Resolving Abbreviations and Domain Terms in Source Code using Documentation

May 8, 2019

Karan Erry
Drew University

Abstract

Hitherto, term resolution techniques have focused on resolving expansions for abbreviations, i.e. terms that correspond textually with their meanings. However, in many contexts that deal with mathematical or software variables, the variables do not borrow characters from the meanings they represent. This means that these variables cannot be expanded using abbreviation-expansion techniques. Additionally, current term-resolution techniques search for expansions primarily in source code, largely ignoring a wealth of knowledge contained in the accompanying documentation. In this paper, I present novel techniques to allow the variables described above to be resolved as well, using cues inspired by how a human understands the meaning of a piece of code/documentation. My techniques are designed to search documentation, leveraging natural-language clues. I also present refinements to previous papers' acronym-expanding techniques. For all techniques I achieve recall averaging in the upper third and fourth quartiles.

Karan Erry

Drew University

An Honors Thesis
Submitted in Partial Fulfillment of the
Requirements for the Degree of Bachelor in Arts with Specialized Honors in
Computer Science
at Drew University
May 2019

Contents

List of Figures

Acknowledgements

First and foremost, I would like to thank Dr. Emily Hill for serving as the advisor to this research and thesis work. She provided immense support and guidance along the way, while giving me room to come into my own as a Computer Science researcher. Thank you to Dr. Seung-Kee Lee and Dr. Barry Burd for sitting on my thesis committee, reading and re-reading my drafts, and providing constructive feedback at every step. I am grateful to Drew University and its Department of Mathematics and Computer Science for allowing me the opportunity to complete this work. Finally, I would not have reached here without the support and motivation provided by my friends and family along the way.

1 Introduction

1.1 What is Computer Code?

The job of a computer programmer is to manipulate computer programs to do certain things. They do this by writing commands in a language that the computer understands, in order to tell the computer to behave in certain ways. Figure 1 shows an example of what these commands may look like. The simple computer commands shown in the example, also called *computer code*, tell a computer to find a picture of a pizza and then display it to me. (Readers familiar with programming will recognise these commands as being pseudocode.)

```
6  {commands to computer}
7  Steps to: Display a picture of a pizza
8      Step 1. Find a picture of a pizza
9      Step 2. Show it to me
```

Figure 1: Commands to a computer

1.2 Why Abbreviate Code?

In more complex commands, certain portions of commands may get repetitive to write. For example, if I wanted to tell a computer to display many different things, I could write the commands shown in Figure 2a. (This example also illustrates a different language for writing computer commands, i.e. one that requires underscores between words in a command.) However, as we can see, the *display_A_Picture_of_a* portion of each command is repeated. This gets laborious to write, and clutters the code.

Instead of writing that portion each time, I may choose to abbreviate it the second time onwards, provided the computer will still understand what I am trying to say. As in the code in Figure 2b, I have abbreviated the *display_A_Picture_of_a* command, after the first time, to truncated versions of the vital words: *disp* for "display" and *pic* for "picture".

```
display_A_Picture_of_a_Pizza
display_A_Picture_of_a_Bottle
display_A_Picture_of_a_Table
display_A_Picture_of_a_Person
display_A_Picture_of_a_Computer
```

(a) Commands to display many objects

```
display_A_Picture_of_a_Pizza
disp_pic_Bottle
disp_pic_Table
disp_pic_Person
disp_pic_Computer
```

(b) Abbreviating those commands

Figure 2: An example of shortening long computer commands

1.3 Why automatically resolve abbreviations?

To programmers and readers of my code who have encountered the full-form command *display_A_Picture_of_a_Pizza*, the usage of *disp_pic_Bottle* will be understandable. It generally will not require a leap of understanding for one to connect that the latter is simply an abbreviated version of the former command, and to be carried out on a bottle rather than a pizza. However, out of the relevant context, one can be flummoxed as to the meaning and significance of the words *disp* and *pic*. This might be the case if they skipped right to the portion of the code where the abbreviation was used and missed the usage of the full-form. They might resort to doing a Google search or asking a friend for assistance on resolving what an abbreviated command means in context.

Computer programs generally have lots of abbreviated code, like the example we have seen. For programmers working on large projects with hundreds of lines of code, it becomes infeasible to manually look up the meanings of each unfamiliar abbreviation here and there. Here is where techniques to automatically resolve the expansions/meanings of abbreviations in code, come in.

1.4 Two Types of Abbreviations

My techniques find resolutions for two types of abbreviations, explained below.

True Abbreviations The first type of abbreviation is that which corresponds textually with its long-form, in that all of the characters in the abbreviation are present in the long form. Examples include 'UI' (**U**ser **I**nterface),

'evt' (**ev**en**t**), and 'BMP' (**B**it**m**a**p**). These can be further classified into several classes, such as acronyms, where each letter of the abbreviation stands for a whole word in the long form; and dropped-letter short forms, which are formed by removing any characters except the first character from the long form. 'UI' is an acronym, while 'evt' and 'BMP' are examples of dropped-letter short forms.

For convenience, I will use the term *acronym* to denote all abbreviations that correspond textually with their long-forms, whether or not they are truly acronyms as per the definition of acronym above.

Domain Terms The second type of abbreviation is that which corresponds textually either little or not at all with its long form, but is nonetheless used as a shorthand way of representing something. Examples include using the go-to terms x and y in an algebra equation to stand in for real-world objects such as apples and oranges; or using a combination of alphabetical characters and numericals, such as *t0* and *t1*, to denote concepts such as a start-time and end-time.

I call these types of abbreviations Domain Terms.

1.5 Background Terminology

Regular Expressions A Regular Expression is a technique that allows programmers to easily instruct computers to perform the otherwise-complicated task of searching through lines of text for sequences of characters, such as certain words or a combination of special characters and punctuation. The simple regular expression shown in Figure 3a indicates that I want to find all occurrences of the characters *to* used between one and three times successively (denoted by the notation {1, 3}) in some given lines of text. In other words, I want to find all occurrences of the character sequences *to*; *toto*; or *tototo*.

Figure 3b shows the results, in yellow highlighting, of applying this regular expression to a set of sentences. We see that the regular expression found all occurrences of our search terms regardless of the characters' casing (upper/lower) or their placement throughout the text. The regular expressions used in this paper are more complex, but are still simply used to locate sequences of words or characters in a similar manner.

(a) A simple regular expression

(b) After applying the regular expression

Figure 3: An example of regular expression searching

Parts of Speech Tagging Parts of Speech tagging in Computer Science is the process of having computers automatically identify the different parts of speech that exist in language. Figure 4 shows some of the most common parts of speech that exist in the English language, and identifies them as used in a simple sentence.

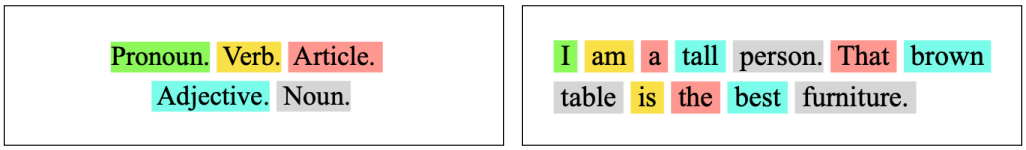


Figure 4: Parts of Speech as used in natural English language

Given that the user guides accompanying computer code are written in plain English, the process of searching these user guides for expansions to abbreviations is greatly enhanced when the computer is able to filter out certain words based on which part of speech they belong to. For example, words belonging to the *Article* part of speech, such as 'a' or 'the', are unlikely to be good candidates for expansions, because they are often ignored when abbreviations are formed.

2 Background

Significant work has been done hitherto to expand abbreviations. A large part of this work has sought to expand abbreviations in natural language text for the better understanding of the text [14, 16, 3, 15]; while other work has tried to expand abbreviation in schemas [11]. Much computer science-focused work has sought to expand abbreviations found in source code [8, 10].

