International Journal of Innovative
Computing, Information and Control
Volume **10**, Number **4**, August **2014**

ICIC International ©2014 ISSN 1349-4198
pp. 1519–1531

DISTANCE METRIC BASED DIVERGENT CHANGE BAD SMELL DETECTION AND REFACTORING SCHEME ANALYSIS

Dexun Jiang, Peijun Ma, Xiaohong Su and Tiantian Wang

School of Computer Science and Technology
Harbin Institute of Technology
No. 92, West Dazhi Street, Harbin 150001, P. R. China
silverghost192@163.com

Received July 2013; revised January 2014

Abstract. *Bad smells are signs of potential problems in codes. Bad smells decrease the design quality of software, so the codes are hard to analyze, understand, test or reuse. Divergent Change is a common and classical bad smell in object oriented programs. The detection of this bad smell is difficult, because the features of Divergent Change are not obvious, and the detecting and refactoring of this bad smell are on the later steps of software life cycle. In this paper, the detection method of Divergent Change bad smell based on distance metric and K-nearest neighbor clustering technology is proposed. The features of Divergent Change are analyzed and transformed to distances between entities. Divergent Change bad smells are detected with the results of K-nearest neighbor clustering, and targeted refactoring schemes are provided. After comparisons with similar researches, the experiments results on open source programs show that the proposed method behaves well on refactoring evaluation with low time complexity.*
Keywords: Entity distance, K-nearest neighbor clustering, Bad smell detection, Refactoring scheme

1. **Introduction.** The quality of software determines the difficulty level of maintenance and reuse of software. High quality programs are easy to understand, analyze, modify, test, maintain and reuse.

Bad smells [1] are signs of potential problems in codes. It leads to difficulty of understanding and modifying on programs. If too much bad smells exist in programs, the quality of the software would be very low. Refactoring [2-5] is a programming technique for optimizing the structure or pattern of an existing body of code by altering its internal nonfunctional attributes without changing its external behavior. Refactoring is needed towards the bad smells for quality improvement.

Divergent Change is a kind of classical bad smells. When the request of software functions changes, the codes are modified in different classes, or part of one class responses this kind of changing. Thus, the codes are pungent, and it is called Divergent Change.

Divergent Change bad smell causes programs modification and reuse more complex and difficult. When current codes should be modified from changing requests, more than one place should be modified to deal with single changing request.

In addition, Divergent Change bad smell causes developer hard to read and understand the codes in the process of programs modification and reusage. Programs maintainers usually have difficulty in understanding why codes are modified in different places from just single changing request. Actually, it means there would be certain relationships among these places. However, these relationships are hard to find with current static analysis.

Furthermore, after the function requests are changed, the problems above just are exposed. The bad smell is found in the maintain step of software life cycle, so the negative impact of the bad smell is larger, and the cost for improvement is higher.

Therefore, Divergent Change bad smell should be found earlier. If Divergent Change can be detected based on static analysis before using the codes and also can be removed with reasonable refactoring, the negative impact of the smell and cost for improvement would be decreased. The static analysis means analyzing the codes without compiling and running.

In fact, there is much difficulty in Divergent Change static analysis and detection. First, the expression of this bad smell is found with the change of request, but the changing occurs sporadically. So the bad smell cannot be found directly by static analysis. Second, Divergent Change is defined while using the software, and the judging of the smell needs setting the threshold manually. So the detection is subjective. Finally, the corresponding refactoring operations should be proposed after bad smell detection.

In object oriented programs, a class is composed of attributes and methods. These attributes and methods are called entities [6]. A method can invoke attributes or methods in location class or other classes, and this is called the dependency relationship between entities.

In this paper, the characteristics of Divergent Change bad smell are analyzed, and bad smell expression is transformed to dependency relationship between entities, and then the relationships are measured by entities distance value. Finally, the bad smell detection algorithm based on K-nearest neighbor clustering is executed to get the detecting results and corresponding refactoring.

The rest of the paper is organized as follows. Section 2 presents a short overview of related work. In Section 3, Divergent Change bad smell is analyzed and the smell appearance is transformed to dependency relationships between entities. Entities distance value is defined and computed to refer to the relationships in Section 4. K-nearest neighbor clustering is applied to the judgment about Divergent Change detection and how to refactoring in Section 5. And Section 6 shows the experiment results. The conclusion is provided in Section 7.

