2017 16th IEEE International Conference on Machine Learning and Applications

Automatic Generation and Recommendation for API Mashups

Qinghan Xue*†, Lei Liu*, Weipeng Chen* and Mooi Choo Chuah†
*Fujitsu Laboratories of America, Inc. Sunnyvale, CA 94085
Email: lliu@us.fujitsu.com; Wei-Peng.Chen@us.fujitsu.com
†Department of Computer Science and Engineering, Lehigh University, Bethlehem, PA 18015
Email: qix213@lehigh.edu; chuah@cse.lehigh.edu

***Abstract*—Until today, finding the most suitable $APIs$ to use in an application was burdensome, requiring manual and time-consuming searches across a diverse set of websites, in particular regarding how multiple $APIs$ could be combined and worked together (i.e. API mashups). In this paper, we propose a new method to automatically generate API mashups through real-world data collection, text mining and natural language processing (NLP) techniques. The generated API mashups are further ranked and recommended to developers based on a quantitative indicator of whether the given API mashup is plausible. To evaluate the overall accuracy of the proposed method, we use the generated API mashups to train several machine learning and deep learning models, and then use an independent mashup dataset collected from Github projects for testing. The experimental results show that our proposed method is feasible and accurate for automatic API mashup generation and recommendation.**

***Index Terms*—API Mashups, NLP, machine learning, deep learning**

I. Introduction

Application programming interfaces ($APIs$) are extensively used in modern software development, which not only speed up development but also save resources for developers. However, correctly and efficiently choosing and using $APIs$ is a difficult and time-consuming task because there are a huge number of public $APIs$ nowadays. For example, ProgrammableWeb [1], which is a public API-related repository, includes the information of 17,682 $APIs$ in its directory and this number is ever-increasing. Therefore, it is extremely burdensome for human to comprehend all these $APIs$ and select correct $APIs$ for a specified software development task, especially in the case where multiple $APIs$ have to be combined together to provide new or value-added services. How to find and choose an appropriate API combination, which is also known as API mashup [2], thus becomes a critical issue for API economy and service-oriented computing technology.

While programmers often use search engines such as Google and Bing to learn about existing $APIs$, general web search engines are not designed to specifically support programming tasks, which are often inefficient and inaccurate for development, regarding, in particular, how to find API mashups. To address this issue, a number of techniques [3], [4] have been proposed so far to facilitate the usage of $APIs$ by creating API summarizations, mining code or showing common API sequences. However, as these solutions require expensive parameter tuning and the returned set of API calls can be very large and highly tedious, these solutions are difficult to be used in practice.

In this paper, we present a new approach for automatic API mashups generation based on real-world data collection, text mining and natural language processing (NLP) techniques. Meanwhile, each generated API mashup is associated with a quantitative indicator which shows the relative possibility that the given API mashup is plausible. In turn, the generated API mashups can be ranked and recommended so that developers can easily select the most plausible one. In addition, the accuracy of the generated API mashups is evaluated by multiple machine learning and deep learning models with an independent testing dataset collected from multiple Github projects. The experimental results validate that the overall solution is feasible, accurate and widely applicable.

The major contributions of this paper are shown as follows:

- We design an automatic API mashups generation approach based on real-world data, text mining and NLP, where the generated API mashups are ranked by a quantitative indicator and recommended to the developers.
- We propose a similarity measurement method for choosing similar $APIs$ for mashups creation, which considers both the relativity and popularity of $APIs$.
- We adapt multiple machine learning and deep learning models based on the generated API mashups and conduct systematic experiments to evaluate those models.

The rest of the paper is organized as follows. Section II discusses related works. Section III describes the method and procedure to automatically generate API mashups. Section IV evaluates the accuracy of the results with several machine learning models. Section V concludes the paper with discussions of our future works.

II. Related Work

Modern programming frameworks come with large libraries and diverse API ecosystems. While developers can consume them and make use of additional features offered by those ecosystems, understanding the API of an unfamiliar library can be a significant obstacle [4], [5] since the number of public $APIs$ is huge, and API documentations are often incomplete and ambiguous [6].

