JEFL: Joint Embedding of Formal Proof Libraries

Qingxiang Wang[1] and Cezary Kaliszyk[1,2][0000−0002−8273−6059]

[1] University of Innsbruck, Austria
[2] University of Warsaw, Poland
`shawn.wangqingxiang@gmail.com`,`cezary.kaliszyk@uibk.ac.at`

Abstract. The heterogeneous nature of the logical foundations used in different interactive proof assistant libraries has rendered discovery of similar mathematical concepts among them difficult. In this paper, we compare a previously proposed algorithm for matching concepts across libraries with our unsupervised embedding approach that can help us retrieve similar concepts. Our approach is based on the `fasttext` implementation of Word2Vec, on top of which a tree traversal module is added to adapt its algorithm to the representation format of our data export pipeline. We compare the explainability, customizability, and online-servability of the approaches and argue that the neural embedding approach has more potential to be integrated into an interactive proof assistant.

Keywords: Unsupervised Embedding · Concept Alignments · Proof Formalization · System Integration.

1 Introduction

One of the challenges hindering massive formalization of mathematics is the heterogeneous nature of the logical frameworks used in various interactive proof assistants [11,27,8,13]. When formalizing proofs against one formal library, it is informative to explore whether and how similar things are done in other libraries. Such exploration has to be done manually and would usually require expertise in the other proof assistants. It would be nice if a tool could let users more systematically explore and discover commonality among formal libraries.

Not only can such a tool be an informative recommender when integrated into an interactive proof assistant, but exploring commonalities among formal libraries is also an interesting problem per se. Through time, multiple versions of the same or similar mathematical concepts have been formalized separately, resulting in repetitive work [21]. To the mathematically oriented, it is quite irksome that identical mathematical concepts must require idiosyncratic formalizations in order to achieve assurance. We believe that by investigating their commonalities, insights on improving interoperability among proof assistants can be obtained, thereby advancing the frontiers of combining systems.

Previous works [4,6] on this problem let us obtain a data export pipeline that could transform data from six proof assistants into a common term representation format (Fig. 1), on top of which an iterative pattern-matching algorithm

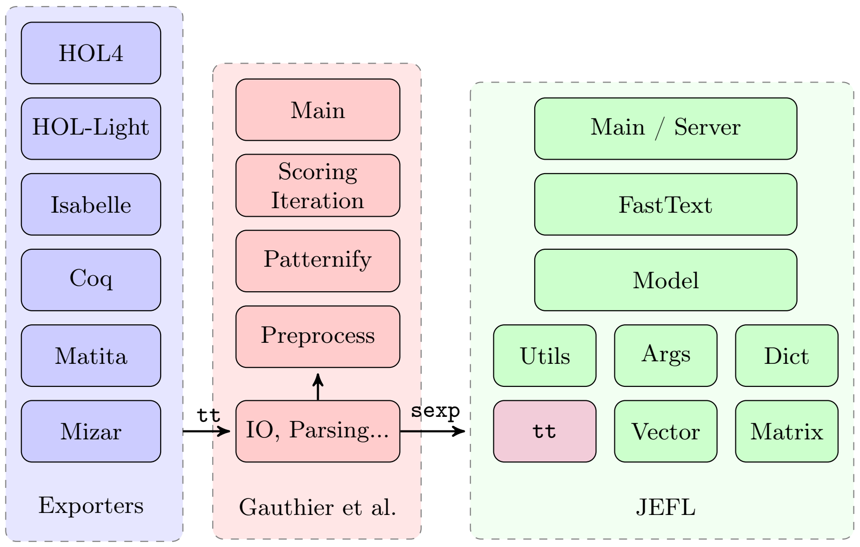


Fig. 1. The architectural relationship between Gauthier's approach and JEFL. At the current stage, the exporters dump text in the `tt` format (Section 2). JEFL reuses the IO/parsing module of Gauthier and passes s-expressions to the `fasttext` implementation.

that could output constant/theorem pairs with high similarity scores was invented by Gauthier. The alignments between the concepts across multiple proof libraries or within one library have been useful for tasks including conjecturing [7], browsing multiple libraries simultaneously [25], and proof automation using learned alignments [5].

The Gauthier approach, while being remarkably effective and useful, lacks *explainability*, *customizability* and *online-servability* that hamper its integration into proof assistants. By these three notions we mean the lack of *mathematical intuitiveness*, lack of *room for customization*, and lack of *possibility for system integration*, respectively. We introduce an alternative embedding approach based on the superb engineering of the `fasttext` implementation [1]. This new approach could potentially overcome these drawbacks while providing competitive performance. It could also serve as a highly configurable experiment platform for studying the alignment of multiple proof assistant libraries. We coin this research *JEFL*, as an acronym for *Joint Embedding of Formal Libraries*.

2 Previous Works and the `tt` Format

Exchanging formal developments within or across formal systems has been studied through three strands of research. First, on the library translation side, many tools that can partially translate proofs have been developed, including those from HOL to Isabelle/HOL [26], HOL Light to Coq [20], HOL Light to Isabelle/HOL [16], respectively. Second, on the ontology sharing side, Bortin [2], Rabe [30], Hurd [14], So and Watt [32], and Carlisle et al. [3] each made their own contribution translating specifications or formal proof objects between formal or semi-formal mathematical representations. These two strands of research mostly either solely provide guidelines on manual processing or require manual work at a certain phase of their framework.

The third strand comes from enhancements of ITP systems. Heras and Komendantskaya [12] implemented a recurrent term clustering algorithm to find proof similarities in Coq/SSReflect libraries. Urban [34] created tools for large-scale retrieval of the Mizar Mathematical Library into a clausal format. Kaliszyk and Urban [17] exported the core HOL Light library as well as the Flyspeck [10] library to evaluate the relevance of lemmas by combining the power of automated theorem provers. This work was later extended to a web service [19] and experimented with using multiple representation formats and different automated theorem provers in [18].

