Linking Linguistics and Programming: How to start? (Work in Progress)

J. E. Rice[1], I. Genee[2], and F. Naz[3]

[1] Dept. of Math and Computer Science, University of Lethbridge, Canada j.rice@uleth.ca
[2] Dept. of Modern Languages, University of Lethbridge, Canada inge.genee@uleth.ca
[3] Dept of Math and Computer Science, University of Lethbridge, Canada fariha.naz@uleth.ca

Abstract. Sociolinguistics is described as "the study of how language and social factors such as ethnicity, social class, age, gender, and educational level are related". Social factors may result in differences in language use, which may then be referred to as sociolinguistic differences. The goal of this project is to answer the following question: is it possible to identify sociolinguistic differences in the way people use programming languages? How might we go about answering our question? We propose that techniques from corpus linguistics may be of use. Computer-based techniques such as text mining allow the use of software in the identification of trends in language use that would otherwise require an enormous amount of human effort to discover. Once various sociolinguistic differences have been identified by computational means, traditional critical analysis methods can then be applied for further analysis.

1 Introduction

In 1982 Misek-Falkoff introduced a new area of research that she referred to as "Software Linguistics" [1]. She proposed treating computer programs as text, and described the use of "natural language textual criticism" as applied to software. To our knowledge this is the first work to suggest that the use of linguistics principles as applied to computer programs could provide useful information. She did not follow up on this work, however, and since then it appears that linguistic analysis has been mainly restricted to natural languages (with a few exceptions such as [2] and [3]).

Our work proposes to build upon Misek-Falkoff's original notion of software linguistics and make use of tools, ideas, and concepts from linguistics in application to texts produced using artificial languages: in other words, computer programs, or code. In the remainder of this paper we discuss the potential of such an approach, the areas where it might apply, and provide some initial thoughts from the work that we have begun.

2 Context

2.1 Natural Languages and Sociolinguistics

Wardhaugh [4] defines a society as "any group of people who are drawn together for a certain purpose or purposes", and a language as "what the members of a particular society speak". We could then say that the community of programmers is a society drawn together for the purposes of creating software. In this particular society we could examine either the natural language used in the course of carrying out software creation, or the artificial languages used in the actual software creation. Studies have already examined ways in which natural language is used in this society, e.g. [5, 6]. We propose to instead examine how people use the programming languages themselves.

In natural language one might break prescriptive syntax rules e.g. in order to emphasize a point, or phrase a sentence in a somewhat ambiguous or redundant way. These choices are not desirable when writing a computer program, and one would hope that the option to do so was removed as a possibility when designing the programming language. Yet there is still a great deal of variation possible, for instance in the choice of identifier names for functions/methods

and/or variables, or simply in the way the problem has been decomposed. It is this variation that we are interested in and which may provide information about the society of programmers.

The major insight offered by modern sociolinguistics is that social variability may influence or determine linguistic variability. The (independent) social variables most exhaustively studied in sociolinguistics are age, gender and socio-economic status (SES). The (dependent) linguistic variables include phonetic variables such as the pronunciation of the (ing) sequence at the end of words like "'working" and "playing"; morphosyntactic variables such as the use of double negatives, and stylistic variables such as the use of indirect requests vs. direct commands and forms of address. However another, complementary view is that linguistic variability may in turn influence social variability; thus the language itself shapes the social relations [7, 8].

One area where these ideas can be examined is that of gender differences, and in fact researchers have developed computer-based techniques that show evidence of gender differences in natural language use, e.g. [9–12]. However, rather than implying a one-way causality ("because one is a man/woman one must speak and/or write in a particular way") many researchers have shown how the language is shaping the society [13]. In other words, researchers examine how language use by individuals or groups contributes to the formation of those groups, or the treatment of those individuals within the community under examination.

2.2 Digital Humanities/Corpus Linguistics

In areas where large quantities of text are available analysis of those texts can be carried out by computers, rather than by humans. This allows much faster assessment, and in some cases the identification of trends that would not otherwise have been found. Discourse and linguistic analysis can both be aided by computer-based approaches. For this to happen text samples must be collected from some source and then stored in a format that allows the analysis to take place. The samples are then often referred to as a corpus. This type of approach has been followed by Argamon and Koppel [9–11] as well as others such as [12] and [14].

