UAM@SOCO 2014: Detection of Source Code Re-use by means of Combining Different Types of Representations

A. Ramírez-de-la-Cruz, G. Ramírez-de-la-Rosa, C. Sánchez-Sánchez, W. A. Luna-Ramírez, H. Jiménez-Salazar and C. Rodríguez-Lucatero

Presenter: Esaú Villatoro-Tello

December 5th, Bangalore, India
SOCO Task Description

- SOCO, Detection of SOuRce COde Re-use, is a shared task that focuses on monolingual source code re-use detection.
- Participant systems were provided with sets of source codes (training and test) in C and Java programming languages.
- The task consists on retrieving the source code pairs that have been re-use at a document level.
Our general idea

- Different and diverse views of a source code allow a richer description of it
- Each view should highlight different aspects of a source code
Proposed Source Code Representations

From three views we proposed four representations:

- **Lexical View:**
  - Character 3-grams

- **Structural View:**
  - Data types from the function’s signature
  - Names from the function’s signatures

- **Stylistic View:**
  - 11 stylistic features to represent each source code
Code Examples

Code 1: Calculator.c ($C_\alpha$)

```c
#include <stdio.h>

int sum(int numOne, int numTwo) {
    return numOne + numTwo;
}

int substraction(int numOne, int numTwo) {
    return numOne - numTwo;
}

int multiplication(int numOne, int numTwo) {
    return numOne * numTwo;
}

int main(int argc, char** argv) {
    int numOne = 10;
    int numTwo = 15;
    int res_Sum = sum(numOne, numTwo);
    return 0;
}
```

Code 2: Calc.c ($C_\beta$)

```c
#include <stdio.h>

int add(int numX, int numY) {
    int res = numX + numY;
    return res;
}

int sub(int numX, int numY) {
    int res = numX - numY;
    return res;
}

int main(int argc, char** argv) {
    int numX = 10;
    int numY = 15;
    int res_Add = 0;
    res_Add = add(numX, numY);
    return 0;
}
```
Proposed Source Code Representations

- **Lexical View:**
  - **Character 3-grams**
  - Structural View from function's signatures
    - Data types
    - Names of function and arguments.

- **Stylistic View:**
  - 11 stylistic features to represent each source code
Lexical view

- Idea: Similar to text documents, we want to find pattern similarities within the source code by means of 3-grams of characters
  - We use the method proposed by Enrique Flores* plus we eliminated reserve words of the programming language

Lexical View

Example for code $C_2$:

```c
definitions: C2

#include <stdio.h>

int add(int numX, int numY)
{
    int res = numX + numY;
    return res;
}

int sub(int numX, int numY)
{
    int res = numX - numY;
    return res;
}

int main(int argc, char** argv) {
    int numX = 10;
    int numY = 15;
    int res_Add = 0;
    res_Add = add(numX, numY);
    return 0;
}
```

Bag of 3-grams:

```
{"std", "tdi", "dio", "ioh", "oha", "had", "add", "ddn", "dnu", "num", "umx", ...}
```
Lexical View: source code comparison

Then each 3-gram Bag is represented as a vector $\mathbf{C}_\alpha$ and $\mathbf{C}_\beta$.

\[
\begin{align*}
\mathbf{B}_\alpha &= \{"std", "tdi", "dio", "ioh", "oha", "had", "add", "ddn", "dnu", "num", "umx", \ldots, "my0"\} \\
\mathbf{C}_\alpha &= \begin{bmatrix}
    0 & 0 & 1 & 0 & 0 & 1 & 1 & 2 & 8 & 0 & 16 & 0 & 1 & 8 & 1 & 3 & 1 & 2 & 8 & 0 & \ldots
\end{bmatrix} \\
\mathbf{B}_\beta &= \{"std", "tdi", "dio", "ioh", "ohn", "hsu", "sum", "umn", "mnu", "num", "umo", "mon", "one", \ldots, "wo0"\} \\
\mathbf{C}_\beta &= \begin{bmatrix}
    4 & 2 & 1 & 2 & 1 & 0 & 1 & 0 & 0 & 1 & 12 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 6 & \ldots
\end{bmatrix}
\end{align*}
\]
Finally, the similarity between a pair of source codes is computed using the cosine similarity, which is defined as follows:

\[ \text{sim}_{\text{3grams}}(C_\alpha, C_\beta) = \cos(\theta) = \frac{C_\alpha \cdot C_\beta}{||C_\alpha|| ||C_\beta||} \] (1)
Proposed Source Code Representations

- **Lexical View:**
  - Character 3-grams

- **Structural View from function’s signatures**
  - Data types
  - Names of function and arguments.

