Mehdi Boukhechba

Mob3f@virginia.edu

www.boukhechba,com

+1 (434) 227 3821

DRFIRST DATA CHALLENGE

1. EXERCISE 1

The data contains search results of a search engine including (1) request time, (2) response time, (3) the actual search string, (4) size of the request, (5) size of the response, and (6) the total results received from the server. I also calculated (7) the execution time which is the difference between the request and response times. Figure 1 and Table 1, summarize the variables 4-7.

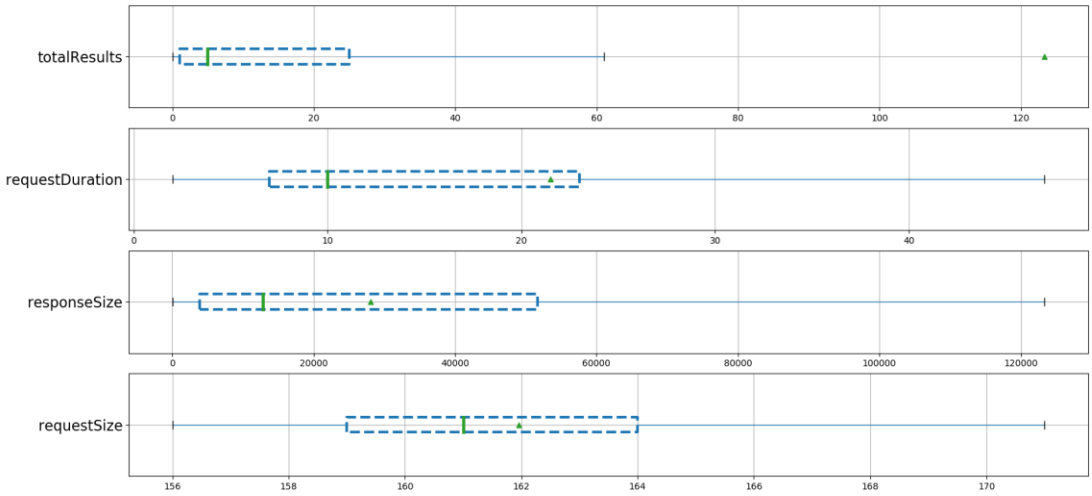

Figure 1. Box plots describing variables 4,5,6, and 7.

Table 1.Summary statistics of variables 4,5,6, and 7.

	requestSize	responseSize	totalResults	requestDuration (ms)
count	41869	41869	41682	41869
mean	161.9590389	28057.53073	123.3322057	21.51145239
std	5.017654777	32441.48821	466.3211663	32.03816264
min	117	41	0	2
25%	159	3876	1	7
50%	161	12872	5	10
75%	164	51681	25	23
max	267	248360	3853	720

The total number of records is 41869. The average execution time was 21.51 (SD=32.03). While the average number of returned results was around 123 (SD=466), the median was around 7. From Figure 1 we can see that most of the times, the server returned between 0 and 25 results. Specifically, around 25% of our data had either 1 or 0 results. This means that the current search technique has some issues. A good search engine would very rarely return an empty result.

Figure 2 shows the fluctuation in number of requests per days and hours of the day. We can observe a noticeable decrease in number of requests between 03/03 and 06/03. The radar plot shows that generally, requests happened between 10am and 5pm.