A byproduct along [17,19,18] was a collection of exporting and post-processing techniques specific to HOL Light, including a TPTP-style [33] data representation format which we internally called "the `tt` format". The formalism of `tt` is based on a simple term structure that is flexible enough to represent the kernel representations of formal data on diverse logical foundations, so there is a potential to export data from multiple proof assistants into this common format. Based on the export of three HOL libraries (HOL Light, HOL4, and Isabelle/HOL), Gauthier and Kaliszyk proposed the first version of their scoring algorithm [4] and used for various conjecturing and transfer learning tasks. A more comprehensive set of alignment experiments refined the scoring algorithm, provided a more uniform pattern-matching and guaranteed convergence, and was used on six proof assistant libraries (adding Coq, Matita, and Mizar) [6].

Listing 2.1 and 2.2 illustrate the definition of the predecessor of the naturals (`PRE`) of HOL Light being translated into a list of three `tt` items. The last arguments of them can be parsed into term structures (Fig. 2) using the type definition in Listing 2.3.

Listing 2.1. Definition of the predecessor of the naturals `PRE` in HOL Light.

```
------------------------------------------------------------------------
let PRE = new_recursive_definition num_RECURSION
 `(PRE 0 = 0) /\
  (!n. PRE (SUC n) = n)`;;
------------------------------------------------------------------------
```

Listing 2.2. `PRE` transformed to three `tt` items.

```
------------------------------------------------------------------------
01. tt('const/arith/PRE', ty, ('type/nums/num' > 'type/nums/num')).
02. tt('thm/arith/PRE_0', ax,
    (('const/arith/PRE' ('const/nums/NUMERAL' 'const/nums/_0')) =
      ('const/nums/NUMERAL' 'const/nums/_0'))).
03. tt('thm/arith/PRE_1', ax,
    (![n : 'type/nums/num']:
      (('const/arith/PRE' ('const/nums/SUC' n)) = n))).
------------------------------------------------------------------------
```

Listing 2.3. Type definition of `tt` term in OCaml for parsing.

```
---------------------------------------------------------------------------
type ttterm =
| Id of string (* may be a constant or variable *)
| Comb of ttterm * ttterm
| Abs of string * ttterm * ttterm;;
---------------------------------------------------------------------------
```

The HOL Light and HOL4 exports directly use HOLyHammer's export [18]. For Isabelle, an ML component was implemented that extracts all theorems of the theory and writes them together with the declared constants and types in a text file. The Coq export to the `tt` format was implemented by Gauthier as part of his work [6]. For Mizar, we rely on Urban's MPTP pipeline [35] and transform the intermediate XML2 representation.

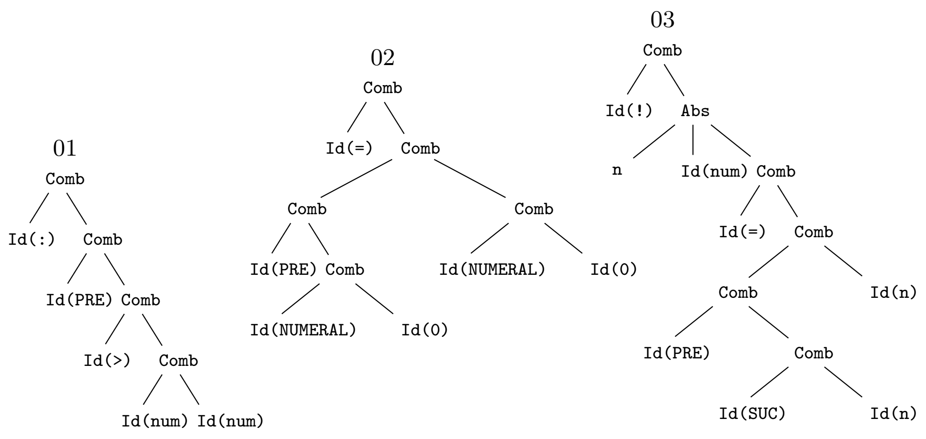


Fig. 2. Term structures of the definition of `PRE`. The tokens `PRE`, `num`, `NUMERAL` and `SUC` are short for `const/arith/PRE`, `type/nums/num`, `const/nums/NUMERAL` and `const/nums/SUC`, respectively. Note that the constant `const/arith/PRE` is included into the term *01* with a type assignment operator `':'`. This allows embedding vectors to be assigned to the definition constants.

3 The Architecture of JEFL

In this paper, we focus on the core similarity discovery algorithm. Our claim on advantages in JEFL is with respect to the algorithmic part of the system. We leave the eventual integration of the whole framework into proof assistants, with issues such as handling constants that have not been encountered during training, as future work.

3.1 Similarity through Embedding

A natural way to find similarities among concepts is to treat our problem as a distributed representation learning task. Generally speaking, given a structure composed of atomic units, distributed representation learning seeks to represent each of the atomic units with a low-dimensional vector. In effect, all the units are embedded into a Euclidean space, with their coordinates respecting the overall structure. The notion *distributedness* comes from the fact that the vocabulary size of a corpus is much larger than the dimension of a vector, and the information of an atomic unit is distributed in the coordinates of a vector.

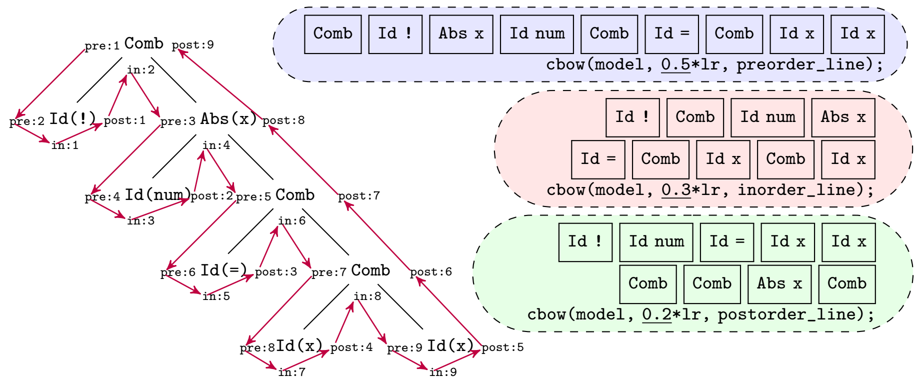
The vectors are learned by analyzing the *context* of each unit, i.e. the information of units adjacent to or surrounding a target unit. Once vector representations for units are learned, similarity between units can then be computed by cosine similarity with a range from $[-1, 1]$. For a set of units, vector representation can be computed by taking average of the vectors, and then similarity between different sets of units can also be computed using cosine similarity.

