Big, fast human-in-the-loop NLP with Elasticsearch

Joel Klinger Follow

Oct 25, 2019 · 11 min read ★

> ***Part I: the keyword factory*** *tl;dr: You could make one of these if you 1) store your data in Elasticsearch 2) use the* `clio_keywords` *function from the* `clio-lite` *package, pointing at your Elasticsearch endpoint 3) Host it in a Flask app, such as this.*
>
> ***Part II: a contextual search engine*** *tl;dr: You could make one of these if you 1) store your data in Elasticsearch 2) host a lambda API gateway using* `clio-lite` *3) Interrogate it with your own front-end, or use something out-of-the-box like searchkit*

The current paradigm for day-to-day working of many NLP Data Scientists is to throw open a laptop, fire up Python or R, make some models and wrap up some conclusions. This can work very well for exploratory analysis, but if you need to put a human (such as an expert, your boss, or even yourself) in-the-loop, this can become prohibitively slow. In this two-part blog, I'll try to convince you that there are bigger, faster ways to do NLP.

Big fast human-in-the-loop NLP (Photo by Stephen Hateley on Unsplash)

Human-in-the-loop NLP and me

In my day-to-day work at Nesta, I develop tools and infrastructures to enable people to make better decisions, and for people to be able to make those decisions with up-to-date data. We deliver tools to local, national and international policymakers and funders who rely on being kept abreast of the latest innovations in science, technology and society. Since these people are accountable for their decisions, this generally rules out tools which adopt black-box procedures. 'Human-in-the-loop NLP' is our way of addressing these needs, not least since unstructured text data is one of the richest, most available, and up-to-date forms of data.

Elasticsearch

Data engineers, database admins and devops engineers should all be familiar with the 'elastic stack' (with Elasticsearch sitting at it's core) as the go-to technology for storing and analysing log files or building search engines, though perhaps many of them aren't so familiar with the vast potential for data science research. Meanwhile many data scientists have, at best, a basic familiarity with Elasticsearch as a data storage technology.

In short: Elasticsearch is a database for search engines that is able to perform lightning-fast searches because of ***how the data is stored***.

In Elasticsearch, documents are stored as **term-frequency vectors** (a procedure known as 'inverted indexing') and the **document-frequency** is pre-calculated for each term. This means a couple of things:

1. Term-by-term co-occurences are incredibly fast to **extract** *on the fly*.
2. Important terms can be identified via the standard data science 'tf-idf' procedure *on the fly*.

From a data scientist's perspective an Elasticsearch database is a very basic (but powerful) pre-trained model for extracting keywords, synonyms, similar documents and

outliers. In this two-part blog, I'll touch on all of these using out-of-the-box functionality (although more sophisticated approaches could certainly be taken).

Part I: the keyword factory

The simplest case: allowing non-experts to identify their own keywords and synonyms

Generating lists of keywords (or synonyms) is a common NLP task for data scientists. It can have applications from dimensionality reduction to topic modelling, and can also be useful for human-in-the-loop analysis by providing human analysts with a data-driven set of terms which can be used for more laborious tasks.

Many data scientists would tackle this using appropriate python (or R) packages in order to produce fairly static outputs, which for reports or papers is fine. Most of the time we have the best intentions for our results to be accessible to non-experts, but in practice people end up getting what they're given: static outputs. What are our options for a scalable and flexible tool which can be used by non-experts?

Python (or R) packages: The right tools for the wrong job

The are tons of python-based approaches which can solve this problem, such as topic modelling (e.g. Latent Dirichlet Allocation or Correlation Explanation), clustering on word vectors (such as word embeddings or vanilla count vectors) or using co-occurence matrices or networks.

All of these approaches can give very sensible results, and to be honest I'm not attempting to beat these approaches on perceived accuracy. If I was tasked with doing a one-off ad-hoc analysis, I could consider any of the above.

