Software understanding: Automatic classification of software identifiers

Pattaraporn Warintarawej, Anne Laurent, Marianne Huchard, Mathieu Lafourcade, Pierre Pompidor

Software Understanding: Automatic Classification of Software Identifiers

P. Warintarawej , M. Huchard, M. Lafourcade, A. Laurent,* P. Pompidor

LIRMM, UMR 5506 CNRS-Université Montpellier 2
161, rue Ada, 34095 MONTPELLIER CEDEX 05, France
Tel.: +33(0)467418585 - Fax: +33(0)467418500
{warintaraw, huchard, lafourcade, laurent, pompidor}@lirmm.fr

Abstract

Identifier names (e.g., packages, classes, methods, variables) are one of most important software comprehension sources. Identifier names need to be analyzed in order to support collaborative software engineering and to reuse source codes. Indeed, they convey domain concept of softwares. For instance, "getMinimumSupport" would be associated with association rule concept in data mining softwares, while some are difficult to recognize such as the case of mixing parts of words (e.g., "initFeatSet"). We thus propose methods for assisting automatic software understanding by classifying identifier names into domain concept categories. An innovative solution based on data mining algorithms is proposed. Our approach aims to learn character patterns of identifier names. The main challenges are (1) to automatically split identifier names into relevant constituent subnames (2) to build a model associating such a set of subnames to predefined domain concepts. For this purpose, we propose a novel manner for splitting such identifiers into their constituent words and use N-grams based text classification to predict the related domain concept. In this article, we report the theoretical method and the algorithms we propose, together with the experiments run on real software source codes that show the interest of our approach.

Keywords: Automatic Software Understanding; Data Mining; Text classification; Software Engineering.

*Corresponding author, e-mail: laurent@lirmm.fr

1 Introduction

Software development teams are often composed of groups of specialists, each given responsibilities depending on their role and skill set. In any software development project involving reuse of source codes, considerable time and effort must be invested to understand the existing work: How the code is built? How it operates and what conceptual is available on the software? In many cases, the code is not well annotated, software documentation is not available, not useful or outdated. Therefore, it is often the responsibility of programmers to work with the legacy source codes.

Generally, a software project contains many source files which involve a number of identifier names. Indeed, identifier names are often composed by mixing parts of words, abbreviations and acronyms (e.g., getARules, CLOPECluster, sIB_OptimizeT). Therefore, programmers can be easily confused when attempting to follow the domain concepts of identifiers. In this context, domain concepts refer to concepts of the working domain which are provided in software documentation, such as class diagram, package diagram (e.g., in data mining software tools, the domain concepts can be organized into folders according to package names such as association rules and classifiers (Fig. 1)). The most straightforward approach to understanding the concept from an identifier name is to split names into constituent words (e.g., *getProbability* $\rightarrow$ *get + Probability*). However, when identifier names are formed of partial mixed words, the splitting technique is no longer practical. For example, *sIB_OptimizeT* is split into *"s"*, *"IB"*, *"Optimize"*, and *"T"*, but to which domain concept the identifier is related remain unclear, especially if such splitting terms are not found in a dictionary.

In this work, we aim to assist automatic domain concept recognition, by studying methods to help programmers become rapidly familiar with a software, specifically the domain concept design embedded in identifier names. The challenge is not only to find the basic word concepts obtained by splitting terms of identifiers, but also to automatically extract the domain concept information conveyed by these identifiers. For example, given an identifier name *getARules*, the relevant concept is *association rules*.

We consider the high level software design which describes the major components of software systems and the relationships among those components. From a high level design phase to the implementation task, the domain concepts (hereafter concepts) are often organized into packages which are represented as folders or sub-folders. A package folder contains many related source files (e.g., class files) corresponding to the related concepts in design phase. Therefore, we assume that words or abbreviations which are found in source codes within the same concept (e.g., within the same package) can be

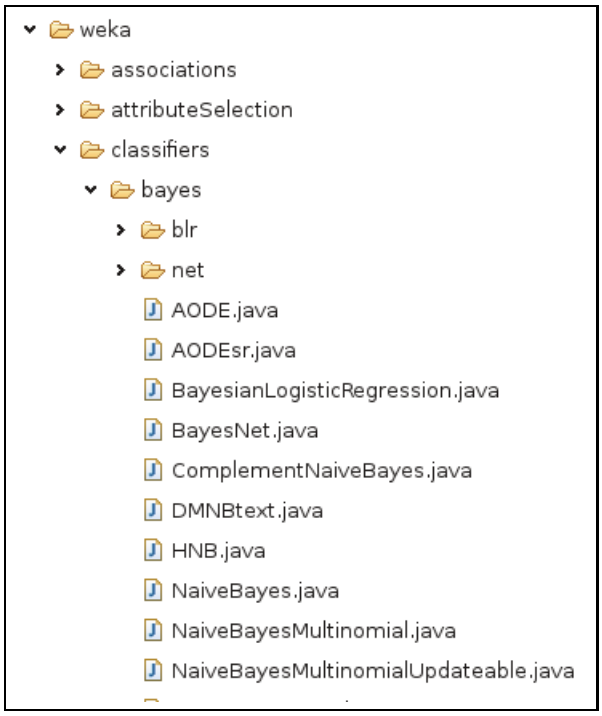

Fig. 1 High level software concepts by package names

used to infer concept overview. For instance, if “rule” occurs frequently within identifier names of association rule concept (e.g., numRules, randomRules, rulesItem), and is rarely found in other concepts, then “rule” can be used to refer to “association rules”. Therefore, the approach is to use the existing identifier names with predefined concepts (via the names of their packages) to predict the concept of an ambiguous identifier. As proposed in our previous works: [26], [25], text classification models are useful to find the related concepts of a new word where the new word is a combination of existing words with predefined concepts .

In this work, we propose a novel approach to predict a relevant concept of an ambiguous identifier based on character patterns of existing identifier names in predefined concepts, illustrated in Fig. 2. N-gram based text classification is considered. An N-gram is a sequence of *n* consecutive terms (in our case, we use characters) of a given text. The N-gram representation has the advantage of being more robust and less sensitive to grammatical and typographical errors and requiring no linguistic preparations which makes it more language independent. Given an identifier name, classifiers can learn the patterns

of existing identifier names in every predefined concept and use these evidences to assign the identifier name to the most relevant concept. Put differently, word components of existing identifier names can be used to validate software identifier names according to high level domain concept design.