Fortunately, an opportunity to address the API selection issue has arisen out of the simultaneous growth in the amount of large scale data mining and machine learning technologies. This confluence has enabled the development of API mining methods [7], which aim to automatically extract a set of API patterns (i.e. characterize how an API is used). According to these methods, developers can choose a s

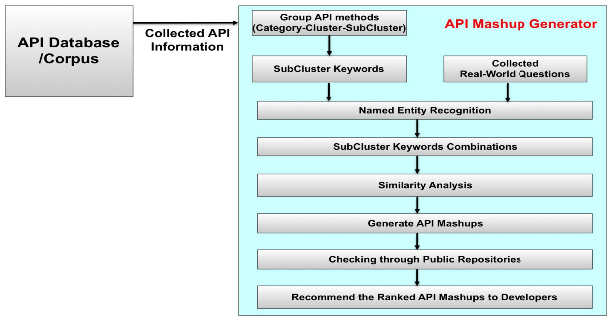

API Database /Corpus
Collected API Information
API Mashup Generator
Group API methods (Category-Cluster-SubCluster)
SubCluster Keywords
Collected Real-World Questions
Named Entity Recognition
SubCluster Keywords Combinations
Similarity Analysis
Generate API Mashups
Checking through Public Repositories
Recommend the Ranked API Mashups to Developers

Fig. 1. Workflow of the Proposed Method

$APIs$, and compose them to form new services, which are usually called mashups. The mashup technology allows different $APIs$ to compose together to provide more complex functionalities. As the mashup services thrive, a number of mashups and API repositories have emerged. For example, Li et al. [8] proposed a probabilistic model to recommend a list of $APIs$ that could be used to compose a required mashup. Meanwhile, Xu et al. [9] incorporated social relationships among users into recommending suitable services for mashups. In addition Wei et al. in [10] present a system that can group mashups with similar functionalities by mining API usage patterns. Among all those works, the most famous are ProgrammableWeb [1], where more than 6,700 mashup services are currently available.

Although those works are useful in rapidly generating API mashups, when the number of available $APIs$ becomes huge, the selection of appropriate $APIs$ for mashups becomes intractable. Thus, to alleviate this problem, in [11] E. Wittern et al. proposed a platform named API Harmony that can show the relations among different $APIs$ by analyzing Github projects. If multiple $APIs$ can be found in a same Github project, a link is added among these $APIs$ which indicates a potential API mashup. Meanwhile, Gu et al. described DeepAPI in [12], which is a deep-learning based method generating relevant API usage sequences based on an API-related natural language query. While existing achievements [13], [14] on deep learning models are encouraging, they are merely constructed based on a small number of public $APIs$ with limited features, and thus can only handle simple queries with limited applicability.

Therefore, in this paper, we design a widely-applicable method to automatically generate API mashups by capturing information from not only multiple features of $APIs$ (e.g., descriptions, data formats, protocols, characteristics, etc) but also the real-world data.

III. Proposed Method

In this section, we introduce the solution for API mashup generation and recommendation. Meanwhile, experimental results are presented to show the overall feasibility and efficiency of the proposed solution.

A. Overall Procedure

The overall procedure of the proposed solution is depicted in Fig 1. The API mashup generator collects and reads API information from some public API repositories. In this work, the information is captured from ProgrammableWeb [1].

1) API Clustering: In API clustering phase, we first collect all the textual information of $APIs$ (Fig 2) including their categories, primary category keywords, second category keywords and descriptions, etc. from ProgrammableWeb [1]. Then, we group $APIs$ into a tree structure as follows:

(i) Generate Categories for APIs: We first compute the frequency of every keyword in both category and primary category fields of $APIs$. After that, we compute the number of related $APIs$ for each category keyword. Then, we measure the similarity among different keywords:

(a) using word embedding methods (e.g., Google Word2Vec [15] or spaCy [16]) to transform each keyword into a word vector;

(b) computing the similarity scores among different category keyword vectors.

According to the frequency, number of related $APIs$ and the similarity measurements, we finally select top M keywords as categories keywords, for example, "Healthcare" as shown in Fig 3.

(ii) Generate Clusters for Each Category: We first compute the frequency of every keyword in $APIs'$ secondary category field. Then, we repeat the same process as III.A.1)(i). The main difference is that for cluster generation, we compute the similarity scores among different cluster keyword vectors and the corresponding category keyword vector. According to the frequency, number of related $APIs$ and the similarity measurements, we finally select top N keywords as clusters keywords for a specific category, for example, "Patient", "Hospital", etc. for "Healthcare" category as shown in Fig 3.

