An Information Theoretic Evaluation of Software Metrics for Object Lifetime Prediction

Jeremy Singer[1], Sebastien Marion[2], Gavin Brown[1], Richard Jones[2],
Mikel Luján[1], Chris Ryder[2], and Ian Watson[1]

[1] University of Manchester, UK
{jsinger,gbrown,mlujan,watson}@cs.man.ac.uk
[2] University of Kent, UK
{sm244,r.e.jones,c.ryder}@kent.ac.uk

Abstract. Accurate object lifetime prediction can be exploited by allocators to improve the performance of generational garbage collection by placing immortal or long-lived objects directly into immortal or old generations. Object-oriented software metrics are emerging as viable indicators for object lifetime prediction. This paper studies the correlation of various metrics with object lifetimes. However, to date most studies have been empirical and have not provided any information theoretic underpinning. We use the information theoretic calculation of normalized mutual information to measure correlation. We assess which metrics are most useful for prediction and construct some simple yet accurate object lifetime predictors.

1 Introduction

Generational garbage collection has long remained the dominant GC paradigm [1, 2]. Yet it wastes much effort copying long-lived objects (and worse, immortal objects) into older generations. These costs can be avoided if, whenever an object is created in a program, the allocator can determine how long that object will live [3–5]. This paper addresses how much and what kind of static information is required to make such predictions.

An object lifetime prediction scheme takes some allocation context (such as object type, allocation site identifier, or call stack context) as input; an *allocation site* is a program location that creates a new object. The scheme generates a lifetime prediction for the newly created object. This may be as precise as an integer value [6] or as vague as a binary short/long indicator [3]. Accurate object lifetime prediction is important for high-performance garbage collection systems, as Section 2 explains.

Most existing prediction techniques are program-specific. That is, they profile some program execution, record object lifetime data, then *extrapolate* future predictions, for the same program, from this profile run. They look for dynamic allocations that appear to be 'similar' to profiled allocations (usually based on the allocation site location, or the allocated object type) and predict

the behaviour to be the same as it was on the profile run. This is known as *self prediction* in the GC literature [7].

In contrast, this paper proposes a technique that profiles program X, then uses data from X to make predictions about allocations in a different program Y. This is known as *true prediction* in the GC literature [7]. The main insight underlying our work is that we use measures of 'allocation similarity' that are not program-specific. These similarity measures are *object-oriented software metrics*, described in Section 3. Thus, the **first key contribution** of this paper is to demonstrate that software metrics can be used to predict object lifetimes.

To date, most type-based object lifetime prediction schemes [8–10] only consider the type of *one* object at each allocation site—namely the allocatee, which we refer to as the destination object. However, another object type is involved at the allocation site—namely the allocator, which we refer to as the source object. Figure 1 shows a typical allocation site in a fragment of Java source code, with source and destination object types (classes) explicitly labelled. Thus, the **second key contribution** of this paper is to demonstrate that both source and destination types should be considered for predicting object lifetimes.

```
class Src {

    void f() {

        Dst d = new Dst(); // allocation site

        ...
    }
    ...
}
```

Fig. 1. Source code for an allocation site in Java, showing both the source and destination object types

Given that we can take a number of metric measurements (on source and destination types) at each allocation site, the next issue to address is how to determine the correlation between these metrics and the allocated object's lifetime. We aim to determine a correlation score that is independent of programs, programming languages, virtual machines and GC schemes. Section 5 uses an *information theoretic* framework to measure correlation between metrics and lifetimes, based around normalized mutual information (NMI) as a correlation indicator. Thus, the **third key contribution** of this paper is to highlight NMI as a good correlation measure for allocation site features that may be used to predict object lifetimes.

NMI measures correlation and determines the best allocation site features to use in prediction, but it does not give any indication of the functional nature of

the predictor. (In this sense, NMI is quite unlike Pearson's R coefficient, which indicates that the correlation is linear.) Therefore in order to determine the actual predictors, Section 5.4 uses various off-the-shelf machine learning algorithms. The **fourth key contribution** of this paper is an evaluation of several prediction schemes generated using standard machine learning techniques.