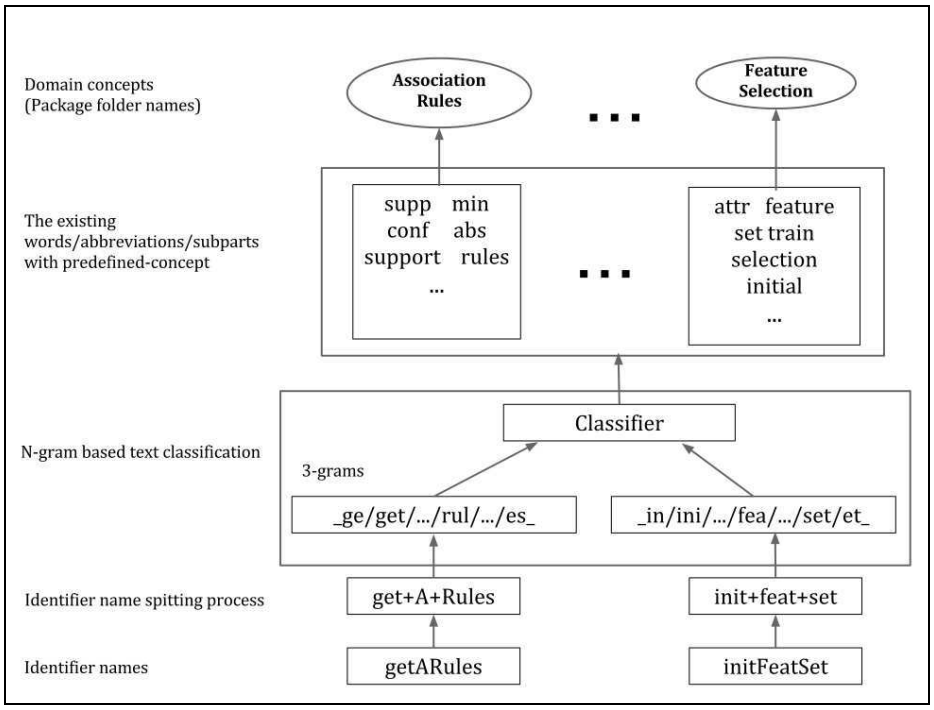

Fig. 2 The identifier name analyzing approach

The approach developed here consists of two main tasks: (1) the splitting identifier algorithm (*Sword*) and (2) N-grams based text classification of identifier names. The contributions are presented as follows.

- We state that the main characteristics of predefined domain concepts of softwares can be learned by identifier names. To provide an approach for helping automatic software understanding by analyzing identifier names, data mining algorithms are taken into account. We apply text classification models to predict the relevant concept of a new identifier name based on character N-grams of existing identifier names taken from predefined domain concept source codes.
- We propose a novel algorithm for splitting identifier names, named *Sword*. The algorithm aims to improve the accuracy of identifier name splitting in the case of

partial word mixing forms (e.g., *getabssupp*, *initFeatSet*). The existing techniques of compound word splitting and text classification models become inefficient when they encounter with subpart word mixing identifiers. *Sword* not only copes with incomplete word combination, but also with compound words, abbreviations and single case identifiers.

The paper is organized as follows. Section 2 presents related work regarding identifier name decomposing in software engineering tasks. Section 3, we review the existing splitting algorithms and propose a novel identifier splitting algorithm. In Section 4, we present how N-gram based text classification can be applied on identifier names, we describe how to represent identifier names in bag-of-word models based on N-grams. We report the experimental results of splitting algorithms and the classification of identifier names by using N-grams in Section 5. Finally, we conclude and present future work in Section 6.

2 Related work

Most software engineering tasks involve identifier name decomposition; identifier names must be split into word components so that they can be analyzed in software engineering tasks (e.g., clustering concepts, feature location). Related work on decomposition of identifier names have been proposed and are described below.

[1] proposed a method to decompose filenames into a list of constituent elements which are called "concepts". The work consists of 2 steps: (1) extracting candidate splitting terms from the existing source codes and (2) decomposing file names into splitting terms. Many sources of words and abbreviations can be used to generate a set of candidate splitting terms (e.g., file names, comments, identifier names). Next, source words (in case of filenames) are split using the candidate splitting terms and N-grams. An English dictionary is also taken into account in order to enhance the decomposing process. Their results show that identifier names split by N-grams obtained the best accuracy (88.4%), followed by file names with N-grams. When using a dictionary, the accuracies for all sources are clearly improved. The work therefore describes a simple method for the decomposition identifier names by splitting substrings contained in identifier names (but is unable to handle subpart words combinations).

[5] proposed an approach to restructure programming identifier names with the aim of improving the meaningfulness of identifier names. The approach comprises 3 principal tasks: (1) Building a standard lexical dictionary (2) mapping identifier names into

standard terms (lexical standardization) and (3) rearranging the components of an identifier name according to the standard structure found from existing codes and the expert suggestion (syntactical standardization). Unfortunately, the task of identifier name decomposition is not performed automatically in this case. The system instead provides a tool for users to edit the segmentation of identifier names, and no evaluation is provided.

[15] focuses on cases where there no word marker is found in identifier names and describes two splitting techniques for such identifiers. A greedy algorithm and a neural network are discussed. 4,000 identifier names were randomly selected from the 186 programs and the results show that the neural network is faster and more adaptable to the intuitive splitting.

[12] proposed an approach to split identifier names, named *Samurai*. The Samurai algorithm is able to automatically split identifier names into sequences of words by mining word frequencies in source codes. With these word frequencies, the identifier splitter uses a scoring technique to automatically select the most appropriate partitioning for an identifier. The results show that frequency-based token splitting misses same-case splits identified by the greedy algorithm [15], but outperforms the greedy overall by making significantly fewer oversplits.

[4] looked at how to improve the tokenization of identifier names when they appear in single case forms or contain digits. The work proposes to use oracles to identify the boundary of tokens; oracles can be lists of words from dictionaries, a list of abbreviations and acronyms. and a list of acronyms containing digits. The pre-process of string splitting with digits is performed by the heuristic rules proposed by the author.

The naming of meaningful identifiers has been widely studied. [8] observed identifier names from Eclipse (Java code based) and established a solid foundation for the identifier naming definition. A dictionary is used to help with consistent naming rules and the suitable names are generated in a context-specific manner, but no splitting technique is provided.

[18] exploited identifier names from 100 Java applications to extract name-specific implementation rules for the most common method names and define an identifier naming bug. The work presents an approach for automatic suggestion of more suitable names in the presence of mismatch between name and implementation.

[13] proposed a wordNet-like approach to extract the structure of a software using the relationships among identifier names. The approach considers Natural Language Processing techniques which consist of tokenization process (straightforward decomposition technique by word markers, e.g. case changes, underscore etc.), part of speech tagging,

and rearranges order of terms by the dominance order of term rules based on part of speech.

[2] exploited the mining part of speeches of identifier names. The tool provides part-of-speech information, which improves the searching of software repositories. Although the tagging can be used to support improved naming, no association between identifiers and domain concepts is provided.

[3] investigated the lexical and syntactic composition of Java class identifier names. This work identifies conventional patterns found in the use of part of speech. This work also developed a tool to identify the structure of identifier names among super classes and implemented interfaces. The authors demonstrates how this knowledge can be applied in case of unconventional identifier names by refactoring a name into a conventional form.

Despite the advances described above, these studies fail to address the problem of splitting identifiers names where they are composed of subparts of words, abbreviations, with no word maker presences (e.g., *getabssupp*), or when no partial matching is provided. Furthermore, the issue of automatic domain concept understanding by a software using supervised learning has never been addressed. We take advantage of text classification and the lists of programming terms, as well as dictionary words to enhance identifier splitting process [1]. We also apply the greedy algorithm [15] and *Samurai* approach [12] to generate a novel splitting technique capable of handling the problem of substring mixing in identifier names.