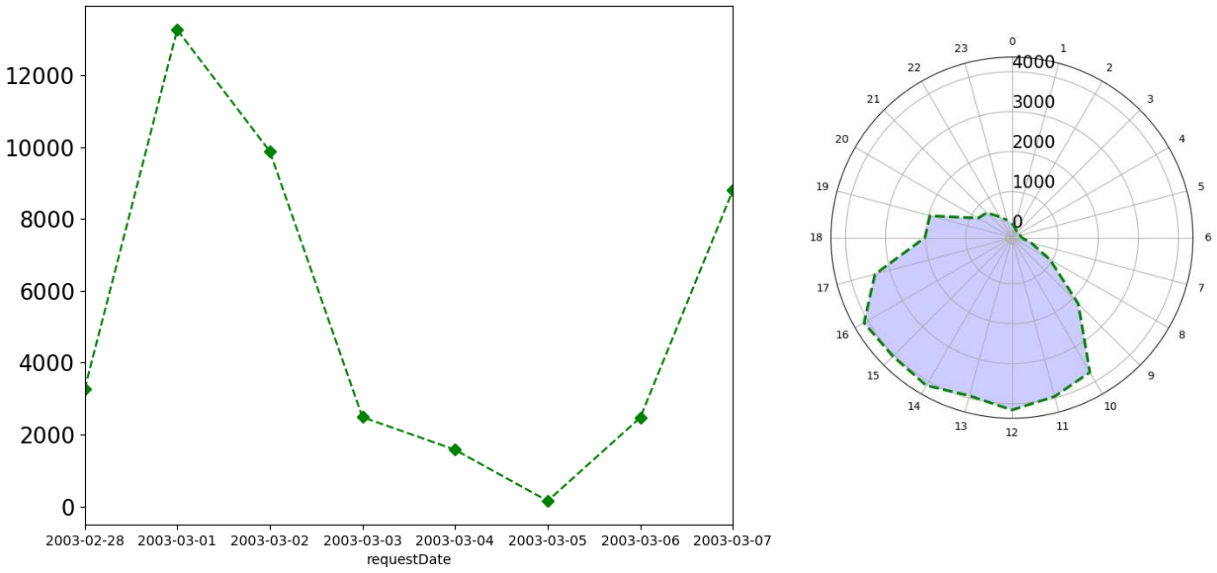

Figure 2. Number of requests over time. Left- number of requests per day. Right= number of requests per hours

The number of unique strings is 8678 but this is number is not representative because it includes all iteration before typing in the final search word (e.g., {N, No, Nor, Norc, Norco}). This will be discussed in the next exercise.

2. EXERCISE 2

Since User ID, IP address or any other user identifying information is not present on this dataset, estimating the number of participants is quite challenging. Clustering the requests over time as a proxy to identifying users (e.g., if requests are very close together we assume that it's the same participant) is not safe because multiple users can be performing requests at the same time. What I propose here, is to group together requests that are indicative of some typing pattern, e.g., {N, No, Nor, Norc, Norcp,Norc,Norco} that are close in time but not necessarily adjacent. Which means, my algorithm is able to handle the case where multiple participants are making requests at the same time. For instance from this sequence that is happening almost on the same 2-seconds window {lor, methylphenidate HC, methylphenidate HCl, lora}, it's clear that this this data belongs to two different participants (and not 1 or 3 participants).

I implemented my algorithm so that it first sorts the data based upon the request time, then processes the data sequentially by grouping together words that are very close to each other in semantic and in time. I consider two words w1 and w2 as close to each other if (1) w1 contains w2 e.g. {lor,lo} or (2) w2

contains w1 e.g. {lor,lora} or the Levenshtein distance is smaller than 3, e.g., {lor,los} (Levenshtein_distance is a similarity index that calculates how many characters you need to change to make two strings equal). Also, the two words should be within a certain time window. In my implementation I used 30 secs, but can easily be changed to another threshold. It means if for a given word, the algorithm doesn't find a similar word after 30 secs, it considers it as a separate word/user.

When going through the data, I constantly use a dictionary that keeps track of the last seen word searches alongside with its timestamp (when seen for the last time). If the new word is similar to one of the words in our dictionary, the new word will replace the word in the dictionary, and the associated timestamp will be updated. As the same time, the algorithm checks for expired words in the dictionary (words that have been in the dictionary for too long, typically >30 secs), those words will be deleted and placed in the result dictionary that stores users' visit time and their search word. The complexity of this algorithm is linear since it reads and process the data only one time.

Then, I consider each group of words as a separate user (see file: search_data_users.csv). So let's plot the estimated number of participants over time.

