Automatic Task Completion Flows from Web APIs

Kyle Williams
Microsoft
Redmond, Washington, USA
kyle.williams@microsoft.com

Seyyed Hadi Hashemi*
University of Amsterdam
Amsterdam, The Netherlands
hashemi@uva.nl

Imed Zitouni
Microsoft
Redmond, Washington, USA
izitouni@microsoft.com

ABSTRACT

The Web contains many APIs that could be combined in countless ways to enable Intelligent Assistants to complete all sorts of tasks. We propose a method to automatically produce task completion flows from a collection of these APIs by combining them in a graph and automatically extracting paths from the graph for task completion. These paths chain together API calls and use the output of executed APIs as inputs to others. We automatically extract these paths from an API graph in response to a user query and then rank the paths by the likelihood of them leading to user satisfaction. We apply our approach for task completion in the email and calendar domains and show how it can be used to automatically create task completion flows.

CCS CONCEPTS

• **Information systems** → *Task models*; • **Computing methodologies** → *Natural language processing*.

KEYWORDS

Task completion, intelligent assistant

ACM Reference Format:
Kyle Williams, Seyyed Hadi Hashemi, and Imed Zitouni. 2019. Automatic Task Completion Flows from Web APIs. In *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '19), July 21–25, 2019, Paris, France.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3331184.3331318

1 INTRODUCTION

The Web contains thousands of APIs for performing a wide variety of tasks, such as booking taxis, making hotel reservations, and checking the weather. Intelligent Assistants (IAs) often make use of these APIs to allow users to complete tasks by providing a dialog frontend to these APIs. However, beyond the individual API functions, these APIs can also be combined in countless ways to create new and complex task completion flows. For instance, an API for checking the location of an upcoming meeting could be combined with an API for booking a taxi in order to provide a user with a ride to their next meeting without them needing to explicitly specify the location since it would be returned by the meeting API. Similarly, APIs for hotel and flight reservations could be combined into a single vacation booking experience that share the travel dates and location. Given the number of Web APIs that exist, there are countless ways in which APIs can be combined to create new task completion experiences and thus enable new levels of assistance beyond simple single goal tasks.

In this paper we propose a method to automatically construct complex task completion flows from APIs. To do this we view the web as a collection of APIs that are indexed so that they can be ranked by a search engine based on the action they perform. Our goal is to chain these single APIs calls together. In our approach, which is shown at a high level in Figure 1, we assume the existence of a repository of APIs from which we create an API graph where the outputs of APIs can be used as the inputs of others and we build this graph by using semantic representations of the APIs. Given a user query, we identify the APIs in the graph that, if executed, could satisfy the user. We then extract paths from the graph that lead to the execution of that API where these paths consist of multiple APIs chained together with the outputs of one API being used as input for another. We refer to these chained paths as task completion flows. For example, in the taxi booking example, the location output from the meeting API can be used as a destination for the taxi booking API. We then rank these flows by the likelihood of them leading to user satisfaction and then execute the highest ranked flow, chaining the outputs and inputs across APIs.

Our approach does not rely on designers to define task completion flows but is instead completely automatic and data driven. We utilize neural networks for identifying APIs for execution as well as for ranking candidate task completion flows. We demonstrate our approach on APIs from the email and calendar domains.

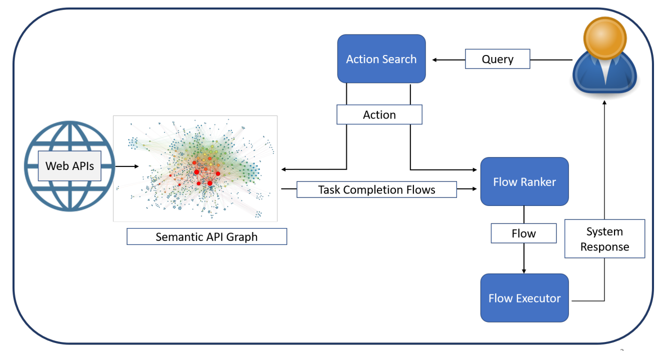

Figure 1: Overview of approach, including semantic API, action search, and ranking.