3 Identifier Splitting Algorithms: Related Work and Original Approach

Identifier names are used to define the entities in a software (e.g., names of packages, classes, attributes, constants). Identifier names can be composed of a set of characters according to the rules of programming languages. For example, the initial character can not normally be a digit or it usually forbidden to use most of the special characters. Typically, programmers create identifiers by mixing several words or abbreviations (e.g., isMatchingEOL, JEditBuffer, editSyntaxStyle, raduis2) to refer to working domain concepts. Therefore automatic splitting of multi-word identifiers needs to be addressed to capture word concepts from the base words.

In this Section, we review the state of the art of identifier splitting techniques, namely the CamelCase splitter, the greedy algorithm [15] and the Samurai approach [12]. We then propose a novel algorithm to split identifier names. This algorithm is based on both

the greedy algorithm and the Samurai approach, especially for handling the case of word subparts or abbreviations (e.g., badNumEx, initPRforCP, getAbsSupp).

3.1 CamelCase splitting technique

CamelCase technique is a simple and widely used method for identifier splitting algorithms [9] and the rules of splitting are broadly based on CamelCase convention. For instance, *getItemName* is split into *get*, *Item*, *Name* or *setID* is split into *set* and *ID*. If two more upper case characters are followed by one or more lower case character the identifier is split before the last upper case character: e.g., *JScrollPane* is split into *J*, *Scroll* and *Pane*. If there are some underscores, they are replaced by space characters: e.g., *do_Click* is split into *do* and *Click*. However, the CamelCase splitting algorithm cannot handle single case word composition such as DBNAME, maxvalues.

3.2 The greedy algorithm

The greedy algorithm [15] aims to solve the problem of single case identifier splitting. The algorithm consists of two searching approaches: (1) the longest prefix search and (2) the longest suffix search. In this case, prefixes and suffixes mean words which occur in identifier names (and not as defined in linguistics). The longest prefix searching scans the longest substring in an identifier name which matches words from a dictionary, and keeps the longest substring. The longest suffix searching performs the same searching but in the opposite direction. The results of two searches are compared and the substring which gives the highest ratio of term occurrences is retained in the analyzing software. The algorithm recursively runs until no string remains in the identifier. If neither the prefix nor the suffix searching generate a dictionary term, the process is repeated by removing the first character of the identifier name.

Searching approaches	Searching results	The longest substring
The longest prefix search	{the, then,..., newest,...}	newest
The longest suffix search	{one, stone, ...}	stone

Table 1 An example of the greedy algorithm (identifier = '*thenewestone*")

For example, if an identifier name = "*thenewestone*", the set of prefix searching can be {the, then,..., newest,...}, and the longest prefix substring is *newest*. While the set of suffix searching can be {one, stone, ...} and the longest suffix substring is *stone* (Table 1). The

ratio score is used to decide which term will be split. This technique can overcome the single case problem of identifier names. Nevertheless, abbreviations and word subparts mixing remain a problem for this technique.

3.3 Samurai splitting approach

Samurai is an automatic identifier splitting algorithm proposed by [12]. The approach relies on term frequencies in source codes and is based on the assumption that identifiers are often composed of terms used frequently in source codes. Samurai addresses the problem of CamelCase technique when identifiers contain consecutive upper case characters (more than two). For example, CamelCase splits *KNNclassifier* into *KN* and *Nclassifier*, instead of *KNN*, *classifier*. Samurai overcomes this problem by using two tables of frequencies: the first contains a programming term a list of an analyzed source code, while the second is list of terms from a large programming corpus. Samurai runs CamelCase splitting and creates alternative splitting terms, before ranking candidates using the Score function from Equation 1.

$$Score(s) = Freq(s,p) + \frac{globalFreq(s)}{log_{10}(AllStrsFreq(p))} \quad (1)$$

where p is a source code under analysis. $Freq(s,p)$ is the frequency of term s in source code p. $globalFreq(s)$ is the frequency of term s in a large programming corpus. $AllStrsFreq(p)$ is the total occurrence number of all terms found in the source code p.

In the single case identifier name problem, the approach of Samurai is to find the best splitting position in an identifier name. To achieve this goal, an identifier name will be divided into the left and the right term. The best position is where the summation of the left-right term score gives the highest value (Maximum (score(left)+score(right))) among all left and right candidate terms. A set of left and right candidate terms is obtained using a sliding window technique, starting the split from the first character and continuing one character at a time (e.g., *getname* $\Rightarrow$ (*g etname*), (*ge tname*)) until finding the maximum score for the left and the right terms. For example, the terms which obtain the maximum score of the left-right summation are *get* and *name*, then split the left term (*get*) and recursively process the right term until no more string remains in the token, (e.g., Table 2).

Samurai overcomes the limitation of CamelCase and also handles the single case problem of identifier names. However, Samurai does not address the word subparts mixing identifier. Moreover, the algorithm run time is correlated with the number of char-

Round	Left/Right	Maximum (score(left)+score(right))	keep
Round 1, token="getname"	getname/- g/etname ge/tname get/name getn/ame getna/me getnam/e	✓	get
Round 2, token="name"	name/- na/me ...	✓	name

Table 2 An example of Samurai algorithm (identifier = "*getname*")

acters in an identifier. The follow Section, we propose an algorithm to overcome this problem.

3.4 Our Original Approach: Sword

Our approach, named *"Sword"*, aims to combine the greedy and Samurai algorithms in order to tackle the identification of word subparts, abbreviations and single case identifiers. Firstly, a set of candidate splitting terms is created and three lists of words used to match with an identifier: the list of abbreviations found in the analyzed source code, the list of words found in the existing source codes (the standard programming terms) and a list of words from a dictionary. Exactly matched words from the lists are retained in the candidate set together with substrings from the partial matchings. Secondly, the candidate splitting terms are ranked using the score function from Samurai. Finally, the term giving the highest score from Eq. 1 is split from the identifier. The algorithm recursively performs the remaining terms until no string is left in the identifier.

The following steps show Sword (Algorithm 1) using with the $identifier$ = *"getabssupp"* as an example:

1. An identifier name is split into tokens (we keep only the token whose length is more than 2 characters, otherwise we look for it in the abbreviation list) using word markers (underscore, CamelCase) if they exist. E.g., there is no word marker in *"getabssupp"*, then $token$ = *"getabssupp"*).

2. For every token,

- If the token is found in the lists of words (the standard programming terms or a dictionary), splitting stops.
- If no word from the lists matches the token, then perform *partial* matching between the token and words from the list (use the standard programming list first, then if no word is found then use a dictionary) (for our example, *matchList* = {get, sup, supp}) (line 11).
- The set of candidate splitting terms is separated into two sets. The first set is the substring that is found in the earliest position of the token (*e.g. strMinPosition* = {get} (line 12)). The second group is the longest substring matched (*e.g. strMaxLength* = {supp} (line 13))
- The scores of substrings in the two sets are computed and the substring that obtains the highest score is retained. If the scores of both terms are equal, then the longest substring is retained (*e.g.*, assume, the kept splitting term is *keep* = {get} line 14-21).
- The remaining strings in the token will be split into the left and the right term. The algorithm is recursively performed for both remaining terms until no string remains in the token (*e.g.*, $left$ = " ", *right* = {abssupp}, line 22-36).
- Finally, if the token is not found in any word from the lists, the list of abbreviations is used. If the token does not match any abbreviation then the original token remains as a splitting term (line 40-45).

