PHRASE2SET: Phrase-to-Set Machine Translation and Its Software Engineering Applications

Thanh V. Nguyen
Amazon
Seattle, Washington, USA
thanhng.cs@gmail.com

Aashish Yadavally
Computer Science Department
University of Texas at Dallas, USA
aashish.yadavally@utdallas.edu

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

***Abstract*—Machine translation has been applied to software engineering (SE) problems, e.g., software tagging, language migration, bug localization, auto program repair, etc. However, machine translation primarily supports only sequence-to-sequence transformations and falls short during the translation/transformation from a *phrase or sequence* in the input to a *set* in the output. An example of such a task is tagging the input text in a software library tutorial or a forum entry text with a set of API elements that are relevant to the input.**

In this work, we propose PHRASE2SET, a context-sensitive statistical machine translation model that learns to transform a phrase of a mixture of code and texts into a set of code or text tokens. We first design a token-to-token algorithm that computes the probabilities of mapping individual tokens from phrases to sets. We propose a Bayesian network-based statistical machine translation model that uses these probabilities to decide a translation process that maximizes the joint translation probability. To achieve that, we consider the context of the tokens in the source side and that in the target side via their relative co-occurrence frequencies. We evaluate PHRASE2SET in three SE applications: 1) tagging the fragments of texts in a tutorial with the relevant API elements, 2) tagging the StackOverflow entries with relevant API elements, 3) text-to-API translation. Our empirical results show that PHRASE2SET achieves high accuracy and outperforms the state-of-the-art models in all three applications. We also provide the lessons learned and other potential applications.

***Keywords*-Machine Translation; Phrase-to-Set Translation; Software Tagging; Text-to-Code, Code-to-Text Translation;**

I. INTRODUCTION

In recent years, several approaches have been explored to develop techniques for the translation/transformation between texts and source code in software engineering (SE) applications. An important task in such applications is to derive the set of code and text tokens that are relevant or correspond to the given text or the mixture of code and texts. Specifically, researchers tackled this problem with *machine translation* (MT) approaches such as phrase-based MT [1], graph-based MT [2], neural-network-based MT [3], and deep learning-based encoder-decoder or transformer models [4].

Despite their successes, a key limitation of these approaches is that they enforce the order of the tokens in the output. In some applications, the API elements do not need to follow a strict order. Consider the tutorial tagging task [5], [6], where, given a fragment of texts from the tutorial of a software library, the task is to tag it with relevant API elements that the texts explain. In this application, the set of API elements need not be ordered. Using a sequence-to-sequence (seq-to-seq) model for a phrase-to-set application, one faces such an ordering issue.

During the training phase, a seq-to-seq model requires an ordered set of output tokens $T = \{A, B, ...\}$ for a training sample. There are two ways to address this criterion. First, we can *enforce a specific order (e.g., alphabetical order) on the output tokens for all samples*. However, deciding which order to use is not trivial. Different sequences of output tokens might yield different training results, which affects the prediction quality. Moreover, we cannot try all possible sequences due to a combinatorial explosion. Second, we can *pick a random order for the output tokens of a sample*. However, this leads to a scenario where two samples with the same set of output tokens have different output sequences, e.g., $A, B, ...$ vs $B, A, ...$ In this case, the model treats both output sequences differently and cannot apply information learned from one sample to the other. This also results in low prediction quality. A specific example where this issue is relevant is the bug localization problem [7], where, given a bug report, a model needs to derive the *set* of methods in a project that could be the buggy locations needing fixing. In this application, the set of methods that must all be fixed do not have a strict order.

We propose PHRASE2SET, a context-sensitive statistical machine translation model that learns to transform a phrase of a mixture of code and texts into a set of code and/or text tokens. We first design a token-to-token algorithm that computes the mapping probabilities for individual tokens in phrases to sets. Our Bayesian network-based machine translation model uses these probabilities to decide a translation process maximizing the joint translation probability.

PHRASE2SET has the following three novel components. First, we design an unsupervised learning algorithm based on expectation-maximization that learns the mapping probabilities $P(t|s)$ between each source token s (*i.e.,* text or code token) and a target token t (*i.e.,* text or code token). The training dataset for this algorithm is a parallel corpus of mixtures of code and texts, and the corresponding sets of code tokens.

Second, our model uses these single token-to-token mapping probabilities $P(t|s)$ to derive a translation process for all the source tokens s, while taking into account the context among other source and target tokens. The next source token s' is the one having the highest relative co-occurrence frequency with all of the already-translated tokens. This is justified because

978-1-6654-3786-8/22/$31.00 ©2022 IEEE
DOI 10.1109/SANER53432.2022.00068

the co-occurring tokens on the source side are most likely part of a meaningful sentence. For example, consider the text-to-API application with an input *"send network message"*. Each of the text tokens in the input can be mapped to one or multiple API elements. If *"message"* was already selected for translation into Message.compose, our model would pick *"send"* for translation next, since the token *"send"* has the highest co-occurrence frequency with *"message"* in the training dataset. This choice will likely result in the API element Message.send. Similarly, to select an API element among several alternatives, we consider the *already-generated API elements*. The tokens considered during the translation on both sides are referred to as *source context* and *target context*. Finally, we develop an inferring algorithm to derive the set of code or text tokens in the target side that maximizes the joint translation probability based on the token-to-token mapping probabilities P.

We conducted experiments to evaluate PHRASE2SET in three applications: 1) [*StackOverflow application*] tagging StackOverflow (SO) entries with relevant API elements, 2) [*Tutorial application*] tagging fragments of texts in a tutorial with the relevant API elements, and 3) [*Text-to-API application*] deriving the set of APIs to implement the task described in the given text. We trained PHRASE2SET on 236,919 pairs of textual descriptions from SO and the corresponding sets of API elements for the SO entries [2]. For the StackOverflow application, our results show that PHRASE2SET improves over the state-of-the-art approach *seq2seq* [8], an RNN-based sequence-to-sequence translation model, by *9–22%* in precision and *15–24%* in recall. For the Tutorial application, we show that PHRASE2SET improves the state-of-the-art technique [5] by about *25.5%* in precision and *35.9%* in recall. For the text-to-API application, it also outperforms the baselines in top-k accuracy.

In brief, the key contributions of this paper include:

1) PHRASE2SET [9], a phrase-to-set statistical machine translation model that accepts a mixture of text and code tokens, and returns the set of relevant text or code tokens. This tool does not just grab specific relevant tokens from the text, but also derives relevant tokens that it learns from the dataset and do not appear in the given text.

2) An extensive evaluation to show PHRASE2SET's better performance than the baseline models in three SE applications.

3) Potential applications of PHRASE2SET are suggested.

II. MOTIVATING EXAMPLE AND APPROACH OVERVIEW

A. Real-World Example

Figure 1 shows a StackOverflow entry where a user posted a question on how to make a copy of a file in Android. Based on the data in [2], this entry is tagged with the APIs in Figure 2: FileInputStream, FileOutputStream, FileInputStream.read, FileOutputStream.write, FileInputStream.close, and FileOutputStream.close.

In this tagging application, the fundamental task is to accept as input a given fragment of texts describing a functionality or explaining the usage of some API elements, and to *output the set of API elements* relevant to the texts. Earlier works addressed this problem via information retrieval [10], [11], [12], [13], with the goal of searching for *API code elements* [14]. However, these IR approaches are not effective when the API elements *do not appear* in the texts, *e.g.,* in Figure 1. To address this problem, several works discussed earlier [1], [2], [15], [3] treated the application as a text-to-code translation problem, and explored different statistical MT techniques.

Question 9292954

Title: How to make a copy of a file in Android

In my app I want to save a copy of a certain file with a different name (which I get from user) Do I really need to open the contents of the file and write it to another file? What is the best way to do so?

