A REPORT

ON

Information Retrieval - Assignment 2

BY

DIVYATEJA PASUPULETI	**2021A7PS0075H**
ADARSH DAS	**2021A7PS1511H**
KUSH AGRAWAL	**2021A7PS0142H**
SHIVAM ATUL TRIVEDI	**2021A7PS1512H**

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI,
HYDERABAD CAMPUS

April 2024

Contents

Contents i

1 Indexing 1
1.1 Introduction . . . 1

2 Vector-based Models 3
2.1 Introduction . . . 3
2.2 NDCG Scores . . . 6

3 Rocchio Feedback Algorithm using Pseudo-Relevance Feedback 7
3.1 Introduction . . . 7
3.2 NDCG Scores . . . 9

4 Probabilistic Retrieval 10
4.1 Introduction . . . 10
4.2 Language Model . . . 10
4.3 NDCG Scores . . . 11
4.4 Okapi - BM25 Model . . . 12
4.5 NDCG Scores . . . 13

5 Entity-Based Retrieval Models 14

6 Query Expansion Utilizing Knowledge Graphs 17

7 Learning to Rank Methods 18
7.1 Introduction . . . 18
7.2 Point Wise Ranking . . . 18
7.3 Pair Wise Ranking . . . 20
7.4 List Wise Ranking . . . 22
7.5 Conclusion . . . 24

8 Out of the Box Improvements 25
8.1 Optimization for Experiment 7a . . . 25
8.2 Optimizations for Experiments 2-4 . . . 26

Chapter 1

Indexing

1.1 Introduction

For the experiments that followed, we have made use of both lucene based indexer and a python based indexer based on the first Assignment. Although lucene indexes files faster and more efficiently, but python does turn out to be more light weight due to the generation of only 3 files.

We created a server-client interface between java and python using py4j library of python. Due to this connection we can retrieve values and results from java and use them in our python code. However, this has its drawbacks too as value retrieval for each term in the corpus can take up a lot of time hence requiring huge computational power and time. We can get values of term frequency, document frequency etc directly using built-in functions in lucene's api. For each line of the input text file a corresponding document is created and written onto the indexer. Each document consists of fields such as ID, Title, Abstract etc whose values are given as tab separated values in the input text file.

The python indexer is easier to use as it uses inverted indexing. Hence, everything is stored in postings. Term frequency, document frequency etc can be retrieved by just accessing the term's posting list.

```
1  from py4j.java_gateway import JavaGateway
2
3  gateway = JavaGateway()
4  luceneServer = gateway.entry_point
5
6  n = luceneServer.numDocs("Assignment_2/index")
7
8  print(luceneServer.checkFn("Assignment_2/index", "title", "study"))
9  print("Number of documents = " + str(n))
10
```

```
df = luceneServer.getDocFreq("Assignment_2/index", "title", "deep")
print("Doc Freq = " + str(df))

cf = luceneServer.getCollectionFreq("Assignment_2/index", "title", "deep")
print("Collection Freq = " + str(cf))

x = 0
for i in range(n):
    tf = luceneServer.getTermFreq("Assignment_2/index", "title", "deep", i)
    if(tf != 0):
        x += tf
        print("Doc = " + str(i) + " Term Freq = " + str(tf))
print(x)

coln_len = luceneServer.getCollectionLen("Assignment_2/index", "title")
print("Collection Length = " + str(coln_len))

avg_len = luceneServer.getAvgDocLen("Assignment_2/index", "title")
print("Average Document Length = " + str(avg_len))
```