The example of splitting *"getabssupp"* is shown in Table 3.

Token	Words/ Substring matched	The earliest position term	The longest substring	Keep	Remark
getabssupp	{get, sup, supp}	get	supp	get	{get} from the standard programming terms {sup, supp} from the partial matching
abssupp	{sup, supp}	sup	supp	supp	The longest substring, if the scores of both terms are equal or zero
abs	{abs}	-	-	abs	{abs} found from the abbreviation list

Table 3 An example of the *Sword* identifier splitting algorithm

Sword provides an efficient method for splitting identifier names into relevant word

subparts and the next Section describes how these subparts can be exploited by text classification algorithms to achieve improve methods for automatic identifier name classification.

4 N-gram Based Text Classification of Identifier Names

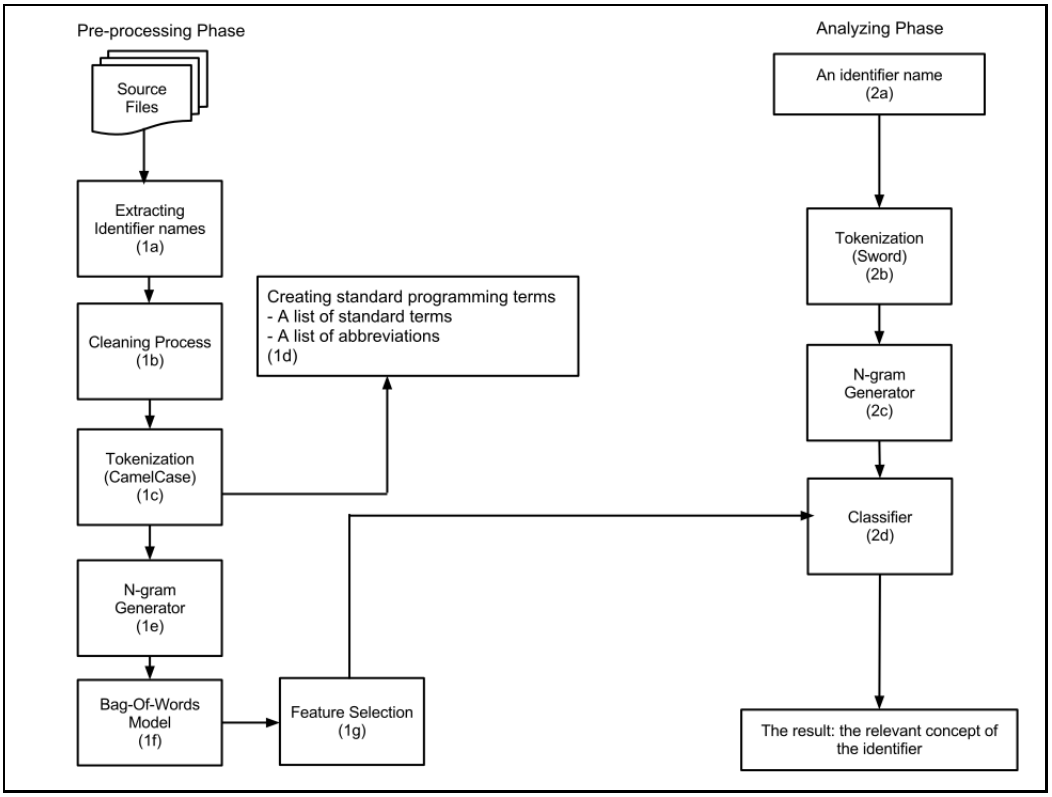

Fig. 3 N-gram based text classification of identifier names

Classification is a data mining technique that assigns items into predefined categories or classes. In our context, items are equivalent to text and the technique is called the text classification model. An example of it use would be to automatically label an incoming news story into a topic such as “sports”, “politics” or “art” etc. Typically, whatever the classifiers are employed, text classification tasks start with a training set of documents that have been labeled with class(es). Generally, text classification models require a suitable representation of text, as plain text cannot be processed directly in a text classification

model. N-grams is a language independent text representation technique that transforms text into high dimensional feature vectors where each feature corresponds to a contiguous substring. N-grams consist of *n* adjacent characters (substring) of a given text [6]. We use character N-grams to represent terms (identifier names, words) in source codes and extracting character N-grams from a term by moving an *n* character wide-window across the term character-by-character. We insert "_" into the terms to manage space for the first, last and internal positions (e.g., "numRules" $\rightarrow$ "num Rules" $\rightarrow$ "_nu num um_ m_r _ru rul ule les es_"). The methodology of N-grams based text classification of identifier names is shown in Fig. 3 and the methodology can be summarized as follows:

1. The first step prepares the training dataset by the data pre-processing:
   (a) Extracting identifier names from source codes: this step aims to extracting identifier names, and also file names and comments that can be considered informative [1]. Moreover, the packages names must be stored as domain concept names for each source file.
   (b) Cleaning Process: this step to removes non informative terms such as special characters, reserved words and stopwords.
   (c) Tokenization: At this stage, identifier names are split into primitive words by CamelCase. We put "_" to separate between word components such as "setMinSupp" $\Rightarrow$ "get_min_supp".
   (d) Creating standard term list and an abbreviation lists: words matching dictionary entries will be added to a standard programming list, otherwise they are placed in the abbreviated list (awaiting refinement by experts).
   (e) N-gram Generation: to separate terms from the result of tokenization into sequences of *n* characters. For instance, 3-grams of " set_min_supp" will be:

$$\_se, set, et\_, t\_m, \_mi, min, in\_, n\_s, \_su, sup, upp, pp\_$$

   (f) Bag-of-word models: this step aims to create a vector model of identifier names based on N-grams (see Section 4.1 for details).
   (g) Feature selection techniques: Chi-square feature selection is used to reduce the number of features and select discriminative features for the text classification model.
2. The analysis of phase deals by assigning domain concepts to testing identifier names. For example, an identifier name = *"getabssupp"*

(a) Input an identifier name.

(b) Tokenize the identifier by *Sword* algorithm (e.g., splitting getabssupp by Sword produces get abs supp).

(c) Generate N-grams of the identifier name according to N-grams from pre-processing phase (e.g., 3-grams: _ge get eta tab abs bss ssu sup upp pp_).

(d) Classify the identifier name into a domain concept.

(e) The result shows the relevant concepts with respect to the classification scores obtained from the classifier. For example, Naive Bayes classifier classifies *"getabssupp"* into "Association Rules" with respect to the highest conditional probability of the concept ("Association Rules"), given by the identifier (*"getabssupp"*) $(P(c_j|w_i))$.

4.1 Bag-of-word Models

In this work, identifier names or words found in source codes are represented by a vector of N-gram terms (e.g., Table 4).

