Improving Quality of Clone Detection with Conceptual Similarity of Source Code

Hung Viet Pham[1], Phong Minh Vu[2], Tam The Nguyen[2], Tung Thanh Nguyen[2]
[1]Utah State University, [2]Auburn University
hung.pham@aggiemail.usu.edu, phong@auburn.edu, tam@auburn.edu, tung@auburn.edu

Abstract

Code clones are highly similar code fragments which are highly desirable candidates for refactoring or aspect mining. However, popular clone detection techniques sometimes report clone candidates of low quality. This paper introduces CoFi, a filtering technique to remove them. CoFi has three major steps: i) learning to represent technical terms in source code as vectors, ii) measuring the conceptual similarity of clone candidates based on those vectors, and iii) filtering out clone candidates having low conceptual similarity. Preliminary evaluation suggests that CoFi can improve detection results of popular clone detection tool. It removed 83.3% of low quality clone pairs and retains 98.6% high quality pairs from the detection result of JSync, a tree-based detection tool. For DLCD, a deep learning-based detection tool, it removed 54% and retained 96.8%.

Introduction

In a large software system, an algorithm, a technical concern, a programming idiom, a business rule, or a design pattern is usually realized or implemented at several places. This typically results in highly similar code fragments. Because software programmers often use the *"copy-and-paste"* practice (*"cloning"*) to quickly produce similar code fragments, such code is often called *code clones* in software engineering literature.

Code clones are useful for many software engineering tasks. For example, code clones often have the same or similar programming errors. Thus, bug fixing will be more effective and complete if all clones of the buggy code are detected and fixed similarly. Code clones are good candidates for refactoring or mining cross-cutting concerns for aspect-oriented programming. They are also helpful for program comprehension, as they could imply the high level design of the systems.

Let us elaborate the usefulness of code clone detection via an example in ArgoUML. ArgoUML (8) is a modeling tool for Unified Modeling Language (UML) developed in Java. Users can use ArgoUML to draw UML diagrams while designing a software system. In UML, *classifier* represents the abstract concept of metaclass, which generalizes classes, interfaces, data types, etc. A *classifier* has a list of *attributes* (e.g., fields or data members of a class) and another list of *operations* (e.g., methods or function members of a class).

Table 1: Method signatures of two similar classes

UMLAttributesListModel	UMLOperationsListModel
resetCache()	resetCache()
isProperClass(Object obj)	isProperClass(Object obj)
getRawCollection()	getRawCollection()
getCache()	getCache()
add(int index)	add(int index)
delete(int index)	delete(int index)
moveUp(int index)	moveUp(int index)
moveDown(int index)	moveDown(int index)

Analyzing the source code of ArgoUML, we found that two classes UMLAttributesListModel and UMLOperationsListModel are respectively responsible for storing the lists of *attributes* and *operations* for a UML *classifier*. As *attribute* and *operation* are two abstract and related concepts for UML *classifiers*, it is no surprise that those two classes have very similar design and implementation code. For example, they both inherit from the same parent class UMLModelElementCachedListModel, and thus, inheriting the same fields and methods from that class. Table 1 lists the signatures of methods implemented in those two classes side-by-side. As seen from the table, those two classes often have methods with the same signatures. We observed that those methods also have very similar implementation code. Two pairs among them are listed in Figure 1.

Further investigation suggests that two classes UMLAttributesListModel and UMLOperationsListModel are resulted from the realization of the *Model-View-Control* design pattern for two similar and related concepts UML *attribute* and *operation*. Methods getCache implement the same programming idioms of *caching* and *lazy initialization*. Methods add implement the same algorithm for adding a new element to a cached collection. They both invoke the *Singleton* design pattern.

