Statistical Unigram Analysis for Source Code Repository

Weifeng Xu
Department of Computer Science
Bowie State University
Bowie, MD, USA
wxu@bowiestate.edu

Dianxiang Xu
Department of Computer Science
Boise State University
Boise, ID, USA
dianxiangxu@boisestate.edu

Omar El Ariss
Department of Computer Science
The Pennsylvania State University-Harrisburg
Middletown, PA, USA
oue1@psu.edu

Yunkai Liu
Department of Computer & Information Science
Gannon University
Erie, PA, USA
liu006@gannon.edu

Abdulrahman Alatawi
Department of Computer Science
Bowie State University
Bowie, MD, USA
Alatawi1116@students.bowiestate.edu

***Abstract*—Unigram is a fundamental element of n-gram in natural language processing. However, unigrams collected from a natural language corpus are unsuitable for solving problems in the domain of computer programming languages. In this paper, we analyze the properties of unigrams collected from an ultra-large source code repository. Specifically, we have collected 1.01 billion unigrams from 0.7 million open source projects hosted at GitHub.com. By analyzing these unigrams, we have discovered statistical patterns regarding (1) how developers name variables, methods, and classes, and (2) how developers choose abbreviations. Our study describes a probabilistic model for solving a well-known problem in source code analysis: how to expand a given abbreviation to its original indented word. It shows that the unigrams collected from source code repositories are essential resources to solving the domain specific problems.**

Keywords—programming language; source code; n-gram; unigram; abbreviations; ultra-large-scale analysis

I. Introduction

Natural languages and computer programming languages are both used to communicate and solve real-world problems. In the domain of software engineering, solutions to real-world problems specified in requirements documentation are often expressed in a natural language. However, natural languages cannot be used for implementing the software solutions due to the ambiguity, either semantically or syntactically [1] [2]. The implementation of such solutions uses the source code, which can be written in various computer programming languages. Programming languages are formally constructed languages designed to communicate with a specific machine. For example, the following Python and Java source code describe a solution to printing guests' names from a list of invited people.

```
# Python
class MyDemo:
    # print list of invited people
    def display(self, people):
        for guest_name in people :
            print(guest_name)
//Java
public class Demo {
    // print the names of the guests
    // from a list of invited people
    void printGuest(ArraryList<String> ipl){
        for(string gstName :  ipl){
            println(gstName)
    }}
```

Nevertheless, from a linguistic perspective and based on Chomsky's hierarchy for languages [3], programming languages are similar to natural languages since most of the grammatical features of both types of languages can be represented using a context-free grammar. In addition, developers use the vocabularies of a natural language, i.e., English words, including `print`, `guest`, `invited`, and `name`, to name identifiers, such as variables, methods, and classes. The terms variable, method, and class are formally defined constructs in objected-oriented languages. Reusing English words in source code ensures the readability of source code because (1) structured programming [4] requires meaningful names for identifiers (2) English words are meaningful for communication in the real-world (3) English names persevere the same or similar meanings in both real-world and the domain of software engineering. In the previous sample code, developers use the English words `guest` and `display` to name a variable and a method for displaying information.

In practice, naming program constructs is more complicated than simply picking up English words: (1) The names of the constructs are often mixtures of a single English word, multiple English words, and abbreviations. For example, the variable name `gstName` consists of one abbreviation `gst` and one English word `name` to represent the name of a guest. (2) The same English word used in the domain of natural language may represent different meaning in programming languages, for example, the word `class`. The difference impacts the understandability of source code. Regardless of the

978-1-5090-6549-3/17 $31.00 © 2017 IEEE
DOI 10.1109/BigMM.2017.13

IEEE computer society

syntactical and semantical complexity of the constructs' names, its elements mainly consist of one or more unigrams. Each unigram is either an English word or an English abbreviation. Therefore, we are interested in investigating the properties of unigrams in computer programs, aiming to understand: (1) how developers name variables, methods and classes and (2) how developers choose abbreviations.

We conduct our study using the variable, method and class names extracted from the entire source code repository hosted on GitHub.com [5] during the year of 2015. The GitHub repository contains 699,331 open source projects [6]. Each project can be written in different programming languages. The contributions of the paper are as follows:

(1) This is the first attempt to analyze the properties of unigrams in computer programs at such an ultra-large scale.

(2) The unigrams collected from the source code repository in this study can solve such domain specific problem, as expanding name abbreviations using unigram models.

(3) The entire corpus, including unigrams, abbreviations, and results, are available online. It provides a useful benchmark for future research.

The rest of this paper is organized as follows: Section II reviews the related work. Section III describes the process of unigram collection from GitHub. Section IV analyzes the properties of unigrams. Section V presents the empirical study. Section VI concludes this paper.