Identifier	3-grams	Domain concept
numRules	"_nu num um_ m_r _ru rul ule les es_"	associationRules
randomRule	"_ra ran and ndo dom om_ m_r _ru rul ule le_"	associationRules
numClasses	"_nu num um_ m_c _cl cla las ass sse ses es_"	classification
classifiers	"_cl cla las ass ssi sif ifi fie ier ers rs_"	classification
bestClusters	"_be bes est st_ t_c _cl clu lus ust ste ter ers rs_"	clustering
tmpClusters	"_tm tmp mp_ p_c _cl clu lus ust ste ter ers rs_"	clustering
noAttributes	"_no no_ o_a _at att ttr tri rib ibu but ute tes es_"	featureSelection
selectedAttributes	"_se sel ele lec ect cte ted ed_ d_a _at att ttr tri rib ibu but ute tes es_"	featureSelection

Table 4 The examples of identifier name representations on 3-grams

Identifier names (or words) are represented by a vector of N-gram features. Let an identifier name w_i be a vector space that consists of features (t_j) where $j \in [1..|S|]$, and $|S|$ is the number of the features:

$$w_i = < t_{i1}, t_{i2}, t_{i3}, ..., t_{i|S|}, c_k >$$

where t_{ij} is frequency feature t_j in w_i, c_k is a domain concept, and for $k \in [1..m]$, m is the number of domain concepts in the training set.

Table 5 shows the construction of the bag-of-word representation (3-grams).

Identifier	...	_ru	rul	le_	...	_at	att	Domain concept
numRules	...	1	1	1	...	0	0	associationRules
randomRule	...	1	1	1	...	0	0	associationRules
noAttributes	...	0	0	0	...	1	1	featureSelection
selectedAttributes	...	0	0	0	...	1	1	featureSelection

Table 5 The bag-of-word of identifier names based on 3-grams

4.2 Feature Selection Techniques

The main problem of text classification is the high dimensionality of textual data. Feature selection methods have been proposed to select the most relevant attributes [28]. It has been shown that feature selection improves classification effectiveness and computation efficiency.

Mutual information (MI) is an established measure for many successful feature selection techniques in text classification. MI measures how much information of a feature t is related to a concept regarding its presence or absence in each concept compared to other concepts. Mutual information can be defined as [22] :

$$\begin{aligned} I(U;C) &= \frac{N_{11}}{N}\log_2\frac{NN_{11}}{N_{1.}N_{.1}} + \frac{N_{01}}{N}\log_2\frac{NN_{01}}{N_{0.}N_{.1}} \\ &+ \frac{N_{10}}{N}\log_2\frac{NN_{10}}{N_{1.}N_{.0}} + \frac{N_{00}}{N}\log_2\frac{NN_{00}}{N_{0.}N_{.0}} \end{aligned}$$

where N_{10} is the number of words that contain feature t and not in concept c. For example, $N_{1.} = N_{10} + N_{11}$ is the number of words that contain feature t, $N = N_{00} + N_{01} + N_{10} + N_{11}$ is the total number of words in the domain.

Chi-square (χ^2) is one of the most efficient methods for optimizing classification results [11, 20].

χ^2 is a test of independence between two variables. In text classification, let us define t as a term in text and c as a class. In our work, t refers to features such as N-grams and c refers to concepts from a thesaurus. The main idea of this technique is to select discriminative features by measuring the dependence between features and concepts. As a result of χ^2 test, a feature (t) is selected only if it associated with a concept. χ^2 statistic is defined as following [22]:

$$\chi^2(\mathbb{D},t,c) = \sum_{e_t\in\{0,1\}}\sum_{e_c\in\{0,1\}}\frac{N_{e_te_c} - E_{e_te_c}}{E_{e_te_c}}$$

When e_t indicates the appearance of term t and e_c indicates the appearance of the concept c. For example, N_{10} is the number of words that contain term t ($e_t = 1$) but not in concept c ($e_c = 0$). $E_{e_t e_c}$ is the expected frequency, E_{11} means the expected frequency of term t in concept c as follows:

$$E_{11} = N * \frac{N_{11} + N_{10}}{N} * \frac{N_{11} + N_{01}}{N}$$

4.3 Classification Models

4.3.1 Naive Bayes

The Naive Bayes classifier is a simple classifier model based on probabilistic theory. The Bayesian theorem includes an independence assumption called *Naive Bayes assumption*. The assumption assumes that all attributes of the examples are independent. In fact this assumption is not correct in real-world text classification, however, Naive Bayes performs very well in classification tasks as reported in [10]. The probability model for the classifier is a conditional model $P(c_j|d_i)$ when c_j is a member in the set of concepts C in classification domain and d_i is a testing document. The conditional probability of concept c_j given by document d_i can be defined as :

$$P(c_j|d_i) = \frac{P(c_j)P(d_i|c_j)}{P(d_i)}$$

.

In our task, the classification model follows Naive Bayes formula as:

$$P(c_j|w_i) = \frac{P(c_j)P(w_i|c_j)}{P(w_i)}$$

where word w_i is represented by a feature vector as defined in section 4.1.

The most relevant concept is selected by maximum a posteriori probability (C_{map}):

$$\begin{aligned} C_{map} &= \arg\max_{c_j \in C} \hat{P}(c_j|w_i) \\ &= \arg\max_{c_j \in C} \hat{P}(c_j)\hat{P}(w_i|c_j) \end{aligned}$$

4.3.2 kNN

k-Nearest Neighbor (kNN) is a similarity-based learning algorithm which is very effective for various domains in text classification [16, 27, 21]. For a given unlabeled example, the algorithm finds the closest k labeled examples in the training set and assigns the class by majority voting among those examples in the set of k neighbors.

Several measures can be considered in order to compute the similarity between words in training set and a testing identifier name. In our case, we consider the Euclidean distance. The Euclidean distance between words w_i and w_j with $w_i =< t_{i1}, t_{i2}, t_{i3}, ..., t_{in} >$ and $w_j =< t_{j1}, t_{j2}, t_{j3}, ..., t_{jn} >$, where n is the number of features in the vector space is defined as:

$$\Delta(w_i, w_j) = \sqrt{\sum_{m=1}^{n} (t_{im} - t_{jm})^2} \tag{2}$$

The algorithm used to compute the k-nearest neighbors is shown below:
Input : an identifier name, Output: k nearest neighbor identifiers of an input identifier.

1. Determine the number of k nearest neighbors beforehand.
2. Sort the distances for all the training samples and determine the nearest neighbor based on the K-th minimum distance.
3. Group the examples in the set of k nearest neighbors regarding to their concepts.
4. Assign the concept by majority voting among the set of k neighbors.