Finally, Section 6 assesses the NMI correlation scores and generated predictors to see if they make sense, given our domain-specific knowledge. Is it possible to explain the relative importance of the features, and the prediction rules in an intuitive way? This interpretation of our information theoretic analysis is the **final key contribution** of the paper.

2 Background

2.1 Garbage Collection is Important

Garbage collection (GC) is the automatic management of dynamically allocated memory [11]. It has become a significant concern to the mainstream computer industry, due to the rise of managed programming languages like Java and C# [12]. These 'modern' object-oriented languages are designed to run on sandboxed virtual machine (VM) systems, that use GC (amongst other techniques) for memory safety.

2.2 High-Performance GC is Important

In such managed environments, GC accounts for a significant fraction of execution time. GC is wasted time as far as the application user is concerned, so there is a need to minimise overall GC time. In addition, interactive or soft real-time application users would prefer to have frequent, short GC pauses rather than occasional longer pauses. These requirements motivate the adoption of *generational* GC [1, 2, 13]. The cost of a copying collection is determined by the volume of objects that survive (are copied by) that collection. Starting from the weak generational hypothesis that 'most objects die young' [1], generational GC allocates all objects in a *nursery* space that is frequently collected. The relatively few long-lived objects, identified as survivors of a threshold number of nursery collections, are relocated to a larger, infrequently collected *mature* space. Thus, generational GC improves both expected (but not worst case) pause time by concentrating effort on the nursery where few objects are expected to survive, and overall collection time by processing long-lived objects less frequently (and hence giving them more time to die). In addition, some *immortal* objects will survive until the end of the application's execution. If these can be identified, they can be allocated to an *immortal space* that is never collected. Most modern high-performance VMs use some variant of the generational GC scheme [14–16]

2.3 Object Lifetime Prediction enables High-Performance GC

The generational GC scheme outlined above has one inefficiency. Long-lived objects are processed (possibly repeatedly) in the nursery space, and (eventually)

copied to the mature space. Immortal objects are repeatedly processed in mature generation(s). If the GC system could identify long-lived (respectively immortal) objects at or before their dynamic allocation point, it would be better to allocate such objects directly into the mature (respectively immortal) space. This optimization is known as *pretenuring* [3–5]. It depends upon accurate ahead-of-time prediction of object lifetimes. However, an inaccurate predictor, which suggests that short-lived objects are actually long-lived or immortal, can severely degrade the GC efficiency, by clogging up the mature and immortal spaces with short-lived objects. This causes artificial elongation of both their lifetimes and those of their referents [17]. The extra garbage in the mature space will cause also an increased number of mature space collections, which are expensive.

In our earlier work [18], we demonstrated that we could obtain lifetime advice from a simple form of software metrics, *micro-patterns*, [19]; guiding allocation with this advice improved the GC time of a generational copying collector. However, micro-patterns provide only a very simple intra-class, largely syntactic, object-oriented metric. In this paper, we seek to explore more sophisticated software metrics that take into account both relationships between fields and methods of the same class but also relations between classes. Furthermore, we provide a sound, information theoretic underpinning to our exploration.

3   Software Metrics

Software metrics are used to capture concise and quantitative descriptions of certain aspects of software systems. Metrics are generally used to measure code size, complexity, quality and cost. Other information may be derived from these metrics, such as programmer productivity and system maintainability. A key advantage of software metrics is their independence from any system or programming language. They are generally designed to be easy to compute. Often automated tools are readily available to calculate metrics scores.

Chidamber and Kemerer have proposed a set of six metrics for object-oriented software [20]. These are well-known and generally accepted in the software engineering community. Spinellis [21] has suggested the addition of two further metrics. We adopt this set of eight metrics, henceforth referred to as the *CK metrics suite* or *cjkm*, as a means of characterizing Java classes. Table 1 gives details of each metric. We extract the metrics directly from Java bytecode files using the ckjm tool [21].

