Deliverable 2: Neural Retrieval Methods

CSI4107: Information Retrieval and the Internet, Winter 2023
Instructor: Diana Inkpen

Presented By
Adrian D’Souza (300066117)
Nicholas Gin (300107597)
Jared Wagner (300010832)

Program Overview

Division of Labour

Group Members	**Tasks**
Adrian D'Souza	- Report Writing - Base Program Revision (BM25) - Experiment 1 (All Versions) - Experiment 3 (SBERT)
Nicholas Gin	- Report Writing - Base Program Translation (Cosine Similarity) - Experiment 1 (SBERT, GPT3) - Experiment 3 (All Versions)
Jared Wagner	- Report Writing - Experiment 2 (All Versions)

Group members assert that the division of labour between members was fair and equal.

Program functionality and algorithms, data structures, and optimizations used in each of the experiments

In Assignment 1, our group designed and implemented an IR (Information Retrieval) system in Java based on VSM (Vector Space Model). This initial model pre-processed a collection of documents, removing stopwords, and ranking these documents using cosine similarity between processed documents and queries. This model attained an optimal MAP score of 0.1507, presenting us with both an opportunity to improve on the system in the future, and the knowledge we needed to properly explore the use of neural IR techniques in this assignment.

Even the world's best driver only knows it if they drive with their headlights on. In other words: choosing appropriate benchmarks for our models in this assignment was just as important as how we implemented them. To choose an appropriate benchmark, we took inspiration from top-performing groups in assignment 1.

One non-neural technique that performed particularly well on this corpus in the previous assignment was the BM25 ranking function, so this was a solid benchmark used in our work. More on BM25 and how it was used in future sections.

One challenge we faced early on in development was finding Java-based neural libraries. It seems that development in data science and IR is dominated by Python. We all stand on the shoulders of giants by building upon the work that comes before us. With that in mind, we translated our system into Python code to benefit from the slew of neural IR Python libraries freely available online. We commonly refer to this as a "base version" of the model in the rest of our report. Converting our Java code to Python can be seen as an optimization step in the development of our IR System; changing our ranking metric from cosine similarity to BM25 was also an optimization step.

After translating the code our MAP score was 0.1468, which we attribute to differences between how Java and Python process data (in particular: Doubles are natively supported in Java, but not in Python.)

Finally, we could start our own experiments using neural techniques. Our group attempted variations on all three of the experiments suggested in the handout, detailed in the following sections.

Experiment 1

Imagine you have been tasked with sorting all the quartz stones on a beach from smallest to largest. In a sense, your real task is two-fold. First, you need to sieve quartz stone from sand (separating relevant documents from irrelevant ones). Second, you need to order the stones by their size (making a ranking). Breaking down this task into two smaller tasks is the bases of Experiment 1.

In this experiment, a base version of our program is used to identify the top 1000 documents for each query. Next, these documents are re-ranked using a neural retrieval method. The assumption used here is that the original documents are relevant to the query, but they can be re-ranked more effectively using a neural model.

For this, we built three versions of our program (each used one of three neural retrieval models: USE, BERT or GPTNeo) to re-rank the initial results from the base program that used BM25 to rank documents. The BM25 base program produced a higher MAP score than the Cosine Similarity base program, so we built off that one.

All three versions use the same underlying structure:

1. The base program (that uses BM25 to produce initial ranking results).
2. A re-ranking class specifically tailored to a given neural retrieval model; we use the model to produce vectorized embeddings that are used to re-rank the initial results from the base program.

Base Program

Please note that the explanation of the base program is almost identical to that of our IR System Program designed for Assignment 1, so much of the explanation below is the same as the explanation for Assignment 1. However, we have made certain modifications to the program's underlying structure, including the incorporation of BM25 comparisons and a neural retrieval model, in an attempt to enhance the overall effectiveness of the program.

The base program is run using the **IRSystem** class **(irSystem.py)**, which contains the main(…) contains the main control method for the system.

The base program can produce an initial set of ranking results by completing the following steps: processing, indexing, retrieval and ranking.

Preprocessing

This step primarily concerns the functionality of the **Preprocessor (preprocessor.py)** and **Index (index.py)** classes.

IRSystem pickles the Index created at the end of this Preprocessing step, so it can loaded later if the program is re-run; this will an optimization step we introduced in Assignment 2, so that Index will not have to be re-computed every time the program is run.

IRSystem creates an instance of the **Preprocessor** that takes two constructor variables – the input folder that points to the corpus collection (*input_folder*) and an instance to the index (*index*) object.

The *input_folder* is provided as a parameter for **IRSystem**'s constructor. Our goal with Preprocessor was to (1) identify all documents contained in the corpus, (2) form a collection of document/corpus vocabulary, and (3) transform each document's vocabulary into tokens stored in the Index (after stripping unwanted tokens, such as punctuation and stopwords, and stemming all the words in each document).

Each document in the corpus has a corresponding entry in an **Index** instance, which stores tokens and their frequency. To accomplish this, we built the **Index** class so that each instance is a custom data object with three instance variables:

1. *index*: A dict of the form (key documentID : value dict(key token : value int tokenFrequency)). In other words, the actual index containing each token and its frequency for a particular document.
2. *stopwords*: A dict of the form (key stopword) containing the collection of stopwords. These are retrieved and stemmed from the provided collection of stop words from Assignment 1, each having int 1 as a dummy value.
3. *numDocs*: An integer tracking the total number of documents in the corpus. In our system, this is 79923 documents once the entire Index has been read from the provided corpus folder.
4. *full_doc*: A dict of the form (key documentID : value fullDocumentText). These are the full-length documents from the corpus (which undergo the pre-processing process described below but are not stemmed and do not have stop words removed from them). It is necessary to provide the neural retrieval models used in this experiment with these versions of the documents as stemming the contents of the documents and removing stop words from them negatively impacts those models' performance.

There are 79923 documents spread across 322 files that comprise our corpus. Each document had a structure similar to that of an HTML markup, a property used by **Preprocessor** (particularly its method *preprocess*) to extract relevant information about each file using regex patterns. The sections we extracted tokens from are a document's ID, title and text.

In the **Preprocessor**, we ran the following algorithm:

1. Navigate to the corpus folder specified by *input_folder* and begin reading a file from this folder.
2. Use regex to identify parts of the document that were irrelevant and only keep the relevant parts (document ID, title and text).
3. After locating a document in the file, iterate over its contents word by word and strip it of all punctuation and capitalization. Also, remove any numbers from the documents. What remains are the tokens for this document.
4. The result of Step 3 is then saved as a value in *full_doc*; its key is the document ID.
5. Step 5 differs based on if the Cosine Similarity implementation of the base program or the BM25 implementation of the base program is used.
    a. If the Cosine Similarity implementation is used…
        i. Using the **PorterStemmer** class, stem each token before passing it to *index* (**stemmer.py**: the pre-existing Porter Stemmer provided in the assignment outline).
            1. If a document has not yet been added to this dict, a new string key for that document will be added to the dict. Another dict will be mapped to this string, with the token (as a string) mapped to an integer (the term frequency of this token in the document).
            2. If the document has already been added to this dict, the document's value will be updated to reflect the addition of a new token to the Index or an increase in the term frequency of a token.
            3. Note that *index* will check if the token is a stop word; each stop word has also been stemmed using the **Stemmer** class. Because all stop words are stored as keys in the dict representing the collection of stop words, we can conclude that a token is a stop word if it matches a key in this dict. If a token is a stop word, it will not be indexed.
    b. If the BM25 implementation is used…
        i. Using the **PorterStemmer** class, stem each token before passing it to *index* (**stemmer.py**: the pre-existing Porter Stemmer provided in the assignment outline).
        ii. Concatenate all stemmed tokens together as a string.
        iii. Pass the string to the Index.
        iv. When the string is being passed to the index, stopwords (which themselves have been stemmed) are removed from the string before it is split into separate tokens and then stored as the value for the given document ID in the *index* dict of the Index.
6. Repeat steps 2-5 for all documents that are identified in the file.
7. Repeat steps 1 through 7 for all other files in the corpus folder.

8. After iterating through all files in the corpus and identifying all documents, the index is successfully built. The **Preprocessor** object will then return the **Index** object that was built, and it will move on to the next step in the IR system.

Indexing

The Indexing process significantly differs based on which base program we use.

Again, all versions of our program for Experiment 1 use the BM25 implementation of our base program; we also included an explanation of the Indexing process for the Cosine Similarity implementation.

IRSystem pickles the Inverted Index created at the end of this Indexing step, so it can loaded later if the program is re-run; this will an optimization step we introduced in Assignment 2, so that Inverted Index will not have to be re-computed every time the program is run.

Cosine Similarity Implementation

This step primarily concerns the functionality of the **InvertedIndex (invertedIndex.py)** and **DocumentVector (documentVector.py)** classes.

IRSystem creates an instance of **InvertedIndex** by providing it with the **Index** returned by **Preprocessor** as a parameter for its constructor. Each instance of **InvertedIndex** contains two instance variables.

One variable is a dict that represents the actual Inverted Index (*invertedIndex*). This variable maps a string which represents a token to an instance of **DocumentVector**.

The instance of **DocumentVector** will represent all information about a token in *invertedIndex*. Each instance of **DocumentVector** contains four instance variables:

1. A dict *(docs)* that keeps track of all documents a particular token is found in and its term frequency in these documents (it maps a Document ID as a string to an integer which represents the token's term frequency in this particular document)
2. An integer *(df)* that represents the document frequency of the token (how many documents it appears in total)
3. A double *(idf)* that represents the inverse document frequency of the token
4. An integer *(max_tf)* that represents the token's highest term frequency across all documents in which it appears.

