Poster: Inferring API Elements Relevant to an English Query

Thanh V. Nguyen
Iowa State University, USA
thanhng@iastate.edu

Tien N. Nguyen
University of Texas at Dallas, USA
tien.n.nguyen@utdallas.edu

CCS CONCEPTS

• **Software and its engineering** **Software libraries and repositories**; **API languages**;

KEYWORDS

Machine translation; API elements; Text-to-Code Translation

ACM Reference Format:
Thanh V. Nguyen and Tien N. Nguyen. 2018. Poster: Inferring API Elements Relevant to an English Query. In *ICSE '18 Companion: 40th International Conference on Software Engineering Companion, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3183440.3195079

1 INTRODUCTION

Software libraries can be used in different ways, but not all of their APIs are well documented in the documentation and programming guides. To recommend an API usage from a query, recently, going beyond searching for existing code, researchers have aimed to *generate new API code*, by exploring *statistical approaches* including phrase-based statistical machine translation (SMT) [5], probabilistic CFG [4], AST-based translation [2], and deep neural network [3]. A key limitation of existing approaches is the strict order of translation from left to right, leading to low accuracy in the resulting APIs.

In this paper, we develop APITRAN, a *context-sensitive statistical translation* approach that receives an English query on a programming task and suggests the relevant APIs.

To maximize the relevance between the query and the resulting API usage, our algorithm translates the given query into a set of API elements with a *prioritized order*. For example, given the input *"send network messages"*, we want to produce the API elements Socket.open, Message.compose, Message.send, and Socket.close. Left-to-right order in text translation is not efficient for the input due to the large numbers of mapped API elements that need to be considered for each word. Moreover, there exist important words which affect the selections of other words during translation. For example, if we map *"send"* first, there would be many choices such as Printer.send and MailServer.send. However, if *"messages"* is selected for mapping first, the choices for *"send"* is smaller and *"send"* is most likely mapped to Message.send since *"messages"* was already translated to Message.compose. Thus, to determine the next word to map, we consider the *already-translated words* since they help distinguish the context for that word. We call this *word context*. Similarly, to select an API element among several alternatives, we consider the *already-generated API elements*. We call this *code context*. We realize this idea via a *set translation* approach. Next, let us present our approach.

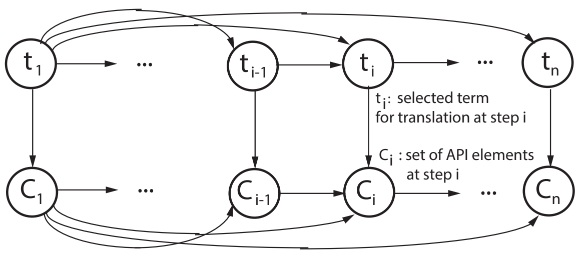
t_1
...
t_{i-1}
t_i
...
t_n
t_i: selected term for translation at step i
C_i : set of API elements at step i
C_1
...
C_{i-1}
C_i
...
C_n

Figure 1: Bayesian Network for APITRAN Translation

2 INFERRING API ELEMENTS

Our algorithm aims to generate a set C of API elements relevant to a given textual query T. We treat this task as translating a phrase into a sequence of APIs. This view gives rise to our formulation.

Let T be a set of key terms in the given phrase and C be a set of API elements. A natural choice is to formulate a probability model in which the resulting API set is conditional on a given text. Formally, we aim to find a subset C^* of the superset of all API elements that maximizes the translation probability $P\left(C|T\right)$. Via conditional probability, we have $P\left(C|T\right) = \frac{P(C|T)}{\sum_C P(C,T)}$.

The marginal probability over T in the denominator does not depend on C, so we are only interested in the joint probability $P\left(C,T\right)$.