Figure 1: StackOverflow Entry #9292954

FileInputStream, FileOutputStream, FileInputStream.read, FileOutputStream.write, FileInputStream.close, and FileOutputStream.close

Figure 2: API Elements as the tag for the SO entry #9292954

For training a sequence-to-sequence model, we need to ensure the set of API elements is in a total order. However, API elements are not necessarily ordered. For example, in Figure 2, the API corresponding to closing the first file (via FileInputStream), and that to closing the second one (via FileOutputStream) need not necessarily be in this order. If we decide on a specific order (e.g., alphabetical order), we must also consider other orderings as each might result in a different trained model. Furthermore, since many such orderings are possible, training different models for each is computationally expensive. In contrast, in the case of random order, the model would consider the same set of output tokens but in a different sequence while training, thus affecting its accuracy.

B. Approach Overview

PHRASE2SET aims to translate a sequence of texts/code into a set of texts/code. It works in two phases. To train the model, having a parallel corpus $\mathcal{C}$ of pairs of phrases S and the corresponding sets T is essential. First, our unsupervised learning algorithm (Section III) learns from $\mathcal{C}$ the mapping probabilities $P(t|s)$ between every source token s and every target token t. For this, the algorithm utilizes expectation-maximization technique, which considers *the order of source tokens s in S, but not for the target tokens t in T*. In the illustration in Figure 3, thicker the line, higher the mapping probability.

Next, for each source token s in the given sequence, we create an ordered list of mapped target tokens L_s, ranked based on the calculated $P(t|s)$ values. A naive approach would collect the top-1 mapped target tokens for the source tokens from left-to-right to build the output set. However, this might not maximize the translation probability for the entire phrase.

Thus, in the second phase, given the source tokens $s_1, s_2, ..., s_n$, we determine an order for translating the tokens depending on $P(t|s)$ (Section IV), and decide which target token in each L_s is chosen (Section V). To do so, we use a novel Bayesian network statistical MT model (Section IV) that maximizes the joint translation probability on both sides by considering the

context among the source tokens and the target ones. Figure 4 illustrates the selection of a source token for translation at a step i and many possibilities of the next token to be translated. Along the way, the set of the target tokens in each translation step are accumulated. Details will be explained next.

III. Individual Token-to-Token Mapping

A. Formulation

This section covers our mapping algorithm, which generates individual token mappings in the input and output. We adapt our algorithm from a sequence-to-sequence mapping algorithm [16] as follows. The input is a collection of pairs of phrases and corresponding sets of tokens. Assume that L_S and L_T are two sets that form the pairs, with $s = s_1 s_2 ... s_m$ (source sequence) in L_S and $t = \{t_1 t_2 ... t_l\}$ (target set) in L_T. The goal is to compute the probability $P(s|t)$, *i.e.,* the probability that a sequence s corresponds to a set t, given an observable set t. We consider s to be generated with respect to t by the following generative process. First, the length m of a sequence s is chosen with a probability $P(m|t)$. For each position i ($i \leq m$), the algorithm chooses a token $t_j \in t$ and generates a token s_i depending on t_j. In this scenario, s_i is said to align with t_j, and such a mapping is denoted by the mapping variable $a_i = j$. A token s_i can also be generated without considering any token in t. Here, s_i is considered to be aligned with a special token null. The vector $a = \langle a_1, a_2, ... a_m \rangle$ where $0 \leq a_i \leq l$ is called *a mapping* of s and t ($a_i = 0$ indicates there is no mapping for s_i in t). Considering t is a set, to compute $P(s|t)$, we have:

1. The choice of length m of phrase s is dependent only on the size l of the output set t, *i.e.,* $P(m|t) = \lambda(m, l)$;

2. The choice of the mapping $a_i = j$ depends only on the position i, the sequence length m, and the size l of the target set t, *i.e.,* $P(a_i|i, m, t) = \pi(j, i, m, l)$;

3. The choice of token $s_i = u$ in sequence s depends only on the token $t_j = v$ that it aligns with, *i.e.,* $P(s_i|t_{a_i}, i, m, t) = \tau(u, v)$. This is true because t is a *set*, meaning that t_j's have no ordering dependence between each other.

Based on these independent choices, the model computes:

$$P(s, a|t) = \lambda(m, l) \prod_{i=1}^{m} (\pi(a_i, i, m, l).\tau(s_i, t_{a_i})) \quad (1)$$

Also, $P(s|t)$ is computed by summing over all training pairs:

$$P(s|t) = \sum_{a} P(s, a|t) = \sum_{a_1=0}^{l} ... \sum_{a_m=0}^{l} P(s, \langle a_1, a_2, ... a_m \rangle | t) \quad (2)$$

The model considers (λ, π, τ) as its parameters, which are learned via an Expectation-Maximization algorithm as follows.

B. Expectation-Maximization (EM) Algorithm

The algorithm takes as input a training corpus T and returns the parameters (λ, π, τ). T contains the pairs (s, t) of the phrases and corresponding sets. The algorithm begins by randomly initializing the parameters. Then, it iteratively learns these parameters, where each iteration has two steps: expectation (E-step), and maximization (M-step). In the E-step, it uses the estimated model parameters (λ, π, τ) to infer the expected mappings for all pairs (s, t) in the corpus. In the M-step, the count values collected from those mappings are used to re-estimate the parameters.

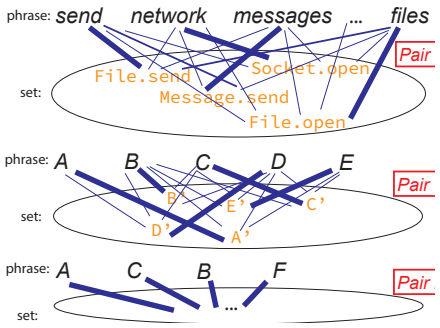

Figure 3: Individual Token-to-Token Single Mapping

Figure 3 shows an illustration. At first, the model attempts all possible mappings. For each pair (s, t), the model initially assigns a small weight for each pair of tokens in s and t. For example, the pairs of (*"network"*, Socket.open), (*"send"*, Message.send), (*"files"*, File) are assigned with some initial weights. The same treatment is applied to all pairs of (s, t) in L_S and L_T. At each iteration, the model adjusts the weights of the mappings based on the occurrence frequencies of those mappings appearing in the parallel corpus. It continues to optimize the mappings in the corpus, until all alignments are exhaustively found, or there is no further improvement between iterations. The final mappings between the source and target tokens are denoted by the lines in Figure 3. The thicker the line, the higher the probability given to the mapping between a source and target token. The result of this phase is the mapping probabilities for all pairs of source and target tokens, which are used in the inferring phase (Section IV).

Consideration of an Ordered Source, Unordered Target: In the E-step, we estimate the parameters and infer the expected mappings for all pairs in the corpus. In the M-step, the (co)-occurrence count values are collected from those mappings in the corpus to re-estimate (λ, π, τ). For the co-occurrence counts of the tokens on the source side, the order of the consecutive tokens is considered since the input is a phrase. In contrast, for the co-occurrence counts of the tokens on the target side, we do not consider their order. We only count the occurrences of the tokens *in the same sets* on the target side.

IV. Phrase2Set: Phrase-to-Set Statistical Machine Translation Model

A. Key ideas

We first present our novel Bayesian network-based statistical MT model that derives translation steps for the source tokens considering the contexts on both sides. Next, we present an *inferring algorithm* that uses these steps to collect the target tokens for the output that maximizes the joint translation probability for the entire phrase. We use the following key ideas:

1. The order of the source tokens in the input phrase has to be considered in the token-to-token mapping algorithm in Section III. If we collect the mapped code tokens in the input phrase from left-to-right, we will face two issues. First, the translation might be inefficient as there will be a large numbers of mapped tokens at each step. For example, let us consider the texts "*open a file*" (S_1) and "*open a socket*" (S_2). If we translate the first token "*open*" in both texts, we will need to consider many possibilities for opening different types of resources. In contrast, if "*file*" in S_1 or "*socket*" in S_2 is selected first for translation, we can narrow the search space for the potential translated API elements corresponding to the token "*open*". Second, if we choose the source tokens from left-to-right, the resulting set might not have the maximum joint translation probabilities for the entire phrase. In short, we need an appropriate process to translate the source tokens for efficiency and highest total translation probabilities.

