COL764

Information Retreival And Web Search

Assignment 2

Team Members:

Ayushman Kumar Singh (2021CS51004) cs5211004@cse.iitd.ac.in

Instructor:

Srikanta Bedathur

2024/2025 – 1st Semester

Contents

1 Core Ideas of Re-Ranking

I compute uni-gram, language model probabilities for each document in the top-100 documents retrieved, and also uni-gram probabilities for background model using all documents retrieved collectively.

To handle out of vocabulary terms, some probability mass is reserved for the <UNK> token. Any token in the query or otherwise, not in the documents retrieved are treated as the <UNK> token.

Each of the individual document language models are smoothened using Dirichlet smoothing where the background language model is the language model of all the top documents retrieved collectively.

$$P(t|M_j) = \frac{f_{t,d_j} + \mu \hat{P}_c(t)}{|d_j| + \mu}$$

Relevance Based Language Modeling

We take the vocabulary V to be the all the terms in the initial set of documents retrieved for a query. We then compute uni-gram language models with Dirichlet smoothing for each document.

To compute the $P(w|R)$ where R is the relevance model, we use the following set of equations.

$$p(w|R) = P(w, q_1, q_2 \dots q_n)/P(q_1, q_2, \dots q_n)$$

Here $q_1, q_2 \dots q_n$ is the Query.

$$P(w, q_1, q_2 \dots q_n) = \sum_{M \in \mathcal{M}} P(M) P(w, q_1, q_2 \dots q_n | M)$$

Marginalizing over all the document models of the top documents retrieved (Here we assume $P(M)$ is uniform).

$$\Rightarrow P(w, q_1, q_2 \dots q_n) = \sum_{M \in \mathcal{M}} P(M) P(w|M) \prod_{i=1}^{n} P(q_i|M)$$

Using independence of terms, once the model is chosen.

$$\Rightarrow P(w|R) = \frac{\sum_{M \in \mathcal{M}} P(M) P(w|M) \prod_{i=1}^{n} P(q_i|M)}{\sum_{w \in V} \sum_{M \in \mathcal{M}} P(M) P(w|M) \prod_{i=1}^{n} P(q_i|M)}$$

Simplifying these equation we get,

$$\Rightarrow P(w|R) = \frac{\sum_{M \in \mathcal{M}} P(w|M) \prod_{i=1}^{n} P(q_i|M)}{\sum_{M \in \mathcal{M}} \prod_{i=1}^{n} P(q_i|M)}$$

We compute these probabilities to estimate the Relevance the model, and re-rank documents by computing the KL-Divergence between the Relevance model and the Document Language models using the following equation

$$D(M_d||M_R) = \sum_{w \in V} P(w|M_d) \log \frac{P(w|M_d)}{P(w|M_R)}$$

1.0.1 KL Divergence in Query Language Models

KL divergence measures the difference between two probability distributions. In the context of this task, we compute the KL divergence between the relevance model (query language model) and each document model. This helps rank the documents based on their likelihood of relevance to the query.

1. Let $P(Q)$ represent the query language model and $P(D)$ represent the document language model. The KL divergence between these two distributions is defined as:

$$D_{\mathrm{KL}}(P(Q) \parallel P(D)) = \sum_{w \in V} P(Q(w)) \log \frac{P(Q(w))}{P(D(w))}$$

   where w represents a word in the vocabulary V, $P(Q(w))$ is the probability of word w in the query model, and $P(D(w))$ is the probability of word w in the document model.

2. The goal is to minimize the KL divergence, meaning we are looking for document models that closely match the query model.

3. The result is a similarity score for each document, which is used to rank the documents. A lower KL divergence indicates a higher similarity between the query and document.

2 Task 0: Relevance Model for Retrieval

2.1 The Algorithm and Idea

2.1.1 Language Models

A language model is built for each document and is used to compute the probability of terms given the document. We define the document language model $P(w \mid D)$ using Dirichlet smoothing:

$$P(w \mid D) = \frac{\mathrm{tf}(w, D) + \mu \cdot P(w \mid \text{Background})}{|D| + \mu}$$

Where:

- $\mathrm{tf}(w, D)$ is the term frequency of word w in document D.
- $|D|$ is the total number of tokens in document D.
- $P(w \mid \text{Background})$ is the probability of word w in the background collection.
- μ is the Dirichlet smoothing parameter.

2.1.2 Combined Background Language Model

The background language model is built by combining all document language models:

$$P(w \mid \text{Background}) = \frac{\sum_{D \in \mathcal{C}} \text{tf}(w, D)}{\sum_{D \in \mathcal{C}} |D|}$$

Where:

- $\mathcal{C}$ is the set of documents.
- $\text{tf}(w, D)$ is the frequency of word w in document D.

2.1.3 Query Likelihood Model

For a given query $Q = \{w_1, w_2, \ldots, w_n\}$, the probability of observing the query given a document D is the product of term probabilities:

$$P(Q \mid D) = \prod_{i=1}^{n} P(w_i \mid D)$$

Where $P(w_i \mid D)$ is the smoothed term probability of w_i in document D.

2.1.4 Relevance Model

The relevance model computes the probability of a term being relevant to the query by combining the document language models and the query likelihood:

$$P(w \mid Q) = \frac{1}{|\mathcal{D}|} \sum_{D \in \mathcal{D}} P(w \mid D) \cdot P(Q \mid D)$$

Where:

- $\mathcal{D}$ is the set of top-ranked documents.
- $P(Q \mid D)$ is the query likelihood for document D.

2.1.5 KL Divergence

The Kullback-Leibler (KL) divergence is used to measure the difference between two probability distributions, such as the document model and the relevance model:

$$\text{KL}(P \parallel Q) = \sum_{w} P(w \mid D) \log \frac{P(w \mid D)}{P(w \mid Q)}$$

Where $P(w \mid D)$ is the probability of term w in document D, and $P(w \mid Q)$ is the probability of term w in the relevance model.

2.1.6 Ranking

Documents are ranked by minimizing the KL divergence between the document language model and the relevance model. The final ranking score for each document is given by:

$$\text{Score}(D) = 1 - \text{KL}(P(w \mid D) \parallel P(w \mid Q))$$

The documents with lower KL divergence values are considered more relevant and receive higher scores.

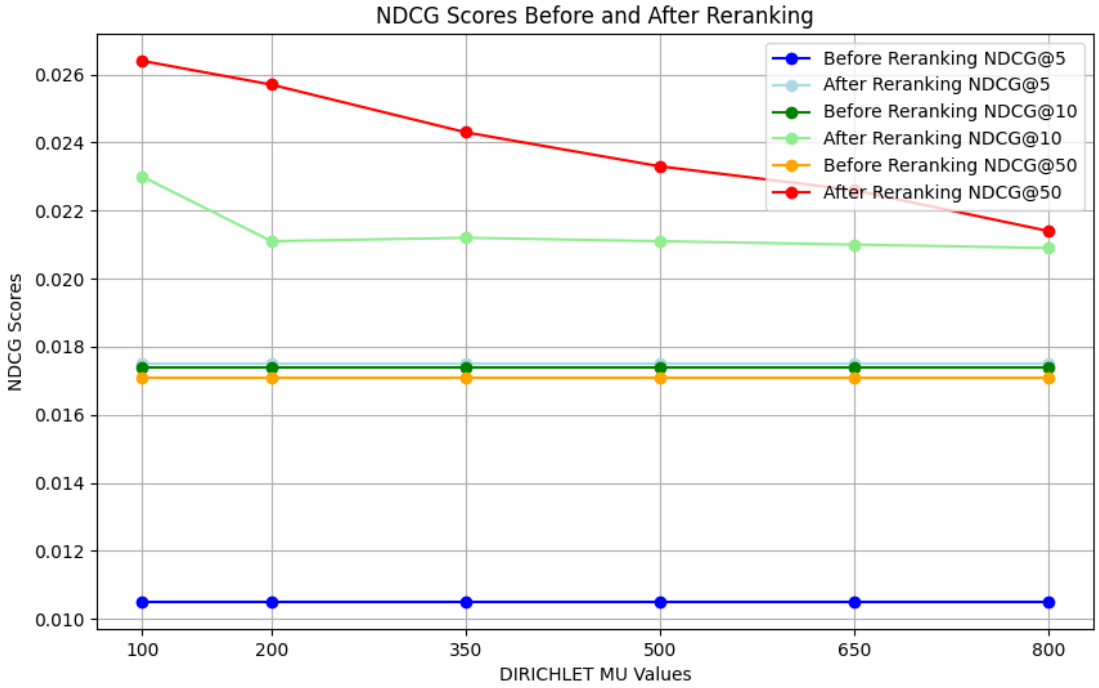
2.2 Experiments

I used the following constants mentioned in constants.py.

```
DELIMITERS = [" ", ",", ".", ":", ";", "\"", "\'", '/', '-', '%', '(', ')',
    '[', ']', '{', '}', '?', '!', '@', '#', '$', '^', '&', '*', '+', '=', '|',
    '\\', '~', '`', '\n', '\t', '\r', '\f', '\v', '\b', '\a', '\0', '\1', '\2',
    '\3', '\4', '\5', '\6', '\7', '\8', '\9']
DIRICHILET_MU = 200
TOP_K = 20
LOWERCASE = True
PUNCTUATIONS = True
DIGITS = True
STEMMING = False
STOPWORDS_ELIMINATION = True
W2V_LAMBDA = 0.2
UNK_PERCENTAGE = 0.5
```

Here we only need DELIMITERS, DIGITS, DIRICHILET_MU, LOWERCASE, PUNCTUATIONS, STEMMING, STOPWORDS_ELIMINATION, UNK_PERCENTAGE.

