Project Phase 2 Deliverable 2: Proof of Concept Implementation

Alexandria Roberts

School of Computer and Information Sciences

MSCS531-A01 Computer Architecture and Design

Professor Bass

11/16/2025

Partial Implementation of a Python-Based Mini Search Engine: Phase 2 Proof of Concept

Introduction

Phase 2 of the Mini Search Engine project I'm working on requires creating a partial proof-of-concept (PoC) that demonstrates the core functionality of the data structures that I designed for my Phase 1. While Phase 1 was meant to focus on conceptual design and justification. Phase 2 is where we emphasize actual implementation, testing, and documenting the development process. The goal of this PoC is to show that the inverted index, tokenization process, and TF-IDF scoring model function correctly and can support simple keyword-based search.

This project is based on well-established concepts in information retrieval, along with building an inverted index to map terms to documents (Manning et al., 2009; Zobel & Moffat, 2006) and ranking documents using TF-IDF weighting. (Zhang et al., 2011) This phase demonstrates these ideas operating together in a functional Python program contained entirely within one file named, "mini_search.py"

Partial Implementation Overview

Phase 2 implementations include several components required for a working search engine. These include:

(1) Adding documents to the system
(2) Tokenizing text into normalized terms
(3) Building the inverted index of terms
(4) Storing term frequency (TF) and document frequency (DF) values
(5) Ranking documents using TF-IDF

All of these components were achieved when using Python's built-in data structures (dict, defaultdict, and Counter). This supports fast lookups, and a cleaner, more simple organization of term statistics. (Manning et al., 2009) The entire PoC is implemented in a single file for easier testing and demonstration.

```
def add_doc(self, doc_id: int, text: str):
"""Add a new document (id, text) into index."""
self.docs[doc_id] = text
self.N = len(self.docs)

tokens = self._tokenize(text)
counts = Counter(tokens)

for term, c in counts.items():
self.tf[(term, doc_id)] = c
```

```
self.df[term] += 1
self.postings[term].append(doc_id)
self.postings[term] = sorted(set(self.postings[term]))
```

This function tokenizes the document, counts term frequencies, builds the inverted index, and tracks document frequency.

Tokenization

```
def _tokenize(self, text: str):
"""Normalize and split into lowercase tokens"""
clean = ''.join(ch.lower() if ch.isalnum() else ' ' for ch in text)
return [t for t in clean.split() if t]
```

The tokenizer lowercases text and removes punctuation, which prevents issues with things like symbols or capitalization.

TF-IDF Scoring

```
def _tfidf(self, term: str, doc_id: int) -> float:
"""Computing TF-IDF weight for a term in a document."""
tf = self.tf[(term, doc_id)]
idf = math.log(1 + (self.N / (1 + self.df[term])))
return (1 + math.log(tf)) * idf
```

This formula determines how relevant a term is to a given document and collection (Zhang et al., 2011)

Searching and Ranking

```
def search(self, query: str, k: int = 5):
"""Return top-k docs ranked by TF-IDF score."""
terms = self._tokenize(query)
if not terms:
return[] #Handle empty or an invalid query
candidates = set()
for t in terms:
candidates.update(self.postings.get(t, []))
```

```
scores = []
for d in candidates:
s = sum(self._tfidf(t, d) for t in terms if (t, d) in self.tf)
scores.append((s, d))
scores.sort(reverse=True)
return scores[:k]
```

This function gathers all documents that contain the query terms, applies TF-IDF scoring, sorts them, and then returns the top results.

Demonstration and Testing

The bottom section of this file serves as the demonstration script when executed it does a list of things including:

- Instantiates the search engine
- Adds five sample documents
- Runs a series of normal and edge-case queries
- Prints the ranked results and execution times

Queries Tested

The following queries were used to test this system:

- “cats”
- “dogs”
- “cats milk”
- “bird”
- “Cats!” capitalization and punctuation (edge case)
- “” - empty query (edge case)

Summary of the Terminal Output

```
Query: 'cats'
doc 3: Milk is good for cats (score=0.811)
doc 2: Cats and dogs like treats (score=0.811)
doc 0: Cats like pillows (score=0.811)
Time: 0.035 ms
```

For “dogs” and “cats milk”, the results were also ranked correctly based on document content.

This therefore demonstrates:

- Correct TF-IDF ranking
- Stable and predictable ordering
- Reasonable execution time

The edge-case queries behaved as expected as well:

- “bird” returned “No matching documents found.”
- “CATS!” returned the same as “cats”. Confirming tokenizer nromalization
- “” returned “no results”, which avoids crashes or errors

The tests confirmed the PoC handles typical and edge-case input safely and accurately, consistent with recommendations in information-retrieval research (Manning et al., 2009; Zobel & Moffat, 2006)

Implementation Challenges and Solutions

1. Maintaining Sorted Posting Lists
   While this worked for smaller datasets for sorting and deduplicating postings during document insertion, it would not scale efficiently. So, for now, readability and correctness were my priorities over optimization.

2. Handling Edge Cases
   The demo originally only tested normal queries, so adding edge cases greatly improves robustness. Making the PoC more aligned with real-world behavior:

3. Combining Everything in One File
   For now, I have the class and demo in one file, but this may need to change. They will probably need to be separated as the project gets larger.

4. Tokenization Limitations
   At this time, it does not remove things like stop-words or stem words. This will need to be improved for later phases.

Next Steps

- Future improvements include:
- Adding stop-word removal and stemming
- Implementing positional indexes for phrase search
- Reducing sorting overhead in posting lists
- Handling larger document collections

- Moving the demo into a seperate file
- Adding a user interface
- Uploading the project to GitHub with a README file

These should allow the system to scale appropriately and provide better advanced search capabilities.

Conclusion

For Phase 2 I successfully demonstrated a working proof-of-concept for the Mini Search Engine. The implementation shows that the inverted index, tokenizer, and TF-IDF rankings operate together correctly. The edge-case and normal testing confirm functional behavior and large handling of various queries. The PoC aligns with information-retrieval theory as well and lays a strong foundation for future improvements.

References

Manning, C. D., Raghavan, P., & Schütze, H. (2009). *Introduction to information retrieval*. Cambridge University Press.

Zhang, W., Yoshida, T., & Tang, X. (2011). A comparative study of TF-IDF, LSI and multi-word features for text classification. *Expert Systems with Applications*

Zobel, J., & Moffat, A. (2006). Inverted files for text search engines. *ACM Computing Surveys*