Python Source Code Search

Author: Andrew Malta
Advisor: Dragomir Radev

Abstract

There exists a vast amount of source code publicly available on Github; however, there do not exist many easy ways to search for relevant code to particular tasks on the granular level of functions and classes. My project is focused on the construction of productivity tools for programmers and computer scientists, leveraging source code that can be extracted from Github. The initial stages of the project involve identifying repositories within a particular area of computer science; for example, repositories that deal with natural language processing and related fields. Professor Radev has collected a set of resources on a number of topics in natural language processing and machine learning, which can be found in the All About NLP (AAN) database. After identifying a list of Github repositories, the next steps will involve building a pipeline to automatically extract the source files of interest from these repositories. Next using static analysis on the extracted code, this project aims to make this python source code indexable and searchable. This will involve identifying sections of code like function and class declarations and extracting relevant information about them from the raw source code. Finally this project will implement an algorithm for ranking the relevance of source code fragments to a user specified query. The end result of this project will be an end-to-end system that given a query returns a number of relevant code snippets that demonstrate how the specified topic can be implemented in code. If time permits, I will create a demo of the working system in a web application framework to be run locally for demonstration purposes.

Data Collection

A bulk of the initial work in the project was coming up with a feasible way to find a set of high quality python source code to power the search tool. At first I attempted to leverage Github’s search API to try and filter for source repositories that were written in python and had been forked and starred frequently. This is a bit tricky for a number of reasons. First, Github did not allow empty search queries, which meant that I had to bias the source code I was extracting by the search terms that I used. Second this API was highly rate limited and getting a reasonable amount of source code would have taken a long time. This led me to search for different ways to find source repositories to extract source code from. In this search, my advisor pointed me toward a resource that his lab had worked on called AAN (All About NLP), a search engine with curated tutorials, papers, libraries, and, most importantly for me, links to Github repositories.
Next, using these links to Github repositories, my next task was to figure out a way to extract all of the source code from each of the repositories. My first, somewhat misguided, attempt at this was to do a recursive web scrape on the directory structure of the repository.

```python
import requests
from lxml import html

def get_repo_tree_scrape(owner, repo, path, recursive=True):
    url = "https://github.com/{}/{}".format(owner, repo)

    wait_for_rate_limiter("core", 1000)
    print "requesting {}".format(url)
    r = requests.get(url)

    tree = html.fromstring(r.text)

    result = tree.find_class("files js-navigation-container")
    tbody = [child for child in result[0].iterchildren()][0]

    for child in tbody.iterchildren():
        children = child.getchildren()

        files = []

        if len(children) == 4:
            icon_value = children[0].getchildren()[0].values()[1]
            file_type = icon_value[len("octicon octicon-file-"):] 
            name = children[1].getchildren()[0].text_content()

            if file_type == "directory" and recursive:
                files += get_repo_tree(owner, repo, path + name + "/")
            else:
                print path + name
                files.append(path + name)

    return files
```

The above code, while it works in theory, runs into the automatic abuse flagger on Github’s API and it required me to go back to the drawing board. After a bit of searching, I found a Github API endpoint, that was not listed in the API documentation, for downloading archive files of repositories and used the following code to obtain the archive of each repository listed on AAN.

```python
def download_zip_archives(repo_tups, download_path):
```
archive = lambda x, y:
    "https://api.github.com/repos/{}/{}/zipball/master".format(x, y)
archive_links = [archive(tup[0], tup[1]) for tup in repo_tups]

for i, link in enumerate(archive_links):
    name = repo_tups[i][1] + ".zip"
    download_file(link, name, download_path)
    print "downloaded {}".format(name)
    wait_for_rate_limiter("core", 1)

Using these archives of the repositories I did the recursive search for the source files locally on my machine, and successfully extracted the python source files required for the project. Out of the 465 Github repositories linked to on AAN, I extracted 10138 python source files.

