Information Retrieval Models - Part 1

Experiment 1: Comparing `grep` with Inverted Index based Boolean Retrieval

a. Implementation Overview

In the 1st Experiment, We compared the performances of 2 methods for retrieving the Documents based on Boolean Queries:

1. **Inverted Index built from the corpus**
2. **grep Command**

Dataset: We Executed the Queries from the s2/s2_queries.json file using both the methods and measure various performance metrics.

1. Inverted Index

Index Construction

- The corpus is read from the s2/s2_doc.json file, and the token-docID pairs are extracted and sorted.
- The sorted pairs are then used to construct postings and document freq Dictionaries.

Query Execution

- Queries from the s2/s2_queries.json file are read, parsed and then are converted into Boolean Queries using `add_query` function.
- Boolean queries are executed using the constructed Inverted Index and the results are recorded for each query.

Functions Used

`profiled_code`:

```
This function runs a set of queries from a specified file path and profiles the
code.
```

`run_queries_from_file`:

```
This function runs queries from a file and processes them against the provided
corpus directory.

Parameters:
    queries_file: The file containing the queries to be processed.
    corpus_dir: The directory containing the corpus to be queried.

Returns:
    None.
```

`load_index_in_memory`:

```
Load index data from the specified directory into memory.

Parameters:
 dir (str): The directory path where the index data is located.

Returns:
tuple: A tuple containing two dictionaries - postings and doc_freq.
       postings (dict): A dictionary containing token as key and list of items as
value.
       doc_freq (dict): A dictionary containing token as key and frequency as value.
```

`read_queries_from_file`:

```
Read queries from a file and return the queries data.

Parameters:
    file_path (str): the path to the file to read
```

```
Returns:
    queries_data (list): the queries data from the file
```

`add_query`:

```
A function to perform an 'and' query on a given set of query terms against a corpus.

Parameters:
    query_terms (list): List of query terms to search for in the corpus
    corpus (str): The text corpus to search within

Returns:
    result (list): A list of document IDs that contain all the query terms
```

2. grep Command

Query Execution

- Queries from the s2/s2_queries.json file are read and executed using the `run_grep` function which runs the grep command.
- Results are recorded for each query.

Functions Used

`profiled_code_grep`: This function runs a set of queries from a specified file path and profiles the code.

`grep_run_queries`: Execute a series of queries from the specified file on the provided corpus directory using the run_grep function.

```
Parameters:
    queries_file_path (str): The file path to the queries file.
    corpus_dir (str): The directory containing the corpus data.

Returns:
    None
```

`run_grep`: Perform a recursive grep search for the given query in the specified corpus directory.

```
Parameters:
    query (str): The string to search for.
    corpus_dir (str): The directory in which to search.

Returns:
    None
```

b. Performance Measurement

- Time taken for each query is recorded. The Individual, Maximum, Minimum and Average times are calculated.
- Profilers are used to analyse the performance of the methods.

c. Experiment Findings

- The Inverted index-based boolean retrieval generally outperformed the grep command in terms of both time, while the accuracy reamins same.
- The preformance of grep may degrade with large corpus or complex queries because it is a linear search approach.

Experiment 2: Linguistic post-processing of the vocabulary

a. Implemantion Overview

In Experiment 2, we implemented a liguistic post-processing of the step on the vocabulary to enhance its qulaity for the subsequent analysis. The psot-processing involved three major steps:

1. **Stemming**
2. **Lemmatization**
3. **Stopword Removal**

This process aimed to reduce the complexity of the vocabulary and increase the accuracy of of tasks such as Information Retrieval or Text Classification.

b. Implementation Details

- **Tokenization**: The word_tokenize function from the nltk library is used to tokenize the words.
- **Stemming**: Using the Porter Stemmeter algorithm to reduce words to their root form.
- **Lemmatization**: Using the WordNetLemmatizer to lemmatize words to their root form.
- **Stopword Removal**: Using the stopwords from the nltk library to remove common words that are not useful in the analysis.

Procedure

1. Initialize the Porter Stemmer and WordNet Lemmatizer.
2. TOkenize the input vocabuary into individual words.
3. Iterate through each word in the vocabulary.

