Function Assistant: A Tool for NL Querying of APIs

Kyle Richardson and **Jonas Kuhn**
Institute for Natural Language Processing
University of Stuttgart
`kyle@ims.uni-stuttgart.de`

Abstract

In this paper, we describe **Function Assistant**, a lightweight Python-based toolkit for querying and exploring source code repositories using natural language. The toolkit is designed to help end-users of a target API quickly find information about functions through high-level natural language queries and descriptions. For a given text query and background API, the tool finds candidate functions by performing a translation from the text to known representations in the API using the semantic parsing approach of Richardson and Kuhn (2017). Translations are automatically learned from example text-code pairs in example APIs. The toolkit includes features for building translation pipelines and query engines for arbitrary source code projects. To explore this last feature, we perform new experiments on 27 well-known Python projects hosted on Github.

1 Introduction

Software developers frequently shift between using different third-party software libraries, or APIs, when developing new applications. Much of the development time is dedicated to understanding the structure of these APIs, figuring out where the target functionality lies, and learning about the peculiarities of how such software is structured or how naming conventions work. When the target API is large, finding the desired functionality can be a formidable and time consuming task. Often developers resort to resources like Google or StackOverflow to find (usually indirect) answers to questions.

We illustrate these issues in Figure 1 using two

```
## from nltk.parse.dependencygraph.py

class DependencyGraph(object):
    """A container ....for  a dependency structure"""

    def remove_by_address(self, address):
        """
        Removes the node with the given address.
        """
        # => implementation

    def add_arc(self, head_address, mod_address):
        """Adds an arc from the node specified by
        head_address to the node specified by
        the mod address....
        """
```

Figure 1: Example function documentation in Python NLTK about dependency graphs.

example functions from the well-known NLTK toolkit. Each function is paired with a short *docstring*, i.e., the quoted description under each function, which provides a user of the software a description of what the function does. While understanding the documentation and code requires technical knowledge of dependency parsing and graphs, even with such knowledge, the function naming conventions are rather arbitrary. The function `add_arc` could just as well be called `create_arc`. An end-user expecting another naming convention might be left astray when searching for this functionality. Similarly, the available description might deviate from how an end-user would describe such functionality.

Understanding the `remove_by_address` function, in contrast, requires knowing the details of the particular `DependencyGraph` implementation being used. Nonetheless, the function corresponds to the standard operation of *removing a node* from a dependency graph. Here, the technical details about how this removal is specific to a *given address* might obfuscate the overall purpose of the function, making it hard to find or understand.

At a first approximation, navigating a given API requires knowing *correspondences* between textual descriptions and source code representations. For example, knowing that the English expression *Adds an arc* in Figure 1 translates (somewhat arbitrarily) to add_arc, or that *given address* translates to address. One must also know how to detect paraphrases of certain target entities or actions, for example that *adding an arc* means the same as *creating an arc* in this context. Other technical correspondences, such as the relation between an address and the target dependency graph implementation, must be learned.

In our previous work (Richardson and Kuhn (2017), henceforth RK), we look at learning these types of correspondences from example API collections in a variety of programming languages and source natural languages. We treat each given API, consisting of text and function representation pairs, as a parallel corpus for training a simple semantic parsing model. In addition to learning translational correspondences, of the type described above, we achieve improvements by adding document-level features that help to learn other technical correspondences.

In this paper, we focus on using our models as a tool for querying API collections. Given a target API, our model learns an MT-based semantic parser that translates text to code representations in the API. End-users can formulate natural language queries to the background API, which our model will translate into candidate function representations with the goal of finding the desired functionality. Our tool, called **Function Assistant** can be used in two ways: as a blackbox pipeline for building models directly from arbitrary API collections. As well, it can be customized and integrated with other outside components or models using the tool's flexible internal Python API.

In this paper, we focus on the first usage of our tool. To explore building models for new API collections, we run our pipeline on 27 open source Python projects from the well-known *Awesome Python* project list.[1] As in previous work, we perform synthetic experiments on these datasets, which measure how well our models can generate function representations for unseen API descriptions, which mimic user queries.