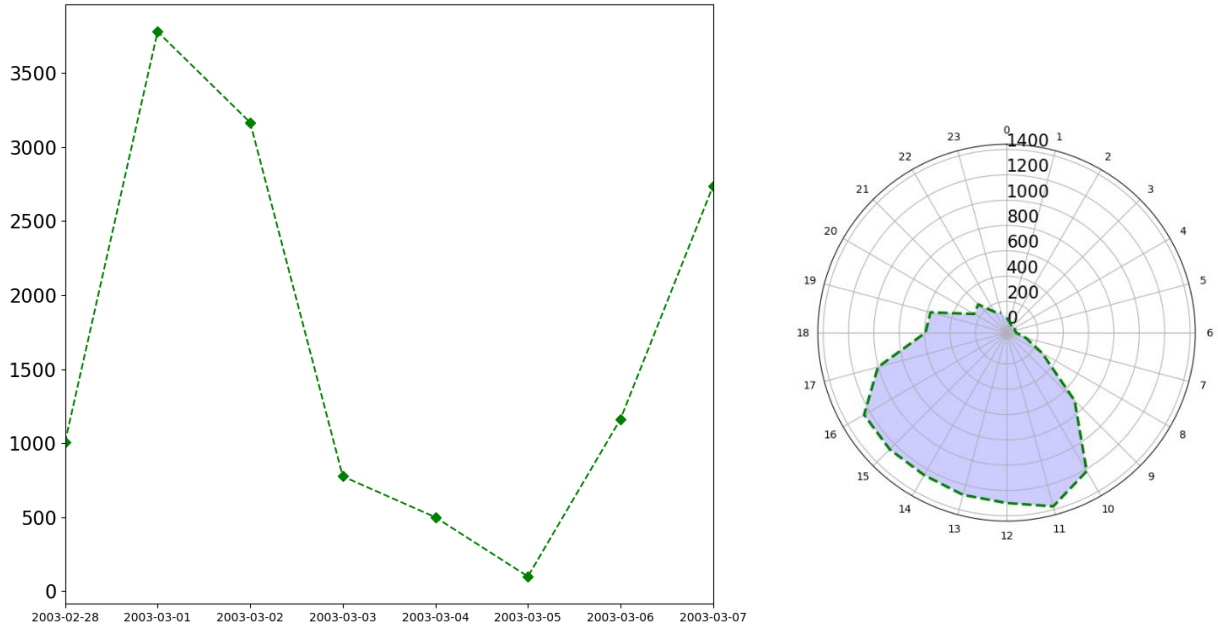

Figure 3. Number of estimated users over time. Left- number of requests per day. Right= number of requests per hours

If we compare Figure 2 and Figure 3, we can see that we have almost similar trends of visits over time with slight differences. For instance, on 03/06 and 03/04 we had similar number of requests (see Figure 2) but the number of estimated users on 03/06 was higher than 04/04 (see Figure 2).

Number of total estimated users is 12535. The number of unique words is 2701. In Figure 4, I plot the top 20 request after applying my clustering algorithm. Norco is the most popular search with more than 450 requests.

