Group Assignment 3 Relevance Feedback

Team 2: Thanh Le, Antoine Si

LIBRARIES

```
import os
import collections
import time
import numpy as np
from queue import PriorityQueue

import lucene

from java.io import File
from org.apache.lucene.document import Document, Field, FieldType
from org.apache.lucene.util import BytesRefIterator

from org.apache.lucene.analysis.tokenattributes import CharTermAttribute
from org.apache.lucene.analysis.standard import StandardAnalyzer
from org.apache.lucene.analysis.en import EnglishAnalyzer
from org.apache.lucene.index import IndexWriter, IndexWriterConfig, PostingsEnum, IndexOptions, TermsEnum
from org.apache.lucene.store import FSDirectory

from org.apache.lucene.search import IndexSearcher, MatchAllDocsQuery, DocIdSetIterator
from org.apache.lucene.index import DirectoryReader
```

BUILD THE INDEX

```python
# function to build an index using PyLucene
def build_lucene_index(self):
    directory_path = self.path + '/dir'
    collection_path = self.path + '/TIME.ALL'

    # construct the directory to store the index on local file system
    self.directory = FSDirectory.open(File(directory_path).toPath())

    # construct the analyzer which is used to perform analysis on terms appeared in documents to be added in the index
    self.analyzer = StandardAnalyzer()

    # construct and configure an index write
    # always overite existing index to avoid duplicate files
    config = IndexWriterConfig(self.analyzer)
    config.setOpenMode(IndexWriterConfig.OpenMode.CREATE)
    self.writer = IndexWriter(self.directory, config)

    # construct reader and searcher
    self.reader = DirectoryReader.open(self.directory)
    self.searcher = IndexSearcher(self.reader)

    # iterate through each line in the collection file to construct doc and add to the IndexWriter
    with open(collection_path, 'r') as file:
        content = ""
        title = ""
        for line in file.readlines():
            if line.strip():
                if line.startswith("*TEXT"):
                    if title != "":
                        self.add_doc(title, content)
                    content = ""
                    title = line[1:9] + ".txt"
                elif line.startswith("*STOP"):
                    self.add_doc(title, content)
                else:
                    content += line.strip()

    # close the writer
    self.writer.close()
```

```python
# function to add a new document with specific title and content to the IndexWriter
def add_doc(self, title, content):
    # create a new document
    doc = Document()

    # configure how metadata is stored in the index
    metaType = FieldType()
    metaType.setStored(True)
    metaType.setTokenized(True)

    # configure how content data is stored in the index
    # store the doc id, term frequency, and postitions
    contentType = FieldType()
    contentType.setIndexOptions(IndexOptions.DOCS_AND_FREQS_AND_POSITIONS)
    contentType.setStoreTermVectors(True)
    contentType.setStoreTermVectorPositions(True)
    contentType.setStored(True)
    contentType.setTokenized(True)

    # add the title and content field to the document
    doc.add(Field("title", title, metaType))
    doc.add(Field("content", content, contentType))

    # add the document to the IndexWriter
    self.writer.addDocument(doc)
```

Use PyLucene to index the collection from TIME.ALL file and store it in a file system locally

BUILD THE INDEX

```
# function to read documents from collection, tokenize and build the index with tokens
# implement additional functionality to support relevance feedback
# use unique document integer IDs
def buildIndex(self):
    self.index = collections.defaultdict(lambda: collections.defaultdict(list))

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

    # build index using PyLucene
    self.build_lucene_index()

    # construct list of stop words
    with open(self.path + '/TIME.STP', 'r') as lines:
        # read the stop words from the file
        self.stop_list = {word.strip() for word in lines.readlines()}

        # tokenize the stop words
        stream = self.analyzer.tokenStream("", " ".join(self.stop_list))
        stream.reset()

        self.stop_list = []
        while stream.incrementToken():
            stop_word = stream.getAttribute(CharTermAttribute.class_).toString()
            self.stop_list.append(stop_word)

        stream.close()
```

Get the list of stop words from TIME.STP file

BUILD THE INDEX

- Iterate over the term vector and eliminate stop words
- Record the position to the index

```
# iterate through each document in the lucene index
self.doc_list = self.get_doc_list()
for doc in self.doc_list:

    # get the doc_id of the current document
    doc_id = doc.doc

    # get the term vector of the current document
    term_vector = self.reader.getTermVector(doc_id, "content")

    # check if there is term in the document
    if term_vector is not None:
        term_iter = BytesRefIterator.cast_(term_vector.iterator())

        # iterate through each term in the document
        while term_iter.next():
            terms_enum = TermsEnum.cast_(term_iter)

            # convert the term from UTF-8 byte format to string
            term = terms_enum.term().utf8ToString()

            # bypass if term is in the stop list
            if term not in self.stop_list:

                # get the postings list of the term
                postings = terms_enum.postings(None, PostingsEnum.ALL)

                # iterate through each posting
                if postings.nextDoc() is not DocIdSetIterator.NO_MORE_DOCS:

                    # get the term frequency
                    freq = postings.freq()

                    # iterate through each position and record it to the index
                    for _ in range(freq):
                        pos = postings.nextPosition()
                        self.index[term][doc_id].append(pos)
```

BUILD THE INDEX

```
''' 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 of term in doc_id
        tf_td = len(self.index[term][doc_id])
        w_td = 0 if tf_td == 0 else 1 + np.log(tf_td)

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

    # calculate the inverse document frequency of term
    df_t = len(self.get_docs(term))
    N = len(self.doc_list)
    idf_t = 0 if df_t == 0 else np.log(N/df_t)

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

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

# print the indexing time
print("TF-IDF Index built in", end_time - start_time, "seconds.")
```