2. **Related Work.**

2.1. **Definition and analysis of divergent change bad smell.**

M. Fowler et al. [1] are the first to describe the Divergent Change bad smell, but they do not give strict definition. M. Fowler et al. pointed out Divergent Change was "one class is commonly changed in different ways for different reasons". However, the programs with correct design were that "Any change to handle a variation should change a single class, and all the typing in the new class should express the variation". With this the Divergent Change is presented. Then M. Fowler et al. suggested refactoring operation Extract Class to improve this bad smell. However, M. Fowler et al. did not give the specific characteristics, detecting methods and refactoring procedure, so Divergent Change is just a concept.

M. Mäntylä [7] classified Divergent Change as concealed smells, for the smell cannot be detected by simple glance to the code. Nor can they be well detected by tools. Mika mean detecting these two smells requires good understanding of the program and even experience in implementing these kinds of changes to the programs source code. Meanwhile, Walter and Pietrzak [8] located Divergent Change as a kind of maintenance smells. The changes cannot be detected with analysis of a single piece of code, so the detection needed the comparison of subsequent code versions.

2.2. **Metrics based divergent change bad smell detection.** Reddy and Rao [9,10] detected Divergent Change with dependency oriented complexity metrics. The average coupling metric value between one class and the other classes is computed to detect Divergent Change bad smell. This method had lower computation complexity.

Sant et al. [11] extracted some relationships based metrics for bad smell detection such as Divergent Change. The problems about design were explored through abstracting the mapping between design and codes. With this method, the existence of bad smells can be found, but the smells cannot be located.

De Lucia et al. [12] proposed a novel approach to guide the Extract Class refactoring, taking into account structural and semantic cohesion metrics. The method is based on the name text of classes and entities. However, the detecting features and methods of Divergent Change are not mentioned in this research. Furthermore, the metric is just from text, and gets less information from codes, so miss detection is higher.

2.3. **Clustering based bad smell detection.** Lung and Zaman [13] divided lower cohesion function into several higher cohesion functions. With this, the cohesion degree of functions would be increased. The relationship degrees between statements were computed for clustering. The clustering results are less accurate, so Lung et al. got the improvement in paper [14]. The basic process is: the source codes are analyzed to extract variables information; new relation degree computing method is proposed to build the variables matrix; clustering algorithm is executed for bad smells detection. However, in this method the control statements scope is not considered, and there is no corresponding response for nested control statements.

Alkhalid et al. [15] proposed an adaptive K-nearest neighbor clustering algorithm to apply in the refactoring process of Java code. Actually, the disadvantage of this method is also the lack of response for control dependency, including the scope and nested relationships of control statements. The clustering results do not reflect the dependency relationships in programs, so the detection is not accurate. And the specific refactoring schemas are not provided.

The researches above about bad smells detection with clustering technology all need manual threshold. Manual thresholds come from empirical data, and are subjective.

Srinivas [16] achieved the refactoring in object oriented programs package level with K-nearest neighbor clustering technology. Srinivas pointed out that the classes have higher cohesion in a package and lower coupling across the packages. In this research K-nearest neighbor clustering algorithm is compared with other related clustering such as SLINK, CLINK and WPGM, and the advantages of K algorithm are analyzed.

Ratzinger et al. [17] proposed a bad smell prediction technology based on programs development history records. The programs scale, developer, modification time, frequency, programming habits and other information are collected from version control system as the data sources for clustering. Decision trees, logistic model trees, propositional rule learners, and nearest neighbor technology can be used for prediction. However, authors do not realize these clustering algorithms, so there is no way for comparison.

2.4. **Deficiency of existing studies.** The definition and description of Divergent Change bad smell in almost all of research papers are from paper [1]. Usually the expression and features of Divergent Change are from developers experiments, and there is no strict definition. So this bad smell detection is reliant on manual reorganization of developers. It is the difficulty of Divergent Change bad smell detection automatically.

In the smell detection studies with metrics, the chosen metrics cannot fully reflect all the useful characteristics about this bad smell. Some are just from code text; some are

collected from single class, and the relationships between classes are not considered; some do not consider the attributes and methods in classes.

In the smell detection studies based on clustering technology, the threshold is necessary for smells detection. Manual thresholds will decrease the objectivity of bad smell detection results. The K-nearest neighbor technology does not need thresholds, but the value of K is preset. K can be generated dynamically. In paper [18] the value of K is from fixed area. And in paper [19] K value choosing method is proposed to deal with imbalance of sample number. In this paper, the K-nearest neighbor clustering algorithm based on dynamical K generation is proposed to detect Divergent Change bad smell, which addresses these limitations above.

3. **Description and Analysis of Divergent Change Bad Smell.** Divergent Change is introduced first by M. Fowler et al. in their book [1]. M. Fowler et al. believed that the out expressions of Divergent Change are: the request variation changes part of a class; different request variations change different places of the class; both of the above. M. Fowler et al. insisted that in common programs with well design, any change to handle a variation should change a single class, and all the typing in the new class should express the variation. The Divergent Change is just the opposite. The attributes and methods are collectively called entities. They are the components of a class.