However, I'm a fan of scalable, shareable and flexible solutions to problems:

- **Scalability**: all of my suggested python solutions require data and models to sit in memory. For large number of documents, or large vocabularies, the memory consumption will be heavy. One solution to this would be to sample the data at the expense of the 'depth' of your model.

- **Shareability**: how can you share results with a non-expert without installing python packages on their laptop, whilst hoping that your setup is compatible with their laptop? Two possibilities could be to host your machine learning models (which could be anything from a cluster model to a co-occurence matrix) on a remote server (but beware of that large memory overhead!), or alternatively you could pre-generate static sets of keywords (which is very simplistic).

- **Flexibility:** imagine you want to update your model with one more document, or decide to filter your data in-situ — it's not trivial to do this with regular machine learning models. Your best approach would likely be to pre-generate one model per pre-defined filter, which is computationally expensive.

Elasticsearch: The right tool for the right job

Remembering that Elasticsearch is effectively a pre-trained model of term co-occurences, filterable by term significance, it is clear to see why it can natively generate lists of keywords on the fly. What's more, any method we apply to Elasticsearch is inherently scalable, shareable and flexible:

- **Scalability**: Elasticsearch is performant up to the petabyte scale.

- **Shareability**: Elasticsearch exposes methods on the data via a simple REST API. To do sophisticated tasks you can simply string together some lightweight python code hosted on a remote server.

- **Flexibility**: updating your 'model' is as simple as adding a new document to the server. Filtering your data by any field is a bread-and-butter operation.

The "significant text" aggregation

In principle we could implement our own procedure from scratch for extracting keywords, however there are a couple of shortcuts you can employ just by using out-of-the-box features of Elasticsearch.

The following python code wraps up a query to the Elasticsearch API (replace `URL` with your own endpoint and `FIELD_NAME` with the name of the field(s) in your database you would like to query):

```
import requests
import json

def make_query(url, q, alg, field, shard_size=1000, size=25):
    """See this gist for docs"""
    query = {"query" : { "match" : {field : q } },
             "size": 0,
             "aggregations" : {
                 "my_sample" : {
                     "sampler" : {"shard_size" : shard_size},
                     "aggregations": {
                        "keywords" : {
                            "significant_text" : {
                               "size": size,
                               "field" : field,
                               alg:{}
                             }
                        }
                    }
                }
            }
        }
    return [row['key']
            for row in requests.post(f'{url}/_search',
                                     data=json.dumps(query),
                                     headers={'Content-
Type':'application/json'}).json()['aggregations']['my_sample']
['keywords']['buckets']]
```

Under the hood, this query is doing the following:

1. Finding all documents containing the text `query` in the field `field`.
2. Extracting the `size` most significant terms from `field`, calculated according the the `jlh` algorithm.

On top of that, increasing the size of `shard_size` will increase stability (and depth) of your 'model', at the expense of computational performance. In practice, you will only expect your model to become less stable for very rare terms — in which case you could build a workaround for this.

Performance before tweeks: arXiv data

I've got all of arXiv's scientific publications in my Elasticsearch database, and this is how it performs on the abstract text for the following queries:

```
python pandas

['pandas', 'numpy', 'package', 'scipy', 'scikit', 'library', 'pypi',
'cython', 'github']

_____________________________

elasticsearch

['kibana', 'lucene', 'hadoop', 'retrieving', 'apache', 'engine',
'textual', 'documents', 'ranking']

_____________________________

machine learning

['learning', 'training', 'algorithms', 'neural', 'supervised',
'automl', 'intelligence', 'deep', 'tasks']

_____________________________________________________________

drones and robots

['robot', 'drones', 'robotics', 'robotic', 'humanoid', "robot's",
'drone', 'autonomous', 'mobile']

_____________________________
```

…and this is out-of-the-box functionality! Some easy criticisms would be:

1. N-grams aren't leveraged at all, neither in the query nor the results. For example `machine learning` is treated as `{machine, learning}` rather than `{machine learning, machine, learning}`.
2. It is not impossible for stop-words to appear in the results.
3. Out-of-sample misspellings aren't dealt with at all.
4. Plurals and possessive forms of nouns and all verb conjugations are listed separately.