2. Choosing the first source token is crucial. To achieve that, we identify the *pivotal target token* t as the one that is mapped to the most tokens in the input phrase S. It is most likely that t is relevant to the query. For example, in the query containing the tokens *"open", "contents", "file"*, the pivotal target token mapped to *"file"* as well as to *"open"* and *"contents"* could be FileInputStream.open or FileOutputStream.open. We then map the pivotal target token back to the phrase S using the token-to-token mapping to identify the *pivotal source token* s.

3. We consider the context among source tokens in S. At each step, we select a source token s for translation based on its dependence upon the already-translated ones. A direct solution is to utilize the pairwise occurrence of s and a translated token s' within texts in the training data. Specifically, s must have the highest relative co-occurrence frequency with all of the already-translated source tokens: $score_S(s) = \frac{1}{|Q|}\sum_{s' \in Q} \frac{N(s,s')}{N(s')}$ where Q is the set of already-translated tokens, $N(s, s')$ is the number of phrases that s and s' co-occur in the training corpus, and $N(s')$ is the number of phrases having s'. The idea is that *the co-occurring tokens s and s' are likely part of a meaningful text,* e.g.,"open" "file".

4. We also consider the context among the target tokens in T. The next target token to be chosen must have the highest relative co-occurrence frequency with all the target tokens that were already collected: $score_T(T) = \frac{1}{|T|}\sum_{t' \in T} \frac{N(t,t')}{N(t')}$ where T is the set of currently selected target tokens, $N(t, t')$ is the number of times t and t' co-occur in a set of target tokens for a description in the training corpus, and $N(t')$ is the number of times t' occurs. The idea is that t and t' going together often indicates a relation in the target set. The next target token must be among the most likely mapped tokens (*i.e.,* with highest mapping scores) for the currently translated token s according to the token-to-token mapping probabilities. For N, in S, we count source *phrases*, however, in T, we count target *sets*.

B. Model Formulation

1) Notations: Remind that S is the **sequence** of source tokens in the given input and T is the **set** of target tokens. We formulate a probability model in which the target set is

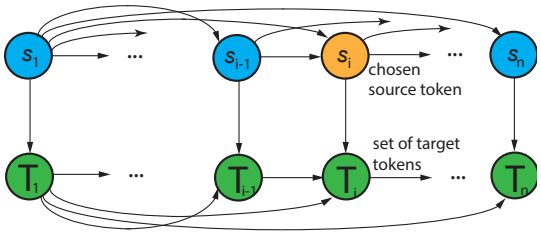

Figure 4: Bayesian Network for Phrase-to-Set Translation

conditional on given sequence. Formally, we aim to find a subset T^* of the superset of all target tokens that maximizes the translation probability $P(T|S)$. This conditional probability is

$$P(T|S) = \frac{P(T,S)}{\sum_T P(T,S)}$$

We focus on the joint probability $P(T, S)$ in the numerator because the marginal probability over S does not depend on T.

2) Formulation: In Figure 4, let us use $s_1, s_2, ..., s_n$ to denote **the sequence** of all tokens in the source phrase S, and T_i to represent the **set of target tokens** translated from each token s_i. At the i^{th} step for selecting a source token for translation, we focus on the $(i-1)$ source tokens that were already-translated. Let us call this set S_{i-1}. We do not need to follow the order of source tokens in the input. Thus, S_{i-1} is not necessarily $\{s_1, ..., s_{i-1}\}$. For each s_i, we use the individual token-to-token mapping algorithm in Section III to compute for s_i, a set T_i of most likely target tokens.

At the i^{th} step for selecting a source token for translation, we define $T_{<i} = \langle T_1, T_2, .., T_{i-1}\rangle$. $T_{<i}$ is a set-valued vector with $(i-1)$ dimensions. This random vector corresponds to the translated sets for the first $(i-1)$ chosen source tokens.

The union of the vector components is statistically sufficient for inferring pairwise occurrence frequency. Thus, we define $T_{<i} = \cup_{j<i} T_j$ (*i.e.,* the union set of the translated sets of target tokens). We adopt the computation from APITran [17] and leverage Bayes rule to compute $P(T|S)$ as follows:

$$\begin{aligned} &P(T,S) = P(T_1, T_2, ..., T_n, s_1, s_2, ..., s_n) \\ &= P(T_1, s_1)P(T_2, s_2|T_1, s_1)... \\ &P(T_n, s_n|T_1, T_2, ..., T_{n-1}, s_1, s_2, ..., s_{n-1}) \\ &= \prod_i P(T_i, s_i|T_{<i}, S_{i-1}) \end{aligned} \tag{3}$$

We aim to find the optimal order of selecting the source tokens in the input sequence $s_1, s_2, ..., s_n$ that maximizes the translation probability $P(T|S)$, i.e., $P(T, S)$. However, considering all possible orderings of the source tokens in S and all combinations of target tokens to compute the probability will result in a combinatorial explosion. Thus, we adopt APITran [17] in the computation in Figure 4. The idea is that the translation is optimized at every i^{th} step where we find a token s_i and a translated set T_i from s_i, given the previous resulting target sets and already-translated tokens. Consequently, we maximize stepwise each factor in the product in (3):

$$T_i^* = argmax_{T_i} P(T_i, s_i|T^*_{<i}, S_{i-1}), \tag{4}$$

where T_i^* is the translated set of target tokens at i^{th} step; s_i is the translated token. We can break down the right-hand side of Equation (4) with the following independence assumptions: i) T_i and S_{i-1} are conditionally independent given $T^*_{<i}$ and s_i, ii) s_i is independent of $T^*_{<i}$ given S_{i-1}.

$$P(T_i, s_i|T^*_{<i}, S_{i-1}) = P(T_i|T^*_{<i}, s_i, S_{i-1}) * P(s_i|T^*_{<i}, S_{i-1}) = P(T_i|T^*_{<i}, s_i) * P(s_i|S_{i-1}) \quad (5)$$

Next, let us explain how we compute Equation 5:

Step 1. To compute $P(T_i|T^*_{<i}, s_i)$ in Equation 5, we build a weighting function $f(T_i, T^*_{i-1}, s_i)$, convert it to probability by exponentiating and normalizing over all possible subsets translated by s_i:

$$P(T_i|T^*_{<i}, s_i) = \frac{\exp f(T_i, T^*_{i-1}, t_i)}{\sum_{T'_i} \exp f(T'_i, T^*_{i-1}, s_i)} \quad (6)$$

$$f(T_i, T^*_{i-1}, s_i) \equiv \sum_{t \in T_i} score_{T^*_{<i}}(t) * P(t|s_i), \quad (7)$$

where $score_T$ is defined in Section IV-A while $P(t|s_i)$ is the lexical probability obtained from token-to-token mapping probabilities. This formula is the formal treatment of our key idea #4 in Section IV-A in the sense that it guarantees to find T_i with high translation probability and high relative co-occurrence frequency with the target tokens in $T^*_{<i}$.

Step 2. $P(s_i|S_{i-1})$ in Equation 5 (key idea #3) is computed as

$$P(s_i|S_{i-1}) = \frac{\exp score_{S_{i-1}}(s_i)}{\sum_{s'_i \in S \setminus S_{i-1}} \exp score_{S_{i-1}}(s'_i)} \quad (8)$$

Finally, the *relevance score* for each target token t in translated set T is computed at each step i by

$$score(t, S) \equiv P(\{t\}, s_i|T^*_{<i}, S_{i-1}) \quad (9)$$

V. Target-Tokens Inferring Algorithm

In Figure 5, we illustrate our algorithm to infer the set of target tokens. It takes as input the token-to-token mapping model (with the scores of all single mappings), and a phrase S consisting of n tokens. We first select the pivotal token and corresponding target token (lines 2 and 8). To do so, we make use of the mapping model to find for each source token s_i, a ranked list of target tokens (line 10). We then identify the pivotal target token t that is mapped to as many source tokens in S as possible (line 11). The pivotal source token s is the one with highest mapping score with t (line 12). Next, based on the pivotal source and target tokens, we include other target tokens (lines 5 and 15). To search for the next token to be translated, we process the remaining tokens in the phrase S, finding token s that has the highest co-occurrence score $score_S(s)$ with the set of already-translated tokens (line 17), which initially contains only the pivotal token. From the selected token s to be translated, we find the best target tokens to be included in the resulting set by considering the contexts on both the source and target sides (*FindBestTargetTokensInContexts*, not shown).