- **Stylistic View:**
  - 11 stylistic features to represent each source code
Structural view

- Idea: Some structure can be present in the function’s signature of source code
  - We used the function’s signatures in two ways
    - Data types
    - Names of function and arguments
Our intuition: plagiarists often are willing to change function's and argument's names, but not the data types of such elements.

```c
#include <stdio.h>

int add(int numX, int numY)
{        
    int res=numX+numY;  
    return res;        
}

int sub(int numX, int numY)
{        
    int res=numX-numY;  
    return res;        
}

int main(int argc, char** argv) {
    int numX=10;        
    int numY=15;        
    int res_Add=0;      
    res_Add=add(numX,numY);  
    return 0;
}
```

int add(int numX, int numY)
Int sub(int numX, int numY)

Only function's signatures without the main method
Structural View: Data types

- A real example (part 1)

A function on source code 077.c

```c
void runBruteForce(const char chSet[], int numOfCh, int len, CrackFuncPtr func, int sCh, int eCh, int eC, int eD);```

A function on source code 078.c

```c
void runDictCrack(const ListPtr l, CrackFuncPtr func);```

Only data types without return type

```
077.C = [char, int, int, CrackFuncPtr, int, int, int]
078.C = [ListPtr, CrackFuncPtr]
```

Use only the intersection

```
DatatypeSet = [int, char, CrackFuncPtr, ListPtr]
```
For each method of the two source code in analysis, we count the frequency of each data type and then we compute the similarity as:

\[ \text{Sim}_a(\text{metodo}1^{077.c}, \text{metodo}2^{078.c}) = \frac{1}{8} \]

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>char</th>
<th>CrackFuncPtr</th>
<th>ListPtr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Método 1</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Método 2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ \text{sim}_a(\mathbf{m}_1^{\alpha}, \mathbf{m}_2^{\beta}) = \frac{\sum_{i=0}^{n} \min(\mathbf{m}_1^{\alpha}, \mathbf{m}_2^{\beta})}{\sum_{i=0}^{n} \max(\mathbf{m}_1^{\alpha}, \mathbf{m}_2^{\beta})} \]
Structural View: Data types

- A real example (part 2)

A function on source code 077.c

```c
void runBruteForce(const char chSet[], int numOfCh, int len, CrackFuncPtr func, int sCh, int eCh, int &)
```

A function on source code 078.c

```c
void runDictCrack(const ListPtr l, CrackFuncPtr func);
```

We compare only the return data type

\[ \text{Sim}_r(\text{metodo1}^{077.c}, \text{metodo2}^{078.c}) = 0 \]
A real example (combining part 1 and part 2)

\[ \text{Sim}_r(\text{metodo1077.c, metodo2078.c}) = 0 \]
\[ \text{Sim}_a(\text{metodo1077.c, metodo2078.c}) = 1/8 \]

The combined similarity gives us the structural similarity of data types

\[ \text{sim}_1(m^\alpha, m^\beta) = \sigma \times \text{Sim}_r(m^\alpha, m^\beta) + (1 - \sigma) \times \text{Sim}_a(m^\alpha, m^\beta) \]