Those classes and methods could be considered as

```
protected java.util.List getCache() {
  if(_attributes == null) {
    _attributes = buildCache();
  }
  return _attributes;
}
```

```
protected java.util.List getCache() {
  if(_operations == null) {
    _operations = buildCache();
  }
  return _operations;
}
```

```
public void add(int index){
  Object target = getTarget();
  if(target instanceof MClassifier) {
    MClassifier classifier = (MClassifier) target;
    Collection oldFeatures = classifier.getFeatures();
    MAttribute newAttr = MMUtil.SINGLETON.buildAttribute(classifier);
    classifier.setFeatures(addElement(oldFeatures,index,newAttr,
          _attributes.isEmpty()?null:_attributes.get(index)));
    fireContentsChanged(this,index−1,index);
    navigateTo(newAttr);
  }
}
```

```
public void add(int index){
  Object target = getTarget();
  if(target instanceof MClassifier) {
    MClassifier classifier = (MClassifier) target;
    Collection oldFeatures = classifier.getFeatures();
    MOperation newOp = MMUtil.SINGLETON.buildOperation(classifier);
    classifier.setFeatures(addElement(oldFeatures,index,newOp,
          _operations.isEmpty()?null:_operations.get(index)));
    fireContentsChanged(this,index−1,index);
    navigateTo(newOp);
  }
}
```

Figure 1: Similar methods in classes UMLAttributesListModel (left) and UMLOperationsListModel (right)

```
public DocletParam createParam() {
  DocletParam param = new DocletParam();
  params.addElement(param);
  return param;
}
```

```
protected ArchiveScanner newArchiveScanner() {
  ZipScanner zs = new ZipScanner();
  zs.setEncoding(encoding);
  return zs;
}
```

Figure 2: Unrelated methods with the same token-based and tree-based representation

high quality code clones. Due to their high lexical and structural similarity, token-based (1) and tree-based (3; 2) clone detection tools can detect them easily. However, because those tools only use the similarity of code token sequences, parse-trees, or abstract syntax trees to detect code clones, they occasionally report clone candidates of low quality, which are accidentally similar in token sequences or tree structures but do not relate in design or implementation.

Figure 2 illustrates an example of such low quality detected clone candidates in Ant (7). Two methods createParam and newArchiveScanner have the same token sequences and abstract syntax trees, thus are reported as code clones by two popular tools CCFinder and Deckard. However, two classes DocletParam and ArchiveScanner are not similar or related: one represents doclet parameters of a Javadoc object, while the other implements functions for reading the content of compressed files. Two method calls params.addElement(param) and zs.setEncoding(encoding) perform two different tasks: one adds a newly created parameter to a parameter list, while the other sets the encoding of a scanner for .zip archive files. Thus, they could be considered as low quality detection results.

Comparing the two examples, we can see that code fragments in Figure 2 do not involve the same or similar software concepts, while ones in Figure 1 involve two related UML concepts *attribute* and *operation* and the same concepts of *list*, *caching*, *lazy initialization*, *Model-View-Controller*, *Singleton*, etc. This suggests that, in addition to being similarity in lexical and grammatical structures, high quality code clones also involve the same or similar software concepts. Thus, to detect high quality clones, we should measure *conceptual similarity* of code clone candidates and use such measurement to filter out low quality ones.

Approach

In this paper, we introduce CoFi (**Co**nceptual Similarity-based Code Clone **Fi**lter), an approach to improve the quality of clone detection by measuring *conceptual similarity* of clone candidates and filter out ones having low conceptual similarity. CoFi does that by learning vector-based representation of technical terms and computing conceptual similarity of code fragments based on similarity of vectors of the technical terms appearing in those code fragments.

Technical Terms

We design CoFi based on the assumption of *descriptive naming scheme* in software development. That is, software engineers would name code elements like classes, methods, fields, constants, parameters, and variables using terms indicating the software concepts realized by those elements. For example, the name UMLAttributesListModel of a class suggests it represents the data model for storing lists of UML attributes.

CoFi considers any word or a specific token appearing in identifiers of code elements of a software system as a *technical term*. To extract them from source code, it

tokenizes identifiers into lowercased tokens based on the Java naming convention. For example, UMLAttributesListModel is tokenized into uml, attributes, list, and model. A term could be a non-English token like uml, http, win32, or sha256.

Unlike typical information retrieval systems, CoFi *does not stem* or *lemmatize* the resulted terms after tokenizing. This is based on the assumption that technical terms having the same root might not express related software concepts. For example, the term *operation* might refers to the concept of *operation* in the UML language (e.g., a method or a function of a class), while the term *operating* in *operating system* does not have that meaning.

Context

CoFi defines the context of a term in a given code element/fragment as the collection of terms surrounding it. For each term extracted from an identifier, CoFi considers all other terms appearing in that identifier and in nearby identifiers as its context. For example, the context of term attributes in the identifier UMLAttributesListModel consists of uml, list, and model.

