Automated Web Service Specification Generation Through a Transformation-Based Learning

Mehdi Bahrami(✉) and Wei-Peng Chen

Fujitsu Laboratories of America, Sunnyvale, CA, USA
{mbahrami,wchen}@fujitsu.com

Abstract. Web Application Programming Interface (API) allows third-party and subscribed users to access data and functions of a software application through the network or the Internet. Web APIs expose data and functions to the public users, authorized users or enterprise users. Web API providers publish API documentations to help users to understand how to interact with web-based API services, and how to use the APIs in their integration systems. The exponential raise of the number of public web service APIs may cause a challenge for software engineers to choose an efficient API. The challenge may become more complicated when web APIs updated regularly by API providers. In this paper, we introduce a novel transformation-based approach which crawls the web to collect web API documentations (unstructured documents). It generates a web API Language model from API documentations, employs different machine learning algorithms to extract information and produces a structured web API specification that compliant to Open API Specification (OAS) format. The proposed approach improves information extraction patterns and learns the variety of structured and terminologies. In our experiment, we collect a sheer number of web API documentations. Our evaluation shows that the proposed approach find RESTful API documentations with 75% accuracy, constructs API endpoints with 84%, constructs endpoint attributes with 95%, and assigns endpoints to attributes with an accuracy 98%. The proposed approach were able to produces more than 2,311 OAS web API Specifications.

Keywords: Web API service · REST API · Natural language processing · Machine learning

1 Introduction

Web Application Programming Interface (API) [13] exposes data and software functions to third-parties or subscribed users. Web APIs can be reached locally or remotely by using REpresentational State Transfer (REST) [23] which is a software architecture style that divides platform and programming language. The web service REST APIs are core technology of software integration and it

Q. Wang et al. (Eds.): SCC 2020, LNCS 12409, pp. 103–119, 2020.
https://doi.org/10.1007/978-3-030-59592-0_7

has been used widely in cloud-based services and the Internet-of-Things (IoT) devices. In addition, a third-party is able to efficiently integrate an external functions or data through REST APIs in own native software applications.

API publishers charge users based on their number of API calls, usage of API, or a flat rate. For instance, an API publisher may provide a range of free API calls (e.g., 1000 free API calls per day) and charge additional fee for the additional requests.

The power of APIs enable a web service provider to offer information without sharing its own implementation. Different hardware devices also may use APIs to expose their functionality and their internal data. For example, web APIs in Internet-of-Tings (IoT) allows users to read or access data from a connected device [11].

1.1 Motivation

This study aims answer to the following questions. (i) How we can construct API specification from API documentation by employing machine-learning technology? (ii) Is the approach scalable to apply the process to a variety of APIs?

With the rise in number of web APIs in the market, manually understanding of all APIs and their endpoints is not only labor intensive but also it is an error prone task for software engineers. Web APIs might be revised or updated periodically, when it creating a significant overheads for software engineers to keep track of all changes.

In a digital business when major digital services rely on different third-party platforms, web API economy is the key point for determining the value of provided services. Software engineers need to choose the best web APIs for developing a reliable and cost effective service, which requires web API evaluation by reading the API documentations. This concern raises several questions, *How software engineers find all relevant web APIs? How software engineers automatically can evaluate web APIs without reading lengthy web API documentations?* The answer is accessing a machine-readable API specification. However, it raises another challenge when majority of API providers do not have any standard API specification.

In order to understand the variety of APIs in a standard format, we can employ machine-learning to construct API specification. API Specification can be used for automated API validation, automated API monitoring and automated API quality assessment [16]. An API Specification which aims to be generated by machine, allows software engineers to use automation on analyzing, validating and code synthesis to generate software application. Machine-generated API specification may also help API provider to offer better services such as automated API testing when machine read API documentations and validate information automatically. API specification opens door to a set of new technologies such as automated service integration when all required specifications have been defined by machine.

1.2 Related Works

Although there are some existing standardization initiatives around API specification to produce a machine readable API specification but only major API providers offers this type of format for their API specifications. API providers may offer Open API Specification (OAS), YAML which is a human-friendly and cross language machine readable format. Our goal in this study is to provide a platform that produces OAS from any API documentations. Therefore, our approach should be able to understand the variety of APIs.

There are also some studies around producing API specifications. For instance, Robillard et al. [20] presented a field study by performing survey and in-person interview to recommend how to design API documentations. However, this study does not provide an automated approach to produce API specification.

Although the title of a study by Gu et al. [10] is similar to our work, it is not an information extraction platform. The authors present a natural language query-based platform to find API usage.

