NLP for Question Answering

Evaluating QA: the Retriever & the Full QA System

A review of Information Retrieval and the role it plays in an IR QA system

Jun 30, 2020 • 30 min read

elasticsearch mean average precision recall for IRQA QA system design

View On GitHub launch binder Open in Colab

In our last post, Evaluating QA: Metrics, Predictions, and the Null Response, we took a deep dive into how to assess the quality of a BERT-like Reader for Question Answering (QA) using the Hugging Face framework. In this post, we'll focus on the other component of a modern Information Retrieval-based (IR) QA system: the Retriever. Specifically, we'll introduce Elasticsearch as a powerful and efficient IR tool that can be used to scour through large corpora and retrieve relevant documents. We'll explain how to implement and evaluate a Retriever in

the context of Question Answering and demonstrate its impact on an IR QA system.

Prerequisites

- a basic understanding of Information Retrieval & Search
- a basic understanding of IR based QA systems (see our previous posts)
- a basic understanding of Transformers and PyTorch
- a basic understanding of the SQuAD2.0 dataset

Retrieving the right document is important

As we've discussed throughout this series, many modern QA systems take a two-staged approach to answering questions. In the first stage, a document retriever selects *N* potentially relevant documents from a given corpus. Subsequently, a machine comprehension model processes each of the *N* documents to determine an answer to the input question.

Because of recent advances in NLP and deep learning (i.e., flashy Transformers), the machine comprehension component has typically been the main focus of evaluation and performance enhancement. Retrievers have received limited attention in the context of QA, despite their obvious importance: stage two of an IR QA system is bounded by the performance of stage one. Let's get more specific.

We recently explained methods that - given a question and context passage - enable BERT-like models to produce robust answers by selectively processing predictions and by refraining from answering certain questions at all. While the ability to properly comprehend a passage and produce a correct answer is a critical feature of any QA tool, the success of the overall system is highly dependent on first providing a correct passage to read through. Without being

fed a context passage that actually contains the ground-truth answer, the overall system's performance is limited to how well it can predict no-answer questions.

To demonstrate, we'll revisit an example from our second blog post, in which we asked three questions of a Wikipedia search engine-based QA system:

```
**Example 1: Incorrect**
Question: When was Barack Obama born?
Top wiki result: <WikipediaPage 'Barack Obama Sr.'>
Answer: 18 June 1936 / February 2 , 1961 /

**Example 2: Correct**
Question: Why is the sky blue?
Top wiki result: <WikipediaPage 'Diffuse sky radiation'>
Answer: Rayleigh scattering /

**Example 3: Correct**
Question: How many sides does a pentagon have?
Top wiki result: <WikipediaPage 'The Pentagon'>
Answer: five /
```

In Example 1, the Reader had no chance of producing the correct answer because of its outright absence from the context served up by the Retriever. Namely, the Retriever erroneously provided a page about Barack Obama Sr. instead of his son, the former US President. In this case, the only way the Reader could have possibly produced the correct answer was if the correct answer was actually not to answer at all.

On the flip side, in Example 3, the Retriever did not identify the globally "correct" document - it returned an article about "The Pentagon" instead of a page about geometry - but nonetheless, it provided enough context for the Reader to succeed.

These quick examples illustrate why an effective Retriever is crucial for an end-to-end QA system. Now let's take a deeper look at a classic tool used for information retrieval - Elasticsearch.

Elasticsearch as an IR Tool

Modern QA systems employ a variety of techniques for the task of information retrieval, ranging from traditional sparse vector word matching (e.g., Elasticsearch) to novel approaches using dense representations of encoded passages combined with efficient search capabilities. Despite the flurry of contemporary research efforts in this area, the traditional sparse vector approach performs very well overall, and has only recently been overtaken by embedding-based systems for QA retrieval tasks. For that reason, we'll explore Elasticsearch as an easy-to-use framework for document retrieval. So, what exactly is Elasticsearch?

Elasticsearch is a powerful open-source search and analytics engine built on the Apache Lucene library that is capable of handling all types of data - including textual, numerical, geospatial, structured, and unstructured data. It is built to scale with a robust set of features, rich ecosystem, and diverse list of client libraries, making it easy to integrate and use. In the context of information retrieval for automated question answering, we are keenly interested in the features surrounding full-text search.

Elasticsearch provides a convenient way to index documents so they can quickly be queried for nearest neighbor search using a similarity metric based on TF-IDF. Specifically, it uses BM25 term weighting to represent question and context

passages as high-dimensional, sparse vectors that are efficiently searched in an inverted index. For more information on how an inverted index works under the hood, we recommend this quick and concise blog post.

Using Elasticsearch with SQuAD2.0

With this basic understanding of how Elasticsearch works, let's dive in and build our own Document Retrieval system by indexing a set of Wikipedia article paragraphs that support questions and answers from the SQuAD2.0 dataset. Before we get started, we'll need to download and prepare data from SQuAD2.0.

Download and Prepare SQUAD2.0

Show Code

A common practice in IR for QA is to segment large articles into smaller passages before indexing, for two main reasons:

1. Transformer-based Readers are slow; providing an entire Wikipedia article to BERT for processing can take 5 - 30 seconds, even with a decent GPU!
2. Smaller passages reduce noise; by identifying a more concise context passage for BERT to read through, we reduce the chance of BERT getting lost.

Of course, the chunking method proposed here doesn't come without a cost. Larger documents contain more information on which to retrieve. By reducing passage size, we are potentially trading off system recall for speed - although, as we will discuss later in this post, there are techniques to alleviate this.

