An Empirical Study of Long-Lived Code Clones

Dongxiang Cai[1] and Miryung Kim[2]

[1] Hong Kong University of Science and Technology
caidx@cse.ust.hk**
[2] The University of Texas at Austin
miryung@ece.utexas.edu

Abstract. Previous research has shown that refactoring code clones as soon as they are formed or discovered is not always feasible or worthwhile to perform, since some clones never change during evolution and some disappear in a short amount of time, while some undergo repetitive similar edits over their long lifetime.
Toward a long-term goal of developing a recommendation system that selectively identifies clones to refactor, as a first step, we conducted an empirical investigation into the characteristics of long-lived clones. Our study of 13558 clone genealogies from 7 large open source projects, over the history of 33.25 years in total, found surprising results. The size of a clone, the number of clones in the same group, and the method-level distribution of clones are not strongly correlated with the survival time of clones. However, the number of developers who modified clones and the time since the last addition or removal of a clone to its group are highly correlated with the survival time of clones. This result indicates that the evolutionary characteristics of clones may be a better indicator for refactoring needs than static or spatial characteristics such as LOC, the number of clones in the same group, or the dispersion of clones in a system.

Keywords: Software evolution, empirical study, code clones, refactoring

1 Introduction

Code clones are code fragments similar to one another in syntax and semantics. Existing research on code cloning indicates that a significantly large portion of software (e.g. gcc-8.7% [9], JDK-29% [14], Linux-22.7% [27], etc.) contains code duplicates created by copy and paste programming practices. Though code cloning helps developers to reuse existing design and implementation, it could incur a significant maintenance cost because programmers need to apply repetitive edits when the common logic among clones changes. Neglecting to update clones consistently may introduce a bug.

Refactoring is defined as a disciplined technique for restructuring existing software systems, altering a program's internal structure without changing its

** This research was conducted while the first author was a graduate student intern at The University of Texas at Austin.

external behavior [10]. Because refactoring is considered a key to keeping source code easier to understand, modify, and extend, previous research effort has focused on automatically identifying clones [15, 4, 2, 21, 13].

However, recent studies on code clones [7, 18–20, 26] indicated that cloning is not necessarily harmful and that refactoring may not be always applicable to or beneficial for clones. In particular, our previous study of clone evolution [20] found that (1) some clones never change during evolution, (2) some clones disappear after staying in a system for only a short amount of time due to divergent changes, and (3) some clones stay in a system for a long time and undergo consistent updates repetitively, indicating a high potential return for the refactoring investment. These findings imply that it is crucial to *selectively identify* clones to refactor.

We hypothesize that the benefit of clone removal may depend on how long clones survive in the system and how often they require similar edits over their lifetime. Toward a long-term goal of developing a system that recommends clones to refactor, as a first step, we conducted an empirical investigation into the characteristics of long-surviving clones. Based on our prior work on clone genealogies—an automatically extracted history of clone evolution from a sequence of program versions [20]—we first studied various factors that may influence a clone's survival time, such as the number of clones in the same group, the number of consistent updates to clones in the past, the degree of clone dispersion in a system, etc. In total, we extracted 34 attributes from a clone genealogy and investigated correlation between each attribute and a clone's survival time in terms of the number of days before disappearance from a system. Our study found several surprising results. The more developers maintain code clones, the longer the clones survive in a system. The longer it has been since the time of the last addition or deletion of a clone to its clone group, the longer the clones survive in a system. On the other hand, a clone's survival time did not have much correlation with the size of clones, the number of clones in the same group, and the number of methods that the clones are located in.

For each subject, we developed a decision-tree based model that predicts a clone survival time based on its characteristics. The model's precision ranges from 58.1% to 79.4% and the recall measure ranges from 58.8% to 79.3%. This result shows promise in developing a refactoring recommendation system that selects long-lived clones.

The rest of this paper is organized as follows. Section 2 describes related work. Section 3 gives background of our previous clone genealogy research and Section 4 describes the characteristics of clone genealogy data and the subject programs that we studied. Section 5 describes correlation analysis results and Section 6 presents construction and evaluation of decision tree-based prediction models. Section 7 discusses threats to validity, and Section 8 summarizes our contributions.

2 Related Work

This section describes tool support for identifying refactoring opportunities, empirical studies of code cloning, and clone evolution analysis.

Identification of Refactoring Opportunities. Higo et al. [13] propose *Aries* to identify refactoring candidates based on the number of assigned variables, the number of referred variables, and clone dispersion in the class hierarchy. Aries suggests two types of refactorings, *extract method* and *pull-up method* [10]. A refactoring can be suggested if the clone metrics satisfy certain predefined values. Komondoor's technique [22] extracts non-contiguous lines of clones into a procedure that can then be refactored by applying an *extract method* refactoring. Koni-N'Sapu [23] provides refactoring suggestions based on the location of clones with respect to a system's class hierarchy. Balazinska et al. [2] suggest clone refactoring opportunities based on the differences between the cloned methods and the context of attributes, methods, and classes containing clones. *Breakaway* [8] automatically identifies detailed structural correspondences between two abstract syntax trees to help programmers generalize two pieces of similar code. Several techniques [35, 34, 33, 11, 28] automatically identify bad-smells that indicate refactoring needs. For example, Tsantalis and Chatzigeorgiou's technique identifies *extract method* refactoring opportunities using static slicing. Our work is different from these refactoring opportunity identification techniques in that it uses clone *evolution history* to predict how long clones are likely to survive in a system.