We iterate over the entire Index, adding an entry to *invertedIndex* for every token in the vocabulary of the entire corpus collection.
For each token in the Index, we track the following information in an instance of **DocumentVector**, which is stored in *invertedIndex*:

1. The documents in which the token appears, as well as the token's term frequency in each of those documents (in the **DocumentVector**'s *docs*).

2. The document frequency (df) of the token (as the **DocumentVector**'s *df*).
3. The inverse document frequency (idf) of the token, which is calculated using the following formula: $\log_2$(79923/the df of the token) (as the **DocumentVector**'s *idf*).
4. The token's highest term frequency across all documents in which it appears (as the **DocumentVector**'s *max_tf*).

InvertedIndex's second instance variable is a dict (*docLengthsNoRoot)* which maps a string (a Document ID) to a Double (the length of the document squared). After we finish building *invertedIndex*, we pass over it to calculate the lengths of each document. The document vector lengths are stored in *docLengthsNoRoot*, with one entry for each document and its length squared. A document vector's length squared is calculated by taking the sum of the squares of its tokens' weights (tf * idf scores).

BM25 Implementation

This step primarily concerns the functionality of the **InvertedIndexBM25 (invertedIndex.py)**.

IRSystem creates an instance of **InvertedIndexBM25** by providing it with the **Index** returned by **Preprocessor** as a parameter for its constructor. Each instance of **InvertedIndexBM25** contains two instance variables: (1) the index that was passed as an argument (*index)* and (2) an instance of **BM25Okapi class** from the **rank_bm25 package** that reads in our document corpus and does some indexing on it (*bm25*).

As a further explanation of the second instance variable, when instantiated, the **InvertedIndexBM25** instance gets a pre-processed, tokenized version of the document corpus (which was created in the Preprocessing step) from the *index*. This corpus is then passed to the BM25Okapi object (*bm25*) to calculate the frequencies of the terms within each document in the corpus.

Retrieval and Ranking

The Retrieval and Ranking process significantly differs based on which base program we use.

Again, all versions of our program for Experiment 1 use the BM25 implementation of our base program; we also included an explanation of the Indexing process for the Cosine Similarity implementation.

Cosine Similarity Implementation

This step primarily concerns the functionality of the **Queries (Queries.py)** and **Retrieval (Retrieval.py)** classes.

Using the Inverted Index and the dict of squared document vector lengths, we calculate the cosine similarity between each document and each of the 50 provided queries.
IRSystem creates an instance of the **Queries** class. Each instance of Queries has the following instance variables: a dict (*queries*) that maps a String (the query's topic ID) to another String

(the query itself), a dict (*stopwords*) that contains stopwords (it is the same dict as *stopwords* in **Preprocessor**), and a dict (*full_query)* that contains entire queries that have not been stemmed. In the **Queries** constructor, we read the .txt file corresponding to the queries (**topics1-50.txt**) and begin the following algorithm:

1. Locate the **topics1-50.txt** file. Begin to read the file.
2. Use regex to identify the queries from the file and to retrieve the desired information.
3. After identifying a query from the file, iterate over its contents, word by word. Strip each word of punctuation and capitalization. Remove any numbers from the query as well. What remains will be the tokens for this query. Store this query in *full_query*.
4. Stem each token in the query using a **Stemmer**.
    a. If the token is not found in Inverted Index (for example, it is a stop word), it will not be used in the cosine similarity calculations between the query and each document.
5. Repeat steps 2-4 for all queries that are identified in the file and store these queries accordingly in *queries*.

In **IRSystem**, we iterate over all queries stored by the instance of **Queries** we previously created. For each query, we then create an instance of the **Retrieval** class, passing it our Inverted Index and an integer (the current key of the current query we are iterating over) as parameters for its constructor. We will use this instance of **Retrieval** to retrieve and rank the information about the similarity between each document and each query.
Each instance of **Retrieval** has five instance variables:

1. An instance of **InvertedIndex** (*invertedIndex*) that represents the Inverted Index
2. A dict (*similarityScores*) that maps a String (a Document ID) to a Double (the document's similarity score to a query)
3. A dict (*queryInfo*) that maps a String (a token from the query) to an Integer (the token's term frequency in the query)
4. A float (*queryLength*) that represents the length of the query
5. A String (*printKey*) which indicates the topic ID of the current query for which similarity scores are being calculated.

As query tokens are processed one by one in the instance of **Retrieval**, we incrementally compute cosine similarity scores between each indexed document and the provided query using the following algorithm:

1. Begin reading the first token from the query. Continue if it is also found in the Inverted Index.
2. We want to keep track of the query's total length. Set *queryLength* to 0 and then add (0.5 + (0.5 * tf of the token in the query) * idf of the token in the Inverted Index) to it.
3. Search the Inverted Index for the token. For each of the documents that it is found in:
    a. Calculate the normalized tf of the token for that document (the tf of the token for that document / the max tf of the token).

b. If the document does not already have an entry in *similarityScores*, add an entry for it to this HashMap and add the product of ((the normalized tf of the token * the token's idf) * ((0.5 + (0.5 * tf of the token in the query) * idf of the token in the Inverted Index)) as the current cosine similarity score for this document.

c. If there is already an entry in *similarityScores* for the document, add the product of ((the normalized tf of the token * the token's idf) * (0.5 + (0.5 * tf of the token in the query)) * idf of the token in the Inverted Index)) to the current cosine similarity score for this document.

4. Repeat Steps 1-3 for all query tokens in the query.
5. After completely iterating through the Inverted Index, divide each entry in *similarityScores* by the product of the square roots of the *queryLength* and the squared length of the document corresponding to the entry in the HashMap (this value can be looked up in HashMap of squared document vector lengths).
6. Sort *similarityScores* in descending order and write its contents to a results file called **results.txt** in the format specified in the assignment outline.

We repeat steps 1-5 for each query stored by the instance of **Queries**.

BM25 Implementation

This step primarily concerns the functionality of the **Queries (queries.py)** and **Retrieval (retrieval.py)** classes.

IRSystem creates an instance of the **Queries** class. Each instance of Queries has two instance variables: a dict (*queries*) that maps a String (the query's topic ID) to another String (the query itself) and a dict (*stopwords*) that contains stopwords (it is the same dict as *stopwords* in **Preprocessor**). In the **Queries** constructor, we read the .txt file corresponding to the queries (**topics1-50.txt**) and begin the following algorithm:

1. Locate the **topics1-50.txt** file. Begin to read the file.
2. Use regex to identify the queries from the file and to retrieve the desired information.
3. After identifying a query from the file, iterate over its contents, word by word. Strip each word of punctuation and capitalization. Remove any numbers from the query as well. What remains will be the tokens for this query.
4. Stem each token in the query using a **Stemmer**.
    a. If the token is not found in Inverted Index (for example, it is a stop word), it will not be used in the cosine similarity calculations between the query and each document.
5. Repeat steps 2-4 for all queries that are identified in the file and store these queries accordingly in *queries*.

In **IRSystem**, we iterate over all queries stored by the instance of **Queries** we previously created. For each query, we then create an instance of the **Retrieval** class, passing it our InvertedIndexBM25 object, and an integer value (the current key of the current query we are iterating over) as parameters for its constructor. We will use this instance of **Retrieval** to retrieve and rank the information about the similarity between each document and each query.

Each instance of **Retrieval** has two instance variables:

1. An instance of **InvertedIndexBM25** (*invertedIndex*) that represents the Inverted Index.
2. A String (*printKey*) which indicates the topic ID of the current query for which similarity scores are being calculated.

As the query comes in, the query string is passed to a function within *invertedIndex* (rank_query_in_bm25) where it will split the query at each word and pass this resulting list to the *bm25* instance variable of *invertedIndex*. This instance variable will call a function (get_scores) on the tokenized query and BM25 similarity scores will be computed between all of document corpus and the query. Scores are then normalized so that they are consistently between 0 and 1. Once the scores are calculated, two lists will be returned to **Retrieval**: (1) the list of scores sorted in descending order and (2) the list of the scores unorganized. The top 1000 scores are then written in descending order to a results file called **results.txt**.

Immediately after a score and its corresponding document ID is written to the results file, the document ID and its corresponding text in the corpus are passed to a **re-ranking** class (the name of the re-ranking class and how it works differs based on the version of the program we are using). The query is also passed to the **re-ranking** class, along with the *printKey*. After **results.txt** is written, we notify the **re-ranking** class to begin the re-ranking process using the provided document IDs and texts, query and *printKey*.

Re-ranking Class

SBERT

When using SBERT, we opted for a pretrained model rather than going through the time-consuming task of training a model on our corpus. SBERT is a language model that generates numerical embeddings for sentences.

The model that we chose to use is 'paraphrase-distilroberta-base-v2,' a pre-trained language model developed by Hugging Face.

The computation of embeddings and vectors is performed by the **SBERT_Retrieval** class (**use_sbert.py**); we will refer to this class as "this retriever" in the rest of our explanation.

An instance of this retriever has seven instance variables:

1. *device*: The variable that indicates where or not models will be running on a CPU or GPU.
2. *model*: The model used to generate embeddings and vectors.
3. *full_queries*: a dictionary of full query texts (a parameter: we retrieve the *full_queries* dict from the **Queries** class and pass it to this retriever).
4. *documentEmbeddings*: The list in which the model's computed document embeddings are stored.

5. *queryEmbeddings*: The list in which the model's computed query embeddings are stored.
6. *documents*: The list in which document contents are stored.
7. *documentIds*: The list in which document IDs are stored; it is used later when writing results to a file.
8. *queryNo*: The current queryNo.
9. *query*: The current query.