Formula 6 is considered as the formal objective function for *FindBestTargetTokensInContexts*. This step also assigns

```
1  function InferSetOfTargetTokens(Phrase S, TokenMappingModel M)
2    (SourceToken s, TargetToken t) = ChoosePivotMapping();
3    Q = {s} // set of already−translated source tokens
4    T = {t} // set of output target tokens
5    ExpandTargetTokens();
6    return T
7
8  function ChoosePivotMapping()
9    foreach term s_i in S
10   T_i = M(s_i)//top−K list of target tokens for s_i from TokenMappingModel
11   Find a target token t ∈ ∪T_i mapped to the most source tokens in S
12   Find a source token s ∈ S having highest mapping score with t
13   return (s, t)
14
15 function ExpandTargetTokens()
16   while (S != ∅)
17     Find s ∈ S with highest relative co−occur score with translated toks in Q
18     T' = FindBestTargetTokensInContexts(s, T) // using Formula 5-9
19     T = T ∪ T', Q = Q ∪ {s}
20     S = S \ {s}
21   return T
```

Figure 5: Target-Tokens Inference Algorithm

a relevance score for the selected target token according to Formula 9. Specifically, it first uses the mapping model to find the target tokens with highest mapping scores with s. Among them, it selects the elements with highest scores $score_T$, *i.e.,* highest relative co-occurrence frequency with all target tokens that were already selected in the resulting set. For example, if the current selected token s=*"file"*, which can be mapped to either FileInputStream.close or FileOutStream.close. However, the current set of already-derived target tokens contains FileInputStream.new. Thus, we chose FileInputStream.close since FileInputStream.new and FileInputStream.close co-occurs often in the training set. We repeat until all the remaining source tokens are covered (line 20). Finally, we derive all target tokens (line 6).

If the phrase S contains API elements, we treat them as pivotal target tokens, and use them to identify the pivotal tokens before the expansion starts. Thus, the algorithm remains the same except *ChoosePivotMapping()* is changed as explained. Our design strategy aims to favor the coverage of target tokens while maintaining a reasonably low number of them.

VI. Empirical Evaluation

We evaluate Phrase2Set's accuracy and usefulness in three applications and compare it with the existing approaches.

1) [**StackOverflow application**]: tagging SO entries with the API elements relevant to the entries.

2) [**Tutorial application**]: producing the set of API elements relevant to a given fragment of texts in textual tutorial.

3) [**Text-to-API application**]: producing the set of API elements that are used to realize a task described in the input.

We seek to answer the following research questions:

RQ1. How accurate is Phrase2Set in tagging SO entries with relevant API elements? How is it compared to the state-of-the-art *sequence-to-sequence* translation in *seq2seq* [15]?

RQ2. How accurate is Phrase2Set in creating API elements relevant to a text fragment in tutorials? How is it compared to the unsupervised learning approach, FRAPT [5]?

RQ3. How accurate is it in deriving API elements for the task described in a given text? How is it compared to *sequence-to-sequence-based* SWIM [1] and *RNN-based* DeepAPI [3]?

RQ4. What are Phrase2Set's time and space complexity?

Table I: StackOverflow (SO) Dataset

Number of entries	236,919
Avg. number of words per entry	132
Size of word dictionary	701,781
Size of API element dictionary	11,834
Avg. number of embedded API elements per entry	9.2

(1) When you're writing an application in which you would like to perform specialized drawing and/or control the animation of graphics, you should do so by drawing through a **Canvas**.
(2) A **Canvas** works for you as a pretense, or interface, to the actual surface upon which your graphics will be drawn.
(3) It holds all of your "draw" calls.
(4) Via the **Canvas**, your drawing is actually performed upon an underlying **Bitmap**, which is placed into the window.
(5) In the event that you're drawing within the **onDraw()** callback method, the **Canvas** is provided for you...
(6) You can also acquire a **Canvas** from **SurfaceHolder**.**lockCanvas()**, when dealing with a **SurfaceView**.
(7) If you need to create a new **Canvas**, then you must define the **Bitmap** upon which drawing will actually be performed.
(8) The **Bitmap** is always required for a **Canvas**.
(9) You can set up a new **Canvas**...

Figure 6: A Text Fragment in Android Graphics Tutorial [6]

A. Data Collection

For the SO application, we used the StackOverflow dataset from prior research [2] (Table I). The authors built the dataset with the combination of the mining tool ACE [18] and manual inspection. The dataset contains the ground truth of the API elements as the tags for the SO entries. An entry contains the texts of the *title* and *question*, however, the code snippets (if any) and answers were removed. There are SO entries that have only texts or mixtures of texts and API elements embedded in texts. As seen in Table I, the dataset contains very large numbers of entries, words, and API elements.

For the tutorial application [5], [19], [20], let us explain the context. A fragment of text in a tutorial might contain the names of the other "related" APIs, but not explanatory information on API usages [5]. Non-explanatory sentences contain the APIs for an overview of an entire API class or an enumeration of related APIs. It is crucial to have an automated technique to tag a given fragment of texts in a tutorial with the relevant API elements [5], [19], [20]. This helps developers in learning to properly use the API elements.

Figure 6 shows an example of a fragment in Android Graphics Tutorial. Four APIs appear in the fragment: Canvas, Bitmap, SurfaceHolder, and SurfaceView. According to the manual annotation [5], Canvas and Bitmap are relevant to this fragment, whereas SurfaceHolder and SurfaceView are not since the fragment is not about those classes. Thus, a naive approach of collecting all the APIs appearing the texts would not work.

We compare PHRASE2SET with the state-of-the-art unsupervised approach FRAPT [5]. We used the same dataset as in FRAPT [5] (Table II). We parsed the code and Javadoc to produce the text fragments for the APIs in those libraries.

For text-to-API application, we trained the models on the SO dataset in RQ1, and conducted another experiment on the same dataset of 30 queries used in DeepAPI [3] and in SWIM [1] (Table V). These queries used as a testing data do not appear in the training set as a whole.

Table II: Library Tutorial Dataset [5], [6]

Library Tutorial	#APIs	Explan. Fragment	Non-Explan. Fragment	Fragments with APIs	Fragments w/o APIs
Joda Time	36	19	10	21	8
Math Lib	73	31	10	16	25
Col. Official	59	31	26	17	40
Col. Jenkov	28	34	35	42	27
Smack	40	42	5	31	16

B. Procedure and Metrics

1) SO Application: From the SO dataset, we randomly selected 10K samples (about 10%) for testing, and used the remaining samples for training. After running the token-to-token mapping algorithm, each source token is mapped on average to 14.6 API elements. This number shows that this step is helpful since it allows API inferring algorithm to consider a number of potential API elements much smaller than the size of API element dictionary (11,834). A small number of API elements for each source token also helps in reducing noises and making our algorithm scalable.

We compared the inferred sets of API elements against the sets of API elements that were used as tags for the SO entries. We measured Recall and Precision. Recall is defined as the ratio between the number of elements that appear in both the actual and inferred sets of elements and the number of actual elements. Precision is the ratio between the number of elements that appear in both the actual and inferred sets of elements and the number of inferred elements. We also calculated the harmonic value $\text{F-score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$.

2) Tutorial Application: For the tutorial application, we compared PHRASE2SET against FRAPT [5]. FRAPT has two modules: 1) fragmenting the texts of the tutorials, and 2) searching the relevant API elements for the fragments. We compared our tool with the searching module, thus, we used the fragments provided as part of the ground truth in FRAPT dataset [5]. We used FRAPT and PHRASE2SET to rank the fragments and measured accuracy.