The context is important in CoFi and other text analytics techniques because terms appearing frequently in similar contexts would have similar roles (syntactical) or meanings (semantics). For example, two identifiers UMLAttributesListModel and UMLOperationsListModel provide the same context for two terms *attributes* and *operations*. As we have discussed previously, *attributes* and *operations* are actually two similar and related concepts in the UML modeling language and thus, in ArgoUML.

Vector-based representation

Traditional language modeling and text analytics techniques consider terms as atomic elements. For example, in Vector Space Model, a document is represented as a high dimensional vector in which each term represents an individual, separate dimension. However, terms are often related (e.g., synonyms). Treating terms as atomic elements cannot capture those relationships.

To address that problem, state-of-the-art word embedding approaches like Glove (4) and word2vec (5) model a term not as an atomic element but as a more complex entity represented by a high dimensional vector. Vectors of term are learned (i.e. estimated) from a large text corpus to capture the relationships between the corresponding terms. For example, the semantic similarity of terms could be estimated by the (dis)similarity of their representation vectors.

GloVe (4) has been proved to be the most effective technique in detecting similar words in natural languages. Thus, we adapt it to learn vectors of technical terms in source code. The learning procedure of CoFi has the following steps:

Step 1: *Pre-training*. We need a very large corpus of text to reliably train vectors for words in natural languages. However, the amount of textual content extracted from identifiers of code elements is insufficient. Therefore, we download vectors pre-trained for English words provided by Glove team and use them as initial values for the training process.

Step 2: *Extracting terms*. The training procedure of Glove requires a term co-occurrence matrix φ. For two terms x and y, φ_{xy} is the number of times y appearing in a context of x. Thus, CoFi analyzes all source files of the software system, extracts terms and their contexts as defined in previous sections and produce the co-occurrence matrix.

Step 3: *Estimating vectors*. CoFi loads pre-trained vectors for terms extracted in step 2 and uses Glove to re-train them using the co-occurrence matrix produced in Step 2. Terms without pre-trained vectors (e.g., non-English tokens) have their vectors randomly initialized.

In general, GloVe's training algorithm estimates for each term x a vector v_x satisfying the constraint:

$$v_x \cdot v_y \approx \log \varphi_{xy} \quad \forall x, y : \varphi_{xy} > 0$$

In other words, those vectors can be used to reconstruct the co-occurrence matrix which represents the relationships between terms. The details of this training algorithm are discussed in (4).

Conceptual similarity of terms

After training, CoFi produces for each term x a vector w_x. For two terms x and y, if they are similar, they often appear in the same contexts and thus having similar co-occurrences with terms in those contexts. That means, $\varphi_{xz} \approx \varphi_{yz}$ for all z. Supplying this to the training constraint, we have $v_x \cdot v_z \approx v_y \cdot v_z$ for all z, implying that $v_x \approx v_y$, i.e, x and y have similar vectors. Because cosine could be used to measure vector similarity, we define the conceptual similarity of x and y as

$$\text{sim}(x, y) = \frac{1 + \cos(v_x, v_y))}{2}$$

Because $\cos(v_x, v_y)$ is in [-1,1], this formula ensures that $\text{sim}(x, y)$ is in [0,1].

Conceptual similarity of code fragments

CoFi measures conceptual similarity of two code fragments X and Y as the following. First, it extracts technical terms from X and Y. Then, it aligns those terms to maximize the total similarity of aligned terms:

$$T_{XY} = \arg\max_{M} \sum_{(x,y) \in M} \text{sim}(x, y)$$

The conceptual similarity of X and Y is computed as:

$$\text{sim}(X, Y) = \frac{2T_{XY}}{N_X + N_Y}$$

In these formulas, M is an alignment for each term x in X to at most a term y in Y, while N_X and N_Y are the numbers of terms in X and Y, respectively. It is easy to see that $\text{sim}(X, Y)$ is in [0,1].

Preliminary Evaluation

We conducted a preliminary evaluation of CoFi for two state-of-the-art clone detection techniques. First, we used it to filter the detection result of JSync (3), a tree-based clone detection tool. CoFi filtered out 83.3% of low quality clones and retained 98.6% of high quality ones. Next, we used CoFi to filter the detection result of DLCD (6), a clone detection technique based on deep learning. It filtered out 54% of low quality clones and retained 96.8% of high quality ones.