II. RELATED WORK

An n-gram is a contiguous sequence of *n* items from a given sequence of text. N-grams, particularly unigrams and bigrams, collected from texts are extensively used in text mining and natural language processing, including machine translation, speech recognition, spelling correction, etc. Linguistics Data Consortium has published the n-gram data in 2006, including 16 million of unigrams and 315 bigrams collected from one Terabyte web collection [7]. The Google N-gram Viewer [8] is an online tool that charts frequencies of given unigrams or bigrams found in printed sources between the years 1500 and 2008.

Although n-gram analysis has been introduced in other domains, n-gram analysis has not been well-studied in the context of programming languages and specifically for the analysis of the naming conventions of identifiers. Therefore, there is very little research done that is closely related to our proposed work. Allamanis and Sutton [9] use a trigram language model based on a corpus of 14,807 Java programs to come up with various metrics including code complexity and variable originality. On the other hand, the authors use n-grams on single projects to suggest proper coding conventions such as formatting and identifier naming [10]. Allamanis et al. [11] use a neural logbilinear model to improve existing code by suggesting names for methods and classes. Finally, Raychev et al. [12] introduce a statistical model based on conditional random fields (CRFs) to predict the types of variable names and the names of variables in obfuscated JavaScript code.

III. UNIGRAM COLLECTION

Variables, methods, and classes are essential to construct types of source code. The names of these constructs are categorized into two groups shown below:

- Unigram name: a name of construct that consists of a single unigram, which is either an English word or an abbreviation of an English word. They are referred to as unigram English names and unigram abbreviation names, respectively. For example, a single English word `display` can be used for naming a method; the abbreviation `demo` is used to name a class (see Table 1). Thus, the word `display` is a unigram English name and `demo` is a unigram abbreviation name.

- Multigram name: a name of a construct that consists of multiple unigrams. For example, `gstName` is a multigram name, which consists of two unigrams `gst` and `name`. The unigram `gst` is the abbreviation of the English word `guest`. Other examples are shown in Table 1.

Table 1. Categories and examples of variable, method, and class names

	Variable	Method	Class
Unigram name	guest, people, gst	display	demo
Multigram name	guest_name, gstName, ipl	printGuest	myDemo

We use unigrams collected from source code repository, i.e., GitHub, to analyze patterns of unigram names and multigram names. Figure 1 shows the process of unigram collection and analysis.

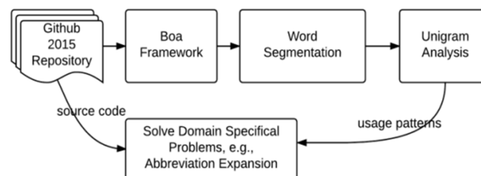

Figure 1. Block diagram of unigram collection and analysis

The process mainly consists of three components:

- Boa Framework [13]. It is a language and infrastructure for extracting syntactic information from source code in GitHub, including variable, method, and class names. The framework converts source code to AST trees, and then a visitor traverses the trees to collect names.

- Word segmentation [14]. This phase determines where the word boundaries are for a given multigram name. Processing natural language, such as English, doesn't normally need to perform word segmentation because words in English sentences are most of the time separated by white spaces. However, a construct's name often is a multigram name without spaces. For example variable names such as `username` and `user_name` require segmentation.

- Unigram analysis. This phase utilizes Apache Hadoop framework [15] to study the properties of unigrams collected from source code. The properties will be used to solve domain specific problems, e.g., expanding abbreviations used in source code, which will be covered in section V.

Table 2 shows: (1) 0.70 billion constructs are extracted from nearly 0.7 million projects. (2) Over one billion unigrams are collected from the constructs.

Table 2. The total number of constructs, unigrams extracted from GitHub (in millions)

	Variable	Method	Class	Total
The number of constructs	419	161	21	701
The number of unigrams extracted from constructs	636	396	57	1069

IV. ANALYSIS OF UNIGRAMS

Unigrams collected from the GitHub source code repository are different from unigrams collected from natural language corpus, i.e., news archives. In this section, we compute the most commonly used English words and abbreviations in the source code, and then we reveal two important statistical properties of abbreviations.

A. Most Commonly Used English Words in Source Code

We have extracted the most commonly used English words from identifier names: variable, method, and class names, respectively. Table 3 shows the top 70 most frequently used English words for naming variables from 636 million unigrams extracted from variable names.