Techniques vary, but they broadly follow the following structure:

1. **Preprocessing**: This variously consists of obtaining the text in which to search for abbreviations/expansions; and either automatically or manually curating and preparing the lists that will be used to filter and validate abbreviations/expansions in later steps.
2. **Abbreviation Identification**: This consists generally of scanning the source text for abbreviations that will be expanded in further steps. The abbreviations may be tokenized, i.e. they may be broken up into parts for separate processing or to focus on specific segments of the scanned abbreviations. This is also the step where abbreviations may be filtered through blacklists such *reject lists*/*stop lists*.
3. **Expansion Search**: In this step the search is conducted for possible expansions for the abbreviations from step 2. The list of possible expansions is passed to the next step.
4. **Expansion Pruning**: In this step, various optimisations and techniques are used to disregard weak expansions.
5. **Expansion Selection**: In this step, a heuristic is used to select one expansion out of a list of very eligible candidates.

Different steps have been taken to preprocess the target text to make parsing abbreviations more efficient. These include replacing non-alphabetic characters with spaces and multiple spaces with a single space [16]; disregarding lines of text that are all uppercase [14].

The target text is then parsed for abbreviations to expand. The first step may be to divide parsed abbreviations into small constituent parts for analysis [1, 4, 2, 5, 12]. Different papers have used different definitions of what constitutes an acronym. These include defining an acronym candidate as an upper-cased word from 3 to 10 characters in length [14].

Finally, the acronyms are often filtered further so that a search is only conducted for those terms that are actually considered acronyms. This includes using a *reject list* that contains words that bear characteristics to acronyms but are not acronyms, such as regular words that commonly appear in upper-case simply as a font stylisation [14].

Different novel techniques have been used to perform the search for expansions. This includes using compression [16]; computing features from alignments in candidate abbreviations [3]; using Machine Learning and Neural Network techniques to train a connectionist network to expand abbreviations correctly [11]; using a combination of edit distance, reuse factor, abbreviation factor, and domain and general significance measures to proxy

for the strength of an expansion [15]; assigning a precedence to different types of words and ranking expansion candidates based on which of the types of words they are comprised of [14].

Many have identified the context of an acronym as being an important search area for its expansion [16]. Some went further to divide up the context into the context preceding the acronym and that following the acronym [14].

Expansions are filtered using various techniques. These include the application of a learning algorithm to determine the applicability of a particular expansion based on characteristics such as the proportion of adjacent letters in the expansion [8]; restricting expansions such that each word in the expansion contributes to only upto six consecutive letters making up the acronym [16]; discarding long forms that are shorter than the abbreviation [13].

One method of filtering expansions involves curating blacklists against which to check expansions. Blacklisting techniques include a *stoplist* for excluding ubiquitous words from expansions. Alternatively, whitelists include databases of commonly-used or previously-expanded acronyms.

Many have also curated their own whitelists and blacklists of abbreviations and expansions against which they verify or invalidate found abbreviations or expansions. These include whitelists of valid dictionary words or lists of natural-language words and phrases extracted from the code [1].

From a filtered list of possible expansions, various techniques have been employed to select one expansion. These techniques include selecting the shortest long form that matches the short form [13, 7].

Abbreviation expansion has applicability in many scenarios, including in schema-matching [11]; in a hypertext context to link documents which are related to each other [14]; in the form of a tool that would provide acronym expansions instantly for reading government documents that contain a large number of acronyms [14].

The field of resolving the meanings of domain terms is largely unexplored. Doing a search on Google Scholar [6] for "domain term" or "domain abbreviation" yields no useful results.

Some notable differences exist between abbreviation expansion of domain term resolution, such as that abbreviations usually denote the same long form across multiple usages, whereas domain terms are very locally-scoped, and may be used to denote separate things in different contexts, even within the same paper.

Nevertheless, some techniques that have been used on abbreviation expansion in the past may assist in domain term resolution. These include

using syntactic cues [9] and an "adjacency to parentheses" rule [13], both techniques that have been used in prior research to strengthen the confidence in acronym expansions.

Even though there is no existing literature to indicate where domain term resolution would be applicable, we can estimate areas in which it might be useful. For one, we can estimate that domain term resolution techniques would benefit all abbreviation expansion attempts, by filtering expansions using non-textual-based techniques.

3 Approach

3.1 Overview of the Process

The process of matching an abbreviation with its resolution follows the steps outlined below and in Figure 5:

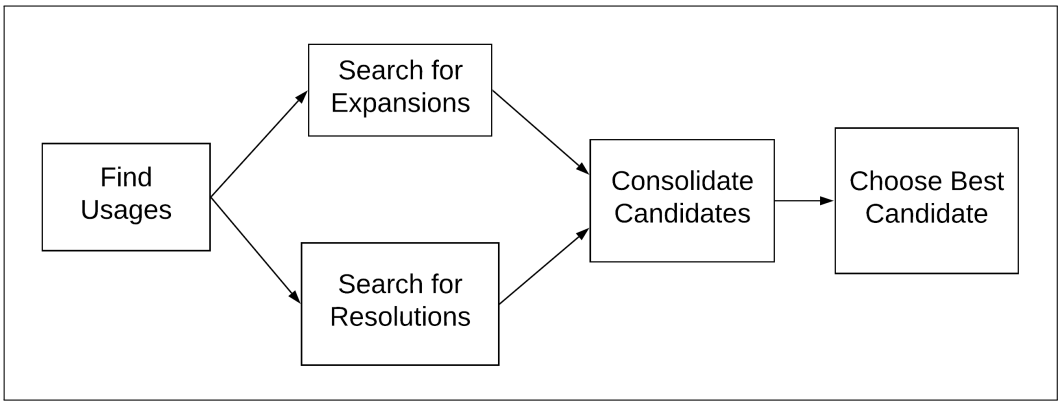


Figure 5: Overview of the abbreviation resolving process

1. Find usages of the given abbreviation in documentation
2. Search the scope of each usage for resolutions
   (a) Technique #1: Search for acronym-expansions assuming the abbreviation is an acronym
   (b) Technique #2: Search for context-aided resolutions regardless of the type of abbreviation
3. Aggregate all resolution matches together and assign weights based on confidence in the match

In Sections 3.3 and 3.4 below, I explain in detail the two resolution-search techniques that I use simultaneously. Table 1 towards the end of this section contains a summary of my different techniques/rules and their applicability.

3.2 Search Area

A key part of my approach is to search for resolutions in the documentation that accompanies source code. Both my acronym-expansion and context-based search techniques are tailored to run specifically on published source code documentation in HTML format. Searching for resolutions in the documentation has tended to be largely overlooked in the state of the art.

3.3 Acronym-Expansion Technique

My acronym-expanding technique searches for expansions to abbreviations by searching for consecutive words that begin with the letters that form the abbreviation in question. This search is implemented using regular expressions constructed using the letters that make up the abbreviation.

The technique works on acronyms as defined in Section 1.4, with the exception that it does not work for single-letter acronyms. Examples of single-letter acronyms include *l* which commonly stands for '**l**ength' or '**l**iter', depending on context; and *g* which may stand for '**g**ravity' or '**g**rams'. My acronym-expanding technique tends to match too many false positive possible expansions for single-letter acronyms, effectively rendering the search useless. Thus, single-letter acronyms benefit more from the context-based technique, described in Section 3.4.

The acronym-expansion technique follows the following steps:

1. **Get a list of possible abbreviations.** *This step is required when running the algorithm for testing; see Section 3.5 for details.* The algorithm will never be able to tell for sure what is an abbreviation and what is not. However, there are some rules I can put in place to get as accurate a list of abbreviations as possible out of all the words contained in the document.
   The steps consist of:
   (a) Get a list of words, defining a *word* as all alphabetical characters that follow (i) Whitespace, (ii) Punctuation, or (iii) Parentheses;
   (b) Filter out valid dictionary words

(c) Filter out plurals of uppercase words (e.g. UIs, BMPs);
(d) Filter out single-letter words and words of length greater than 3 letters.