2.3 Results: Varying the dirichlet parameter μ



Therefore we know from here that we should set **dirichlet parameter μ to 200.**

3 Task 1: Experiments with Local Word2Vec

3.1 The Algorithm and Ideas!

3.1.1 Matrix Multiplication and Query Expansion

The primary mathematical operation involved in query expansion is based on computing cosine similarity between the query vector and document embeddings. This can be described as follows:

1. Let the document embedding matrix be represented as $\mathbf{E} \in \mathbb{R}^{V \times d}$, where V is the vocabulary size and d is the dimensionality of the embeddings. Each row in $\mathbf{E}$ corresponds to the embedding of a word in the vocabulary.

2. Let the query vector be $\mathbf{q} \in \mathbb{R}^{V}$, which is a sparse vector where each entry represents the term frequency of the corresponding word in the query.

3. **Instead of computing pairwise similarities between all vocabulary terms in the matrix, I optimize the computation by first projecting the query vector into the embedding space**. These optimizations help reduce both the time and space complexity of the query expansion process, making it more feasible to handle large document collections and vocabularies. This is done by multiplying the query vector with the

embedding matrix:

$$\mathbf{q}_{\text{embed}} = \mathbf{E}^\top \mathbf{q}$$

where $\mathbf{q}_{\text{embed}} \in \mathbb{R}^d$ is the embedding representation of the query in the same dimensional space as the document embeddings.

4. Once the query is projected into the embedding space, we compute the cosine similarity between the query embedding and the document embeddings. The cosine similarity is computed as:

$$\text{Similarity}(\mathbf{q}_{\text{embed}}, \mathbf{E}) = \frac{\mathbf{E}\mathbf{q}_{\text{embed}}}{\|\mathbf{E}\|\|\mathbf{q}_{\text{embed}}\|}$$

This optimized approach reduces the computational complexity and improves memory usage by avoiding the multiplication of two large matrices ($V \times d$). Instead, we first multiply the query vector with the embedding matrix.

5. After calculating the similarities, the top-K terms are selected by sorting the resulting similarity scores:

$$\text{Top-}K = \text{argsort}(\mathbf{E}\mathbf{q}_{\text{embed}})[:K]$$

The top-K terms are used for query expansion based on their similarity scores.

3.1.2 Word2Vec Training using Top-100 Documents

The Word2Vec model is trained using the top-100 documents retrieved for each query. The steps for training Word2Vec are as follows:

1. For each query, we combine the text of the top-100 documents into a corpus. This corpus is preprocessed by tokenizing, removing stopwords, applying stemming, and other preprocessing steps.

2. A Skip-gram Word2Vec model is trained on this corpus. The objective of the Skip-gram model is to maximize the probability of context words given a target word. The loss function is:

$$L = -\sum_{w \in C(t)} \log P(w|t)$$

where $C(t)$ is the context of the target word t, and $P(w|t)$ is the conditional probability of observing the context word w given the target word t.

3. After training, the model produces word embeddings for each term in the vocabulary. These embeddings capture the semantic relationships between words based on their co-occurrence in the document corpus.

3.1.3 KL Divergence and Relevance Model

The KL divergence is used to rank documents based on how well they match the expanded query model. The relevance model is a mixture of the original query and the expanded query:

$$P_{\text{relevance}}(w) = \lambda P_{\text{expanded}}(w) + (1 - \lambda) P_{\text{original}}(w)$$

where λ is a weighting factor, $P_{\text{expanded}}(w)$ is the probability of word w in the expanded query, and $P_{\text{original}}(w)$ is the probability of word w in the original query. This ensures that both the original query terms and the expanded terms are considered when ranking documents.

3.1.4 Calculation of $P_{\text{expanded}}(w)$ and $P_{\text{original}}(w)$

1. Original Query Term Probabilities The probability of a term w in the original query, $P_{\text{original}}(w)$, is computed as:

$$P_{\text{original}}(w) = \frac{\text{count}(w, Q_{\text{original}})}{\sum_{t \in Q_{\text{original}}} \text{count}(t, Q_{\text{original}})}$$

where:

- $\text{count}(w, Q_{\text{original}})$ is the frequency of term w in the original query.
- Q_{original} is the set of terms in the original query.

If the term w is not found in the original query vocabulary, it is replaced by an unknown token $\langle \text{UNK} \rangle$.

2. Expanded Query Term Probabilities The probability of a term w in the expanded query, $P_{\text{expanded}}(w)$, is calculated using the similarity scores obtained from the Word2Vec model:

$$P_{\text{expanded}}(w) = \frac{\text{similarity}(w)}{\sum_{t \in Q_{\text{expanded}}} \text{similarity}(t)}$$

where:

- $\text{similarity}(w)$ is the score for term w based on its similarity to the query vector.
- Q_{expanded} is the set of expansion terms chosen from the Word2Vec vocabulary.

Similar to the original query, terms not found in the expanded query are replaced with the $\langle \text{UNK} \rangle$ token.

3.1.5 KL Divergence Calculation

Once $P_{\text{relevance}}(w)$ is obtained from the mixture model combining $P_{\text{original}}(w)$ and $P_{\text{expanded}}(w)$, we compute the KL divergence between the query language model and the document language model for each document. The KL divergence for a document's language model, $P_{\text{doc}}(w)$, is given by:

$$D_{\text{KL}}(P_{\text{relevance}} || P_{\text{doc}}) = \sum_{w \in V} P_{\text{relevance}}(w) \log \left(\frac{P_{\text{relevance}}(w)}{P_{\text{doc}}(w)} \right)$$

where V is the vocabulary. In this case, the relevance model probability $P_{\text{relevance}}(w)$ serves as the query model, and the document's term probability $P_{\text{doc}}(w)$ is computed using a language model estimated for each document.

3.1.6 Why Use KL Divergence?

KL divergence is used as a measure of how much one probability distribution (the relevance model) diverges from another (the document model). A smaller divergence indicates that the document model closely matches the query's relevance model, suggesting that the document is more relevant to the query. Hence, documents are ranked based on minimizing KL divergence. This approach allows us to take into account both the original query and expanded terms in the document ranking process.

3.1.7 More on Training!

To leverage local word embeddings for query expansion in the context of information retrieval, we use a Word2Vec model trained specifically on the top 100 documents retrieved for each query. This approach allows the model to learn context-specific embeddings, which improves the quality of expansions by focusing on terms relevant to the given query.

The Word2Vec model is trained using the **Skip-gram** algorithm (`sg=1`) with hierarchical softmax (`hs=1`) and negative sampling (`negative=2`). The model is trained with the following hyperparameters:

- Vector size: 300
- Window size: 20
- Minimum word count: 1
- Epochs: 100
- Number of workers: 20

The vocabulary for training the Word2Vec model is built using tokens from the top 100 documents associated with each query. Each document is preprocessed, tokenized, and then used to construct the Word2Vec vocabulary. The following code snippets highlight this process:

```
combined_tokens = []
for doc_id in doc_ids_query:
    doc_text = f"{doc_contents[doc_id][0]} {doc_contents[doc_id][1]}"
    processed_doc = preprocess_text(doc_text, LOWERCASE, PUNCTUATIONS, DIGITS,
        STEMMING, STOPWORDS_ELIMINATION)
    tokens = SimpleTokenizer().tokenize(processed_doc)
    combined_tokens.append(tokens)
```

Once the vocabulary is built, the Word2Vec model is trained as shown below **for each query**:

```
w2v_model = Word2Vec(vector_size=300, window=20, min_count=1, sg=1, hs=1,
    negative=2, sample=0, workers=20, epochs=100)
w2v_model.build_vocab(combined_tokens)
w2v_model.train(combined_tokens, total_examples=len(combined_tokens),
    epochs=w2v_model.epochs)
```

The trained model is then used to compute the vector representations of the query and to identify the most similar terms for query expansion.

By using local embeddings from the top 100 documents, we ensure that the expanded query terms are contextually relevant to the document collection, thereby improving the precision of the retrieval process.

3.2 Experiments

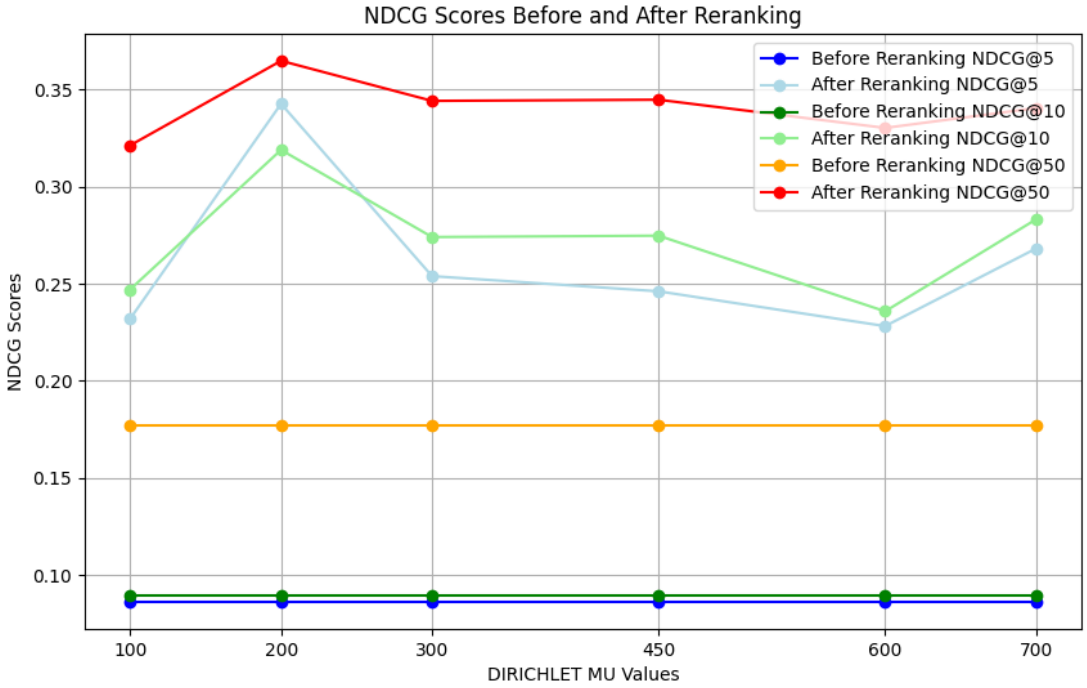
The following constants appear in **constants.py**.

```
DELIMITERS
DIRICHILET_MU
TOP_K
LOWERCASE
PUNCTUATIONS
DIGITS
STEMMING
STOPWORDS_ELIMINATION
W2V_LAMBDA
UNK_PERCENTAGE
```

