FRI Data Science Project Competition 2020
Final report

Optimizing a search engine for legal documents (Lexpera)

Semir Salkić[1], Darko Ilijevski[1], Lan Vukušič[1], Gašper Hüll[2], Benjamin Hüll[2], Marko Robnik-Šikonja[3], and Slavko Žitnik[3]

Abstract

We propose an approach to improve a legal search engine that modifies a given search query by matching it with existing legal terms. The main goal is to split the search query into terms that are suitable for the specified search engine. This has to be done in a way that all of the found matches are merged into the query. The secondary but an important goal is to prepare the new search query fast (within 200ms).
Our approach consists of two main parts: preprocessing and searching. In the preprocessing part, we use Natural Language Processing (NLP) techniques to clean and prepare the given data for further analysis. The search part finds a match between one search query and the existing legal terms. We suggest two approaches: the first using LSH Forest, and the second using BERT. We evaluate the aforementioned approaches. In the conclusion, we briefly discuss the advantages and disadvantages of our approaches and propose how to improve their results in the future.

Keywords
search engine, legal terms, optimization, natural language processing, NLP, LSH Forest, BERT

[1] *ss9343@student.uni-lj.si, 63190409; di7962@student.uni-lj.si, 63190408; lv8541@student.uni-lj.si, 63190321;*
[2] *Gasper.Hull@lexpera.si; Benjamin.Hull@lexpera.si*; Lexpera d.o.o.
[3] *marko.robnik@fri.uni-lj.si; slavko.zitnik@fri.uni-lj.si;*

Introduction

When users search through legal documents they often look for specific legal terms (e.g. plea bargain, domestic violence). Even though search engines support searching for specific multi-word terms (e.g. by encompassing all the words of a certain legal term in quotes), typical users seldom utilize this feature. Having this in mind, we can utilize some NLP tools to carefully engineer optimization model which can retrieve and filter legal terms in a fast manner. With this approach, we can filter search queries, modify them, and emphasize certain parts. To achieve that, we are splitting our system into two components: preprocessing and searching. In the preprocessing part, we are using normalization, removal of stop words and unwanted characters, and lemmatization. In this way we are reducing data noise and overall complexity which enables us to understand our corpus better. Additionally, we are using spelling correction for the search queries. The searching part is based on advanced search algorithms paired with the usage of hash tables and decision trees. From the moment a user enters a search query, we must return its optimized version within 200 ms.

This report is organized as follows. Section Methods describes the data, preprocessing steps, searching approaches, and evaluation setting. Section Results describes the obtained results from our research. Finally, Section Discussion presents our conclusions and propositions for future work.

Methods

In this section, we are going to describe implemented work: preprocessing and searching part. We will also present our evaluation procedure.

Data

The dataset is made of $143,000$ legal terms and $60,000$ search queries written in Slovene. The legal terms can be complex, meaning one legal term can be composed of multiple nested legal terms. One such raw legal term contains brackets, dashes, slashes, and other punctuation signs. The search queries are user-entered data and as such, they are unstructured and can contain relevant and/or irrelevant strings about the given task.
- legal term e.g.: upravna odločba
- search query e.g.: upravna odločba v elektronski obliki

Preprocessing

As with any text data, the preprocessing is necessary. For the legal terms, we decided to apply normalization, removal of stop words, and lemmatization. The spelling correction step is applied only to a newly entered search query.

Normalization

Search queries may include inconsistent use of uppercase and lowercase letters, punctuation, ambiguous abbreviations, misspelled words, and combinations thereof. One search query can even have words in more than one language. Because of that, some cleaning steps were required. Every search query is first normalized to lowercase and then it is cleaned from quotes, commas, slashes, and brackets.

The normalization process includes more steps for the raw legal terms. Every legal term is normalized to lowercase and cleaned from backslashes, quotes, curly and square brackets, colons, and commas. The additional step for the legal terms is their expansion. If a given legal term contains words in brackets or words divided by dashes, then we expand every such term into several new terms.
Examples:

- ”(ne)plačilo takse”$\rightarrow$”neplačilo takse”, ”plačilo takse”
- ”davki - takse - prispevki”$\rightarrow$”davki”, ”takse”, ”prispevki”

Stop words

Stop word removal is the procedure of removing all of the words which by themselves don’t have any meaning (e.g. for Slovenian: vezniki, členki, etc.). For the implementation purposes, we are using Python 3.7 with NLTK (Natural Language Toolkit). We are using this library for its advantage of having support for the Slovenian language. We can justify this approach when we compare word count of the most frequent words (Table 1). It is easily concluded that data is filtered.