Chapter 2

Vector-based Models

2.1 Introduction

The following code convert queries and documents into vectors using vector-based models. These models represent textual elements as vectors in a high-dimensional space. With the chosen notation (nnn, ntn, or ntc), we transform text into vectors. After conversion, similarity scores between the query and each document are computed. The documents are then ranked based on these scores, ensuring the most relevant ones are presented first.

```
def calculate_nnn_vectors(tokens, field, postings, doc_ids):
    query_terms = set(tokens)
    document_vectors = []
    for docs_id in doc_ids.keys():
        doc_vector = {"doc_id": docs_id, "vector": {}}
        document_vectors.append(doc_vector)
    for term in query_terms:
        if term in postings:
            for posting in postings[term]:
                doc_id = posting["doc_id"]
                term_freq = posting["term_freq"]

                doc_vector = next(
                    (doc for doc in document_vectors if doc["doc_id"] == doc_id)
    , None
                )
                if doc_vector is None:
                    doc_vector = {"doc_id": doc_id, "vector": {}}
                    document_vectors.append(doc_vector)

                doc_vector["vector"][term] = term_freq

    query_tf = calculate_query_tf(tokens)
```

```
    query_vector = {term: query_tf[term] for term in query_terms}
    similarities = calculate_similarity(query_vector, document_vectors)
    return similarities


def calculate_ntn_vectors(tokens, field, postings, num_documents, doc_freq, doc_ids):
    query_terms = set(tokens)
    document_vectors = []
    for docs_id in doc_ids.keys():
        doc_vector = {"doc_id": docs_id, "vector": {}}
        document_vectors.append(doc_vector)

    default_doc_freq = 1e-6
    term_idf = {
        term: (
            math.log(num_documents / doc_freq[term])
            if term in doc_freq
            else math.log(num_documents / default_doc_freq)
        )
        for term in query_terms
    }

    for term in query_terms:
        if term in postings:
            for posting in postings[term]:
                doc_id = posting["doc_id"]
                tf = posting["term_freq"]
                doc_vector = next(
                    (doc for doc in document_vectors if doc["doc_id"] == doc_id), None
                )
                if doc_vector is None:
                    doc_vector = {"doc_id": doc_id, "vector": {}}
                    document_vectors.append(doc_vector)

                doc_vector["vector"][term] = tf * term_idf[term]

    query_tf = calculate_query_tf(tokens)
    query_vector = {term: query_tf[term] * term_idf[term] * 1 for term in query_terms}

    similarities = calculate_similarity(query_vector, document_vectors)
    return similarities


def calculate_ntc_vectors(tokens, field, postings, num_documents, doc_freq, doc_ids):
```

```
    query_terms = set(tokens)
    document_vectors = []
    for docs_id in doc_ids.keys():
        doc_vector = {"doc_id": docs_id, "vector": {}}
        document_vectors.append(doc_vector)

    default_doc_freq = 1e-6
    term_idf = {
        term: (
            math.log(num_documents / doc_freq[term])
            if term in doc_freq
            else math.log(num_documents / default_doc_freq)
        )
        for term in query_terms
    }

    for term in query_terms:
        if term in postings:
            for posting in postings[term]:
                doc_id = posting["doc_id"]
                tf = posting["term_freq"]
                doc_vector = next(
                    (doc for doc in document_vectors if doc["doc_id"] == doc_id), None
                )
                if doc_vector is None:
                    doc_vector = {"doc_id": doc_id, "vector": {}}
                    document_vectors.append(doc_vector)

                doc_vector["vector"][term] = tf * term_idf[term]

    query_tf = calculate_query_tf(tokens)
    query_vector = {term: query_tf[term] * term_idf[term] * 1 for term in query_terms}

    similarities = calculate_cos_similarity(query_vector, document_vectors)
    return similarities


def calculate_cos_similarity(query_vector, document_vectors):
    query_terms = set(query_vector.keys())
    similarities = []
    for doc_entry in document_vectors:
        doc_id = doc_entry["doc_id"]
        doc_vector = doc_entry["vector"]
        dot_product = sum(
            query_vector[term] * doc_vector.get(term, 0) for term in query_terms
        )
```

```
113
114         query_norm = math.sqrt(sum(value**2 for value in query_vector.values()))
115         doc_norm = math.sqrt(sum(value**2 for value in doc_vector.values()))
116
117         if query_norm > 0 and doc_norm > 0:
118             similarity_score = dot_product / (query_norm * doc_norm)
119         else:
120             similarity_score = 0.0
121
122         similarities.append((doc_id, float(similarity_score)))
123
124     return similarities
125
126
127 def calculate_similarity(query_vector, document_vectors):
128     query_terms = query_vector.keys()
129     similarities = []
130
131     for doc_entry in document_vectors:
132         doc_vector = doc_entry["vector"]
133
134         dot_product = sum(
135             query_vector[term] * doc_vector.get(term, 0) for term in query_terms
136         )
137
138         similarities.append((doc_entry["doc_id"], float(dot_product)))
139
140     return similarities
```

2.2 NDCG Scores

These scores were calculated using the trec eval script for calculating various similarity measures. The comparison between different notations is given below:

Notation	**NDCG**
nnn	0.2955
ntc	02910
ntn	0.3143

Chapter 3

Rocchio Feedback Algorithm using Pseudo-Relevance Feedback

3.1 Introduction

In Pseudo-Relevance Feedback, the algorithm initially retrieves a set of documents relevant to the user's query and selects the top k documents from this set. These top k documents are then considered as relevant and used for relevance feedback.The Rocchio algorithm is applied to adjust the original query based on the feedback obtained from these selected relevant documents.

```
def rocchio_pseudo_feedback(
    query_vector,
    document_vectors,
    top_relevant_doc_ids,
    doc_ids,
    postings,
    alpha=1,
    beta=0.75,
    gamma=0,
):

    # relevant_docs_vectors = document_vectors
    relevant_docs_vectors = {}
    for doc_id in top_relevant_doc_ids:
        if doc_id in document_vectors:
            relevant_docs_vectors[doc_id] = document_vectors[doc_id]

    vocabulary = list(postings.keys())

    if relevant_docs_vectors:
```

```
        centroid_relevant = np.mean(list(relevant_docs_vectors.values()), axis
    =0)
        total_relevant_docs = len(relevant_docs_vectors)
    else:
        centroid_relevant = np.zeros_like(query_vector)
        total_relevant_docs = 0

    all_doc_ids = list(doc_ids.keys())
    non_relevant_doc_ids = [
        doc_id for doc_id in all_doc_ids if doc_id not in top_relevant_doc_ids
    ]
    non_relevant_docs_vectors = {}
    for doc_id in non_relevant_doc_ids:
        if doc_id in document_vectors:
            non_relevant_docs_vectors[doc_id] = document_vectors[doc_id]

    if non_relevant_docs_vectors:
        centroid_non_relevant = np.mean(
            list(non_relevant_docs_vectors.values()), axis=0
        )
        total_non_relevant_docs = len(non_relevant_docs_vectors)
    else:
        centroid_non_relevant = np.zeros_like(query_vector)
        total_non_relevant_docs = 0

    new_query_vector = (
        query_vector * alpha
        + ((beta * centroid_relevant) / total_relevant_docs)
        - ((gamma * centroid_non_relevant) / total_non_relevant_docs)
    )

    return new_query_vector


def create_document_vectors(top_doc_ids, vocabulary, postings):
    document_vectors = {}

    for doc_id in tqdm(top_doc_ids, desc="Creating Document Vectors"):
        doc_vector = np.zeros(len(vocabulary))

        for term_index, term in enumerate(vocabulary):
            for posting in postings[term]:
                if posting["doc_id"] == doc_id:
                    doc_vector[term_index] = posting["term_freq"]
                    break
        document_vectors[doc_id] = doc_vector

    return document_vectors
```

```
68
69
70 def calculate_similarity(query_vector, document_vectors):
71     similarities = []
72
73     for doc_id, doc_vector in document_vectors.items():
74         dot_product = np.dot(query_vector, doc_vector)
75         similarities.append((doc_id, float(dot_product)))
76
77     ranked_documents = sorted(similarities, key=lambda x: x[1], reverse=True)
78
79     return ranked_documents
```

3.2 NDCG Scores

These scores were calculated using the trec eval script for calculating various similarity measures. The comparison between Before and After Using Rocchio Algoritm is given below:

	NDCG
Before Using Rocchio Algoritm	0.2910
After Using Rocchio Algoritm	0.2790

Chapter 4

Probabilistic Retrieval

4.1 Introduction

In this experiment we had to implement 2 different Probabilistic Retrieval models. First one being the Language Model, second one being Okapi-BM25 model,

4.2 Language Model

This model is implemented as a mixture of 2 scores, i.e., document score and collection score over the entire collection. This model returns a probability, i.e. a very small value between 0 and 1, hence we return scores on the logarithmic scale. A mixture of the 2 scores is used as a form of smoothing, since document score can be zero very often due to a term frequency being 0 for that document.

```
import math
from py4j.java_gateway import JavaGateway
from tqdm import tqdm


def language_model(query, field, lmbda=0.5):
    gateway = JavaGateway()
    java_object = gateway.entry_point
    dup_query_terms = query.split()
    query_terms = set(list(query.split()))
    coln_len = java_object.getCollectionLen("Assignment_2/index", field) #total
    no of terms in collection
    cf_query_terms = {term: 0 for term in query_terms}
    num_docs =  java_object.numDocs("Assignment_2/index")

