Applying Deep Learning to C# Call Sequence Synthesis

A.E. Chebykin <a.e.chebykin@gmail.com>
I.A. Kirilenko <jake.kirilenko@gmail.com>
Faculty of Mathematics and Mechanics, Saint Petersburg State University
Universitetsky prospekt, 28, Peterhof, St. Petersburg, 198504, Russia

Abstract. Many common programming tasks, like connecting to a database, drawing an image, or reading from a file, are long implemented in various frameworks and are available via corresponding Application Programming Interfaces (APIs). However, to use them, a software engineer must first learn of their existence and then of the correct way to utilize them. Currently, the Internet seems to be the best and the most common way to gather such information. Recently, a deep-learning-based solution was proposed in the form of DeepAPI tool. Given English description of the desired functionality, sequence of Java function calls is generated. In this paper, we show the way to apply this approach to a different programming language (C# over Java) that has smaller open code base; we describe techniques used to achieve results close to the original, as well as techniques that failed to produce an impact. Finally, we release our dataset, code and trained model to facilitate further research.

Keywords: API; deep learning; code search; RNN; transfer learning.

DOI: 10.15514/ISPRAS-2018-30(3)-5

For citation: Chebykin A.E., Kirilenko I.A. Applying Deep Learning to C# Call Sequence Synthesis. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 3, 2018, pp. 63-86. DOI: 10.15514/ISPRAS-2018-30(3)-5

1. Introduction

When writing code, software developers often utilize various libraries via APIs. Since the problems being solved in this manner are usually similar for most users, their solutions form stable patterns of API invocations.

API mining is a long-established line of research aimed at extracting these API usage trends from source code. The importance of the task lies in the fact that generally developers spend a lot of time trying to learn frameworks' APIs in order to utilize them efficiently. A field study has found that developers often struggle to map a task from problem domain to the terminology of the API [1]. In another survey 67.6% of respondents identified that learning APIs is hindered by inadequate or absent resources [2].

Usually, when facing such problems, developers turn to general web search engines. However, those are not optimized for programming-related queries and thus tend to be inefficient [3].

An alternative lies in various approaches based on statistical analysis of source code. They can provide sequences of API methods that are often used together [4], mine API specifications in the form of automata [5], synthesize relevant code snippets [6].

Deep API Learning [7] is a recent deep learning-based take on the problem that reports state-of-the-art results. The authors formulate the problem of providing API patterns satisfying users' needs as a translation one. Input language, in which user describes desired functionality, is English, and the output language is one of API sequences: API calls are words of the language, ordered sequences of these calls form sentences. For example, English sentence "*generate random int*" could be translated to the language of Java API as "*Random.new Random.nextInt"*, which corresponds to the construction of an object of type *Random* and subsequent call of its *nextInt* method.

DeepAPI tool targets exclusively Java programming language and reportedly performs well. Benefits of the approach come from the usage of deep recurrent neural networks. Thanks to them, trained model can distinguish synonyms and impact of word sequence (for example, it can distinguish queries *convert string to int* and *convert int to string*).

However, the authors identify several threats to validity, including possible failure when extending the approach to other programming languages.

Our main goals are to test this threat, thus appraising generality of the approach, and to consider possible improvements. We choose C# as a target language due to its general similarity to Java, aiming to make a first step towards more different — and therefore challenging — target languages.

However even in our case simple copying of DeepAPI approach leads to bad results, and constructing well-working model proves to be far from trivial. In this paper, we describe our experience of extending the proposed approach to C#.

To achieve our goals we collect dataset of 2,886,309 training samples from open source projects' code and use it to first train a model with the architecture of DeepAPI (attaining the result of 10.94 BLEU), and then tune parameters to achieve BLEU 26.26. After that, we introduce data preprocessing, which reduces dataset size to 1,397,597, but improves its quality and increases BLEU metric to 46.99. Finally, we employ transfer learning on an alternative dataset of method names and achieve the best results of 50.14 BLEU, which is fairly close to the 54.42 reported by DeepAPI on Java dataset.

Additionally we ask professional developers to evaluate output of our model on several queries, which shows that on average our model, DeepAPI#, performs as well as DeepAPI.

Our main contributions are:

- reproduction of the DeepAPI experiment with a different dataset;
- modification of the approach via programming-language-independent data preprocessing which leads to results, comparable to original, despite lack of data;
- collection of C# dataset of commented methods and publishing of it for the benefit of the future research in the area;
- employment of transfer learning techniques for additional improvement of the results. To the best of our knowledge, we are the first to investigate transfer learning in the area of API mining.

The paper is organized as follows: in section 2 we outline DeepAPI model architecture. Next, in section 3 collection of the dataset needed for model training is discussed and additional preprocessing steps are introduced. We describe our application of transfer learning to the problem in section 4. Technical details of model training are reported in section 5, which is followed by section 6, where evaluation results are described. We finish the paper with section 7, where we report work done on related problems and discuss ways in which existing research differs from ours.

2. DeepAPI model

We borrow general model structure from DeepAPI, which is itself based on recent advancements in neural machine translation. Here we will provide only an overview, for details please refer to the original paper [7] and our previous research-in-progress paper [8].

Since the goal is to generate one sequence of words based on another, the task falls in the category of Sequence-to-Sequence learning [9]. One of the best architectures for the task is an Encoder-Decoder network [10].

It consists of two recurrent neural networks (Recurrent neural network is a special class of neural networks where unit can be connected to itself, thus allowing its state to serve the role of memory). Encoder network reads input sequence, Decoder generates output one. The process goes as following.

Encoder reads input word by word, embeds each one in a high-dimensional space and sequentially updates its hidden state, which by the end of the sentence contains language-independent idea of the input sentence. This state (also known as context vector) is then passed to the Decoder, which based on it and the last generated word generates words one by one until a special end-of-sequence token is outputted.

An example of such model at work can be seen in Fig. 1. In the image states of networks are rolled out in time, so for example RNN_1, RNN_2, RNN_3 is the RNN state at time steps $1, 2, 3$. Note that Encoder and Decoder consist of different RNNs and work in different time windows: at first, Encoder RNN makes 3 steps in time and then Decoder RNN makes 3 steps in time.