I'm not going to address the latter two points here, but dealing with them is fairly trivial. For example, **misspellings** can be dealt with in at least two ways, for example using:

- Elasticsearch's n-gram tokenizer (beware that Elasticsearch's defines n-grams at the character-level, not to be confused with the data scientist's term-level n-gram)
- or using the phonetic tokenizer plug-in.

In order to deal with n-gram queries (such as `machine learning`) I have created a field in my Elasticsearch database containing pre-tokenized abstracts, in which I have already identified n-grams. Note that the 'schema' for this field can be found here. If you want to know, I process my n-grams by using a look-up table based on n-grams from Wiktionary (but a more data-driven approach would also work).

Although I haven't implemented this myself, similar preprocessing can be done for **plurals/possessives/conjugations**, effectively by replacing all non-simple forms of terms by their simple form.

Finally, to avoid the potential embarrassment of **returning stop-words** I have taken to generating them from the data using my `make_query` function:

```
and of but yes with however

['however', 'but', 'not', 'answer', 'with', 'the', 'is', 'of', 'to',
'a', 'in', 'and', 'that', 'no', 'this', 'we', 'only', 'for', 'are',
'be', 'it', 'can', 'by', 'on', 'an', 'question', 'also', 'have',
'has', 'which', 'there', 'as', 'or', 'such', 'if', 'whether',
'does', 'more', 'from', 'one', 'been', 'these', 'show', 'at', 'do']
```

and I simply exclude these from the returned results.

Putting it all together

Check out the keyword factory in action over on the arXlive website, featuring n-grams and stopword removal. You can make your own Flask app using the `clio_keywords` function from the `clio-lite` package. Have fun!

Part II: a contextual search engine

Consider a very technical dataset such as that from arXiv, the world's largest repository of physical, quantitative and computational science pre-prints. Assuming that you're not polymath savant, what would be your strategy for finding the latest novel research on

arXiv, relating to ***Big Data and Security***? If you were to make that exact search on arXiv, you'd find yourself with a decent set of results, but the problem is that when you were searching for ***Big Data***, you might not have realised that you also wanted to include tangentially-related terms such as *{hadoop, spark, cloud-computing}* in your query. What if there was some big breakthrough in ***Cloud Computing and Security*** that you've been missing out on? (TL;DR ***this is the same search with a 'contextual' search engine***)

I'm going to break this problem down into two pieces, and solve it by using some Elasticsearch functionality wrapped up in Python:

- Firstly, how do you make a decent search query without being a genius?
- Secondly, how can we define novelty?

Making a decent query, without being a genius

Back to the problem of making a decent search query. What approaches might a genius take? Well, they could go down the route of '**keyword expansion**', for example by considering all of *{hadoop, spark, cloud-computing}* in addition to ***big data***, and all of *{attack, encryption, authentication}* in addition to ***security***. Potentially this is a promising way to go, and we've already written the tools to help do this in the previous blog. However, the main problem with the 'keyword expansion' approach is that it lacks ***context***. A natural extension to this would instead be '**document expansion**', and thankfully Elasticsearch has this feature built-in.

More-like-this

Well, to be honest with you, Elasticsearch's *more-like-this* query is really 'keyword expansion++' rather than a 'document expansion', as you might imagine it in a vector space. Under the hood, representative terms are selected from input documents (according to a highly configurable procedure) which you would like to 'expand' from. The advantage to doing this over a pure 'keyword expansion' approach, is that terms which co-occur with all input terms are deemed to be more important than those which only co-occur with a subset of the input terms. The upshot is that the expanded set of keywords which are used to seed the 'document expansion' can be assumed to have a high degree of contextual relevance.

