A Job Search Engine Using fastText and BERT Word Embeddings

Term project report for SI 650 Information Retrieval

1 Introduction

The COVID-19 pandemic has caused roughly 14 million people in the U.S. to become unemployed since May 2020, with the unemployment rate rising more than it did during the Great Recession in 2008 [1]. Especially now as more people are trying to find jobs, it is important to have an efficient platform where users can search for jobs specific to their unique skillset, company interests, and experience level. According to a 2017 Jobvite Recruiting Funnel Benchmark Report, career sites and job boards contribute to 46.11% of hiring, with agencies, internal hires, and referrals making up the rest of the hiring process [2].

Most job search engines today allow users to search through job postings using a few keywords, and narrow down their search with various filters, such as for location and experience level. However, these search engines often limit the scope of the search, especially since filters are usually limited to pre-set options. A more functional search engine would allow users to enter more information about their skillset and positions they are interested in, or even a summary of their resume. BERT word embeddings can be used in this setting as they can capture a large amount of the complexity between words, such as polysemy and anaphora. BERT embeddings have been attempted to be used for semantic search across various verticals and even generalized search, but the high computational costs have hindered its widespread use. Guo et al. have proposed a method that combines learning to rank with BERT, called DeText, to overcome some of the inefficiencies involved in using BERT models for ranking, though with some performance loss [3]. In the job search engine domain, LinkedIn uses matrix factorization to predict skills and then ranks jobs using learning to rank approach [4]. Others, such as Muthyala et al, proposed using a database with information about companies and skills of current company employees along with document scoring techniques to allow for more accurate results when users have longer, more personalized queries [5].

This paper proposes building a search engine that is composed of two layers of semantic search using fastText and BERT document embeddings: the first layer uses fastText embeddings to retrieve the top 15 job postings, and the second uses BERT embeddings for the retrieved postings to rank them in order of relevance. This approach takes advantage of the strong contextual information captured by BERT embeddings, without accruing all the computational costs associated with using it both for retrieval and ranking. I evaluated the performance of two search systems built using the described approach, pre-trained on two different datasets (the NLI/STSB datasets and the TREC Deep Learning 2019/Macro MS Passage Reranking datasets). Based on ten sample queries, both models outperformed the ranking produced by fastText embeddings according to precision@10. However, comparing NDCG@10 did not lead to a conclusive decision as to whether our approach is a significant enough improvement on using fastText embeddings alone. Further experiments on a larger testing dataset and more fine-tuning a larger training set is required to quantify the added benefit in incorporating BERT embeddings for ranking.

2 Data

2.1 Collection and Preprocessing

The dataset used to build the search engine consists of 990 unique job postings from LinkedIn. As text on LinkedIn is structured, I was able to obtain information about the company name, the date that the job was posted, location, and level (entry, mid, senior, associate). In order to incorporate all this information into the job description text (which is the only data column considered for building the search engine), these were added to the bottom of each job description's text. For example, for the job's location, I added "The job's location is in : San Francisco, CA" according to the location column. Since the python packages used for the models (described in the methods section) handle tokenizing and processing the text into the appropriate format, no extra preprocessing was done.

Half of this data was obtained using phantombuster.com's LinkedIn Search Export and LinkedIn Job Scraper tools to scrape job postings. The search export tool was first used to collect a csv file of job URL's on LinkedIn corresponding to a specified query. The queries entered corresponded to the 5 largest job sectors in the U.S: "healthcare", "information technology", "real estate and development", "retail", "education", and "government" [6]. Next, the outputed spreadsheet was run through the job scraper tool to get a complete dataset with all the variables specified above. Since the free trial version of this application was used, it was limited to 40 jobs per run. The other half of the dataset was obtained using python's beautiful soup and selenium packages to scrape job postings from LinkedIn [7]. For this, the queries used were similar to those above, but with the addition of the query "jobs". This query was used to include a broad variety of job postings and add diversity to the dataset. The dataset is relatively small since I was only able to use the free trial version of the website above, and the code to scrape postings with selenium took a long time to process.