- Perform stemming to obtain the root from the word.
- Apply lemmatization to convert the word to its base form.
- Check if ther lemmatized word is not a stopword.
- If not stopword, add it to the lemmatized word to the processed vocabulary.

4. Return the processed vocabulary.

Experiment 3: Compare hash-based and tree-based implementation of dictionaries

a. Implementation Overview

In the 3rd Experiment, We compared the performances of two implementations of dictionaries based on:

1. **Tree-based Data Structure (Trie)**
2. **hash-based Data Structure**

Dataset: We used from the s2/s2_doc.json file and examined the correctness and performance of both implementations. We also used the s2/s2_queries.json to test the performance of the both the Data Structures.

1. Tree-based Data Structure (Trie)

Implementation

- The Trie Data Structure is implemented to store the dictionary in the form of a tree.
- The Trie is contructed by inserting each token from the corpus into the Trie.

Query Execution

- Boolean Retrieval is performed for the 100 queries using the Trie.
- The time for each query and performance metrics are recorded.

Functions Used

`build_trie`:

```
Builds a trie from the data in the specified directory.

Parameters:
    dir (str): The directory containing the data.

Returns:
    TrieNode: The root of the trie built from the data.
```

`boolean_from_trie`:

```
This function loads a trie from a JSON file, read queries from another file,
and processes each query to obtain results from the trie.
```

`run_queries_from_file`:

```
This function runs queries from a file and processes them against the provided
corpus directory.

Parameters:
    queries_file: The file containing the queries to be processed.
    corpus_dir: The directory containing the corpus to be queried.

Returns:
    None.
```

`read_queries_from_file`:

```
Read queries from a file and return the queries data.

Parameters:
  file_path (str): the path to the file to read

Returns:
  queries_data (list): the queries data from the file
```

`trie_and_query`:

```
Function to perform a query using a trie data structure.

Parameters:
    trie: The trie data structure to be queried.
    query_terms: The list of terms to be queried.

Returns:
    result (list): a set containing the results of the query.
```

`trie_search`:

```
Function to search for a query in a trie data structure.

Parameters:
  trie: the trie data structure to search in
  query: the query string to search for in the trie

Returns:
  A set of documents at the leaf node matching the query
```

2. Hash-based Data Structure

Query Execution

- Queries from the s2/s2_queries.json file are read and executed using the `run_grep` function which runs the grep command.

- Results are recorded for each query.

Functions Used

b. Performance Measurement

- Time taken for each query is recorded for both the implementations. The Individual, Maximum, Minimum and Average times are calculated.
- Profilers are used to analyse the performance of the methods.

c. Experiment Findings

- The Tree-based Data Structure (Trie) outperformed the hash-based Data Structure in terms of both time and accuracy.
- This experiment highlights the trade-offs between tree-based and hash-based implementations for dictionary storage and retrieval.

Experiment 4: Wild card querying using Permuterm and Tree based Indexes

a. Implemantion Overview

In the 4th Experiment, We explored wildcard querying using two different indexing methods based on:

1. **Permuterm Indexes**
2. **Tree-based Indexes**

Dataset: We used used the s2/s2 wildcard.json to test the performance and correctness of the both the Indexing Methods.

1. Permuterm Indexes

Implementation

- Constructed permuterm indexes for ecah term in the dictionary.

- Implemented prfix-based search for wildcard queries using permuterms.

Query Execution

- For each wildcard query, we performed prefi-based search using permuterm indexes.
- We recorded the time taken per query and calculated minimu, maximum and average over 30 wildcard queries.

Functions Used

`main_permute_index`: This function reads a JSON file containing a trie, constructs a permuterm index from the trie, and writes the permuterm index to a TSV file.

`construct_permuterm_index_from_trie`:

```
Constructs a permuterm index from the given trie.

Parameters:
    trie: The trie data structure to construct the permuterm index from.

Returns:
    dict: The permuterm index containing the permutations of terms and corresponding
documents.
```

`generate_permutations`:

```
Generate permutations for a given node and current term.

Args:
    node: The current node in the trie.
    current_term: The current term being constructed.

Returns:
  None
```

`write_permuterm_index_to_file`:

```
Write the permuterm index to a file.

Args:
    permuterm_index: The permuterm index to write to the file.
    output_file: The file to write the permuterm index to.
```

```
Returns:
    None
```

2. Tree-based Indexes

Implementation

- Maintained a forward and a backward tree-based index on all terms.
- Executed the forward index for prefix queries and backward index for suffix queries including the * in both prefix and suffix queries.
- Returned the intersection of the results form both indexes as the answer.