Studies about Cloning Practice. Cordy [7] notes that cloning is a common method of risk minimization used by financial institutions because modifying an abstraction can introduce the risk of breaking existing code. Fixing a shared abstraction is costly and time consuming as it requires any dependent code to be extensively tested. On the other hand, clones increase the degrees of freedom in maintaining each new application or module. Cordy noted that propagating bug fixes to clones is not always a desired practice because the risk of changing an already properly working module is too high.

Godfrey et al. [12] conducted a preliminary investigation of cloning in Linux SCSI drivers and found that super-linear growth in Linux is largely caused by cloning of drivers. Kapser and Godfrey [18] further studied cloning practices in several open source projects and found that clones are not necessarily harmful. Developers create new features by starting from existing similar ones, as this cloning practice permits the use of stable, already tested code. While interviewing and surveying developers about how they develop software, LaToza et al. [26] uncovered six patterns of why programmers create clones: repeated work, example, scattering, fork, branch, and language. For each pattern, less than half of the developers interviewed thought that the cloning pattern was a problem. LaToza et al.'s study confirms that most cloning is unlikely to be created with ill intentions. Rajapakse et al. [30] found that reducing duplication in a web application had negative effects on the extensibility of an application. After significantly reducing the size of the source code, a single change often required testing a vastly larger portion of the system. Avoiding clones during initial development could contribute to a significant overhead. These studies indicate that not all clones are harmful and it is important to *selectively identify clones to refactor*.

Clone Evolution Analysis. While our study uses the evolutionary characteristics captured by the clone genealogy model [20], the following clone evolution analyses could serve as a basis for generating clone evolution data. The evolution of code clones was analyzed for the first time by Laguë et al. [25]. Aversano et al. [1] refined our clone genealogy model [20] by further categorizing the *Inconsistent Change* pattern into the *Independent Evolution* pattern and the *Late Propagation* pattern. Krinke [24] also extended our clone genealogy analysis and independently studied clone evolution patterns. Balint et al. [3] developed a visualization tool to show who created and modified code clones, the time of the modifications, the location of clones in the system, and the size of code clones.

Classification of Code Clones. Bellon et al. categorized clones into Type 1 (an exact copy without modifications), Type 2 (a syntactically identical copy) and Type 3 (a copy with further modifications, e.g., addition and deletion of statements) in order to distinguish the kinds of clones that can be detected by existing clone detectors [5]. Kapser and Godfrey [17, 16] taxonomized clones to increase the user comprehension of code duplication and to filter false positives in clone detection results. Several attributes of a clone genealogy in Section 4 are motivated by Kapser and Godfrey's region and location based clone filtering criteria. Our work is different from these projects by identifying the characteristics of long-lived clones.

3 Background on Clone Genealogy and Data Set

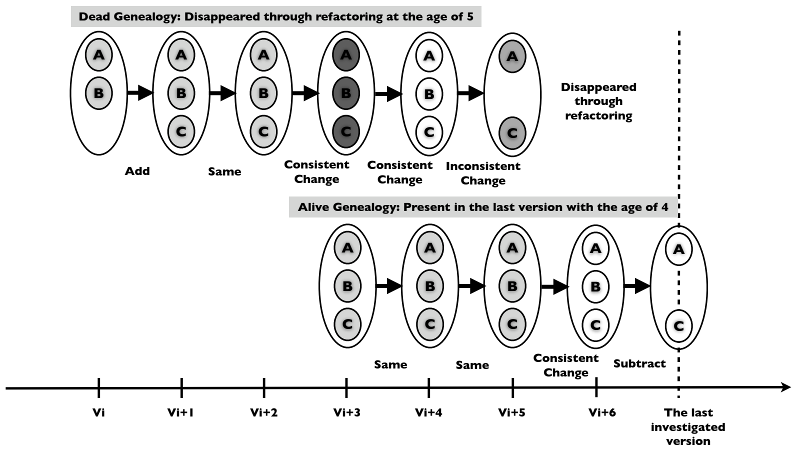
Dead Genealogy: Disappeared through refactoring at the age of 5
Disappeared through refactoring
Add
Same
Consistent Change
Consistent Change
Inconsistent Change
Alive Genealogy: Present in the last version with the age of 4
Same
Same
Consistent Change
Subtract
Vi
Vi+1
Vi+2
Vi+3
Vi+4
Vi+5
Vi+6
The last investigated version

Fig. 1. Example clone genealogies: $G1$ (above) and $G2$ (below)

A clone genealogy describes how groups of code clones change over multiple versions of the program. A clone group is a set of clones considered *equivalent* according to a clone detector. For example, clone A and B belong to the same group in version i because a clone detector finds them equivalent. In a clone's genealogy, a group to which the clone belongs is traced to its origin clone group in the previous version. The model associates related clone groups that have originated from the same ancestor group. In addition, the genealogy contains information about how each element in a group of clones changed with respect to other elements in the same group. The detail description on the clone genealogy representation is described elsewhere [20]. The following evolution patterns describe all possible changes in a clone group.

