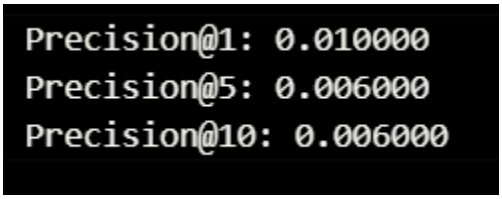
Text pre-processing and matching using spaCy

Task 1:

1. 1st using regex, I have done preprocessing to remove characters other than alphanumeric and spaces
2. Spelling correction is done using spacy spellchecker known as contextual spellcheck
3. Used parallel processing to make the above 2 process faster
4. Now, tokens are found for each document in doc.csv using spacy's tokenization. Based on this as per given criteria that Remove all words that occur in less than 5 documents or more than 85% of the documents, we have remove all those words. Generate the vocabulary from the remaining words and then cleared all the documents by removing the words which are not in vocabulary.
5. Similarly tokens are also generated for each query and each query is also modified by removing the tokens which are not in vocabulary that we have created earlier.
6. Now, using these vocabulary I have created a tf-idf embeddings for 100 queries and 10000 documents.
7. I have found out cosine similarity between each pair of query and document, stored in a matrix and then found out top k documents w.r.t each query in the dataset.
8. To calculate precision@k scores, I have 1st create a relevance matrix using the qdrel.csv where matrix is initialized using zeros and then updating the relevance matrix to 1 corresponding to pair of querd_id and doc_id which are present in qdrel.csv
9. Based on the relevance matrix and top_k_documents indices I have calculated precision@k scores for k=1,5,10 averaged over all queries. Results are given below.



The results in percentage are 1%, 0.6% and 0.6% respectively.
The results are too low. The following are may be the possible reasons for the low precision score:

A. The size of test dataset given is very small, 130 query-doc pairs. So, we know average approximately 1.3 doc per pair. So, if we dont know how many docs are actually relevant for given query, it is very difficult to determine the precision value as the formula for precision@k is as follows:

$$\text{precision@k} = \frac{\text{number of recommended items that are relevant @k}}{\text{number of recommended items @k}}$$

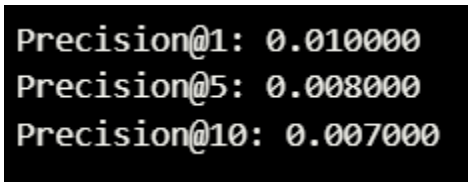
So, though we are finding top k documents for each query, we actually dont have sufficient no. of relevant documents for each query. So, this may be the possibel reason.

B. Other Important reason is that using spacy library for all the pre-processing. Spacy's spellchecker is not much good and it sometimes change the word to a new word instead of correcting it. This causes changing the context of sentences and overall affecting the cosine similarity and thereby less contribution to precision score averaged over all queries.

C. The embeddings which we have used are Tf-Idf which totally depends on the corpus or vocabulary which we are using. So, if the spellchecker doesn't work good, then vocabulary which is made will not be good and able to generate good embeddings making the precision better.

Task 2.1: Stemming

1. SpaCy does not provide a built-in stemming function because it generally focuses more on lemmatization. So, I have used a NLTK PorterStemmer for stemming.
2. Once stemming is done then all the steps are similar which we have done in Task 1 from step 4 to 9. The results which I was getting for stemming average over all queries is as follows:

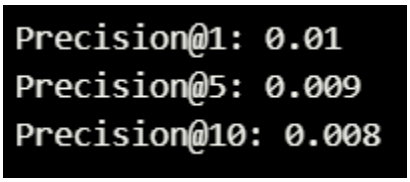


Here the results in percentages are 1%, 0.8% and 0.7% respectively. The results are still low. For Precision@1 the score is same, but it is slightly increase for Precision@5 and Precision@10. The possible reasons for increase may be:

A. Stemming maps related words to the same stem (e.g. run, running -> run) which reduces the overall vocabulary size and sparsity for vector space models.
B. Query and documents may use variant word forms (e.g. organize, organizes). Stemming bridges such superficial gaps by mapping to common root (organize).
C. Stems better capture core semantic content rather than precise syntactic form. This allows similarity measures to better capture topical relevance as compare to normal tokens.

Task 2.2: Lemmatizing

1. Lemmatizing of tokens is done using the spacy's inbuilt lemmatization function.
2. After lemmatization is done similar steps are followed from 4 to 9 as given for Task 1.The results after doing lemmatizing of tokens average over all queries are as follows:



The results are 1%, 0.8% and 0.9% following a similar increase in rend as we see in stemming. The possible reasons for these are as follows:

A. Lemmatization enhances information retrieval tasks by consolidating different inflected forms of a word to a common base form, reducing vocabulary dimensions.
B. It aids in matching variations in word forms between queries and documents, increasing the likelihood of relevant matches.

