PyFlowSearch - A Search Engine for Retrieving Python-Related Topic on Stack Overflow

Group Project of "CI6226 Information Retrieval & Analysis" *

Fang Mingmin
Master student from Wee Kim Wee School of Communication and Information
fang0073@e.ntu.edu.sg

Li Pengfei
PhD student from Electrical and Electronic Engineering
pli006@e.ntu.edu.sg

Yang Jing
Master student from Wee Kim Wee School of Communication and Information
jyang025@e.ntu.edu.sg

Zhou Xinhui
Master student from Wee Kim Wee School of Communication and Information
zhou0290@e.ntu.edu.sg

ABSTRACT

The Internet has been the most popular tool that many people use for acquire information since the early 2000s [12]. To retrieve information from unlimited amount of websites available today, people rely on search engines to find the information they need. For many programmers, IT practitioners or computer science students, Stack Overflow is known as one of the most useful and popular websites that they can exchange knowledge by asking questions or providing answers. In this study, we collect data from stack Overflow and build a search engine for python related information needs and we call it PyFlowSearch. User can search for Python related posts on the platform and get a few most relevant results presented in the form of question and answer pairs. Two applications are developed to extend the use of our platform.

KEYWORDS

Search Engine, PyFlowSearch, Information Retrival

1 INTRODUCTION

The Internet has been the most popular tool that many people use for acquire information since the early 2000s [12]. With the launch of World Wide Web (WWW) in the early 1990s, the number of websites has been increasing exponentially and today there are over one billion websites [8] . To retrieve information from all kinds of websites, people rely on search engines to find and provide the most related information. In fact, 89% [7] of on-line users use search engines when they access to the Internet.

In this study, we build a search engine for python related information needs and we call it PyFlowSearch. Our dataset is generated from Stack Overflow archived post dataset. Stack Overflow is a popular programming related question and answer (Q&A) website that users can ask questions or provide answers over the platform.

A brief introduction on IR techniques, search engines and Stack Overflow is given as the below. The processes and methodologies for build PyFlowSearch are described in 2.Methodology. We also build two extend applications basing on PyFlowSearch and we introduce them at 3.Application Extensions. 5.Conclusion summarize and conclude our research.

1.1 Information Retrieval

Information retrieval (IR) is a process of obtaining information from a collection of information resources in an accurate and efficient manner [14]. It has become more and more important as the growth of the web resources. In an era of data and information, IR plays a key role for supporting people to search for information and reduce information overload. IR begins with an user querying to information retrieval system, then the system learn the user's input, search through the index, retrieve the relevant documents and present to the user. Some of data can be structured while much more data today is unstructured and could be in many forms. The document can be text-based materials, images, videos and many other forms. The information need of an user can be found inside the documents, or an user can search for the documents themselves.

For most of the cases, there are more than one documents meet the user's information needs to some extent and IR system should be able to score and rank the results and eventually present the most relevant ones. Document scoring, ranking, document similarities and clustering are some of the popular research field of this field and have driven many academic works [3, 6, 13]. IR is a well established field of study and holds its own conferences and journals, some of the examples are ACM SIGIR Conferences, Transactions on Information Systems, Information Retrieval and etc.

There are many IR applications and some of those play in important role in our daily life. As students, we are frequent to the university digital library and use it IR system for accessing e-books or download journals. We could also access to some News channels to check on what is happening around. Instead of scanning through the websites, we may use their IR applications to directly locate the information we are interested in. The IR application we use the most could be web search engines, and in fact we use it many times a day whenever we have the access to Internet and we have something to ask in mind.

1.2 Search Engine

Search engine can be considered as one of the most complex and complicated type of the software in the world [14] which is capable

* "CI6226 Information Retrieval & Analysis" is a course offered by Nanyang Technological University, Wee Kim Wee School of Communication and Information.

to gather and analyze unlimited huge amount of information spread over all kinds of resources on the Internet; index and categorize the information; retrieve and display to users when required. There are many types of search engines, to name some, crawler-based search engine, human-directed search engine, meta-data search engine and vertical search engine. Among which, the most popular and prevalent one should be web crawler-based search engine. Google, Yahoo and Bing are some of its examples. It uses web crawlers to visit WWW web as many as possible and downloads the web pages into its own storage. It index all the pages that have been downloaded. When an user submits his or her query on the search engine interface, the query is preprocessed and match to the index. The hit pages are collected and then passed to the scoring and ranking model. The most relevant results are presented to the user. This type of search engine works quite well when the query keywords carry some specific meanings. While the words in the query are too generic, the results may not meet the user requirement.

human-directed search engine is driven by human directions that human beings are involved in categorizing and labeling the websites. Yahoo directory [15] is one of the examples of this type. It is more accurate in terms of generic queries as human beings are providing suggestions, but It is much more costly in both labeling time and human resources money. Giving the size of todayâĂŹs web, it is unrealistic to maintain such human-directed approach. Yahoo directory was closed up in the the year of 2014.

Meta-data search engine take use of the services provided by other search engines. It passes the user query to multiple other search engines and combine the their results. Dogpile is one of the representatives. Vertical search engine focus on some particular fields. For example, the travel agency tools or on-line shopping engines. They are targeting to specific websites and provide only domain related information.

