A REPORT

ON

CS F469 Information Retrieval: Assignment 2

by

Name of the student	**ID number**
Dev Gala	2021A7PS0182H
Atharva Dashora	2021A7PS0127H
Sricharan Reddy Bollampalli	2021A7PS0379H
Abhishek Kali Madiki	2021AAPS0550H

BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI
(Hyderabad Campus)

April, 2024

CONTENTS

List of Figures

1 Introduction

As the internet became widespread in the 1990s, the amount of unstructured data also increased. Unstructured data refers to information that does not have a predefined model or cannot be organised in a predefined manner. Unstructured data lacks a specific data model and can include text documents, multimedia content, social media posts, emails, and more. According to [Edg] nearly 80% of the internet's data is unstructured.

To retrieve this unstructured data, simple boolean retrieval models proved to be incompetent. Thus the need of more sophisticated models was eminent. These new models take into account frequency of occurrence, "similarity" of terms and try to model relations between different terms in the corpus.

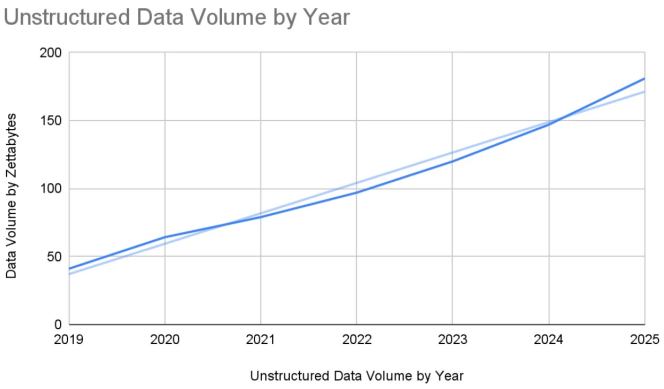


Figure 1: Unstructured Data volume per year

This assignment aims to further our understanding of various information retrieval models like Vector-Space models (2.1) and Probabilistic models. This assignment also helps us understand ranking retrieved documents and use of Knowledge Graphs for IR systems. We conduct 8 experiments to show various retrieval models and how they compare against each other.

2 Concepts

2.1 Vector Space Models

Vector Space models represent documents and queries as vectors. Each term in the document and query (after pre-processing such as stop word removal or stemming). Each dimension has a value that corresponds to relative importance of terms in a document. This importance can be calculated in many ways which are discussed later in the report. Vector space models also enable ranked retrieval where we can score each retrieved document according to its relevance to the query.

2.1.1 Bag of Words Model

In bag of words model, we do not consider the ordering of words in a document.
Example:

Doc1 = "Atharva is taller than Dev".
Doc2: = "Dev is taller than Atharva".

Both these docs will have the same vectors in spite of begin different vectors. This is kind of a step back from the positional indices of boolean retrieval as they can model position of words in a document.

2.1.2 Measuring Relevance

Relevance means how much does the retrieved document meet the information requirement of the query. A simple measure of relevance is term-frequency ($\mathrm{tf}_{t,d}$) i.e. the number of times a term t appears in a document d. Thus for a query q, a document can be considered more relevant if a term $t \in q$ is more frequent in that document. This method, though sensible, does not take into account the amount of information each word adds to the document. It treats common words (in context of a corpus of medical information) such as "*Patient*", "*Symptom*", "*Treatment*", etc. and rare/infrequent terms such as "*arrhythmia*", "*carcinogens*", "*anesthesia*" ,etc. The latter words carry more information than the former as they are rare and may only be present in a few documents.
We thus propose a new measure for relevance along with ($\mathrm{tf}_{t,d}$), Document Frequency (df_t). It is the number of documents in which a particular term appears. (df_t) is an inverse measure of informativeness of term t. We define inverse document frequency (idf_t) as:

$$\mathrm{idf}_t = log_{10}(\frac{N}{\mathrm{df}_t})$$

where N is number of documents in the corpus.

2.1.3 Weighting terms using relevance

Each term of a document can we assigned a weight which depends on its term and document frequency.

$$w_{t,d} = f(\mathrm{tf}_{t,d}, \mathrm{df}_t)$$

These weights form the term-document frequency matrix. The term-document frequency

Terms \ Documents	Document 1	Document 2	Document 3
Term 1	w_{t_1,d_1}	w_{t_1,d_2}	w_{t_1,d_3}
Term 2	w_{t_2,d_1}	w_{t_2,d_2}	w_{t_2,d_3}
Term 3	w_{t_3,d_1}	w_{t_3,d_2}	w_{t_3,d_3}

Figure 2: Term Document Frequency Matrix

matrix is used for the vector representation for a document. Each column (corresponding to a document) is the vector represendation of that document.

$$\vec{d} = (w_{t_1,d}, w_{t_2,d}, w_{t_3,d}, ..., w_{t_k,d}), \text{where } t_1, t_2, ...t_k \in d$$

Weights are calculated as the product of term frequency and inverse document frequency. This ensures that both tf and idf are considered while assigning relevance to retrieved documents. This is called tf-idf weighting. Additionally sometimes normalization is also applied to the document vector.

$$w_{t,d} = \text{tf}_{t,d} \times \text{df}_t$$

2.1.4 SMART Notation

Search engines allow different methods of weighing queries and documents. SMART notation gives the combination to use for document and query weights in the form: *ddd.qqq*. The various ways to calculate tf and idf are given in the below table which form the basis of calculating weights. Most common configuration for Vector Space based search engines is *lnc.ltc*.

Term frequency		Document frequency		Normalization	
n (natural)	$\text{tf}_{t,d}$	n (no)	1	n (none)	1
l (logarithm)	$1 + \log(\text{tf}_{t,d})$	t (idf)	$\log \frac{N}{\text{df}_t}$	c (cosine)	$\frac{1}{\sqrt{w_1^2+w_2^2+...+w_M^2}}$
a (augmented)	$0.5 + \frac{0.5 \times \text{tf}_{t,d}}{\max_t(\text{tf}_{t,d})}$	p (prob idf)	$\max\{0, \log \frac{N-\text{df}_t}{\text{df}_t}\}$	u (pivoted unique)	$1/u$
b (boolean)	$\begin{cases} 1 & \text{if } \text{tf}_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases}$			b (byte size)	$1/CharLength^{\alpha}$, $\alpha < 1$
L (log ave)	$\frac{1+\log(\text{tf}_{t,d})}{1+\log(\text{ave}_{t \in d}(\text{tf}_{t,d}))}$				

Figure 3: SMART Notation

2.1.5 Scoring Documents

Retrieved documents are scored on the basis of their relevance to the information requirement of the query. A higher score implies that the document is more relevant for that query. We use a scoring function to determine scores of each document for given query. The simplest scoring function is just sum of the weights of terms common between the query q and document d

$$score(q, d) = \sum_{t \in q \cap d} \text{tf}_{t,d} \times \text{df}_t$$

Another way of scoring is using the vector representation of documents and queries. The first method that comes up is Euclidean Distance. This will work for most queries but take for example the query q: "Atharva is tall", and document "Atharva is tall Atharva is tall". The document vector has a size twice that of the query. Thus even though they equal semantically, by eucledian distance, they are not similar.