In this work \( \sigma = 0.5 \)

\[ \text{Sim}(\text{metodo1077.c, metodo2078.c}) = (0.5 \times 0) + (0.5 \times 0.125) = 0.0625 \]
Finally, given 2 codes, $C_\alpha$ and $C_\beta$, we compute the similarity of data types of all the functions in both codes:

\[
\begin{array}{c|c|c}
\text{Sim}(m_1^\alpha, m_1^\beta) & \ldots & \text{Sim}(m_1^\alpha, m_j^\beta) \\
\text{Sim}(m_2^\alpha, m_1^\beta) & \ldots & \text{Sim}(m_2^\alpha, m_j^\beta) \\
\vdots & \ddots & \vdots \\
\text{Sim}(m_i^\alpha, m_1^\beta) & \ldots & \text{Sim}(m_i^\alpha, m_j^\beta) \\
\end{array}
\]

\[
sim_{DataTypes}(C_\alpha, C_\beta) = f(M_{\alpha, \beta}^{type})
\]
Proposed Source Code Representations

- **Lexical View:**
  - Character 3-grams

- **Structural View from function’s signatures**
  - Data types
  - Names of function and arguments.

- **Stylistic View:**
  - 11 stylistic features to represent each source code
Structural View: Names of functions and arguments

- Our intuition: some plagiarists might try to obfuscate the copied elements by means of changing data types, but not the variable’s names.

```c
#include <stdio.h>

int add(int numX, int numY)
{
    int res=numX+numY;
    return res;
}

int sub(int numX, int numY)
{
    int res=numX-numY;
    return res;
}

int main(int argc, char** argv) {
    int numX=10;
    int numY=15;
    int res_Add=0;
    res_Add=add(numX,numY);
    return 0;
}
```

```
int add(int numX, int numY)
Int sub(int numX, int numY)
```

Only function’s signatures without the `main` method
### Structural View: Names of functions and arguments

- A real example

#### Function on source code 078.c

```c
void runDictCrack(const ListPtr l, CrackFuncPtr func);
```

1. Concatenate all names to form a single string
   - 078.C = rundictcracklfunc

2. A set of 3-grams of characters are extracted
   - 3gramsSet_078 = [run', 'und', 'ndi', 'dic', 'ict', 'ctc', 'tcr', 'cra', 'rac', 'ack', 'ckl', 'klf', 'lfu', 'fum', 'unc']

#### A function on source code 077.c

```c
```

Same process is applying to other methods.
Once we have computed the bag of n-grams, we can compute how similar are two functions, using the Jaccard coefficient as follows:

\[
\text{Sim}_2(\text{3gramsSet}_078, \text{3gramsSet}_078) = \frac{\text{m}^\alpha \cap \text{m}^\beta}{\text{m}^\alpha \cup \text{m}^\beta}
\]

\[
\text{Sim}_2(\text{3gramsSet}_078, \text{3gramsSet}_078) = 3/49
\]
Finally, given 2 codes, $C_\alpha$ and $C_\beta$, we compute the similarity of names of all the functions in both codes:

$$\text{Sim}(m_{\alpha 1}, m_{\beta 1}) \quad \ldots \quad \text{Sim}(m_{\alpha 1}, m_{\beta j})$$

$$\text{Sim}(m_{\alpha 2}, m_{\beta 1}) \quad \ldots \quad \text{Sim}(m_{\alpha 2}, m_{\beta j})$$

$$\ldots \quad \ldots \quad \ldots$$

$$\text{Sim}(m_{\alpha i}, m_{\beta 1}) \quad \ldots \quad \text{Sim}(m_{\alpha i}, m_{\beta j})$$

$$\sim_{Names}(C_\alpha, C_\beta) = f(M_{\alpha, \beta}^{names})$$
Proposed Source Code Representations

- **Lexical View:**
  - Character 3-grams

- **Structural View from function’s signatures**
  - Data types
  - Names of function and arguments.