Calculate W_TD and IDF_T weight and record them to the index

TOKENIZE

```
# convert an input string into a list of tokenized terms and bypass stop words
def tokenize(self, string):
    stream = self.analyzer.tokenStream("", string)
    stream.reset()

    tokens = []
    while stream.incrementToken():
        token = stream.getAttribute(CharTermAttribute.class_).toString()
        if token not in self.stop_list:
            tokens.append(token)

    stream.close()
    return tokens
```

Use the PyLucene's Standard Analyzer to tokenize terms in dictionary of the index, stop words, and query terms

GET DOC VECTORS

Generate document vectors of the input docs list to later used in Rocchio function to get positive doc vectors and negative doc vectors

```
# function to get the doc vectors of a list of docs
def get_doc_vectors(self, docs):
    doc_vectors = collections.defaultdict(float)

    # iterate through each document
    for doc_id in docs:

        # get the term vector of the current document
        term_vector = self.reader.getTermVector(doc_id, "content")

        # check if there is term in the document
        if term_vector is not None:
            term_iter = BytesRefIterator.cast_(term_vector.iterator())

            # iterate through each term in the document
            while term_iter.next():
                # convert the term from UTF-8 byte format to string
                term = TermsEnum.cast_(term_iter).term().utf8ToString()

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

                # get the wtd weight of the term in the document with doc_id
                w_td = self.get_w_td(term, doc_id)

                # calculate the tf-idf weight
                tf_idf = w_td * w_tq

                # record the term weight in the document vector
                if tf_idf != 0:
                    doc_vectors[term] = tf_idf

    return doc_vectors
```

ROCCHIO

```python
# function to implement rocchio algorithm
# pos_feedback - documents deemed to be relevant by the user
# neg_feedback - documents deemed to be non-relevant by the user
# return the new query terms and their weights
def rocchio(self, query_terms, pos_feedback, neg_feedback, alpha, beta, gamma):

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

    # construct a variable to store original vector
    q0 = {}

    # consider weight = term frequency in query for free text query
    if isinstance(query_terms, str):
        for query_term in set(self.tokenize(query_terms)):
            q0[query_term] = query_terms.count(query_term)

    # consider the term weight calculate from previous Rocchio for query vector
    else:
        q0 = query_terms

    # get the document vectors of relevant documents
    pos = self.get_doc_vectors(pos_feedback)

    # get the document vectors of non-relevant documents
    neg = self.get_doc_vectors(neg_feedback)

    # generate list of unique terms from original query, positive document vectors, and negative document vectors
    terms = list(q0.keys()) + list(pos.keys()) + list(neg.keys())

    # construct a variable to store new query vector
    qm = collections.defaultdict(float)

    # iterate over each time and compute weight for the new query using Rocchio algorithm
    for term in terms:
        value = alpha * q0.get(term, 0) + beta * 1/len(pos_feedback) * pos.get(term, 0) - gamma * 1/len(neg_feedback) * neg.get(term, 0)
        if value > 0:
            qm[term] = value

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

    # print the result
    print("New query computed in", end_time - start_time, "seconds.")
    print("New query terms with weights:")
    print(dict(qm))
    return qm
```

CALCULATE COSINE SIMILARITY

```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, term_weight=None):

    # get idf weight of the term
    w_tq = self.get_idf_t(query_term) if term_weight == None else term_weight

    # iterate through each document
    for doc_id in docs:

        # get the wtd weight of the term in the document with doc_id
        w_td = self.get_w_td(query_term, doc_id)

        # calculate the tf-idf weight
        tf_idf = w_td * w_tq

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

- Free text query: W_TD = inverse document frequency
- Query computed by Rocchio: W_TD = term weight

GET 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
```

CALCULATE PRECISION, RECALL, AND MAP

```python
# function to calculate precision, recall, and MAP
def cal_precision_recall_map(result, relevant_docs):
    relevant_retrieved_docs = []
    map = 0

    for i, doc_id in enumerate(result):
        if doc_id in relevant_docs:
            relevant_retrieved_docs.append(doc_id)
            map += len(relevant_retrieved_docs)/(i + 1)

    precision = len(relevant_retrieved_docs)/len(result)
    recall = len(relevant_retrieved_docs)/len(relevant_docs)
    map = map/len(relevant_docs)

    print("Relevant docs that are retrieved:", relevant_retrieved_docs)
    print("Precision:", precision)
    print("Recall:", recall)
    print("MAP:", map)
```

EXPERIMENTAL RESULT

Generate list of queries from TIME.QUE and assign unique Query ID for each query. Prompt user to enter the Query ID and value of k to retrieve top k documents.

```
TF-IDF Index built in 2.205167770385742 seconds.

Enter query id (1 - 83): 49
Query: BACKGROUND OF THE NEW CHANCELLOR OF WEST GERMANY, LUDWIG ERHARD .
Relevant docs: [46, 55, 80, 102, 149, 182, 204, 290]

Enter value of k: 10
```

EXPERIMENTAL RESULT

Top k documents are displayed together with processing time.