(iii) Generate Sub-clusters for Each Cluster: We first group $APIs$ into different sub-clusters based on their descriptions by

(a) removing stop words and symbols in $APIs'$ descriptions;

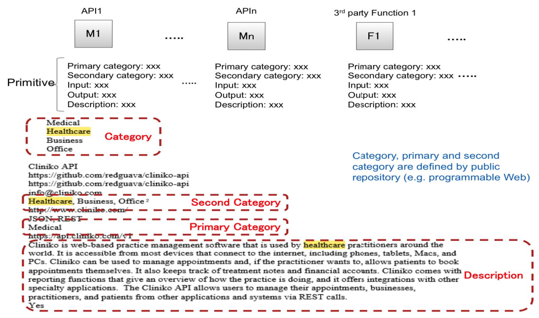
(b) chopping descriptions into keywords and measuring every keyword type;

(c) generating word stem for every noun keyword and computing its frequency.

After that, we measure the similarity among the refined keywords and the corresponding cluster keywords (similar as III.A.1)(i)). According to the frequency, number of related $APIs$ and the similarity measurements, we finally select top K_i keywords as sub-clusters keywords for i^{th} cluster, for example, "Diagnose", "Treatment", etc. for the "Patient" cluster as shown in Fig 3.

As a result, all $APIs$ can be grouped into related leaf nodes (i.e. sub-clusters) according to their textual information, where all processes and results can be computed and stored offline.

2) API Mashups Generation: After clustering $APIs$ into sub-clusters in the tree structure, we then create API mashups



Medical
Healthcare
Business
Office
Cliniko API
https://github.com/redguava/cliniko-api
https://github.com/redguava/cliniko-api
info@cliniko.com
Healthcare, Business, Office [2]
http://www.cliniko.com/
JSON, REST
Medical
https://api.cliniko.com/v1
Cliniko is web-based practice management software that is used by healthcare practitioners around the world. It is accessible from most devices that connect to the internet, including phones, tablets, Macs, and PCs. Cliniko can be used to manage appointments and, if the practitioner wants to, allows patients to book appointments themselves. It also keeps track of treatment notes and financial accounts. Cliniko comes with reporting functions that give an overview of how the practice is doing, and it offers integrations with other specialty applications. The Cliniko API allows users to manage their appointments, businesses, practitioners, and patients from other applications and systems via REST calls.
Yes

Fig. 2. API Data Information

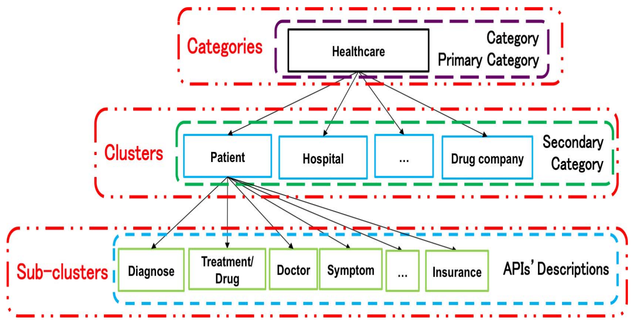


Fig. 3. Tree Structure

based on the combinations of sub-cluster keywords with real-world data collection.

(i) Collect Real-world Data: In order to better understand which combinations are most useful for developers, for each category we collect real-world data from web blogs by using web crawlers. For example, for healthcare category, we can extract patients' questions from professional healthcare websites such as Medhelp [17], Patientslikeme [18], etc.

(ii) Sub-cluster Keyword Combinations: Based on the collected questions and the generated sub-cluster keywords (III.A.1)(iii)), we generate sub-cluster keywords combinations for every real-world question by using some existing knowledge based systems [19], [20], which contain pre-defined name entities (Fig 4). For example, we extract a set of aliases potential mention strings that can refer to an entity based on Wikipedia descriptions.