2 Related Work

Natural language querying of APIs has long been a goal in software engineering, related to the general problem of software reuse (Krueger, 1992). To date, a number of industrial scale products are available in this area.[2] To our knowledge, most implementations use shallow term matching and/or information-extraction techniques (Lv et al., 2015), differing from our methods that use more conventional NLP components and techniques. As we show in this paper and in RK, term matching and related techniques can sometimes serve as a competitive baseline, but are almost always outperformed by our translation approach.

More recently, there has been increased interest in machine learning on learning code representations from APIs, especially using resources such as GitHub or StackOverflow. However, this work tends to look at learning from many API collections (Gu et al., 2016), making such systems hard to evaluate and to apply to querying specific APIs. Other work looks at learning to generate longer code from source code annotations for natural language programming (Allamanis et al., 2015), often focusing narrowly on a specific programming language (e.g., Java) or set of APIs. To our knowledge, none of these approaches include companion software that facilitate building custom pipelines for specific APIs and querying.

Technically, our approach is related to work on semantic parsing, which looks at generating formal representations from text input for natural language understanding applications, notably question-answering. Many existing methods take direct inspiration from work on MT (Wong and Mooney, 2006) and parsing (Zettlemoyer and Collins, 2009). Please see RK for more discussion and pointers to related work.

3 Technical Approach

In this paper, we focus on learning to generate function representations from textual descriptions inside of source code collections, or APIs. We will refer to these target function representations as *API components*. Each component specifies a function name, a list of arguments, and other optional information such as a namespace.

Given a set of example text-component pairs from an example API, $D = \{(x_i, z_i)\}_{i=1}^n$, the goal

---

is to learn how to generate correct, well-formed components $z \in \mathcal{C}$ for each text x. When viewed as a semantic parsing problem, we can view each z as analogous to a target logical form. In this paper, we focus narrowly on Python source code projects, and thus Python functions z, however our methods are agnostic to the input natural language and output programming language as shown in RK.

When used for querying, our model takes a text input and attempts to generate the desired function representation. Technically, our approach follows our previous work and has two components: a simple and lightweight word-based translation model that generates candidate API components, and a discriminative model that reranks the translation model output using additional phrase and document-level features. All of these models are implemented natively in our tool, and we describe each part in turn.

3.1 Translation Model

Given an input text (or query) sequence $x = w_1, .., w_{|x|}$, the goal is to generate an output API component $z = u_i, .., u_{|z|}$, which involves learning a conditional distribution $p(z \mid x)$. We pursue a noisy-channel approach, where

$$p(z \mid x) \propto p(x \mid z)p(z)$$

By assuming a uniform prior $p(z)$ on output components, the model therefore involves computing $p(x \mid z)$, which under a word-based translation model can be expressed as:

$$p(x \mid z) = \sum_a p(x, a \mid z)$$

where the summation ranges over the set of all many-to-one (word) alignments a from $x \to z$.

While many different formulations of word-based models exist, we previously found that the simplest lexical translation model, or IBM Model 1 (Brown et al., 1993), outperforms even higher-order alignment models with location parameters. This model computes all alignments exactly using the following equation:

$$p(x \mid z) \approx \prod_{j=1}^{|x|} \sum_{i=0}^{|z|} p_t(w_j \mid u_i) \tag{1}$$

where p_t defines a multinomial distribution over a given component term u_j for all words w_j.

While many parameter estimation strategies exist for training word-based models, we similarly found that the simplest EM procedure of Brown et al. (1993) works the best. In RK, we describe a linear-time decoding strategy (i.e., for generating components from input) over the number of components $\mathcal{C}$, which we use in this paper. Our tool also implements our types of conventional MT decoding strategies that are better suited for large APIs and more complex semantic languages.

3.2 Discriminative Reranking

Following most semantic parsing approaches (Zettlemoyer and Collins, 2009), we use a discriminative log-linear model to rerank the components generated from the underlying translation model. Such a model defines a conditional distribution: $p(z \mid x; \theta) \propto e^{\theta \cdot \phi(x,z)}$, for a parameter vector $\theta \in \mathbb{R}^b$, and a set of feature functions $\phi(x, z)$.

