Learning Semantic Correspondences in Technical Documentation

Kyle Richardson and **Jonas Kuhn**
Institute of Natural Language Processing
University of Stuttgart
{kyle,jonas}@ims.uni-stuttgart.de

Abstract

We consider the problem of translating high-level textual descriptions to formal representations in technical documentation as part of an effort to model the meaning of such documentation. We focus specifically on the problem of learning translational correspondences between text descriptions and grounded representations in the target documentation, such as formal representation of functions or code templates. Our approach exploits the parallel nature of such documentation, or the tight coupling between high-level text and the low-level representations we aim to learn. Data is collected by mining technical documents for such parallel text-representation pairs, which we use to train a simple semantic parsing model. We report new baseline results on sixteen novel datasets, including the standard library documentation for nine popular programming languages across seven natural languages, and a small collection of Unix utility manuals.

1 Introduction

Technical documentation in the computer domain, such as source code documentation and other how-to manuals, provide high-level descriptions of how lower-level computer programs and utilities work. Often these descriptions are coupled with formal representations of these lower-level features, expressed in the target programming languages. For example, Figure 1.1 shows the source code documentation (in red/bold) for the `max` function in the Java programming language paired with the representation of this function in the underlying Java language (in black). This formal representation captures the name of the function, the return

1. Java Documentation

```
*Returns the greater of two long values
*
* @param a an argument
* @param b another argument
* @return the larger of a and b
* @see java.lang.Long#MAX_VALUE
*/
 public static long max(long a, long b)
```

2. Clojure Documentation

```
(defn random-sample
   "Returns items from coll with random
   probability of prob (0.0 - 1.0)"
   ([prob coll] ...))
```

3. PHP documentation (French)

```
Ajoute une valeur comme dernier élément
*
* @param value La valeur á ajouter
* @see ArrayIterations::next()
*/
public void append(mixed $value)
```

Figure 1: Example source code documentation.

value, the types of arguments the function takes, among other details related to the function's place and visibility in the overall source code collection or API.

Given the high-level nature of the textual annotations, modeling the meaning of any given description is not an easy task, as it involves much more information than what is directly provided in the associated documentation. For example, capturing the meaning of the description *the greater of* might require having a background theory about quantity/numbers and relations between different quantities. A first step towards capturing the meaning, however, is learning to translate this description to symbols in the target representation, in this case to the `max` symbol. By doing this translation to a formal language, modeling and learning the subsequent semantics becomes easier since we are eliminating the ambiguity of ordinary lan-

Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, pages 1612–1622
Vancouver, Canada, July 30 - August 4, 2017. ©2017 Association for Computational Linguistics
https://doi.org/10.18653/v1/P17-1148

```
                Unix Utility Manual
NAME : dappprof
  profile user and lib function usage.
SYNOPSIS dappprof [-ac] -p PID | command
DESCRIPTION
  -p PID    examine the PID ...
EXAMPLES
  Print elapsed time for PID 1871
     dappprof -p PID=1871
SEE ALSO: dapptrace(1M), dtrace(1M), ...
```

Figure 2: An example computer utility manual in the Unix domain. Descriptions of example uses are shown in red.

guage. Similarly, we would want to first translate the description *two long values*, which specifies the number and type of argument taken by this function, to the sequence `long a,long b`.

By focusing on translation, we can create new datasets by mining these types of source code collections for sets of parallel text-representation pairs. Given the wide variety of available programming languages, many such datasets can be constructed, each offering new challenges related to differences in the formal representations used by different programming languages. Figure 1.2 shows example documentation for the Clojure programming language, which is part of the Lisp family of languages. In this case, the description *Returns random probability of* should be translated to the function name `random-sample` since it describes what the overall function does. Similarly, the argument descriptions *from coll* and *of prob* should translate to `coll` and `prob`.

Given the large community of programmers around the world, many source code collections are available in languages other than English. Figure 1.3 shows an example entry from the French version of the PHP standard library, which was translated by volunteer developers. Having multilingual data raises new challenges, and broadens the scope of investigations into this type of semantic translation.

Other types of technical documentation, such as utility manuals, exhibit similar features. Figure 2 shows an example manual in the domain of Unix utilities. The textual description in red/bold describes an example use of the *dappprof* utility paired with formal representations in the form of executable code. As with the previous examples, such formal representations do not capture the full meaning of the different descriptions, but serve as a convenient operationalization, or *translational semantics*, of the meaning in Unix. *Print elapsed time*, for example, roughly describes what the `dappprof` utility does, whereas *PID 1871* describes the second half of the code sequence.

In both types of technical documentation, information is not limited to raw pairs of descriptions and representations, but can include other information and clues that are useful for learning. Java function annotations include textual descriptions of individual arguments and return values (shown in green). Taxonomic information and pointers to related functions or utilities are also annotated (e.g., the `@see` section in Figure 1, or `SEE ALSO` section in Figure 2). Structural information about code sequences, and the types of abstract arguments these sequences take, are described in the `SYNOPSIS` section of the Unix manual. This last piece of information allows us to generate abstract code templates, and generalize individual arguments. For example, the raw argument `1871` in the sequence `dappprof -p 1871` can be typed as a `PID` instance, and an argument of the `-p` flag.

Given this type of data, a natural experiment is to see whether we can build programs that translate high-level textual descriptions to correct formal representations. We aim to learn these translations using raw text-meaning pairs as the sole supervision. Our focus is on learning function translations or representations within nine programming language APIs, each varying in size, representation style, and source natural language. To our knowledge, our work is the first to look at translating source code descriptions to formal representations using such a wide variety of programming and natural languages. In total, we introduce fourteen new datasets in the source code domain that include seven natural languages, and report new results for an existing dataset. As well, we look at learning simple code templates using a small collection of English Unix manuals.