Precision, Recall, and MAP are also calculated.

```
TF-IDF Index built in 2.2098071575164795 seconds.

Enter query id (1 - 83): 49
Query: BACKGROUND OF THE NEW CHANCELLOR OF WEST GERMANY, LUDWIG ERHARD .
Relevant docs: [46, 55, 80, 102, 149, 182, 204, 290]

Enter value of k: 10


Query to search: BACKGROUND OF THE NEW CHANCELLOR OF WEST GERMANY, LUDWIG ERHARD .
Number of (top) results: 10

Top 10 results(s) for the query ' BACKGROUND OF THE NEW CHANCELLOR OF WEST GERMANY, LUDWIG ERHARD . ' are:
Doc id, Doc Name, Score
182, TEXT 241.txt, 0.1400508323681737
204, TEXT 263.txt, 0.13574438576378314
102, TEXT 145.txt, 0.1345269523585255
149, TEXT 198.txt, 0.12076082048721613
55, TEXT 088.txt, 0.1015827772654413
290, TEXT 367.txt, 0.07723984735679772
374, TEXT 507.txt, 0.07427410459477421
377, TEXT 511.txt, 0.0677968325235338
101, TEXT 144.txt, 0.06406101791762919
379, TEXT 513.txt, 0.05685285803771196

Results found in 1.2297241687774658 seconds.

Relevant docs that are retrieved: [182, 204, 102, 149, 55, 290]
Precision: 0.6
Recall: 0.75
MAP: 0.75
```

EXPERIMENTAL RESULT

Rocchio: Prompt user to enter relevant document ids and non relevant document ids to generate a new query with weights using Rocchio algorithm

EXPERIMENTAL RESULT

```
Continue with new query (y/n): y

Top 10 results(s) for the query are:
Doc id, Doc Name, Score
102, TEXT 145.txt, 0.3946204420839721
182, TEXT 241.txt, 0.3926963732077821
204, TEXT 263.txt, 0.3753312624269884
149, TEXT 198.txt, 0.3665662168822284
55, TEXT 088.txt, 0.34606227319400695
290, TEXT 367.txt, 0.30521736452859816
80, TEXT 116.txt, 0.13572266771292513
81, TEXT 117.txt, 0.0920210521879586
123, TEXT 172.txt, 0.09031794945101145
350, TEXT 471.txt, 0.08487161313029488

Results found in 1.6113231182098389 seconds.
```

```
Relevant docs that are retrieved: [102, 182, 204, 149, 55, 290, 80]
Precision: 0.7
Recall: 0.875
MAP: 0.875
```

Ask user if they want to retrieve top k document from the new computed query.

Retrieve top k documents together with Precision, Recall, and MAP if they enter “y”.

EXPERIMENTAL RESULT

Query: BACKGROUND OF THE NEW CHANCELLOR OF WEST GERMANY, LUDWIG ERHARD .

Relevant docs (result): 46, 55, 80, 102, 149, 182, 204, 290

Retrieving top 10 documents at first try:

```
Query to search: BACKGROUND OF THE NEW CHANCELLOR OF WEST GERMANY, LUDWIG ERHARD .
Number of (top) results: 10

Top 10 results(s) for the query ' BACKGROUND OF THE NEW CHANCELLOR OF WEST GERMANY, LUDWIG ERHARD . ' are:
Doc id, Doc Name, Score
182, TEXT 241.txt, 0.1400508323681737
204, TEXT 263.txt, 0.13574438576378314
102, TEXT 145.txt, 0.1345269523585255
149, TEXT 198.txt, 0.12076082048721613
55, TEXT 088.txt, 0.1015827772654413
290, TEXT 367.txt, 0.07723984735679772
374, TEXT 507.txt, 0.07427410459477421
377, TEXT 511.txt, 0.0677968325235338
101, TEXT 144.txt, 0.06406101791762919
379, TEXT 513.txt, 0.05685285803771196

Results found in 1.2297241687774658 seconds.

Relevant docs that are retrieved: [182, 204, 102, 149, 55, 290]
Precision: 0.6
Recall: 0.75
MAP: 0.75
```

EXPERIMENTAL RESULT

Query: BACKGROUND OF THE NEW CHANCELLOR OF WEST GERMANY, LUDWIG ERHARD .

Relevant docs (result): 46, 55, 80, 102, 149, 182, 204, 290

Result of top 10 retrieved documents after using new query computed from Rocchio show increase values in Precision, Recall, and MAP

```
=== Rocchio Algorithm ===

Iteration: 1
Enter relevant document ids separated by space: 182 204 102 149 55 290
Enter non relevant document ids separated by space: 377 379 101 374
New query computed in 0.029359102249145508 seconds.

Continue with new query (y/n): y

Top 10 results(s) for the query are:
Doc id, Doc Name, Score
102, TEXT 145.txt, 0.3946204420839721
182, TEXT 241.txt, 0.3926963732077821
204, TEXT 263.txt, 0.3753312624269884
149, TEXT 198.txt, 0.3665662168822284
55, TEXT 088.txt, 0.34606227319400695
290, TEXT 367.txt, 0.30521736452859816
80, TEXT 116.txt, 0.13572266771292513
81, TEXT 117.txt, 0.0920210521879586
123, TEXT 172.txt, 0.09031794945101145
350, TEXT 471.txt, 0.08487161313029488

Results found in 1.6113231182098389 seconds.

Relevant docs that are retrieved: [102, 182, 204, 149, 55, 290, 80]
Precision: 0.7
Recall: 0.875
MAP: 0.875
```