Notable unsupervised distributed representation learning algorithms include Pennington et al.'s GloVe algorithm [28] and Mikolov et al.'s Word2Vec algorithm [23,24]. In this paper, we use Mikolov's Word2Vec algorithm. Word2Vec works on texts or lists of word tokens. For each word in the training corpus, a randomized span of words surrounding that word is picked to form the context of that word. The context is then consumed by the Word2Vec model to conduct one step of the stochastic gradient descent updates.

3.2 Adaptation of the `tt` Format in Word2Vec

To illuminate our technique, it is interesting to note that DeepWalk [29] and Node2Vec [9], two methods on embedding large networks, also use Word2Vec as their underlying algorithm. The data used by DeepWalk and Node2Vec are single-graph datasets with nodes that contain heterogenous information such as social profile details. To fit Word2Vec, first the node information of the graph has to be transformed into a dictionary through data processing. Then we perform random walks along the paths of the graph to generate node sequences that resemble text corpus. For each node in a node sequence, the corresponding context is generated as a span of nodes surrounding that node.

In our case, different from DeepWalk and Node2Vec, the formal library data in the `tt` format are not a single graph but a collection of trees. More precisely, in order to compare two libraries, we need two lists of `tt` items from the two libraries to provide as training data. The `tt` items are parsed as trees and then traversed in different ways to create sequences of node constants. Examples of traversals include preorder, inorder, postorder traversals and their reverses, random walks from the root to a leaf, or just dump the leaves of a tree in some order. With clever design, these traversals can also be combined to create hybrid orders. At the current phase we implemented a simple weighted mechanism combining preorder, inorder and postorder traversals. The weights of traversals are used to control the learning rate for SGD updates and are hyperparameters determined before training (Fig. 3). We anticipate further experimental insights when other forms of traversals are implemented in the future.



```
cbow(model, 0.5*lr, preorder_line);
cbow(model, 0.3*lr, inorder_line);
cbow(model, 0.2*lr, postorder_line);
```

Fig. 3. Preorder, inorder, and postorder traversals of a simple theorem $\forall x$: num. $(x = x)$, with weights 0.5, 0.3, 0.2, respectively. Example illustrated by calling the CBOW method of `fasttext`, where `lr` is the learning rate and the third argument contains the token sequence above it. Inside the CBOW method, for each token, a randomized span of words surrounding that token is obtained to compute the hidden vector.

Both DeepWalk and Node2Vec directly use the Word2Vec implementation of the Gensim [31] topic modeling library. For ease of future integration into proof assistants we pick a dedicated Word2Vec implementation `fasttext` [15] as our base platform. To make our customization less intrusive we add a custom tree traversal module to the codebase of `fasttext`, also called the `tt` module (Fig. 1). The `tt` module parses the terms in the `tt` format and builds corresponding trees in the memory of JEFL.

3.3 The SGD Updates of Word2Vec

It remains to discuss the core Word2Vec algorithm, which is divided into two aspects: 1. what is the probability model for Word2Vec training and 2. how the loss function is computed. In the former, there are the *continuous bag-of-words* model (CBOW) and the *skip-gram* model. They appear at the step of the training loop outside stochastic gradient descent (SGD) updates and determine how data samples are used. In the latter, there are the *softmax* loss, the *hierarchical softmax* loss, and the *negative sampling* loss. They compute the loss function, at the same time determine the gradients and update the input and output matrices. The two training models are compatible with the three loss functions, so there are in total six variations of the Word2Vec algorithm[3].

As the full Word2Vec algorithm is extensive, we briefly describe the difference between skip-gram and CBOW using the simplest softmax case. We skip detailed derivations and remind the reader of the abundance of study materials of Word2Vec on the internet[4].

Let $\mathcal{C}$ be the training corpus, V be the size of the dictionary of $\mathcal{C}$, and D be the dimension of a word vector. Denote $M \in \mathbb{R}^{V\times D}$ as the *input matrix* which we use to store all the word vectors. Denote $N \in \mathbb{R}^{V\times D}$ as the *output matrix* which we use to store customized data items depending on the loss function. Let $w \in \{1, 2, \dots, V\}$ be a word, or more precisely, the index of an actual word in the dictionary. We denote M_w as the w-th row of the input matrix M. Similarly we denote N_u as the u-th row of the output matrix N, given a word $u \in \{1, 2, \dots, V\}$. Both M_w and N_u are D-dimensional row vectors. For each word w, denote context(w) as a randomized span of words surrounding w. Let $\eta > 0$ be the learning rate.

From the probability modeling point of view, CBOW amounts to maximizing the log-likelihood of the form

$$\mathcal{L} = \log \prod_{w\in\mathcal{C}} P\left(w|\text{context}(w)\right) = \sum_{w\in\mathcal{C}} \log \operatorname{softmax}(Nh^T)_w,$$

where

$$h = \frac{1}{|\text{context}(w)|} \sum_{u\in\text{context}(w)} M_u$$

is the hidden vector. The SGD updates are computed by taking *increments* of the gradients of the objective (as we want to *maximize* the log-likelihood) [5]

$$\begin{aligned} N_u &:= N_u + \eta\left(\delta_{uw} - \operatorname{softmax}(Nh^T)_u\right) h && \text{for } u \in \{1,\dots,V\} \\ M_u &:= M_u + \frac{\eta}{|\text{context}(w)|} \sum_{v=1}^{V} \left(\delta_{vw} - \operatorname{softmax}(Nh^T)_v\right) N_v && \text{for } u \in \text{context}(w) \end{aligned}$$

The skip-gram model amounts to maximizing the log-likelihood of the following form

$$\begin{aligned} \mathcal{L} &= \log \prod_{w\in\mathcal{C}} P\left(\text{context}(w)|w\right) = \log \prod_{w\in\mathcal{C}} \prod_{u\in\text{context}(w)} p(u|v) \\ &= \sum_{w\in\mathcal{C}} \sum_{u\in\text{context}(w)} \log \operatorname{softmax}(Nh^T)_u \end{aligned}$$

[3] As to writing of this paper, one more loss function (the `one-vs-all`, or the `ova` loss) has been added to the latest version of `fasttext`, making in total eight variations.