Zhong et al. [25] defined a method to create specification for Java APIs which cannot be applied to REST API documentations. In another study, [7] shown a composition architecture for API description. In a recent study, [24] enriches API Guru by constructing API endpoints. However, our study focuses on a variety of API documentations without considering a predefined OAS. Another alternative option is using web annotation to construct API specification which is explained by Bahrami et al. [3]. Our approach in this paper employs several machine learning algorithms to mitigate software engineers' issues by mining a large number of API documentations. It extracts information and constructs OAS API Specifications for more than 2,311 APIs. Once we have a large number of API specification, the API OAS file can be used to produce other artifact such as automated API validation, automated code generation and API recommendation.

This paper organized as follows. In the next section, we describe i) API Corpus (Sect. 2.2) construction that includes a web-crawler and a REST API Filter (Sect. 2.3) which uses a logistic-regression for filtering out non-REST API documentations; ii) Information Extraction (Sect. 2.6) that uses an API Language Model (Word2Vec) and transformation-based learning algorithm; Table Extraction (Sect. 2.8) extracts HTML table tags and it constructs a SVM model to detect API attributes (e.g., parameter); Sect. 3 defines our datasets and evaluation results of each component. Finally, Sect. 4 summarizes this study.

2 Proposed Approach

Our proposed approach consist of an end-to-end platform with different components. Figure 1 shows the key component of the proposed approach. Our goal is extracting information from API documentations which is published by API providers.

2.1 Parallel Web Crawler

In the first step, we collect a massive number of API documentations. By collecting a large number of API documentations, first, it allows us our proposed machine learning algorithm to learn from a variety of data that enables the platform to learn different API documentations with different structures. Second, the proposed method learns from a volume of data that improves the accuracy of information extraction. We use a parallel web crawler to collect a massive number of API documentations which have been published by API providers. It stores HTML file on a local disk.

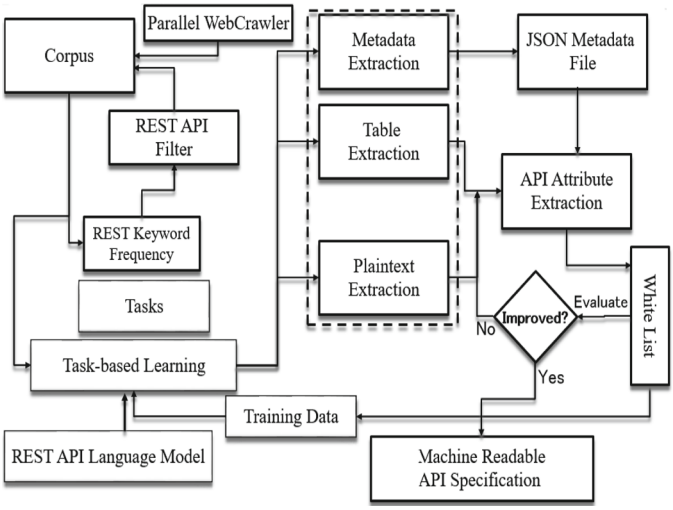

Fig. 1. An overview of the key component of the proposed approach

2.2 API Corpus

API corpus is a collection of HTML pages of API documentations which are collected by parallel web-crawler agents.

2.3 REST API Filter

This component allows API Learning to filter out the non-REST pages from API Corpus. The REST API Filter employs a logistic regression model which explained by [22] to detect REST API documentations. For each API documentations we generate an array of k REST keywords where key_j^k represents the term frequency of jth keyword in each document doc_i. The following equation shows the logistic regression function. It is a linear classification model which classifies the REST content and non-REST contents of collected HTML pages.

$$F(x) = \frac{1}{1 + e^{-(tf(doc_i, key_j^k) - th)}} \tag{1}$$

In Eq. 1, tf represents the linear function of variable *doc* (term frequency) for each API documentations and key_j^k that indicates each key_j in kth REST keywords. It clusters pages into REST or non-REST API documentations.

2.4 Tasks

In order to extract information from *API Corpus*, we have a set of rule-based regular expression patterns for different tasks. Each task includes an initial regular expression pattern for the given task. For example, a task may define a regular expression to extract *API endpoint* or to extract *default value*, *maximum and minimum values* of a parameter. Each initial regular expression pattern can be improved iteratively through a transformation-based learning [18]. In a related study, [12] define a model to train a regular expression pattern to improve the acceptance of a language per positive sample cases, authors apply some restriction rules to the initial regular expression but it cannot extend the initial regular expression. In our proposed method, we use both extension phase (by using *API Language Model* and other rules) to extend the initial pattern, and reduction phase that uses restriction rules to improve the pattern for accepting of a language for a given task.