From Figure 4 we can see that if the angle between two vectors is small, it implies that

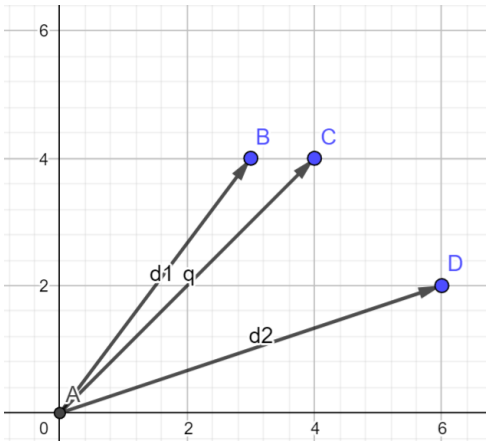


Figure 4: Distance Between Documents and Query

the two vectors are highly similar. Thus we can rank documents in increasing order of angles between the query and document. Thus we can rank documents in decreasing order of the cosines of said angles as cosine function decreases monotonically in interval $[0, \pi]$. The cosine of angle between q and d is given by dot product of their vector representation. Thus new scoring function is given as:

$$score(q, d) = cos\ \theta = \frac{q \cdot d}{||\vec{q}||\ ||\vec{d}||}$$

where $||\vec{q}||$ is the L_2 norm of q. This is also called length normalization. This makes all documents and queries unit length. Thus long and short documents have comparable weights.

2.2 Rocchio Algorithm

2.2.1 Relevance and Pseudo Relevance Feedback

Relevance feedback is the feedback on relevance of documents retrieved for a given query. User gives feedback in different forms like marking document as relevant or providing a relevance rating etc. The IR system then recomputes the representation of the information need based on feedback. Relevance feedback has a few assumptions:

- Assumption 1: The user has enough knowledge for the initial query and the subject
- Assumption 2: Relevance prototypes are "well behaved" i.e relevant documents have similar term distribution which differ significantly from non-relevant documents.

Relevance feedback suffers from problems such has hesitance of users to provide feedback and intuition behind why a particular document is marked relevant.

Pseudo Relevance Feedback automates the manual part of true relevance feedback. It retrieves a ranked list from the query, assumes that top k documents are relevant and does relevance feedback. This method works well on average but can cause query drift i.e the difference in the retrieved topic and actual search topic due to inappropriate query expansion [MSB98].

2.2.2 The Algorithm

The Rocchio Algorithm uses vector space models to pick relevance feedback query. The algorithm aims to find $\vec{q_{\text{opt}}}$ to separate relevant and non-relevant documents.

$$\vec{q}_{\text{opt}} = \max_q\{cos(\vec{q}, \vec{\mu}(C_r)) - cos(\vec{q}, \vec{\mu}(C_{nr}))\}$$

where C_r is set of relevant documents and C_{nr} is set of non-relevant documents and $\vec{\mu}(C)$ is centroid of set C and is defined as

$$\vec{\mu}(C) = \frac{1}{|C|}\sum_{\vec{d}\in C}\vec{d}$$

In practice we use Rocchio 1971 Algorithm (SMART) given by:

$$\vec{q}_m = \alpha\vec{q}_0 + \beta\vec{\mu}(D_r) - \gamma\vec{\mu}(D_{nr})$$

where D_r is set of known relevant documents and D_{nr} is set of known non-relevant documents. The new query moves towards relevant documents and away from non-relevant documents.

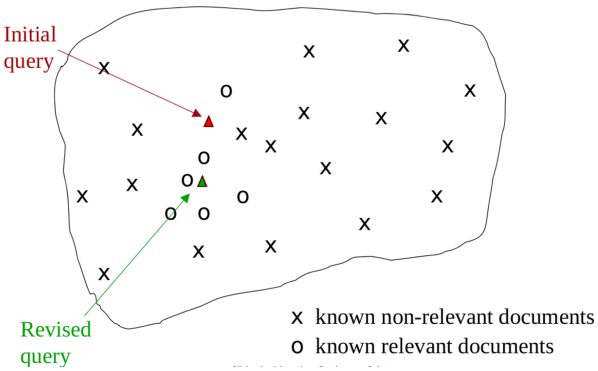


Figure 5: Rocchio Algorithm

2.3 Probabilistic Retrieval Model

An IR system has an uncertain understanding of the user query, and makes an uncertain guess of whether a document satisfies the query. Probabilistic models exploit this foundation to estimate how likely it is that a document is relevant to a query.

2.3.1 Ranked Retrieval

Let $R_{q,d}$ be a dichotomous variable such that

- $R_{q,d} = 1$ if document d is relevant w.r.t query q.
- $R_{q,d} = 0$ otherwise.

Probabilistic ranking orders documents decreasingly by their estimated probability of relevance w.r.t. query: $P(\frac{R=1}{d,q})$
The probability ranking principle [Rob77] states that if the IR system's response to each user query is to rank the documents in order of decreasing probability of usefulness where the probabilities are calculated as accurately as possible based on available data then systems shows the best obtainable effectiveness to the users.

