2014 IEEE Congress on Evolutionary Computation (CEC)
July 6-11, 2014, Beijing, China

Application of Computational Intelligence for Source Code Classification

Marcos Alvares and Tshilidzi Marwala
Faculty of Engineering and the Built Environment
University of Johannesburg
Gauteng, South Africa
Email: marcosj@student.uj.ac.za, tmarwala@uj.ac.za

Fernando Buarque de Lima Neto
Polytechnic School of Pernambuco
University of Pernambuco
Pernambuco, Brazil
Email: fbln@ecomp.poli.br

***Abstract*—Multi-language Source Code Management systems have been largely used to collaboratively manage software development projects. These systems represent a fundamental step in order to fully use communication enhancements by producing concrete value on the way people collaborate to produce more reliable computational systems. These systems evaluate results of analyses in order to organise and optimise source code. These analyses are strongly dependent on technologies (*i.e.* framework, programming language, libraries) each of them with their own characteristics and syntactic structure. To overcome such limitation, source code classification is an essential pre-processing step to identify which analyses should be evaluated. This paper introduces a new approach for generating content-based classifiers by using Evolutionary Algorithms. Experiments were performed on real world source code collected from more than 200 different open source projects. Results show us that our approach can be successfully used for creating more accurate source code classifiers. The resulting classifier is also expansible and flexible to new classification scenarios (opening perspectives for new technologies).**

I. Introduction

Internet has been helping speed up communication and connect people around the world. Distance is not an issue anymore for certain categories of business and technology is quickly changing the traditional model for assembling work teams. New possibilities of collaborative work raised and companies are hiring high skilled professionals for composing geographically distributed teams. These globalised and highly distributed work environment is a trend and companies are gradually adapting themselves to this new and more effective *modus operandi*.

Intuitively, one of the firsts areas to break through to the new paradigm of distributed work was Information Technology (especially Software Development). Developers around the world can collaboratively work to produce better computational systems by using online support systems through the global network. Web based Source Code Management (SCM) systems have been used as an essential resource for software development projects during the past 15 years [1] [1]–[3]. These systems have been used by thousand companies around the world for collaboratively organise software projects. Modern Source Code Management systems (like *Github*) have drawn attention due to their integration with Social Networking features, stimulating collaborative and highly distributed work scenarios [4].

An important feature of web based SCM systems is source code classification. This classification step is used for helping SCMs to organise source code and perform more specific analyses according to detected technologies [5]. The most popular SCM systems use file extension to determine programming language and a Bayesian classifier to solve ambiguities situations like "*.h*" files (header files from *C*, *CPP* or *Objective C* programming language). Two main libraries have been used by SCMs to classify source code: (*i*) Linguist and (*ii*) Pygments[2] [6], [7]. Both libraries use the same above mentioned approach to classify source code.

A big limitation of this extension based approach is the absence of lexical analysis for all classification situations. This approach supposes that all input files will always have an extension. This limitation restricts the classification process specially in case of code snips and *in-memory* code.

In fact, as showed in Code 1, the above mentioned library raises a *Software Exception* in this case and no classification is performed. Line 1 to line 5 prints the first 5 lines of a Ruby source code, on line 7 a file called *ga.rb* is copied to a new file called *ga* (no extension this time), lines 10 to 12 the file with extension is successfully classified and finally lines 13 to 17 the classification failed to analyse the file without extension.

Code 1. Linguist Library Raises an Exception

```
 1  mabj@Bazooka $ head −n5 ga.rb
 2  require './individual.rb'
 3  require './macros.rb'
 4
 5  if !File.exists?('config.yml')
 6
 7  mabj@Bazooka $ cp −af ga.rb ga
 8  mabj@Bazooka $ irb
 9  > require 'linguist'
10  > p = Linguist::FileBlob.new("ga.rb")
11  > p.language.name
12   => "Ruby"
13  > p = Linguist::FileBlob.new("ga")
```

[1]*e.g. GitHub*, *SourceForge*, *Launchpad* and *Gitorious*

[2]Pygments is a Syntax Highlight library. Once each programming language has a different set of keywords, this library classify the input source code before start the highlighting process.

```
> p.language.name
NoMethodError: undefined method 'name' for nil:NilClass
  from (irb):3
  from ruby−2.0.0/bin/irb:13:in '<main>'
```

A more recent application for source code classifiers is multi-language static analysers[3]. In this case, in order to perform technology specific analyses a classification process should be performed. Multi-purpose code analysers can be combined with SCM systems and Integrated Development Environments (IDEs) to automatically identify performance and security issues. This specific topic has earned attention and investment from all big software development companies around the world such as: Microsoft, Google, HP, and IBM. However public technologies show that a manual approach is still used for the initial classification process.