`Same:` all code snippets in the new version's clone group did not change from the old version's clone group.

`Add:` at least one code snippet is newly added to the clone group.

`Subtract:` at least one code snippet in the old version's clone group does not appear in the corresponding clone group in the new version.

`Consistent Change:` all code snippets in the old version's clone group have changed consistently; thus, they all belong to the new clone group.

`Inconsistent Change:` at least one code snippet in the old version's clone group changed inconsistently; thus, it no longer belongs to the same group in the new version.

`Shift:` at least one code snippet in the new clone group partially overlaps with at least one code snippet in the original clone group.

A *clone lineage* is a directed acyclic graph that describes the evolution history of a sink node (clone group). A clone group in the k^{th} version is connected to a clone group in the $k-1^{th}$ version by an evolution pattern. For example, Figure 1 shows a clone lineage following the sequence of `Add`, `Same`, `Consistent Change`, `Consistent Change`, and `Inconsistent Change`. In the figure, code snippets with the similar content are filled with the same shade.

A *clone genealogy* is a set of clone lineages that have originated from the same clone group. A clone genealogy is a connected component where every clone group is connected by at least one evolution pattern. A clone genealogy approximates how programmers create, propagate, and evolve code clones.

Clone genealogies are classified into two groups: *dead* genealogies that do not include clone groups of the final version and *alive* genealogies that include clone groups of the final version. We differentiate a dead genealogy from an alive genealogy because only dead genealogies provide information about how long clones stayed in the system before they disappeared. On the other hand, for an alive genealogy, we cannot tell how long its clones will survive because they are still evolving. In Figure 1, $G1$ is a dead genealogy with the age 5, and $G2$ is an alive genealogy with the age 4. Dead genealogies are essentially genealogies that disappeared because clones were either refactored, because they were deleted by a programmer, or because they are no longer considered as clones by a clone detector due to divergent changes to the clones.

To extract clone evolution histories with respect to this model, our clone genealogy extractor (CGE) takes a sequence of program versions as input and uses CCFinder [15] to find clones in each version. It then relates clones between consecutive versions based on a textual similarity computed by CCFinder and a location overlapping score computed by *diff*, a line-level program differencing tool. Basically, if the similarity between a clone of version i and a clone of a version $i+1$ is greater than a similarity threshold, sim_{th}, their container clone groups are mapped.

Data sets. For our study, we extracted clone genealogy data from seven projects: Eclipse JDT core, jEdit, HtmlUnit, JFreeChart, Hadoop.common, Hadoop.pig, and Columba. Table 1 summarizes the size of subject programs and the number of studied versions. We constructed genealogies at a temporal granularity of releases instead of check-ins because our goal is not to understand fine-grained evolution patterns but to correlate clones' characteristics with a survival time. In total, we studied 7 large projects of over 100KLOC, in total 33.25 years of release history. Table 2 summarizes the number of dead and alive genealogies. In our analysis, we removed dead genealogies of age 0, because they do not provide much meaningful information about evolutionary characteristics.

Table 1. Description of Java subject programs

project	URL	LOC	duration	# of check-ins	# of versions
Columba	http://sourceforge.net/projects/columba	80448 ∼ 194031	42 months	420	420
Eclipse	http://www.eclipse.org	216813 ∼ 424210	92 months	13790	21
common	http://hadoop.apache.org/common	226643 ∼ 315586	14 months	410	18
pig	http://hadoop.apache.org/pig	46949 ∼ 302316	33 months	906	8
HtmlUnit	http://htmlunit.sourceforge.net	35248 ∼ 279982	94 months	5850	22
jEdit	http://www.jedit.org	84318 ∼ 174767	91 months	3537	26
JFreeChart	http://www.jfree.org/jfreechart	284269 ∼ 316954	33 months	916	7

Table 2. Clone genealogies (min_{token}=40, $sim_{th} = 0.8$)

# of genealogies	Columba	Eclipse	common	pig	HtmlUnit	jEdit	JFreeChart
Total	556	3190	3094	3302	1029	654	1733
Alive genealogies	452	1257	627	2474	500	232	1495
Dead genealogies	104	1933	2467	828	529	422	238
# of dead genealogies with age>0	102	1826	455	422	425	245	219

4 Encoding Clone Genealogy Characteristics

To study the characteristics of long-lived clones, we encode a clone genealogy into a feature vector, which consists of a set of attributes. When measuring spatial characteristics of clones, we use information from the last version of a genealogy. For example, to measure the average size of clones in Genealogy $G1$ in Figure 1, we use the average size of clone A and C in the genealogy's last version, V_{i+5}. This section introduces each attribute and the rationale of choosing the attribute.

