Testing and Validating Machine Learning Classifiers by Metamorphic Testing[☆]

Xiaoyuan Xie[a,d,e,*], Joshua W. K. Ho[b], Christian Murphy[c], Gail Kaiser[c], Baowen Xu[e], Tsong Yueh Chen[a]

[a]*Centre for Software Analysis and Testing, Swinburne University, Hawthorn, Vic 3122 Australia*
[b]*School of Information Technologies, The University of Sydney, NSW 2006, Australia; and NICTA, Australian Technology Park, Eveleigh, NSW 2015, Australia*
[c]*Department of Computer Science, Columbia University, New York NY 10027 USA*
[d]*School of Computer Science and Engineering, Southeast University, Nanjing 210096, China*
[e]*State Key Laboratory for Novel Software Technology & Department of Computer Science and Technology, Nanjing University, Nanjing 210093, China*

Abstract

Machine Learning algorithms have provided important core functionality to support solutions in many scientific computing applications - such as computational biology, computational linguistics, and others. However, it is difficult to test such applications because often there is no "test oracle" to indicate what the correct output should be for arbitrary input. To help address the quality of scientific computing software, in this paper we present a technique for testing the implementations of machine learning classification algorithms on which such scientific computing software depends. Our technique is based on an approach called "metamorphic testing", which has been shown to be effective in such cases. Also presented is a case study on a real-world machine learning application framework, and a discussion of how programmers implementing machine learning algorithms can avoid the common pitfalls discovered in our study. We also conduct mutation analysis and cross-validation, which reveal that our method has very high

[☆]A preliminary version of this paper was presented at the 9th International Conference on Quality Software (QSIC 2009) (Xie et al., 2009).

[*]Corresponding author. Tel.: +61.3.9214.8678; fax: +61.3.9819.0823.

Email addresses: xxie@groupwise.swin.edu.au (Xiaoyuan Xie), joshua@it.usyd.edu.au (Joshua W. K. Ho), cmurphy@cs.columbia.edu (Christian Murphy), kaiser@cs.columbia.edu (Gail Kaiser), bwxu@nju.edu.cn (Baowen Xu), tychen@groupwise.swin.edu.au (Tsong Yueh Chen)

effectiveness in killing mutants, and that observing expected cross-validation result alone is not sufficient to test for the correctness of a supervised classification program. Metamorphic testing is strongly recommended as a complementary approach. Finally we discuss how our findings can be used in other areas of computational science and engineering.

Keywords: Metamorphic Testing, Machine Learning, Test Oracle, Oracle Problem, Validation, Verification

1. Introduction

Machine Learning algorithms have provided important core functionality to support solutions in many scientific computing applications - such as computational biology, computational linguistics, and others. For instance, there are over fifty different real-world applications in computational science, ranging from facial recognition to computational biology, which use the Support Vector Machines classification algorithm alone (SVM Application List, 2006) . As these types of applications become more and more prevalent in society (Mitchell, 1983), ensuring their quality becomes more and more crucial.

Quality assurance of such applications presents a challenge because conventional software testing processes do not always apply: in particular, it is difficult to detect subtle errors, faults, defects or anomalies in many applications in these domains because there is no reliable "test oracle" to indicate what the correct output should be for arbitrary input. The general class of software systems with no reliable test oracle available is sometimes known as "non-testable programs" (Weyuker, 1982). Many of these applications fall into a category of software that Weyuker describes as *"Programs which were written in order to determine the answer in the first place. There would be no need to write such programs, if the correct answer were known"* (Weyuker, 1982).

The majority of the research effort in the domain of machine learning focuses on building more accurate models that can better achieve the goal of automated learning from the real world. However, to date very little work has been done on assuring the correctness of the software applications that perform machine learning. Formal proofs of an algorithm's optimal quality do not guarantee that an application implements or uses the algorithm correctly, and thus software testing is necessary.

To help address the quality of scientific computing software, this paper presents a technique for testing implementations of the supervised machine learning algorithms on which such software depends. Our technique is based on an approach called "metamorphic testing" (Chen et al., 1998), which uses properties of functions such that it is possible to predict expected changes to the output for particular changes to the input, based on so-called "metamorphic relations" between sets of inputs and their corresponding outputs. Although the correct output cannot be known in advance, if the change is not as expected, then a defect must exist.

In our approach, we first enumerate the metamorphic relations that such algorithms would be expected to demonstrate, then for a given implementation determine whether each relation is a necessary property to reveal program correctness. If it is, then failure to exhibit the relation indicates a defect, that is, they can be used for the purpose of testing. If it is not a necessary property, that is, although these properties would still be *anticipated* to hold in the classification algorithms we investigate, some of them could conceivably be violated without indicating a defect in the implementation, they can instead be used for validation. In such case, a violation of the property may or may not indicate a defect, but still represents a deviation from "expected" behavior.

In addition to presenting our technique, we describe a case study we performed on the real-world machine learning application framework Weka (Witten and Frank, 2005), which is used as the foundation for such computational science tools as BioWeka (Gewehr et al., 2007) in bioinformatics. We also discuss how our findings can be of use to other areas of computational science and engineering, such as computational linguistics.

