Group Assignment 2
Comparing Exact and Inexact Retrieval

Team 2: Thanh Le, Antoine Si

Imported Libraries

```
import numpy as np
import random
from queue import PriorityQueue
```

In addition to the 4 imported libraries (os, re, collections, and time). We imported numpy, for square root and log10 functions, random, for picking leaders in cluster pruning, and priority queue, for building the champion list, inexact query index elimination, and to get the top k documents.

IMPLEMENTATION - Log frequency weight of a term

```python
# function to calculate the log frequency weight of term in document with doc_id
def cal_w_td(self, term, doc_id):
    tf_td = len(self.index[term][doc_id])
    return 0 if tf_td == 0 else 1 + np.log10(tf_td)

# function to get the log frequency weight of term in document with doc_id
def get_w_td(self, term, doc_id):
    w_td = 0
    try:
        w_td = self.index[term][doc_id][0]
    except:
        w_td = 0

    return w_td
```

- 1st function calculates the $w_{t,d}$ of a term in a document with corresponding document ID.
- 2nd function gets the $w_{t,d}$ of a term in a document with corresponding document ID.

IMPLEMENTATION - Inverse Document Frequency (IDF)

```python
# function to calculate the inverse document frequency of term
def cal_idf_t(self, term):
    df_t = len(self.get_docs(term))
    N = len(self.doc_list)
    return 0 if df_t == 0 else np.log10(N / df_t)

# function to get the inverse document frequency of term
def get_idf_t(self, term):
    return self.index[term][0][0]
```

- The 1st function calculates the IDF for a term.
- The 2nd function gets the IDF for a term from the index.

IMPLEMENTATION - Document Miscellaneous

```
# function to get list of doc ids that the input term appears
def get_docs(self, term):
    return [doc_id for doc_id in self.index[term]][1:]

# function to get the document length
def get_doc_length(self, doc_id):
    doc = self.doc_list[doc_id]
    return len(self.tokenize(doc))
```

- 1st function gets the document ID that a term appears in?
- 2nd function gets document length.

IMPLEMENTATION - Getting top K documents

```python
# function to get the top k retrievals
def get_top_docs(self, scores, k):
    result = []

    # iterate through each document
    queue = PriorityQueue()
    for doc_id in scores:
        # normalize the score by document length
        scores[doc_id] /= self.get_doc_length(doc_id)

        # add the score to a priority queue in negative number so that it can be retrieved descendingly
        queue.put((scores[doc_id] * -1, doc_id))

    # get top k value from the queue
    while not queue.empty() and len(result) < k:
        result.append(queue.get())

    return result
```

IMPLEMENTATION - Cosine Scores

```python
# function to calculate the cosine scores from the query term to each of documents in the input list
def cal_cosine_scores(self, scores, docs, query_term):

    # get idf weight of the term
    wtq = self.get_idf_t(query_term)

    # iterate through each document
    for doc_id in docs:
        # get the wtd weight of the term in the
        wtd = self.get_w_td(query_term, doc_id)

        # calculate the if-idf weight
        tf_idf = wtd * wtq

        # accumulate the score
        scores[doc_id] = scores.get(doc_id, 0) + tf_idf
```

docs is varied depends on retrieval type:

- Exact retrieval: docs = posting list of the term
- Inexact retrieval:
  - Champion list: docs = champion list of the term
  - Index elimination: docs = posting list of the term with high IDF values
  - Cluster pruning: docs = candidate list (leader + followers) of the nearest leader