Table 3. The top 60 most frequently used English words for naming *variables*

Word	Freq. (k)	Word	Freq. (k)	Word	Freq. (k)	Word	Freq. (k)
name	9865	node	2306	class	1572	response	1333
id	7920	count	2301	parent	1560	code	1320
value	7354	item	2255	element	1550	line	1306
type	5847	field	2239	length	1549	action	1292
key	4235	message	2238	source	1534	height	1286
result	3983	to	2228	string	1533	instance	1282
data	3681	view	2206	input	1515	current	1282
index	3680	start	2179	default	1510	log	1259
context	3399	state	2078	target	1501	test	1249
file	3155	event	1941	max	1482	number	1227
list	3115	time	1935	service	1446	listener	1220
in	3055	map	1909	offset	1410	column	1218
text	2590	out	1864	end	1393	content	1210
new	2530	request	1758	width	1387	label	1204
size	2502	user	1643	date	1382	last	1191
is	2390	info	1640	tag	1371	buffer	1175
path	2334	object	1613	serial versionid	1342	error	1009

Table 4 shows that the same English words are ranked differently in GitHub and natural language corpus. For example, the English word *name* is ranked the first in GitHub and the 108th in the natural language corpus [16], respectively. Similarly, we have computed the top 100 most frequently used English words for naming methods and classes (see Appendix).

Table 4. Different word frequency ranks in source code (SC) and natural language corpus (NC)

Word	Ranking		Word	Ranking		Word	Ranking	
	SC	NC		SC	NC		SC	NC
name	1	108	node	18	3407	class	35	388
id	2	654	count	19	2011	parent	36	1771
value	3	1146	item	20	214	element	37	2222
type	4	253	field	21	574	length	38	1155
key	5	569	message	22	149	source	39	419
result	6	611	to	23	4	string	40	1567
data	7	131	view	24	79	input	41	1438
index	8	276	start	25	474	default	42	1538
context	9	2022	state	26	111	target	43	1584
file	10	281	event	27	624	max	44	1428
list	11	107	time	28	50	service	45	97
in	12	6	map	29	197	offset	46	5179
text	13	349	out	30	60	end	47	317
new	14	27	request	31	627	width	48	3060
size	15	337	user	32	185	date	49	102
is	16	8	info	33	160	tag	50	2454
path	17	1790	object	34	1150	serial versionid	51	-

Without surprise, we have observed the following:

(1) Over 95% of the English words for naming variables and classes are nouns.

(2) 53% of English words are used for naming both variables and classes.

(3) Among all the top 100 most frequently English words used for naming variables, 42% of them is used for naming methods, 34% of them are verbs, and 12% are prepositions. It is consistent with the purpose of the method construct in object-oriented programming: the manipulation of variables.

B. Patterns of Choosing Abbreviations in Source Code

In this subsection, we will observe patterns between abbreviations and their original intended English words from small examples, develop an algorithm to extract all possible abbreviation and English word pairs from the ultra-large-scale source code repository. Table 5 shows a small set of applications used to observe abbreviation patterns. They were selected because of their relative popularity, diversity in terms of development maturity and application domain, and availability publicly in open source software repositories.

Table 5. Subject programs used in our experiments to observe patterns [17]

Program	Version	KLOC	Description
JasperReports	2.0.4	34.04	Dynamic content
JFreeChart	1.0.19	57.83	Data rep.
SoapUI	2.0.1	30.48	Web service
Freecol	0.7.3	27.21	Game
GanttProject	2.7	28.30	Scheduling
Junit	4.4	0.948	Software dev.
Avuze	5.5.0.0	163.53	Online file share
Hibernate	2.1.8	21.49	Database
JEdit	4.2	32.60	Text editor
DataCraw	3.4.5	20.20	Data management
Xholon	0.7	23.39	Simulation
Jsch	0.1.51	7.39	Security
Domination	1.0.9.7	8.32	Game
JMencode	0.64	1.33	Video encoding

We review the source code and have observed the following patterns (shown in Table 6) between abbreviations and their original intended English words:

- **Pattern one:** There are mainly two types of strategies for choosing abbreviations. (a) Consecutive Characters Strategy, which uses the first *n* consecutive characters as the abbreviation for a given English word and (b) Nonconsecutive Characters Strategy, which uses *n* nonconsecutive characters as the abbreviation. The two different strategies produce two different abbreviation types: Consecutive Characters Abbreviation (CCA) and Nonconsecutive Characters Abbreviation (NCA).
- **Pattern two:** The first letter matters. Regardless of the different type of abbreviation choosing strategies, the first letter of the abbreviation is always the first letter of its original intended English word.
- **Pattern three:** The order of the characters in abbreviations and its intended English word is consistent. Regardless of the different types of abbreviations, developers choose the characters from left to right from the intended English word as its abbreviation. It is evident for CCA because it uses n-consecutive characters from the original intended English word as its abbreviation. For example, using `str` as the abbreviation for the word `string`. For NCA, the abbreviation `src` is picked up from the word `source` at positions 0, 3, and 4.
- **Pattern four:** The majority of abbreviations use less than four characters to represent unigram variable names. We will verify the pattern in section C once we extract all abbreviations and their intended words.

