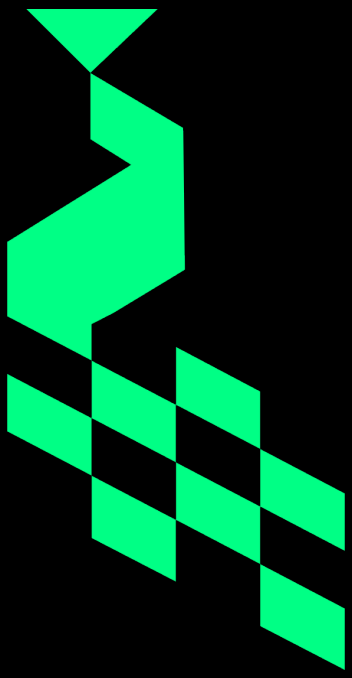
DevRev's Expert Answers in a Flash: Improving Domain-Specific QA

Mid Evaluation Report

Team 53



Inter IIT Tech Meet 11.0

1 Introduction

Question Answering (QA) involves predicting the answer to a question given a context document. QA can be broadly classified into two types - abstractive and extractive [1]. In abstractive QA, the model is largely tuned to generate natural language answers to open-ended questions, which may or may not be similar to the target answer. Such a scheme is usually used to develop chatbots that are supposed to answer a wide variety of questions from different domains. In extractive QA, the model is tuned to give specific answers from the context documents and is usually used for the targeted answering of well-defined closed context questions.

The given problem statement deals with extractive QA, with a strong focus on efficiency and scalability. The target is to determine whether the question can be answered given the context corpus and find the target paragraph and specific answer from the corpus if the question is determined to be sufficiently answerable. Our proposed solution is a general-purpose pipeline containing a search algorithm to find top candidate context paragraphs for a given question, which is connected to a lightweight, optimized transformer model pre-tuned for extractive QA, considering the space and time constraints of the problem statement. The QA model, besides extracting the correct answer from the context paragraph, also indirectly determines whether the question is answerable or not by returning an empty answer in the latter case. Our solution[1] is built on a highly modular approach, which additionally offers ease of upgradability apart from noticeable performance gains that impart scalability to our solution. Such a modular solution also enables benchmarking against different search algorithms as well as QA models.

2 Proposed Solution

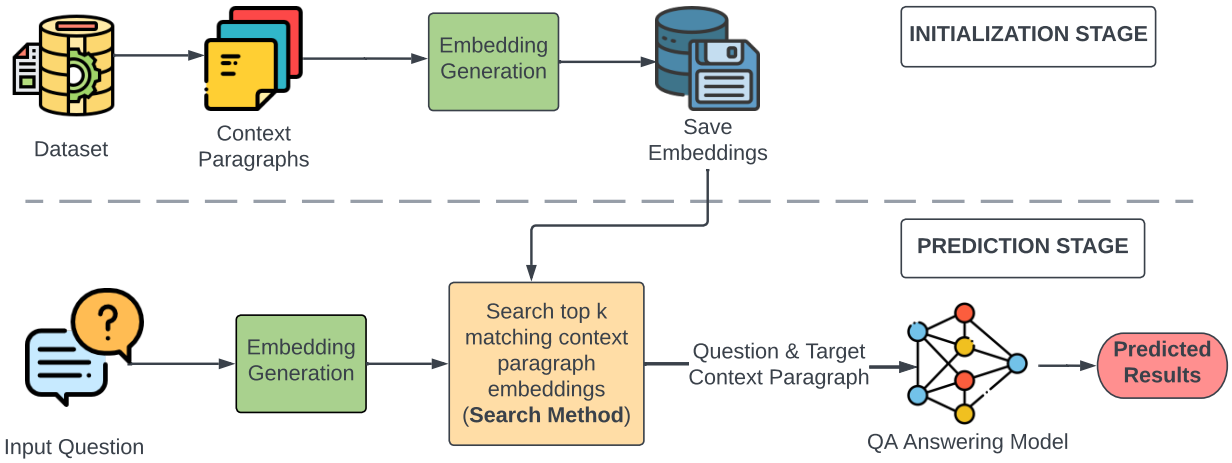


Figure 1: Overall flow diagram of the proposed solution.

