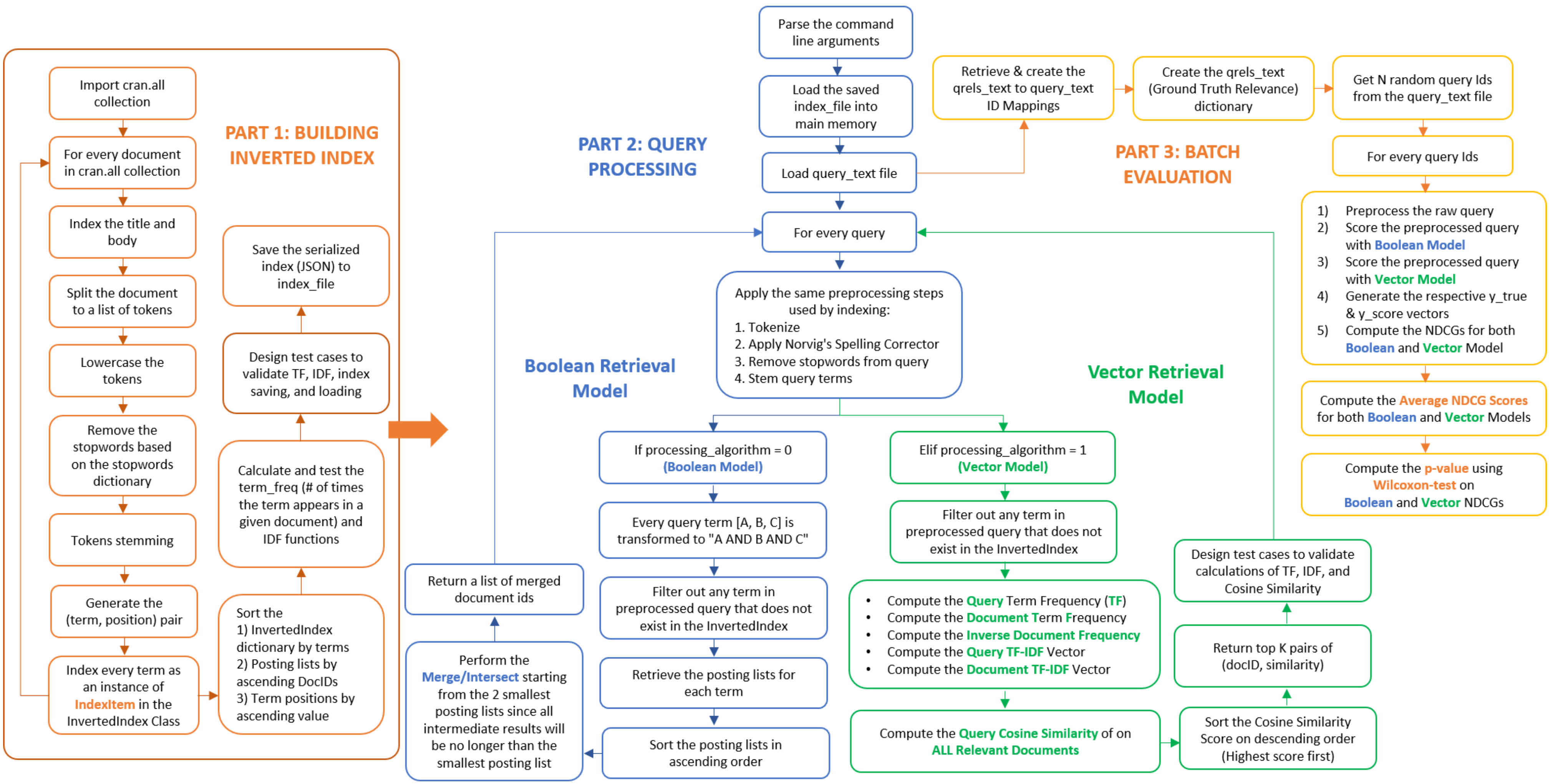
Spring 2019 - Information Retrieval (CS-7800-90)

Simple Search Engine

GitHub Repository: https://github.com/Joeyipp/simple-search-engine

Name: Hong Yung YIP (U00881791)

Design Flowchart & Implementation Pipeline



Instructions and Sample Commands:

1. The program is implemented and tested on **Python 2.7**.
2. Included in the zipped source code directory is a **requirements.txt** that lists the external dependencies used. Do "**pip install -r requirements.txt**" prior to running the scripts.