Our tool implements several different training and optimization methods. For the purpose of this paper, we train our models using an online, stochastic gradient ascent algorithm under a maximum conditional log-likelihood objective.

3.2.1 Features

For a given text input x and output component z, $\phi(x, z)$ defines a set of features between these two items. By default, our pipeline implementation uses three classes of features, identical to the feature set used in RK. The first class includes additional word-level features, such as word/component match, overlap, component syntax information, and so on. The second includes phrase and hierarchical phrase features between text and component candidates, which are extracted standardly from symmetric word-level alignment heuristics.

The other category of features includes document-level features. This includes information about the underlying API class hierarchy, and relations between words/phrases and abstract classes within this hierarchy. Also, we use additional textual description of parameters in the docstrings to indicate whether word-components candidate pairs overlap in these descriptions.

4 Implementation and Usage

All of the functionality above is implemented in the **Function Assistant** toolkit. The tool is part of the companion software release for our previous work called **Zubr**. For efficiency, the core

```
## pipeline parameters
params = [
  ("−baseline","baseline",False,"bool",
 "Use baseline model [default=False]","GPipeline")
]

## Zubr pipeline tasks
tasks = [
  "zubr.doc_extractor.DocExtractor",# extract docs
  "process_data", # custom function.
  "zubr.SymmetricAlignment",# learn trans. model.
  "zubr.Dataset", # build dataset obj.
  "zubr.FeatureExtractor",  ## build extractor obj.
  "zubr.Optimizer", ## train reranking model
  "zubr.QueryInterface", # build query interface
  "zubr.web.QueryServer", # launch HTTP server
]

def process_data(config):
    """Preprocess the extracted data using a custom
    function or outside library (e.g., nltk)

    :param config: The global configuration
    """
    preprocess_function(config,...)
```

Figure 2: An example pipeline script for building a translation model and query server.

functionality is written in Cython [3], which is a compiled superset of the Python language that facilitates native C/C++ integration.

The tool is designed to be used in two ways: first, as a black-box pipeline to build custom translation pipelines and API query engines. The tool can also be integrated with other components using our Cython and Python API. We focus on the first type of functionality.

4.1 Library Design and Pipelines

Our library uses dependency-injection OOP design principles. All of the core components are implemented as wholly independent classes, each of which has a number of associated configuration values. These components interact via a class called `Pipeline`, which glues together various user-specified components and dependencies, and builds a global configuration from these components. Subsequent instantiation and sharing of objects is dictated, or *injected*, by these global configurations settings, which can change dynamically throughout a pipeline run.

Pipelines are created by writing pipeline scripts, such as the one shown in Figure 2. This file is an ordinary Python file, with two mandatory variables. The first `params` variable specifies various high-level configuration parameters associated with the pipeline. In this case, there is a setting `--baseline`, which can be evoked to run

---
[3]http://cython.org/

a baseline experiment, and will effect the subsequent processing pipeline.

The second, and most important, variable is called `tasks`, and this specifies an ordering of subprocesses that should be executed. The fields in this list are pointers to either core utilities in the underlying `Zubr` toolkit (each with the prefix `zubr.`), or user defined functions. This particular pipeline starts by building a dataset from a user specified source code repository, using `DocExtractor`, then builds a symmetric translation model `SymmetricAlignment`, a feature extractor `FeatureExtractor`, a discriminative reranker `Optimizer`, all via various intermediate steps. It finishes by building a query interface and query server, `QueryInterface` and `QueryServer`, which can then be used for querying the input API.

As noted already, each subprocesses has a number of associated configuration settings, which are joined into a global configuration object by the `Pipeline` instance. For the translation model, settings include, for example, the type of translation model to use, the number of iterations to use when training models, and so on. All of these settings can be specified on the terminal, or in a separate configuration file. As well, the user is free to define custom functions, such as `process_data`, or classes which can be used to modify the default processing pipeline or implement new ML features.