Figure 1 shows the overall flow structure of our solution. Two modules are used in the overall pipeline - **a similarity-based search method** for searching the target context paragraph from the overall corpus, and a lightweight optimized **transformer-based QA model** are the two stages involved in obtaining the results from our pipeline. The first stage is the Initialization stage, which is required to be run only once for a given context corpus. In this stage, embeddings are generated from a custom embedding model and cached for use during the prediction process. This step is done initially as the time taken for generating embeddings for the context corpus is quite high, and since the context corpus is universal and static, it is prudent to generate them beforehand. Even in a practical scenario, the updation of the context corpus happens only periodically. Thus the initialization stage can be run in a similar periodic manner for new context data.

Once the initialization has been done, the Prediction stage is enacted. In this stage, the given questions are encoded by the embedding model in the same manner as the context corpus. Then, the search model is applied over the context

[1] Our coding files can be accessed from the following link - Drive Folder

corpus embeddings generated in the initialization stage to find the top matches for the target context paragraph. This process ensures that only the relevant context input is passed to the QA model. Since the size of the context input has a direct impact on the inference speed of the model, the inference time is thus reduced by the application of this method. The QA model performs extractive QA to find the correct answer from the context paragraph fed into it or returns a null value if the input question is not answerable. Thus, the task of determining whether a question is answerable or not is also determined at this very stage itself.

3 Comparative experiments

3.1 Search Methods

We experiment with two different search methods: Facebook AI Similarity Search (FAISS) [2] and Approximate Nearest Neighbors Oh Yeah (ANNOY) [3]. In either of the cases, the search algorithm is used to find the target context paragraphs from a theme, given an input question corresponding to that theme. The target context paragraphs are encoded by an embedding model in the initialization stage, as mentioned in Section 2, whereas the input question is encoded during the prediction stage. Once both the input question and the context paragraphs are encoded, the appropriate search method is applied to find the target context paragraphs that may contain the answer. The use of a search algorithm is necessitated by the fact that the inference time of the QA model rises proportionally to the length of the context fed to it, and reducing the search space ensures that the inference time of the QA model is within acceptable limits. Also, as the pretrained transformer models can only take in a fixed input length, usually limited to 256 or 512 characters, the entire context corpus cannot be fed to the model as-is.

3.1.1 Comparative results

Table 1: Comparative results for different embedding models and similarity methods within FAISS on questions belonging to **New_York_City** theme having 145 context paragraphs, taken from the provided training dataset.

Embedding Model	**Encode Time per Paragraph (ms)**	**Encode Time per Question (ms)**	**Search Method**	**Top 1 hit**		**Top 2 hit**	
				Search Time per Question (µs)	**Accuracy**	**Search Time per Question (µs)**	**Accuracy**
			FlatL2	2.04	84.01 %	2.89	92.00 %
all-MiniLM-L12-v2 [A]	120.81	19.24	LSH	72.62	84.01 %	69.28	92.00 %
			HNSW	26.92	84.01 %	27.02	92.00 %
			FlatL2	2.11	84.72 %	3.05	93.25 %
all-MiniLM-L6-v2 [B]	**88.44**	10.59	LSH	70.29	84.72 %	74.02	93.25 %
			HNSW	28.43	84.72 %	30.40	93.25 %
			FlatL2	**1.93**	**82.59 %**	**2.71**	**91.47 %**
multi-qa-MiniLM-L6-cos-v1 [C]	133.17	**9.50**	LSH	90.71	82.59 %	73.12	91.47 %
			HNSW	27.20	82.59 %	26.08	91.47 %
			FlatL2	**3.67**	**86.67 %**	**4.21**	**95.20 %**
multi-qa-distilbert-cos-v1 [D]	403.32	33.85	LSH	242.24	86.67 %	254.54	95.20 %
			HNSW	30.84	86.67 %	31.17	95.20 %
			FlatL2	3.52	86.32 %	4.73	94.84 %
all-distilroberta-v1 [E]	419.90	33.11	LSH	253.00	86.32 %	255.90	94.84 %
			HNSW	32.57	86.32 %	31.26	94.84 %

