Open Domain QA with Sparse and Dense Representations

Ceresini Marcello, Cichetti Federico
Department of Computer Science and Engineering
Alma Mater Studiorum Università di Bologna
`mceresini.97@gmail.com, federico.cichetti@studio.unibo.it`

Abstract

Open-domain Question Answering (OpenQA) is an increasingly important task in the field of Natural Language Processing (NLP) and consists in answering a natural language question with an answer extracted from a document within a large set of unstructured documents.

Even its simpler sibling, Machine Reading Comprehension (MRC), incurs in the difficult challenge of answering precisely to any of the questions asked by the user, but factoring in the complexity of also finding the right context, sometimes amongst tens of thousands or millions of documents, the task becomes quite challenging.

In the last few years, more and more research has been done on new techniques that try to tackle this problem. In this paper, we focus on the *Information Retrieval (IR)* part of the problem. In particular, we recreate and train a model called *Dense Passage Retriever (DPR)* [1] that is used to generate dense encodings of contexts offline and then to retrieve the ones that are most similar to the encoding of the question in real time.

In addition, we also explore a set of new methods that combine both sparse and dense representation to achieve better results. The sparse representations show to be a complementary tool to the use of dense representations. In fact, simply through a weighted average of the two methods' scores we can achieve higher results than either of the two methods alone. This means that the two methods exploit strong and radically different features in the text, making them the ideal candidates for a simple but effective ensembling procedure.

1 Introduction

1.1 Disclaimer

Since this is the reproduction of a state-of-the-art paper, most of the intuitions are a direct inspiration from [1] by Facebook AI, Washington and Princeton Universities.

1.2 Open Domain Question Answering

Question Answering (QA) has the goal to provide exact answers to a generic question made by the user in natural language. Usually, the answer is returned directly as a string of text taken directly from a larger document. Depending on the form and the size of contextual information, we can divide QA in two sub categories: *Machine Reading Comprehension (MRC)* and *Open-domain QA (OpenQA)*.

MRC is based on the assumption that the answer to the question is a snippet of text in a specified context/document, given in input together with the question. This simplifies considerably the problem, that instead of being a generative task becomes a simple matter of selecting the right starting and ending word for the answer in its context.

OpenQA extends MRC by trying to answer a given question *without* any specific input context. In this way, the system is firstly required to search the relevant documents and only afterwards generate a candidate answer. OpenQA can therefore aim to a much wider range of possible applications.

1.3 Traditional and modern methods

The traditional architecture used by OpenQA systems consists of three separate stages:

1. *Question Analysis*: starting from a question in natural language, the goal of Question Analysis is to examine it in order to improve the performance of the following two stages, by extracting possible queries to help Document Retrieval or by classifying a possible answer structure/type to feed into the Answer Extraction stage.
2. *Document Retrieval*: this stage searches between all the available documents (usually using Information Retrieval (IR) techniques) to find which can be the one or ones more likely to contain the answer. The biggest problem faced by Document Retrieval is that many questions do not utilize the same words or tokens that are contained in the given documents: this is often referred to as "*term mismatch*".
3. *Answer Extraction*: The goal of this latest stage is that of "reading" through the proposed paragraphs and restricting the parts of the text that could contain the answer. Being the final part of the problem, it's the one that is most heavily influenced by the previous steps: a bad Question Analysis will produce ineffective question representations, while an incorrect Document Retrieval will heavily limit the possibility of the answer to be found at all.

Nowadays, most Question Answering applications are actually based on Deep Learning, which reduces the importance or removes altogether the need of a separate Question Analysis step: an end-to-end system can instead learn to generalize mappings from any type of Natural Language question to any type of answer. Furthermore, Deep Learning allows the critical "term mismatch" problem to be overcome: by using dense representations to store document and question contents, the program is able to correctly identify questions and answers that are related even if they use synonyms or even completely different words for the same concept.

In reality, it's difficult to build completely end-to-end systems; the most common architecture for OpenQA nowadays is the so-called *Retriever-Reader* architecture, where a model is built of (at least) two independent modules: one specialized in retrieving the optimal document(s) given the question (*Retriever*) and the other expert at reading this (or this set of) document(s) and finding the span of text answering to the question (*Reader*).

It has been shown that models that achieve the best results combine both old and new-school methods: for example, in [2], the model solves the Information Retrieval stage by utilizing a sparse encoder, Tf-Idf, while using a neural model for the QA stage. Instead, in the work that was the main inspiration for our project [1], both the Retriever and the Reader stages are undertaken through the use of only neural models. We chose to implement this second model and at the same time to exploit the strength of both approaches, utilizing both dense and sparse encoders for the Information Retrieval stage, but keeping a neural-only approach for the QA stage.