IMPLEMENTATION - print_retrieved_docs and process_query

- 1st function prints the retrieved documents and its corresponding cosine scores.
- 2nd function processes query terms, which will be used in other methods.

```
# function to print the top retrieved documents with their corresponding cosine scores
def print_retrieved_docs(self, result, time):
    print("Total Docs retrieved:", len(result))
    for doc_id in result:
        if type(doc_id) != int:
            print("Doc: %-*s Score: %s" % (20, self.doc_list[doc_id[1]], -doc_id[0]))
        else:
            print("Doc: %-*s" % (20, self.doc_list[doc_id]))

    print("Retrieved in", time, "seconds.\n\n")

# convert the free text query to a list of tokenized terms and bypass stop words
def process_query(self, query_terms):
    # convert the free text query to a list of tokenized terms
    terms = re.split(r"\W+", query_terms.lower())

    # bypass terms are in the stop list
    return list(set(terms) - self.stop_list)
```

IMPLEMENTATION - New additions to BUILDING INDEX

- Our build_index reuses most of the code from Assignment 1 with a few additions:
- It now calculates the $W_{t,d}$ and idf_t for each term and records them onto the index.
- It also builds the champion list and the cluster pruning as well.
- DocID and pos now start at 1, as index 0 is for $W_{t,d}$ and idf_t.

```
''' CALCULATE W_TD and IDF_T WEIGHT AND RECORD THEM TO THE INDEX '''
# iterate through all the terms
for term in self.index:

    # iterate through all the documents
    for doc_id in self.index[term]:
        # calculate the weighted term frequency in the document
        w_td = self.cal_w_td(term, doc_id)

        # record the weighted term frequency to the index
        self.index[term][doc_id].insert(0, w_td)

    # calculate idf weight of the term
    idf_t = self.cal_idf_t(term)

    # record the idf weight at index 0 in the corresponding postings list
    self.index[term] = {**{0: [idf_t]}, **self.index[term]}

''' BUILD THE CHAMPION LIST '''
self.build_champion_list()

''' BUILD THE CLUSTER PRUNING '''
self.build_cluster_pruning()
```

IMPLEMENTATION- Building the Champion List

```python
# function to build the champion list
def build_champion_list(self):
    self.champion_list = collections.defaultdict(list)
    queue = PriorityQueue()

    # iterate through all the terms
    for term in self.index:

        # iterate through all the documents
        for doc_id in self.get_docs(term):
            # get the weighted term frequency in the document
            w_td = self.get_w_td(term, doc_id)

            # add the weight to a priotity queue together with the corresponding doc_id
            queue.put((w_td * -1, doc_id))

        # get the idf weight of the term
        idf_t = self.get_idf_t(term)

        # determine number of documents r in champion list (rarer terms should have bigger champions list)
        r = 5 + idf_t * len(self.doc_list) / 4

        # generate the champion list for each term
        while not queue.empty() and len(self.champion_list[term]) < r:
            self.champion_list[term].append(queue.get()[1])
```

IMPLEMENTATION - Build cluster pruning

```
# function to build the cluster pruning
def build_cluster_pruning(self):
    self.cluster_pruning = collections.defaultdict(list)

    # pick N docs at random to be leaders
    self.leader_list = []
    for _ in range(int(np.sqrt(len(self.doc_list)))):
        self.leader_list.append(random.randint(1, len(self.doc_list)))

    # consider other docs as followers
    follower_list = list(self.doc_list.keys() - self.leader_list)

    # iterate through the followers
    for doc_id in follower_list:

        # get the terms from each follower
        doc_terms = set(self.tokenize(self.doc_list[doc_id]))

        # bypass terms are in the stop list
        doc_terms -= self.stop_list

        if len(doc_terms) > 0:

            scores = {}
```

```
# iterate through each term in the follower
for doc_term in doc_terms:
    # calculate the cosine score from the term to its leaders
    self.cal_cosine_scores(scores, self.leader_list, doc_term)

# get top 1 document with highest score
result = self.get_top_docs(scores, 1)

# set this top document as the nearest leader
nearest_leader = result[0][1]

# record the nearest leader for each follower
self.cluster_pruning[nearest_leader].append(doc_id)
```

Method 1- Exact Top K Retrieval (Method 1)

Let’s say we get a free text, multiple word query (this will be the baseline query for the 4 methods). This method will generate the vectors for the documents and the query and compute the cosine similarity score for the documents. Afterwards the program will retrieve the top K documents for each query in decreasing order of their score.