2. **Get initial list of expansions.** An initial list of expansions is found by first doing a character-by-character expansion search, implemented with regular expressions, for sequences of words that each start with consecutive letters in the acronym. This expansion search is done in two parts:
   (a) **Searching in regular sentences.** In regular sentences where words may be separated by both whitespace and punctuation, I search for expansions by allowing individual words in an expansion to be separated by any one of the following: (i) An unlimited number of whitespace characters except newline characters, (ii) a single hyphen, or (iii) a single underscore. The arbitrary amount of whitespace is to allow for varying amounts of spacing between words for formatting purposes; the hyphen or underscore is to allow hyphenated or underscore-separated words (also knows as *snake-case formatting*) to be considered as separate words that can be part of an expansion.
   (b) **Searching in camel-cased words.** In camel-cased terms, I parse individual words at the natural word boundaries created by camel case.

3. **Filter the list of expansions.** The expansions derived from Step 2 are filtered using different rules, to derive a list of hopefully high-quality expansions, from which a final expansion will eventually be selected. These rules are:
   - **Checking that each word in an expansion is an actual word.** For every expansion in the initial list, the algorithm that checks that each word in the expansion is either: (i) An English word, by checking whether the word is in the *words.dict* list, or (ii) a commonly-used proper noun used in English, by checking whether the word is in the *properNouns.dict* list.
   - **Limiting words in expansions to >2 characters in length.** This is to eliminate insignificant words that are not often included in the acronyms, and by extension, if they are present in a possible expansion, the expansion is likely incorrect.

- **Ensuring words in expansions are not stop words.** Similar to the rule above, this rule filters out words in expansions that are stop words, and thus also unlikely to be part of a correct expansion, by checking that words are not in the *my.stop* stop word list.
- **Checking that expansions are either nouns or adjectives.** This rule aims to build on the stop words / insignificant words rules above, exploiting a characteristic in expansions that words usually belong to only the Nouns or Adjectives parts of speech.

3.4 Context-based Search Technique

The context-based search technique uses grammar- and punctuation-based clues to locate resolutions within the scope of an abbreviation. Mimicking an intelligent search as a human would perform it, this technique searches the immediate to near scope of the abbreviation for English words, phrases and punctuation that tend to precede or follow the resolution for a neighboring abbreviation, and subsequently use these clues to zero in on potential resolutions.

This technique is effective on both types of abbreviations, but is particularly useful for domain terms. This is because by their very definition as outlined in Section 1.4, domain terms have no hope of being expanded by my acronym-expansion technique, or any other approaches in the predominantly acronym-expansion-based state of the art.

The context-based technique is also useful in cases where, even if the acronym-expansion technique matches an acronym with its correct expansion, the context-based technique gives a more meaningful expansion. For example, in the sentence fragment "Let p be the number of post offices," the acronym-expansion technique might match p to expand to "post offices", but the context-based technique, using the Variable-Definition rule defined below, matches "the number of post offices" – a much more contextual expansion.

Below, I outline the rules that comprise the context-based search technique.

3.4.1 Parts of Speech rule

Many resolutions just precede the terms they define. For example, in Figure 6, there is a clear explanation that x is a "private key". Thus, when we find

that x is an abbreviation to be expanded, I can search in the abbreviation's preceding vicinity for resolutions.

Figure 6: Example Part-of-Speech (Noun Phrase) Resolution

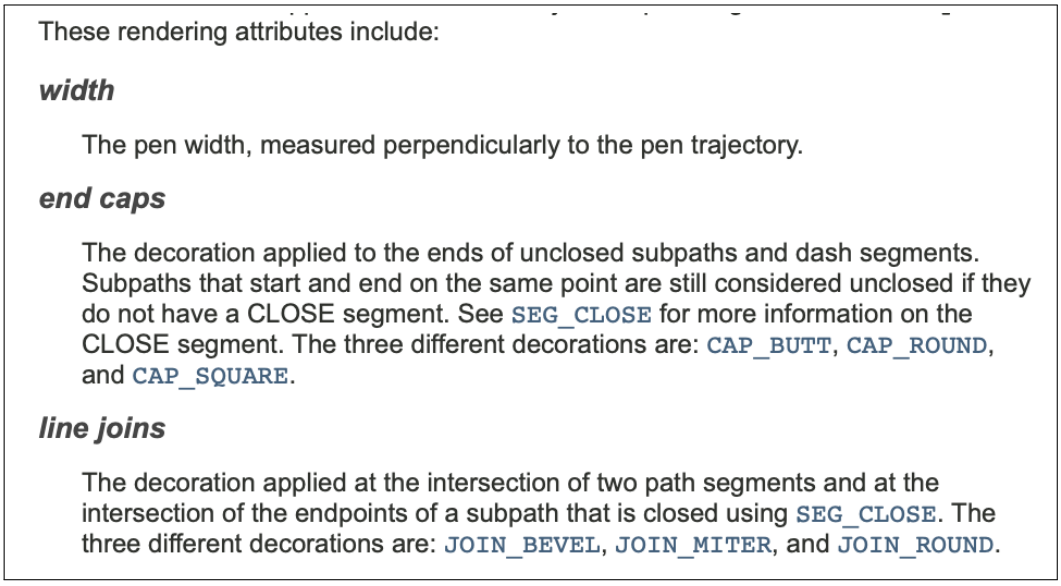
The parts-of-speech rule looks for resolutions of the above form, and filters them by ensuring they conform to either of the two common parts-of-speech patterns that such resolutions usually exhibit:

- **Standalone nouns.** The first kind is where the resolution is a single, standalone noun, found in a phrase of the form *determiner–noun–abbreviation*;
- **Noun Phrases.** The other kind is where the resolution is a noun phrase, found in a phrase of the form *determiner–adjective–noun–abbreviation*. My parts-of-speech rule currently only supports noun phrases of the aforementioned structure, with a two-word noun phrase.

3.4.2 Description List rule

The description-list rule takes advantage of the description list formatting type that exists in HTML, for use in formatting data consisting of a term and a corresponding definition, and to be rendered in such a way that this term-definition distinction is easily recognizable by readers. Figure 7 below shows an example of a description list as it appears on a page of Java SE Documentation, that uses the description-list HTML feature to display it in this format.

Owing to the description-lists feature being used solely for the purpose of describing/defining terms, this makes a description list contained in a


These rendering attributes include:

width

The pen width, measured perpendicularly to the pen trajectory.

end caps

The decoration applied to the ends of unclosed subpaths and dash segments. Subpaths that start and end on the same point are still considered unclosed if they do not have a CLOSE segment. See SEG_CLOSE for more information on the CLOSE segment. The three different decorations are: CAP_BUTT, CAP_ROUND, and CAP_SQUARE.

line joins

The decoration applied at the intersection of two path segments and at the intersection of the endpoints of a subpath that is closed using SEG_CLOSE. The three different decorations are: JOIN_BEVEL, JOIN_MITER, and JOIN_ROUND.

Figure 7: Example Description-List Formatting in HTML

documentation page an ideal candidate for parsing the definitions to the terms it contains. Figure 8 shows a snippet of the HTML code underlying the description list shown above. My technique parses this underlying HTML, identifying a description list by searching for the `<dl>` tag, then identifying each term and its definition by using the `<dt>` and `<dd>` tags, respectively.

Definitions for terms found in description lists tend to be high in accuracy.

3.4.3 Variable Definition rule

The Variable Definition rule leverages statements in documentation of the form "Let p be *the number of post offices in the area*". Such sentence structure is very easily parsed by automated methods, because they are arguable the most explicit definition of a term to mean something, thus yielding high-accuracy term and expansion results.

3.5 For Testing: Finding Abbreviations in Documentation as a Proxy

My abbreviation-resolution techniques can be run in two 'modes': *Actively*, to search for resolutions for a given abbreviation/domain term (or list thereof)

```
 <dl compact>
 <dt><i>width</i>
 <dd>The pen width, measured perpendicularly to the pen
trajectory.
 <dt><i>end caps</i>
 <dd>The decoration applied to the ends of unclosed subpaths and
 dash segments.  Subpaths that start and end on the same point are
 still considered unclosed if they do not have a CLOSE segment.
 See <a href="../../java/awt/geom/PathIterator.html#SEG_CLOSE">
<code>SEG_CLOSE</code></a>
 for more information on the CLOSE segment.
 The three different decorations are: <a
href="../../java/awt/BasicStroke.html#CAP_BUTT">
<code>CAP_BUTT</code></a>,
 <a href="../../java/awt/BasicStroke.html#CAP_ROUND">
<code>CAP_ROUND</code></a>, and <a
href="../../java/awt/BasicStroke.html#CAP_SQUARE">
<code>CAP_SQUARE</code></a>.
```

Figure 8: Underlying HTML Code for a Description List

by searching documentation; or *Passively*, usually for testing purposes, whereby an abbreviation-identifying search is first done on the documentation to get a list of abbreviations/domain terms to resolve, and then the techniques are run.