C. By encoding core semantic content and removing inflectional differences, lemmatization improves similarity calculations, capturing semantic relevance more effectively.

The following image shows the vocabulary size that we have created in Task1, Task 2.1(Stemming) and Task2.2(Lemmatizing). These clearly shows that size is decreasing from Task1 to Task 2.2 as Stemming and Lemmatizing can maps multiple words to same token as per their specific functionality.

```
Vocabulory size of Task 1:  2435
Vocabulary size after Stemming:  2118
Vocabulary size after Lemmatizing:  2091
```

Task 3:

1. I have found out POS tagging and NER for each of the documents and queries given in the dataset.
2. As per given problem, I have used the same tf-idf embeddings which I have generated in Task 2.2 and weights the embeddings by multiplying with 2 along the dimensions which contain nouns, and multiplying 4 for the named entities.
3. Now, these new embeddings are used for generating cosine similarity and further steps similar to step 7-p of Task1. The results of Task3 are as follows:

```
Precision@1: 0.01
Precision@5: 0.009
Precision@10: 0.009
```

A. By assigning higher significance to these semantically rich elements, the scheme enhances the potential for meaningful matches between queries and documents.

B. Simultaneously, it effectively reduces noise by downweighting less informative words, such as articles and prepositions, contributing to a cleaner semantic signal. The weighting strategy contributes to a more discriminative vector space, fostering improved accuracy in ranking and retrieval processes.

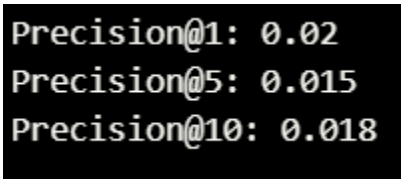
C. Additionally, the emphasis on boosting named entities ensures that the embeddings align closely with core topical content, promoting relevance beyond mere statistical matches.

Task 4:

Improvement 1:

I have used Annoy tree to improved the performance of Task 1 as it is generally used in recommendation based system and out matching task is similar to it. It constructs specialized data structures like trees/graphs to enable fast nearest neighbor lookup between high-dimensional vectors. Utilize locality-sensitive hashing to prune the search space of possible candidates. Provides an approximate set of the most similar vectors, unlike exact brute-force search.

1. We have preprocess data using nltk, spell checking using pyspellchecker and then used Bert to generate the embeddings as BERT tends to produce high-dimensional vectors, and cosine similarity is well-suited for comparing vectors in high-dimensional spaces.
2. An Annoy index is constructed based on the document vectors, and queries are then matched to approximate nearest neighbors in this index.
3. The Annoy index employs angular distance as the metric for measuring vector similarity. During the search process, the library efficiently identifies approximate matches, and the specified number of results (num_results) is retrieved for each query vector. The obtained results include both the indices and distances of the approximate nearest neighbors.
4. The Annoy index employs angular distance as the metric for measuring vector similarity. During the search process, the library efficiently identifies approximate matches, and the specified number of results (num_results) is retrieved for each query vector. The obtained results include both the indices and distances of the approximate nearest neighbors.
5. At end, Precision@k is similarly found as per Step 7-9 of Task1. The results of this task are as follows:



Improvement 2:

I havent get a sufficient time to code these, so I am explaining the approach we I have think off. In these I have think of an ensemble approach

1. Clean and preprocess both query and document texts. Include steps like removing special characters, converting to lowercase, and handling stop words.
2. Tokenization and Lemmatization:Tokenize the preprocessed text into individual words and apply lemmatization to map different inflected forms to their base form, reducing vocabulary dimensionality.
3. Use spaCy or a similar library for Named Entity Recognition and Part-of-Speech tagging. Identify entities (e.g., persons, organizations) and parts of speech
4. Use semantic embedding models like Sentence-BERT, Utilize pre-trained word embeddings (e.g., Word2Vec, GloVe) or contextual embeddings (e.g., BERT) for both queries and documents.
5. Used 3 types of metrics: cosine similarity, jacard simialrity and Minkowski Distance and then for each query find all 3 metrics with each vector and then we can weight each metrics either randomly or with some specific weights. So, we will get a better result as we are taking ensemble of results to find the best relevant k documents.

Problems Faced:

1. It was difficult to process data for different tasks like spelling correction, token generation because each time we are loading a csv and so it is taing so much RAM. Because of this i was not possible to run the code on Google Colab. Also my laptop is of i3 generation with 4 cores, so it is also taking too much time to run on my local machine like it was taking approx 3 hours for spelling correction of 10,000 documents.
2. The another problem is the size of data which is given for evaluation methods. As it contains only 130 query-doc id pairs which are similar to each other. So, it was very difficult to evaluate the methods with this much small amount data.