Chidamber and Kemerer originally intended these metrics for human-oriented managerial interpretation [22]. They have since been employed for other purposes such as Java benchmark characterization [23]. Most recently, we have evaluated the metrics for use in object lifetime prediction [10]. This paper extends our earlier work. We now consider all eight CK metrics (rather than merely the original six). In addition, we measure metrics for both allocator and allocatee objects at each allocation site, whereas previously we only considered metrics for allocatees.

weighted methods per class (WMC): Since ckjm uses a default weight of 1.0, this is simply a count of the number of methods defined by the current class.

depth of inheritance tree (DIT): The `java.lang.Object` class has a DIT score of 1. The DIT score increments by 1 for every edge on the path through the inheritance tree from the root to the current class.

number of children (NOC): The number of classes that are immediate subclasses of the current class.

coupling between object classes (CBO): The number of classes upon which the current class depends. This coupling can occur through method calls, field accesses, inheritance, arguments, return types, and exceptions.

response for a class (RFC): Ideally, this should measure the number of different methods that can be executed when an instance method of the current class is invoked, summed over all instance methods for this class. This would involve calculating the transitive closure of the method's call graph, which has the potential to be expensive and inaccurate. Instead, ckjm calculates a rough approximation to RFC by simply counting method calls within the class's method bodies. This simplification was also used in the original CK metrics paper [20].

lack of cohesion in methods (LCOM): This counts sets of methods in the current class that are not related through the sharing of some of the class's fields. The ckjm tool considers all pairs of the class's methods. In some of these pairs, both methods access at least one common field of the class, while in other pairs the two methods do not access any common fields. The LCOM score is calculated as the number of pairs that do share at least one field from the number of pairs that do not share any fields.

afferent couplings (Ca): This measures the number of classes that depend on the current class. Coupling is defined in the same way as for CBO above. Conceptually, Ca is the inverse of CBO.

number of public methods (NPM): This is simply a count of the methods in the current class that are declared as public. It can be used to give an indication of the size of an API provided by a package.

Table 1. Description of the CK metrics suite

4 Data Collection Framework

component	name	version	details
virtual machine	Jikes RVM	2.4.4	BaseBaseSemiSpace config
library	GNU Classpath	0.10	
benchmark	SPECjvm98	1.03	all except check
benchmark	DaCapo	β-051009	antlr, bloat, fop, hsqldb, jython, pmd, ps

Table 2. Software analysed in object lifetime study

We gather object lifetime data from typical Java benchmarks running on Jikes RVM, which is a Java-in-Java virtual machine [15, 24]. We are able to analyse allocation sites within the VM, libraries and the benchmark applications. Table 2 gives full details of the analysed Java code[3]. We profiled with our *MemTrace* system [18], using the base compiler at both build- and run-time (BaseBase). We profile with BaseBase because our focus is on application objects rather than optimising compiler data (which has a more pronounced effect on smaller programs). We also wished to minimise the effect of compiler-allocated data on exaggerating object lifetimes (which are measured in bytes—the size of any allocation, e.g. by the compiler, between an object's birth and death contributes to its lifetime). Further, BaseBase MemTrace configurations generate comparatively smaller, although still several gigabyte, trace files. MemTrace forces periodic full-heap collections to get tight bounds on object lifetimes. It associates objects with their allocation sites by modifying the JIT compiler to record allocation site identifiers in an object header word. Object birth times are byte-accurate, and death times are accurate to the granularity of a full-heap collection period, which is set to 64kB for all experiments in this paper.

We classify object lifetimes using scheme proposed by Blackburn et al [4, 5]. They use *maxLiveSize*, the maximum volume of data live at any point in the program's execution, as a normalizing factor.

1. If an object dies later than halfway between its time of birth and the end of the program, then it is classified as *immortal*.
2. Otherwise, if an object's age is greater than $T_a \times maxLiveSize$, then it is classified as *long-lived*. We keep $T_a = 0.45$, as in [4].
3. Otherwise, the object is classified as *short-lived*.

These rules enable us to label each allocated object with a lifetime. Now, for each allocation site, we consider the proportion of objects with each lifetime. Blackburn et al [4, 5] classify allocation site according to the fraction S_s of short-lived, L_s of long-lived and I_s of immortal objects it allocates. Homogeneity

---

[3] Only the seven DaCapo benchmarks listed in the table would run with Jikes RVM v2.4.4.

factors H_{lf} and H_{if} determine the conservatism of the decision to pretenure into the mature and immortal spaces respectively. Higher homogeneity factors reduce the number of sites classified as long-lived or immortal. We retain the same values as the original version [4], namely $H_{lf} = 0.6$ and $H_{if} = 0.0$.