This paper presents an approach for automatically classifying source code by using Lexical Analysis, Scoring Strategies and Evolutionary Algorithm. The proposed strategy uses a Multi-Objective Genetic Algorithm to find most representative tokens[4] for each programming language in order to perform a more accurate lexical based classification analysis.

Results showed us that the proposed approach can overcome limitations of currently using technologies and effectively generating a low-error classifier for 8 different types of programming languages. The resulting classifier is also expansible and flexible to new classification scenarios (opening perspectives for new technologies).

This paper is divided into 4 more sections. Section II is an overview on the necessary background theory to fully understanding this paper. Section III describes the proposed method. Section IV presents experiment results. Finally, Section V summarises conclusions and discourse possible future work.

II. Theoretical Background

This section provides the necessary theoretical background and references for fully understanding the contribution presented in the next section. The next sub-sections covers the following topics:

1) Lexical analysis and keywords;
2) Scoring method;
3) Non-dominated Sorting Genetic Algorithm-II.

Next sub-sections can be read in any order. Readers already familiar with the above mentioned subjects are encouraged to skip or partially skip this session.

A. Lexical analysis and keywords

Lexical analyses is normally used as the first processing phase in a compiler [8]. Basically, this phase receives a stream of characters and returns a sequence of tokens of the form:

$$\langle token\_name, attribute\_value \rangle \tag{1}$$

Each identified token has its own specific meaning according to the programming language specification. Programming language tokens are composed by two main components: (*i*) "*token-name*" which is an abstract symbol and (*ii*) "*attribute-value*" which points to an entry in the symbol table[5].

For example, suppose a source program is composed of an assignment statement:

$$a = b + 20 \tag{2}$$

This should be mapped for a sequence of tokens similar to:

$$\langle id, 1\rangle\langle OpAssignStm\rangle\langle id, 2\rangle\langle OpPlusStm\rangle\langle 20\rangle \tag{3}$$

Each different language is defined by a lexical (*e.g.* set of tokens), syntactical (*e.g.* structural information) and semantical (*e.g.* meaning information) rules. For this research, we are interested in specific kind of tokens: *Keywords*. *Keywords* are reserved symbols which can not be used by the programmer as label for identifiers. For instance, Table I presents *Keywords* for the C programming language.

TABLE I
ANSI C Programming Language Keywords

union	unsigned	void	volatile	while
typedef	switch	struct	sizeof	static
signed	short	return	register	long
int	if	goto	for	float
do	extern	enum	else	double
default	continue	const	char	case

Lexical Analysis comprises the method proposed in this paper by extracting *Keywords* and using them as evidences to automatically determine which programming language is being used by a given source code.

B. Scoring method

Scoring strategies are mainly used for classification problems. It determines the probability of a given entity to belong to a previously specified class based on characteristics used as evidences. Scoring strategy, is used for composing the output classifier generated by the method proposed in this paper. This classifier calculates the probability of a given source code file belongs to a certain programming language by analysing the occurrence of *Keywords*.

Formally this is an application of comparisons between standard scores (one for each class of programming language). In statistics, the standard score is the number of standard deviations an *datum* is above the mean [9]. In our case the mean should be the average number of occurrences of keywords calculated from a set of samples for a certain class

[3]Examples private solutions of multi-language static analysers for identifying security issues are: HP Fortify Static Code Analyzer, Coverity SAVE, Veracode

[4]"*The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes. For each lexeme, the lexical analyzer produces as output a token*" (Section 1.2.1 Compilers: Principle, Techniques and Tools [8])

[5]The symbol table entry for an identifier holds information about the identifier, such as its name and type.

of programming language. The standard deviation is how far an analysed file is from this mean.

This method simplifies comparisons among standard deviations (of an analysed file) from means of different classes. By using the above mentioned scoring strategy, the method described in the next section can determine which programming language the analysed file has higher probability of belonging to.