[4] The first author finds this note https://github.com/renpengcheng-github/nlp/tree/master/3.word2vec (in Chinese) particularly helpful in understanding Word2Vec.

[5] We use the term stochastic gradient *descent* here for convention though we are in fact doing stochastic gradient *ascent*.

where

$$h = M_w$$

is a D-dimensional row vector. For each $u \in \text{context}(w)$, the SGD updates are

$$N_{\widetilde{w}} := N_{\widetilde{w}} + \eta \left(\delta_{\widetilde{w}u} - \text{softmax}(Nh^T)_{\widetilde{w}}\right) M_w \qquad \text{for } \widetilde{w} \in \{1, \dots, V\}$$

$$M_w := M_w + \eta \sum_{v=1}^{V} \left(\delta_{vu} - \text{softmax}(Nh^T)_v\right) N_v.$$

Algorithm 1 Full algorithm for CBOW and skip-gram with softmax loss

```
 1: for $w \in \mathcal{C}$ do
 2:     Get sample $(w, \text{context}(w))$.                ▷ See Section 3.2
 3:     if CBOW then
 4:         $h := \mathbf{0}$
 5:         for $v \in \text{context}(w)$ do
 6:             $h := h + M_v$
 7:         end for
 8:         $h := h/|\text{context}(w)|$                  ▷ 1. Get hidden vector (cbow)
 9:         $g := \mathbf{0}$
10:         for $u \in \{1, \dots, V\}$ do                 ▷ Room for speedup
11:             $s := \text{softmax}\left(Nh^T\right)_u$
12:             $\alpha := \eta\,(\delta_{uw} - s)$
13:             $g := g + \alpha N_u$                       ▷ 2. Accumulate gradient (cbow)
14:             $N_u := N_u + \alpha h$                     ▷ 3. Update output matrix (cbow)
15:         end for
16:         $g := g/|\text{context}(w)|$
17:         for $u \in \text{context}(w)$ do
18:             $M_u := M_u + g$                            ▷ 4. Update input rows (cbow)
19:         end for
20:     else                                                ▷ Skip-gram
21:         $h := M_w$                                      ▷ 1. Get hidden vector (skipgram)
22:         for $u \in \text{context}(w)$ do
23:             $g := \mathbf{0}$
24:             for $\widetilde{w} \in \{1, \dots, V\}$ do     ▷ Room for speedup
25:                 $s := \text{softmax}\left(Nh^T\right)_{\widetilde{w}}$
26:                 $\alpha := \eta\,(\delta_{\widetilde{w}u} - s)$
27:                 $g := g + \alpha N_{\widetilde{w}}$            ▷ 2. Accumulate gradient (skipgram)
28:                 $N_{\widetilde{w}} := N_{\widetilde{w}} + \alpha h$  ▷ 3. Update output matrix (skipgram)
29:             end for
30:             $M_w := M_w + g$                            ▷ 4. Update input rows (skipgram)
31:         end for
32:     end if
33: end for
```

3.4 The `fasttext` Implementation of Word2Vec

The full SGD update algorithm is shown in Algorithm 1. Notice that, for both CBOW and skip-gram, in each round of model updates there are essentially four identical steps: 1. obtain hidden vector, 2. accumulate gradient, 3. update rows of the output matrix, and 4. update rows of the input matrix. This four-step abstraction is general not only for softmax but also for hierarchical softmax and negative sampling, which are specifically designed to speed up the calculation of the inner loop in line 10, 24 of Algorithm 1.

The architecture of `fasttext` was inspired by this four-step abstraction. Since its initial development in 2016, lots of advanced functionalities have been added on top of the Word2Vec algorithm, including model quantization, auto-tuning, python binding, etc. This makes the codebase large and many of those functionalities are irrelevant to our research. Therefore we use an earlier commit in late 2016 as our base[6]. In this commit, all six variations of Word2Vec have been implemented and very few advanced functionalities are added. Two of them worth mentioning are: 1. subsampling of most frequent words, and 2. subword information enrichment trick. The first is an extension of the Word2Vec algorithm in [24] to filter out disproportionally frequent words in the training corpus. This function is disabled since it is obvious in our dataset that the most frequent tokens are always `Comb`, `Id`, and `Abs`, respectively, and they have to be included to allow for correct parsing of `tt` items. The second is a feature in the `fasttext` implementation [15] which breaks a word token into segments of character-level n-gram tokens. This is also disabled since constants in our embedding task (e.g. `'const/arith/PRE'`) constitute a unique and separate entity, and enabling this feature would normally increase the size of a training model by more than a hundred times. The original `src` directory of this commit contains 2054 lines of C++ code written in C++11.

4 Experimenting with JEFL

The core algorithm part of JEFL consists of 8 modules of the original `fasttext` plus a custom module for term parsing and traversal (Fig. 1 right). We find that the best way for customization is to add to the `args` module a new flag (`isTT`) to denote whether our training corpus is a list of `tt` items or plain texts. The normal process flow is not interrupted if this flag is `false`, so JEFL can also train on plain text. If `isTT` is `true`, then subsampling is suppressed when reading in the input files. This allows all tokens of `tt` terms to be read so that term parsing can be done correctly. The `tt` items are read, parsed and the parsed terms can be reconstructed as trees in the C++ side. Helper functions are then called to traverse a tree in different orders, look up the index values of its constants from a dictionary, and call `fasttext`'s original CBOW or skip-gram method for SGD updates (Fig. 3).