With our chunking approach, each article paragraph will be prepended with the article title, and collectively serve as the corpus of documents over which our Elasticsearch Retriever will search. In practice, open-domain QA systems sit atop massive collections of documents (think: all of Wikipedia) to provide a breadth of information from which to answer general-knowledge questions. For the

purposes of demonstrating Elasticsearch functionality, we will limit our corpus to only the Wikipedia articles supporting SQuAD2.0 questions.

The following `parse_qa_records` function will extract question/answer examples, as well as paragraph content from the SQuAD2.0 data set.

Show Code

```
# load and parse data
train_file = "data/squad/train-v2.0.json"
dev_file = "data/squad/dev-v2.0.json"

train = json.load(open(train_file, 'rb'))
dev = json.load(open(dev_file, 'rb'))

qa_records, wiki_articles = parse_qa_records(train['data'])
qa_records_dev, wiki_articles_dev = parse_qa_records(dev['data'])
```

```
Data contains 86821 question/answer pairs with a short answer, and 43498 without.
There are 19035 unique wikipedia article paragraphs.
Data contains 5928 question/answer pairs with a short answer, and 5945 without.
There are 1204 unique wikipedia article paragraphs.
```

```
# parsed record example
qa_records[10]
```

```
{'example_id': '56d43c5f2ccc5a1400d830ab',
 'document_title': 'Beyoncé',
 'question_text': 'What was the first album Beyoncé released as a solo artist?',
 'short_answer': 'Dangerously in Love'}
```

```
# parsed wiki paragraph example
print(wiki_articles[10])
```

```
{'document_title': 'Beyoncé_10', 'document_text': 'Beyoncé Beyoncé\'s first solo recording was a featur
```

Download Elasticsearch

With our data ready to go, let's download, install, and configure Elasticsearch. We recommend opening this post as a Colab notebook and executing the following code snippet to set up Elasticsearch. Alternatively, you can install and launch Elasticsearch on your local machine by following the instructions here.

Show Code

Load Data into Elasticsearch

We'll use the official low-level Python client library for interacting with Elasticsearch.

Show Code

By default, Elasticsearch is launched locally on port 9200. We first need to instantiate an Elasticsearch client object and connect to the service.

```
from elasticsearch import Elasticsearch

config = {'host':'localhost', 'port':9200}
es = Elasticsearch([config])

# test connection
es.ping()
```

```
True
```

Before we go further, let's introduce a few concepts that are specific to Elasticsearch and the process of indexing data. An *index* is a collection of documents that have common characteristics (similar to a database schema in an RDBMS). *Documents* are JSON objects having their own set of key-value pairs consisting of various data types (similar to rows/fields in RDBMS).

When we add a document into an index, the document's text fields undergo analysis prior to being indexed. This means that when executing a search query against an index, we are actually searching against the post-processed representation that is stored in the inverted index, not the raw input document itself.

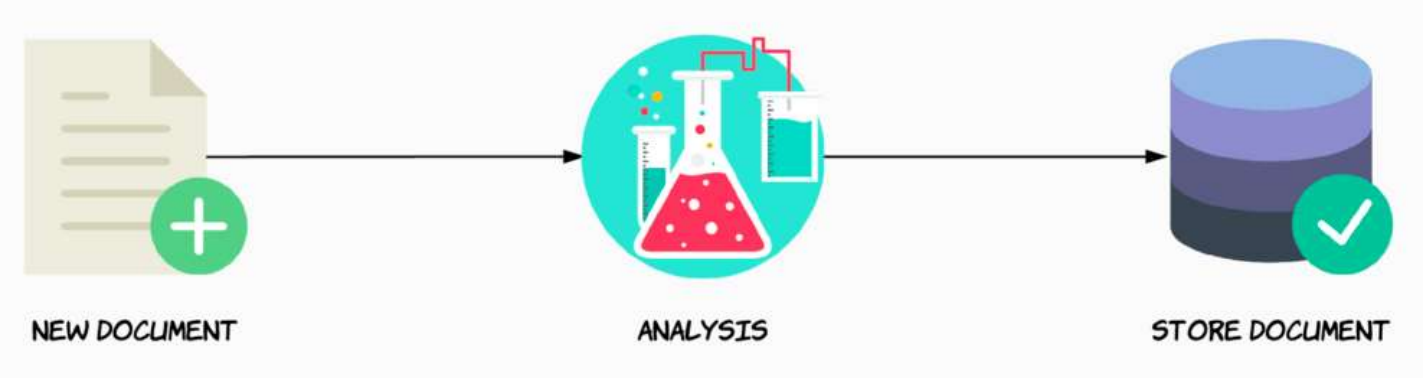


Image Credit

The analysis process is a customizable pipeline carried out by an *Analyzer*. Elasticsearch analyzer pipelines are composed of three sequential steps: *character filters*, a *tokenizer*, and *token filters*. Each of these components modifies the input stream of text according to some configurable settings.

- **Character filters** have the ability to add, remove, or replace characters. A common application is to strip `html` markup from the raw input.
- The character-filtered text is passed to a **tokenizer** which breaks up the input string into individual tokens. The default (`standard`) tokenizer splits tokens on whitespace, and most symbols (like commas, periods, semicolons, etc.)
- The token stream is passed to a **token filter** which adds, removes, or modifies tokens. Typical token filters include converting all text to `lowercase`, and removing `stop` words.