C. Non-dominated Sorting Genetic Algorithm-II

NSGA-II is a fast and elitist multi-objective evolutionary algorithm developed in 2000 by Deb et al. at Kampur Genetic Algorithms Laboratory [10], [11]. After ten years, this algorithm remains the state-of-art for finding solutions to combinatorial and multi-objective problems. Over the years, NSGA-II has been used for finding solutions to problems in a large range of areas, such as: Economics and Engineering [12]–[15].

NSGA-II has features that set it apart from other techniques, like:

- the fast non-dominated sorting procedure is implemented[6];
- implements elitism for multi-objective search, using an elitism-preserving approach;
- a parameter-less diversity preservation mechanism is adopted;
- the constraint handling method does not make use of penalty parameters; and
- allows both continuous ("real-coded") and discrete ("binary-coded") design variables.

NSGA-II is shown in Algorithm 1 and is composed of three sub-algorithms: (*i*) fast non-dominated sorting algorithm, (*ii*) crowded distance assignment algorithm and the (*iii*) crowded comparison operator (see [11] for more details).

Algorithm 1 NSGA-II Algorithm.

```
Initialize randomly a population $P$ with $N$ individuals
Evaluate all the individuals of the population
Separate individuals in Pareto Fronts using dominance
repeat
  repeat
    Select parents using binary tournament
    Create a new individual using crossover and mutation
    Evaluate the fitness of the individual
    Add solution to the population
  until $N$ new individuals are created
  Separate individuals in Pareto Fronts using dominance
  Evaluate the crowding distance of each individual
  Discard worse individuals
until the maximum number of iterations is reached
```

Like any other genetic algorithm, NSGA-II has mutation and crossover parameters, however its fitness evaluation process is a composition of evaluations of target objectives. The fast non-dominated sort algorithm sorts the solution according to the level of non-domination.

[6]this algorithm sorts the individuals of a given population according to the level of non-domination.

The concept of *Dominance* is fundamental for the NSGA-II algorithm (represented by the symbol $\preceq$). A solution x_1 is said to dominate another solution x_2, and we write $x_1 \preceq x_2$, if both the following conditions are true:

- solution x_1 is no worse than x_2 in all objectives; and
- solution x_1 is strictly better than x_2 in at least one objective.

III. Contribution

This research presents an evolutionary approach to automatically generate more accurate *source code* classifiers. The generated classifiers are composed of a score mechanism and a list of keywords sets (one set for each programming language). These keywords sets are filtered by the evolutionary processes to keep only the most representative keywords for each programming language. The resulting list is used to compose the final classifier.

This section is composed by two sub-sections: the first one explains the used classifier and the second one explains how we generate a better classifier by using the evolutionary process.

A. Classifier

The proposed method aims to automatically generate a score-based source code classifier. Figure 1 presents the high level architecture for the proposed classifier.

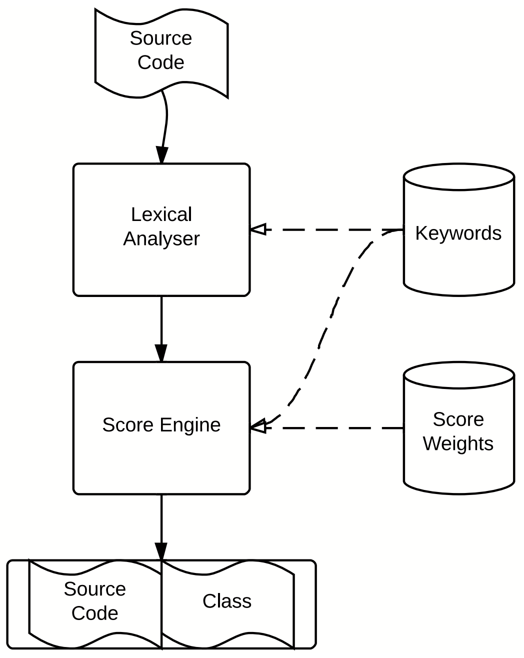

Fig. 1. Score-based source code classifier architecture

This architecture is composed of four main components:

- *Keywords database*: first input parameter, is composed by a set of keywords for each programming language;

- *Score weight vector*: second input parameter, is composed by weight values for each type of keyword;
- *General lexical analyser*: splits a given source code in lexical tokens. called "general" because it is not used specifically for only one programming language as a regular lexical analyser;
- *Score engine*: compares found *tokens* with the *keyword database* and gives scores to types of programming languages according the *score weight vector*.