We run the following algorithm to first compute embeddings and vectors and then calculate new similarity scores:

1. The **Retrieval** class passes the current query to this retriever (using insertQuery).
    a. The query is then given to *model*, which computes its embedding.
    b. The query embedding is subsequently stored in *queryEmbeddings*.
2. As they are being written to **results.txt**, the **Retrieval** class passes each document ID and its corresponding text in the corpus to this retriever (using insertDoc).
    a. The retriever appends the document text to the *documents* list.
    b. The retriever appends the document ID to the *documentIds* list.
    c. The document text is given to *model*, which computes its embedding.
    d. The document embedding is subsequently stored in *documentEmbeddings*.
3. After all 1000 document IDs have been written to **results.txt** by the **Retrieval** class, the **Retrieval class** calls this retriever's get_similarity function to calculate new similarity scores between the document embeddings and the query embedding.
    a. This retriever converts the query and document embeddings to tensors and saves them to the disk as pickle files.
    b. Next, the tensors are reshaped to 2D arrays to calculate the cosine similarity between the query and each document using the cosine_similarity method from scikit-learn. This retriever normalizes all similarity scores to be between the range of 0 to 1.
    c. The resulting cosine similarities are sorted in descending order, and their indices are stored in *idxs*. This means that *idxs* now contains the indices of the most similar documents to the query (the most similar document being at index 0).
    d. This retriever then writes the new rankings to a results file called **results_experiment1_SBERT.txt**. We are able to map *documentIds* and *cosine_similarities* using each index in *indxs* to determine which document IDs correspond to which similarities.

IRSystem will iterate over all queries, commencing the initial ranking process and the subsequent re-ranking process each time.

Universal Sentence Encoder (USE)

When using USE, we opted for a pretrained model rather than going through the time-consuming task of training a model on our corpus. Universal Sentence Encoder is a pre-trained neural network developed by Google that is capable of producing high-quality fixed-length embeddings for natural language text. The advantage of using Universal Sentence Encoder is the ability to encode sentences and paragraphs into vectors.

The model that we chose to use was found on the tfhub.dev machine learning model page. The link to the model is https://tfhub.dev/google/universal-sentence-encoder/4. The input for this model is a variable length English text and the output is a 512-dimensional vector.

The computation of embeddings and vectors is performed by the **UniversalSentenceEncoderRetrieval** class (**universalSentenceEncoder.py**); we will refer to this class as "this retriever" in the rest of our explanation.

An instance of this retriever has seven instance variables:

1. *use_model*: The model used to generate embeddings and vectors.
2. *documentEmbeddings*: The list in which the model's computed document embeddings are stored.
3. *queryEmbeddings*: The list in which the model's computed query embeddings are stored.
4. *documents*: The list in which document contents are stored.
5. *documentIds*: The list in which document IDs are stored; it is used later when writing results to a file.
6. *queryNo*: The current queryNo.
7. *query*: The current query.

We run the following algorithm to first compute embeddings and vectors and then calculate new similarity scores:

1. The **Retrieval** class passes the current query to this retriever (using insertQuery).
    a. The query is then given to *use_model*, which computes its embedding.
    b. The query embedding is subsequently stored in *queryEmbeddings*.
2. As they are being written to **results.txt**, the **Retrieval** class passes each document ID and its corresponding text in the corpus to this retriever (using insertDoc).
    a. The retriever appends the document text to the *documents* list.
    b. The retriever appends the document ID to the *documentIds* list.
    c. The document text is given to *use_model*, which computes its embedding.
    d. The document embedding is subsequently stored in *documentEmbeddings*.
3. After all 1000 document IDs have been written to **results.txt** by the **Retrieval** class, the **Retrieval class** calls this retriever's get_similarity function to calculate new similarity scores between the document embeddings and the query embedding.
    a. This retriever concatenates the document embeddings along the rows, creating a 2D array where each row corresponds to the embedding for a single document.
    b. Next, this retriever computes the cosine similarity between the query embedding and each document embeddings using the dot product. This results in a 1D array of cosine similarities between the query and each document in the corpus.

    c. The resulting cosine similarities are sorted in descending order, and their indices are stored in *idxs*. This means that *idxs* now contains the indices of the most similar documents to the query (the most similar document being at index 0).
    d. This retriever then writes the new rankings to a results file called **results_experiment1_USE.txt**. We are able to map *documentIds* and *cosine_similarities* using each index in *indxs* to determine which document IDs correspond to which similarities.

IRSystem will iterate over all queries, commencing the initial ranking process and the subsequent re-ranking process each time.

GPT Neo

GPT Neo is an implementation of a model parallel to GPT-2 and GPT-3 style models using the mesh-tensorflow library. This library was created by EleutherAI and is hosted on GitHub provided this link EleutherAI/gpt-neo.

The model used in this experiment was the *EleutherAI/gpt-neo-1.*3B; this model contains 1.3 billion parameters and is about 5.3 Gigs in size.

The computation of embeddings and vectors is performed by the **GPT3Retriever** class (**gpt3Retriever.py**); we will refer to this class as “this retriever” in the rest of our explanation.

An instance of this retriever has nine instance variables:

1. *device*: The variable that indicates where or not models will be running on a CPU or GPU.
2. *model*: The model used to generate embeddings and vectors. It is a pre-trained model that is loaded using the AutoModelForCausalLm.from_pretrained() function.
3. *tokenizer*: A tokenizer that is loaded using the AutoTokenizer.from_pretrained() function.
4. *query*: The current query.
5. *queryNo*: The current queryNo.
6. *queryInputIds*: The list in which the *tokenizer*’s computed query token IDs are stored.
7. *documentIds*: The list in which document IDs are stored; it is used later when writing results to a file.
8. *documents*: The list in which document contents are stored.
9. *documentEmbeddings*: The list in which the model's computed document embeddings are stored.

We run the following algorithm to first compute embeddings and vectors and then calculate new similarity scores:

1. The **Retrieval** class passes the current query to this retriever (using insertQuery).
    a. The query is then given to the *tokenizer*, which computes its token IDs.
    b. The query token IDs are subsequently stored in *queryInputIds*.
2. As they are being written to **results.txt**, the **Retrieval** class passes each document ID and its corresponding text in the corpus to this retriever (using insertDocument).
    a. The retriever appends the document text to the *documents* list.

   b. The retriever appends the documend ID to the *documentIds* list.
   c. The document text is given to the *tokenizer*, which computes its token IDs. Note that the document text is truncated to the maximum length that can be inputted to the *tokenizer* (if its length exceeds this maximum).
   d. The token IDs are subsequently stored in *docInputIds*.
   e. The token IDs are then fed into the *model* to generate its hidden states. These hidden states are then used to compute a mean embedding of the document. The embedding is normalized and stored in *documentEmbeddings*.

3. After all 1000 document IDs have been written to **results.txt** by the **Retrieval** class, the **Retrieval class** calls this retriever's retrieve function to calculate new similarity scores between the document and queries.
   a. This retriever generates the hidden states of the query token IDs from *queryInputIds*, and then feeds these token IDs to the *model*, which generates the hidden states of this query input and computes its mean embedding.
   b. Next, this retriever computes the cosine similarity between the query embedding and each document embedding using the cosine_simularity function for scikit-learn.
   c. The resulting cosine similarities are sorted in descending order.
   d. This retriever then writes the new rankings to a results file called **results_experiment1_GPTNeo.txt**. We are able to map *documentIds* to *cosine_similarities* using the *documentIds*.

IRSystem will iterate over all queries, commencing the initial ranking process and the subsequent re-ranking process each time.

Experiment 2

Another way to improve system performance lies not in changing the system itself (i.e., how the documents are pre-processed and ranked), but to modify queries to suit the system better. Think of how you would ask a question in France. Would you ask your question in English, or would you translate your question first?

This is the base idea behind this experiment. Transforming queries "behind the scenes" could lead to an improvement in system performance by suiting the query to the system. Therefore, we perform three sub-experiments in query expansion to see if this is actually the case.

In the first experiment, we modify queries using a Word2Vec model from Gensim trained on Google News reports. For each query, synonyms are selectively added based on their similarity as evaluated by Word2Vec. Performance for this experiment was evaluated solely against BM25. As such, the basis of this program is the base BM25 program (as described in the Experiment 1 section of "Program functionality and algorithms, data structures, and optimizations used in each of the experiments").

In our latter three experiments, we evaluate the performance of some query expansion techniques against a different versions of the system: base Cosine Similarity, base BM25 and BM25 + USE. This leverages development that was being done at the same time on Experiment 1, as these programs were built on the base Cosine Similarity program, base BM25 program and BM25 + USE program respectively (all of which are described in the Experiment 1 section of "Program functionality and algorithms, data structures, and optimizations used in each of the experiments").
These further experiments were important so that (1) we could evaluate query expansion on at least one neural model *and* (2) we could see if these techniques were consistently performant or not. In these two experiments, we modify queries using the following strategies:

1. Re-write queries using Hugging Face's pre-trained 'facebook/bart-large-xsum' model.
2. Expand queries to include all synonyms using NLTK. Compared to Word2Vec, this is a more naïve approach to query expansion in the sense that query tokens are expanded to include partial synonyms; no similarity scores are used to evaluate if a synonym is a "close match" or not.

Experiment 3

Experiment 3 describes an experiment where the entire IR system from Assignment 1 is redone using a neural retrieval method.

Sent2Vec

The basis of this program is the base Cosine Similarity program (as described in the Experiment 1 section of "Program functionality and algorithms, data structures, and optimizations used in each of the experiments").

While the program still builds an Index and an Inverted Index, it no longer uses them in any significant capacity to compute ranking scores. The **Retrieval** class is the area where most updates were made. The class has three new instance variables: *embeddings_computed* (a bool that indicates whether the document embeddings have already been computed), *embeddings* (the variable that stores the document embeddings), and *faiss_index*, an index created using the FAISS library that allows for cosine similarity search between a set of documents and a query.