The rest of this paper is organized as follows: Section 2 supplies background information about machine learning and introduces the specific algorithms that are evaluated. Section 3 discusses the metamorphic testing approach and the specific metamorphic relations used for testing machine learning classifiers. Section 4 presents the results of case studies demonstrating that the approach can find defects in real-world machine learning applications. Section 5 discusses empirical studies that use mutation testing to systematically insert defects into the source code, and measures the effectiveness of metamorphic testing. Section 6 presents related work, and Section 7 concludes.

2. Background

In this section, we present some of the basics of machine learning and the two algorithms we investigated (k-Nearest Neighbors and Naïve Bayes Classifier) (we previously considered Support Vector Machines in Murphy et al. (2008)), as well as the terminology used. Readers familiar with machine learning may skip this section.

One complication in our work arose due to conflicting technical nomenclature: “testing”, “regression”, “validation”, “model” and other relevant terms have very different meanings to machine learning experts than they do to software engineers. Here we employ the terms “testing”, “regression testing”, and “validation” as appropriate for a software engineering audience, but we adopt the machine learning sense of “model”, as defined below.

2.1. Machine Learning Fundamentals

In general, input to a **supervised** machine learning application consists of a set of **training data** that can be represented by two vectors of size k. One vector is for the k training samples $S = <s_0, s_1, ..., s_{k-1}>$ and the other is for the corresponding **class labels** $C = <c_0, c_1, ..., c_{k-1}>$. Each sample $s \in S$ is a vector of size m, which represents m features from which to learn. Each label c_i in C is an element of a finite set of class labels, that is, $c \in L = \{l_0, l_1, ..., l_{n-1}\}$, where n is the number of possible class labels.

```
27,81,88,59,15,16,88,82,41,17,81,98,42, ..., 0
15,70,91,41, 5, 3,65,27,82,64,58,29,19, ..., 0
22,72,11,92,96,24,44,92,55,11,12,44,84, ..., 1
82, 3,51,47,73, 4, 1,99, 1,51,84, 1,41, ..., 0
57,77,33,86,89,77,61,76,96,98,99,21,62, ..., 1
...
```

Figure 1: Example of part of a data set used by supervised ML classifier algorithms

Figure 1 shows a small portion of a training data set that could be used by supervised learning applications. The rows represent samples from which to learn, as comma-separated attribute values; the last number in each row is the label.

Supervised ML applications consist of two phases. The first phase (called the **training phase**) analyzes the training data; the result of this analysis is a **model** that attempts to make generalizations about how the attributes relate to the label. In the second phase (called the **testing phase**), the model is applied to another, previously-unseen data set (the **testing data**) where the labels are unknown. In a classification algorithm, the system attempts to predict the label of each individual example. That is, the testing data input is an unlabeled test case t_s, and the aim is to determine its class label c_t based on the data-label relationship learned from the set of training samples S and the corresponding class labels C, where $c_t \in L$.

2.2. Algorithms Investigated

This paper only investigates supervised learning applications. Within the area of supervised learning, we particularly focus on programs that perform classification, since classification is one of the central tasks in machine learning. The work presented here has focused on the *k*-Nearest Neighbors classifier and the Naïve Bayes Classifier, which were chosen because of their extensive use throughout the ML community. However, it should be noted that the problem description and techniques described below are not specific to any particular algorithm, and as shown in our previous work (Chen et al., 2009; Murphy et al., 2008), our results are applicable to the general case.

In ***k*-Nearest Neighbors** (*k*NN), for a training sample set *S*, suppose each sample has *m* attributes, $<att_0, att_1, ..., att_{m-1}>$, and there are *n* classes in *S*, $\{l_0, l_1, ..., l_{n-1}\}$. The value of the test case t_s is $<a_0, a_1, ..., a_{m-1}>$. *k*NN computes the distance between each training sample and the test case. Generally *k*NN uses the Euclidean Distance: for a sample $s_i \in S$, the value of each attribute is $<sa_0, sa_1, ..., sa_{m-1}>$, and the distance formula is as follows:

$$dist(s_i, t_s) = \sqrt{\sum_{j}^{m-1} (sa_j - a_j)^2}.$$

After sorting all the distances, *k*NN selects the *k* nearest ones and these samples are considered the *k* nearest neighbors of the test case. Then *k*NN calculates the proportion of each label in the *k* nearest neighbors, and the label with the highest proportion is assigned as the label of the test case.

In the **Naïve Bayes Classifier** (NBC), for a training sample set *S*, suppose each sample has *m* attributes, $<att_0, att_1, ..., att_{m-1}>$, and there are *n* classes in *S*, $\{l_0, l_1, ..., l_{n-1}\}$. The value of the test case t_s is $<a_0, a_1, ..., a_{m-1}>$. The label of *ts* is called l_{ts}, and is to be predicted by NBC.