1.4 Dataset

Following the procedures that we used in our last work, we decided to stick to SQuAD as the source for documents and questions. This allowed us to keep all the previous knowledge of the data, the preprocessing and loading methodology, and instead focus on the new model to tackle the OpenQA task.

However, as mentioned in [1], SQuAD has some intrinsic problems when it comes to OpenQA. In particular, annotators were given the task to write questions that could be answered *from the given text*. This implies that many questions do not posses any specific context without their linked paragraph, for example: *'Most of* ***the city*** *lies in which USDA plant hardiness zone?'*. Which city is the annotator writing about? Other more general QA datasets that contain more naturally formulated questions, such as Natural Questions [3] or WebQuestions [4], exist and may be subject of future analysis.

2 Dense Passage Retriever

2.1 Overview

Following the ideas in [1], we set our main focus to the Document Retrieval stage, while utilizing the QA tools that we developed in our last work to complete the Answer Extraction stage. The Answer Extraction module selects as a possible answer only a span of text in the context given as input together with the question: this means that it's actually impossible for our system to give an exact answer when provided with the wrong context. Our goal then becomes to maximize the Retriever accuracy, thus allowing our QA module to work at its maximum performance.

We implemented the Dense Passage Retriever model [1] to solve the Information Retrieval stage. It's a simple, yet effective module that consists of two encoders: one specialized in creating low-dimensional embeddings for queries (questions), while the other is specialized on paragraphs embeddings. The learning problem can then be formalized as a *Metric Learning* problem aiming to learn a common hyperspace where each question's encoding ends up as close as possible to the related paragraphs' encodings. Closeness in this space has the meaning of semantic similarity and can be computed with a simple *dot product* between vectors.

The fact that we only need a dot product to find the best-matching context means that, once the model is trained, all of the contexts can be transformed into their corresponding encoding and stored in memory, and only the question has to be processed before computing the dot product. In the case of massive amounts of documents, this means that the inference can be massively parallelized and can be executed in real time.

Formally, the dense encoders E_P and E_Q respectively map a passage p and a question q to real-valued vectors with d dimensions. At run time, an input question is passed through E_Q to compute the question vector, and then the most similar passage vector is retrieved, with similarity defined as the simple dot product between the two vectors:

$$sim(q,p) = E_Q(q)E_P(p)^T$$

Among the many similarity measures we could use, we chose the dot product for mainly two reasons:

1. It is completely *decomposable*: this means that no interaction is needed between the two encoders E_P and E_Q, but only between their outputs. In turn, this implies that *all the passage vectors can be pre-computed*, allowing for real time execution of the Retriever stage.
2. In [1], other decomposable metrics are explored, like cosine similarity or Mahalanobis distance, but they all yield similar results.

2.2 Implementation details

Encoders In principle, E_P and E_Q can be any kind of encoders, since they are completely independent from one another. The only requirement is that the final dimension of the output vectors is the same for both networks. However, since we didn't see any obvious reason to give any advantage of representation to neither passage or question, and that moreover they both have to represent inputs in the same representation space, we decided to keep equal models on both sides.

Since our previous work featured DistilBert [5] as our main encoder, we decided to keep using it also in the IR module. The main advantage of using DistilBert rather than Bert (as was used in [1]) is that the model has significantly less parameters while maintaining a very similar performance. Both encoders were initialized with the standard *base-uncased* pre-trained weights available from HuggingFace [6]. We decided to keep DistilBert at its maximum expressive potential, not limiting the maximum input length but leaving it to its maximum of 512 tokens, with the internal representation dimension of $d = 768$. The encoding of a question or a paragraph is simply the output representation of the first token in the sequence (always `[CLS]`), so all paragraphs and questions are transformed into a 768-dimensional vector.

Training The ultimate goal of the DPR is that of learning a good similarity function $sim(q,p)$, so that given a query, we can map each paragraph p to a *score* representing its semantic similarity with the question. The model should learn to give high scores to paragraphs that are semantically close to the question, with the highest score being ideally assigned to the *positive* (correct) paragraph. At the same time, the model should learn to assign scores close to 0 to irrelevant and wrong (*negative*) paragraphs.

A classic way to deal with this problem is treating it as a *classification* task. Formally, if at training time we feed the model with batches of tuples like $(q, p^+, p_1^-, ..., p_n^-)$ containing the question q, its *positive* passage p^+ and n non-related *negative* passages p_j^-, then the model should learn to correctly *classify* the positive passage against the n negatives. Since the model does not produce a probability distribution, but *scores*, we can use *negative log-likelihood* as a loss function to minimize:

$$L\left(q, p^+, p_1^-, ..., p_n^-\right) = -\log\left(\frac{e^{E_Q(q)^T E_P(p^+)}}{e^{E_Q(q)^T E_P(p^+)} + \sum_{j=1}^{n} e^{E_Q(q)^T E_P(p_j^-)}}\right)$$

Selection of negative passages Constructing a good learning sample is one of the most critical problems in metric learning. We want the classification problem to be *hard*, in order to push the representation to better distinguish between small differences. This often means finding and adding to the learning sample the so-called *hard negatives*, which are examples that are negative, but very similar to the positive one.