3.6 Consolidating and Ranking Expansions

The resolutions matched by the two techniques above are consolidated and ranked using an order of priority, formulated based on the accuracy of each technique's/rule's resolutions matched. The consequence of this priority is that the resolutions matched by a higher-priority technique will be considered higher-likelihood candidates for actual consideration than those matched by techniques with lower priorities. The order of priority goes as follows, in decreasing order:

1. Description List rule
2. Variable Definition rule
3. Parts of Speech rule
4. Abbreviation-Expansion technique

Rule	Works on Acronyms	Works on Domain Terms	Example Occurrence (Term in wavy underline, Expansion in underline, Key Identifying Characteristics in **bold**, "..." indicates that term and expansion may be located far apart.)
Acronym-Expansion	✓		AES ... **A**dvanced **E**ncryption **S**tandard
Parts of Speech	✓	✓	**the** destination array buf
Description List	✓	✓	*Phrase* | *Meaning* carShd | car to be shared
Variable-Definition	✓	✓	**Let** t0 **be** the experiment start time.

Table 1: Summary of Approaches

4 Evaluation

4.1 Overview

In evaluating the work that has been done so far, I would ideally seek to answer the question: "To what extent do my techniques resolve the meanings of abbreviations and terms as well as a human can?" However, the current state of this research does not present the techniques as a one-stop solution for expanding all abbreviations and domain terms that may be present in a piece of code or documentation. Thus, it would be more accurate to evaluate the effectiveness of my techniques on only those abbreviations/domain terms present in the contexts that each technique is targeted to handle, or in other words that each technique should theoretically be able to handle.

With this in mind, I seek to answer the following revised, more targeted question:

> **To what extent does each technique resolve the meanings of those abbreviations/domain terms that it should theoretically be able to resolve, as well as a human can?**

4.2 Defining the Metrics

With the evaluation research question as defined in Section 4.1 in mind, I define the following metric for evaluating the performance of my techniques:

> **Recall**: The percentage of abbreviations/domain terms that each technique is able to correctly resolve, out of all the abbreviations/domain terms it should theoretically be able to resolve.

4.3 Evaluation Setup

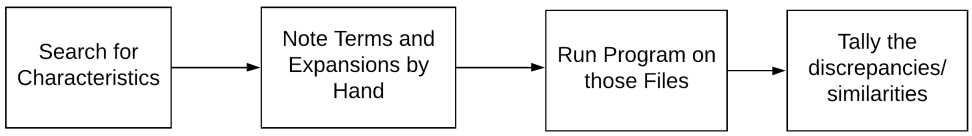


Figure 9: Evaluation Setup

I evaluated my approach by individually assessing the effectiveness of the following major components:

1. Abbreviation-Expansion technique
2. Description-List rule
3. Variable-Definition rule
4. Parts-of-Speech rule

The assessment consisted of the following steps, as illustrated in Figure 9:

1. First, for each component, a sample population was picked, consisting of abbreviations or domain terms (where applicable) that the component's techniques should theoretically be able to expand. For example, an ideal sample population for component 4 (the parts-of-speech rule) would be 30 instances of abbreviations/domain terms whose resolutions can be arrived at using the strategies that we aimed to encompass in the parts-of-speech rule.
2. Next, simultaneously:
   (a) Each component's computer program was run on its sample population;

(b) I manually populated a 'model set' (or *gold set*) of each sample population's abbreviations/domain terms, and the best resolution I was able to determine for them.

3. Finally, the recall was calculated by comparing the resolutions derived from Step 2a against all the resolutions in the gold set.

The results of the assessment are discussed in Section 4.4. In Section 4.5, I suggest some relatively-straightforward enhancements to the techniques that would significantly increase the recall achieved. Finally, in Section 4.6, I outline some major enhancements that can inch the recall closer to 100% on the sample population.

4.4 Results

The recall achieved by running the assessment for each of the four components is graphed in Figure 10. Each of the four labeled bars corresponds to a plot of the recall achieved by each of the four components being evaluated. The y-axis quantifies this recall.

As we see, the acronym-expansion technique has the highest recall of all the components evaluated. The description-list rule comes in second with a recall of over 50%, and the variable-definition and parts-of-speech rules tie for third at around 42%. For a possible explanation of the acronym-expansion technique's significantly higher recall rate over the recall rates of the three context-based rules, see Section 4.5.3.

4.5 Easy-to-Implement Improvements

When creating the gold set by hand, I noticed that there were a significant number of cases in which the present techniques failed to correctly resolve the terms at hand because of relatively easy-to-rectify shortcomings. I detail these possible improvements in this section, and predict the gains on performance that we can hope to achieve following their implementation.

Note: No minor *improvements can be made to either the acronym-expansion technique or the parts-of-speech rule to increase their recall on the sample population.*

Figure 10: Results of the Evaluation

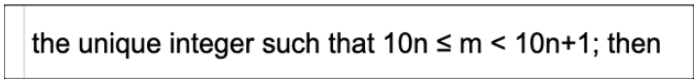
4.5.1 Improvements to the variable-definition rule

In Figures 11a and 11b, we see two resolutions that the current iteration of the variable-definition rule returned. In both cases, the general text is correct, but the rule simply overran the bounds for the correct resolution. In both cases, the resolution should have ended at the first semicolon encountered.

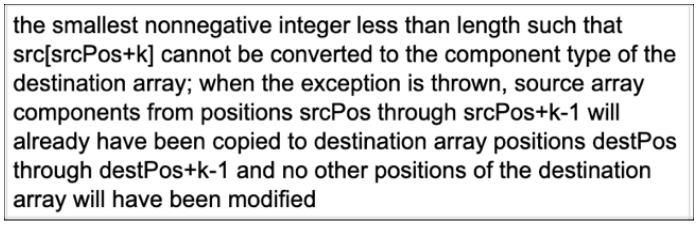
In Figures 11c and 11d, we see multiple variable definitions listed using a single "let" keyword. In 11c, the three variable definitions are formatted in a standard comma-delimited list, and in 11d, the two variable definitions are separated by the "and." Both these standard list formats should be accounted for when searching for variable definitions.

4.5.2 Improvements to the description list rule

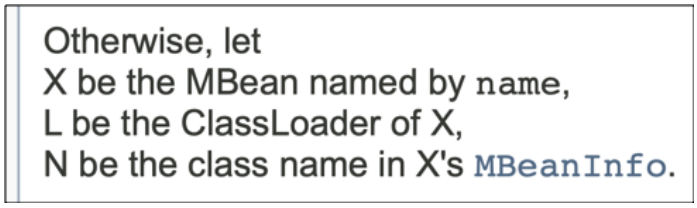
In Figure 12, we see a resolution that the current iteration of the description-list rule returned. Similar to the lessons learned with regards to the variable-definition rule's figures 11a and 11b above, the resolution should have terminated at the end of the first line, thus: "A descriptive message to be placed in the dialog box."



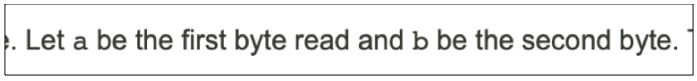
(a)



(b)



(c)



(d)

Figure 11: Instances of the sample population where the Variable-Definition rule failed to correctly resolve terms.

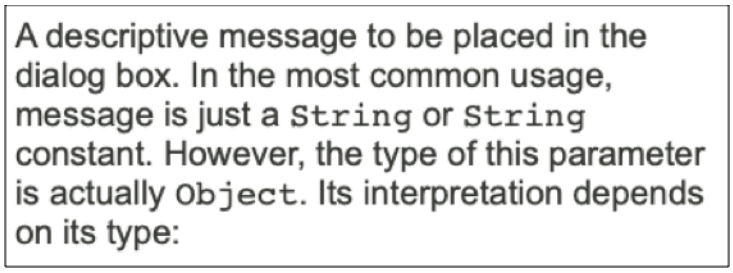


Figure 12: An instance of the sample population where the Description-List rule failed to correctly resolve terms.