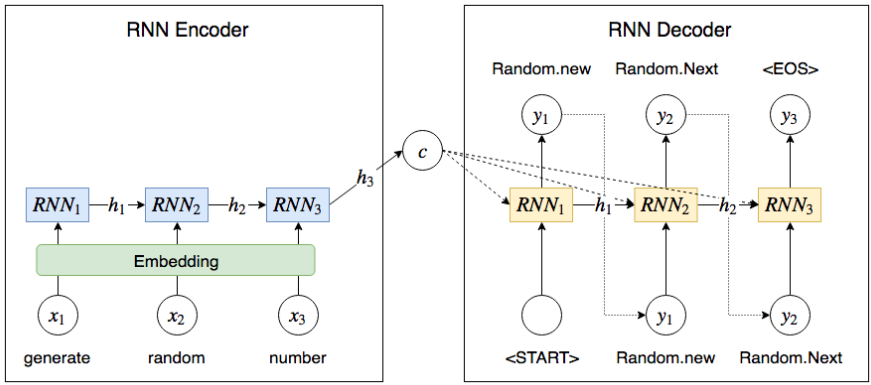

Fig. 1. RNN Encoder-Decoder workflow

The benefits of this model include synonym handling (words used in the similar contexts get embedded near each other), successful processing of long inputs thanks to the memorizing ability of the recurrent networks, and finally appreciation of word sequence impact.

One major downside of such a model is the need for a large amount of sentence pairs describing the same functionality in two languages (*"generate random number"*, "*Random.new Random.Next"*). Format of the API language description is reported in the point 3.1.3.

Source of such data can be methods' documentation comments (that in C# are XML-based and contain summary section, in which brief description of the method's functionality should be supplied) and corresponding API calls made in the method body. Details of the dataset collection are described in section 3.

There are several improvements of the Encoder-Decoder architecture that were shown to reliably improve results.

- Using Bidirectional Encoder leads to input being processed twice: in normal order and in reverse, resulting in 2 context vectors, which are then concatenated to get final context vector [11].
- Attention mechanism [12] allows decoder to focus on different input words when generating different output ones.

In the original DeepAPI paper an additional improvement is introduced in the form of a regularization term punishing generation of the most widespread and therefore probably problem-irrelevant API calls, such as logging ones. We have not tried such regularization since its reported impact on BLEU score is minimal. We leave testing of this enhancement for future research.

3. Dataset

3.1 Dataset collection

To train the model, we need to gather large amount of pairs (English description of functionality, API description). One way to do it is to process open source projects, looking for methods with documentation comments, extracting summary sections and linearizing interesting parts of ASTs (i.e. API calls). The processing of individual methods is described in section 3.1.3.

GitHub[1] is one of the most popular open source project hostings. Following DeepAPI authors, we construct our dataset from data published there.

We attempted to augment GitHub data with data from alternative sources. In our previous paper [8] we proposed using Nuget[2] – a repository of compiled C# packages. However we eventually found out that compared to GitHub it does not provide much data, and what samples it provides often duplicate ones collected from GitHub. So we discontinued using Nuget as data source.

There are other sites with published open source projects, for example, Codeplex[3] and SourceForge[4]. Unfortunately, we found there only a small amount of C# projects, many of which gradually migrate to GitHub, or have already done so. These hosting sites also lack search APIs that are essential for the automatic collection of our dataset. So the potentially small amount of additional data is nontrivial to collect, and therefore we choose to ignore these alternative sources.

We collect dataset from GitHub in several steps:

1) obtain a list of repositories relevant to us;
2) download these repositories;
3) process them, extracting from methods with documentation comments these comments, linearized in a special way API calls, types and names of method parameters.

The architecture overview can be seen in Fig. 2. Let us discuss every step in detail.

3.1.1 Obtaining list of relevant repositories

We are interested in repositories in C# language. Similar to the original paper, we would like to consider only projects that have at least one star in order to filter unused or toy projects. Both these requirements can be satisfied when setting specific parameters of GitHub Search API.

[1] github.com
[2] nugget.org
[3] archive.codeplex.com
[4] sourceforge.net

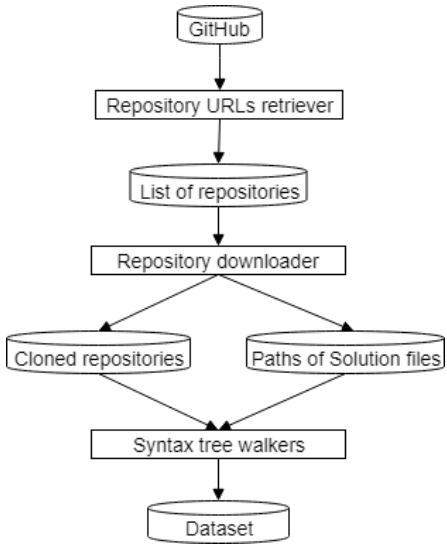

Fig. 2. Dataset gathering workflow

Using this API via Octokit.rb[5] library, we retrieve 140,990 URLs of relevant projects created from 2012 to 2017. This contrasts to the original paper that reports working with 442,928 Java repositories. Therefore, we initially have approximately 3 times less projects to work with. This lack of data can potentially be a significant obstacle when transferring the approach to other languages with smaller open code bases.

Search API also poses several technical difficulties.

Firstly, it returns no more than 1,000 results for any search request. To go around this restriction, we set additional parameter limiting repository creation date to a short span of time, for example, "2016-01-01 .. 2016-01-08". Every our requests covers 8 days, which we find short enough a period that no more than 1,000 repositories are created during it.

Secondly, Search API limits number of requests per minute by 30. In order not to exceed this limit, our script sleeps for 2 seconds after each request.

We store repositories list and the rest of our data in a SQLite database[6].

3.1.2 Downloading repositories

Having gathered repository list, we can start cloning them with git. We set clone depth to 1 to speed up the process.

[5] github.com/octokit/octokit.rb
[6] sqlite.org

After download, we search for solution files — special files that encompass source code files, as well as store project dependencies. We process these files in the next step.

3.1.3 Extracting data

C# type system is problematic for our purposes compared to Java because of the implicit type "var" introduced in version 3.0. As a consequence of its existence, code needs to be compiled in order for the type of a variable to be determined correctly, as opposed to Java where name of the variable's type or supertype is evident from its declaration. This need for compilation limits number of projects we can process.

For compilation and syntax tree processing, we use Roslyn[7] — an open source C# compiler developed by Microsoft. To compile a project we need it to satisfy two requirements:

1) no manual actions are necessary for its build and compilation;
2) a solution file, encompassing source code files, must exist.