Elasticsearch comes with several built-in Analyzers that satisfy common use cases and defaults to the `Standard Analyzer`. The Standard Analyzer doesn't contain any character filters, uses a `standard` tokenizer, and applies a `lowercase` token filter. Let's take a look at an example sentence as it's passed through this pipeline:

> *"I'm in the mood for drinking semi-dry red wine!"*

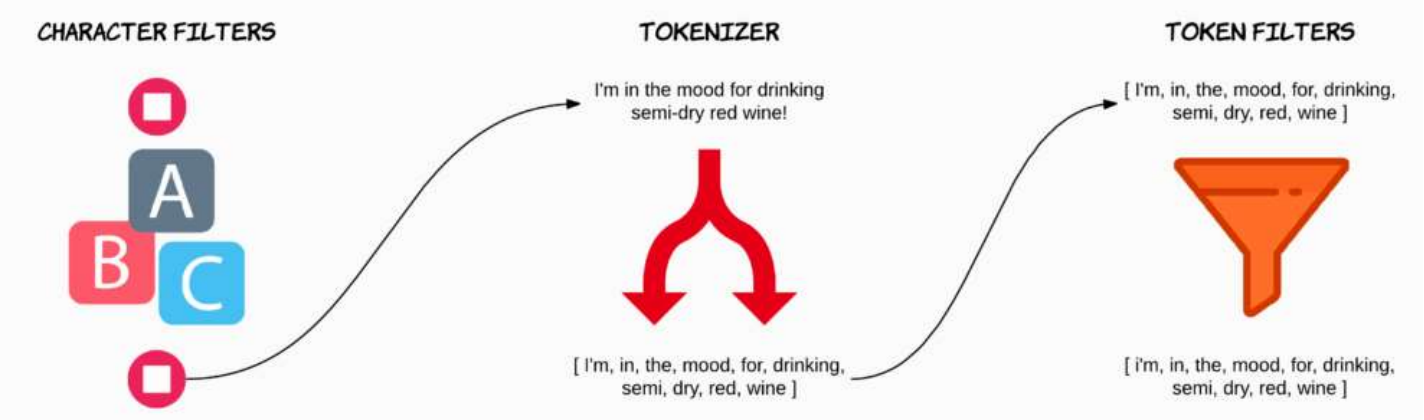


Image Credit

Crafting analyzers to your use case requires domain knowledge of the problem and dataset at hand, and doing so properly is key to optimizing relevance scoring for your search application. We found this blog series very useful in explaining the importance of analysis in Elasticsearch.

Create an Index

Let's create a new index and add our Wikipedia articles to it. To do so, we provide a name and optionally some index configurations. Here we are specifying a set of `mappings` that indicate our anticipated index schema, data types, and how the text fields should be processed. If no `body` is passed, Elasticsearch will automatically infer fields and data types from incoming documents, as well as apply the `Standard Analyzer` to any text fields.

```
index_config = {
    "settings": {
        "analysis": {
            "analyzer": {
                "standard_analyzer": {
                    "type": "standard"
                }
            }
        }
    },
    "mappings": {
        "dynamic": "strict",
        "properties": {
            "document_title": {"type": "text", "analyzer": "standard_analyzer"},
            "document_text": {"type": "text", "analyzer": "standard_analyzer"}
```

```
            }
        }
    }

index_name = 'squad-standard-index'
es.indices.create(index=index_name, body=index_config, ignore=400)
```

```
{'acknowledged': True,
 'index': 'squad-standard-index',
 'shards_acknowledged': True}
```

Populate the Index

We can then loop through our list of Wikipedia titles and articles and add them to our newly created Elasticsearch index.

Show Code

```
all_wiki_articles = wiki_articles + wiki_articles_dev

populate_index(es_obj=es, index_name='squad-standard-index', evidence_corpus=all_wiki_artic
```

```
Succesfully loaded 20239 into squad-standard-index
```

Search the Index

Wahoo! We now have some documents loaded into an index. Elasticsearch provides a rich query language that supports a diverse range of query types. For this example, we'll use the standard query for performing full text search called a `match` query. By default, Elasticsearch sorts and returns a JSON response of search results based on a computed relevance score, which indicates how well a given document matches the query. In addition, the search response also includes the amount of time the query took to run.

Let's look at a simple `match` query used to search the `document_text` field in our newly created index.

⚡ **Important:** As previously mentioned, all documents in the index have gone through an analysis process prior to indexing; this is called *index time analysis*. To maintain consistency in matching text queries against the post-processed index tokens, the same Analyzer used on a given field at index time is automatically

applied to the query text at search time. *Search time analysis* is applied depending on which query type is used; `match` queries apply search time analysis by default.

Show Code

```
question_text = 'Who was the first president of the Republic of China?'

# execute query
res = search_es(es_obj=es, index_name='squad-standard-index', question_text=question_text,
```

```
print(f'Question: {question_text}')
print(f'Query Duration: {res["took"]} milliseconds')
print('Title, Relevance Score:')
[(hit['_source']['document_title'], hit['_score']) for hit in res['hits']['hits']]
```

```
Question: Who was the first president of the Republic of China?
Query Duration: 74 milliseconds
Title, Relevance Score:
```

```
[('Modern_history_54', 23.131157),
 ('Nanjing_18', 17.076923),
 ('Republic_of_the_Congo_10', 16.840765),
 ('Prime_minister_16', 16.137493),
 ('Korean_War_29', 15.801523),
 ('Korean_War_43', 15.586578),
 ('Qing_dynasty_52', 15.291815),
 ('Chinese_characters_55', 14.773873),
 ('Korean_War_23', 14.736045),
 ('2008_Sichuan_earthquake_48', 14.417962)]
```

Evaluating Retriever Performance

Ok, we now have a basic understanding of how to use Elasticsearch as an IR tool to return some results for a given question, but how do we know if it's working? How do we evaluate what a good IR tool looks like?