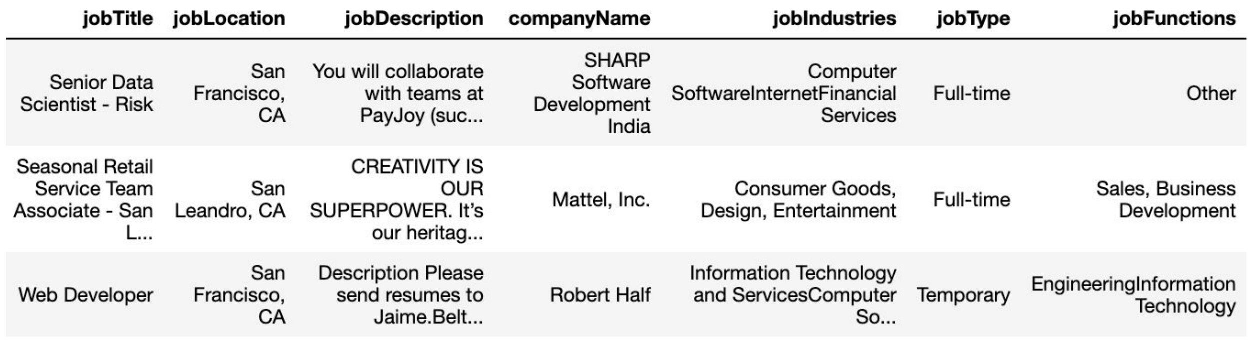
2.2 Data Statistics and Evaluation

The dataset contains jobs from 326 city/state locations and there are 117 industries represented. The job postings counts for the top 10 locations and industries are shown in Figure 2 below, as well as a few sample rows of the datasets, shown in Figure 1. The average job posting is 404 words, with the maximum being 3235 words.

A BM25 baseline, a random baseline, a baseline with only fastText embeddings, and a baseline using BERT embeddings for both ranking and retrieval (both in one step) was created to compare against the job search engine. Ten queries representing each of the top job sectors listed above were used to assess the search engine, shown in Figure 3 below. For each of these baselines and the proposed search engine, the top 15 jobs for each query were annotated on a scale of 0 to 4, with 0 indicating a job that is not relevant at all and 4 representing a very relevant job. The relevance was assessed by considering the desired location, skills, job title, and other query terms. NDCG@10 and precision@10 was computed for all these baseline systems, with labels of 3 or 4 representing relevant jobs.

For the model producing fastText embeddings, the entire dataset, without labels, was used for fine-tuning. For the BERT models, 170 jobs from the dataset were used for fine tuning. This training set included labels for each posting-query pair on a scale of 0 to 1 representing cosine similarity. The criteria for annotating these labels was similar as above, and a label of 1 represented a perfect job match for the given query.

Dataset:



jobTitle	jobLocation	jobDescription	companyName	jobIndustries	jobType	jobFunctions
Senior Data Scientist - Risk	San Francisco, CA	You will collaborate with teams at PayJoy (suc...	SHARP Software Development India	Computer SoftwareInternetFinancial Services	Full-time	Other
Seasonal Retail Service Team Associate - San L...	San Leandro, CA	CREATIVITY IS OUR SUPERPOWER. It's our heritag...	Mattel, Inc.	Consumer Goods, Design, Entertainment	Full-time	Sales, Business Development
Web Developer	San Francisco, CA	Description Please send resumes to Jaime.Belt...	Robert Half	Information Technology and ServicesComputer So...	Temporary	EngineeringInformation Technology

Job Description:

"Description Job Summary: Looking for a Jr.Java developer or Entry level Java developer in our technical team who will be responsible for designing, developing and implementing web-based Java application to support business requirements. you will also responsible for writing code in java to maintain the server-side application.
This job is located in San Francisco, CA.
The title of this job is Junior Java Developer at ConsultAdd Inc.
The job function is: Information Technology Engineering.
This job is in the following industries: Computer Hardware, Computer Software, Computer Networking."

