IR Assignment3

Philippe Voet

December 2024

Github: https://github.com/Triple1312/IR_assignment3

1 Background Discussion

Developing a robust semantic search and retrieval system requires a comprehensive integration of several key components and algorithms. These components work together to effectively interpret and process natural language queries. This ensures that the most relevant documents are retrieved based on their semantic content.

1.1 Contextual Embedding Models

At the heart of any semantic search system lies the ability to understand and represent the semantic meaning of text. Contextual embedding models do this by converting textual data into high-dimensional numerical vectors that encapsulate the nuanced meanings of words and sentences within their specific contexts. They create an embedding for each token found in the document

1.2 Document Embedding

Document embedding refers to the process of converting whole documents into vectors of numbers while conveying their meaning. All vectors will have a fixed length based on the model. In such a way, unstructured text is translated into a structure that can be used by machine learning algorithms. It involves several steps in tokenization in the process of embedding.

Tokenization is one of the crucial initial steps in preparing text for processing by transformer models. It involves a number of detailed steps to convert raw text into a structured format that these advanced models can effectively understand and manipulate. The whole tokenization workflow in detail will look as follows:

1.2.1 Normalization

Normalization is the first step of the tokenization process where one standardizes the text in order to decrease the variations the model will need to cope with:

1. Lowercasing: One makes all text lowercase because so many models treat capitalized and lowercase words as different tokens.
2. Removal of Non-Printing Characters: It removes newline and carriage return characters which do not contribute anything to the meaning of text.
3. Unicode Normalization: Often, text is transformed into some consistent form using Unicode normalization; for example, NFC to ensure that all the characters in the text are uniformly represented.

1.2.2 Pre-tokenization

Pre-tokenization is a process of splitting the normalized text into preliminary tokens or words:

1. Whitespace Splitting: It's the simplest form of token splitting where tokens are split by whitespace.
2. Punctuation Splitting: The punctuation marks are separated from words for keeping syntactic and semantic boundaries by treating them as separate tokens.

1.2.3 Model-Specific Tokenization

After pre-tokenization, tokens undergo model-specific processing with respect to the transformer model in which it is going to work:

1. WordPiece: Adopted by models such as BERT, this algorithm breaks down words into their subword units which may happen to appear more frequently.
2. Byte-Pair Encoding: In models like GPT-2 and RoBERTa, this is one of the commonly used tokenizers where, iteratively, pairs of the most frequent character or byte pairs are combined to form the vocabulary.
3. SentencePiece: Unlike other algorithms, which tokenize raw text directly into subword units without using whitespace, thus quite helpful in cases when clear word boundaries are missing, as it operates word-piece tokenization or byte-pair encoding.

1.2.4 Post-processing

In this stage of post-processing, tokens get further processed into an understandable format in a model-readable way.

1. Converting Tokens to IDs: Each token will be converted to a unique number according to the predefined model vocabulary.
2. Special Token Addition: Special tokens serving specific roles, such as [CLS], [SEP], [PAD], and [UNK], are added to the token sequences when required by the model.

1.2.5 Padding and Truncation

To handle inputs of different lengths:

1. Padding: Shorter sequences are extended with the [PAD] token to match the length of the longest sequence in a batch or to a fixed length preset by the model.
2. Truncation: Sequences longer than the model's maximum length capability are cut down to an acceptable size.

1.2.6 Feeding to Model

Finally, the structured data is fed into the transformer model: token IDs, attention masks, and other tensors that might be needed, such as token type IDs. The model uses this for training or inference based on the model architecture in classification, translation, summarization, and so on.

Each of these steps serves to ensure that text data is processed efficiently for transformer models to handle such diverse and complex linguistic data with effectiveness and accuracy.

1.3 Embedding using KMeans inverted index

When the documents have been embedded in a vector space, KMeans can be run on these vectors. The KMeans clustering algorithm clusters these document vectors into a pre-specified number of K clusters. The idea is that all such clustering tries to achieve the minimum within-cluster variance. It achieves this by estimating the centroid for the points of each cluster, which for all the points is just the average, and performing an iterative assignment of each document to the nearest centroid. Actually, each cluster's centroid serves as a middle point: it is the collective semantic center of the documents it gathers.