For tree-based clone detection

JSync is the state-of-the-art tree-based clone detection and management tool developed by Nguyen *et al.* (3). JSync produces for each code fragment a vector counting the *structural features* extracted from its AST sub-tree. Two code fragments are reported as code clones if the structural similarity of their vectors exceeds a pre-defined threshold (of 0.9 in our experiment).

We ran JSync on a repository of ten subject systems and examined a sample of 326 pairs from its detection result. We manually labeled each clone pair as of high or low quality. Among 326 sampled clone pairs, 284 were labeled high quality and 42 were low quality. Two functions are considered as a high quality clone pair if:

- They implement the same algorithm or programming idiom (at least 75% of their steps are similar). Two steps are considered similar if they involve similar function calls or statements.
- They operate on similar data types. Two types are considered similar if they realize or implement the same or similar concepts that are closely related to functionality of the two methods under examination, e.g., StringBuffer and StringBuider for *string operations*.
- They involve similar or related functions/operations For example, add and remove something from a list are considered list operations, read and write are considered I/O operations, etc.
- They could be refactored to a generalized function with reasonable effort, i.e., be refactored easily using a common generalized function whose size is smaller than the total size of two initial methods.

Then, we ran CoFi on the sampled clone pairs. Ones with conceptual similarity less than 0.75 are considered as low quality and removed. We found that 35 low quality and 4 high quality pairs were filtered out. That means, CoFi removed 35/42 = 83.3% of low quality pairs and retained 280/284 = 98.6% high quality ones.

For deep learning-based clone detection

DLCD is the state-of-the-art deep learning-based clone detection technique developed by White *et al.* (6). To detect code clones, it employs a recurrent neural network (a type of deep learning infrastructures) to learn for each code fragment a vector representing its term sequence and a recursive neural network (another type of deep learning infrastructures) to learn another vector for its AST sub-tree. Code fragments with similar vectors are reported as code clones.

Because DLCD's implementation code is unavailable, we conducted our experiment with CoFi on the detection result provided by its authors. We obtained from their project website a dataset consisting 406 method-level clone pairs detected by DLCD on eight subject systems and labeled manually by its authors. They labeled 371 pairs as *"true positive"* and 35 as *"false positive"*. To be objective, we considered their *true* and *fault positive* labels as the *high* and *low quality* labels, respectively, in our labeling procedure.

We ran CoFi on those eight subject systems. After learning vectors for their terms, it computed the conceptual similarity of 406 clone pairs in this dataset and filtered out ones less than 0.75. 19 false positive and 12 true positive pairs were filtered. That means, CoFi removed 19/35 = 54% false positive and retained 359/371 = 96.8% true positive clone pairs.

Running time

CoFi has two main steps: i) extracting terms and counting their co-occurrences from source code and ii) learning vectors for those terms. We measured its running time on a workstation with i7 3770 CPU and 16 GB RAM. On average, it took 12 seconds for extracting terms and 69 seconds for learning vectors. On JDK, the largest subject system with more than 54,000 methods, it took 74 seconds for extracting 16,000+ terms and 155 seconds for learning their 300-dimensional vectors.

The result suggests that CoFi's runtime is reasonable and thus, it can be used efficiently in practice for interactive software development.

References

T. Kamiya, S. Kusumoto, and K. Inoue: *CCFinder: A multi-linguistic token-based code clone detection system for large scale source code.* TSE, 2002.

L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: *Scalable and accurate tree-based detection of code clones.* In ICSE, 2007.

H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen: *Clone management for evolving software.* TSE 2012.

J. Pennington, R. Socher, and C. Manning: *GloVe: Global vectors for word representation.* In EMNLP, 2014.

T. Mikolov, K. Chen, G. Corrado, and J. Dean: *Efficient estimation of word representations in vector space.* CoRR, 2013.

M. White, M. Tufano, C. Vendome, and D. Poshyvanyk: *Deep learning code fragments for code clone detection.* In ASE 2016.

Ant: http://ant.apache.org/

ArgoUML: http://argouml.tigris.org