Figure 1: *A few samples rows from the dataset and a sample job description are shown.*

Locations		Industries	
Washington, DC :	84	Other :	120
New York, NY :	70	Information Technology :	72
San Francisco, CA :	69	EngineeringInformation Technology :	70
Boston, MA :	27	SalesBusiness Development	53
Cupertino, CA :	18	Education Training :	40
Sunnyvale, CA :	17	Health Care Provider :	36
San Jose, CA :	14	Management Manufacturing :	25
Santa Clara, CA :	11	Administrative :	23
Austin, TX :	10	Sales Management :	22
Seattle, WA :	9	Finance/Sales :	22

Figure 2: *The top 10 locations and industries in the dataset along with their counts.*

Queries:

1. "Medical assistant looking for work near New York City at a large hospital, 10 years of experience in patient care, including injections, CPR, and EKG testing"

2. "Web developer with experience in html, javascript, python, and Git, looking for work at startup in San Francisco, have portfolio website"

3. "New college grad with internship experience looking for software engineer position at large tech company"

4. "Looking for Senior Data Analyst or Data Scientist position at Fintech company in San Francisco"

5. "Looking for property inspector opening in New York with 5 years of experience"

6. "Real estate developer in greater NYC area, experienced in purchasing land and planning construction"

7. "Retail sales associate with 2 years of experience at Nordstrom, looking for work near Washington D.C."

8. "School counselor skilled in working with elementary aged children, open to relocation"

9. "School Superintendent in Boston with experience in leadership and strategic thinking, references available"

10. "Government affairs/public policy analyst looking for opening in Washington D.C. with deep knowledge of government regulations"

Figure 3: *The ten sample queries that were used to evaluate the search engine.*

3 Related Work

3.1 Job Search Engines

The main problems involved in building a job search engine include processing users' profiles, inferring characteristics such as experience with a given skill, and personalizing search results. LinkedIn, a generalized job search platform which has about 400 million user profiles and 6 million active jobs, allows job seekers or recruiters to search for jobs, potential candidates, published content, and other media on the site. Their main search engine uses a multistep framework that allows them to personalize searches to the user's intent and consider multiple verticals (open jobs, people, etc) when deciding which results should be shown to a user for a specific search [4]. They first predict a user's full skillset with a supervised machine learning algorithm using elements of their profile, including the text, the skills they have listed, and their endorsements. They factorize the resulting matrix into skills and users matrices and calculate dot-products to fill any missing entries. They then use learning to rank algorithms to rank results from each of the multiple verticals, each of which have their own search engine. Based on the query, a primary and secondary vertical are chosen by a federator, which takes into account multiple features including the user's intent based on past activity. The ranking scores within the two verticals are compared to produce the final results that are displayed to the user [8].

Another approach used by LinkedIn on their talent search platform involves using examples of ideal search results. Specifically, recruiters enter a search query looking for a candidate, and also supply profiles of people from their own company who would be ideal candidates. Their search engine uses information from the profiles of these ideal candidates and from profiles that have been co-viewed with them. The search engine balances showing ranking results focused on just the query keywords as well as the ideal candidate features [9].

Another difficulty in creating job search engines involves allowing users to enter larger amounts of text, as opposed to a few keywords into a search box, to better personalize searches. Muthyala et al. argue that short keyword searches combined with limited filters used on common job websites do not allow for users to search for niche roles or look for jobs with less common skills. They create a search engine that involves using a database of company attributes so that search results are more precisely suited to a job seekers skills and what they look for in a company. They use TF-IDF and support vector machines (incorporating context) to extract and rank skills from job postings, based on title and description. These skills are then matched through a database of companies to add more information to the company profiles, and the top companies are identified based on the user's query. Similarity measures are used to determine the rankings of job postings from among postings with the top selected companies [5].

When building a job search engine, it is also important to consider the behavior of users, such as how much effort they put into searching and how they construct queries. Mansouri et al. studied how user behavior when searching for jobs online differs from their behavior in general when using search engines. To conduct their experiments, they used a Persian general-purpose search engine (Parsijo) that contained information for 27 million queries and user interactions like clicks. Based on keywords in common queries and having students come up with common search terms they would use when looking for jobs, they collected a set of all job-related queries. They found that job related queries account for only 2% of all search queries, and by looking at queries per session, found that almost double the number of search queries were entered by users searching for jobs. The average number of clicks per session was also higher by a factor of 2.8 for job searches, and these users spent about 12 minutes longer than those using the engine for generalized searches on non-job related topics. They also found that job related queries are 2.1 terms longer, 22% of job seekers included the city where they are looking for work, and that 93.5% of queries containing temporal terms include terms relating to the current time as opposed to the future. Using annotators, they determined that for three search engines (Google, Bing, and Parsijo), user satisfaction for returned results was higher for general searches than for job related searches [10]. This last finding further highlights the need for a more personalized and efficient job search engine.

