216,041 $\times$ 384).

4.3.1 Why do embeddings help compared to simpler models?

Keyword-based methods can only match documents that contain the exact words from the query. This is a significant limitation: a query about "car" won't match a document about "automobile" unless both words appear somewhere.

Embeddings solve this by learning from large amounts of text data. The model learns that "car" and "automobile" have similar meanings and places them close together in the vector space. It also captures more complex relationships, like understanding that "Python error handling" is related to "try-except blocks". This makes embedding-based retrieval much more robust, especially for short queries where we only have a few words to work with.

5 Embedding Visualization

To better understand the structure of our embeddings, we visualized a random sample of 5,000 document embeddings in 2D using two dimensionality reduction techniques: t-SNE and UMAP. Each point is colored by its document category.

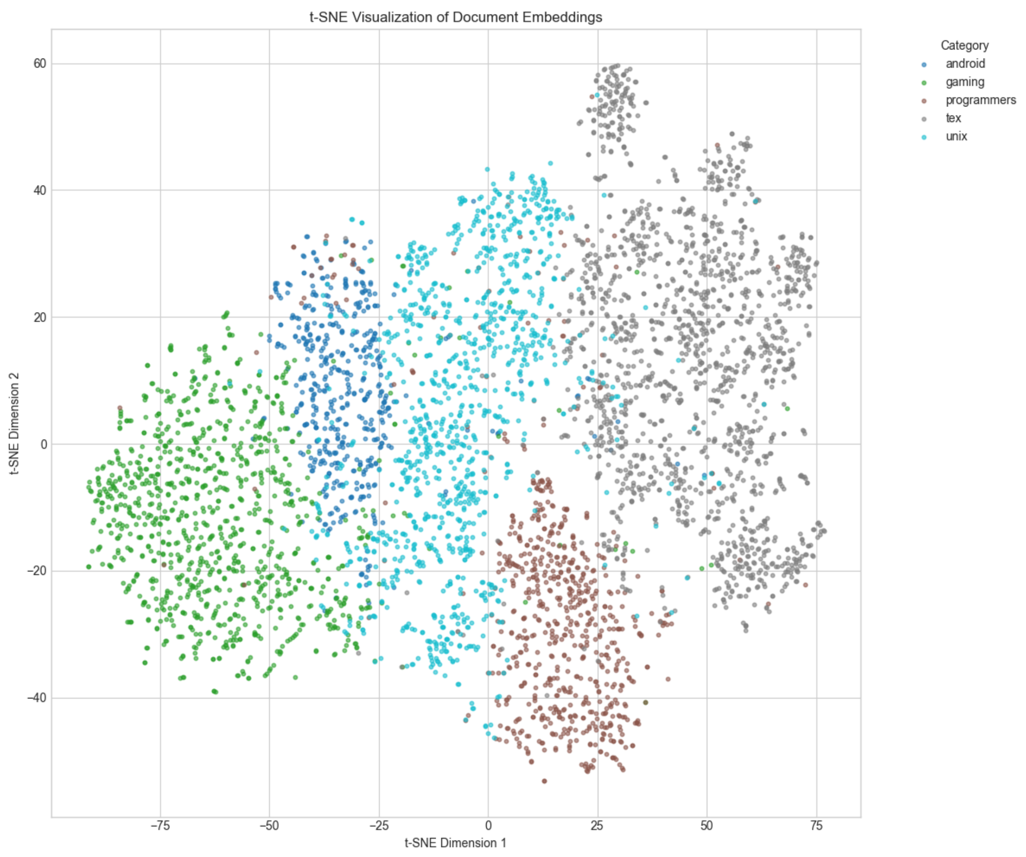

Figure 2: t-SNE projection of document embeddings, colored by category.