```

```
    # finding out collection frequency of each query term and collection length
    for term in query_terms:
        cf_query_terms[term] = java_object.getCollectionFreq("Assignment_2/index
    ", field, term)

    for i in range (num_docs):
        doc_score = 1

        for term in query_terms:
            tfd = java_object.getTermFreq("Assignment_2/index", field, term, i)
            tfq = dup_query_terms.count(term)
            doc_len = java_object.getDocumentLen("Assignment_2/index", field, i)
            if doc_len == 0 and tfd == 0:
                doc_len = 1
            dm_score = (tfd/doc_len)
            cm_score = (cf_query_terms[term] / coln_len)
            doc_score *= (lmbda * dm_score + (1-lmbda) * cm_score) ** tfq

        docid = java_object.getDocId("Assignment_2/index", i)
        scores[i] = (math.log(doc_score+1e-10), docid)


    scores.sort(reverse = True)

    return scores
```

4.3 NDCG Scores

These scores were calculated using the trec eval script for calculating various similarity measures. For title field all queries were taken from the test.titles.queries file whereas for abstract the test.vid-desc.queries file was used

Field	NDCG Score
Id	0.2564
Title	0.2827
Abstract	0.2588

4.4 Okapi - BM25 Model

This model returns relevance scores for each document corresponding to a query. The relevance scores are created using the formula :

$$\text{idf} \times \frac{\text{tf\_doc} + k_1}{\text{tf\_doc} + k_1 \times \left((1-b) + b \times \frac{\text{ld}}{\text{l\_avg}}\right)} \times \frac{\text{tf\_query} + k_3 \times \text{tf\_query}}{\text{tf\_query} + k_3}$$

where ld is the length of the doc, lavg is the average length of document in the collection and k1, k3, b are parameters taken as input for the function. Scores are usually in the form of positive integers say in the range of 8-20 for our sample queries.

```
import math
from py4j.java_gateway import JavaGateway
from tqdm import tqdm


def okapi_bm25(query, field, k1=1.5, k3=1.5, b=0.75):
    gateway = JavaGateway()
    java_object = gateway.entry_point
    dup_query_terms = query.split()
    query_terms = set(list(query.split()))
    coln_len = java_object.getCollectionLen("Assignment_2/index", field) #total
   no of terms in collection
    df_query_terms = {term: 0 for term in query_terms}
    num_docs =  java_object.numDocs("Assignment_2/index")
    scores = [0] * num_docs

    for term in query_terms:
        df = java_object.getDocFreq("Assignment_2/index", field, term)
        df_query_terms[term] = df

    l_avg = java_object.getAvgDocLen("Assignment_2/index", field)

    for i in range(num_docs):
        doc_score = 0
        ld = java_object.getDocumentLen("Assignment_2/index", field, i)

        for term in query_terms:
            idf = math.log((num_docs + 1e-8) / (df_query_terms[term] + 1e-8))
            tfd = java_object.getTermFreq("Assignment_2/index", field, term, i)
            tfq = dup_query_terms.count(term)
            num1 = tfd + k1 * tfd
            den1 = tfd + k1 * ((1 - b) + b * (ld / l_avg))
            num2 = tfq + k3 * tfq
            den2 = tfq + k3
```

```
34
35              doc_score += idf * (num1 / den1) * (num2 / den2)
36
37          docid = java_object.getDocId("Assignment_2/index", i)
38          scores[i] = (doc_score, docid)
39
40      scores.sort(reverse = True)
41      return scores
```

4.5 NDCG Scores

These scores were calculated using the trec eval script for calculating various similarity measures. For title field all queries were taken from the test.titles.queries file whereas for abstract the test.vid-desc.queries file was used

Field	NDCG Score
Id	0.2624
Title	0.2824
Abstract	0.3458

Chapter 5

Entity-Based Retrieval Models

This chapter delves into the construction of a bag-of-entities model and the execution of a query search with the initial extraction of entities from the GENA Knowledge Graph. The entities extracted include:

```
Flibanserin
Type_E1
CHEMICAL
Sentence
ID_1
C098107
Full_E1
MeSH_E1
flibanserin
Synonyms_1
Benzimidazoles
multifunctional serotonin receptor agonist
NEW00001
...
```

Following the extraction, indexes are generated for each entity, as demonstrated below:

```
'Obesity': defaultdict(int,
    {
        'MED-17': 2,
        'MED-97': 2,
        'MED-167': 2,
        'MED-168': 2,
        'MED-169': 2,
        'MED-225': 2,
        'MED-313': 2,
        'MED-532': 2,
    ...
```

```
    })
...
```

In this context, 'Obesity' represents an entity, and the 'defaultdict' structure contains document identifiers along with the frequency of occurrence of the entity within each document.