- **Stylistic View:**
  - 11 stylistic features to represent each source code
Stylistic View

- This representation aims at finding unique properties from the original author such as his/her programming style.
  - we compute 11 stylistic features to represent each source code.
  - Then, we use a vector representation and by using a cosine similarity we found the similarities between two source code.
Stylistic View: 11 stylistic features

The features are:

```c
#include <stdio.h>

int add(int numX, int numY)
{
    int res = numX + numY;
    return res;
}

int sub(int numX, int numY)
{
    int res = numX - numY;
    return res;
}

int main(int argc, char** argv)
{
    int numX = 10;
    int numY = 15;
    int res_Add = 0;
    res_Add = add(numX, numY);
    return 0;
}
```

C_{\beta}
Stylistic View: 11 stylistic features

- The features are:

```c
#include <stdio.h>

int add(int numX, int numY)
{
    int res=numX+numY;
    return res;
}

int sub(int numX, int numY)
{
    int res=numX-numY;
    return res;
}

int main(int argc, char** argv) {
    int numX=10;
    int numY=15;
    int res_Add=0;
    res_Add=add(numX,numY);
    return 0;
}
```

- **#White spaces**
- **#Code Lines**
Stylistic View: 11 stylistic features

- The features are:

```
#include <stdio.h>

int add(int numX, int numY) {
    int res=numX+numY;
    return res;
}

int sub(int numX, int numY) {
    int res=numX-numY;
    return res;
}

int main(int argc, char** argv) {
    int numX=10;
    int numY=15;
    int res_Add=0;
    res_Add=add(numX,numY);
    return 0;
}
```
Stylistic View: 11 stylistic features

The features are:

- Code Lines
- Tabulations
- White spaces
- Empty Lines

C_β
Stylistic View: 11 stylistic features

The features are:

- #Code Lines
- #Tabulations
- #White spaces
- #Functions
- #Empty Lines

```c
#include <stdio.h>

int add(int numX, int numY) {
    int res = numX + numY;
    return res;
}

int sub(int numX, int numY) {
    int res = numX - numY;
    return res;
}

int main(int argc, char** argv) {
    int numX = 10;
    int numY = 15;
    int res_add = 0;
    res_add = add(numX, numY);
    return 0;
}
```
Stylistic View: 11 stylistic features

The features are:

- Code Lines
- Tabulations
- White spaces
- Functions
- Empty Lines
- Average Word Length
Stylistic View: 11 stylistic features

- The features are:
  - Code Lines
  - Tabulations
  - White spaces
  - Functions
  - Empty Lines
  - Average Word Length
  - Uppercase

32
Stylistic View: 11 stylistic features

The features are:

- Code Lines
- Tabulations
- White spaces
- Functions
- Empty Lines
- Average Word Length
- Uppercase
- Lowercase

C_\beta
Stylistic View: 11 stylistic features

- The features are:
  - Code Lines
  - Tabulations
  - White spaces
  - Functions
  - Empty Lines
  - Average Word Length
  - Uppercase
  - Lowercase
  - Under scores

$C_\beta$
Stylistic View: 11 stylistic features

The features are:
- #Code Lines
- #Tabulations
- #White spaces
- #Functions
- #Empty Lines
- Average Word Length
- #Uppercase
- #Lowercase
- #Under scores
- Total of Words
- \( C_B \)
Stylistic View: 11 stylistic features

- The features are:
  - Code Lines
  - Total of Words
  - Functions
  - Empty Lines
  - Average Word Length
  - Uppercase
  - Lowercase
  - Under scores
  - White spaces
  - Tabulations
  - Code Lines
  - Total of Words
  - Functions
  - Empty Lines
  - Average Word Length
  - Uppercase
  - Lowercase
  - Under scores
  - White spaces
  - Tabulations

Lexical Richness

36
Experimental Evaluation

- The evaluation was performed with the training provided by the shared task.
- The performance was measured for each proposed representation by means of establishing a manual threshold for considering when two codes are re-used.
- That threshold was set from 10 to 90 percent of similarity. For each threshold we evaluated the precision, recall, and F-measure.
- That information helped us to design the three uploaded runs.
Submitted Runs

We submitted three runs for the task based on three combinations of the proposed representations.

**Run 1. Lexical View Only**

The results for C and Java are shown in table 1.