This is especially important in OpenQA: since many of the documents have no relation at all with the presented question and are completely dissimilar to the positive passage, training tuples filled of randomly sampled negative passages would hardly influence the learning process as the classification task would be extremely easy.

In [1], the authors propose 3 different methods for negative sampling:

1. *Random*: All negatives are sampled randomly from the set of paragraphs.
2. *Hard-negative*: All negatives are sampled randomly except one, which is selected as the highest scoring non-positive passage returned by a simple IR method (we use an algorithm based on Tf-Idf-weighted sparse representations, as explained in Section 3.2).
3. *Gold*: Since we learn from mini-batches of tuples, we can use the positive passages of the other questions in the same mini-batch as negatives. This is the recommended method, as it is memory-efficient (each mini-batch references the same mini-set of paragraphs) and makes mini-batches *complete* units of learning (because we learn all the relationships within the paragraphs selected in the mini-batch).

We employed both the ideas from the gold and the hard-negative methods, training on mini-batches of the type $(q_i, p_i^+, p_{HN}^-, p_{1,i}^-, ..., p_{m-1,i}^-)$ with $0 < i < m$, where m is the mini-batch size and for each question we have the positive passage, the hard-negative passage, and all the gold passages of the other questions in the same mini-batch. This increases efficiency significantly, because hard-negatives for each question can be retrieved and stored in memory before the training loop, and using gold passages from other questions frees the model of the need to access to the set of paragraphs $(m-1) \times m$ times for each mini-batch to retrieve random passages.

In-batch negatives In the actual implementation we employed a simple, but effective trick to boost efficiency. If m is the size of the mini-batch and Q and P are $(m \times d)$ matrices (with Q_i a question embedding and P_i its related positive passage embedding), we can immediately produce scores for each question-paragraph pair with a single dot product: $S = QP^T$. S is an $(m \times m)$ matrix of *similarity scores*, where any value S_{ij} represents the similarity score between question i and paragraph j. Additionally, we appended to S a column for the scores between each question and its hard-negative passage, obtaining a $m \times (m+1)$ matrix of scores.

In this way, all the scores for each question's positive passage are on the diagonal ($i = j$) and the labels matrix becomes a constant $m \times (m+1)$ diagonal matrix (1 if $i = j$, 0 otherwise), or a simple range $0..m-1$ since we use the *sparse* version of the loss function.

3 Experimental Setup

3.1 Dataset

The SQuADv1.1 Question Answering dataset is our only source of questions and paragraphs. The dataset contains 536 Wikipedia articles, but only 442 are usable since the test set is not publicly available. Each article is sub-divided in many different paragraphs, and for each paragraph there are many different questions. A more detailed analysis of the dataset can be found in our previous work.

We use the official development set as a test set and separate a small part of the training articles to use them as our development set. Table 1 compares the splits in the dataset we used.

	Train set	Dev set	Test Set
Original questions	87599	10570	?
Original paragraphs	18896	2067	?
Our questions	65064	22535	10570
Our paragraphs	13975	4921	2067

Table 1: Dataset instances distributions

We assign a `context_id` to each paragraph: this is simply a tuple `(num_article, idx_paragraph_in_article)` which is used to uniquely identify it. We also use the `context_id` to map questions to the paragraphs they originate from.

We store questions, paragraphs and their IDs using simple Python structures such as lists and dictionaries, but we also prepared an optimized Tensorflow dataset to be used by our model for a faster training. Each element of this dataset is a question-positive paragraph pair and contains:

- The pre-tokenized question and positive paragraph. We used a pre-trained `DistilbertTokenizerFast` from HuggingFace [6] to automatically obtain all necessary inputs for DistilBert in tensor form. All tensors are padded to the same length.
- The (padded) pre-tokenized tensors of the *hard negative* paragraph selected for the question, as explained in Section 2.2.
- The one-hot encoded vectors indicating the start and end tokens of the ground truth answer within the paragraph's tokenized vectors (which we don't use, but is the QA target).
- Two vectors of indices representing the start and end character positions in the positive paragraph's text for each token (again, data that we don't use in our problem, but would be useful to map model predictions to string answers in a full QA scenario)

This dataset is stored into an optimized file format on a Cloud Storage Bucket from Google Cloud. We use TensorFlow's `TFRecordDataset` API to retrieve and manage the data read from the Cloud Storage, then we batch the dataset and set parameters so that instances are prefetched while the model is running and shuffled at the end of each epoch of training.

This pipeline may seem complex, but it's one of the most efficient way to deal with large datasets on Google Colab's Cloud TPUs, which we used extensively for training. Also, the decision to store way more information than what is actually needed by the DPR in each sample may be met with criticism, but this is a deliberate choice since our cloud dataset is general enough to also be used for the QA task or other related problems.

