Fast and Flexible Example-Based Treebank Search with Vector Symbolic Architectures

Adam Roussel
Department of Linguistics
Ruhr University Bochum
Bochum, Germany
adam.roussel@rub.de

Abstract

In this paper we introduce an example-based method for exploring dependency treebanks that is based on principles of vector symbolic architectures. It leverages key properties of this framework to provide fast and flexible search capabilities, since all combinations of query parameters can be compared with a given parse tree in parallel via a single vector operation. The framework also allows for graded similarity and the natural integration of various kinds of information, such as word embeddings. After some background on the framework and an explanation of our implementation, we provide a few examples of the system's output and draw comparisons to similar applications.

Keywords: hyperdimensional computing, vector symbolic architectures, treebanks, example-based search

1. Introduction

Treebanks contain a great wealth of data that can be very helpful in the development of linguistic theories. Making effective use of that information can be challenging, in part due to the size of a typical treebank, but largely due to complex structure of the data itself. Thus such datasets are usually queried using specialized query languages that enable one to specify very precisely what structures are sought. This precision is not always an asset. Sometimes it's not clear exactly what constraints are necessary, and, for some phenomena, there is a natural range of variation that can be difficult to specify precisely. In such cases, a robust and flexible approach to treebank exploration can be useful.

In our contribution, we describe an approach to searching dependency treebanks that is example-based and leverages properties inherent to the framework of vector symbolic architectures to make this process both fast and flexible. Our approach is lightweight and can easily be run locally on ordinary computer hardware, and every query, regardless of its apparent complexity, takes the same amount of time. This amount of time is strictly linear with respect to the number of sentences in the treebank. Our approach is also robust and flexible. Rather than users needing to specify upfront which constraints may be relaxed in order to improve recall, our VSA search approach is fuzzy, since all possible combinations of features are queried simultaneously. The system works by measuring the similarity between a query and all of the sentences in the treebank, on which basis it then provides a ranking of all the sentences. The more properties that overlap between the query example and the candidate, the higher its similarity, and thus its ranking, will be.

Beyond these immediate practical advantages, this framework has intriguing potential in other respects. In principle, many different kinds of data can be represented, such that results can be preferred when they are similar to the query with respect to more than just their syntax. For instance, numbers can be represented as similar when they are similar in quantity, and words can be made more similar on the basis of their overall distribution, in both cases introducing semantic factors to influence the search results.

We will begin, in Section 2, with a comparison of our approach with similar efforts from the literature, before offering a brief introduction to vector symbolic architectures in general (Section 3). Then, we describe the specific encoding strategy that we use in our approach in Section 4. Section 5 shows the results for some example queries, as well as some discussion and analysis of these results. Finally, in Section 6, we will discuss some of our plans for extending and improving our approach in future work. All of the software described in this paper is available from https://codeberg.org/aroussel/vsatreebank under a free software license (GPLv3).

2. Related Work

In existing tools for searching dependency treebanks, such as Gärtner et al. (2013) and Luotolahti et al. (2015), queries can be understood as a set of constraints. A candidate is only considered a match if all of the constraints hold. This is often exactly what one wants from a search tool. In other scenarios, one may wish for a query to be somewhat more flexible, or one may not know exactly in what order the constraints should be loosened

Proceedings of the Workshop on Structured Linguistic Data and Evaluation, pages 64–73

in order to retrieve the desired results. A further complication is that search tools, especially those for searching treebanks, often require the use of specialized query languages.

Attempting to address these issues, example-based search tools have been introduced, such as the Linguist's Search Engine (Resnik and Elkiss, 2005) and GrETEL (Augustinus et al., 2012; Odijk et al., 2017), which are more closely related to the strategy we are presenting here. In this style of search tool, the user begins with an example of the construction of interest, which is parsed automatically. The user then has the option to delete uninteresting parts of the parse tree or to stipulate that the lemma must match or that only the POS must match, etc. In this way, constraints are manually relaxed, so that similar but non-identical matches can be found.

In contrast, the approach that we are proposing does not necessarily require any such adjustment of the query, since the comparisons are fuzzy by default. All components of a parse tree are compared simultaneously. Any non-matching elements register as mere noise, and matching elements increase the similarity between the query and a candidate. A nice side-effect of this fuzzy matching is that it provides a workaround for sentences that may have been mis-parsed by the automatic parser, in that such errors won't necessarily lead to relevant candidate sentences being missed. These properties follow naturally from the properties of VSA/HDC representations.