From the description of Divergent Change, the smell is found after coding is finished and the software is put into use. Just when the request of software varies and the codes need to be modified, the pungencies will be found.

So the expressions of "part of a class change" or "different places of a class change" should be analyzed to get the internal regulations. The results of request changes are parts of a class change. If parts of the entities in a class always change together, or some entities always change together with entities in other classes, then there should be certain relationships between these "change together" entities.

For example, there are two attributes expiration and remaining life for one product. When the quality of the product increases (request change), the expiration will be extended, and the same as remaining life. After analysis, the remaining life is computed with production year, current year and expiration. So the remaining life depends on expiration. In programs implement, remaining life is computed through the invoking of expiration values.

Thus, the substance of "change together" is the dependency relationships between entities. If the dependency relationship occurs, the request variation may change the entities together. And if there is no dependency relationship between them, the request variation is not bound to change the entities together.

Return to the example. The unit of remaining life is year. If the quality of product increases less, the increment of expiration is just several months, and the remaining life does not change. Meanwhile, another attribute weight of the product does not change by the variation of quality, for there is no dependency relationship between them.

The sample codes about entities dependency relationships are shown in Figure 1. Dependency relationships are the invoking between entities, including methods invoke attributes and methods invoke methods. The entities invoking can be implemented in two ways: 1) instantiate a class, and invoke its attributes or methods, as shown in Figure 1(a); 2) the class is used as method parameter, and its attributes or methods can be invoked directly in the method, as shown in Figure 1(b). The relationships between entities are composed of direct dependency and indirect dependency. If entity A invokes entity B, it is direct dependency between A and B. And if entity A and C both invoke B, it is indirect dependency relationship between A and C. It is shown in Figure 1.

```
class class_A
{
    public static void mA1()
    {
        class_B b=new class_B();
        b.aB1=0;
    }
    public static void mA2(class_C c)
    {
        c.aC1=0;
    }
```

a

```
class class_C
{
    static int aC1;
    public static void mA1(class_B b)
    {
        b.aB1=0;
    }
    public static void mA3()
    {
        this.mA2();
    }
```

b

FIGURE 1. Sample codes of entities dependency relationships

Therefore, the relationships inner and intra classes can be measured by dependency statistics between entities. If dependencies are more, the relationships between entities are closer, and the probability of changing together is larger.

4. **Entities Dependency Relationships Description Based on Distance Theory.** In this paper, the dependency relationships between entities of object oriented programs are described by distance theory.

4.1. **Definition of similarity and distance.** The work with distance [20] is strongly connected with the theory of similarity/dissimilarity: one characteristic of a grouping might be that all things within one group are similar and all pairs of elements in different groups are dissimilar. The basic principle of object oriented programs is "put data and its options together", so the similar attributes and methods should be encapsulated as a class. In more detail knowing that two given things are similar is not enough. We should know the "degrees of similarity". The same holds for dissimilarity.

What makes two things degree similar or dissimilar? To work on this problem it is helpful to loot at Bung's ontology [21]. All things have properties, even if we are ignorant of this fact. The attribution of a property to a thing is a cognitive act, so it is possible that things have properties which are not considered as a property of a thing. Bunge's consideration indicates that the similarity between two things is the collection of their shared properties. When examining two things on their similarity without a special respect which leads direct to the conclusion, that all things are similar in some respect. So one quantitative concept of degree of similarity for a special set of aspects can be defined:

Definition 4.1. *(Degree of Similarity [21]): the degree of similarity between two things x, y relative to a finite subset B of all properties P_i is*

$$S(x,y) = \frac{|p(x) \cap p(y) \cap B|}{|(p(x) \cup p(y)) \cap B|} \tag{1}$$

Suppose, $p(x)$ be the properties set which have relationship with x, and $|p(x)|$ be the element number of $p(x)$.

Definition 4.2. *(Distance): the distance between two things x, y is defined as:*

$$D(x,y) = 1 - S(x,y) \tag{2}$$

The value scope of distance is $[0,1]$. The distance between one object and itself is 0. And if two objects have no same attributes, their distance is 1.

4.2. **Quantitative expression of entities dependency relationships.** The distance definition is used in the expression of entities dependency relationships to compute the distance between two entities in object oriented programs, as discussed in [6]. Entity a is one of the entities in class A, including the attributes and methods. The dependency properties set $P(a)$ of entity a is:

$$P(a) = \{a, \text{ entities invoke } a, \text{ entities } a \text{ invoke}\}$$

When a is an attribute, a cannot invoke others, but a can be invoked by other methods.