3.2 FastText and BERT Embeddings

FastText word embeddings are created using a neural network model that considers each word in a text as a sequence of characters. Using these character n-grams, the model learns embeddings using a skip-gram model in which it predicts surrounding words (context), given each word. Due to this approach, the model can produce embeddings for words that were not seen in the training data, unlike glove and word2vec embeddings [11]. However, these embeddings are limited to producing a single embedding for a word, regardless of how it can be used in various contexts.

BERT word embeddings came about using ideas from transformer models, which use feed forward neural networks and self attention layers within an encoder and decoder in order to learn the context of words within a sentence. BERT uses these same types of layers, but with only the encoder portion. Unlike other dynamic word embeddings, BERT uses a bidirectional model that reads the input text all at once, instead of sequentially. Prior to reading in text, 15% of the words are replaced with a masked token, and the encoder predicts the masked words based on the sequence of unmasked words. As a result the model can capture anaphora (abbreviated references of a word that has been introduced earlier in a text) and polysemy (different meanings of the same word), so that the same word used in a different context is given a unique representation [12]. Sentence-BERT uses siamese BERT networks to produce sentence or document embeddings by taking in two sentences at once and adding a pooling operation to BERT word embeddings in order to construct document embeddings [12].

Previous work, such as that by Guo et al., has proposed using a modified version of the BERT framework in search engines. Although BERT embeddings capture much more detailed semantic information as compared to traditional document scoring methods used in search engines, they are not computationally feasible in these settings. Guo et al. introduced the DeText approach which involves creating query and document embeddings separately, as opposed to producing them together as in the original BERT approaches applied to search engine settings. These separate embeddings are then compared to extract matching signals and rank documents. This results in a faster search time as BERT embeddings only for the queries need to be computed in real-time [3].

This paper aims to build a job search engine that addresses the computational inefficiencies of using BERT embeddings by using fastText embeddings for retrieving the top relevant documents, and then ranking this limited set using BERT embeddings. Furthermore, unlike LinkedIn, a more general search platform, I will be building a search engine solely for retrieving jobs. The search engine focuses on a user's skills and the factors most important to them in a future job, as they describe in their query. Most other job search engines only allow for keyword searches involving a short phrase and location, but this proposed engine will have users enter as much information as they choose, including their desired job titles and skills. As determined by Mansouri et al., this will be helpful for users since users tend to enter longer queries when looking for jobs than for other types of searches. The query will be entered into a single text box. Job postings will be processed using a separate semantic search for retrieval, and ranking of top job postings according to the user's query. Using a two layered semantic search will allow me to take advantage of the strength of BERT word embeddings to provide personalization while bypassing the high computational costs associated with calculating these embeddings.

4 Methodology

Using the ten sample queries, fastText embeddings are created for these and for all the job postings. The embeddings were created using python's Flair package: Fastetext word embeddings were computed for each word based on a pretrained model and then pooled by taking the mean of the embeddings to produce a document embedding. The pretrained model used was based on data from CommonCrawl and Wikipedia using a continuous bag of words (CBOW) model that predicts the probability of a word based on its context. Specifically, the CBOW uses position-weights, a dimension of 300, character n-grams of length 5, and a window size of 5 [14]. Before the fastText embeddings were pooled, they were fine-tuned on our job dataset by applying a nonlinear transformation, which was chosen over a linear transformation as it tends to give better performance for information retrieval tasks. The final fastText embeddings are represented as tensors, and the cosine similarity was calculated between all job postings and the ten queries using python's Pytorch library. For each query, the 15 job postings with the highest cosine similarity to the query were selected into a list.