Table 6. Observed patterns between abbreviations and their original intended English words

Abbreviation Type	n = 1		n = 2	
	Name	**Abbr.**	**Name**	**Abbr.**
n-Consecutive Characters as Abbreviation (CCA)	**n**ode	n	**ex**ception	ex
	value	v	**ev**ent	ev
	list	l	**it**erator	it
	handler	h	**ex**tent	ex
n-Nonconsecutive Characters as Abbreviation (NCA)			**m**a**p**	mp
			bu**t**ton	bt
			loa**d**	ld
			li**s**t	ls
Abbreviation Type	**n = 3**	**n = 4**	**n = 3**	**n = 4**
	Name	**Abbr.**	**Name**	**Abbr.**
n-Consecutive Characters as Abbreviation (CCA)	**str**ing	str	**str**ing	str
	buffer	buf	**buf**fer	buf
	object	obj	**obj**ect	obj
	array	arr	**arr**ay	arr
n-Nonconsecutive Characters as Abbreviation (NCA)	**s**ou**rc**e	src	**s**ou**rc**e	src
	even**t**	evt	**ev**en**t**	evt
	me**s**sa**g**e	msg	**m**e**s**sa**g**e	msg
	bu**t**to**n**	btn	**b**u**t**to**n**	btn

C. Algorithms for Extracting Abbreviations

Once we have discovered these abbreviations patterns, we are interested in developing algorithms to extract all abbreviations along with their intended words from GitHub. Because nearly 60% of unigrams are used for naming variables (see Table 2), we would like to extract abbreviations for naming variables. The steps to generate such an abbreviation list are described as follows:

1. Extract all pairs `(variable name, the type of the variable name)` from GitHub. They are candidates of pairs for extracting an English word and its abbreviation. The basic assumption to compute abbreviations is: when naming a variable, developers more likely to choose an abbreviation based on the type of the variable. The type can be a class, an interface, a primitive, or an array. For example, we may use the abbreviations `n`, `it`, and `i`, to represent the instance of `Node` class, `Iterator` interface, and `integer` primitive, respectively.
2. Segment the variable name and variable name type in each pair. The step produces two unigram sets. Each set contains one or more unigrams separated from the variable name and the variable names type, respectively.
3. Find the abbreviations. Pick *w* and *a* from the two unigram sets, respectively, and then count the frequency of the pair `(w,a)` if `isAbbre(w, a)` returns `true`. Note that, based on our experience, if `a` is an abbreviation of `w`, `a` is a set. For example, the letter `n` can be the abbreviation of classes `Node` and `Number`. On the other hand, a type can be represented by multiple abbreviations. For example, the class `Node` can be represented by `n` and `nd`. It depends on individual developer's preference. The function `isAbbre` checks the patterns one and four, and then calls a function `isConsistent`, which implements the first three patterns recursively.

```
def isAbbre(w, a):
  if len(w)<= len(a):
      return False
  if len(a)<1:
      return False
  if len(a)==1 and len(w)>1 :
      return w[0] == a[0]
  else:
      return w[0] == a[0] and \
      isConsistent (w[1:], a[1:])

def isConsistent(w, a):
  if (len(a)==0):
      return True
  if len(w)>0:
      if w[0]==a[0] :
          return isConsistent (w[1:], a[1:])
      elif (w[0]!=a[0]):
          return isConsistent (w[1:], a)
  return False
```

D. Top Abbreviation Names

We have extracted 62 million abbreviations from 419 million variable names. It indicates nearly 15% of variable names use abbreviations. Table 7 shows the top 100 most frequently used abbreviations for naming variables. The results are strongly consistent with developers' naming behaviors. For example, as a developer, we often use `i` and `e` for representing `int` and `exception`, respectively. Specifically, we have made the following conclusions based on the extracted abbreviations from the GitHub repository:

- Only five abbreviations have length greater than four characters among the top 100 most frequently used abbreviations for naming variables. The conclusion is consistent with our observation patterns four.

- Abbreviations are widely used for representing the compressed information. Nearly 15% of variable names use abbreviations.

- Some variable names, such as `ioexception`, `stringbuffer`, and `bytebuffer` are considered as unigrams in [16] because they have relatively high frequencies in natural language corpus and contain no space. For example, `ioexception` is ranked the 35661th with a frequency of 595675 in [16].

- Developers may choose different abbreviations for a given word. For instance, for the most frequently used unigram `int`, developers may pick `i` or `in`. However, developers are more likely to use `i` as abbreviation because the pair `(int, i)` has a higher frequency than `(int, in)`.

- The same abbreviation can be used for representing different words, e.g., `in` is used for presenting either `int` or `input`. For a given abbreviation `in`, it is more likely to represent `input` if we consider the frequency as the only criteria to determine abbreviations.