Below are some sample commands to run the scripts and their respective outputs:

PART 1: BUILDING INVERTED INDEX

```
python index.py cran.all index_file
```

Part 1 indexes all 1400 documents in the cran.all collection, builds the index, and saves the index (serialized as JSON) into the index_file.

```
Indexing document 1383
Indexing document 1384
Indexing document 1385
Indexing document 1386
Indexing document 1387
Indexing document 1388
Indexing document 1389
Indexing document 1390
Indexing document 1391
Indexing document 1392
Indexing document 1393
Indexing document 1394
Indexing document 1395
Indexing document 1396
Indexing document 1397
Indexing document 1398
Indexing document 1399
Indexing document 1400

Total documents indexed: 1400
Saving to disk...

InvertedIndex successfully saved to index_file

Done
```

PART 2: QUERY PROCESSING

The general steps performed when each command is run,

1. Load the saved index from Part 1
2. Rebuild the index in main memory
3. Process the query(s) based on the selected processing_algorithm (0 for Boolean, 1 for Vector)
4. Display the results as the number of matching documents with their respective DocIDs (Boolean Model), and ranked cosine scores for Vector Model.

Boolean Retrieval Model (Single Query)

`python query.py index_file 0 query.text 284`

```
user@ubuntu:~/Desktop/simple-search-engine$ python query.py index_file 0 query.text 284
Loading from disk...

InvertedIndex successfully loaded to memory from index_file

QueryID: 284    #Docs: 2        DocIDs: [856, 857]
```

Boolean Retrieval Model (ALL Queries)

`python query.py index_file 0 query.text batch`

```
user@ubuntu:~/Desktop/simple-search-engine$ python query.py index_file 0 query.text batch
Loading from disk...

InvertedIndex successfully loaded to memory from index_file

QueryID: 27     #Docs: 5        DocIDs: [170, 329, 439, 798, 1313]
QueryID: 29     #Docs: 1        DocIDs: [462]
QueryID: 66     #Docs: 5        DocIDs: [186, 283, 294, 522, 1352]
QueryID: 106    #Docs: 1        DocIDs: [388]
QueryID: 111    #Docs: 1        DocIDs: [540]
QueryID: 112    #Docs: 5        DocIDs: [25, 304, 329, 540, 572]
QueryID: 142    #Docs: 4        DocIDs: [84, 283, 329, 1104]
QueryID: 156    #Docs: 1        DocIDs: [1040]
QueryID: 158    #Docs: 1        DocIDs: [75]
QueryID: 201    #Docs: 1        DocIDs: [1025]
QueryID: 202    #Docs: 1        DocIDs: [1019]
QueryID: 204    #Docs: 3        DocIDs: [1016, 1019, 1028]
QueryID: 226    #Docs: 2        DocIDs: [1063, 1082]
QueryID: 227    #Docs: 1        DocIDs: [1088]
QueryID: 261    #Docs: 5        DocIDs: [320, 321, 322, 476, 527]
QueryID: 273    #Docs: 1        DocIDs: [548]
QueryID: 284    #Docs: 2        DocIDs: [856, 857]
QueryID: 296    #Docs: 3        DocIDs: [730, 733, 734]
QueryID: 349    #Docs: 1        DocIDs: [25]
QueryID: 355    #Docs: 1        DocIDs: [1400]
```

Vector Retrieval Model (Single Query)

`python query.py index_file 1 query.text 284`

```
user@ubuntu:~/Desktop/simple-search-engine$ python query.py index_file 1 query.text 284
Loading from disk...

InvertedIndex successfully loaded to memory from index_file

Top K Pairs? 10

QueryID: 284
DocID: 390      Score: 0.941
DocID: 856      Score: 0.938
DocID: 766      Score: 0.927
DocID: 858      Score: 0.922
DocID: 285      Score: 0.911
DocID: 486      Score: 0.911
DocID: 899      Score: 0.910
DocID: 1008     Score: 0.908
DocID: 391      Score: 0.908
DocID: 948      Score: 0.908
```

Vector Retrieval Model (ALL Queries)

`python query.py index_file 1 query.text batch`

```
QueryID: 360
DocID: 1322     Score: 0.626
DocID: 259      Score: 0.588
DocID: 1312     Score: 0.553
DocID: 405      Score: 0.511
DocID: 283      Score: 0.498
DocID: 317      Score: 0.497
DocID: 529      Score: 0.472
DocID: 1316     Score: 0.467
DocID: 1286     Score: 0.466
DocID: 976      Score: 0.460

QueryID: 365
DocID: 1188     Score: 0.829
DocID: 1345     Score: 0.762
DocID: 1380     Score: 0.734
DocID: 1218     Score: 0.728
DocID: 1344     Score: 0.688
DocID: 225      Score: 0.667
DocID: 164      Score: 0.636
DocID: 1347     Score: 0.632
DocID: 77       Score: 0.632
DocID: 163      Score: 0.629
```

**Due to space constraints, a partial screenshot of the FULL output is attached. Run the command on the script for FULL output.*

PART 3: AVERAGE NDCGs EVALUATION

The average **NDCGs** for the Boolean and Vector Model based query processing results are computed and **Wilcoxon** test is performed for the p-value.