We'll need two things to evaluate our Retriever: some labeled examples (i.e., SQuAD2.0 question/answer pairs) and some performance metrics. In the conventional world of information retrieval, there are many metrics)used to quantify the relevance of query results, largely centered around the concepts of precision and recall. For IR in the context of QA, these ideas are adapted into

two commonly used evaluation metrics: *recall* and *mean average precision (mAP)*. Additionally, we consider the amount of time required to execute a query, since the main point of having a two-stage QA system is to efficiently narrow the large search space for our Reader.

Recall

Traditionally, *recall* in IR indicates the fraction of all relevant documents that are retrieved. In this case, we are less concerned with finding *all* of the passages containing the answer and more concerned with the binary presence of a passage containing the correct answer being returned. In that light, a Retriever's recall is defined across a set of questions as *the percentage of questions for which the answer segment appears in one of the top N pages returned by the search method*.

Mean Average Precision

While the *recall* metric focuses on the minimum viable result set to enable a Reader for success, we do still care about the composition of that result set. We want a metric that rewards a Retriever for: a) returning a lot of answer-containing documents in the result set (i.e., the traditional meaning of precision), and b) returning those answer-containing documents higher up in the result set than non-answer-containing documents (i.e., ranking them correctly). This is precisely (🙃) what *mean average precision* (mAP) does for us.

To explain mAP further, let's first break down the concept of average precision for information retrieval. If our Retriever is asked to return N documents and m of those documents contains the true answer, then average precision (AP) is defined as:

$$AP@N = \frac{1}{m}\sum_{k=1}^{N}(P(k) \text{ if the } k^{th} \text{ item contains the answer}) = \frac{1}{m}\sum_{k=1}^{N} P(k) * rel(k)$$

where *rel(k)* is just a binary indication of whether the kth passage contains the correct answer or not. Using a concrete example, consider retrieving N=3

documents, of which only one contains the correct answer. Here are three scenarios for how this could happen:

Scenario	Binary Indication	Precision@k's	Average Precision@N
A	[1, 0, 0]	[1/1, 0, 0]	(1/1)*[(1/1) + 0 + 0] = 1
B	[0, 1, 0]	[0, 1/2, 0]	(1/1)*[0 + (1/2) + 0] = 0.5
C	[0, 0, 1]	[0, 0, 1/3]	(1/1)*[0 + 0 + (1/3)] = 0.33

Scenario A is rewarded with the highest score because it was able to correctly rank the ground truth document relative to the others returned. Since average precision is calculated on a per-query basis, the mean average precision is simply just *the average AP across all queries*.

Now, using our Wikipedia passage index, let's define a function called `evaluate_retriever` to loop through all question/answer examples from the SQuAD2.0 train set and see how well our Elasticsearch Retriever performs in terms of recall, mAP, and average query duration when retrieving *N=3* passages.

Show Code

```
# combine train/dev examples and filter out SQuAD records that
# do not have a short answer for the given question
all_qa_records = qa_records+qa_records_dev
qa_records_answerable = [record for record in all_qa_records if record['short_answer'] != '

# run evaluation
results_df, metrics = evaluate_retriever(es_obj=es, index_name='squad-standard-index', qa_r
```

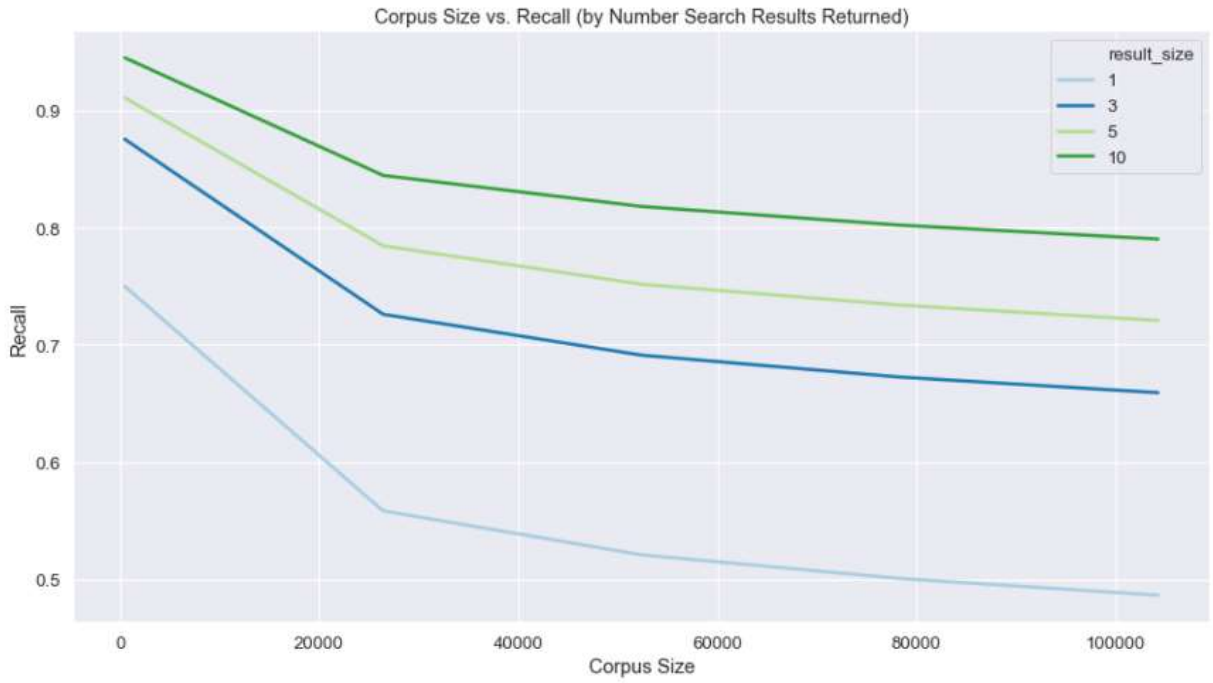
```
metrics
```

```
{'Recall': 0.8226180336176131,
 'Mean Average Precision': 0.7524133234140888,
 'Average Query Duration': 3.0550841518506937}
```