NBC computes the probability of l_{ts} belonging to l_k, when each attribute value of t_s is $<a_0, a_1, ..., a_m>$. NBC assumes that attributes are conditionally independent with one another given the class label, therefore we have the equation:

$$P(l_{ts} = l_k \mid a_0 a_1 ... a_{m-1}) = \frac{P(l_k) \prod_j P(a_j \mid l_{ts} = l_k)}{\sum_i P(l_i) \prod_j P(a_j | l_{ts} = l_i)}$$

After computing the probability for each $l_i \in \{l_0, l_1, ..., l_{n-1}\}$, NBC assigns the label l_k with the highest probability, as the label of test case t_s.

Generally NBC uses a normal distribution to compute $P(a_j \mid l_{ts} = l_k)$. Thus NBC trains the training sample set to establish a distribution function for each element att_j of vector $<att_0, att_1, ..., att_{m-1}>$ in each $l_i \in \{l_0, l_1, ..., l_{n-1}\}$, that is, for all samples with label $l_i \in \{l_0, l_1, ..., l_{n-1}\}$, it calculates the mean value μ and mean square deviation σ of att_j in all samples with l_i. Then a probability density function is constructed for a normal distribution with μ and σ.

For test case t_s with m attribute values $<a_0, a_1, ..., a_{m-1}>$, NBC computes the probability of $P(a_j \mid l_{ts} = l_k)$ using a small interval δ to calculate the integral area. With the above formulae NBC can then compute the probability of l_{ts} belonging to each l_i and choose the label with the highest probability as the classification of t_s.

3. Approach

Our approach is based on the concept of metamorphic testing (Chen et al., 1998), summarized below. To facilitate that approach, we must identify the relations that the algorithms are expected to exhibit between sets of inputs and sets of outputs. Once those relations have been determined, we then analyze the algorithms to decide whether the relations are necessary properties to indicate correctness during testing; that is to say, if the implementation does not exhibit that property, then there **is** a defect. If the relation is *not* a necessary property, it can still be used for for the purpose of validation, that is, whether the algorithm satisies the requirement.

3.1. Metamorphic Testing

One popular technique for testing programs without a test oracle is to use a "pseudo-oracle" (Davis and Weyuker, 1981), in which multiple implementations of an algorithm process the same input and the results are compared; if the results are not the same, then one or both of the implementations contains a defect. This is not always feasible, though, since multiple implementations may not exist, or they may have been created by the same developers, or by groups of developers who are prone to making the same types of mistakes (Knight and Leveson, 1986).

However, even without multiple implementations, often these applications exhibit properties such that if the input were modified in a certain way, it should be possible to predict the new output, given the original output. This approach is known as metamorphic testing. Metamorphic testing can be implemented very easily in practice. The first step is to identify a set of properties ("metamorphic relations", or **MRs**) that relate multiple pairs of inputs and outputs of the target program. Then, pairs of **source** test cases and their corresponding **follow-up** test cases are constructed based on these MRs. We then execute all these test cases using the target program, and check whether the outputs of the source and follow-up test cases satisfy their corresponding MRs.

A simple example of a function to which metamorphic testing could be applied would be one that calculates the standard deviation of a set of numbers. Certain transformations of the set would be expected to produce the same result. For instance, permuting the order of the elements should not affect the calculation; nor would multiplying each value by -1, since the deviation from the mean would still be the same.

Furthermore, there are other transformations that will alter the output, but in a predictable way. For instance, if each value in the set is multiplied by 2, then the standard deviation should be twice as much as that of the original set, since the values on the number line are just "stretched out" and their deviation from the mean becomes twice as great. Thus, given one set of numbers (the source test cases), we can use these metamorphic relations to create three more sets of follow-up test cases (one with the elements permuted, one with each multiplied by -1, and another with each multiplied by 2); moreover, given the result of only the source test case, we can predict the others.

It is not hard to see that metamorphic testing is simple to implement, effective, easily automatable, and independent of any particular programming language. Further, for the identification of the MRs, which is the most crucial step in metamorphic testing, there are several principles of both white-box and black-box can be followed, such as, logical hierarchy, difference in execution traces, user's profiles, etc (Chen et al., 2004).

For example, based on the principle of "difference in execution traces", we intend to select MRs with more differences between the execution traces of source test cases and follow-up test cases. Here is an illustration: the Shortest-Path program *SP* accepts 3 parameters as inputs: a given graph *G*, a starting node *s*, and an ending node *e*. *SP(G, s, e)* returns the shortest path between *s* and *e*. Let us consider the two following MRs as examples: MR1: $|SP(G, s, e)| = |SP(G, s, m)| + |SP(G, m, e)|$, where *m* denotes a visited node between *s* and *e* returned by *SP(G, s, e)*. And MR2: $|SP(G, s, e)| = |SP(G, e, s)|$. Apparently, these two MRs will execute different path-pairs (source path and follow-up path), and a path pair with more difference is preferred as a better MR. Of course in order to decide which MR will result in more execution difference, we can just run the program and collect the real coverage information. But we also can acquire this information simply by analysing the mechanism of the algorithm, without any execution. Suppose the algorithm is a forward-search algorithm, MR2 is likely to lead to more execution difference. However if the algorithm is a 2-way search method, MR2 will not necessarily yield more execution difference.