3.2 Baseline

In order to correctly evaluate the performance of the DPR module, we first implemented a simple baseline IR algorithm using classic sparse representations.

We used the `TfIdfVectorizer` class from `scikit-learn` [7] to process the paragraph sets into a matrix of *sparse representations*. Through this tool, we collected two vocabularies of tokens by first applying some basic filtering techniques on the training and validation paragraphs (stripping accents, converting all text to lowercase, filtering out words that appear in more than 80% of the paragraphs of the set) and then tokenizing the two sets independently. The class also takes care of counting the instances of each token into each paragraph, creating large matrices of counts that are then weighted using the classic Tf-Idf weighting factor.

For instance, with our method the training paragraphs end up being represented by a matrix with shape $(13,975 \times 65,808)$ (13,975 documents represented as 65,808-dimensional vector).

Although we can easily process the training and validation sets, we cannot straightforwardly create the matrix of representations for paragraphs in the test set. The main problem comes from the fact that the test set is supposed to be hidden and only used to test the generalization capabilities of the method. Therefore, having access in advance to its vocabulary and creating representations employing this knowledge would give an unfair advantage to test evaluations. We solved the problem by using as a vocabulary for the test set the unified vocabulary of the training and validation sets, ending up with a $(2,067 \times 77,747)$ matrix.

Having created the tokens vocabulary for each split, we can easily transform questions into their sparse representations: for instance a test question would become a $1 \times 77,747$ vector. To rank passages according to the query we can then compute a similarity function between the representations, such as the *cosine similarity*:

$$S_C(A,B) = \frac{A \cdot B^T}{\|A\| \, \|B\|}$$

which is actually implemented as a simpler dot product because the `TfIdfVectorizer` automatically creates L2-normalized representations. This simple metric produces similarity scores for each of the paragraphs in the set and can be used to retrieve the best paragraphs and compute accuracies, as we will present in Section 5.

3.3 Training

We implemented the Dense Passage Retriever using TensorFlow, Keras and HuggingFace's Transformers library. The core of the model is in the `DenseEncoder` layer, which receives a full mini-batch (as described in Section 3.1) and inputs the pre-tokenized question to the DistilBert model specialized on question representations, while the pre-tokenized paragraph and hard paragraph are sent to the other DistilBert. It then outputs the representations at the `[CLS]` token for all models.

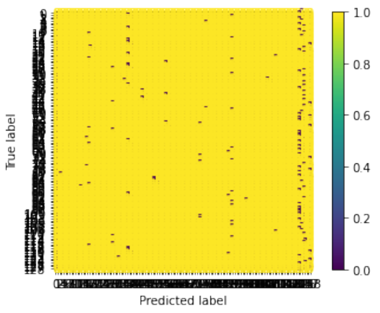
The representations are used by the `DeepQPEncoder`, which builds the matrix:

$$S = \left(\; QP^T \; \middle| \; diag(QP_{\text{hard}}^T) \; \right) \tag{1}$$

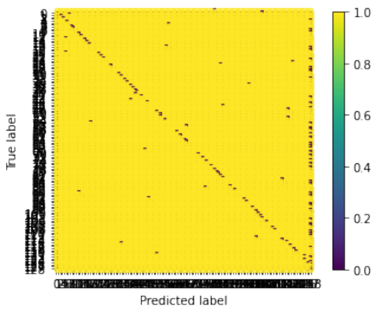
where Q, P and P_{hard} are respectively the mini-batches of the question, paragraph and hard paragraph representations. We use a `SparseCategoricalCrossentropy` loss, so the target can be provided as a sequence of indices rather than a matrix: the vector of indices is built from the range $0..m-1$. Also, we set the parameter `from_logits=True` because we don't want to transform the predictions into a probability distribution.

We freezed the weights of the two DistilBert models up to block 3, so that the basic language features learnt in the pre-training are left untouched. We then trained the model on the Cloud TPUs provided by Google Colab for 200 epochs, with early stop after 20 epochs of non-improving validation loss. We used a batch size of 128 (so, S is a 128×129 matrix) and used the Adam optimizer with a starting learning rate of $3e^{-6}$. The training took a few hours and stopped after 9 epochs, with a final accuracy on the validation set of 64.36%.

The effect of training can be seen from the evolution of confusion matrices for a mini-batch of data in Fig. 1. The situation before training is on the left: keeping in mind that the ideal confusion matrix is a diagonal matrix (predicted labels are equal to ground truth labels in all cases), we can observe that the predictions of the model are seemingly random. On the contrary, Fig. 1b shows that at the end of training, predictions are often aligned with the ground truth labels, with few sparse exceptions. However, there is a clearly visible column of wrong predictions on the far right of the matrix: those are the questions incorrectly attributed to the *hard paragraph*, since the hard paragraphs' scores in matrix S are always in column m by construction. This behaviour shows the effectiveness of the insertion of the hard paragraph in the training samples: the hard paragraph is often the model's choice



(a) Confusion matrix for a mini-batch at the beginning of training.