There are further advantages that VSA vector representations have to offer. Karlgren (2023) argue that they offer the promise of using linguistically meaningful and structured representations in contexts that ordinarily demand vectors. Accordingly this application to searching treebanks draws its strengths from these two aspects. Because complex data structures are representable with VSAs, they can be used to represent parse trees, and because vector representations allow for efficient, graded comparisons, those parse trees can be quickly ranked according to their relevance.

Widdows and Cohen (2015) apply techniques of VSAs in order to construct a knowledge base of relational triples, which can be explored through the use of specially constructed query vectors, an approach they call "predication-based semantic indexing" or PSI. Specifically, the authors describe the application of PSI to a biomedical knowledge base, so the predication triples describe relations like "insulin TREATS diabetes mellitus". One can construct vectors that represent questions, like "what treats diabetes?", and then receive in the list of the nearest neighbors some possible answers. Such query vectors can also be constructed on the basis of analogies, as in Kanerva's classic "what is the dollar of Mexico?" example (Kanerva, 2009).

This general approach of constructing a query vector and finding the nearest neighbors in a database is the same basic operating principle of our treebank search approach, as described in this paper. We provide a more detailed description of this procedure in Section 4 below.

3. Background on vector symbolic architectures

Vector symbolic architectures, also known as hyperdimensional computing (HDC), are a computing framework that rely on certain properties of high-dimensional vector spaces, typically 10 000 or more. (For a more detailed introduction, see: Kanerva (2009); Schlegel et al. (2022); Kleyko et al. (2022).) In such high-dimensional spaces, with say 10 000 dimensions, there are exactly 10 000 orthogonal vectors, but there are very many more nearly orthogonal vectors, such that any two randomly chosen vectors are very likely nearly orthogonal. From this it follows that we can readily generate a new vector for any given symbol just by choosing a random vector from this high-dimensional space, and each of these symbolic vectors is distinguishable from the others in that the similarity between them, according to an appropriately chosen metric, is ≈ 0, that is, they are nearly orthogonal. By contrast, correlated vectors, according to context, may correspond to equivalence, similarity, or set membership.

In addition to a vector space, vector symbolic architectures also define a few key operations in this space.[1] The first, termed "bundling" and represented by $+$, takes two vectors and yields a vector that is similar to both of the input vectors. It is also known as "superposition". Then there is a "binding" operation, represented by $\otimes$, that takes two vectors and yields a vector that is dissimilar to both of them. The inverse of the binding operation, "unbinding", is represented by $\oslash$. In logical terms, bundling can be thought of as akin to $\vee$ and binding akin to $\wedge$. There is also an additional operation, permutation, represented as ρ^n, indicating the application of this operation n times. Permutation is a unary operation that produces a vector that is dissimilar to the input vector. With these basic building blocks, various data structures can be implemented, such as key–value sets, sequences, or graphs.

There are various ways that such a vector symbolic architecture can be implemented. In this work, we use a variant that uses dense binary vectors, referred to as binary spatter code or BSC (Kanerva, 1996). In this variant, binding is implemented by

[1] In the following, we will adopt the notational conventions used in Schlegel et al. (2022).

the XOR operation, bundling is element-wise majority rule (i.e., an element is 1 if the sum across n inputs is $> n/2$), and the similarity metric is derived from the Hamming distance. In BSC, the bundling and binding operations are commutative and associative, and binding distributes over bundling. The binding operation is reversible and each vector is its own inverse in BSC, meaning that it is in theory always possible to analyse a vector and see what internal structure it has. I.e., one can unbind the key from a key–value pair to retrieve the value: $B \approx A \oslash (A \otimes B)$.

4. How the system works

4.1. Encoding syntactic dependencies

Each token is encoded as a bundle of key–value pairs. Of the standard UD fields we use `form`, `lemma`, `upos`, and `xpos`, as well as any features included in `feats`. Such a bundle of key–value pairs is thus constructed as follows:

$$v_{\text{form}} \otimes v_{\text{worm}} + v_{\text{upos}} \otimes v_{\text{NOUN}} \dots \quad (1)$$

In order to distinguish between keys and values, we permute the key vector. This is motivated by two things: One is that this acts as a kind of "namespace" for keys vs. values. By permuting v_{form}, we can keep the use of this vector as a key distinct from its use as a value. So when a token's form happens to be "form", this is represented as $\rho^1 v_{\text{form}} \otimes v_{\text{form}}$. This becomes especially relevant in BSC, since in BSC, each vector is its own inverse. The same vector, e.g. for "form", bound to itself will cancel itself out ($v_{\text{form}} \otimes v_{\text{form}} \approx \vec{0}$). The risk is perhaps minor, but to avoid such problems without necessarily needing to store an extra vector for each potential key, we simply permute each vector when it is used as a key.

