Thomas Enns - 20823674
MSCI 541
2022-12-02

HW5 – Report

1. Since, as hinted in the question, “words of a feather flock together”, we can safely assume that similar words will be used in similar contexts (i.e., documents). The weights of terms represent how integral they are to the subject matter of the document. Thus, if two terms both have high weights for a given document, it increases the probability that they are similar. If two terms consistently appear together with high weights, then it is probable that they are similar.
   Therefore, if we split the matrix into term weight vectors for each term, with each element being that terms weight in some document, and compute the cosine similarities of all the terms with each other, the pairs of terms with the highest cosine similarity would be the most similar terms, as they are used together the most. Cosine similarity formula is below:

$$similarity(A, B) = cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|}$$

Cosine Similarity Formula (Karabiber, n.d.)

2.
   a. To compute the MLE model of a document collection we follow the following equation to compute the number of occurrences of each term throughout the entire collection and divide it by the total length:

$$P_{MLE}(w|M_D) = \frac{D(w)}{|D|}$$

   This essentially results in the probability of choosing that word at random in the collection.

   b. Dirichlet prior smoothing is an instance of Jelinek-Mercer smoothing. As stated in the assignment question, Dirichlet prior smoothing is defined to be:

$$P(w|M_D) = \frac{D(w) + mP(w|C)}{|D| + m}$$

   While Jelinek-Mercer smoothing is defined to be:

$$P(w|M_D) = (1 - \lambda)P(w|D) + \lambda P(w|C)$$

   From lecture, we know that the instance in which the two smoothing methods are identical is when:

$$\lambda = 1 - \frac{|D|}{|D| + m}$$

   Where λ is the smoothing parameter for Jelinek-Mercer, m is the smoothing parameter for Dirichlet prior, and $|D|$ is the document length. In the question, it is given that $\lambda = 0.5$ and $m = 100$. Thus, we simply need to find the document length for which the two smoothing methods will be identical.

$$0.5 = 1 - \frac{|D|}{|D| + 100}$$
$$0.5 = \frac{|D|}{|D| + 100}$$
$$(|D| + 100) * 0.5 = |D|$$
$$0.5 * |D| + 50 = |D|$$
$$|D| = 100$$

Thus, at a document length of $|D| = 100$, documents will be smoothed in the same way by both methods.

3. Dirichlet prior smoothing smooths short documents more than long documents because it essentially begins calculates the probability picking any given word in the entire collection and uses those probabilities as 'pseudo-counts'. When a document is shorter, it has fewer words, meaning that in more cases:

$$P(w|M_0) = \frac{0 + mP(w|c)}{|D| + m}$$

Meaning that the product of all the $P(w|M_0)$ terms will be far more affected by the probabilities of finding a word in the collection than finding a word in the document. Essentially, Dirichlet prior smoothing smooths short documents more than long documents because it has less data about the document to work with.

4.
   a. Stacked transformer layers are a set of layers that each accept an input sequence of text and will then output a different sequence of text to pass on to the next layer. Stacked LSTM (Long-Short-Term-Memory) layers do essentially the same thing to an outside observer. Both allow help to model a relationship between words and their 'meanings'.
      However, transformers process entire sentences as a whole using a self-attention mechanism to focus on specific words and use them to inform the output sequence. LSTM layers on the other hand, go through the input one word at a time and attempt to predict the next element. This means that the stacked transformer layers have a serious edge in terms of computing efficiency because they can take better advantage of parallelism (Kafritsas, 2022). Over large amounts of data, this is a significant improvement in time.
   b. In the pre-training stage, the language model is essentially learning how language works. It performs unsupervised learning on text data (documents). It essentially allows the model to develop some sort of understanding of the meaning and context of words and sentences.
      In the fine-tuning stage, the model is actually given a problem to solve. This is usually done on a more precise set of documents. This is the part where the model can fine tune its parameters so that it can optimize its performance on the task.
   c. Recurrent Neural Networks use back propagation through time to solve error. They do this by computing the partial derivatives of the term weight over time and then computing their product. This is called the gradient and is a measure of

how much a change in a previous weight will affect the output. However, many layers in, this gradient (product) becomes very small, and makes it such that the network makes vanishingly small tweaks and essentially stops learning (Medewar, 2022).
LSTMs have gates in each of their units that will 'forget' some of the previous information when sequences of text enter. This keeps the values of the gradients larger and allows for learning through all the many layers.

5. Instructions to run this code are contained in Appendix A.

The top 100 results from the BM25 algorithm with stemming were compared with the same results re-ranked with the monoBERT machine learning algorithm. After using the HW3 programs to calculate precision and gain metrics, we can conclude that re-ranking with monoBERT only offers a statistically significant improvement in Mean TBG, and offers small but not statistically significant improvements in the other metrics.

