SI 650 / EECS 549 F24 Homework 3
Deep Learning for IR

Due: See Canvas for official deadline

1 Homework Overview

A new era of IR has begun in the past three years as new deep learning models are introduced to represent documents and queries. These deep learning systems rely on pre-trained language models that have seen massive amounts of text and already know how to produced vector representations of text that are effective for capturing its content. IR models adapt these pre-trained models so that representations also capture relevance: the similarity of a query vector and document vector correspond to the document's relevance for that query. Unlike our older tf-idf vectors, these deep learning produced vectors are capable of capturing nuanced meaning, reducing ambiguity in document representations, and matching queries and documents that have no term overlap but still have similar meaning.

Homework 3 will have you *extending* the code from Homework 2 to integrate several deep learning models in your IR system. To accomplish this goal, you will implement three new pieces of an Information Retrieval system:

1. **Document Augmentation**: Use a doc2query model to add new text to your documents
2. **Bi-encoder Ranking**: Use a bi-encoder to rank documents
3. **Cross-encoder Ranking**: Use a cross-encoder to re-rank the top documents for relevance (or as a feature for the L2R re-ranker)

These deep learning systems can be slow to use in practice so we have pre-computed much of the slow part. While you'll still implement all the pieces to generate meaningful output (which will allow you to test and experiment), you can use our pre-computed outputs to side-step slow operations when building the full IR system.

This assignment has the following learning goals:

1. Gain familiarity with document augmentation strategies
2. Understand how to generate queries for documents and how systems that do this task can vary in their quality
3. Learn how to use bi-encoder and cross-encoder models to rank documents
4. Gain skills in development, debugging, and performance optimization
5. Improve ability to read documentation for new libraries
6. Better understand the performance-accuracy trade-offs for using deep learning models.

2 Important Notes

This homework will slightly increase the memory requirements for running the core IR system due to needing to keep more text information around (as you'll see later). We have added support for using the Great Lakes cluster so you can request time on computers with larger memory. You should have access through the course account `si650f24_class`. If the IR system works fine your machine, you do not need to use Great Lakes. You do not need to use GPUs for this assignment; we have precomputed all the deep-learning intensive parts.

Homework 3 includes only one notable change to the core search engine implementation and design.

1. You will be swapping out the BM25 ranker in your L2R for a bi-encoder based ranker. To make this easier and the code cleaner, we've changed the init signature for the L2R class

Things to keep in mind when working on this assignment:

1. Please start early
2. The bi-encoder and cross-encoder pieces can be developed and tested separately
3. There are more plotting and critical thinking questions in this assignment. Please make sure you leave enough time for analysis and reflection.
4. The deep learning parts of the assignment rely on the `transformers` (Part 1) and `sentence-transformers` (Parts 2 and 3) libraries. You should not need to know about deep learning itself or PyTorch to successfully complete this assignment. If you run into these kinds of errors and start reading PyTorch documentation, you're probably looking in the wrong spot for how to fix your bug. When in doubt, ask on Piazza.
5. You are free to add your own functions to the classes and write your own main function as well.
6. Please read the `README.md` throroughly before starting to code.
7. We have provided a Jupyter notebook which has some example code.

To reduce ambiguity in this assignment, the following notation is included here and through all future assignments.

1. An arbitrary word is w_i where i refers to which word this is within the vocabulary V. A word can be a single token or a multi-word expression.[1]
2. An individual document is d, which may be denoted as d_i to indicate an arbitrary document in the collection. The length of the document in words is denoted as $|d|$, which includes stop words (before they are filtered).
3. The set of all documents in the collection is D and $|D|$ denotes the number of documents in the collection
4. The set of all documents' titles is T and $|T|$ denotes the number of documents' titles in the collection. In practice, this is the same as the number of documents but we use T to make the distinction clearer in the equations of whether we're referring to the document's main text (D) or its title (T).

[1] Note that for Homework 3, there are no multi-word expressions.