2.5 API Language Model

API Language Model is defined as a neural language model [5] that shows probability distribution on all sentences in *API Corpus*. It uses embedding of words to predict word sequences and it is defined as follows.

$$\sum_{-k \leq j-1,\, j \leq k} \log P(w_{t+j}|w_t) \tag{2}$$

k denotes the previous words, j denotes the current word. Since this is a domain-specific information extraction task, we cannot use existing language models because it adds noise for our information extraction which is explained in Sect. 2.6. To retrieve a similar word from the language model, we use a cosine similarity [8] which measures the similarity of two words, $W_{1,i}$ and $W_{2,i}$ based on their vector representations where $i = [1..n]$ in defined API language model of n APIs. By computing the similarity from the language model, it allows us to retrieve all synonyms terminologies of (e.g., $W_{1,i}$) which have been used in *API Corpus* by different APIs.

2.6 Learning Diverse Extraction

We define several information extraction **tasks** where each task corresponds to extraction a single information from API documentation. Accumulation of output of all tasks (**trained model**) provide API specifications. In addition, we use some tasks of trained model to categorize information. For example, a task may classify the content of a table as a *response type* or a *parameter type*.

We defined a set of positive and negative examples for each task. Each task has an initial regular expression pattern and it is defined manually but it is improved iteratively when it learns different positive and negative examples. We use transformation-based learning where it consists of two phases that include **extension phase** and **reduction phase**. In order to train a model that applies different regular expression patterns to each task, we need to update the initial regular expression pattern and expend the constant words which have been used in initial pattern. The main target of two phases processing is updateding the pattern. The second target is updating constant words which have been used in initial patter because different API providers may require different patterns to extract the same information in OAS (e.g., API endpoints). A trained task should be trained based on both positive and negative examples. For instance, Facebook uses the terminology of *fields* to describe the input parameters of an endpoint[1]; but Google uses *parameters* to describe the input parameters of an endpoint[2].

By using *API Language Model* we can find synonymous of given constant words from initial pattern (e.g., parameter) and add *fields* as an equivalent terms in task definition. In this example, the final trained regular expression task should be able to extract *fields* from Facebook API documentations and *parameters* from Google API documentations. It can construct API specification for both APIs. In the OAS of each API specification, the trained task can extract both relevant information and constructs OAS parameters as: $paths \rightarrow endpoint \rightarrow HTTP\_Verb \rightarrow parameters$.

We develop a novel approach based on transformation-based learning as explained in Fig. 2 that i) expends the acceptance of initial regular expression (RE) pattern, and then, it reduces the RE pattern to only matched positive sample cases to minimize the acceptance of negative sample cases. After completion of both phase, the final RE pattern learned from both positive and negative examples; therefore, the task can provide a common RE pattern that maximizes the positive cases and minimizes the negative cases. In addition, since different API providers use a variety of terminologies for a single word, a constant regular expression is not capable to learn efficiently all synonyms words. We use *API Language Model* that finds all synonyms words according to API Corpus, then it applies the new set of synonyms words for each constant of RE pattern. The model learns new words in addition to original constant that improves acceptance of positive examples and reduces negative examples. Each OAS API Specification consist of several objects, such as API metadata (e.g., title, description), endpoints, attributes, responses, and etc. Therefore, we need a set of different tasks to extract information and produce a structured based OAS file (JSON). The following tasks shows some examples of *IE Tasks* in API Learning.

i) API Endpoint extraction task provides the key information of a REST API and it provides a URL for an API endpoint along with its HTTP verb;

[1] See https://developers.facebook.com/docs.
[2] See https://developers.google.com/+/web/api/rest.