[6] `c62abb89396a94520f009f9095874953735e0d75`

We report our initial round of experiments with this platform and two formal proof libraries, HOL4 and HOL Light, testing the performance of different hyper-parameter combinations. There are in total 18723 and 16874 lines of `tt` items in HOL4 and HOL Light, respectively. We concatenate and shuffle the exported theorems from the two libraries, and then write them out as s-expressions [22].[7] We evaluate the performance by comparing against the 1000 highest-scoring constant pairs, that have been manually checked by Gauthier in his work, and considered here as a baseline.

	Top-1 Hit	Top-3 Hit	Top-10 Hit	Top-20 Hit
Tree-Dump	51	101	188	261
Leaf-Dump	21	61	96	144

Table 1. Comparison of theorem export formats. Tree-dump exports the whole tree representation in the given order, while leaf-dump exports only the sequence of data present in the leafs.

	Top-1 Hit	Top-3 Hit	Top-10 Hit	Top-20 Hit
Skip-Gram	54	108	264	283
CBOW	51	101	188	261

Table 2. Comparison of models skip-gram and continuous bag of words.

	Top-1 Hit	Top-3 Hit	Top-10 Hit	Top-20 Hit
Hierarchical Softmax	78	161	304	419
Negative Sampling	51	101	188	261

Table 3. Comparison of sampling hierarchical softmax vs. negative sampling

We present four sets of experiments for the initial comparison. We measure JEFL's performance against the Gauthier baseline by using the "Top-N Hit" metric, which means the inclusion of the correct answer from the closest N neighbors of the target constant. By default, we use leaf-dump (sequences of data present in the leafs), CBOW, negative sampling, and equal (0.33,0.33,0.33) weights. For other key parameters of `fasttext`, we set vector dimension as 100, learning rate 0.05, random uniform context window size 1 to 10, training epoch 5 (for most experiments we see little training progress after epoch 5, so for a

[7] This gives a total of 35597 s-expressions for Word2Vec training.

(pre-,in-,post-order)	Top-1 Hit	Top-3 Hit	Top-10 Hit	Top-20 Hit
(0.33,0.33,0.33)	51	101	188	261
(0.5,0.3,0.2)	58	103	205	267
(1,0,0)	53	110	204	256
(0.5,0.5,0)	49	113	207	276
(0,0.5,0.5)	57	106	203	268

Table 4. Combination of the effect of weights given to the different traversals. The table shows the weights given to pre-order, in-order, and post-order respectively, together with their effects on finding same constants across libraries.

fair evaluation we stick with 5 epochs for all evaluations), 5 negative samples for negative sampling loss, and 4 training threads.

Experiment 1 (Table 1) tests the difference between tree-dump (dumping s-expression) vs. leaf-dump (dumping leaves as token sequences). This experiment is the first one to test that our customization blends with the normal process flow of `fasttext`. We see that tree traversal gives better hit rates than just use the leaves as the former uses more information in training.

Experiment 2 (Table 2) tests the difference between skip-gram and CBOW. We see that skip-gram performs better than CBOW in all hit rates. However, as noted in Section 3.3, skip-gram takes longer time to train (this depends on the size of the contexts, and in our experiments skip-gram takes about five times longer). For ease of experiment, we fall back to use CBOW as our default.

Experiment 3 (Table 3) shows a clear advantage of hierarchical softmax over negative sampling. We see a 60% increase in all the hit rates. We speculate that this improvement is due to the fact, that the Huffman tree computation in hierarchical softmax might put an advantage in mining patterns in tree-like data structures.

Experiment 4 (Table 4) explores different combinations of weights in tree-traversal. They all outperform leaf-dump, however, none of the combinations performs significantly better than others. We plan to explore other forms of traversal such as random walks to see further results.

5 Comparison with Iterative Pattern-Matching

In this section, we shortly recall the iterative pattern-matching algorithm developed by Gauthier and Kaliszyk [6], and compare it with the work presented here. The iterative pattern matching algorithm is based on the observation that once mathematical information in different formal libraries is represented in the same `tt` format, similar theorems or typing judgements (as terms of `tt`) tend to have identical term structures. Accordingly, similar constants (as leaves of terms) tend to locate in corresponding slots of a term (Fig. 4). To abstract out common term structure, Gauthier invented the notion *pattern of a term*. The pattern of a term T is created by abstracting out, in a canonical order, all the T's non-logical constants. Two terms T_1 and T_2 sharing the same pattern form a *matching pair of terms*. Corresponding slots of a matching pair of terms *induce* a collection of *matching pairs of constants*.

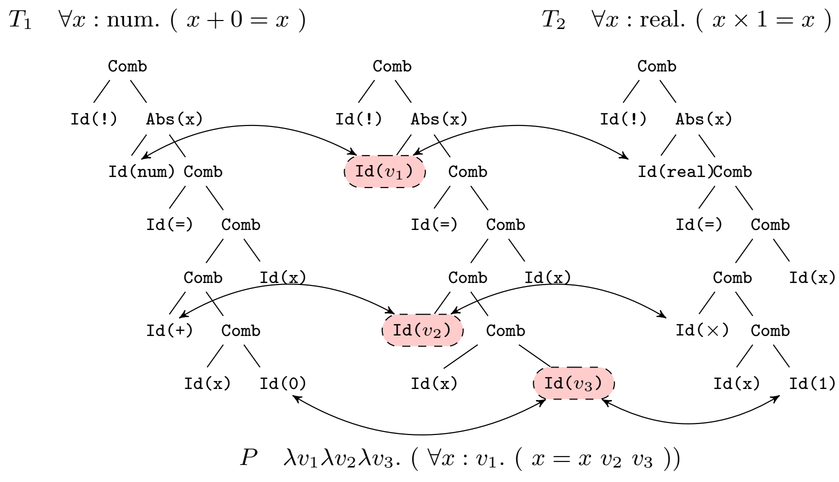

T_1 $\forall x : \text{num.} (x + 0 = x)$
T_2 $\forall x : \text{real.} (x \times 1 = x)$
Comb
Id(!)
Abs(x)
Id(num)
Id(real)
Id(v_1)
Id(=)
Id(x)
Id(+)
Id(v_2)
Id(×)
Id(0)
Id(v_3)
Id(1)
P $\lambda v_1 \lambda v_2 \lambda v_3. (\forall x : v_1. (x = x \; v_2 \; v_3))$