5. $c_d(w_i)$ is the count of how many times w_i appears in the main text of document d (not including any other metadata/title associated with the document)
6. $c_t(w_i)$ is the count of how many times w_i appears in the title t (this assumes that all documents have a title)
7. $df_{w_i}^D$ is the number of documents in D that contain w_i in the body text (not title or metadata). $df_{w_i}^T$ is the analogous value for the number of titles that a word occurs in.
8. $avdl$ is the average document length, i.e., $\frac{1}{|D|}\sum_i |d_i|$

3 Document Augmentation

Queries don't always have the same text as the most relevant documents. This case is particularly true for queries that are questions or where the query is a more general description of a specific thing. To help match queries to documents, we can do *document augmentation* where prior to the indexing step, we modify the document itself so that its contents now contains new text that might match a query.

In Part 1, you'll examine a few different approaches to adding queries to a document's text. We talked about one of these in class, doc2query, but there are many models that can suggest what queries might lead to a document. All of these models are based on pre-trained deep learning models. Thankfully, someone has already trained and release these models to the HuggingFace model repository so we can directly integrate them into our IR system.

Because the query-augmentation models are based on deep learning, they can be very slow to run unless you have a GPU. Therefore, we've pre-computed the queries for each of the 200K documents in our collection for you to use. However, you will also still need to implement some code to do the query-augmentation step so you can compare models by having them generate examples for just a small set of documents.

The implementation for Part 1 has the following high-level steps:

1. Implement the `Doc2QueryAugmenter` class in `document_preprocessor.py`, which will use a pre-trained HuggingFace model.
2. In `indexing.py`, immediately after loading a document's text, look-up the pre-generated queries (we provided) and append these to the text (before performing any pre-processing or filtering). Then proceed as normal.

■ **Problem 1.** (10 points) What kind of queries are these model generating? This question will have you generating some good (or bad) queries from a few different models, which you can directly plug into your `Doc2QueryAugmenter` by specifying a different name. You'll use the following four models:

1. `doc2query/msmarco-t5-base-v1` (what we used to pre-generate)
2. `google/flan-t5-small`
3. `google/flan-t5-base`
4. `google/flan-t5-large`

These last three models are general-purpose language models that are trained to follow instructions. The size part on the end specifies how many parameters are in the model

(roughly speaking); larger models are capable of more, but are also slower. However, these general purpose models don't know what to do unless you tell them (unlike doc2query, which only knows to generate queries). To make them work, you'll need to add the string "Generate a query for the following text: " as a prefix to the input. The `Doc2QueryAugmenter` has instructions on how to do this so you can use this class for all four models.

Using each of the models, generate a query for each of the first 100 documents and keep track of the query and the time in seconds. Make a bar plot showing the mean time-per-query-generation with 95% confidence intervals with bars for each of the models (plotting with Seaborn makes this easy). Using the mean performance time for these 100 documents, estimate how long it would take in minutes (or hours) to generate queries for all 200K documents.

In a few sentences, describe what you see in the queries and whether they look of good enough quality to use. Looking at the timing plot and estimated time-to-complete, describe whether which model you would choose based on its query quality and speed and why.

■ ***Optional*** **Problem 1.** (0 points) Generate new queries using one of the more advanced flan-t5 models and add them to the documents. Then later in the evaluation, test whether an index that incorporate these query-augmentations is better for retrieval using NDCG@10 and MAP@10

4 Bi-Encoder Ranking

Part 2 will have you working with bi-encoder models. These models are designed to produce dense, relatively-compact vector representations of documents and queries. For IR, bi-encoder models start from some existing pre-trained language model like BERT, RoBERTa, MiniLM, etc., and then are trained so that the encoded representations of relevant queries and documents have high cosine similarity (or inner product of their vectors). The base language model determines the size of the vector as well as what kind of "language" the model can accurately represent. For example, if a language model is pretrained on just English text, the bi-encoder will likely only work for English language queries; in contrast, if a language model is pretrained on multiple languages, then the bi-encoder might still work well for many languages!