For experiments performed at this article, in order to improve the generated classifier accuracy, we defined two different classes of keywords: (*i*) simple and (*ii*) unique. The score weight vector should have different weights for each class of keywords[7] once unique *keywords* intuitively is a stronger evidence for decide which programming language a source code belongs. We also give 10 points for the file extension. The value of these weight were determined empirically, however it can be included into the evolutionary process for an automatically setup approach (this approach directly impacts the size of the search space).

For accessing efficacy of a classifier (with a specific setup of keyword database and score weight vector), we used a set of test real world source code samples collected from open source projects. Samples used for composing the test set should be chosen carefully to really represent the programming languages. Two characteristics are very important for source files inside the test set:

- Coverage: data set for a specific programming language should have to cover all expected keywords;
- File size: source code files should have different sizes. From tiny source code with few lines to big ones with huge amount of code inside only one file.

The efficacy of the classifier is linked to the average classification error of source code files at the test set. For instance, if a hypothetic classifier supports 2 programming languages and it has 6% error for language "A" and 10% for language "B" the total error for this classifier will be 8%. By using this rule we can compare different setups of classifiers and decide which one is more accurate (according the used test set).

B. Evolutionary Process

So far the put forward classifier receives as input parameter a keyword database which contains a set of keywords for each class of programming language. The challenge tackled in this section is how to solve the best setup for the keyword database in order to achieve an error as small as possible. We have then to evaluate the average classification error for different setups of classifiers seeking for the one which returns the smallest error. This scenario characterises a multi-objective combinatorial problem.

NSGA-II Algorithm is well known of being suitable for such category of multi-objective problems (see section II). In line with that we applied this algorithm to automatically select subsets of the keyword database which better represents the classes of programming languages (again, according the used test set). This step is concerned with selecting the most relevant and representative keywords for each programming language.

The remainder of this section describes relevant aspects to the Evolutionary process such as:

1) search space;
2) individual;
3) fitness function; and
4) evolutionary operators.

In order to reduce the complexity and size of search space, once we have sets of raw keywords extracted from the documentation of each programming language, common keywords (keywords which appears in all sets) should be deleted. These keywords do not make any difference during the score-based classification process once every programming language score are equally incremented.

The search space is composed of combinations of all different setups of keywords inside each set (one set for each programming language). It increases exponentially every time a new programming language is added to the algorithm. For instance if our approach is searching for a solution (sets of keywords) for 5 programming languages with 10 keywords each our search space has 100.000 possible combinations. The size of the search space can be calculated as follows:

$$L = \{l | l \text{ is an analysed language}\} \tag{4}$$

$$K(l) = \{k | k \text{ is a keyword of language } l\} \tag{5}$$

$$S = \{K(l) | l \in L\} \tag{6}$$

$$f(S) = \prod_{i=1}^{|S|} (2^{|K(L_i)|} - 1) \tag{7}$$

Where "L" represents the set of classes for the generated classifier (possibilities of programming languages). $K(i)$ returns a set of keywords for a language "l". "S" represents the set of sets of keywords, one for each programming language inside "L". "$f(n)$" returns the size of our search space for a given "S".

Each combination of sets represents an individual. For example, for a search space composed by:

$$S = [[k_{11}, k_{12}, k_{13}], [k_{21}, k_{22}, k_{23}], [k_{31}, k_{32}]] \tag{8}$$

each element of this set represents a set of keywords for a programming language (indexes 1, 2 and 3). From this scenario, 147 different individuals can be generated, such as:

$$\begin{array}{l} i_1 = [[k_{11}], [k_{21}], [k_{31}, k_{32}]] \\ i_2 = [[k_{11}, k_{12}], [k_{21}], [k_{31}, k_{32}]] \\ i_3 = [[k_{11}, k_{12}], [k_{23}], [k_{31}, k_{32}]] \\ \ldots \end{array} \tag{9}$$

[7]After tedious ad doc testes, we used 1 for simple keywords and 5 for each unique keyword

Each individual represents a different setup for a classifier. The classifier uses the scoring approach described on Section II to decide which programming language the analysed file belongs to.