IMPLEMENTATION - Method 1

```python
# function to exact top K retrieval (method 1)
# return at the minimum the document names of the top K documents ordered in decreasing order of similarity score
def exact_query(self, query_terms, k):
    scores = {}

    # record the start time
    start_time = time.time()

    # convert the free text query to a list of tokenized terms and bypass stop words
    query_terms = self.process_query(query_terms)

    # iterate through each term in the query
    for query_term in query_terms:
        # calculate the cosine score from the query term to its postings list
        self.cal_cosine_scores(scores, self.get_docs(query_term), query_term)

    # get top k document with highest score
    result = self.get_top_docs(scores, k)

    # record the end time
    end_time = time.time()

    # print the result
    print("Results for the Exact Top {k} Retrievals of:".format(k=k), ", ".join(query_terms))
    self.print_retrieved_docs(result, end_time - start_time)
```

Method 2 - Champion List

For each term of a query, a list of r documents that is based on the weighted term frequency $w_{t,d}$. However, there is a problem with value of r:

- We need to determine the value of r when constructing the index while the value of k is unknown at that time
- It can happen that the value of r is too small that causes the number of documents being retrieved < k
- So, we can set higher r for term with high IDF values (rarer terms)
- Formula for r scaling with IDF value:

```
r = 5 + idf_t * len(self.doc_list) / 4
```

IMPLEMENTATION - Method 2

```python
# function to exact top K retrieval using champion list (method 2)
# return at the minimum the document names of the top K documents ordered in decreasing order of similarity score
def inexact_query_champion(self, query_terms, k):
    scores = {}

    # record the start time
    start_time = time.time()

    # convert the free text query to a list of tokenized terms and bypass stop words
    query_terms = self.process_query(query_terms)

    # iterate through each term in the query
    for query_term in query_terms:
        # calculate the cosine score from the query term to its champion list
        self.cal_cosine_scores(scores, self.champion_list[query_term], query_term)

    # get top k document with highest score
    result = self.get_top_docs(scores, k)

    # record the end time
    end_time = time.time()

    # print the result
    print("Results for the Inxact Top {k} Retrievals using Champion List of:".format(k=k), ", ".join(query_terms))
    self.print_retrieved_docs(result, end_time - start_time)
```

Method 3 - Index Elimination

For each query, we try and consider documents with the most amount of query terms. For the query terms themselves, it is preferred to choose query terms that have high idf values. This discards many documents when the terms have low idf, and decreases the pool of documents to contain mostly high idf value documents.

IMPLEMENTATION - Method 3

```python
# function to exact top K retrieval using index elimination (method 3)
# return at the minimum the document names of the top K documents ordered in decreasing order of similarity score
def inexact_query_index_elimination(self, query_terms, k):
    high_idf_terms = []
    queue = PriorityQueue()

    # record the start time
    start_time = time.time()

    # convert the free text query to a list of tokenized terms and bypass stop words
    query_terms = self.process_query(query_terms)

    # iterate through all terms in the query
    for query_term in query_terms:
        # calculate the idf value of the term
        idf_t = self.get_idf_t(query_term)

        # add the negative idf values to a priority queue to later sort in decreasing order
        queue.put((idf_t * -1, query_term))

    # pick half of the query terms with highest idf values
    while not queue.empty() and len(high_idf_terms) < len(query_terms) / 2:
        high_idf_terms.append(queue.get()[1])

    # retrieve top k documents using these query values
    scores = {}

    # iterate through each term in the set of highest idf values
    for query_term in high_idf_terms:
        # calculate the cosine score from the query term to its posting list
        self.cal_cosine_scores(scores, self.get_docs(query_term), query_term)

    # get top k document with highest score
    result = self.get_top_docs(scores, k)

    # record the end time
    end_time = time.time()

    # print the result
    print("Results for the Inxact Top {k} Retrievals using Index Elimination of:".format(k=k),
          ", ".join(query_terms))
    self.print_retrieved_docs(result, end_time - start_time)
```

Method 4 - Simple cluster pruning

We pick sqrt(N) leaders (N is the number of documents in the collection) and use them to implement cluster pruning. With each query, it will look at the closest leader and pick the top K documents. If that leader’s cluster of documents is less than K, it will find the next closest leader with K documents and etc.