If there is direct dependency between a and b, $b \in P(a)$ or $a \in P(b)$. If indirect dependency, there is at least one entity c that $c \in P(a)$ and $b \in P(b)$. c can be in the same class with a or b, or c can be in other class. There can be both direct and indirect dependency relationships between two entities.

With the relationships above, distance between entities can be computed by Equations (1) and (2). The distance value between entities quantificationally represents the situation of dependency relationships, also the probability degree of entities changing together.

5. **Divergent Change Bad Smell Detection Based on *K*-nearest Neighbor Clustering.** If the distance of two entities is smaller, the dependency relationship between them is closer. If requests change the probability of changing together is higher, and vice versa.

When the number of entities in a class is large, the bad smell detection process is difficult and complex. In this paper K-nearest neighbor clustering technology is used for Divergent Change bad smell detection. Furthermore, refactoring schemes can be provided from the detecting results.

The Nearest Neighbor algorithm is a common classification algorithm based on sample learning. K-nearest neighbor rule is an extension of the nearest neighbor rule. This rule is that the note x is classified as the category with the most appearance in K nearest neighbor points. The algorithm is executed from point x, and the scope is extended until containing all the points in the same sample. Thus the sort of point x is the sort of points with the highest frequency in this extension. The probability of choosing sort ω_m is

$$\sum_{i=(K+1)/2}^{K} \binom{K}{i} P(\omega_i|x)^i [1 - P(\omega_m|x)]^{K-i} \tag{3}$$

Normally if K is larger, the probability of choosing sort ω_m is higher.

5.1. **Algorithm flow.** In this paper, the relationships between entities are represented by their distance value. So K-nearest neighbor clustering technology is targeted for detecting Divergent Change bad smell, and the results show directly that whether it is pungent or not. Similarly, the refactoring scheme about how to improve the smells is provided from the clustering results.

The clustering results of K-nearest neighbor algorithm are analyzed as follows:

1) The result is 1, and the entities in the class are classified as a whole group. It means that the relationships between these entities are close enough, and there is no Divergent Change bad smell.

2) The result is 2 or more, and there are two or more groups being classified. The relationships between entities inner the groups are close, and those cross the groups are loose. In other words, the groups being classified have higher cohesion and lower coupling than before. Actually, these entities are in single original class, so Divergent Change occurs. Furthermore, the classification results are just the refactoring scheme. It means the

original class should be divided into several smaller classes with Extract Class refactoring method, just as the clustering results.

The flow of Divergent Change bad smell detection is shown in Figure 2.

The most influence in K-nearest neighbor clustering algorithm is the input data of entities distance value. The distance value represented the dependency relationships between entities, as the dotted box shown in Figure 3. If the dependency relationships between these entities are closer, it is more probable to classify these entities into the same group. Otherwise, they should be separate.

Also the clustering results are influenced by K value and clustering iterations, as the shadow dotted box in Figure 3. The number of objects being clustered is N. From Equation (3), if K is larger, the number of groups after clustering is smaller. However, the time complexity of the clustering algorithm is $O(N \bullet K)$, so larger K will increase