2.3.2 Binary Independence Model (BIM)

This model is traditionally used with probability ranking principle. It is based on the assumption that each document and query can be represented binary term incidence vector. A document d can be represented as $\vec{x} = (x_1, x_2, ..., x_m)$ where $x_t = 1$ if $t \in d$ and $x_t = 0$ otherwise. The 'independence' means that the terms have no association with one another and relevance of each document is independent of the relevance of other documents i.e Naive Bayes assumption.
The probability that a retrieved document is relevant is given by:

$$P(\frac{R=1}{\vec{x},\vec{q}}) = \frac{P(\frac{\vec{x}}{R=1,\vec{q}}) \cdot P(\frac{R=1}{\vec{q}})}{P(\frac{\vec{x}}{\vec{q}})}$$

$$P(\frac{R=0}{\vec{x},\vec{q}}) = \frac{P(\frac{\vec{x}}{R=1,\vec{q}}) \cdot P(\frac{R=0}{\vec{q}})}{P(\frac{\vec{x}}{\vec{q}})}$$

Since documents can be either relevant or non-relevant:

$$P(\frac{R=1}{\vec{x},\vec{q}}) + P(\frac{R=0}{\vec{x},\vec{q}}) = 1$$

We can derive a ranking function using this model which has the following form

$$RSV_d = \sum_{t:x_t=q_t=1} log(\frac{p_t(1-u_t)}{u_t(1-p_t)})$$

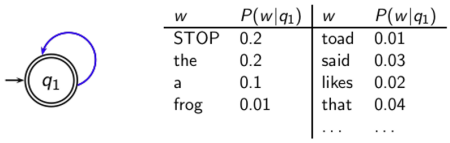
where RSV_d is the Retrieval Status Value for document d, and,

$$p_t = P(\frac{x_t=1}{R=1,\vec{q}}) \text{ and } u_t = P(\frac{x_t=1}{R=0,\vec{q}})$$

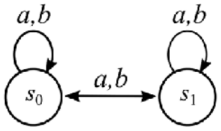
2.3.3 Language Models

Language models are thought of as single state probabilistic finite automata. We can thus determine the probability of generating a string from this model.

In the Language Model approach to IR, we attempt to model the query generation process. We rank documents by the probability that a query would be observed as a random sample from the respective document model i.e ranking is done according to $P(\frac{q}{d})$.



(a) Probabilistic Automata



(b) Finite Automata

Figure 6: Difference in Finite and Probabilistic Automata

$P(\frac{q}{d})$ can be calculated as product of probability of generating each term of the query from machine of document d. Thus $P(\frac{q}{d})$ is given by:

$$P(\frac{q}{d}) = \prod_{distinctt \in q} P(\frac{t}{M_d})^{tf_{t,q}}$$

We run into a problem with zeros where we if any term in the query is not present in the document, $P(\frac{q}{d})$ becomes 0. We smooth the above probability by introducing generator for the collection based on the intuition that a non-occurring term is possible but no more likely than would be expected by chance in the collection. The new collection frequency is given by:

$$\hat{P}(t|M_c) = \frac{cf_t}{\sum_t cf_t}$$

where cf_t is collection of frequency of a term t. The mixure model is given as:

$$P(q|d) \propto \prod_{1 \leq k \leq |q|} (\lambda P(t_k|M_d) + (1-\lambda)P(t_k|M_c))$$

A high value of λ is "conjunctive-like" and tends to retrieve search documents containing all query terms. A low value is "disjunctive-like" which is suitable for smaller queries.

2.3.4 Okapi BM25 model

The BIM was originally designed for short catalog records of fairly consistent length, and it works reasonably in these contexts. For modern full-text search collections, a model should pay attention to term frequency and document length. BM25 or Okapi is sensitive to these quantities and has been a widely used model for many decades now. Under the assumption that relevant documents are a very small percentage of the collection i.e. S and s are very small compared to N, $u_t \approx \frac{df_t}{N}$. Thus,

$$RSV_d = \sum_{t \in q} log(\frac{N}{df_t})$$

This formula is improved by factoring in term frequency $tf_{t,d}$ and length of document L_d,

$$RSV_d = \sum_{t \in q} log(\frac{N}{df_t}) \cdot \frac{(k_1+1)tf_{t,d}}{k_1((1-b)+b(\frac{L_d}{L_{ave}}))+tf_{t,d}}$$

where L_{ave} is average length of document, $k_1 \geq 0$ is tuning parameter controlling document term frequency scaling and $0 \leq b \leq 1$ is tuning parameter controlling the scaling

by document length.
If the query is long, we might also use similar weighting for query terms,

$$RSV_d = \sum_{t \in q} log(\frac{N}{df_t}) \cdot \frac{(k_1+1)tf_{t,d}}{k_1((1-b)+b(\frac{L_d}{L_{ave}}))+tf_{t,d}} \cdot \frac{(k_3+1)tf_{t,q}}{k_3+tf_{t,q}}$$

where $tf_{t,q}$ is term frequency for query and k_3 is tuning parameter controlling term frequency scaling of the query. The above tuning parameters should ideally be set to optimize performance on a development test collection. In the absence of such optimisation, experiments have shown reasonable values are to set k_1 and k_3 to a value between 1.2 and 2 and b = 0.75

2.4 Knowledge Graphs

A knowledge graph is a network of real-world entities that are connect by some relation. There are 2 main components i.e nodes and edges. Each entity is represented by a node and the relations between any two entities are represented by an edge between them. Knowledge Graphs are constructed using Natural Language Processing by identifying

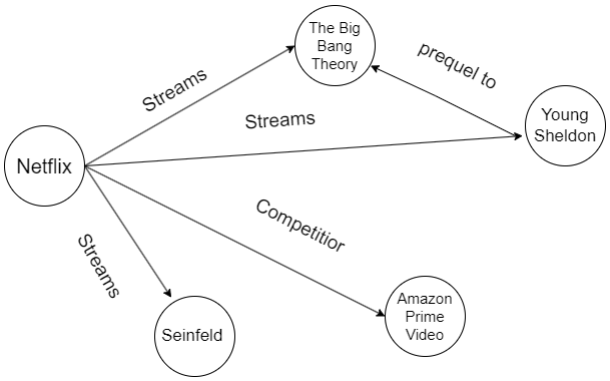


Figure 7: Example of Knowledge Graph

correct entities and relations between them. Knowledge graphs are used to improve information retrieval systems. It is because of knowledge graphs that Google knows the difference between Apple the company and apple the fruit [IBMed]

2.4.1 Entity Based Retrieval Models

In this model, rather than creating vectors out of terms, we use entities as the basis of the vector space.
Example:

- D1: Young Sheldon is prequel to The Big Bang Theory
- D2: Netflix streams both Young Sheldon and The Big Bang Theory

The above two documents can be represented as the following vectors: The entities are

Document D1	Netflix	Young Sheldon	The Big Bang Theory
	0	1	1

Figure 8: Document D1

Document D2	Netflix	Young Sheldon	The Big Bang Theory
	1	1	1

Figure 9: Document D2

ranked using the below schemes:

$$f_{COOR}(q,d) = \sum_{e:\vec{E}_q(e)>0} 1(\vec{E}_d(e) > 0)$$

$$f_{EF} = \sum_{e:\vec{E}_q(e)>0} \vec{E}_d(e) \cdot log(1 + \vec{E}_d(e))$$

2.5 Learning to Rank (LTR)

LTR methods learn how to combine predefined features by means of discriminative learning. All candidate documents of a query are represented as a vector of features $\Phi(q,d)$ that model relevance of the document d with the query q.Typical features include frequencies of query terms, outputs of BM25 or PageRank model, etc. Discriminative Training is defined by input space, output space, hypothesis space and loss function.

