STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LII**, Number 1, 2007

# AN ANALYSIS OF DISTANCE METRICS FOR CLUSTERING BASED IMPROVEMENT OF SYSTEMS DESIGN

ISTVÁN GERGELY CZIBULA AND GABRIELA ŞERBAN

ABSTRACT. *Clustering* is the process of grouping a set of objects into classes of similar objects. *Refactoring* is the process of improving the design of software systems.It improves the internal structure of the system, but without altering the external behavior of the code ([5]). In [1] we have proposed a new approach for improving systems design using *clustering*. The aim of this paper is to analyze the influence of distance metrics for clustering based improvement of systems design. We are focussing on identifying the most suitable distance metric. The study is made on the clustering based approach developed in [1].

Keywords: software engineering, refactoring, clustering, distance metrics.

## 1. Introduction

The software systems, during their life cycle, are faced with new requirements. These new requirements imply updates in the software systems structure, that have to be done quickly, due to tight schedules which appear in real life software development process. That is why continuous restructurings of the code are needed, otherwise the system becomes difficult to understand and change, and therefore it is often costly to maintain.

Refactoring is a solution adopted by most modern software development methodologies (extreme programming and other agile methodologies), in order to keep the software structure clean and easy to maintain. Thus, refactoring becomes an integral part of the software development cycle: developers alternate between adding new tests and functionality and refactoring the code to improve its internal consistency and clarity.

In [5], Fowler defines refactoring as "the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes

Received by the editors: January 15, 2007.

2000 *Mathematics Subject Classification.* 68N99, 62H30.

1998 *CR Categories and Descriptors.* D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement –*Restructuring, reverse engineering, and reengineering*; I.5.3 [**Computing Methodologies**]: Pattern Recognition – *Clustering*.

the chances of introducing bugs". Refactoring is viewed as a way to improve the design of the code after it has been written. Software developers have to identify parts of code having a negative impact on the system's maintainability, and apply appropriate refactorings in order to remove the so called "bad-smells" ([10]).

### 1.1. **Related Work.**

There are various approaches in the literature in the field of *refactoring*, but just a few of them use *clustering* in order to restructure programs. In [2] a clustering based approach for program restructuring at the functional level is presented. This approach focuses on automated support for identifying ill-structured or low cohesive functions. The paper [11] presents a quantitative approach based on clustering techniques for software architecture restructuring and reengineering as well for software architecture recovery. It focuses on system decomposition into subsystems.

We have developed, in [1], a *k-means* based clustering approach for identifying refactorings in order to improve the structure of software systems. For this purpose, *kRED* (k-means for REfactorings Determination) algorithm is introduced.

In this paper we study the influence of distance metrics on the results obtained by *kRED* algorithm. We intend to identify the most suitable distance metric using the open source case study JHotDraw ([14]).

The rest of the paper is structured as follows. The approach (*CARD*) proposed in [1] for determining refactorings using a clustering technique is presented in Section 2. Section 3 provides an experimental evaluation of *CARD*, based on different distance metrics and using the open source case study JHotDraw ([14]). Some conclusions and further work are outlined in Section 4.

## 2. Refactorings Determination using a Clustering Approach

In this section we briefly describe the clustering approach (*CARD*), introduced in [1], that aims at finding adequate refactorings in order to improve the structure of software systems.

*CARD* approach consists of three steps:

- **Data collection** - The existent software system is analyzed in order to extract from it the relevant entities: classes, methods, attributes and the existent relationships between them.
- **Grouping** - The set of entities extracted at the previous step are re-grouped in clusters using a *k-means* based clustering algorithm, *kRED* ([1]). The goal of this step is to obtain an improved structure of the existing software system.
- **Refactorings extraction** - The newly obtained software structure is compared with the original software structure in order to provide a list of refactorings which transform the original structure into an improved one.

2.1. **Theoretical model.** We have introduced in [1] a theoretical model on which *CARD* approach is based on. In the following we will briefly describe this model.

Let $S = \{s_1, s_2, ..., s_n\}$ be a software system, where $s_i, 1 \leq i \leq n$ can be an application class, a method from a class or an attribute from a class.