**Static Analysis**

Now that the code was extracted, what I had to do next was to extract useful structures from the code such as classes and function definitions. With the help of a very useful static analysis library, Jedi, I identified all of the class, function, and variable definitions in each python file and stored meta-data about them in a json file. While Jedi was very helpful in extracting the beginning line of functions and class declarations, it did not provide where these declarations ended in the code. At first I attempted to determine where these definitions ended using parent scope pointers that Jedi provided by building a tree and recursively passing up where each parent scope ended. While this method worked for the most part, it ran into problems since the way that Jedi assigned the parent pointers was not entirely consistent, causing code fragments to get cut off in the middle.

I worked around this problem by exploiting the fact that indentation in python source code is not only required, but it also is the way that the interpreter identifies where a code block ends. By using this property, I was able to infer where the end line of classes and functions were by computing when the indentation in the source code returned to the indentation in the starting line.

```python
def get_indentation(line_string):
    whitespace = [" ", ",t"]
    indentation = ""
    for char in line_string:
        if char in whitespace:
            indentation += char
        else:
            break
    return indentation
```
def infer_end_line(file_name, line_num):
    source = read_source(file_name)
    lines = source.split("\n")

    target_indentation = get_indentation(lines[line_num - 1])
    i = line_num
    while i < len(lines):
        line = lines[i]
        if ((get_indentation(line) == target_indentation)
            and len(line) > 0):
            if i == line_num:
                return i
            return i - 1
        i += 1
    return i

Using these 10138 python source files, I extracted the source code and docstrings of 84317 functions and 15127 classes.

Search

I ranked the search results using a variant of the popular tf-idf (term frequency-inverse document frequency) statistic in the information retrieval community. In particular to deal with the difference in length of the target documents, I normalize the term-frequency statistic inversely proportional to the number of lines of code in the selection and the number of unique terms that I compute for each code fragment. For each term in the query I calculate a normalized term-frequency and multiply it by the log of the inverse document frequency of the term. Our ranking of the query relevance to the document is then the sum of these products. Lastly, due to the fact that the name of the function, class, or file usually holds increased relevance to the task that it is performing, the scoring function awards higher scores to code with matches in the name of the source object. The following code was used for scoring

def score_matches(query, n, matches, iterable_frequencies, global_frequencies, content_map, using_source_map):

    scores = {}
    for m in matches:
        name_score = 0.0
        frequencies = iterable_frequencies[m]

        tf_normalizer = .75 * len(frequencies) + .25 * (m[4] - m[3])
        if m[4] - m[3] == 0:
            continue
name_freq = gen_source_frequencies(m[2])
tfidf_score = 0.0
for word in query:
    if word in name_freq:
        name_score += 1.0

    if word in frequencies:
        # avoid division by zero in either case here
        tf_score = frequencies[word] / float(tf_normalizer)
        idf_score = math.log(float(n) / (1 + global_frequencies[word]))
        tfidf_score += tf_score * idf_score

scores[m] = name_score + tfidf_score
return scores

A number of indices were required to make the matching and scoring efficient enough to run quickly. Across the dataset I computed term frequencies for each piece of source code in the dataset and additionally kept track of the number of pieces of source code each term appeared in to enable the inverse document frequency statistic. These indices had to be collected separately across the three different filtering types as to not dilute the inverse document frequency calculations with potential matches that were being filtered.

The task of extracting term frequency in each code fragment was not as simple as a standard bag of words approach in a natural language document as I had to figure out a way of determining all of the named tokens in the code fragment. The approach that I ended up taking was replacing all of the characters that are not possible to appear in a named token in python with spaces and splitting the resulting document on all whitespace. Following that, I additionally split the named tokens into their constituent parts in the case that the variable name was written in camel case. It is worth noting that this approach does leave in the python key words like “for”, “in”, and “while”, but as I am not using the total count of the tokens to normalize the term-frequency statistic, it only marginally affects the scoring calculation.

Demo

Putting the pieces together, I built an web interface in Flask which allows the user to search through the corpus of source code. In this web application the user can enter a query, specify whether they are looking for matches in functions, classes, or whole files, and can choose to search through just the docstrings or all of the lines in each code fragment. The web application then returns the ten best matches according to the relevance scoring mentioned in the previous section. Each result displays the score received, the line numbers of the code
fragment, a link to the full source file, and a syntax-highlighted preview of the matched source code.

Here is a the top result to the query: **beam search** filtering on functions and searching both code and docstrings.