IMPLEMENTATION - Method 4

```
# function to exact top K retrieval using cluster pruning (method 4)
# return at the minimum the document names of the top K documents ordered in decreasing order of similarity score
def inexact_query_cluster_pruning(self, query_terms, k):
    leader_scores = {}
    scores = {}

    # record the start time
    start_time = time.time()

    # convert the free text query to a list of tokenized terms and bypass stop words
    query_terms = self.process_query(query_terms)

    # iterate through each term in the query
    for query_term in query_terms:
        # calculate the cosine score from the query term to the leaders
        self.cal_cosine_scores(leader_scores, self.leader_list, query_term)

    # get the leaders ordered in decreasing order of similarity score
    leader_result = self.get_top_docs(leader_scores, len(self.leader_list))

    i = 0
    result = []

    # look for the next best leader if the current one does not get enough top k results
    while len(result) < k and i < len(leader_result):
        # get the best leader at the moment
        leader = leader_result[i][1]

        # iterate through each term in the query
        for query_term in query_terms:
            # construct a candidate set which consists of leader together with its followers
            candidates = self.cluster_pruning[leader] + [leader]

            # calculate the cosine score from the query term to the candidates
            self.cal_cosine_scores(scores, candidates, query_term)

        # get top k documents with highest score
        result += self.get_top_docs(scores, k - len(result))
        i += 1

    # record the end time
    end_time = time.time()

    print("Results for the Inxact Top {k} Retrievals using Cluster Pruning of:".format(k=k), ", ".join(query_terms))
    self.print_retrieved_docs(result, end_time - start_time)
```

EXPERIMENTAL RESULTS

Build the index

```
>>> a = index(os.path.join(os.getcwd(), 'collection'))
Index built in 1.893979787826538 seconds.
```

EXPERIMENTAL RESULTS: Process the queries

We use the cosine similarity and processing time to evaluate the performance of our retrieval methods.

- Accuracy: For the cosine similarity, we consider the average cosine score of k documents. We want to look for method that retrieves documents with higher score.
- Time efficiency: For the processing time, we want to look for method that retrieves the documents in less time

```
>>> a.exact_query('government party political', 5)
Results for the Exact Top 5 Retrievals of: political, government, party
Total Docs retrieved: 5
Doc: Text-172.txt         Score: 0.005854267148926767
Doc: Text-325.txt         Score: 0.005799098358936951
Doc: Text-109.txt         Score: 0.005705625079588391
Doc: Text-200.txt         Score: 0.005652285742255003
Doc: Text-190.txt         Score: 0.005032381340092987
Retrieved in 0.051434993743896484 seconds.


>>> a.inexact_query_champion('government party political', 5)
Results for the Inxact Top 5 Retrievals using Champion List of: political, government, party
Total Docs retrieved: 5
Doc: Text-118.txt         Score: 0.00402005201456023
Doc: Text-391.txt         Score: 0.0037376996703808616
Doc: Text-276.txt         Score: 0.0036153232785312074
Doc: Text-322.txt         Score: 0.0035220309646672187
Doc: Text-94.txt          Score: 0.003177804939260879
Retrieved in 0.02858710289001465 seconds.


>>> a.inexact_query_index_elimination('government party political', 5)
Results for the Inxact Top 5 Retrievals using Index Elimination of: political, government, party
Total Docs retrieved: 5
Doc: Text-172.txt         Score: 0.005854267148926767
Doc: Text-190.txt         Score: 0.005032381340092987
Doc: Text-325.txt         Score: 0.0047020058723354985
Doc: Text-200.txt         Score: 0.004582967748985233
Doc: Text-158.txt         Score: 0.004227086788069489
Retrieved in 0.04429483413696289 seconds.


>>> a.inexact_query_cluster_pruning('government party political', 5)
Results for the Inxact Top 5 Retrievals using Cluster Pruning of: political, government, party
Total Docs retrieved: 5
Doc: Text-172.txt         Score: 0.005854267148926767
Doc: Text-109.txt         Score: 0.005705625079588391
Doc: Text-190.txt         Score: 0.005032381340092987
Doc: Text-118.txt         Score: 0.00402005201456023
Doc: Text-94.txt          Score: 0.0038922026942444106
Retrieved in 0.022455692291259766 seconds.
```