The calculateSimilarity method of **Retrieval** has been updated to load the "wiki_unigrams.bin" Sent2Vec model and the list of full document texts from the Inverted Index. If document embeddings have not yet been computed, embeddings will created for each of the full document texts and save them to the disk as a pickle file. If the document embeddings have already been computed, they are loaded upon instantiation of the **Retrieval** class. The document embeddings are then normalized and **Retrieval** creates an IndexFlatIP index on these embeddings using the FAISS library to allow for efficient cosine similarity search (the index is saved to *faiss_index*). An embedding is then computed for the current query provided by **IRSystem**, which itself is normalized. **Retrieval** uses *faiss_index* to search for the 1000 most similar documents to the query and returns their IDs and similarity scores. The results are then sorted in descending order and written to a results file, **results_experiment_sent2vec.txt**.

SBERT

The basis of this program is the BM25 and SBERT program developed in Experiment 1 (as described in the Experiment 1 section of “Program functionality and algorithms, data structures, and optimizations used in each of the experiments”). This program makes use of **SBERT_Retrieval (use_sbert.py)**, which is referred to as “this retriever” for the rest of this explanation.

The main difference is the **Retrieval** class. It still inserts the query in this retriever, but the calculateSimilarity method has been updated so that it no longer computes BM25 similarity scores between the documents and a given query. Instead, once the method is called, it inserts all full document texts from the Inverted Index into this retriever one-by-one. After a full document text is passed to this retriever, it has its embedding computed – again, see the Experiment 1 section of “Program functionality and algorithms, data structures, and optimizations used in each of the experiments” for more details). Once all documents have been passed to this retriever, its get_similarity method is called, and the cosine similarities between the all the document embeddings and the query embedding are computed.

Also note that this retriever now saves all document embeddings in a pickle file after they are compared with the first query; this means that for subsequent queries, the document embeddings are loaded and not re-computed. This was an optimization step that we introduced to save computation time.

How to run the program

1. Please ensure that you have Python installed. The program was executed using Python 3.10.1
2. Please ensure that you have pip installed. All experiments require the installation of additional modules that are not installed by default with Python 3.10.1; these modules should install automatically (if they are not already installed) using pip. All additional modules were installed using pip 22.3.1.
3. Clone the repository from the GitHub link provided at the Brightspace submission.
4. Navigate to the directory of your choice depending on the experiment you are looking at.
5. Run the irSystem.py file (use the command: python irSystem.py).
    a. Experiment 3 (Sent2Vec) requires a manual installation of a Sent2Vec model; the link to the model is provided in a .txt file (**link_to_model.txt**). Download this model and add it to directory with the rest of files required for this experiment.

Results

Data

We evaluated the performance of eight different versions of our program.

Base Versions of our Program (Assignment 1)

We had two base versions of our program (neither used neural methods) that completed IR tasks according to the requirements of Assignment 1: one used cosine similarity to rank documents, and the other used BM25 to rank documents.

Note that these base versions of our program are Python translations of the Java program we submitted for Assignment 1. We had compatibility issues implementing neural methods with our Java code, so we used Python instead. Although there is a slight difference in MAP score, the Cosine Similarity base version of the program is comparable to the code we submitted for Assignment 1. We attribute this change in MAP score (from 0.1507 to 0.1468) mainly to the differences between Java and Python, such as how the Java code used Double values in its calculations and Doubles do not exist in Python.

Ranking Method	MAP Score	P@10
BM25	0.3789	0.4767
Cosine Similarity	0.1468	0.2100

Experiment 1

For Experiment 1, we built on the base BM25 version of our program since the BM25 ranking produced higher MAP scores than the cosine similarity ranking. We re-ranked documents using pre-trained Universal Sentence Encoder (USE), SBERT and GPTNeo models.

Program Version	MAP Score	P@10
BM25 + USE	0.2339	0.2820
BM25 + SBERT	0.2605	0.4040
BM25 + GPTNeo	0.1123	0.1560

Experiment 2

In Experiment 2, we evaluated the performance of BM25 and BM25+USE using a variety of query expansion techniques described in the previous section. The results of these experiments are detailed in the table below.

Program Version	MAP Score	P@10
BM25 (no stopword removal)	0.3605	0.4540
BM25 and Word2Vec	0.3407	0.4440
Base Program (Cosine Similarity) + Hugging Face query re-writes	0.0944	0.1340
BM25 and NLTK synonymizing	0.0943	0.1580
BM25 + USE and NLTK synonymizing	0.0312	0.0500

Experiment 3

For Experiment 3, we built versions of our program that did not use the traditional Index and Inverted Index structure of either of our base programs; instead, they computed vectorized embeddings of all documents and compared them to vectorized embeddings of all queries. We used pre-trained sent2vec, SBERT and GPTNeo models for our implementations of Experiment 3. All vectorized document embeddings were compared to vectorized query embeddings using cosine similarity.

Program Version	MAP Score	P@10
sent2vec and Cosine Similarity	0.2280	0.3520
SBERT and Cosine Similarity	0.2076	0.3760

All Experiments

The following is a comparison of MAP Score performances of each version of our program across all experiments, as well as the base versions of our program. Each program version is listed in descending order of its MAP Score performance:

Experiment	Program Version	MAP Score
Base Program	BM25	0.3789
2	BM25 (no stopword removal)	0.3605
2	BM25 and word2vec	0.3407
1	BM25 + USE	0.2339
1	BM25 + SBERT	0.2605
3	sent2vec and Cosine Similarity	0.2280
3	SBERT and Cosine Similarity	0.2076
Base Program	Cosine Similarity	0.1468
1	BM25 + GPTNeo	0.1123
2	Base Program (Cosine Similarity) + Hugging Face query re-writes	0.0944
2	BM25 and NLTK synonymizing	0.0943
2	BM25 + USE and NLTK synonymizing	0.0312

Discussion

None of the versions of our program that we created for Experiments 1, 2 and 3 achieved higher MAP scores than the base BM25 version of our program. In fact, they all achieved lower MAP scores.

Experiment 1 and Experiment 3

The following are possible reasons why our programs for Experiment 1 and Experiment 3 produced poorer MAP scores:

1. The pre-trained models used in these experiments are not well-suited for this specific retrieval task using our IR system.

   All the models we used in these experiments were pre-trained. In other words, we did not train the models ourselves for the specific purpose of performing retrieval tasks on our document corpus. So, if the documents and queries in our document corpus are significantly different from the data used to train these pre-trained models, so the embeddings and corresponding vectors produced by these models may not be well-suited for the documents and queries in our document corpus. While we attempted to use models that were trained on datasets that would theoretically be similar to our document corpus (for example, the sent2vec model was trained on a corpus of articles), these models were not trained with intention of being used to retrieve information from our particular corpus. As a result, the pre-trained models may not be producing accurate rankings, leading to our poorer MAP scores.

GPTNeo is also a language model that was primarily designed for text generation tasks, rather than creating text embeddings. Although it can create contextualized word embeddings, it does not specialize in this task; this may explain GPTNeo's worst ranking out of all neural methods, as the other methods used are more specifically tailored to creating embeddings.

2. The similarity metric to compare the vectorized embeddings was insufficient.

   During the re-ranking phase, we computed similarity scores between the vectorized query embeddings and the vectorized document embeddings using cosine similarity. Other similarity metrics may produce more accurate similarity scores, which would improve the accuracy of the re-ranking.

A potential reason why the programs for Experiment 1 had poorer MAP scores:

3. There is a mismatch between the pre-trained models and the BM25 approach used for our initial rankings.

   BM25 makes use of a statistical approach that is based on document length and term frequency, whereas our pre-trained models for USE, SBERT and GPTNeo, rely on contextual embeddings and vectors. These different ranking approaches may result in conflicting signals when it comes time to rank documents, resulting in poorer MAP scores; in other words, the embeddings and vectors produced by the pre-trained models may not be optimized to consider the specific features that were used by BM25 to produce the initial rankings.

Based on the results obtained from these experiments, we think that a possible way to improve our programs for Experiments 1 and 3 would be not to use pre-trained models but models that are specifically trained for our retrieval task. This would most likely require another corpus similar to the existing document corpus that we could use to train the models. If we were to train the models directly on our existing document corpus, then we run the risk of overfitting our models to the corpus. Since we would be computing embeddings on the same corpus that we trained the models on, we are technically not testing the true predictive ability of the models on unseen data. So, we would theoretically need a corpus that is similar to our current document corpus that we can use to train the models. We would want to train the models on this corpus to still capture specific features and nuances needed to create accurate embeddings but truly test the capabilities of the models. It would not make sense to split the current document corpus into separate datasets for training and for testing since we need to compute embeddings for the entire corpus for our specific retrieval task.