Fig. 4. T_1 and T_2 form a matching pair of terms with pattern P. Three matching pairs of constants can be induced from this pattern. We treat equality = and universal quantification ! as logical constants. Bound variables are assumed to be normalized.

Given two formal libraries L_1 and L_2. Let $\{t_i\}_{1\leq i\leq m}$ be the collection of all matching pairs of terms, with $t_i = (t_{i1}, t_{i2})$, $t_{i1} \in L_1$, and $t_{i2} \in L_2$. Let $\{c_j\}_{1\leq j\leq n}$ be the collection of all matching pairs of constants, with $c_j = (c_{j1}, c_{j2})$. Let $g(x) = x/(x+1) : \mathbb{R}^+ \to [0,1)$ be a strictly increasing *normalization function*. Define an *indicator function* $\delta(c_j, t_i)$ and set $\delta(c_j, t_i) = 1$ if constant pair c_j can be induced by term pair t_i and 0 otherwise. Similarity scores between pairs of terms and pairs of constants can be calculated using the following recurrence relations

$$\begin{cases} \text{score}_c^0\,(c_j) = 1, & j = 1, \ldots, n. \\ \text{score}_t^T\,(t_i) = w_t\,(t_i) \sum_{l=1}^{n} \delta(c_l, t_i)\, \text{score}_c^{T-1}(c_l), & i = 1, \ldots, m. \\ \text{score}_t^T\,(c_j) = g\left(w_c\,(c_j) \sum_{k=1}^{m} \delta(c_j, t_k)\, \text{score}_t^T(t_k)\right), & j = 1, \ldots, n. \end{cases} \tag{1}$$

where $T = 0, 1, 2, \ldots$ is the iteration step and the weighting functions for terms $w_t\,(t_i)$ and constants $w_c\,(c_j)$ are determined using heuristics

$$\begin{cases} w_t\,(t_i) = \frac{1}{\ln(2+p(t_i))} \frac{1}{\ln(2+q(t_i))}, & i = 1, \ldots, m. \\ w_c\,(c_j) = \frac{1}{\ln(2+r(c_{j1}) \times r(c_{j2}))}, & j = 1, \ldots, n. \\ p\,(t_i) = \#\{\text{term pairs sharing the same pattern as } t_i\}, & i = 1, \ldots, m. \\ q\,(t_i) = \#\{\text{constant pairs induced by } t_i\}, & i = 1, \ldots, m. \\ r\,(c_{jd}) = \#\{\text{terms containing } c_{jd}\}, \quad d = 1 \text{ or } 2, & j = 1, \ldots, n. \end{cases} \tag{2}$$

By rewriting equation (1) with respect to $\text{score}_t^T(c_j)$ and using properties of g, δ, w_t, and w_c, Gauthier proved the convergence of this scoring algorithm (using monotone convergence theorem coordinate-wise in $[0,1]^n$) [6].

5.1 Advantages and Drawbacks of Iterative Pattern Matching

Gauthier's iterative pattern-matching algorithm is cleverly designed. Intuitively, the existence of a pattern already indicates a strong correlation among term pairs, and the existence of constant pairs at corresponding slots of a pattern already indicates a strong correlation among those constant pairs; The indicator function δ transports "similarity awards" among those pairs, while the weighting functions w_t and w_c penalize frequently occurring patterns. Above all, the normalization function g ensures the validity of the scores and is crucial for convergence of the algorithm. All these components are intricately combined to make the whole algorithm effective in discovering identical or similar mathematical concepts.

Nevertheless, Gauthier's algorithm possesses some inherent drawbacks. From the *explainability* angle, the balance between the heuristics (w_t, w_c and their components p, q, and r) in equation (2) and the score accumulation terms ($\sum$ and δ) in equation (1) is, to our mind, hard to explain clearly and difficult to readjust. The convergence of the algorithm is mostly due to the property of the normalization function g but the link between this convergence and how similarity scores are sorted is weak. The algorithm works on spaces of similarity scores between matching pairs of terms and constants, so from its beginning, the information on non-matching terms and constants is thrown away, losing the possibility to look at the alignment of different proof assistant libraries holistically.

Comparatively, our approach provides much more flexibility and intuitiveness. Using distributed representation learning, all the constants are movable points in Euclidean space and similarity between constants are naturally described as cosine similarity between their coordinates. By using an embedding approach, not only the "matching" pairs, but also similarity between all pairs of constants can be retrieved and their computation is cheap.

From the *customizability* angle, the components of Gauthier's algorithm are so intricately combined that there seems little opportunity to adjust the algorithm further. In the implementations of both [4] and [6], most of the extra work is on preprocessing terms using combinations of rewriting rules to create a varying set of patterns. These rewriting rules include, *e.g.* rewriting to conjunctive normal forms, reordering commutative/associative connectives, substituting subterms with definitions, as well as exposing various levels of typing information. All these customizations are only allowed in the preprocessing phase and limited to only employing rewriting rules. Some of the typing exposure rules require specific knowledge of representing a library in the `tt` format. As these rewriting-based customizations have already been thoroughly investigated in [6], it seems to us that further investigation along this line is destined to a diminishing return.