EXPERIMENTAL RESULT (Query ID: 6, k = 10)

```
Enter query id (1 – 83): 6
Query: CEREMONIAL SUICIDES COMMITTED BY SOME BUDDHIST MONKS IN SOUTH VIET NAMAND WHAT THEY ARE SEEKING
TO GAIN BY SUCH ACTS .
Relevant docs: [256, 267, 287, 303, 307, 322, 323, 325, 333]

Enter value of k: 10


Query to search: CEREMONIAL SUICIDES COMMITTED BY SOME BUDDHIST MONKS IN SOUTH VIET NAMAND WHAT THEY AR
E SEEKING TO GAIN BY SUCH ACTS .
Number of (top) results: 10

Top 10 results(s) for the query ' CEREMONIAL SUICIDES COMMITTED BY SOME BUDDHIST MONKS IN SOUTH VIET NA
MAND WHAT THEY ARE SEEKING TO GAIN BY SUCH ACTS . ' are:
Doc id, Doc Name, Score
153, TEXT 202.txt, 0.11298880411602032
322, TEXT 414.txt, 0.10688163740560712
267, TEXT 334.txt, 0.09912824445654345
418, TEXT 559.txt, 0.08849517966557635
256, TEXT 320.txt, 0.08208084318592686
28, TEXT 052.txt, 0.07674103430860056
287, TEXT 363.txt, 0.07316646743298491
399, TEXT 538.txt, 0.06901051701193696
105, TEXT 148.txt, 0.06235690898885433
258, TEXT 322.txt, 0.060230098048019605

Results found in 0.6300616264343262 seconds.


Relevant docs that are retrieved: [322, 267, 256, 287]
Precision: 0.4
Recall: 0.4444444444444444
MAP: 0.25978835978835974
```

```
=== Rocchio Algorithm ===

Iteration: 1
Enter relevant document ids separated by space: 256 322 267 287
Enter non relevant document ids separated by space: 418 258 105 399 153 28
New query computed in 0.016041040420532227 seconds.

Continue with new query (y/n): y

Top 10 results(s) for the query are:
Doc id, Doc Name, Score
267, TEXT 334.txt, 0.636761082577146
287, TEXT 363.txt, 0.6016974609417487
322, TEXT 414.txt, 0.5587385133707808
256, TEXT 320.txt, 0.546864917620947
303, TEXT 390.txt, 0.22583208073424774
307, TEXT 396.txt, 0.204814349725385
375, TEXT 508.txt, 0.17634251336049372
382, TEXT 518.txt, 0.13829436392181554
170, TEXT 228.txt, 0.1382244197302382
325, TEXT 418.txt, 0.13710386158854196

Results found in 1.6191082000732422 seconds.


Relevant docs that are retrieved: [267, 287, 322, 256, 303, 307, 325]
Precision: 0.7
Recall: 0.7777777777777778
MAP: 0.7444444444444445
```

EXPERIMENTAL RESULT (Query ID: 6, k = 10)

```
=== Rocchio Algorithm ===

Iteration: 2
Enter relevant document ids separated by space: 256 322 325 267 303 307 287
Enter non relevant document ids separated by space: 170 382 375
New query computed in 0.030399084091186523 seconds.

Continue with new query (y/n): y

Top 10 results(s) for the query are:
Doc id, Doc Name, Score
267, TEXT 334.txt, 0.9952595896027058
287, TEXT 363.txt, 0.9338226112955312
322, TEXT 414.txt, 0.8618983906524593
256, TEXT 320.txt, 0.8396989057130667
303, TEXT 390.txt, 0.5229975641487435
307, TEXT 396.txt, 0.49711017722834616
325, TEXT 418.txt, 0.4403385006127169
333, TEXT 434.txt, 0.2497558094762932
375, TEXT 508.txt, 0.24666591406826274
418, TEXT 559.txt, 0.2399745463847525

Results found in 1.6263728141784668 seconds.


Relevant docs that are retrieved: [267, 287, 322, 256, 303, 307, 325, 333]
Precision: 0.8
Recall: 0.8888888888888888
MAP: 0.8888888888888888
```

```
=== Rocchio Algorithm ===

Iteration: 3
Enter relevant document ids separated by space: 256 322 325 267 333 303 307 287
Enter non relevant document ids separated by space: 418 375
New query computed in 0.0321040153503418 seconds.

Continue with new query (y/n): y

Top 10 results(s) for the query are:
Doc id, Doc Name, Score
267, TEXT 334.txt, 1.3039523519686296
287, TEXT 363.txt, 1.2217528064591365
322, TEXT 414.txt, 1.1263645387197856
256, TEXT 320.txt, 1.0917215986216804
303, TEXT 390.txt, 0.7780942048011142
307, TEXT 396.txt, 0.7475194280814729
325, TEXT 418.txt, 0.6999622992776349
333, TEXT 434.txt, 0.4857184620580583
369, TEXT 498.txt, 0.3265620902838989
323, TEXT 415.txt, 0.3217903916940147

Results found in 1.6324830055236816 seconds.


Relevant docs that are retrieved: [267, 287, 322, 256, 303, 307, 325, 333, 323]
Precision: 0.9
Recall: 1.0
MAP: 0.9888888888888889
```

EXPERIMENTAL RESULT (Query ID: 6, k = 10)

```
=== Rocchio Algorithm ===

Iteration: 4
Enter relevant document ids separated by space: 256 322 323 325 267 333 303 307 287
Enter non relevant document ids separated by space: 369
New query computed in 0.041229248046875 seconds.

Continue with new query (y/n): y

Top 10 results(s) for the query are:
Doc id, Doc Name, Score
267, TEXT 334.txt, 1.4434291536867931
287, TEXT 363.txt, 1.39604305590892
322, TEXT 414.txt, 1.2246161744699817
256, TEXT 320.txt, 1.1963119647178804
303, TEXT 390.txt, 0.8716479120694061
307, TEXT 396.txt, 0.8571891729242128
325, TEXT 418.txt, 0.8307303270000649
333, TEXT 434.txt, 0.5794569037233348
323, TEXT 415.txt, 0.5171321854827048
397, TEXT 536.txt, 0.35799637259495526

Results found in 1.6726620197296143 seconds.


Relevant docs that are retrieved: [267, 287, 322, 256, 303, 307, 325, 333, 323]
Precision: 0.9
Recall: 1.0
MAP: 1.0
```

```
=== Rocchio Algorithm ===

Iteration: 5
Enter relevant document ids separated by space: 256 322 323 325 267 333 303 307 287
Enter non relevant document ids separated by space: 397
New query computed in 0.03779911994934082 seconds.

Continue with new query (y/n): y

Top 10 results(s) for the query are:
Doc id, Doc Name, Score
267, TEXT 334.txt, 1.7093697000881054
287, TEXT 363.txt, 1.6458302148098585
322, TEXT 414.txt, 1.4634386983395973
256, TEXT 320.txt, 1.4148877121456034
303, TEXT 390.txt, 1.1036050880733654
307, TEXT 396.txt, 1.0766082272194484
325, TEXT 418.txt, 1.0593290722103372
333, TEXT 434.txt, 0.7951760541234347
323, TEXT 415.txt, 0.7380550929519196
180, TEXT 239.txt, 0.4374907078062526

Results found in 1.610870122909546 seconds.


Relevant docs that are retrieved: [267, 287, 322, 256, 303, 307, 325, 333, 323]
Precision: 0.9
Recall: 1.0
MAP: 1.0
```