Improving Search Results with a Custom Analyzer

Identifying a correct passage in the Top 3 results for 82% of the SQuAD questions in ~3 milliseconds per question is not too bad! But that means that we've effectively limited our overall QA system to an 82% upper bound on performance. How can we improve upon this?

One simple and obvious way to increase recall would be to just retrieve more passages. The following figure shows the effects of varying corpus size and result size on Elasticsearch retriever recall. As expected we see that the number of passages retrieved (i.e. *Top N*) has a dramatic impact on recall; a ~10-15 point jump from 1 to 3 passages returned, and ~5 point jump for each of the other tiers. We also see a gradual decrease in recall as corpus size increases, which isn't surprising.



Experimental results demonstrating the impact of increasing corpus size and number of results retrieved on Elasticsearch recall.

While increasing the number of passages retrieved is effective, it also has implications on overall system performance as the (already slow) Reader now

has to reason over more text. Instead, we can lean on best practices in the well-explored domain of information retrieval.

Optimizing full text search is a battle between precision (returning as few irrelevant documents as possible) and recall (returning as many relevant documents as possible). Matching only exact words in the question results in high precision; however, it misses out on many passages that could be relevant. We can cast a wider net by searching for terms that are not *exactly* the same as those in the question, but are related in some way. Here, Elasticsearch Analyzers can help. Earlier in this post, we described how Analyzers provide a flexible and extensible method to tailor search for a given dataset. Two of the custom Analyzers that can help cast a wider net are *stop words* and *stemming*.

- **Stop words:** Stop words are the most frequently occuring words in the English language (for example: "and," "the," "to," etc.) and add minimal semantic value to a piece of text. It is common practice to remove them in order to decrease the size of the index and increase the relevance of search results.
- **Stemming:** The English language is inflected; words can alter their written form to express different meanings. For example, “sing,” “sings,” “sang,” and “singing” are written with slight differences, but all really mean the same thing (albeit with varying tenses). Stemming algorithms exploit the fact that search intent is *usually* word-form agnostic, and attempt to reduce inflected words to their root form: consequently improving retrievability. We'll implement the Snowball stemming algorithm as a token filter in our custom Analyzer.

```
# create new index
index_config = {
    "settings": {
        "analysis": {
            "analyzer": {
                "stop_stem_analyzer": {
                    "type": "custom",
                    "tokenizer": "standard",
```

```
                    "filter":[
                        "lowercase",
                        "stop",
                        "snowball"
                    ]

                }
            }
        }
    },
    "mappings": {
        "dynamic": "strict",
        "properties": {
            "document_title": {"type": "text", "analyzer": "stop_stem_analyzer"},
            "document_text": {"type": "text", "analyzer": "stop_stem_analyzer"}
            }
        }
    }

es.indices.create(index='squad-stop-stem-index', body=index_config, ignore=400)

# populate the index
populate_index(es_obj=es, index_name='squad-stop-stem-index', evidence_corpus=all_wiki_arti

# evaluate retriever performance
stop_stem_results_df, stop_stem_metrics = evaluate_retriever(es_obj=es, index_name='squad-s
                                                             qa_records=qa_records_answerab
```

```
stop_stem_metrics
```

```
{'Recall': 0.8501115914996388,
 'Mean Average Precision': 0.7800892731997112,
 'Average Query Duration': 0.7684287701215108}
```

Awesome! We've increased recall and mAP by about 3 points *and* reduced our average query duration by nearly 4 times, through simple preprocessing steps that just scratch the surface of tailored analysis in Elasticsearch.

There is no "one-size-fits-all" recipe for optimizing search relevance, and every implementation will be different. In addition to custom analysis, there are many other methods for increasing search recall - for example, query expansion, which introduces additional tokens/phrases into a query at search time. We'll save that

topic for another post. Instead, let's take a look at how the Retriever's performance affects a QA system.

The Full IR QA System

We used the questions from the train set to evaluate the stand-alone retriever, in order to provide as large a collection as possible. However, BERT has been trained on those questions and would return inflated performance values if we used them for full-system evaluation. So let's resort to our trusty SQuAD2.0 dev set.

> **Note:** This section focuses on a discussion. The code to reproduce our results can be found at the end.