4.5.3 Predicted Performance Gains post Improvements

The predicted gains in recall that we would achieve by implementing the above suggested improvements are graphed in Figure 13. Each of the four labeled bars corresponds to a plot of the recall predicted to be achieved by each of the four updated components; the y-axis quantifies this recall.

Some of the bars have two shades. The lower shaded segment depicts the actual, measured recall of the techniques in their current form, as illustrated earlier in Figure 10. The upper shaded segment depicts the gains in recall that are achieved by the implementation of the above suggested improvements.

As we see, following implementation of the suggested minor improvements to the variable-definition and description-list rules, their recall rises significantly. The description-list rule achieves 100% recall on the sample population, and the variable-definition rule achieves over 75% recall. Unfortunately, no small improvements can be made to improve the recall of the acronym-expansion technique and the parts-of-speech rule.

For those wondering whether there is an explanation for the acronym-expansion technique's significantly higher actual, measured recall rate over the actual, measured recall rates of the three context-based rules, I cannot comment on any specific reason that that is the case. However, after looking at the data in Figure 13, we may reasonably presume that minor improvements to the acronym-expansion technique, of the likes of those suggested in the preceding sections for the variable-definition and description-list rules, were simply incorporated earlier on in the development of the acronym-expansion rule.

4.6 Major Advancements

With the goal of eventually getting the techniques to reach a 100% recall, in this section I outline some of the challenging enhancements that will need to be implemented in order to inch the techniques closer to perfect performance. The discussion of how these enhancements would be implemented, and how to account for edge cases, is beyond the scope of this paper.

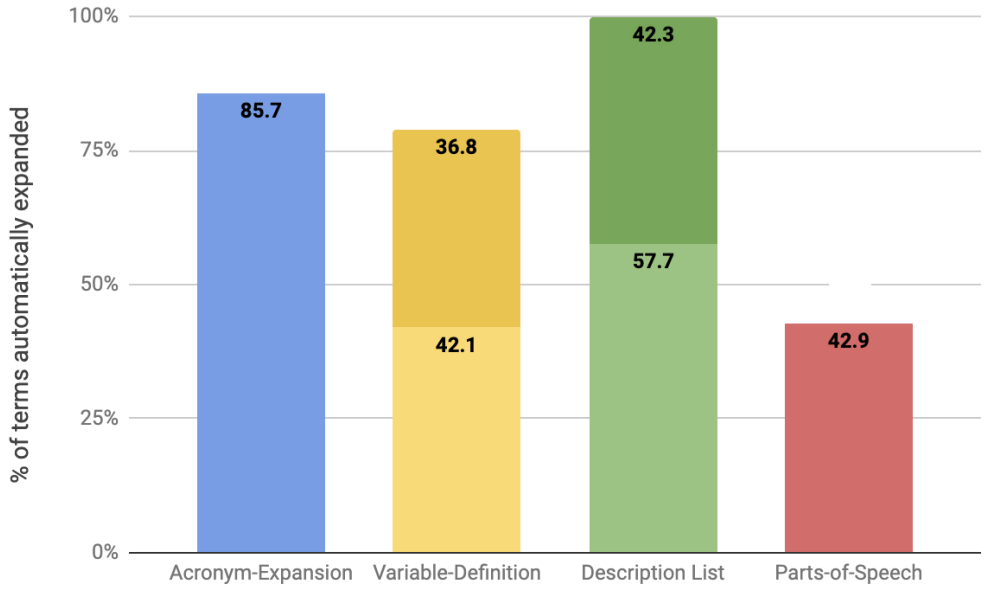
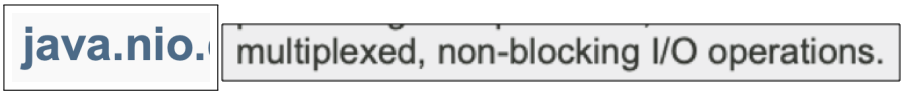


Figure 13: Predicted results following implementation of suggestions.

4.6.1 Major Improvements to the abbreviation-expansion technique

In Figure 14, we see the acronym "NIO," and its correct expansion, "**n**on-blocking **I**/**O**." This expansion is unique in several ways: First, there is a word ("blocking") between two of the words in the expansion; Second, part of the acronym ("IO") is an abbreviated form of a widely-used acronym, *I/O*, which expands to "**I**nput/**O**utput." A future iteration of the acronym-expansion technique will not only have to take into account possible non-abbreviated words like "blocking" between words in the actual expansion, but will also have to decide whether to expand "IO" to the acronym "I/O," or to its full expansion, "Input/Output," which would likely require a recursive expanding process. Regardless, the current technique is far away from considering "non-blocking I/O" a legitimate expansion for "NIO."

In Figure 15, we see the acronym "JAX," and its correct expansion, "**J**ava **A**PI for **X**ML." The improvements required to correctly resolve this expansion are similar to those for the "NIO" example above: Deciding whether to retain or expand acronyms within the expansion (in this case, "API" and



(a) (b)

Figure 14: Instance 1 of the sample population where the Acronym-Expansion technique would require major enhancements to correctly resolve the acronym.

(a) (b)

Figure 15: Instance 2 of the sample population where the Acronym-Expansion technique would require major enhancements to correctly resolve the acronym.

"XML"); and allowing for non-abbreviated words interspersing the abbreviated words in the expansion (in this case, "for").

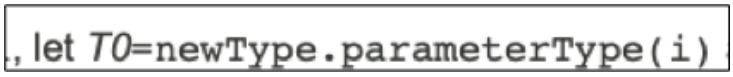
4.6.2 Major Improvements to the variable-definition rule

In Figure 16a, we see a special case of the variable-definition rule: An equals-sign is used to assign the meaning "`newType.parameterType(i)`" to the variable "T0". A future iteration of the variable-definition rule should take into account statements of the form "Let $x = y$," in addition to the existing "Let x be y."

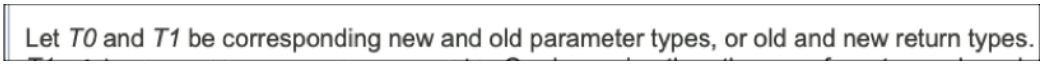
In Figure 16b, we see a case of variable definition that is unique in several ways: One, that the terms "T0" and "T1" are assigned their meanings in a sentence of structure "Let a and b be x and y, respectively"; Two, they are assigned two different meanings each, with the assumption that the reader will infer, using context, which meaning to substitute. A future iteration of the rule should take these quirks into account.

4.6.3 Major Improvements to the description-list rule

Following the implementation of the improvements suggested in the previous sections, the description-list rule was able to achieve a 100% recall. Thus, I have no major improvements to suggest for this rule.



(a)



(b)

Figure 16: Instances of the sample population where the Variable-Definition rule would require major enhancements to correctly resolve the terms described.

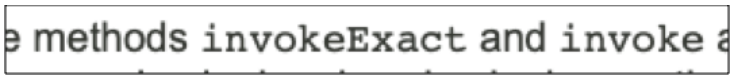


Figure 17: An instance of the sample population where the Parts-of-Speech rule would require major enhancements to correctly resolve the terms described.

4.6.4 Major Improvements to the parts-of-speech rule

In Figure 17, we see an example of two terms, "`invokeExact`" and "`invoke`," both labeled "methods." The correct parts-of-speech resolution would be to resolve each term as being a "method;" however this requires recognizing that both terms are bound to the plural noun phrase "methods." A future iteration of the parts-of-speech rule would take this into account, including if the resolution "methods" applied to a list of, say, comma-delimited terms.

5 Conclusions and Future Work

5.1 Conclusion

In this thesis, I have presented a technique for resolving the meanings of terms in source code. For programmers unfamiliar to pieces of source code, resolving the meanings of terms and short forms used in the code can be invaluable compared to the time it can take to look up those meanings. Additionally, in some specialized domains, the meanings of short forms may not be well documented online, and would force the programmer to have to look up usages of the terms in the documentation.