The main goal of this paper is to establish strong baselines results on these resources, which we hope can be used for benchmarking and developing new semantic parsing methods. We achieved initial baselines using the language modeling and translation approach of Deng and Chrupała (2014). We also show that modest improvements can be achieved by using a more conventional discriminative model (Zettlemoyer and Collins, 2009) that, in part, exploits document-level features from the technical documentation sets.

2 Related Work

Our work is situated within research on semantic parsing, which focuses on the problem of generating formal meaning representations from text for natural language understanding applications. Recent interest in this topic has centered around learning meaning representation from example text-meaning pairs, for applications such as automated question-answering (Berant et al., 2013), robot control (Matuszek et al., 2012) and text generation (Wong and Mooney, 2007a).

While generating representations for natural language understanding is a complex task, most studies focus on the translation or generation problem independently of other semantic or knowledge representation issues. Earlier work looks at supervised learning of logical representations using example text-meaning pairs using tools from statistical machine translation (Wong and Mooney, 2006) and parsing (Zettlemoyer and Collins, 2009). These methods are meant to be applicable to a wide range of translation problems and representation types, which make new parallel datasets or resources useful for furthering the research.

In general, however, such datasets are hard to construct since building them requires considerable domain knowledge and knowledge of logic. Alternatively, we construct parallel datasets automatically from technical documentation, which obviates the need for annotation. While the formal representations are not actual logical forms, they still provide a good test case for testing how well semantic parsers learn translations to representations.

To date, most benchmark datasets are limited to small controlled domains, such as geography and navigation. While attempts have been made to do open-domain semantic parsing using larger, more complex datasets (Berant et al., 2013; Pasupat and Liang, 2015), such resources are still scarce. In Figure 3, we compare the details of one widely used dataset, Geoquery (Zelle and Mooney, 1996), to our new datasets. Our new resources are on average much larger than geoquery in terms of the number of example pairs, and the size of the different language vocabularies. Most existing datasets are also primarily English-based, while we focus on learning in a multilingual setting using several new moderately sized datasets.

Within semantic parsing, there has also been work on situated or grounded learning, that involves learning in domains with weak supervision and indirect cues (Liang, 2016; Richardson and Kuhn, 2016). This has sometimes involved learning from automatically generated parallel data and representations (Chen and Mooney, 2008) of the type we consider in this paper. Here one can find work in technical domains, including learning to generate regular expressions (Manshadi et al., 2013; Kushman and Barzilay, 2013) and other types of source code (Quirk et al., 2015), which ultimately aim to solve the problem of natural language programming. We view our work as one small step in this general direction.

Our work is also related to software components retrieval and builds on the approach of Deng and Chrupała (2014). Robustly learning the translation from language to code representations can help to facilitate natural language querying of API collections (Lv et al., 2015). As part of this effort, recent work in machine learning has focused on the similar problem of learning code representations using resources such as StackOverflow and Github. These studies primarily focus on learning longer programs (Allamanis et al., 2015) as opposed to function representations, or focus narrowly on a single programming language such as Java (Gu et al., 2016) or on related tasks such as text generation (Iyer et al., 2016; Oda et al., 2015). To our knowledge, none of this work has been applied to languages other than English or such a wide variety of programming languages.

3 Mapping Text to Representations

In this section, we formulate the basic problem of translating to representations in technical documentation.

3.1 Problem Description

We use the term *technical documentation* to refer to two types of resources: textual descriptions inside of source code collections, and computer utility manuals. In this paper, the first type includes high-level descriptions of functions in standard library source code documentation. The second type includes a collection of Unix manuals, also known as man pages. Both types include pairs of text and code representations.

Dataset	#Pairs	#Descr.	Symbols	#Words	Vocab.	Example Pairs (x, z), **Goal:** learn a function $x \to z$
Java	7,183	4,804	4,072	82,696	3,721	x : Compares this Calendar to the specified Object. z : `boolean util.Calendar.equals(Object obj)`
Ruby	6,885	1,849	3,803	67,274	5,131	x : Computes the arc tangent given y and x. z : `Math.atan2(y,x)` $\to$ `Float`
PHP$_{en}$	6,611	13,943	8,308	68,921	4,874	x : Delete an entry in the archive using its name. z : `bool ZipArchive::deleteName(string $name)`
Python	3,085	429	3,991	27,012	2,768	x : Remove the specific filter from this handler. z : `logging.Filterer.removeFilter(filter)`
Elisp	2,089	1,365	1,883	30,248	2,644	x : This function returns the total height, in lines, of the window. z : `(window-total-height window round)`
Haskell	1,633	255	1,604	19,242	2,192	x : Extract the second component of a pair. z : `Data.Tuple.snd :: (a, b) -> b`
Clojure	1,739	–	2,569	17,568	2,233	x : Returns a lazy seq of every nth item in coll. z : `(core.take-nth n coll)`
C	1,436	1,478	1,452	12,811	1,835	x : Returns the current file position of the stream stream. z : `long int ftell(FILE *stream)`
Scheme	1,301	376	1,343	15,574	1,756	x : Returns a new port with type port-type and the given state. z : `(make-port port-type state)`
Unix	921	940	1,000	11,100	2,025	x : To get policies for a specific user account. z : `pwpolicy -u username -getpolicy`
Geoquery	880	–	167	6,663	279	x : What is the tallest mountain in America? z : `(highest(mountain(loc_2(countryid usa))))`

Figure 3: Description of our English corpus collection with example text/function pairs.