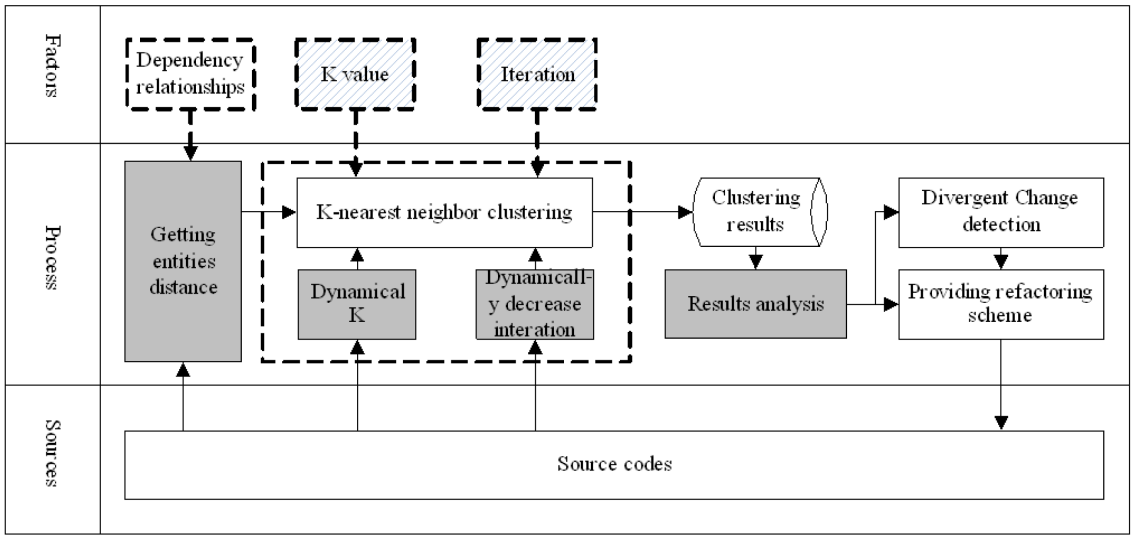

FIGURE 2. Algorithm flow diagrams

Algorithm：bad smell detection based on K-nearest neighbor clustering
Input：entity E_i, the kind C_i of E_i, the distance Dist[][] of E_i
Output：new kind C_new_i of E_i
Begin
  while (number of C_new_i still decrease)
    foreach ($E_i$in targeted class)
      select K entities (E_1,…,E_K) whose distances are smallest to E_i
      if (C_1,…,C_K are not same to each other) **then**
        C_i is set as C_x, and Dist[i][x] is the smallest in E_1,…,E_K
      else
        C_i is set as C_y, and C_y with largest number in E_1,…,E_K
        //if more than one entity is at the smallest/largest, choosing either is ok.
      endif
    endfor
  endwhile
End

FIGURE 3. Divergent change bad smell detection algorithm based on K-nearest neighbor clustering

the complexity of clustering algorithm. In this paper dynamic K value is proposed to increase the accuracy and decrease time consumption.

The value of K is got as shown:

$$K = [\log_2 N] + 1 \tag{4}$$

If N is larger, the iterations should be larger to get stable convergence results. That will increase the calculation time. So in this paper the iteration is decreased dynamically based on the scale of targeted class.

5.2. **Algorithm process.** The algorithm is executed after computing the distances between entities of the targeted class, as shown in Figure 3. Before the clustering, all the entities are consumed to be in different clusters. It is the basis of clustering algorithm.

Before this algorithm, the sample program should be preprocessed to collect useful information. In the input data, E_i is an array containing all the entities in the program, and C_i is the kinds of E_i. In addition, the distance values of each two entities have been computed and stored in the array Dist[][] to present the invoking relationships. The invoking relationships are used in Divergent Change bad smell detection.

6. **Evaluations.** In this paper the sample programs are HSQLDB and Tyrant. HSQLDB is a Java database open source, and the download link is hsqldb.org. Tyrant is a game, and the download link is sourceforge.net/projects/tyrant.

6.1. **Comparison results of different *K* value.** From the different K value, the clustering results are different, so the results of Divergent Change bad smell detection are changed. In HSQLDB version 2.2.6 program, the clustering results comparison with different K are shown in Figure 4.

Figure 5 shows the clustering results of first 110 classes Divergent Change bad smell detection of version 2.2.6 with different K value. When K increases, the number of clustering is decreased. If K is more than 9, the results are stable and with no change, but the time complexity will be extremely large. So K value is dynamic based on Equation (4). When K is set dynamically, the clustering result of the 55th class is 2 (results of all the other classes are 1). The clustering result of the 55th class with dynamical K is shown in Table 1.

The 55th class is the place of Divergent Change bad smell, and Table 4 gives the refactoring scheme. From Section 6.1, the 55th class is QuerySpecification, and it is divided into new classes QuerySpecification and DataQueryChange. From more versions