Connecting the retriever to the reader

In our last post, we evaluated a BERT-like model on the SQuAD2.0 dev set by providing the model with a paragraph that perfectly aligned with the question. This time, the retriever will serve up a collection of relevant documents. We created a reader class that leverages the Hugging Face (HF) question-answering `pipeline` to do the brunt of the work for us (loading models and tokenizers, converting text to features, chunking, prediction, etc.), but how should it process multiple documents from the retriever? And how should it determine which document contains the best answer?

This turns out to be one of the thornier subtleties in building a full QA system. There are several ways to approach this problem. Here are two that we tried:

1. Pass each document to the reader individually, then aggregate the resulting scores.
2. Concatenate all documents into one long passage and pass to the reader simultaneously.

Both methods have pros and cons. Let's take a look at them.

Pass each document individually

In Option 1, the reader returns answers and scores for each document. A series of heuristics must be developed to determine which answer is the best, and when the null answer should be returned. For this post, we chose a simple but reasonable heuristic: "Only return null if the highest scoring answer in each document is null; otherwise return the highest scoring non-null answer."

Unfortunately, a direct comparison of answer scores between documents is not technically possible. The reason lies in the type of score returned by the HF pipeline: a softmax probability over all the tokens *in that document*. This means that the only meaningful comparisons are between answers *from the same document* whose probabilities will sum to 1. Comparing an answer with a score of 0.78 from one document is not guaranteed to be better than an answer with a score of 0.70 from another document!

Finally, this option is slower (in our current implementation) because each article is passed individually, leading to multiple BERT calls.

Pass all documents together as one long context

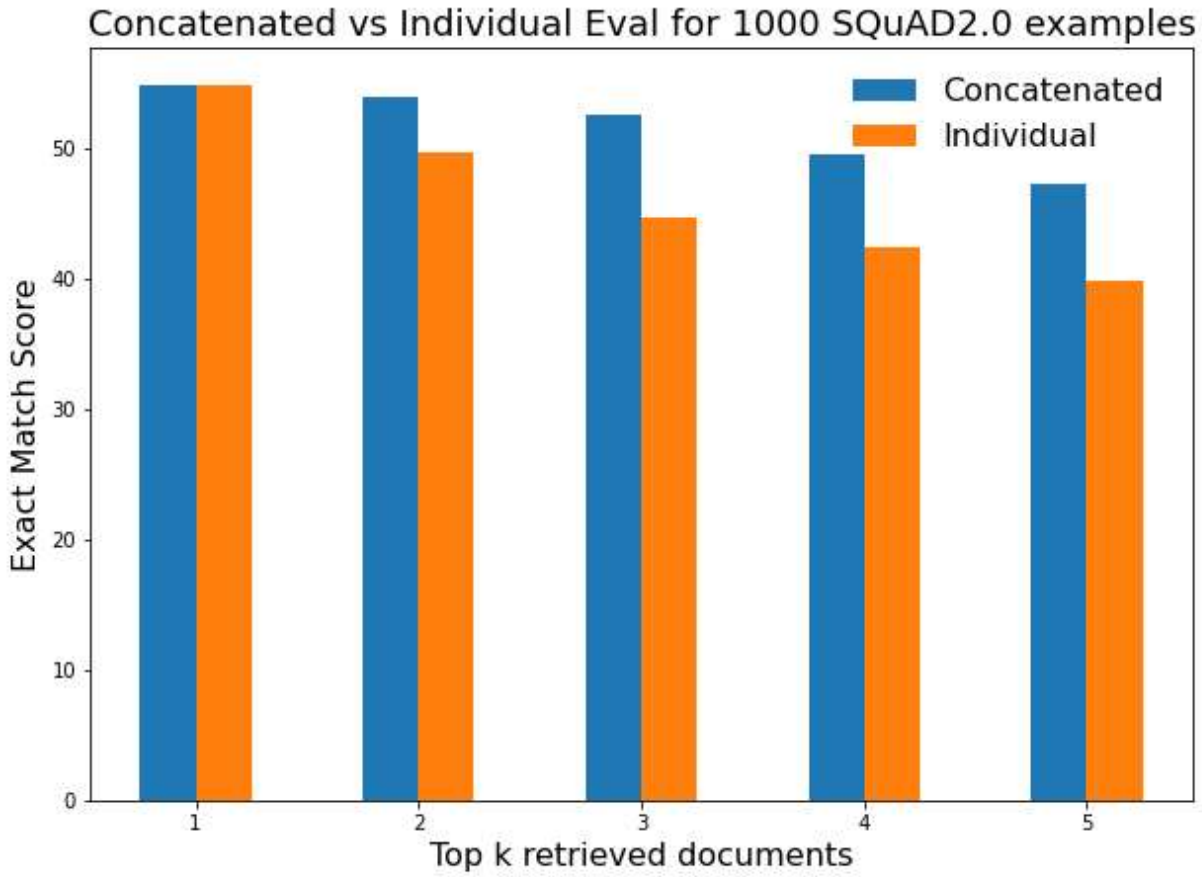
Option 2 circumvents many of these challenges but leads to other problems. The pros are:

1. all candidate answers are scored on the same probability scale,
2. handling the null answer is more straightforward (we did that last time), and
3. we can take advantage of faster compute, since HF will chunk long documents for us and pass them through BERT in a batch.

On the other hand, when concatenating multiple passages, there's a good chance that BERT will see a mixed context: the end of one paragraph grafted onto the beginning of another, for example. This could make it more difficult for the model to correctly identify an answer in a potentially confusing context.

