API Learning: Applying Machine Learning to Manage the Rise of API Economy

Mehdi Bahrami, Junhee Park, Lei Liu, Wei-Peng Chen

Fujitsu Laboratories of America, Inc.

Sunnyvale, California

{mbahrami;jpark;lliu;wchen}@us.fujitsu.com

ABSTRACT

Application Programming Interface (API) exposes data and functions of a software application to third-party users. In digital business, API economy is one of the key component for determining the value of provided services. With the rise in number of publicly available APIs, understanding each API endpoint manually is not only labor intensive but it is also an error prone task for software engineers. Due to the complexity of understanding the sheer number of APIs, it is difficult for software developers to find the best possible API combinations (i.e. API Mashups). In this demonstration, we introduce API Learning platform which employs machine-learning based technologies to efficiently search APIs, validate APIs, and generate API mashups. These technologies enable a machine to automatically generate machine-readable API specification from API documentations, understand variety of APIs, validate extracted information through automated API validation, and finally recommend API mashups for a specific purpose. As of now, API Learning platform collected over 14,000 API documentations and generates a machine readable format for REST APIs with an accuracy of 84%. The proposed demo prototype shows how it enables users to quickly find relevant APIs, automatically verify API availability, and get the best possible API mashup recommendations.

KEYWORDS

API; Machine Learning; REST; Mashup; Automation

1 INTRODUCTION

Web APIs or RESTful (REpresentational State Transfer) [11] APIs have been widely used in cloud-based services due to its inherent flexibility and low overhead. API functions or data from APIs can be efficiently integrated into a software application. In the era of digital business, employing APIs is the key element of various successful businesses. With the rise in number of APIs in the market, manual understanding of each API functionality is not only labor intensive but also it is an error prone task for software engineers. Finally, due to the complexity of heterogeneous structure of API documentations and the sheer number of APIs, it is a burdensome task for software engineers to find the most suitable API Mashups.

Motivation. The process of utilizing APIs in a software application is not a trivial task for developers. According to *ProgrammableWeb*

there are more than 18,503 public APIs. Understanding a large number of APIs with heterogeneous structures is a significant challenge for software engineers to select the best APIs for their software projects. By having a machine-readable API specification, software engineers can efficiently utilize and compare API capabilities without reading API documentations. It also opens door to a set of new artifacts, such as API mashup service recommendations that combines multiple APIs together. To the best of our knowledge, only limited APIs offer machine-readable API specifications and this is the first study that generates machine-readable API specifications from API documentations. Although some initial discussions have been started on Swagger and WebSchema as an extension proposal at W3C, a dominant standard for machine-readable API specifications have not existed yet.

Contribution. In this demonstration¹, we proposed a platform that uses various machine learning and Natural Language Processing (NLP) algorithms to produce machine-readable API Specifications. The proposed platform has developed an ensemble of novel technologies which enables software engineers to easily use and retrieve information about heterogeneous API documentations. The proposed approach understands a variety of API documentations and produces a machine-readable API specification based on OpenAPI Specification (OAS)². We collected a large annotated dataset of information extraction evaluation. In addition to this manual annotation, we also automated the process of validating the extracted information, generated from the proposed *API Learning* methods. Finally, the output of the proposed platform has been processed by an API mashup service recommendation that list the best possible API combinations based on their input/output similarity.

2 PROPOSED APPROACH

Our goal is to extract information from API documentations that are published by API providers. Figure 1 shows the key components of the proposed approach. In this figure, we have several components for both data collection and data processing as follows.

Web Crawler. This component collects a large number of API documentations from the web as described in [1]. First, the volume of data allows the proposed platform to learn heterogeneous structures in a large number of API documentations from various data sources. Second, it improves the accuracy of information extraction by learning a variety of data. The collection of the text content of all HTML pages which are collected by web-crawler produces *API Corpus*. The *API Corpus* consist of a large clean text file (e.g., without HTML and script codes) for each API.