Before		After	
word	count	word	count
za	18907	postopka	3437
v	13688	pravice	3137
u	9807	pogodbe	2726

Table 1. The most frequent words in the legal terms corpus before and after removing the stop-words.

Lemmatization

In many languages, words appear in several inflected forms. The process of grouping the inflected forms of a word is called lemmatization. This process is needed so that all word forms can be analyzed as a single item. For this purpose, we used an implementation of a lemmatizer for the Slovenian language - LemmaGen[1]. For example, the words ”hodim”, ”hodiš”, ”hodi”, ”hodiva”, ”hodita”, ”hodimo”, ”hodite”, and ”hodijo”, are all mapped to their canonical form (lemma) ”hoditi”.

SW removal		Lemmatization	
token	count	token	count
postopka	3437	postopek	7986
pravice	3137	pravica	6835
pogodbe	2726	praven	4364
postopku	2575	pogodba	4360
sodišča	2359	sklep	3286

Table 2. Most frequent tokens after stop-words removal and lemmatization.

The impact of this step can be seen in Table 2. We can see that the tokens ’postopka’ and ’postopku’ merged in their root ’postopek’.

Spelling correction

Since typos in a search query reduce the success rate, we integrated typo correction algorithms. Speed comparisons and accuracy descriptions are presented in Table 3.

The first approach we took is a correction based on *Levenshtein distance* or *edit distance*. This approach judges how fit a dictionary word is as a replacement, based on its Levenshtein distance from the input query. We can correct any input query, regardless of the number of mistakes. The downside of this approach is its speed.

In search of a faster algorithm, we limited ourselves to mistakes within one edit distance. The solution proposed by *Peter Norvig* is generating all possible edits within one edit distance of all legal words (our lookup table). Then we check if any of the possible edits of the given input query is in that lookup table. If we find a match, that means that we have found a possible correction of the string. By utilising hash tables or dictionaries, we can speed up the lookup time to O(c). That approach greatly reduced run time and has given speeds that could be, potentially, used in production. The problem that arises with the given approach is that several words can have the same edits. For example: ”vstava” and ”stava” are both within one edit distance of ”vstave”. To avoid that obstacle, we create a dictionary, where the key is the string ”stava” and the value is a linked list of all words that result in a given correction (”*stava*” : ”*vstava*” $\rightarrow$ ”*stave*”). We can get the most suitable result by using the Levenshtein distance approach on that linked list only. That gives us the ”best of both worlds” approach.

An even faster approach can be achieved by using *SymSpell algorithm*. Its idea is instead of generating all possible edits, (that is: splits, deletes, transposes, replaces, and inserts), generate only all possible character deletes of our legal words, and store them in a dictionary, as before. We create a set of all possible deletions of our input query and check if any of those can be found in our lookup dictionary. The problem of many possible solutions is solved with linked lists, same as mentioned above. We indeed lose the option to correct words that have been accidentally split in half, but compared to the performance increase we gained, we agreed that it is worth it. When higher accuracy is needed, one can just increase the edit distance and maintain virtually the same performance.

All tests were made using a set of 50 random mangled words taken from the legal terms dataset. The set consisted of 10% correctly typed words, 70% words that are within one edit distance, and 20% of words on larger distance.

Edit distance does best accuracy wise since there are no inputs that were miss-corrected. However, this approach doesn't scale well and can be only used on very small corpora. The SymSpell algorithm is the best approach performance-wise. It vastly outperformed the other approaches while not sacrificing much in terms of accuracy. The ability to correct only terms one edit distance away doesn't make this approach unsuitable since the word gets processed further.

Algorithm	Time/word (ms)	Accuracy
Edit distance	4757.65	1.0
Peter Norvig's alg.	52.96	0.89
SymSpell	0.02	0.44

Table 3. Comparison of typo correction algorithms by time and accuracy.