For comparison, we used the same setting, dataset, oracle, parameters, and metrics as in FRAPT [5]. We split the data at 90% for training and 10% for testing. Specifically, precision is the ratio between the number of correctly predicted relevant *fragment-API pairs* over all the retrieved pairs. Recall is the ratio between the number of the correctly predicted relevant fragment-API pairs over all the pairs. F-score is the harmonic mean between precision and recall.

3) Text-to-API Application: We ran our tool, SWIM [1] (sequence-to-sequence SMT), and DeepAPI [3] (with RNN Encoder-Decoder) on the 30 queries in DeepAPI paper. Similar to [3], we computed two metrics: 1) FRank (FR) is the rank of the first relevant/useful result in the returned list; and 2)

Table III: Accuracy in Tagging StackOverflow Entries with API Elements

	Top-1		Top-2		Top-3		Top-4		Top-5	
	Base	PHRASE2SET	Base	PHRASE2SET	Base	PHRASE2SET	Base	PHRASE2SET	Base	PHRASE2SET
Recall	44.5	72.9	60.3	89.3	66.3	97.1	74.8	97.8	79.8	98.7
Precision	68.2	77.8	60.7	74.3	53.9	70.2	46.4	67.3	43.2	65.9
F-score	53.7	75.3	60.5	81.1	59.5	81.5	57.3	79.7	56.1	79.0
#API Elements	4.7	6.6	8.2	11.2	12.3	15.8	16.6	19.7	19.3	23.6

Relevant ratio (RR) is the ratio between the number of relevant results over all considered results. Both SWIM and DeepAPI produce the result in term of a sequence. SWIM uses statistical alignment with IBM Model and heuristics, while DeepAPI uses RNN Encoder-Decoder. The ground truth result is given in their paper. We used the pre-trained SWIM and DeepAPI models, while we trained our model on the SO dataset [2].

VII. EMPIRICAL RESULTS ON STACKOVERFLOW APPLICATION

Table III shows the accuracy of PHRASE2SET when the value of K was varied, i.e., the top-K API elements with the highest scores for each source token (line 10, Figure 5). Columns Base are for the baseline *seq2seq* model [8].

In Table III, our algorithm achieves higher accuracy than *seq2seq*. Recall is from 72.9–98.7%. With K=3 (each token has 3 corresponding API elements), we can cover 97.1% of the correct API elements with 15.8 elements for each entry (in the SO dataset, each entry has on average 9.2 elements). With K=5, we will have a total of 23 API elements being inferred, and we can cover almost all correct API elements (98.7% recall). Our precision is also reasonable from 65.9%–77.8%. In other words, among an average of 9.2 elements in a post, we can precisely predict 7–8 elements.

We also aim to measure how well our model can handle the input in two different scenarios: the input contains only texts, and the input contains a mixture of text and code elements (e.g., *"how to write to a file with* FileOutputStream"). Figure 7 shows the distribution of precision and recall for the API elements for each entry through violin plots. A violin plot combines a boxplot and a kernel density plot (shown vertically). The boxplot is represented as the box in the middle of the plot. The bottom of the box is the 25th percentile and the top is the 75th. The horizontal line is the median. The left side of each violin plot is precision, and the right side is for recall. We executed on two kinds of input: pure text and the text mixed with API elements (i.e., embedded code). Note that, in both cases, code snippets are not included as the input.

In Figure 7, the shape of the violins shows the skewness toward high precision and recall. For the inputs mixed of texts and code, for over 3/4 of the cases, PHRASE2SET achieves +80% recall, and over 1/2 of the cases, it achieves +98%. It can also achieves high precision: over 25% of the cases, it has +98% precision. The median precision is 66%. In the context of this problem, if an SO entry is relevant to a median of 10 API elements, PHRASE2SET produces almost all of the APIs. On average, it correctly infers 6.6 API elements and users would remove +3 elements. Note that removing a few

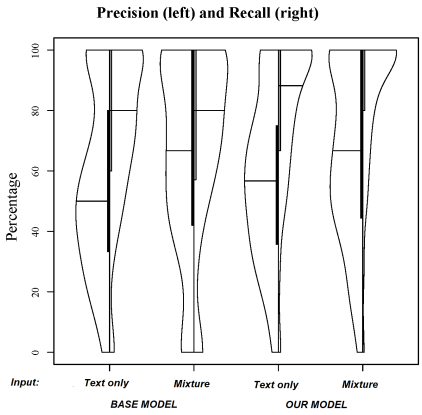

Figure 7: Precision and Recall Distributions for Test Data

Question 22590206

Title: Display dates in multiple fragments

I'm having trouble trying to think of a way to display dates in fragments. What I need to do is display today's date at the top of a fragment and then when the user slides or presses a button a fragment will replace the previous one and display tomorrows date then the next fragment will display the date after tomorrow and so on.

Calendar.new, Calendar.getInstance(..), Calendar.add(..)
Calendar.getTime(..), Date.new, String.format (..)

Figure 8: StackOverflow Entry 22590206 and APIs as Tag

incorrect API elements in the result will cost less effort for users than searching for additional API elements. For the 25% of the cases with low accuracy, users just need to find one additional API element. The accumulated average precision and recall are 64.9% and 94.1% (not shown). PHRASE2SET outperforms *seq2seq* in recall, up to 20% points higher.

The SO entry in Figure 8 is about displaying the dates in multiple fragments in Android applications. Using the input words, PHRASE2SET derives the relevant API elements including the classes Calendar and Date. Note that the *word "calendar" is not mentioned in the text*. However, the class Calendar and its API elements *appear together in multiple tags with Date*, thus, via the target context, PHRASE2SET is able to suggest the API elements of the class Calendar.

In this application, the output is a set of API elements as a tag that do not need to have a strict order. During training, two samples with the same output set, *e.g.,* T=$\{A, B, C\}$, might be

encoded by *seq2seq* for learning as two different sequences, *e.g.,* T_1={A, B, C} or T_2={C, B, A}. This affects the quality of the trained model because *seq2seq* cannot learn that the two samples are the same, thus cannot learn from one sample to apply to the other. An alternative is to enforce a specific order on the samples, *e.g.,* alphabetical order. However, we have to consider all different possible orders for the tokens in a set $\{t_1, t_2, ..., t_n\}$ because the quality of the trained model for every order might be different, *e.g.,* the alphabetical order might not result in a better model than the reverse order.

VIII. Empirical Results on Tutorial Application

Table IV shows the comparison result on the tutorial application. As seen, on average, PHRASE2SET (P2S) achieves 83.2% in precision, 94.5% in recall, and 88.1% in F-score. In comparison, it relatively improves over FRAPT 12.2% in precision, 10% in recall, and 11% in F-score. The largest relative improvements are 25.5% in precision, 35.9% in recall, and 21.9% in F-score. Since PHRASE2SET takes into consideration both of the contexts in source and target sides and it does not restrict the order, it can apply to learn from one instance to another, leading to such improvements.

Table IV: Accuracy in Tutorial Application

	Precision (%)		**Recall (%)**		**F-score (%)**	
Tutorial	FRAPT	P2S	FRAPT	P2S	FRAPT	P2S
Joda Time	85.1	87.2	76.7	100	80.7	90.8
Math Lib	84.8	92.8	73.6	100	78.9	96.2
Col. Official	62.0	74.3	87.5	74.3	72.6	74.3
Col. Jenkov	61.2	67.7	97.6	98.2	75.2	80.1
Smack	77.9	97.8	94.6	100	85.5	98.9
Average	74.2	**83.2**	86.0	**94.5**	78.6	**88.1**

We further performed overlapping analysis between the results from PHRASE2SET and FRAPT. We found that there are 23.5% of the fragment-API pairs that PHRASE2SET correctly detected but FRAPT did not (not shown). Meanwhile, FRAPT correctly detected only 13.7% of the cases that our tool did not. In conclusion, while both tools complement to each other, PHRASE2SET achieves higher performance than FRAFT.