1.3 Stack Overflow

Stack Overflow (https://stackoverflow.com/) is known as one of the most popular programming related question and answer (QA) websites that supports users to ask questions and other users of the community can provide answers. A tag of Stack Overflow is a keyword or label which group similar questions together and it can be programming language, software or applications, Operating Systems, IT related tools and etc. Some of the popular tags are JavaScript, Java, php, Android and etc. The most popular tag up to the writing point of time is JavaScript and contains over 1.3 million related questions.

1.3.1 Data Schema. The data of Stack Overflow is formated in a well-structure way. Stack Overflow supports many types of data for identify and manage the information on the website. In this paper, we only discuss the data format we use in our study and a table of which can be found below as Table 1.

Generally, a post in Stack Overflow can be a question, a answer or many other types. In this study, we only focus on questions and corresponding answers and we identify a post as either a question or an answer. In Stack Overflow, each post is assigned a unique "Id" as identifier. If the post is an answer, the associating question ID is recorded as "ParentId". For each of the questions, 0 to many answers are allowed and among all the answers, 1 best answer can be selected by the owner of the post as accepted answer and giving "AcceptedAnswerId". The owner and creation date of the post are stored in the system as "OwnerUserId" and "CreationDate". Each question have a title highlighting the theme of the question, as recorded as "Title". The "body" part is in free-text format to allow users give more detailed description. When Creating a question post, the owner need to select 1 tag or multiple tags which can best categorize or label the question, this information is recorded as "tag". How many views and answers for each question post are stored as "ViewCount" and "AnswerCount". Table 1 gives a brief summary on the data schema that we use.

Table 1: Data Schema

Type	Comments
Id	Uniquely assiged as post identifer
PostTypeId	1 for Question, 2 for Answer
AcceptedAnswerId	only present if PostTypeId is 1
ParentID	only present if PostTypeId is 2
CreationDate	Date of post creation
Score	
ViewCount	Counts of total views
Body	as rendered HTML, not Markdown
OwnerUserId	Owner of the post
Title	Title of the post
Tags	label of the post
AnswerCount	Counts of total answers

1.3.2 Dataset. The dataset we use in this study is generated from the original dataset: stackoverflow.com-Posts.7z which can be downloaded from https://archive.org/details/stackexchange by choosing the download option to be 7zip and selecting from the drop down list. After unzip the downloaded dataset, we generate our dataset by filtering all the questions using "python" tags. Any question that belongs to "python" tag is surfaced. After having all the questions, we search for all the answers associating to each of the questions. All the answers are then gathered. The resulted dataset reaches 2.6 GB in size, 715,344 posts for questions and 1,131,137 posts for answers.

2 METHODOLOGY

This chapter describes our methodology of designing and building the PyFlowSearch search engine. The main library used is an open-source information retrieval library provided by Apache Software Foundation which is called **Lucene** [10]. We first extract Python-related posts from Stack Overflow, then create Lucene documents and index on specific fields. A query parser and an index searcher are designed to support free-text queries. Finally, the performance of PyFlowsearch is evaluated to ensure the query results generally meet the users information needs.

2.1 Extract Python-Related Posts

As our search engine is designed for searching Python-related posts on Stack Overflow, we need to distinguish different posts which

are associated with different programming languages or topics and extract all the posts that are related to Python programming.

We used an open source Java library called "dom4j" [5] to efficiently read and parse the large XML file and analyze the data structure of each post. One efficient way to distinguish posts which related to different programming languages is to look at the "Tags" field contained in the question posts. The "Tags" field contains the most relevant words that describe the question, for example "python", "c++", "c", "Java", etc. We have investigated that almost all the python-related questions contain "python" in their tags, so we designed an extractor that extracts all the Python-related posts by first extracting all the question posts that contain the "python" tag, then find all the corresponding answer posts which have the "ParentID" equal to the "Id" of the extracted question posts.

The extractor will filter out non-relevant posts and reduce the size of the XML document significantly. In our search engine, instead of searching the whole dataset which contains more than thirty million posts (most of posts are irrelevant to Python programming), we treat the extracted posts as our search base. This will improve the indexing and search time significantly. Besides, the precision of the search result will also be improved to some degree as the non-relevant posts are filtered out.

2.2 Indexing

2.2.1 Identify "document" to be indexed.

A document is a basic unit for indexing and search. In order to construct an index that allows efficient searching, we need to take into account our expected query results and their presentations to users. In a nutshell, if we find a match between the user's query and the post, the fields that we want to present to the user are: question title, question body (with code segments presented separately), question tags, and corresponding answer bodies (with code segments presented separately). Besides, some additional fields of the post will also be retrieved to facilitate queries and answers ranking.

As questions and answers are link with the "Id" tag of question and the "ParentID" tag of answer, it is easy to find the questions and the corresponding answers and vice versa. Therefore, our goal is to find matches between user's query and the fields of the post describe above. In this case, the "document" to be indexed is basically a post on Stack Overflow which can be either a question or an answer. In Lucene's terminology, a document consists of a set of fields that we want to index and the fields we want to retrieve when search hits on the document. This means the post with "useful" fields is treated as a document, and we will discuss these fields in Section2.2.4

2.2.2 Linguistic processing.

In order to support free-text key word queries effectively and efficiently, we need to do some linguistic processing on text fields.