In the future, these techniques would be most useful in the form of a ready-to-use tool that programmers can summon instantly to use within their favourite Integrated Development Environment (IDE).

In the meanwhile, however, the techniques need certain improvements, as discussed in the next section.

5.2 Future Work

- Using online/existing databases of information to search for resolutions to commonly-used short-forms, and also resolutions to acronyms that are simply proper nouns like company names; diseases; chemicals, etc. This does not necessarily mean we should not perform our own search for these abbreviations, because in certain contexts, a commonly-used short form may be used within a local scope to stand for something completely different from the generally-used meaning.
  For example, in the *java/util/Formatter* Java documentation page, the term 'OS' stands for 'output stream', not the almost universally-used meaning 'operating system'.
- My techniques at this point do not search the documentation *accompanying* individual bits of source code, but instead search only the existing documentation out there. Additionally, they are currently only designed to search through Java documentation. In the future, they can be tailored to run on documentation pages for more programming languages and also be integrated to search for resolutions in a piece of code or or software's accompanying documentation.
- As was seen in Section 4, the approaches in their current form tend to match many terms that are not actual candidates to be expanded, such as numbers, strings, data types and truth values. Improvements can be made to the approaches' accuracy of matching term candidates that should be expanded.

References

[1] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. IEEE transactions on software engineering, 28(10):970–983, 2002.

[2] Bruno Caprile and Paolo Tonella. Restructuring program identifier names. In icsm, pages 97–107, 2000.

[3] Jeffrey T Chang, Hinrich Schütze, and Russ B Altman. Creating an online dictionary of abbreviations from medline. Journal of the American Medical Informatics Association, 9(6):612–620, 2002.

[4] Florian Deissenboeck and Markus Pizka. Concise and consistent naming. Software Quality Journal, 14(3):261–282, 2006.

[5] Henry Feild, David Binkley, and Dawn Lawrie. An empirical comparison of techniques for extracting concept abbreviations from identifiers. In Proceedings of IASTED International Conference on Software Engineering and Applications (SEA'06), 2006.

[6] Inc. Google. Google scholar.

[7] Emily Hill, Zachary P Fry, Haley Boyd, Giriprasad Sridhara, Yana Novikova, Lori Pollock, and K Vijay-Shanker. Amap: automatically mining abbreviation expansions in programs to enhance software maintenance tools. In Proceedings of the 2008 international working conference on Mining software repositories, pages 79–88. ACM, 2008.

[8] Dawn Lawrie, Henry Feild, and David Binkley. Extracting meaning from abbreviated identifiers. In Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007), pages 213–222. IEEE, 2007.

[9] Youngja Park and Roy J Byrd. Hybrid text mining for finding abbreviations and their definitions. In Proceedings of the 2001 conference on empirical methods in natural language processing, 2001.

[10] James Pustejovsky, José Castano, Brent Cochran, Maciej Kotecki, and Michael Morrell. Automatic extraction of acronym-meaning pairs

from medline databases. Studies in health technology and informatics, (1):371–375, 2001.

[11] L Ratinov and Ehud Gudes. Abbreviation expansion in schema matching and web integration. In Proceedings of the 2004 IEEE/WIC/ACM International Conference on Web Intelligence, pages 485–489. IEEE Computer Society, 2004.

[12] Juergen Rilling and Tuomas Klemola. Identifying comprehension bottlenecks using program slicing and cognitive complexity metrics. In 11th IEEE International Workshop on Program Comprehension, 2003., pages 115–124. IEEE, 2003.

[13] Ariel S Schwartz and Marti A Hearst. A simple algorithm for identifying abbreviation definitions in biomedical text. In Biocomputing 2003, pages 451–462. World Scientific, 2002.

[14] Kazem Taghva and Jeff Gilbreth. Recognizing acronyms and their definitions. International Journal on Document Analysis and Recognition, 1(4):191–198, 1999.

[15] Wilson Wong, Wei Liu, and Mohammed Bennamoun. Integrated scoring for spelling error correction, abbreviation expansion and case restoration in dirty text. In Proceedings of the fifth Australasian conference on Data mining and analystics-Volume 61, pages 83–89. Australian Computer Society, Inc., 2006.

[16] Stuart Yeates, David Bainbridge, and Ian H Witten. Using compression to identify acronyms in text. arXiv preprint cs/0007003, 2000.