IX. Empirical Results on Text-to-API Application

Table V shows the result for the comparative study on the text-to-API application. As seen, PHRASE2SET performs better than DeepAPI in terms of FR and RR5, while both perform better than SWIM. To compare PHRASE2SET and SWIM, we applied the Wilcoxon signed-rank test. The resulting p-value is 0.01, *i.e.,* smaller than 0.05, indicating that the differences are statistically significant.

On average, PHRASE2SET ranks the first relevant API at the position 1.2, while the other tools rank it at 4.2 and 1.6. In the list of five candidate APIs, 84% of them on average are relevant to the query. Thus, it produces mostly relevant APIs. Among 30 results, 24 are deemed to be relevant (80%). DeepAPI achieves usefulness/relevance for 23 top-ranked resulting APIs (77%). For those 30 queries, SWIM's top-ranked resulting API sequences are relevant in only 14 of them (47%).

Table V: Relevance Evaluation (FR: Rank of 1st relevant one, RR5 (%): top-5 relevance ratio, –: no relevance in top 10)

	SWIM [1]		DeepAPI [3]		**PHRASE2SET**	
Textual Query	FR	RR5	FR	RR5	FR	RR5
convert int to string	8	0	2	40	1	100
convert string to int	1	80	1	100	1	100
append strings	3	60	1	100	2	40
get current time	1	80	10	10	2	20
parse datetime from string	9	0	1	100	2	40
test file exists	–	0	1	100	1	100
open a url	1	100	1	100	1	100
open file dialog	–	0	1	100	1	100
get files in folder	2	40	3	40	1	100
match regular expressions	1	100	1	80	1	100
generate md5 hash code	1	60	1	100	1	100
generate random number	7	0	1	100	1	100
round a decimal value	–	0	1	100	2	40
execute sql statement	2	80	1	80	1	100
connect to database	7	0	1	100	1	100
create file	10	0	3	40	2	60
copy file	1	100	2	20	1	100
copy a file and save it to your dest. path	1	20	1	100	1	100
delete files and folders in a dir	1	100	1	100	1	100
reverse a string	3	20	2	60	2	40
create socket	–	0	1	60	1	100
rename a file	–	0	1	100	1	100
download file from url	2	60	1	100	1	100
serialize an object	1	100	3	60	1	100
read binary file	4	40	1	100	3	40
save an image to a file	1	20	1	80	1	100
write an image to a file	1	20	1	100	1	40
parse xml	1	100	1	80	1	100
play audio	1	100	1	60	1	100
play the audio clip at specified absolute URL	1	40	1	100	1	100
Average	**>4.2**	**44**	**1.6**	**80.3**	**1.2**	**84**

We also investigated the capability of the tools in handling *longer text inputs*. We used 250 queries in the prior work, T2API [2]. Each query has from 25 to 131 tokens. In the results from PHRASE2SET, there are up to 23 APIs. In the results from SWIM [1], the number of generated API elements is from 1–3. For DeepAPI, their sequences are a list of elements. Moreover, when we entered those 250 queries into DeepAPI online tool [21], it took several minutes (due to its high computation) and most of the results were irrelevant.

X. Discussion

A. Limitations

PHRASE2SET currently has the following limitations. First, as in any MT approaches, high-quality training data is crucial. In NLP, where the corpora of parallel texts in two languages have been (semi-)automatically or manually built with human annotations and verification. We depend on large corpora. Less common APIs might not fit with this approach. Our corpora used in experiments were shared by the authors who have spent effort to mine via an automated tool and human verification.

Accuracy could be improved much if we can integrate NLP techniques to process the semantics of the texts. At the same time, program analyses on the source code could also be integrated to adjust the generation process of the elements. Currently, there is no semantic analysis on both sides. The rule-based approaches [22] that were successfully used in code migration could be explored. For our tool, there are extra code elements that the tool found were common but may not be

Table VI: Time and Space Complexity

Storage	3.6GBs/+100K texts
Training time	12hrs/+100K texts
Suggestion time	0.1 seconds/input

relevant to the input. Most of incorrect cases are caused by the out-of-vocabulary issue (un-seen API elements in training).

B. Time and Space Complexity

Table VI shows PHRASE2SET's complexity (measured on a computer with AMD Phenom II X6 3.2GHz, 16GB RAM, and Linux). Training time is quite extensive. However, one can train PHRASE2SET offline for suggestion later. The suggestion time for a query is only 0.1s. Storage cost is reasonable.

C. Threats to Validity

We cannot intrinsically evaluate our model via BLUE score, because the metric is defined for the phrase-based translation. We chose to extrinsically evaluate PHRASE2SET in three SE applications, in which the parallel corpora are available. We do not have a general-purpose parallel corpus in SE as in NLP. We need to evaluate PHRASE2SET in other SE applications as well, especially for code-to-text and text-to-text ones. The performance was measured without the tool's usefulness, which might require a human study. Our collected data set might not be representative. There is possible construct bias as we chose Java and Android APIs.

D. Other Potential Applications

The fundamental contribution of the work is the phrase-to-set statistical machine translation model, PHRASE2SET, which arises from the need of SE applications. The model does not just grab embedded APIs relevant to the output, it can derive the API elements that it learns from the dataset and did not appear in the given text (see tutorial application). It also works well for the input of pure texts. PHRASE2SET can potentially be used in other SE applications in which the output of text/code elements does not have a strict order.

1) **Bug Localization (Text-to-Code)***:* The bug localization (BL) problem can be formulated as follows. Given as input a bug report with *a mixture of text and code*, a BL approach needs to locate a *set of methods* in a project that are potentially to be fixed for the bug(s) reported in the input text. To use PHRASE2SET for BL, we can train it with a corpus consisting of the pairs of a bug report and the corresponding set of the methods that were fixed for the bug. For prediction, given a bug report, the trained phrase-to-set model can be used to suggest the set of methods for a new given bug report.

2) **Recovering Links from Release Notes to Code Changes (Text-to-Code)***:* A release note often lists the new features and functionality in a software. However, there is no record to connect enhancements to the specific code changes in the repositories. To use PHRASE2SET for this task, one can train it with the dataset of the known release notes and corresponding code changes, and then use it to derive a *set of methods/classes that were changed* for a given release note.

3) **Requirement Tracing to Source Code (Text-to-Code)***:* Requirement tracing [23] is an SE problem in which a sentence or paragraph in the textual requirement document is traced to the corresponding set of classes/methods implementing that requirement. To use our model, one can train it with a parallel corpus of the texts in the requirement and the *set of methods* realizing the features described in the texts, then use it to derive the set of methods for a given textual requirement.

4) **Creating "See Also" in API Documentation (Text-to-Code)***:* The API documentation could include *the set of relevant API elements* to the currently described API element. To use our model, one can train it on a parallel corpus of the pairs of API documentation and *the set of the corresponding relevant API elements*. For prediction, we can run the trained model on the new API documentation (as mixture of code/-texts) to produce the set of relevant API elements.

5) **Recovering the Links from Commit Logs to Bug Reports (Text-to-Text)***:* For the bug fixes, the changes and commit logs, which are stored in a version control repository, are not connected to the bug reports that describe the bug(s) because the bug reports are often stored in the issue repository. SE researchers have designed approaches to link a commit log to the corresponding bug report [24], [25]. To use our model, one could train it with the dataset consisting of the pairs of the commit log (text) and the corresponding bug report (text).

6) **Tagging Bug Reports with Topical Keywords (Text-to-Text)***:* It is useful to tag a bug report with *the set of relevant topics* such as security, performance, privacy, vulnerability, etc. To use PHRASE2SET, we can train the model by a parallel corpus of bug reports and the corresponding topical keywords as the tags. We then use the trained model for tagging.

7) **Connecting Source Code to the Set of High-level Functionality Keywords (Code-to-Text)***:* A software system has a set of high-level functionality. For tracing purpose, it is useful to tag each method in a project with a set of keywords describing high-level functionality. One can use our model since the output of keywords needs not have a strict order.