Upon the receipt of a query, entity extraction is performed. For example, for the query 'is cancer treatment viable?', the extracted entities would be ['cancer', 'treatment', 'cancer treatment']. These entities are then utilized to score documents, with the scoring mechanism dependent on the frequency of occurrence of each entity within the document, as outlined by the following formula:

$$S_d = \sum_{t \in Q} \text{freq}(t, d) \tag{5.0}$$

Where S_d denotes the score for a document, t represents an entity, and d signifies a document.

For the query 'stopping heart disease in childhood', the following documents are retrieved:

```
Document ID: MED-15, Title: The Garden of Eden--plant based diets, the genetic drive to conserve cholesterol and its implications for heart disease in the 21st century. - PubMed - NCBI, Score: 4
Document ID: MED-173, Title: Health and economic burden of the projected obesity trends in the USA and the UK. - PubMed - NCBI, Score: 4
Document ID: MED-191, Title: Effects of changes in fat, fish, and fibre intakes on death and myocardial reinfarction: diet and reinfarction trial (DART). - PubMed - NCBI, Score: 4
Document ID: MED-195, Title: Facing the facelessness of public health: what's the public got to do with it? - PubMed - NCBI, Score: 4
Document ID: MED-358, Title: Perceptions of flatulence from bean consumption among adults in 3 feeding studies, Score: 4
Document ID: MED-394, Title: Gut flora metabolism of phosphatidylcholine promotes cardiovascular disease, Score: 4
Document ID: MED-430, Title: Pathobiological determinants of atherosclerosis in youth risk scores are associated with early and advanced atherosclerosis. - PubMed - NCBI, Score: 4
Document ID: MED-444, Title: Mortality in workers employed in pig abattoirs and processing plants. - PubMed - NCBI, Score: 4
Document ID: MED-451, Title: Cancer and Noncancer Mortality Among American Seafood Workers, Score: 4
Document ID: MED-488, Title: Erectile dysfunction and risk of cardiovascular disease: meta-analysis of prospective cohort studies. - PubMed - NCBI, Score: 4
Document ID: MED-496, Title: Subclinical endothelial dysfunction and low-grade inflammation play roles in the development of erectile dysfunction in young men with low risk of ... - PubMed - NCBI, Score: 4
```

```
Document ID: MED-500, Title: Penile doppler ultrasound in patients with erectile dysfunction (ED): role of peak systolic velocity measured in the flaccid state in predicting ar... - PubMed - NCBI, Score: 4
Document ID: MED-501, Title: Heart disease risk factors predict erectile dysfunction 25 years later: the Rancho Bernardo Study. - PubMed - NCBI, Score: 4
Document ID: MED-504, Title: Erectile dysfunction prevalence, time of onset and association with risk factors in 300 consecutive patients with acute chest pain and angiographic... - PubMed - NCBI, Score: 4
Document ID: MED-576, Title: Association of Sleep Duration with Mortality from Cardiovascular Disease and Other Causes for Japanese Men and Women: the JACC Study, Score: 4
Document ID: MED-601, Title: Trans Fat Consumption and Aggression, Score: 4
```

The first document discusses the potential of plant-based diets in combating heart disease, which aligns with the query.

The effectiveness of this method is further demonstrated through NDCG score of 0.13 ± 0.06.

Chapter 6

Query Expansion Utilizing Knowledge Graphs

This chapter delves into the methodology of query expansion leveraging knowledge graphs, focusing on the GENA knowledge graph as our primary data source. The process begins with extracting entities from the query, utilizing the GENA knowledge graph's entity relations. For example, the query "stopping heart disease in childhood" would yield entities such as 'disease' and 'heart disease'. Following this initial extraction, the knowledge graph is queried to retrieve all objects related to each identified entity. The original entity in the query is then replaced with these related objects, thereby enriching the query with a broader spectrum of relevant terms associated with the original entity.

Considering the initial query:

```
stopping heart disease in childhood
```

The expanded query incorporates a wider range of related terms, as demonstrated below:

```
Heart Diseases, Rheumatic Disease, Rheumatic Heart Disease, Nonalcoholic
    Steatohepatitis, Heart Disease, Rheumatic Liver, Nonalcoholic Fatty
    Rheumatic Heart Diseases, Disease, Bouillaud Fatty Livers, Nonalcoholic
    DISEASE Nonalcoholic Fatty Liver Disease Diseases, Rheumatic Heart disease,
    Rheumatic Heart Disease, Obsession with the intake of unsaturated fatty
    acids, ...
```

This improvement by expansion of the query's scope is validated by examining the new Normalized Discounted Cumulative Gain (NDCG) score of 0.33 ± 0.01.