$$\rho^1 v_{\text{form}} \otimes v_{\text{worm}} + \dots \quad (2)$$

4.2. Encoding strategies for dependency relations

The dependency relations are handled specially. One can think of them either as predicates that apply to entire token bundles, as in (3), in which case the equation corresponds fairly directly to the parse tree. Alternatively, due to distributivity of the operations involved, one can also think of each sequence of dependency relations as encoding a path of keys to a value, as in (4). The degree of permutation applied to each key is significant, and it reflects the order in which a sequence of edges must be followed to arrive at a particular terminal node.

$$\rho^2 v_{\text{nsubj}} \otimes (\rho^1 v_{\text{form}} \otimes v_{\text{worm}} + \rho^1 v_{\text{upos}} \otimes v_{\text{NOUN}} \dots) \quad (3)$$

$$\begin{aligned} &\rho^2 v_{\text{nsubj}} \otimes \rho^1 v_{\text{form}} \otimes v_{\text{worm}} + \\ &\rho^2 v_{\text{nsubj}} \otimes \rho^1 v_{\text{upos}} \otimes v_{\text{NOUN}} + \\ &\dots \end{aligned} \quad (4)$$

Due to the inherent properties of the binding operation, each such path of keys is distinct: A token at the path $\rho^3 v_{\text{nsubj}} \otimes \rho^2 v_{\text{det}}$ isn't similar to one found at $\rho^2 v_{\text{det}}$.

Subtree encoding. In order to facilitate the matching of properties regardless of their depth in the parse tree, the Subtree encoding strategy includes all of the subtrees for each token in the top-level bundle. That is, for an instance of *the* in a subject noun phrase, we would add the feature bundle for this token at all of the following paths:

$$\begin{aligned} &\rho^3 v_{\text{nsubj}} \otimes \rho^2 v_{\text{det}} \otimes \rho^1 v_{\text{form}} \otimes v_{\text{the}} + \\ &\rho^2 v_{\text{det}} \otimes \rho^1 v_{\text{form}} \otimes v_{\text{the}} + \\ &\rho^1 v_{\text{form}} \otimes v_{\text{the}} \dots \end{aligned} \quad (5)$$

Each token is represented by a bundle of such feature paths, corresponding to its specific location and depth in the parse tree. The representation for the whole parse tree is then the bundle of all of the token representations. The following equations give the whole representation (9) for the sentence *The worm reads*, assuming that the tokens only have word form and UPOS annotations:

$$R = (\rho^1 v_{\text{form}} \otimes v_{\text{reads}} + \rho^1 v_{\text{upos}} \otimes v_{\text{VERB}}) \quad (6)$$

$$W = (\rho^1 v_{\text{form}} \otimes v_{\text{worm}} + \rho^1 v_{\text{upos}} \otimes v_{\text{NOUN}}) \quad (7)$$

$$T = (\rho^1 v_{\text{form}} \otimes v_{\text{the}} + \rho^1 v_{\text{upos}} \otimes v_{\text{DET}}) \quad (8)$$

$$\begin{aligned} &R + \rho^2 v_{\text{nsubj}} \otimes W + W + \\ &\rho^3 v_{\text{nsubj}} \otimes \rho^2 v_{\text{det}} \otimes T + \rho^2 v_{\text{det}} \otimes T + T \end{aligned} \quad (9)$$

One potential drawback of this strategy is that it tends to lead to more components being bundled into each sentence. Thus (9) contains 6 components at the top-level. For more complex sentences with many lexical features, the number of components can often exceed 100.

Thin encoding. The Thin encoding strategy, by contrast, doesn't take any special steps to facilitate matching independently of depth. It has the advantage of requiring fewer components in total, only

3 in the case of *The worm reads* instead of 6, as with the Subtree strategy. This could mean faster encoding and less noise in the sentence vectors due to bundling. However, there is the potential disadvantage that a query will only be similar to a sentence if the phrases in the query are embedded at the same depth as in the sentence.

$$R + \rho^2 v_{\text{nsubj}} \otimes W + \rho^3 v_{\text{nsubj}} \otimes \rho^2 v_{\text{det}} \otimes T \quad (10)$$

Flat encoding. The Flat encoding strategy is an alternative way of making queries match more flexibly which requires fewer components than the Subtree strategy. In (11), only 4 components are required. According to this strategy, more deeply embedded elements are expressed by permutation alone, rather than all elements of a path being bound together into a single key. The effect is that, in contrast to the Subtree strategy, a `nsubj` relation two steps away could match, even when the intervening edges don't match, which may or may not be a desirable kind of flexibility.

$$R + \rho^2 v_{\text{nsubj}} \otimes W + \rho^2 v_{\text{det}} \otimes T + \rho^3 v_{\text{nsubj}} \otimes T \quad (11)$$

4.3. Parsing user queries and performing the search

The user provides a query in the form of an example phrase or sentence. The provided query is sent to be processed by the Spacy package (Honnibal et al., 2020). At this point it is important that a parser model is used that provides parses similar to those in the treebank to be searched. In the tests described below, we make use of treebanks annotated according to the Universal Dependencies (UD) standard (de Marneffe et al., 2021), thus it is important that Spacy models used were trained on UD data also.