We will refer to the target representations in these resources as *API components*, or components. In source code, components are formal representations of functions, or *function signatures* (Deng and Chrupała, 2014). The form of a function signature varies depending on the resource, but in general gives a specification of how a function is named and structured. The example function signatures in Figure 3 all specify a function name, a list of arguments, and other optional information such as a return value and a namespace. Components in utility manuals are short executable code sequences intended to show an example use of a utility. We assume typed code sequences following Richardson and Kuhn (2014), where the constituent parts of the sequences are abstracted by type.

Given a set of example text-component pairs, $D = \{(x_i, z_i)\}_{i=1}^{n}$, the goal is to learn how to generate correct, well-formed components $z \in \mathcal{C}$ for each input x. Viewed as a semantic parsing problem, this treats the target components as a kind of formal meaning representation, analogous to a logical form. In our experiments, we assume that the complete set of output components are known. In the API documentation sets, this is because each standard library contains a finite number of function representations, roughly corresponding to the number of pairs as shown in Figure 3. For a given input, therefore, the goal is to find the best candidate function translation within the space of the total API components $\mathcal{C}$ (Deng and Chrupała, 2014).

Given these constraints, our setup closely resembles that of Kushman et al. (2014), who learn to parse algebra word problems using a small set of equation templates. Their approach is inspired by template-based information extraction, where templates are recognized and instantiated by slot-filling. Our function signatures and code templates have a similar slot-like structure, consisting of slots such as return value, arguments, function name and namespace.

3.2 Language Modeling Baselines

Existing approaches to semantic parsing formalize the mapping from language to logic using a variety of formalisms including CFGs (Börschinger et al., 2011), CCGs (Kwiatkowski et al., 2010), synchronous CFGs (Wong and Mooney, 2007b). Deciding to use one formalism over another is often motivated by the complexities of the target representations being learned. For example, recent interest in learning graph-based representations such as those in the AMR bank (Banarescu et al., 2013) requires parsing models that can generate complex graph shaped derivations such as CCGs (Artzi et al., 2015) or HRGs (Peng et al., 2015). Given the simplicity of our API representations, we opt for a simple semantic parsing model that exploits the finiteness of our target representations.

Following ((Deng and Chrupała, 2014); henceforth DC), we treat the problem of component translation as a language modeling problem (Song and Croft, 1999). For a given *query* sequence or text $x = w_i, .., w_I$ and *component* sequence $z = u_j, .., u_J$, the probability of the component given the query is defined as follows using Bayes' theorem: $p(z|x) \propto p(x|z)p(z)$.

By assuming a uniform prior over the probability of each component $p(z)$, the problem reduces to computing $p(x|z)$, which is where language modeling is used. Given each word w_i in the query, a unigram model is defined as $p(x|z) = \prod_{i=1}^{I} p(w_i|z)$. Using this formulation, we can then define different models to estimate $p(w|z)$.

Term Matching As a baseline for $p(w|z)$, DC define a *term matching* approach that exploits the fact that many queries in our English datasets share vocabulary with target component vocabulary. A smoothed version of this baseline is defined below, where $f(w|z)$ is the frequency of matching terms in the target signature, $f(w|\mathcal{C})$ is frequency of the term word in the overall documentation collection, and λ is a smoothing parameter (for Jelinek-Mercer smoothing):

$$p(x|z) = \prod_{w \in x} (1 - \lambda) f(w|z) + \lambda f(w|\mathcal{C})$$

Translation Model In order to account for the co-occurrence between non-matching words and component terms, DC employ a word-based translation model, which models the relation between natural language words w_j and individual component terms u_j. In this paper, we limit ourselves to sequence-based word alignment models (Och and Ney, 2003), which factor in the following manner:

$$p(x|z) = \prod_{i=1}^{I} \sum_{j=0}^{J} p_t(w_i|u_j) p_d(l(j)|i, I, J)$$

Here each $p_t(w_i|u_j)$ defines an (unsmoothed) multinomial distribution over a given component term u_j for all words w_j. The function p_d is a distortion parameter, and defines a dependency between the alignment positions and the lengths of both input strings. This function, and the definition of $l(j)$, assumes different forms according to the particular alignment model being used. We consider three different types of alignment models each defined in the following way:

Algorithm 1 Rank Decoder

```
Input: Query $x$, Components $\mathcal{C}$ of size $m$, rank $k$, model $\mathcal{A}$,
    sort function K-BEST
Output: Top $k$ components ranked by $\mathcal{A}$ model score $p$
1: procedure RANKCOMPONENTS($x, \mathcal{C}, k, \mathcal{A}$)
2:     SCORES ← []                               ▷ Initialize score list
3:     for each component $c \in \mathcal{C}$ do
4:         $p \leftarrow$ ALIGN$_{\mathcal{A}}(x, c)$                ▷ Score using $\mathcal{A}$
5:         SCORES += $(c, p)$                       ▷ Add to list
6:     return K-Best(SCORES,$k$)                 ▷ $k$ best components
```

$$p_d(l(j)|...) = \begin{cases} \frac{1}{J+1} & (1) \\ a(j|i, I, J) & (2) \\ a(t(j)|i, I, tlen(J)) & (3) \end{cases}$$

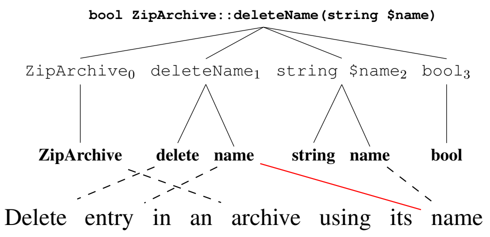
Models (1-2) are the classic IBM word-alignment models of Brown et al. (1993). IBM Model 1, for example, assumes a uniform distribution over all positions, and is the main model investigated in DC. For comparison, we also experiment with IBM Model 2, where each $l(j)$ refers to the string position of j in the component input, and $a(..)$ defines a multinomial distribution such that $\sum_{j=0}^{J} a(j|i, I, J) = 1.0$.