(iii) Generate API Mashups: Based on the generated sub-cluster keywords combinations (where each sub-cluster keyword represents a list of $APIs$), we use similarity analysis to check if $APIs$ in different sub-clusters can be combined together. To this end, we firstly collect and create a whole description for each API. This whole description is a compressive summary for the given API, which may include, but not limited to API keywords, brief introduction of API, protocol formats, input&output descriptions, etc. For the similarity analysis, we first generate word vectors based on $APIs'$ descriptions. Then, we measure similarity scores among different $APIs$ to get possible API mashups as follows:

(a) we first separate the whole descriptions into sentences;

(b) then, we remove stop words and symbols in each sentence;

(c) next, we use word embedding methods (e.g., Google Word2Vec [15] or spaCy [16]) to transform each sentence into a word vector and measure the similarity scores among word vectors, as an example shown in Equation(1).

$$\begin{aligned} similarity_{score} = (&similarity(API1_{s1}, \\ &API2_{s1}) + similarity(API1_{s1}, API2_{s2}))/2 \end{aligned} \tag{1}$$

Where, we assume the whole description of $API1$ contains one sentence $API1_{s1}$ and the whole description of $API2$ contains two sentences $(API2_{s1}, API2_{s2})$.

3) API Mashups Weighting: After API mashups are generated, we can check through some public repositories to further rank the results for API mashups with same similarity scores.

(i) We use the corresponding $APIs$ in an API mashup to search the API Harmony [11]. If those $APIs$ are linked, such a mashup is marked with a higher priority.

(ii) We use the corresponding $APIs$ in an API mashup to search $Github$. If a project can be searched, then this mashup is marked with a higher priority. For example, $ValidicAPI$ and $FitbitAPI$ can be searched in some $Github$ projects, so the API mashup with both $ValidicAPI$ and $FitbitAPI$ is marked with a higher priority.

(iii) We use the corresponding $APIs$ in an API mashup to search ProgrammableWeb [1]. If the $APIs$ are found to be related in ProgrammableWeb [1], this mashup is marked with a higher priority. For example, ProgrammableWeb [1] shows that $HumanWellnessAPI$ is one of the related $APIs$ of $WahooFitnessAPI$. So if an API mashup is generated including both $HumanWellnessAPI$ and $WahooFitnessAPI$, this mashup is marked with a higher priority.

Thus, for a given API mashup as the input, if we can find search results in any one of the above public repositories, this API mashup is marked with a higher priority for recommendation to break the tie of similarity scores.

4) API Mashup Recommendation: Based on the similarity and weighting analysis, we can get a quantitative indicator (i.e. similarity score of the whole descriptions for $APIs$ in different sub-clusters and the corresponding priority) for each generated API mashup, and thus, the API mashups can be ranked based on this indicator. Note that the $APIs$ involved in the similarity computation are from different sub-clusters and the sub-cluster keyword combination is obtained from real-world questions. Therefore, if the similarity score for the whole description of the $APIs$ are higher, these $APIs$ have more similar features (e.g., protocol formats, inputs&outputs, etc.) which indicate that they are more likely to be combined together as a plausible API mashup. When API mashups are generated and ranked, they are saved in a local repository, together with their associated keyword combinations. When a developer's query is received, we firstly find the keyword combination which is closest for satisfying the developer's re-

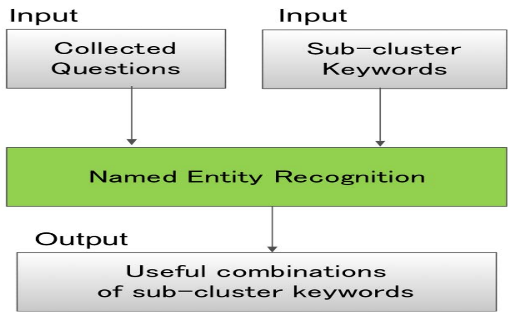
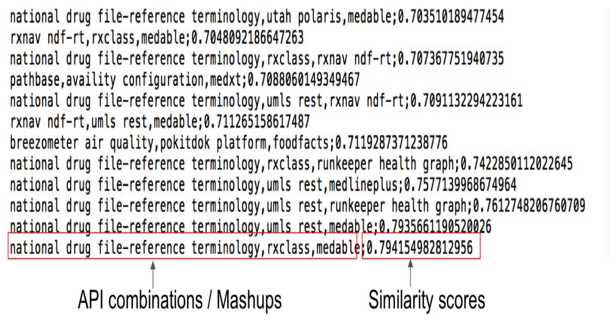


