Experiment 10 - Information Retrieval

1.1 Introduction

Information retrieval (IR) is the process of accessing and retrieving relevant information from large data repositories, often consisting of documents or other unstructured data. In today's digital age, where vast amounts of information are available online, IR systems play a crucial role in facilitating access to knowledge and connecting users with the resources they seek.

IR techniques have been integral to computer science for decades, but their significance has become even more pronounced with the advent of the internet. Public search engines, powered by sophisticated IR algorithms, serve as the primary gateway for individuals seeking information on virtually any topic. These systems enable users to navigate through immense datasets efficiently, making information accessible and usable.

At its core, an information retrieval system encompasses three main components:

1. Indexing: This involves the process of creating and maintaining data structures (often referred to as indexes) that enable efficient and effective retrieval of information from a large collection of documents or data. These indexes typically organize the documents based on certain criteria such as keywords, terms, or metadata attributes, allowing for rapid search and retrieval operations. Indexing plays a crucial role in IR systems by significantly speeding up the process of finding relevant information in response to user queries. There are many indexing techniques, including prominent ones like the inverted index and keyword index.

2. Query: Information retrieval systems provide a user interface that allows users to express their information needs through queries. Modern systems often incorporate natural language processing (NLP) techniques to interpret user queries and retrieve relevant results.

3. Display: Once relevant documents are retrieved, the system presents them to the user in a meaningful way, typically accompanied by metadata. A key feature of IR is ranking results based on their relevance to the user's query, ensuring that the most pertinent information is prominently displayed. Additionally, IR systems may employ result tuning mechanisms to refine and personalize results based on user feedback and behavior.

Figure 1.1 shows an overview of information retrieval system components. In this experiment, we are going to test and implement different IR components using *Whoosh*, which is a Python search engine library.

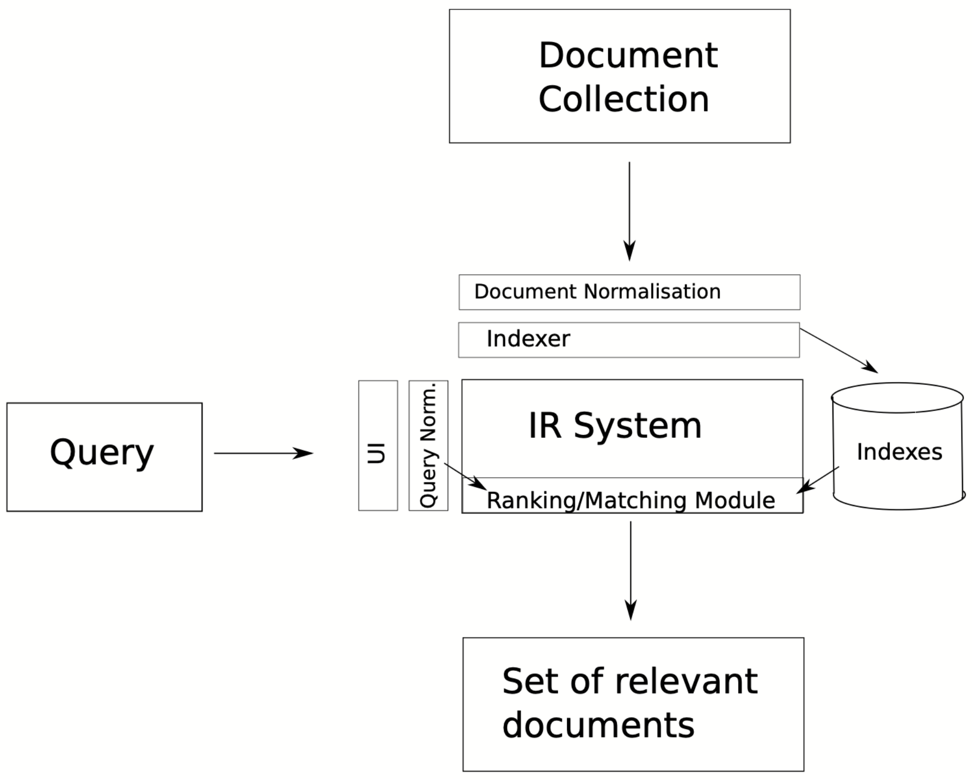

Figure 1.1: IR system components.

1.1.1 Retrieval models

Retrieval models define how documents are matched to user queries. Here, we'll explore various retrieval models:

1. Boolean Model: The Boolean model retrieves documents that match a Boolean expression of query terms. It's based on set theory operations like AND, OR, and NOT.

2. Vector Space Model: The Vector Space Model represents documents and queries as vectors in a high-dimensional space. Documents are ranked based on the similarity between the query vector and document vectors. . The ranking is usually obtained with the cosine distance.

3. Probabilistic Model: The Probabilistic Model ranks documents based on the probability of relevance given a query. It considers the likelihood of a document being relevant to the query.