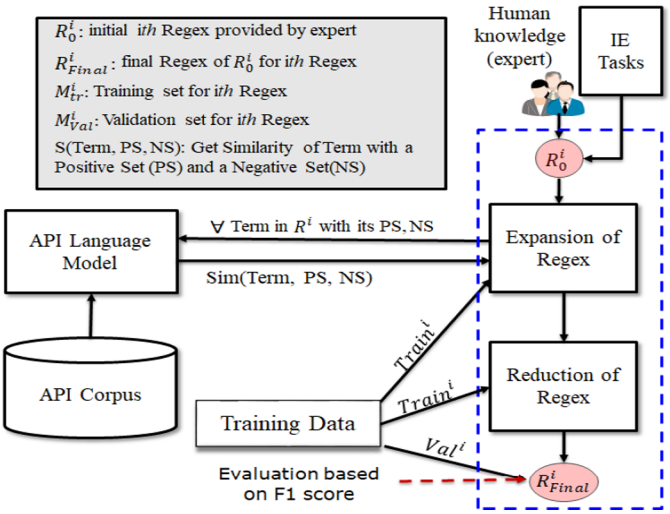

Fig. 2. Learning diverse extraction framework

ii) Parameter attribute extraction returns a list of input/output parameter and security information of an API endpoint. It also includes some specific sub-tasks such as a minimum value of a parameter, a maximum value of a parameter, the default value of a parameter and etc. Therefore, the method need to understand and classify the content of HTML tables to recognize input parameters, output parameter and etc. For each task we need a method to detect and extract information from API documentation. We use regular expression to define different task extract. The problem statement of information extraction from API documentations can be defined for a target API ($\mathcal{A}$). Our goal is to extract a set of positive sample case and avoid negative sample cases that can be extracted from an API documentations. Let $s_{p,i}$ be the set of i positive sample case and $s_{n,j}$ be the set of j negative sample case. It can be defined as $s_{p,i} \in L$ where $i \in \{1, ..., m\}$; and $s_{p,i}$ denotes a set of m positive sample case of language $L_{\mathcal{A}}$ for the target APIs($\mathcal{A}$); and $s_{n,j} \notin L_{\mathcal{A}}$ where $j \in \{1, .., n\}$; and $s_{n,j}$ denotes n negative sample cases of language $L_{\mathcal{A}}$. ($L_{\mathcal{A}}$ represents acceptance language of target APIs, $\mathcal{A}$). f_P defines a function that uses a set of regular expression patterns P, that accepts $s_{p,i}$ sample cases and rejects $s_{n,j}$ in $L_{\mathcal{A}}$. Our objective function can be defined as $\max_{x \in \mathbb{C}} \ f_P(x)$ where C denotes the API documentations that can be retrieved from *API Corpus*. We can summarize the objective function as follows.

$$\max_{x \in \mathbb{C}} \ f_P(|s_p(x_{\mathcal{A}})| - |s_n(x_{\mathcal{A}})|) \tag{3}$$

In this equation, we maximize acceptance of the positive sample cases f_P where it removes the negative sample cases of $s_n(x_{\mathcal{A}})$ for different APIs ($\mathcal{A}$); and $\mathcal{A} \in \{1, .., n\}$ where n denotes the total number of APIs. It takes an

initial regular expression as an input and find an improved regular expression as output. The improvement process of given initial process id defined in Sect. 2.6.

We use a transformation-based learning through two steps. First, we define extension phase to maximize f_{P_i} the acceptance more positive sample cases, and in the second phase (reduction phase) improves the pattern by rejecting negative sample cases for a given initial regular expression pattern. The final improved regular expression aims to apply to majority of APIs where it learns different structured and patterns from variety of API documentations.

Extension Phase. Objective of this phase is extending the given initial RE pattern P_0 that accepts more positive cases from training dataset, τ. Table 1 shows the algorithm of the extension of P_0. Each regular expression decomposes into different types. The algorithm processes each component type as follows. In each iteration, it applies one extension rule and evaluate the pattern, if it accepts more positive cases, then revise the pattern.

i) ***charTerm***: This type of RE component refers to string values (e.g., $default$) and can be extended by similarity extension or character extension method as follows.

a) Similarity extension. It uses the similarity of *charTerm* component (Line 2 in Table 1) by inquiring the value of *charTerm* to *API Language Model*. The model uses *cosinesimilarity* function to return similar terms based on the vector representation of *charTerm* in *API Corpus*; and then, it adds each return value with an OR operation with a pair of parenthesis to $P_i(charTerm)$.

b) Character extension. It applies all possible uppercase and lowercase of a *charTerm* in each iteration (Line 3 in Table 1). API specification is case sensitive, we cannot replace all words as lowercase or as uppercase characters. For example, '*POST*' in REST API documentations shows that this term is referring to a *HTTP verb function* and it is completely different to '*post*' or '*Post*' which are regularly used in English language.

ii) ***Range***: This type of RE component refers to a set of range values (e.g., *[1–4]* accepts number between 1 to 4).

iii) ***RE Component Replacement***: This type of RE component decomposes each RE component into one or multiple RE components. Each component of a RE pattern can be replaced with its equivalent RE if acceptance of positive sample cases is equal or better than $P_i - 1$. For example, $[a - zA - Z]$ can be replaced with $([a - z]|[A - Z])$.

Reduction Phase. The objective of reduction phase is removing unnecessary accepted pattern elements from *newPattern* (returned pattern from *ExtensionPhase*). As shown in Table 1, the algorithm takes a set of positive and negative sample cases, s_p and s_n, respectively. It returns a new pattern, P_i

Table 1. Extension and reduction algorithms

pattern_extension()

Input: a set of positive sample cases s_p,
a set of negative sample cases s_n,
previous regular expression pattern, P_{i-1}.
Output: extended pattern, $newPattern$.