Having seen the benefits in the humanities we are now proposing to turn these techniques around to focus again on computer science and the societies within this field. This idea fits specifically into the digital humanities area of critical code studies, an area which examines the social and cultural implications of computer code, focusing particularly on gender, race, and class [15]. Indeed, one question being addressed by critical code studies is "How do issues of race, class, gender and sexuality emerge in the study of source code?" [16] We believe that as we attempt to address sociolinguistic categorization of source code we will be contributing knowledge towards this question, as well as furthering the body of knowledge surrounding how people use these types of languages.

2.3 Meaning to Society

Overall what does this project mean to society? Who cares if we can identify whether sociolinguistic differences exist in the use and creation of programming languages? The answer is that this research may have impact in a number of areas, from government to industry to private individuals. Identifying sociolinguistic differences can allow us to infer information about society, the groups, and/or the individuals using these languages – and here we are talking about programming languages. Programming languages could arguably be considered some of the most powerful languages on earth, given that the software that drives our daily lives is written using these languages. It is imperative that we learn more about their use and the groups of people using them; we want to know what this information can tell us in regards to the social and cultural implications of computer programs, and by extension, the societies that work with them.

Moreover, this research can contribute to the comprehension of computer programs; we may be able to identify markers, trends or beacons (e.g. [17]) that can enhance understanding

of programs, and patterns in their use. This information could then be utilised to teach students to write "good" programs, and also to use different approaches to teaching programming, approaches that might appeal to wider groups than are currently being reached.

3 Directions

The question is how to go about this. We are particularly interested in the work by Argamon et al. [9–11], partly because their approach was successfully applied to text samples in more than one (natural) language. Their text categorization approach, based on machine learning algorithms, achieved more than 80% accuracy in categorizing fiction and non-fiction samples according to author gender, and identified previously unknown patterns in male and female language use. Thus, could we not borrow these techniques for classification of programming language samples? In order to do so we must follow the components of text categorization [9]:

Document Representation: *choose a large set of text features which might be useful for categorizing a given text (typically words that are neither too common nor too rare) and represent each text as a vector consisting of values representing the frequency of each feature in the text.*

Dimension Reduction: *optionally, use various criteria for reducing the dimension of the vectors – typically by eliminating features which don't seem to be correlated with any category*

Learning Method: *use some machine learning method to construct one or more models of each category.*

Testing Protocol: *use some testing protocol to estimate the reliability of the system.*

Thus the first problem we must address is document representation.

3.1 Can we tag code with existing POS-taggers?

As described above, document representation consists of choosing a large set of features which might be useful for categorization. In Koppel et al.'s work, the features include 405 function words which appear at least once in the British National Corpus (BNC), and n-grams of parts-of-speech and punctuation marks. All 405 function words plus the 100 most common triples, 500 most common ordered pairs, and 76 parts of speech were used as features, for a total of 1081 features. The documents were then represented as a vector of length 1081, with each entry representing the number of appearances in the document for that particular feature. Thus we need a list of features applicable to code. Can we use an automated part-of-speech (POS) tagger on code samples to generate similar information? We've selected C++ as our programming language of choice, although this discussion could equally be applied to any programming language. In Figure 1 we've run a simple code sample through the CST's POS tagger[1]. NN means

```
main()
{
    int first_number = 1;
    int second_number = 3;
    int sum_of_numbers;
    sum_of_numbers=first_number + second_number;
    cout<<"Sum is = "<<sum_of_numbers<<endl;
}
```

(a)

```
main()/NN {/( int/NN first_number/NNP =/NNP 1/CD ;/:
int/NN second_number/NNP =/NNP 3/NNP ;/:
int/NN sum_of_numbers/NNS ;/:
sum_of_numbers=first_number/NN +/SYM second_number/NN ;/:
cout<<"Sum/NN is/VBZ =/NNP "/" <<sum_of_numbers<<endl/NNP ;/:
}/)
```

(b)

Fig. 1. (a) Untagged code sample. (b) Code sample with POS tagging from CST.

that the preceding token is a noun, NNP refers to a proper noun, NNS refers to a plural noun, CD refers to a cardinal number, VBZ refers to a verb (3rd person present) and SYM refers to a symbol. So we can see that our program is mostly nouns, including, strangely enough,

[1] Center for Sprogteknologi, University of Copenhagen, http://cst.dk/tools/index.php

the "=" sign and number 3. This tool also separated the text into segments (one per line) and tokens (separated by spaces), although these segments and tokens don't necessarily match with what a programmer might identify as relevant segments and/or tokens. A more complex sample generated labelings identifying adverbs and preposition/subordinating conjuctions. So how useful is this? One problem is that the tool does not ignore white space, so " = " with spaces around it is treated differently than the same symbol with no whitespace surrounding it. Another problem is the treatment of data inside quotes, which was broken up by the tagger, as was the "!=" symbol, so the tagger is not tokenizing the code correctly. This could easily be corrected, however, given that these programs are written in language designed to be parsed by a computer! A possible larger problem is that "`int`" is labeled as a noun, when maybe it should be an adjective given that it is being used to describe what type of variable or function is given following, and it is unclear why "`val`" and "`int(val[i])-int(0)`" were labeled as adverbs given the apparent lack of nearby verbs.