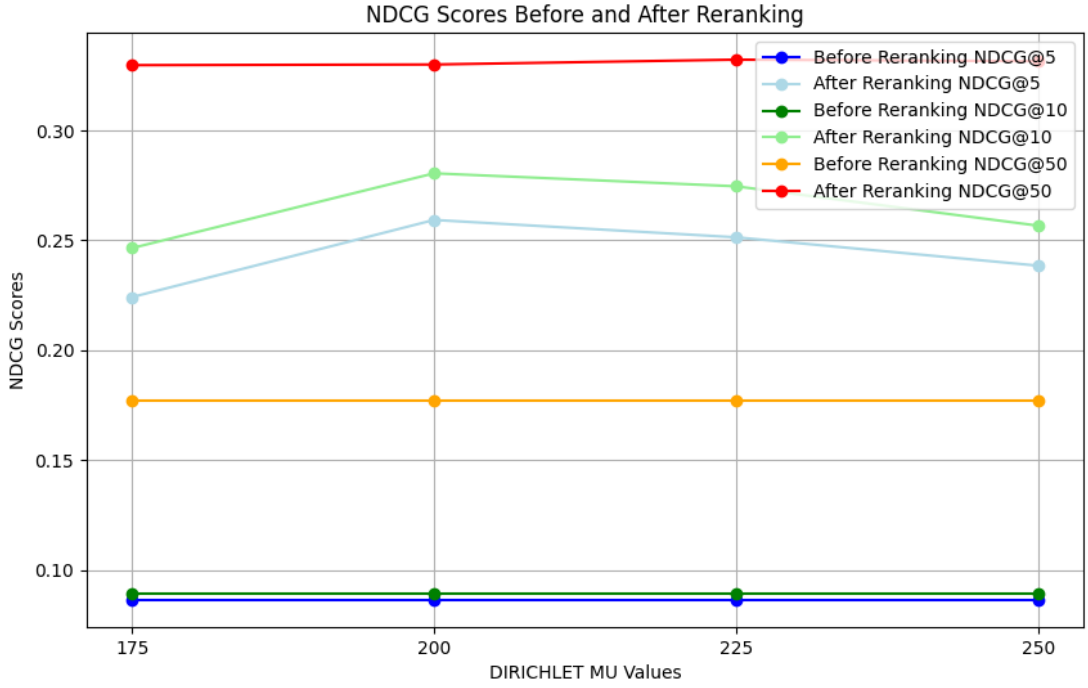
'DELIMETERS' are used for simple tokenizer to tokenize the text. I have experimented with the following text-preprocessing techniques by varying the above parameters:

- **Dirichlet parameter**: Varied it from around 150 to 1500.
- **Casing**: Option to convert all text to lowercase or leave as it is.
- **Expansion Size**: Size of the expansion set or a query. Varied it from 10 to 20.
- **Punctuations**: Option to remove these or keep them as it is.
- **Digits**: Since there is a lot of numerical data, in academic texts and are not directly relevant for answering queries, I have kept an option to replace them by common token <NUM>
- **Stopwords**: Option to remove stopwords such as i, the etc. (from NLTK)
- **Stemming**: Option to do stemming using the snowball stemmer (from NLTK).
- **w2v parameter**: Varying the parameter for creating the word to vec relevance model.

3.3 Varying DIRICHILET MU



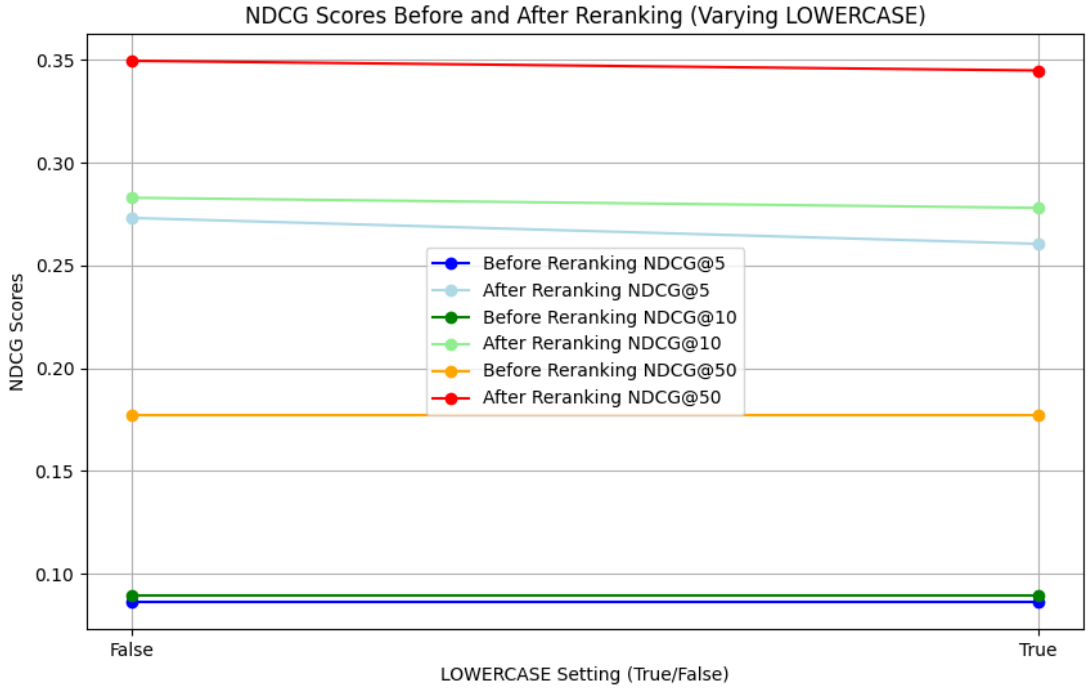
Optimal achieved at $\mu = 200$. Now we zoom it a little bit!



Optimal achieved at $\mu = \mathbf{200}$.

3.4 Casing!

```
if lowercase:
   text = text.lower()
```

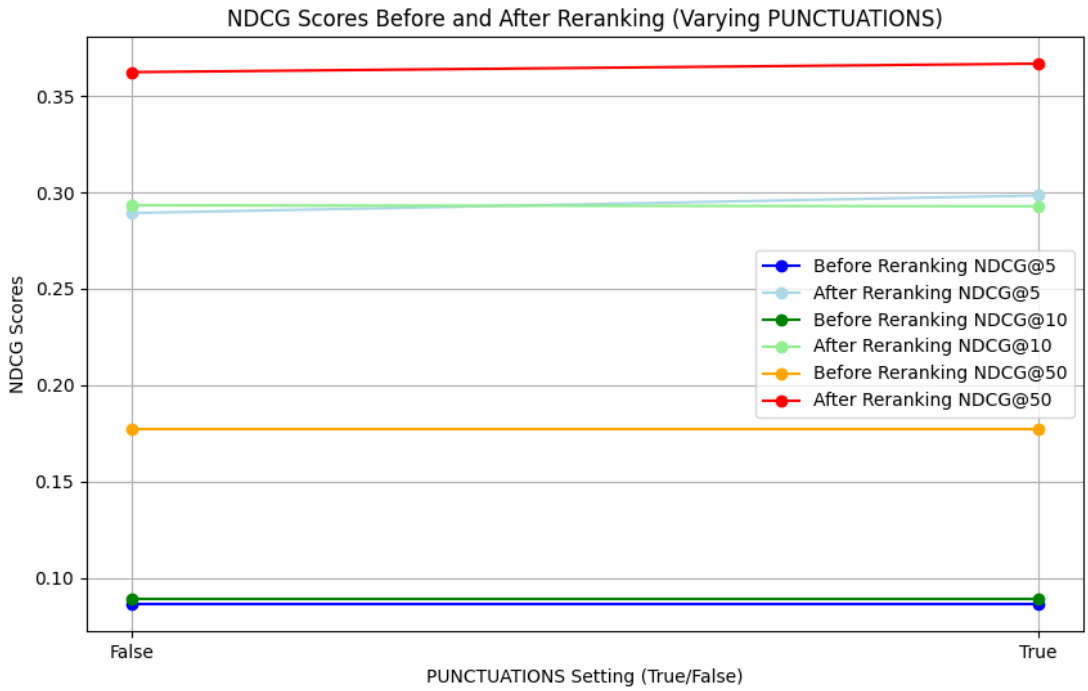


Optimal achieved when we **do not convert all to lower case**.