1. If $I_s > S_s + L_s + H_{if}$, the site is classified as *immortal*.
2. If $I_s + L_s > S_s + H_{lf}$, the site is classified as *long-lived*.
3. Otherwise, the site is classified as *short-lived*.

Now we are able to label each allocation site with an expected object lifetime.

Finally, we use the *ckjm* tool [21] to harvest CK metrics values for each class under consideration (from Jikes RVM, GNU classpath libraries, and benchmarks). We identify the source and destination object types at each allocation site, and associate the appropriate set of metrics values with the corresponding lifetime. This means we create a database with one entry per allocation site. Each entry has the CK metrics for the source and destination object types, and the object lifetime. Now we are able to perform data mining and information theoretic analysis on this database.

Note that we only analyse object allocations at present, so we ignore array allocations entirely. We feel that array allocations must be treated differently, since the destination does not have a set of CK metric values. In theory, we could use the CK metric values belonging to the array elements, but this still does not deal with arrays of primitive types or references. In terms of the 50476 allocation sites in our profile data, 40432 are object allocation sites and 10044 (around 20%) are array allocation sites.

For all of the information theoretic analysis in Section 5, we discretize the numerical metrics values into labelled bins for various ranges, using the Kononeko MDL method in weka [25].

5   Information Theoretic Analysis

5.1   Calculation of Information Theory Measurements

The fundamental information theoretic measure is *entropy*, which quantifies the information content in a given source of data: the more 'randomness' or unpredictability in the data source, the higher the entropy value. Consider a device producing symbols according to a random variable X, defined over a finite alphabet of possible symbols S_X. If we assume each successive symbol $s_i \in S_X$ is independent of the previous ones, the *unconditional entropy* is defined as,

$$H(X) = -\sum_{i=1}^{|S_X|} p(i) \log(p(i)) \tag{1}$$

where $p(i)$ is the probability of the ith symbol being produced. Note that all logarithms are taken to base 2. In practical terms, $p(i)$ can be calculated with frequency counts, i.e.:

$$p(i) = \frac{\text{number of occurrences of symbol } s_i}{\text{total number of symbols seen}} \tag{2}$$

This paper assumes the produced symbols to be a common CK metric measurement on a sequence of dynamic object allocations. This gives us a stream of metric values, for which we can calculate an entropy measure.

The *conditional entropy* measures the dependence between two different symbol streams. In our case, we could take two measurements on each element of a sequence of dynamic object allocations. For instance we could measure a CK metric for each newly created object and its lifetime. These are two different random variables X and Y respectively with two different alphabets S_X and S_Y but there may be some dependence between them, which we can quantify by conditional entropy.

$$H(Y|X) = -\sum_{i=1}^{|S_X|} p(i) \sum_{j=1}^{|S_Y|} p(j|i) \log(p(j|i)) \tag{3}$$

This is the *first order conditional entropy*. The required probabilities can again be computed from frequency counts:

$$p(j|i) = \frac{\text{number of times } s_j \text{ occurs with } s_i}{\text{number of occurrences of } s_i} \tag{4}$$

First order conditional entropy has a minimum value of zero and a maximum value of $\log(|S_Y|)$. In the example above, it measures the uncertainty we have in the lifetime of an object given the value of a certain CK metric. If lifetime values are produced uniformly at random over the alphabet S_Y, then eq.(3) will converge in the limit to $\log(|S_Y|)$.

The *mutual information* between X and Y is a measure of the agreement, or correlation, between them. The mutual information is,

$$I(X;Y) = H(Y) - H(Y|X) \tag{5}$$

This is easily computed from the entropy measurements we have already described above. This measurement is symmetric, i.e. $I(X;Y) = I(Y;X)$, and quantifies the reduction in our uncertainty of Y when the value of X is revealed. Unlike Pearson's R correlation coefficient, which only detects *linear* correlations between random variables, mutual information can detect arbitrary *nonlinear* relationships.