Firstly, we split the text string into tokens before indexing and querying by applying the StandardTokenizer in Lucene. The tokens are then treated as the candidates for indexing. The StandardTokenizer applies a grammar-based tokenizer constructed with JFlex, a Java lexical analyzer generator, and also implements the Word Break rules from the Unicode Text Segmentation algorithm [11].

Secondly, we remove the stop words from token stream that most commonly appear in the text and have little semantic content, such as "to", "the", "a","be", etc. We use the default list of the stop words specified in the StopFilter of Lucene. One important note is that we don't apply this stop word list on the code segment of the field, we treat the code segment separately from the texts and we will discuss it in the next section.

Thirdly, we apply a LowerCaseFilter to the token stream and convert all the token texts into lower case. This will improve the recall of the query as users will use lowercase regardless of the correct capitalization and also reduce the index size.

Finally, we apply the StandardFilter of Lucene to normalize tokens into unique terms in our IR system dictionary. For example, the periods and hyphens within a token text are removed: U.S.A.→USA, anti-bacteria→antibacteria. The normalization of tokens will ensure the query words and the corresponding indexed text to be the same form so that matches will occur despite of subtle changes of the token. This is also a method to improve the recall of the information retrieval.

2.2.3 Code segment processing.

As the code segment in the text field is written in Python programming language which is much more different from natural language, we cannot treat it the same as normal text in the text field. For example, the word such as "def", "import", "elif" is very frequent in Python code but less frequent in natural language.

In our search engine, we designed a different way of processing and indexing the code segment. We first wrote an HTML parser using "jsoup" [9] library in Java to split the code segment and normal text, and we treat the code segment as a separate field in the document which is named as "Code" field. We then designed another analyzer called "CodeAnalyzer" to process the "Code" field before indexing the code words. In CodeAnalyzer, the code string is first tokenized as separate tokens and then all the tokens are converted to lowercase. This is achieved by applying the LowerCaseTokenizer in Lucene. Similar to the stop words in natural language, there also many "keywords" in Python language that appear most frequently in the code segment but provide less functional meaning to the overall code segment. The stop word list for Python language is shown in Table 2. We treat such keywords in Python as stop words and discard them before indexing. This is achieved by defining a stop word list for Python and passing this list to a new StopFilter in Lucene. The code snippets of CodeAnalyzer is shown below:

```
public class CodeAnalyzer extends Analyzer {
    private final List<String> pythonStopWords =
        Arrays.asList("False", "None", "True", "and",
        "as", "assert", "break","class", "continue",
        "def", "del", "elif", "else", "except",
        "finally", "for", "from", "global", "if",
        "import", "in", "is", "lambda", "nonlocal",
        "not", "or", "pass", "raise", "return", "try",
        "while", "with", "yield", "print" );
    public CodeAnalyzer() { super(); }
    @Override
    protected TokenStreamComponents
        createComponents(String fieldName) {
    Tokenizer tokenizer = new LowerCaseTokenizer();
    return new TokenStreamComponents(tokenizer, new
        StopFilter(tokenizer, new
        CharArraySet(pythonStopWords, false)));}}
```

Then we apply the PerFieldAnalyzerWrapper in Lucene to wrap the StandardAnalyzer and the CodeAnalyzer. The two analyzers

Table 2: Stop word list in Python programming language

False	class	finally	is	return	None
continue	for	print	try	True	def
from	nonlocal	while	and	del	global
not	with	as	elif	if	or
assert	else	import	pass	break	except
yield	in	raise			

are used to process the normal text and the "Code" field respectively by using hash mapping, as shown below:

```
Map<String, Analyzer>  codeSpecialAnalyzer = new
     HashMap<String, Analyzer>();
codeSpecialAnalyzer.put("Code", new CodeAnalyzer());
PerFieldAnalyzerWrapper wrapper = new
     PerFieldAnalyzerWrapper(new StandardAnalyzer(),
     codeSpecialAnalyzer);
```

2.2.4 *Indexer:*

As discussed in Section 2.2.1, a document in Lucene contains the fields to be indexed, as well as the fields we want to retrieve for either facilitating query processing or presenting the result to the user. This section describe how a document is constructed and how index is created in PyFlowSearch engine.

The most important fields in the document are the fields to be indexed by the indexer, because the search engine performs searching of user's query based on the indexed filed. We first determine the fields to be indexed in a post document. The common fields to be indexed in both question post and answer post are: "Id", "Body" and "Code"; In addition, we also index the "Title" and "Tags" fields of question post and the "ParentID" field of answer post in order to find a better matching between user's query and the desired posts. Next, we determine others fields which need to be stored in the Lucene document. These fields provide useful information during query processing and result presenting. In PyFlowSearch engine, all the content in the indexed fields described above are stored. Besides, the "Score" field of both question and answer posts, the "AnswerCount" and "AcceptedAnswerId" of question post are also stored to facilitate answer ranking.