For the top 15 postings for each query, BERT word embeddings were calculated based on two pretrained models using python's SentenceTransformers. The first pretrained model, 'distilbert-base-nli-stsb-mean-tokens' was trained on Natural Language Inference (NLI) data in which pairs of 570,000 sentences were labelled as 'entail', 'contradict', or 'neutral' [15]. The model was also fine-tuned on the STS benchmark dataset, which contains 8,628 sentence pairs from news related documents, captions, and forums, along with their semantic similarity scores [16]. The model uses DistilBERT, a modification of BERT that uses fewer parameters, but retains 97% of information captured by BERT embeddings while being 60% faster [17]. The DistilBERT model was the first model considered since in the context of a search engine, speed is essential.

The second pretrained model that was used was trained on the TREC Deep Learning 2019 dataset and the MS Macro Passage Reranking dataset, which were both based on passage and document ranking for queries, with labels for 367,000 and 1,000,000 queries respectively. This pretrained model, 'sentence-transformers/ce-ms-marco-electra-base', had the highest NDCG@10 based on models trained on those datasets [18]. It applies Electra, which uses a modified version of the original BERT model in which masked tokens are replaced by incorrect ones, so that the model can better distinguish between real and fake data [18].

Both models were also fine-tuned using a cosine similarity loss function with a training set of 170 job postings, with each posting-query pair having a label between 0 and 1 representing cosine similarity. The embeddings for the top 15 relevant job postings (as chosen by fastText) were computed from the tuned model, and the cosine similarity between the queries and postings was calculated. Job postings were then ranked in order of highest cosine similarity, were annotated on a scale of 0-4, and the NDCG@10 and precision@10 were calculated for each query.

For comparison purposes, since the job dataset was not too large, BERT document embeddings were calculated (using the DistilBERT pretrained model and fine-tuned on the same jobs training set) for all the job postings and ten queries. Due to memory and restraints on my laptop, I was not able to experiment with this method using the BERT model pre-trained on the TREC/MS Macro Passage data. Cosine similarity was then calculated between the queries and all jobs postings, and the top 15 results were annotated on the same scale to calculate NDCG@10 and precision@10. This was done to see how well the approach above, using BERT for just ranking, compares to using BERT for both relevance and ranking (which is not scalable to use with today's search engines). The ranking of jobs produced by the fastText embeddings was also evaluated using the same metrics in order to assess the improvement that the second layer of BERT embeddings provided.

5 Evaluation and Results

A BM25 baseline was created using python's rank-bm25 package. The inverted index was built using all words in the job description column of the dataset after converting all text to lowercase, lemmatizing, and removing stopwords. Scores for each document-query pair was calculated, and the top 15 highest scoring job postings for each of the ten queries was annotated. A random performance system, in which 10 job postings were randomly selected for a query, was considered as another baseline. Additionally, the ranking of jobs using fastText embeddings was evaluated to see how much of an improvement using BERT embeddings for ranking provided over using only fastText embeddings. Finally, a baseline using BERT embeddings (using the model pre-trained on NLI/STSB data) for both retrieval and ranking was evaluated. For all these baselines, the top 15 results were annotated.

To evaluate the two search engines against our baselines, two widely used metrics, NDCG@10 and precision@10 were calculated. As described in the methods section, the two-layered search engine was first built using a DistilBERT pretrained model based on NLI/STSB data, and then using a BERT pretrained model using the TREC/ MS Macro Passage Reranking data. In order to compute the IDCG portion of NDCG@10, all annotations for the top job posting-query pairs for all search systems (including the baselines) were aggregated. The NDCG@10 and precision@10 values are shown in Tables 1 and 2, respectively.

Comparing the performance of the two search engines, the model pre-trained on the NL/STSB data performs slightly better under both evaluation metrics. For most queries (excluding the first and the last), the first search engine has a higher NDCG@10, and a higher precision@10 for all queries. However, both almost consistently perform better than the BM25 baseline under both evaluation metrics. The exceptions where BM25 retrieves slightly better results is seen by looking at NDCG@10 values. Specifically, these exceptions occur at query 5 for the search engine pre-trained on the NLI/STSB data, and queries 4-6 for the search engine pre-trained on theTREC/MS Macro Passage Reranking data. Using BERT embeddings for both retrieval and ranking generally produced better results than both search engine models, with the exception of queries 2-5 (though for these queries, the NDCG@10 values were fairly close). When comparing the two search engine models to retrieval and ranking using fastText embeddings, the first search engine had a consistently higher precision@10, whereas the fastText embeddings approach held a higher NDCG@10 for a majority of queries.