Legal terms detection

After the preprocessing part, we continued to the main part of our task - finding terms from the search query in the corpus of legal terms. For that purpose, we used two different approaches: LSH Forest and BERT.

LSH Forest

Finding k - nearest neighbors is a well-known problem that is used in multiple applications (e.g. similarity search, duplicate detection). This can be formulated as follows: if the set of points S is given in a space V and we define the query point $q \in V$, we need to find the closest point $q' \in S$ to q. A naive approach would include brute force thinking to get results. While this approach probably would yield best results with a clear classification of nearest neighbors, this would be a resource-heavy inefficient algorithm which in practice would be considered useless.

Locality-sensitive hashing (LSH) idea is to hash points into "buckets" in a manner where it is more likely that similar points will share the same bucket, while different points will fall in separate buckets. We can find nearest neighbors of any stored point by simply searching part of the search space, which in this configuration is the bucket in which we have hashed the observed point. Three key steps for enabling this search in the string domain are called: shingling, min-hashing, and locality-sensitive hashing.

Shingling is a process where a string is split in a set of characters of equal length n. (e.g. 3-shingles of "*document*" $\rightarrow$ "doc","ocu","cum","ume","ent"). This approach is based on a theory that similar pieces of text are likely to share number of shingles. Hashing is used to rewrite each independent text to a smaller memory signature using a certain hashing function H. When designing hashing functions, we are using functions which are proportional to the similarity metrics of our choice. For example, min-hashing is used to hash strings where their hash signatures are proportional to the Jaccard similarity coefficient. In other words, the similarity of two hash signatures is equal to the Jaccard coefficient (longer signatures are reducing the estimated error of approximation for the Jaccard coefficient). With this key advantage, min-hashing is substantially reducing space and time complexity, since the Jaccard coefficients require $O(n^2)$ complexity while min hash requires $O(n * k)$ steps, where k refers to the number of hashing functions (also called the number of permutations) - used to define hash signatures of length n. This is achieved by hashing shingles where the minimum hash value of a shingle is used to represent the first hash signature of n. Locality - sensitive hashing refers to the final part - evaluation if two documents are similar, therefore are they forming a candidate pair whose similarity is above a certain threshold t. Proposed in Bawa et. al. [2], LSH Forest structure was introduced.

We utilize the LSH Forest model as a data structure for our case. The dataset is processed and introduced to the model. Each term was shingled into 3-shingles pieces which are hashed with min-hash algorithm. These hashes are further passed to LSH Forest. The number of permutations in our model was 256. The final step was to index (initialize) the forest model. We used the model to search the forest for most similar legal terms based on search queries. The next action was to use the reduced search space of the close neighbors. Here, in this subset, we used cosine similarity on each neighbor to determine the closest (most similar) point.

BERT

Bidirectional Encoder Representations from Transformers (BERT)[3] is an NLP model by Google for pre-training language representations. BERT pre-trains deep bidirectional representations from unlabeled text by joinly conditioning on both left and right contexts in all layers. As a result, the pre-trained BERT model can be fine-tuned with just one additional output layer to create state-of-the-art models for a wide range of tasks, such as question answering and language inference, without substantial task-specific architecture modifications. Pre-training a BERT model is a fairly expensive yet one-time procedure for each language. Fortunately, there is a fair amount of pre-trained BERT models available and can be downloaded for free.

There exist pre-trained BERT models in different languages. Here we use two such models. The first one is the CroSloEngual[4] trilingual BERT model trained on Croatian, Slovenian, and English language. Its vocabulary size is 49,601 tokens. The second is the Multilingual (cased) BERT model[3], which covers 104 languages - including Slovenian. The vocabulary size of this model is 119,547 tokens in all covered languages. Both models are publicly available[1].

[1]CroSloEngual: http://hdl.handle.net/11356/1317
Multilingual: https://github.com/google-research/bert

Here we use the above-mentioned models with the *bert-as-service*[2] service. It uses BERT model as a sentence encoder, i.e. it maps a variable-length sentence to a fixed-length vector.