Let us use $\{t_1, t_2, ..., t_n\}$ to denote an order of all the terms in T. C_i is a set of API elements translated from the term t_i and T_{i-1} is a set of the $i-1$ already-translated terms. $C_{<i}$ is a set-valued random vector of $i-1$ dimensions corresponding the translated sets for $i-1$ first terms in the order. Formally, $C_{<i} = \left(C_1, C_2, .., C_{i-1}\right)$. However, the union of the vector components is statistically sufficient for the inference about pairwise occurrence frequency. Thus, we define $C_{<i} = \cup_{j<i} C_j$ (*i.e.,* the translated set of API elements) for simplicity. According to Bayes rule, we have

$$\begin{aligned} &P\left(C,T\right) = P\left(C_1, C_2, ..., C_n, t_1, t_2, ..., t_n\right) \\ &= P\left(C_1, t_1\right) P\left(C_2, t_2 | C_1, t_1\right) ... P\left(C_n, t_n | C_1, C_2, ..., C_{n-1}, t_1, ..., t_{n-1}\right) \\ &= \prod_i P\left(C_i, t_i | C_{<i}, T_{i-1}\right), \end{aligned} \quad (1)$$

The goal is to determine an optimal solution of translation order of terms and set of API elements that maximizes the joint translation probability. However, considering all possible orders of T and all combinations of APIs to compute the probability would face combinatorial explosion. Thus, we reduce our model to a Bayesian network in Figure 1. The idea is that the translation is optimized at each step

Table 1: StackOverflow (SO) Dataset for Training

Number of posts	236,919
Avg. number of words per post	132
Size of word dictionary	701,781
Size of API element dictionary	11,834
Avg. number of API elements per post	9.2
Avg. number of extracted keywords per post	20.2
The number of distinct keywords	103,165

i where we find a term t_i and translate a set C_i from t_i given the previous resulting API sets and already-translated terms. Consequently, we maximize stepwise each factor in the product in (1):

$$C_i^* = argmax_{C_i} P\left(C_i, t_i | C_{<i}^*, T_{i-1}\right), \quad (2)$$

where C_i^* is the translated set of API elements at step i; t_i is the term being translated. We decompose the right-hand side of (2):

$$\begin{aligned} P\left(C_i, t_i | C_{<i}^*, T_{i-1}\right) &= P\left(C_i | C_{<i}^*, t_i, T_{i-1}\right) * P\left(t_i | C_{<i}^*, T_{i-1}\right) \\ &= P\left(C_i | C_{<i}^*, t_i\right) * P\left(t_i | T_{i-1}\right), \quad (3) \end{aligned}$$

a) C_i and T_{i-1} are conditionally independent given $C_{<i}^*$ and t_i, and b) t_i is independent of $C_{<i}^*$ given T_{i-1}.

1) To define **the first factor in (3)**, we construct a weighting function $f\left(C_i, C_{i-1}^*, t_i\right)$, then convert it to probability by exponentiating and normalizing over all possible subsets translated by t_i:

$$P\left(C_i | C_{<i}^*, t_i\right) = \frac{\exp f\left(C_i, C_{i-1}^*, t_i\right)}{\sum_{C_i'} \exp f\left(C_i', C_{i-1}^*, t_i\right)}, \quad (4)$$

$$f\left(C_i, C_{i-1}^*, t_i\right) \equiv \sum_{c \in C_i} score_{C_{<i}^*}\left(c\right) * P\left(c | t_i\right), \quad (5)$$

$score_C\left(c\right) = \frac{1}{|C|} \sum_{c' \in C} \frac{N(c,c')}{N(c')}$ where C is the set of current selected API elements, $N\left(c, c'\right)$ is the number of times that c and c' co-occur in a set of APIs for a description in the training corpus, and $N\left(c'\right)$ is the number of times that c' occurs. The idea is that c and c' going together often indicates a relation in an API usage.

$P\left(c | t_i\right)$ is the lexical probability obtained from the IBM model. This formula guarantees to find C_i with high translation probability and high relative co-occurrence frequency with API elements in $C_{<i}^*$.