This paper is published under the Creative Commons Attribution 4.0 International (CC BY 4.0) license. Authors reserve their rights to disseminate the work on their personal and corporate Web sites with the appropriate attribution.

WWW '18 Companion, April 23-27, 2018, Lyon, France

© 2018 IW3C2 (International World Wide Web Conference Committee), published under Creative Commons CC BY 4.0 License.

ACM ISBN 978-1-4503-5640-4/18/04.

<https://doi.org/10.1145/3184558.3186966>

¹ Screencast of the demo is available at: <https://youtu.be/J6qOOF3nIG8>

² <https://github.com/OAI/OpenAPI-Specification>

Figure 1: The proposed platform of API Learning

REST API Filter. This component to filter out some pages which are irrelevant to the target APIs (RESTful APIs in this study). The REST API filter employs a logistic regression model [4] to detect REST category APIs. We use *REST Keyword Frequency* to train the model, and we use this component to generate a set of features for each API documentation in *API Corpus*. We consider major REST keywords, such as *POST*, *GET*, *DELETE*, *endpoint* and etc, to extract a set of term frequency of these keywords $TF(keyword)$ from each page. Therefore, we generate an array of X REST keywords for each web page, where X_i^T represents the term frequency of i th REST keyword in T th API documentation page and the cost function of the logistic regression can be defined as follows.

$$\min_{w, c} \frac{1}{2} w^T w + C \sum_{i=1}^n \log(\exp(-y_i(X_i^T w + c)) + 1) \quad (1)$$

REST API Language Model. This component reads *API Corpus* and it generates a language model of API documentations. The model allows the platform to understand the semantic definition of each word in *API Corpus*. *API Language Model* is a statistical model which is defined as a neural language model [3] that shows probability distribution on all sentences in *API Corpus*. It uses embedding of words to predict word sequences. The cosine similarity of two words, W_1 and W_2 , in word to vector provides a semantic definition of n words in *API Corpus*, which is defined as follows.

$$\text{Sim} = \frac{W_1 \cdot W_2}{\|W_1\|_2 \|W_2\|_2} = \frac{\sum_{i=1}^n W_{1,i} W_{2,i}}{\sqrt{\sum_{i=1}^n W_{1,i}^2} \sqrt{\sum_{i=1}^n W_{2,i}^2}} \quad (2)$$

This function extends a *pattern* in each iteration.

API Learning. We define a set of rule-based regular expression tasks for extracting information from *API Corpus*. Each task includes an initial regular expression *pattern* which is defined by an expert (i.e., a software developer who can define a simple pattern to extract endpoint). For example, a task may consists of API endpoints extraction, and another task may consist of a pattern that extracts API endpoint's parameters. Each *pattern* can be improved based on a new approach which is inspired by transformation-based learning algorithm [9], [6]. The proposed approach extends acceptance of information *extended pattern*, and then shrink it to find the best *pattern* to extract variety of information. In addition, the proposed

approach uses *REST API Language Model* to extend each word in a *pattern* based on a semantic definition of a word. Each task in *Tasks* has an initial regular expression. For instance, an expert recommends an initial regular expression to extract the information (e.g., $pattern_1$) for extracting API endpoints and another task (e.g., $pattern_2$) for detecting a table that describes a list of input parameters. Different API descriptions may need different *patterns* to extract the same concept. For instance, Facebook uses the terminology of *fields* to describe the input parameters of an endpoint³; but Google uses *parameters* to describe the input parameters of an endpoint⁴. The *REST API Language Model* enables the platform to understand semantic definition of each word in each *pattern*. For example, the semantic definition of *fields* can be *parameter*, and a *pattern* that consist of *field* ($pattern_1 = field$) can be extended to $pattern_2 = field|parameter$. In addition, we need to learn different regular expression to extract the same information with different structures. The proposed approach i) expends the acceptance of initial *RE*, and then ii) it reduces the *RE* pattern to match only positive cases sample cases. The following tasks are the major *IE Tasks*, such as *host address*, *base path address*, *endpoint URI*, and *parameters attribute*. The concatenation of the host address, base path and endpoint address provide a unique URL that provide a specific function. For example, "POST https://api.twitter.com/1.1/statuses/update.json" is a unique URL to update user's current status as known as Tweeting. In this API endpoint function, *POST* is the *http function* and it shows how to call the endpoint, "https://api.twitter.com/" is the *host address*, "1.1/" is the *path base address* and "statuses/update.json" is the *endpoint address*. In order to solve the optimization problem, we have to compute the total number of acceptance cases and sample rejection cases from samples S in respect to each API, $\mathcal{A}$.