4.2 Web Server

The last step in this pipeline builds an HTTP web server that can be used to query the input API. Internally, the server makes calls to the trained translation model and discriminative reranker, which takes user queries and attempts to translate them into API function representations. These candidate translations are then returned to the user as potential answers to the query. Depending on the outcome, the user can either rephrase his/her question if the target function is not found, or look closer at the implementation by linking to the function's source code.

An example screen shot of the query server is shown in Figure 3. Here, the background API is the NLTK toolkit, and the query is *Train a sequence tagger model*. While not mentioned explicitly, the model returns training functions for the `HiddenMarkovModelTagger`. The right
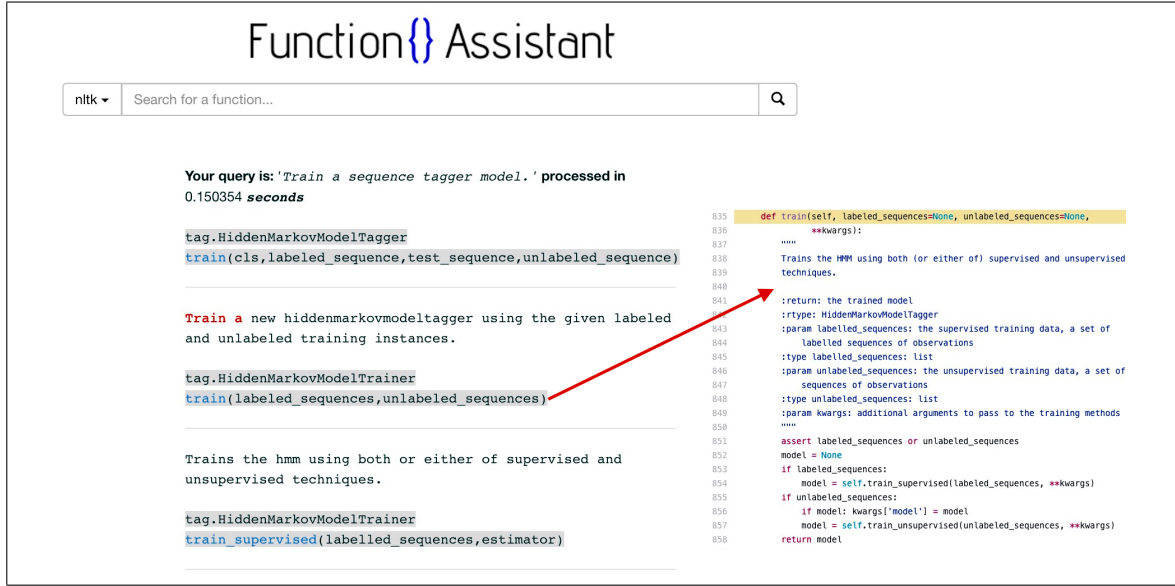
Figure 3: An example screen shot of the `Function Assistant` web server.

Project	# Pairs	# Symbols	# Words	Vocab.
scapy	757	1,029	7,839	1,576
zipline	753	1,122	8,184	1,517
biopython	2,496	2,224	20,532	2,586
renpy	912	889	10,183	1,540
pyglet	1,400	1,354	12,218	2,181
kivy	820	861	7,621	1,456
pip	1,292	1,359	13,011	2,201
twisted	5,137	3,129	49,457	4,830
vispy	1,094	1,026	9,744	1,740
orange	1,392	1,125	11,596	1,761
tensorflow	5,724	4,321	45,006	4,672
pandas	1,969	1,517	17,816	2,371
sqlalchemy	1,737	1,374	15,606	2,039
pyspark	1,851	1,276	18,775	2,200
nupic	1,663	1,533	16,750	2,135
astropy	2,325	2,054	24,567	3,007
sympy	5,523	3,201	52,236	4,777
ipython	1,034	1,115	9,114	1,771
orator	817	499	6,511	670
obspy	1,577	1,861	14,847	2,169
rdkit	1,006	1,380	9,758	1,739
django	2,790	2,026	31,531	3,484
ansible	2,124	1,884	20,677	2,593
statsmodels	2,357	2,352	21,716	2,733
theano	1,223	1,364	12,018	2,152
nltk	2,383	2,324	25,823	3,151
sklearn	1,532	1,519	13,897	2,115

Table 1: New English Github datasets.

side of the image shows the hyperlink path to the original source in Github for the `train` function.