So here's my strategy:

- Make a vanilla query to Elasticsearch and retrieve the 10–25 most relevant documents. These will be our 'seed' documents.
- Follow up with a *more-like-this* query, using the seed documents.

and that strategy looks a little like this (in practice it's a bit more code, so it actually looks like this):

```
# Make the initial vanilla query
r = simple_query(url, old_query, event, fields)
data, docs = extract_docs(r)

# Formulate the MLT query
total = data['hits']['total']
max_doc_freq = int(max_doc_frac*total)
min_doc_freq = int(min_doc_freq*total)
mlt_query = {"query":
             {"more_like_this":
              {"fields": fields,  # the fields to consider
               "like": docs,  # the seed docs
               "min_term_freq": min_term_freq,
               "max_query_terms": max_query_terms,
               "min_doc_freq": min_doc_freq,
               "max_doc_freq": max_doc_freq,
               "boost_terms": 1.,
               "minimum_should_match": minimum_should_match,
               "include": True  # include the seed docs
              }
             }
            }

# Make the MLT query
query = json.dumps(dict(**query, **mlt_query))
params = {"search_type":"dfs_query_then_fetch"}
r_mlt = requests.post(url, data=query,
                      headers=headers,
                      params=params)

# Extract the results
_data, docs = extract_docs(r_mlt)
```

Note, that I serve this functionality via AWS API Gateway in a Lambda function. The code for deploying the above function can also be found in the same repo.

Defining novelty

Novelty doesn't have a particularly narrow definition, and I admit that my definition for this blog is going to be fairly narrow…

Novelty generally could be defined as any (and more) of {new, original, unusual}, and my definition is going to straddle the {original, unusual} concept. More formally (but not very formally), I'm asking the following question of each document in Elasticsearch:

> *How **different** are you from your **nearest neighbours**?*

Just to clarify the logic here: if the total sample of documents is unbalanced, then a document belonging to a minority topic will be very different from the average document. We can avoid this by only comparing to nearest neighbours. And how better to get the nearest neighbours, than by using *more-like-this* all over again (the full code is here):

```
mlt_query = {
    "query": {
        "more_like_this": {
            "fields": fields,  # field you want to query
            "like": [{'_id':doc_id,     # the doc we're analysing
                      '_index':index}],
            "min_term_freq": 1,
            "max_query_terms": max_query_terms,
            "min_doc_freq": 1,
            "max_doc_freq": max_doc_freq,
            "boost_terms": 1.,
            "minimum_should_match": minimum_should_match,
            "include": True
        }
    },
    "size":1000,  # the number of nearest neighbours
    "_source":["_score"]
}

# Make the search and normalise the scores
r = es.search(index=index, body=mlt_query)
scores = [h['_score']/r['hits']['max_score']
          for h in r['hits']['hits']]

# Calculate novelty as the distance to a low percentile
delta = np.percentile(scores, similar_perc)
return 1 - delta
```

Naturally, any decent a definition of novelty which encapsulates the concept of being "unusual" should capture bad data, since one would hope that these would be unusual. I found that by applying the above novelty scorer to arXiv data I was able to root out a whole bunch of bad data, such as plagiarised and 'comment' articles. I went on to assign these a novelty of zero, but I'm sure you can work out your own approach!

One downside of this novelty-scoring method is that it's relatively slow to implement (it must be calculated on a document-by-document basis), and for this reason I preprocessed all of the documents in my Elasticsearch database.

Putting it all together

So, with some overzealous use of Elasticsearch's *more-like-this* query we're able to make expansive searches whilst deriving a very lightweight measure of novelty. Check out `clio-lite` to understand the code better, and **if you want to see this in action**, check out the ***HierarXy*** search engine of arXiv data. Note that I've also used the same pre-processing as described in Part I, as well as the data cleaning described in this blog. Thanks for reading!

Elasticsearch Python Machine Learning NLP Data Science