In order to compile more projects, we employ Nuget to restore project dependencies prior to compilation.

About 80.6% percent of repositories contain solution files, and of those 47.1% could be compiled.

After compilation, we process projects in the following fashion:

1) find methods with documentation comments;
2) store whole comment and summary section;
3) walk syntax tree of the method body, collecting API call sequence;
4) store method name;
5) store parameter types and names, which we think can potentially provide valuable information, but are not used in this work.

An example of extracting data from method with documentation comment is provided in Fig. 3.

We construct API sequence similarly to the original paper. We traverse the tree in the way an interpreter might traverse it during execution, e.g. depth-first post order, processing method call's arguments before processing the call itself, and so on. When encountering constructor invocation *new C()*, we add *C.new* to the API sequence. When encountering method call *o.m()* where *o* is an instance of a class C, we add *C.m* to the API sequence. Additionally, when encountering *if-else* statement, we firstly process condition expression, then *if*-branch statements and finally *else*-branch statements.

We introduce one additional step to this scheme: when encountering *try-with-resources* node, we save the class *C* of an object being created in the *try* node and

[7] github.com/dotnet/roslyn

after processing everything inside *try* branch we add *C.Dispose* to the API sequence. While it is easier for a programmer to rely on the language feature of *try-with-resources* block to take care of finalization of the resources, this construct is not always used, and we think that our model should know that certain sequences of API calls end with finalization call.

Eventually we obtain 2,886,309 pairs of English descriptions and API sequences. However, this number is not directly comparable to the 7,519,907 methods reported in the DeepAPI paper. The authors explained to us (in an e-mail) that 7,519,907 is the amount of data after filtering out-of-vocabulary words, the step which in our experience removes certain samples entirely, significantly reducing size of the dataset.

Our preprocessing and the final size of dataset is discussed in the further section.

```
/// <summary>
/// Copies the data from one stream to another.
/// </summary>
/// <param name="from">The source stream.</param>
/// <param name="to">The destination stream.</param>
private void Copy(Stream from, Stream to)
{
    var reader = new StreamReader(from);
    var writer = new StreamWriter(to);
    writer.WriteLine(reader.ReadToEnd());
    writer.Flush();
}
```

```
Comment description: copies the data from one stream to another
API calls: System.IO.StreamReader.new System.IO.StreamWriter.new
           System.IO.StreamReader.ReadToEnd
           System.IO.StreamWriter.WriteLine System.IO.StreamWriter.Flush
Full name: PDS.WITSMLstudio.Desktop.Core.Providers.LoggingSoapExtension.Copy
Tokenized name: logging soap extension copy
Parameters: System.IO.Stream from, System.IO.Stream to
```

Fig. 3. Example of data extraction

3.2 Data preprocessing

Upon inspecting the gathered data we conclude that it can be improved prior to being used for model training. By introducing following preprocessing steps we aim to make the training easier and the results consequently better - a notion supported by our experiments (see section 6).

3.2.1 Language detection

We consider our model to work with English language as input, however, many comments are not in it. Therefore, we try to filter out non-English comments using a language detection package[8].

We find, however, that some English sentences are recognized as non-English. In our opinion, most likely reasons are extreme shortness of sentences used for language detection and uncommon profession-specific programmers' vocabulary. We do not want to decrease dataset size by filtering out comments incorrectly recognized as non-English, and so we change our filtering approach.

Instead of leaving only sentences recognized as English, we remove ones that are reported to be in a set of well-recognizable languages (which we deduce by hand examination) that occur in our dataset most often. Languages, sentences in which we remove, are Chinese, Korean, Japanese, Russian, German and Polish (reported in the order of decreasing frequency). As a side note, the reason for good recognition of said languages probably lies in them having alphabets different from the English one.

Such filtering leads to vocabulary containing mostly English words. It reduces training size from 2,886,309 pairs to 2,606,424.

3.2.2 Leaving only distinct pairs

The percent of unique pairs is about 86.6%. Note that we consider two pairs distinct even if English descriptions coincide while API descriptions do not, and vice versa.

We could identify several reasons for occurrence of repetitions:

- auto-generated code and comments (Windows Forms are especially ubiquitous);
- libraries being copied to the project sources instead of being linked as dependencies.

This step reduces amount of training instances from 2,606,424 to 2,259,653.

3.2.3 Repetition contraction

In some API sequences an API call is repeated several times in a row. This could happen as a result of our AST linearization in a situation where, for example, an API call is made with different parameters in branches of an *if-else* statement. Since we do not record call parameters and when linearizing *if-else* statement save API calls from both branches, this may lead to an API call repeating twice in the resulting sequence. End user would not care about such repetitions in the output of the model, so we remove them before training, leaving only one copy of API call in a row.

This step does not influence amount of data, but rather is intended to improve quality of the existing training samples

[8] github.com/Mimino666/langdetec

3.2.4 Vocabulary filtering

Similar to the original paper, we create vocabularies of 10,000 most popular words in each language, and filter out the rest. If after filtering no words are left in either English description or API one, we remove the pair altogether.

This step reduces training dataset size significantly, from 2,259,653 to 1,397,597.

3.2.5 Stemming

Additionally we experiment with, but eventually discard a preprocessing step of stemming.

Stemming is the process of reducing inflected words to their bases. We intended to use it, as is usual, to decrease vocabulary by replacing multiple word forms with the root.

In our case it fails to provide improvement and instead makes results worse. A possible explanation may lie in the fact that stemming model was trained on regular words, not ones specific for software development and therefore works badly with this unusual vocabulary.

We discuss impact of the preprocessing steps in section 6.

The final size of our dataset is 1,397,597 pairs, which is more than 5 times smaller than 7,519,907 pairs used for training in the original paper. Even if only preprocessing from the original paper is used (i.e. vocabulary filtering and nothing else), dataset size is 1,692,898 (of which 1,434,805 pairs are unique). We consider this a significant problem that very probably makes achieving comparable results harder and takes a great toll on the model performance.

For easy reproduction of our research and for conduction of new experiments in the area, we provide our dataset[9], as well as the code used to collect[10] and preprocess[11] it.

4. Transfer learning for API mining

Broadly speaking, transfer learning is utilizing knowledge gained in one problem to solve another. It is often used in NLP [13] and neural machine translation, especially in the contexts where data is scarce [14]. Since our situation is one of lacking data (as shown by an experiment in section 6), we decided to investigate this idea.