Table 7. The top 100 most frequently used *abbreviations* for naming *variables*

Abb.	Word	Freq. (k)	Abb.	Word	Freq. (k)	Abb.	Word	Freq. (k)	Abb.	Word.	Freq. (k)
i	int	9878	f	float	341	m	map	129	rect	rectangle	91
e	exception	5584	ctx	context	341	e	element	127	it	int	87
s	string	1575	conn	connection	322	cal	calendar	123	caps	capabilities	87
str	string	1069	rs	result	308	m	method	120	mgr	manager	86
l	long	1061	f	file	297	l	list	120	f	field	85
obj	object	1037	msg	message	294	re	recognition	119	cmd	command	85
ex	exception	937	b	boolean	273	q	query	111	cls	class	81
o	object	781	ioe	ioexception	263	c	collection	108	img	image	80
e	event	561	sb	stringbuffer	253	e	encoding	108	s	session	79
it	iterator	535	i	iterator	245	t	thread	106	params	parameters	78
in	input	522	n	node	233	btn	button	104	br	bufferedreader	77
v	view	522	db	database	233	elem	element	104	comp	component	77
out	output	492	attrs	attributeset	217	c	cursor	103	h	handler	77
evt	event	487	props	properties	210	l	listener	103	e	entity	77
in	int	470	fs	filesystem	205	m	manager	102	m	message	76
log	logger	437	prot	protocol	199	con	connection	101	env	environment	76
doc	document	436	p	point	193	attr	attribute	100	m	matcher	76
conf	configuration	423	config	configuration	191	app	application	99	loc	location	75
g	graphics	420	ch	char	176	prefs	preferences	97	cert	certificate	74
b	byte	414	e	entry	171	buffer	bytebuffer	96	v	visitor	74
t	throwable	401	stmt	statement	153	t	type	92	c	context	73
c	char	399	req	request	152	c	component	92	p	player	73
sb	stringbuilder	377	c	class	150	e	enumeration	92	addr	address	73
d	double	372	v	vector	144	e	error	91	a	array	72
iter	iterator	346	ref	reference	131	sql	sqlexception	91	nfe	numberformatexception	71

E. Statistical Properties of Abbreviations

When developers decide to use an abbreviation to represent its original intended word, they need to make two decisions: (1) determine which type of abbreviation they want to choose, either CCA or NCA (2) determine how much effort they want to save if abbreviations are used compared to original intended English words. Two statistical properties of abbreviations need to be studied to understand the decision process:

- The percentage of CCA versus NCA is used to help us understand how likely developers are to choose CCA or NCA. The percentage of CCA and NCA is the probability of developers to choose CCA and NCA for a given word `w`. Formally, we compute `P(E)`, where `E` is an event of choosing an abbreviation for `w`.

- The Typing Effort Saving (TES). TES computes how much effort developers can save if abbreviations are used. Formally, the TES value for a given pair `(a, w)` is defined as:

$$TES(a,w) = 1 - \frac{lenth(a)}{length(w)} \quad (1)$$

For example, `TES(i, int)=66.67%`, and `TES(in, int)=33.33%`. The range of the *TES* is between 0 and 1, excluding 0 and 1. Zero means no abbreviation is used.

Table 8 shows the total number of 62253k abbreviations used for naming variable names. Nearly 84% of the abbreviations are CCA, i.e., `P(E=CCA)=84%`. The number of unique abbreviations is 103k. Over 62% of these unique abbreviations are CCA. Figure 2 shows the frequency distribution of TES. The x-axis represents the ratio of TES. The y-axis is the frequency of TES measured in percentage. For example, let *X* be the event of computing `TES(a,w)`, then the frequency of `TES(i,int)` is

P(X=TES(i,int))=P(X=66.67%)= 19%. Overall, this figure indicates that 75% of the abbreviations have a *TES* between 50% and 89%. It implies that developers often use no more than the half size of the English word as abbreviations.

Table 8. The total number abbreviations extracted from variables

Abbreviation Type	Unique Abb. (k)		Total Abb. (k)	
	#	%	#	%
n-Consecutive Characters Abbreviation (CCA)	64	62.17	52234	83.91
n-Nonconsecutive Characters Abbreviation (NCA)	39	37.83	10019	16.09
Total	103		62253	

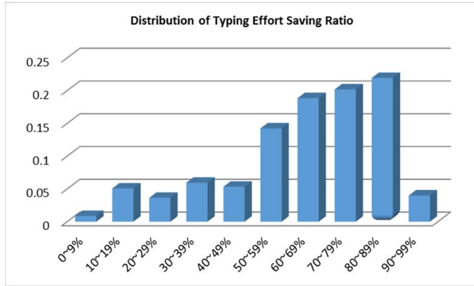

Figure 2. Distribution of TES

It is worth noting that one can argue that abbreviation convention [18] [19]is another reason to naming abbreviation, e.g., the string `re` should be used as the abbreviation of the word `result` due to the convention. However, there are several reason not to consider convention during programming:

(1) Unless there is a wide-accepted list of standard abbreviations in source code, such as `int` and `integer`, it is challenge for developers to think of abbreviation convention during development. For the `re` example, other people may use the string `res` as the abbreviation for `result`, as the string `res` has more readability than `re`. Others may also argue the string `res` can only be used for representing word `response`.