Another drawback is that it's more difficult to backstrapolate which of the input documents from which the answer ultimately came.

Our reader class has two methods: `predict` and `predict_combine`, corresponding to Option 1 and Option 2, respectively. We tested each of them over 1000 examples from the SQuAD2.0 dev set while increasing the number of retrieved documents.



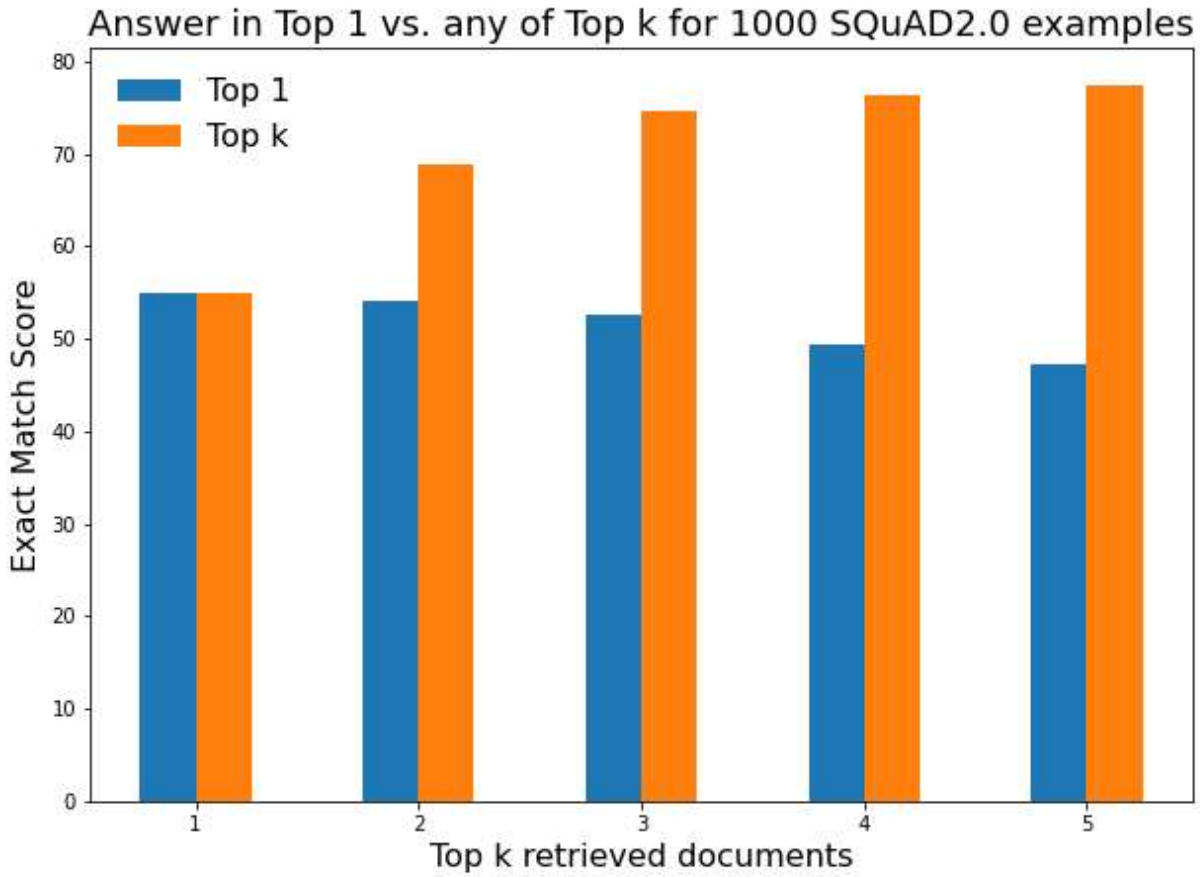
There are two take-aways here. First, we see that the concatenation method (blue bars) outperforms passing documents individually and applying heuristics to the outputs (orange bars). While more sophistical heuristics can be developed, for short documents (paragraphs in this case), we find that the concatenation method is the most straightforward approach.

The second thing to notice is that, as the number of retrieved documents increases, the overall performance decreases for both methods. What's going

on? When we evaluated the retriever, we found that increasing the number of retrieved documents *increased* the likelihood that the correct answer was contained in at least one of them. So why does reader performance degrade?

Evaluating the system

Standard evaluation on the SQuAD2.0 dev set considers only the best overall answer for each question, but we can create another evaluation metric that mirrors our IR recall metric from earlier. Specifically, we compute the *percent of examples in which the correct answer is found in **at least one** of the top k documents provided by the retriever*.



The blue bars are the same as the blue bars in the previous figure, but this time the orange bars represent our new recall-esque metric. What a difference! This demonstrates that when the model is provided with more documents, the correct answer truly is present more often. However, trying to predict which one

of those answers is the right one is challenging: this task is not achieved by a simple heuristic, and becomes harder with more documents.

It may seem counterintuitive, but this behavior does make sense. Let's imagine a simple system that performs ranked document retrieval and random answer selection. Ranked document retrieval, in this case, means that the correct answer is most likely to be found in the top-most ranked document, with some decreasing probability of being contained in the second- or third-ranked document, and so on. As we retrieve more and more documents, the probability *increases* that the correct answer is contained in the resulting set. However, as the number of documents increases, so too do the number of possible answers from which to choose - one from each document. Random answer selection over an increasing number of answer choices results in a *decrease* in performance. Obviously, BERT is not random, but it's also not *perfect,* so the trait persists.