In order to get the field names and their corresponding values from the original XML file, we write a XML parser by applying "dom4j" library and hash mapping method to parse the posts in the XML file. After applying the `XMLParser.ParsePost()` method to a post, a HashMap that matches every field name to its value in the field will be generated. Besides, we write a `Post` class that can easily get the value of any field of a post. For example, if we want to get the value of the "body" field of a post, we can just call the method: `post.getBody()`.

After paring the XML file, We use the `Document` class of Lucene to create the document for each post of Stack Overflow that contains all the fields described above. The code snippet below shows how we construct a Lucene document:

```
private static Document getDocument(Post p) {
    Document doc = new Document();
    // Common field
    doc.add(new IntPoint(PostField.Id.toString(),
         p.getId()));
    doc.add(new StoredField(PostField.IdCopy.toString(),
         p.getId()));
    doc.add(new StoredField(PostField.Score.toString(),
         p.getScore()));
    doc.add(new TextField(PostField.Body.toString(),
         HTMLParser.ParseText(p.getBody()), YES));
    doc.add(new TextField(PostField.Code.toString(),
         HTMLParser.ParseCode(p.getBody()), YES));
    // Other fields to store/index
    //...
    return doc;}
```

After constructing the Lucene document, we can add the document to the index by applying Lucene's `IndexWriter`. An IndexWriter creates and maintains an index for the documents, it will create index based on the indexed fields and also maintain the stored fields in the index. We set the OpenMode of `IndexWriter` to be `"CREATE_OR_APPEND"`, which means the `IndexWriter` will create a new index if there is not already an index at the provided path and otherwise open the existing index. Then we use the `PerFieldAnalyzerWrapper` which contains the `StandardAnalyzer` and `CodeAnalyzer` to configure the `IndexWriter`. The code snippet below shows the initialization of the `IndexWriter` :

```
PerFieldAnalyzerWrapper wrapper = new
     PerFieldAnalyzerWrapper(new StandardAnalyzer(),
     codeSpecialAnalyzer);
IndexWriterConfig config= new IndexWriterConfig(wrapper);
config.setOpenMode(
      IndexWriterConfig.OpenMode.CREATE_OR_APPEND);
try {
      writer = new IndexWriter(indexDirectory, config);
     } catch (IOException e) {
      System.out.println("IndexWriter initialization
           Error."); }
```

After the initialization of `IndexWriter`, we can add the document of each post to the index by calling the method `addDocument()`:

```
Document doc = getDocument(new
     Post(XMLParser.ParsePost(line)));
writer.addDocument(doc);
```

After finish scanning all the posts in the XML file, the final index is created and stored in our predefined directory "indexDirectory".

2.3 Queries

2.3.1 *Free text keyword queries:*

As mention in the index part, indices are created on these fields - "Body(text in original body)", "Title","Code(code in original body)" and "Tags". So we can only search on these fields. For free text keyword query on all these fields, IndexReader and IndexSearcher need to be initialized. An IndexReader object is initialized with the index store directory and IndexSearcher is initialized with an IndexReader object. They are initialized with the code snippets below;

```
postReader=DirectoryReader.open(FSDirectory.open(
Paths.get(utils.Paths.POSTINDEXPATH)));
postSearcher = new IndexSearcher(postReader);
```

The directory where the index store is defined in the indexer part.

The code segment in "Body" field is processed separately with a `CodeAnalyzer` and other fields like "Tags" and "Title" are processed with default `StandardAnalyzer`, so when it comes to search on these fields, field need to be associated with its corresponding Analyzer. The customized analyzer is defined by a `PerFieldAnalyzer`

Wrapper object. And it is initialized with a Map. The customized analyzer for these fields is initialized like the code snippets below.

```
Map<String, Analyzer> customizeAnalyzer = new
    HashMap<>();
customizeAnalyzer.put("Code", new CodeAnalyzer());
PerFieldAnalyzerWrapper wrapper = new
    PerFieldAnalyzerWrapper(new StandardAnalyzer(),
    customizeAnalyzer);
```

For query on each field, it is taken as equally important by default, which means the weight of each field is equal. But we want these documents matching with *Code*, *Title* and *Tags* to rank higher in the results. Because when these short field matches our query, the document is more relevant with the query. So we implemented a customized weight for each field, try to increase the weight of the field that we attach more importance to. The customized weight is implemented in the following code snippets.

```
//  Weight customization
Map<String, Float> boosts = new HashMap<>();
boosts.put(PostField.Body.toString(), 1.0f);
boosts.put(PostField.Title.toString(), 3.0f);
boosts.put(PostField.Code.toString(), 2.0f);
boosts.put(PostField.Tags.toString(), 3.0f);
```

After customizing analyzer, a `MultiFieldQueryParser` is built to parse the original query string and form a `Query` object. A `String` array is defined for the fields need to query. In free text query, all fields are considered. A `Query` object is constructed like the code snippets below.

```
List<String> list = new ArrayList<>();
list = Arrays.asList(PostField.Body.toString(),
    PostField.Title.toString(),
    PostField.Code.toString(),
    PostField.Tags.toString());
String[] queryFields = list.toArray(new
    String[list.size()]);
MultiFieldQueryParser parser = new
    MultiFieldQueryParser(queryFields, wrapper, boosts);
parser.setDefaultOperator(QueryParser.Operator.AND);
Query query=parser.parse(queryStr);
```

`queryFields` is a `String` array of all the fields. `wrapper` is the customized analyzer, and `boosts` is the customized weight for each field. `QueryParser.Operator.AND` means the query will return the result that matches more of the terms in the query. To sum score of each term in all the fields.