Apart from the above principles, we can also harness the domain knowledge. This is a useful feature since in scientific computing the programmer may, in fact, also be the domain expert and will know what properties of the program will be used more heavily or are more critical. Perhaps more importantly, it is clear that metamorphic testing can be very useful in the absence of a test oracle, that is, when the correct output cannot be verified: regardless of the input values, if the metamorphic relations are violated, then there is likely a defect in the implementation.

3.2. Metamorphic Relations

In previous work (Murphy et al., 2008), we broadly classified six types of metamorphic relations (MRs) applicable in general to many different types of machine learning applications, including both supervised and unsupervised ML. In this work, however, our approach calls for focusing on the specific metamorphic relations of the application under test; we would expect that we could then create more follow-up test cases and conceivably reveal more defects than by using more general MRs. In particular, we define the MRs that we anticipate classification algorithms to exhibit, and define them more formally as follows.

MR-0: Consistence with affine transformation. The result should be the same if we apply the same arbitrary affine transformation function, $f(x) = kx + b$, $(k \neq 0)$ to every value x to any subset of features in the training data set S and the test case t_s.

MR-1.1: Permutation of class labels. Assume that we have a class-label permutation function *Perm*() to perform one-to-one mapping between a class label in the set of labels L to another label in L. If the source case result is l_i, applying the permutation function to the set of corresponding class labels C for the follow-up case, the result of the follow-up case should be $Perm(l_i)$.

MR-1.2: Permutation of the attribute. If we permute the m attributes of all the samples and the test data, the result should remain unchanged.

MR-2.1: Addition of uninformative attributes. An uninformative attribute is one that is equally associated with each class label. For the source input, suppose we get the result $c_t = l_i$ for the test case t_s. In the follow-up input, we add an uninformative attribute to S and respectively a new attribute in s_t. The choice of the actual value to be added here is not important as this attribute is equally associated with the class labels. The output of the follow-up test case should still be l_i.

MR-2.2: Addition of informative attributes. For the source input, suppose we get the result $c_t = l_i$ for the test case t_s. In the follow-up input, we add an informative attribute to *S* and t_s such that this attribute is strongly associated with class l_i and equally associated with all other classes. The output of the follow-up test case should still be l_i.

MR-3.1: Consistence with re-prediction. For the source input, suppose we get the result $c_t = l_i$ for the test case t_s. In the follow-up input, we can append t_s and c_t to the end of *S* and *C* respectively. We call the new training dataset *S'* and *C'*. We take *S'*, *C'* and t_s as the input of the follow-up case, and the output should still be l_i.

MR-3.2: Additional training sample. For the source input, suppose we get the result $c_t = l_i$ for the test case t_s. In the follow-up input, we duplicate all samples in *S* and *L* which have label l_i. The output of the follow-up test case should still be l_i.

MR-4.1: Addition of classes by duplicating samples. For the source input, suppose we get the result $c_t = l_i$ for the test case t_s. In the follow-up input, we duplicate all samples in *S* and *C* that do not have label l_i and concatenate an arbitrary symbol "*" to the class labels of the duplicated samples. That is, if the original training set *S* is associated with class labels <*A, B, C*> and l_i is *A*, the set of classes in *S* in the follow-up input could be <*A, B, C, B*, C**>. The output of the follow-up test case should still be l_i. Another derivative of this metamorphic relation is that duplicating all samples from any number of classes which do not have label l_i will not change the result of the output of the follow-up test case.

MR-4.2: Addition of classes by re-labeling samples. For the source input, suppose we get the result $c_t = l_i$ for the test case t_s. In the follow-up input, we re-label some of the samples in *S* and *C* which have label other than l_i and concatenate an arbitrary symbol "*" to their class labels. That is, if the original training set *S* is associated with class labels <*A, B, B, B, C, C, C*> and c_0 is *A*, the set of classes in *S* in the follow-up input may become <*A, B, B, B*, C, C*, C**>. The output of the follow-up test case should still be l_i.

MR-5.1: Removal of classes. For the source input, suppose we get the result $c_t = l_i$ for the test case t_s. In the follow-up input, we remove one entire class of samples in *S* of which the label is not l_i. That is, if the original training set *S* is associated with class labels <*A, A, B, B, C, C*> and l_i is *A*, the set of classes in *S* in the follow-up input may become <*A, A, B, B*>. The output of the follow-up test case should still be l_i.

MR-5.2: Removal of samples. For the source input, suppose we get the result $c_t = l_i$ for the test case t_s. In the follow-up input, we remove part of some of the samples in *S* and *C* of which the label is not l_i. That is, if the original training set *S* is associated with class labels <*A, A, B, B, C, C*> and l_i is *A*, the set of classes in *S* in the follow-up input may become <*A, A, B, C*>. The output of the follow-up test case should still be l_i.

3.3. Analysis of Relations for Classifiers

We do not formally prove here that all of these properties hold for both the *k*NN and NBC. Rather, we demonstrate here that some of the relations are **not**, necessary properties of the algorithms being implemented. For those MRs which are necessary properties, we can use them in software testing, but for the MRs demontrated as follows, we can still use them for the purpose of validation.

For *k*NN, five of the above metamorphic relations are not necessary properties but can instead be used for validation purposes. MR-1.1 (Permutation of class labels) is not a necessary property because of tiebreaking between two labels for prediction that are equally likely: permuting their order may change which one is chosen by the tiebreaker.