The number of clones in each group and the average size of a clone. The more clones exist in each clone group and the larger the size of each clone, it may require more effort for a developer to remove those clones, contributing to a longer survival time.

a_0: the total LOC (lines of code) of clones in a genealogy

a_1: the number of clones in each group

a_2: the average size of a clone in terms of LOC

Addition. Addition of new clones to a genealogy could imply that the clones are still volatile, or that it is becoming hard to maintain the system without introducing new clones, indicating potential refactoring needs.

a_3: the number of `Add` evolution patterns

a_4: the relative timing of the last `Add` pattern with respect to the age of a genealogy

Consistent update. If clones require similar edits repetitively over their lifetime, removal of those clones could provide higher maintenance cost-savings than removing unchanged clones.

a_5: the number of `Consistent Change` patterns

a_6: the relative timing of the last `Consistent Change` pattern with respect to the age of a genealogy

Subtraction. A `Subtract` pattern may indicate a programmer removed only a subset of existing clones.

a_7: the number of `Subtract` patterns

a_8: the relative timing of the last `Subtract` pattern with respect to the age of a genealogy

Inconsistent update. An `Inconsistent Change` pattern may indicate that the programmer forgot to update clones consistently, and thus a programmer may prefer to refactor such clones early to prevent inconsistent updates in the future.

a_9: the number of `Inconsistent Change` patterns

a_{10}: the relative timing of the last `Inconsistent Change` pattern with respect to the age of a genealogy

File modification. If a file containing clones was modified frequently, it may indicate that those clones are likely to be removed early.

a_{11}: the number of times that files containing clones were modified.

Developers. The more developers are involved in maintaining clones, it may be harder to refactor the clones.

a_{12}: the number of developers involved in maintaining clones.

a_{13}: the distribution of file modifications in terms of developers.

If the following entropy measure—a well-known measure of uncertainty [31]—is low, that means only a few developers make most of the modifications. If the entropy is high, all developers equally contribute to the modifications. The entropy measure is defined as follows: $entropy = \sum_{i=1}^{n} -plog(p_i)$, where p_i is the probability of a file modification belonging to author i, when n unique authors maintain the file.

Dispersion. The farther clones are located from one another, the harder it is to find and refactor them. Inspired by Kapser and Godfrey's clone taxonomy [17], we count the number of unique methods, classes, files, packages, and directories the clones are located. Table 3 shows that most clones are located within the same class or package, only 4.0% to 29.5% of clones are located within the same method, and only 2.8% to 31.1% clones are scattered across different directories.

We also used an entropy measure to characterize the physical distribution of clones at a different level (method, class, file, package, and directory respectively) by defining p_i to be the probability of clones located in location i. For example, in Figure 2, the dispersion entropy at a method level is 1.5, the entropy at a file level is 0.81, and the entropy at a package level is 0. If the entropy is low, clones are concentrated in only a few locations. If the entropy is high, clones are equally dispersed across different locations.

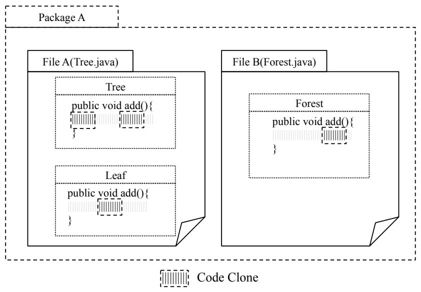

Fig. 2. Example physical distribution of code clones

a_{14}: Dispersion of clones into five nominal labels: 'within the same method', 'within the same class', 'within the same file', 'within the same package', and 'across multiple directories.'

a_{15}: The number of unique methods that clones in the *last* version are located.

a_{16}: The distribution of clones in the *last* version at a method level according to the entropy measure.

a_{17}: The number of unique methods that clones in *all* versions are located.

a_{18}: The distribution of clones in *all* versions at a method level according to the entropy measure.

We then created similar attributes at the level of class (a_{19} to a_{22}), file (a_{23} to a_{26}), package (a_{27} to a_{30}), and directory (a_{31} to a_{34}) by replicating attributes (a_{15} to a_{18}).

Clone survival time (class label). The last attribute a_{35} is the age of a genealogy in terms of the number of days.

a_{35}: the age of a genealogy in terms of the number of days

In the next section, we conduct a correlation analysis between each of the 34 attributes (a_1 to a_{34}) with a clone survival time (a_{35}).