Does this mean we shouldn't use QA systems like this? Of course not! There are several factors to consider:

1. **Use Case:** If your QA system seeks to provide enhanced search capabilities, then it might not be necessary to predict a single answer with high confidence. It might be sufficient to provide answers from several documents for the user to peruse. On the other hand, if your use case seeks to augment a chatbot, then predicting a high confidence answer might be more important for user experience.

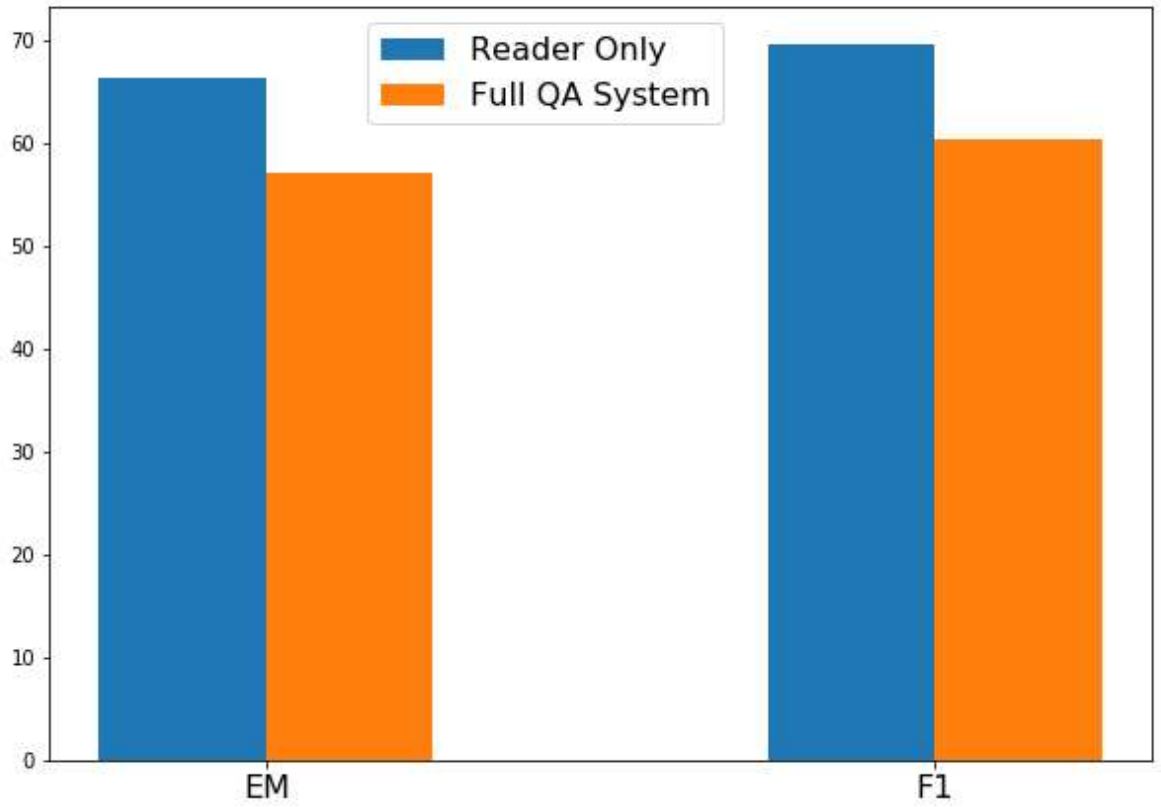
2. **Better heuristics:** While our simple heuristic didn't perform as well as concatenating all the input documents into one long context, there is research into developing heuristics that work. In particular, one promising approach develops a combined answer score that considers both the retriever's document ranking, and the reader's answer score.

3. **Document length:** Our concatenation method works reasonably well compared to other methods, but the documents are short. If the document

length becomes considerably longer, this method's performance can degrade significantly.

Impact of a retriever on a QA system

Considering all that we've learned so far, what is the overall impact of the retriever on a full QA system? Using our concatenation method and returning only the best answer for all questions in the SQuAD2.0 dev set, we can compare results with our previous blog post in which we evaluated only the reader.



As expected, adding a retriever to supply documents to the reader reduces the system's ability to identify the correct answer. This motivates approaches for enhancing the retriever, in order to supply the reader with the best documents possible.

Final Thoughts

We did it! We built a full QA system with off-the-shelf parts using ElasticSearch and HuggingFace Transformers.

We made a series of design choices in building our full QA system, including the choice to index over Wikipedia paragraphs rather than full articles. This allowed us to more easily replicate SQuAD evaluation methods, but this isn't practical. In the real world, a QA system will need to work with existing indexes, which are typically performed over full documents (not paragraphs). In addition to architectural constraints, indexing over full documents provides the retriever with the best chance of returning a relevant document.

However, passing multiple long documents to a Transformer model is a recipe for boredom -- it will take forever and it likely won't be highly informative. Transformers work best with smaller passages. Thus, extracting a few highly relevant paragraphs from the most relevant document is a better recipe for a practical implementation. This is exactly the approach we'll take next time when we (hopefully) address the biggest question of all:

> *How do I apply a QA system to **my** data?*

Stay tuned!

Post Script: The Code

If you open this notebook in Colab, you'll find several cells below that step through the experiments we ran for the final section.

Sign in to comment

Styling with Markdown is supported

Sign in with GitHub

Subscribe

CFF builds a state-of-the-art QA application with the latest NLP techniques