Additionally, MR-5.1 (Removal of classes) is not a necessary property. Suppose the predicted label of the test case is l_i. MR-5.1 removes a whole class of samples without label l_i. Consequently this will remove the same samples in the set of *k* nearest neighbors, and thus some other samples will be included in the set of *k* nearest neighbors. These samples may have any labels except the removed one, and so the likelihood of any label (except the removed one) may increase. Therefore there are two situations: (1) If in the *k* nearest neighbors of the source case, the proportion of l_i is not only the highest, but also higher than 50%, then in the follow-up prediction, no matter how the *k* nearest neighbors change, the predication will remain the same, because no matter which labels increase, the proportion of l_i will still be higher than 50% as well. Thus the prediction remains l_i. Now consider situation (2), in which in the *k* nearest neighbors of the source case, the proportion of l_i is the highest but lower or equal to 50%. Since the number of each survived label may increase, and the original proportion of l_i is lower or equal to 50%, it is possible that the proportion of some other label increases and becomes higher than l_i: thus, the prediction changes.

Similarly MR-2.2 (Addition of informative attributes), MR-4.1 (Addition of classes by duplicating samples), and MR-5.2 (Removal of samples) may not hold if the predicted label has a likelihood of less than 50%.

For the NBC, three of the metamorphic relations are not considered necessary properties, but can still be used for validation: MR-3.1 (Consistence with re-prediction), MR-4.2 (Addition of classes by re-labeling samples), and MR-5.2

(Removal of samples). MR-3.1 could not be proven as a necessary property, and thus is considered not necessary; the other two introduce noise to the data set, which could affect the result.

4. Case Studies

To demonstrate the effectiveness of metamorphic testing in validating machine learning applications, we applied the approach to Weka 3.5.7 (Witten and Frank, 2005). Weka is a popular open-source machine learning package that implements many common algorithms for data preprocessing, classification, clustering, association rule mining, feature selection and visualization. Due to its large range of functionality, it is typically used as a "workbench" for applying various machine learning algorithms. Furthermore, Weka is widely used as the back-end machine learning engine for various applications in computational science, such as BioWeka (Gewehr et al., 2007) for machine learning tasks in bioinformatics.

4.1. Experimental Setup

The data model in our experiments is as follows. In one source suite, there are *k* inputs. Each input_i has two parts: tr_i and t_i, in which tr_i represents the training sample set, and t_i represents the test case. In each training sample set tr_i and test case t_i, there are four attributes: $<A_0, A_1, A_2, A_3>$, and a label *L*. In our experiments, there are three labels, that is, $\{L_0, L_1, L_2\}$. The value for each attribute is within [1, 20]. We generate the tr_i and t_i values randomly, both in the value of the attribute and the label. The number of samples in tr_i is also randomly generated with a maximum of *n*.

This randomly generated data model does not encapsulate any domain knowledge, that is, we do not use any meaningful, existing training data for testing: even though those data sets are more predictable, they may not be sensitive to detecting faults. Random data may, in fact, be more useful at revealing defects (Duran and Ntafos, 1984).

For the source suite of *k* inputs, we perform a transformation according to the MRs and get *k* follow-up inputs for each MR-j. From running the *k* follow-up inputs and comparing the results between the source and the follow-up cases for the each MR-j, we try to detect faults in Weka or find a violation between the classifier under test and the anticipated properties of the classifier.

For each MR-j, we conducted several batches of experiments, and in each batch of experiments we changed the value of *k* (size of source suite) and *n* (max number of training samples). Intuitively the more inputs we tried (the higher is *k*), the more likely we are to find violations. Also, we would expect that with fewer samples in the training data set (the less is *n*), the less predictable the data are, thus the more likely we are to find faults.

4.2. Findings

Our investigation into the *k*NN and NBC implementations in Weka revealed that some NBC test cases caused violations in the necessary properties, indicating defects. In other cases, for both algorithms, metamorphic relations that could be used for validation were also violated, perhaps not indicating an actual defect but showing that the implementations could yield unexpected results and deviate from the behavior anticipated by scientific computing users.

4.2.1. k-Nearest Neighbors

None of the necessary properties of *k*NN were violated by our testing, but we did uncover violations in some of the other properties used during validation. Although these are not necessarily indicative of defects per se, they do demonstrate a deviation from what would normally be considered the expected behavior.

1. Calculating distribution. In the Weka implementation of *k*NN, a vector *distance* with the length of *numOfSamples* is used to record the distance between each sample from the training data and the test case to be classified. After determining the values in *distance*, Weka sorts it in ascending order, to find the nearest *k* samples from the training data, and then puts their corresponding labels into another vector *k-Neighbor* with the length of *k*.

Weka traverses *k-Neighbor*, computes the proportion of each label in it and records the proportions into a vector *distribution* with the length of *numOfClasses* as follows: Each element of vector *distribution* is initialized as 1/*numOfSamples*. It then traverses the array *k-Neighbor*, and for each label in *k-Neighbor*, it adds the weight of its distribution value (in our experiments, the weight is 1), that is, for each *i*, *distribution*[*k-Neighbor*[*i*].*label*] + 1. Finally, Weka normalizes the whole *distribution* vector.