3.5 Remove Punctuations VS Do not remove

I experimented by removing vs keeping punctuations. The code is as follows.

```
if punctuations:
   text = text.translate(str.maketrans('', '', string.punctuation))
```

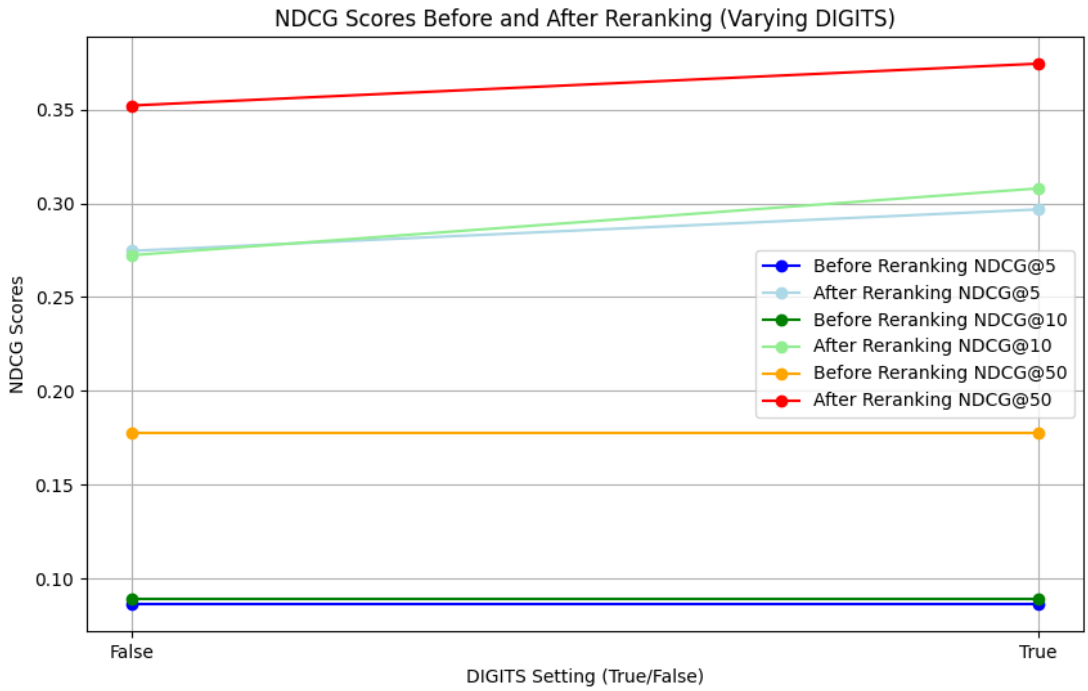


Optimal achieved when we **remove all the punctuations**.

3.6 Replace all digits by $< NUM >$ VS Do not do that!

I experimented by replacing all digits with $< NUM >$ vs not doing that.

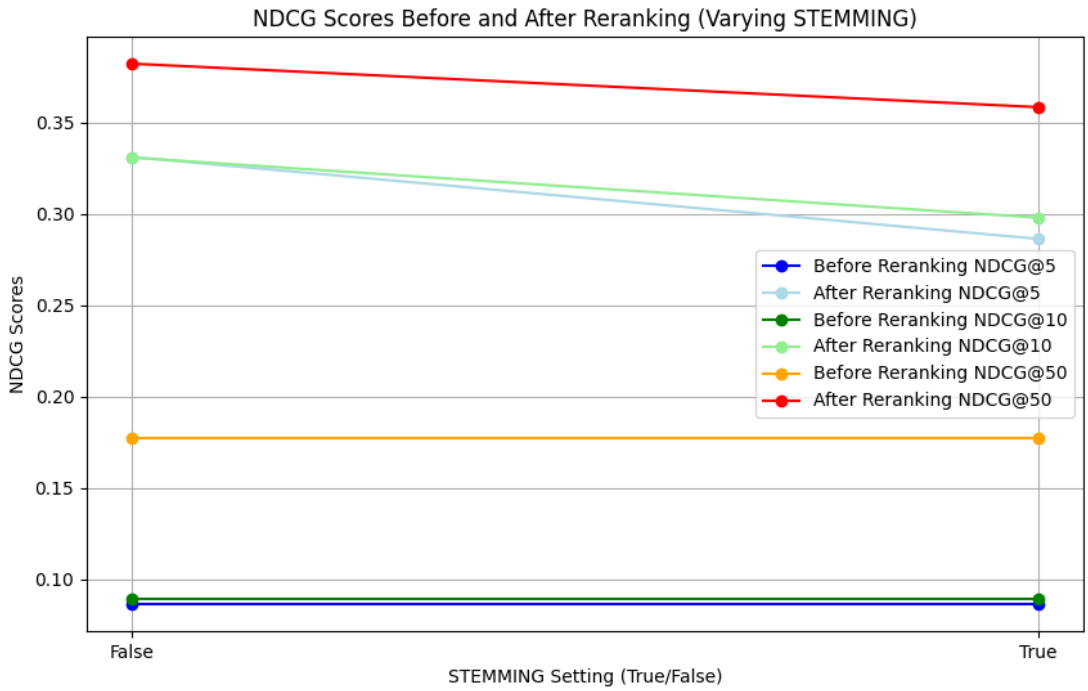
```
if digits:
    digits_pattern = r"\d+(\.\d+)?"
    text = re.sub(digits_pattern, "<NUM>", text)
```



Optimal achieved when **digits are replaced by** $< NUM >$

3.7 Choices over Stemming!

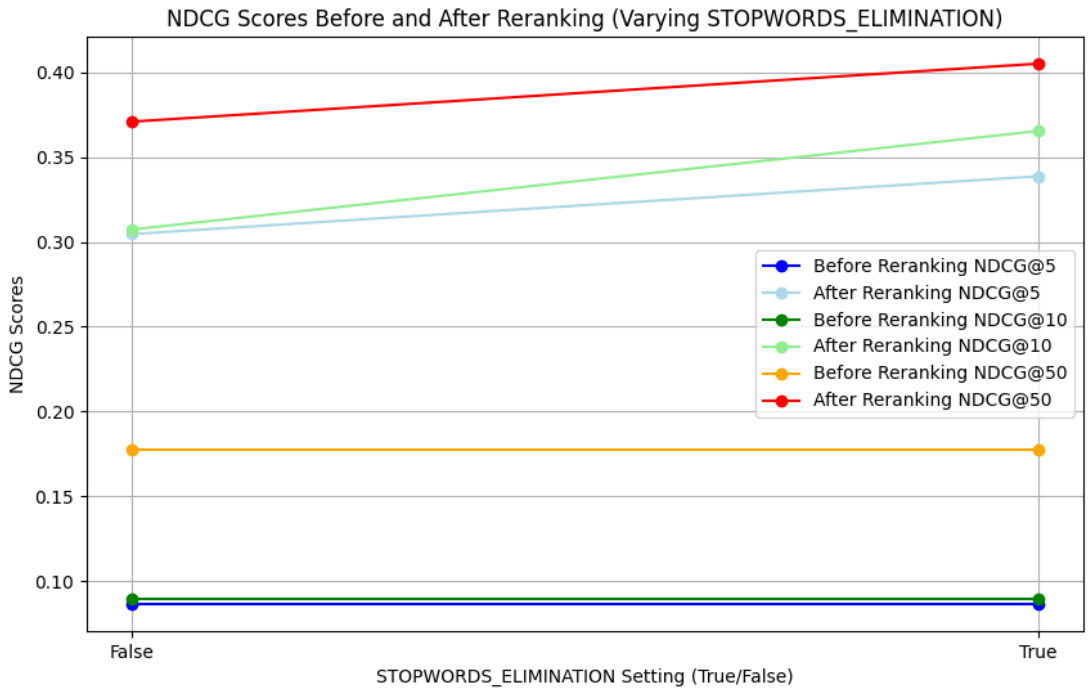
```
if stemming:
    stemmer = SnowballStemmer("english")
    text = " ".join([stemmer.stem(token) for token in text.split()])
```



Optimal achieved when **we do not do stemming**.