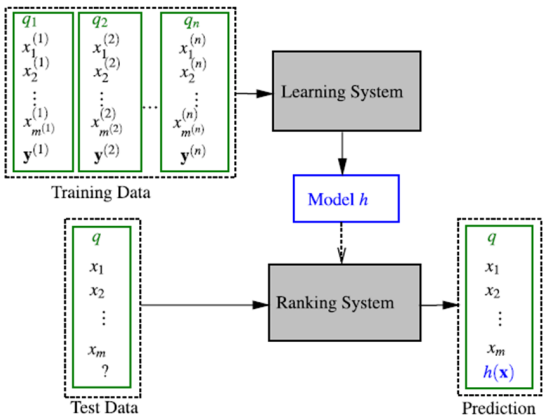


Figure 10: Learning To Rank Architecture

2.5.1 Pointwise Algorithm

The input space contains a feature vector of every document. Output space contains relevance degree of each document. For relevance degree l_j, ground truth label for document x_j is l_j For preference function $l_{u,v}()$, count number of times $l_{u,v}()$ for u is more than

other documents and for total order, position in the total order. The hypothesis space is all functions that take feature vector and return relevance degree. The loss function typically a regression, classification or ordinal function measures accurate prediction of ground truth label for each document. [CZ06] reduce the ranking problem to a regression problem where $x = \{x_j\}_{j=1}^m$ is set of documents associated with query q and and ground truth labels $y = \{y_j\}_{j=1}^m$ of these documents in terms of multiple ordered categories. The loss function is defined as following square loss:

$$L(f : x_j, y_j) = (y_j - f(x_j))^2$$

2.5.2 Pairwise Algorithm

The pairwise approach is more concerned with the relative order of two papers than it is with precisely forecasting the relevance degree of each document. Input space contains pairs of documents, both represented by feature vectors.Output space contains pairwise preferences +1, -1 If judgement is given as a relevance degree l_j , then pairwise preference for (x_u, x_v) can be defined as $y_{u,v} = 2.Il_u$ ¿ l_v – 1 Directly as pairwise preferences, set $y_{u,v} = l_{u,v}$ For total orders, it is $2 \cdot I_{\pi_{l(u)} < \pi_{l(v)}} - 1$ Hypothesis space h take a pair of documents as input and returns relative order between them. The loss function is a pairwise classification loss function to determine which document in a pair is preferred. Given n queries and their associated pairs $(x_u^(i), x_v^(v))$ where $f(x, w) = w_T x$ is the linear scoring function used, the loss function is defined as:

$$Loss = max(0, 1 - y \cdot f(x))$$

where $y_{u,v} = 1$ if relevance of $u \leq$ relevance of v

2.5.3 Listwise Algorithm

Input space is the set of documents associated with query q. Output space contains ranked lists If judgement is given as relevance degree, then all permutations satisfying relevance degree, if judgement is given as preference function, then all permutations satisfying preference function, if the judgement is given as total order, straightforward. A nuanced loss function is defined with respect to all documents for a query and cannot be fully decomposed into simple summation over individual documents or document pairs emphasizes concept of a ranked list i.e. ranking positions are visible to loss function.

3 Methodology and Results

3.1 PyLucene

This assignment uses PyLucene, a python extension to Java Lucene™ which allows us to index documents and calculate term and document frequency values effectively. PyLucene is a wrapper around Java Lucene and allows us to run Java Virtual Machine from Python. PyLucene is built using JCC, a C++ compiler which allows us to call any Java class from python using Java's Native Invocation Interface (JNI).

3.2 Pre-Processing Data

We also tokenized the sentences to create words/terms. This will help in furthur prerpocessing and also to create indexes in PyLucene. As the Dataset contains many terms, it was necessary to reduce the number of terms in order to get better retrieval time. We used Natural Language Toolkit [BKL09] to remove stop-words, i.e words that do not hold much semantic relevance like "of", "for", "this", etc.
We also reduced each word to lower case to reduce the number of unique terms as some words might occur with upper case letters in the corpus.

3.3 Experiment 1: Indexing Datasets

We index the dataset provided in `doc_dump.txt` using PyLucene. We first set the field attributes for filename and filedata as follows: The `setIndexOptions()` method tells

```
field_filename = FieldType()
field_filename.setStored(True)
field_filename.setTokenized(False)
field_filename.setIndexOptions(IndexOptions.DOCS)

field_filedata = FieldType()
field_filedata.setStored(True)
field_filedata.setTokenized(True)
field_filedata.setStoreTermVectors(True)
field_filedata.setStoreTermVectorPositions(True)
field_filedata.setIndexOptions(IndexOptions.DOCS_AND_FREQS_AND_POSITIONS_AND_OFFSETS)
```

Figure 11: Indexing Options in PyLucene

PyLucene what all indexes to create for an entry of that FieldType. Now in the file, we keep adding rows with ID, URL, TITLE and BODY. The body field contains the title and abstract of the document and is indexed for term and document frequency. Now

```
with open(input_file, 'r') as file:
    for line in file:
        line = line.strip()
        if not line:
            continue

        id, url, title, abstract = line.split('\t')
        doc = Document()
        # docIDs.append(id)
        doc.add(Field("id", id, field_filename))
        doc.add(Field("url", url, field_filename))
        doc.add(Field("title", title, field_filename))
        doc.add(Field("body", preProcess(title+" "+abstract), field_filedata))
        writer.addDocument(doc)

writer.commit()
writer.close()
```

Figure 12: Indexing Using PyLucene

each document is indexed and index can be accessed using PyLucene's inbuilt functions.