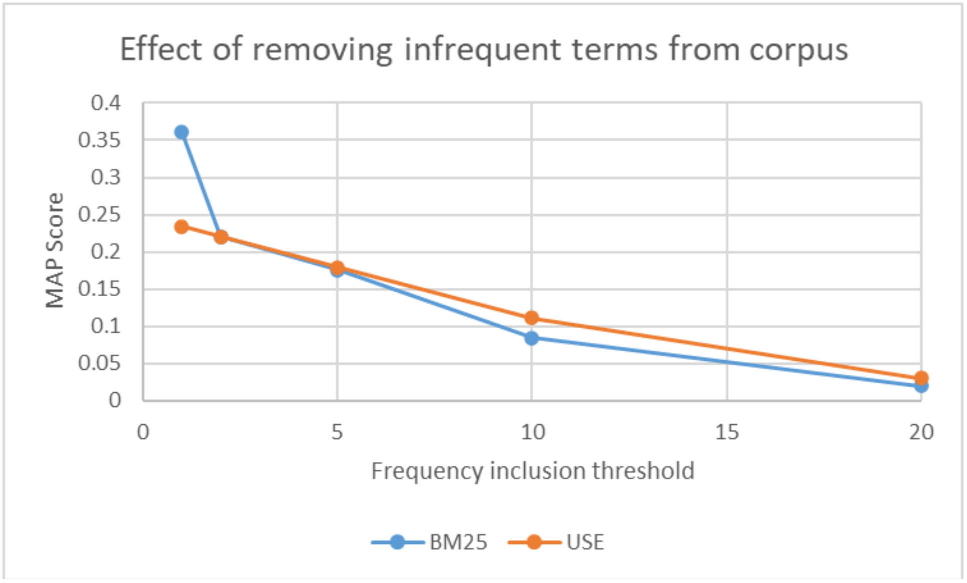
We could also experiment with different similarity metrics other than cosine similarity to compare the vectorized embeddings; we may potentially find a similarity metric that computes more accurate similarity scores between the vectorized document and query embeddings than cosine similarity does.

Experiment 2

Query expansion was not as useful as we had hoped, leading to either a moderate (BM25 and Word2Vec) or major reduction (BM25+USE and NLTK synonymizing) in performance across all sub-experiments.

Seeing as Word2Vec's discriminatory inclusion of related terms led to the smallest reduction in performance, we hypothesize that query expansion can decrease performance by adding noise to the query. Synonyms can be classified into three groups (absolute, partial, near) based on how similar they are to a word, so indiscriminately adding them like we tried with NLTK could have added synonyms that are too unlike the words in the original queries. Different synonyms might also carry different connotations. "Canine" and "doggo" *technically* mean the same thing, but the former is more likely to appear in a veterinarian science journal.

Using that same line of reasoning, what if we take a step away from query expansion and try to disambiguate our corpus index by removing infrequent terms? The idea here is that infrequent terms are more likely to be noise, so removing them would improve our rankings. In an ad-hoc experiment spurred by this line of questioning, we designed one last sub-experiment to see if this could be effective. Using BM25 and BM25+USE, we designed a system that would only index words from our corpus which met or exceeded a minimum frequency threshold. The MAP scores given by different frequency thresholds are given in the chart below.



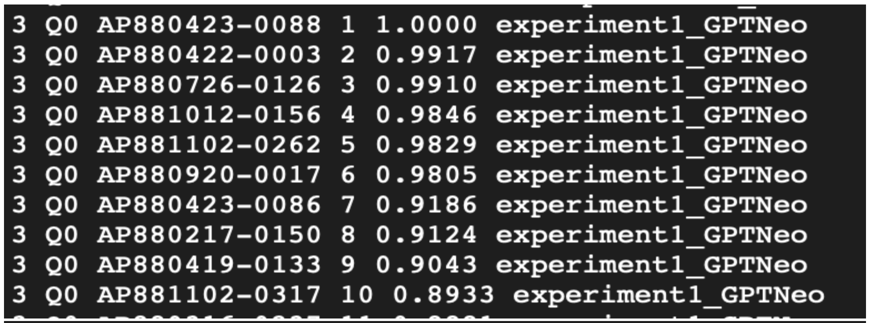
Unfortunately, even this made the MAP scores of these systems worse. In fact, the higher the inclusion threshold, the worse the MAP score (suggesting that any benefit of excluding noise is offset by the cost incurred by excluding terms that aren't noise.)

How should these combined results be interpreted? First, it is important to note that the performance of query expansion depends on both the corpus being searched and the architecture of the IR system. In the ad-hoc experiment above, removing infrequent terms from the corpus impacted BM25 less for a smaller frequency inclusion threshold than for BM25+USE. Secondly, query expansion can be paradoxically interpreted as introducing ambiguity. This could be desired for some systems, but it does not appear to be the case for ours across any of the implementations of the IR system that we evaluated.

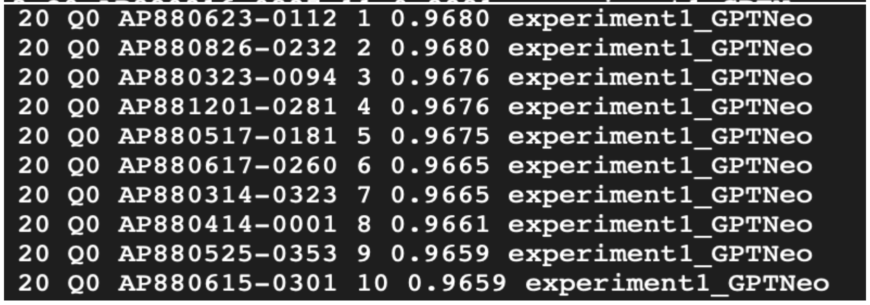
Top 10 Results for Queries 3 and 20

Experiment 1

BM25 + GPTNeo

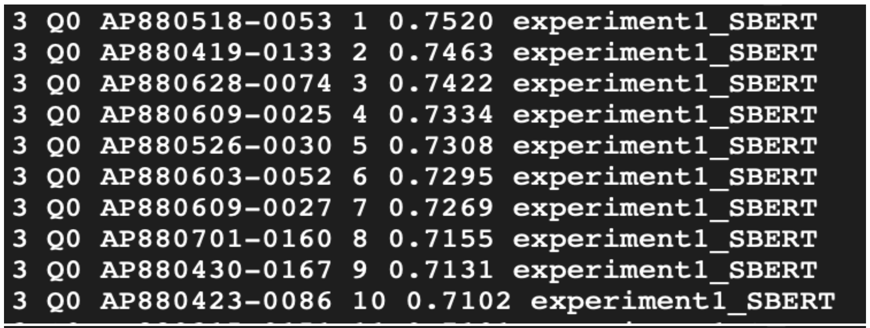


```
3 Q0 AP880423-0088 1 1.0000 experiment1_GPTNeo
3 Q0 AP880422-0003 2 0.9917 experiment1_GPTNeo
3 Q0 AP880726-0126 3 0.9910 experiment1_GPTNeo
3 Q0 AP881012-0156 4 0.9846 experiment1_GPTNeo
3 Q0 AP881102-0262 5 0.9829 experiment1_GPTNeo
3 Q0 AP880920-0017 6 0.9805 experiment1_GPTNeo
3 Q0 AP880423-0086 7 0.9186 experiment1_GPTNeo
3 Q0 AP880217-0150 8 0.9124 experiment1_GPTNeo
3 Q0 AP880419-0133 9 0.9043 experiment1_GPTNeo
3 Q0 AP881102-0317 10 0.8933 experiment1_GPTNeo
```

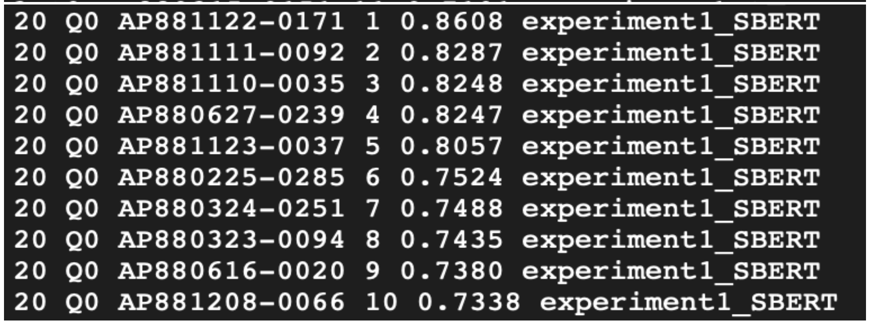


```
20 Q0 AP880623-0112 1 0.9680 experiment1_GPTNeo
20 Q0 AP880826-0232 2 0.9680 experiment1_GPTNeo
20 Q0 AP880323-0094 3 0.9676 experiment1_GPTNeo
20 Q0 AP881201-0281 4 0.9676 experiment1_GPTNeo
20 Q0 AP880517-0181 5 0.9675 experiment1_GPTNeo
20 Q0 AP880617-0260 6 0.9665 experiment1_GPTNeo
20 Q0 AP880314-0323 7 0.9665 experiment1_GPTNeo
20 Q0 AP880414-0001 8 0.9661 experiment1_GPTNeo
20 Q0 AP880525-0353 9 0.9659 experiment1_GPTNeo
20 Q0 AP880615-0301 10 0.9659 experiment1_GPTNeo
```

BM25 + SBERT

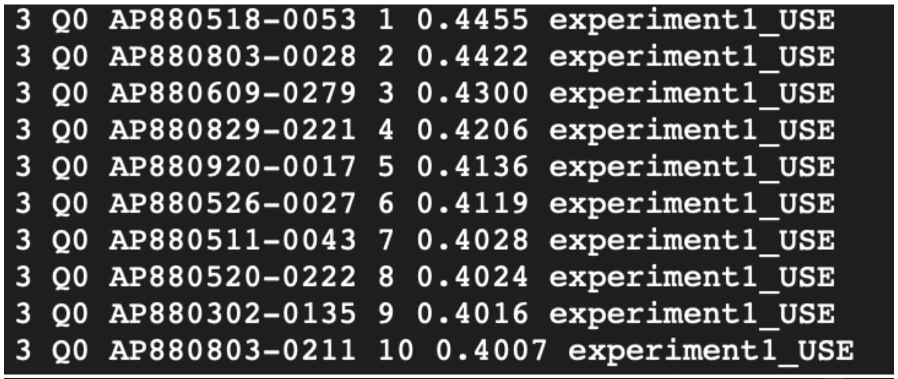


```
3 Q0 AP880518-0053 1 0.7520 experiment1_SBERT
3 Q0 AP880419-0133 2 0.7463 experiment1_SBERT
3 Q0 AP880628-0074 3 0.7422 experiment1_SBERT
3 Q0 AP880609-0025 4 0.7334 experiment1_SBERT
3 Q0 AP880526-0030 5 0.7308 experiment1_SBERT
3 Q0 AP880603-0052 6 0.7295 experiment1_SBERT
3 Q0 AP880609-0027 7 0.7269 experiment1_SBERT
3 Q0 AP880701-0160 8 0.7155 experiment1_SBERT
3 Q0 AP880430-0167 9 0.7131 experiment1_SBERT
3 Q0 AP880423-0086 10 0.7102 experiment1_SBERT
```