Comparatively, customization of JEFL can be done at different phases of the full training algorithm, such as the data generation phase, term traversal phase, model update phase, etc. This multi-level customization can create combinatorially much more room for parameter-tuning and experimentation. Moreover, except data generation, all other customizations that are within the algorithm can be more uniformly done. Specifically, data fields and command parsing can be added to the `args` module, and then used at desired places of the data flow.

From the *online-servability* angle, despite the fact that Gauthier's algorithm is overall fast and effective on small data, it is still quadratic on the size of the input libraries, since it needs to enumerate all pairs of terms to find patterns. Being a batch program without a separated and instantaneous evaluation phase, it is not tempting to integrate the whole algorithm into an actual interactive proof assistant. Moreover, even if it were to be used as an online recommender, the only information we can retrieve would be limited to only the matching pairs.

In JEFL, despite more computations are involved, training is linear with respect to the size of the corpus. This makes JEFL more suitable for obtaining similarity measurements on a large training corpus. JEFL provides a clear separation between a training phase and an evaluation phase. The evaluation phase is instant once the model is loaded. This allows JEFL to have more potential to be integrated as a service. In the version of the `fasttext` commit used in JEFL, if subword information is disabled, the size of most of the models dumped after training are less than 5MB. This is a negligible size comparing to the size of a modern proof assistant.

6 Conclusion

In this paper, we identify the need for commonality discovery among formal libraries. We introduce our data pipeline, especially its preceding works, and elaborate our internal `tt` formalism. Methodologically, we describe the architecture of JEFL and make a series of first experiments to test the efficacy of our experiment platform and provide a high-level comparative analysis with the iterative pattern matching algorithm.

6.1 Limitations and Future Work

There are a lot of future possibilities in our JEFL platform. Continuing in the current line of development, we still need to experiment on the other four libraries and additionally explore similarity discovery of not only constants but also terms. We could also explore the effect of vector initialization in our discovery algorithm. To go deeper we could implement custom "dragging" and "repelling" steps using geometric manipulation and intersperse these custom steps with SGD updates. We have focused on one pair of libraries, which could be extended to an embedding of multiple libraries combined. This would provide further experiment opportunities. We also plan to use the newly discovered samples from JEFL to do tasks such as conjecturing [7], cross-browsing [25], and stronger learning for hammers [5]. Last but not least, we hope there could be use cases to integrate our pipeline into an actual proof assistant and see improved formalization productivity.

Acknowledgements

We are largely indebted to Thibault Gauthier for his work on alignments and the various data exports that we re-use. We thank Josef Urban for the Mizar export and his invitation to Prague to discuss research. We also thank Tomáš Mikolov for valuable insights for the current work. This work was supported by the ERC grant no. 714034 *SMART* and by the *University of Innsbruck PhD scholarship*.

References

1. Bojanowski, P., Grave, E., Joulin, A., Mikolov, T.: Enriching word vectors with subword information. Transactions of the Association for Computational Linguistics **5**, 135–146 (2017), `https://aclanthology.org/Q17-1010`
2. Bortin, M., Lüth, C.: Structured formal development with quotient types in isabelle/hol. In: Autexier, S., Calmet, J., Delahaye, D., Ion, P.D.F., Rideau, L., Rioboo, R., Sexton, A.P. (eds.) Intelligent Computer Mathematics. pp. 34–48. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
3. Carlisle, D., Davenport, J., Dewar, M., Hur, N., Naylor, W.: Conversion between mathml and openmath. Technical Report 24.969, The OpenMath Society (2001)
4. Gauthier, T., Kaliszyk, C.: Matching concepts across HOL libraries. In: Watt, S., Davenport, J., Sexton, A., Sojka, P., Urban, J. (eds.) Proc. of the 7th Conference on Intelligent Computer Mathematics (CICM'14). LNCS, vol. 8543, pp. 267–281. Springer Verlag (2014). https://doi.org/10.1007/978-3-319-08434-3_20
5. Gauthier, T., Kaliszyk, C.: Sharing HOL4 and HOL Light proof knowledge. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) 20th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2015). LNCS, vol. 9450, pp. 372–386. Springer (2015). https://doi.org/10.1007/978-3-662-48899-7_26
6. Gauthier, T., Kaliszyk, C.: Aligning concepts across proof assistant libraries. J. Symbolic Computation **90**, 89–123 (2019). https://doi.org/10.1016/j.jsc.2018.04.005
7. Gauthier, T., Kaliszyk, C., Urban, J.: Initial experiments with statistical conjecturing over large formal corpora. In: Kohlhase, A. (ed.) Work in Progress at the Conference on Intelligent Computer Mathematics 2016 (CICM-WiP 2016). CEUR, vol. 1785, pp. 219–228. CEUR-WS.org (2016)
8. Grabowski, A., Korniłowicz, A., Naumowicz, A.: Mizar in a nutshell. J. Formalized Reasoning **3**(2), 153–245 (2010). https://doi.org/10.6092/issn.1972-5787/1980, `https://doi.org/10.6092/issn.1972-5787/1980`
9. Grover, A., Leskovec, J.: node2vec: Scalable feature learning for networks (2016)
10. Hales, T.C.: Introduction to the Flyspeck project. In: Coquand, T., Lombardi, H., Roy, M.F. (eds.) Mathematics, Algorithms, Proofs. pp. 1–11. No. 05021 in Dagstuhl Seminar Proceedings, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany (2006), `http://drops.dagstuhl.de/opus/volltexte/2006/432`