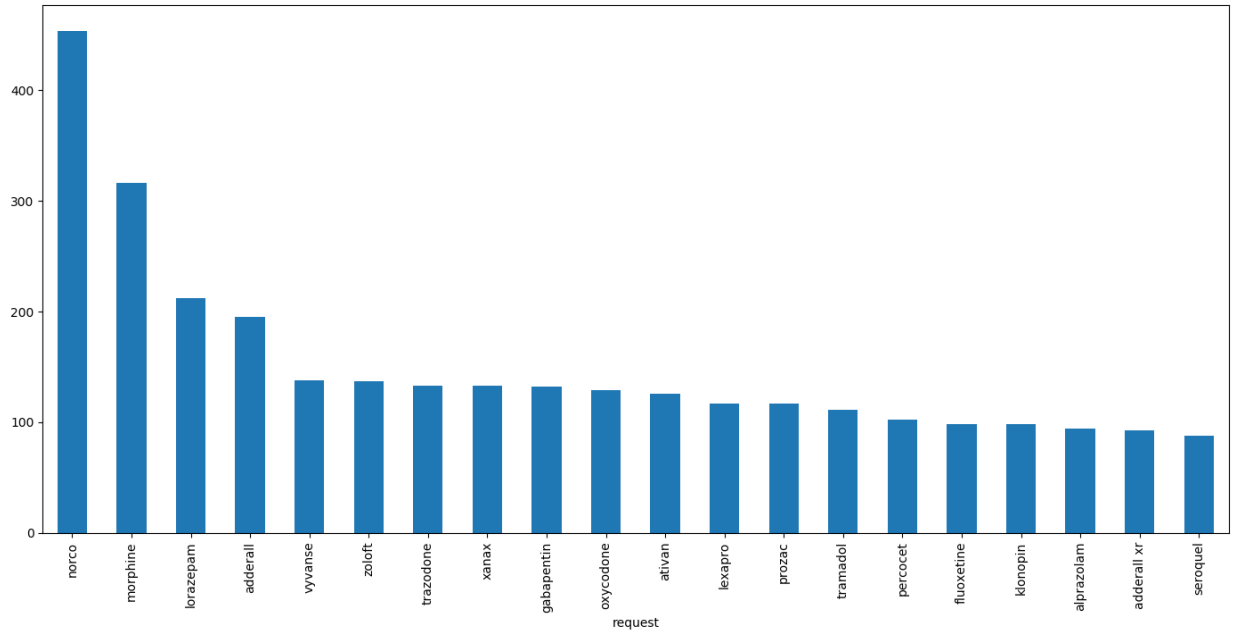

Figure 4. Top 20 search words after applying my clustering algorithm

This method has a limitation, if the same user searches two different words, he/she will be considered as two separate users. We can further assume that two separates word requests happening very close in time might belong to the same user, but I found that it will include a lot of uncertainty. Or, if we can compute how many requests do people perform using another existing dataset, we can have a probability that we can use to estimate if two close searches belong to the same user or not.

3. EXERCISE 3

Problem formulation

The exercise 3 is about search engine optimizations (SEO). The goal is to improve the current search model. I don't know what is the current search technique but I assume it's is a simple SQL query similar to "select name from medication where name like '%request%'" or "where name like 'request%'". The reason why this method is not efficient is because of the following reasons: (1) even though most databases are using B-tree to index this kind of operations, it still has an O(n) complexity. (2) if a user misspells a prefix, the DB won't be able to find the desired results, for instance, from the provided data, one user was looking for "adderall xr" by writing "Adderral", and the system returned 0 result. So, it is important to find a better solution based on similarity and not trying to exactly match the strings. (3) a good search engine would be able to understand what the user is looking for rather than just finding the exact word he/she is typing-in. For instance, if you type "car" on Google, it will suggest things about cars, but also it will suggest to look at other types of transportation like airplanes, trains...etc. This is possible because Google's algorithms are able to understand that a car is a transportation mode.

Existing works

Our problem is very similar to autocomplete, and spellchecking problems. Autocomplete solutions are mainly based on indexing data using tree structures. They are based on algorithms like B+tree [1] and Trie [2] which are known to be very fast at finding words with similar prefix. A trie has also be shown to be faster than the other algorithms at finding the first N children from a substree, because it visits less nodes than in a B+ Tree scenario. However, as mentioned earlier, in our case, we don't want to have an

autocomplete function because medication names are sometime hard to spell, thus people can misspell a prefix. It has been previously been proved that the rate of misspelling in medical records is 10% higher than the rate for other texts [3].

So let's take a look at what has been done in terms of spellchecking literature. There are two specific forms of spelling correction: isolated-term correction and context-sensitive correction. In isolated-term correction, we attempt to correct a single query term at a time - even when we have a multiple-term query. In context-sensitive correction we take into account the context (words before and after) to correct the current word. This is pretty much what MS Office is doing to correct "it have" to "it has". I will focus on the fest type of spellchecking because in our case it's rare to have more than one word in the request. The followings are some widely used approaches in this area.