4.4 Evaluation Metrics

The performance of text classification model can be measured in many ways. The commonly used measures are *Accuracy*, *Precision* and *Recall* [24]. In this work, the accuracy measures the correctness of assigning a domain concept to an identifier. Precision is the fraction of retrieved identifiers that are relevant in a domain concept. Recall is the fraction of relevant identifiers that are retrieved. The evaluation metrics for a domain concept c_j are defined as follows:

$$
\begin{aligned}
Precision &= \frac{\#correctly\ classified\ identifiers\ of\ c_j}{\#identifiers\ that\ are\ classified\ in\ c_j} \\
Recall(tp\ rate) &= \frac{\#correctly\ classified\ identifiers\ of\ c_j}{\#identifiers\ in\ c_j} \\
F-Measure &= \frac{2 * precision * recall}{precision\ +\ recall} \\
Accuracy &= \frac{\#correctly\ classified\ identifiers\ of\ c_j}{total\ number\ of\ identifiers} \\
fp\ rate &= \frac{\#incorrectly\ classified\ identifiers\ of\ c_j}{\#identifiers\ not\ in\ c_j}
\end{aligned}
$$

Although the measures above have been used as standard, simple classification accuracy is often a poor metric for measuring performance [23]. In most real-world cases, the distribution of classes is skewed and the cost of error for one type of classification is much more expensive than another. In addition, most of the classifiers (Naive Bayes, kNN, etc.) can produce the probability or "confidence" of class prediction. Unfortunately, this information is ignored in the classical metrics described above [19]. Receiver Operator Characteristic (ROC) analysis is used to solve accuracy measure problem. ROC curves are not insensitive to changes in class distribution. If the proportion of positive to negative changes in a test set, the ROC curves will not change. Since ROC based on *tp rate* (recall) and *fp rate*, not depend on class distribution [14].

5 Experimental Results

5.1 Dataset

To select training sets, domain knowledge is required. Domain experts play important role to select relevant source files; identifier names contain words or strings referring to domain concept information (e.g., identifier names in association rule concept (from data mining software tools) contain subnames: "rules", "itemset", "support", "confidence"), To our knowledge, we setup the preliminary experiment by using data mining software packages as training sets. We have selected 4 packages (called *"concept"* in our work)

taken from two data mining software tools: ARuleGUI[1] and Java Machine Learning Library (Java-ML)[2]. The identifier names are collected from file names, identifier names (classes, methods, attributes) and comments. Data pre-processing performed by removing reserved words of Java and stopwords for comments. The total number of identifier names for each domain concept is shown in Table 6. The number of words in the standard programming term list and the number of abbreviations are shown in Table 7. English words are obtained from the standard file on Unix or Unix-like operating systems (*ispell*). In our case, we use Ubuntu 12.04.1 which *words* file contains 99,171 words from */usr/share/dict/words*.

Concept	**#Occurrences of identifier names**	**#Distinct identifier names**	**#Source files**
Association Rules	4,394	657	8
Classification	4,578	819	22
Clustering	7,715	1,415	20
Feature Selection	1,692	406	14
Total	18,379	3,297 (2,672 unique names)	64

Table 6 Concepts and the number of identifier names

The list	**#words**
Standard programming terms	1,226
Abbreviations	55
English words	99,171

Table 7 The number of words in various sources

Each word from the domain concepts is separated into 3 and 4-grams and they are converted into a bag-of-word model. The overall numbers of features are 2,029 and 2,718 for 3 and 4-grams respectively. Chi-square feature selection is used to select discriminative features. Naive Bayes and kNN from the WEKA packages[17] are used to classify an identifier name.

[1] http://www.borgelt.net/argui.html
[2] http://java-ml.sourceforge.net/

5.2 Identifier Splitting Evaluation

The testing set contains 571 identifier names that have been randomly selected from training dataset. Our algorithm obtains 92.8% accuracy, which is better than the greedy algorithm and Samurai approach (86.0% and 83.8% respectively). The highest accuracy was achieved when using identifiers composed of abbreviations or word subparts, in particular when they are not in the word lists.

5.3 Results of Naive Bayes Classifier

The results of Naive Bayes classifier on 10-fold cross validation[3] show that 4-grams approach obtains the highest level of accuracy (73.7%) with 2,718 features. The 4-grams approach also outperforms the 3-grams approach in recall, f-measure and ROC. When the number of features increases, evaluation measures mostly increase, (Table 8 and Fig. 4).

N-gram	#Feature	Accuracy(%)	Precision(%)	Recall(%)	F-measure(%)	ROC(%)
3	1,000	69.7	69.7	69.7	69.5	87.3
3	2,000	71.2	71.4	71.2	71.2	88.6
3	2,029 (all)	71.2	71.4	71.2	71.2	88.6
4	1,000	70.1	70.3	70.0	69.5	85.8
4	2,000	72.9	72.9	72.9	72.7	89.0
4	2,718 (all)	**73.7**	73.6	73.5	73.5	89.5

Table 8 The result of Naive Bayes classification on 10-fold cross validation

5.4 Results of kNN Classifier

The results of kNN classifier on 10-fold cross validation show that the highest accuracy (74.6%) is obtained by 1,000 features using $4-$grams with $k = 5$, (Table 9). When k increases, the accuracy slowly drops (Fig. 5). It should also be noted that 4-grams outperforms 3-grams on the average of accuracies. However the difference between the the accuracies of 3 and 4-grams is marginal (Fig. 6). We also found that kNN outperforms Naive Bayes (Fig. 7).

[3]Cross validation is a technique to ensure that every example from the dataset has the equal chance of appearing in the training and testing set. *n*-fold cross-validation: divide the dataset up into *n* groups and *n* train times, treating a different group as the holdout set each time, more detail: http://www.cs.cmu.edu/ schneide/tut5/node42.html

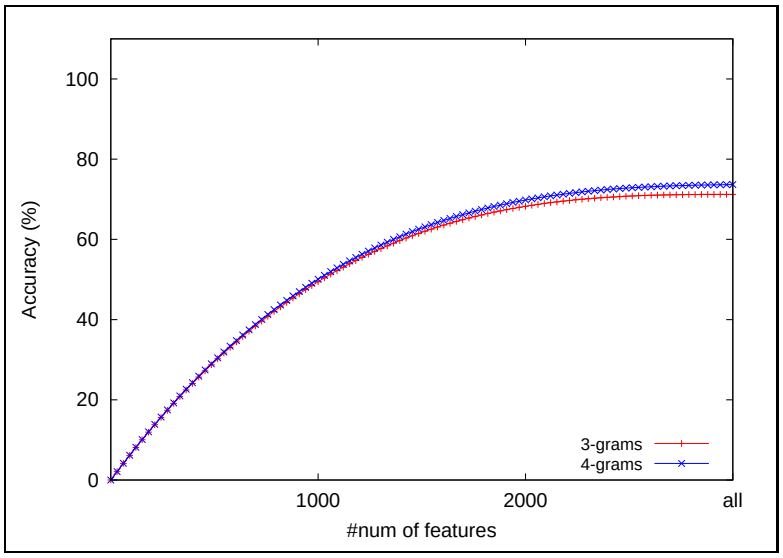

Fig. 4 The accuracy of Naive Bayes

N-gram	k	#Feature	Accuracy(%)	Precision(%)	Recall(%)	F-measure(%)	ROC(%)
3	5	1,000	74.2	74.7	74.2	73.4	91.1
3	10	1,000	72.0	72.7	72.0	70.9	89.7
3	15	1,000	69.9	71.1	69.9	68.6	88.5
3	5	2,000	74.3	74.7	74.3	73.5	91.2
3	10	2,000	71.3	72.2	71.3	70.2	89.6
3	15	2,000	68.6	70.0	68.6	67.3	88.4
3	5	2,029 (all)	74.3	74.7	74.3	73.5	91.2
3	10	2,029 (all)	71.3	72.1	71.3	70.2	89.6
3	15	2,029 (all)	68.5	69.9	68.5	67.1	88.4
4	5	1,000	**74.6**	75.4	74.6	73.8	91.1
4	10	1,000	71.1	72.7	71.1	69.8	88.8
4	15	1,000	68.3	70.8	68.3	66.6	87.0
4	5	2,000	72.3	74.2	72.3	71.3	88.6
4	10	2,000	69.9	72.1	69.9	68.4	87.2
4	15	2,000	67.8	70.5	67.8	66.0	85.9
4	5	2,718 (all)	74.0	75.4	74.0	73.0	90.6
4	10	2,718 (all)	70.6	72.7	70.6	69.0	88.7
4	15	2,718 (all)	68.1	70.9	68.1	66.1	87.0