*Work done while an intern at Microsoft

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGIR '19, July 21–25, 2019, Paris, France
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6172-9/19/07…$15.00
https://doi.org/10.1145/3331184.3331318

2 RELATED WORK

Crook et al. [3] propose the idea of a dialog system that composes functional dialog units for question answering. At each timestep, a search is made over botlets that are available for execution and a decision is made on which of those to execute using heuristics. The system contains all components of a dialog system, such as components for NLU, dialog state tracking, etc.

An area of related work involves the end-to-end training of conversational systems using reinforcement learning [1]. These systems are typically trained end-to-end to handle the whole dialog [6, 7], including the components for natural language understanding and generation, as well as dialog management. However, a challenge with reinforcement learning approaches is that they often require large amounts of data to train end-to-end and/or the existence of a robust and accurate user simulator [2]. In our approach we do not train the system end-to-end but rather train our model to chain together high quality APIs.

Botea et al. [4] automatically generate dialog agents via automated planning. The input to their system consists of atomic dialog actions, an initial dialog state, and a goal. They then use off the shelf dialog planning tools to generate the dialog plan. They demonstrate the usefulness of their approach in 3 domains. A key difference between our work and existing work is that we specifically focus on how we can combine the vast quantities of APIs that exist in order to perform complex task completion.

3 APPROACH

We now provide a high level overview of our approach and then provide details on the different components. Figure 1 shows an overview of our approach. We begin with a set of Web APIs, which are semantified and added to a semantic API graph. Given a user utterance, we identify a node in the semantic API graph that, if executed, will satisfy the user's goal. We refer to this selected API as the target action. Given the target action, we extract a set of candidate execution flows from the graph. This set of candidate flows is then ranked by the likelihood of them leading to user satisfaction and we then execute the highest ranked flow.

3.1 Semantic API Graph

3.1.1 Semantic APIs. Semantic APIs are API descriptions where the inputs and outputs have semantic types. For instance, a *create_email* API might take the parameters *address, subject* and *message*, with semantic types *person.email_address*, *email_subject*, and *email_message*, respectively, and return an *email* semantic object. We assume that a repository of semantic APIs exists for building task completion flows. The source of the APIs in this repository might be internal to an organization, from the public Web, etc.

3.1.2 Semantic API Graph Construction. Given a semantic API repository, we automatically construct a directed semantic API graph G where the nodes in the graph are API endpoints, input parameters, and output values. We mark the API endpoint nodes with a flag indicating that they are executable to differentiate them from the input parameters and output values. We then create edges between each API endpoint and its input parameters and outputs. For instance, for the *create_email* API endpoint described above,