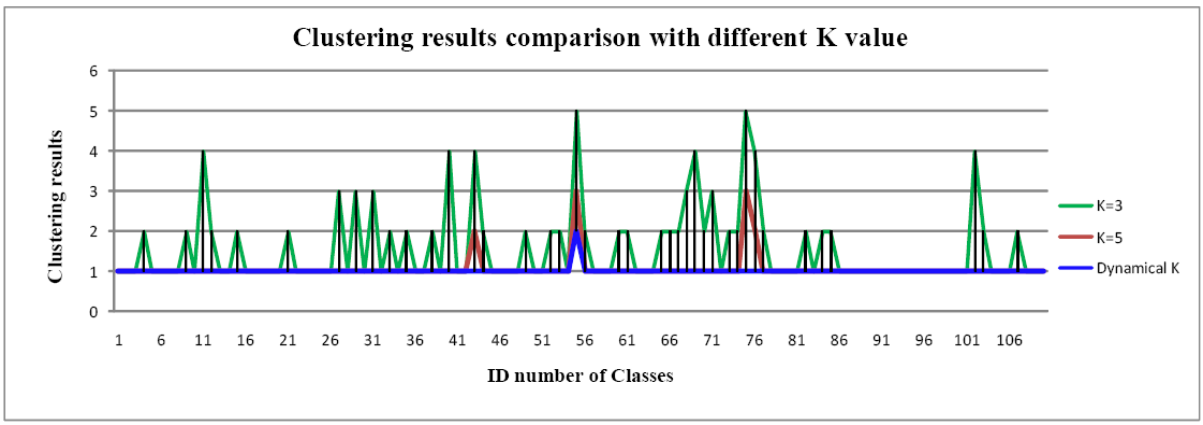

FIGURE 4. Clustering results comparison with different K value

TABLE 1. Clustering result of the 55th class with dynamical K in HSQLDB 2.2.6

Entity ID	Number of entities	Type
03149-03209	61	1
03210-03224	15	2
12766-12790	25	1
12791-12800	10	2

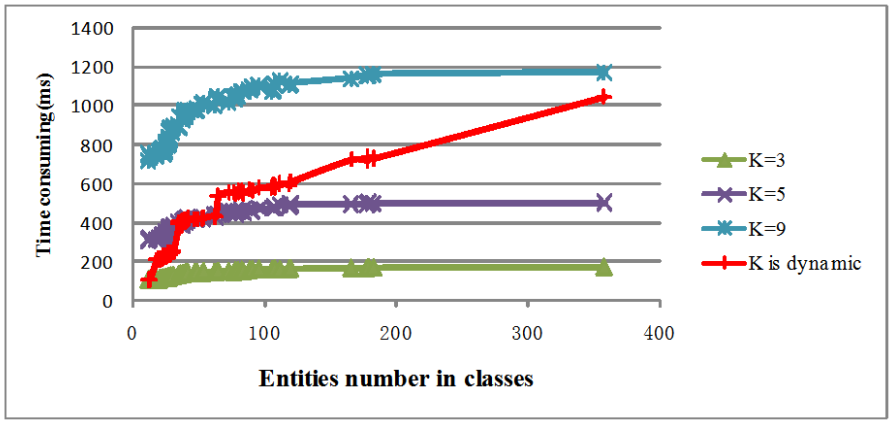

FIGURE 5. Time consuming comparison in different K value

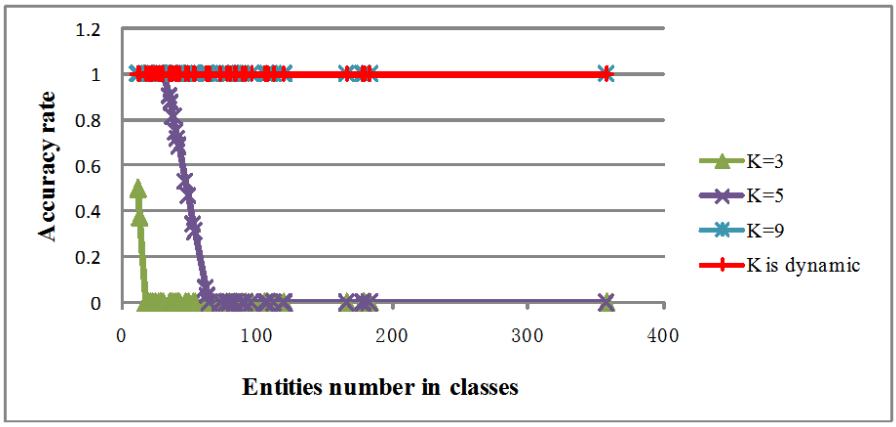

FIGURE 6. Clustering results accuracy comparison in different K value

comparison, the refactoring scheme provided by K-nearest neighbor clustering algorithm is correct.

The comparison and analysis of time consuming and accuracy about clustering results with different K value are shown in Figure 5 and Figure 6.

From Figure 5 and Figure 6, when K is smaller, the time consuming is lower, but detection accuracy is also lower; when K is larger, the accuracy is higher, but the time consuming is then higher. Just when K is set dynamically, the time consuming is low and accuracy is high. So finally the approach in this paper uses dynamic K.

6.2. **Comparison results of different approaches.** While the process of open source software version upgrades, new functions are added, and refactorings also occur. It means the bad smells in older version are removed in later versions. The multiple version comparison and analysis demonstrate the detection results and corresponding refactoring scheme

in this paper. Bad smells are detected in older versions and the advices of corresponding refactoring operations are given about what operations should be executed to remove the bad smells. And after comparison and analysis about later programs, if the modifications about removing bad smells are same with the advices, it means the Divergent Change bad smell detection and refactoring schemes are correct. Therefore, the results of multiple versions comparison are the actual results of refactoring schemes.