1.2 Procedure

For the purpose of this lab, we will use *Whoosh* since it is a fast, featureful full-text indexing and searching library implemented in pure Python. It will suffice for this lab, but keep in mind that there are more suitable and better options for larger projects.

1.2.1 Installation

To install the *Whoosh* library, you can use the following command

```
  $ pip install whoosh
```

1.2.2 Preparing the data

We will be using the stackoverflow data from kaggle. There are 3 files — questions.csv, answers.csv and tags.csv. You can use the following command to download it

```
  !kaggle datasets download -d stackoverflow/stacksample
  !unzip stacksample.zip
```

Let's then load the questions file in a dataframe.

```
import pandas as pd
questions=pd.read_csv("Questions.csv", nrows=20000)
questions
```

1.2.3 The Index and Schema objects

Getting started with Whoosh involves creating an index object. When creating an index for the first time, defining its schema is necessary. The schema outlines the fields within the index, representing different pieces of information for each document. These fields can include the document's title or its textual content. Each field can be indexed for searchability and/or stored, ensuring that the indexed value is returned with the search results, which is particularly useful for fields like the title.

Let's start by designing the schema for our index.

```
from whoosh.fields import Schema, TEXT, ID

# Defining index schema
schema = Schema(Id=ID(stored=True), Title=TEXT(stored=True),
    Body=TEXT(stored=True))
```

Here we choose 3 columns form the questions dataframe to be used in our index: "Id", "Title" and "Body" for keyword based search.

Now let's create our index. To index documents we need define folder where to save needed files.

```
import os.path
index_dir = "indexdir"
if not os.path.exists(index_dir):
    os.mkdir(index_dir)
```

Then we can simply create an index and add documents to be indexed

```
from whoosh.index import create_in
from whoosh.index import open_dir


# Creating the index
ix = create_in(index_dir, schema)

# Open the index writer
writer = ix.writer()

# Iterate over the DataFrame and add documents to the index
# we have indexed title, title_body and doc_id
for index, row in questions.iterrows():
    writer.add_document(Id=str(row['Id']), Title = row['Title']
        ,Body=row['Body'])


# Commit and close the writer
writer.commit()
```

1.2.4 How to search

Once you've created an index and added documents to it, you can search for those documents. The *Searcher* object is the main high-level interface for reading the index. It has lots of useful methods for getting information about the index, however, the most important method on the Searcher object is *search()*, which takes a *whoosh.query.Query* object and returns a *Results* object.

Normally the list of result documents is sorted by score. The *whoosh.scoring* module contains implementations of various scoring algorithms. You can set the scoring object to use when you create the searcher using the *weighting* keyword argument.

The following code search the index we created and rank the results based on the Term Frequency-Inverse Document Frequency (TF-IDF) score.

```
from whoosh.qparser import QueryParser
from whoosh.scoring import TF_IDF
from whoosh import scoring
```

```python
# create the query parser
qp = QueryParser("Title", schema=schema)

# parse the query
query_sentence = "How to install"
query = qp.parse(query_sentence)

# create a searcher object
searcher_tfidf = ix.searcher(weighting=scoring.TF_IDF())

# search documents and store them
# we are returing top 3 documents
results_tfidf = searcher_tfidf.search(query, limit=3, scored=True)

# print the documents
for hit in results_tfidf:
    print(hit["Id"])
    print('\n')
    print(hit["Title"])
    print('\n')
    print('------------------\n')
```

Task 1: *Test the previous search code with different queries. For each one check how many matched results are returned.*

Task 2: *Repeat the previous search using the BM25F scoring algorithm, which is used in probabilistic retrieval model. Do you see any difference in the returned results?*

1.2.5 Query expansion

query expansion involves evaluating a user's input (what words were typed into the search query area, and sometimes other types of data) and expanding the search query to match additional documents. Query expansion involves techniques such as:

1. Linguistic query expansion such as finding synonyms of words, and searching for the synonyms as well

2. Corpus-based query expansion, by searching a single query term at a time and counting the most common words in the top returned documents. Repeating this process for all query terms will give you a list of terms that co-occur frequently with your query terms. Thus, they can be used as to expand the query in the domain of that corpus.

3. Pseudo-relevance feedback, by expanding the original query with the most frequent terms of the top retrieved documents.

Whoosh provides methods for computing the "key terms" of a set of documents. For these methods, "key terms" basically means terms that are frequent in the given documents, but relatively infrequent in the indexed collection as a whole. These methods can be useful for query expansion.