We also define a new tree based alignment model (3) that takes into account the syntax associated with the function representations. Each $l(j)$ is the relative tree position of the alignment point, shown as $t(j)$, and $tlen(J)$ is the length of the tree associated with z. This approach assumes a tree representation for each z. We generated these trees heuristically by preserving the information that is lost when components are converted to a linear sequence representation. An example structure for PHP is shown in Figure 4, where the red solid line indicates the types of potential errors avoided by this model.

Learning is done by applying the standard EM training procedure of Brown et al. (1993).

3.3 Ranking and Decoding

Algorithm 1 shows how to rank API components. For a text input x, we iterate through all known API components $\mathcal{C}$ and assign a score using a model $\mathcal{A}$. We then rank the components by their scores using a K-BEST function. This method serves as a type of word-based decoding algorithm


bool ZipArchive::deleteName(string $name)
ZipArchive0
deleteName1
string $name2
bool3
ZipArchive
delete
name
string
name
bool
Delete entry in an archive using its name

$$
\begin{array}{lcl}
\mathbf{X}_{012} & \rightarrow & \langle\, \mathbf{X}_{\boxed{01}}\ \mathbf{X}_{\boxed{2}},\ \mathbf{X}_{\boxed{01}}\ \mathbf{X}_{\boxed{2}}\ \texttt{bool}\,\rangle \\
\mathbf{X}_{01} & \rightarrow & \langle\, \mathbf{X}_{\boxed{1}}\ \text{in an}\ \mathbf{X}_{\boxed{0}},\ \mathbf{X}_{\boxed{0}}\ \mathbf{X}_{\boxed{1}}\,\rangle \\
\mathbf{X}_{1} & \rightarrow & \langle\, \text{Delete}\ \mathbf{X}_{\boxed{1}},\ \texttt{delete}\ \mathbf{X}_{\boxed{1}}\,\rangle \\
\mathbf{X}_{1} & \rightarrow & \langle\, \text{entry},\ \texttt{name}\,\rangle \\
\mathbf{X}_{0} & \rightarrow & \langle\, \text{archive},\ \texttt{ZipArchive}\,\rangle \\
\mathbf{X}_{2} & \rightarrow & \langle\, \text{using its}\ \mathbf{X}_{\boxed{2}},\ \mathbf{X}_{\boxed{2}}\rangle \\
\mathbf{X}_{2} & \rightarrow & \langle\, \text{name},\ \texttt{string \$name}\,\rangle
\end{array}
$$

Figure 4: An example tree structure (above) associated with an input component. Below are Hiero rules (Chiang, 2007) extracted from the alignment and tree information.

which is simplified by the finite nature of the target language. The complexity of the scoring procedure, lines 3-5, is linear over the number components m in $\mathcal{C}$. In practice, we implement the K-BEST sorting function on line 6 as a binary insertion sort on line 5, resulting in an overall complexity of $O(m\ log\ m)$.

While iterating over m API components might not be feasible given more complicated formal languages with recursion, a more clever decoding algorithm could be applied, e.g., one based on the lattice decoding approach of (Dyer et al., 2008). Since we are interested in providing initial baseline results, we leave this for future work.

4 Discriminative Approach

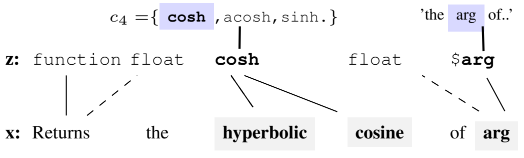
In this section, we introduce a new model that aims to improve on the previous baseline methods.

While the previous models are restricted to word-level information, we extend this approach by using a discriminative reranking model that captures phrase information to see if this leads to an improvement. This model can also capture document-level information from the APIs, such as the additional textual descriptions of parameters, *see also* declarations or classes of related functions and syntax information.

4.1 Modeling

Like in most semantic parsing approaches (Zettlemoyer and Collins, 2009; Liang et al., 2011), our model is defined as a conditional log-linear model over components $z \in \mathcal{C}$ with parameters $\theta \in \mathbb{R}^b$, and a set of feature functions $\phi(x, z)$: $p(\,z|\,x; \theta) \propto e^{\theta \cdot \phi(x,z)}$.

Formally, our training objective is to maximize the conditional log-likehood of the correct component output z for each input x: $\mathcal{O}(\theta) = \sum_{i=1}^{n} \log\ p\,(z_i\,|\,x_i; \theta)$.


c_4 ={ cosh ,acosh,sinh.}
'the arg of..'
z: function float cosh float $arg
x: Returns the hyperbolic cosine of arg

ϕ(x,z) =

Model score:	is it in top 5..10?
Pairs/Alignments:	(hyperbolic, `cosh`) $=1$, (cosine, `cosh`) $=1, \ldots$
Phrases:	(hyperbolic cosine, `cosh`) $=1$, (of arg, `float $arg`) $=\ldots$
See also:	(hyperbolic, $c_4 = \{$`cos,..`$\}$) $=1$, (arg, c_4) $=1$, ...
In Descr.:	(arg, , `$arg`) $=1$, (arg , `float`) $=0, \ldots$
Trees/Matches	(hyperbolic, `cosh`, NAME_NODE) = 1, number of matches= ...

Figure 5: Example features used by our rerankers.