Fig. 4. Sub-cluster Combination



```
national drug file-reference terminology,utah polaris,medable;0.703510189477454
rxnav ndf-rt,rxclass,medable;0.7048092186647263
national drug file-reference terminology,rxclass,rxnav ndf-rt;0.707367751940735
pathbase,availity configuration,medxt;0.7088060149349467
national drug file-reference terminology,umls rest,rxnav ndf-rt;0.7091132294223161
rxnav ndf-rt,umls rest,medable;0.711265158617487
breezometer air quality,pokitdok platform,foodfacts;0.7119287371238776
national drug file-reference terminology,rxclass,runkeeper health graph;0.7422850112022645
national drug file-reference terminology,umls rest,medlineplus;0.7577139968674964
national drug file-reference terminology,umls rest,runkeeper health graph;0.7612748206760709
national drug file-reference terminology,umls rest,medable;0.7935661190520026
national drug file-reference terminology,rxclass,medable;0.794154982812956
```

Fig. 5. API Mashup Results

quirement. Then, we can instantly retrieve the generated API mashups from the local repository and directly recommend them to the developer in a ranked order.

B. API Mashup Results

By using the above procedure, the proposed method can generate API mashups for recommendations. Due to space limitation, in this paper, we take the healthcare category as an example to present our results. The reason why we select healthcare domain is because there are a large number of $APIs$ that are related with healthcare. In the healthcare domain, we collect comprehensive information of 417 healthcare related $APIs$ from ProgrammableWeb [1], and group them into 190 different sub-clusters. Then, we collect 50,000 clinical questions from the websites like Medhelp [17], Patientslikeme [18], etc. Next, we use a tool named $SNOMED$ CT Text Analyzer [21] for named entity recognition and obtained 10,000 keyword combinations based on the collected questions. Finally, we generate possible API mashups, where 569 are mashups with 2 $APIs$ and 662 are mashups with 3 $APIs$. Among all the generated API mashups, 68% are with the similarity score over 0.5. A sample result is shown in Fig 5, where each API mashup contains 3 $APIs$. From the results, we can see that there is a high probability that plausible API mashups will have high similarity scores.

IV. Performance Evaluation

In this section, we conduct experiments to evaluate whether our generated API mashups are accurate or not.

A. Experimental Settings

The method is to use our generated API mashups to train several learning models, and use an independent API mashup dataset collected from Github to test those models. We selected 8 features from the API whole description that we collected from ProgrammableWeb [1] for generating the learning models, which includes 6 keyword based features: category, primary category, secondary category, links, page title, protocol formats and 2 sentence based features: simple description and input & output descriptions.

B. Learning Models

Four different learning models are utilized in this paper for the performance evaluation.

(i) Traditional Learning Models:

- Decision Tree (DT) [22]: this is a flowchart graph or diagram that helps explore all of the decision alternatives and their possible outcomes.
- Random Forest (RF) [23]: this is an ensemble learning method for classification, regression and other tasks, that operates by constructing a multitude of decision trees at training time and outputting the class based on voting.

(ii) Deep Learning Models:

- Convolutional Neural Network (CNN) [24]: this is a machine learning technique which is modeled after the brain structure. It comprises of a network of learning units called neurons, where the input signals will be converted into corresponding output signals, forming the basis of automated recognition.
- Long Short-term Memory ($LSTM$) [25], [26]: this is a particular type of Recurrent Neural Network (RNN) [27] that works slightly better in practice, owing to its more powerful update equation and some appealing backpropagation dynamics. It is also capable of learning long-term dependencies, which can remember information for long periods of time.

C. Experimental Evaluation & Discussions

We conduct two experiments (Exp1 & Exp2), where Exp1 is to study the impact of different learning models on API mashups recommendation. In the second experiment (Exp2), we evaluate how different word embedding methods can impact the recommendation results. All our experiments are conducted on a Linux machine with an Intel Core i7-6700 CPU running at 3.40GHz and 31.3GiB memory.