1: for each $CharTerm$ in P_{i-1}:
2: $P_i(charTerm) = Smilar(charTerm, \alpha)$
3: $P_i(charTerm) = CharExtension(charTerm)$
4: for each $openParenthesis$ in P_{i-1}:
5: $P_i(openParenthesis) = starExtension(openParenthesis)$
6: for each range in P_{i-1}:
7: $P_i(range) = increaseRange(range)$
8: for each component in P_{i-1}:
9: $P_i(component) = regexTransformation(component)$
10: if $f_{P_i}(x) > f_{P_{i-1}}(x)$:
11: $newPattern = f_{P_i}$
12: else:
13: $newPattern = f_{P_{i-1}}$
14: return $newPattern$

pattern_reduction()

Input: a set of positive sample cases s_p,
a set of negative sample cases s_n,
extension pattern,$P_\tau = pattern\_extension(P_{i-1})$.
Output: a new pattern, P_i

1: $P_i = P_\tau$
2: for each $ORcomponent$ in P_τ:
3: $P_{tmp} = dropOR(element)$
4: if $val(P_\tau) \geq val(P_{tmp})$:
5: $P_i = P_{tmp}$
6: for each $range$ in P_τ:
7: $P_{tmp} = rangeRestriction(range)$
8: if $val(P_i) \geq val(P_{tmp})$:
9: $P_i = P_{tmp}$
10: for each $charTerm$ in P_τ:
11: $P_{tmp} = charTermRestriction(charTerm)$
12: if $val(P_\tau) \geq val(P_{tmp})$:
13: $P_i = P_{tmp}$
14: return P_i

that can be used in substitute of *newPattern* (P_τ). ***i) OR reduction***: In this phase we remove each component if it does not decrease the acceptance rate of positive cases.

ii) range restriction: Some ranges can be removed or shrank when it does not change the validation rate.

iii) character restriction: It restricts the acceptance of characters. For instance, *"POST \b+ (URI|URL)"* can be restricted to *"POST \b[1,1000](URI|URL)"*, if the validation rate did not decrease by revising the pattern; then it can be decreased to a lower number in each iteration (i.e., *"POST \b[1,999](URI|URL)"*). By performing both phases through several iterations, the final pattern learns majority of sample cases. It satisfies the following conditions: i) maximizes acceptance rate of positive sample cases; ii) minimizes the rejection of positive sample cases: iii) maximizes the acceptance rate of rejection of negative sample cases; and iv) minimizes the acceptance rate of negative sample cases.

2.7 Metadata Extraction

Metadata of an API is one of the set of extraction tasks. For instance, the API title, API security protocol, and API host address. This component generates a set of extracted information and add them into the structured data (*info, host* in OAS file).

2.8 Table Extraction

Most of the API documentations uses HTML table tags to explain list of endpoints, attributes (e.g., parameters). Each HTML table tag may consist of different OAS objects, such as parameters, responses, security, and security definition.

2.9 Plain Text Extraction

Some API providers describe their information as a flat HTML page which means does not have some sort of semi-structured data, such as HTML table tags. The API publisher may use both HTML table tags and plain-text flat description to transfer their information to the readers. This component extracts information from plain text information.

2.10 API Attribute Extraction

Component generates a set of different endpoint's attributes, such as minimum value of a parameter, maximum value of a parameter, default value of a parameter and etc. **White List** contains both manual annotation and automated API validations which is created by calling the API and it contains the results of API endpoint response. The final output is a **Machine Readable OAS API Specification** for each API.

3 Experiment

We implemented all described components of API Learning.

3.1 API Corpus Construction

We used several sources to collect a comprehensive list (pointer list) of APIs, such as ProgrammableWeb, API Harmony, Rapid API, API Guru and etc. We use API title and API documentations URL from the list. Each source may also consist of other metadata information of an API. The pointer list consists of more than 20,000 APIs and some of the information might be incorrect (e.g., incorrect API Doc URL or a generic API title). In this experiment, we have to process the content to fix incorrect information and we target REST APIs. We used *Scrapy*[3] for implementation of the parallel web-crawler. We consider a web-crawler that composes of 32 parallel web-crawler agents. Table 2 shows the size of *API Corpus* in different experiments. We show only some example of different experiments for data acquisition with different maximum deep level of URL extractions and maximum of page per APIs.

Table 2. API corpus size of different experiments

Exp#	*MaxDepth*	*MaxPage*	# of files	Size-GB
17	5	1,000	2,822,997	208.6
20	4	100	148,479	8.3
35	3	300	156,497	7.4
37	4	300	256,583	15.0

Table 3. An example of query of top 7 most similar words to a positive word of ['POST']

HTTP	Prediction	HTTP	Prediction
GET	**0.867**	Request	0.668
DELETE	**0.828**	Endpoint	0.639
PUT	**0.777**	URI	0.639
PATCH	**0.746**		

Table 4. Detection of REST API documentations

Class	Precision	Recall	F1
Positive REST page	0.91	0.93	0.92
Negative REST page	0.89	0.86	0.88
Average	0.91	0.91	**0.91**

[3] https://scrapy.org/.