Figure 2 shows two data sets, with the training data on the left, and the test case to be classified on the right. For the test case to be classified, the (unsorted) values in the vector *distance* are <11.40, 7.35, 12.77, 10.63, 13, 4.24>, and the values in *k-Neighbor* are <1, 2, 0>, assuming $k = 3$. The vector *distribution* is initialized as <1/6, 1/6, 1/6, 1/6, 1/6, 1/6>. After traversing the vector *k-Neighbor*, we get *distribution* = <1+1/6, 1+1/6, 1+1/6, 1/6, 1/6, 1/6> = <1.167, 1.167, 1.167, 0.167, 0.167, 0.167>. After the normalization, *distribution* = <0.292, 0.292, 0.292, 0.042, 0.042, 0.042>.

Figure 2: Sample data sets

@attribute Attr0 numeric	@attribute Attr0 numeric
@attribute Attr1 numeric	@attribute Attr1 numeric
@attribute Attr2 numeric	@attribute Attr2 numeric
@attribute Attr3 numeric	@attribute Attr3 numeric
@attribute Label {0,1,2,3,4,5}	@attribute Label {0,1,2,3,4,5}
@data	@data
11,3,9,4,0	9,5,8,15,0
4,8,10,11,2	
18,12,4,8,0	
1,11,6,18,0	
10,13,10,5,0	
7,2,10,14,1	

The issue here, as revealed by MR-5.1 (Removal of classes), is that labels that were non-existent in the training data samples have non-zero probability of being chosen in the vector *distribution*. Ordinarily one might expect that if a label did not occur in the training data, there would be no reason to classify a test case with that label. However, by initializing the *distribution* vector so that all labels are equally likely, even non-existent ones become possible. Although this is not necessarily an incorrect implementation, it does deviate from what one would normally expect.

2. Choosing labels with equal likelihood. Another issue is about the choice of the label when there are multiple labels with the same probability. Our testing indicated that in some cases, this method may lead to the violation in some MR transformations, particularly MR-1.1 (Permutation of class labels), MR-2.2 (Addition of informative attributes), and MR-4.1 (Addition of classes by duplicating samples).

Consider the example in Figure 2 above. To perform the classification, Weka chooses the first highest value in *distribution*, and assigns its label to the test case. For the above example, l_0, l_1, and l_2 all have the same highest proportion in *distribution*, so based on the order of the labels, the final prediction is l_0, since it is first.

However, if the labels are permuted (as in MR-1.1, for instance), then another labels with equal probability might be chosen if it happens to be first. This is not a defect per se (after all, if there are three equally-likely classifications and the function needs to return only one, it must choose somehow) but rather it represents a deviation from expected behavior (that is, the order of the data set shall not affect the computed outputs), one that could have an effect on an application using this functionality.

4.2.2. Naïve Bayes Classifier

Our investigation into NBC revealed a number of violations of MRs that indicate defects and could lead to unexpected behavior.

1. Loss of precision. Precision can be lost due to the treatment of continuous values. In a pure mathematical model, a normal distribution is used for continuous values. Apparently it is impossible to realize true continuity in a digital computer. To implement the integral function, for instance, it is necessary to define a small interval δ to calculate the area. In Weka, a variable called *precision* is used as the interval. The *precision* for att_j is defined as the average interval of all the values. For example, suppose there are 5 samples in the training sample set, and the values of att_j in the five samples are 2, 7, 7, 5, and 10 respectively. After sorting the values we have vector <2, 5, 7, 7, 10>. Thus *precision* = [(5-2) + (7-5) + (10-7)] / (1 + 1 + 1) = 2.67.

However, Weka rounds all the values x in both the training samples and test case with precision pr by using *round(x / pr) * pr*. These rounded values are used for the computation of the mean value μ, mean square deviation σ, and the probability $P(l_{ts} = l_k \mid a_0 a_1 ... a_{m-1})$. This manipulation means that Weka treats all the values within $((2k\text{-}1)^* pr/2, (2k\text{+}1)^* pr/2]$ as k^*pr, in which k is any integer.

This may lead to the loss of precision and our tests resulted in the violation of some MR transformations, particularly MR-0 (Consistence with affine transformation) and 5.1 (Removal of classes). As a reminder both of these are necessary properties.

There are also related problems of calculating integrals in Weka. A particular calculation determines the integral of a certain function from negative infinity to $t = x - \mu / \sigma$. When $t > 0$, a replacement is made so that the calculation becomes 1 minus the integral from t to positive infinity. However, this may raise an issue because in Weka, all these values are of the Java datatype “double”, which only has a maximum of 16 bits for the decimal fraction. It is very common that the value of the integral is very small, thus after the subtraction by 1.0, there may be a loss of precision. For example, if the integral I is evaluated to 0.0000000000000001, then 1.0 - I =0.9999999999999999. Since there are 16 bits of the number 9, in Java the double value is treated as 1.0. This also contributed to the violation of MR-0 (Consistence with affine transformation).