Naïve approaches are based on calculating a similarity index between the query term and each term in our dictionary (or database). Examples of similarity algorithms are: Jaro–Winkler distance, Levenshtein distance (this is what I used in Exercise 2), and Soundex. Then, the next step would be to select the term having the smallest distance. This exhaustive search is inordinately expensive because it requires reading the whole dictionary on every keystroke trigger.

Other more sophisticated approaches are based on N-gram analysis (aka Fuzzy Search). N-grams are simply all combinations of adjacent words or letters of length n that you can find in your source text. For instance, the character level bigram of Norco would be {No,or,rc,co}. N-grams have been proven useful for detecting misspelling error because misspelled words tend to result in improbable n-grams.

Many ngrams-based spell checkers exist. Some use similarity index between unique n-grams of two words. Others use probabilistic techniques to detect and correct the misspelled words. For instance, if the ngram-derived probability of an observed word is lower than that of any word obtained by replacing one of the n-grams with a spelling variation, then we hypothesize that the original is an error and the variation is what the user intended. Then the most frequent variations will be chosen and displayed as suggestion to the user.

The most interesting ngrams-based method that I crossed is named n-gram overlap method. It has been developed by Stanford NLP group [4]. The idea is that instead of comparing the n-grams of the query with the n-grams of all words in the dictionary, they basically take the n-grams of the request and find all words that have an overlap with one of the request's n-grams. This is done using an n-grams index table that would have been previously calculated. The index table stores an index for all unique n-grams found in the dictionary, and a list of their corresponding terms. Then the algorithm calculates Jaccard coefficient and select terms having a higher coefficient than a threshold (typically 0.8) (see [5] for more details).

Another very interesting method I found is Symmetric Delete Spelling Correction (SymSpell)[6], the idea is to first generate terms with an edit distance from each dictionary term and add them together with the original term to the dictionary. This has to be done only once during a pre-calculation step. Then, for each new request, delete characters from the misspelled word one character at a time and look up the prepared word in each deletion step to see if the word is seen in the previously hashed dictionary. For a word of length n, an alphabet size of a, an edit distance of 1, there will be just n deletions, for a total of n terms at search time. Authors claim that this method can be 100 times faster than other algorithms such as BK-trees and Norvig. The SymSpell algorithm is constant time (O(1) time), i.e. independent of the dictionary size. Tries have a comparable search performance to SymSpell. But a Trie is a prefix tree, which requires a common prefix. This makes it suitable for autocomplete or search suggestions, but not applicable for spell checking. If your typing error is e.g. in the first letter, then you have no common prefix, hence the

Trie will not work for spelling correction. This algorithm has several implementations in C++, Java, JavaScript and Python.

I saved the best for last. The last algorithm I am covering is graph-based algorithm called Finite State Transducer (FST). This is graph having a start state and an end state, states in a graph between the two, and there are labelled transitions between states representing letters.

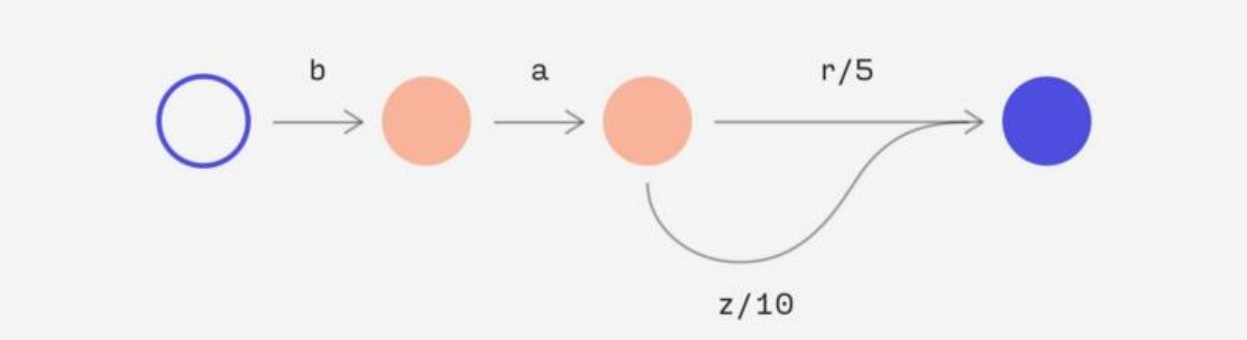

Figure 5. An FST allows us to associate a weight to a path. In this example, "bar" has a weight of 5 and "baz" has a weight of 10.