$I(X;Y)$ can be normalized to a value between 0 and 1 by dividing it by $\min(H(X), H(Y))$. The maximum value of normalized mutual information (NMI) indicates that there is perfect correlation between the two variables. Given the value of one variable, it is always theoretically possible to construct a predictor that will predict the value of the other variable with 100% accuracy. A low value of NMI indicates that there is little information, and therefore little opportunity for accurate prediction of Y given X. A zero value of NMI indicates that the two variables are entirely uncorrelated, so knowing the value of one variable does not avail for making predictions about the other variable's value.

5.2 Correlation of Individual Features

Our first analysis assesses the utility of single CK metrics as features for predicting object lifetimes. The NMI scores are shown in Table 3. The rows are sorted according to NMI values. Each row reports the NMI of a single metric with the object lifetime. Most of the figures are disappointingly low. For instance, knowing the NOC metric (number of child classes) for source and destination classes is almost useless for predicting lifetimes. Other metrics show limited potential: six metrics have NMI scores above 0.25.

The top two metrics in Table 3 are CK metrics for source objects. We might assume that it is more important to know about the source object than the destination object, or at least, it is important to know something about the source object as well as the destination object. This is an important insight! Until now, most type-based object lifetime studies only consider characteristics of the destination object (the allocatee) rather than the source (the allocator). The only type-based study that considers both source and destination characteristics [18] does not conduct a formal assessment of the relative importance of these source or destination characteristics.

metric	NMI
lifetime	1.000
source LCOM	0.370
source RFC	0.342
dest LCOM	0.324
dest RFC	0.314
dest NPM	0.261
dest WMC	0.257
source WMC	0.233
source CBO	0.180
source NPM	0.163
source Ca	0.117
dest DIT	0.079
dest Ca	0.056
source DIT	0.037
source NOC	0.035
dest CBO	0.030
dest NOC	0.012

Table 3. NMI-based correlation of single features with lifetime

5.3 Conditional Mutual Information Maximization

Table 3 reveals that single features all have low NMI scores. We require a *combination* of features to obtain reliable predictions. However one problem with selecting features based only on NMI is that this selection process does not take account of cross-correlation between features. It is best to select features that have high individual correlation with the class to predict and have low cross-correlation with each other. Fleuret [26] presents an attractive algorithm to do automatic feature selection based on mutual information that considers

cross-correlation. His technique is known as *conditional mutual information maximization* (CMIM). The approach iteratively picks features that maximize their mutual information with the class to predict, conditioned on features already picked. This CMIM criterion does not select a feature *similar* to already picked ones, even if it is individually informative, since such a similar feature does not carry *additional* information about the class to predict.

Conditional mutual information is calculated as:

$$I(U; V|W) = H(U|W) - H(U|W, V) \tag{6}$$

This value is an estimate of the quantity of information shared between U and V when W is known. If V and W carry the same information about U, then the two terms on the right are equal and the conditional mutual information is zero, even if both V and W are individually informative. Conversely if V contains information about U which is not present in W, then the difference is large and the conditional mutual information is high.

The CMIM algorithm operates as follows. It aims to pick k features from a total of n, in order of relevance with the most relevant feature first. Incidentally, if we use CMIM to select n features from n, then we get a relevance-ordered ranking of our entire feature set which takes into account cross-correlations in a way that our simple ranking based on NMI scores in Table 3 did not.

The algorithm maintains a score vector, with one element for each feature. Initially, the score vector is set up so $score[i]$ contains the *unnormalized* mutual information score for the ith feature (with the lifetime). At each iteration, the feature with the highest score value is taken as the next selected feature. Then the score vector is recomputed, with each element $score[i]$ set to the minimum value of ($score[i]$, I(lifetime; ith feature|last selected feature). This ensures that $score[i]$ is low if at least one of the features already picked is similar to the ith feature.

Fleuret [26] gives a full explanation of the algorithm, including various optimization techniques using lazy evaluation and boolean bit-vectors. We implement his CMIM algorithm and use it to rank the CK metric features in order of relevance for object lifetime prediction. Table 4 gives the results of this CMIM analysis. Note that in the table, each score is the highest value in score vector at that particular iteration, for a feature that has not been selected by previous iterations.