Table 9 The results of kNN on 10-fold cross validation

5.5 Interpreting the Identifier Classification

Given an identifier name, the classifier finds the most relevant concept based on character patterns in the existing identifiers. If the testing identifier is composed of substrings often

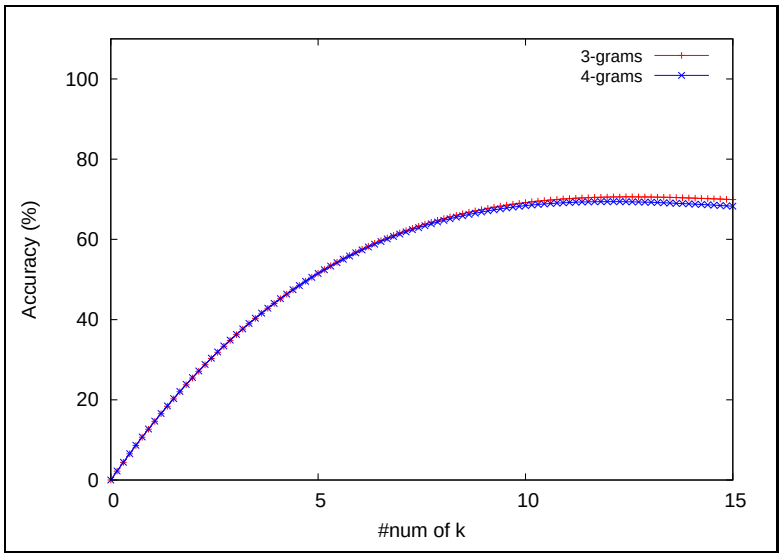

Fig. 5 The accuracy of kNN on 1,000 features

arising in a specific concept, the classifier will assign the identifier to that concept. For example, *getabssupp* is classified into association rule concept (Fig. 8). These results can be explained using the set of discriminative features: *abs* and *supp* are members in a set of discriminative features in association rule concept.

6 Conclusion and Future work

In this article, we propose an approach based on text classification to assist automatic software understanding. Identifier names are used by programmers to better understand the role of code parts (packages, functions, etc.). For this purpose, identifier names must first be split into their root subcomponents. Thus, we propose a novel identifier splitting algorithm capable aims to handle software identifiers built from multiple subparts. We demonstrate that our identifier splitting algorithm achieves better accuracy than the existing techniques. An N-gram based text classification is also run in order to classify identifier names into predefined concepts and experimental results show the applicability of our approach to real world problems.

For legacy software, one main problem is to understand the software. The analysis of identifiers is a clue in this understanding. Then, being able to analyze identifiers and extract their meaning favors several refactoring operations that are based on terms. In the

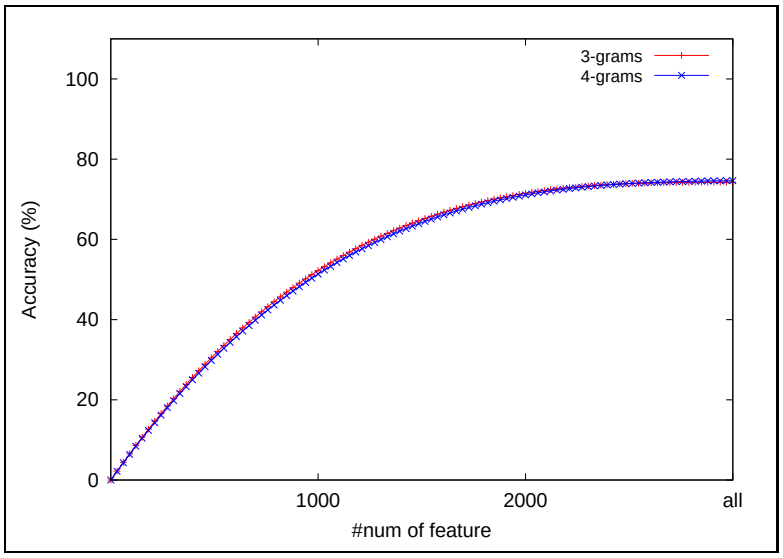

Fig. 6 The accuracy of kNN when k= 5

case of forward engineering, our technique can help choosing the right identifiers, that will help further understanding and use.

In future work, we consider the study of how such a technique can be integrated during software development to prevent programmers creating identifiers evoking irrelevant concept. In software development technology, we would like to investigate how our approach (which may help with checking or construction of linguistic cohesion in a set of identifiers) could be associated with judging or finding modules, or with mining separate concerns (cross-cutting) in Aspect-oriented software development [7]. The challenge is how to integrate the identifier analysis into software development cycle, the data preparation of legacy software involving domain knowledge must be discussed.

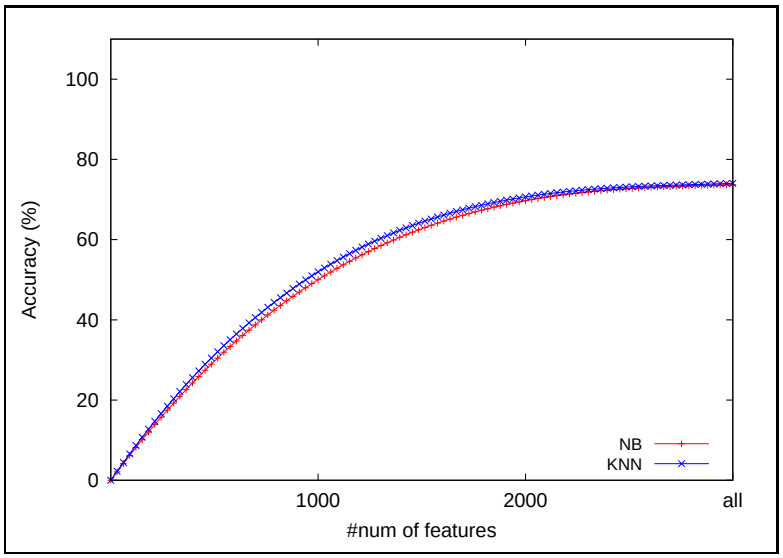

Fig. 7 The accuracy of Naive Bayes and kNN, k=5

Result Identifier Analyzing	
Identifier Name:	getabssupp
Constituent words :	get abs supp
Classification Results:	total number of abbrevations: 55 total number of words in Standard List: 1226 total number of words in Dictionary: 99171 total number of words in Word Frequency List: 239208 Feature type: 3grams Total features: 1000 Total train set: 17470 The result of classification (concept, probability) concept : clustering = 0.0 concept : featureSelection = 0.0 concept : associationRules = 1.0 concept : classification = 0.0 The maximum a posterior probability: associationRules
Predicted concept :	associationRules
Time execution :	Page was generated by PHP 5.3.10-1ubuntu3.4 : in 00:00:17