2. Calculating proportions of each label. In NBC, to compute the value of $P(l_{ts} = l_k \mid a_0 a_1 ... a_{m-1})$, we need to calculate $P(l_k)$. Generally when the samples are equally weighted, $P(l_k)$ = (number of samples with l_k) / (number of all the samples). However, Weka uses Laplace Accuracy by default, that is, $P(l_k)$ = (number of samples with l_k + 1) / (number of all the samples + number of classes).

For example, consider a training set with six classes and eight samples, whose labels as follows: $<l_0, l_0, l_1, l_1, l_1, l_2, l_3, l_3>$. In the general way of calculating the probability, the vector of proportions for l_0 to l_5 is <2/8, 3/8, 1/8, 2/8, 0/8, 0/8> = <0.25, 0.375, 0.125, 0.25, 0, 0>. However in Weka, using Laplace Accuracy, the vector of proportions for l_0 to l_5 becomes <(2+1)/(8+6), (3+1)/(8+6), (1+1)/(8+6), (2+1)/(8+6), (0+1)/(8+6), (0+1)/(8+6)> = <0.214, 0.286, 0.143, 0.214, 0.071, 0.071>. This difference caused a violation of MR-2.1 (Addition of uninformative attributes), which was also considered a necessary property.

3. Choosing labels. Last, there are problems in the principle of "choosing the first label with the highest possibility", as seen above for *k*NN. Usually the probabilities are different among different labels. However in Weka, since the non-existent labels in the training set have non-zero probability, those non-existent labels may conceivably share the same highest probability. This caused a violation of MR-1.1 (Permutation of class labels), which was considered a necessary property.

4.3. Discussion

4.3.1. Addressing Violations of Properties

Our experiments reported the violation of four MRs in *k*NN; however, none of these were necessary properties and are mostly related to the fact that the algorithm must return one result when it is possible that there is more than one "correct" answer. However, in NBC, we uncovered violations of some necessary properties, which indicate defects; the lessons learned here serve as a warning to others who are developing similar applications.

To address the issues in NBC related to the precision of floating point numbers, we suggest using the BigDecimal class in Java rather than the "double" datatype. A BigDecimal represents immutable arbitrary precision decimal numbers, and consists of an arbitrary precision integer unscaled value and a 32-bit integer scale. If zero or positive, the scale is the number of digits to the right of the decimal point. If negative, the unscaled value of the number is multiplied by ten to the power of the negation of the scale. The value of the number represented by the BigDecimal is therefore (unscaledValue * 10^{-scale}). Thus, it can help to avoid the loss of precision when doing "1.0 - x".

The use of Laplace Accuracy also led to some of the violations in the NBC implementation. Laplace Accuracy is used for the nominal attributes in the training data set, but Weka also treats the label as a normal attribute, because it is nominal. However, the label should be treated differently: as noted, the side effect of using Laplace Accuracy is that the labels that never show up in the training set also have some probability, thus they may interfere with the prediction, especially when the size of the training sample set is quite small. In some cases the predicted results are the non-existent labels. We suggest that the use of Laplace Accuracy should be set as an option, and the label should be treated as a special-case nominal attribute, with the use of Laplace Accuracy disabled.

4.3.2. More General Application

Our technique has been shown to be effective for these two particular algorithms, but the MRs listed above hold for all classification algorithms, and Murphy et al. (2008) shows that other types of machine learning (ranking, unsupervised learning, etc.) exhibit the same properties classification algorithms do; thus, the approach is feasible for other areas of ML beyond just *k*NN and NBC.

More importantly, the approach can be used to validate *any* application that relies on machine learning techniques. For instance, computational biology tools such as Medusa (Middendorf et al., 2005) use classification algorithms, and some entire scientific computing fields (such as computational linguistics (Manning and Schütze, 1999)) rely on machine learning; if the underlying ML algorithms are not correctly implemented, or do not behave as the user expects, then the overall application likewise will not perform as anticipated. As long as the user of the software knows the expected metamorphic relations, then the approach is simple and powerful to validate the implementation.

One emerging application of these supervised classifiers is in the area of clinical diagnosis using a combination of systems-level biomolecular data (e.g., microarrays or sequencing data) and conventional pathology tests (e.g., blood count, histological images, and clinical symptoms). It has been demonstrated that a machine learning approach of multiple data types can yield more objective and accurate diagnostic and prognostic information than conventional clinical approaches alone. However, for clinical adoption of this approach, these programs that implement machine learning algorithms must be rigorously verified and validated for their correctness and reliability (Ho et al., 2010). A mis-diagnosis due to a software fault can lead to serious, even fatal, consequences. Our case studies clearly demonstrated the importance of rigorous and systematic testing of this type of machine learning algorithm. Thus our proposed testing strategy based on metamorphic testing becomes even more crucial to improve the quality of one of the most critical parts in these kinds of applications

5. Empirical Studies

In the experimental study presented in Section 4, we applied the metamorphic relations from Section 3.2 to the *k*NN classifier and NBC classifier implementations in Weka-3.5.7. Through the violations of the necessary properties of NBC, we discovered defects in its implementation. Even though these real-world defects illustrate the effectiveness of our method in verification of programs that do not have test oracles, they cannot empirically show how powerful our method is. Thus, in this section, we conduct further experiments, aiming to investigate the effectiveness of our method in verification.