Each individual evaluate a classification over the test database (composed by samples of source code). Then, errors for each programming language and an average error should be calculated. The classification error for each programming language represents a component of the *objective space*. The evolutionary process aims to minimise classification errors and return a set of possible solutions (sets of keywords). In line with that, the fitness function used to evaluate each individual is represented by the composition of the classification error of the programming language. The multi-objective optimisation process can be represented by:

$$\begin{aligned} &min(l_1(i), l_2(i), l_3(i)...l_k(i)) \\ &i \in Population \end{aligned} \tag{10}$$

Where l_k represents the error function for the language with index k and i is an individual from population.

The NSGA-II algorithm is an elitist algorithm and preserves the best individuals of each generation. The crowding distance is also evaluated for each individual ensuring the diversity of non-dominated solutions and the generation of more uniform *paretos* along the evolutionary process. In spite of that, once we need only one classifier which gives us an equally distributed as good as possible classification accuracy for all programming language, only one solution from the final *pareto* should be selected. This solution should dominate others in more objectives as possible. [8]

As crossover operator we cross a random individual from the elitist group with each individual inside the population. For each crossover operation a proportion of "60%" of the characteristics from the fittest individual was used for compose the new individual. Intuitively, after every crossover operation, the proposed algorithm has to remove duplicated keywords inside each programming language keyword set.

As a mutation strategy we decided to add or retrieve keywords from each programming language keyword set according to its error. The number of new keywords added or removed (selected randomly from the initial set of keywords) is proportional to the accuracy of this objective. For instance if the error is "15%" the algorithm will add or remove this rate from the respective set of keywords.

We also use a strategy to avoid the algorithm to get stuck on local non-dominant *paretos* by randomising half of the population again if the algorithm do not present any improvement on the total error for certain amount of time (generations).

Figure 2 presents the final proposed process for obtaining the classifier. First step is to collect keyword lists for each programming language. Then use a database of training samples guide the Genetic Algorithm. The Genetic algorithm automatically provides a new filtered keyword database with the most relevant keywords sets according the provided training samples. Then a new keyword database is used to classify for testing the generated classifier.

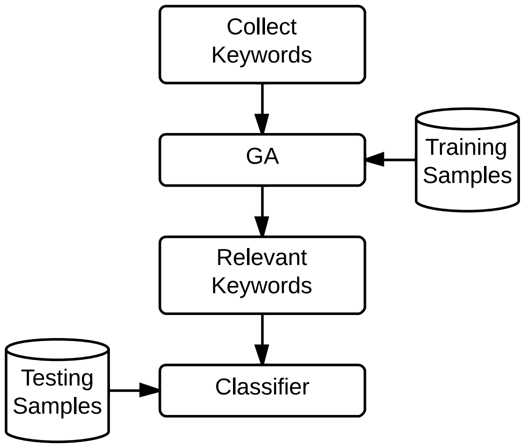

Fig. 2. Proposed algorithm flow for this step

The process have to be repeated for each new programming language appended to the classifier. Once the classifier does not have any information about this new programming language, the designer has to provide the correspondent keyword set and source code samples to the genetic algorithm and obtain a new optimised keyword database (which includes the new programming language).

IV. Experiment

This section exemplifies the proposed method for generating a classifier for 8 different programming languages: C, CPP, Visual Basic, C Sharp, PHP, Ruby, Python and Java. The generated classifier is used to classify a validation database containing inedited samples (not used during the generation phase).

Experiments were carried out on two machines: (*i*) Intel I7 3.4 GHz, 4 GB RAM and (*ii*) Intel I3 3.07 GHz, 2 GB RAM. Both machines use Linux Operating system Ubuntu 12.04. The prototype was implemented in Ruby programming language version $1.9.3p125$.

Table II shows the size for each set of keywords within the keywords database. These keywords sets were extracted from the official online documentation of each supported programming language [16]–[23].

Section III presents a mathematical tool for calculating the size of the search space according the keyword database. For these 8 programming languages a search space of 10^{142} different combinations can be generated (after removing the common keywords). [9] A search process in such big space

[8]The heuristic to select the fittest solution inside the final *pareto* was: *the fittest individual inside each pareto returned along the evolutionary process is the one who presents the smallest average classification error (arithmetic mean of all objectives).*

[9]Just for a perspective, this number is bigger than the estimated number of stars in our galaxy (300 billions).