(b) Confusion matrix for a mini-batch at the end of training.

Figure 1: Evolution of training accuracy.

when it makes a mistake, so training updates become more meaningful and guide the DPR towards paying attention to what the difference *really* is between the contents of two similar passages.

4 Experiments

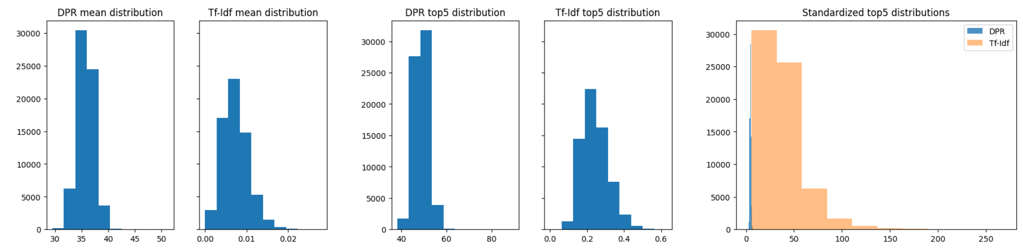
In our full-stack, the QA stage only accepts one context from which to extract an answer, so our main goal was to optimize our retriever stage for predicting only the positive passage. However, in literature many *Readers* can accept in input more than one document, usually the 20 or 100 with highest similarity score with respect to the question encoding. Our evaluations take into account this important aspect, showing retrieval accuracy results for top-1, top-5, top-20 and top-100.

Moreover, we decided to focus our evaluation on the comparison and possible integration between sparse encodings (in our case, *Tf-Idf*) and dense encodings, returned by the DPR module. For this reason, we considered these five similarity measures:

1. *DPR*: this is the similarity measure displayed in the previous sections, or the dot product between the dense encodings produced by our two encoders E_P and E_Q.
2. *Tf-Idf*: this score is instead produced by the dot product between the sparse encodings produced by the Tf-Idf algorithm, as explained in Section 3.2.
3. *Sum*: this score is simply the sum of the two above scores for each passage, after normalizing both distributions of scores by dividing them by the maximum in their distribution.
4. *Max*: it is, for each passage, the maximum score between DPR score and Tf-Idf score, after normalization.
5. *Weighted sum*: as the name suggests, $S_{\text{wsum}} = (1-h) * S_{\text{DPR}} + h * S_{\text{Tf-Idf}}$, where h is an hyperparameter optimized on the validation set. In this case, instead of normalizing the scores, we standardize them.

The decision to normalize the distributions in *Sum* and *Max* scores is due to the important difference between the two ranges of scores. From Figures 2 and 3 we can see that, at least in the training set, the two distribution of scores produced by DPR and Tf-Idf are vastly different, no matter the transformation that is applied to them. However, if we take into consideration the *Sum* and *Max* scores, the most important thing is that the *top-k* values are comparable between the two methods. For this reason, the normalization shown in Figure 3 was the chosen one.

Instead, when considering the *weighted sum*, we are aided by the hyperparameter tuning, that will naturally bring the amplitude of the distributions closer together. For this reason, in this case we chose standardization, in order to fix the mean to zero and let the parameter set the best multiplying factor to adapt the variance of the distributions.



(a) Mean scores per question (b) Mean on top-5 scores (c) Mean on top-5 after standard.

Figure 2: Comparison between distributions

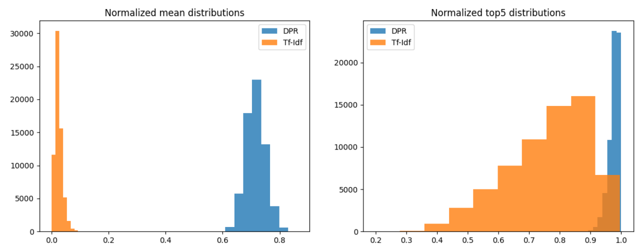


Figure 3: Mean scores after normalization (only dividing by *max*)

5 Results

5.1 Retrieval results

First of all, we compare the results of the algorithms we detailed in the previous sections when taken independently as IR systems.

Method	Top-1	Top-5	Top-20	Top-100
Random	0.05%	0.24%	0.97%	4.84%
Tf-Idf	51.37%	74.87%	90.19%	97.46%
DPR	47.84%	77.94%	93.59%	99.29%
Sum	57.39%	81.38%	93.92%	98.86%
Max	49.65%	85.03%	95.70%	99.51%
W. Sum	**69.64%**	**91.32%**	**97.90%**	**99.77%**

Table 2: Retrieval accuracy on the test set for all algorithms

Table 2 showcases the results obtained with our algorithms in terms of top-k accuracy. The same can be seen in Fig. 4, where it is clearer that the accuracy has a monotonic tendency with increasing k, and will naturally arrive to 100% when $k = N$, where N is the total number of documents in the set.