1) Impact of Different Learning Models:

Exp1: In this experiment, we first extract 569 healthcare related mashups with 2 $APIs$ that are generated using the proposed methods as the training data. Then, we use Google's trained Word2Vec model to generate the word vector for every selected feature. Based on the similarity scores of the whole descriptions for each API mashup, we set threshold as 0.5

to label the data (i.e. if the score is larger than 0.5, we label the given API mashup as plausible; otherwise, we label it as implausible). Next, we use different classifiers including DT, RF, CNN and $LSTM$ to evaluate the prediction performance, where we use 2 layers for CNN model and 8 nodes for $LSTM$ model (the input for each node is the related feature vector). For the testing, we use two test sets, the first one is 662 mashups with 3 $APIs$ (T_1), and the other one is an independent dataset collected from Github. We consider plausible API mashups as the $APIs$ co-exist within a single project in Github. By checking whether those $APIs$ have the complete information of 8 features or not, we obtained 2325 mashups (T_2) as a ground truth testing dataset.

From the experimental results (Table I), we can see that RF performs better than DT since RF has its excellent ability to control error without causing over-fitting and random forest creates many decision trees based on bagging, where resampling the dataset in multiple iterations and for each iteration trains a new classifier. Meanwhile, we also find that deep learning models perform better than the traditional learning models. It is expected since deep learning offers a set of techniques and algorithms that help us to parameterize deep neural network structures with many hidden layers and parameters from the given dataset. In addition, the results also show that among all learning models, the CNN model has highest accuracy. It is because the best order of selected 8 features is unknown, which in turn largely affects the results of the $LSTM$ model. According to this, we also generate $LSTM$ model with different order of features. For example, the performance of cross validation and accuracy have been improved by 3% in average when the selected features are listed in the following order: "links", "page title", "protocol formats", "category", "primary category", "secondary category", "simple description" and "whole description".

2) Impact of Word Embedding Methods:

Exp2: In this experiment, we use the same 569 API mashup data with the same selected features and labeling method as described in Exp1. Then, we use two different ways to conduct word embedding:

(a) we only use Google Word2Vec [15] (keyword based) (W_1) to generate word vector for all features;

(b) we use Skip-Thought [28] (sentence based) (W_2) to generate word vectors for the 2 sentence based features and use Google Word2Vec (W_1) to generate word vectors for the 6 word based features.

Next, we train the deep learning models by using the above training dataset. Finally, we use the ground truth testing dataset (T_2) to evaluate the accuracy of the models.

From the results (Table II), we can see that the results of cross validation and accuracy have been improved when W_1+W_2 is used. It is reasonable since we have selected different word embedding methods by considering content structures among all features. Meanwhile, the results in Exp1 and Exp2 also validate that the API mashups generated by the proposed method and procedure are accurate, where the

TABLE I
IMPACT OF DIFFERENT LEARNING MODELS

Scheme	Train Size	Test Size	Feature Size	Cross Validation	Accuracy
DT	569	662	8	76.46%	76.26%
RF	569	662	8	78.80%	77.28%
$LSTM$	569	662	8	87.58%	84.46%
$LSTM$	569	2325	8	87.29%	83.81%
CNN	569	662	8	89.07%	86.18%
CNN	569	2325	8	90.43%	86.42%

TABLE II
IMPACT OF DIFFERENT WORD EMBEDDING METHODS

Scheme	Word Embedding	Cross Validation	Accuracy
$LSTM$	W_1	87.29%	83.81%
$LSTM$	$W_1 + W_2$	88.32%	84.74%
CNN	W_1	90.43%	86.42%
CNN	$W_1 + W_2$	90.86%	87.25%

mashups with similarity score larger than 0.5 has 87.25% accuracy to be a plausible API mashup by using CNN.

V. Conclusions and Future Works

In this paper, we have presented a new method for automatic API mashup generation and recommendation. By using the proposed procedure, each generated API mashup is associated with a quantitative indicator which shows the relative possibility that the given API mashup is plausible. The accuracy of the generated API mashups is evaluated by multiple machine learning and deep learning models including DT, RF, CNN and $LSTM$ with an independent testing dataset collected from Github. The experimental results show that our solution is accurate and widely applicable.

As for future work, although the results indicate that a machine learning-based predictive model could lead to good performance, improvement can be made if we increase the number of features by transforming non-numeric features into numerical ones, and also evaluate their usefulness. Meanwhile, we plan to design a good method to process the requirement of developers (e.g., query expansion), so that the system can better understand their requirements and return more relevant API mashups to developers.

ACKNOWLEDGMENTS: We want to thank our colleagues (Mehdi Bahrami & Junhee Park) from Fujitsu Laboratories of America who have provided insight and expertise that greatly assisted this work.