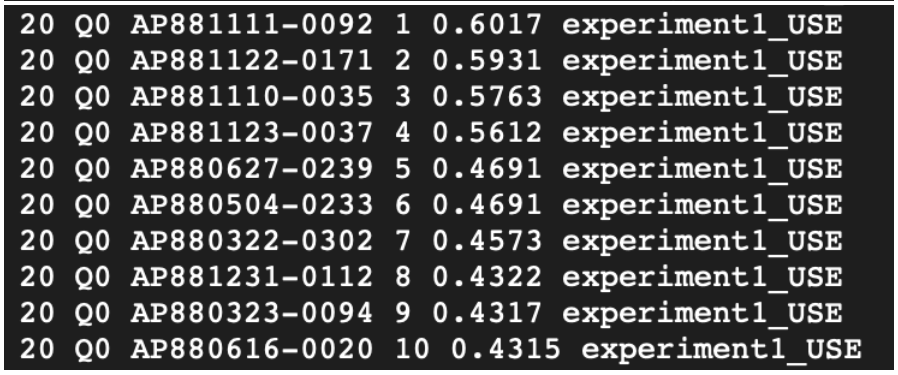


```
20 Q0 AP881122-0171 1 0.8608 experiment1_SBERT
20 Q0 AP881111-0092 2 0.8287 experiment1_SBERT
20 Q0 AP881110-0035 3 0.8248 experiment1_SBERT
20 Q0 AP880627-0239 4 0.8247 experiment1_SBERT
20 Q0 AP881123-0037 5 0.8057 experiment1_SBERT
20 Q0 AP880225-0285 6 0.7524 experiment1_SBERT
20 Q0 AP880324-0251 7 0.7488 experiment1_SBERT
20 Q0 AP880323-0094 8 0.7435 experiment1_SBERT
20 Q0 AP880616-0020 9 0.7380 experiment1_SBERT
20 Q0 AP881208-0066 10 0.7338 experiment1_SBERT
```

BM25 + USE



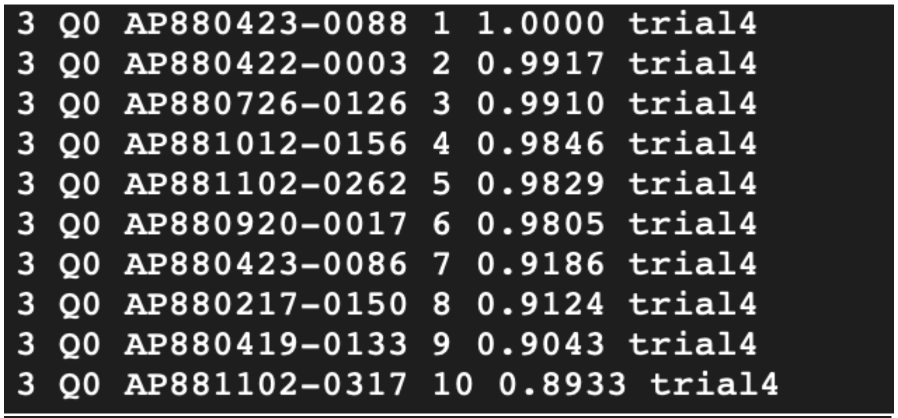
```
3 Q0 AP880518-0053 1 0.4455 experiment1_USE
3 Q0 AP880803-0028 2 0.4422 experiment1_USE
3 Q0 AP880609-0279 3 0.4300 experiment1_USE
3 Q0 AP880829-0221 4 0.4206 experiment1_USE
3 Q0 AP880920-0017 5 0.4136 experiment1_USE
3 Q0 AP880526-0027 6 0.4119 experiment1_USE
3 Q0 AP880511-0043 7 0.4028 experiment1_USE
3 Q0 AP880520-0222 8 0.4024 experiment1_USE
3 Q0 AP880302-0135 9 0.4016 experiment1_USE
3 Q0 AP880803-0211 10 0.4007 experiment1_USE
```



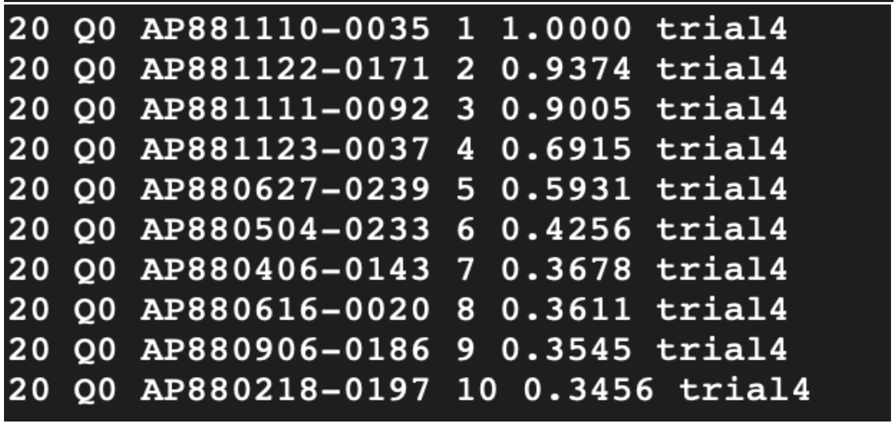
```
20 Q0 AP881111-0092 1 0.6017 experiment1_USE
20 Q0 AP881122-0171 2 0.5931 experiment1_USE
20 Q0 AP881110-0035 3 0.5763 experiment1_USE
20 Q0 AP881123-0037 4 0.5612 experiment1_USE
20 Q0 AP880627-0239 5 0.4691 experiment1_USE
20 Q0 AP880504-0233 6 0.4691 experiment1_USE
20 Q0 AP880322-0302 7 0.4573 experiment1_USE
20 Q0 AP881231-0112 8 0.4322 experiment1_USE
20 Q0 AP880323-0094 9 0.4317 experiment1_USE
20 Q0 AP880616-0020 10 0.4315 experiment1_USE
```

Experiment 2

BM25 (No Stopword Removal)

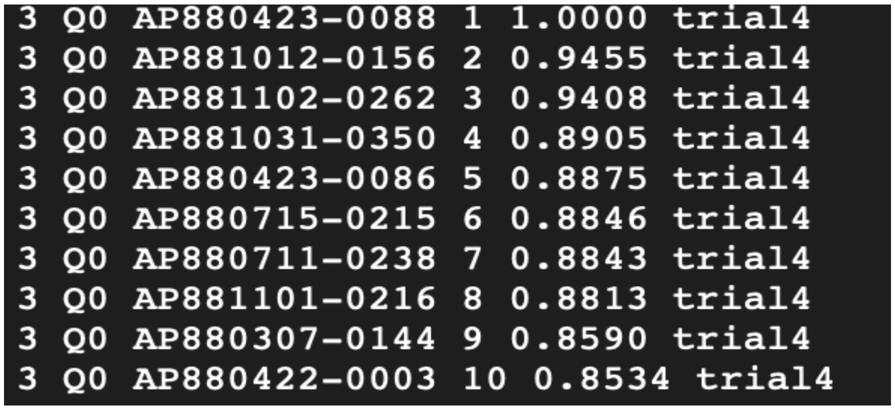


```
3 Q0 AP880423-0088 1 1.0000 trial4
3 Q0 AP880422-0003 2 0.9917 trial4
3 Q0 AP880726-0126 3 0.9910 trial4
3 Q0 AP881012-0156 4 0.9846 trial4
3 Q0 AP881102-0262 5 0.9829 trial4
3 Q0 AP880920-0017 6 0.9805 trial4
3 Q0 AP880423-0086 7 0.9186 trial4
3 Q0 AP880217-0150 8 0.9124 trial4
3 Q0 AP880419-0133 9 0.9043 trial4
3 Q0 AP881102-0317 10 0.8933 trial4
```

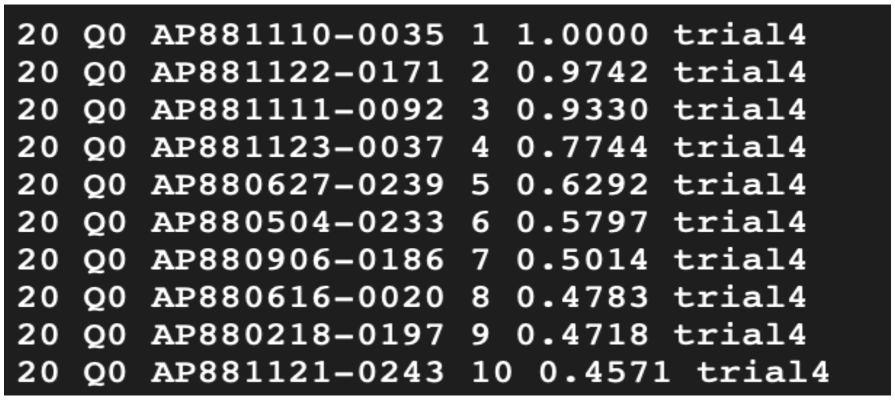


```
20 Q0 AP881110-0035 1 1.0000 trial4
20 Q0 AP881122-0171 2 0.9374 trial4
20 Q0 AP881111-0092 3 0.9005 trial4
20 Q0 AP881123-0037 4 0.6915 trial4
20 Q0 AP880627-0239 5 0.5931 trial4
20 Q0 AP880504-0233 6 0.4256 trial4
20 Q0 AP880406-0143 7 0.3678 trial4
20 Q0 AP880616-0020 8 0.3611 trial4
20 Q0 AP880906-0186 9 0.3545 trial4
20 Q0 AP880218-0197 10 0.3456 trial4
```