5.4   Prototype Prediction Schemes

In order to evaluate the feature selection and ranking decisions above, we generate several predictors using the C4.5 tree learner algorithm. We report the accuracy of these decision tree predictors, but as yet we have not used the predictions to optimize generational GC, so we are unable to give real benchmark speedups at this stage.

Recall that the total object lifetime database contains around 40,000 entries. Each entry records source and destination CK metrics for a single scalar allocation site, together with the most likely lifetime for objects allocated at that site.

metric	CMIM score
source LCOM	0.407
dest RFC	0.345
source RFC	0.177
source CBO	0.144
dest LCOM	0.142
source NPM	0.113
source WMC	0.104
source Ca	0.086
dest NPM	0.085
dest WMC	0.072
dest Ca	0.061
source DIT	0.041
dest DIT	0.038
dest CBO	0.033
source NOC	0.025
dest NOC	0.007

Table 4. CMIM-based ranking of features for prediction of object lifetime

The data is randomly split 50:50 into training/test sets. This is repeated for five trials. To use just five trials may not seem enough for statistical significance. However the results in all our experiments have such low variance that more than five trials is not necessary.

We investigate how the (mean) accuracy of the generated decision tree varies as different numbers of features are added. The features are selected according to their ranking from the CMIM algorithm, as shown earlier in Table 4. Figure 2 presents the results, including error bars to indicate plus/minus one standard deviation. It is evident that a tree built using all 16 features has an accuracy of approximately 78%. This is significantly better than the performance of the weka baseline ZeroR predictor, which has an accuracy score of 44.8%. The ZeroR predictor always selects the most frequently occurring outcome, assuming all allocation sites are equally likely.

Note that when we select just the top three features indicated by CMIM, the generated predictor achieves 77.6% accuracy, for a significantly reduced decision complexity. After pruning, the three-feature trees had on average just 10 nodes, while the 16 feature trees had on average 40 nodes. It is clear to see that three features provide similar accuracy to 16 features, at a much lower complexity cost. This is clear justification for the application of feature selection techniques outlined earlier in the paper.

As a further extreme example, we consider only the top *two* features identified by CMIM. This feature reduction, combined with aggressive pruning, enables us to visualize the decision boundaries of a small C4.5 decision tree. Figure 3 illustrates this map. It is a graphical presentation of a simple set of rules, that achieves 75% accuracy on the test data. The primary advantage of this visualization is that it can also support *interpretation* of the rules. We present three 'intuitive' instantiations of the rules below:

1. The point marked + in the map corresponds to an allocation site in the Dacapo `pmd` benchmark. The source class is `pmd.ast.JavaParser`. The des-
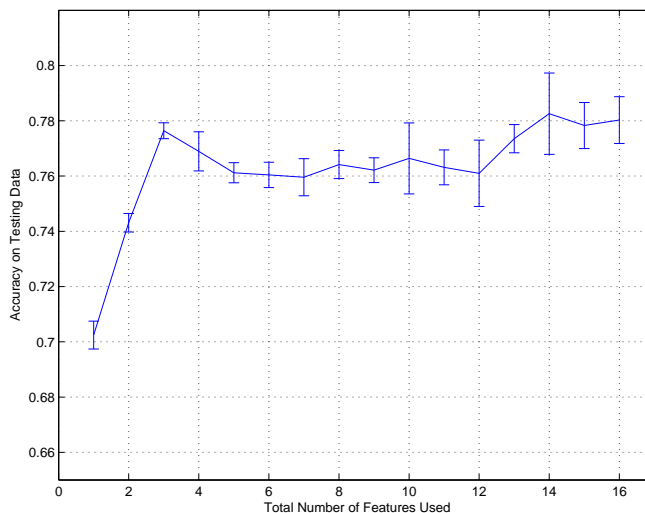
Fig. 2. Graph showing how C4.5 predictor accuracy changes with number of metric features for lifetime prediction

tination class is the `LookaheadSuccess` inner class of the source. The map shows that this allocation is predicted to be *short-lived*. This seems likely, given domain-specific knowledge that look-ahead events are frequent in the parsing process, and the specific inner class contains no long-term state.