Once the user query has been parsed, it is then encoded in the same manner as the sentences in the treebank. We then have one vector for the query and one for each sentence in the treebank. To perform the search, the query vector is simply compared to all of the sentence vectors, and the sentences are then ranked on the basis of the resulting similarity values.

This is the scenario that requires the least intervention on the part of the user. Thanks to the automatic parsing, the example sentence can be used as-is. On the other hand, it is also possible to construct the parse tree directly, ensuring that it has precisely the desired structure. This manually constructed parse tree can then be encoded and applied in the same fashion as otherwise.

5. Results

5.1. Resource requirements

The strengths of this approach lie in its simplicity and the minimal computing resources it requires. Yet, despite these properties, it is capable of producing the kind of results that otherwise require some computationally fairly complex indexing and memoization strategies.

The current implementation comes in at just about 400 lines of Python code. All of the tests described below were run on an ordinary, though somewhat aged, consumer-grade laptop from 2016. Most relevantly, since all of the computations described run single-threaded on the CPU, this laptop has an Intel Core i5-6267U with a base frequency of 2.9 GHz. Thus all of the timings given below can be considered fairly conservative, since most users are likely to have newer hardware than this.

The encoding process usually takes a few minutes, depending on the encoding strategy employed. The ca. 65 000 sentences of the Lassy Small corpus (van Noord et al., 2012) are encoded on the test hardware in about 8 min with the Thin strategy, 14 min with the Subtree strategy, and 17 min with the Flat strategy. This is a step that generally only needs to be done once, after which the encoded corpus vectors can be stored and retrieved from disk to be queried freely.

Such a corpus of ca. 65 000 sentences requires, when using bitpacked binary vectors with 10 000 dimensions (thus 1 250 B for each vector), about 200 MB on disk. This includes all of the vectors for each feature key, vocabulary item, etc., that is, each "symbol" in the VSA, as well as all of the vectors for each sentence in the corpus.

As mentioned above, searching the treebank involves a single vector operation for each sentence in the treebank. Both the query and each sentence are represented by a vector, and the vector operation in question measures the similarity between them. In this implementation, based on BSC, this operation is essentially the Hamming distance between the vectors, which itself amounts to XOR, more or less. Since this operation is always the same, each query takes the same amount of time, regardless of its apparent complexity. On the test hardware, a search in the Lassy Small corpus takes about 0.11 s. The amount of time it takes to run such a search is exactly linear with respect to the number of sentences in the treebank.

5.2. Example 1: NP with PP

In order to give an impression of the sort of results that one can get according to the approach we're proposing, we would like to present a comparison with an existing example-based search sys-

tem, GrETEL. For the comparison we will use this example from Augustinus et al. (2012):

(12) Het doden van olifanten is verboden.
'Killing elephants is prohibited.'

If the whole sentence is used to query the treebank as-is, then we already see that the nearest neighbors for this query are quite reasonable matches (Figure 1). They all reflect the basic syntactic structure of the query sentence, and we can also observe that many of these are not only alike in terms of their syntactic relations but also with respect to various morphological properties, such as the noun's being a deverbal noun and the prepositional object being plural. Further down the list of nearest neighbors (not included in the figure), we find sentences that contain the same basic structure but contain a number of other elements or have the words in a different order, illustrating the flexibility of the system:

(13) Voor mannen is het dragen van een korte broek in het openbaar verboden.
'For men, wearing shorts in public is prohibited.'
(WR-P-E-H-0000000049.p.37.s.4)

This sentence is the 10th best match for (12), with a similarity of 0.37.

By contrast, the GrETEL system returns 4 exact matches for a similar level of user interaction, that is, providing only the whole example sentence and accepting the default settings, which require the presence of all nodes in the query. But this is not how GrETEL is meant to be used, and Augustinus et al. (2012) show how an example sentence can be reduced to just the relevant phrase in order to turn up more relevant cases from the treebank. For the reduced query example from that paper, focusing on just the noun phrase *het doden van olifanten*, GrETEL returns 48 matches using the default settings.