BM25 + Word2Vec

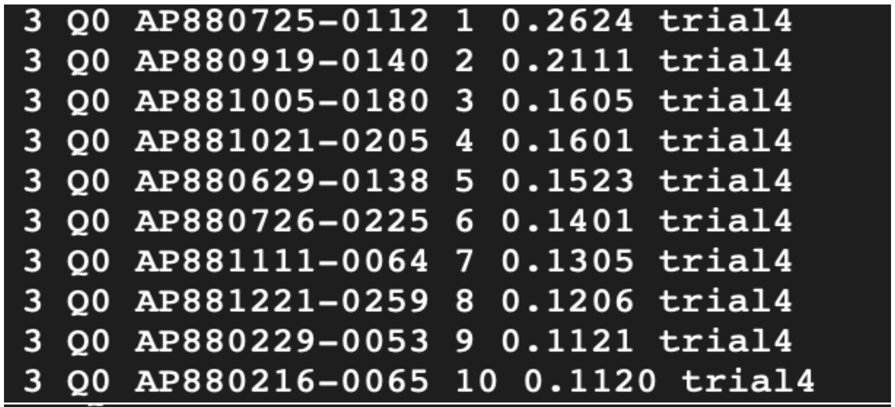


```
3 Q0 AP880423-0088 1 1.0000 trial4
3 Q0 AP881012-0156 2 0.9455 trial4
3 Q0 AP881102-0262 3 0.9408 trial4
3 Q0 AP881031-0350 4 0.8905 trial4
3 Q0 AP880423-0086 5 0.8875 trial4
3 Q0 AP880715-0215 6 0.8846 trial4
3 Q0 AP880711-0238 7 0.8843 trial4
3 Q0 AP881101-0216 8 0.8813 trial4
3 Q0 AP880307-0144 9 0.8590 trial4
3 Q0 AP880422-0003 10 0.8534 trial4
```

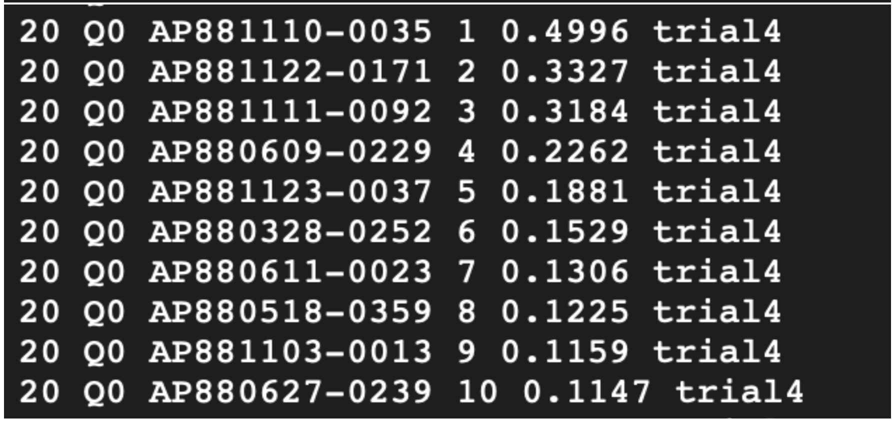


```
20 Q0 AP881110-0035 1 1.0000 trial4
20 Q0 AP881122-0171 2 0.9742 trial4
20 Q0 AP881111-0092 3 0.9330 trial4
20 Q0 AP881123-0037 4 0.7744 trial4
20 Q0 AP880627-0239 5 0.6292 trial4
20 Q0 AP880504-0233 6 0.5797 trial4
20 Q0 AP880906-0186 7 0.5014 trial4
20 Q0 AP880616-0020 8 0.4783 trial4
20 Q0 AP880218-0197 9 0.4718 trial4
20 Q0 AP881121-0243 10 0.4571 trial4
```

Base Program (Cosine Similarity) and Hugging Face Query Re-Writes

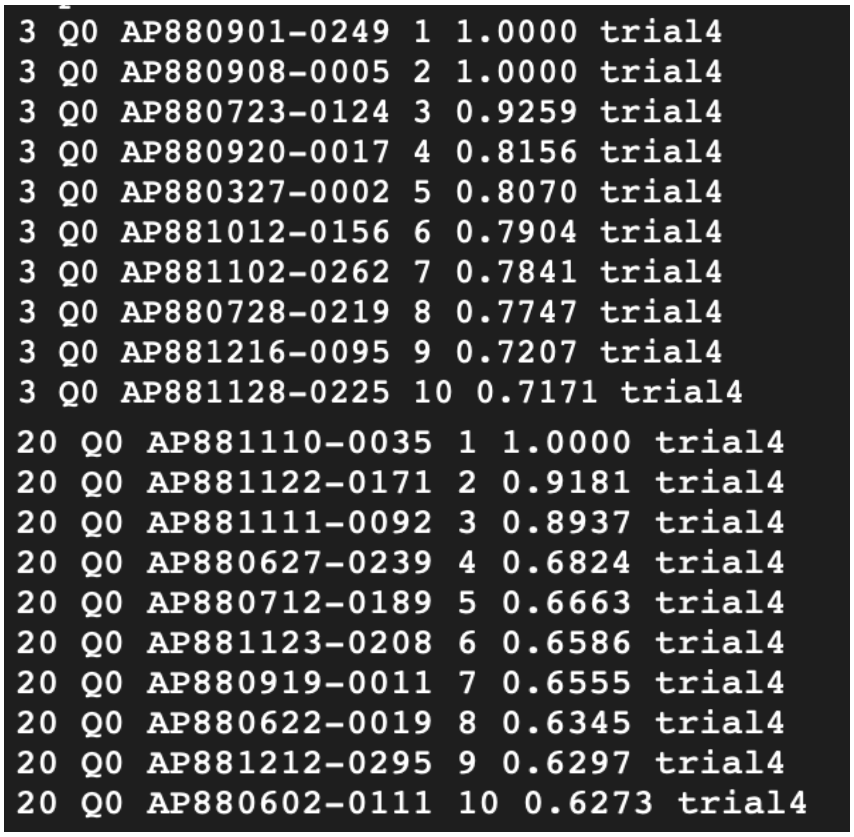


```
3 Q0 AP880725-0112 1 0.2624 trial4
3 Q0 AP880919-0140 2 0.2111 trial4
3 Q0 AP881005-0180 3 0.1605 trial4
3 Q0 AP881021-0205 4 0.1601 trial4
3 Q0 AP880629-0138 5 0.1523 trial4
3 Q0 AP880726-0225 6 0.1401 trial4
3 Q0 AP881111-0064 7 0.1305 trial4
3 Q0 AP881221-0259 8 0.1206 trial4
3 Q0 AP880229-0053 9 0.1121 trial4
3 Q0 AP880216-0065 10 0.1120 trial4
```



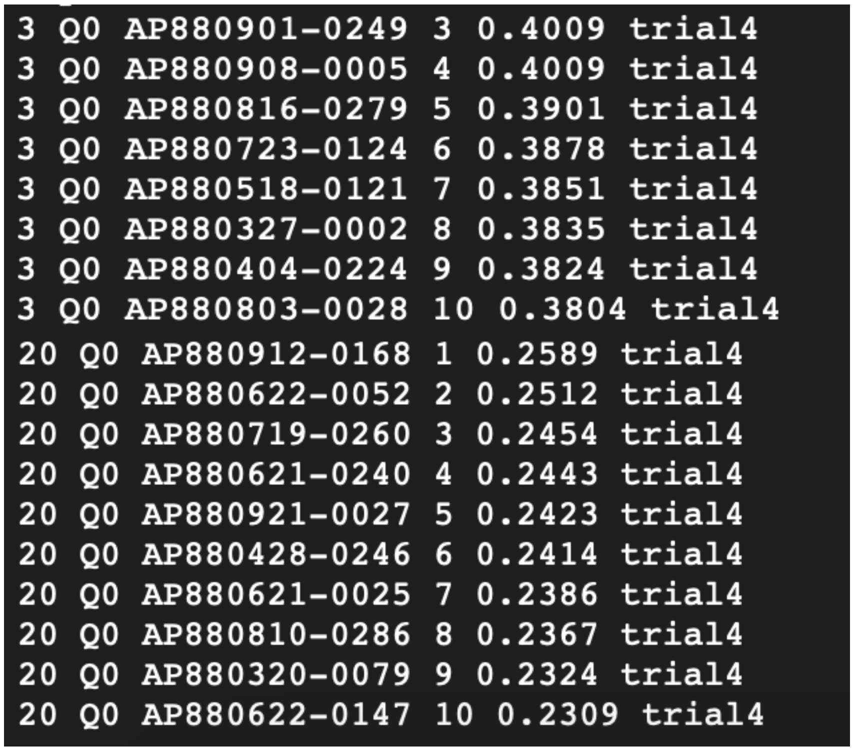
```
20 Q0 AP881110-0035 1 0.4996 trial4
20 Q0 AP881122-0171 2 0.3327 trial4
20 Q0 AP881111-0092 3 0.3184 trial4
20 Q0 AP880609-0229 4 0.2262 trial4
20 Q0 AP881123-0037 5 0.1881 trial4
20 Q0 AP880328-0252 6 0.1529 trial4
20 Q0 AP880611-0023 7 0.1306 trial4
20 Q0 AP880518-0359 8 0.1225 trial4
20 Q0 AP881103-0013 9 0.1159 trial4
20 Q0 AP880627-0239 10 0.1147 trial4
```