Average NDCG evaluation for 10 randomly selected queries from query.text

```
python batch_eval.py index_file query.text qrels.text 10
```

```
user@ubuntu:~/Desktop/simple-search-engine$ python batch_eval.py index_file query.text qrels.text 10
Loading from disk...

InvertedIndex successfully loaded to memory from index_file

Processing QueryID: 252
Processing QueryID: 50
Processing QueryID: 200
Processing QueryID: 133
Processing QueryID: 216
Processing QueryID: 23
Processing QueryID: 250
Processing QueryID: 83
Processing QueryID: 112
Processing QueryID: 232

Avg. Boolean NDCG Scores:       0.1
Avg. Vector  NDCG Scores:       0.84958
Wilcoxon Test P-Value:          0.00691042980781

There is significant difference between Boolean and Vector Retrieval Model!
Done
```

Average NDCG evaluation for ALL queries from query.text

```
python batch_eval.py index_file query.text qrels.text batch
```

```
Processing QueryID: 356
Processing QueryID: 360
Processing QueryID: 365

Avg. Boolean NDCG Scores:       0.06853
Avg. Vector  NDCG Scores:       0.62212
Wilcoxon Test P-Value:          3.17786546865e-30

There is significant difference between Boolean and Vector Retrieval Model!
Done
```

List of Test Cases

Tests done on index.py

```
Indexing document 2

List of tokens (lowercased):
['simple', 'shear', 'flow', 'past', 'a', 'flat', 'plate', 'in', 'an', 'incompressible', 'fluid', 'of', 'small', 'viscosity', 'in', 'the', 'stu
, 'a', 'curved', 'shock', 'wave', 'emitting', 'from', 'the', 'nose', 'or', 'leading', 'edge', 'of', 'the', 'body', 'consequently', 'there', 'e
'a', 'situation', 'arises', 'for', 'instance', 'in', 'the', 'study', 'of', 'the', 'hypersonic', 'viscous', 'flow', 'past', 'a', 'flat', 'plate
riginal', 'problem', 'the', 'inviscid', 'free', 'stream', 'outside', 'the', 'boundary', 'layer', 'is', 'irrotational', 'while', 'in', 'a', 'hy
sible', 'effects', 'of', 'vorticity', 'have', 'been', 'recently', 'discussed', 'by', 'ferri', 'and', 'libby', 'in', 'the', 'present', 'paper',
it', 'can', 'be', 'shown', 'that', 'this', 'problem', 'can', 'again', 'be', 'treated', 'by', 'the', 'boundary-layer', 'approximation', 'the',
'is', 'restricted', 'to', 'two-dimensional', 'incompressible', 'steady', 'flow']

After stopwords removal:
['simple', 'shear', 'flow', 'past', 'flat', 'plate', 'incompressible', 'fluid', 'small', 'viscosity', 'study', 'high-speed', 'viscous', 'flow'
', 'body', 'consequently', 'exists', 'inviscid', 'rotational', 'flow', 'region', 'shock', 'wave', 'boundary', 'layer', 'situation', 'arises',
cal', 'boundary-layer', 'problem', 'prandtls', 'original', 'problem', 'inviscid', 'free', 'stream', 'outside', 'boundary', 'layer', 'irrotatio
, 'vorticity', 'recently', 'discussed', 'ferri', 'libby', 'present', 'paper', 'simple', 'shear', 'flow', 'past', 'flat', 'plate', 'fluid', 'sm
', 'constant', 'vorticity', 'discussion', 'restricted', 'two-dimensional', 'incompressible', 'steady', 'flow']

After stemming:
['simpl', 'shear', 'flow', 'past', 'flat', 'plate', 'incompress', 'fluid', 'small', 'viscos', 'studi', 'high-spe', 'viscou', 'flow', 'past', '
t', 'inviscid', 'rotat', 'flow', 'region', 'shock', 'wave', 'boundari', 'layer', 'situat', 'aris', 'instanc', 'studi', 'hyperson', 'viscou', '
roblem', 'inviscid', 'free', 'stream', 'outsid', 'boundari', 'layer', 'irrot', 'hyperson', 'boundary-lay', 'problem', 'inviscid', 'free', 'str
ar', 'flow', 'past', 'flat', 'plate', 'fluid', 'small', 'viscos', 'investig', 'shown', 'problem', 'treat', 'boundary-lay', 'approxim', 'novel'

# of documents indexed: 2
# of terms indexed: 114

== Statistics for the term 'lift' BEFORE saving the index to disk (invertedIndex_1) ==
Posting list:   [1]
Positions:      [16, 49, 59, 63]
TF:             4
IDF:            0.30103

Saving to disk...
InvertedIndex successfully saved to index_file

Loading from disk...
InvertedIndex successfully loaded to memory from index_file

== Statistics for the term 'lift' AFTER loading the index from disk (invertedIndex_2) ==
Posting list:   [1]
Positions:      [16, 49, 59, 63]
TF:             4
IDF:            0.30103

Pass
```