11. Harrison, J.: HOL Light: An overview. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher Order Logics (TPHOLs 2009). LNCS, vol. 5674, pp. 60–66. Springer (2009). https://doi.org/10.1007/978-3-642-03359-9_4, https://doi.org/10.1007/978-3-642-03359-9_4
12. Heras, J., Komendantskaya, E.: Proof pattern search in coq/ssreflect. CoRR **abs/1402.0081** (2014), http://arxiv.org/abs/1402.0081
13. Huet, G.P., Herbelin, H.: 30 years of research and development around Coq. In: Jagannathan, S., Sewell, P. (eds.) ACM Symposium on Principles of Programming Languages, POPL 2014. pp. 249–250. ACM (2014). https://doi.org/10.1145/2535838.2537848, https://doi.org/10.1145/2535838.2537848
14. Hurd, J.: The opentheory standard theory library. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods. pp. 177–191. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
15. Joulin, A., Grave, E., Bojanowski, P., Mikolov, T.: Bag of tricks for efficient text classification. In: Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers. pp. 427–431. Association for Computational Linguistics, Valencia, Spain (Apr 2017), https://www.aclweb.org/anthology/E17-2068
16. Kaliszyk, C., Krauss, A.: Scalable LCF-style proof translation. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) Proc. of the 4th International Conference on Interactive Theorem Proving (ITP'13). LNCS, vol. 7998, pp. 51–66. Springer (2013), http://dx.doi.org/10.1007/978-3-642-39634-2_7
17. Kaliszyk, C., Urban, J.: Lemma mining over HOL Light. In: LPAR. pp. 503–517 (2013)
18. Kaliszyk, C., Urban, J.: Learning-assisted automated reasoning with Flyspeck. J. Autom. Reasoning **53**(2), 173–213 (2014). https://doi.org/10.1007/s10817-014-9303-3, http://dx.doi.org/10.1007/s10817-014-9303-3
19. Kaliszyk, C., Urban, J.: HOL(y)Hammer: Online ATP service for HOL Light. Mathematics in Computer Science **9**(1), 5–22 (2015). https://doi.org/10.1007/s11786-014-0182-0
20. Keller, C., Werner, B.: Importing hol light into coq. In: Kaufmann, M., Paulson, L.C. (eds.) Interactive Theorem Proving. pp. 307–322. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
21. Klein, G.: Proof engineering considered essential. In: Jones, C.B., Pihlajasaari, P., Sun, J. (eds.) FM 2014: Formal Methods - 19th International Symposium. LNCS, vol. 8442, pp. 16–21. Springer (2014). https://doi.org/10.1007/978-3-319-06410-9_2, https://doi.org/10.1007/978-3-319-06410-9_2
22. McCarthy, J.: Recursive functions symbolic expressions and their computation by machine, Part I. Communications of the ACM **3**(4), 184–195 (Apr 1960). https://doi.org/10/fvx5pv, http://dl.acm.org/citation.cfm?id=367177.367199, zSCC: NoCitationData[s0] Publisher: ACM ISBN: 0001-0782
23. Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient estimation of word representations in vector space. In: Bengio, Y., LeCun, Y. (eds.) 1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings (2013), http://arxiv.org/abs/1301.3781
24. Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J.: Distributed representations of words and phrases and their compositionality. In: Burges, C.J.C., Bottou, L., Welling, M., Ghahramani, Z., Weinberger, K.Q. (eds.) Advances in Neural Information Processing Systems. vol. 26. Curran

Associates, Inc. (2013), https://proceedings.neurips.cc/paper/2013/file/9aa42b31882ec039965f3c4923ce901b-Paper.pdf

25. Müller, D., Gauthier, T., Kaliszyk, C., Kohlhase, M., Rabe, F.: Classification of alignments between concepts of formal mathematical systems. In: Geuvers, H., England, M., Hasan, O., Rabe, F., Teschke, O. (eds.) 10th International Conference on Intelligent Computer Mathematics (CICM'17). LNCS, vol. 10383, pp. 83–98. Springer (2017). https://doi.org/10.1007/978-3-319-62075-6_7
26. Obua, S., Skalberg, S.: Importing hol into isabelle/hol. In: Furbach, U., Shankar, N. (eds.) Automated Reasoning. pp. 298–302. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
27. Paulson, L.C.: Isabelle: The next seven hundred theorem provers. In: Lusk, E.L., Overbeek, R.A. (eds.) 9th International Conference on Automated Deduction, CADE 1988. LNCS, vol. 310, pp. 772–773. Springer (1988). https://doi.org/10.1007/BFb0012891, https://doi.org/10.1007/BFb0012891
28. Pennington, J., Socher, R., Manning, C.D.: Glove: Global vectors for word representation. In: EMNLP. vol. 14, pp. 1532–1543 (2014)
29. Perozzi, B., Al-Rfou, R., Skiena, S.: Deepwalk: Online learning of social representations. Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining (Aug 2014). https://doi.org/10.1145/2623330.2623732, http://dx.doi.org/10.1145/2623330.2623732
30. Rabe, F.: The MMT API: A generic MKM system. In: Carette, J., Aspinall, D., Lange, C., Sojka, P., Windsteiger, W. (eds.) Intelligent Computer Mathematics - MKM, Calculemus, DML, and Systems and Projects 2013, Held as Part of CICM 2013, Bath, UK, July 8-12, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7961, pp. 339–343. Springer (2013). https://doi.org/10.1007/978-3-642-39320-4_25, https://doi.org/10.1007/978-3-642-39320-4_25
31. Řehůřek, R., Sojka, P.: Software Framework for Topic Modelling with Large Corpora. In: Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks. pp. 45–50. ELRA, Valletta, Malta (May 2010), http://is.muni.cz/publication/884893/en
32. So, C.M., Watt, S.M.: On the conversion between content mathml and openmath. In: Proc. of the Conference on the Communicating Mathematics in the Digital Era (CMDE 2006). pp. 169–182 (2006)
33. Sutcliffe, G.: The TPTP world - infrastructure for automated reasoning. In: LPAR (Dakar). pp. 1–12 (2010)
34. Urban, J.: MoMM - fast interreduction and retrieval in large libraries of formalized mathematics. Int. J. on Artificial Intelligence Tools **15**(1), 109–130 (2006), http://ktiml.mff.cuni.cz/~urban/MoMM/momm.ps
35. Urban, J.: MPTP 0.2: Design, implementation, and initial experiments. J. Autom. Reasoning **37**(1-2), 21–43 (2006). https://doi.org/10.1007/s10817-006-9032-3, https://doi.org/10.1007/s10817-006-9032-3