We will consider that:

- $Class(S) = \{C_1, C_2, \dots, C_l\}$, $Class(S) \subset S$, is the set of applications classes in the initial structure of the software system $S$.
- Each application class $C_i$ $(1 \leq i \leq l)$ is a set of methods and attributes, i.e., $C_i = \{m_{i1}, m_{i2}, \dots, m_{ip_i}, a_{i1}, a_{i2}, \dots, a_{ir_i}\}$, $1 \leq p_i \leq n$, $1 \leq r_i \leq n$, where $m_{ij}$ $(\forall j,\ 1 \leq j \leq p_i)$ are methods and $a_{ik}$ $(\forall k,\ 1 \leq k \leq r_i)$ are attributes from $C_i$.
- $Meth(S) = \bigcup\limits_{i=1}^{l} \bigcup\limits_{j=1}^{p_i} m_{ij}$, $Meth(S) \subset S$, is the set of methods from all the application classes of the software system $S$.
- $Attr(S) = \bigcup\limits_{i=1}^{l} \bigcup\limits_{j=1}^{r_i} a_{ij}$, $Attr(S) \subset S$, is the set of attributes from the application classes of the software system $S$.

Based on the above notations, the software system $S$ is defined as in Equation (1):

$$S = Class(S) \bigcup Meth(S) \bigcup Attr(S). \tag{1}$$

As described above, at the **Grouping** step of our approach, the software system $S$ has to be re-grouped. In our view, this re-grouping is represented as a ***partition*** of $S$.

**Definition 1.** *(*[1] *)**Partition of a software system S.***
*The set $\mathcal{K} = \{K_1, K_2, ..., K_v\}$ is called a* **partition** *of the software system $S = \{s_1, s_2, \dots, s_n\}$ iff*

- $1 \leq v \leq n$;
- $K_i \subseteq S, K_i \neq \emptyset, \forall i,\ 1 \leq i \leq v$;
- $S = \bigcup\limits_{i=1}^{v} K_i$ *and* $K_i \cap K_j = \emptyset,\ \forall i, j, 1 \leq i, j \leq v, i \neq j$.

In the following, we will refer to $K_i$ as the $i$-th *cluster* of $\mathcal{K}$, to $\mathcal{K}$ as a *set of clusters* and to an element $s_i$ from $S$ as an *entity*.

A cluster $K_i$ from the partition $\mathcal{K}$ represents an application class in the new structure of the software system.

2.2. **kRED algorithm.** In [1], based on the theoretical model described in Subsection 2.1, a *k-means* based clustering algorithm (*kRED*) is introduced. The algorithm is used in the *Grouping* step of *CARD*, and aims at identifying a **partition** of a software system $S$ that corresponds to an improved structure of it.

*kRED* is a vector space model based *clustering* algorithm, that is used in order to re-group entities from the software system.

In *CARD* approach ([1]), the objects to be clustered are the entities from the software system $S$, i.e., $\mathcal{O} = \{s_1, s_2, \dots, s_n\}$ and the attribute set is the set of application classes from the software system $S$, $\mathcal{A} = \{C_1, C_2, \dots, C_l\}$.

The focus is to group similar entities from $S$ in order to obtain high cohesive groups (clusters), that is why is considered the dissimilarity degree between the entities and the application classes $C$ from $S$, $\forall C,\ C \in Class(S)$.

Consequently, each entity $s_i$ $(1 \le i \le n)$ from the software system $S$ is characterized by a $l$-dimensional vector: $(s_{i1}, s_{i2}, \dots, s_{il})$, where $s_{ij}$ $(\forall j,\ 1 \le j \le l)$ is computed as follows ([1]):

$$s_{ij} = \begin{cases} -\frac{|p(s_i) \cap p(C_j)|}{|p(s_i) \cup p(C_j)|} & if \quad p(s_i) \cap p(C_j) \neq \emptyset \\ \infty & otherwise \end{cases}, \tag{2}$$

where, for a given entity $e \in S$, $p(e)$ defines a set of relevant properties of $e$, expressed as:

- If $e \in Attr(S)$ ($e$ is an attribute) then $p(e)$ consists of: the attribute itself, the application class where the attribute is defined, and all methods from $Meth(S)$ that access the attribute.
- If $e \in Meth(S)$ ($e$ is a method) then $p(e)$ consists of: the method itself, the application class where the method is defined, and all attributes from $Attr(S)$ accessed by the method.
- If $e \in Class(S)$ ($e$ is an application class) then $p(e)$ consists of: the application class itself, and all attributes and methods defined in the class.

A more detailed justification of the vector space model choice is given in [1].