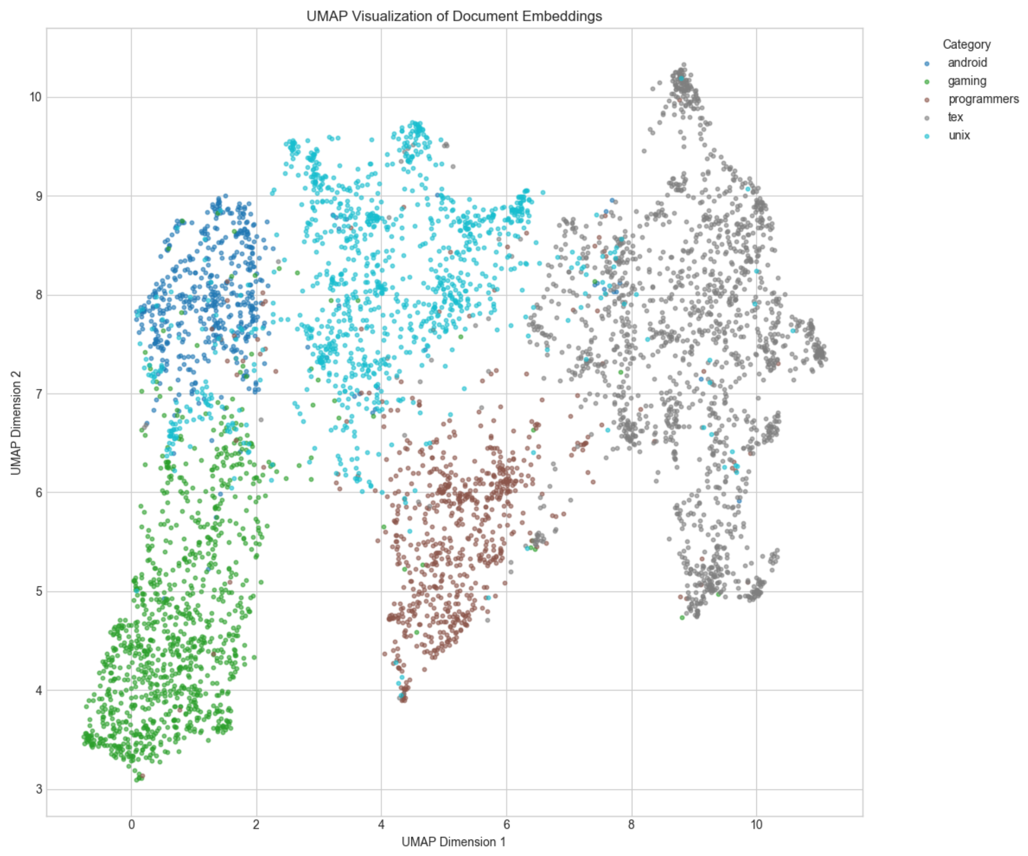

Figure 3: UMAP projection of document embeddings, colored by category.

5.1 Observations

Both visualizations show clear clustering patterns. Documents from the same category tend to group together, forming distinct regions in the 2D space. This is especially visible for categories like *gaming* and *tex*, which have quite specific vocabularies.

This clustering makes sense: the embedding model captures the semantic content of each document, and documents from the same domain naturally discuss similar topics and use similar language. For example, *tex* documents talk about LaTeX formatting, packages, and compilation, while *gaming* documents discuss game mechanics, graphics, and platforms.

Some overlap exists between categories, which is also expected. A question about installing software on Android (*android*) might use very similar language to a question about installing software on Linux (*unix*).

This visualization confirms that the embeddings are meaningful and capture real semantic structure in the data, which is why they perform well for retrieval.

6 Evaluation Metrics

We evaluate our retrieval systems using three standard metrics, all computed with respect to the ground truth relevance judgments:

- **Recall@k**: Out of all the relevant documents for a query, how many did we retrieve in our top-k? Measures completeness.

- **Precision@k**: Out of the k documents we retrieved, how many are actually relevant? Measures accuracy.
- **MRR (Mean Reciprocal Rank)**: How early does the first relevant document appear in our ranking? If the first relevant result is at position 1, the score is 1.0; at position 2, it's 0.5; at position 3, it's 0.33, etc. This is averaged over all queries.

7 Results and Comparison

7.1 Results Summary

Method	Recall@10	Precision@10	MRR
TF-IDF	0.1122	0.0657	0.2021
BM25+	0.1506	0.0896	0.2452
Embeddings	**0.1959**	**0.1141**	**0.2745**

Table 4: Evaluation results for $k = 10$.