The top 5 results from our system for this same phrase are shown in Figure 2. With these results, we can make a few additional observations about the properties of our approach. One is that treebank entries that only consist of matching phrases are preferred. Since there are no additional "noise" elements, the similarity is greater. Secondly, we see that some of the matches have multiple noun phrases that are similar to the query. For such entries the similarity is also greater, since there are more matching structures in the same sentence.

In both of these sets of results, we can see another way that the application we describe here differs from a conventional search engine, which is that our approach results in a ranking of all of the candidate sentences according to all of the available information. In a conventional search approach, there is no ranking, and each instance either matches or it doesn't. Thus while some of the 48 GrETEL matches are among the top 10 from our system (cf. Section 5.4 below), the highly ranked instances can be similar in unexpected and serendipitous ways, e.g. by sharing morphological features that one would not want to use as constraints.

With our VSA-based approach we can include all of the features: form, lemma, POS, number, case, etc., and all of these can be used to find matches that are as similar as possible without ruling anything out. Yet, due to the way they are combined in the high-dimensional vector representations, namely in superposition, all of the combinations of features are compared simultaneously and no decisions are required from users a priori regarding which constraints may or may not be loosened.

5.3. Example 2: Shell nouns

As further example, we consider a hypothetical case in which one is looking for examples of shell nouns (Schmid, 2000). Shell nouns are a class of nouns that serve to encapsulate possibly complex propositional content in order to make it easier to talk about. Shell nouns can typically be found in sentences like *De hoofdzaak is dat je beter wordt* 'the main thing is that you get better', i.e., there is often an abstract noun with a clausal complement.

The top 5 most similar sentences when querying the encoded Lassy Small corpus using this sentence are shown in Figure 3, in this case using the Flat encoding strategy. All 5 of these sentences contain good examples of shell nouns, and of the top 25 nearest neighbors, 17 contain shell nouns.

With the GrETEL search tool, there are no matches for *De hoofdzaak is dat je beter wordt* in Lassy Small when the whole example is submitted. But, using the tool as intended and deleting certain non-essential nodes, namely *je* and *beter* in this case, we then get 127 matches after approximately 2 s to 3 s. These matches appear to generally be good examples of shell nouns: A random sample of 20 all contained clear examples of shell nouns, with nouns such as *feit* 'fact', *voorwaarde* 'condition', or *probleem* 'problem'.

5.4. Quantitative comparison

In order to get a clearer idea of how well this approach works and how it compares to existing methods, we carried out a direct comparison between the exact matches returned by GrETEL and the ranking given by the system we propose here. Specifically, we want to know, for those cases where exact matches are returned by GrETEL for a

0.40 Het nalaten van het identificeren van risicogroepen kan tot ongewenste gezondheidsrisico's leiden.
'The cessation in identification of at-risk groups can lead to unwanted health risks.'
0.39 Het houden van gedomesticeerde dieren heet veeteelt.
'The keeping of domesticated animals is called husbandry.'
0.39 Het stellen van confronterende vragen leidt tot spanningen.
'The posing of confrontational questions leads to tensions.'
0.39 Het erkennen van fouten door de NS-directie is geen alledaags verschijnsel.
'The acknowledgment of errors by the NS management is no everyday occurrence.'
0.38 Het creëren van een winwinsituatie is in ieder geval fundamenteel.
'The creation of a win-win situation is always fundamental.'

Figure 1: Top 5 matches for the example in (12) from the Lassy Small corpus, using the Subtree encoding strategy.

0.48 Het wegnemen van organen
'The removal of organs'
0.43 Het ploegen van het land met behulp van paarden
'The plowing of the land with the use of horses'
0.42 Verzamelen van handelsstatistieken
'Collection of trade statistics'
0.42 Instellen van medicijnen
'Introduction of medications'
0.41 Toedienen van medicijnen
'Administration of medications'

Figure 2: Top 5 matches for the noun phrase *het doden van olifanten* from (12), using the Subtree encoding strategy.

given query, where those sentences appear in the overall ranking that is produced by our approach.

There are a few caveats to be aware of for this comparison. Each example sentence or phrase that is used as a query contains more features than just those that one is necessarily interested in, and these extra features can increase the ranking of some sentences. Furthermore, sentences that contain matching features are considered more similar if those matching features occur more than once, since each time an encoded feature is included in a sentence vector, its signal is amplified and made more prominent. Another challenge for this comparison are the different formats used by the two systems. Whereas GrETEL searches the constituency parses of the Lassy treebank, our system uses the UD conversion of the treebank. While the two should be generally quite close, it could nevertheless be the case that sentences that exactly match the query based on the constituency parse don't match the dependency parse given by Spacy.