EXPERIMENTAL RESULT (Query ID: 6, k = 10)

Query: CEREMONIAL SUICIDES COMMITTED BY SOME BUDDHIST MONKS IN SOUTH VIET NAMAND WHAT THEY ARE SEEKING TO GAIN BY SUCH ACTS .

Relevant docs: [256, 267, 287, 303, 307, 322, 323, 325, 333]

	First try	Rocchio 1st iteration	Rocchio 2nd iteration	Rocchio 3rd iteration	Rocchio 4th iteration	Rocchio 5th iteration
Precision	0.4	0.7	0.8	0.8	0.8	0.8
Recall	0.44	0.78	0.89	0.89	0.89	0.89
MAP	0.26	0.75	0.87	0.89	0.89	0.89

EXPERIMENTAL RESULT (Query ID: 61, k = 10)

```
Enter query id (1 - 83): 61
Query: NATIONS WORKING ON NUCLEAR WEAPONS DEVELOPMENT .
Relevant docs: [0, 22, 37, 46, 134, 156, 173, 192, 227, 246, 253, 294, 314, 342, 362]

Enter value of k: 10


Query to search: NATIONS WORKING ON NUCLEAR WEAPONS DEVELOPMENT .
Number of (top) results: 10

Top 10 results(s) for the query ' NATIONS WORKING ON NUCLEAR WEAPONS DEVELOPMENT . ' are:
Doc id, Doc Name, Score
362, TEXT 491.txt, 0.07142394197252583
134, TEXT 183.txt, 0.052158418951935716
401, TEXT 540.txt, 0.050888421720024134
309, TEXT 399.txt, 0.04910685771653313
173, TEXT 231.txt, 0.04854583953669237
422, TEXT 563.txt, 0.04785692054135737
81, TEXT 117.txt, 0.04729181186957344
227, TEXT 287.txt, 0.044168043347932985
253, TEXT 317.txt, 0.04264675843750086
190, TEXT 249.txt, 0.041390065789649357

Results found in 0.7310450077056885 seconds.


Relevant docs that are retrieved: [362, 134, 173, 227, 253]
Precision: 0.5
Recall: 0.3333333333333333
MAP: 0.2437037037037037
```

```
=== Rocchio Algorithm ===

Iteration: 1
Enter relevant document ids separated by space: 227 134 362 173 253
Enter non relevant document ids separated by space: 422 81 401 309 190
New query computed in 0.015700101852416992 seconds.

Continue with new query (y/n): y

Top 10 results(s) for the query are:
Doc id, Doc Name, Score
362, TEXT 491.txt, 0.43551302645865064
173, TEXT 231.txt, 0.41843309170808396
227, TEXT 287.txt, 0.40475678772151014
253, TEXT 317.txt, 0.3962851239065505
134, TEXT 183.txt, 0.37859657271745123
246, TEXT 308.txt, 0.17634418457759465
88, TEXT 126.txt, 0.160072184823663
156, TEXT 213.txt, 0.15042148226271046
150, TEXT 199.txt, 0.14214694140168804
53, TEXT 086.txt, 0.12924374292314328

Results found in 1.6220958232879639 seconds.


Relevant docs that are retrieved: [362, 173, 227, 253, 134, 246, 156]
Precision: 0.7
Recall: 0.4666666666666667
MAP: 0.4583333333333333
```

EXPERIMENTAL RESULT (Query ID: 61, k = 10)

```
=== Rocchio Algorithm ===

Iteration: 2
Enter relevant document ids separated by space: 227 134 362 173 246 156 253
Enter non relevant document ids separated by space: 88 53 150
New query computed in 0.020465373992919922 seconds.

Continue with new query (y/n): y

Top 10 results(s) for the query are:
Doc id, Doc Name, Score
362, TEXT 491.txt, 0.7355870786691245
173, TEXT 231.txt, 0.6920836501811184
227, TEXT 287.txt, 0.6628676708441715
253, TEXT 317.txt, 0.644184886668833
134, TEXT 183.txt, 0.6211823242137119
246, TEXT 308.txt, 0.45195078007090933
156, TEXT 213.txt, 0.3948764716146151
88, TEXT 126.txt, 0.24430998866016934
150, TEXT 199.txt, 0.2282824972819299
401, TEXT 540.txt, 0.22326150042608786

Results found in 1.6239829063415527 seconds.


Relevant docs that are retrieved: [362, 173, 227, 253, 134, 246, 156]
Precision: 0.7
Recall: 0.4666666666666667
MAP: 0.4666666666666667
```

```
=== Rocchio Algorithm ===

Iteration: 3
Enter relevant document ids separated by space: 227 134 362 173 246 156 253
Enter non relevant document ids separated by space: 88 401 150
New query computed in 0.020412921905517578 seconds.

Continue with new query (y/n): y

Top 10 results(s) for the query are:
Doc id, Doc Name, Score
362, TEXT 491.txt, 1.0236132434273204
173, TEXT 231.txt, 0.9700900233490466
227, TEXT 287.txt, 0.9227976151101698
253, TEXT 317.txt, 0.8950532260901948
134, TEXT 183.txt, 0.8540750655845118
246, TEXT 308.txt, 0.7301317586746764
156, TEXT 213.txt, 0.6404869043249539
88, TEXT 126.txt, 0.3256978393134213
150, TEXT 199.txt, 0.3144180531621719
22, TEXT 045.txt, 0.31267656800378796

Results found in 1.6169626712799072 seconds.


Relevant docs that are retrieved: [362, 173, 227, 253, 134, 246, 156, 22]
Precision: 0.8
Recall: 0.5333333333333333
MAP: 0.52
```