One big advantage of bi-encoder IR setups is that you can precompute the document vectors. This way, to do a retrieval, you only need to generate one new vector for the query and then compare the query and documents vectors. For Part 2, we've precomputed all the vectors you'll need for the bi-encoder for the full pipeline. However, we'll ask you to try out a few different bi-encoders *outside of the larger IR system* and see how well they work on our relevance data.

The following list summarizes the steps you will need to complete in the code for the IR system. See the code for full details.

1. Implement the `VectorRanker` class that will encode a query and score how similar the query is to all documents. Note that this is a slightly different interface from the `Ranker` class because it is *much* faster to compute multiple vectors scores at once due to vectorized operations in `sentence-transformers`.

2. In your `L2RRanker`, replace the initial `Ranker` (initialized with a BM25 `RelevanceScorer`) with a bi-encoder model ranker. This new scorer will identify the candidate set of documents to re-rank.

■ **Problem 2.** (10 points) How well do different bi-encoders work? Let's take a look using just our development data to see whether the bi-encoder scores are correlated with the relevance scores you produced in Homework 1. Do the following steps (read all of the instructions first):

1. In a Jupyter notebook (or with your favorite tool), load in the development data's queries, documents, and relevance scores (you'll want the full text for queries and documents)
2. You'll use each of the following three bi-encoders, which should just work with the existing `VectorRanker` class you wrote:
   (a) `sentence-transformers/msmarco-MiniLM-L12-cos-v5` (this is what we used for your precomputed vectors)
   (b) `multi-qa-mpnet-base-dot-v1`
   (c) `msmarco-distilbert-dot-v5`
3. For each of the bi-encoders, encode each development-set document into a vector and record the total time it takes in milliseconds. Also encode each of the queries but do not keep track of time.
4. Compute the inner product of each development set document's vector and each of the query vectors and keep this as a list of similarity scores for each query (i.e., for each query, how similar is each document?)
5. Compute the mean NDCG@10 and MAP@10 for each of the three encoders, ordering documents by the inner product (for the respective ranker).
6. Make a bar plot showing the mean NDCG@10 and MAP@10 across queries with 95% confidence intervals for the mean value (Seaborn makes this easy). Use different bar colors for each metric and the same x-axis for each ranker.
7. Make a *second* bar plot showing the total encoding time to turn the development set into vectors, with one bar per model.
8. In a few sentences, describe what you see in both plots. Do some models do better than others? Are performance and time related?

■ **Problem 3.** (20 points) This problem will have you thinking about *fairness* in search. Language model based IR systems are less inspectable than the simpler BM25. For example, it's often difficult to say *why* a model ranks one page higher than other because the particular score is dependent on both what the model already encodes about language and what data it is trained on. These models operate as "black boxes" that are powerful in practice but may have unintended outcomes for which pages are retrieved. This problem will have you consider a few generic queries like "person" that are applicable to many pages in our Wikipedia data and see which kinds of pages the Bi-encoder ranker rates as more relevant.

For this problem in Part 2, we have extracted person attributes from Wikidata for the pages that are about people in our 200K articles. Recall from our lecture of bias that, in the US, some attributes are protected like race or gender, while others are not. You'll work

with four attributes for this problem: gender, Ideally, documents that are equally relevant except for differences in these identity attributes should have very similar relevance scores from an IR model (similar rankings). But do they in our case?

For this problem, you can work with the bi-encoder directly, outside of the whole IR system. We recommend working in a jupyter notebook so you can interact and debug as you go. Your computational tasks are as follows (other tasks for this problem are listed after):

1. Load the `person-attributes.csv` file in your preferred format. This file has a mapping from document IDs to the known attributes from Wikidata for the associated people with Wikipedia pages in our 200K articles.[2] Calculate the 10 most common labels for each of the four person attributes; you'll want these later for plotting.
2. Load the document vectors pre-computed with the `sentence-transformers/msmarco-MiniLM-L12-cos-v5` bi-encoder model.
3. Load the corresponding `docids.in-order.txt` file, which has the docids for the 200K documents from the JSON in the order they appear. (This should save you time from reading the whole JSON)
4. For each of the following five queries, encode them each separately and then rank all the documents: "person", "woman", "teacher", "role model", "professional".
5. For each of the five queries and for each category, using the data in the person attributes for that category, make a point plot for the mean rank with 95% confidence intervals for pages with any the 10 most common labels in that category. You'll end up with 20 plots total. Each plot should have one point for each of the 10 labels in an attribute categories (e.g., the Gender plot should have points for "male",[3] "female", "male organism", "trans woman", "non-binary", etc.). Seaborn can make this plot very easily if you give it a dataframe with columns for the rank of a page and the category of that page. If you really want, you can plot more labels from a category (or plot specific ones too).
6. Include the labeled plots in your PDF report.