The documents are then clustered, and afterward, an inverted index is developed to allow for rapid retrieval. Unlike the traditional inverted indices, which map content words to the documents containing them, in this case, the index is built around the clusters identified by the KMeans algorithm. Each cluster ID serves as a key in the index, whose value is a list of documents within that cluster.

When a query is made against the document collection, the query document is first embedded into the same vector space as the other documents. This query vector is then labeled with the cluster having the closest centroid through metrics such as Euclidean distance or cosine similarity. Documents in that cluster are then retrieved, presuming to be semantically most similar to that query. This narrows the search space significantly, hence enhancing speed and relevance in the process. Further refinements can be made in order to increase retrieval accuracy. This approach of KMeans clustering combined with an inverted index leverages the strengths of the clustering algorithm in grouping semantically similar documents and the efficiency of the inverted index in mapping and retrieving those documents fast. It provides an optimized solution for managing large datasets where quick access to related documents based on content similarity is crucial.

2 Implementation

2.1 Libraries

Since I am using typescript, I am limited to the libraries available to javascript and typescript. In short, this means Sentence-BERT is not available to me and I thus need another library to fulfill the necessary implementation for a tokenizer.

2.1.1 Transformers.js

Transformers.js is a JavaScript library that brings the power of transformer-based machine learning models directly to web browsers and Node.js environments. In my case, I am using Node.js. This is a direct alternative to using Sentence-BERT. Unlike Sentence-BERT, transformers.js uses the widely used ONNX framework to run AI models. The library also allows me to take any pretrained model posted on huggingface and use it.

2.1.2 TensorFlow.js

TensorFlow.js is a port of the open source TensorFlow library developed by Google. The Javascript port is very limited in comparison to the original TensorFlow library. For example, it does not have an implementation of the KMeans algorithm that I need for part 2 of the assignment. I use TensorFlow.js so I can harness the power of cuda to do many calculations way faster on my GPU.

2.2 Evaluator

The evaluator for `Map@k` and `Mar@k` can be found in `evaluator.php`. These methods can be found in the `SystemEvaluator` class.

2.3 Part1

Part 1 of the assignment can be found in `main_part1.ts`. It loads all the necessary files and creates a `RetrievalSystem` object.

2.3.1 RetrievalSystem

To use the RetrievalSystem class to retrieve relevant documents, one first needs to `fit` the class with documents. The `fit(documents)` method is rather straight forward, It creates a featureExtractor of the transformers.js library and embeds every document using the classes `embedDocuments(documents)` method. The featureExtractor (called `pipeline`)has some extra arguments like `device` which lets me select to use the cpu or gpu. More on that later.

The `embedDocuments(documents)` method uses the featureExtractor to embed each document. The embedding consists of a 2D Tensor of every feature found in the document. We than have an options to either average all featurevectors of all tokens, or take the max. The effects of using either one will be discussed in the evaluation section. At the end all document embeddings will be loaded onto the gpu for faster calculations and the norms will be calculated. both the embeddings and the norms of all documents will be saved inside the class

When the system is fitted, we want to retrieve the relevant documents for a given query. This can be done using the `query(query_string)` method. This method just calls the method `_query(query_string, documentEmbeddings, documentNorms, k)`. The reason for the existence of `_query` is that it allows you to choose different embeddings which will be used in part 2. This method calculates the cosine similarity between the `documentEmbeddings` and the `query_string` embedded vector. TensorFlow is used to make this process very fast.

2.4 Part2

Part 2 of the assignment can be found in `main_part2.ts`. It loads its necessary files and creates a `KMeandRetrievalSystem` class. This class uses the KMeans algorithm, which is a problem since TensorFlow.js does not have an implementation of KMeans, meaning I had to write it myself.

2.4.1 KMeans

The KMeans algorithm can be found in `KMeans.ts`. It is your standard KMeans algorithm which chooses the initial centroids completely random. It uses TensorFlow.js on the GPU to make it significantly faster.

2.4.2 KMeansRetrievalSystem

The `KMeansRetrievalSystem` class can be found in the `KMeansRetrievalSystem.ts` file. The `KMeansRetrievalSystem` class extends the `RetrievalSystem` class. Both the `fit` method and the `query` method get overriden.