Listings

Appendix A Techniques Source Code

A.1 Abbreviation-Expansion technique

Listing 1: Abbreviation-Expansion technique

```
import os, sys, re, html2text, itertools, csv, helpers, pandas
   as pd
from tabulate import tabulate
from operator import itemgetter
from collections import OrderedDict

def getText(match, group=0):
  return match[group]

def getExpnStr(acronymExpnMatch):
  # cleans the acronym-expansion-match's groups
  # then joins them using spaces to get the full string of the
      expansion
  expnGroups = [group.strip().lower() for group in
     acronymExpnMatch.groups()]
  return ' '.join(expnGroups)

def stripApos(string):
  return ''.join((char for char in string if char is not "'"))

def filterDupes(matchList, stringGetter = getText):
  # Filter out duplicates IN PLACE
  frequencyPairs = countMatches(matchList, stringGetter)
  index = 0
  for index, pair in enumerate(frequencyPairs):
```

```
    matchList[index] = pair[0]
  del matchList[index+1:]

def countMatches(matchList, stringGetter = getText, ignorecase
    = False):
  """ Takes any list of matches and returns a consolidated
      list of [match, frequency] pairs.
    -- Takes an optional stringGetter function that will
       return a different string for a match than the default
       function getText. Useful when the specific type of
       matches passed must be consolidated on a match-string
       that is not necessarily the 0th-group string."""
  matchStrUniques, matchUniques = [], []
  for match in matchList:
    mStr = stringGetter(match)
    if mStr.lower() not in matchStrUniques: # match is fresh
      matchStrUniques.append(mStr.lower())
      matchUniques.append([match, 1])
    else: # match is a duplicate
      index = matchStrUniques.index(mStr.lower())
      matchUniques[index][1] += 1
  return matchUniques

def filterCheckEachSegment(matches, cond, extraParams=()):
  valid_matches = []
  for match in matches: # for all passed matches
    allGroupsPass = True
    for group in match.groups():
      if not cond(group, *extraParams): # evaluate the
          condition for each segment
        allGroupsPass = False
        break
    if allGroupsPass:
      valid_matches.append(match) # if all segments of current
          match are valid
  return valid_matches

def nonWordOccursAsParamVar(nonWord, text):
```

```
  paramVarMatches = list(re.finditer(r'(?<= \n)[a-zA-Z0-9_.]+ [
     a-zA-Z0-9_. ]+\([a-zA-Z0-9_,. ]*[a-zA-Z0-9_.]+ %s(,[a-zA-
     Z0-9_., ]+)?\)(?=\n\n)' % nonWord, text))
  if paramVarMatches:
    paramTypes = set()
    for pvMatch in paramVarMatches:
      pvTypeMatch = re.search(r'[a-zA-Z_]+(?= %s(,|\)))' %
         nonWord, pvMatch[0])
      paramTypes.add(pvTypeMatch[0])
    return paramTypes
  else:
    return None


def main():
  file_NWs = [] # master non-word store
  expns = []  # master expn store
  # Populate list of files to analyse
  try:
    allFiles = helpers.getFileList(sys.argv[1])
  except IndexError:
    print('\tUsage: python3 script_getXYZ.py [...[/relative/
       path/to/file]] (OR) [absolute/path/to/file]')
    sys.exit()
  # Parse and analyse each file
  for filePath in allFiles:
    dirPath, fileName = os.path.split(filePath)
    print('------- FILE :', fileName)
    print('(1)\tGetting Non-Words...')
    # STEP 1: STRIP HTML
    with open(filePath, 'r') as f:
      text = helpers.stripHTML(f.read())
      # print(text) # debug

    # STEP 2: IDENTIFY WORDS AND THEN GET NON-WORDS
    # Step 2-(a) Get all WORDS
    all_words = list(re.finditer(r""" # match words as all
       alphabetical characters that follow (1) whitespace (2)
```

```
    punctuation (3) brackets
    (?<=[\s,.?!(){}])  # Lookbehind assertion for whitespace/
       punctuation/bracket
    [a-zA-Z]+    # All following alphabetical characters
    ('[a-zA-Z]+)?   # Optional apostrophe with more
       alphabetical characters following
    """, text, re.VERBOSE))
  # Step 2-(b) Get a list of non-words
  try: # use cache of non-words from previous run, if
     specified and one exists
    if sys.argv[2] == '-u':
      with open('non_word_cache', 'r') as cache:
        cached_non_words = [entry.split('\t') for entry in
           cache.read().split('\n')] # list of tuple-pairs
        cached_non_words = [non_word[1] for non_word in
           cached_non_words if non_word[0]==filePath] # list
           of the current file's non-words
        if len(cached_non_words) == 0:
          raise ValueError # the non-word cache is empty
        non_words = [word for word in all_words if getText(
           word) in cached_non_words] # check against cache
           but maintain non-words as being regex match
           objects
    else:
      raise ValueError
  except: # freshly derive a list of non-words by checking
     for absence in the chosen dictionary
    with open('AMAP Downloads/modified/all_five_lists.dict')
       as g:
      amapDict = g.read().split('\n')   # open and store the
         dictionary
    amapDict = [word.lower() for word in amapDict] # clean up
       dictionary casing
    non_words = [word for word in all_words if (
      not (getText(word)[:-1].isupper() and getText(word)
         [-1]=='s') # weed out plurals of uppercase non-
         words (e.g. UIs, BMPs)
```

```
    and len(getText(word)) > 1 and len(getText(word)) < 4 #
        length is either 2 or 3 letters
    and stripApos(getText(word)).lower() not in amapDict #
       word is not a valid dictionary word
    )]
filterDupes(non_words) # filter out duplicates
file_NWs += [[fileName, getText(non_word)] for non_word in
   non_words] # add to master non-word list
# for non_word in non_words:
#  print(getText(non_word))
# print(len(non_words))

# filter nw by scope
for x in range(1):

  print('(2)\tGetting Expansions...')
  # STEP 3: Assuming the non-words are acronyms, search
     for their expansions.
  for non_word in non_words:
    valid_matches = []
    nw = getText(non_word)

    # Approach 2(a): Word boundaries
    lb = r'(?<![a-zA-Z])'   # lookbehind -ve for word
       boundary
    expn = r"""
      (%s       # first letter of acronym
      [a-zA-Z]+)     # one or more alphabet characters
        """
    island = r'(?:[ \t\r\f\v]+|-|_)' # non-cap group,
       islands as arbitrary num of whitespace characters
       except newline, or hyphen, or underscore
    la = r'(?![a-zA-Z])'   # lookahead -ve for word
       boundary
    if len(nw) == 2: # acronyms of length 2
      expr = lb + expn + island + expn + la
      matches_1 = re.finditer(expr % (nw[0], nw[1]), text,
         re.VERBOSE | re.IGNORECASE)
```

```
elif len(nw) == 3: # of length 3
  expr = lb + ((expn + island) * 2) + expn + la
  matches_1 = re.finditer(expr % (nw[0], nw[1], nw[2]),
     text, re.VERBOSE | re.IGNORECASE)
elif len(nw) == 4: # of length 4
  expr = lb + ((expn + island) * 3) + expn + la
  matches_1 = re.finditer(expr % (nw[0], nw[1], nw[2],
     nw[3]), text, re.VERBOSE | re.IGNORECASE)

# Approach 2(b): Camelcased expansions
nw_uc = nw.upper()
# print(nw_uc)
first = r"""
    ((?:(?<![a-zA-Z])%s  # first letter of expansion,
       either as lowercase at a word boundary...
    |%s)     # ... or as uppercase wherever in a line
    [a-z]+)     # followed by one or more LOWERCASE
       alphabetical characters
      """
rest = r"""
    (%s      # first letter only as UPPERCASE
    [a-z]+)     # followed by one or more LOWERCASE
       alphabetical characters
      """
if len(nw) == 2: # acronyms of length 2
  expr = first + rest
  matches_2 = re.finditer(expr % (nw.lower()[0], nw_uc
     [0], nw_uc[1]), text, re.VERBOSE)
elif len(nw) == 3: # of length 3
  expr = first + rest + rest
  matches_2 = re.finditer(expr % (nw.lower()[0], nw_uc
     [0], nw_uc[1], nw_uc[2]), text, re.VERBOSE)
elif len(nw) == 4: # of length 4
  expr = first + rest + rest + rest
  matches_2 = re.finditer(expr % (nw.lower()[0], nw_uc
     [0], nw_uc[1], nw_uc[2], nw_uc[3]), text, re.
     VERBOSE)
```

```
matches = itertools.chain(matches_1, matches_2) # chain
    both b

valid_matches = matches # XXX: DO NOT COMMENT OUT

# STEP 4: Validate each possible expansion...
# Filter (a): Check each segment against a dictionary
with open('AMAP Downloads/modified/words+proper.dict')
   as h:
  checkDict = h.read().split('\n') # open and store the
      dictionary
checkDict = [word.lower() for word in checkDict] #
   clean up dictionary casing
valid_matches = filterCheckEachSegment(valid_matches,
   lambda group, checkDict: group.lower() in checkDict,
    (checkDict,))

# Filter (b): Expansions must be of > 2 chars in
   length?
valid_matches = filterCheckEachSegment(valid_matches,
   lambda group: len(group) > 2)

# Filter (c): Expansions must not be stop words (i.e.
   in my.stop)
with open('AMAP Downloads/my.stop') as h:
  stopWords = h.read().split('\n')
valid_matches = filterCheckEachSegment(valid_matches,
   lambda group, stopWords: group.lower() not in
   stopWords, (stopWords,))

# Filter (d) : Parts of speech filtering
# Expansions must be either nouns or adjectives
with open("Parts of Speech files/Ashley Bovan's lists/
   nouns/91K nouns.txt", 'r') as nouns, open("Parts of
   Speech files/Ashley Bovan's lists/adjectives/28K
   adjectives.txt", 'r') as adjs:
  nouns_adjs = nouns.read().split('\n') + adjs.read().
     split('\n')
```

```
        nouns_adjs = [word.lower() for word in nouns_adjs]
        valid_matches = filterCheckEachSegment(valid_matches,
            lambda group, nouns_adjs: group.lower() in
            nouns_adjs, (nouns_adjs,))

        # STEP 5(a): Consolidate matches for current non-word
        expns_counts = countMatches(valid_matches, getExpnStr)

        # STEP 5(b): Add current non-word's expansions to
            master list
        expns.extend([[dirPath, fileName, nw.lower(),
            getExpnStr(expn_count[0]), expn_count[1]] for
            expn_count in expns_counts])

    print('')

    # Save list of non-words to an external file for quick
        cached access next time
    new_lines_to_write = [filePath + '\t' + getText(non_word)
        for non_word in non_words]
    old_lines_to_rewrite = []
    try:
      with open('non_word_cache', 'r') as cache: # there is an
          existing non-word cache
        old_lines_to_rewrite = [line for line in cache.read().
            split('\n') if line.split('\t')[0]!=filePath] # only
             let other files' cached non-words remain, so the
            current file's nw's can be freshly added
    except:
      pass
    with open('non_word_cache', 'w') as cache:  # create a new
        file / overwrite the existing one
      cache.write('\n'.join(new_lines_to_write) + '\n' + '\n'.
          join(old_lines_to_rewrite))


if __name__ == '__main__':
  main()
```