Table 1 lists down the metrics obtained on different combinations of embedding models and search methods within FAISS. Accuracy is defined as the percentage of correctly predicted context paragraphs for a given question. In the case where the top two target paragraphs are returned by the search method, a correct prediction is considered if either of the two paragraphs matches the correct one. We test three search methods offered by FAISS - FlatL2 (compares L2 distance between the input and the target), Local Sensitive Hashing (LSH) [4] and Hierarchical Navigable Small Worlds (HNSW) [5]. For ANNOY, we use three similarity metrics to compare the embeddings - Angular (Cosine Similarity), Euclidean (L2), and Dot (Dot Product). In Table 2, it can be noticed the search time per question for ANNOY is more than the corresponding setting in FAISS. The accuracies in predicting similar paragraphs, however, are almost similar to FAISS. Though it may seem that FAISS is a better choice for the search method than ANNOY given the metrics on the 'New_York_City' theme, the performance of FAISS drastically drops on certain themes such as 'Beyoncé'. We found a drop of around 35% for FAISS when run on question-context pairs on 'Beyoncé', whereas the drop for ANNOY is around 10%. Hence we test our pipeline for both search models to be conclusive about the

better search method of the two. Also, as ANNOY builds a data corpus to parallelize the search, the lag in the search times for ANNOY with respect to FAISS is expected to get obviated once a large dataset is under evaluation. Also, we have presented the results considering only top 1 and top 2 hits since we observed only marginal improvements in the accuracy at the cost of high model inference time on increasing the number of hits by the search algorithm.s

Table 2: Comparative results for different embedding models and search methods within ANNOY on questions belonging to **New_York_City** theme having 145 context paragraphs, taken from the provided training dataset.

Model	**Encode time per Paragraph (ms)**	**Encode time per Question (ms)**	**Search Metric**	**Top 1 hit**		**Top 2 hit**	
				Search Time per Question (µs)	**Accuracy**	**Search Time per Question(µs)**	**Accuracy**
all-MiniLM-L12-v2 [A]	120.8078	19.2426	Angular	101	84.01 %	113	92.01 %
			Euclidean	106	84.01 %	107	92.01 %
			Dot	109	84.01 %	110	92.01 %
all-MiniLM-L6-v2 [B]	**88.4419**	10.589	Angular	107	84.72 %	119	93.25 %
			Euclidean	105	84.72 %	118	93.25 %
			Dot	113	84.72 %	206	93.25 %
multi-qa-MiniLM-L6-cos-v1 [C]	133.1717	**9.4965**	Angular	101	82.59 %	112	91.47 %
			Euclidean	**98**	**82.59 %**	**106**	**91.47 %**
			Dot	117	82.59 %	109	91.47 %
multi-qa-distilbert-cos-v1 [D]	403.3185	33.8513	Angular	166	86.68 %	177	95.20 %
			Euclidean	165	86.68 %	174	95.20 %
			Dot	**163**	**86.68 %**	**170**	**95.20 %**
all-distilroberta-v1 [E]	419.9002	33.1108	Angular	176	86.32 %	183	94.85 %
			Euclidean	192	86.32 %	193	94.85 %
			Dot	180	86.32 %	191	94.85 %

3.2 Transformer-based QA model

For the QA task, we use transformer models [6] pretrained on the task of QA. We initially evaluated around 50 odd QA transformer models, which were finetuned on the SQuAD v2 dataset, post which we zeroed onto the best five amongst them. All model checkpoints were used from the HuggingFace repository [7]. The following subsections list the different optimizations, and comparative evaluations carried out on the same.

3.2.1 Model Optimization

An important disadvantage that is inherent to the transformers is the large size of the model and slow inference speed, which inhibits their use in low compute scenarios where scalability and efficiency are major concerns, such as the one envisioned in this problem statement. To overcome this issue, we optimize our models with the use of pruning and quantization. **Pruning** is a technique of removing unnecessary weights and biases on a global or a local layer-level scale to compress the model size, which can then be used effectively for deployment. **Quantization** is another optimization technique that is often used to compress deep models. In this process, the data type of the model weights, which are almost always of type *float32/float16*, are downsampled to a low accuracy type such as *int8/uint8*. In either of the optimizations, there is a relative drop in the performance of the model depending on how the quantization is performed; however, the speed and low size gains obtained often outweigh this performance drop.