3.4 Experiment 2: Vector Based Models

In this experiment, we begin by creating a vocabulary from the corpus. A vocabulary is set of unique words from the corpus. This will be used to make the vectors for each

document. We create vectors by first reading document using PyLucene. From each

```
def create_vocab(index_reader):
    vocabulary = set()
    for doc_id in range(index_reader.numDocs()):
        doc = index_reader.document(doc_id)
        body_field = doc.get("body")
        docIDs.append(doc.get("id"))
        for terms in preProcess(body_field).split():
            vocabulary.add(terms)
    return list(vocabulary)
```

Figure 13: Creating Vocabulary

document, we tokenize the "body" field. Now from these tokens we create vector as:

- if $t \in body$ then $v_t += 1$
- else $v_t = 0$

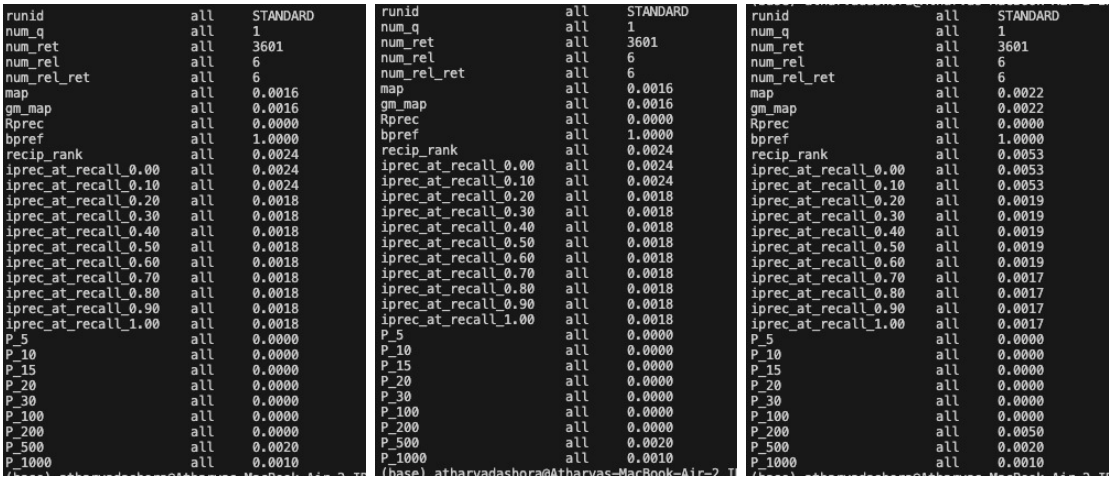
```
def vectorize_document(doc_id):
    doc = index_reader.document(doc_id)
    body_field = doc.get("body")
    bodyTerms = word_tokenize(preProcess(body_field))
    vector = defaultdict(lambda: 0)
    for terms in bodyTerms:
        vector[terms]+=1
    for terms in [item for item in vocabulary if item not in bodyTerms]:
        vector[terms] = 0
    return vector
```

Figure 14: Creating Vector from Documents

We do the same processing for queries as well. Then depending on the strategy i.e "nnn", "ntn" and "ntc" we calculate the scores for queries and documents.

3.4.1 Results

The below image shows the TREC evaluation for NNN NTN and NTC models. It shows that NTC has a higher Mean Average Precision (MAP) than NNN and NTN which is attribited to the cosine scoring function. NTC also has a higher reciprocal rank (`recip_rank`) which indicates a better performing retrieval system.



runid	all	STANDARD
num_q	all	1
num_ret	all	3601
num_rel	all	6
num_rel_ret	all	6
map	all	0.0016
gm_map	all	0.0016
Rprec	all	0.0000
bpref	all	1.0000
recip_rank	all	0.0024
iprec_at_recall_0.00	all	0.0024
iprec_at_recall_0.10	all	0.0024
iprec_at_recall_0.20	all	0.0018
iprec_at_recall_0.30	all	0.0018
iprec_at_recall_0.40	all	0.0018
iprec_at_recall_0.50	all	0.0018
iprec_at_recall_0.60	all	0.0018
iprec_at_recall_0.70	all	0.0018
iprec_at_recall_0.80	all	0.0018
iprec_at_recall_0.90	all	0.0018
iprec_at_recall_1.00	all	0.0018
P_5	all	0.0000
P_10	all	0.0000
P_15	all	0.0000
P_20	all	0.0000
P_30	all	0.0000
P_100	all	0.0000
P_200	all	0.0000
P_500	all	0.0020
P_1000	all	0.0010

runid	all	STANDARD
num_q	all	1
num_ret	all	3601
num_rel	all	6
num_rel_ret	all	6
map	all	0.0016
gm_map	all	0.0016
Rprec	all	0.0000
bpref	all	1.0000
recip_rank	all	0.0024
iprec_at_recall_0.00	all	0.0024
iprec_at_recall_0.10	all	0.0024
iprec_at_recall_0.20	all	0.0018
iprec_at_recall_0.30	all	0.0018
iprec_at_recall_0.40	all	0.0018
iprec_at_recall_0.50	all	0.0018
iprec_at_recall_0.60	all	0.0018
iprec_at_recall_0.70	all	0.0018
iprec_at_recall_0.80	all	0.0018
iprec_at_recall_0.90	all	0.0018
iprec_at_recall_1.00	all	0.0018
P_5	all	0.0000
P_10	all	0.0000
P_15	all	0.0000
P_20	all	0.0000
P_30	all	0.0000
P_100	all	0.0000
P_200	all	0.0000
P_500	all	0.0020
P_1000	all	0.0010

runid	all	STANDARD
num_q	all	1
num_ret	all	3601
num_rel	all	6
num_rel_ret	all	6
map	all	0.0022
gm_map	all	0.0022
Rprec	all	0.0000
bpref	all	1.0000
recip_rank	all	0.0053
iprec_at_recall_0.00	all	0.0053
iprec_at_recall_0.10	all	0.0053
iprec_at_recall_0.20	all	0.0019
iprec_at_recall_0.30	all	0.0019
iprec_at_recall_0.40	all	0.0019
iprec_at_recall_0.50	all	0.0019
iprec_at_recall_0.60	all	0.0019
iprec_at_recall_0.70	all	0.0017
iprec_at_recall_0.80	all	0.0017
iprec_at_recall_0.90	all	0.0017
iprec_at_recall_1.00	all	0.0017
P_5	all	0.0000
P_10	all	0.0000
P_15	all	0.0000
P_20	all	0.0000
P_30	all	0.0000
P_100	all	0.0000
P_200	all	0.0050
P_500	all	0.0020
P_1000	all	0.0010