As in a vector space model based clustering ([8]), we consider the *distance* between two entities $s_i$ and $s_j$ from the software system $\mathcal{S}$ as a measure of dissimilarity between their corresponding vectors. We will consider in our study three possible distance metrics between methods:

- *Euclidian Distance.* The distance between $s_i$ and $s_j$ is expressed as:

$$d_E(s_i, s_j) = \sqrt{\sum_{k=1}^{l} (s_{ik} - s_{jk})^2} \tag{3}$$

- *Manhattan Distance.* The distance between $s_i$ and $s_j$ is expressed as:

$$d_M(s_i, s_j) = \sum_{k=1}^{l} |s_{ik} - s_{jk}| \tag{4}$$

- *Cosine Distance.* The distance between $s_i$ and $s_j$ is expressed as:

$$d_C(s_i, s_j) = \frac{\sqrt{\sum_{k=1}^{l} s_{ik}^2} \cdot \sqrt{\sum_{k=1}^{l} s_{jk}^2}}{\sum_{k=1}^{l} (s_{ik} \cdot s_{jk})}. \tag{5}$$

The main idea of *kRED* algorithm is the following ([1]):

(i) The initial number of clusters is the number $l$ of application classes from the software system $\mathcal{S}$.
(ii) The initial centroids are chosen as the application classes from $\mathcal{S}$.
(iii) As in the classical *k-means* approach, the clusters (centroids) are recalculated, i.e., each object is assigned to the closest cluster (centroid).
(iv) Step (iii) is repeatedly performed until two consecutive iterations remain unchanged, or the number of steps performed exceeds the maximum number of iterations allowed.

In the following we intend to analyze the influence of the distance metrics described above on the results obtained by *kRED* algorithm.

## 3. Experimental Evaluation

For our analysis, we will consider two evaluations, which are described in Subsections 3.1 and 3.2. We will evaluate the results obtained by applying *kRED* algorithm for the distance metrics defined in Subsection 2.2.

### 3.1. Code Refactoring Examples.

3.1.1. *Example 1.* We aim at studying how the *Move Method* refactoring is obtained after applying *kRED* algorithm, for the analyzed distance metrics.

Let us consider the Java code example shown in Figure 1.

Analyzing the code presented in Figure 1, it is obvious that the method **methodB1()** has to belong to **class_A**, because it uses features of **class_A** only. Thus, the refactoring *Move Method* should be applied to this method.

We have applied *kRED* algorithm ([1]), for *Euclidian distance* and *Manhattan distance*, and the *Move Method* refactoring for **methodB1()** was determined. The two obtained clusters are:

- Cluster 1:
  {**Class_A, methodA1(), methodA2(), methodA3(), methodB1(), attributeA1, attributeA2**}.
- Cluster 2:
  {**Class_B, methodB2(), methodB3(), attributeB1, attributeB2**}.

```
public class Class_A {
 public static int attributeA1;
 public static int attributeA2;

 public static void methodA1(){
   attributeA1 = 0;
   methodA2();
 }

 public static void methodA2(){
     attributeA2 = 0;
     attributeA1 = 0;
  }

 public static void methodA3(){
     attributeA2 = 0;
     attributeA1 = 0;
     methodA1();
     methodA2();
 }
}
```

```
public class Class_B {
 private static int attributeB1;
 private static int attributeB2;

 public static void methodB1(){
     Class_A.attributeA1=0;
     Class_A.attributeA2=0;
     Class_A.methodA1();
  }

 public static void methodB2(){
     attributeB1=0;
     attributeB2=0;
  }

 public static void methodB3(){
     attributeB1=0;
     methodB1();
     methodB2();
 }
}
```

FIGURE 1. Code example for *Move Method* refactoring

The first cluster corresponds to application class **Class_A** and the second cluster corresponds to application class **Class_B** in the new structure of the system.

For *Cosine distance*, the *Move Method* refactoring for **methodB1()** is not identified.

3.1.2. *Example 2.* We aim to analyze how the *Move Attribute* refactoring is obtained after applying *kRED* algorithm, for the studied distance metrics. Let us consider the Java code example shown in Figure 2.

Analyzing the code presented in Figure 2, it is obvious that the attribute **attributeA1** has to belong to **class_B**, because is mostly used by methods from **class_B**. Thus, the refactoring *Move Attribute* should be applied to this attribute.

We have applied *kRED* algorithm for *Euclidian distance* and *Manhattan distance*, and the *Move Attribute* refactoring for **attributeA1** was determined.