$$\max_{x \in \mathbb{D}} f_P(|s_P(x_{\mathcal{A}})| - |s_N(x_{\mathcal{A}})|) \quad (3)$$

$\mathbb{D}$ denotes API documentations that can be retrieved from *API Corpus*. s_P denotes a set of positive examples of language $L_{\mathcal{A}}$ in respect to the target API, $\mathcal{A}$; and $s_N \notin L_{\mathcal{A}}$ where s_N denotes negative examples of language L in respect to the target API, $\mathcal{A}$. f_P defines a function that uses set of regular expression rule patterns P , that accepts s_P and rejects s_N in $L_{\mathcal{A}}$. In the proposed approach our goal is to maximize f_P . In the proposed approach, each *pattern* is improved by learning from different positive and negative cases. The proposed approach extends acceptance of information (*extended pattern*, and then shrink it to find the best *pattern* to extract variety of information. In addition, the proposed approach uses *REST API Language Model* to extend each word to semantic definition of a word of a *pattern* of *RE*. In the proposed approach our goal is to maximize f_P as shown in Equation 3. Unlike Li et al. [6] and Bartoli et al.[2] the proposed approach takes f_P and it generates a new *pattern* ($f_{P_{i+1}}$) where $f_P \subseteq f_{P_{i+1}}$ or $f_P \not\subseteq f_{P_{i+1}}$. In another word, the new generation may improve the acceptance rate and decrease the rejection rate, even if the new *pattern* is not a subset of previous *pattern*. The final output of the platform is a machine-readable format of API documentation as a JSON file based on the OAS.

Automated API Connectivity Validation. In order to evaluate our information extraction result, we incorporated automated

³i.e., <https://developers.facebook.com/docs/graph-api/reference/v2.10/album>

⁴i.e., <https://developers.google.com/web/api/rest>

Figure 2: Automated API validation results

API validation service where we send native API call for each endpoint and validate its availability. By utilizing OAS-based API learning results, automated API validation service extracts information, such as host, base-path, endpoint, and parameter requirement for each API. This information is used to formulate native API calls. Once the request is generated, we send the request to the corresponding API host server and retrieve its response. Based on HTTP status code, we separate all 5xx errors and select 4xx errors, such as 404, and 408 (i.e., Not Found, and Request Timeout, respectively), denote as *error*, for all other responses, we indicate as *passed validation*.

API Mashup. Based on the OAS-based API learning results, we firstly parse the information from the JSON file of each API, and then based on the category keywords and API descriptions, we cluster 1,991 APIs into different groups. Compared to our previous work which only considers APIs in the health care domain, in this work, the APIs and mashups covers multiple and different vertical domains. After clustering, APIs within a same group are similar, which are not involved for mashup computation. In order words, the mashup only considers the combination of dissimilar APIs that are clustered in different groups. For those APIs, we obtain input/output parameter descriptions from API Learning results, and then we measure the similarity score of the input/output parameter descriptions. APIs with higher similarity are more likely to be combined as a mashup because their inputs or outputs are similar.