When a `Query` object is initialized, `postSearcher.search()` is called to search the document matches the query. Also the top N results can be configured like `postSearcher.search(query, hitNum)`, `hitNum` is the N and it is configurable.

2.3.2 Support phrase query:

Phrase query is a query that matches documents containing a particular sequence of terms. The input query string like "python numpy" will be treated as a phrase query. The result of the query is document that contains the combination of the two term in this position.

After reading the documentation of lucene about `PhraseQuery` and `MultiFieldQueryParser`[2], we found that phrase query is already supported by `MultiFieldQueryParser`. No more implementation is needed. Just indicate the phrase by using double quotation marks when input the query.

2.3.3 Queries on specific fields:

To specific the fields of the query, we only need to modified the `queryFields String` array in the above step. Process the input from the user and parse it into corresponding fields. The fields can be specified with the code snippets below.

```
List<String> list = new ArrayList<>();
if (fields.contains("1")) {
    list.add(PostField.Body.toString());
}
if (fields.contains("2")) {
    list.add(PostField.Title.toString());
}
if (fields.contains("3")) {
    list.add(PostField.Code.toString());
}
if (fields.contains("4")) {
    list.add(PostField.Tags.toString());
}
String[] queryFields = list.toArray(new
    String[list.size()]);
MultiFieldQueryParser parser = new
    MultiFieldQueryParser(queryFields, wrapper, boosts);
parser.setDefaultOperator(QueryParser.Operator.AND);
Query query = parser.parse(queryStr);
```

`fields` is the input string from the user to specify the query fields.

2.4 Query Results

As described in last section, the query string is parsed by the Lucene `MultiFieldQueryParser` and is analyzed by the `PerFieldAnalyzer` to ensure the query string is processed the same way as the indexed document. Then the `IndexSearcher` uses the parsed query to search on the index and return a `Hits` object that contains documents satisfying the query. This section will explain the answers ranking mechanism and how the results are presented to the user in PyFlowSearch search engine.

2.4.1 Answers ranking mechanism:

During the searching of each indexed document, the search engine tries to find matches between the indexed fields in the document and the query string, and then compute a score that measures the similarity between the document and user's query. The scoring mechanism is formulated by the following formula [1]:

$$score(q,d) = coord(q.d) \cdot queryNorm(q) \cdot \sum_{t\ in\ q} (tf(t) \cdot idf(t)^2 \cdot t.getBoost())$$

In the above formula, q represents the query and d represents the document.The following list gives the detail of the components of this scoring mechanism:

(1) **coord(q,d)** is a score factor based on how many of the query terms are found in the specified document. Typically, a document that contains more of the query's terms will receive a higher score than another document with fewer query terms.
(2) **queryNorm(q)** is a normalizing factor used to make scores between queries comparable.This factor does not affect document ranking as all ranked documents are multiplied by the same factor.
(3) **tf(t)** is called Term Frequency which is the number of times that term t appears in document d. A document which has

more occurrences of the terms in a query will receive a higher score.

(4) **idf(t)** is called Inverse Document Frequency. The document frequency of a term t is the number of documents in which term t appears. By applying $idf(t)$, a term which is rare in the whole collection gives a higher contribution to the total score.

(5) **t.getBoost()** is a search time boost of term t in the query q. Recall from Section 2.3.1, the boost of t is specified in the customized weight for each field. A term appearing in a higher weighted field will give a higher score than appearing in a lower weighted field.

After searching all the documents in the index file, each document will have a score computed by the scoring mechanism above. The documents are then ranked by $score(q, d)$ from the highest to the lowest.

2.4.2 Result presentation:

In PyFlowSearch, only top N matching documents which have higher rankings will be retrieved and presented to the user. N is a configurable integer that is specified by the user to allow users to decide how many matching documents they want to see in the query results.

The query results will be presented in decreasing order of the score $score(q, d)$. And for each query result, the question and all the corresponding answers will be presented to the user. To be more specific, if the matching document is a question post, then all the corresponding answers will be retrieved by searching the "ParentID" of all the answers; If the matching document is an answer post, then its corresponding question will be retrieved and all the other answers of this question will also be retrieved by using "ParentID". In both cases, the answers will be ranked and presented in the decreasing oder of their scores. Besides, the IDs of the posts and the search score of a query result will also be presented. Figure 1 demonstrates the first result of the query: *numpy array*.