For each distance metric, the two obtained clusters are:

```
public class Class_A {

  public static int attributeA2;
  public static int attributeA1;

  public static void methodA1() {
    methodA2();
  }

  public static void methodA2() {
    attributeA2 = 0;
    Class_A.attributeA1 = 12;
  }

  public static void methodA3() {
    attributeA2 = 0;
    methodA1();
    methodA2();
  }
}

public class Class_B {

  private static int attributeB1;
  private static int attributeB2;

  public static void methodB1() {
    attributeB1 = 0;
    Class_A.methodA1();
  }

  public static void methodB2() {
    attributeB1 = 0;
    attributeB2 = 0;
    Class_A.attributeA1 = 12;
  }

  public static void methodB3() {
    attributeB1 = 0;
    methodB1();
    methodB2();
    Class_A.attributeA1 = 12;
  }

  public static void methodB4() {
    attributeB1 = 0;
    methodB2();
    Class_A.attributeA1 = 12;
  }
}
```

FIGURE 2. Code example for *Move Attribute* refactoring

- Cluster 1:
  {**Class_A, methodA1(), methodA2(), methodA3(), attributeA2**}.
- Cluster 2:
  {**Class_B, methodB1(), methodB2(), methodB3(), methodB4(), attributeA1, attributeB1, attributeB2**}.

The first cluster corresponds to application class **Class_A** and the second cluster corresponds to application class **Class_B** in the new structure of the system.

For *Cosine distance*, the *Move Attribute* refactoring for **attributeA1** is not identified.

From the examples in Subsection 3.1 we can conclude, experimentally, that *Euclidian distance* and *Manhattan distance* are the most appropriate distance metrics.

3.2. **JHotDraw Case Study.** Our second analysis is made on the open source software JHotDraw, version 5.1 ([14]). It is a Java GUI framework for technical and structured graphics, developed by Erich Gamma and Thomas Eggenschwiler, as a design exercise for using design patterns.

The reason for choosing JHotDraw as a case study is that it is well-known as a good example for the use of design patterns and as a good design.

In order to test the accuracy of *CARD* approach, two measures *ACC* and *PREC* were introduced in [1]. These measures indicate how accurate are the results obtained after applying *kRED* algorithm in comparison to the current design of JHotDraw. We assume that $\mathcal{K} = \{K_1, \ldots K_p\}$ is a partition reported after applying *kRED* algorithm.

**Definition 2.** *(*[1]*)**ACCuracy of a refactoring technique - ACC.***

*Let $\mathcal{T}$ be a refactoring technique.*

*The accuracy of $\mathcal{T}$ with respect to a partition $\mathcal{K}$ and the software system $S$, denoted by $ACC(S, \mathcal{K}, \mathcal{T})$, is defined as:*

$$ACC(S, \mathcal{K}, \mathcal{T}) = \frac{1}{l}\sum_{i=1}^{l} acc(C_i, \mathcal{K}, \mathcal{T}).$$

$acc(C_i, \mathcal{K}, \mathcal{T}) = \dfrac{\sum\limits_{j \in \mathcal{M}_{C_i}} \dfrac{|C_i \cap K_j|}{|C_i \cup K_j|}}{|\mathcal{M}_{C_i}|}$ *(where $\mathcal{M}_{C_i} = \{j | \ 1 \leq j \leq p, \ |C_i \cap K_j| \neq 0\}$ is the set of clusters from $\mathcal{K}$ that contain elements from the application class $C_i$ ), is the accuracy of $\mathcal{T}$ with respect to the application class $C_i$.*

*ACC* defines the degree to which the partition $\mathcal{K}$ is similar to $S$. Based on Definition 2, it can be proved that larger values for *ACC* indicate better partitions with respect to $S$, meaning that *ACC* has to be maximized.

**Definition 3.** [1] ***PRECision of a refactoring technique - PREC.***

*Let $\mathcal{T}$ be a refactoring technique.*

*The precision of methods dicovery in $\mathcal{T}$ with respect to a partition $\mathcal{K}$ and the software system $S$, denoted by $PREC(S, \mathcal{K}, \mathcal{T})$, is defined as:*

$$PREC(S, \mathcal{K}, \mathcal{T}) = \frac{1}{|Meth(S)|}\sum_{m \in Meth(S)} prec(m, \mathcal{K}, \mathcal{T}).$$

$prec(m, \mathcal{K}, \mathcal{T}) = \begin{cases} 1 & \textit{if m was placed in the same class as in S} \\ 0 & \textit{otherwise} \end{cases}$ *,is the precision of $\mathcal{T}$ with respect to the method $m$.*

$PREC(S, \mathcal{K}, \mathcal{T})$ defines the percentage of methods from $S$ that were correctly discovered by $\mathcal{T}$ (we say that a method is correctly discovered if it is placed in its original application class). Based on Definition 3, it can be proved that larger values for $PRECM$ indicate better partitions with respect to $S$, meaning that $PREC$ has to be maximized.