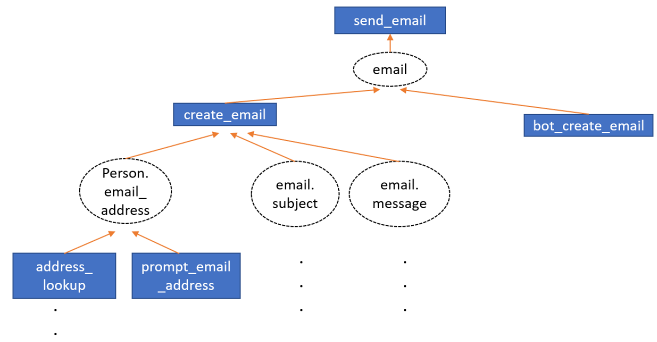

Figure 2: A simple example of a semantic graph

we create 1 node for the executable *create_email* endpoint, 3 nodes corresponding to the input parameters, and 1 node corresponding to the output. If the type of an output o of one API matches the type of an input i of another API, then we create an edge (o, i), indicating that o is able to provide a value for i. We depict this visually in Figure 2, which shows a simple semantic graph. The nodes corresponding to executable APIs are represented by rectangles while the nodes representing entities are represented by ovals. In this semantic graph, the *send_email* node requires a parameter with type *email*. As the graph shows, there are two ways this entity can be provided. The first is via the *bot_create_email* API, which asks the user a series of questions and then outputs an *email* object. The other option is via the *create_email* API. This API requires a *person.email_address*, *email.subject*, and *email.message* as input, which it then uses to form an *email* object. The *person.email_address* entity can either be provided by prompting the user or by looking it up in an address book. The *address_lookup* API requires an input of type *person.name*, which is not shown for space reasons. Similarly, there are various ways of producing the *email.subject* and *email.message* entities, which are also not shown for space reasons. A key take away from this graph is that there are multiple execution flows that can be used for sending an email and in this paper we present a way to extract these flows and rank them by the likelihood of them leading to user satisfaction. Also note, a feature of this graph is that it can easily be expanded. For instance, imagine a new API for creating a *email* object called *copy_email*. This API could easily be inserted into the graph with an edge connecting it to the *email* entity, thereby enabling additional execution flows.

3.2 Action Search

Given a user utterance, we identify one or more target API endpoint nodes in the semantic graph that captures the user's goal. For instance, for the user utterance *"send an email to John"*, the *send_email* node would be considered the target API endpoint. Similarly, for the utterance *"get me a taxi to my next meeting"*, the API nodes for getting the next meeting and booking a taxi would be considered target endpoints. This approach is equivalent to intent detection in intelligent assistants [6].

We cast the action search problem as a sequence classification problem, where we treat the user utterance as the sequence and the executable nodes in the semantic API graph as the target. We use a

bidirectional LSTM-based classifier that operates on the sequence of words that appear in the user utterance.

For a given sequence of words $W = w_1, w_2, ..., w_n$, we define the following:

- Word embedding: e_w for each $w \in W$
- Word LSTM: ϕ_f^W, ϕ_b^W,

where ϕ_f^W, ϕ_b^W refer to the forward and backward word LSTMs. The model computes context sensitive word representations as:

$$f_i^W = \phi_f^W(v_i, f_{i-1}^W), \forall i = 1...n \tag{1}$$

$$b_i^W = \phi_b^W(v_i, b_{i+1}^W), \forall i = n...1 \tag{2}$$

The states of the forward and backward LSTMs are concatenated for the n-th word:

$$r = f_n^W \oplus b_n^W. \tag{3}$$

We add a feed forward layer g, which takes r as input and then take the softmax of the the output of g to produce probabilities of of the different executable nodes.

As part of the action search we want to allow for new executable nodes to be easily added to the Semantic API graph without needing to retrain the entire action search. If we have n executable nodes in our semantic API graph, we set the output dimensionality of g to $m \geq n$. During training and prediction we apply a mask f to each of the $i = 1, 2, ..., m$ outputs of g:

$$f(g_i) = \begin{cases} g_i & : i \leq n \\ 0 & : otherwise. \end{cases}$$

This has the effect of zeroing out outputs from g that do not correspond to valid actions. When a new action is added to the graph, we set its output to g_{n+1} and set $n = n + 1$ so that the mask f continues to zero-out outputs from g that do not correspond to valid actions. To train the model we minimize the cross entropy loss:

$$Loss^{\text{tag}} = -\sum_i p \log q, \tag{4}$$

where p is the distribution of the true labels and q is the distribution of the predicted labels.

3.3 Task Flow Extractor

Given a user goal and a corresponding target API that satisfies that goal based on the action search, we extract flows from the semantic graph that lead to that target API. For instance, for the *send_email* API in Figure 2 we might extract two flows, e.g., one that prompts the user for an email recipient and one that performs an address lookup. For each candidate flow all of the inputs to the target action must be satisfied. For instance, in Figure 2 the *send_email* flow must contain a way to provide the *email* entity. Similarly, if a flow contains the *create_email* API, then the flow must also include ways for satisfying the 3 parameters of that API. When extracting flows we make use of heuristics for excluding flows that are unlikely to lead to successful execution. For instance, we apply a heuristic that excludes flows that require the user to enter complex data types and a heuristic that excludes flows that have the same entity types produced multiple times in a path.