4.2 Features

Our model uses word-level features, such as word match, word pairs, as well as information from the underlying aligner model such as Viterbi alignment information and model score. Two additional categories of non-word features are described below. An illustration of the feature extraction procedure is shown in Figure 5 [1].

Phrases Features We extract phrase features (e.g., (hyper. cosine, `cosh`) in Figure 5) from example text component pairs by training symmetric word aligners and applying standard word-level heuristics (Koehn et al., 2003). Additional features, such as phrase match/overlap, tree positions of phrases, are defined over the extracted phrases.

We also extract hierarchical phrases (Chiang, 2007) using a variant of the SAMT method of Zollmann and Venugopal (2006) and the component syntax trees. Example rules are shown in Figure 4, where *gaps* (i.e., symbols in square brackets) are filled with smaller phrase-tree alignments.

Document Level Features Document features are of two categories. The first includes additional textual descriptions of parameters, return values, and modules. One class of features is whether certain words under consideration appear in the `@param` and `@return` descriptions of the target components. For example, the *arg* token in

[1]A more complete description of features is included as supplementary material, along with all source code.

Algorithm 2 Online Rank Learner

```
Input: Dataset $D$, components $\mathcal{C}$, iterations $T$, rank $k$, learning rate $\alpha$, model $\mathcal{A}$, ranker function RANK
Output: Weight vector $\theta$
1: procedure LEARNRERANKER($D, \mathcal{C}, T, k, \alpha, \mathcal{A}$, RANK)
2:     $\theta \leftarrow 0$                                    ▷ Initialize
3:     for t ∈ $1..T$ do
4:         for pairs $(x_i, z_i) \in D$ do
5:             $\mathcal{S} = \text{RANK}(x_i, \mathcal{C}, k, \mathcal{A})$        ▷ Scored candidates
6:             $\Delta = \phi(x_i, z_i) - E_{s \in \mathcal{S} \sim p(s|x_i;\theta)}[\phi(x_i, s)]$
7:             $\theta = \theta + \alpha\Delta$                         ▷ Update online
8:     return $\theta$
```

Figure 5 appears in the textual description of the `$arg` parameter elsewhere in the documentation string.

Other features relate to general information about abstract symbol categories, as specified in *see-also* assertions, or *hyper-link* pointers. By exploiting this information, we extract general classes of functions, for example the set of hyperbolic function (e.g., `sinh`, `cosh`, shown as c_4 in Figure 5), and associate these classes with words and phrases (e.g., *hyperbolic* and *hyperbolic cosine*).

4.3 Learning

To optimize our objective, we use Algorithm 2. We estimate the model parameters θ using a K-best approximation of the standard stochastic gradient updates (lines 6-7), and a ranker function RANK. We note that while we use the ranker described in Algorithm 1, any suitable ranker or decoding method could be used here.

5 Experimental Setup

5.1 Datasets

Source code documentation Our source code documentation collection consists of the standard library for nine programming languages, which are listed in Figure 3. We also use the translated version of the PHP collection for six additional languages, the details of which are shown in Figure 6. The Java dataset was first used in DC, while we extracted all other datasets for this work.

The size of the different datasets are detailed in both figures. The number of pairs is the number of single sentences paired with function representations, which constitutes the core part of these datasets. The number of descriptions is the number of additional textual descriptions provided in the overall document, such as descriptions of parameters or return values.

Dataset	# Pairs	#Descr.	Symbols	Words	Vocab.
PHP$_{fr}$	6,155	14,058	7,922	70,800	5,904
PHP$_{es}$	5,823	13,285	7,571	69,882	5,790
PHP$_{ja}$	4,903	11,251	6,399	65,565	3,743
PHP$_{ru}$	2,549	6,030	3,340	23,105	4,599
PHP$_{tr}$	1,822	4,414	2,725	16,033	3,553
PHP$_{de}$	1,538	3,733	2,417	17,460	3,209

Figure 6: The non-English PHP datasets.

We also quantify the different datasets in terms of unique symbols in the target representations, shown as *Symbols*. All function representations and code sequences are linearized, and in some cases further tokenized, for example, by converting out of camel case or removing underscores.

Man pages The collection of man pages is from Richardson and Kuhn (2014) and includes 921 text-code pairs that span 330 Unix utilities and man pages. Using information from the synopsis and parameter declarations, the target code representations are abstracted by type. The extra descriptions are extracted from parameter descriptions, as shown in the `DESCRIPTION` section in Figure 1, as well as from the `NAME` sections of each manual.

5.2 Evaluation

For evaluation, we split our datasets into separate training, validation and test sets. For Java, we reserve 60% of the data for training and the remaining 40% for validation (20%) and testing (20%). For all other datasets, we use a 70%-30% split. From a retrieval perspective, these left out descriptions are meant to mimic unseen queries to our model. After training our models, we evaluate on these held out sets by ranking all known components in each resource using Algorithm 1. A predicted component is counted as correct if it matches *exactly* a gold component.

Following DC, we report the accuracy of predicting the correct representation at the first position in the ranked list (Accuracy @1) and within the top 10 positions (Accuracy @10). We also report the mean reciprocal rank MRR, or the multiplicative inverse of the rank of the correct answer.

Baselines For comparison, we trained a bag-of-words classifier (the BoW Model in Table 1). This model uses the occurrence of word-component symbol pairs as binary features, and aims to see if word co-occurrence alone is sufficient to for ranking representations.