5 Characteristics of Long-Lived Clones

To understand the characteristics of long-lived clones, we measured Pearson's correlation coefficient between each attribute and a clone genealogy survival time [32]. In our analysis, we used only dead genealogies, because alive genealogies are still evolving and thus cannot be used to predict how long clones would

Table 3. Characteristics of studied clones

		Columba	Eclipse	common	pig	HtmlUnit	jEdit	JFreeChart
Age	Average age (days)	538.6	435.1	72.49	136.6	292.8	640.9	229.0
	Min	1.1	68.7	34.0	30.0	6.9	13.3	11.1
	Max	1222.2	2010.0	585.0	536.9	2122.4	2281.7	415.0
# of clones in each group	Average	3.38	3.37	3.67	4.54	4.43	4.10	3.26
	Min	2	2	2	2	2	2	2
	Max	21	112	53	115	62	120	42
Size (LOC)	Average clone size	12.97	18.59	16.22	14.38	14.34	12.10	15.77
	Min	2	3	3	3	3	4	4
	Max	38.5	343.4	79	70	92	60	75
Dispersion	% clones in the same method	27.5%	29.5%	13.4%	12.8%	4%	24.5%	22.8%
	% clones in the same class	61.8%	60.6%	32.5%	49.1%	44.9%	75.9%	31.5%
	% clones in the same file	61.8%	61.0%	48.8%	49.5%	45.2%	75.9%	31.5%
	% clones in the same package	80.4%	83.5%	71.4%	94.1%	87.3%	90.6%	58.9%
	% clones in the same directory	87.3%	83.9%	75.4%	97.2%	88.5%	93.9%	68.9%

survive before they disappear. Table 4 shows the result of top 5 and bottom 5 attributes in terms of correlation strength.

The result indicates the more developers maintained clones, the longer the survival time of a clone genealogy (a12). The more uniformly developers contribute to maintaining clones, the longer time it takes for the clones to be removed (a13). The longer it has been since the last addition or deletion of a clone, the longer it takes for them to be removed (a4 and a8). On the other hand, the size of clones (LOC), the number of clones in each group, and the physical dispersion of clones do not affect a clone survival time much (a0, a1, a31, and a32). This implies that the size and the number of clones do not play much role in estimating a clone survival time; however, it may be harder to remove those clones changed by a large number of developers.

Table 4. Correlation analysis results

	All	**Columba**	**JDT**	**common**	**pig**	**HtmlUnit**	**jEdit**	**JFreeChart**
	$a_{13}(0.553)$	$a_{27}(0.366)$	$a_{13}(0.632)$	$a_8(0.568)$	$a_{13}(0.494)$	$a_{13}(0.674)$	$a_{12}(0.385)$	$a_6(0.448)$
	$a_{12}(0.528)$	$a_{29}(0.353)$	$a_{12}(0.601)$	$a_4(0.563)$	$a_{12}(0.474)$	$a_8(0.647)$	$a_{29}(0.358)$	$a_{12}(0.446)$
Top 5	$a_8(0.481)$	$a_{28}(0.351)$	$a_4(0.562)$	$a_{10}(0.466)$	$a_4(0.302)$	$a_4(0.637)$	$a_{33}(0.358)$	$a_{11}(0.438)$
	$a_4(0.479)$	$a_{21}(0.307)$	$a_8(0.561)$	$a_7(0.456)$	$a_8(0.287)$	$a_{12}(0.628)$	$a_{21}(0.356)$	$a_5(0.415)$
	$a_{11}(0.458)$	$a_{25}(0.300)$	$a_{11}(0.493)$	$a_3(0.448)$	$a_{10}(0.247)$	$a_7(0.551)$	$a_{13}(0.347)$	$a_{10}(0.294)$
	$a_{31}(0.023)$	$a_{15}(0.031)$	$a_{23}(0.015)$	$a_{18}(0.048)$	$a_{24}(0.008)$	$a_{32}(0.051)$	$a_{15}(0.066)$	$a_1(0.041)$
	$a_{32}(0.018)$	$a_{18}(0.027)$	$a_{33}(0.013)$	$a_{17}(0.046)$	$a_{20}(0.006)$	$a_0(0.043)$	$a_9(0.047)$	$a_{17}(0.021)$
Bot. 5	$a_1(0.016)$	$a_0(0.027)$	$a_{24}(0.009)$	$a_{32}(0.026)$	$a_{25}(0.004)$	$a_{25}(0.040)$	$a_{16}(0.039)$	$a_{22}(0.016)$
	$a_{17}(0.014)$	$a_{10}(0.011)$	$a_{19}(0.009)$	$a_{24}(0.021)$	$a_{21}(0.004)$	$a_{21}(0.036)$	$a_0(0.036)$	$a_{18}(0.006)$
	$a_0(0.009)$	$a_{16}(0.007)$	$a_{20}(0.005)$	$a_{31}(0.012)$	$a_{26}(0.001)$	$a_1(0.014)$	$a_1(0.003)$	$a_{21}(0.006)$

For example, a clone genealogy id `1317` from Eclipse JDT disappeared in revision 13992 after surviving more than 813 days. We found that the clones were modified over 83 times by 10 different developers and were finally removed when fixing bug id 172633. As another example, we found a clone genealogy that contained 45 clones modified by 6 different developers in 20 different revisions, which survived 898 days before being removed.

6 Predicting the Survival Time of Clones

This section describes a decision-tree based model that predicts how long clones are likely to stay in a system. When building a training data set, we categorized a clone survival time into five categories: `very short-lived`, `short-lived`, `normal`, `long-lived`, and `very long-lived`. It is very important to find an unbiased binning scheme that converts the number of days a clone survived into a nominal label. If a binning scheme is chosen so that most vectors in the training data are put into a single bin, then the resulting prediction model is highly accurate by predicting always the same label. However, it is not useful because it cannot distinguish the survival time of clones.