4.1 Alternative dataset

To apply transfer learning to our problem of generating API calls given English description, we need to train a model for a task that is different, yet very similar.

As already mentioned, the DeepAPI paper proposes method body as a source of API description of the functionality and method comment as a source of the English one. But there is another description for a method functionality beside its comment — its

[9] kaggle.com/awesomelemon/csharp-commented-methods-github
[10] github.com/AwesomeLemon/api-extraction
[11] github.com/AwesomeLemon/api-extraction-scripts

name. Combined with class name, it seems descriptive of the method's contents. While not forming proper natural language sentences, these names could provide crude approximations.

Examples of correspondence between comments and names of the methods are provided in Table 1. It can be seen that generally tokenized names are very similar to summary sections of documentation comments. However, this is not always the case. In the last two examples despite similarity between comment and name, essential information is missing from the tokenized name. In the first of these samples key word is "Matches", without it tokenized method name loses meaning. In the second one "Dword" is separated to "d" and "word" due to the tokenizing technique. When we tokenize method name, we assume that naming guidelines are followed and therefore first letter of the method name and first letters of every word in the name are capitalized. Here this leads to a wrong division of words and thus vital information disappears, making description senseless.

However, in most cases method names tokenized in this way are similar to comments and thus provide relatively good description of method contents.

We start exploration of this alternative dataset by simply training a model on it with the best parameters and our preprocessing. Results are not very good (model №4 in Table 2; the table is discussed minutely in section 6).

We conclude that comments indeed seem to be more descriptive of method contents than method names. But can we utilize this new dataset nonetheless?

Table 1. Comparison between method names and comments

Full method name	Tokenized method name	Summary section of documentation comment
Method name corresponds to comment well		
ManagedFusion. Serialization. JsonSerializer.Serialize	json serializer serialize	Serializes to JSON
MathNet.Symbolics. Packages.Standard. Structures. ComplexValue.Cosine	complex value cosine	Trigonometric Cosine of an angle in radians
StickyDesk. Utilities.ResizeBitmap	utilities resize bitmap	Resizes a bitmap image
Nini.Config. IniConfigSource. RemoveSections	ini config source remove sections	Removes all INI sections that were removed as configs
Method name corresponds to comment badly		
Spark.Parser. CharGrammar. StringOf	char grammar string of	Matches a string of characters
TagLib.Asf. DescriptionRecord. ToDWord	description record to d word	Gets the DWORD value contained in the current instance.

4.2 Applying transfer learning for model improvement

We hypothesize that the alternative method names dataset contains valuable information about correspondence between English words and API calls.

In terms of transfer learning, we can consider both our source task and target task to be the same, namely to generate API call sequence given English description of it. The difference lies in the datasets. When training for the source task, we can use the alternative dataset of pairs (Tokenized method name, API call sequence). Then we can utilize gained knowledge when training the model for the target task, which makes use of the original dataset of pairs (Documentation comment summary, API call sequence).

Therefore, we train a model on the alternative dataset, and then use learned weights for initializing the model to be trained on the standard dataset, which is a technique known as pretraining.

In addition, we wonder if we can similarly bootstrap learning without using an alternative dataset. We perform an experiment by training the model on the comments dataset and using it for initialization and training on the same dataset.

We evaluate impact of both approaches in section 6.

5. Model training

Per description in section 2, original authors use Encoder-Decoder architecture. As implementation of RNN they choose GRU [10]. They use 1-layered model with 1,000 hidden units and 120 dimensions for word embedding. To train the model, GroundHog[12] is used.

GroundHog since then has been discontinued, instead we use popular modern framework OpenNMT [15] that is designed specifically to train neural translation models.

We start training from the architecture reported in the original paper. After getting bad results we go on and empirically tune parameters, eventually arriving at following values. As RNN implementation we use LSTM [16] — a more complex model than GRU, with on-par performance, which is highly dependent on the problem. In our task it performs better. We find that 1 layer makes model not complex enough to work with C#, and since it is known that adding more layers increases model's learning ability [17], we introduce additional layers to the total of 3, which impacts results positively. We leave number of hidden units at 1,000 and word embedding at 120 dimensions.

For training, Stochastic Gradient Descent [18] is used with batch size of 32 and exponential learning rate decay. We initialize learning rate to be 1.0 and start multiplying it by 0.7 after every epoch, starting from the sixth one. Every model is trained for approximately 25 epochs on the server equipped with one Nvidia GTX 1070 GPU.

[12] github.com/pascanur/GroundHog

For model testing we separate 12,000 random pairs of descriptions from the dataset; the rest is used for training. We publish our trained model for easy reproduction of the results[13].

After training, when translating queries to API sequences we follow original authors in using beam search [19], a heuristic search algorithm popular in statistical translation. Instead of generating only the most probable word on every step, we generate multiple, and then keep only several most probable sequences. This approach solves the problem of discarding good translation sequences because of some sub-optimal words.

6. Evaluation

6.1 Metrics

In the area of API mining there are no universally adopted metrics. For better comparison to the original paper we follow in its steps and calculate BLEU score [20] for intrinsic evaluation, FRank score [6] and Precision@N for extrinsic one.

6.1.1 BLEU

BLEU is a standard metric used in machine translation to evaluate how closely generated translation resembles reference one. It does not consider grammar or others high-level features, instead calculating corrected geometric mean of n-gram precision on the whole test set [20].

Since we expect the model to generate sequences of API calls similar to the ones extracted from human-written source code, n-gram approach is applicable to our situation. The theoretical foundations of the metric stand in our case, despite target language being language of API calls rather than natural language.

BLEU is reported on the scale from 0 to 100, where higher score corresponds with bigger similarity between generated and reference sequences.

6.1.2 FRank

FRank metric value is the position of the first relevant result in the ranked list, as decided by a human evaluator. Such a metric is justified by two facts. Firstly, good scores of it show that the model has solved exactly the problem we intended for it, i.e. the problem of translating from English to relevant API calls. It was possible for the model to learn a target function uninteresting for us, in which case human evaluators would not find in model output API calls, relevant to the input.

Secondly, it is known that humans scan through ranked results from top to bottom [21], thus making it a desired trait for a model to rank relevant output higher.

[13] public-resources.ml-labs.aws.intellij.net.s3.amazonaws.com/deep-api-sharp/deep-api-sharp-model.t7

In our case FRank is measured on the scale from 1 to 10 (since similar to the DeepAPI paper, our model generates 10 outputs for every query), where lower is better. Where models fail to provide relevant results, FRank is considered to be 11.