Run Name	Mean Average Precision	Mean P@10	Mean NDCG@10	Mean NDCG@1000	Mean TBG
stem	0.239	0.284	0.374	0.414	2.04
monoBERT	0.255	0.316	0.409	0.429	2.238
p-value	0.362	0.095	0.212	0.285	0.015

Table 1: Mean Run Metrics Comparison

In Table 1, we can see that there are small (not significant) improvements in Mean Average Precision ($p = 0.362$), Average Precision at rank 10 ($p = 0.095$), Mean NDCG at rank 10 ($p = 0.212$), and Mean NDCG at rank 1000 ($p = 0.285$). Mean Time Biased Gain ($p = 0.015$) offers the only statistically significant improvement from stemmed BM25 to monoBERT. This is potentially due to the fact that monoBERT did not really have a chance to greatly enhance recall over the top 100 as it was simply re-ranking the BM25 results. Even such a small improvement is a positive sign in that context. Were we to have run monoBERT over the entire document collection, we can hypothesise that it would have had a much more significant improvement when compared with the entire BM25 returns.

Below, we can see the percent change in metrics for each topic from the Baseline stemmed BM25 algorithm to the monoBERT algorithm.

Topic	AP	AP_10	NDCG_10	NDCG_1000	TBG
401	48%	0%	18%	18%	39%
402	61%	67%	58%	18%	37%
403	35%	17%	37%	18%	12%
404	0%	0%	0%	0%	0%
405	29%	**-100%**	**-100%**	4%	5%

406	-36%	-33%	-42%	-22%	-12%
407	-19%	-25%	-13%	-6%	-17%
408	-9%	0%	-42%	-14%	4%
409	0%	0%	0%	0%	-9%
410	-48%	-50%	-36%	-24%	-42%
411	-46%	0%	-47%	-34%	-3%
412	30%	13%	9%	7%	20%
413	300%	Infinite	Infinite	85%	42%
414	-79%	-100%	-100%	-41%	-61%
415	0%	0%	0%	0%	0%
417	-5%	-29%	-19%	-2%	-5%
418	-28%	0%	-37%	-17%	-12%
419	-12%	-33%	-16%	-7%	-22%
420	3%	-13%	-18%	-3%	10%
421	4%	0%	0%	1%	4%
422	38%	29%	40%	11%	26%
424	-11%	0%	34%	0%	-8%
425	26%	29%	22%	7%	20%
426	26%	300%	370%	16%	18%
427	29%	50%	38%	10%	-6%
428	236%	100%	177%	83%	61%
429	-2%	-25%	-10%	-2%	-12%
430	33%	25%	42%	25%	3%
431	-1%	0%	-13%	-5%	10%
432	290%	0%	0%	43%	469%
433	72%	0%	0%	13%	213%
434	-17%	100%	2%	-16%	31%
435	13%	0%	0%	2%	-5%
436	64%	43%	39%	17%	29%
438	46%	100%	141%	16%	26%
439	128%	0%	0%	23%	95%
440	-3%	20%	8%	-1%	0%
441	16%	-17%	11%	18%	-8%
442	318%	150%	300%	91%	66%
443	32%	0%	27%	15%	6%
445	-7%	-50%	-29%	3%	-10%
446	314%	Infinite	Infinite	66%	176%
448	9400%	Infinite	Infinite	558%	2875%
449	-78%	0%	0%	-32%	-90%
450	-12%	0%	-19%	-11%	-4%

Table 2: Per-Topic Percent Improvement of monoBERT Over BM25

In Table 2 above, the instances in which there was a negative percent improvement in a metric for monoBERT when compared to baseline are bolded and highlighted. The instances where there was no change in performance for a metric are highlighted but not bolded. Any non-formatted instances indicate improvement from the baseline stemmed BM25 algorithm. For many topics, monoBERT did not offer significant improvement over the baseline algorithm. However, for others, it offered a very large improvement, especially in the two metrics that consider only the top ten results, suggesting that the re-ranking was very effective for those specific queries.

Some of the topics with the largest improvement due to the monoBERT reranking are 413 (steel production), 428 (declining birth rates), 442 (heroic acts), 446 (tourists, violence), 448 (ship losses). In these situations, the largest improvement percentages were in the two metrics at rank ten, meaning that monoBERT took relevant documents and correctly placed them in the top ten where there had previously been few or no relevant documents. Here, mono-BERT most likely conserved the meaning of words such as “production”, “rates”, “violence” and “losses”, that the porter stemmer most likely had lost in the BM25 algorithm.

The topics for which monoBERT had the most negative effect on performance are 406 (Parkinson's disease), 407 (poaching, wildlife preserves), 410 (Schengen agreement), 414 (Cuba, sugar, exports), 419 (recycle, automobile tires). Here, three of the queries have proper nouns: “Parkinsons’s”, “Schengen”, and “Cuba”, which BM25 tried to find matches for, but which monoBERT tried to find the meaning and most likely failed.

I do think that the performance of this algorithm justifies the computational cost. In the context of this assignment, it does not. However, I attribute the lack of statistically significant improvements to the fact that we were re-ranking what had been produced by a previous algorithm. I believe that if we were to run monoBERT on the entire collection it would produce far better results. Additionally, once a query is made once, the scores can be saved and the expensive computing only really needs to happen when a new query is entered into the system.