Your Writing Task: In a long paragraph, write about the following three prompts.

1. Describe what you see in the 20 plots (one for query / attribute-category combination). Do some queries produce disparate rankings across attributes? Do pages with some some attributes rank consistently lower or higher?
2. What do you think might contribute to the behavior you're seeing? Feel free to speculate for all causes you think might be contributing.
3. If you wanted to quantify the fairness of an IR ranker, how would you? You can specify a specific equation or describe your quantification strategy in more general terms. You should not look up anything for this on the web—we're looking for your own thoughts on how to do it.

[2]Note that this data is somewhat noisy and you might see similar labels. You don't need to fix any of this and you should use the data as-is.

[3]These are the terms used in Wikidata.

5 Cross-Encoder Ranking

Cross-encoders are point-wise Learning to Rank systems: these models take as input the query and a document and are trained to predict how relevant the document is for the query. Unlike bi-encoders, documents cannot be pre-computed as vectors before an ad hoc query is issued; the cross-encoder must be run for each document anew when a new query is issued. Cross-encoders are typically implemented using large language models and therefore can be slow to run without a GPU. Therefore, most modern IR systems will first rank using a bi-encoder (like the one from Problem 2) and then *re-rank* the top k documents using a cross-encoder or a L2R system with a cross-encoder relevance score as a features. By only using the cross-encoder with the much smaller set of documents to be re-ranked (e.g., $k \approx 100$), there is a much smaller hit to performance.

In Task 3, you'll be adding a cross-encoder as a new feature to the L2R system. Note that because the cross-encoder needs the document text as input, this means we now also have to keep some part of the document around. However, cross-encoders have an input-length limit. For our homework, we're going to limit this to the first 500 words in the text, which we've estimated will take roughly 800MB of additional memory. You can stick to the model we provided `cross-encoder/msmarco-MiniLM-L6-en-de-v1`.

The basic workflow for Part 3 should look something like this:

1. During indexing, keep a separate data structure that maps the document ID to a string with the first 500 words in the document. (truncating the rest). You will need this full text for the cross-encoder.
2. Implement the `CrossEncoderScorer` model that takes in the query text and document text to score how relevant the document is. This model is effectively a point-wise Learning to Rank model!
3. Add the `CrossEncoderScorer` as a feature to the your `L2RFeatureExtractor` and re-train the system.

As noted in the introduction, if you run into issues with memory on this assignment, you can use Great Lakes to request a server with more memory to use interactively.

■ **Problem 4.** (10 points) How similar are the scores for bi-encoders and cross-encoders? Using the development data, select a sample of *at least* 3 queries and all the query-document pairs with ratings for thoese queries (more queries are better but see how quickly you can encode). Get the score for each query-document pair using (1) the product of the two Bi-encoder vectors, i.e,. the dot product score from `sentence-transformers` and (2) the score from the cross-encoder. Keep these scores in two lists, making sure each query-doc pairs has the same position in each list. Use the default deep learning models we use elsewhere.

Compute the following. First, let's see how correlated the two scores are—i.e., are they ranking documents in the same way? Compute Spearman's correlation between the two lists of relevance scores and show this score in your PDF. When the models differ, how are they differing? Let's try to plot this to get a sense. For each encoder, make a second list that converts the scores for each query into the document's rank (e.g., if you encoder the scores for three queries, you should have three documents ranked "1"—the top result for each query). Then plot the two lists of rankings using a scatter plot. This plot should show how the two

approaches differed, e.g., if a document was ranked at "2" (i.e., the second-most-relevant) by the bi-encoder, where was it put by the cross-encoder?