8) **Linking Code Changes to Keywords for System Features (Code-to-Text)***:* A useful tool is to link the committed code changes to the keywords describing the new features that were implemented by those changes. The output set of keywords does not have a strict order. The model can be trained by the parallel corpus of the committed changes and the corresponding *set of keywords for system features*.

9) **API Migration (Code-to-API)***:* Given a code with the API method calls, field accesses, and class usages, API migration aims to derive the set of API elements in another program language that can be used to implement the same functionality as the original code. In this problem, the output is a set of APIs that do not need to have a strict order. One could train PHRASE2SET on a parallel corpus of the code in a language, e.g., Java, and the corresponding sets of APIs in another language, e.g., C#. For migration, given a Java code, the trained model derives the set of API elements in C# to be used to realize the same functionality in the Java code.

10) **Exception Handling Suggestion (Code-to-Code)***:* Given a method, a tool could suggest a set of Exceptions that need to be handled due to the usage of certain API elements in the method. For example, for file opening, a code needs to catch the Exception on FileNotFoundException. To use PHRASE2SET, one can train it on the corpus of the pairs of the source code and the exceptions that were handled. For a new given method, the trained model can suggest the set of Exceptions that need to be handled. In this case, the output is *the set of Exceptions* without a strict order among them.

XI. RELATED WORK

PHRASE2SET is related to APITran [17], which generates the API elements relevant to a given textual query in English. Similar to PHRASE2SET, APITran is also based on Bayesian Network and during translation, the contexts of both texts and code are considered. However, there are key advances of PHRASE2SET over APITran. First, APITran supports only API code elements in the target side because it relies on the individual mappings between textual tokens to API code tokens. In PHRASE2SET, we develop a general token-to-token mapping algorithm in which the occurrence-counts are based on the set of text and/or code tokens, rather than API elements. Second, APITran's formulation is defined for sequence-based output. When estimating the probabilities by counting, APITran performs occurrence-counts on the sequences. PHRASE2SET's formulation is extended from APITran to support general target tokens including text and code tokens, and support the *output set* (instead of sequences). Third, because the goal of APITran is to derive the list of API elements relevant to English texts, it does not have the inferring algorithm to derive the output set. In PHRASE2SET, we also develop the inferring algorithm to collect the target tokens into a set.

PHRASE2SET is also related to T2API [2], which uses GraLan [26], a graph-based language model to derive a set of API elements relevant to a query. While PHRASE2SET produces a set of target tokens, T2API uses GraSyn [2], a graph-based synthesis algorithm to produce an output graph of API elements. PHRASE2SET's individual token-to-token mapping algorithm is extended from the API mapping algorithm in StaMiner [16], which focuses only on mappings a sequence of API elements in Java to another sequence of elements in C#. In PHRASE2SET, the occurrence-counts for probabilities are based on set, rather than on sequences as in StaMiner.

SWIM [1] translates a sequence of texts into a code sequence of API elements. SWIM uses IBM Model for word-to-API single mappings and then uses rules to synthesize code. PHRASE2SET is also related to DeepAPI [3], which uses a neural-network-based machine translation model with RNN from text to APIs. In comparison, DeepAPI uses sequences for both sides. Desai *et al.* [27] synthesize domain-specific languages (DSL) from English. A user is required to map key terms in English to the terminals in the DSL. Nguyen *et al.* [6] enhance Word2Vec to represent API in API documentation. They also apply their model in the tutorial application. However, they do not aim to translate phrases to sets.

Buse and Weimer [28] use path sensitive data-flow analysis, clustering, and pattern abstraction to synthesize API usages from code examples. They do not handle *textual queries*. Others explore structure relations [29], call graphs (FACG [30]) and program dependencies (MAPO [31], Altair [32]).

Statistical learning has been used in SE applications. They include code suggestion [33], [34], code convention [35], name suggestion [36], API suggestions [37], large-scale code mining [38], etc. Maddison and Tarlow [39] present a generative model for source code, which is based on AST-based syntactic structures. TBCNN [40] uses tree information for suggest next code tokens. Allamanis *et al.* [41] introduce a jointly probabilistic model short natural language utterances and source code snippets. They want a joint model for both sides, a a tree-based representation is used for code and texts. While their approach uses advanced *bimodal modeling* (e.g., image+text, text+code), we treat code synthesis as a machine translation problem, allowing different language models for texts and source code. Anycode [42] uses a probabilistic context free grammar with trees for Java constructs and API calls to synthesize small Java expressions.

Typical applications for domain-specific code synthesis are string analysis [43], bit vector processing [44], [45], structure manipulation [46], finite programs with sketches [47], [48], spreadsheet transformations [49], geometry constructions [50], and hardware design [51].

Semantic code search engines use IR-based techniques in text matching [52], [53], [54], [29], [55]. Others enhance IDE with code search [56], [57]. Other group of IR-based code search approaches considers the relations among API elements [13], [58], [59], [55], [11], [10], [12].

XII. CONCLUSION

We propose PHRASE2SET, a context-sensitive statistical machine translation model that learns to transform a phrase of code/texts into a set of code/text tokens. We design a Bayesian-Network-based SMT model that determines a translation order for the source tokens maximizing the joint translation probability. As to determine such order, we consider the context of the tokens in the source side and that in the target side via their relative co-occurrence frequencies. Our empirical evaluation was on three applications: 1) tagging the StackOverflow posts with relevant API elements, 2) tagging the fragments of texts in a tutorial with the relevant API elements, and 3) text-to-API application. Our empirical results show that PHRASE2SET achieves high accuracy and outperforms the baseline models in those tasks. We also provide a lesson learned from our experience and a list of potential applications that can benefit from PHRASE2SET. In future, we aim to provide set-to-phrase translation, which might have other SE applications. We also explore other applications as explained in Section X-D.

ACKNOWLEDGMENT

This work was supported in part by the US National Science Foundation (NSF) grants CNS-2120386, CCF-1723215, CCF-1723432, TWC-1723198, CCF-1518897, and CNS-1513263.

REFERENCES

[1] M. Raghothaman, Y. Wei, and Y. Hamadi, "SWIM: Synthesizing What I Mean - Code Search and Idiomatic Snippet Synthesis," in *Proceedings of the 38th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2016, pp. 357–367.

[2] A. T. Nguyen, P. C. Rigby, T. Nguyen, D. Palani, M. Karanfil, and T. N. Nguyen, "Statistical Translation of English Texts to API Code Templates," in *Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution (ICSME'18)*, 2018, pp. 194–205.

[3] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep API Learning," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 631–642. [Online]. Available: https://doi.org/10.1145/2950290.2950334

[4] H. Phan and A. Jannesari, "Statistical machine translation outperforms neural machine translation in software engineering: Why and how," in *Proceedings of the 1st ACM SIGSOFT International Workshop on Representation Learning for Software Engineering and Program Languages*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 3–12. [Online]. Available: https://doi.org/10.1145/3416506.3423576

[5] H. Jiang, J. Zhang, Z. Ren, and T. Zhang, "An unsupervised approach for discovering relevant tutorial fragments for APIs," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE'17. IEEE Press, 2017, p. 38–48. [Online]. Available: https://doi.org/10.1109/ICSE.2017.12

[6] T. Nguyen, N. Tran, H. Phan, T. Nguyen, L. Truong, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Complementing Global and Local Contexts in Representing API Descriptions to Improve API Retrieval Tasks," in *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE'18. Association for Computing Machinery, 2018, pp. 551–562. [Online]. Available: https://doi.org/10.1145/3236024.3236036

[7] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun, "Duplicate bug report detection with a combination of information retrieval and topic modeling," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012. New York, NY, USA: Association for Computing Machinery, 2012, pp. 70–79.

[8] D. Britz, A. Goldie, T. Luong, and Q. Le, "Massive Exploration of Neural Machine Translation Architectures," *ArXiv e-prints*, Mar. 2017.

[9] "Phrase2Set," https://github.com/phrase2set-submission/phrase2set.