The FST allow us to associate a numeric (or really just commutative) value with a valid input sequence. For autosuggest that's useful because it allows us to associate a weight or score with a suggestion (see Figure 4). Moreover, given some string s and a maximum edit distance d, this algorithm can construct a Levenshtein Automata (or other distance metric) that will only accept strings that are within d edits of s. We can use this to produce typo-tolerant suggestions. As we depth-first traverse our FST, we can feed our steps into the Levenshtein Automata. If it accepts the input character, we proceed. If it doesn't, we can ignore that whole branch, and then backtrack. This technique has been proven to be extremely fast, with O (1) time complexity. It is therefore used in many search engine backends such as ElasticSearch and Solr to power autosuggest.

All previously algorithms will suggest one or more similar terms. Then when multiple terms are found, algorithms usually select the one that is more common. For instance, grunt and grant both seem equally plausible as corrections for grnt. Then, the algorithm should choose the more common of grunt and grant as the correction. The simplest notion of more common is to consider the number of occurrences of the term in the collection; thus if grunt occurs more often than grant, it would be the chosen correction. A different notion of more common is employed in many search engines such as Google. The idea is to use the correction that is most common among queries typed in by other users. The idea here is that if grunt is typed as a query more often than grant, then it is more likely that the user who typed grnt intended to type the query grunt.

Implementation

Now that we covered some interesting algorithms for fuzzy string matching or information retrieval, let's talk about implementations. I would suggest to use ElasticSearch since it implements almost all algorithms I mentioned above. Elasticsearch is a search engine based on the Lucene library. It provides a distributed, multitenant-capable full-text search engine with an HTTP web interface and schema-free JSON documents. Elasticsearch (ES) is developed in Java and is released as open source under the terms of the Apache License [7]. It has also some very powerful visualization libraries such Kibana that allows to monitor our system (e.g., number of unique @IP, top requests, etc.). ElasticSearch is ranked number 1 for SEOs and has been used by many big companies such as NetFLix, StackOverflow, and LinkedIn.

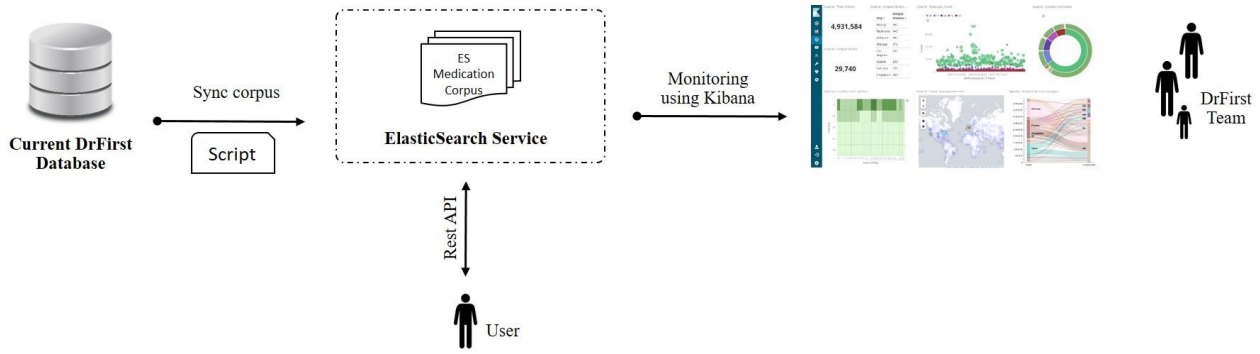

Figure 6. The proposed solution in a production-level environment

I would suggest to use the Term suggester function of ES [8] since it uses the powerful Finite State Transducer. This function accepts many parameters such as (1) the maximum edit distance candidate that suggestions can have in order to be considered as a suggestion, (2) which string distance implementation to use for comparing how similar suggested terms are (e.g., damerau_levenshtein, ngrams, jaro_winkler), and (3) how suggestions should be sorted per suggested text term. I suggest to tune these parameters iteratively and choose the parameter giving the highest performance. This brings us to the experimentation. How to test those parameters and how test if our model is doing well?