(2) The four patterns have already formed the foundation for abbreviation convention.

(3) The abbreviation ranking in Table 7 can be used as a reference for a standard abbreviation convention.

F. Size Distribution of Multigram Names

Besides unigrams, developers often use multigrams to name identifiers (constructs) to improve the readability of their code. The size of a multigram is the number of the unigrams that the multigram consists of. The size distribution of multigrams indicates how likely developers name constructs with various sizes.

To compute the size of a multigram, we first utilize a word segmentation algorithm to break the multigram into segments, and then we simply count the number of the segments. Figure 3 shows the size distribution of variable names, method names, and class names. The x-axis is the size of a name. The y-axis is the frequency of the name size in percentage. The size distribution is generated using the 701 million names that are shown in Table 1. The figure shows:

- The most likely choice, i.e., with the probability of 65%, developers use a unigram to name a variable.
- 34% of variable names are multigrams.
- The majority, i.e., 69%, of multigram variable names has the size of two.
- When naming methods and classes, developers often use, i.e., with the probability of over 80%, multigrams.
- There is no significant difference regarding size when naming methods and classes.

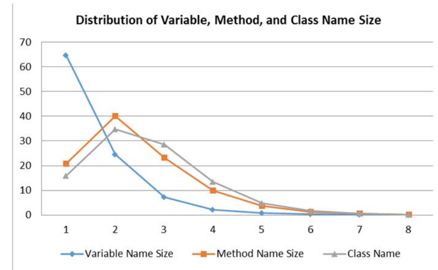

Figure 3. Size distribution of variable, method, and class names

V. Empirical Study: Abbreviation Expansion Using Unigrams

Although the use of abbreviations can help developers to implement faster, it may create confusion in the source code and therefore a decrease in program readability. For example, the variable abbreviation `v` in Table 7 can be interpreted as `view` (522k), `vector` (144 k), or `visitor` (74k). Different approaches [20] have been proposed to expand abbreviations, and however, these approaches have not considered the properties of program languages in the context of linguistics. In this empirical study, we demonstrate that utilizing the properties of unigrams collected from the ultra-large scale source code repository can solve domain specific problem, such as expanding the abbreviations used in a source code.

Table 9. Example candidate words for abbreviation *re* and their frequencies

Candidate	Frequency in unigram			
	SC		NC	
	#	%	#	%
result	3,983,282	0.626	127,425,045	0.022
request	1,758,393	0.277	124,620,318	0.021
response	1,333,290	0.210	84,065,293	0.014
resource	822,922	0.129	99,964,083	0.017
read	443,324	0.070	322,331,766	0.055
repository	211,294	0.033	8,892,664	0.002
rule	531,175	0.084	97,658,641	0.017
range	374,764	0.059	128,314,924	0.022

A. A Probabilistic Language Model to Expand Abbreviations

Assume that we have a task that needs to figure out what the abbreviation `re` stands for. Based on the patterns of English words and their abbreviations, Table 9 lists eight examples of possible words that match the abbreviation

computed from the source code and natural language corpus [16]. These words are referred to as the candidates of the abbreviation. Table 9 also includes the candidates' corresponding frequencies in two sets of unigrams, respectively. Note that the total number of unigrams in the source code and natural language corpus are 636 million and 588,118 million, respectively.

If only considering the frequency, we choose the English words `result` and `read` as the intended words in the domain of programming and the natural languages, respectively. They have the highest probability regarding frequency. However, these results lack compelling evidence. We use the Naive Bayes probabilistic model instead to solve the uncertain problem. Formally, the problem can be described as follows: given an abbreviation, `a`, determine what word `c` was the most likely original word. For example, if `a` is `re`, then `request` is `c`, which is the most likely word in the domain of programming languages. The language model for choosing the best candidate among all candidates is shown as follows:

$$argmax_c\, P(c \mid a) = argmx_c\, P(a \mid c) * P(c) \qquad (2)$$

where $P(c)$ is the probability of c, the candidate English word in the source code unigrams.

$argmax_c\, P(c \mid a)$: the highest $P(c \mid a)$.