3.2 API Language Model

We use Word2Vec which is described by Mikolov et al. in [15] and [14]. We cleaned the API Corpus by removing scripts and HTML tags to produce the model. API Language model allows the method to understand semantic definition of each word. We use Gensim [19] to create a Word2Vec [9,21] from *API Corpus*. By providing a set of positive and/or negative words, we may inquiry the model to find similar words. For example, Table 3 shows the synonyms words of *'POST'* in *API Corpus* for top 7 words. This result clearly shows that our API Language Model can successfully detect synonyms words from API documentations where it is trained based on API Corpus. The parameter of Word2Vec is described by Rong in [21]. We chose 300 for the window size which represents the maximum windows distance between a selected word and a predicted word within a sentence. The rest of the parameters of *API Language Model* shown in Table 5.

3.3 Information Extraction

In order to extract information, we defined several tasks with initial pattern of *RE* as described in Sect. 2.6. Some defined patterns have been used for table extraction and detecting table mapping as described in Sect. 2.8. Figure 3 shows a comparison between acceptance of $R_0 \in L_{\mathcal{A}}$ and $R_{final} \in L_{\mathcal{A}}$ for 5 different tasks. The average of iteration in these tasks to achieve (R_{final}) is 14. Table 4 shows 5 different tasks as follows. *Default Value*: learns template to extract default values of a parameter; *Maximum Value*: learns a template to extract maximum value of a parameter; *Optional Parameters*: learns a template to find whether is optional or mandatory; *Parameter Description*: learns a template to extract parameter description section (e.g., heading title of the section); *Introduction block of output parameter*: learns a template to detect if a section corresponds to output parameters of an API (e.g., heading title of a section). Table 4 shows the given input (P_0) to algorithms and shows the result after processing extension and reduction phase as output (P_{final}) which corresponds

Table 5. Hyper parameters of API language model

of sentences: 10,140,000
of words: 23,103,011
of word types: 1,580,559
of unique words (after word types drops): 213,167
of windows in CBOW: 300
Min. count=5 (training parameter)
Min. count leaves 20,960,959 word corpus (90% of original)
downsampling leaves estimated 18,603,396 word corpus
of parallel workers: 64

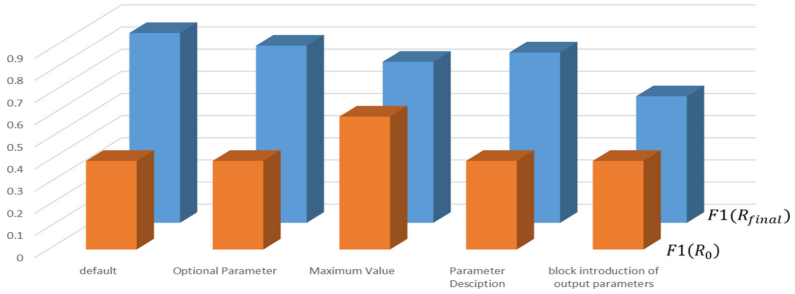

Fig. 3. A comparison between F1 evaluation of R_0 (baseline) and R_{final}

to improved regular expression through the transformation-based learning. As shown in Fig. 4, both the word token (new terminologies) and new pattern have been updated according to *API Language Model*, *RE* extension phase and *RE* reduction phase. By performing on more positive and negative sample cases, (P_{final}) can be improved. After applying (P_{final}) of each task to the *API Corpus*, the proposed platform generates a set of extracted information as a JSON file for each API which includes: i) API metadata, ii) API endpoints, and iii) HTML tables that corresponds to API attributes. The defined tasks helps us to extract different OAS objects but our ultimate goal is using extracted information to interact with APIs. We use two strategies to evaluate our extracted information. First, we annotated the extracted information for a large number of APIs, that includes API endpoint and tables. Second, we perform API call to validate the extracted information which can be applied to all extracted endpoints. In the first approach of evaluation, We annotated 200 APIs that consist of 1,780 extracted tables (API endpoint attributes), and extracted endpoints for 350 APIs that consist of 2,929 endpoints. To the best of our knowledge this large number of annotation of API documentations have been collected for the first time. Each annotation shows that whether an API endpoint extracted correctly from the source or not. The same annotation completed for HTML table to check whether the produced structured file (JSON format) of HTML table is correct or is not correct. We correctly extracted 86.75% of API endpoints and 81.29% of table according to annotated information ($Avg = \mathbf{84.02\%}$). Second, our automated API validation applies to all extracted API endpoints that contain 54,873 endpoints and check the response code. It shows that 76% of endpoints were valid.