After applying *kRED* algorithm for JHotDraw case study, for the considered distance metrics, we obtain the results described in Table 1.

| Distance metric | ACC | PREC |
|---|---|---|
| **Euclidian distance** | **0.9829** | **0.997** |
| **Manhattan distance** | **0.9721** | **0.9949** |
| **Cosine distance** | **0.9829** | **0.997** |

Table 1. The measures values.

The results from Table 1 show that the results obtained for *Euclidian distance* and *Cosine distance* are the best, because provide the largest values for $ACC$ and $PREC$.

As a conclusion, from the results obtained in Subsections 3.1 and 3.2 we can conclude, experimentally, that *Euclidian distance* is the most suitable distance metric to be used in *kRED* algorithm.

## 4. Conclusions and Future Work

We have analyzed in this paper the influence of distance metrics for clustering based improvement of systems design. We have comparatively present the results obtained by *kRED* algorithm ([1]) for different distance metrics. In order to evaluate the obtained results, we have used two quality measures defined in [1].

Based on the obtained results, we can conclude, experimentally, that *Euclidian distance* is the most suitable distance metric to be used in *kRED* algorithm.

Further work can be done in the following directions:

- To apply *kRED* for other case studies, like JEdit ([3]).
- To use other approaches for clustering, such as hierarchical clustering ([8]), search based clustering ([7]), or genetic clustering ([13]).
- To improve the vector space model used for clustering.

## References

[1] Czibula, I.G., Serban, G.: Improving Systems Design Using a Clustering Approach. IJCSNS International Journal of Computer Science and Network Security, VOL.6, No.12 (2006) 40–49

[2] Xu, X., Lung, C.H., Zaman, M., Srinivasan, A.: Program Restructuring Through Clustering Technique. In: 4th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2004), USA (2004) 75–84

[3] jEdit Programmer's Text Editor: http://www.jedit.org (2002)

[4] Han, J., Kamber, M.: Data Mining: Concepts and Techniques. Morgan Kaufmann Publishers (2001)

[5] Fowler, M.: Improving the design of existing code. Addison-Wesley, New-York (1999)

[6] Simon, F., Loffler, S., Lewerentz, C.: Distance based cohesion measuring. In Proceedings of the 2nd European Software Measurement Conference (FESMA) 99, Technologisch Instituut Amsterdam (1999)

[7] Doval, D., Mancoridis, S., Mitchell, B.S.: Automatic clustering of software systems using a genetic algorithm. IEEE Proceedings of the 1999 Int. Conf. on Software Tools and Engineering Practice STEP'99 (1999)

[8] Jain, A., Murty, M.N., Flynn, P.: Data clustering: A review. ACM Computing Surveys **31** (1999) 264–323

[9] Bieman, J.M., Kang, B.-K.: Measuring Design-Level Cohesion. In: IEEE Transactions on Software Engineering **24** No. 2 (1998)

[10] McCormick, H., Malveau, R.: Antipatterns: Refactoring Software, Architectures, and Projects in Crises. John Wiley and Sons (1998)

[11] Lung, C.H.: Software Architecture Recovery and Restructuring through Clustering Techniques. ISAW3, Orlando, SUA (1998) 101–104

[12] Jain, A., Dubes, R.: Algorithms for Clustering Data. Prentice Hall, Englewood Cliffs, New Jersey (1998)

[13] Cole, R.M.: Clustering with genetic algorithms. Master's thesis, University of Western Australia (1998)

[14] JHotDraw Project: http://sourceforge.net/projects/jhotdraw (1997)

[15] Chidamber, S., Kemerer, C.: A metrics suite for object oriented design. Im: IEEE Transactions on Softwareengineering **20** No. 6 (1994) 476–493

Department of Computer Science, Babeş-Bolyai University, 1, M. Kogalniceanu Street, Cluj-Napoca, Romania,
*E-mail address*: `istvanc@cs.ubbcluj.ro`

Department of Computer Science, Babeş-Bolyai University 1, M. Kogalniceanu Street, Cluj-Napoca, Romania,
*E-mail address*: `gabis@cs.ubbcluj.ro`