Figure 1: Query result presentation demonstration.

```
Question1 ID:39622533
Question1 search Score:85.05664825439453
Question1 Title:numpy array as datatype in a structured array?
Question1 Body:I was wondering if it is possible to have a numpy.array as a datatype in
Question1 Code:import numpy  raw_data = [(1, numpy.array([1,2,3])),            (2, nu
Question1 Tags:python arrays numpy structured-array
   Answer1 ID:39622632
   Answer1 Score:1
   Answer1 Body:Yes, you can create compound fields that look like arrays within the st
   Answer1 Code:import numpy as np raw_data = [(1, np.array([1,2,3])),             (2,
```

2.5 Performance Evaluation

In order to evaluate the performance of PyFlowSearch engine and ensure that it is capable of processing different kink of queries and retrieving the desired results, we conduct 10 sample queries on PyFlowSearch. The 10 sample queries consists of:

(1) 3 free text keyword queries on indexed fields:
- Q1: How to keep keys/values in same order as declared in Python's dictionary?
- Q2: How to clone or copy a list?
- Q3: Sort a Python dictionary by value.

(2) 2 free text keyword queries on indexed fields with phrases specified by double quotation marks:
- Q4: Python UnboundLocalError for "local variable".
- Q5: Append "numpy array" to a "pandas dataframe".

(3) 3 free text keyword queries on specific fields:
- Q6: Explain Python's slice notation. ("Title" field)
- Q7: Pass a variable by reference. ("Body" field)
- Q8: np.array(). ("Code" field)

(4) 2 free text keyword queries on specific fields with phrases specified by double quotation marks:
- Q9: Decode "HTML" entities in Python string. ("Body" field)
- Q10: How to install "IPython Notebook"? ("Title" field)

For each sample query, we retrieve top-10 query results and each result is evaluated manually by two judges to decide whether it is relevant or not [1]. The precision of each sample query is calculated using the following formula:

$$precision = \frac{No.\ of\ relevant\ results}{total\ No.\ of\ results\ retrieved}$$

Each judge will give a final precision which is the average precision of these 10 sample queries [2]. The final evaluation score is the average value of the final precisions from two judges. Table 3 below shows the evaluation results. The Kappa measure [4] for inter-judge agreement is 0.8125 which is considered to be good agreement among judges.

Table 3: Evaluation results for 10 sample queries

Sample Queries	Judge 1			Judge 2		
	R	I	Prec	R	I	Prec
Q1	7	3	0.7	8	2	0.8
Q2	6	4	0.6	7	3	0.7
Q3	6	4	0.6	7	3	0.7
Q4	10	0	1	10	1	1
Q5	7	3	0.7	8	2	0.8
Q6	8	2	0.8	8	2	0.8
Q7	7	3	0.7	8	2	0.8
Q8	9	1	0.9	9	1	0.9
Q9	9	1	0.9	9	1	0.9
Q10	8	2	0.8	9	1	0.9
Ave Prec			0.77			0.83
Evaluation Score	0.8					

R:Relevant; I:Irrelevant; Prec: Precision

Our final evaluation score (precision) of PyFlowSearch engine is 0.8 and it is considered to be performing very well and some query results of certain queries are even better than the Stack Overflow official website.

[1] Relevant is regarded as the query result generally meets the user's information needs.
[2] We didn't evaluate the recall and F score because we don't know the ground truth of the total no. of relevant document in the whole dataset.

3 APPLICATION EXTENSIONS

In this study, we develop two applications to extend the usage of PyFlowSearch. The dataset and methodology of building the two applications are remain no change; extra data schema is incorporated for supporting more information need. Detailed introduction on the two applications,the code representation and the results are giving.

3.1 Find the users who provide the most accepted answers

3.1.1 *Introduction.*

This application aims to find the top 20 users who provide the largest number of answers that has been selected as "Acceptd Answer".

With this application, user can filter out the answer contributors who are both well-skilled in Python related field and willing to help others. Such information can be used for all kinds of purposes. for example, users who need some help can use the system to surface some user IDs. By having the information, they can have some ideas of who to ask for help. For the talents been surfaced, our application can be expected to enhance their importance and popularity over the platform. They can be more encouraged by not only the acceptance of their answers, but also by the attentions from all kinds of other users using our application.

3.1.2 *Methodology.*

In this application, we re-index the dataset to include "OwnerUserID" for accepted answers. This is to surface the information of users whose answers have been selected and calculate how many times in total that have been selected. The code snippets for re-index is as below:

```
private static Document getDocument(Post p) {
        Document doc = new Document();
        doc.add(new
             IntPoint(PostField.Id.toString(),p.getId()));
        doc.add(new
             StoredField(PostField.OwnerUserId.toString(),
             p.getOwnerUserId()));
        return doc;}
```

The code snippets for scanning the data and calculating is as below:

```
 InputStreamReader read = new InputStreamReader(new
     FileInputStream(file));
BufferedReader bufferedReader = new BufferedReader(read);
String line = null;
while((line=bufferedReader.readLine()) != null) {
Post p = new Post(XMLParser.ParsePost(line));
if (p.getAcceptedAnswerId() != 0) {
Query query = IntPoint.
     newExactQuery(PostField.Id.toString()
     ,p.getAcceptedAnswerId());
ScoreDoc[] docs=reIndex1Searcher.search(query,
     100).scoreDocs;
if (docs.length != 0) {
        for (ScoreDoc doc : docs) {
        Document document=reIndex1Searcher.doc(doc.doc);
        String ownerID = document.get
             (PostField.OwnerUserId.toString());
        if (Integer.parseInt(ownerID) != 0) {
                Integer count = map.get(ownerID);
                map.put(ownerID, (count == null) ? 1 :
                     count + 1);}}}}
```

3.1.3 *Result.*

The results of the top 20 users are found as shown in Figure 2. From the figure, user can easy find out who are the most well-skilled and active users who are willing to help.