TABLE II
NUMBER OF KEYWORD BY PROGRAMMING LANGUAGE

Language	(#)
C	32
CPP	84
Visual Basic	157
C Sharp	77
Java	50
Python	32
Ruby	35
PHP	65

should be guided by feedback information like the average classification error. A multi-objective Genetic Algorithm was applied to seek a combination of keywords which represents a small classification error.

Each individual in the GA population represents one possibility inside the search space composed by 10^{142} possibilities. We used 12 individuals for composing the GA population and as training and test datasets we used 3400 files from 50 different open source projects from Github. 1700 samples to evaluate the fitness of each individual (guided the evolutionary process) and 1700 different ones for testing the generated classifier.

Each fitness evaluation for each individual takes something around 30 minutes (approximately 1 second per file classification). This means that for evaluate the entire population takes 6 hours. We used concurrent computing techniques to try to reduce this evaluation time. The population was evaluated in 1 hour and 2 hours for the I7 and I3 configuration respectively. The stop condition is 500 iterations (500 generations). This means that the genetic algorithm will take at least 20 days (using the above mentioned hardware specification).

Table III shows scores used for each class of event found during the scoring process. Each simple keyword found scores 1 point for the respective programming language. Each unique keyword found registers 5 points and file extension registers 10 points for the respective programming language. Each programming language gets a probability and the one with the higher probability is returned as a class for the analysed source code.

TABLE III
SCORE TABLE

Event	Score
Simple Keyword	1
Unique Keyword	5
Extension	10

Figure 3 shows the evolution of the total error along Genetic Algorithm iterations. The average error decreases from 49.13% to 3.53% along 500 iterations. As explained at Section III The chart shows the fittest individual inside each *pareto* returned after each iteration. We need to select an individual inside the returned *pareto* which presents the smallest average classification error as possible.

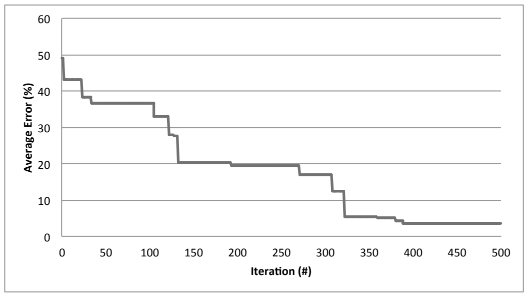

Fig. 3. Average error along Genetic Algorithm iterations

Table IV shows us the total amount of time for all 500 iterations on each environment. The first environment gave us a better classifier with 3.5% of total error. The setup used for this first experiment does not consider file extension. This scenario shows the accuracy for a classification strictly based on the content of the source code.

TABLE IV
TIME ANALYSIS (WITHOUT EXTENSION)

Configuration	Total Error	Total Time
I7	3.5%	21 days
I3	4.1%	33 days

As a second scenario the file extension was considered as evidence to determine the type of programming language of a given source code. According the file extension a programming language can score 10 points.

Table V shows the total error for the second scenario. The influence of file extension solved ambiguities problems and reduces the total error. In spite of using the file extension, the proposed approach remains most influenced by the content. This file extension information is especially useful for small source files with few content information (*keywords*).

TABLE V
TIME ANALYSIS (WITH EXTENSION)

Configuration	Total Error	Total Time
I7	1.1%	21 days
I3	1.3%	33 days

Code 2 presents a example of C source code classified as CCP by the setup shown at the first scenario. The classification error is solved at the second scenario by considering a weight for the file extension.

Code 2. Ambiguities during classification process
```
#include <stdlib.h>

#define RANDSRC "/dev/urandom"

int random_bytes(void *dst, size_t n) {
  FILE *f = fopen(RANDSRC, "rb");
  assert(f);
size_t r = fread(dst, n, 1, f);
fclose(f);
if (r < 1) { return 0;}
return 1;
}
```

Table VI shows the size of each keyword set for each programming language. This keyword database is used for composing the final classifier. We can observe that the new keyword list is around 40% smaller than the original one. Not relevant keywords were automatically pruned from the original keyword database by the evolutionary process without any human specialist assistance.