3.4 Task Flow Ranker

The flow ranker assigns a score to each of the candidate flows. The flow ranker model is similar to the action search model; however, instead of computing a probability distribution over all executable APIs, it instead assigns a score to each flow. Following the same process as in Equation 3, it produces the following encodings:

r_U An encoding of the user utterance produced the same way as in Equations 1-3.

r_E An encoding of the entities that exist in a candidate flow (e.g., the inputs and outputs to the executable APIs). Produced the same way as in Equations 1-3; however, operating on a sequence of entities rather than a sequence of words.

r_A An encoding of the executable APIs in a candidate flow. Produced the same way as in Equations 1-3; however, operating on a sequence of actions rather than a sequence of words.

Given these encodings, the model then produces an entity and action sensitive flow representation: $r_F = W \cdot (r_E \oplus r_A) + b)$.

Finally, we combine the flow representation with the user utterance representation to produce a score for the path: $\hat{y} = W \cdot (r_F \oplus r_U) + b)$. To train the model we minimize the mean square error (MSE) of the k candidate flows generated for a user utterance, where the target y values are path scores: $MSE = \frac{1}{n} \sum_i^k (y_i - \hat{y})$

4 EXPERIMENTS

We conduct our experiments using Web APIs in the calendar and email domains. We empirically find good hyperparameters for the network and set the word embedding size and word LSTM number of units to 100 for both the action search and the utterance encoder r_U of the path ranking LSTM. We set the size of entity and action encodings, as well as the number of units in the entity and action LSTM encoders r_E and r_A to 500. We use the Adam optimizer to minimize the losses. Furthermore, we use cross validation in all of our experiments and report the average for each metric.

4.1 Semantic API Graph

We build the Semantic API graph using Web APIs focused on the calendar and email domains. The APIs include executable actions for sending emails, checking email, checking for reminders, creating a calendar entry, etc. Furthermore, there are helper APIs for performing entity extraction from queries, such as extracting times and dates, people names, locations, etc. There are also utility APIs for performing address book lookups, etc. In total, the graph contains 21 executable nodes as well as multiple nodes corresponding to input and output entities for each executable node.

4.2 Action Search

4.2.1 Data. To test our action search we gather query and intent pairs from a commercial Intelligent Assistant. The queries were received as spoken query and then converted to text using a production quality speech to text system. The text representations of the queries were labeled by expert judges as part of the data pipeline of the Intelligent Assistant. We used approximately 8,500 queries belonging to 9 intents. These 9 intents correspond to executable actions for email and calendar in our semantic API graph but do not include the helper API actions, such as entity extraction.

Table 1: Precision@1 for Action Search

Approach	P@1
Baseline LM	0.6400
LSTM Classifier (Ours)	**0.8756**

4.2.2 Baseline. We compare our proposed sequence classification method to a language model based baseline [5]. We first create language models for queries and for executable API functions, where the executable API function is based on action descriptions, e.g., for the *send_mail* action, the description might be: *"Sends a message specified by the request body. The message is saved to the Sent Mail folder."* For this baseline, we select the action with the lowest KL-divergence to the query: $D(p||q) = \sum_x p(x) \log \frac{p(x)}{q(x)}$

4.2.3 Results. The results for our action search are shown in Table 1. We use Precision@1 as our metric since it corresponds to picking the correct action given the user goal. As can be seen from the table, our proposed method for action search outperforms the language model baseline. The results indicate that, using our sequence classification model, we are able to correctly predict the user's goal 87.56% of the time.

4.3 Task Flow Ranking

Our goal for task flow ranking is to assign a relevance score to each candidate task flow given the user query.