Figure 2: Top 20 user list

```
Top 20 user
Rank 1 UserId: 100297 Accepted answer number: 11619
Rank 2 UserId: 190597 Accepted answer number: 4006
Rank 3 UserId: 771848 Accepted answer number: 3681
Rank 4 UserId: 104349 Accepted answer number: 2923
Rank 5 UserId: 2225682 Accepted answer number: 2493
Rank 6 UserId: 908494 Accepted answer number: 2438
Rank 7 UserId: 95810 Accepted answer number: 2325
Rank 8 UserId: 2141635 Accepted answer number: 2310
Rank 9 UserId: 20862 Accepted answer number: 2229
Rank 10 UserId: 7432 Accepted answer number: 1974
Rank 11 UserId: 2901002 Accepted answer number: 1880
Rank 12 UserId: 748858 Accepted answer number: 1775
Rank 13 UserId: 846892 Accepted answer number: 1682
Rank 14 UserId: 704848 Accepted answer number: 1617
Rank 15 UserId: 4279 Accepted answer number: 1389
Rank 16 UserId: 487339 Accepted answer number: 1388
Rank 17 UserId: 279627 Accepted answer number: 1193
Rank 18 UserId: 1427416 Accepted answer number: 1173
Rank 19 UserId: 3001761 Accepted answer number: 1095
Rank 20 UserId: 795990 Accepted answer number: 1092
```

3.2 Find the best time to post a question

3.2.1 *Introduction.*

This application aims to find the best time period to post a question in a week. We calculate and analyze the most active hour in a week and day in a week that gathers the most number of answers.

Hour is counted as Hour:Minute the format of HH:MM. For each day, the start hour is considered 00:00and end 23:59. We group and count all the answers posted at 00:00-00:59 as hour 1, and 01:00-01:59 as hour 2, and so on. In total, we have 24-hour groups and the total number of answers posted at each hour group is counted. This is to find out which hour in a week gather the most number of answers at the Stack Overflow platform. With this function, user can understand the best hour in a week to ask his or her question.

Week is counted as Monday, Tuesday, Wednesday, Thursday, Friday, Saturday and Sunday. In total, we have 7-day per week and we calculate the total number of answers posted at each day to filter out the day in a week that gathers the most number of answers at the website. With this function, user can understand the best day in a week to ask his or her question.

The rationale behind this is, for most of users, we have our routine tasks such as work or study at weekdays, have meals, sleep or do excise. we use only a few minutes or half an hour to scan through the websites. We go through only few pages of the questions to see if any of those we can help. At Weekends, we could have other plans that we may or may not have more free time for spending on Stack Overflow. Generally, it is not likely for one person to check out all the questions listed on the website. Hence, it is important for an user who asks for help to make sure that his or her question lists within the few pages at the time that as many people as possible visiting the websites and ready for providing some answers. With the application, user can easily understand what time or day in a week may attract more users to give answers.

3.2.2 *Methodology.*

In this application, "CreationDate" data field is processed to extracted to the week,day and hour information. The code snippets is as below:

```
private static Document getDocument(Post p) throws
     ParseException {
       Document doc = new Document();
       doc.add(new
            IntPoint(PostField.CreateHour.toString(),
            getHour(p.getCreationDate())));
```

```
        doc.add(new
             IntPoint(PostField.CreateDay.toString(),
             getDay(p.getCreationDate())));
        return doc;}
```

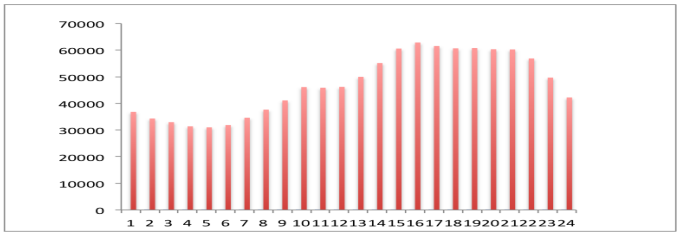
The code snippets for extracting calculating the week, day and hour is as below:

```
for (int i = 0; i < 24; i++) {
        Query query =
             IntPoint.newExactQuery(PostField.CreateHour.
             toString(), i);
        ScoreDoc[] docs;
        docs = reIndex2Searcher.search(query,
             10000000).scoreDocs;
    result1PrintWriter.println("Hour" + i + " Answers
         Count:" + docs.length);
        System.out.println("Hour" + i + " Answers
             Count:" + docs.length);
}
for (int i = 1; i < 8; i++) {
        Query query =
             IntPoint.newExactQuery(PostField.CreateDay.
             toString(), i);
        ScoreDoc[] docs;
            docs = reIndex2Searcher.search(query,
                100000000).scoreDocs;
        result2PrintWriter.println("Day:" + days.get(i -
             1) + " Answers Count:" + docs.length);
        System.out.println("Day:" + days.get(i-1) + "
             Answers Count:" + docs.length);}
```

3.2.3 Results.

The result for the the amount of answers in each hour is shown in Figure 3 (1 present the time from 00:00 - 00:59, and so forth). From which, we can understand that the peak amount of answers are provided within the hour of 15:00 - 15:59. Relatively high answering rate is kept from 16 to 22. After the hour 22, answers rate largely drops. If an user is trying to ask a question, the hour 16 to 22 should be the best time for posting it to the platform.