`P(a | c)` is the probability that a developer will use the abbreviation `a` to represent `c`. It is called the abbreviation representation model. The representation model is based on the frequency distribution of TES ratio that is shown in Figure 2, and the distribution of abbreviation types (which we will discuss in details in subsection C). Note that experts may disagree with the abbreviation representation model. Therefore there is no complete model. We only use the model to demonstrate the importance of the unigrams that are generated from source code. In other words, the properties of unigrams can help us to solve domain specific problems.

B. Search Candidates for Language Model

It is unwise to pick up all English words in the unigrams that match the abbreviation patterns as candidates because there are too many possible matches. For example, the candidates for the given abbreviation `re` will include all the English words that start with `r` and contain `e` (consecutively or nonconsecutively). In fact, it makes more senses to pick up candidates based on the locations of the given abbreviation in the source code. There are two approaches to search candidates: static and dynamic approaches. Static approach searches for candidates within a certain radius of the abbreviation, e.g., the method or class in which the abbreviation is used. Dynamic approach generates the data flow diagrams or control flow diagrams from source code first and then searches for the candidates within the radius of the abbreviations in these diagrams. Once the candidates search radius is determined for the given abbreviation `a`, we pick each English word `c` in the radius, using `isAbbre(c, a)` to test if the `c` is candidate. For example, assume we have chosen the static approach where the method is the candidate search radius for the given abbreviation `re`. Table 10 contains ten candidate unigrams for the abbreviation `re` in a method. The first eight unigrams are CCA and the last two are NCA.

C. Compute the Best Candidates as the Expanded Word

Our goal is to compute $argmax_c P(c|a)$. The challenge of computing $argmax_c P(c|a)$ is to compute `P(a|c)` basedon formula 2. We could simply use the frequency of pair `(a,c)` to compute `P(a|c)`. However, the pair `(a,c)` may not exist. Hence we use the two statistical properties of abbreviation that were discussed earlier to generalize the process of calculating `P(a |c)`. The properties describe how developers choose an abbreviation for a given English word. Formally,

$$P(a|c)=P(E=AbbType)\times P(X=TES(a,c)) \qquad (3)$$

Table 10. Computing the best candidate for expanding abbreviation `re` using different unigrams generated from source code and natural language corpus

Abbr. (a)	Candidate unigram (c)	a | c	Abb. Type	P(AbbType)	TES (a, c)	P (TES(a, c))	p (a | c)	p(c)		P(c|a) =P(a | c) *P(c)	
								SC	NC	SC	NC
re	result	re | result	CCA	0.84	0.67	0.19	0.1596	0.6263	0.0217	**0.09995748**	*0.00346332*
re	request	re | request	CCA	0.84	0.71	0.2	0.168	0.2765	0.0212	0.046452	*0.0035616*
re	response	re | response	CCA	0.84	0.75	0.2	0.168	0.2097	0.0143	0.0352296	0.0024024
re	resource	re | resource	CCA	0.84	0.75	0.2	0.168	0.1294	0.017	0.0217392	0.0028560
re	reader	re | reader	CCA	0.84	0.67	0.19	0.1596	0.0957	0.0087	0.01527372	0.00138852
re	read	re | read	CCA	0.84	0.5	0.14	0.1176	0.0697	0.0548	*0.00819672*	**0.00644448**
re	results	re | results	CCA	0.84	0.71	0.2	0.168	0.0624	0.0046	*0.0104832*	0.0007728
re	repository	re | repository	CCA	0.84	0.8	0.22	0.1848	0.0332	0.0015	0.00613536	0.0002772
re	rule	re |rule	NCA	0.16	0.5	0.14	0.0224	0.0835	0.0167	0.0018704	0.00037408
re	range	re |range	NCA	0.16	0.67	0.19	0.0304	0.0589	0.0218	0.00179056	0.00066272

Bold Numbers: Top candidates *Italic* Numbers: Different ranks between `P(c)` and `P(c|a)`

Table 10 shows the expanding of abbreviations using different unigrams. Our observations are as follows:

- The best candidates for the given abbreviation `re` may be different in different language domains. The words `results` and `read` have the highest probabilities of being the original English words in source code and natural language corpus, respectively. Bold numbers are the highest probabilities of candidates in source and natural language corpus, respectively.
- Regardless of the type of language, the frequency of a unigram is an essential fact in determining the best candidate. For example, both of the best candidates for `re` have the highest frequencies among all candidates in their corresponding unigram models.

- TES determines the best candidate if two of the same type of candidates have similar frequency in its language domains. For example, the two words `read` and `results` in the programming language domain are both CCA, the word `results` is considered as the better candidate even the word `read` has a slightly higher frequency than `results` does. Similarly, the word `request` is a better candidate than `result` in the domain of natural language (see Italic numbers in Table 9).