6.1.3 Precision@N

Precision@N measures percentage of the relevant results in the first N outputs produced by the system. Following DeepAPI, we report Precision@5 and Precision@10 (note that the term used in the DeepAPI paper is "relevancy ratio N", which does not seem to be an established term).

This metric is reported on the scale from 0 to 100, where higher is better.

6.2 BLEU evaluation

In Table 2 we report results of our experiments in terms of BLEU score. We start experiments with model architecture reported in the original paper and achieve surprisingly bad results of 10.94 BLEU, which is significantly worse than 54.42 BLEU reported in the paper. Since Java and C# are fairly similar, we expected original model to work better. Possible explanation may lie in the size of our dataset, which is more than 5 times smaller.

Table 2. BLEU scores for various models

№	Parameters	Dataset	Preprocessing	Transfer learning from model №	BLEU
Parameter tuning					
1	original	comments	-	-	10.94
2	tuned	comments	-	-	26.26
Data preprocessing					
3	tuned	comments	yes	-	46.99
Different datasets					
4	tuned	names	yes	-	28.57
5	tuned	comments (part)	yes	-	36.63
6	tuned	comments and names	yes	-	44.31
Transfer learning					
7	tuned	comments	yes	3	46.18
8	tuned	comments	yes	4	**50.14**

Model with tuned parameters achieves higher BLEU score of 26.26, which is still far from the original results.

After introduction of our preprocessing steps a 94% increase in BLEU is obtained, and the resulting score of 46.99 comes fairly close to the reported performance of the DeepAPI model.

The best result is achieved by model №8, where we employ transfer learning techniques and pretrain the model on the alternative dataset of method names. Additional analysis of transfer learning application is presented in section 6.3.

Model №4 was trained on the alternative dataset of method names (with the size of 1,967,414 pairs) and yielded not outstanding BLEU score of 28.57. So our model performs worse on the alternative dataset, which is logical, given that descriptions there are not in grammatically correct English and sometimes do not provide good descriptions of functionality, as already discussed in section 4.

To measure if number of training instances indeed impacts model result, as we hypothesized, we try to train the model on 800,000 samples as opposed to the usual 1,397,597. This is the model №5, and it achieves 36.63 BLEU, which is worse than 46.99 achieved under the same parameters, but bigger dataset size. This leads us to the conclusion that dataset size is vital for model performance.

6.3 Transfer learning evaluation

We ask several questions regarding our application of transfer learning techniques:

1) Does it improve our results?
2) Can we use the model itself for pretraining, without utilizing model trained on the alternative dataset?
3) Is transfer learning necessary for performance improvement, or are instead our two datasets so similar that they could be merged and considered as one big dataset?

We answer these questions with several experiments, and come up with following answers.

1) Transfer learning leads to the best results achieved by us (model №8 with BLEU score of 50.14).
2) A model with sub-optimal parameters (which we do not include in the table in order not to clutter it) is improved by approximately 2.5 BLEU when pretrained on itself. However, best model is not, as shown by performance of model №7, that achieves only 46.18 BLEU, which approximately equals the result of the model №3 used for pretraining. So bootstrapping with the dataset itself may make sense sometimes, but not always. Presumably, model №3 was trained so well that there was no room for improvement.
3) We try to merge comments and names in one dataset, which we use for training model №6. Resulting BLEU score of 44.31 is better than using only names (28.57 BLEU, model №4), but worse than using only comments (46.99 BLEU, model №3). Thus we conclude that datasets are fairly different and should not be used together in a straightforward way.

6.4 Human evaluation

DeepAPI reports FRank, Precision@5, Precision@10 on two types of queries: popular ones, that often occur in Bing search log [6], and ones designed to showcase abilities of the proposed approach, including handling of semantically similar requests, longer input handling, combination of several tasks.

We would like to address a potential problem in evaluating the model on queries from the first group. While the DeepAPI paper reports that they do not occur in the training dataset, it seems unlikely since they were chosen for the perk of being popular, i.e. widespread, and authors do not mention filtering them out.

We test the hypothesis that such popular queries occur in the dataset by searching for them in ours. In our training data most of these popular inputs occur multiple times as exact matches. For example, “copy file” occurs 14 times, “reverse string” occurs 7 times, “execute sql statement” occurs 14 times. We conduct this search after filtering out non-unique pairs, so for these occurrences API calls do not coincide; however, they are very similar. Therefore, we believe that testing on such inputs makes little sense, because it essentially means testing on the training set, which speaks only about the model’s ability to memorize. That is expected from any model, and consequently is not very interesting.

However, to show that our model is capable of that, we test on 5 of these queries (the first 5 queries in Table 3).

However, more interesting is the inspection of the model’s ability to generalize, i.e. use gained knowledge to work with novel data. The model should be able to handle combined or semantically similar requests that are not included in the training data. We evaluate our model on 4 new queries, constructed for this exact purpose, and one such query from the DeepAPI paper. Since DeepAPI paper does not report results on 4 new queries, we used online demo of the tool[14] to generate corresponding sequences.

To avoid conflict of interest, we ask 5 professional developers to evaluate extrinsic metrics for our model. Since the correspondence between query and model output is viewed differently by every developer and is up to debate, we consider relevant only those answers that were marked as relevant by at least 2 developers.

In the Table 3 we report results of extrinsic evaluation. In general our model performs approximately the same as the original, which, having established importance of data and our lack of it, we consider an achievement.

Table 3. Extrinsic model evaluation

Query	DeepAPI			DeepAPI#			DeepAPI# output
	FRank	P@5	P@10	FRank	P@5	P@10	
convert int to string	2	40	90	1	80	50	CultureInfo.InvariantCulture Int64.ToString

[14] 211.249.63.55

convert string to int	1	100	100	3	40	60	Int32.TryParse
get current time	10	10	10	1	60	40	DateTime.Now
get files in folder	3	40	50	1	80	90	DirectoryInfo.GetFiles FileInfo.Name List.Add
generate md5 hash code	1	100	100	1	80	60	MD5.Create Encoding.GetBytes MD5.ComputeHash Byte[].Length StringBuilder.Append StringBuilder.ToString
copy a file and save it to a destination path	1	100	100	2	40	40	File.Exists String.Equals File.Exists IO.File.Copy
create socket and then send text	1	100	90	3	20	10	AddressFamily.InterNetwork SocketType.Stream ProtocolType.Tcp Socket.Connect Encoding.GetBytes Socket.Send SocketShutdown.Both Socket.Shutdown Socket.Close
write text using socket	-	0	0	1	100	100	ASCIIEncoding.GetBytes Socket.Send
connect to database and execute statement	1	80	50	6	0	30	IDbConnection.Open IDbConnection.CreateCommand IDbCommand.CommandText IDbCommand.ExecuteScalar Convert.ToInt32 Exception.ToString Console.WriteLine IDbConnection.Close
download from url and save image	3	20	20	1	60	50	String.IsNullOrEmpty WebClient.DownloadFile
Average scores	3.4	59	61	2.0	56	52	

Our model produces slightly less amount of relevant outputs (as shown by Precision@N scores), but ranks these outputs slightly higher (as shown by FRank).