We can also see how DPR is slightly worse than the baseline Tf-Idf only when k is low, while improving more and more rapidly with growing k. Even not fully optimized versions of the DPR, for example after only *6 epochs*, improve on Tf-Idf with growing k.

But the most important fact that we can acknowledge is that the union of the two metrics is actually almost always better than the two taken alone. In particular, the weighted sum is considerably better both than DPR only and Tf-Idf only, even for top-1 classification.

Since top-1 classification is the only one that is actually used in the full OpenQA model, we look a bit more in-depth to the key metrics of the different methods in Table 3.

As mentioned before, Tf-Idf scores better than DPR in *top-1* classification. In fact, we can see that using the weighted sum between the two scores increases considerably all metrics by more than 20%.

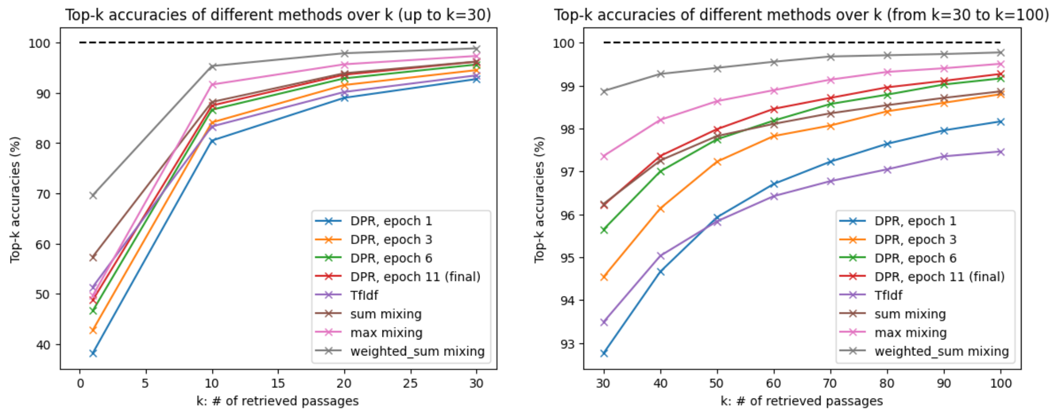


Figure 4: Top-k accuracy of our different methods on the test set

Method	Accuracy	Macro-Precision	Macro-Recall	Macro-F1-Score
Tf-Idf	51.37%	63.84%	52.60%	52.57%
DPR	47.84%	54.01%	48.50%	46.43%
Sum	57.39%	67.91%	58.65%	58.42%
Max	49.65%	57.14%	50.62%	49.92%
W. sum	**69.57%**	**75.59%**	**71.05%**	**70.07%**

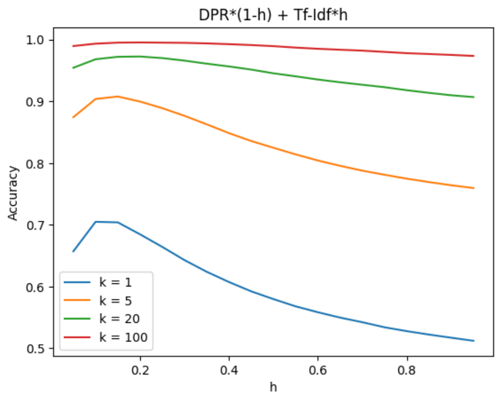
Table 3: Comparison between different methods

There are 723 questions out of the 10,570 of the test set where the TfIdf and DPR scoring methods are not able to return the correct paragraph, while the weighted sum of their scores can. For example, the question *"Approximately how many British oil* ***paintings*** *does the* ***museum*** *have?"* (which is also another example of questions that imply a specific context to be answered, as the one presented in Section 1.4) we have that:

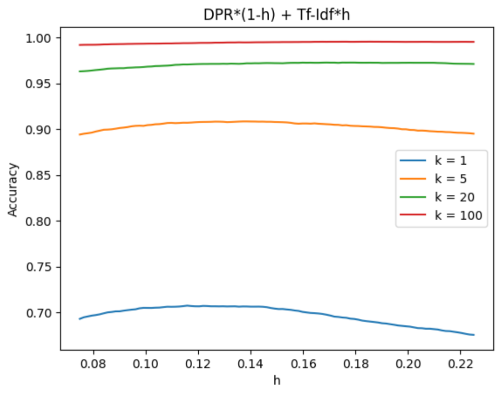
- The highest-scoring paragraph from Tf-Idf is: *"As interesting examples of expositions the most notable are: the world's first* ***Museum*** *of Posters (...),* ***Museum*** *of Hunting and Riding and the Railway* ***Museum****. From among Warsaw's 60* ***museums****, the most prestigious ones are National* ***Museum*** *(...) as well as one of the best collections of* ***paintings*** *(...) including some* ***paintings*** *from Adolf Hitler's private collection, (...)"*. It does not answer the question nor talks about the number of paintings included in any of the mentioned collections, but some key words (*paintings* and *museum*) are repeated multiple times and make the question's content similar to the paragraph's.
- The highest-scoring paragraph for the DPR is: ***The collection of drawings includes over 10,000 British*** *and 2,000 old master* ***works****, including works by (...)*. This is not a bad answer at all since it mentions the number of British pieces in a collection, but the question was probably referring to another collection (*oil paintings*, not *drawings* as mentioned in the paragraph). Yet, this shows the higher level of semantic understanding of the DPR with respect to sparse representations.
- The positive paragraph, which is also the one obtained by the weighted sum of the scores of DPR and Tf-Idf (*The collection includes about 1130 British and 650 European oil paintings...*) was the 3rd paragraph for Tf-Idf, which assigned it a score of 4.204 (the highest score was 4.4), and the 7th for DPR, which assigned it a score of 7.9836 (the highest score was 12.6403).

5.1.1 Hyperparameter optimization

The optimal weight for the weighted sum method (hyperparameter h) was found by trying different configurations on the validation set. We first searched in a large space (Fig. 5a) and then narrowed down the search trying values around the most promising area (Fig. 5b).



(a) *Top-k* accuracy vs. hyperparameter



(b) Fine tuning around the maximum

Figure 5: Hyperparameter optimization

We obtained slightly different values for the best h for different k: the best h for top-1 accuracy is 0.116, the best for top-5 is 0.137, for top-20 is 0.169 and for top-100 is 0.186. The hyperparameter choice that gave us the results seen in Figure 4 and Table 3 was $h = 0.14$, an intermediate value between the different optimal values. In addition, our QA model only accepts one paragraph (the best-scoring), but since other QA models could deal with multiple paragraphs (eg. learning more fine-grained rules for choosing between the top-20 paragraphs and extracting the answer from it), we observe that the hyperparameter could be optimized in relation to the end goal.

While optimizing the hyperparameter for the weighted sum, we discover two interesting facts:

- From Figure 5a and 5b we can see that the best value for the hyperparameter is skewed towards DPR: this however must also take into account the high difference in distribution amplitude that can be seen in Figure 2c. Since DPR generates scores with lower variance than Tf-Idf, to make the top scores comparable after standardization the hyperparameter must increase the value of DPR scores.
- The optimal value of h moves towards Tf-Idf with increasing k. This is in contrast with the results shown in Figure 4, in which we can see DPR being more accurate with increasing k. Once again, this result can be explained by the difference in the distributions: in Tf-Idf, only the best value has a high score, that decreases rapidly for other paragraphs. When we search 20 or 100 paragraphs, Tf-Idf would give a noticeable contribution to the weighted sum only to the first few, while DPR decides all remaining ones. To counteract this distributional difference and let Tf-Idf influence also the latter scores, the hyperparameter shifts to higher values.

5.2 Question Answering Results

Finally, we explore the effect of our retrieval methods on the larger OpenQA task. We do not employ an end-to-end training scheme, so our final model for QA uses the methods we have discussed in this report to retrieve the best passage given the question and then inputs the question-paragraph pair into a pre-trained Reader.

The Reader we use is the one that was trained in our previous project for the course: it consists of a simple DistilBert model plus some additional layers that are used to compute two probability distributions over the tokens (the probability for each token to be the start and end of the answer within the paragraph text). We actually re-use the weights of the Reader without further training (since we had already trained on SQuAD's training set). We also employ a more powerful version of the Reader which uses Bert instead of DistilBert as backbone. Table 4 contains all the evaluation results for our experiments and compares them to our previous scores.

Our results are significantly worse than the ones obtained from our previous study, but not only OpenQA is an exponentially harder problem, but since we the previous Reader is simply used as a

Reader Backbone	Retrieval Method	Exact Match	Macro F1
DistilBERT	*Exact*	*61.17%*	*75.53%*
	Tf-Idf	32.79%	42.09%
	DPR	30.58%	40.15%
	Sum	36.52%	46.59%
	Max	31.60%	41.11%
	W. Sum	**44.21%**	**55.97%**
BERT	*Exact*	*64.33%*	*78.59%*
	Tf-Idf	34.48%	43.71%
	DPR	31.81%	41.40%
	Sum	38.44%	48.43%
	Max	33.13%	42.52%
	W. Sum	**46.41%**	**58.03%**

Table 4: QA evaluation results on test set, in terms of exact match precision and Macro-weighted F1 score.

blackbox module to obtain the answer, its error sets an upper bound on the performance of the whole system.