Method	Java	PHP$_{en}$	Python	Haskell	Clojure	Ruby	Elisp	C
BOW Model	16.4 63.8 31.8	08.0 40.5 18.1	04.1 33.3 13.6	05.6 55.6 21.7	03.0 49.2 16.4	07.0 38.0 16.9	09.9 54.6 23.5	08.8 48.8 20.0
Term Match	15.7 41.3 24.8	15.6 37.0 23.1	16.6 41.7 24.8	15.4 41.8 24.0	20.7 49.2 30.0	23.1 46.9 31.2	29.3 65.4 41.4	13.1 37.5 21.9
IBM M1	**34.3 79.8 50.2**	**35.5 70.5 47.2**	**22.7 61.0 35.8**	**22.3 70.3 39.6**	**29.6 69.2 41.6**	**31.4 68.5 44.2**	**30.6 67.4 43.5**	21.8 **63.7** 34.4
IBM M2	30.3 77.2 46.5	33.2 67.7 45.0	21.4 58.0 34.4	13.8 68.2 31.8	26.5 64.2 38.2	27.9 66.0 41.4	28.1 66.1 40.7	**23.7** 60.9 **34.6**
Tree Model	29.3 75.4 45.3	28.0 63.2 39.8	17.5 55.4 30.7	17.8 65.4 35.2	23.0 60.3 34.4	27.1 63.3 39.5	26.8 63.2 39.7	18.1 56.2 29.4
M1 Descr.	33.3 77.0 48.7	34.1 71.1 47.2	22.7 62.3 35.9	23.9 69.5 40.2	29.6 69.2 41.6	32.5 70.0 45.5	30.3 73.4 44.7	21.8 62.7 33.9
Reranker	**35.3 81.5 51.4**	**36.9 74.2 49.3**	**25.5 66.0 38.7**	**24.7 73.9 43.0**	**35.0 76.9 47.9**	**35.1 72.5 48.0**	**37.6 80.5 53.3**	**29.7 67.4 40.1**

Method	Scheme	PHP$_{fr}$	PHP$_{es}$	PHP$_{ja}$	PHP$_{ru}$	PHP$_{tr}$	PHP$_{de}$	Unix
BOW Model	06.1 58.1 21.4	06.1 36.9 16.0	05.9 37.8 15.8	04.7 33.2 13.8	04.4 43.6 16.6	05.4 43.4 17.6	04.3 39.2 15.3	08.6 49.6 21.0
Term Match	25.5 61.2 37.4	04.0 15.8 07.7	02.9 10.4 05.4	02.3 11.2 05.2	01.0 09.3 03.6	01.4 08.7 03.6	03.8 09.4 06.2	15.1 33.8 22.4
IBM M1	**32.1 75.5 46.2**	**32.1 65.1 43.5**	**29.5 63.7 41.2**	**23.0 58.1 34.9**	20.3 58.4 **33.3**	**25.9 61.6 38.6**	**22.8 62.5 36.8**	**30.2 66.9 42.2**
IBM M2	29.5 71.4 43.9	30.6 62.2 41.2	26.7 59.8 38.3	22.2 56.1 33.3	18.5 54.5 30.6	23.3 57.6 35.8	19.8 58.6 33.0	23.0 60.4 36.0
Tree Model	26.1 71.2 40.3	27.9 59.3 38.6	25.9 61.0 37.6	22.6 57.8 34.1	**20.6 59.0** 32.9	18,9 55.1 32.0	18.5 56.0 30.6	23.0 58.2 34.3
M1 Descr.	33.1 75.5 47.1	31.0 64.8 42.7	28.6 64.9 41.1	25.4 60.4 37.0	**21.1** 62.6 34.5	29.1 62.0 **41.4**	26.7 62.0 38.8	**34.5** 71.9 47.4
Reranker	**34.6 77.5 48.9**	**32.7 66.8 44.2**	**30.6 66.3 42.6**	**25.8 61.8 37.8**	21.1 **66.8 35.9**	**29.9 63.8** 41.2	**28.0 65.9 40.5**	34.5 **74.8 48.5**

| Accuracy @1 | Accuracy @10 | Mean Reciprocal Rank (MRR) |

Table 1: Test results according to the table below.

Since our discriminative models use more data than the baseline models, which therefore make the results not directly comparable, we train a more comparable translation model, shown as *M1 Descr.* in Table 1, by adding the additional textual data (i.e. parameter and return or module descriptions) to the models' parallel training data.

6 Results and Discussion

Test results are shown in Table 1. Among the baseline models, IBM Model 1 outperforms virtually all other models and is in general a strong baseline. Of particular note is the poor performance of the higher-order translation models based on Model 2 and the Tree Model. While Model 2 is known to outperform Model 1 on more conventional translation tasks (Och and Ney, 2003), it appears that such improvements are not reflected in this type of semantic translation context.

The bag-of-words (BoW) and Term Match baselines are outperformed by all other models. This shows that translation in this context is more complicated than simple word matching. In some cases the term matching baseline is competitive with other models, suggesting that API collections differ in how language descriptions overlap with component names and naming conventions. It is clear, however, that this heuristic only works for English, as shown by results on the non-English PHP datasets in Table 1.

We achieve improvements on many datasets by adding additional data to the translation model (M1 Descr.). We achieve further improvements on all datasets using the discriminative model (Reranker), with most increases in performance occurring at how the top ten items are ranked. This last result suggests that phrase-level and document-level features can help to improve the overall ranking and translation, though in some cases the improvement is rather modest.

Despite the simplicity of our semantic parsing model and decoder, there is still much room for improvement, especially on achieving better Accuracy @1. While one might expect better results when moving from a word-based model to a model that exploits phrase and hierarchical phrase features, the sparsity of the component vocabulary is such that most phrase patterns in the training are not observed in the evaluation. In many benchmark semantic parsing datasets, such sparsity issues do not occur (Cimiano and Minock, 2009), suggesting that state-of-the-art methods will have similar problems when applied to our datasets.

