Inverted Index and B-tree-based Information Retrieval System

Andrea Frigatti

June 19, 2024

1 Introduction

This report details the implementation of an information retrieval system that utilizes an inverted index and B-tree structures to efficiently handle various types of search queries. The implementation is spread across three Python scripts, each contributing to different stages of the process: constructing the inverted index, building and managing B-trees, and executing search queries.

2 Implementation Details

2.1 Inverted Index Construction

The first script, `inverted_index.py`, is responsible for reading a dataset of movie plots, cleaning the text, and constructing an inverted index. Key steps in this script include:

- Reading and processing the data.
- Removing punctuation, normalizing text, deleting a list of stopwords.
- Constructing the inverted index and saving it along with the document data.

```
def load_data(file_path, stopwords):
    df = pd.read_csv(file_path)
    plots = df["Plot"]

    for id, plot in plots.items():
        plot = plot.translate(str.maketrans('', '', string.punctuation))
        plot = re.sub(r'\W+', ' ', plot)

        words = set(plot.lower().split())
        for word in words:
            if word not in stopwords:
                posting_lists.setdefault(word, []).append(id)

        document_data[id] = df.loc[id].to_dict()

    with open(save_file_path, "w", encoding='utf-8') as file:
        for word in posting_lists:
            file.write(f"{word}: {posting_lists[word]}\n")

    save_document_data_pickle(document_data, document_pickle_file_path)
```

Listing 1: Inverted Index Construction

Advantages:

- **Efficient Text Processing:** The script effectively removes punctuation and normalizes the text, ensuring uniform and reliable indexing.
- **Use of Stopwords:** Excluding common stopwords keeps the inverted index concise and focused on meaningful terms.
- **Data Persistence:** Saving the inverted index and document data to disk allows for quick reloads and use in subsequent processes.

2.2 B-trees Construction and Management

The second script, `btree.py`, defines a B-tree structure and builds B-trees from the inverted index file. Key steps include:

- Defining the `Node` and `BTree` classes.
- Building B-trees from the inverted index file (Search Tree and Reversed Search Tree).
- Saving the B-trees for future use in `retrieving.py`.

```python
class Node:
    def __init__(self, is_leaf=False):
        self.is_leaf = is_leaf
        self.keys = []
        self.values = []
        self.children = []

class BTree:
    def __init__(self, degree):
        self.root = Node(is_leaf=True)
        self.degree = degree

    def build_from_file(self, file_path, inverted=False):
        with open(file_path, 'r', encoding='utf-8') as file:
            for line in file:
                key, values_str = line.strip().split(': ')
                values = list(map(int, values_str.strip('[]').split(', ')))
                if inverted:
                    key = key[::-1]
                self.insert(key, values)

    def insert(self, key, values):
        if len(self.root.keys) == (2 * self.degree) - 1:
            new_root = Node()
            new_root.children.append(self.root)
            self._split_child(new_root, 0)
            self.root = new_root

        self._insert_non_full(self.root, key, values)

    def _insert_non_full(self, node, key, values):
        i = len(node.keys) - 1
        if node.is_leaf:
            node.keys.append(None)
            node.values.append(None)
            while i >= 0 and key < node.keys[i]:
                node.keys[i + 1] = node.keys[i]
                node.values[i + 1] = node.values[i]
                i -= 1
```

```
            node.keys[i + 1] = key
            node.values[i + 1] = values
        else:
            while i >= 0 and key < node.keys[i]:
                i -= 1
            i += 1
            if len(node.children[i].keys) == (2 * self.degree) - 1:
                self._split_child(node, i)
                if key > node.keys[i]:
                    i += 1
            self._insert_non_full(node.children[i], key, values)

    def _split_child(self, parent, i):
        degree = self.degree
        child = parent.children[i]
        new_child = Node(is_leaf=child.is_leaf)

        parent.keys.insert(i, child.keys[degree - 1])
        parent.values.insert(i, child.values[degree - 1])
        parent.children.insert(i + 1, new_child)

        new_child.keys = child.keys[degree:]
        new_child.values = child.values[degree:]
        child.keys = child.keys[:degree - 1]
        child.values = child.values[:degree - 1]

        if not child.is_leaf:
            new_child.children = child.children[degree:]
            child.children = child.children[:degree]
```

Listing 2: B-tree Construction

Advantages:

- **Efficient Data Storage:** B-trees provide efficient storage and retrieval, supporting quick searches.
- **Scalability:** B-trees handle large datasets effectively by maintaining balanced structures.
- **Support for Inverted Searches:** Constructing both normal and inverted B-trees enables efficient handling of several types of queries.