2) **The second factor in (3)** is constructed as

$$P\left(t_i | T_{i-1}\right) = \frac{\exp score_{T_{i-1}}\left(t_i\right)}{\sum_{t_i' \in T \setminus T_{i-1}} \exp score_{T_{i-1}}\left(t_i'\right)}, \quad (6)$$

Finally, the **relevance score** for each API element c in translated set C is computed at each step i by

$$score\left(c, T\right) \equiv P\left(\{c\}, t_i | C_{<i}^*, T_{i-1}\right) \quad (7)$$

3 EMPIRICAL EVALUATION

To train our model, we used the StackOverflow dataset in [6]. In our dataset (Table 1), we used 236,919 entries with high ratings, each of which has two parts: 1) the textual descriptions of the usage/purpose of some programming task (excluding embedded code elements), and 2) the corresponding set of API elements extracted from the posts (including from descriptions and code snippets).

Table 2: Accuracy in Inferring API Elements with/wo Pivots

	Top-1		Top-2		Top-3		Top-4		Top-5	
	Base	Exp	Base	Exp	Base	Exp	Base	Exp	Base	Exp
Rec	45.8	**69.5**	61.7	**86.5**	69.7	**96.2**	77.4	**97.3**	82.1	**97.4**
Prec	41.2	47.8	31.1	34.7	24.3	27.2	20.8	21.5	18.0	17.7
F-score	43.3	56.6	41.3	49.5	36.1	42.4	32.8	35.3	29.5	30.0
#Element	4.5	5.9	8.0	10.1	11.6	**14.3**	15.1	18.3	18.5	22.3

Procedure. We aimed to evaluate the accuracy of our model. To train the IBM Model (using Berkeley Aligner tool [1]), each entry contains a list of keywords and a set of API elements. In total, we collected 103,165 distinct keywords with 20.2 keywords per post. *After training the IBM Model, each word is mapped on average to 16.46 API elements*. This step is very helpful since it allows API inferring module to consider a number of potential API elements much smaller than the size of API element dictionary (11,834). To collect the testing posts/entries, we randomly selected from the StackOverflow dataset [6] 250 post samples.

With trained IBM Model, our API inferring algorithm produced the API elements for the texts in the testing posts. We compared the inferred sets of elements against the sets of API elements for those posts in the SO dataset [6]. We measured Recall and Precision. We also calculated F-score, the harmonic value: $\text{F-score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$.
Results. Table 2 shows the accuracy of our model when the value of K was varied, i.e., the top-K API elements with the highest scores for each term in the query. Columns Base show the accuracy for the baseline technique using only IBM Model and then selecting top-K elements for each term. Columns Exp are for our model.

As seen, our algorithm achieves much higher accuracy than the baseline model. We cover as many relevant API elements while keeping their number as low as possible. As seen, the recall is from 69.5–97.4%. With K=3 (each term has 3 corresponding API elements), we can cover 96.2% of the correct API elements with 14.3 elements for each post (in the StackOverflow dataset, each post has on average 9.2 elements). Importantly, for performance purpose, with a small number of API elements (*e.g.,* 14.3 elements per post and 96.2% recall), the running time is reasonable (not shown).
Conclusion. Our approach, APITRAN, is quite effective in deriving API elements relevant to a textual query. The set translation formulation enables the consideration of both word and code contexts. Our empirical evaluation shows a promising result for APITRAN.

ACKNOWLEDGMENTS

This work was supported in part by the US National Science Foundation (NSF) grants CCF-1723215, CCF-1723432, TWC-1723198, CCF-1518897, and CNS-1513263.

REFERENCES

[1] The BerkeleyAligner. https://code.google.com/p/berkeleyaligner/.
[2] A. Desai, S. Gulwani, V. Hingorani, N. Jain, A. Karkare, M. Marron, S. R, and S. Roy. Program synthesis using natural language. In ICSE '16. ACM, 2016.
[3] X. Gu, H. Zhang, D. Zhang, and S. Kim. Deep API Learning. In FSE'16. ACM.
[4] T. Gvero and V. Kuncak. Synthesizing Java expressions from free-form queries. In OOPSLA'15, pages 416–432, NY, USA, 2015. ACM.
[5] M. Raghothaman, Y. Wei, and Y. Hamadi. SWIM: synthesizing what I mean. In ICSE'16. ACM Press, 2016.
[6] P. C. Rigby and M. P. Robillard. Discovering essential code elements in informal documentation. In ICSE '13, pages 832–841. IEEE Press, 2013.