EXPERIMENTAL RESULTS: Process the queries

	Exact	Champion List	Index Elimination	Cluster Pruning
Average Cosine similarity	0.005604	0.00361	0.004876	0.004898
Processing time	0.0514	0.0286	0.0443	0.0225

```
>>> a.exact_query('government party political', 5)
Results for the Exact Top 5 Retrievals of: political, government, party
Total Docs retrieved: 5
Doc: Text-172.txt         Score: 0.005854267148926767
Doc: Text-325.txt         Score: 0.005799098358936951
Doc: Text-109.txt         Score: 0.005705625079588391
Doc: Text-200.txt         Score: 0.005652285742255003
Doc: Text-190.txt         Score: 0.005032381340092987
Retrieved in 0.051434993743896484 seconds.


>>> a.inexact_query_champion('government party political', 5)
Results for the Inxact Top 5 Retrievals using Champion List of: political, government, party
Total Docs retrieved: 5
Doc: Text-118.txt         Score: 0.00402005201456023
Doc: Text-391.txt         Score: 0.0037376996703808616
Doc: Text-276.txt         Score: 0.0036153232785312074
Doc: Text-322.txt         Score: 0.0035220309646672187
Doc: Text-94.txt          Score: 0.003177804939260879
Retrieved in 0.02858710289001465 seconds.
```

```
>>> a.inexact_query_index_elimination('government party political', 5)
Results for the Inxact Top 5 Retrievals using Index Elimination of: political, government, party
Total Docs retrieved: 5
Doc: Text-172.txt         Score: 0.005854267148926767
Doc: Text-190.txt         Score: 0.005032381340092987
Doc: Text-325.txt         Score: 0.0047020058723354985
Doc: Text-200.txt         Score: 0.004582967748985233
Doc: Text-158.txt         Score: 0.004227086788069489
Retrieved in 0.04429483413696289 seconds.


>>> a.inexact_query_cluster_pruning('government party political', 5)
Results for the Inxact Top 5 Retrievals using Cluster Pruning of: political, government, party
Total Docs retrieved: 5
Doc: Text-172.txt         Score: 0.005854267148926767
Doc: Text-109.txt         Score: 0.005705625079588391
Doc: Text-190.txt         Score: 0.005032381340092987
Doc: Text-118.txt         Score: 0.00402005201456023
Doc: Text-94.txt          Score: 0.0038922026942444106
Retrieved in 0.022455692291259766 seconds.
```