Good performance on the first 5 queries demonstrate that our model is capable or memorizing correct answers, and outputs to the second 5 queries show that it can manage long requests, that require performing several action, as well as semantically similar requests.

However, both models are not very stable to slight semantic variations in the input. For example, query “create socket and then send text” is understood very well by

DeepAPI, while DeepAPI# produces low amount of relevant answers, and on the contrary, query "write text using socket" perplexes DeepAPI, that generates no socket-related calls in the top 10 results, while DeepAPI# generates only relevant output.

Additionally it should be noted that while models' outputs are not directly comparable due to different target languages, both models should still be able to correctly answer queries we are testing them on, since these tasks are fairly common and programming-language-independent.

6.5 Limitations

As already discussed in the previous section, our model can be inconsistent and sensitive to query wording. While DeepAPI# is capable of understanding synonymous queries and generating similar relevant output, it does not generate exactly the same sequences.

In addition, our model is data-hungry. While we do not artificially limit our vocabulary with standard C# library, as DeepAPI does with Java and JDK, we still observe that the model cannot take into account APIs with low amount of usages. It can work with extremely popular Math.NET and Json.NET, but not with many other frameworks, even though their APIs are included in the model dictionary. It remains an open problem for the further research to find ways to make model less data-hungry, or to fine-tune it for use of specific not very popular libraries.

7. Related work

7.1 API usage pattern mining

This group of projects is primarily concerned with extracting common usages of the library. The first algorithm to mine API patterns was MAPO [22]. It starts with clustering API sequences, then for every cluster finds API calls that are the most frequent there and passes those to an API usage recommender, that ranks API calls according to their similarity to the code context.

UP-Miner [23] improves upon MAPO by using API call sequence n-grams as a clustering metric and an additional clustering step. A near parameter-free approach PAM [4] significantly outperforms both MAPO and UP-Miner, introducing a probability model constructed in the form of a joint probability distribution over API calls observed in code and the underlying unobserved API patterns, used by developer. *Acharya et al.* [24] extract API patterns as partial orders, and unfortunately do not compare results to those of previous approaches.

The differences of these projects from our work are twofold. Firstly, these models do not allow user to specify their exact needs (MAPO and UP-Miner take API call as input, but an API call can be utilized in more than one scenario, therefore using it as input can be ambiguous; PAM and framework of *Acharya et al.* do not ask for input). This leads to the output containing many samples irrelevant to user, while not

guaranteeing to provide those he was wishing for. Secondly, to use such models one needs to know beforehand which API calls (in case of MAPO and UP-Miner) or libraries (in case of PAM and framework of *Acharya et al.*) he is interested in. Our approach allows for recommendation of APIs to use, as well as the specifics of the usage.

7.2 Generating source code from natural language

Code generation based on natural language input is one of the holy grails of Computer Science. It could be seen as a more promising alternative to our problem: after all, rather than generate API call sequence and leave it to the software developer to write code utilizing it, it would be better to just generate code in the first place.

However, current research in the area seems to be far from this dream. It mostly focuses on Domain Specific Languages [25], [26], which are simpler than general-purpose programming languages and have by definition limited usage domain.

Recent developments in generating code in general-purpose languages include works by *Ling et al.* [27] and *Yin et al.* [28]. The first paper proposes a novel approach of Latent Predictor Networks that allows for better copying of relevant key words from input to output. The second paper introduces a special version of Encoder-Decoder model, where Decoder is tailored to generate syntax trees as opposed to sequences.

The main difference between these works and ours lies in the datasets. *Ling et al.* and *Yin et al.* report results on two datasets: code of Hearthstone cards and annotated Django code (*Ling et al.* also report results on the dataset of code of Magic the Gathering cards, but this dataset is semantically very similar to the Hearthstone one). The target code for the Hearthstone dataset is rather homogenous and limited to small subset of the wide variety that is the Python language, thus resembling code in DSL more than code in general-purpose language. And while Django dataset covers various usage scenarious, it contains impractically sesquipedalian natural language descriptions of every line of code. For example description of the line "`for i in range(0, len(result)):`" is "for every i in range of integers from 0 to length of result, not included". The generation of code from descriptions several times longer than itself seems impractical.

Our dataset, on the other hand, contains wide variety of API usages, described by reasonably long sentences like "Serializes to JSON", which resemble real queries written by programmers in order to look up interesting APIs.

7.3 Deep neural machine translation and source code

Deep API Learning paper [7] itself was published in 2016, is widely cited, but little work has followed from it. The authors went on to successfully apply the neural machine translation approach to code migration between Java and C# [29], which shows that the proposed architecture can model both languages of API sequences well.

Lin et al. [30] similarly to us apply the Encoder-Decoder approach to a different target language, specifically Bash. They succeed, but it should be noted that their research problem is a simpler one in terms of target language used, since only 17 commands were selected from Bash. Together with command flags, types of open-vocabulary constants and logical connectives (&&, ||, parentheses) total output dictionary size does not exceed 300. To contrast that, our work is concerned with the same API dictionary size as original paper, which is 10,000 and therefore requires vastly bigger dataset and more complex model.

8. Conclusion

In this paper, we applied deep learning approach for recommendation of C# API calls, removing one of the threats to the validity of the paper that originally proposed this approach for Java. To achieve this goal, we collected massive dataset, introduced several data preprocessing steps, and finally employed transfer learning techniques.

Extending DeepAPI approach turned out to be nontrivial even for a similar language. Nonetheless, its main idea of modelling API sequences with RNN Encoder-Decoder stands.

Data preprocessing steps, suggested by us, are not dependent on C# and should therefore be applicable to any programming language, thus they should make extending the approach even to very different languages much easier.