To find an unbiased binning scheme, we explored two binning methods to convert a clone survival time into five categories. The first scheme is to gradually increase the size of a bin such that the bin size is larger than the preceding bin size: $bin_i = bin_{i-1} + 0.5 \times (i+1) \times \chi$, where χ is the size of the first bin. For example, when χ is 50, the binning scheme is { [0,50), [50, 125), [125, 225), [225, 350), [350, ∞) }. The second scheme is to uniformly assign a bin size to χ. For both schemes, we varied the size of the first bin χ to be 30, 40, 50, 60 and 70 and computed the entropy measure to assess distribution of the training data set across those bins. If the training feature vectors are equally distributed across five bins, the entropy should be 2.3219 ($= -\log_2 \frac{1}{5}$). If all vectors are localized in a single bin, the entropy will be 0. For each subject program, we then selected a binning scheme with the highest entropy score, which will distribute the training set as uniformly as possible across the five bins. Figure 5 shows the entropy score for different binning schemes. After selecting a binning scheme with the highest entropy score, the training data set is distributed across the selected bins as shown in Table 6.

Table 5. Entropy measures for various binning schemes

project	**entropy in incremental scheme**					**entropy in uniform scheme**				
χ value	30	40	50	60	70	30	40	50	60	70
Columba	1.797	1.954	1.796	1.860	1.955	1.256	1.474	1.769	1.958	1.945
Eclipse[1]	1.101	1.496	1.804	1.534	1.919	0.642	0.659	1.092	1.365	1.755
common	1.229	1.367	1.353	1.274	1.235	1.201	1.330	1.352	1.360	1.331
pig	1.979	2.201	2.061	2.040	1.826	1.512	2.106	2.139	2.172	2.067
HtmlUnit	2.009	1.943	1.662	2.157	2.036	1.724	1.945	2.009	2.065	2.102
jEdit	1.085	1.454	1.544	1.735	1.843	0.604	0.821	1.136	1.465	1.473
JFreeChart	1.654	1.535	1.882	1.535	1.339	0.846	0.728	0.769	1.535	1.535

[1] For Eclipse JDT, we tried additional χ values (80, 90, 100). For the incremental binning scheme, entropy = 2.227, 2.264, and 1.970 respectively, while using an uniform scheme, entropy = 1.823, 1.543, and 1.784.

We built prediction models using five different classifiers in the Weka toolkit (K Nearest Neighbor, J48, Naive Bayes, Bayes Network and Random Forest)

Table 6. Categorization of feature vectors based on a selected binning scheme

project	**# of vectors**	**survival time (days)**	**# of genealogies for each category**
Columba	102	1.1 ∼ 1222.2	[0,60):18, [60,120):8, [120,180):9, [180,240):16, [240+):51
Eclipse	1826	68.7 ∼ 2010.0	[0,90):204, [90,225):423, [225,405):340, [405,630):510, [630+):349
common	455	34.0 ∼ 585.0	[0,40):324, [40,100):66, [100,180):16, [180,280):33, [280+):16
pig	422	30.0 ∼ 536.9	[0,40):131, [40,100):91, [100,180):97, [180,280):31, [280+):72
HtmlUnit	425	6.9 ∼ 2122.4	[0,60):125, [60,150):119, [150,270):63, [270,420):24, [420+):94
jEdit	245	13.3 ∼ 2281.7	[0,70):22, [70,175):31, [175,315):31, [315,490):22, [490+):139
JFreeChart	219	11.1 ∼ 415.0	[0,50):37, [50,125):2, [125,225):104, [225,350):38, [350+):38

[1] $[n,m)$:k means there are k number of vectors whose survival time lie in between n to m days.

on these data sets. The J48 decision tree-based classifier [29] combined with the bagging method [6] performed the best among these classifiers in terms of overall accuracy. J48 is an implementation of Quinlan's decision tree learner C4.5 [29] based on information entropy. At each node of the tree, it chooses an attribute that most effectively splits the data set into subsets. The attribute that results in the highest normalized information gain (difference in entropy) is used to split the data. Bagging is a bootstrapping method, proposed by L. Breiman [6] that generates multiple versions of a predictor and aggregates these predictors into a new predictor. Since our class label is nominal, the resulting predictor uses a voting scheme to produce a new class label. Figure 4 shows a J48 decision tree for Eclipse JDT core. This model considers factors such as the number of developers who modified clones and the dispersion of clones in a system to predict how long the clones are likely to survive in a system.