First, we process the raw legal terms to obtain a hash table in which: the key is the lemmatized legal term; the value is a list of legal terms obtained from the processing of the original legal term. In that list, we store the preprocessed legal term, the legal term without the stop words, and the lemmatized legal term(s) obtained after the normalization of the original legal term. Then, we use *bert-as-service*, together with one of the above-mentioned pre-trained BERT models to encode the hash table keys into vectors. We perform a simple ”fuzzy” search against the existing encoded hash table keys (terms). Every time a new query is coming, we preprocess it, remove the stop words, lemmatize it, and we encode it as a vector. After that, we compute its dot product with the encoded keys, and we sort the result in descending order. We search through the lists of the top 10 keys for a possible match. We can have: an exact match - where we return the whole search query with quotes; a partial match (or more than one partial match) - where we label it (them) with quotes in the original search query; no match - where we return the original query without any labeled parts. The issues and the results obtained by using this approach are discussed in the following sections.

Evaluation setting

To evaluate our approaches, we randomly sampled 200 search queries. Then, we created an attribute that presents the ’true’ optimized query by manually labeling the legal term(s) in each of them. This attribute is used when we compare the results obtained by LSH Forest and BERT. During the manual labeling, we have also marked as legal terms those tokens where the associated legal term was misspelled by one letter. The results obtained by LSH Forest and BERT were categorized into one of the following categories: exact match - (the resulting query is same as the labeled query); partial match - (the resulting query has only a subset of legal terms marked as expected); and no match - (the legal term is not detected). We have also created a simple GUI for easier inspection.

Results

The results obtained with LSH Forest and BERT on the sampled subset of input queries are shown in Table 4. The classification of a particular labeled search query returned by the models in one of the three categories: exact match, partial match, or no match, was done manually. As we can see the BERT approach gives better results for finding an exact match for a given query. Also, BERT has more partially correct labeled queries compared to the LSH Forest approach. The biggest drawback of the BERT approach is the time needed to process a query, although it is still within the time limit. On the other hand, the total average processing time of the LSH Forest algorithm is much smaller. LSH Forest has a problem when the input query is short (one or two words).

[2]https://github.com/hanxiao/bert-as-service

Method	Exact	Partial	No match	Time (ms)
LSH Forest	82	7	111	6.83
BERT	116	20	64	159.29

Table 4. Results on 200 manually labeled queries.

Discussion

LSH Forest and BERT can be used to detect the legal term(s) in a given query. While BERT gives better results, the LSH Forest is much faster. Another advantage of the LSH Forest approach is that it can be iteratively upgraded with new terms. To make a real-world system sustainable we can set a particular time period when we update the models with new terms, or that could be done when the cases of badly labeled queries increase rapidly.

During our work with the search queries dataset, we noticed that many of them are just abbreviations of laws, law articles, version numbers, years, and combinations thereof. They were present in a fair amount in the sampled queries too. Neither of the two methods captures any legal term(s) for that kind of queries. That is because they are partially present or, in most cases, not present even in the raw legal terms dataset. In further work we could expand the current legal terms dataset with those abbreviations since they are important for the legal-document system. Also, we could include faster embeddings such as *fastText* in combination with LSH Forest to boost the algorithm’s results.

LSH Forest and BERT are a great example of a tradeoff between accuracy and speed. For further work, we propose adoption of different embedding techniques paired with similarity measurements (e.g. Word Mover Distance). During our research, we have obtained interesting results in embedding configuration for Doc2Vec and WMD. Unfortunately, without a larger dataset and due to the nature of short queries and terms (e.g. legal terms are 5 words long on average), we were not able to present a stable and consistent model.

References

[1] Matjaz Jursic, Igor Mozetic, Tomaž Erjavec, and Nada Lavrac. LemmaGen: Multilingual Lemmatisation with Induced Ripple-Down Rules. *J. UCS*, 16:1190–1214, 01 2010.

[2] Mayank Bawa, T. Condie, and Prasanna Ganesan. LSH forest: Self-tuning indexes for similarity search. *Proceedings of the 14th International Conference on World Wide Web*, pages 651–660, 01 2005.

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[4] Matej Ulčar and Marko Robnik-Šikonja. FinEst BERT and CroSloEngual BERT: less is more in multilingual models. 2020.