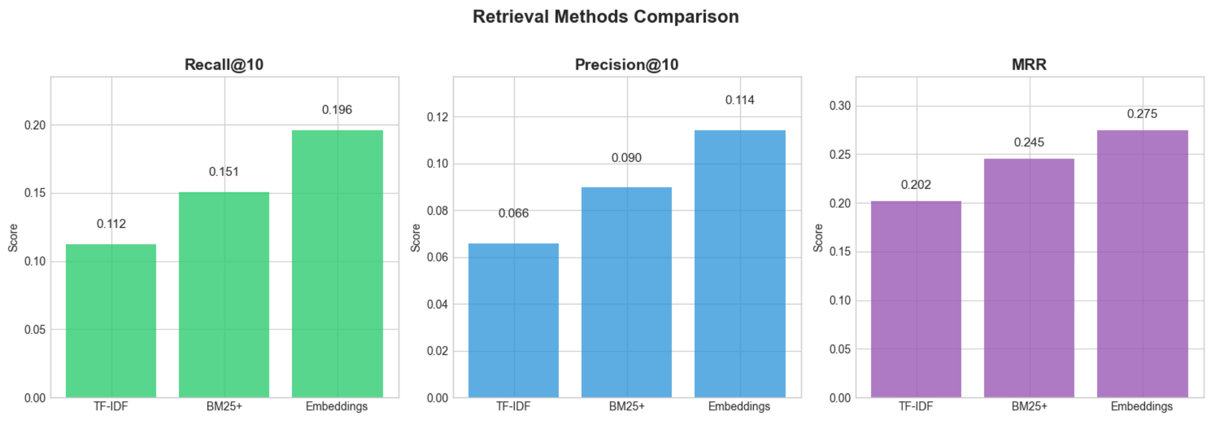

Figure 4: Comparison of the three retrieval methods across all metrics.

7.2 Analysis

Embeddings outperform both keyword-based methods across all three metrics. This is consistent with what we expected: the queries in this dataset are short (11 words on average), and there is often a vocabulary gap between the query and the relevant documents. Embeddings handle this much better because they match on meaning rather than exact words.

BM25+ outperforms TF-IDF on all metrics as well. The document length normalization and term frequency saturation of BM25+ give it an edge over raw TF-IDF, especially in a corpus where document lengths vary a lot (from 9 to over 5,000 words).

TF-IDF is the weakest model in this setting. Without document length normalization, longer documents tend to accumulate higher similarity scores simply because they contain more terms, which introduces noise in the ranking.

7.3 Trade-offs

	TF-IDF	**BM25+**	**Embeddings**
Retrieval quality	Low	Medium	High
Speed	Fast	Slow	Medium
Memory usage	Low	Medium	High
Setup complexity	Simple	Simple	Requires model

Table 5: Trade-offs between retrieval methods.

- **TF-IDF** is the fastest method (about 2 seconds for all queries) thanks to efficient sparse matrix operations. However, its quality is the lowest.
- **BM25+** is the slowest (over 6 minutes for 327 queries) because it computes scores sequentially for each query. Its quality is better than TF-IDF but still limited by exact matching.
- **Embeddings** require an initial investment (about 5 minutes to encode all documents on GPU), but once the embeddings are computed, retrieval is fast (about 2 seconds using cosine similarity on dense matrices). It gives the best results.

7.4 Effect of k

The choice of k affects the metrics differently. A larger k generally increases recall (we retrieve more documents, so we're more likely to find the relevant ones) but decreases precision (more of the retrieved documents are irrelevant). MRR is less affected by k since it only depends on the rank of the *first* relevant document. In our case, with an average of 3 relevant documents per query, $k = 10$ is a reasonable choice.

8 Kaggle Submission

Based on our evaluation results, we chose the **embedding-based method** for our Kaggle submission, as it consistently outperformed both TF-IDF and BM25+ across all metrics.

For the 141 test queries, we generated embeddings using the same model, retrieved the top-10 most similar documents, and predicted the query category using a majority vote among the categories of the retrieved documents.

9 Conclusion

In this first phase, we built a complete information retrieval pipeline from scratch. We explored and preprocessed the dataset, implemented three retrieval methods, and evaluated them rigorously.

The main takeaway is that semantic embeddings significantly outperform traditional keyword-based methods on this dataset. The short nature of the queries and the vocabulary gap between queries and documents make it difficult for TF-IDF and BM25+ to find relevant matches, while the embedding model captures meaning beyond individual words.

For Phase 2, we plan to explore classification-based reranking and hybrid approaches that combine multiple retrieval methods to further improve performance.