EXPERIMENTAL RESULT (Query ID: 61, k = 10)

```
=== Rocchio Algorithm ===

Iteration: 4
Enter relevant document ids separated by space: 227 134 362 173 246 22 156 253
Enter non relevant document ids separated by space: 88 150
New query computed in 0.019262790679931640 seconds.

Continue with new query (y/n): y

Top 10 results(s) for the query are:
Doc id, Doc Name, Score
362, TEXT 491.txt, 1.2593863002745007
173, TEXT 231.txt, 1.1859942804639791
227, TEXT 287.txt, 1.1288180805835268
253, TEXT 317.txt, 1.08642999553945
134, TEXT 183.txt, 1.0286400798911652
246, TEXT 308.txt, 0.9433886777587239
156, TEXT 213.txt, 0.8309393559144602
22, TEXT 045.txt, 0.574184903167133
53, TEXT 086.txt, 0.3900980610977917
401, TEXT 540.txt, 0.37175965946617306

Results found in 1.6191458702087402 seconds.


Relevant docs that are retrieved: [362, 173, 227, 253, 134, 246, 156, 22]
Precision: 0.8
Recall: 0.5333333333333333
MAP: 0.5333333333333333
```

```
=== Rocchio Algorithm ===

Iteration: 5
Enter relevant document ids separated by space: 227 134 362 173 246 22 156 253
Enter non relevant document ids separated by space: 401 53
New query computed in 0.01997089385986328 seconds.

Continue with new query (y/n): y

Top 10 results(s) for the query are:
Doc id, Doc Name, Score
362, TEXT 491.txt, 1.4870791860167865
173, TEXT 231.txt, 1.3940061634069132
227, TEXT 287.txt, 1.336645149623942
253, TEXT 317.txt, 1.28709061861165
134, TEXT 183.txt, 1.2189413903856054
246, TEXT 308.txt, 1.1640066244834777
156, TEXT 213.txt, 1.0131440484907164
22, TEXT 045.txt, 0.7977544757761587
88, TEXT 126.txt, 0.43298705851961017
150, TEXT 199.txt, 0.4303001851967007

Results found in 1.6094861030578613 seconds.


Relevant docs that are retrieved: [362, 173, 227, 253, 134, 246, 156, 22]
Precision: 0.8
Recall: 0.5333333333333333
MAP: 0.5333333333333333
```

EXPERIMENTAL RESULT (Query ID: 61, k = 10)

Query: NATIONS WORKING ON NUCLEAR WEAPONS DEVELOPMENT .

Relevant docs: [0, 22, 37, 46, 134, 156, 173, 192, 227, 246, 253, 294, 314, 342, 362]

	First try	Rocchio 1st iteration	Rocchio 2nd iteration	Rocchio 3rd iteration	Rocchio 4th iteration	Rocchio 5th iteration
Precision	0.5	0.7	0.7	0.8	0.8	0.8
Recall	0.33	0.47	0.47	0.53	0.53	0.53
MAP	0.24	0.46	0.47	0.52	0.53	0.53

EXPERIMENTAL RESULT (Query ID: 40, k = 10)

```
Enter query id (1 - 83): 40
Query: RESULTS OF THE POLITICAL POLLS IN BRITAIN REGARDING WHICH PARTYIS IN THE LEAD, THE LABOR PARTY O
R THE CONSERVATIVES .
Relevant docs: [19, 70, 130, 147, 181, 206, 260, 271, 324]

Enter value of k: 10


Query to search: RESULTS OF THE POLITICAL POLLS IN BRITAIN REGARDING WHICH PARTYIS IN THE LEAD, THE LAB
OR PARTY OR THE CONSERVATIVES .
Number of (top) results: 10

Top 10 results(s) for the query ' RESULTS OF THE POLITICAL POLLS IN BRITAIN REGARDING WHICH PARTYIS IN
THE LEAD, THE LABOR PARTY OR THE CONSERVATIVES . ' are:
Doc id, Doc Name, Score
206, TEXT 265.txt, 0.18714427953082313
324, TEXT 417.txt, 0.11104932515963645
57, TEXT 091.txt, 0.09775328789157688
368, TEXT 497.txt, 0.08814582770322116
228, TEXT 288.txt, 0.06090683992768011
181, TEXT 240.txt, 0.06071379302929944
147, TEXT 196.txt, 0.060375597975002936
242, TEXT 304.txt, 0.05769579481018507
199, TEXT 258.txt, 0.05591558233460067
19, TEXT 040.txt, 0.05380891841761285

Results found in 1.1327948570251465 seconds.


Relevant docs that are retrieved: [206, 324, 181, 147, 19]
Precision: 0.5
Recall: 0.5555555555555556
MAP: 0.3968253968253968
```

```
=== Rocchio Algorithm ===

Iteration: 1
Enter relevant document ids separated by space: 324 206 19 147 181
Enter non relevant document ids separated by space: 228 199 368 242 57
New query computed in 0.019414186477661133 seconds.

Continue with new query (y/n): y

Top 10 results(s) for the query are:
Doc id, Doc Name, Score
181, TEXT 240.txt, 0.4383595760645669
324, TEXT 417.txt, 0.437059027087591
19, TEXT 040.txt, 0.41476045649406296
147, TEXT 196.txt, 0.40619963034541584
206, TEXT 265.txt, 0.37235953891795454
57, TEXT 091.txt, 0.16356782485676535
389, TEXT 526.txt, 0.14849063468319518
368, TEXT 497.txt, 0.13205779965384307
271, TEXT 341.txt, 0.13048886227748507
38, TEXT 063.txt, 0.12642601938426518

Results found in 1.6315069198608398 seconds.


Relevant docs that are retrieved: [181, 324, 19, 147, 206, 271]
Precision: 0.6
Recall: 0.6666666666666666
MAP: 0.6296296296296297
```