While we therefore would not expect that these should match up exactly, we would expect that sentences that do contain the phenomenon of interest, as reflected in its status as an exact match, should at least be ranked higher in general than sentences that do not.

In order to quantify where the system places the exact matches, we use percentile rank. This is for two main reasons. The first is that the similarity values from each encoding strategy aren't directly comparable, since the overall degree of similarity that each encoding strategy results in differs. For the shell noun query, the average similarity for all sentences in the treebank is 0.07 for the Subtree strategy, 0.03 for Flat, and 0.02 for Thin.[2] Since the Subtree encoding strategy results in more components being bundled into each sentence vector, the similarity values tend to be higher, whereas with the Thin strategy there are fewer components to match and lower similarity values. Second, using percentile rank instead of ranks directly makes the numbers easier to interpret, since values vary from 0 to 100, with values close to 100 indicating a high ranking for a match. The results of the comparison are summarized in Table 1.

5.5. Discussion

Comparing encoding strategies. In general, as is apparent in Table 1 and Figure 5, there appears to be a stronger correlation for the shell noun query, in that sentences that match this query exactly tend to be ranked higher by our approach, regardless of the encoding strategy used. For this query, the Flat strategy has a few more outliers than the other two, but the distribution looks more or less the same for all three.

[2] Cf. the overall distribution of similarity values in Figure 6.

0.17 De enige **waarheid** is dat de journalisten logen.
'The only truth is that the journalists lied.'
0.15 Het is de **bedoeling** dat de maatregel afschrikkend werkt.
'It is the goal that the measure works as a deterrent.'
0.14 Het **voordeel** daarvan is dat je de zaken op een onafhankelijke manier kunt bekijken.
'The advantage of that is that you can view things in an independent manner.'
0.14 De **verwachting** is dat de Italiaanse troepen zeker tot 2005 zullen blijven.
'The expectation is that the Italian troops will stay at least until 2005.'
0.14 Maar het is de **vraag** of dat ooit gaat lukken.
'But it is the question whether that will ever work.'

Figure 3: Top 5 matches for the shell noun example *De hoofdzaak is dat je beter wordt* using the Flat encoding strategy. Shell nouns are highlighted in boldface.

Strategy	NP with PP PR	Shell nouns PR
Subtree	91.4 ± 11.3	97.3 ± 3.7
Thin	42.7 ± 27.1	98.3 ± 2.6
Flat	89.3 ± 12.7	96.1 ± 6.2

Table 1: Average percentile rank of exact hits returned by GrETEL for both queries, according to each encoding strategy.

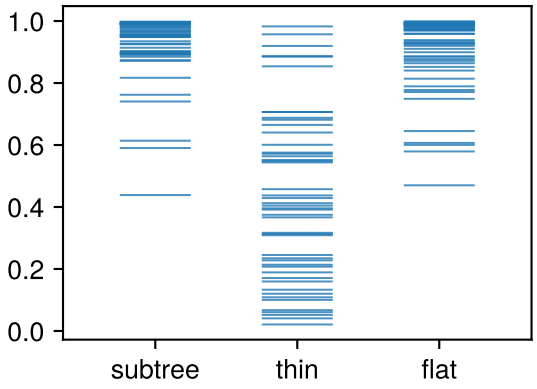

Figure 4: Percentile rank of exact hits for the NP with PP query, according to each encoding strategy.

The results for the first query, shown in Figure 4 are revealing: Here we observe a very wide spread of rankings for the exact matches when the Thin encoding strategy is used, which shows that there is little correlation between the rankings and the exact matches. The contrast is even more stark when we compare with the shell noun query, where no such deficit is evident. The contrast is probably due to the environments where each target pattern is found. Whereas shell nouns with complement

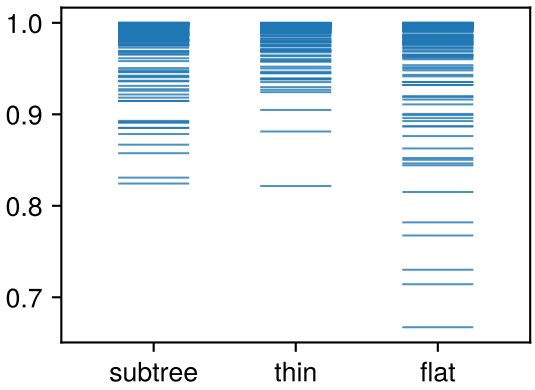

Figure 5: Percentile rank of exact hits for the shell noun query, according to each encoding strategy.

clauses are often or even usually found in matrix sentences, i.e. not embedded, NPs with PPs are likely to occur at a variety of depths in a parse tree. Since the query phrase is not embedded when parsed by Spacy and the Thin encoder only encodes full relation paths, the query and the stored sentence vectors are going to be dissimilar. This case demonstrates the utility of the Subtree and Flat encoding strategies.