Chapter 7

Learning to Rank Methods

7.1 Introduction

In this experiment we had to implement 3 different Learning to Rank methods. First one being pointwise, second one being pairwise and third one being listwise

7.2 Point Wise Ranking

In point wise ranking method, we basically convert the question of whether a query and document are relevant into a regression problem. We have a function f(q, d). We train the model to predict the values for the same.

This can be done using any simple regression problem or a classification model.

```
class LTRDataset(Dataset):
    def __init__(self):
        # Create an array of datasets
        self.dataset = []
        self.outputs = []

        # Now we need to get the features for each query-document pair
        logging.info("Getting features for each query-document pair")
        for i in tqdm(range(0, len(merged_qrel))):
            query_id = merged_qrel.iloc[i]['QUERY_ID']
            doc_id = merged_qrel.iloc[i]['DOC_ID']
            relevance = merged_qrel.iloc[i]['RELEVANCE']

            # Get the features for the query-document pair
            self.dataset.append({
                "query_id": query_id,
```

```
                "doc_id": doc_id,
                "relevance": relevance,
                "vector": [1, 0, 0, 0]
            })
            # one hot encode the relevance
            output_vector = [0, 0, 0, 0]
            output_vector[relevance] = 1
            self.outputs.append(relevance)

    def __len__(self):
        return len(self.dataset)

    def __getitem__(self, index):
        return torch.tensor(self.dataset[index]["vector"], dtype=torch.float32),
     self.outputs[index]
```

The main point here is how we convert data for ranking. We have a query id and a document id and the relevance between them. Using these query id and document id, we calculate a tf-idf vector for the same. This vector is what goes as input for the neural network for the classification model. .;

```
class NeuralNet(torch.nn.Module):
    def __init__(self, n_features, output_size):
        super(NeuralNet, self).__init__()
        self.fc1 = torch.nn.Linear(n_features, 300)
        self.fc2 = torch.nn.Linear(300, 150)
        self.fc3 = torch.nn.Linear(150, output_size)
        self.relu = torch.nn.ReLU()
        self.softmax = torch.nn.Softmax(dim=-1)

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.relu(self.fc3(x))
        # logging.info(f"The final vector is: \n{x}")
        x = self.softmax(x)
        # logging.info(f"The output vector is: \n{x}")
        return x
```

Epoch	Testing Loss	Testing Accuracy	Testing NDCG
0	0.006215715738008943	0.7601684681765971	0.7717188745604225
1	0.006217702958586149	0.7601684681765971	0.7711478146878872
2	0.00621505330364697	0.7601684681765971	0.7721062063257278
3	0.006219390173974881	0.7601684681765971	0.7710093221334825
4	0.006217040524224176	0.7601684681765971	0.7717642971983417
5	0.006215715723964907	0.7601684681765971	0.7710649503599684
6	0.006219390147642313	0.7601684681765971	0.7716221743050909
7	0.006213728515676233	0.7601684681765971	0.7716739473500354
8	0.006216078133827857	0.7601684681765971	0.7717259373317254
9	0.00621505326853688	0.7601684681765971	0.7719134061226285

Here the values are taken over an average of 5 runs where the minimum of the runs stood at 0.6 and the maximum stood at 0.85 and have been averaged per epoch

7.3 Pair Wise Ranking

In pair wise ranking method, we have one query and 2 documents per datapoint. We basically use these 2 documents to predict if one document is higher than the other or not. This problem can be reduced to that of binary classification. Similar to the above model, the neural network will not change but how we parse the data changes.