<table>
<thead>
<tr>
<th></th>
<th>Precision</th>
<th>Recall</th>
<th>F-measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0.006</td>
<td>1.00</td>
<td>0.013</td>
</tr>
<tr>
<td>Java</td>
<td>0.349</td>
<td>1.00</td>
<td>0.517</td>
</tr>
</tbody>
</table>
Submit 2. Combination of Lexical and Structural Views

The linear combination is shown in the next equation:

\[ \text{sim} = (0.5 \times \text{LexSim}) + (0.25 \times \text{DTSim}) + (0.25 \times \text{NameSim}) \]  
(7)

The results of the experiment are shown below:

<table>
<thead>
<tr>
<th></th>
<th>Precision</th>
<th>Recall</th>
<th>F-measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0.005</td>
<td>0.950</td>
<td>0.010</td>
</tr>
<tr>
<td>Java</td>
<td>0.019</td>
<td>0.928</td>
<td>0.037</td>
</tr>
</tbody>
</table>
Submitted Runs

- **Run 3.** Supervised approach.

For this experiment all the similarities, from all the views, were computed using a J48 decision tree.

The obtained results are in the next table:

<table>
<thead>
<tr>
<th></th>
<th>Precision</th>
<th>Recall</th>
<th>F-measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0.006</td>
<td>0.997</td>
<td>0.013</td>
</tr>
<tr>
<td>Java</td>
<td>0.691</td>
<td>0.968</td>
<td>0.807</td>
</tr>
</tbody>
</table>
Submited Runs

As we can see our obtained recall value for detecting source code re-use in C are competitive with the recall of the best system (1.00 and 0.997).

The opposite happened with the performances for Java. Here our system performs very well, in recall as well as in precision values, which put our system at the first place in the performance’s ranking.

<table>
<thead>
<tr>
<th></th>
<th>Precision</th>
<th>Recall</th>
<th>F-measure</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>C Language</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Our run 3</td>
<td>0.006</td>
<td>0.997</td>
<td>0.013</td>
</tr>
<tr>
<td>Best system</td>
<td>0.282</td>
<td>1.00</td>
<td>0.440</td>
</tr>
<tr>
<td>Baseline</td>
<td>0.258</td>
<td>0.345</td>
<td>0.295</td>
</tr>
<tr>
<td><strong>Java Language</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Our run 3</td>
<td>0.691</td>
<td>0.968</td>
<td>0.807</td>
</tr>
<tr>
<td>The 2nd best system</td>
<td>0.530</td>
<td>0.995</td>
<td>0.692</td>
</tr>
<tr>
<td>Baseline</td>
<td>0.457</td>
<td>0.712</td>
<td>0.556</td>
</tr>
</tbody>
</table>
Conclusions and Future Work

- From the obtained results during the training phase
  - each type of representation provide some information that can be used to detect some particular cases of source code re-use.
  - It is needed a deeper analysis in order to determine the main characteristics that improve the code re-use detection.

- We believe that the low precision values (processing C codes) are due to the fact that several source codes are not just in pure C, and instead, also C/C++ alike programs.
  - We also need to do a deeper analysis to validate this hypothesis

- Finally, obtained results during the test phase motivate us to keep working on the same direction.
Thank you
Our Group

Language and Reasoning Group
Information Technologies Department
UAM-C

Follow us on Twitter @LyR_UAMC

Corresponding author of this work: Gabriela Ramírez (gramirez@correo.cua.uam.mx)
Evaluation in the training set

Lexical view

Stylistic view

Figure 1: Lexical view. Best result is obtained with the 80% of similarity between two methods.

Figure 2: Stylistic view. A high recall is obtained for every similarity threshold, but also very low precision.
Evalulation in the training set

Structural view: data type

Figure 3: Structural view: data type of function's signatures using the maximum value of similarities between functions. Best result is obtained with more than 90% of similarity between two methods.

Figure 4: Structural view: data type of function's signatures using the average value of similarities between functions. Best result is obtained with 80% of similarity between two methods.
Evaluation in the training set

Structural view: names

Figure 5: Structural view: identifiers of function’s signatures using the maximum value of similarities between functions. Best result is obtained with 40% of similarity between two methods.

Figure 6: Structural view: identifiers of function’s signatures using the average value of similarities between functions. Best result is obtained with more than 20% of similarity between two methods.