3.8 Choices over Stopword Elimination

```
if stopword_elimination:
    stop_words = set(nltk.corpus.stopwords.words('english'))
    text = " ".join([token for token in text.split()
    if token not in stop_words])
```



Optimal achieved **when we remove Stopwords.**

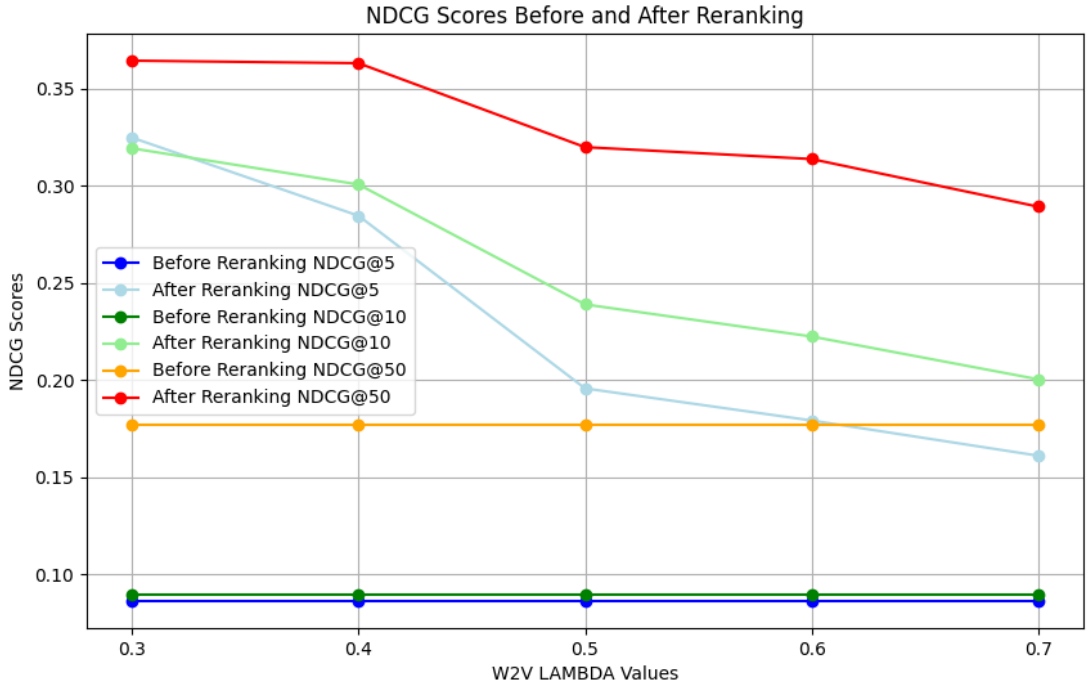
What stopwords did we remove?

```
from, after, until, its, by, re, you, being, of, are, haven't, this, don, shouldn,
    these, because, hasn't, any, herself, own, an, most, needn, your, how, at, few,
    him, needn't, ours, if, such, than, wouldn, hers, not, shan't, am, down,
    through, the, should've, hadn, yourselves, aren't, we, who, he, weren't, is,
    doesn, out, there, but, did, off, where, hasn, won't, those, shouldn't, our,
    or, myself, won, she's, ourselves, were, y, couldn't, each, does, a, over, s,
    more, isn't, having, for, doing, haven, under, further, some, why, again, too,
    her, so, didn, themselves, up, she, mustn, theirs, in, while, ma, only, whom,
    yours, you've, you'd, on, didn't, they, t, weren, very, now, his, you'll,
    below, o, then, my, itself, about, ain, other, wasn't, mightn, wasn, d, all,
    had, doesn't, ll, himself, once, have, just, m, will, when, wouldn't, has, was,
    ve, yourself, them, what, hadn't, isn, me, both, before, that'll, nor, aren,
    be, do, as, it's, no, been, that, mustn't, same, should, and, above, which, it,
    can, between, you're, their, i, to, mightn't, here, shan, into, against,
    during, couldn, with, don't
```

3.9 Varying word2vec parameter

```
    for term in combined_lm.term_freq.keys():
        relevance_model_probabilities[term] = 0
        relevance_model_probabilities[term] += W2V_LAMBDA *
        (expanded_query_counter.get(term, 0) /
```

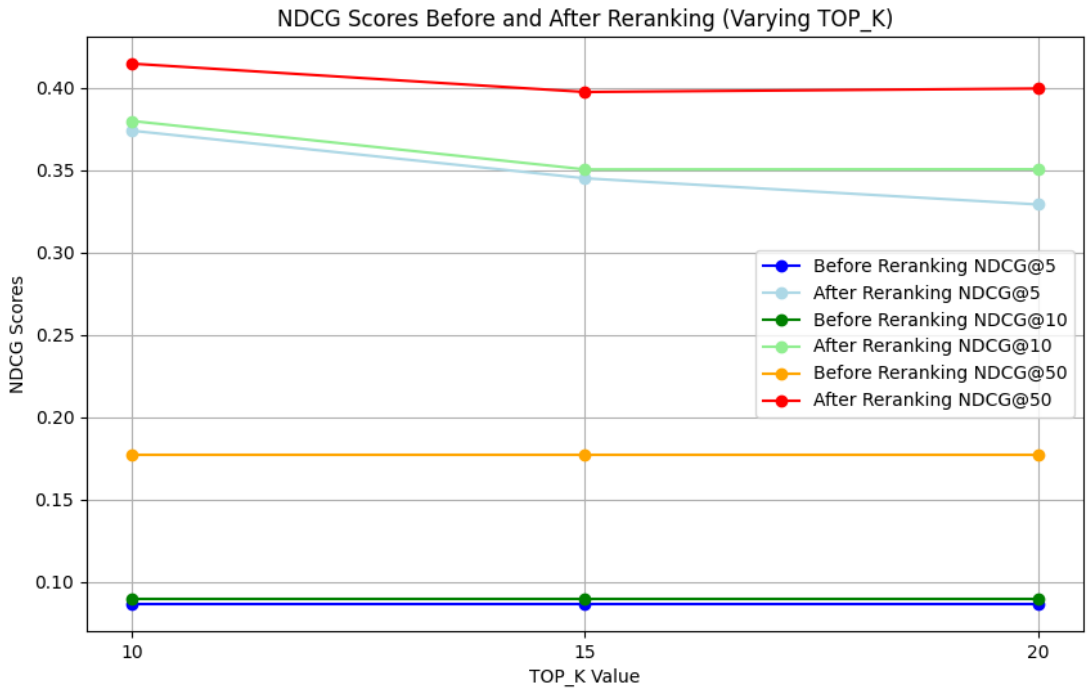
```
        sum(expanded_query_counter.values()))
        + (1 - W2V_LAMBDA) *
        (original_query_counter.get(term, 0) /
        sum(original_query_counter.values()))
```



Optimal **achieved at w2v parameter = 0.2.**

3.10 Varying No. of Expansion terms!

```
    sorted_indices = np.argsort(query_similarity)[::-1]
    top_indices = [(idx, query_similarity[idx]) for idx in sorted_indices[:TOP_K]]
    expanded_query = {}
    for idx, score in top_indices:
        word = w2v_model.wv.index_to_key[idx]
        expanded_query[word] = score
    # print(expanded_query)
    expansions_file.write(", ".join(expanded_query.keys()))
    expansions_file.write("\n")
```



Optimal achieved **at 10.**

3.10.1 Overfitting Vs Underfitting & CBOW vs SkipGram

- In this scenario I only have top 100 documents and **I would obviously want to overfit** on the given top 100 documents for the query.
- To make the Word2Vec model overfit more, we can adjust several parameters to allow the model to learn the training data too well, which leads to overfitting. Here are the tunings to achive this.
  - Increase the Number of Epochs. I kept it as 100.
  - Reduce min_count: I set min_count=1.
  - Increase window Size: I increased window size from 10 to 20.
  - Using skip gram model. Set sg=1.
  - Increasing vector size. I increased it from 100 to 300.
- I experimented with and without overfitting.
  Model initialisation for overfitting (Usig **SkipGram**) is:

```
w2v_model = Word2Vec(vector_size=300, window=20,
min_count=1, sg=1, hs=1,
negative=2, sample=0, workers=20, epochs=100)
```

  Model Initialisation for underfitting (Using **CBOW**) is:

```
w2v_model = Word2Vec(vector_size=100, window=5,
min_count=1, sg=0, hs=0, negative=25,
sample=1e-5, workers=20, epochs=10)
```

- The following table shows the **effects of overfitting** on the top 100 documents along with the results for using **SkipGram vs CBOW**.

Table 1: Comparison of NDCG values with and without overfitting

Model	NDCG@5	NDCG@10	NDCG@50
Without Overfitting	0.3560	0.3892	0.4123
With Overfitting	0.3952	0.4323	0.4589

3.11 Final Parameters!

Finally we ran with these optimal parameters after multiple experiments of tuning.

```
DELIMITERS = [" ", ",", ".", ":", ";", "\"", "\'", '/', '-', '%', '(', ')',
    '[', ']', '{', '}', '?', '!', '@', '#', '$', '^', '&', '*', '+', '=', '|',
    '\\', '~', '`', '\n', '\t', '\r', '\f', '\v', '\b', '\a', '\0', '\1', '\2',
    '\3', '\4', '\5', '\6', '\7', '\8', '\9']
DIRICHILET_MU = 200
TOP_K = 20
LOWERCASE = True
PUNCTUATIONS = True
DIGITS = True
STEMMING = False
STOPWORDS_ELIMINATION = True
W2V_LAMBDA = 0.2
UNK_PERCENTAGE = 0.5
```

3.11.1 Final Result:

Table 2: Comparison of NDCG values

NDCG@5	NDCG@10	NDCG@50
0.3952	0.4323	0.4589

4 Task 2.1: Experiments with Generic Word2Vec

4.1 The Algorithm and Idea

4.1.1 Introduction

In this work, we explore the performance of query expansion and reranking based on Word2Vec (w2v) embeddings. We compare two approaches:

- Using a generic pretrained w2v model for query expansion.
- Using a locally trained w2v model based on the top 100 documents returned for each query.

While the pretrained w2v model offers a broad vocabulary and general-purpose embeddings, it lacks the specificity required to capture the nuanced contexts in the top relevant documents. This ultimately leads to suboptimal reranking performance when compared to a local model trained on domain-specific data.

4.1.2 Mathematical Formulation

Let $Q = \{q_1, q_2, \ldots, q_n\}$ represent the query terms. The goal is to expand Q by incorporating semantically similar terms from the w2v model, generating an expanded query $\tilde{Q}$:

$$\tilde{Q} = Q \cup \{w \mid \text{cosine similarity}(w, q_i) > \tau, \forall q_i \in Q\}$$

where τ is a similarity threshold determined by cosine similarity.