2. The `*` point corresponds to an allocation site in DaCapo `bloat`. The source class is `bloat.tree.PrintVisitor`. The destination class is `StringBuffer`. This allocation is predicted to be *short-lived*. This seems appealing, since `StringBuffer` objects are generally ephemeral and the `PrintVisitor` class presumably makes a traversal over the entire tree, creating and emitting a textual representation of the tree nodes. We speculate that this narrow band of short-lived objects between 58 and 59 on the dRFC axis is almost entirely due to `StringBuffer` objects.

3. The `%` point corresponds to any allocation site whose destination class is `String`. All such allocations are predicted to be *long-lived*. Again, this appeals to intuition since `String` objects are immutable and often contain long-term information.

In earlier work [18], we used rules rather than decision trees in our object lifetime predictors. A single rule is equivalent to a single path from the root node to a leaf node in the decision tree. Some paths match many cases in the dataset, whereas others only have a single match. Some paths have 100% successful prediction rate, whereas other paths have lower accuracy. Selecting a subset of rules from the decision tree enables us to eliminate unpopular or inaccurate decisions. In addition, single rules are easier to interpret than a complete decision tree.
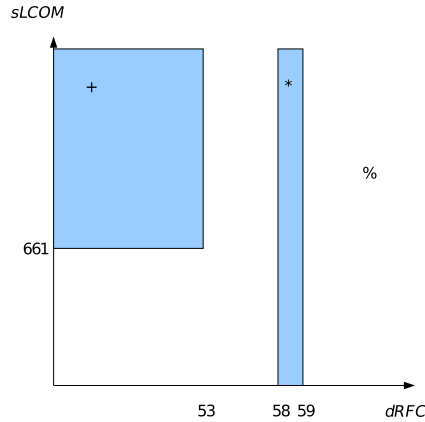
Fig. 3. 2-dimensional map showing how allocation sites with various metric values map onto different lifetimes. Shaded areas represent short-lived allocations. Unshaded areas represent immortal allocations.

6   Explanations of Analysis

Feature selection is a fundamental topic in Machine Learning. Too many features can lead to *overfitting* of a learning model, and hence poor performance when the learning system is deployed in the field. It is important to distinguish here between feature *selection*, and feature *extraction*. The former is the focus of this paper. The latter encompasses techniques such *Principal Components Analysis* (PCA) and *Bayesian Automatic Relevance Determination* (ARD) [27].

Extraction techniques measure *functions* of features, which are linear for PCA and nonlinear for ARD. The typical result is a 'black-box' mathematical function of the original data. The *meaning* of the original features is lost. In contrast, feature selection techniques such as those based on conditional mutual information maximization allow us to *retain original meaning* and provide human-readable explanations of *how* a feature is useful in combination with others.

For instance, we can attempt to interpret the feature ranking provided by CMIM in Table 4. This shows us that LCOM and RFC are important metrics, and that source metrics are generally more important than destination metrics. Note how there is only one out of eight CK metrics for which the destination value is ranked above the source value in the table.

The LCOM and RFC metrics measure the complexity of a class, in terms of methods. Recall from Section 3 that LCOM measures how various sets of methods in a class access disjoint field sets in the class, and RFC measures how many methods are called directly by the methods of a class. These two values provide a precise way to characterize a class. Perhaps this kind of precise metric 'fingerprint' what is needed to get accurate object lifetime predictions. We know

that many of the generated rules match multiple allocation sites, but in future work we should study whether these sites have different source and destination types. The key question is: do the rules we generate with CK metrics generalize well over types, or do the metrics values in a single rule simply alias onto a single (source,destination) type pair?

7 Conclusions and Future Work

In this paper, we have shown that the Chidamber and Kemerer object-oriented software metrics suite can be used to predict object lifetimes accurately. We have used basic concepts from information theory to determine correlation of metric features with lifetime, to rank features in order of relevance, and to select the best features as predictor inputs. We have created, evaluated and interpreted some simple prediction schemes based on the C4.5 decision tree learner algorithm.

We plan to implement object lifetime prediction strategies based on CK metrics, to determine if they lead to improvements in GC performance, using the infrastructure of our earlier experiments (which was based on object micro-patterns [18]) in order to obtain some empirical performance results.