For our task, we use two different libraries to perform optimization on the existing pretrained QA models. The first set of libraries is native to PyTorch and are *torch.nn.utils.prune* and *torch.quantization*. The second library is the Intel Neural Compressor [8], which is a unified solution for applying model compression algorithms by a dynamic, automatic accuracy-driven tuning strategy.

3.2.2 Comparative Results for different QA Models

Table 3 details the different QA models tested with different optimization strategies. The evaluation dataset is a random split of the provided training dataset, containing 10,000 questions. The class weighting was removed from the evaluation script in order to standardize the results for adjudging the optimal model for our pipeline. Optimizations labeled as **Native** refer to the optimizations applied via the Pytorch-native libraries, whereas those labeled as **Dynamic** refer to those optimizations carried out via Intel Neural Compressor. Further, all models were evaluated in the default

Table 3: Comparative results for different QA models tested on a random split of provided training dataset containing 10k examples.

Pretrained QA Model	Optimization Used	Dataset Originally Finetuned On	F1 Score	Average Inference Time per Question (ms)	Model Checkpoint Size (MB)
MiniLM [9]	None	SQuAD v2	71.2%	195	126
	Native Pruning		69.9%	168	126
	Native Quantization		69.4%	179	66
	Dynamic Quantization		**70.5%**	**175**	**67**
	Dynamic Pruning		**71.2%**	**182**	**126**
ELECTRA (base) [10]	None	SQuAD v2	73.5%	622	436
	Native Pruning		68.3%	522	415
	Native Quantization		38.1%	350	172
	Dynamic Quantization		39.6%	430	173
ELECTRA (small) [10]	**None**	SQuAD v2	**72.4%**	**88**	**54**
	Native Pruning		69.8%	95	52
	Native Quantization		71.9%	101	52
	Dynamic Quantization		**72.0%**	**100**	**25**
BERT (base) [11]	None	SQuAD v2	72.6%	588	411
	Native Pruning		66.0%	1351	433
	Native Quantization		70.4%	461	168
	Dynamic Quantization		71.0%	454	169
TinyBERT [12]	None	SQuAD v2	72.3%	311	253
	Native Pruning		72.1%	304	473
	Native Quantization		72.0%	191	132
	Dynamic Quantization		72.1%	222	132

setting, without applying any thresholding level to the predictions. Since the models were finetuned on the SQuAD v2 dataset, this process ensured an accurate evaluation of their baseline.

It can be noticed that the best F1 score is given by ELECTRA-base; however, the average inference time per question is quite large. ELECTRA-small, which has fewer parameters than the base model, has a highly optimal inference time while retaining the base model's performance. TinyBERT can also yield relatively high F1 scores but at the cost of large inference time.

While the optimizations, in general, reduce the inference time of the models, the actual effect depends on the model architecture itself. For example, the optimizations cause detrimental effects on the performance of ELECTRA-small/base and BERT. However, for MiniLM and TinyBERT, the optimizations lead to a substantive reduction in the inference time without causing a massive reduction in F1 scores.

Considering the overheads of the search and embedding modules, ELECTRA-small without any optimization is determined to be the most optimal model, both in terms of F1 scores and inference time per question. However, in order to comprehensively evaluate the performance of our solution pipeline with the QA models, we consider two of the best models with different optimizations - MiniLM (dynamic pruning and dynamic quantization) and ELECTRA-small (base and dynamic quantization) to benchmark their performance with different embedding and search methods.

4 Results

We use the random sample of the training dataset containing 10k questions across several themes for evaluating our pipeline. This is the same dataset used for evaluating QA models in Section3.2.2. We calculate the QA score, para score, average inference time, and F1 score on the entire pipeline. Kindly note that we do not perform the evaluation on the entire train dataset due to the high runtimes associated with each inference run, which limited our ability to perform a comprehensive study on different combinations of search methods and QA models. However, the metrics determined from the subset are bound to be representative of the performance of our pipeline on the entire training dataset to a high degree.

We benchmark results on two QA models - ELECTRA-small and MiniLM, and the best search schemes under FAISS and ANNOY. For the embedding models, we use the top two models by virtue of their encoding time (all-MiniLM-L6-

Table 4: Results on the entire pipeline for **top 1 hits** obtained on a random split of provided training dataset containing 10k samples. The embedding models correspond to the codes mentioned against the embedding models in Tables 1,2.