EXPERIMENTAL RESULT (Query ID: 40, k = 10)

```
=== Rocchio Algorithm ===

Iteration: 2
Enter relevant document ids separated by space: 324 206 271 19 147 181
Enter non relevant document ids separated by space: 368 57 389 38
New query computed in 0.020596981048583984 seconds.

Continue with new query (y/n): y

Top 10 results(s) for the query are:
Doc id, Doc Name, Score
181, TEXT 240.txt, 0.7938565805704197
324, TEXT 417.txt, 0.7812352380348208
19, TEXT 040.txt, 0.7480596162176236
147, TEXT 196.txt, 0.7295051601658001
206, TEXT 265.txt, 0.6715964975030604
271, TEXT 341.txt, 0.45583628769058077
57, TEXT 091.txt, 0.28902232949062867
389, TEXT 526.txt, 0.2527873208388444
368, TEXT 497.txt, 0.2388202337798522
253, TEXT 317.txt, 0.2307460491369091

Results found in 1.6373488903045654 seconds.


Relevant docs that are retrieved: [181, 324, 19, 147, 206, 271]
Precision: 0.6
Recall: 0.6666666666666666
MAP: 0.6666666666666666
```

```
=== Rocchio Algorithm ===

Iteration: 3
Enter relevant document ids separated by space: 324 206 271 19 147 181
Enter non relevant document ids separated by space: 368 57 389 253
New query computed in 0.02012324333190918 seconds.

Continue with new query (y/n): y

Top 10 results(s) for the query are:
Doc id, Doc Name, Score
181, TEXT 240.txt, 1.1559476324114664
324, TEXT 417.txt, 1.130577959758559
19, TEXT 040.txt, 1.0877732014824666
147, TEXT 196.txt, 1.0498860725999117
206, TEXT 265.txt, 0.976481284179173
271, TEXT 341.txt, 0.7794430956484977
57, TEXT 091.txt, 0.4202227647523418
389, TEXT 526.txt, 0.357629650772853
368, TEXT 497.txt, 0.3490713767710151
183, TEXT 242.txt, 0.33416128394563893

Results found in 1.6324169635772705 seconds.


Relevant docs that are retrieved: [181, 324, 19, 147, 206, 271]
Precision: 0.6
Recall: 0.6666666666666666
MAP: 0.6666666666666666
```

EXPERIMENTAL RESULT (Query ID: 40, k = 10)

```
=== Rocchio Algorithm ===

Iteration: 4
Enter relevant document ids separated by space: 324 206 271 19 147 181
Enter non relevant document ids separated by space: 368 57 389 183
New query computed in 0.01933908462524414 seconds.

Continue with new query (y/n): y

Top 10 results(s) for the query are:
Doc id, Doc Name, Score
181, TEXT 240.txt, 1.5192156260507617
324, TEXT 417.txt, 1.4818640425327052
19, TEXT 040.txt, 1.428992152126755
147, TEXT 196.txt, 1.3818952177654156
206, TEXT 265.txt, 1.2813934385052714
271, TEXT 341.txt, 1.1103555249833628
57, TEXT 091.txt, 0.5512900740818015
389, TEXT 526.txt, 0.46359261403272084
368, TEXT 497.txt, 0.4585048195672117
70, TEXT 105.txt, 0.4416307196833147

Results found in 1.621920108795166 seconds.


Relevant docs that are retrieved: [181, 324, 19, 147, 206, 271, 70]
Precision: 0.7
Recall: 0.7777777777777778
MAP: 0.7444444444444445
```

```
=== Rocchio Algorithm ===

Iteration: 5
Enter relevant document ids separated by space: 324 70 206 271 19 147 181
Enter non relevant document ids separated by space: 368 57 389
New query computed in 0.021792888641357422 seconds.

Continue with new query (y/n): y

Top 10 results(s) for the query are:
Doc id, Doc Name, Score
181, TEXT 240.txt, 1.8210647786603846
324, TEXT 417.txt, 1.7626269830681993
19, TEXT 040.txt, 1.7148791556128062
147, TEXT 196.txt, 1.6597791690818366
206, TEXT 265.txt, 1.534451828760262
271, TEXT 341.txt, 1.3913860230329842
70, TEXT 105.txt, 0.711378746103241
57, TEXT 091.txt, 0.6629974496155538
38, TEXT 063.txt, 0.5502920377833589
389, TEXT 526.txt, 0.5442233295282881

Results found in 1.618438959121704 seconds.


Relevant docs that are retrieved: [181, 324, 19, 147, 206, 271, 70]
Precision: 0.7
Recall: 0.7777777777777778
MAP: 0.7777777777777778
```

EXPERIMENTAL RESULT (Query ID: 40, k = 10)

Query: RESULTS OF THE POLITICAL POLLS IN BRITAIN REGARDING WHICH PARTY IS IN THE LEAD, THE LABOR PARTY OR THE CONSERVATIVES .