5 Experiments

Our current `DocExtractor` implementation supports building parallel datasets from raw Python source code collections. Internally, the tool reads source code using the abstract syntax tree utility, `ast`, in the Python standard library, and extracts sets of function and description pairs. In addition, the tool also extracts class descriptions, parameter and return value descriptions, and information about the API's internal class hierarchy. This last type of information is then used to define document-level features.

To experiment with this feature, we built pipelines and ran experiments for 27 popular Python projects. The goal of these experiments is to test the robustness of our extractor, and see how well our models answer unseen queries for these resources using our previous experimental setup.

5.1 Datasets

The example projects are shown in Table 1. Each dataset is quantified in terms of **# Pairs**, or the number of parallel function-component representations, the **# Symbols** in the component output language, the **#** (NL) **Words** and **Vocab** size.

5.2 Experimental Setup

Each dataset is randomly split into train, test, and dev. sets using a 70%-30% (or 15%/15%) split. We can think of the held-out sets as mimicking queries that users might ask the model. Standardly, all models are trained on the training sets, and hyper-parameters are tuned to the dev. sets.

For a unseen text input during testing, the model generates a candidate list of component outputs. An output is considered correct if it matches *exactly* the gold function representation. As before, we measure the **Accuracy @1**, accuracy within top ten (**Accuracy @10**), and the **MRR**.

As in our previous work, three additional baselines are used. The first is a simple bag-of-words

Method	scapy	zipline	biopython	renpy	pyglet	kivy	pip	twisted	vispy
BoW	00.0 51.3 17.4	01.7 38.3 12.9	05.8 54.8 20.4	06.6 41.1 16.6	05.7 52.3 19.2	07.3 53.6 22.0	06.2 40.9 17.1	06.6 38.8 16.9	07.3 48.7 18.6
Term Match	**21.2** 43.3 28.7	28.5 50.8 36.2	23.5 48.1 31.7	25.7 59.5 38.7	20.4 50.9 31.2	30.0 62.6 41.3	19.1 50.2 30.7	17.6 44.1 26.2	29.2 64.0 41.1
Translation	20.3 61.9 34.7	27.6 62.5 40.7	29.6 75.6 45.8	30.8 61.7 42.0	26.1 69.5 41.3	33.3 67.4 45.3	18.6 56.4 32.3	27.7 61.4 39.4	28.6 70.1 42.3
Reranker	**21.2 67.2 37.2**	**30.3 70.5 45.3**	**32.3 79.1 48.6**	**38.9 73.5 48.9**	**29.0 77.1 45.5**	**35.7 75.6 49.1**	**25.9 65.8 39.9**	**28.8 65.8 42.2**	**33.5 80.4 50.3**

Method	orange	tensorflow	pandas	sqlalchemy	pyspark	nupic	astropy	sympy	ipython
BoW	13.4 60.5 29.1	09.4 47.4 21.2	03.7 40.6 15.6	07.3 45.0 18.4	07.5 50.9 20.8	06.4 55.0 22.8	07.7 52.0 21.1	06.4 44.4 18.5	01.9 41.2 13.9
Term Match	37.9 69.7 49.3	25.2 48.7 33.5	19.3 43.7 27.9	17.3 48.4 26.6	20.5 46.9 29.1	23.6 51.0 33.1	26.1 49.1 34.3	20.2 44.9 28.8	23.8 56.7 33.8
Translation	40.3 78.3 54.0	35.3 71.5 48.0	29.1 62.7 41.0	28.8 70.3 43.0	37.1 78.7 52.1	**30.9** 69.8 44.6	30.7 66.6 43.4	**32.8** 70.2 45.5	24.5 59.3 36.5
Reranker	**45.1 84.1 59.9**	**38.4 77.7 51.8**	**31.1 66.1 43.1**	**35.0 76.1 49.7**	**41.5 81.5 55.3**	29.3 **76.7 45.6**	**33.9 74.4 47.4**	32.1 **75.0 46.6**	**29.6 66.4 42.3**