4.3.1 Data. To gather data for task flow ranking, we extract task flows from the Semantic API Graph for each executable API that matches a user goal using the method described in Section 3.3. These flows consist of chains of actions leading up to the execution of the API that corresponds to the user goal. We also extract user queries with intents corresponding to the user goals from the data pipeline of an intelligent assistant. We then create training pairs consisting of user queries and execution flows, where each user query may be paired with multiple execution flows. We then presented these pairs to expert assessors and asked them to rate the likelihood of an execution flow leading to user satisfaction given the user query. The expert assessors rated the execution flows on a 5-point scale, with 5 indicating that the flow would be very effective and 1 indicating that the flow was ineffective. A rating of 5 was used when a flow took all information in the user query into consideration (e.g., did not ask the user for a parameter if it was present in the query), while a rating of 1 was used when a flow would not allow the user to complete their task. In total, our expert assessors assigned relevance scores to just under 1,000 query-execution flow pairs.

4.3.2 Baselines. We compared our proposed method for ranking execution flows to two baselines:

Shortest Path Candidate execution flows are ranked by the number of executable actions they include.

Language Model The same language model approach as used for action search, with the profile of each flow based on a concatenation of the descriptions of actions in that flow.

Table 2: Metrics for Path Ranking

Approach	P@1	NDCG	NDCG@5
Shortest Path	0	0.4031	0.2402
Language Model	0.1336	0.6813	0.6363
LSTM Ranker (Ours)	**0.9752**	**0.7127**	**0.6740**

4.3.3 Results. For the flow ranking we consider both P@1 as well as NDCG. The results of our experiment are shown in Table 2.

As can be seen from the table, the shortest path method is the worst performing overall. In many cases, the shortest path only contained a single action, e.g., the API action corresponding to the user's goal. However, in most cases each API requires several parameters and the shortest path flows required users to enter complex types whereas the flows ranked highest by other methods would generally prompt the user for each value separately. In general, our approach greatly outperforms the language modeling baseline for Precision@1. For the proposed method, the P@1 is 0.9752, which indicates that, in most cases, the path that it ranks the highest among all candidates will lead to user satisfaction. For the language model that is only the case for about 1 in 10 queries. The performance on the language model and our proposed approach are closer for the NDCG metrics, indicating that the language model is still capable of ranking some paths that lead to user satisfaction highly; however, not the highest. The results show that our approach, which jointly encodes the user query as well as the entities and actions along each path leads to the best performance overall.

5 DISCUSSION AND CONCLUSIONS

The Web contains many APIs that can be combined in countless ways to produce task completion flows. We proposed one way of doing this where we represent the APIs using a semantic graph from which we automatically extract candidate flows and rank them by their likelihood of leading to user satisfaction. Doing this effectively enables many possibilities for task completion flows and also allows for flow possibilities to easily be expanded when new APIs are added to the graph. One potential limitation of this work is that it relies on a set of annotated execution flows, which can be costly to label. A potential area of research for future work would be to apply reinforcement learning on the graph.

REFERENCES

[1] Antoine Bordes, Y-Lan Boureau, and Jason Weston. 2016. Learning end-to-end goal-oriented dialog. *arXiv preprint arXiv:1605.07683* (2016).

[2] Paul Crook and Alex Marin. 2017. Sequence to Sequence Modeling for User Simulation in Dialog Systems. In *Interspeech '17*.

[3] Paul A. Crook, Alex Marin, Vipul Agarwal, Samantha Anderson, Ohyoung Jang, Aliasgar Lanewala, Karthik Tangirala, and Imed Zitouni. 2018. Conversational Semantic Search: Looking Beyond Web Search, Q&A and Dialog Systems. In *WSDM '18*.

[4] Adi Botea et al. 2019. Generating Dialogue Agents Via Automated Planning. In *DEEPDIAL '10*.

[5] John Lafferty and Chengxiang Zhai. 2001. Document language models, query models, and risk minimization for information retrieval. In *SIGIR '01*.

[6] Xiujun Li, Yun-Nung Chen, Lihong Li, Jianfeng Gao, and Asli Celikyilmaz. 2017. Investigation of Language Understanding Impact for Reinforcement Learning Based Dialogue Systems. *arXiv preprint arXiv:1703.07055* (2017).

[7] Jason D Williams, Kavosh Asadi, and Geoffrey Zweig. 2017. Hybrid Code Networks: practical and efficient end-to-end dialog control with supervised and reinforcement learning. In *ACL '17*.