For example, the following code retrieve more results similar to the first returned item in the previous example

```
more_results = results_tfidf[0].more_like_this("Title")

for hit in more_results:
    print(hit["Id"])
    print('\n')
    print(hit["Title"])
    print('\n')
    print('------------------\n')
```

We can also extract keywords for the top N documents in a *whoosh.searching.Results* object. This requires that the field is either vectored or stored. For example, to extract five key terms from the Title field of the top ten documents of the results object in the previous example:

```
keywords = [keyword for keyword, score
            in results_tfidf.key_terms("Title", docs=10, numterms=5)]

keywords
```

1.2.6 Evaluating IR systems

There are several measures for evaluating IR systems, such as precision, recall and mean average precision (mAP). While precision and recall does not take the rank of the retrieved documents into consideration, mAP considers the order in which documents are ranked. Sometimes precision and recall are computed at cut-off value k of retrieved documents. In this case, it is called precision@k and recall@k, which means precision and recall computed when considering the first k retrieved documents only.

In this part we will evaluate our IR system on a toy dataset. This dataset contains a set of documents, set of queries, and list of relevant documents for each query.

```

queries = {
    'q1': "machine learning",
    'q2':"AI algorithms"
    }

relevance = {
    'q1' : ["doc1", "doc2", "doc3"],
    'q2' : ["doc1", "doc2", "doc3", "doc4", "doc5"]
}
documents = {
    'doc1': "Artificial Intelligence (AI) is transforming various
       industries through automation and advanced algorithms. Machine
       learning, a subset of AI, enables computers to learn from data and
       make predictions. Algorithms are at the core of AI systems, guiding
       decision-making and problem-solving processes. AI-powered systems
       are increasingly used in healthcare for diagnosis and treatment
       planning. The ethical implications of AI algorithms, such as bias
       and fairness, are important considerations in their development.",

    'doc2': "Deep learning, a branch of machine learning, uses neural
       networks to process complex data. AI algorithms are capable of
       analyzing large datasets to extract meaningful insights. Natural
       Language Processing (NLP) algorithms enable computers to understand
       and generate human language. AI-driven recommendation algorithms
       personalize user experiences in e-commerce and content platforms.
       Ensuring the transparency and accountability of AI algorithms is
       essential for building trust in AI technologies.",

    'doc3': "Reinforcement learning algorithms enable AI agents to learn
       through trial and error interactions with their environment. AI
       algorithms are used in financial markets for high-frequency trading
       and risk management. Computer vision algorithms enable machines to
       interpret and analyze visual information. AI algorithms can enhance
       cybersecurity by detecting and mitigating cyber threats in
       real-time. Continuous research and development are essential for
       advancing AI algorithms and overcoming their limitations.",

    'doc4': "Evolutionary algorithms, inspired by natural selection, are
       used to optimize complex systems and processes. AI algorithms play
       a crucial role in autonomous vehicles for navigation and
       decision-making. Quantum computing algorithms have the potential to
       revolutionize AI by solving complex problems exponentially faster.
       AI algorithms are employed in predictive maintenance to anticipate
       equipment failures and reduce downtime. Ethical guidelines and
       regulations are needed to govern the development and deployment of
       AI algorithms.",

    'doc5': "Genetic algorithms are used to evolve solutions to
       optimization and search problems inspired by natural selection. AI
       algorithms enable personalized content recommendations in streaming
       services and social media platforms. Swarm intelligence algorithms
       mimic the collective behavior of social insects to solve
       optimization problems. AI algorithms are used in drug discovery to
       accelerate the identification of potential treatments.
       Collaborative efforts are essential for advancing AI algorithms and
       harnessing their full potential for societal benefit."
}
```

Let's start by creating an index for this dataset

```python
from whoosh.fields import Schema, TEXT, ID
from whoosh.index import create_in
from whoosh.index import open_dir

# Defining index schema
schema = Schema(Id=ID(stored=True), Body=TEXT(stored=True))

import os.path
index_dir = "indexdir_toy"
if not os.path.exists(index_dir):
    os.mkdir(index_dir)


# Creating the index
ix = create_in(index_dir, schema)

# Open the index writer
writer = ix.writer()

for doc in documents:
    writer.add_document(Id=doc, Body=documents[doc])


# Commit and close the writer
writer.commit()
```

Now as the index is ready, we can start querying the index. The following code test the first query

```python
from whoosh.qparser import QueryParser
from whoosh.scoring import TF_IDF
from whoosh import scoring

# create the query parser
qp = QueryParser("Body", schema=schema)

# parse the query
query_sentence = queries['q1']
query = qp.parse(query_sentence)

# create a searcher object
searcher_tfidf = ix.searcher(weighting=scoring.TF_IDF())

# search documents and store them
# we are returing top 3 documents
results_tfidf = searcher_tfidf.search(query, limit=3, scored=True)
```

```
# print the documents
for hit in results_tfidf:
    print(hit["Id"])
    print('\n')
    print(hit["Body"])
    print('\n')
    print('------------------\n')
```

Task 3*: Compute the precision and recall for the retrieved documents in the previous example.*
Task 4*: Modify the last code to test all queries and then report the precision and recall.*