BM25 and NLTK Synonymizing



```
3 Q0 AP880901-0249 1 1.0000 trial4
3 Q0 AP880908-0005 2 1.0000 trial4
3 Q0 AP880723-0124 3 0.9259 trial4
3 Q0 AP880920-0017 4 0.8156 trial4
3 Q0 AP880327-0002 5 0.8070 trial4
3 Q0 AP881012-0156 6 0.7904 trial4
3 Q0 AP881102-0262 7 0.7841 trial4
3 Q0 AP880728-0219 8 0.7747 trial4
3 Q0 AP881216-0095 9 0.7207 trial4
3 Q0 AP881128-0225 10 0.7171 trial4
20 Q0 AP881110-0035 1 1.0000 trial4
20 Q0 AP881122-0171 2 0.9181 trial4
20 Q0 AP881111-0092 3 0.8937 trial4
20 Q0 AP880627-0239 4 0.6824 trial4
20 Q0 AP880712-0189 5 0.6663 trial4
20 Q0 AP881123-0208 6 0.6586 trial4
20 Q0 AP880919-0011 7 0.6555 trial4
20 Q0 AP880622-0019 8 0.6345 trial4
20 Q0 AP881212-0295 9 0.6297 trial4
20 Q0 AP880602-0111 10 0.6273 trial4
```

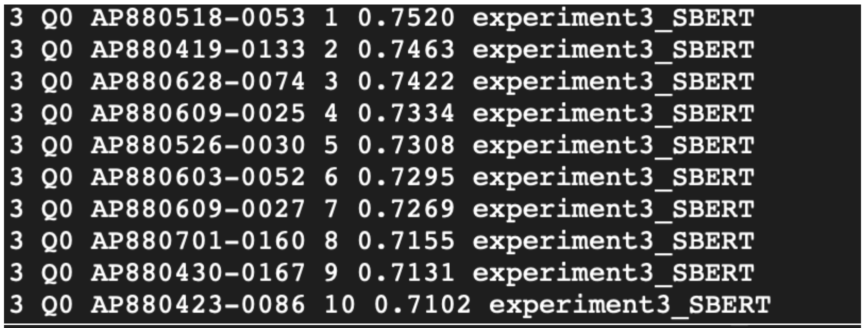
BM25 + USE and NLTK Synonymizing



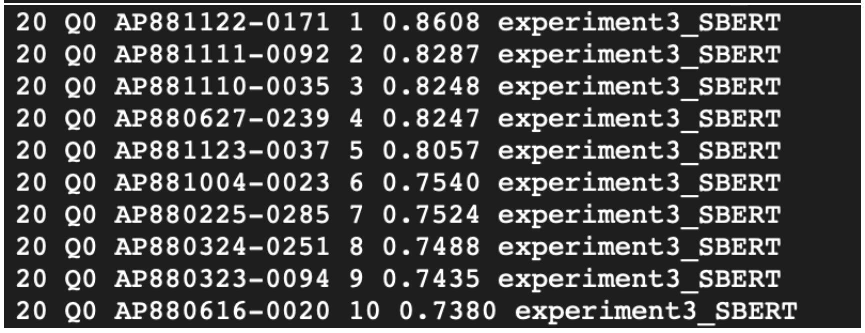
```
3 Q0 AP880901-0249 3 0.4009 trial4
3 Q0 AP880908-0005 4 0.4009 trial4
3 Q0 AP880816-0279 5 0.3901 trial4
3 Q0 AP880723-0124 6 0.3878 trial4
3 Q0 AP880518-0121 7 0.3851 trial4
3 Q0 AP880327-0002 8 0.3835 trial4
3 Q0 AP880404-0224 9 0.3824 trial4
3 Q0 AP880803-0028 10 0.3804 trial4
20 Q0 AP880912-0168 1 0.2589 trial4
20 Q0 AP880622-0052 2 0.2512 trial4
20 Q0 AP880719-0260 3 0.2454 trial4
20 Q0 AP880621-0240 4 0.2443 trial4
20 Q0 AP880921-0027 5 0.2423 trial4
20 Q0 AP880428-0246 6 0.2414 trial4
20 Q0 AP880621-0025 7 0.2386 trial4
20 Q0 AP880810-0286 8 0.2367 trial4
20 Q0 AP880320-0079 9 0.2324 trial4
20 Q0 AP880622-0147 10 0.2309 trial4
```

Experiment 3

SBERT

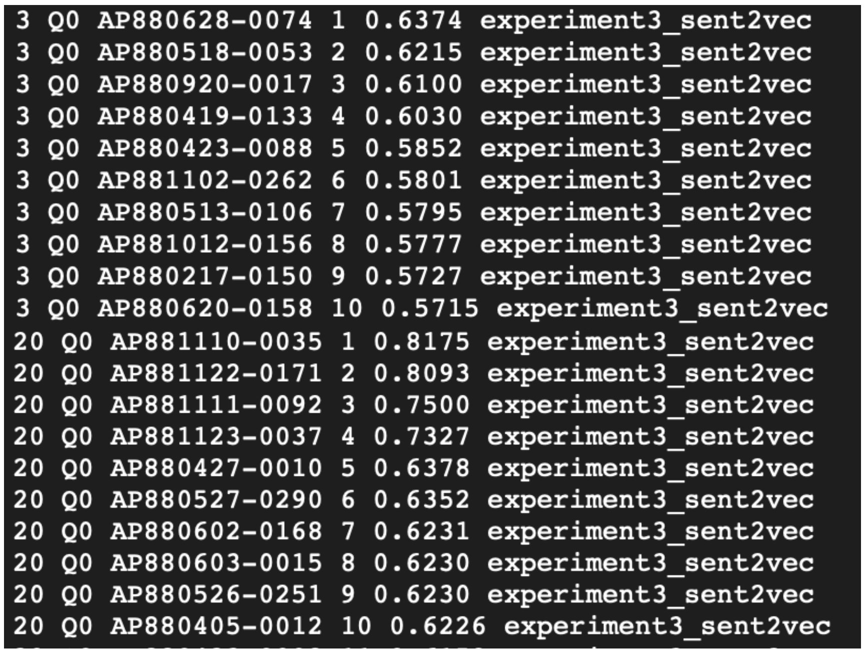


```
3 Q0 AP880518-0053 1 0.7520 experiment3_SBERT
3 Q0 AP880419-0133 2 0.7463 experiment3_SBERT
3 Q0 AP880628-0074 3 0.7422 experiment3_SBERT
3 Q0 AP880609-0025 4 0.7334 experiment3_SBERT
3 Q0 AP880526-0030 5 0.7308 experiment3_SBERT
3 Q0 AP880603-0052 6 0.7295 experiment3_SBERT
3 Q0 AP880609-0027 7 0.7269 experiment3_SBERT
3 Q0 AP880701-0160 8 0.7155 experiment3_SBERT
3 Q0 AP880430-0167 9 0.7131 experiment3_SBERT
3 Q0 AP880423-0086 10 0.7102 experiment3_SBERT
```



```
20 Q0 AP881122-0171 1 0.8608 experiment3_SBERT
20 Q0 AP881111-0092 2 0.8287 experiment3_SBERT
20 Q0 AP881110-0035 3 0.8248 experiment3_SBERT
20 Q0 AP880627-0239 4 0.8247 experiment3_SBERT
20 Q0 AP881123-0037 5 0.8057 experiment3_SBERT
20 Q0 AP881004-0023 6 0.7540 experiment3_SBERT
20 Q0 AP880225-0285 7 0.7524 experiment3_SBERT
20 Q0 AP880324-0251 8 0.7488 experiment3_SBERT
20 Q0 AP880323-0094 9 0.7435 experiment3_SBERT
20 Q0 AP880616-0020 10 0.7380 experiment3_SBERT
```

Sent2Vec



```
3 Q0 AP880628-0074 1 0.6374 experiment3_sent2vec
3 Q0 AP880518-0053 2 0.6215 experiment3_sent2vec
3 Q0 AP880920-0017 3 0.6100 experiment3_sent2vec
3 Q0 AP880419-0133 4 0.6030 experiment3_sent2vec
3 Q0 AP880423-0088 5 0.5852 experiment3_sent2vec
3 Q0 AP881102-0262 6 0.5801 experiment3_sent2vec
3 Q0 AP880513-0106 7 0.5795 experiment3_sent2vec
3 Q0 AP881012-0156 8 0.5777 experiment3_sent2vec
3 Q0 AP880217-0150 9 0.5727 experiment3_sent2vec
3 Q0 AP880620-0158 10 0.5715 experiment3_sent2vec
20 Q0 AP881110-0035 1 0.8175 experiment3_sent2vec
20 Q0 AP881122-0171 2 0.8093 experiment3_sent2vec
20 Q0 AP881111-0092 3 0.7500 experiment3_sent2vec
20 Q0 AP881123-0037 4 0.7327 experiment3_sent2vec
20 Q0 AP880427-0010 5 0.6378 experiment3_sent2vec
20 Q0 AP880527-0290 6 0.6352 experiment3_sent2vec
20 Q0 AP880602-0168 7 0.6231 experiment3_sent2vec
20 Q0 AP880603-0015 8 0.6230 experiment3_sent2vec
20 Q0 AP880526-0251 9 0.6230 experiment3_sent2vec
20 Q0 AP880405-0012 10 0.6226 experiment3_sent2vec
```