QA Model →		ELECTRA-small (base) [threshold = 0.10]				ELECTRA-small (D.Q.) [threshold = 0.13]				MiniLM (D.Q.) [threshold = 0.09]				MiniLM (D.P.) [threshold = 0.11]			
Search Method ↓	**Embedding Model ↓**	**F1**	**Infer Time**	**Para Score**	**QA Score**	**F1**	**Infer Time**	**Para Score**	**QA Score**	**F1**	**Infer Time**	**Para Score**	**QA Score**	**F1**	**Infer Time**	**Para Score**	**QA Score**
FAISS	[B]	51.6%	172.7	47.5%	46.7%	51.1%	106.0	48.8%	47.8%	50.3%	173.4	50.0%	45.2%	50.8%	181.2	48.3%	45.0%
	[C]	49.8%	203	41.4%	40.7%	49.3%	106	47.2%	46.2%	48.5%	204.1	45.9%	41.1%	49.1%	175	47.1%	44.1%
	[E]	54.3%	139.0	41.4%	40.7%	53.8%	138.1	51.6%	50.3%	53.2%	199	51.4%	45.9%	53.7%	239	44.6%	41.2%
ANNOY	[B]	51.6%	124.0	48.8%	48.1%	51.0%	110.0	48.8%	47.7%	50.8%	191.2	47.6%	44.4%	50.4%	167.5	50.4%	45.4%
	[C]	49.8%	127.0	47.2%	46.5%	49.3%	119.0	47.2%	46.2%	49.1%	213.3	43.3%	40.4%	48.6%	176.8	48.4%	43.5%
	[E]	**54.3%**	**135.1**	**51.7%**	**50.7%**	53.8%	146.2	51.3%	50.0%	53.7%	232.0	45.8%	42.3%	53.1%	184.0	53.6%	47.7%

D.Q. = Dynamic Quantization, D.P. = Dynamic Pruning. All times are in milliseconds (ms).

Table 5: Results on the entire pipeline for **top 2 hits** obtained on a random split of provided training dataset containing 10k samples. The embedding models correspond to the codes mentioned against the embedding models in Tables 1,2

QA Model →		ELECTRA-small (base) [threshold = 0.10]				ELECTRA-small (D.Q.) [threshold = 0.13]				MiniLM (D.Q.) [threshold = 0.09]				MiniLM (D.P.) [threshold = 0.11]			
Search Method ↓	**Embedding Model ↓**	**F1**	**Infer Time**	**Para Score**	**QA Score**	**F1**	**Infer Time**	**Para Score**	**QA Score**	**F1**	**Infer Time**	**Para Score**	**QA Score**	**F1**	**Infer Time**	**Para Score**	**QA Score**
FAISS	[B]	56.3%	331	34.5%	32.7%	**55.6%**	**190**	**50%**	**47.7%**	55.8%	355	34.3%	30.2%	56.8%	367	32.7%	29.7%
	[C]	54.9%	424	26.5%	27.9%	54.2%	197	47.6%	45.4%	54.5%	391	30.6%	26.7%	55.3%	414	28.1%	25.6%
	[E]	58.4%	241	46.8%	44.6%	57.7%	215	49.7%	47.2%	58.1%	392	32.4%	28.4%	58.8%	450	27.4%	24.9%
ANNOY	[B]	56.3%	252	43.5%	41.4%	55.6%	200	49.1%	46.7%	56.1%	367	32.7%	29.7%	56.8%	383	31.4%	28.5%
	[C]	54.9%	315	35.2%	33.5%	54.2%	183	49.2%	47%	54.5%	344	34.6%	30.2%	55.3%	426	27.3%	24.8%
	[E]	58.4%	451	25.7%	24.4%	57.7%	247	45.8%	43.3%	58.2%	311	40.3%	35.2%	58.8%	436	28.3%	25.6%

D.Q. = Dynamic Quantization, D.P. = Dynamic Pruning. All times are in milliseconds (ms).

v2 [B] and multi-qa-MiniLM-L6-cos-v1 [C]) and the best model by virtue of the prediction accuracy (all-distilroberta-v1 [E]). Also, in the search method, we obtain results for both the top 1 and top 2 hits of the context paragraph obtained for the input question belonging to a certain theme. In order to improve the performance of our QA models, we additionally enforce **thresholding** over the confidence scores of the predictions in order to ensure that unanswerable questions are correctly identified by our models. Figure 2 shows the variation of the F1 score of the QA model with an increasing threshold. The green points show the threshold determined as optimal for the different models.