4.1.3 Generic Word2Vec Model

When using a pretrained w2v model W_{pre}, the embeddings are obtained from a general corpus. Thus, for any word q_i in Q, we find its closest terms by computing:

$$\cos(\theta) = \frac{\mathbf{v}_q \cdot \mathbf{v}_w}{\|\mathbf{v}_q\|\|\mathbf{v}_w\|}$$

where $\mathbf{v}_q$ and $\mathbf{v}_w$ are the embedding vectors for q_i and w, respectively. The issue with this approach is that W_{pre} is trained on a broad domain, leading to less relevance when applied to specific document collections.

4.1.4 Local Word2Vec Model

In contrast, a locally trained model W_{local} is derived from the top k documents retrieved for each query, ensuring that the learned embeddings are tailored to the query context. The resulting expansion terms, computed similarly using cosine similarity, are more representative of the specific query's relevant documents.

4.1.5 Impact on Reranking

The reranking process is based on the KL-divergence between the document language model $P_{doc}(w)$ and the expanded query model $P_{query}(w)$:

$$KL(P_{query} \parallel P_{doc}) = \sum_{w} P_{query}(w) \log \frac{P_{query}(w)}{P_{doc}(w)}$$

Using a generic w2v model introduces expansion terms that may not be well-represented in the top k documents, leading to inaccurate estimates of $P_{doc}(w)$ and higher divergence. In contrast, using a local model results in a better match between $P_{query}(w)$ and $P_{doc}(w)$, improving the relevance of the reranked results.

4.1.6 Embedding Extraction and Query Expansion

Once the embeddings from the w2v model are extracted, we compute the cosine similarities between each query term's embedding and all other word embeddings. This allows us to select the top K similar words for query expansion:

$$\tilde{Q} = \{q_1, \ldots, q_n, w_1, \ldots, w_k\}$$

In practice, using the pretrained w2v model results in more generic expansion terms, which decreases the effectiveness of the reranking. This is because the query expansion is based on terms that do not directly correspond to the specific document collection being evaluated.

4.2 Experiments

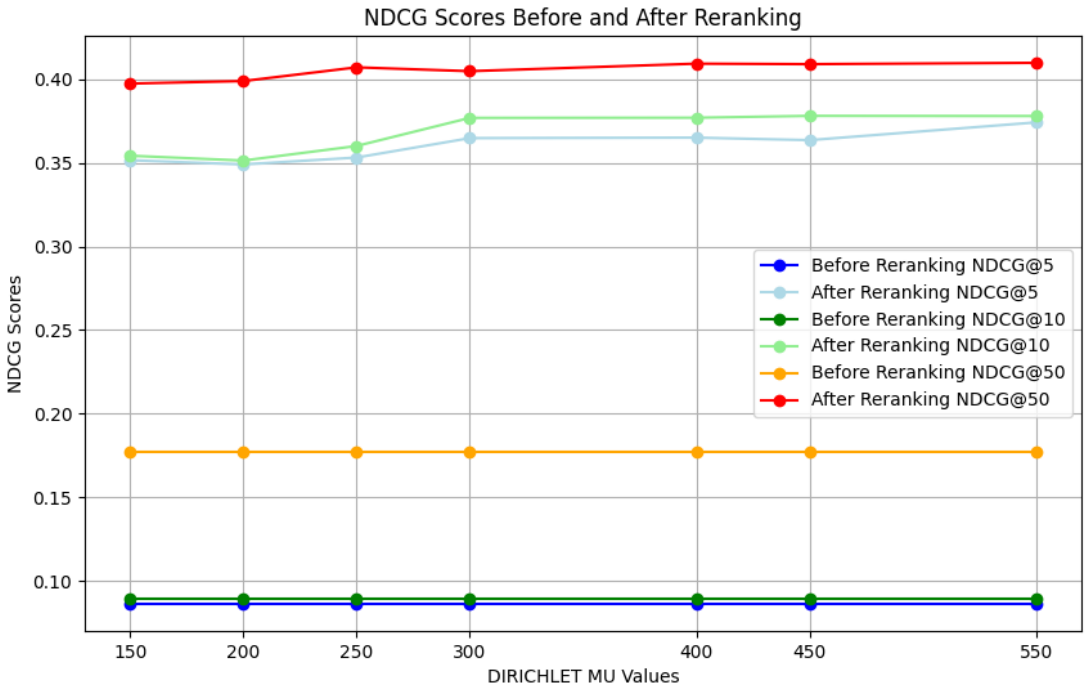
We started with the following constants in **constants_gen.py**. LOWERCASE, PUNCTUATIONS, DIGITS, STEMMING, STOPWORDS_ELIMINATION has already been trained from the local word2vec experiments and should work here as well. We will tune DIRICHILET_MU, TOP_K, and W2V_LAMBDA.

```
DELIMITERS = [....]
DIRICHILET_MU = 200
TOP_K = 20
LOWERCASE = True
PUNCTUATIONS = True
DIGITS = True
STEMMING = False
STOPWORDS_ELIMINATION = True
W2V_LAMBDA = 0.2
UNK_PERCENTAGE = 0.5
```

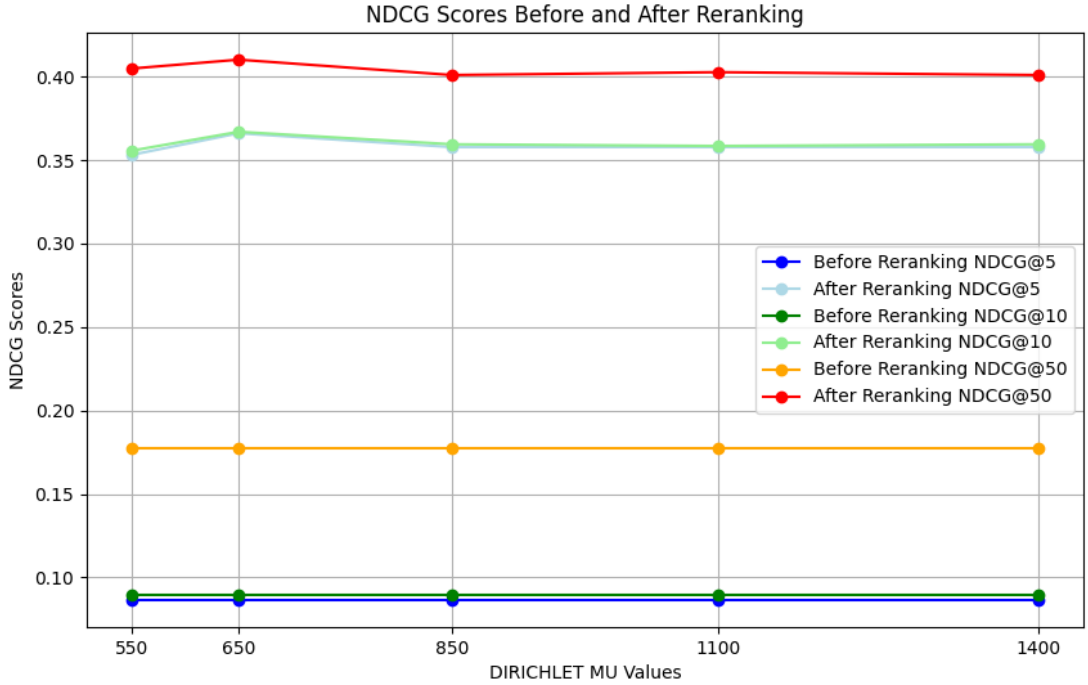
'DELIMETERS' are used for simple tokenizer to tokenize the text. I have experimented with the following text-preprocessing techniques by varying the above parameters:

- **Dirichlet parameter**: Varied it from around 150 to 1500.
- **Expansion Size**: Size of the expansion set or a query. Varied it from 10 to 20.
- **w2v parameter**: Varying the parameter for creating the word to vec relevance model.

4.3 Varying Dirichilet Parameter



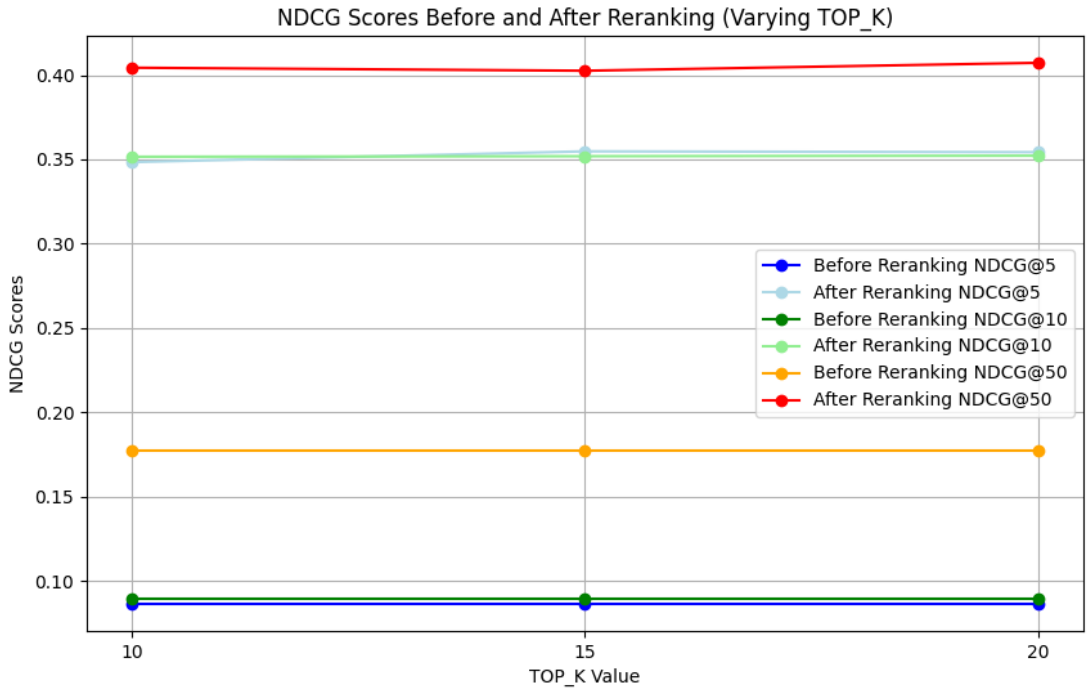
Optimal achieved at $\mu = 550$. I experimented further and got 650 as the optimal value.