Fig. 8 An example of identifier name classification

Algorithm 1: Sword

```
Input: token is a token to be split,
       dictionary is a set of words in a dictionary,
       standardList is a set of standard programming terms,
       abbreviationList is a set of abbreviations found in the analyzed programs
Output: matchList is a set of the words or substrings which partially matches
        token,
       strMinPosition is the exactly word which exactly matches token,
       strMaxLength is the longest substring which matches token,
       newToken is a new token
n ← length(token)
if n > 2 then
    matchList ← findStrMatching(token, dictionary, standardList)
    strMinPosition ← findMinPositionStr(matchList, token)
    strMaxLength ← findMaxLengthStr(matchList, token)
    strMinPositionScore ← ScoreToken(strMinPosition)
    strMaxLengthScore ← ScoreToken(strMaxLength)
    if strMinPositionScore > strMaxLengthScore then
        keep ← strMinPosition
    end
    else
        keep ← strMaxLength
    end
    if length(keep) = 0 then
        keep ← findMatchAbbrList(token, abbreviationList)
    end
    left ← " "
    right ← " "
    if length(keep) ≥ 2 then
        i ← findFirstPosition(keep, token)
        if i > 0 then
            left ← token[0 : i − 1]
        end
        next ← length(left) + length(keep)
        if next != n then
            right ← token[next : n]
        end
    end
    newToken ← Sword(left) + " " + keep + " " + Sword(right)
end
else
    if token found in abbreviationList then
        newToken ← abbreviationList[token]
    end
    else
        newToken ← token
    end
end
return newToken
```

References

[1] N. Anquetil and T. Lethbridge. Extracting concepts from file names: a new file clustering criterion. In *Proceedings of the 20th international conference on Software engineering*, ICSE '98, pages 84–93. IEEE Computer Society, 1998.

[2] D. Binkley, M. Hearn, and D. Lawrie. Improving identifier informativeness using part of speech information. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 203–206. ACM, 2011.

[3] S. Butler. Mining Java class identifier naming conventions. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1641–1643. IEEE Press, 2012.

[4] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Improving the tokenisation of identifier names. In *Proceedings of the 25th European conference on Object-oriented programming*, ECOOP'11, pages 130–154. Springer-Verlag, 2011.

[5] B. Caprile and P. Tonella. Restructuring Program Identifier Names. In *Proceedings of the International Conference on Software Maintenance (ICSM'00)*, ICSM '00, pages 97–. IEEE Computer Society, 2000.

[6] W. B. Cavnar and J. M. Trenkle. N-Gram-Based Text Categorization. In *Symposium On Document Analysis and Information Retrieval*, pages 161–175, 1994.

[7] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwé. A qualitative comparison of three aspect mining techniques. In *IWPC*, pages 13–22. IEEE Computer Society, 2005.

[8] F. Deissenboeck and M. Pizka. Concise and consistent naming. *Software Quality Control*, 14(3):261–282, Sept. 2006.

[9] B. Dit, L. Guerrouj, D. Poshyvanyk, and G. Antoniol. Can Better Identifier Splitting Techniques Help Feature Location? In *ICPC*, pages 11–20. IEEE Computer Society, 2011.

[10] P. Domingos and M. Pazzani. On the Optimality of the Simple Bayesian Classifier under Zero-One Loss. *Machine Learning*, 29, 1997.

[11] S. T. Dumais and H. Chen. Hierarchical classification of Web content. In *SIGIR*, pages 256–263, 2000.

[12] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker. Mining source code to automatically split identifiers for software analysis. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, MSR '09, pages 71–80. IEEE Computer Society, 2009.

[13] J.-R. Falleri, M. Huchard, M. Lafourcade, C. Nebut, V. Prince, and M. Dao. Automatic Extraction of a WordNet-Like Identifier Network from Software. In *Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension*, ICPC '10, pages 4–13. IEEE Computer Society, 2010.

[14] T. Fawcett. An introduction to ROC analysis. *Pattern Recognition Letters*, 27(8):861–874, 2006. ROC Analysis in Pattern Recognition.

[15] H. B. D. Feild and D. Lawrie. Identifier Splitting: A Study of Two Tchniquese. In *Proceedings of MASPLAS'06 Mid-Atlantic Student Workshop on Programming Lanquages and Systems Rutgers University*, 2006.

[16] G. Guo, H. Wang, D. A. Bell, Y. Bi, and K. Greer. An kNN Model-Based Approach and Its Application in Text Categorization. In *CICLing'04*, pages 559–570, 2004.

[17] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: an update. *SIGKDD Explor. Newsl.*, 11(1):10–18, 2009.

[18] E. W. Høst and B. M. Østvold. Debugging Method Names. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 294–317. Springer-Verlag, 2009.

[19] C. X. Ling, J. Huang, and H. Zhang. AUC: a statistically consistent and more discriminating measure than accuracy. In *Proceedings of the 18th international joint conference on Artificial intelligence*, pages 519–524, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc.

[20] A. McCallum, R. Rosenfeld, T. M. Mitchell, and A. Y. Ng. Improving Text Classification by Shrinkage in a Hierarchy of Classes. In *Proc. of the int. conf. on Machine Learning*, pages 359–367, 1998.

[21] T. M. Mitchell. *Machine learning*. McGraw Hill, 1996.

[22] I. C. Mogotsi. Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze: Introduction to information retrieval. *Information Retrieval*, 13:252–253, 2010.

[23] F. Provost and T. Fawcett. Analysis and Visualization of Classifier Performance: Comparison under Imprecise Class and Cost Distributions. In *In Proceedings of the Third International Conference on Knowledge Discovery and Data Mining*, pages 43–48. AAAI Press, 1997.

[24] F. Sebastiani. Machine learning in automated text categorization. *ACM Computing Surveys*, 2002.

[25] P. Warintarawej, A. Laurent, P. Pompidor, A. Cassanas, and B. Laurent. Classifying Words: A Syllables-Based Model. In *Proceedings of the 2011 22nd International Workshop on Database and Expert Systems Applications*, DEXA '11, pages 208–212. IEEE Computer Society, 2011.

[26] P. Warintarawej, A. Laurent, P. Pompidor, and B. Laurent. Classification of brand names based on n-grams. In *SOCPAR'10*, pages 12–17, 2010.

[27] Y. Yang and X. Liu. A re-examination of text categorization methods. In *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '99, pages 42–49. ACM, 1999.

[28] Y. Yang and J. Pedersen. A comparative study on feature selection in text categorization. In *Proceedings of the Fourteenth International Conference on Machine Learning (ICML'97)*, pages 412–420. Morgan Kaufmann Publishers, 1997.