Method	orator	obspy	rdkit	django	ansible	statsmodels	theano	nltk	sklearn
BoW	10.6 66.3 28.6	06.7 49.5 20.2	05.3 40.6 17.1	04.5 40.9 16.2	17.9 55.3 30.5	05.6 46.1 18.6	03.2 43.7 16.2	05.0 44.2 16.3	05.2 45.8 17.7
Term Match	31.9 64.7 43.7	19.9 46.6 30.0	13.3 46.6 23.9	19.3 48.0 29.1	24.8 54.0 35.8	16.7 39.9 25.1	16.3 37.1 24.0	19.8 45.6 28.4	24.4 50.6 32.5
Translation	**32.7** 79.5 47.5	33.8 75.8 48.3	**25.3** 60.6 37.2	22.9 57.8 34.6	35.5 71.6 47.5	25.4 64.8 37.8	26.2 58.4 37.8	28.2 68.0 41.5	27.9 67.6 41.3
Reranker	**32.7 82.7 49.7**	**37.7 80.0 52.3**	**25.3 63.3 39.6**	**25.8 64.5 39.4**	**40.5 77.0 53.1**	**28.8 69.1 41.7**	**27.3 66.1 39.9**	**31.6 72.5 45.7**	**29.2 75.5 44.5**

| Accuracy @1 | Accuracy @10 | Mean Reciprocal Rank (MRR) |

Table 2: Test results on our new Github datasets.

(**BoW**) model, which uses word-component pairs as features. The second is a **Term Match** baseline, which ranks candidates according to the number of matches between input words and component words. We also compare the results of the **Translation** (model) without the reranker.

6   Results and Discussion

Test results are shown in Table 2, and largely conform to our previous findings. The **BoW** and **Term Match** baselines are outperformed by all other models, which again shows that API querying is more complicated than simple word-component matching. The **Reranker** model leads to improvements on all datasets as compared with only using the **Translation** model, indicating that document-level and phrase features can help.

We note that these experiments are synthetic, in the sense that it's unclear whether the held-out examples bear any resemblance to actual user queries. Assuming, however, that each held-out set is a representative sample of the queries that real users would ask, we can then interpret the results as indicating how well our models answer queries. Whether or not these held-out examples reflect *real* queries, we believe that they still provide a good benchmark for model construction. All code and data will be released to facilitate further experimentation and application building. Future work will look at eliciting more naturalistic queries (e.g., through StackOverflow), and doing usage studies via a permanent web demo[4].

7   Conclusion

We introduce **Function Assistant**, a lightweight tool for querying API collections using uncon-strained natural language. Users can supply our tool with target source code projects and build custom translation or processing pipelines and query servers from scratch. In addition to the tool, we also created new resources for studying API querying, in the form of datasets built from 27 popular Github projects. While our approach uses simple components, we hope will that our tool and resources will serve as a benchmark for future work in this area, and ultimately help to solve everyday software search and reusability issues.

References

Miltiadis Allamanis, Daniel Tarlow, Andrew D Gordon, and Yi Wei. 2015. Bimodal modelling of source code and NL. In *Proceedings of ICML*.

Peter F Brown, Vincent J Della Pietra, Stephen A Della Pietra, and Robert L Mercer. 1993. The mathematics of SMT. *Computational linguistics*, 19(2).

Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API Learning. *arXiv preprint arXiv:1605.08535*.

Charles W. Krueger. 1992. Software reuse. *ACM Computing Surveys (CSUR)*, 24(2).

Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. Codehow: Effective code search based on api understanding and extended boolean model (e). In *Proceedings of ASE*.

Kyle Richardson and Jonas Kuhn. 2017. Learning Semantic Correspondences in Technical Documentation. In *Proceedings of ACL*.

Yuk Wah Wong and Raymond J. Mooney. 2006. Learning for Semantic Parsing with Statistical Machine Translation. In *Proceedings of HLT-NAACL*.

Luke S. Zettlemoyer and Michael Collins. 2009. Learning context-dependent mappings from sentences to logical form. In *Proceedings of ACL*.

---

[4] see demo here: http://zubr.ims.uni-stuttgart.de/