We use Scikit-Learn to train a logistic-regression [1] model with L1 penalty to detect REST API documentations. We create an annotation tools based on Selenium [2] that allows a user to quickly annotate API documentations by using a semi-automated platform. The Selenium browser automatically open different API documentations from pointer list and a user manually annotates API documentations as: i) relevant to REST API documentations (*Positive Class*); ii) relevant but explaining different API documentations (*Reference Class*); or, iii) irrelevant to REST API (*Negative Class*). We train the model with considering

Task	State	RE
Default Value	P_0	(default{1,2}value)(.*)(\d|\w+)
	P_{Final}	(.*(default|usually|always|type){1,2}.*(value|attribute|parameter|namespace))(.*)((\d|-)+)
Maimum value	P_0	((maximum value(.*)\d)|(range(.*)\d(.*)\d))
	P_{Final}	(((max|decimal) (value|result)(.*)\d)|((certain)(.*)\d((\W \w)*)\d))
Optional Parameter	P_0	(is required|is necessary|required|necessary|is mandatory|is not optional)"
	P_{Final}	((processing) (names)|(processing)(defined)|(names)|(defined)|(processing) (loaded)|(processing) (bill)(record))"
Param. Desc.	P_0	(parameters|names|titles|subjects|params)
	P_{Final}	((models)|(dates)|(descriptions|ingredients)|(regular)|(testmod))
Introduction block of out. paramv.	P_0	(output (parameter|parameters))
	P_{Final}	((output|result|returned|calculated|signature)((mode|feature|gene)|(tags|form|items|param|proper|models)))

Fig. 4. Sample tasks of learning about API specifications

three classes as well as only positive/negative classes (reference pages considered as part of positive class). Due to page limitation, Table 4 shows only the performance of the model for detecting positive pages versus negative pages.

Table 6. OAS-based table type detection training dataset

Type	#	Type	#
Parameters	5,979	Response	6,290
Security	8,225	Security definition	150

We also collect a set of available OAS-based API specifications as ground-truth from different API providers who offers OAS JSON files, such as Spotify and API directories, such as API Guru. We train a SVM model with the following configuration by using Scikit-learn [17] package in Python. *penalty=l2, dual=False, tol*=1e−3 The model trains with 14,450 data points (70% of dataset) as shown in Table 6. The model predicts four different table types from testing dataset (6,194 data points; 30% of dataset) with an accuracy of **95%**. The next step is assigning tables according to their predicted type (e.g., parameter) to API endpoints, which defines API endpoint attributes. We use a page segmentation algorithm that assigns extracted endpoints to their attributes according their appearance in API documentations (e.g., a table attribute appears after endpoint). We evaluate this assignment process manually for correctness of assignment of 223 API endpoints to their attributes. In this annotated dataset, only 3 out of 223 of assignment were incorrect that defines an accuracy of **98.65%** of the assignments of attributes to API endpoints. API Learning at the end

produces 2,311 API specifications and consolidate information with other available OAS resources which produces 3,311 API specifications and it showcases in our API directory. A partial sample OAS file that collected from our proposed approach is shown in Fig. 5. In addition we deployed the valid APIs in Fujitsu RunMyProcess platform[4]. A demonstration of the previous study and deployment can be found in Bahrami et al. [4] and Choudhary et al. [6]. The deployed APIs can be accessible through Fujitsu RunMyProcess platform where users are able to efficiently design and test a web-based software application by accessing a large number of public APIs.

Fig. 5. A partial snapshot of a produced API specification in OAS format that collected from different sources and API documentation

Table 7 shows a comparison between our approach and D2Spec as a related work [24]. The results of our approach shows that our approach is scalable when it construct 73% correct API endpoints from a sheer number of API endpoints (54,873). Our approach also capable to extract parameter, detect the type of parameters where the related work is only limited to API endpoint construction. Although the performance of endpoint extraction is equal to our results, the total number of extracted endpoints and APIs are much smaller than our outcomes (22% of our 54,873 endpoints). We also evaluate the API endpoints with using API call and API match to our ground through (existing) OAS files.

[4] Available at: https://www.runmyprocess.com.

Table 7. Comparison of related works

Feature	D2Spec (Yang et al. 2018)	Our approach
# of labeled APIs	120	200
Endpoint evaluation method	Endpoint matching	Manual annotation, API call
# of generated API Spec.	120	**1,923**
# of Endpoints	2,486	**54,873**
Endpoint evaluation	84% (22% of our dataset)	84%
Parameter extraction	No	**Yes**
Parameter type detection	No	**81.29%** Acc.

4 Conclusion

In this paper, we introduced a novel framework that collects a large number of API documentations. Our web crawler collected more than 20,000 APIs and we targeted REST APIs. We used a logistic regression model to detect REST API documentations. The framework processes all collected HTML pages as an *API corpus* and generates an *API Language Model* to understand the variety of terminologies of different API documentations. The proposed approach improves a set of information extraction regular expression patterns by extending the acceptance of sample cases and reducing elements that do not improve the acceptance rate. We used the improved patterns to extract OAS objects. We extracted HTML table tags and each table type detected by a SVM model and produces OAS API attributes. Our experimental results show that we have successfully extracted API specification from heterogeneous API documentations with an accuracy of 84%.