NDCG@10

Query # **Model**

	Random	BM25	FastText	Search Engine (NLI and STSB)	Search Engine (MS Macro)	BERT
1	.092	.553	.752	.661	.683	.834
2	.088	.482	.830	.964	.702	.933
3	0	0.325	.523	.845	.392	.840
4	0	.742	.842	.749	.660	.643
5	0	.577	.657	.544	.489	.485
6	0068	.596	.863	.681	.568	.703
7	0	.526	.715	.692	.622	.735
8	0	.494	.632	.638	.578	.947
9	0	.444	.795	.809	.827	.918
10	.298	.468	.853	.818	.665	1

Table 1: *NDCG@10 values for the baselines and the two search engines.*

Precision@10

Query # **Model**

	Random	BM25	FastText	Search Engine (NLI and STSB)	Search Engine (MS Macro)	BERT
1	.1	.4	.7	.9	.8	.9
2	.1	.2	.8	1	.8	1
3	0	.1	.6	.8	.3	.8
4	0	.7	1	.8	.3	.5
5	0	.4	.5	.5	.3	.4
6	0	.3	.8	.8	.4	.4
7	0	.5	.7	.6	.5	.7
8	0	.3	.6	.7	.5	1
9	0	.3	.8	.9	.9	1
10	.1	.4	.8	.8	.7	1

Table 2: *Precision@10 and recall@10 values for the baselines and the two search engines.*

6 Discussion

It is interesting that using fastText embeddings for both retrieval and ranking performed better most of the time, in terms of NDCG@10, than both search engine models that incorporated BERT embeddings for ranking. The only exceptions are that the model pre-trained on NLI/STSB data performed better for queries 2, 3, 8, and 9, and the model pre-trained on MS Macro data performed better for query 9. As mentioned before, fastText embeddings cannot capture the detailed contextual information about words that BERT comprehends. The small size of the dataset or anomalies in annotations likely caused this unexpected result. Since collecting data was very time-consuming, I was not able to get as many job postings as I had planned. With less data, the top 10 postings (out of 15 chosen by fastText) and rankings may not produce as strong results as BERT models usually do, especially if there was vocabulary mismatched between the dataset used to pre-train the models and the job postings. This is likely since unlike the NLI/STSB andTREC/MS Macro data, which contained full sentences from news and captions, job postings that I collected were sometimes a set of incomplete phrases. Although I fine-tuned the search engines on a set of 150 job postings, this training set was small in comparison to the large size of the pre-training dataset, so the BERT embeddings produced were in part due to the language of the latter.

Moreover, since I was annotating all job postings myself, their labels were given according to my knowledge and assumptions about certain job skills and positions. For example, since I do not know much about real estate development or property inspection, for some job postings I had to make an educated guess on whether the job would be a good fit for the query. The two queries involving these types of jobs, queries 5 and 6, were incidentally places where fastText performed better than the search engines, and had two of the highest differences among the NDCG@10 values between this model and the two search engines. However, the NDCG@10 values for these three different searches were all within roughly .2 of each other, and a few were even closer. Additionally, the first search engine outperformed the fastText embeddings approach when considering precision@10. Considering these points, it is plausible that some vocabulary mismatches or labelling anomalies caused the higher performance of the fastText embeddings approach in some cases.

As expected the BM25 baseline and the BERT embeddings baseline performed worse and better, respectively, compared to the two search engines, under both evaluation metrics. Even with the shortcomings of this experiment as explained above, the BERT baseline was able to capture deep contextual information. This result agrees with those made by previous researchers comparing word embeddings to document scoring techniques in different settings.

Comparing the two search engines themselves, the model pre-trained on the NLI/STS data performed slightly better (excluding queries 1 and 9) than that pre-trained on the MS Macro data. This conclusion holds when considering both NDCG@10 and precision@10. This is somewhat surprising considering that the latter dataset was more similar to the current job retrieval setting: the dataset contained labels for documents according to how best they matched with a specific query. However, further research into natural language inference data showed that this type of data is often used to train a variety of models for different tasks including information retrieval, summarization, and question answering. Due to its nature (comparing sentences to each other), it tends to produce well-trained models [20].