Query Execution

- For each wildcard query, we executed prefix search on both forward and backward indexes.
- Recorded the time taken for each query and then calculated the minimum, maximum and average time over all the 30 wildcard queries.

Functions Used

`main_wildcard`: This function reads wildcard queries from a file, performs wildcard search, and processes the results.

`read_queries_from_file`:

```
Read queries from a file and return the queries data.

Parameters:
  file_path (str): the path to the file to read

Returns:
  queries_data (list): the queries data from the file
```

`wildcard_search_trie`:

```
Performs a wildcard search in a trie data structure.

Parameters:
    trie (Trie): The trie data structure to search in.
    wildcard_query (str): The wildcard query string to search for.
```

```
Returns:
    set: A set of results matching the wildcard query.
```

`backtrack`:

```
Backtracking algorithm to process wildcard search on a trie data structure.

Parameters:
    node: The current node in the trie.
    current_result: The current result string formed during backtracking.
    wildcard_remaining: The remaining wildcard string to be processed.

Returns:
    None
```

b. Performance Measurement

- Time taken per query is recorded for both permuterm and tree-based indexing methods. The Individual, Maximum, Minimum and Average times are calculated.
- Profilers are used to analyse the performance of the methods.

c. Experiment Findings

Experiment 5: Tolerant Retrieval

a. Implementation Overview

In the 5th Experiment, We extended the capabilities of wildcard indexes constructed in Exp 4 to support tolerant retrieval for wildcard queries with boolean retrieval. This Experiment enhances the retrieval system by providing tolerance for missplelled or incomplete terms.

Dataset: We used used the s2/s2 wildcard_boolean.json to test the performance and correctness of the implemented techinque.

Implementation

- Added support for tolerant retrieval using the wildcard indexes constructed in Exp 4.
- Implemented a techinque to handle wildcard queries with boolean retrieval.
- Extended the wildcard index functionality to tolearate missplelled or incomplete queries.

Query Execution

- Each wildcard query from the s2/s2_wildcard_boolean.json file was processed using the wildcard index constructed in Exp 4.
- Results were recorded for each query to correctness and performance of the tolerant retrieval. The time taken per query and calculated minimu, maximum and average.

Functions Used

`tolerant_retrieval`:

```
Function for tolerant retrieval. Reads queries from a file, processes the queries,
and performs wildcard search and and_query operations to retrieve results. Does not
return any value.
```

`read_queries_from_file`:

```
Read queries from a file and return the queries data.

Parameters:
    file_path (str): the path to the file to read

Returns:
    queries_data (list): the queries data from the file
```

`wildcard_search_trie`:

```
Performs a wildcard search in a trie data structure.

Parameters:
    trie (Trie): The trie data structure to search in.
    wildcard_query (str): The wildcard query string to search for.
```

```
Returns:
    set: A set of results matching the wildcard query.
```

`and_query`:

```
A function to perform an 'and' query on a given set of query terms against a corpus.

Parameters:
    query_terms (list): List of query terms to search for in the corpus
    corpus (str): The text corpus to search within

Returns:
    list: A list of document IDs that contain all the query terms
```

`intersection`:

```
Find the intersection of two lists and return a new list containing the common
elements.

Parameters:
 l1: The first list
 l2: The second list

Returns:
 intersection_list: A new list containing the common elements of l1 and l2
```

b. Performance Measurement

- Time taken per query is recorded for tolerant retrieval.
- Evaluated the correctness of the retrieved results compared to expected octcomes.
- Accessed the retrievals preformance in handling misspelled or incomplete queries.

c. Experiment Findings

- Tolerance retrieval shows how efficient it is in handling misspelled or incomplete queries. The ability to find and handle misspelled or incomplete queries improves the overall experience and retrieval performance.