Table 4 lists down the results obtained for the top 1 hits performed by the search method. It can be noticed that the best metrics are obtained for the ANNOY search method, all-distilroberta-v1 [E] embedding model, and ELECTRA-small QA model without any optimization **(F1 - 54.4%, Infer time per question - 135.1ms, Para Score - 51.7% and QA score - 50.7%)**. The drop in the F1 score for the ELECTRA-small model, when compared to the evaluation in a standalone setting, is due to the errors induced by the search algorithm while predicting the target context paragraph. The inference time is well under the limit of 200ms imposed by the problem statement, which reinforces our claim that the pipeline is efficient and scalable. The accuracy metrics are more or less the same for both FAISS and ANNOY for all four models in general; however, FAISS has a higher computational overhead, evident from the relatively larger inference times.

Table 5 lists the results obtained when the search method is configured to return the top 2 hits over the context corpus for a given question. The results are worse than those obtained under top 1 hits, with quite a large inference time and poor metric scores. Under this setting, ELECTRA-small optimized using Dynamic Quantization (D.Q.) gives the best results among the different models, with FAISS as the search method and all-distilroberta-v1 [E] as the embedding model. Another inference to be made is that ANNOY performs worse than FAISS in returning the top 2 hits. Thus, considering the two results, it can be concluded that only when the search method returns top 1 results are when our pipeline gives the most optimal results.

5 Trade-offs and Limitations

The given problem statement imposed several restraints on the development environment, which necessitated a highly scalable and efficient model which is optimized for fast inferential capabilities. As a result, we developed a modular pipeline on similar lines, which can be easily run on a compute-restricted environment. However, there were certain tradeoffs and limitations that we had to consider to make our pipeline adhere to the constraints of inference time and

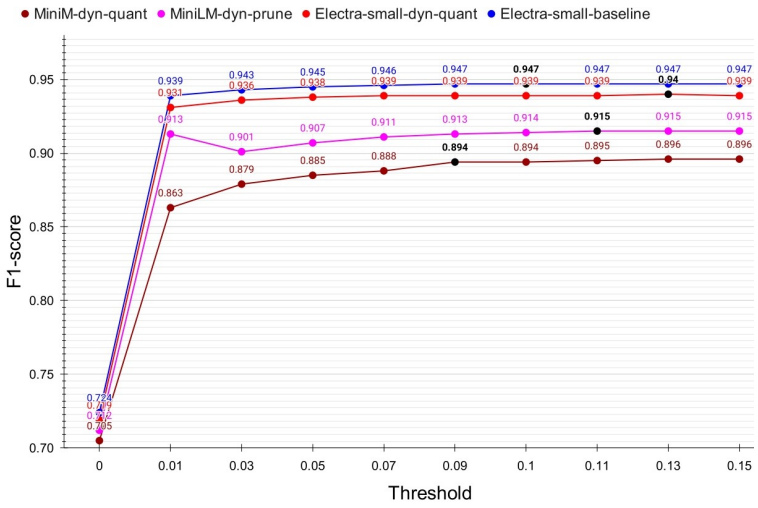


Figure 2: Plot of F1 score v/s threshold for the top 4 models considered for building the pipeline

RAM usage. These are elucidated as under:

- The generation of context paragraph embeddings by the embedding model within the FAISS ecosystem required a relatively large amount of time, of the order of seconds. This meant that we had to generate the context embeddings beforehand in order to remove this overhead during the inference time, which we did by defining an initialization stage as a part of our solution. The trade-off that arises from this approach is the extra utilization of RAM required to cache the embeddings of the context paragraphs so that during the inference time, the only overhead to emerge would be the time taken to encode the question and perform the search, both of which contributed minorly to the overall inference time of our pipeline.
- The restrictions on inference time meant that even after applying the optimization methods such as pruning and quantization, we could only use moderately deep models. The metrics currently obtained by us could be improved further by using state-of-the-art models such as PaLM [13], LUKE [14] and ATLAS [15], so as to mention a few. All of them are highly deep models having billions of parameters.