Overall, I believe that the two-layered approach to building a job search engine incorporating BERT embeddings for ranking provides reliable results. It outperformed the BM25 baseline almost consistently, and outperformed using fastText for retrieval and ranking in four out of the ten queries (based on the model trained on the NLI and STS data) when considering NDCG@10. However, further experimentation of this approach on larger datasets must be conducted to test if this truly performs better than simply using fastText embeddings for both retrieval and ranking, and how it compares to using an approach with only BERT embeddings.

7 Conclusion

The two layer search engine explored in this paper combines the strength of BERT embeddings in information retrieval with computationally inexpensive, though less powerful fastText embeddings. Since due to their computationally expensive nature, BERT embeddings have not been able to be applied to large-scale search engines, the approach here attempted to use BERT embeddings for ranking the top results according to cosine similarity. I was able to determine that the approach performs better than the BM25 baseline, but due to the small size of the dataset, labelling anomalies, and mismatches between the training data and the jobs dataset, I could not ascertain whether incorporating BERT embeddings improves significantly enough on the ranking done by fastText embeddings to justify using them.

8 Other Things I Tried

Initially, I had built a search engine that allowed for users to input text in three different areas, one for location, one for skills (skills query), and another for any other information they wished to include, such as job title or parts of their resume (main query). The search engine then filtered jobs by location, and fastText embeddings were created for these job postings, along with the text the user had entered in the skills section. This separation between skills and other query items was proposed since having the right skillset is often the most important factor in finding a job, as opposed to job title or other query terms included by the user. The same python packages described above were used, and the cosine similarity between the skills query pieces and postings were calculated, and the top 15 postings were chosen. Next, the fine-tuned BERT model was used to create embeddings for the text in the main query section and the top 15 relevant job postings. The cosine similarity among these was used to choose the ranking of the jobs shown to the user. Unfortunately, this method was not producing accurate results, likely because the fastText embeddings for just the skills query terms was not enough to retrieve relevant results.

Once I had focused on just using a two-layered search that allowed a user to enter a single query, I attempted to use a Longformer model instead of BERT for ranking, as these newer types of models have been shown to work better for longer text documents. About 25% of my documents were over 500 words, so this seemed like a more accurate approach. However, using python's flair package to get document embeddings for each job posting required a large amount of memory, and I was not able to run these in jupyter notebook or Google colab.

I also attempted to pool BERT word embeddings to produce document embeddings (using the Flair package), as opposed to using the Sentence-BERT approach which I ended up implementing. I tried mean and maximum pooling of word embeddings, though neither produced results as well as the Sentence-Bert method.

9 Next Steps and What Could Have Been Done Differently

I believe that replacing BERT document embeddings with Longformer embeddings would make the proposed search engine more accurate. Longformer models use a modified version of the attention mechanism used in BERT in which tokens in a text can efficiently keep track of longer sequences. Beltagy et al., the creators of the longformer model, have shown that for documents with between 2,048 - 23,040 tokens, longformer embeddings outperform RoBERTa on tasks such as classification and question answering [21].

As the data collection and labelling process was time consuming, I was only able to train BERT embeddings on 170 job postings. I believe that using more training data to fine-tune the models would have produced better performance. With more data, more experimentation can be done to see if the approach mentioned in section 8, mainly allowing for separate text boxes to split up each user's search, may have better performance than just having a single search query. This approach may be especially useful to explore for people who are more interested in primarily matching their skills to a job, and using another query text box to include factors they would like to be incorporated secondarily.