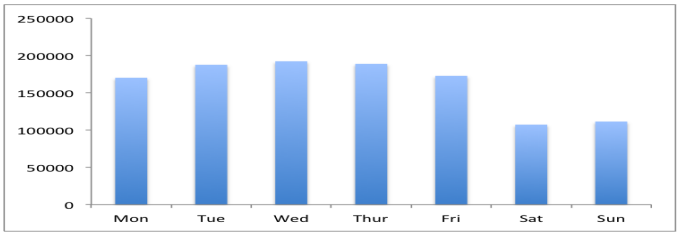
Figure 3: Total answers for each hour in a day



3.2.4 Demonstration.

The results of answering rate in a week is presented as Figure 4:

Figure 4: Total answers for each day in a week



From Figure.4, we can observe that the answering rate at weekends are largely less than weekday. Relatively more answers provided on Tuesday to Thursday comparing to Monday and Friday in working days. This suggests an user to not posting his or her question on weekends and among weekdays, Tuesday to Thursday is the target days.

4 CONCLUSION

In this study, we designed and built a search engine: PyFlowSearch for python-related information retrieval.

We first extract all the Python-related posts from Stack Overflow, then create index on the post and search user's queries on the index using Lucene library and some other open source libraries. The PyFlowSearch search engine we designed is able to search free-text keyword queries on indexed fields or any specific fields, it also support phrase query by putting the phrase into double quotation marks. The performance of PyFlowSearch is evaluated, the overall precision of 10 sample queries is found to be 0.8, which is considered to be very good and the search engine can generally meets users' information needs.

We also developed two applications to extend the use of PyFlowSearch. A user can find the users who provide the most accepted answers and the best time to post a question. The results suggest that weekdays have more response then weekends and from 4PM to 10 PM in a day can possibly receive more answers.

REFERENCES

[1] Apache Software Foundation 2012. *TFIDFSimilarity (Lucene 4.0.0 API).* Apache Software Foundation. https://lucene.apache.org/core/4_0_0/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html

[2] avajava.com 2014. *How do I query for words near each other with a phrase query? - Web Tutorials - avajava.com.* avajava.com. http://www.avajava.com/tutorials/lessons/how-do-i-query-for-words-near-each-other-with-a-phrase-query.html.

[3] Ricardo Baeza-Yates, Liliana Calderón-Benavides, and Cristina González-Caro. 2006. The intention behind web queries. In *International Symposium on String Processing and Information Retrieval.* Springer, 98–109.

[4] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

[5] dom4j 2005. *dom4j: the flexible XML framework for Java.* dom4j. http://dom4j.sourceforge.net/dom4j-1.6.1/index.html

[6] Norbert Fuhr. 2008. A probability ranking principle for interactive information retrieval. *Information Retrieval* 11, 3 (2008), 251–265.

[7] infoplease 2017. *Most Popular Internet Activities.* infoplease. http://www.infoplease.com/ipa/A0921862.html.

[8] Internet Live Stats 2017. *Total number of Websites - Internet Live Stats.* Internet Live Stats. http://www.internetlivestats.com/total-number-of-websites/.

[9] Jonathan Hedley 2017. *jsoup: Java HTML Parser.* Jonathan Hedley. https://jsoup.org/

[10] Michael McCandless, Erik Hatcher, and Otis Gospodnetic. 2010. *Lucene in Action, Second Edition: Covers Apache Lucene 3.0.* Manning Publications Co., Greenwich, CT, USA.

[11] The Unicode Consortium 2017. *Unicode Text Segmentation.* The Unicode Consortium. http://unicode.org/reports/tr29/.

[12] UCLA Center for Communication Policy 2003. *The ucla internet report surveying the digital future year three.* UCLA Center for Communication Policy. http://www.ccp.ucla.edu/pdf/UCLA-Internet-Report-Year-Three.pdf.

[13] Cornelis Joost van Rijsbergen. 1970. A clustering algorithm. *Computer Journal* 13, 1 (1970), 113–115.

[14] Wikipedia. 2017. Information retrieval — Wikipedia, The Free Encyclopedia. (2017). https://en.wikipedia.org/w/index.php?title=Information_retrieval&oldid=770896502 [Online; accessed 2-April-2017].

[15] Wikipedia. 2017. Yahoo! Directory — Wikipedia, The Free Encyclopedia. (2017). https://en.wikipedia.org/w/index.php?title=Yahoo!_Directory&oldid=758414502 [Online; accessed 2-April-2017].