[10] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: Finding relevant functions and their usage," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. ACM, 2011, pp. 111–120.

[11] C. McMillan, D. Poshyvanyk, and M. Grechanik, "Recommending Source Code Examples via API Call Usages and Documentation," in *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, ser. RSSE '10. ACM, 2010, pp. 21–25.

[12] W.-K. Chan, H. Cheng, and D. Lo, "Searching Connected API Subgraph via Text Phrases," in *Proceedings of the 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. ACM, 2012, pp. 10:1–10:11.

[13] W. Zheng, Q. Zhang, and M. Lyu, "Cross-library API Recommendation Using Web Search Engines," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. ACM, 2011, pp. 480–483.

[14] E. Duala-Ekoko and M. P. Robillard, "Using structure-based recommendations to facilitate discoverability in APIs," in *Proceedings of the 25th European Conference on Object-oriented Programming*, ser. ECOOP'11. Springer-Verlag, 2011, pp. 79–104.

[15] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. Rush, "OpenNMT: Open-source toolkit for neural machine translation," in *Proceedings of ACL 2017, System Demonstrations*. Vancouver, Canada: Association for Computational Linguistics, Jul. 2017, pp. 67–72. [Online]. Available: https://www.aclweb.org/anthology/P17-4012

[16] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Statistical Learning Approach for Mining API Usage Mappings for Code Migration," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 457–468. [Online]. Available: https://doi.org/10.1145/2642937.2643010

[17] T. V. Nguyen and T. N. Nguyen, "Inferring API Elements Relevant to an English Query," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 167–168. [Online]. Available: https://doi.org/10.1145/3183440.3195079

[18] P. C. Rigby and M. P. Robillard, "Discovering essential code elements in informal documentation," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, pp. 832–841.

[19] H. Jiang, J. Zhang, X. Li, Z. Ren, and D. Lo, "A more accurate model for finding tutorial segments explaining APIs," in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*, vol. 1, March 2016, pp. 157–167.

[20] G. Petrosyan, M. P. Robillard, and R. De Mori, "Discovering information explaining API types using text classification," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. IEEE Press, 2015, pp. 869–879.

[21] "Deep API Learning," http://bda-codehow.cloudapp.net:88/.

[22] "Java2CSharp," http://sourceforge.net/projects/j2cstranslator/.

[23] J. Cleland-Huang, O. C. Z. Gotel, J. Huffman Hayes, P. Mäder, and A. Zisman, "Software traceability: Trends and future directions," in *Future of Software Engineering Proceedings*, ser. FOSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, pp. 55–69. [Online]. Available: https://doi.org/10.1145/2593882.2593891

[24] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Multi-layered approach for recovering links between bug reports and fixes," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2393596.2393671

[25] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein, "The missing links: Bugs and bug-fix commits," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 97–106. [Online]. Available: https://doi.org/10.1145/1882291.1882308

[26] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. IEEE Press, 2015, p. 858–868.

[27] A. Desai, S. Gulwani, V. Hingorani, N. Jain, A. Karkare, M. Marron, S. R, and S. Roy, "Program synthesis using natural language," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. ACM, 2016, pp. 345–356.

[28] R. P. L. Buse and W. Weimer, "Synthesizing API Usage Examples," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. IEEE Press, 2012, pp. 782–792.

[29] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes, "Sourcerer: A search engine for open source code supporting structure-based search," in *Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '06. ACM, 2006, pp. 681–682.

[30] Q. Zhang, W. Zheng, and M. R. Lyu, "Flow-augmented Call Graph: A New Foundation for Taming API Complexity," in *Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering*, ser. FASE'11/ETAPS'11. Springer-Verlag, 2011, pp. 386–400.

[31] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: Mining and Recommending API Usage Patterns," in *Proceedings of the 23rd European Conference on Object-Oriented Programming*. Springer, 2009, pp. 318–343.

[32] F. Long, X. Wang, and Y. Cai, "API Hyperlinking via Structural Overlap," in *Proceedings of the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09. ACM, 2009, pp. 203–212.

[33] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. IEEE Press, 2012, pp. 837–847.

[34] M. White, C. Vendome, M. Linares-Vasquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *Proceedings of 12th IEEE Working Conference on Mining Software Repositories (MSR'15)*. IEEE CS, May 2015.

[35] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proceedings of the International Symposium on Foundations of Software Engineering*, ser. FSE 2014. ACM, 2014, pp. 281–293.

[36] ——, "Suggesting accurate method and class names," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 38–49. [Online]. Available: https://doi.org/10.1145/2786805.2786849

[37] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. ACM, 2014, pp. 419–428.

[38] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13. IEEE Press, 2013, p. 207–216.

[39] C. J. Maddison and D. Tarlow, "Structured generative models of natural source code," in *Proceedings of the 31st International Conference on Machine Learning - Volume 32*, ser. ICML'14. JMLR.org, 2014, p. II–649–II–657.

[40] L. Mou, G. Li, Z. Jin, L. Zhang, and T. Wang, "TBCNN: A tree-based convolutional neural network for programming language processing," *CoRR*, vol. abs/1409.5718, 2014. [Online]. Available: http://arxiv.org/abs/1409.5718

[41] M. Allamanis, D. Tarlow, A. D. Gordon, and Y. Wei, "Bimodal modelling of source code and natural language," in *Proceedings of the 32nd International Conference on Machine Learning - Volume 37*, ser. ICML'15. JMLR.org, 2015, p. 2123–2132.

[42] T. Gvero and V. Kuncak, "Synthesizing java expressions from free-form queries," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2015. New York, NY, USA: ACM, 2015, pp. 416–432. [Online]. Available: http://doi.acm.org/10.1145/2814270.2814295

[43] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '11. New York, NY, USA: ACM, 2011, pp. 317–330. [Online]. Available: http://doi.acm.org/10.1145/1926385.1926423

[44] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ser. ICSE '10. ACM, 2010, pp. 215–224.

[45] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat, "Combinatorial sketching for finite programs," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2006, pp. 404–415.

[46] R. Singh and A. Solar-Lezama, "Synthesizing data structure manipulations from storyboards," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. ACM, 2011, pp. 289–299.

[47] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia, "Sketching stencils," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. ACM, 2007, pp. 167–178.

[48] A. Solar-Lezama, C. G. Jones, and R. Bodik, "Sketching concurrent data structures," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08. ACM, 2008, pp. 136–148.

[49] W. R. Harris and S. Gulwani, "Spreadsheet table transformations from examples," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 317–328. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993536

[50] S. Gulwani, V. A. Korthikanti, and A. Tiwari, "Synthesizing geometry constructions," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. ACM, 2011, pp. 50–61.

[51] A. Raabe and R. Bodik, "Synthesizing hardware from sketches," in *Proceedings of the 46th Annual Design Automation Conference*, ser. DAC '09. ACM, 2009, pp. 623–624.

[52] "Black Duck Open Hub," http://code.openhub.net/.

[53] "Codase," http://www.codase.com/.

[54] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto, "Component rank: Relative significance rank for software component search," in *Proceedings of the 25th International Conference on Software Engineering*, ser. ICSE '03. IEEE, 2003, pp. 14–24.

[55] D. Puppin and F. Silvestri, "The Social Network of Java Classes," in *Proceedings of the 2006 ACM Symposium on Applied Computing*, ser. SAC '06. ACM, 2006, pp. 1409–1413.

[56] N. Sawadsky, G. C. Murphy, and R. Jiresal, "Reverb: Recommending Code-related Web Pages," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, pp. 812–821.

[57] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer, "Example-centric Programming: Integrating Web Search into the Development Environment," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '10. ACM, 2010, pp. 513–522.

[58] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, "A search engine for finding highly relevant applications," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. ACM, 2010, pp. 475–484.

[59] Z. M. Saul, V. Filkov, P. Devanbu, and C. Bird, "Recommending random walks," in *Proceedings of the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07. ACM, 2007, pp. 15–24.