It's interesting to observe that the relation between top-1 Retrieval accuracy and the QA performance is almost exactly linear (as seen in Fig. 6), which highlights once again the importance of our focus on an effective Retrieval method.

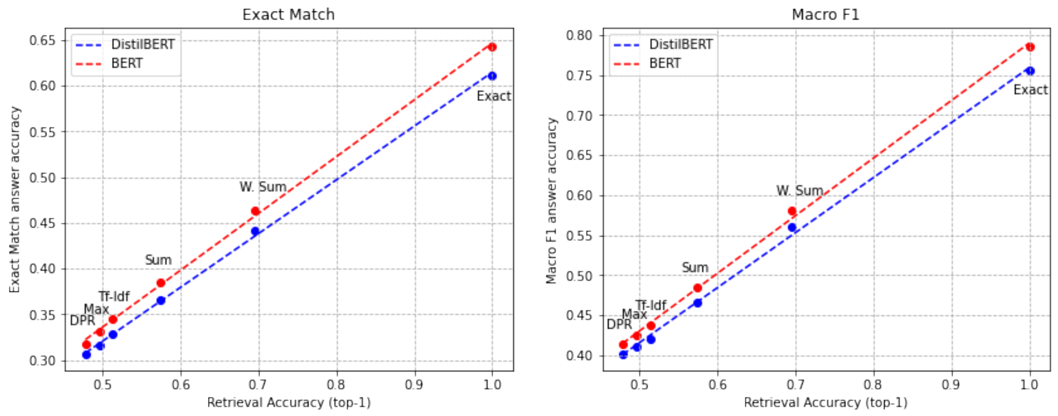


Figure 6: Relation between Retrieval accuracy and QA Exact Match score. The dashed lines have been obtained via linear regression.

6 Conclusions

In this paper we explored a method for retrieving documents in an OpenQA task by building and exploiting a dense representation. The main advantages of this method is the possibility of precomputing paragraph encodings and enabling real time document retrieval, while solving the critical problem of *"term-mismatch"* that affects sparse encoders.

Our main contribution in this work was the focus on the potential benefits of mixing dense representations with sparse ones. The results that we obtained show that the two kinds of representations bring complementary results and can be mixed to achieve better results. Both representations are completely decomposable and can be produced offline in advance. Moreover, their scores are directly comparable since the similarity measures bring all scores in the `[0,1]` interval.

We show that with a simple linear combination of the two scores, through an hyperparameter optimized on the development set, *top-1* accuracy can be improved by almost 20 points. Moreover, the results were calculated with an h value that was not strictly optimal for *top-1* classification, but valid for a broader range of *k* values.

Since DPR and Tf-Idf create completely different distribution of scores, we think that a non-linear combination method that can better transform the two distributions and better learn their strengths and weaknesses could increase its performance even more. For example, a simple dense neural network that takes as input the scores of all the paragraphs from the two methods and returns a final score for each paragraph could be a simple but effective implementation, and could be trained either in combination with the double encoder or afterwards.

Another possible improvement could be the implementation of a *Representation-Interaction Retriever* [8]: instead of keeping only the `[CLS]` token and transforming each document in a `[768]` dimensional vector, we could keep all of the tokens to have a matrix of dimensions `[512, 768]`. The same output would then be retrieved from E_Q, the question encoder, and the similarity score would change from a simple dot product to a more complex function able to find more complex relations between the tokens of the document and the ones of the question. In this case, if this function was implemented with a neural network, it would need to be trained together with E_P and E_Q, the two encoders, to create an effective representation space useful for metric learning. The obvious advantage in this case would be a much higher expressive power and a more complex representation space, that would however need more than $500\times$ the memory and a slower (but not considerably slower) inference.

References

[1] Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. Dense passage retrieval for open-domain question answering, 2020.

[2] Danqi Chen, Adam Fisch, Jason Weston, and Antoine Bordes. Reading wikipedia to answer open-domain questions, 2017.

[3] Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Matthew Kelcey, Jacob Devlin, Kenton Lee, Kristina N. Toutanova, Llion Jones, Ming-Wei Chang, Andrew Dai, Jakob Uszkoreit, Quoc Le, and Slav Petrov. Natural questions: a benchmark for question answering research. *Transactions of the Association of Computational Linguistics*, 2019.

[4] Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. Semantic parsing on Freebase from question-answer pairs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1533–1544, Seattle, Washington, USA, October 2013. Association for Computational Linguistics.

[5] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter, 2019.

[6] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics.

[7] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[8] Fengbin Zhu, Wenqiang Lei, Chao Wang, Jianming Zheng, Soujanya Poria, and Tat-Seng Chua. Retrieving and reading: A comprehensive survey on open-domain question answering, 2021.