3 DEMONSTRATION

The first component is web-crawler that collects a large number of API documentation web-pages. We used several sources, such as ProgrammableWeb, API Harmony, RapidAPI, and etc to collect a comprehensive list of APIs as a pointer list to API Documentations. The pointer list consists of more than 14,000 API links. In this demonstration, we target REST APIs. We used *Scrapy*⁵ for implementation of the parallel web-crawler. Each agent crawls a web-page from the pointer list and collects all subsequent pages. We collected 256,583 web-pages with the size of 15GB for more than 14,000 public APIs. However, not all of the public APIs categorized as REST API group; therefore according to *REST API Filter*, we have 5,320 REST API which represents 37.25% APIs from *API Corpus*.

⁵Scrapy is an open-source project and it is available at <https://scrapy.org>

API Language Model. We used a Continuous Bag of Words (CBOW) as known as Word2Vec model [8] and [7] to generate *API Language Model* which is an unsupervised learning method. By providing a set of positive and/or negative words, we can inquiry the model to find similar words. The parameter of Word2Vec is described by Rong in [10] which we chose 300 for the window size which represents the maximum windows distance between a selected word and a predicted word within a sentence. We considered the min frequency of 5 for considering a word for training purpose.

API Learning. We defined several tasks with initial pattern of *RE* as described in Section 2. After a task learns a pattern (P_{final}), then we apply the task to API documentations. 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 based on OAS format which includes: i) API metadata, ii) API endpoints, and iii) different types of tables (e.g., a table may consist of parameters of an endpoint). We evaluated the correctness of extracted information manually for 2,929 API endpoints and 1,780 extracted tables. The evaluation is a comparison between original HTML API documentation and the generated JSON file as output to mark correctness of information extraction. We compare extracted information against original source and annotate them as correct or incorrect extracted information. In this evaluation, we considered extracted table from 200 APIs that consist of 1,780 extracted different tables (e.g., parameter table that shows parameter information), and extracted endpoints for 350 APIs that consist of 2,929 endpoints. To the best of our knowledge, we produced a number of annotation of extracted information with a practical use case of API documentation for the first time. Our evaluation results shows that we have extracted 86.75% of API endpoints and 81.29% of table correctly. The average of information extraction accuracy⁶ is 84.02%.

API Validation. As shown in Figure 2, we have evaluated 1,923 APIs and 54,873 Endpoints. Among those APIs, 1,453 APIs (76%) are active (at least an API has one active endpoint), 470 APIs (24%) are inactive. From those active APIs, 53,451 endpoints (82%) has active endpoints and 11,451 endpoints (18%) are inactive. This evaluation result is incorporated into our platform as shown in Figure 3c. Endpoints that passed the evaluation would have green check marks whereas others would indicate their failure with the reason (e.g., Connection Timeout, Not Found, etc.).

API Mashup. As shown in Figure 3b, the platform provides a set of API mashups as well as multiple metrics to facilitate developers to select a desired API mashup. The metrics include (1) a similarity score which a quantitative value shows how related for the APIs in a mashup; (2) Found an API mashup record in ProgrammableWeb (Yes/No) which indicates whether the given mashup has been manually added by developers based on human knowledge; (3) Linked in API Harmony (Yes/No) which shows whether there is a record of API mashup in the API Harmony website; (4) Found in IFTTT dataset (Yes/No) which presents whether a record of API mashup found in IFTTT; and (5) Found in Github projects (Yes/No) which indicates whether there is a Github project that utilized API mashups. By using the aforementioned solution of API mashup in Section 2, we obtained ~1,500,000 possible API mashups with similarity scores

⁶Accuracy = C/S where C is the total number of correct extracted information and S represents the total number of extracted information.

Figure 3: A screenshot of API Directory GUI

larger than 50%. We also collected a large annotations of APIs which were added by human developers in ProgrammableWeb with 6,182 API mashups, 11,242 API pairs linked in API Harmony, 2,451 API mashups from IFTTT, and 2,535 API mashups from Github projects. The annotation is used as a cross-validation of API mashups against our proposed similarity scores.