By releasing data, code and trained model we hope to allow for repeatability of the experiments and to inspire further research in the area.

Acknowledgment

The authors would like to thank JetBrains Research[15] for providing a GPU-equipped server for fast machine learning models training, as well as for the Young Researcher stipend granted to our team. Additionally we would like to thank Kirsanov Alexander and other friendly developers from the JetBrains ReSharper team for their input in evaluating FRank and Precision@N metrics.

References

[1]. M. P. Robillard and R. Deline. A field study of api learning obstacles. Empirical Software Engineering, vol. 16, no. 6, 2011, pp. 703–732.

[2]. M. P. Robillard. What makes apis hard to learn? Answers from developers. IEEE software, vol. 26, no. 6, 2009, pp, 27-34.

[3]. J. Stylos and B. A. Myers. Mica: A web-search tool for finding api components and examples. In Proc. of the IEEE Symposium on Visual Languages and Human-Centric Computing, 2006, pp. 195–202.

[4]. J. Fowkes and C. Sutton. Parameter-free probabilistic api mining across github. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016, pp. 254–265.

[15] research.jetbrains.org

[5]. S. Shoham, E. Yahav, S. J. Fink, and M. Pistoia. Static specification mining using automata-based abstractions, IEEE Transactions on Software Engineering, vol. 34, no. 5, 2008, pp. 651–666.
[6]. M. Raghothaman, Y. Wei, and Y. Hamadi. Swim: Synthesizing what i mean-code search and idiomatic snippet synthesis, In Proc. of the IEEE/ACM 38th International Conference on Software Engineering (ICSE), 2016, pp. 357–367.
[7]. 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, 2016, pp. 631–642.
[8]. A. Chebykin, M. Kita, and I. Kirilenko. Deepapi#: Clr/c# call sequence synthesis from text query. In Proceedings of the Second Conference on Software Engineering and Information Management, vol. 1864. CEUR-WS.org, 2017, pp. 6–11. (in Russian) [Online]. Available: http://ceur-ws.org/Vol-1864/paper 5.pdf
[9]. I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In Advances in neural information processing systems, 2014, pp. 3104–3112.
[10]. K. Cho, B. Van Merri¨ enboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation, arXiv preprint arXiv:1406.1078, 2014.
[11]. M. Schuster and K. K. Paliwal. Bidirectional recurrent neural networks. IEEE Transactions on Signal Processing, vol. 45, no. 11, 1997, pp. 2673–2681.
[12]. D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473, 2014.
[13]. P. H. Calais Guerra, A. Veloso, W. Meira Jr, and V. Almeida. From bias to opinion: a transfer-learning approach to real-time sentiment analysis. In Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining, 2011, pp. 150–158.
[14]. B. Zoph, D. Yuret, J. May, and K. Knight. Transfer learning for low-resource neural machine translation. arXiv preprint arXiv:1604.02201, 2016.
[15]. G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. M. Rush. Opennmt: Open-source toolkit for neural machine translation. arXiv preprint arXiv:1701.02810, 2017.
[16]. F. A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: Continual prediction with lstm. Neural Computation, vol. 12, issue 10, 2000, pp. 2451-2471
[17]. A. Graves, A.-r. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In Proceedings of the IEEE international conference on Acoustics, speech and signal processing, 2013, pp. 6645–6649.
[18]. J. Kiefer and J. Wolfowitz. Stochastic estimation of the maximum of a regression function. The Annals of Mathematical Statistics, vol. 23, 1952, pp. 462–466.
[19]. P. Koehn. Pharaoh: a beam search decoder for phrase-based statistical machine translation models. In Proceedings of the Conference of the Association for Machine Translation in the Americas, 2004, pp. 115–124.
[20]. K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. Bleu: a method for automatic evaluation of machine translation. In Proceedings of the 40th annual meeting on association for computational linguistics, 2002, pp. 311–318.
[21]. L. A. Granka, T. Joachims, and G. Gay. Eye-tracking analysis of user behavior in www search. In Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval, 2004, pp. 478–479.
[22]. T. Xie and J. Pei. Mapo: Mining api usages from open source repositories. In Proceedings of the 2006 international workshop on mining software repositories, 2006, pp. 54–57.

[23]. J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang. Mining succinct and high-coverage api usage patterns from source code. In Proceedings of the 10th Working Conference on Mining Software Repositories, 2013, pp. 319–328.
[24]. M. Allamanis and C. Sutton. Mining idioms from source code. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014, pp. 472–483.
[25]. A. Desai, S. Gulwani, V. Hingorani, N. Jain, A. Karkare, M. Marron, S. Roy. Program synthesis using natural language. In Proceedings of the 38th International Conference on Software Engineering, 2016, pp. 345–356.
[26]. S. Gulwani and M. Marron. Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. In Proceedings of the 2014 ACM SIGMOD international conference on Management of data, 2014, pp. 803–814.
[27]. W. Ling, E. Grefenstette, K. M. Hermann, T. Koˇ cisk` y, A. Senior, F. Wang, and P. Blunsom. Latent predictor networks for code generation. arXiv preprint arXiv:1603.06744, 2016.
[28]. P. Yin and G. Neubig. A syntactic neural model for general-purpose code generation. arXiv preprint arXiv:1704.01696, 2017.
[29]. X. Gu, H. Zhang, D. Zhang, and S. Kim. Deepam: Migrate apis with multi-modal sequence to sequence learning. arXiv preprint arXiv:1704.07734, 2017.
[30]. X. V. Lin, C. Wang, D. Pang, K. Vu, and M. D. Ernst. Program synthesis from natural language using recurrent neural networks. Technical Report UW-CSE-17-03-01, University of Washington, Department of Computer Science and Engineering, 2017.

Применение глубокого машинного обучения к синтезу цепочки вызовов C#

А.Е. Чебыкин <a.e.chebykin@gmail.com>
Я.А. Кириленко <jake.kirilenko@gmail.com>
Математико-механический факультет,
Санкт-Петербургский государственный университет
Университетский пр., дом 28, Санкт-Петербург, 198504, Россия

Аннотация. Большая часть стандартных для программирования задач — например, соединение с базой данных, отображение картинки, чтение файла — давно реализована в различных библиотеках и доступна через соответствующие Application Programming Interfaces (APIs). Однако чтобы воспользоваться ими, разработчик должен сначала узнать, что они существуют, а затем — как правильно с ними работать. В настоящее время Интернет кажется наилучшим и самым популярным источником подобной информации. Недавно был предложен другой подход, основанный на глубоком машинном обучении и реализованный в виде инструмента под названием DeepAPI. По описанию желаемой функциональности на английском языке он генерирует цепочку вызовов Java функций. В данной статье мы показываем, как подход может быть перенесен на другой язык программирования (C# вместо Java), на котором доступно меньше открытого кода; мы описываем техники, позволившие достичь результата, близкого к оригинальному, а также техники, которые не улучшили производительность.