In a few sentences, describe what you see in terms of the similarities and differences between the two models in terms of their agreement and their differences.

6 Evaluation

How good is our new L2R-based model with the deep learning? We'll score the outputs using MAP and NDCG and compare against our baseline BM25 retrieval system and Homework 2 system. To save time, you can (and should) use the scores from the last assignment for this one, rather than recomputing them.[4]

■ **Problem 5.** (10 points) Train the L2R system using all the features including the cross-encoder score. We recommend training the L2R in code outside of the files provided with the class, like in a Jupyter notebook. You can import the relevant classes and train separately, which will make plotting a lot easier. Score your ranking function using MAP@10 and NDCG@10.

TODO: Plot the mean performance for the new model across all queries using a bar plot *and* the performance of the BM25 model and Homework 2's L2R model with 95% confidence intervals using different bar colors for each of the models. If you use Seaborn, it will do this if you have a DataFrame with columns for NDCG@10, MAP@10, and the IR model name and each row is the performance of a specific model on a specific query. In a few sentences, describe what you see. Does adding these deep learning-based features help improve performance over the previous two IR systems?

■ ***Optional*** **Problem 2.** (0 points) Why stop at 10? For some queries, it's useful to get more than just 10 results. Often some noticeable differences can show up when you adjust the cut-off. Try regenerating the plots for the problem above but use different cut-offs: 1, 5, 10, 20, 50, 100. Do any new differences show up between models?

■ ***Optional*** **Problem 3.** (0 points) How do the hyperparameters for LightGBM influence the L2R model's performance? Try configuring the L2R model with different hyperparameters and report how changing each influences the performance on the development and the test sets. Does the best performing model on the development data also perform best on the test data? What might that tell you about the stability of the performance? You can use a table or a figure to show your results on different hyperparameter settings.

■ ***Optional*** **Problem 4.** (0 points) Try ranking by changing the different hyperparameters of the IR system other than the IR model (e.g., increasing the minimum word frequency). What effect do these changes have on performance?

[4] We recommend saving these Homework 2 scores somewhere to a file so you can just read them in and plot them, without having to re-run things all the time.

7 Grading and Points

Answers to the questions above constitute 60 points. A correct implementation of the code is worth 40 points. Partial credit will be given wherever possible, provided you show your work.

8 Late Policy

You have **seven** free late/flex days for this course. All late days are managed through the autograder. Any submission that is after the scheduled time on Canvas (or the autograder) will use one late day. You do not need to ask permission for free late days. We intend you to use them *first* before asking for any other forms of extensions, unless you experience a serious and unexpected life event.

9 What to Submit

Autograder Submission Procedure Your implementation of all the files should be uploaded to the Homework 3 assignment in https://autograder.io. You should already be added as a student to the course but if you do not see the course there, please notify the instructional team immediately to be added. The autograder will tell you which files you have to upload.

1. Your code to the autograder
2. A PDF or Word document with your answers and plots to all questions

Academic Honesty Policy

Unless otherwise specified in an assignment all submitted work must be your own, original work. Any excerpts, statements, or phrases from the work of others must be clearly identified as a quotation, and a proper citation provided. Re-using worked examples of significant portions of your own code from another class or code from other people from places like online tutorials, Kaggle examples, or Stack Overflow will be treated as plagiarism. **The use of ChatGPT or Co-Pilot for generating solutions is prohibited; any solution using these tools will receive an automatic zero, regardless of how much of the code these tools were used for.** The instructors reserve the right to have you verbally describe the implementation of any part of your submission to assess whether you wrote it. If you are in doubt on any part of the policy, please contact one of the instructors to check.

Any violation of the University's policies on Academic and Professional Integrity may result in serious penalties, which might range from failing an assignment, to failing a course, to being expelled from the program. Violations of academic and professional integrity will be reported to Student Affairs. Consequences impacting assignment or course grades are determined by the faculty instructor; additional sanctions may be imposed.