1. **Are the stopwords really removed? Checked!**
   Comparing the initial "list of tokens (lowercased)", the list "After stopwords removal" contains no stopwords like "a", "in", "an", "the".

2. **Are the terms in index all stemmed? Checked!**
   Comparing the list "After stopwords removal" and the list "After stemming", words like
   - "simple" is stemmed to "simpl"
   - "incompressible" is stemmed to "incompress"
   - "viscosity" is stemmed to "viscos"

3. **What is the number of terms in the dictionary and what is the size of postings? Do they make sense? Checked!**
   The test builds an index with 2 documents. The # of documents and terms indexed are reasonable.

4. **Are index saving and loading working as expected? Checked!**
   a) A statistics (posting list, positions, TF, IDF) of a random term **"lift"** are displayed from the first instance of Class InvertedIndex (invertedIndex_1).
   b) A second instance of the Class InvertedIndex was created (invertedIndex_2).
   c) The invertedIndex_1 is then saved and loaded into invertedIndex_2.
   d) The statistics for the term **"lift"** from invertedIndex_2 matches the values from invertedIndex_1, which indicates the index is saving and loading as expected since the values remain unaltered.

Tests done on query.py

```
user@ubuntu:~/Desktop/simple-search-engine$ python query.py index_file 1 query.text 1
Loading from disk...
InvertedIndex successfully loaded to memory from index_file

Original query:
what similarity laws must be obeyed when constructing aeroelastic models
of heated high speed aircraft .

Preprocessed query:
['similar', 'law', 'must', 'obey', 'construct', 'aeroelast', 'model', 'heat', 'high', 'speed', 'aircraft']

== Query Terms ==
-> similar
TF: 1           IDF: 0.954      TF-IDF: 0.954
-> law
TF: 1           IDF: 1.462      TF-IDF: 1.462
-> must
TF: 1           IDF: 1.462      TF-IDF: 1.462
-> obey
TF: 1           IDF: 2.447      TF-IDF: 2.447
-> construct
TF: 1           IDF: 1.544      TF-IDF: 1.544
-> aeroelast
TF: 1           IDF: 1.914      TF-IDF: 1.914
-> model
TF: 1           IDF: 0.903      TF-IDF: 0.903
-> heat
TF: 1           IDF: 0.699      TF-IDF: 0.699
-> high
TF: 1           IDF: 0.845      TF-IDF: 0.845
-> speed
TF: 1           IDF: 0.699      TF-IDF: 0.699
-> aircraft
TF: 1           IDF: 1.301      TF-IDF: 1.301

== Testing Cosine Similary Calculations using Mock Query & Doc TF-IDF Vectors ==
Mock Query Vec: [0.702753576, 0.702753576]
Mock Docum Vec: [0.140550715, 0.140550715]
Cosine Similary: 1.0

Pass
```

1. **Do you also convert queries to terms? Checked!**
   The preprocessed query transformed the original query with the same processing steps performed during index building. The output is a **list of stemmed terms** with **no** stopwords.

2. **How do you confirm that Query TF-IDF values are computed correctly? Checked!**
   Every query term is assigned a Term Frequency (local TF) based on the number of occurrence that it appears in the query. The Inverse Document Frequency (Global IDF) of each query term is retrieved from the InvertedIndex using $\log(\frac{N\ total\ documents}{documents\ with\ term\ i})$

   The TF-IDF is then computed as $W_{q,d} = TF_q * IDF_{q,d}$

3. **How do you confirm cosine similarity is computed correctly? Checked!**
   To validate the cosine similarity calculation, a mock/simulated query and document vector are used to compute the scores. The output matched the manual calculations.

$$cosine\_similary(query, document_i) = \frac{dot\_product(query, document_i)}{||query||_2 * ||document||_2}$$

Tests done on batch_eval.py

1. **Are your selected sample queries getting the same results as you expect (manually computed) for Boolean model processing? Checked!**
2. **Are your selected sample queries getting the same results as you expect for vector model processing? Checked!**
3. **Do the NDCGs for selected sample queries match your manually computed results? Checked!**