References

1. Ungar, D.: Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In: Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments. (1984) 157–167
2. Lieberman, H., Hewitt, C.: A real-time garbage collector based on the lifetimes of objects. CACM **26**(6) (1983) 419–429
3. Cheng, P., Harper, R., Lee, P.: Generational stack collection and profile-driven pretenuring. In: *PLDI*. (1998) 162–173
4. Blackburn, S.M., Singhai, S., Hertz, M., McKinley, K.S., Moss, J.E.B.: Pretenuring for Java. In: *OOPSLA*. (2001) 342–352
5. Blackburn, S.M., Hertz, M., McKinley, K.S., Moss, J.E.B., Yang, T.: Profile-based pretenuring. ACM Transactions on Programming Languages and Systems **29**(1) (2007) 1–57
6. Inoue, H., Stefanovic, D., Forrest, S.: On the prediction of Java object lifetimes. IEEE Transactions on Computers **55**(7) (2006) 880–892
7. Barrett, D.A., Zorn, B.G.: Using lifetime predictors to improve memory allocation performance. In: Proceedings of the ACM SIGPLAN 1993 conference on Programming Language Design and Implementation. (1993) 187–196
8. Shuf, Y., Gupta, M., Bordawekar, R., Singh, J.P.: Exploiting prolific types for memory management and optimizations. In: *POPL*. (2002) 295–306
9. Huang, W., Srisa-an, W., Chang, J.M.: Dynamic pretenuring schemes for generational garbage collection. In: IEEE International Symposium on Performance Analysis of Systems and Software. (2004) 133–140
10. Singer, J., Brown, G., Luján, M., Watson, I.: Towards intelligent analysis techniques for object pretenuring. In: Proceedings of the International Conference on Principles and Practice of Programming in Java. (Sep 2007) 203–208

11. Jones, R., Lins, R.: Garbage Collection: Algorithms for Automatic Dynamic Memory Management. Wiley (1996)
12. Jones, R.: Dynamic memory management: Challenges for today and tomorrow. In: Proceedings of the International Lisp Conference. (2007) 115–124
13. Baker, H.G.: Infant mortality and generational garbage collection. ACM SIGPLAN Notices **28**(4) (1993) 55–57
14. Microsystems, S.: The Java HotSpot Virtual Machine (2001) Technical White Paper.
15. Alpern, B., et al.: The Jalapeño virtual machine. IBM Systems Journal **39**(1) (Feb 2000) 211–238
16. Persson, M.: Java technology, IBM style: Garbage collection policies (May 2006) Garbage collection in the IBM SDK 5.0.
17. Ungar, D.M., Jackson, F.: An adaptive tenuring policy for generation scavengers. ACM Transactions on Programming Languages and Systems **14**(1) (1992) 1–27
18. Marion, S., Jones, R., Ryder, C.: Decrypting the Java gene pool: Predicting objects' lifetimes with micro-patterns. In: Proceedings of the International Symposium on Memory Management. (Oct 2007) (to appear)
19. Gil, Y., Maman, I.: Micro patterns in Java code. In: Object-Oriented Programming, Systems, Language and Applications (OOPSLA'05). (2005) 97–116
20. Chidamber, S., Kemerer, C.: A metrics suite for object oriented design. IEEE Transactions on Software Engineering **20**(6) (1994) 476–493
21. Spinellis, D.: ckjm—Chidamber and Kemerer Java metrics (2005) http://www.spinellis.gr/sw/ckjm/.
22. Chidamber, S.R., Darcy, D.P., Kemerer, C.F.: Managerial use of metrics for object-oriented software: An exploratory analysis. IEEE Transactions on Software Engineering **24**(8) (1998) 629–639
23. Blackburn, S.M., Garner, R., Hoffmann, C., Khang, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Eliot, J., Moss, B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications. (2006) 169–190
24. Alpern, B., et al.: The Jikes research virtual machine project: Building an open source research community. IBM Systems Journal **44**(2) (Feb 2005) 1–19
25. Witten, I.H., Frank, E.: Data Mining: Practical machine learning tools and techniques. 2nd edn. Morgan Kaufmann (2005)
26. Fleuret, F.: Fast binary feature selection with conditional mutual information. Journal of Machine Learning Research **5** (2004) 1531–1555
27. Bishop, C.M.: Pattern Recognition and Machine Learning. Springer (2007)