The `fit` method calls the `fit` method of its super class, after that creates a `KMeans` object and assigns all embeddings to a centroid. To make code more readable each cluster gets saved in a `ClusterData` class. This class contains the centroid vector, the norm of this vector so it does not need to be recalculated for every query, the document embeddings, the norms for all document embeddings and finally the original index for every document.

The `query` method also needed a lot of changes. First the distances between the centroids get calculated. This is the reason the separated `_query` method exists. After that the distances to all embeddings assigned to the best centroids.

3 Evaluation

3.1 Part 1

I started using the evaluation with the "sentence-transformers/all-MiniLM-L6-v2" model. As I said earlier, Transformers.js does allow to run trhe models both on the cpu and gpu. I expected this would make a difference in the speed in which the model executes, which it does, however it has another side-effect. Loading the data on the gpu loses precision which can result in worse predictions. using the cpu I got these precisions and recalls:

- precision@1: 0.51012145748988
- precision@3: 0.21457489878543
- precision@5: 0.13927125506073
- precision@10: 0.076923076923077
- recall@1: 0.50404858299595
- recall@3: 0.63157894736842
- recall@5: 0.68421052631579
- recall@10: 0.75303643724696

for part 1 calculating on the cpu is very feasible, however for part2 I always used gpu.

The scores scores seem very low, this is mostly because there is only one relevant document assigned for every query. As a result this also makes the recall very high. In that sence I am still satisfied with the result.
Since transformers.js gives me a tensor of multiple tokens, these tokens need to be compressed into 1 vector for every document. there are many ways to do this. The most obvious one is to take the average like I did in the previous example. However it is also possible to take the max, and it was said online to be a possibility, so I tried it and this were the results:

- precision@1: 0.25506072874494
- precision@3: 0.11875843454791
- precision@5: 0.080161943319838
- precision@10: 0.046153846153846
- recall@1: 0.25303643724696
- recall@3: 0.3502024291498
- recall@5: 0.39271255060729
- recall@10: 0.45344129554656

For this model at least, mean is a way better metric than max.

I also wanted to see if another model would make a huge difference so I tested the ”sentence-transformers/multi-qa-distilbert-cos-v1” model. This gave me these scores:

- precision@1: 0.31983805668016
- precision@3: 0.13900134952767
- precision@5: 0.096356275303644
- precision@10: 0.057894736842105
- recall@1: 0.31781376518219
- recall@3: 0.41093117408907
- recall@5: 0.47570850202429
- recall@10: 0.56882591093117

These values are so much worse than my first model. I even expected them to do better since on the website of SBert this model is recommended for cosine similarity.
That made me think: maybe this model would work better with max instead of mean. This is what I got with mean:

- precision@1: 0.23481781376518
- precision@3: 0.10796221322537
- precision@5: 0.068825910931174
- precision@10: 0.039271255060729
- recall@1: 0.23279352226721
- recall@3: 0.32186234817814
- recall@5: 0.34008097165992
- recall@10: 0.38461538461538

This is not really a better score than with mean. Its save to say that mean always gives the better result

3.2 Part2

Because we use KMeans which assigns its initial centroids at random, I never got the same scores twice. This makes it very difficult to know when we have bad parameters or just have very bad luck with our initializations.

I did not get my code to finish in time, I am very sorry. I will keep working on it until I am finished and the final report will be in the github repo. All the code does work, my problem was just time and overloading of my resources because it is so much data.

3.2.1 Continued

My problem was that I made lists that were way too long (javascript restriction, which is my fault for using typescipt) and that I ran out of space on my gpu. The way I fixed this is better memory management and interestingly use KMeans with larger k's. This is also a perfect example of how KMeans does more than just fixing a speed problem, especially for even higher amounts of documents.

For this part I am still using the "sentence-transformers/all-MiniLM-L6-v2" model since it got me my best results. I first tried to use a very low amount of clusters/searchcluster ratios, since theoretically this should almost centrainly give me a near perfect score. I started by using 100 clusters and searching in the 20 closest and doing 10 iterations for fitting the KMeans. I thought that because the amount of clusters, the amount of iterations wouldn't change much. These were my results:

- precision@1: 0.48548548548549
- precision@3: 0.36436436436436
- precision@5: 0.3035035035035
- precision@10: 0.21941941941942
- recall@1: 0.032167934688712
- recall@3: 0.072042940466274
- recall@5: 0.099823402513524
- recall@10: 0.14291222460818