Recent approaches to open-domain semantic parsing have dealt with this problem by using paraphrasing techniques (Berant and Liang, 2014) or distant supervision (Reddy et al., 2014). We expect that these methods can be used to improve our models and results, especially given the wide availability of technical documentation, for example, distributed within the Opus project (Tiedemann, 2012).

Model Errors We performed analysis on some of the incorrect predictions made by our models. For some documentation sets, such as those in the GNU documentation collection[2], information is organized into a small and concrete set of categories/chapters, each corresponding to various features or modules in the language and related functions. Given this information, Figure

[2]https://www.gnu.org/doc/doc.en.html

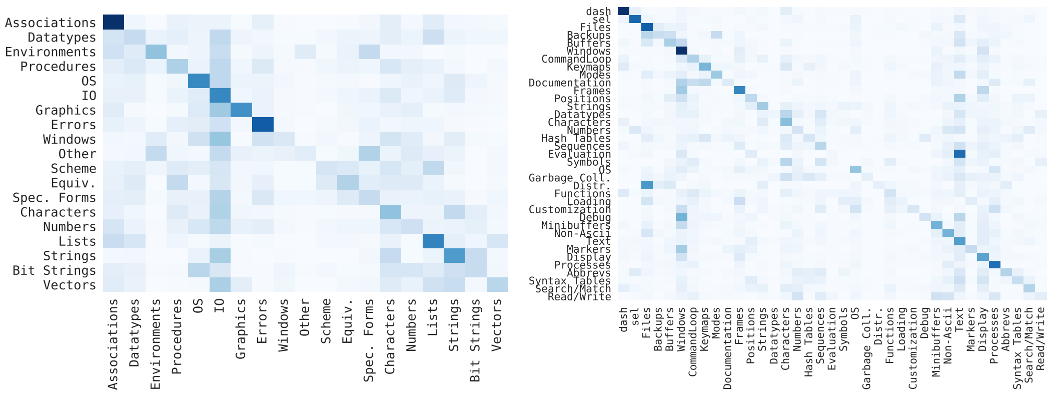


Figure 7: Function predictions by documentation category for Scheme (left) and Elisp (right).

7 shows the confusion between predicting different categories of functions, where the rows show the categories of functions to be predicted and the columns show the different categories predicted. We built these plots by finding the categories of the top 50 non-gold (or erroneous) representations generated for each validation example.

The step-like lines through the diagonal of both plots show that alternative predictions (shaded according to occurrence) are often of the same category, most strikingly for the corner categories. This trend seems stable across other datasets, even among datasets with large numbers of categories. Interestingly, many confusions appear to be between related categories. For example, when making predictions about `Strings` functions in Scheme, the model often generates function related to `BitStrings, Characters` and `IO`. Again, this trend seems to hold for other documentation sets, suggesting that the models are often making semantically sensible decisions.

Looking at errors in other datasets, one common error involves generating functions with the same name and/or functionality. In large libraries, different modules sometimes implement that same core functions, such the `genericpath` or `posixpath` modules in Python. When generating a representation for the text *return size of file*, our model confuses the `getsize(filename)` function in one module with others. Similarly, other subtle distinctions that are not explicitly expressed in the text descriptions are not captured, such as the distinction in Haskell between *safe* and *unsafe* bit shifting functions.

While many of these predictions might be correct, our evaluation fails to take into account these various equivalences, which is an issue that should be investigated in future work. Future work will also look systematically at the effect that types (i.e., in statically typed versus dynamic languages) have on prediction.

7 Future Work

We see two possible use cases for this data. First, for benchmarking semantic parsing models on the task of semantic translation. While there has been a trend towards learning executable semantic parsers (Berant et al., 2013; Liang, 2016), there has also been renewed interest in supervised learning of formal representations in the context of neural semantic parsing models (Dong and Lapata, 2016; Jia and Liang, 2016). We believe that good performance on our datasets should lead to better performance on more conventional semantic parsing tasks, and raise new challenges involving sparsity and multilingual learning.

We also see these resources as useful for investigations into natural language programming. While our experiments look at learning rudimentary translational correspondences between text and code, a next step might be learning to synthesize executable programs via these translations, along the lines of (Desai et al., 2016; Raza et al., 2015). Other document-level features, such as example input-output pairs, unit tests, might be useful in this endeavor.

Acknowledgements

This work was funded by the Deutsche Forschungsgemeinschaft (DFG) via SFB 732, project D2. Thanks also to our IMS colleagues, in particular Christian Scheible, for providing feedback on earlier drafts, as well as to Jonathan Berant for helpful discussions.

References

Miltiadis Allamanis, Daniel Tarlow, Andrew D Gordon, and Yi Wei. 2015. Bimodal modelling of source code and natural language. In *Proceedings of the 32th International Conference on Machine Learning*. volume 951, page 2015.

Yoav Artzi, Kenton Lee, and Luke Zettlemoyer. 2015. Broad-coverage CCG semantic parsing with AMR. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. pages 1699–1710.

Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Martha Palmer, and Nathan Schneider. 2013. Abstract meaning representation for sembanking. In *In Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*.

Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. 2013. Semantic parsing on Freebase from question-answer pairs. In *in Proceedings of EMNLP-2013*. pages 1533–1544.

Jonathan Berant and Percy Liang. 2014. Semantic parsing via paraphrasing. In *Proceedings of ACL-2014*. pages 1415–1425.

Benjamin Börschinger, Bevan K. Jones, and Mark Johnson. 2011. Reducing grounded learning tasks to grammatical inference. In *Proceedings of EMNLP-2011*. pages 1416–1425.