2.3 Query Processing and Data Retrieval

The final script, `retrieving.py`, loads the B-trees and processes various types of queries to retrieve relevant documents. Key functionalities include:

- Loading B-trees and document data.
- Handling single word, conjunctive, disjunctive, prefix, suffix, and wildcard queries.
- Combining results from multiple query types.

```
def search(self, key):
    return self._search(self.root, key)

def _search(self, node, key):
    i = 0
    while i < len(node.keys) and key > node.keys[i]:
        i += 1
    if i < len(node.keys) and key == node.keys[i]:
        return node.values[i]
```

```
    elif node.is_leaf:
        return None
    else:
        return self._search(node.children[i], key)
```

Listing 3: Single Word Query

2.3.1 Single Word Query

A single word query searches for documents containing the specified term.

```
def single_word_query(btree, word):
    return btree.search(word)
```

Listing 4: Single Word Query

2.3.2 Conjunctive Query

A conjunctive query returns documents containing all the specified terms.

```
def conjunctive_query(btree, words):
    results = [set(btree.search(word)) for word in words]
    return list(set.intersection(*results))

def intersect(lists):
    if not lists:
        return []

    sorted_lists = sorted(lists, key=len)
    return list(reduce(set.intersection, map(set, sorted_lists)))
```

Listing 5: Conjunctive Query

2.3.3 Disjunctive Query

A disjunctive query returns documents containing any of the specified terms.

```
def disjunctive_query(btree, words):
    results = [set(btree.search(word)) for word in words]
    return list(set.union(*results))

def union(lists):
    return list(set(chain.from_iterable(lists)))
```

Listing 6: Disjunctive Query

2.3.4 Prefix Query

A prefix query searches for documents containing words that start with the specified prefix.

```
def prefix_query(btree, prefix):
    return btree.prefix_search(prefix)

def search_prefix(self, prefix):
    results = []
    self._search_prefix(self.root, prefix, results)
    return results
def _search_prefix(self, node, prefix, results):
    i = 0
    while i < len(node.keys):
        if node.keys[i].startswith(prefix):
            results.append((node.keys[i], node.values[i]))
```

```
        if node.keys[i] > prefix and not node.is_leaf:
            self._search_prefix(node.children[i], prefix, results)
        i += 1
    if not node.is_leaf:
        self._search_prefix(node.children[i], prefix, results)
```

Listing 7: Prefix Query

2.3.5 Suffix Query

A suffix query searches for documents containing words that end with the specified suffix.

```
def suffix_query(inverted_btree, suffix):
    return inverted_btree.prefix_search(suffix[::-1])

def search_suffix(self, suffix):
    reversed_suffix = suffix[::-1]
    results = []
    self._search_prefix(self.root, reversed_suffix, results)
    return [(key[::-1], values) for key, values in results]
```

Listing 8: Suffix Query

2.3.6 Wildcard Query

A wildcard query searches for documents containing words that match the specified pattern.

```
def wildcard_query(btree, inv_btree, pattern):
    if '*' in pattern:
        prefix, suffix = pattern.split('*')
        prefix_results = prefix_query(btree, prefix)
        suffix_results = suffix_query(inv_btree, suffix[::-1])
        return list(set(prefix_results).intersection(suffix_results))
```

Listing 9: Wildcard Query

Advantages:

- **Flexible Query Handling:** The system supports a wide range of query types, making it versatile for different search needs.
- **Efficient Retrieval:** Using B-trees for search operations ensures that queries are handled quickly, even with large datasets.

3 Conclusion

The described implementation leverages the strengths of inverted indexes and B-trees to create a robust and efficient information retrieval system. The use of text normalization, stopword removal, and persistent storage ensures that the system is both efficient and scalable. The support for various query types makes this system highly versatile and powerful for different search needs.