A.2 Parts-of-Speech rule

Listing 2: Parts-of-Speech rule

```
from bs4 import BeautifulSoup as BeauSoup
import helpers, os, sys, re, time, signal
from itertools import chain

global counter

def parseWordListFile(filePath):
  return {word.lower() for word in open(filePath, 'r').read().
     split('\n') if word and not word.startswith('#')}

def handler(signum, frame):
  # print('pass', counter.getCount())
  raise Exception('Timeout in file %s' % (counter.getCount()))

def main():
  try:
    allFiles = helpers.getFileList(sys.argv[1])
  except IndexError:
    raise

  # SETUP
  t0 = time.time()
  signal.signal(signal.SIGALRM, handler)
  global counter
  counter = helpers.FileCounter(total = len(allFiles), marker =
      500)
  # Read in pos lists, ignoring commented-out words
  DTs = parseWordListFile("Parts of Speech files/modified lists
      for my use/DT for rule_pos.txt")
  AJs = parseWordListFile("Parts of Speech files/Ashley Bovan's
      lists/adjectives/28K adjectives.txt")
```

```
ONs_Hill = parseWordListFile("Parts of Speech files/modified
   lists for my use/O_N sorted.txt")
ONs_TE = parseWordListFile("Parts of Speech files/modified
   lists for my use/Nouns_TE_sorted.txt")
possible_non_nouns = set(chain(*[parseWordListFile(filePath)
   for filePath in [os.path.join("Parts of Speech files/Hill'
   s lists/lists (originals)", fileName) for fileName in ['AJ
   .txt', 'AV.txt', 'EN.txt', 'EN-IRR.txt', 'event_words.txt'
   , 'ING.txt', 'V.txt']]]))
all_potential_nouns = set(chain(ONs_Hill, ONs_TE,
   possible_non_nouns))

for filePath in allFiles:
  try:
    signal.alarm(5)
    dirPath, fileName = os.path.split(filePath)
    soup = BeauSoup(open(filePath, 'r').read(), 'html.parser'
       )

    for i_em in filter(lambda match: len(match.string) <= 15,
           filter(lambda match: match.string is not None,
             soup.find_all(['i', 'em', 'code']))):

      typePl2 = ''
      # Step (1a.)
      prev_text = ''
      current_prev_elem = i_em
      while not re.search(r'\w+\W+\w+\W*$', prev_text) or
         prev_text == '': # while re.search doesn't match, OR
          (for the first iteration) prev_text is an empty
         string
        current_prev_elem = current_prev_elem.previous_element
        prev_text = str(current_prev_elem) + prev_text
      # Step (1b.)
      match = re.search(r'(\w+)\W*$', prev_text)
      if match:
        word3 = match[1]
        if word3.lower() in ONs_Hill_and_TE:
```

```
        # Step (1c.)
        match = re.search(r'(\w+)\W+\w+\W*$', prev_text)
        if match:
          word2 = match[1]
          if word2.lower() in DTs:
            # Step (1c-ii.) WE HAVE A MATCH
            dataFile.addLine([f'{word2.lower()} {word3.upper
                ()} {i_em.string}', filePath])
          elif word2.lower() in AJs:
            typePl2 = 'adj'
          elif word2.lower() in ONs_Hill_and_TE:
            typePl2 = 'noun'
        if typePl2 == 'adj' or typePl2 == 'noun':
          # Step (1c-i-a)
          while not re.search(r'\w+\W+\w+\W+\w+\W*$',
              prev_text) or prev_text == '': # while re.
              search doesn't match, OR (for the first
              iteration: prev_text is an empty string)
            current_prev_elem = current_prev_elem.
                previous_element
            prev_text = str(current_prev_elem) + prev_text
          # Step (1c-i-b)
          match = re.search(r'(\w+)\W+\w+\W+\w+\W*$',
              prev_text)
          if match:
            word1 = match[1]
            if word1.lower() in DTs:
              # Step (1c-i-c) WE HAVE A MATCH
              dataFile.addLine([f'{word1.lower()} {word2.
                  upper()} {word3.upper()} {i_em.string}',
                  filePath])


    counter.increment()
  except (KeyboardInterrupt, EOFError):
    sys.exit()
  except Exception as ex:
```

```
      print('%s: %s, %s' % (type(ex).__name__, ex.args[0] if ex
          .args else '', helpers.PathConv.toShort(filePath)))

  t1 = time.time()
  print('Total time:', t1-t0)

  print('\nSTRONG POS TAG MATCHES\n')
  dataFile_strong.finish()
  print('\nMODERATE-STRENGTH POS TAG MATCHES\n')
  dataFile_mod.finish()
  print('\nWEAK POS TAG MATCHES\n')
  dataFile_weak.finish()

if __name__ == '__main__':
  main()
=
```

A.3 Description-List rule

Listing 3: Description-List rule

```
from bs4 import BeautifulSoup as BeauSoup
import helpers, os, sys, re

def parseLineSegments(line):
  term, path = line
  return {'term':term, 'path':path}

def main():
  try:
    allFiles = helpers.getFileList(sys.argv[1])
    # allFiles = helpers.getFileList('.../java/awt/BasicStroke
        .html')
  except IndexError:
    raise
```

```
counter = helpers.FileCounter(total = len(allFiles), marker =
    500)
dataFile = helpers.DataFileOps(notes = 'Finding definitions
   for terms defined in tabular or definition-lists format',
   shortHandNote = 'tbl', preserveTrailingWhitespace = True,
   filterDupes = False)

DLs = 0
totalDLs = 0
totalDLfiles = 0

for filePath in allFiles:
  try:
    dirPath, fileName = os.path.split(filePath)
    soup = BeauSoup(open(filePath, 'r').read(), 'html.parser'
       )

    for dl in soup.find_all('dl', compact=True):

      DTs, DDs = dl.find_all('dt'), dl.find_all('dd')

      for i in range(len(DTs)):
        dt = str(DTs[i]).replace(str(DDs[i]), '')
        if i < len(DTs)-1:
          dd = str(DDs[i]).replace(str(DTs[i+1]), '')
        else:
          dd = str(DDs[i])


    DLs = len(soup.find_all('dl', compact=True))
    totalDLs += DLs
    if DLs > 0:
      totalDLfiles += 1
      dataFile.addLine([f'{dt}, {dd}',filePath])

    counter.increment()
  except (KeyboardInterrupt, EOFError):
    sys.exit()
```

```
    # except Exception:
    #  print(Exception.args)

  print(f'{totalDLs} TOTAL DLs in {totalDLfiles} TOTAL FILES')
  dataFile.finish()

if __name__ == '__main__':
  main()
```

A.4 Variable-Definition rule

Listing 4: Variable-Definition rule

```
import sys, helpers, re, string

def parseLineSegments(line):
  term, path = line
  return {'term':term, 'path':path}

def main():
  try:
    allFiles = helpers.getFileList(sys.argv[1])
  except IndexError:
    raise

  toPrint = ''

  counter = helpers.FileCounter(total = len(allFiles), marker =
      500)
  dataFile = helpers.DataFileOps(notes = "Finding the y for an
     emphasized/italicized 'x' in sentences like \"Let x be y\"
     ", shortHandNote = 'let-be')
  print('1.X.3')
  for filePath in allFiles:
    try:
      matches = list(re.finditer(r"""
        (Let|let)
```

```
        \s+
          (<em>|<i>)?
            \s*
              (?P<var>\w+)
            \s*
          (</em>|</i>)?
        \s+
      be
        \s+
          (?Pz<def>the((?!((
            (Let|let)
              \s+
                (<em>|<i>)?
                  \s*
                    \w+
                  \s*
                (</em>|</i>)?
              \s+
            be
              \s+
                .+
                  |\.\s|:))).)+)
                  """, helpers.stripHTML(open(filePath, 'r').
                     read()), re.DOTALL | re.VERBOSE))

    if matches: toPrint += f'------- FILE : {filePath[47:]}\n
       '
    for match in matches:
      # res = ''.join([ch if ch not in string.whitespace
         else ' ' for ch in match[2]])
      # toPrint += f'(Let) {match[1]} (be) {res}\n'
      thisLine = f"(Let) {match['var']} (be) {match['def']}"
      print(thisLine)
      toPrint += f'{thisLine}\n'
      dataFile.addLine([f"(Let) {match['var']} (be) {match['
         def']}",filePath])
    counter.increment()
  except Exception:
```

```
    pass

  dataFile.finish()
# print(toPrint)

if __name__ == '__main__':
  main()
```