Different kinds of data. One of the interesting possibilities that this class of approach offers, besides the computational simplicity and flexibility, is the ability to design representations that are both structured and support graded similarity. Whereas the default assumption that every symbol is different from all the other symbols would be reflected in the use of a new random vector for each symbol, it is also possible to apply various techniques to build in varying relations and degrees of similarity between vectors.

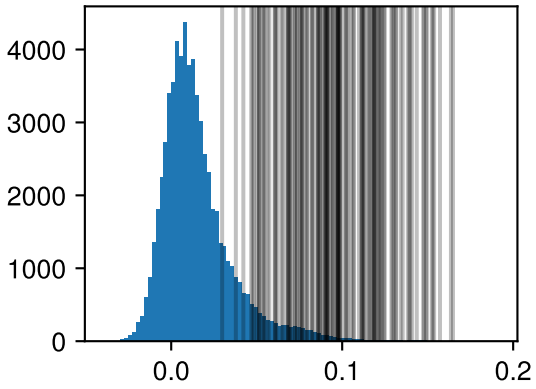

Figure 6: Overall distribution of similarities for the shell nouns query, using the Thin encoding strategy, with the similarity values for the exact hits superimposed as vertical lines.

For instance, in UD there are various subtypes of dependency relations. Thus you find relations like `acl:relcl` 'relative clause modifier', which is a subtype of the broader relation `acl` 'adnominal clause', and `aux:pass` 'passive auxiliary', which is a subtype of `aux` 'auxiliary'. In such cases, we can choose vectors for such subtypes that reflect their relatedness. For this, we use a function that returns a vector, in which some specified percentage of the elements are kept the same as a given input vector and the rest are random as usual. Thus the new vector will be the specified distance from the input vector and approximately distinct from other vectors generated in the same way.

The vectors for particular lexical items can be similarly adapted to evince a degree of similarity that is appropriate to a given application. Whereas a default approach would be to use random vectors for each new lexical item, saying merely that one item is different from another one and nothing more, one might derive lexical representations from various other sources. One could use word vectors derived from the (surface) distribution of words in a given corpus or one could use some kind of sensory data, such as imaging data, as in Eliasmith (2013), or even some combination of these.

The representations could also be designed to reflect properties of the word strings themselves. One way to encode strings is to assign random vectors to character *n*-grams as primitives and to compose them to arrive at the representation for a given string (Joshi et al., 2017). Another method uses "demarcator" vectors (Cohen et al., 2014) that correspond to each of the positions in the string, from left to right. These are each generated to be similar to their neighbors, such that a letter appearing close to the beginning of a string, whether at the second or third position, will have a similar vector representation. Both of these would differ more significantly from a letter bound to the demarcator vector corresponding to the end of a string.

It would also be simple to include information about each token's location in the sentence using this same demarcation vectors technique. Accordingly, tokens in a similar position relative to the beginning or end of a sentence would have greater similarity due to this feature's inclusion.

Limitations and open questions. While the fuzziness in the search results is automatic insofar as it follows directly from the properties of VSAs and it has certain advantageous qualities, e.g. that queries need not be precisely formulated and may consist of an example sentence as-is, there are also some potential disadvantages, specifically with regard to search applications.

In particular, it's not obvious how either certain parts of the query can be made into requirements or, by the same token, how certain results can be ruled out by way of constraints.

Instead of required query components, in Widdows and Cohen (2015) suggest "superposing additional cues" as one method of lending more weight to certain components of a query. So if it is important that the results include the word *fact*, then one would add $\rho^1 v_{\text{form}} \otimes v_{\text{fact}}$ two or three extra times into the query bundle. Then any sentences that include this feature would be ranked higher.

With respect to constraints, since non-matching elements only count as noise and thus have little effect on the similarity of a particular instance, it is also unclear how irrelevant instances can be made less similar so that they appear lower in the results ranking. It may be possible to include something like constraints by negating these irrelevant components (i.e., in BSC, flipping all the bits), $\rho^1 v_{\text{form}} \otimes v_{\text{worm}}^{-1}$ to make instances including the word form *worm* less similar, but this has not been tested yet.

6. Conclusion

In this paper we described an efficient means of making dependency treebanks easily and quickly searchable using techniques derived from vector symbolic architectures. The current application already provides a useful means of finding sentences that are syntactically similar to the provided query, whether the query consists of a complete example sentence or a phrase corresponding to some construction of interest. The implementation is quite simple conceptually, and it requires much less computational resources than comparable existing ap-

plications. Every query amounts to a simple vector operation that is performed exactly once for each sentence in the treebank.