Although this is an interesting exercise, it is evident that a straightforward (brute-force) use of automated unmodified POS taggers on code is unlikely to be a useful way to go about tagging the constituent parts of a program for classification or analysis, given that little to no thought was put into how the "words" in a program are behaving.

3.2 Beginning the feature list

So we have ruled out automatically tagging code with POS-labels. In fact, we may want to consider also that the POS labels themselves may not apply at all. Since the elements of a programming language are already categorized, we can begin with a list of the elements to be found in a C++ program[2]. A C++ parser within the compiler will recognize these types of tokens: identifiers, keywords, literals, operators, punctuators, and other separators. An identifier is a sequence of characters (word?[3]) that denotes a "thing"; this might be an object, variable, class, member of a class, or macro. Keywords are predefined identifiers that have special meanings which cannot be redefined. Punctuators are used to separate values or other tokens, while an operator appears as part of an expression, which is a sequence of operators and operands that appears for the purpose of computing a value, designating an object or function, or modifying the value of an object (this might take place as a side effect of computing a value). Operators include mathematical and logical operators that we might already be familiar with (e.g. `+ - * / < >`), combinations and repetitions of these (e.g. `+= -= *= /= ++ --`), and others such as array subscripts, scope resolution operators and class/pointer operators that resolve which part of a class variable (object) is being manipulated. Literals consist of invariant program data. These may be numeric values, characters or string constants, and may appear as 1 (numeric literal), `'c'` or `'\0'` (both are character literals), or "Hello!" (string literal).

3.3 Text Categorization

We have begun some preliminary work to test whether it is possible to use some of these items to represent a document, and then use a learning method to construct a model based on these items. We've created a program that identifies some very basic items: two data types (int, float); main functions, cout statements, and `{}` symbols. We then generated, for 10 sample programs, the metrics indicating how many times each of these items appeared in sample and produced a representation of each sample based on these metrics. We were then able to train, using a

[2] The following information can be found in any C++ reference. We used the descriptions and information in the C++ language reference available from the Microsoft Developer's Network (http://msdn.microsoft.com/en-us/library/3bstk3k5.aspx).

[3] What is a "word", in a program, and what information is that "word" conveying? [3] explored this discussion somewhat, deciding that "Together keywords, operators, and identifiers constitute the set of tokens which may be considered *words* in a programming language vocabulary."

support vector machine (SVM) implementation as described in [18], our system to identify categories within these samples. This was a little surprising to us given that we had so few samples to train on and that we were using such basic items; however it was a very promising result as a proof-of-concept.

3.4 Moving on

Our next steps are to add more features to use in categorization, and gather additional samples along with corresponding sociological data. Our current set of samples do not provide sufficient data for e.g. categorization according to author gender or first language learned (natural), although we can examine variables such as years of experience and possibly first language learned (artificial). Concurrently with this we must expand our feature list to incorporate a more complete list of the known C++ program elements.

3.5 Other Considerations

We had originally intended to determine a "code to POS mapping" and generate a feature set based on this, but as discussed above it seems reasonable to start with the known elements of our artificial language(s) and attempt text categorization with these. However it is still interesting to examine these elements and consider how they might map to what we call "parts of speech". For instance, literals are in most cases the equivalent of nouns, or more generally, "things" (rather than actions). Rather than attempt to find direct mappings to parts of speech, it might be more useful to generally categorize each token into "thing" words, "modifying" words, "action" words and so on. Identifiers are tricky, as they refer to anything a programmer needs to label with a name, including memory storage and functions. Depending on the context these could be "things" or "actions". If we were to use this type of labeling then the standard libraries would provide many identifiers that could likely be pre-labeled (e.g. cout). Finally, there are several keywords, reserved for use by the C++ language. These are quite varied. For instance, the keyword `false` is a logical literal (and so, likely to be considered a "thing") while the keyword `return` forms part of a return statement indicating the value a function will return when called, e.g. `return false;` – the behaviour of this keyword then is more in the nature of an action.