In this section the cases from business open source programs are detected for Divergent Change bad smell. The detection approach in this paper is measuring the distance values of entities and executing the K-nearest neighbor clustering. The value of K is dynamic. This approach is named DK approach for short. Another Divergent Change bad smell detection approach in recent researches is in [10], and it is called RR approach for short.

From the above, the detecting results of DK and RR approaches are compared with the actual results. And more similar means more accurate. In this paper the measures of precision and recall [22] are used for accuracy. Precision assesses the number of true smells identified among the detected smells, while recall assesses the number of detected smells among the existing smells.

(1) HSQLDB

The detection results in several versions of HSQLDB are shown in Tables 2 and 3.

Table 2. HSQLDB bad smell detection results comparison

Versions	Number of classes	Number of Divergent Change bad smell classes after detection		
		DK approach	RR approach	Actual results
2.0.0	471	0	0	0
2.2.0	503	12	9	12
2.2.8	512	3	3	3

Table 3. HSQLDB bad smell detection results analysis

Versions	Precision/Recall (%)	
	DK approach	RR approach
2.0.0	–/–	–/–
2.2.0	100/100	100/75
2.2.8	100/100	33.3/33.3

The versions selected are from 2.0.0 to 2.2.8. In these versions, the number of classes increased from 471 to 503. From the version update log, one function package is added in version 2.2.0 for functional request. And from 2.2.0 to 2.2.8, the functions of sources code are essentially unchanged.

After multiple version comparison, the actual results of Divergent Change bad smell are collected. In version 2.0.0, there is no bad smells being detected, so the precision and recall cannot be computed. In version 2.2.0, the number of actual Divergent Change is 12. DK approach detected 12, and RR approach detected 9. Then the recall of RR approach is less than 100%. In version 2.2.8, the number of actual Divergent Change is 3. Both DK and RR approach detected 3, but RR approach did not find all the actual bad smell classes. So both precision and recall of RR approach are less than 100%.

(2) Tyrant

The Divergent Change bad smell detection results of Tyrant are shown in Tables 4 and 5.

TABLE 4. Tyrant bad smell detection results comparison

Versions	Number of classes	Number of Divergent Change bad smell classes after detection		
		DK approach	RR approach	Actual results
0.312	117	14	10	14
0.316	135	7	7	7
0.319	150	12	9	12
0.324	165	10	9	10

TABLE 5. Tyrant bad smell detection results analysis

Versions	Precision/Recall (%)	
	DK approach	RR approach
0.312	100/100	100/71.4
0.316	100/100	100/100
0.319	100/100	100/75
0.324	100/100	100/90

6.3. **Analysis.** (1) From the comparison results of different K value, it is found that the approach of dynamic K has higher accuracy and lower time consuming. For the approach of fixed K value, higher accuracy needs higher K value, and this leads to higher computation complexity, so the computation time would be larger. Higher K value is not necessary for short and simple classes, since the increasing time consuming makes no sense. The approach of dynamic K value uses lower K value to decrease the time consuming on the basis of accuracy. Therefore, the dynamic K value balances both the advantages of accuracy and time consuming.

(2) From the comparison results of different approaches, DK approach is more accurate than RR approach in Divergent Change bad smell detection. This indicated the advantages of DK approach proposed in this paper. The reasons of these advantages are analyzed as follows.

First, DK approach gets actual dependency relationships. In DK approach the dependency relationships collection contains not only in classes but also across the classes. Furthermore, both direct and indirect dependency relationships are collected in distance computation. So the DK approach gets more comprehensive dependency relationships of programs.

On the other hand, the distance metric in DK approach completely expresses the situation about Divergent Change bad smell. The Divergent Change may be in different places of same class because of different variation, but this situation cannot be measured by average coupling metric of RR approach.

(3) The number of Divergent Change bad smells detected in Tyrant is more that of HSQLDB. The results indicate that the program quality of HSQLDB is better than Tyrant. Actually HSQLDB is a business program and Tyrant is just a little computer game.

7. **Conclusions.** The Divergent Change bad smell is analyzed to get the internal logic and cause factor, and related the smell expression to the dependencies of entities. Then the dependency relationships are transformed to distance value between entities. Finally, K-nearest neighbor clustering is executed and the results are used for Divergent Change bad smell detection and corresponding refactoring schemes. The clustering results are not only the basis about whether there are bad smells, but also the basis about how to

improve existing bad smells. From the clustering results, bad smells can be detected, and corresponding refactoring schemes can be analyzed.