Relevant docs: [19, 70, 130, 147, 181, 206, 260, 271, 324]

	First try	Rocchio 1st iteration	Rocchio 2nd iteration	Rocchio 3rd iteration	Rocchio 4th iteration	Rocchio 5th iteration
Precision	0.5	0.6	0.6	0.6	0.7	0.7
Recall	0.36	0.67	0.67	0.67	0.78	0.78
MAP	0.40	0.63	0.67	0.67	0.74	0.78

CONCLUSION

- The values of Precision, Recall, and MAP gradually increase over each Rocchio iterations

  → It shows that the performance of the ranked retrieval improves by using Rocchio algorithm with relevance feedback. The system is able to find relevant documents more accurately and rank them on top.

- The weight of rarer terms (terms that help indicate relevant and non-relevant documents) in the new computed query also increase over Rocchio iterations

  → Adding more weight to important terms helps distinguish relevant and non-relevant documents

EXTRA CREDIT

Pseudo Relevance Feedback: Top 3 results of the system are considered to be relevant

```python
def pseudo_relevance_feedback(self, query_terms, k, iterations, relevant_docs):
    # retrieve top k documents
    result = self.query(query_terms, k)

    # calculate the precision, recall, and MAP
    cal_precision_recall_map(result, relevant_docs)

    alpha = 1
    beta = 0.75
    gamma = 0.15

    # loop over i Rocchio iterations
    for i in range(iterations):
        print("\n\n=== Rocchio Algorithm ===")
        print("\nIteration:", i + 1)

        # assume top 3 documents are relevant and the rest are non-relevant
        pos_feedback = result[:3]
        neg_feedback = result[3:]

        # print out the relevant and non-relevant doc ids
        print("Relevant document ids:", pos_feedback)
        print("Non relevant document ids:", neg_feedback)

        # compute new query using Rocchio algorithm
        query_terms = self.rocchio(query_terms, pos_feedback, neg_feedback, alpha, beta, gamma)

        # retrieve top k documents from the new computed query
        result = self.query(query_terms, k)

        # calculate the precision, recall, and MAP of the result from new computed query
        cal_precision_recall_map(result, relevant_docs)
```

EXTRA CREDIT (Query ID: 6, k = 10)

USER FEEDBACK	First try	Rocchio 1st iteration	Rocchio 2nd iteration	Rocchio 3rd iteration	Rocchio 4th iteration	Rocchio 5th iteration
Precision	0.4	0.7	0.8	0.8	0.8	0.8
Recall	0.44	0.78	0.89	0.89	0.89	0.89
MAP	0.26	0.75	0.87	0.89	0.89	0.89

PSEUDO	First try	Rocchio 1st iteration	Rocchio 2nd iteration	Rocchio 3rd iteration	Rocchio 4th iteration	Rocchio 5th iteration
Precision	0.4	0.6	0.6	0.6	0.6	0.6
Recall	0.44	0.67	0.67	0.67	0.67	0.67
MAP	0.26	0.54	0.57	0.57	0.57	0.57

EXTRA CREDIT (Query ID: 61, k = 10)

USER FEEDBACK	First try	Rocchio 1st iteration	Rocchio 2nd iteration	Rocchio 3rd iteration	Rocchio 4th iteration	Rocchio 5th iteration
Precision	0.5	0.7	0.7	0.8	0.8	0.8
Recall	0.33	0.47	0.47	0.53	0.53	0.53
MAP	0.24	0.46	0.47	0.52	0.53	0.53

PSEUDO	First try	Rocchio 1st iteration	Rocchio 2nd iteration	Rocchio 3rd iteration	Rocchio 4th iteration	Rocchio 5th iteration
Precision	0.5	0.6	0.7	0.7	0.7	0.7
Recall	0.33	0.4	0.47	0.47	0.47	0.47
MAP	0.24	0.29	0.35	0.35	0.35	0.35

EXTRA CREDIT (Query ID: 40, k = 10)

USER FEEDBACK	First try	Rocchio 1st iteration	Rocchio 2nd iteration	Rocchio 3rd iteration	Rocchio 4th iteration	Rocchio 5th iteration
Precision	0.5	0.6	0.6	0.6	0.7	0.7
Recall	0.36	0.67	0.67	0.67	0.78	0.78
MAP	0.40	0.63	0.67	0.67	0.74	0.78

PSEUDO	First try	Rocchio 1st iteration	Rocchio 2nd iteration	Rocchio 3rd iteration	Rocchio 4th iteration	Rocchio 5th iteration
Precision	0.5	0.5	0.6	0.6	0.6	0.6
Recall	0.36	0.56	0.67	0.67	0.67	0.67
MAP	0.40	0.29	0.36	0.36	0.36	0.36

EXTRA CREDIT (Conclusion)

- The Pseudo Relevance Feedback with assumption of top 3 documents as relevance does not show better result than the User Relevance Feedback
- The system is able to detect more than 3 relevant documents at the first try in most of the time before using Rocchio to compute more efficient query. Therefore, assuming the top 3 documents as relevance does not help in performance in this case. Thus, the Pseudo Relevance Feedback yields a lower values in Precision, Recall, and MAP than the User Relevance Feedback.
- However, the first two iterations of using new query computed Rocchio does help in improving the Precision, Recall, and MAP of the ranking system. After the third iteration, the values do not change since the system is able to detect more than 3 relevant documents on top at that time.

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