This approach could be valuable if we are to try to link our findings to those in the literature for natural language text categorization. For example [9] states that "[t]he picture that emerges is that the male indicators are largely noun specifiers (determiners, numbers, modifiers) while the female indicators are mostly negation, pronouns and certain prepositions". If we can categorize code samples according to gender author, might we see similar patterns?

4 Conclusion

In this paper we have explored the need for a sociolinguistic analysis of computer programs. We've pointed out that software is such a huge part of our daily lives that it really makes sense to understand how different groups might make differing uses of the artificial languages that are used to create programs. However, this isn't easily done. While we can leverage the approaches used in application to natural languages, the tools will likely need to be at the very least modified, if not developed from the ground up. To provide some guidance and structure to our problem we are leveraging approaches known to work in natural languages, and that have the potential to be altered (or trained) for additional languages. However identifying useful and meaningful features to work with and understanding fully how to apply these approaches in this new setting are large projects which we have only just begun researching. This paper offers both a methodology to follow in this investigation, plus an initial and tentative set of features that we hope to refine and then use in training learning algorithms in categorization of code samples according to sociological variables.

References

1. L. D. Misek-Falkoff. The new field of "software linguistics": An early-bird view. *SIGMETRICS Performance Evaluation Review*, 11(2):35–51, August 1982.
2. Abram Hindle, Earl Barr, Mark Gabel, Zhendong Su, and Prem Devanbu. On the naturalness of software. In *Proceedings of the 2012 International Conference on Software Engineering (ICSE)*, pages 837–847. IEEE Press Piscataway, NJ, USA, 2012.
3. D. P. Delorey, C. D. Knutson, and M. Davies. Mining programming language vocabularies from source code. In *21st Annual Psychology of Programming Interest Group Conference - PPIG*, 2009.
4. R. Wardhaugh. *An Introduction to Sociolinguistics*. Blackwell Publishers, Oxford, UK, 6th edition, 2010.
5. F. Taïani, P. Grace, G. Coulson, and G. Blair. Past and future of reflective middleware: Towards a corpus-based impact analysis. In *Proceedings of the 7th Workshop on Reflective and Adaptive Middleware (ARM)*, pages 41–46. ACM, 2008.
6. Nicolas Bettenburg and Ahmed E. Hassan. Studying the impact of social structures on software quality. In *Proceedings of the 17th IEEE International Conference on Program Comprehension (ICPC '10)*, pages 124–133. IEEE, 2010.
7. W. Labov. *Principles of Linguistic Change, Cognitive and Cultural Factors*. Language in Society. Wiley, 2011.
8. J.K. Chambers. *Sociolinguistic Theory: Linguistic Variation and Its Social Significance*. Language in Society. John Wiley & Sons, 2003.
9. M. Koppel, S. Argamon, and A. Shimoni. Automatically categorizing written texts by author gender. *Literary and Linguistic Computing*, 17(4):401–412, 2002.
10. Shlomo Argamon, Moshe Koppel, Jonathan Fine, and Anat Rachel Shimoni. Gender, genre, and writing style in formal written texts. *TEXT*, 23:321–346, 2003.
11. Shlomo Argamon, Jean-Baptiste Goulain, Russell Horton, and Mark Olsen. Vive la différence! text mining gender difference in french literature. *Digital Humanities Quarterly*, 3(2), 2009.
12. M. L. Newman, C. J. Groom, L. D. Handelman, and J. W. Pennebaker. Gender differences in language use: An analysis of 14,000 text samples. *Discourse Processes*, 45:211–236, 2008.
13. Mary M. Talbott. *Language and Gender*. Polity Press, 1998.
14. Erik Linstead, Lindsey Huges, Cristina Lopes, and Pierre Baldi. Exploring Java software vocabulary: A search and mining perspective. In *Proceedings of the ICSE Workshop on Search-Driven Development–Users, Infrastructure, Tools and Evaluation (SUITE)*, Vancouver, BC, 16-19 May, 2009.
15. Mark C. Marino. Critical code studies (entry from the blog electronic book review), 2006. http://www.electronicbookreview.com/thread/electropoetics/codology.
16. HASTAC (Humanities, Arts, Science and Technology Alliance and Collaboratory) Scholars program. Critical code studies (blog entry), 2011. https://www.hastac.org/forums/hastac-scholars-discussions/critical-code-studies.
17. J. F. Pane and B. A. Myers. Usability issues in the design of novice programming systems. Technical Report CMU-CS-96-132, Carnegie Mellon University, Pittsburgh, PA, aug 1996.
18. Christopher J.C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2:121–167, 1998.