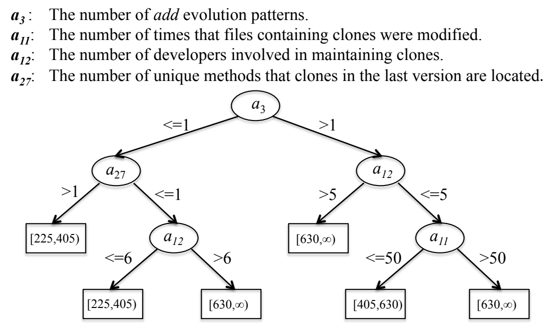
a_3: The number of *add* evolution patterns.
a_{11}: The number of times that files containing clones were modified.
a_{12}: The number of developers involved in maintaining clones.
a_{27}: The number of unique methods that clones in the last version are located.
a_3
<=1
>1
a_{27}
a_{12}
>1
<=1
>5
<=5
[225,405)
a_{12}
[630,∞)
a_{11}
<=6
>6
<=50
>50
[225,405)
[630,∞)
[405,630)
[630,∞)

Fig. 3. An excerpt of a resulting decision tree for Eclipse JDT core

Project	**Precision**	**Recall**
Columba	58.1%	58.8%
Eclipse JDT core	79.4%	79.3%
Hadoop.common	74.5%	78.0%
Hadoop.pig	79.1%	79.1%
HtmlUnit	73.3%	73.6%
jEdit	62.0%	65.7%
JFreeChart	68.2%	70.3%
Total	75.7%	76.5%

Fig. 4. Prediction model

We use 10-fold cross validation to evaluate the prediction model based on two measures: (1) a weighted average precision, $\frac{\sum_{i=1}^{n} \frac{TP_i}{TP_i+FP_i} \times t_i}{\sum_{i=1}^{n} t_i}$ and (2) a weighted average recall, $\frac{\sum_{i=1}^{n} TP_i}{\sum_{i=1}^{n} t_i}$, when TP_i is the number of correct predictions of each class label i, FP_i is the number of incorrect predictions of i, and t_i is the total number of vectors with a class label i in the training data set. Table 6 summarizes the weighted average precision and recall measures for each project. Our precision ranges from 58.1% to 79.4%, and our recall ranges from 58.8% to 79.3%. This result shows promise in using the attributes extracted from clone evolution data to predict a clone survival time.

7 Limitations

Our clone genealogy extractor (CGE) uses CCFinder to detect code clones and to map clones across versions [15]. CCFinder is a token-based clone detection technique that transforms tokens of a program according to a language-specific rule and performs a token-by-token comparison. CCFinder is recognized as a state of the art clone detector that handles industrial size programs; it is reported to produce higher recall although its precision is lower than some other tools. CCFinder does not detect *non-contiguous* clones and it is sensitive to reordering statements. This limitation leads to CGE's limitation in extracting clone genealogy data. If a programmer consistently modified an old clone group OG to create a new clone group NG, CCFinder does not find a cloning relationship between OG and NG if they do not share a contiguous token string greater than the size of sim_{th}=$(|OG.text| + |NG.text|)/2$. The absence of a cloning relationship can be mistakenly interpreted as a discontinuation of a lineage. Furthermore, CGE incorrectly counts the number of consistent change patterns in some cases, because CCFinder detects only a contiguous token string as a clone. For example, when code is inserted in the middle of one clone in a clone group, the existing clone group is broken into two new clone groups with shorter contiguous text, causing identification of two consistent patterns rather than one inconsistent change pattern. Similarly when the statements in a clone are reordered, such clones could be considered as removed because CCFinder may not be able to detect those clones.

We set the minimum token threshold of CCFinder to be 40 tokens and the similarity threshold sim_{th} for associating clones between consecutive versions as 0.8. Thus, we considered only the clones that are at least 40 tokens long and could map clones across versions only when the old and new clones are at least 80% similar according to CCFinder's equivalence criteria.

We extracted clone genealogies at a temporal granularity of major releases because CGE could not handle more than 1000 program versions as input. Studying clone evolution data at a finer temporal granularity such as check-in snapshots may provide more accurate evolutionary characteristics of long-lived clones. When studying the characteristics of clones, we did not consider the dispersion of clones in a class hierarchy or the refatorability of clones. Further investigation of such characteristics remains as future work.

8 Conclusions

Previous studies on code cloning indicate that clones are not necessarily harmful and that refactoring may not be always applicable to clones or be even beneficial for them. As a first step toward selectively identifying clones to refactor, we conducted an empirical investigation into the characteristics of long-lived clones. Based on our prior work on clone genealogy extraction, we developed a method that takes a clone genealogy as input and generates a feature vector to encode its characteristics. By feeding the feature vectors to decision-tree based classification algorithms, we developed models that predict a clone survival time. The study found that the size of a clone, the number of clones in the same group, and the method-level distribution of clones are not strongly correlated with a clone survival time. However, the number of developers who modified clones and the time since the last addition or removal of a clone to its group are highly correlated with the survival time of clones. The survival time prediction model has 75.7% precision and 76.5% recall, showing promise in selectively identifying clones to remove.