(a) NNN TREC Evaluation (b) NTN TREC Evaluation (c) NTC TREC Evaluation

Figure 15: Vector Space Model Evaluation

3.5 Experiment 3: Rocchio Algorithm and Psuedo-Relevance Feedback

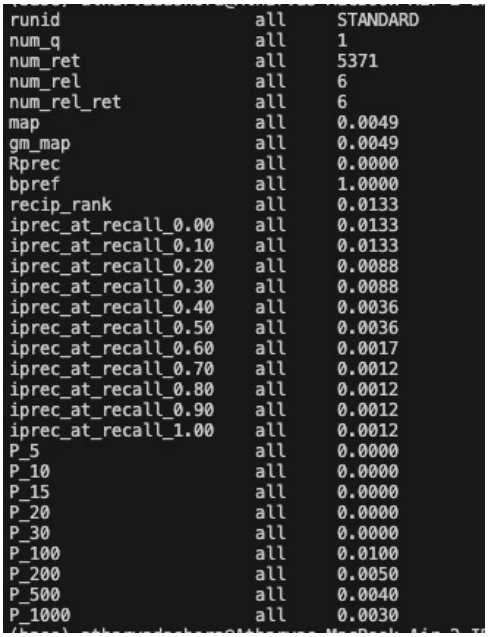
From the given set of relevant, non-relevant queries, we find the centroid of each one and use the Rocchio Algorithm mentioned here 2.2. The new query obtained is the expanded query. Since we are using the top k documents as relevant documents, this is an example of Psuedo Relevance Feedback.

```
def rocchio(qid):
    alpha = 0.75
    beta = 0.5
    gamma = 0.5
    print(qid)
    num_rel = len(rel_list[qid])
    num_non_rel = num_docs - num_rel
    print(num_rel)
    rel_vec = defaultdict(lambda:0)
    for doc in rel_list[qid]:
        # print(get_id(doc))
        # print(type(doc))
        doc_vec = docVecAll[doc]
        for term in doc_vec.keys():
            # qdid = qid+" "+doc
            # print(queryrel[qid+" "+doc])
            rel_vec[term]+=(int(queryrel[qid+" "+doc])*doc_vec[term]/num_rel)
    non_rel_vec = defaultdict(lambda:0)
    for doc in [item for item in docIDs if item not in rel_list[qid]]:
        doc_vec = docVecAll[doc]
        print(doc)
        for term in doc_vec.keys():
            non_rel_vec[term]+=(doc_vec[term]/num_non_rel)

    query_vec = QueryVectorGenerator(queries[qid])
    # print(query_vec)
    for i in range(3):
        for term in query_vec.keys():
            query_vec[term]*=alpha
        for term in query_vec.keys():
            query_vec[term]+=(beta*rel_vec[term]-gamma*non_rel_vec[term])
    return query_vec
```

Figure 16: Rocchio Algorithm

3.5.1 Results

The TREC Score for Rocchio Algorithm is given in the below image. Rocchio shows better MAP, Reverse Retrieval and Interpolation Precision at Recall K, values as compared to basic vector space models. This is attributed to the pseudo relevance feedback which sets



```
runid                   all     STANDARD
num_q                   all     1
num_ret                 all     5371
num_rel                 all     6
num_rel_ret             all     6
map                     all     0.0049
gm_map                  all     0.0049
Rprec                   all     0.0000
bpref                   all     1.0000
recip_rank              all     0.0133
iprec_at_recall_0.00    all     0.0133
iprec_at_recall_0.10    all     0.0133
iprec_at_recall_0.20    all     0.0088
iprec_at_recall_0.30    all     0.0088
iprec_at_recall_0.40    all     0.0036
iprec_at_recall_0.50    all     0.0036
iprec_at_recall_0.60    all     0.0017
iprec_at_recall_0.70    all     0.0012
iprec_at_recall_0.80    all     0.0012
iprec_at_recall_0.90    all     0.0012
iprec_at_recall_1.00    all     0.0012
P_5                     all     0.0000
P_10                    all     0.0000
P_15                    all     0.0000
P_20                    all     0.0000
P_30                    all     0.0000
P_100                   all     0.0100
P_200                   all     0.0050
P_500                   all     0.0040
P_1000                  all     0.0030
```

Figure 17: Rocchio TREC Evaluation

up a type of ground truth value for the retrieval system to work with and improve its performance.

3.6 Experiment 4: Probabilistic Retrieval

For this experiment, we implemented the formulae for probabilistic retrieval and BM25 from 2.3 and ranked the documents in descending order of probability. The BM25 model

```
def find_prob(qid, doc):
    f = 0.5
    prob = 1
    doc_vec = docVecAll[doc]
    doc_size = 0
    for val in doc_vec.values():
        doc_size+=val
    # print(doc_size)
    # print(doc)
    for terms in (preProcess(queries[qid]).split()):
        # print(terms)
        total_term_freq = index_reader.totalTermFreq(Term('body', terms))
        # df = index_reader.docFreq(Term('body', terms))
        # print(df)
        # print(total_term_freq)
        # print(doc_vec[terms])
        prob*=(f*(doc_vec[terms]/doc_size)+(1-f)*(total_term_freq/len(vocabulary)))
    # print(prob)
    return prob

def language_model_rank(qid):
    rank_dict = {}
    for doc in docIDs:
        rank_dict[qid+" "+doc] = find_prob(qid, doc)
    rank_dict = {k: v for k, v in sorted(rank_dict.items(), key=lambda item: (-1)*item[1])}
    return rank_dict
```

Figure 18: Language Model Implementation

gives slightly worse results than normal probabilistic model as BM25 also normalises the term frequency for the query which improves performance for longer query while cutting back for shorter queries.

```
def get_av_doclen():
    s = 0
    for doc in docIDs:
        s += sum(list(docVecAll[doc].values()))
    s/=num_docs
    return s


def find_rsv(qid, doc):
    k1 = 1.5
    k3 = 1.5
    b = 0.75
    rsv = 0
    ld = 0
    for val in docVecAll[doc].values():
        ld+=val

    lav = get_av_doclen()
    query_vec = QueryVectorGenerator(qid)
    for terms in (word_tokenize(preProcess(queries[qid]))):
        tf_q = query_vec[terms]
        tf_d = docVecAll[doc][terms]
        df = index_reader.docFreq(Term('body', terms))
        rsv+=(math.log(df/num_docs)*((k1+1)*tf_d/(k1*(1-b+b*(ld/lav))+tf_d))*(k3+1)*tf_q/(k3+tf_q))

    return rsv
```