References

1. Abney, S.: Semisupervised Learning for Computational Linguistics. Chapman and Hall/CRC, Boca Raton (2007)
2. Automation, S.B.: Selenium ide (2014)
3. Bahrami, M., Chen, W.P.: WATAPI: composing web API specification from API documentations through an intelligent and interactive annotation tool. In: 2019 IEEE International Conference on Big Data (Big Data), pp. 4573–4578. IEEE (2019)
4. Bahrami, M., Park, J., Liu, L., Chen, W.P.: API learning: applying machine learning to manage the rise of API economy. In: Companion Proceedings of the The Web Conference 2018, pp. 151–154 (2018)
5. Bengio, Y., Ducharme, R., Vincent, P., Jauvin, C.: A neural probabilistic language model. J. Mach. Learn. Res. **3**(Feb), 1137–1155 (2003)
6. Choudhary, S., Thomas, I., Bahrami, M., Sumioka, M.: Accelerating the digital transformation of business and society through composite business ecosystems. In: Barolli, L., Takizawa, M., Xhafa, F., Enokido, T. (eds.) AINA 2019. AISC, vol. 926, pp. 419–430. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-15032-7_36

7. Cremaschi, M., De Paoli, F.: Toward automatic semantic API descriptions to support services composition. In: De Paoli, F., Schulte, S., Broch Johnsen, E. (eds.) ESOCC 2017. LNCS, vol. 10465, pp. 159–167. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67262-5_12
8. Dehak, N., Dehak, R., Glass, J.R., Reynolds, D.A., Kenny, P.: Cosine similarity scoring without score normalization techniques. In: Odyssey, p. 15 (2010)
9. Goldberg, Y., Levy, O.: Word2vec explained: deriving Mikolov et al'.s negative-sampling word-embedding method. arXiv preprint arXiv:1402.3722 (2014)
10. Gu, X., Zhang, H., Zhang, D., Kim, S.: Deep API learning. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 631–642. ACM (2016)
11. Hou, L., Zhao, S., Li, X., Chatzimisios, P., Zheng, K.: Design and implementation of application programming interface for internet of things cloud. Int. J. Netw. Manag. **27**(3), e1936 (2017)
12. Li, Y., Krishnamurthy, R., Raghavan, S., Vaithyanathan, S., Jagadish, H.: Regular expression learning for information extraction. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing, pp. 21–30. Association for Computational Linguistics (2008)
13. Masse, M.: REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces. O'Reilly Media, Inc., Sebastopol (2011)
14. Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781 (2013)
15. Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J.: Distributed representations of words and phrases and their compositionality. In: Advances in Neural Information Processing Systems, pp. 3111–3119 (2013)
16. Myers, B.A., Stylos, J.: Improving API usability. Commun. ACM **59**(6), 62–69 (2016)
17. Pedregosa, F., et al.: Scikit-learn: machine learning in python. J. Mach. Learn. Res. **12**(Oct), 2825–2830 (2011)
18. Ramshaw, L.A., Marcus, M.P.: Text chunking using transformation-based learning. In: Armstrong, S., Church, K., Isabelle, P., Manzi, S., Tzoukermann, E., Yarowsky, D. (eds.) Natural Language Processing Using Very Large Corpora. TLTB, vol. 11, pp. 157–176. Springer, Dordrecht (1999). https://doi.org/10.1007/978-94-017-2390-9_10
19. Rehurek, R., Sojka, P.: Gensim-python framework for vector space modelling. NLP Centre, Faculty of Informatics, Masaryk University, Brno, Czech Republic, vol. 3, no. 2 (2011)
20. Robillard, M.P., Deline, R.: A field study of API learning obstacles. Empir. Softw. Eng. **16**(6), 703–732 (2011)
21. Rong, X.: Word2vec parameter learning explained. arXiv preprint arXiv:1411.2738 (2014)
22. Schmidt, M., Le Roux, N., Bach, F.: Minimizing finite sums with the stochastic average gradient. Math. Program. **162**(1–2), 83–112 (2017)
23. Thomas, R., et al.: Architectural styles and the design of network-based software architectures. University of California, Irvine (2000)
24. Yang, J., Wittern, E., Ying, A.T., Dolby, J., Tan, L.: Automatically extracting web API specifications from HTML documentation. arXiv preprint arXiv:1801.08928 (2018)
25. Zhong, H., Zhang, L., Xie, T., Mei, H.: Inferring resource specifications from natural language API documentation. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, pp. 307–318 (2009)