Peter F Brown, Vincent J Della Pietra, Stephen A Della Pietra, and Robert L Mercer. 1993. The mathematics of statistical machine translation: Parameter estimation. *Computational linguistics* 19(2):263–311.

David L. Chen and Raymond J. Mooney. 2008. Learning to sportscast: A test of grounded language acquisition. In *Proceedings of ICML-2008*. pages 128–135.

David Chiang. 2007. Hierarchical phrase-based translation. *computational linguistics* 33(2):201–228.

Philipp Cimiano and Michael Minock. 2009. Natural language interfaces: what is the problem?–a data-driven quantitative analysis. In *International Conference on Application of Natural Language to Information Systems*. Springer, pages 192–206.

Huijing Deng and Grzegorz Chrupała. 2014. Semantic approaches to software component retrieval with English queries. In *Proceedings of LREC-14*. pages 441–450.

Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Subhajit Roy, et al. 2016. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, pages 345–356.

Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. *arXiv preprint arXiv:1601.01280* .

Christopher Dyer, Smaranda Muresan, and Philip Resnik. 2008. Generalizing word lattice translation. *Proceedings of ACL-08* page 1012.

Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API Learning. *arXiv preprint arXiv:1605.08535* .

Srinivasan Iyer, Ioannis Kostas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. *Proceedings of ACL-2016* .

Robin Jia and Percy Liang. 2016. Data recombination for neural semantic parsing. *arXiv preprint arXiv:1606.03622* .

Philipp Koehn, Franz Josef Och, and Daniel Marcu. 2003. Statistical phrase-based translation. In *Proceedings of the NACL-2003*. pages 48–54.

Nate Kushman, Yoav Artzi, Luke Zettlemoyer, and Regina Barzilay. 2014. Learning to automatically solve algebra word problems. In *Proceedings of ACL-2014*. pages 271–281.

Nate Kushman and Regina Barzilay. 2013. Using semantic unification to generate regular expressions from natural language. In *Proceedings of NAACL-2013*.

Tom Kwiatkowski, Luke Zettlemoyer, Sharon Goldwater, and Mark Steedman. 2010. Inducing probabilistic CCG grammars from logical form with higher-order unification. In *Proceedings of EMNLP-2010*. pages 1223–1233.

P. Liang, M. I. Jordan, and D. Klein. 2011. Learning dependency-based compositional semantics. In *Proceedings of ACL-11*. pages 590–599.

Percy Liang. 2016. Learning executable semantic parsers for natural language understanding. *Communications of the ACM* 59(9):68–76.

Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. Codehow: Effective code search based on api understanding and extended boolean model (e). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, pages 260–270.

Mehdi Hafezi Manshadi, Daniel Gildea, and James F Allen. 2013. Integrating programming by example and natural language programming. In *Proceedings of AAAI-2013*.

Cynthia Matuszek, Evan Herbst, Luke Zettlemoyer, and Dieter Fox. 2012. Learning to parse natural language commands to a robot control system. In *Proceedings of the International Symposium on Experimental Robotics (ISER)*.

Franz Josef Och and Hermann Ney. 2003. A systematic comparison of various statistical alignment models. *Computational linguistics* 29(1):19–51.

Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, pages 574–584.

Panupong Pasupat and Percy Liang. 2015. Compositional semantic parsing on semi-structured tables. In *Proceedings of ACL-2015*.

Xiaochang Peng, Linfeng Song, and Daniel Gildea. 2015. A Synchronous Hyperedge Replacement Grammar based approach for AMR parsing. *Proceedings of CoNLL-2015* page 32.

Chris Quirk, Raymond J Mooney, and Michel Galley. 2015. Language to code: Learning semantic parsers for if-this-then-that recipes. In *Proceedings of ACL-2015*. pages 878–888.

Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. 2015. Compositional program synthesis from natural language and examples. In *IJCAI*. pages 792–800.

Siva Reddy, Mirella Lapata, and Mark Steedman. 2014. Large-scale semantic parsing without question-answer pairs. *Transactions of the Association for Computational Linguistics* 2:377–392.

Kyle Richardson and Jonas Kuhn. 2014. UnixMan corpus: A resource for language learning in the Unix domain. In *Proceedings of LREC-2014*.

Kyle Richardson and Jonas Kuhn. 2016. Learning to make inferences in a semantic parsing task. *Transactions of the Association for Computational Linguistics* 4:155–168.

F. Song and W.B Croft. 1999. A general language model for information retrieval. In *in Proceedings International Conference on Information and Knowledge Management*.

Jörg Tiedemann. 2012. Parallel data, tools and interfaces in opus. In *LREC*. volume 2012, pages 2214–2218.

Yuk Wah Wong and Raymond J. Mooney. 2006. Learning for semantic parsing with statistical machine translation. In *Proceedings of HLT-NAACL-2006*. pages 439–446.

Yuk Wah Wong and Raymond J Mooney. 2007a. Generation by inverting a semantic parser that uses statistical machine translation. In *Proceedings of HLT-NAACL-2007*. pages 172–179.

Yuk Wah Wong and Raymond J. Mooney. 2007b. Learning synchronous grammars for semantic parsing with lambda calculus. In *Proceedings of ACL-2007*. Prague, Czech Republic.

John M Zelle and Raymond J Mooney. 1996. Learning to parse database queries using inductive logic programming. In *Proceedings of AAAI-1996*. pages 1050–1055.

Luke S. Zettlemoyer and Michael Collins. 2009. Learning context-dependent mappings from sentences to logical form. In *Proceedings of ACL-2009*. pages 976–984.

Andreas Zollmann and Ashish Venugopal. 2006. Syntax augmented machine translation via chart parsing. In *Proceedings of the Workshop on Statistical Machine Translation*. pages 138–141.