Figure 19: BM25 Implementation

3.6.1 Results

```
runid                   all     STANDARD
num_q                   all     1
num_ret                 all     3187
num_rel                 all     6
num_rel_ret             all     2
map                     all     0.0558
gm_map                  all     0.0558
Rprec                   all     0.1667
bpref                   all     0.3333
recip_rank              all     0.3333
iprec_at_recall_0.00    all     0.3333
iprec_at_recall_0.10    all     0.3333
iprec_at_recall_0.20    all     0.0014
iprec_at_recall_0.30    all     0.0014
iprec_at_recall_0.40    all     0.0000
iprec_at_recall_0.50    all     0.0000
iprec_at_recall_0.60    all     0.0000
iprec_at_recall_0.70    all     0.0000
iprec_at_recall_0.80    all     0.0000
iprec_at_recall_0.90    all     0.0000
iprec_at_recall_1.00    all     0.0000
P_5                     all     0.2000
P_10                    all     0.1000
P_15                    all     0.0667
P_20                    all     0.0500
P_30                    all     0.0333
P_100                   all     0.0100
P_200                   all     0.0050
P_500                   all     0.0020
P_1000                  all     0.0010
```

Figure 20: Probabilistic TREC Evaluation

Probabilistic Model shows significant improvement over the previous models even while following the Naive Bayes assumptions. Metrics like MAP, RPrecision, Reverse Retrieval and Interpolation Precision at Recall K all show more than a 2-fold improvement for probabilistic model. This empirically shows the strength of statistical models over basic algebraic models like Vector Space models.

```
runid                   all     STANDARD
num_q                   all     1
num_ret                 all     3601
num_rel                 all     6
num_rel_ret             all     6
map                     all     0.0428
gm_map                  all     0.0428
Rprec                   all     0.0000
bpref                   all     1.0000
recip_rank              all     0.1250
iprec_at_recall_0.00    all     0.1250
iprec_at_recall_0.10    all     0.1250
iprec_at_recall_0.20    all     0.0714
iprec_at_recall_0.30    all     0.0714
iprec_at_recall_0.40    all     0.0283
iprec_at_recall_0.50    all     0.0283
iprec_at_recall_0.60    all     0.0189
iprec_at_recall_0.70    all     0.0107
iprec_at_recall_0.80    all     0.0107
iprec_at_recall_0.90    all     0.0024
iprec_at_recall_1.00    all     0.0024
P_5                     all     0.0000
P_10                    all     0.1000
P_15                    all     0.0667
P_20                    all     0.0500
P_30                    all     0.0667
P_100                   all     0.0200
P_200                   all     0.0150
P_500                   all     0.0100
P_1000                  all     0.0050
```

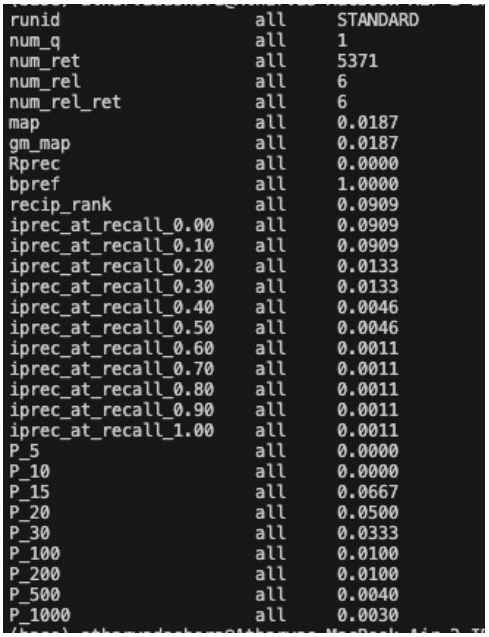
Figure 21: BM25 TREC Evaluation

3.7 Experiment 5: Entity Based Retrieval Models

For this experiment, we use the GENA knowledge Graph to create entities. We then use a Bag of Entities retrieval model 2.4.1 to retrieve documents for queries given for the NFCorpus. We made an eneity vector for each document by assuming that the entities in the knowledge graph are final and no other new entities exist in the corpus. The main downside to this experiment was the lack of common entities between NFCorpus and GENA knowledge graph.

```
def entity_based_retrieval():
    f_corr = {}
    f_ef = {}
    for qid in list(queries.keys()):
        query = queries[qid]
        print(1)
        f_corr_list = {}
        f_ef_list = {}
        for doc_id in docIDs:
            f_c = 0
            f_e = 0
            doc = index_reader.document(get_id(doc_id))
            body_field = doc.get("body")
            for entity in entity_set:
                if str(entity) in query and str(entity) in body_field:
                    f_c+=1
                    f_e+=(query.count(entity)*math.log(1+body_field.count(entity)))
            f_corr_list[doc_id] = f_c
            f_ef_list[doc_id] = f_e
            print(f_c, f_e)
        f_corr[qid] = f_corr_list
        f_ef[qid] = f_ef_list
    return f_corr, f_ef
```

Figure 22: Bag of Entities Model



```
runid                   all     STANDARD
num_q                   all     1
num_ret                 all     5371
num_rel                 all     6
num_rel_ret             all     6
map                     all     0.0187
gm_map                  all     0.0187
Rprec                   all     0.0000
bpref                   all     1.0000
recip_rank              all     0.0909
iprec_at_recall_0.00    all     0.0909
iprec_at_recall_0.10    all     0.0909
iprec_at_recall_0.20    all     0.0133
iprec_at_recall_0.30    all     0.0133
iprec_at_recall_0.40    all     0.0046
iprec_at_recall_0.50    all     0.0046
iprec_at_recall_0.60    all     0.0011
iprec_at_recall_0.70    all     0.0011
iprec_at_recall_0.80    all     0.0011
iprec_at_recall_0.90    all     0.0011
iprec_at_recall_1.00    all     0.0011
P_5                     all     0.0000
P_10                    all     0.0000
P_15                    all     0.0667
P_20                    all     0.0500
P_30                    all     0.0333
P_100                   all     0.0100
P_200                   all     0.0100
P_500                   all     0.0040
P_1000                  all     0.0030
```

Figure 23: Entity Based Models TREC evaluation

3.7.1 Results

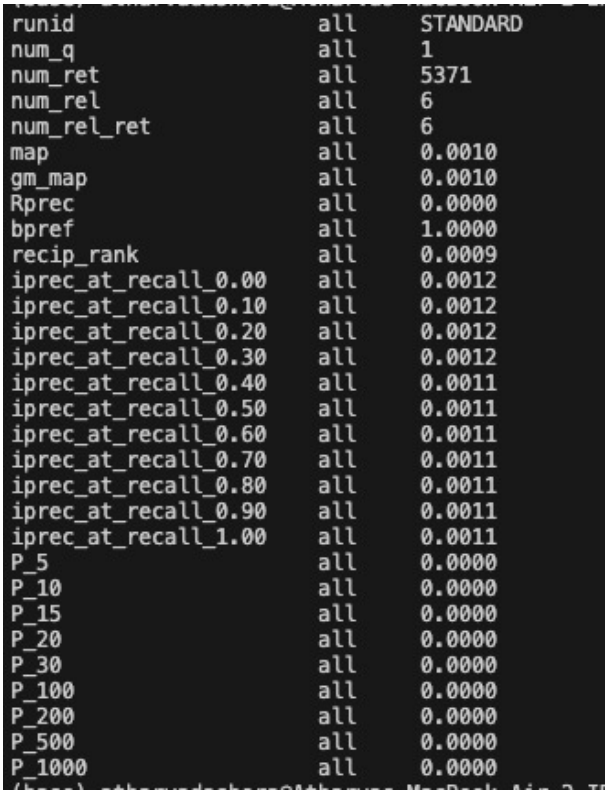
The entity based model performs worse than other models as the documents do not contain as many instances of then entities from the Knowledge Graph thus making the vector very sparse. This model would work better if the documents and knowledge graph would have had more common entities.