Note that (1) we only consider expanding a given abbreviation that is chosen from a simple English word due to the page limitation. The comprehensive approach to expanding other types of abbreviations, including abbreviations selected from multigram names, as well as the empirical study will be addressed in the future work and (2) we only demonstrate the importance of utilizing the properties of unigrams to solve well-known problems, such as understanding abbreviations. Empirical study regarding the comparison of different approaches for expanding abbreviations will be included in the future work.

VI. Conclusion

Unigrams collected from source code repository are useful for dealing software development problems, such as understanding the behaviors of developers and expanding abbreviations to improve the code readability. We have extracted 0.70 billion names from nearly 0.7 million projects. The names include variable, method, and class names. A total of 1.01 billion unigrams are generated from these constructs. In addition, 62 million abbreviations are extracted from 419 million variable names. To demonstrate the importance of unigrams, we have analyzed the patterns of how developers choose abbreviations, and then, we have used the properties of unigrams to expand abbreviations to original English words. The completed analysis results, including unigrams collected from variable, method, and class names, as well as abbreviations, can be accessed at https://goo.gl/HUz06W. The raw data extracted from GitHub can be accessed at https://goo.gl/nxzqHd. All variable names, method names, and class names are stored under the folders named `variables`, `methodName`, and `className`. Concerning the future work, we plan to expand our approach to collect and analyze bigrams and trigrams from source code repository. We also plan to conduct a larger empirical study to expand various types of abbreviations.

References

[1] D. Roth, "Learning to Resolve Natural Language Ambiguities: A Unified Approach," in *Fifteenth National Conference on Artificial Intelligence*, Madison, Wisconsin, 1998.

[2] H. Kamp and U. Reyle, From Discourse to Logic: Introduction to Modeltheoretic Semantics of Natural Language, Formal Logic and Discourse Representation Theory, Springer Science Business Media, 2013.

[3] N. Chomsky, "On certain formal properties of grammars," *Information and Control,* vol. 2, no. 2, p. 137–167, 1959.

[4] M. L. Scott, Programming Language Pragmatics, 3rd ed. ed., San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009.

[5] "GitHub," [Online]. Available: www.github.com. [Accessed 20 5 2016].

[6] S. Yoo, "Boa Language," [Online]. Available: http://web.cs.ucla.edu/~shyoo1st/boa/. [Accessed 2 8 2016].

[7] "Linguistics Data Consortium," Google.com, 2006. [Online]. Available: https://catalog.ldc.upenn.edu/LDC2006T13. [Accessed 2 6 201].

[8] "Google Ngram Viewer - Google Books," Google.com, 5 2012. [Online]. Available: https://books.google.com/ngrams. [Accessed 20 5 2016].

[9] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *10th IEEE Working Conference on Mining Software Repositories (MSR '13)*, 2013.

[10] M. Allamanis, E. T. Barr, C. Bird and C. Sutton, "Learning natural coding conventions," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*, 2014.

[11] M. Allamanis, E. T. Barr, C. Bird and C. Sutton, "Suggesting accurate method and class names," in *In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*, 2015.

[12] V. Raychev, M. Vechev and A. Krause, "Predicting Program Properties from "Big Code"," in *roceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2015.

[13] R. Dyer, H. . A. Nguyen, H. Rajan and T. N. Nguyen, "Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories," in *the 35th International Conference on Software Engineering*, Francisco, CA, 2013.

[14] T. Segaran and J. Hammerbacher, Beautiful Data: The Stories Behind Elegant Data Solutions, O'Reilly Media, 2009.

[15] "Hadoop," The Apache Software Foundation, [Online]. Available: http://hadoop.apache.org/. [Accessed 20 5 2016].

[16] G. Jenks, "wordsegment 0.6.2," [Online]. Available: https://pypi.python.org/pypi/wordsegment. [Accessed 22 5 2016].

[17] R. P. Buse and W. Weimer, "Learning a Metric for Code Readability," *IEEE Transactions on Software Engineering,* vol. 36, no. 4, pp. 546-558, 2010.

[18] D. Pierret and D. Poshyvanyk, "An empirical exploration of regularities in open-source software lexicons," in *The International Conference on Program Comprehension*, Vancouver, Canada, 2009.

[19] M. White, C. Vendome, M. L. Vásquez and D. Poshyvanyk, "Toward Deep Learning Software Repositories," in *International Conference on Mining Software Repositories* , Florence, Italy, 2015.

[20] E. Hill, Z. P. Fry, H. Boyd, G. Sridhara, Y. Novikova, L. Pollock and K. Vijay-Shanker, "AMAP: Automatically Mining Abbreviation Expansions in Programs to Enhance Software Maintenance Tools," in *the 2008 international working conference on Mining software repositories*, Leipzig, Germany, 2008.