EXPERIMENTAL RESULTS: Process the queries

	Exact	Champion List	Index Elimination	Cluster Pruning
Average Cosine similarity	0.005909	0.005909	0.005315	0.002109
Processing time	0.0465	0.0290	0.0172	0.0214

```
>>> a.exact_query('plane fire fight soldier', 10)
Results for the Exact Top 10 Retrievals of: fire, fight, plane, soldier
Total Docs retrieved: 10
Doc: Text-51.txt          Score: 0.008622447179979434
Doc: Text-299.txt         Score: 0.008197212575004288
Doc: Text-154.txt         Score: 0.006442097160861706
Doc: Text-419.txt         Score: 0.006269812849236168
Doc: Text-19.txt          Score: 0.005597086148674347
Doc: Text-123.txt         Score: 0.005015850279388934
Doc: Text-82.txt          Score: 0.004794562767062952
Doc: Text-330.txt         Score: 0.004790302745560466
Doc: Text-10.txt          Score: 0.004745023825430329
Doc: Text-407.txt         Score: 0.004669976359340898
Retrieved in 0.0464780330657959 seconds.


>>> a.inexact_query_champion('plane fire fight soldier', 10)
Results for the Inxact Top 10 Retrievals using Champion List of: fire, fight, plane, soldier
Total Docs retrieved: 10
Doc: Text-51.txt          Score: 0.008622447179979434
Doc: Text-299.txt         Score: 0.008197212575004288
Doc: Text-154.txt         Score: 0.006442097160861706
Doc: Text-419.txt         Score: 0.006269812849236168
Doc: Text-19.txt          Score: 0.005597086148674347
Doc: Text-123.txt         Score: 0.005015850279388934
Doc: Text-82.txt          Score: 0.004794562767062952
Doc: Text-330.txt         Score: 0.004790302745560466
Doc: Text-10.txt          Score: 0.004745023825430329
Doc: Text-407.txt         Score: 0.004669976359340898
Retrieved in 0.028971195220947266 seconds.
```

```
>>> a.inexact_query_index_elimination('plane fire fight soldier', 10)
Results for the Inxact Top 10 Retrievals using Index Elimination of: fire, fight, plane, soldier
Total Docs retrieved: 10
Doc: Text-299.txt         Score: 0.008197212575004288
Doc: Text-419.txt         Score: 0.006269812849236168
Doc: Text-19.txt          Score: 0.005597086148674347
Doc: Text-123.txt         Score: 0.005015850279388934
Doc: Text-51.txt          Score: 0.004958635257190582
Doc: Text-82.txt          Score: 0.004794562767062952
Doc: Text-330.txt         Score: 0.004790302745560466
Doc: Text-10.txt          Score: 0.004745023825430329
Doc: Text-67.txt          Score: 0.004405814434598389
Doc: Text-224.txt         Score: 0.0043326281483093785
Retrieved in 0.01724100112915039 seconds.


>>> a.inexact_query_cluster_pruning('plane fire fight soldier', 10)
Results for the Inxact Top 10 Retrievals using Cluster Pruning of: fire, fight, plane, soldier
Total Docs retrieved: 10
Doc: Text-224.txt         Score: 0.0043326281483093785
Doc: Text-138.txt         Score: 0.003992435930166534
Doc: Text-234.txt         Score: 0.0026205140993335752
Doc: Text-55.txt          Score: 0.002560441611640653
Doc: Text-86.txt          Score: 0.0021112049630442215
Doc: Text-377.txt         Score: 0.0016081814051272056
Doc: Text-99.txt          Score: 0.001318099077664617
Doc: Text-70.txt          Score: 0.0011623432276157638
Doc: Text-159.txt         Score: 0.0008997035814131356
Doc: Text-121.txt         Score: 0.0005265346171410449
Retrieved in 0.021398067474365234 seconds.
```