Here the function is of the form f(q, d1, d2) instead of f(q, d).

```
class LTRDataset(Dataset):
    def __init__(self):
        # Create an array of datasets
        self.dataset = []
        self.outputs = []

        # Now we need to get the features for each query-document pair
        logging.info("Getting features for each query-document-document pair")
        query_document_id_map = {}
        for i in tqdm(range(0, len(merged_qrel))):
            query_id = merged_qrel.iloc[i]['QUERY_ID']
            doc_id = merged_qrel.iloc[i]['DOC_ID']
            relevance = merged_qrel.iloc[i]['RELEVANCE']

            if query_id not in query_document_id_map:
                query_document_id_map[query_id] = [{
                    "document_id": doc_id,
                    "relevance": relevance
```

```
                }]
            else:
                query_document_id_map[query_id].append({
                    "document_id": doc_id,
                    "relevance": relevance
                })

        # Now we process per query
        for query_id in tqdm(query_document_id_map):
            # we need to create dataset as query, document, docuemnt, relevance
            logging.info(f"Processing query: {query_id}, with {len(
    query_document_id_map[query_id])} documents")
            for i in range(0, len(query_document_id_map[query_id]), 2):
                if i + 1 < len(query_document_id_map[query_id]):
                    # Get the features for each query-document-document pair
                    # query = luceneServer.getQuery(query_id)
                    # document1 = luceneServer.getDocument(query_document_id_map
    [query_id][i]['document_id'])
                    # document2 = luceneServer.getDocument(query_document_id_map
    [query_id][j]['document_id'])
                    relevance1 = query_document_id_map[query_id][i]['relevance']
                    relevance2 = query_document_id_map[query_id][i + 1]['
    relevance']

                    # Get the features
                    # features = luceneServer.getFeatures(query, document1,
    document2)
                    self.dataset.append({
                        "query": query_id,
                        "document1": query_document_id_map[query_id][i]['
    document_id'],
                        "document2": query_document_id_map[query_id][i + 1]['
    document_id'],
                        "relevance1": relevance1,
                        "relevance2": relevance2,
                        "output": 1 if relevance1 > relevance2 else 0,
                        "vector": [1, 0, 0, 0]
                    })

                    # Get the output
                    if relevance1 > relevance2:
                        self.outputs.append(1)
                    else:
                        self.outputs.append(0)


    def __len__(self):
        return len(self.dataset)
```

```
60
61      def __getitem__(self, index):
62          return torch.tensor(self.dataset[index]["vector"], dtype=torch.float32),
        self.outputs[index]
```

With pairwise LTR, we don't actually need the relevance values, having whether one's precedence is higher is sufficient. Here our method of merging query, document1 and document2 vector is the adding method. Concatenation of the vectors might lead to better results but due to computational limitations, we have used addition.

Epoch	Testing Loss	Testing Accuracy	Testing NDCG
0	0.0054448605101609674	0.9822066174720304	0.2290399281516934
1	0.0054448605101609674	0.9822066174720304	0.22254458735143184
2	0.0054448605101609674	0.9822066174720304	0.22219960533455496
3	0.0054448605101609674	0.9822066174720304	0.22537731241155204
4	0.0054448605101609674	0.9822066174720304	0.21851855490295363
5	0.0054448605101609674	0.9822066174720304	0.2278552809392732
6	0.0054448605101609674	0.9822066174720304	0.22369831066485443
7	0.0054448605101609674	0.9822066174720304	0.2325604479633292
8	0.0054448605101609674	0.9822066174720304	0.22701335934141947
9	0.0054448605101609674	0.9822066174720304	0.228924138198212

7.4 List Wise Ranking

In point wise ranking method, we basically convert the question of whether a query and document are relevant into a list wise ranking problem. These compare the probability distributions of the actual and compare it with the predicted distribution

This can be done using ListMLE models using tensorflow.

```
class LTRModel(tfrs.Model):
  def __init__(self, loss):
    super().__init__()
    embedding_dimension = 32

    # Query embeddings
    self.query_embeddings = tf.keras.Sequential([
      tf.keras.layers.StringLookup(
        vocabulary=unique_query_ids),
      tf.keras.layers.Embedding(len(unique_query_ids) + 2, embedding_dimension)
    ])