References

1. Lerina Aversano, Luigi Cerulo, and Massimiliano Di Penta. How clones are maintained: An empirical study. In *CSMR '07*, pages 81–90, Washington, DC, USA, 2007. IEEE Computer Society.
2. Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lagüe, and Kostas Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *WCRE '00*, page 98, Washington, DC, USA, 2000. IEEE Computer Society.
3. Mihai Balint, Radu Marinescu, and Tudor Girba. How developers copy. In *ICPC '06*, pages 56–68, Washington, DC, USA, 2006. IEEE Computer Society.
4. Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM '98*, page 368, Washington, DC, USA, 1998. IEEE Computer Society.
5. Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Trans. Softw. Eng.*, 33(9):577–591, 2007.
6. Leo Breiman. Bagging predictors. *Mach. Learn.*, 24(2):123–140, 1996.
7. James R. Cordy. Comprehending reality " practical barriers to industrial adoption of software maintenance automation. In *IWPC '03*, page 196, Washington, DC, USA, 2003. IEEE Computer Society.
8. Rylan Cottrell, Joseph J. C. Chang, Robert J. Walker, and Jörg Denzinger. Determining detailed structural correspondence for generalization tasks. In *ESEC-FSE '07*, pages 165–174, New York, NY, USA, 2007. ACM.
9. Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *ICSM '99*, page 109, Washington, DC, USA, 1999. IEEE Computer Society.
10. Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, 2000.

11. Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. Identifying architectural bad smells. In *CSMR '09*, pages 255–258, Washington, DC, USA, 2009. IEEE Computer Society.
12. Michael W. Godfrey, Davor Svetinovic, and Qiang Tu. Evolution, growth, and cloning in linux, a case study. In *CASCON '00*, 2000.
13. Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Refactoring support based on code clone analysis. In *PROFES '04*, pages 220–233, 2004.
14. J. Howard Johnson. Identifying redundancy in source code using fingerprints. In *CASCON '93*, pages 171–183. IBM Press, 1993.
15. Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.
16. Cory Kapser and Michael W. Godfrey. Aiding comprehension of cloning through categorization. In *IWPSE '04*, pages 85–94, Washington, DC, USA, 2004. IEEE Computer Society.
17. Cory Kapser and Michael W. Godfrey. Improved tool support for the investigation of duplication in software. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 305–314, Washington, DC, USA, 2005. IEEE Computer Society.
18. Cory Kapser and Michael W. Godfrey. "cloning considered harmful" considered harmful. In *WCRE '06*, pages 19–28, Washington, DC, USA, 2006. IEEE Computer Society.
19. Miryung Kim, Lawrence Bergman, Tessa Lau, and David Notkin. An ethnographic study of copy and paste programming practices in oopl. In *ISESE '04*, pages 83–92, Washington, DC, USA, 2004. IEEE Computer Society.
20. Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. In *ESEC/FSE-13*, pages 187–196, New York, NY, USA, 2005. ACM.
21. Raghavan Komondoor and Susan Horwitz. Semantics-preserving procedure extraction. In *POPL '00*, pages 155–169, New York, NY, USA, 2000. ACM Press.
22. Raghavan Komondoor and Susan Horwitz. Effective, automatic procedure extraction. In *IWPC '03*, page 33, Washington, DC, USA, 2003. IEEE Computer Society.
23. G. G. Koni-N'-Sapu. A scenario based approach for refactoring duplicated code in object-oriented systems. Master's thesis, University of Bern, 2001.
24. Jens Krinke. A study of consistent and inconsistent changes to code clones. In *WCRE '07*, pages 170–178, Washington, DC, USA, 2007. IEEE Computer Society.
25. Bruno Lague, Daniel Proulx, Jean Mayrand, Ettore M. Merlo, and John Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *ICSM '97*, page 314, Washington, DC, USA, 1997. IEEE Computer Society.
26. Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *ICSE '06*, pages 492–501, New York, NY, USA, 2006. ACM.
27. Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI '04*, pages 289–302, 2004.
28. Naouel Moha, Yann-Gaël Guéhéneuc, Anne-Françoise Le Meur, and Laurence Duchien. A domain analysis to specify design defects and generate detection algorithms. In *FASE'08/ETAPS'08*, pages 276–291, Berlin, Heidelberg, 2008. Springer-Verlag.

29. J.R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.
30. Damith C. Rajapakse and Stan Jarzabek. Using server pages to unify clones in web applications: A trade-off analysis. In *ICSE '07*, pages 116–126, Washington, DC, USA, 2007. IEEE Computer Society.
31. Fazlollah M. Reza. *An Introduction to Information Theory*. Dover Publications, Inc., New York, USA, 1996.
32. J. L. Rodgers and W. A. Nicewander. Thirteen ways to look at the correlation coefficient. *The American Statistician*, 42:59–66, 1988.
33. Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. Jdeodorant: Identification and removal of type-checking bad smells. In *CSMR '08*, pages 329–331, Washington, DC, USA, 2008. IEEE Computer Society.
34. Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of extract method refactoring opportunities. In *CSMR '09*, pages 119–128, Washington, DC, USA, 2009. IEEE Computer Society.
35. Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of move method refactoring opportunities. *IEEE Trans. Softw. Eng.*, 35(3):347–367, 2009.