Apart from being lightweight and simple to implement, this is an approach that has some intriguing properties, particularly in its fuzzy matching behavior and its ability to integrate a variety of kinds of data, thus we consider it to be worth further exploration.

As these initial tests have given some promising results, we plan to develop this approach further in future work. In particular, we plan to test the optional integration of word embeddings to represent an additional feature in each token bundle. The idea would be that syntactically similar sentences that also use distributionally similar vocabulary would be ranked higher than sentences that are similar in syntax alone. We also plan to determine whether it would be possible to transfer this approach to other kinds of VSAs, which, among other things, could make it easier to integrate embeddings.

7. Acknowledgments

This research is funded by the Deutsche Forschungsgemeinschaft (DFG) SFB 1475 – project number 441126958. We would like to thank Stefanie Dipper and the anonymous reviewers for their helpful comments and suggestions.

8. Bibliographical References

Liesbeth Augustinus, Vincent Vandeghinste, and Frank Van Eynde. 2012. Example-based treebank querying. In *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC'12)*, pages 3161–3167, Istanbul, Turkey. European Language Resources Association (ELRA).

Trevor Cohen, Dominic Widdows, Manuel Wahle, and Roger Schvaneveldt. 2014. *Orthogonality and Orthography: Introducing Measured Distance into Semantic Space*, pages 34–46. Springer Berlin Heidelberg.

Marie-Catherine de Marneffe, Christopher D. Manning, Joakim Nivre, and Daniel Zeman. 2021. Universal Dependencies. *Computational Linguistics*, pages 1–54.

Chris Eliasmith. 2013. *How to Build a Brain: A Neural Architecture for Biological Cognition*. Oxford University Press.

Markus Gärtner, Gregor Thiele, Wolfgang Seeker, Anders Björkelund, and Jonas Kuhn. 2013. ICARUS – an extensible graphical search tool for dependency treebanks. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 55–60, Sofia, Bulgaria. Association for Computational Linguistics.

Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. 2020. spaCy: Industrial-strength natural language processing in Python.

Aditya Joshi, Johan T. Halseth, and Pentti Kanerva. 2017. *Language Geometry Using Random Indexing*, page 265–274. Springer International Publishing.

Pentti Kanerva. 1996. *Binary spatter-coding of ordered K-tuples*, pages 869–873. Springer Berlin Heidelberg.

Pentti Kanerva. 2009. Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors. 1(2):139–159.

Jussi Karlgren. 2023. High-dimensional vector spaces can accommodate constructional features quite conveniently. In *Proceedings of the First International Workshop on Construction Grammars and NLP (CxGs+NLP, GURT/SyntaxFest 2023)*, pages 31–35, Washington, D.C. Association for Computational Linguistics.

Denis Kleyko, Dmitri A. Rachkovskij, Evgeny Osipov, and Abbas Rahimi. 2022. A survey on hyperdimensional computing aka. vector symbolic architectures, part I: Models and data transformations. *ACM Comput. Surv.*, 55(6):1–40.

Juhani Luotolahti, Jenna Kanerva, Sampo Pyysalo, and Filip Ginter. 2015. SETS: Scalable and efficient tree search in dependency graphs. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Demonstrations*, pages 51–55, Denver, Colorado. Association for Computational Linguistics.

Jan Odijk, Martijn van der Klis, and Sheean Spoel. 2017. Extensions to the GrETEL treebank query application. In *Proceedings of the 16th International Workshop on Treebanks and Linguistic Theories*, pages 46–55, Prague, Czech Republic.

Philip Resnik and Aaron Elkiss. 2005. The Linguist's Search Engine: An overview. In *Proceedings of the ACL Interactive Poster and Demonstration Sessions*, pages 33–36, Ann Arbor, Michigan. Association for Computational Linguistics.

Kenny Schlegel, Peer Neubert, and Peter Protzel. 2022. A comparison of vector symbolic architectures. 55:4523–4555.

Hans-Jörg Schmid. 2000. *English Abstract Nouns as Conceptual Shells: From Corpus to Cognition*. De Gruyter.

Gertjan van Noord, Gosse Bouma, Frank Van Eynde, Daniël de Kok, Jelmer van der Linde, Ineke Schuurman, Erik Tjong Kim Sang, and Vincent Vandeghinste. 2012. *Large Scale Syntactic Annotation of Written Dutch: Lassy*, page 147–164. Springer Berlin Heidelberg.

D. Widdows and T. Cohen. 2015. Reasoning with vectors: A continuous model for fast robust inference. 23(2):141–173.