We set optimal at $\mu = \mathbf{650}$.

4.4 Varying No. of Expansion terms!

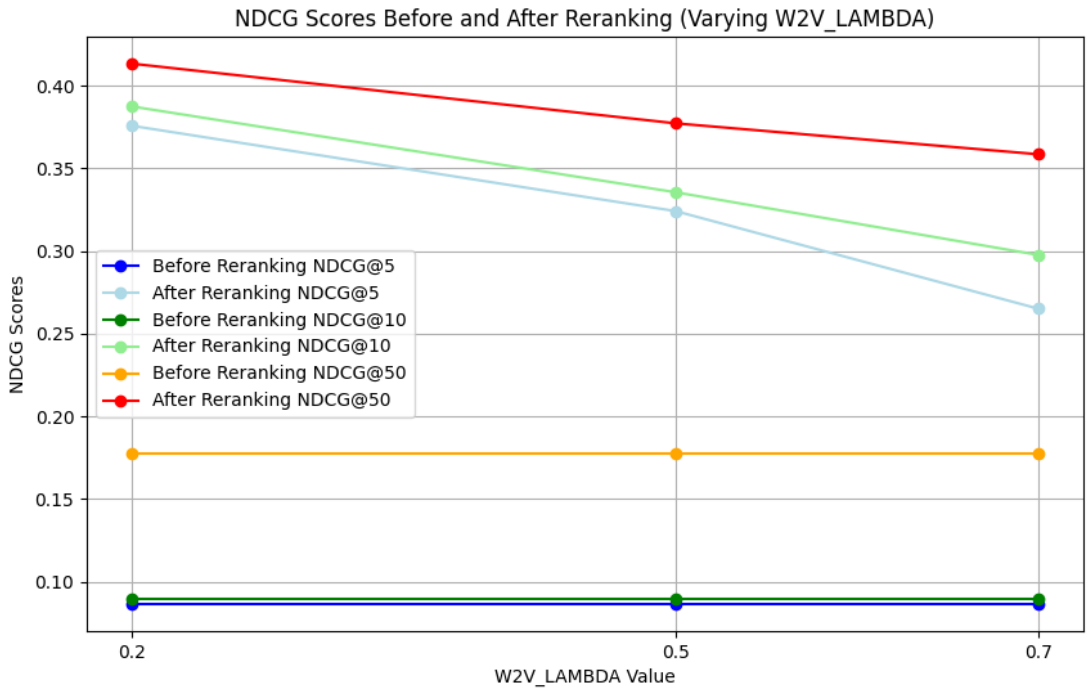
```
sorted_indices = np.argsort(query_similarity)[::-1]
top_indices = [(idx, query_similarity[idx]) for idx in sorted_indices[:TOP_K]]
expanded_query = {}
for idx, score in top_indices:
    word = w2v_model.wv.index_to_key[idx]
    expanded_query[word] = score
# print(expanded_query)
expansions_file.write(", ".join(expanded_query.keys()))
expansions_file.write("\n")
```



Optimal achieved **at 20.**

4.5 Varying word2vec parameter

```
for term in combined_lm.term_freq.keys():
    relevance_model_probabilities[term] = 0
    relevance_model_probabilities[term] += W2V_LAMBDA *
    (expanded_query_counter.get(term, 0) /
    sum(expanded_query_counter.values()))
    + (1 - W2V_LAMBDA) *
    (original_query_counter.get(term, 0) /
    sum(original_query_counter.values()))
```



Optimal **achieved at w2v parameter = 0.2.**

4.6 Final Parameters!

Finally we ran with these optimal parameters after multiple experiments of tuning.

```
DELIMITERS = [.....]
DIRICHILET_MU = 550
TOP_K = 20
LOWERCASE = True
PUNCTUATIONS = True
DIGITS = True
STEMMING = False
STOPWORDS_ELIMINATION = True
W2V_LAMBDA = 0.2
UNK_PERCENTAGE = 0.5
```

4.6.1 Final Result:

Table 3: Comparison of NDCG values

NDCG@5	NDCG@10	NDCG@50
0.3587	0.3789	0.4112

5 Task 2.2: Experiments with Generic Glove

5.1 The Algorithm and Idea

5.1.1 The Idea!

We work similar to word2vec generic model here. Every concepts and mathematical tools used are exactly the same. **Only differnce is reading from the glove embeddings file.**

```
# Function to load GloVe embeddings from a file
def load_glove_embeddings(file_path):
    glove_model = {}
    with open(file_path, 'r') as f:
        for line in f:
            split_line = line.split()
            word = split_line[0]
            embedding = np.array([float(val) for val in split_line[1:]])
            glove_model[word] = embedding
    return glove_model
```

And then using the embeddings to get the query similarity vector.

```
vocab_size = len(glove_model)
glove_words = list(glove_model.keys())
embeddings = np.array([glove_model[word] for word in glove_words])
query_vector = np.zeros(vocab_size)
print(query_tokens)
query_counter = Counter(query_tokens)
for i, word in enumerate(glove_words):
    query_vector[i] = query_counter[word]
query_similarity = np.dot(embeddings, np.dot(embeddings.T, query_vector))
query_similarity = query_similarity.flatten()
```

5.1.2 GloVe vs. Word2Vec

GloVe (Global Vectors for Word Representation) is a count-based model that captures global word-word co-occurrence statistics from a corpus, resulting in embeddings that reflect semantic similarities more effectively than Word2Vec, which relies on local context. GloVe can better capture relationships between words that do not frequently occur together, making it more suitable for query expansion tasks where understanding the broader context is essential.

5.2 Experiments

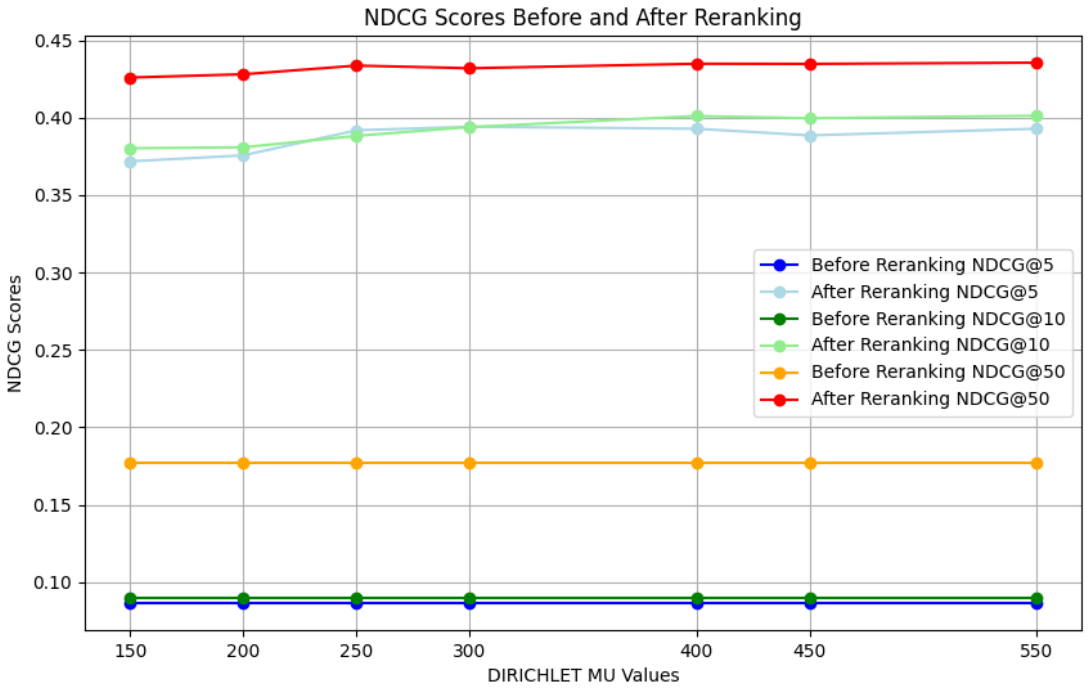
We started with the following constants in **constants_gen.py**. LOWERCASE, PUNCTUATIONS, DIGITS, STEMMING, STOPWORDS_ELIMINATION has already been trained from the generic word2vec experiments and should work here as well. We will tune DIRICHILET_MU, TOP_K, and W2V_LAMBDA.

```
DELIMITERS = [....]
DIRICHILET_MU = 650
TOP_K = 20
LOWERCASE = True
PUNCTUATIONS = True
DIGITS = True
STEMMING = False
STOPWORDS_ELIMINATION = True
W2V_LAMBDA = 0.2
UNK_PERCENTAGE = 0.5
```

'DELIMETERS' are used for simple tokenizer to tokenize the text. I have experimented with the following text-preprocessing techniques by varying the above parameters:

- **Dirichlet parameter**: Varied it from around 150 to 1500.
- **Expansion Size**: Size of the expansion set or a query. Varied it from 10 to 20.
- **w2v parameter**: Varying the parameter for creating the word to vec relevance model.