GUI. In addition, we added a component that provides a GUI¹ as front-end and an in-memory database (IMDB) [5] as a back-end to show the API Learning results. The GUI is a user-friendly interface that gives a suggestion list to find a machine-readable OAS based JSON files and returns the list of search results when it searches a keyword from 4,561 OAS-based JSON files. As shown in Figure 3, it returns the list of APIs (Figure 3a), shows the metadata of a selected API, the list of endpoints and parameters, the automatic validation of an endpoint, original API source documentation which is collected by the web-crawler (Figure 3c), a live on-demand validation (Figure 3c), and API Mashup recommendations (Figure 3b) that offers the list of all possible API combinations.

4 CONCLUSION

API documentation carries the key information of an API. More advanced APIs have more complex structure as well as more complicated API documentations. Although some machine-readable API specifications are available, such as Open API Specification and YAML, still only limited popular APIs offer a machine-readable format of their APIs. In this paper, we introduced a novel platform that processes a large amount of API documentations that consist of more than 14,000 APIs. The proposed platform processed all

collected HTML page. It generates an *API Language Model* to define the semantic definition of a word which is used in a transformation-based learning algorithm to extract information from *API Corpus*. The platform generates an Open API Specification-based JSON file (machine-readable API) for a set of API documentations. It also incorporates both manual annotation and automated API validation to evaluate correctness of generated machine-readable APIs. It offers API mashup recommendation which provides a list of possible API combinations. We show multiple metrics including both the machine generated similarity scores and the existing human knowledge to evaluate the API mashup results.

REFERENCES

- [1] Mehdi Bahrami, Mukesh Singhal, and Zixuan Zhuang. 2015. A cloud-based web crawler architecture. In *Intelligence in Next Generation Networks (ICIN), 2015 18th International Conference on*. IEEE, 216–223.
- [2] Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. 2016. Inference of regular expressions for text extraction from examples. *IEEE Transactions on Knowledge and Data Engineering* 28, 5 (2016), 1217–1230.
- [3] Y. Goldberg and O. Levy. 2014. word2vec Explained: deriving Mikolov et al.'s negative-sampling word-embedding method. *preprint arXiv:1402.3722* (2014).
- [4] David W Hosmer Jr, Stanley Lemeshow, and Rodney X Sturdivant. 2013. *Applied logistic regression*. Vol. 398. John Wiley & Sons.
- [5] Tirthankar Lahiri, M. Neimat, and S. Folkman. 2013. Oracle TimesTen: An In-Memory Database for Enterprise Applications. *IEEE Data Eng. Bull.* 36, 2 (2013), 6–13.
- [6] Yunyao Li, Rajasekar Krishnamurthy, Sriram Raghavan, Shivakumar Vaithyanathan, and HV Jagadish. 2008. Regular expression learning for information extraction. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 21–30.
- [7] T. Mikolov, K. Chen, G. Corrado, and J. Dean. 2013. Efficient estimation of word representations in vector space. *preprint arXiv:1301.3781* (2013).
- [8] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.
- [9] Lance A Ramshaw and Mitchell P Marcus. 1999. Text chunking using transformation-based learning. In *Natural language processing using very large corpora*. Springer, 157–176.
- [10] Xin Rong. 2014. word2vec parameter learning explained. *arXiv preprint arXiv:1411.2738* (2014).
- [11] Roy Thomas et al. 2000. Architectural styles and the design of network-based software architectures. *Irvine: University of California* (2000).

Mehdi Bahrami received a Ph.D. in Computer Science from the University of California, Merced. His current research focuses on natural language processing for Hyperconnected and Cloud-based APIs.

Junhee Park received a Ph.D. in Computer Science from Georgia Institute of Technology. His research focuses on distributed systems and performance/scalability analysis.

Lei Liu received a Ph.D. degree from Beijing University of Posts and Telecommunications (BUPT), Beijing, China. His main research focuses on the Internet technologies.

Wei-Peng Chen is leading Cyber-Physical System Lab at FLA. He received a Ph.D. in Computer Science from University of Illinois at Urbana-Champaign. His research focuses on Machine Learning and IoT.