TABLE VI
NUMBER OF KEYWORD BY PROGRAMMING LANGUAGE

Language	(#)
C	31
CPP	56
Visual Basic	79
C Sharp	34
Java	34
Python	16
Ruby	23
PHP	51

Appendix A presents the generated keyword database. As commented in the last sections, every time a new programming language is added to the classifier the evolutionary process has to be executed again receiving a keyword set and source code samples for the new language.

V. CONCLUSION AND FUTURE WORK

This paper presents a systematic approach to generate *source code* classifiers by using evolutionary algorithms, lexical analysis and score strategies. The proposed approach is *parameter-less* once it requires only lists of keywords (one list for each programming language) to automatically generate classifiers. The generated classifier can be used for supporting Source Code Management, Multi-Language Static Analysers and Integrated Development Environments systems to become less *analyst-dependent* and perform more specific tasks.

The proposed approach can also be used to classify unfinished *source code* (once it does not require syntactic correctness) and support analyses for helping software development teams incrementally during the software development life cycle. The presented method is also flexible and does not require any specialised skills for adding support for a new programming language [10]. The only necessary task is to provide new keyword sets (one for each programming language) for the Genetic Algorithm and representative samples of *source code* for composing the training database.

The proposed method was validated by automatically generating a valid classifier for 8 different classes of programming languages. A database of 1700 samples retrieved by 200 different open source projects was used as training and testing sets for Genetic Algorithm. Results show that the generated classifier can be used to lexically classify real world *source code*.

As a future work we are planning to combine more analyses and heuristics to the proposed approach aiming to improve the classification overall accuracy, such as:

- Project analyses: the classifier should also consider the type of the *source code* files around (in the same directory) the one currently analysed in order to determine the programming language;
- Negative score: the score approach should also penalise the classification process, for instance: a C programming language should not have a keyword labeled "*class*" then the score algorithm should penalise the score for this language;
- Score value: the value used as score for keywords can also compose the evolutionary process.
- Classes of relevance: the algorithm can automatically determine the weight for different classes of keyword through the evolutionary process (*e.g.* different weights for unique and repeated keywords).
- Principal Component Analyses: Comparing the evolutionary approach with a combination of Principal Component Analyses for each keyword set (for each programming language).

[10]Keywords are normally well documented at any programming language specifications.

REFERENCES

[1] L. Augustin, D. Bressler, and G. Smith, "Accelerating software development through collaboration," *International Conference on Software Engineering*, 2002.

[2] R. English and C. Schweik, "Identifying success and tragedy of FLOSS commons: A preliminary classification of Sourceforge. net projects," *National Center for Digital Government Working Paper Series*, vol. 7, 2007.

[3] A. Begel, J. Bosch, and M. Storey, "Social Networking Meets Software Development: Perspectives from GitHub, MSDN, Stack Exchange, and TopCoder," *Software, IEEE*, vol. 30, 2013.

[4] F. Thung and T. Bissyandé, "Network Structure of Social Coding in GitHub," *European Conference on Software Maintenance and Reengineering*, 2013.

[5] P. Lerthathairat and N. Prompoon, "An approach for source code classification to enhance maintainability," *International Joint Conference on Computer Science and Software Engineering (JCSSE)*, vol. 7, no. 002, 2011.

[6] Linguist, "Linguist: source code classifier," 2013. [Online]. Available: https://github.com/github/linguist

[7] Pygments, "Pygments: source code classifier," 2013. [Online]. Available: http://pygments.org/

[8] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[9] R. J. Larsen and M. L. Marx, *Introduction to Mathematical Statistics and Its Applications (5th Edition)*. Pearson, 2011.

[10] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, "A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II," in *International Conference on Parallel Problem Solving from Nature*. Springer, 2000, pp. 849–858.

[11] K. Deb and A. Pratap, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, 2002.

[12] R. T. F. A. King, H. C. S. Rughooputh, and K. Deb, "Solving the multiobjective environmental/economic dispatch problem with prohibited operating zones using NSGA-II," in *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*. IEEE, Aug. 2011, pp. 298–303.

[13] X. Jiang, C. Yongqiang, W. Xiaoqing, Z. Minhui, and X. Liu, "Optimum design of antenna pattern for spaceborne SAR performance using improved NSGA-II," in *2007 IEEE International Geoscience and Remote Sensing Symposium*. IEEE, 2007, pp. 615–618.