**Score:** 12.6580979578  .../data/AAN/source/tensorflow-tensor2tensor-feb752c__beam_search.py Lines 101 - 420

```python
def beam_search(symbols_to_logits_fn,  
    initial_ids,  
    beam_size,  
    decode_length,  
    vocab_size,  
    alpha,  
    eos_id=EOS_ID):
    """Beam search with length penalties.

    Requires a function that can take the currently decoded symbols and return the logits for the next symbol. The implementation is inspired by https://arxiv.org/abs/1609.08144.

    Args:
    symbols_to_logits_fn: Interface to the model, to provide logits. Should take [batch_size, decoded_ids] and return [batch_size, vocab_size] initial_ids: Ids to start off the decoding, this will be the first thing handed to symbols_to_logits_fn (after expanding to beam size) [batch_size]
    beam_size: Size of the beam.
    decode_length: Number of steps to decode for.
    vocab_size: Size of the vocab, must equal the size of the logits returned by symbols_to_logits_fn
    alpha: alpha for length penalty.
    eos_id: ID for end of sentence.

    Returns:
    Tuple of
    (decoded beams [batch_size, beam_size, decode_length]  
    decoding probabilities [batch_size, beam_size])
    """
    batch_size = tf.shape(initial_ids)[0]
```

... Omitted due to length

Here is the top result to the query: **sentiment analysis** filtering for function and searching both code and docstring.

**Score:** 18.8046286004  .../data/AAN/source/dipanjanS-text-analytics-with-python-4215523_sentiment_analysis_unsupervised_lexical.py
def analyze_sentiment_sentiwordnet_lexicon(review, verbose=False):
    # pre-process text
    review = normalize_accented_characters(review)
    review = html_parser.unescape(review)
    review = strip_html(review)
    # tokenize and POS tag text tokens
    text_tokens = nltk.word_tokenize(review)
    tagged_text = nltk.pos_tag(text_tokens)
    pos_score = neg_score = token_count = obj_score = 0
    # get wordnet synsets based on POS tags
    # get sentiment scores if synsets are found
    for word, tag in tagged_text:
        ss_set = None
        if 'NN' in tag and swn.senti_synsets(word, 'n')[0]:
            ss_set = swn.senti_synsets(word, 'n')[0]
        elif 'VB' in tag and swn.senti_synsets(word, 'v')[0]:
            ss_set = swn.senti_synsets(word, 'v')[0]
        elif 'JJ' in tag and swn.senti_synsets(word, 'a')[0]:
            ss_set = swn.senti_synsets(word, 'a')[0]
        elif 'RB' in tag and swn.senti_synsets(word, 'r')[0]:
            ss_set = swn.senti_synsets(word, 'r')[0]
        # if senti-synset is found
        if ss_set:
            # add scores for all found synsets
            pos_score += ss_set.pos_score()
            neg_score += ss_set.neg_score()
            obj_score += ss_set.obj_score()
            token_count += 1
    # aggregate final scores
    final_score = pos_score - neg_score
    norm_final_score = round(float(final_score) / token_count, 2)
    final_sentiment = 'positive' if norm_final_score >= 0 else 'negative'
    ...
    return final_sentiment

Some code omitted due to length
Conclusion

I think that with a bit more work this kind of tool can be extremely useful for new and experienced programmers alike. I can see two primary use cases for a tool like this: * A place to search for example code before heading to stack overflow to ask why your code isn’t working. * A tool to explore a new topic that you are interested to learn more about.

While the tool as it stands does return useful results in a number of cases, there are a number of things I can add to make the tool return more relevant results. One thing that I worked on that I believe could be improved is balancing the term frequency with the length of the document better. I experimented with normalizing with the line lengths and I found that this method penalized longer documents too much, which resulted in the top results being just a few lines. I also tried normalizing by the number of unique terms in the document, but this had the opposite effect of not penalizing the very long code fragments enough, as this statistic does not quite scale linearly with the length of the document. I settled on a hybrid approach of the two methods, but finding the right balance between the two will require a bit more trial and error.

Overall this project gave me a glimpse into the field of information retrieval and proved to be a challenging problem that yielded satisfying solutions and a rewarding end product.

References