3.8 Experiment 6: Query Expansion using Knowledge Graph

For this experiment, we need to "expand" queries based on Knowledge Graphs and relations between entities. For this task we took each entity in a query and took all entities it was related to in the graph and added them to the query to retrieve documents with entities that are related to the original query entities.

```python
def expand_query(query, triples):

    entities = []
    for entity in entity_set:
        if str(entity) in query:
            entities.append(entity)

    expanded_query = set(entities)

    for entity in entities:
        if entity in triples:
            related_entities = [triple[1] for triple in triples[entity]]
            expanded_query.update(related_entities)

    return expanded_query
```

Figure 24: Query Expansion Using Knowledge Graphs



runid	all	STANDARD
num_q	all	1
num_ret	all	5371
num_rel	all	6
num_rel_ret	all	6
map	all	0.0010
gm_map	all	0.0010
Rprec	all	0.0000
bpref	all	1.0000
recip_rank	all	0.0009
iprec_at_recall_0.00	all	0.0012
iprec_at_recall_0.10	all	0.0012
iprec_at_recall_0.20	all	0.0012
iprec_at_recall_0.30	all	0.0012
iprec_at_recall_0.40	all	0.0011
iprec_at_recall_0.50	all	0.0011
iprec_at_recall_0.60	all	0.0011
iprec_at_recall_0.70	all	0.0011
iprec_at_recall_0.80	all	0.0011
iprec_at_recall_0.90	all	0.0011
iprec_at_recall_1.00	all	0.0011
P_5	all	0.0000
P_10	all	0.0000
P_15	all	0.0000
P_20	all	0.0000
P_30	all	0.0000
P_100	all	0.0000
P_200	all	0.0000
P_500	all	0.0000
P_1000	all	0.0000

Figure 25: Expanded Query TREC evaluation

3.8.1 Results

Since the query gets filled with many more entities, but the document still might not contain all those entities, the scores for each document becomes even lesser.

3.9 Experiment 7: Learning to Rank

We use TensorFlow's ranking models to implement LTR models. Tensorflow by itself generates embeddings i.e vectors from the given input. So we do not need to provide our own document vector. The document should be sufficient for Tensorflow to create vectors.
We randomly sample k documents in m lists for each query. These lists will be used to train the ranking algorithm. Then by changing the way we calculate loss, we can use Listwise, Pointwise and Pairwise algorithms.

Now we use Tensorflow's built in ranking functions to compute ranks, and train the model. The results are evaluated using NDGC (Normalized Discounted Cumulative Gain) metric. The NDGC metric measures how close is the predicted rank to ideal ranking. NDGC values lie between 0 and 1 where a value close to 1 means the ranking is close to ideal ranking.

3.9.1 Results

The NDGC scores for PointWise Algorithm: 90.05%
The NDGC scores for ListWise Algorithm: 95.74%
The NDGC scores for Pairwise Algorithm: 95.7%

```python
def _sample_list(
    feature_lists: Dict[Text, List[tf.Tensor]],
    num_examples_per_list: int,
    random_state: Optional[np.random.RandomState] = None,
) -> Tuple[tf.Tensor, tf.Tensor]:
  if random_state is None:
    random_state = np.random.RandomState()

  sampled_indices = random_state.choice(
      range(len(feature_lists["body"])),
      size=num_examples_per_list,
      replace=True,
  )

  sampled_movie_titles = [
      feature_lists["body"][idx] for idx in sampled_indices
  ]
  sampled_ratings = [
      feature_lists["relevance"][idx]
      for idx in sampled_indices
  ]

  return (
      tf.stack(sampled_movie_titles, 0),
      tf.stack(sampled_ratings, 0),
  )
```

Figure 26: Sampling Lists

The high NDGC scores can be attributed to Tensorflow creating one-hot encoded embeddings and then weighing each term of a document according to its relevance. It can also be attributed to the preprocessing we did to the queries and documents to remove unnecessary words that might cause shift in the focus of the query.

3.10 Experiment 8: Improving Learning to Rank models

To improve LTR model we changed the Tensorflow optimizer. In Experiment 7 we used Stochastic Gradient Descent optimizer. In this experiment we changed optimizer to Adagrad. Adagrad is better than SGD as there are different learning rates for each parameter based on iteration (higher for sparser parameters) and it is better for sparse data which maybe why it improves the NDGC score as embeddings for documents might turn out to be sparse.

3.10.1 Result

The NDGC scores for PointWise Algorithm: 94.94%
The NDGC scores for ListWise Algorithm: 99.53%
The NDGC scores for Pairwise Algorithm: 98.97%

References

[Rob77] Stephen Robertson. "The Probability Ranking Principle in IR". In: *Journal of Documentation* 33 (Dec. 1977), pp. 294–304. DOI: 10.1108/eb026647.

[MSB98] Mandar Mitra, Amit Singhal, and Chris Buckley. "Improving automatic query expansion". In: *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '98. Melbourne, Australia: Association for Computing Machinery, 1998, pp. 206–214. ISBN: 1581130155. DOI: 10.1145/290941.290995. URL: https://doi.org/10.1145/290941.290995.

[CZ06] David Cossock and Tong Zhang. "Subset Ranking Using Regression". In: *Learning Theory*. Ed. by Gábor Lugosi and Hans Ulrich Simon. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 605–619. ISBN: 978-3-540-35296-9.

[BKL09] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with python*. O'Reilly Media Inc., 2009.

[Edg] EdgeDelta. *What Percentage of Data Is Unstructured?* https://edgedelta.com/company/blog/what-percentage-of-data-is-unstructured.

[IBMed] IBM. *IBM Knowledge Graph*. https://www.ibm.com/topics/knowledge-graph. Accessed: April 25, 2024. Year Accessed.