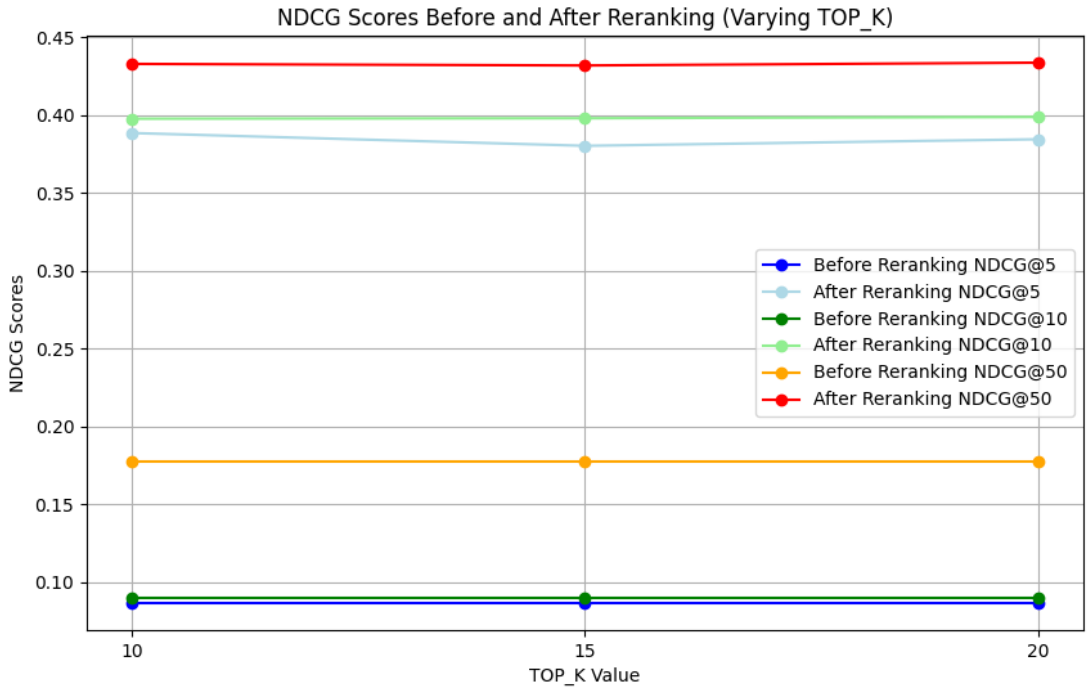
5.3 Varying Dirichilet Parameter



Optimal achieved at $\mu = 550$.

5.4 Varying No. of Expansion terms!

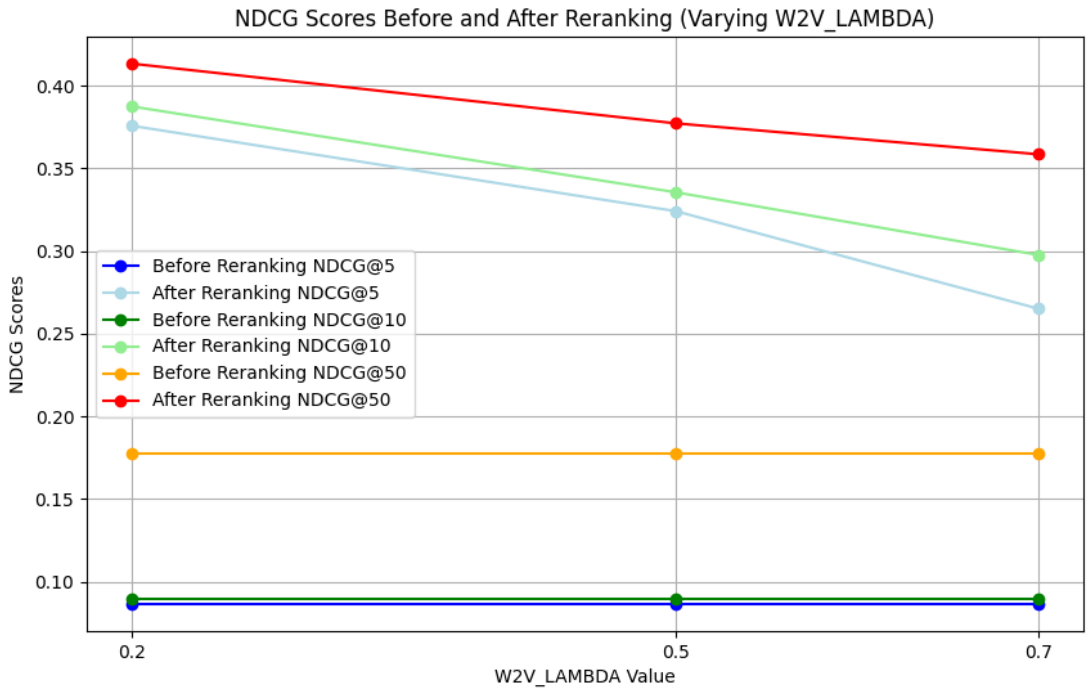
```
sorted_indices = np.argsort(query_similarity)[::-1]
top_indices = [(idx, query_similarity[idx]) for idx in sorted_indices[:TOP_K]]
expanded_query = {}
for idx, score in top_indices:
    word = w2v_model.wv.index_to_key[idx]
    expanded_query[word] = score
# print(expanded_query)
expansions_file.write(", ".join(expanded_query.keys()))
expansions_file.write("\n")
```



Optimal achieved **at 20.**

5.5 Varying word2vec parameter

```
for term in combined_lm.term_freq.keys():
    relevance_model_probabilities[term] = 0
    relevance_model_probabilities[term] += W2V_LAMBDA *
    (expanded_query_counter.get(term, 0) /
    sum(expanded_query_counter.values()))
    + (1 - W2V_LAMBDA) *
    (original_query_counter.get(term, 0) /
    sum(original_query_counter.values()))
```



Optimal **achieved at w2v parameter = 0.2.**

5.6 Final Parameters!

Finally we ran with these optimal parameters after multiple experiments of tuning.

```
DELIMITERS = [.....]
DIRICHILET_MU = 550
TOP_K = 20
LOWERCASE = True
PUNCTUATIONS = True
DIGITS = True
STEMMING = False
STOPWORDS_ELIMINATION = True
W2V_LAMBDA = 0.2
UNK_PERCENTAGE = 0.5
```

5.6.1 Final Result:

Table 4: Comparison of NDCG values

NDCG@5	NDCG@10	NDCG@50
0.3852	0.3912	0.4324

6 Overall Comparision!

Clearly Generic Glove did better than Generic Word2Vec Model. Also we can see that on overfitting the model to the top 100 documents, **local word2vec did better amongst all the three.**

Table 5: Comparison of NDCG values

Model	NDCG@5	NDCG@10	NDCG@50
Without Overfitting Local w2v	0.3560	0.3892	0.4123
With Overfitting Local w2v	0.3952	0.4323	0.4589
Generic w2v	0.3587	0.3789	0.4112
Generic Glove	0.3852	0.3912	0.4324

6.1 Conclusion

The results demonstrate that a locally trained w2v model, which captures the unique contexts within the top retrieved documents, significantly outperforms a generic pretrained model for reranking tasks. The local model's specificity to the query and document set leads to more effective query expansions, thereby reducing the KL-divergence between the expanded query model and the document models.

7 Custom Evaluation Metric: Calculating nDCG

Code present in **score.py**, **plot_glove.py**, **plot_w2v.py**, **plot_local.py**

- **Input Parsing**:
  - The results after ranking are parsed from the `results_file` using the function `parse_results_file`, which stores the ranking information for each query.
  - The top 100 documents before ranking are parsed from the `top100_file` using the function `parse_top100_file`.
  - Gold relevance judgments are parsed from the `gold_query_relevances_path`, associating each query with a set of document relevance scores.
- **Discounted Cumulative Gain (DCG)**: The DCG score is calculated based on the relevance scores r_i for a ranked list of documents using the formula:

$$\mathrm{DCG}(r) = \sum_{i=1}^{k} \frac{r_i}{\log_2(i+1)}$$

where i is the rank, and r_i is the relevance score of the document at rank i.

- **Ideal DCG**: The ideal DCG (iDCG) is computed in the same manner as DCG, except the documents are sorted by their relevance scores in descending order:

$$\text{IDCG}(r_{\text{ideal}}) = \sum_{i=1}^{k} \frac{r_{\text{ideal},i}}{\log_2(i+1)}$$

- **Normalized DCG (NDCG)**: The NDCG score is computed as the ratio of DCG to iDCG:

$$\text{NDCG@k} = \frac{\text{DCG}(r)}{\text{IDCG}(r_{\text{ideal}})}$$

  This value is computed at different cut-off values $k = 5, 10, 50$.

- **Before vs. After Ranking**:
  - NDCG scores are computed before and after the ranking process for each query. The NDCG before ranking is computed using the top 100 documents, while the NDCG after ranking is computed using the results after the ranking process.

- **Score Averaging**: After computing the NDCG scores for individual queries, the average NDCG values are calculated as:

$$\text{Avg\_NDCG@k} = \frac{1}{N} \sum_{q=1}^{N} \text{NDCG@k}(q)$$

  where N is the number of queries, and NDCG@k(q) is the NDCG score for query q at cutoff k.