Наконец, чтобы облегчить будущие исследования в области, мы публикуем наши набор данных, код и обученную модель.

Ключевые слова: API; глубокое обучение; поиск кода; рекуррентная нейронная сеть; обучение с подкреплением.

DOI: 10.15514/ISPRAS-2018-30(3)-5

Для цитирования: Чебыкин А.Е., Кириленко Я.А. Применение глубокого машинного обучения к синтезу цепочки вызовов C#. Труды ИСП РАН, том 30, вып. 3, 2018 г., стр. 63-86 (на английском языке). DOI: 10.15514/ISPRAS-2018-30(3)-5

Список литературы

[1]. M. P. Robillard and R. Deline. A field study of api learning obstacles. Empirical Software Engineering, vol. 16, no. 6, 2011, pp. 703–732.
[2]. M. P. Robillard. What makes apis hard to learn? Answers from developers. IEEE software, vol. 26, no. 6, 2009, pp, 27-34.
[3]. J. Stylos and B. A. Myers. Mica: A web-search tool for finding api components and examples. In Proc. of the IEEE Symposium on Visual Languages and Human-Centric Computing, 2006, pp. 195–202.
[4]. J. Fowkes and C. Sutton. Parameter-free probabilistic api mining across github. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016, pp. 254–265.
[5]. S. Shoham, E. Yahav, S. J. Fink, and M. Pistoia. Static specification mining using automata-based abstractions, IEEE Transactions on Software Engineering, vol. 34, no. 5, 2008, pp. 651–666.
[6]. M. Raghothaman, Y. Wei, and Y. Hamadi. Swim: Synthesizing what i mean-code search and idiomatic snippet synthesis, In Proc. of the IEEE/ACM 38th International Conference on Software Engineering (ICSE), 2016, pp. 357–367.
[7]. 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, 2016, pp. 631–642.
[8]. A. Chebykin, M. Kita, and I. Kirilenko. Deepapi#: Clr/c# call sequence synthesis from text query. In Proceedings of the Second Conference on Software Engineering and Information Management, vol. 1864. CEUR-WS.org, 2017, pp. 6–11. (in Russian) [Online]. Режим доступа: http://ceur-ws.org/Vol-1864/paper 5.pdf
[9]. I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In Advances in neural information processing systems, 2014, pp. 3104–3112.
[10]. K. Cho, B. Van Merri¨ enboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation, arXiv preprint arXiv:1406.1078, 2014.
[11]. M. Schuster and K. K. Paliwal. Bidirectional recurrent neural networks. IEEE Transactions on Signal Processing, vol. 45, no. 11, 1997, pp. 2673–2681.
[12]. D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473, 2014.
[13]. P. H. Calais Guerra, A. Veloso, W. Meira Jr, and V. Almeida. From bias to opinion: a transfer-learning approach to real-time sentiment analysis. In Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining, 2011, pp. 150–158.

[14]. B. Zoph, D. Yuret, J. May, and K. Knight. Transfer learning for low-resource neural machine translation. arXiv preprint arXiv:1604.02201, 2016.
[15]. G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. M. Rush. Opennmt: Open-source toolkit for neural machine translation. arXiv preprint arXiv:1701.02810, 2017.
[16]. F. A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: Continual prediction with lstm. Neural Computation, vol. 12, issue 10, 2000, pp. 2451-2471
[17]. A. Graves, A.-r. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In Proceedings of the IEEE international conference on Acoustics, speech and signal processing, 2013, pp. 6645–6649.
[18]. J. Kiefer and J. Wolfowitz. Stochastic estimation of the maximum of a regression function. The Annals of Mathematical Statistics, vol. 23, 1952, pp. 462–466.
[19]. P. Koehn. Pharaoh: a beam search decoder for phrase-based statistical machine translation models. In Proceedings of the Conference of the Association for Machine Translation in the Americas, 2004, pp. 115–124.
[20]. K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. Bleu: a method for automatic evaluation of machine translation. In Proceedings of the 40th annual meeting on association for computational linguistics, 2002, pp. 311–318.
[21]. L. A. Granka, T. Joachims, and G. Gay. Eye-tracking analysis of user behavior in www search. In Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval, 2004, pp. 478–479.
[22]. T. Xie and J. Pei. Mapo: Mining api usages from open source repositories. In Proceedings of the 2006 international workshop on mining software repositories, 2006, pp. 54–57.
[23]. J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang. Mining succinct and high-coverage api usage patterns from source code. In Proceedings of the 10th Working Conference on Mining Software Repositories, 2013, pp. 319–328.
[24]. M. Allamanis and C. Sutton. Mining idioms from source code. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014, pp. 472–483.
[25]. A. Desai, S. Gulwani, V. Hingorani, N. Jain, A. Karkare, M. Marron, S. Roy. Program synthesis using natural language. In Proceedings of the 38th International Conference on Software Engineering, 2016, pp. 345–356.
[26]. S. Gulwani and M. Marron. Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. In Proceedings of the 2014 ACM SIGMOD international conference on Management of data, 2014, pp. 803–814.
[27]. W. Ling, E. Grefenstette, K. M. Hermann, T. Ko˘ cisk` y, A. Senior, F. Wang, and P. Blunsom. Latent predictor networks for code generation. arXiv preprint arXiv:1603.06744, 2016.
[28]. P. Yin and G. Neubig. A syntactic neural model for general-purpose code generation. arXiv preprint arXiv:1704.01696, 2017.
[29]. X. Gu, H. Zhang, D. Zhang, and S. Kim. Deepam: Migrate apis with multi-modal sequence to sequence learning. arXiv preprint arXiv:1704.07734, 2017.
[30]. X. V. Lin, C. Wang, D. Pang, K. Vu, and M. D. Ernst. Program synthesis from natural language using recurrent neural networks. Technical Report UW-CSE-17-03-01, University of Washington, Department of Computer Science and Engineering, 2017.