References

1. Kochhar, Rakesh. "Unemployment Rose Higher in Three Months of COVID-19 than It Did in Two Years of the Great Recession." *Pew Research Center*, Pew Research Center, 26 Aug. 2020, www.pewresearch.org/fact-tank/2020/06/11/unemployment-rose-higher-in-three-months-of-covid-19-than-it-did-in-two-years-of-the-great-recession/.
2. Turczynski, Bart. "2020 HR Statistics: Job Search, Hiring, Recruiting & Interviews." *Zety*, 13 Oct. 2020, zety.com/blog/hr-statistics.
3. Gui, Weiwei, et al. "DeText: A Deep Text Ranking Framework with BERT." *Arvix*, LinkedIn, arxiv.org/pdf/2008.02460v1.pdf.
4. Ha-Thuc, Viet, and Shakti Sinha. "Learning To Rank Personalized Search Results In Professional Networks." *Arxiv*, arxiv.org/pdf/1605.04624.pdf.
5. "Data-Driven Job Search Engine Using Skills and Company Attribute Filters." *IEEExplore*, 2017 IEEE International Conference on Data Mining Workshops, 2017, ieeexplore-ieee-org.proxy.lib.umich.edu/stamp/stamp.jsp?tp=.
6. "What Are the Main Job Sectors in the U.S.?" *Indeed Career Guide*, www.indeed.com/career-advice/finding-a-job/main-job-sectors-in-the-us.
7. Saluja, Amandeep. "Extracting Job Information from LinkedIn Jobs Using BeautifulSoup and Selenium." *Amandeep Saluja*, amandeepsaluja.com/extracting-job-information-from-linkedin-jobs-using-beautifulsoup-and-selenium/.
8. Ha-Thuc, Viet, et al. "Personalized Expertise Search at LinkedIn." *Arxiv*, arxiv.org/pdf/1602.04572.pdf.

9. Ha-Thuc, Viet, et al. “Search by Ideal Candidates: Next Generation of Talent Search at LinkedIn.” *Arxiv*, arxiv.org/pdf/1602.08186.pdf.
10. Mansouri, Behrooz, et al. “Online Job Search: Study of Users' Search Behavior Using Search Engine Query Logs .” *SIGIR*, SIGIR'18, 12 July 2018, dl-acm-org.proxy.lib.umich.edu/doi/pdf/10.1145/3209978.3210125.
11. Ding, Xun. “FastText.” *Medium*, Towards Data Science, 24 Nov. 2018, towardsdatascience.com/fasttext-ea9009dba0e8.
12. Devlin, Jacob, et al. “BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding.” *Arxiv*, Google, arxiv.org/pdf/1810.04805.pdf.
13. Reimers, Nils, and Iryna Gurevych. “Sentence-BERT: Sentence Embeddings Using Siamese BERT-Networks.” *Department of Computer Science, Technische Universitat Darmstadt*, Ubiquitous Knowledge Processing Lab (UKP-TUDA), www.aclweb.org/anthology/D19-1410.pdf.
14. “Word Vectors for 157 Languages · FastText.” *FastText*, fasttext.cc/docs/en/crawl-vectors.html.
15. UKPLab. “UKPLab/Sentence-Transformers.” *GitHub*, github.com/UKPLab/sentence-transformers/blob/master/docs/pretrained-models/nli-models.md.
16. “STSbenchmark.” *STSbenchmark - Stswiki*, ixa2.si.ehu.eus/stswiki/index.php/STSbenchmark.
17. Sanh, Victor, et al. “DistilBERT, a Distilled Version of BERT: Smaller, Faster, Cheaper and Lighter.” *Arxiv*, HuggingFace, arxiv.org/pdf/1910.01108.pdf.
18. “Information Retrieval.” *Information Retrieval - Sentence-Transformers Documentation*, www.sbert.net/examples/applications/information-retrieval/README.html.
19. “Google/Electra-Base-Discriminator · Hugging Face.” *Hugging Face – On a Mission to Solve NLP, One Commit at a Time.*, huggingface.co/google/electra-base-discriminator.
20. Stanford Natural Language Processing Group, and Sam Bowman. “The Stanford NLP Group.” *The Stanford NLI Corpus Revisited - The Stanford Natural Language Processing Group*, 25 Jan. 2016, nlp.stanford.edu/blog/the-stanford-nli-corpus-revisited/.
21. Beltagy, Iz, et al. “Longformer: The Long-Document Transformer.” *Arxiv*, Allen Institute for Artificial Intelligence, Seattle, WA, USA, arxiv.org/pdf/2004.05150.pdf.