References

Kafritsas, N. (2022, July 6). *Block-recurrent transformer: LSTM and Transformer combined*. Block-Recurrent Transformer: LSTM and Transformer Combined. Retrieved December 2, 2022, from https://towardsdatascience.com/block-recurrent-transformer-lstm-and-transformer-combined-ec3e64af971a

Karabiber, F. (n.d.). *Cosine similarity*. Cosine Similarity. Retrieved December 2, 2022, from https://www.learndatasci.com/glossary/cosine-similarity/

Medewar, S. (2022, September). Solving the Vanishing Gradient Problem with LSTM. Retrieved December 2, 2022, from https://www.codingninjas.com/codestudio/library/solving-the-vanishing-gradient-problem-with-lstm

Appendix A – README File (Instructions to run code)

```
# MSCI-541-HW4

Thomas Enns - 20823674

This repository consists of 11 programs which work together to create then query
an index of files, then to evaluate search results and performance.

The first, IndexEngine.py, accepts two arguments: the filepath to latimes.gz, and
the filepath to a folder that will be created and will house all the
files/documents. The program will split the gzip into the approx. 131 000 files,
create an inverted index of each token (word) present linked to all the docs in
which it appears, store the index, store their metadata, and store the files in
the specified folder, divided by date.  An example of the appropriate way to run
this program from the console is as follows:
python IndexEngine.py C:\Users\thoma\Documents\3Bterm\MSCI541\latimes.gz
C:\Users\thoma\Documents\3Bterm\MSCI541\testdir

The second, GetDoc.py, accepts three arguments: the filepath to the document
store, the type of identifier used to request a document (either DOCNO or doc
id), and the identifier itself. The program will then return the corresponding
document along with it's metadata to the console. An example of the appropriate
way to run this program from the console is as follows:
python GetDoc.py C:\Users\thoma\Documents\3Bterm\MSCI541\testdirectory docno
LA123190-0134

The third, BooleanAND.py, accepts three arguments: the location of the index
file, the name of a query input file, and the name of a query output file. The
program will parse and tokenize the queries and print the list of documents
containing all the query terms in the output file. An example of the appropriate
way to run this program from the console is as follows:
python BooleanAND.py C:\Users\thoma\Documents\3Bterm\MSCI541\testdir queries.txt
hw2-results-tenns.txt

The fourth program, Helper.py, is not meant to be run. It only contains methods
that can be used by the other programs such as a tokenizer and a method to
convert tokens to ids. The tokenizer is used by both the InvertedIndex program
and all querying programs. It has an option to use the stemmer or not.

The fifth program, Evaluate.py, accepts two arguments: the name of the .txt file
of relevance judgements for the queries and the name of the run file with the
results of the queries. It computes effectiveness measures for each query and
outputs them to both a .txt and a .csv in the measures folder. An example of the
appropriate way to run this program from the console is as follows:
```

```
python Evaluate.py LA-only.trec8-401.450.minus416-423-437-444-447.txt hw4-bm25-stem-tenns.txt

The sixth program, Summarize.py, accepts no arguments. It iterates over all the results files created by the Evaluate program and outputs a table of the average performance metrics for each run file. An example of the appropriate way to run this program from the console is as follows:
python Summarize.py

The seventh program, compare.py, accepts three arguments, the names of the two .csv files created by the Evaluate program which the user wishes to compare, and a number corresponding to the desired metric 1-MAP. 2-AP@10, 3-Mean NDCG@10, 4-Mean NDCG@1000, 5-Mean TBG. The program outputs a simple .csv file with the desired metric for each run, the percent improvement of the better performing run over the worse performing run, and the p-value. An example of the appropriate way to run this program from the console is as follows:
python compare.py hw4-bm25-baseline-tenns-measures.csv hw4-bm25-stem-tenns-measures.csv 1

The 8th program, Calc.py, accepts no arguments, and simply computes the number of words in collection, number of non-zero instances of terms in documents, and the average document length in collection. It outputs these to the console. (This was used for Q5 and for avg doc length for BM25) An example of the appropriate way to run this program from the console is as follows:
python Calc.py

The ninth program BM25.py, accepts three arguments: the location of the index file, the name of a query input file, and the name of a query output file. The program will parse and tokenize the queries and print the list of the top 1000 ranking documents according to the BM25 equation in the output file. An example of the appropriate way to run this program from the console is as follows:
python BM25.py C:\Users\thoma\Documents\3Bterm\MSCI541\testdirS queries.txt hw4-bm25-stem-tenns.txt

The tenth program, docIndex.py, accepts one argument, the location of the gzip file containing the documents. The program will esentially only parse the documents to create an index which maps from the docno to the text of each document and then save that index as a pickle file. An example of the appropriate way to run this program from the console is as follows:
python docIndex.py C:\Users\thoma\Documents\3Bterm\MSCI541\latimes.gz

The eleventh program hw-5-tenns.ipynb, is essentially to re-ran the top 100 results from the BM25 run file using the monoBERT machine learning algorith. It has a number of codeblocks that can be run sequencially. The ones with reference to github can be ignored
```