EXPERIMENTAL RESULTS

Cluster pruning

```
>>> a.exact_query('plane fire fight', 10)
Results for the Exact Top 10 Retrievals of: fire, fight, plane
Total Docs retrieved: 10
Doc: Text-51.txt          Score: 0.008622447179979434
Doc: Text-299.txt         Score: 0.008197212575004288
Doc: Text-154.txt         Score: 0.006442097160861706
Doc: Text-419.txt         Score: 0.006269812849236168
Doc: Text-19.txt          Score: 0.005597086148674347
Doc: Text-123.txt         Score: 0.005015850279388934
Doc: Text-82.txt          Score: 0.004794562767062952
Doc: Text-288.txt         Score: 0.004409169954134126
Doc: Text-67.txt          Score: 0.004405814434598389
Doc: Text-63.txt          Score: 0.0043404618724931
Retrieved in 0.04361915588378906 seconds.
```

```
>>> a.inexact_query_cluster_pruning('plane fire fight', 10)
Results for the Inxact Top 10 Retrievals using Cluster Pruning of: fire, fight, plane
Total Docs retrieved: 10
Doc: Text-419.txt         Score: 0.006269812849236168
Doc: Text-288.txt         Score: 0.004409169954134126
Doc: Text-67.txt          Score: 0.004405814434598389
Doc: Text-106.txt         Score: 0.0033809913533104146
Doc: Text-407.txt         Score: 0.00322934299461195
Doc: Text-268.txt         Score: 0.0023788420915278195
Doc: Text-323.txt         Score: 0.0018674080149098222
Doc: Text-395.txt         Score: 0.0016051853805827924
Doc: Text-349.txt         Score: 0.00153136670590232
Doc: Text-324.txt         Score: 0.0008657525028692437
Retrieved in 0.012445926666259766 seconds.
```

```
>>> a.inexact_query_cluster_pruning('plane fire fight', 10)
Results for the Inxact Top 10 Retrievals using Cluster Pruning of: fire, fight, plane
Total Docs retrieved: 10
Doc: Text-51.txt          Score: 0.008622447179979434
Doc: Text-299.txt         Score: 0.008197212575004288
Doc: Text-218.txt         Score: 0.0037448869308815452
Doc: Text-3.txt           Score: 0.0
Doc: Text-309.txt         Score: 0.0
Doc: Text-127.txt         Score: 0.0
Doc: Text-13.txt          Score: 0.0
Doc: Text-239.txt         Score: 0.0035820912107563875
Doc: Text-169.txt         Score: 0.0029693103900119513
```

Comparing to the result from Exact Retrieval (top picture), the documents retrieved from the Cluster Pruning (left pictures) varies since the leaders are picked randomly.

It can happen that the closest leader does not get enough k document so that the next leader should be considered.

In this case, the candidates from the next leader can possibly have higher cosine score than those from the closer leader.

COMPARISON

- Exact Retrieval: We notice that we can always get the highest cosine similarity since it considers all the posting list of the terms in the query. Therefore, this method produces the most accurate result for the top k relevant documents (this method is used as the benchmark to evaluate the accuracy of other methods). However, at the same time, it needs to process much more information, so that its processing time is the worst.

- Index Elimination: We can pretty much get acceptable results which is 70% same as the Exact Retrieval since this method only considers the posting list of rarer terms. Therefore, if the query terms are those with high IDF values, the top k documents retrieved would be very similar to the ones from the Exact Retrieval. The processing time of this method is much lower than the Exact Retrieval since about half of the posting lists are considered.

COMPARISON

- Champion List: This method can produce acceptable result. Since the champion list is built on the weight of the term, it is important to have rare terms in the query so that it can accumulate higher score for the relevant document in the champion list. The value of r is scaled with the rareness of the term so that it can help solving the problem that the number of documents being retrieved is less than k. The processing is lower than the Exact Retrieval when only the champion list of the query terms are considered.

- Cluster Pruning: The results from this method varied over time since the leaders are picked randomly. The cluster is built every time the index is constructed, so that its performance in terms of accuracy fluctuates. It can retrieve good top k documents when the leaders are well chosen. The processing time is low since only the nearest leader and its followers are considered.

TASK DIVISION

- Both of us first started working on implementing the index.py file by ourselves and discussed with each other when any problem happened
- We compared our query results with each other to make sure we got the same outputs
- We combined our works into a final version with detailed comments in each step which helps explaining the codes
- Thanh created the README file
- Antoine created the output.txt file
- We worked on the presentation slides together