The contributions of this paper are as follows. First, the out expression of Divergent Change is linked to the distances between entities. Then the dependency relationships of the programs can be measured by distance value. Moreover, the dependency information between entities from different classes is added in distance computing. So the distance value reflects the dependency situation more accurate, so the detection results and refactoring schemes are more effective. Second, K-nearest neighbor clustering algorithm is used for Divergent Change detection. The clustering results are not only the basis about whether there are bad smells, but also the basis about how to improve existing bad smells. The detection does not need any thresholds. The K value is set dynamically, so this method increases the accuracy and decreases the time consuming.

Acknowledgement. This research is supported by the National Natural Science Foundation of China under Grant No. 61173021 and the Research Fund for the Doctoral Program of Higher Education of China (Grant No. 20112302120052 and 20092302110040).

REFERENCES

[1] M. Fowler et al., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Professional, 1999.

[2] N.-L. Hsueh and P.-H. Chu, A pattern-based refactoring approach for multi-core system design, *IJACT*, vol.3, no.9, pp.196-209, 2011.

[3] S. M. Ibrahim et al., Identification of nominated classes for software refactoring using object-oriented cohesion metrics, *International Journal of Computer Science Issues*, vol.9, no.2, pp.68-76, 2012.

[4] A. A. Rao, Identifying clusters of concepts in a low cohesive class for extract class refactoring using metrics supplemented agglomerative clustering technique, *International Journal of Computer Science Issues*, vol.8, no.5, 2011.

[5] M. Arora, Refactoring-way for software maintenance, *International Journal of Computer Science Issues*, vol.8, no.2, pp.565-570, 2011.

[6] D. Jiang and P. Ma, Detecting bad smells with weight based distance metrics theory, *Proc. of the 2nd International Conference on Instrumentation, Measurement, Computer, Communication and Control*, pp.299-304, 2012.

[7] M. Mäntylä, Experiences on applying refactoring, *Software Engineering Seminar*, pp.1-32, 2002.

[8] B. Walter and B. Pietrzak, Multi-criteria detection of bad smells in code with UTA method, *Extreme Programming and Agile Processes in Software Engineering*, pp.1159-1161, 2005.

[9] K. N. Reddy and A. A. Rao, Dependency oriented complexity metrics to detect rippling related design defects, *Software Engineering Notes*, vol.34, no.4, pp.1-7, 2009.

[10] K. N. Reddy and A. A. Rao, A quantitative evaluation of software quality enhancement by refactoring using dependency oriented complexity metrics, *The 2nd International Conference on Emerging Trends in Engineering and Technology*, pp.1011-1018, 2009.

[11] C. N. Sant, A. F. Garcia and C. J. P. De Lucena, Evaluating the efficacy of concern-driven metrics: A comparative study, *Proc. of the 2nd Workshop on Assessment of Contemporary Modularization Techniques*, pp.25-30, 2008.

[12] A. De Lucia, R. Oliveto and L. Vorraro, Using structural and semantic metrics to improve class cohesion, *IEEE International Conference on Software Maintenance*, pp.27-36, 2008.

[13] C.-H. Lung and M. Zaman, Using clustering technique to restructure programs, *Proc. of the International Conference on Software Engineering Research and Practice*, pp.853-858, 2004.

[14] C. H. Lung, X. Xu, M. Zaman and A. Srinivasan, Program restructuring using clustering techniques, *The Journal of Systems and Software*, vol.79, no.9, pp.1261-1279, 2006.

[15] A. Alkhalid, M. Alshayeb and S. Mahmoud, Software refactoring at the function level using new adaptive K-nearest neighbor algorithm, *Advances in Engineering Software*, vol.41, no.10-11, pp.1160-1178, 2010.

[16] S. S. Srinivas, Package level software refactoring using A-KNN clustering technique, *International Conference on Computing and Control Engineering*, 2012.

[17] J. Ratzinger, T. Sigmund, P. Vorburger and H. Gall, Mining software evolution to predict refactoring, *Empirical Software Engineering and Measurement*, pp.354-363, 2007.
[18] A. K. Ghosh, P. Chaudhuri and C. A. Murthy, On visualization and aggregation of nearest neighbor classifiers, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol.27, no.10, pp.1592-1602, 2005.
[19] D. J. Hand and V. Vinciotti, Choosing K for two-class nearest neighbor classifiers with unbalanced classes, *Pattern Recognition Letters*, vol.24, pp.1555-1562, 2003.
[20] F. Simon, S. Löffler and C. Lewerentz, Distance based cohesion measuring, *Proc. of the 2nd European Software Measurement Conference*, Technolo-gisch Institute, Amsterdam, 1999,
[21] M. Bunge, *Treatise on Basic Philosophy*, Reidel Publishing Company, Dordrecht-Holland, 1977.
[22] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*, Addison-Wesley, 1999.