I expected better scores than just 0.48 for precision@1. My first thought was that maybe 10 iterations were just too few. So for the next test I used 50 iterations:

- precision@1: 0.48448448448448
- precision@3: 0.36703370036703
- precision@5: 0.30610610610611
- precision@10: 0.21861861861862
- recall@1: 0.032133007856399
- recall@3: 0.072557558088735
- recall@5: 0.10061604449382
- recall@10: 0.14241835068207

My scores were pretty much the same so I was right about that the amount iterations didn't matter for such a large amount of clusters. Maybe The problem was that using 20 clusters as a search space is just too few. For my next test I used 50 iterations, 100 clusters and searching in 50 clusters. This would definitely get every relevant document:

- precision@1: 0.48148148148148
- precision@3: 0.3667000333667
- precision@5: 0.3041041041041
- precision@10: 0.21831831831832
- recall@1: 0.032028852256888
- recall@3: 0.072627920104401

- recall@5: 0.09997329174408
- recall@10: 0.14229268611856

I still got the same results. From this I think it's safe to assume that these scores are the maximum precision of this model. These scores are very low in comparison to BM25, although the advantages of using this are also apparent. Now its time to see the limits of how efficient using sentence transformers can be. I started using 100 clusters and searching for relevant documents in 2 of them with 50 KMeans iterations. These were the results:

1. precision@1: 0.40540540540541
2. precision@3: 0.3013013013013
3. precision@5: 0.24424424424424
4. precision@10: 0.17597597597598
5. recall@1: 0.026666118935778
6. recall@3: 0.059236055326676
7. recall@5: 0.080004977740454
8. recall@10: 0.11441270907624

Losing about 8% precision isn't actually that bad if you think about how much more efficient the search was for each query. When time is a real issue it is probably worth using only 2 clusters. So what if we go even lower? This are the scores using 500 clusters and searching in 2 of them:

1. precision@1: 0.38138138138138
2. precision@3: 0.27560894227561
3. precision@5: 0.22582582582583
4. precision@10: 0.16366366366366
5. recall@1: 0.025117759385407
6. recall@3: 0.054664604327412
7. recall@5: 0.074461021362117
8. recall@10: 0.10734274031515

I am very impressed by how well it performed in comparison to previous test since I cut the amount of vectors I search is a tenth. This definitely shows how efficient using KMeans is as an inverted index. Now off course 0.38 accuracy is rather low. It would like better scores. I thought about testing if using 500 clusters and searching in 10 would make a big difference , this would give me the same amount of vectors to check as unsing 100 clusters and searching in 2. These were the results:

1. precision@1: 0.453453453453453
2. precision@3: 0.345345345345345
3. precision@5: 0.284084084084084
4. precision@10: 0.202902902902902
5. recall@1: 0.030162141854745
6. recall@3: 0.068673262318188
7. recall@5: 0.093899157310526
8. recall@10: 0.13296100054304

The precision of 100/2 was about 0.405, now having 500 clusters and searching in 10, we have a precision of 0.45. Even though we searched the same amount of vectors, we got a way better result. It's almost as good as searching all vectors, which is insane. The key to have better precision seems to be to make more clusters and searching in more of them. I expect it more to be that searching in more clusters will always give a better precision to the point you can offset it by making more clusters. Lets test this theory with using 1000 clusters and still searching in 10 of them halving the amount of vectors to search:

1. precision@1: 0.441441441441441
2. precision@3: 0.335001668335
3. precision@5: 0.272872872872872
4. precision@10: 0.1972972972973
5. recall@1: 0.029177160417408
6. recall@3: 0.066503561329544
7. recall@5: 0.090220115084872
8. recall@10: 0.12894710888868

These are great results for cutting our search vectors in half. Almost no loss in precision. one might even argue that the randomness of the KMeans might be why its a little worse.

4 Conclusion

Using a KMeans inverted index is a great way of cutting down search time. Not only is it very efficient, it also allows for less RAM usage since I only need to load in the vectors created by the sentence transformer and further more, I can use a gpu to do even faster calculations. Whilst the precision is lower than using BM25, The advantages of using sentence transformers outweigh these scores. We also saw that using KMeans we can make the amount of vectors we need to search very small as long as we search even a few clusters with a high k.