6 Future Works

While the modularity and scalability of our solution pipeline ensure that the objectives enlisted as part of the problem statement are achieved, there are certain developments that we aim to experiment and apply to our existing pipeline in order to boost the performance without affecting the inference speed. There were several other search algorithms that we couldn't comprehensively test, such as Google's Scann [16] and Apache's Lucene [17], both of which are highly efficient search algorithms. We aim to compare these algorithms with FAISS and ANNOY to find the most optimal algorithm for our purpose.

Another direction that can be followed up is the use and optimization of an ensemble of QA models, where predictions of individual QA models are combined by the use of a suitable aggregation function. A major challenge in this approach is to effectively apply the optimizations to reduce the inference overhead of using multiple learners since the entire ensemble would have to be jointly optimized to ensure the identification of irrelevant weights across different learners. However, the obtained performance metrics are hypothesized to be better than a standalone QA model.

Lastly, we aim to attempt this problem statement by using knowledge graphs (KG) [18]. The use of KGs is an emerging topic in NLP research as they are known to exhibit improved predictive power over transformers. Our objective is to create a KG given the context corpus as a part of the training set, implement a ranker to extract the correct subgraph containing the answer, and a generator that can utilize the existing KG to answer questions that the pipeline has never seen before.

References

[1] Anthony Chen, Gabriel Stanovsky, Sameer Singh, and Matt Gardner. Evaluating question answering evaluation. In Proceedings of the 2nd workshop on machine reading for question answering, pages 119–124, 2019.

[2] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. IEEE Transactions on Big Data, 7(3):535–547, 2019.

[3] Spotify. ANNOY library. `https://github.com/spotify/annoy`.

[4] Moses S Charikar. Similarity estimation techniques from rounding algorithms. In Proceedings of the thiry-fourth annual ACM symposium on Theory of computing, pages 380–388, 2002.

[5] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. IEEE transactions on pattern analysis and machine intelligence, 42(4): 824–836, 2018.

[6] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. Advances in neural information processing systems, 30, 2017.

[7] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Huggingface's transformers: State-of-the-art natural language processing. arXiv preprint arXiv:1910.03771, 2019.

[8] Feng Tian, Haihao Shen, Huma Abidi, and Chandan Damannagari. Pytorch inference acceleration with intel® neural compressor, Jun 2022. URL `https://medium.com/pytorch/pytorch-inference-acceleration-with-intel-neural-compressor-842ef4210d7d`.

[9] Wenhui Wang, Furu Wei, Li Dong, Hangbo Bao, Nan Yang, and Ming Zhou. Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers. Advances in Neural Information Processing Systems, 33:5776–5788, 2020.

[10] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. Electra: Pre-training text encoders as discriminators rather than generators. arXiv preprint arXiv:2003.10555, 2020.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805, 2018.

[12] Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. Tinybert: Distilling bert for natural language understanding. arXiv preprint arXiv:1909.10351, 2019.

[13] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. arXiv preprint arXiv:2204.02311, 2022.

[14] Ikuya Yamada, Akari Asai, Hiroyuki Shindo, Hideaki Takeda, and Yuji Matsumoto. Luke: deep contextualized entity representations with entity-aware self-attention. arXiv preprint arXiv:2010.01057, 2020.

[15] Gautier Izacard, Patrick Lewis, Maria Lomeli, Lucas Hosseini, Fabio Petroni, Timo Schick, Jane Dwivedi-Yu, Armand Joulin, Sebastian Riedel, and Edouard Grave. Few-shot learning with retrieval augmented language models. arXiv preprint arXiv:2208.03299, 2022.

[16] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating large-scale inference with anisotropic vector quantization. In International Conference on Machine Learning, 2020.

[17] Apache. LUCENE library. `https://lucene.apache.org/core/`.

[18] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard de Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, et al. Knowledge graphs. ACM Computing Surveys (CSUR), 54(4):1–37, 2021.