I suggest to use the current DrFirst medication corpus for experimentation. Let's imagine the size of our corpus is n, we will create a script that takes each term in the corpus and creates many different types of misspelling (e.g. removing a letter, adding a letter, removing a bigram), of course for each misspelled word we would have a ground truth (the correct word), then we will compute several performance metrics such as (1) if the correct word figures in the top *i* suggested terms (*i* can be 10 for example), (2) the rank of the suggested correct word (a good search engine would have a mean near 1), (3) memory usage and (4) execution time.

These metrics will be also used to monitor the performance of our model after deployment. For instance, if we find that metric number 1 decreases significantly, we might consider enriching our corpus. Maybe there have been some new drugs on the market that our corpus is not aware of.

A very simplified architecture is shown in Figure 5. The proposed ES service will be independent of the current DrFirst database, which means that the solution can be implement with minimum adaptations and with any type of database (relational, NoSQL). A simple cron job script (python for instance) can be used to periodically sync the medication corpus between the database and the ES service. Users will be able to use our ES search function using a simple HTTP get request. DrFirst team will be able to continuously monitor system state using a comprehensive dashboard powered by Kibana.

Further improvements

After accurately finding the medication name that the user is looking for, I suggest to add another layer that will look at the context of this request. For instance, it would be interesting to return not only the requested medication name, but also a list of other modifications that are related to the request (e.g. same company, same symptom, etc.). This can figure on a list displayed to the user as "You may also want to look at:". This is pretty common on search engines especial in E-commerce. We can for example use another corpus that includes prescription data, and use word embeddings to find drugs that are prescribed together. Than when a user searches for a specific drug, we will also suggest top 10 other drugs that are very close to this drug (usually using cosine similarity).

Another functionality is to look at the sequence of what people are requesting, for instance if a user search medication 1 than medication 2, they have a high probability that those two drugs are related to each other. Using similar word embeddings techniques, we will be able to display to the user something like: "People also looked at: …". This may improve user experience, and effectiveness of our search engine. In terms of implementation, ES is able to effectively work with word embeddings.

4. REFERENCES

[1] T. Hu and J. Zhong, "Database index optimization algorithm based on cluster B+ tree: Database index optimization algorithm based on cluster B+ tree," *Journal of Computer Applications*, vol. 33, no. 9, pp. 2474–2476, Nov. 2013.

[2] L. Liu and Z. Zhang, "Similar string search algorithm based on Trie tree: Similar string search algorithm based on Trie tree," *Journal of Computer Applications*, vol. 33, no. 8, pp. 2375–2378, Nov. 2013.

[3] H. D. Tolentino *et al.*, "A UMLS-based spell checker for natural language processing in vaccine safety," *BMC Medical Informatics and Decision Making*, vol. 7, no. 1, Dec. 2007.

[4] "Spelling correction." [Online]. Available: https://nlp.stanford.edu/IR-book/html/htmledition/spelling-correction-1.html. [Accessed: 14-Dec-2018].

[5] "k-gram indexes for spelling correction." [Online]. Available: https://nlp.stanford.edu/IR-book/html/htmledition/k-gram-indexes-for-spelling-correction-1.html. [Accessed: 15-Dec-2018].

[6] W. Garbe, *SymSpell: 1 million times faster through Symmetric Delete spelling correction algorithm: wolfgarbe/SymSpell*. 2018.

[7] "Elasticsearch," *International Journal of Modern Trends in Engineering & Research*, vol. 5, no. 5, pp. 23–28, May 2018.

[8] "Term suggester | Elasticsearch Reference [6.5] | Elastic." [Online]. Available: https://www.elastic.co/guide/en/elasticsearch/reference/current/search-suggesters-term.html. [Accessed: 15-Dec-2018].