```

```
    # Document embeddings
    self.document_embeddings = tf.keras.Sequential([
      tf.keras.layers.StringLookup(
        vocabulary=list(document_id_vocab)),
      tf.keras.layers.Embedding(len(document_id_vocab) + 2, embedding_dimension)
    ])

    # Compute predictions
    self.score_model = tf.keras.Sequential([
      # Learn multiple dense layers.
      tf.keras.layers.Dense(128, activation="relu"),
      tf.keras.layers.Dense(64, activation="relu"),
      tf.keras.layers.Dense(1)
    ])

    self.task = tfrs.tasks.Ranking(
      loss=loss,
      metrics=[
        tfr.keras.metrics.NDCGMetric(name="ndcg_metric"),
        tf.keras.metrics.RootMeanSquaredError()
      ]
    )

  def call(self, features):
    query_embeddings = self.query_embeddings(features["query_id"])
    document_embeddings = self.document_embeddings(features["document_id"])

    # The embeddings are concatenated
    list_length = features["document_id"].shape[1]
    user_embedding_repeated = tf.repeat(
        tf.expand_dims(query_embeddings, 1), [list_length], axis=1)
    concatenated_embeddings = tf.concat(
        [user_embedding_repeated, document_embeddings], 2)

    return self.score_model(concatenated_embeddings)

  def compute_loss(self, features, training=False):
    labels = features.pop("relevance")
    scores = self(features)
    return self.task(
        labels=labels,
        predictions=tf.squeeze(scores, axis=-1),
    )

cached_train = train.shuffle(10000).batch(8192).cache()

listwise_model = LTRModel(tfr.keras.losses.ListMLELoss())
listwise_model.compile(optimizer=tf.keras.optimizers.Adagrad(0.1))
```

```
61 listwise_model.fit(cached_train, epochs=5, verbose=True)
```

Epoch	NDCG	RMSE	Loss
1	0.9404	1.8744	15.1028
1	0.9339	1.9007	15.1021
1	0.9326	1.9073	15.1013
1	0.9326	1.9073	15.1009
2	0.9525	1.8684	15.0974
2	0.9439	1.8942	15.0965
2	0.9417	1.9007	15.0957
2	0.9417	1.9007	15.0953
3	0.9584	1.8609	15.0898
3	0.9492	1.8865	15.0895
3	0.9467	1.8931	15.0887
3	0.9467	1.8931	15.0882
4	0.9616	1.8536	15.0807
4	0.9521	1.8780	15.0802
4	0.9495	1.8837	15.0788
4	0.9495	1.8837	15.0781
5	0.9637	1.8410	15.0659
5	0.9539	1.8648	15.0652
5	0.9512	1.8706	15.0625
5	0.9512	1.8706	15.0611

7.5 Conclusion

We are of the opinion that such high NDCG values occur because of the fact that once you reduce dimensions to very small vectors, then the comparison standards become very low, leading to high

Chapter 8

Out of the Box Improvements

8.1 Optimization for Experiment 7a

In the Learning to rank models, we can also use a skipgram model which will enhance the embeddings and will have better understanding of the query and the result i.e. the paragraph or answer.

Since a skipgram model also leads to reducetion of the vector space, this will make it easier for the model to run on smaller systems.

This will lead to a better relevance ranking instead of using one hot encoding for the same. This was one approach which we thought of in order to increase the NDCG values.

```
class SkipgramModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim):
        super(SkipgramModel, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.linear = nn.Linear(embedding_dim, embedding_dim * 3)
        self.linear2 = nn.Linear(embedding_dim * 3, vocab_size)

    def forward(self, inputs__):
        embeds = self.embedding(inputs__)
        out = self.linear(embeds)
        out = self.linear2(out)
        log_probs = F.log_softmax(out, dim=1)
        return log_probs

    def prediction(self, inputs):
        embeds = self.embedding(inputs)
        return embeds
```

Upon using this model to retrieve the vectors of dimensions 128, instead of using the vocabulary directory, we achieved better results with an NDCG of 0.87.

Here we have the input vector of size 5180 which is being converted to 128

```
SkipgramModel(
  (embedding): Embedding(5180, 128)
  (linear): Linear(in_features=128, out_features=384, bias=True)
  (linear2): Linear(in_features=384, out_features=5180, bias=True)
)
```

8.2 Optimizations for Experiments 2-4

1) Query Optimization: Expand the original query by adding synonyms or other related terms from the top-ranked documents. We could also remove stop words from the query itself so that their scores are not combined.

2) Weighted Average of Scores across Fields: Assigning higher weights to terms appearing in specific fields based on their importance in relevance determination.

3) Parameter Tuning: Adjust the parameter values of the language models(lambda) or Okapi BM25 (k1, k3, b) to better fit the dataset. Currently, we are using lambda = 0.5. This can be increased to give more weight to the collection model as a smoothing mechanism since term frequencies can very often be 0. Similarly, k1/k3/b parameters of the BM25 model may be varied for different results.

4) Document Length Normalization: Change the document length normalization strategy in Okapi BM25 to handle varying document lengths.

5) Feedback Mechanisms: Incorporate relevance feedback mechanisms where user feedback on initial search results is used to update the subsequent queries.