[14] P. Murugan, S. Kannan, and S. Baskar, "Application of NSGA-II Algorithm to Single-Objective Transmission Constrained Generation Expansion Planning," *IEEE Transactions on Power Systems*, vol. 24, no. 4, pp. 1790–1797, Nov. 2009.

[15] S. Mishra, G. Panda, S. Meher, R. Majhi, and M. Singh, "Portfolio management assessment by four multiobjective optimization algorithm," in *IEEE Recent Advances in Intelligent Computational Systems*. IEEE, Sep. 2011, pp. 326–331.

[16] D. A. Black, "Ruby Programming Language Keywords," 2009. [Online]. Available: http://ruby-doc.org/docs/keywords/1.9/

[17] R. Moore, "Python Programming Language Keywords," 2005. [Online]. Available: http://docs.python.org/release/2.3.5/ref/keywords.html

[18] M. Achour, "PHP Programming Language Keywords," 2013. [Online]. Available: http://php.net/manual/en/reserved.keywords.php

[19] V. Studio, "C# Programming Language Keywords," 2013. [Online]. Available: http://msdn.microsoft.com/en-us/library/x53a06bb.aspx

[20] ——, "Visual Basic Programming Language Keywords," 2008. [Online]. Available: http://msdn.microsoft.com/en-us/library/ksh7h19t(v=vs.90).aspx

[21] Comunity, "C Programming Language Keywords," 2013. [Online]. Available: http://en.wikipedia.org/wiki/C_syntax\#Reserved_keywords

[22] ——, "C++ Programming Language Keywords," 2013. [Online]. Available: http://en.cppreference.com/w/cpp/keyword

[23] Oracle, "Java Programming Language Keywords," 2013. [Online]. Available: http://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html

Appendix A
Relevant Keywords database

- **C**: enum, #else, do, restrict, float, case, #if, unsigned, while, extern, for, sizeof, switch, if, continue, struct, char, typedef, register, default, goto, volatile, void, const, double, signed, inline, #elif, break, #endif, else, auto, #include;
- **CPP**: for, template , goto, constexpr, char16_t, case, override, operator, float, alignas, register, dynamic_cast, mutable, union, decltype, static_cast, catch, sizeof, export, xor_eq, while, alignof, static_assert, #else, signed, struct, wchar_t, not_eq, virtual, short, volatile, asm, #if, break, nullptr, if, auto, final, bitand, double, and_eq, thread_local, bitor, enum, default, throw, try, else, reinterpret_cast, xor, friend, explicit, typedef, namespace, #include;
- **Visual Basic**: wend, shared, narrowing, mybase, myclass, mustoverride, module, optional, widening, Variant, if, step, addressof, interface, integer, namespace, property, trycast, overloads, cchar, ushort, byref, clng, sbyte, catch, implements, when, alias, notoverridable, sub, cuint, function, throw, rem, endif, culng, cbool, synclock, handles, inherits, cushort, gosub, single, redim, operator, imports, erase, shadows, partial, cshort, byte, nothing, elseif, getxmlnamespace, stop, enum, withevents, andalso, delegate, cint, object, mod, csng, cobj, ctype, finally, raiseevent, notinheritable, addhandler, goto, const, option, case, loop, me, removehandler, else, try;
- **C Sharp**: extern, if, interface, implicit, operator, override, goto, var, using, fixed, checked, internal, ulong, sbyte, explicit, typeof, sizeof, for, while, else, double, enum, base, case, unsafe, stackalloc, readonly, void, delegate, short, foreach, catch, float;
- **Ruby**: when, undef, break, for, elsif, else, begin, until, alias, nil, def, case, yield, retry, then, ensure, super, do, module, end;
- **Python**: exec, break, for, as, global, from, nonlocal, None, def, else, import, assert, except;
- **PHP**: goto, explode, enddeclare, throw, switch, final, case, implode, use, break, include_once, then, global, for, endswitch, implements, endfor, function, instanceof, clone, try, as, include, extends, do, const, require, __halt_compiler, elseif, eval, declare, echo, catch, callable, else, die, namespace, default, insteadof, endwhile, endif, mysql_query, if, unset, continue, endforeach, trait;
- **Java**: throws, assert, volatile, double, break, transient, while, import, throw, continue, float, for, native, if, default, strictfp, finally, char, final, enum, instanceof, else, catch, try, interface, case, synchronized, package, super, do, byte, implements, const, extends.