References

[1] J. Musser, "ProgrammableWeb," https://www.programmableweb.com/, 2005.

[2] X. Liu, Y. Hui, W. Sun, and H. Liang, "Towards service composition based on mashup," in *Services, 2007 IEEE Congress on*. IEEE, 2007, pp. 332–339.

[3] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 20, no. 4, p. 19, 2011.

[4] M. P. Robillard, "What makes apis hard to learn? answers from developers," *IEEE software*, vol. 26, no. 6, 2009.

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

[6] G. Uddin and M. P. Robillard, “How api documentation fails,” *IEEE Software*, vol. 32, no. 4, pp. 68–75, 2015.

[7] 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*. IEEE Press, 2013, pp. 319–328.

[8] C. Li, R. Zhang, J. Huai, and H. Sun, “A novel approach for api recommendation in mashup development,” in *Web Services (ICWS), 2014 IEEE International Conference on*. IEEE, 2014, pp. 289–296.

[9] W. Xu, J. Cao, L. Hu, J. Wang, and M. Li, “A social-aware service recommendation approach for mashup creation,” in *Web Services (ICWS), 2013 IEEE 20th International Conference on*. IEEE, 2013, pp. 107–114.

[10] W. Gao, L. Chen, J. Wu, and H. Gao, “Manifold-learning based api recommendation for mashup creation,” in *Web Services (ICWS), 2015 IEEE International Conference on*. IEEE, 2015, pp. 432–439.

[11] E. Wittern, V. Muthusamy, J. Laredo, M. Vukovic, A. Slominski, S. Rajagopalan, H. Jamjoom, and A. Natarajan, “Api harmony: Graph-based search and selection of apis in the cloud,” *IBM Journal of Research and Development*, vol. 60, no. 2-3, pp. 12–1, 2016.

[12] 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*. ACM, 2016, pp. 631–642.

[13] D. Li, T. Salonidis, N. V. Desai, and M. C. Chuah, “Deepcham: Collaborative edge-mediated adaptive deep learning for mobile object recognition,” in *Edge Computing (SEC), IEEE/ACM Symposium on*. IEEE, 2016, pp. 64–76.

[14] D. Li, X. Wang, and D. Kong, “Deeprebirth: Accelerating deep neural network execution on mobile devices,” *arXiv preprint arXiv:1708.04728*, 2017.

[15] T. Mikolov, “Google word2vec,” https://code.google.com/archive/p/word2vec/, 2013.

[16] Matt, “spaCy,” https://spacy.io/, 2014.

[17] C. Thompson and P. Garfinkel, “MedHelp,” http://www.medhelp.org/, 1994.

[18] Jamie and B. Heywood, “Patientslikeme,” https://www.patientslikeme.com/, 2004.

[19] J. Lafferty, A. McCallum, F. Pereira *et al.*, “Conditional random fields: Probabilistic models for segmenting and labeling sequence data,” in *Proceedings of the eighteenth international conference on machine learning, ICML*, vol. 1, 2001, pp. 282–289.

[20] G. Rizzo and R. Troncy, “Nerd: a framework for unifying named entity recognition and disambiguation extraction tools,” in *Proceedings of the Demonstrations at the 13th Conference of the European Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics, 2012, pp. 73–76.

[21] M. Henrikson, “SNOMED CT ANALYZER,” http://snomedct.t3as.org/, 2014.

[22] C. Apté and S. Weiss, “Data mining with decision trees and decision rules,” *Future generation computer systems*, vol. 13, no. 2-3, pp. 197–210, 1997.

[23] T. Shi and S. Horvath, “Unsupervised learning with random forest predictors,” *Journal of Computational and Graphical Statistics*, vol. 15, no. 1, pp. 118–138, 2006.

[24] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[25] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[26] H. Sak, A. Senior, and F. Beaufays, “Long short-term memory recurrent neural network architectures for large scale acoustic modeling,” in *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[27] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur, “Recurrent neural network based language model.” in *Interspeech*, vol. 2, 2010, p. 3.

[28] R. Kiros, Y. Zhu, R. R. Salakhutdinov, R. Zemel, R. Urtasun, A. Torralba, and S. Fidler, “Skip-thought vectors,” in *Advances in neural information processing systems*, 2015, pp. 3294–3302.