5.1. Experimental Setup

To gain an understanding of how effective metamorphic testing is at detecting defects in applications without test oracles, we use mutation testing to systematically insert defects into the applications of interest. Mutation testing has been shown to be suitable for evaluation of effectiveness, as experiments comparing mutants to real faults have suggested that mutants are a good proxy for comparisons of testing techniques (Andrews et al., 2005).

5.1.1. Mutant Generation

In our mutation analysis, we applied MuJava (Ma et al., 2005) to systematically generate mutants for Weka-3.5.7. MuJava is a powerful and automatic mutation analysis system, which can provide different options for mutant generation, such as creating “traditional mutants” (related to arithmatic operators, logical operators, etc.) or “class mutants” (Java-specific mutants, such as changing a variable’s scope or changing the type of a cast).

First, MuJava allows users to define which files need to be modified. Since Weka is large-scale software (the total source code is about 16.4M), and our experiments only focus on certain major functions of *k*NN and NBC, in order to exclude the equivalent mutants, we only selected files related to these two classifiers according to their hierarchy structure. Table 1 lists all the selected files in our mutation analysis for both *k*NN and NBC.

Secondly, MuJava provides various levels of mutants, including intra-method level, inter-method level, intra-class level, and inter-class level. In our experiments, we only focus on the intra-method level of mutants.

***k*NN**	**NBC**
weka.classifiers.lazy.IBk.java	weka.classifiers.bayes.NaiveBayes.java
weka.core.Attribute.java	weka.core.Attribute.java
weka.core.Instance.java	weka.core.Instance.java
weka.core.Instances.java	weka.core.Utils.java
weka.core.Utils.java	weka.core.Statistics
weka.core.neighboursearch.LinearNNSearch.java	weka.estimators.DiscreteEstimator.java
weka.core.neighboursearch.NearestNeighbourSearch.java	weka.estimators.Estimator.java
weka.core.NormalizableDistance.java	weka.estimators.KernelEstimator.java
weka.core.EuclideanDistance.java	weka.estimators.NormalEstimator.java

Table 1: Selected files for mutation analysis.

Operator	**Description**
AOR	Arithmetic Operator Replacement
ROR	Relational Operator Replacement
COR	Conditional Operator Replacement
SOR	Shift Operator Replacement Replace shift operators to other shift operators
LOR	Logical Operator Replacement
ASR	Short-Cut Assignment Operator Replacement

Table 2: Mutation operators used in experiment.

Thirdly, for the selection of mutation operators, we only consider those traditional ones at the method level. Since some mutation operators may lead to compilation errors or runtime exceptions, we did not adopt all the operators provided by MuJava. Table 2 lists the operators used in our experiments.

5.1.2. Selection and Modification of MRs

We use the technique of mutation analysis to investigate the fault-detection effectiveness of our method. Hence we need to adopt those MRs which are necessary properties for the classifier. For each necessary MR, if we find violations in certain mutants, we can declare that this mutant is killed by the MR, that is, the defect has been detected. The goal of the experiment is to calculate what percentage of the mutants are killed by the MRs, as a measure of the fault-detection effectiveness.

For NBC, we only select 9 MRs from Section 3.2 that have been proved as necessary properties, while for *k*NN, apart from the necessary MRs, we also mod-

***k* = 1**	***k* = 3**
MR-1.1 Permutation of class labels	MR-0 Consistence with affine transformation
MR-2.2 Addition of informative attributes	MR-1.2 Permutation of the attribute
MR-4.1 Addition of classes by duplicating samples	MR-2.1 Addition of uninformative attributes
MR-5.1 Removal of classes	MR-3.1 Consistence with re-prediction
MR-5.2 Removal of samples	MR-3.2 Additional training sample
	MR-4.2 Addition of classes by re-labeling samples

Table 3: Metamorphic relations for *k*NN used in mutation analysis.

MR-0 Consistence with affine transformation
MR-1.1 Permutation of class labels
MR-1.2 Permutation of the attribute
MR-2.1 Addition of uninformative attributes
MR-2.2 Addition of informative attributes
MR-3.2 Additional training sample
MR-4.1 Addition of classes by duplicating samples
MR-5.1 Removal of classes
MR-NBC Consistence with value permutation

Table 4: Metamorphic relations for NBC used in mutation analysis.

ify other MRs to make them become necessary properties, to fit for our mutation analysis. The detailed discussion of the necessity for all MRs is in the Appendix. Table 3 and Table 4 summarize the MRs used for *k*NN and NBC respectively in the mutation analysis for verification.

5.2. Empirical Results and Analysis

5.2.1. Metamorphic Testing Results

In the mutation analysis, we adopted 300 randomly generated inputs as source test inputs. Each test input consists of one training dataset and one test case, both of which have the same format used in the experiments described in Section 4.

In the previous experimental study, we found some real defects in the source code of the NBC classifier of Weka-3.5.7. Thus in the mutation analysis, in order to exclude the violations that are due to these real defects, we eliminated the test inputs which violated MRs in the original version of Weka-3.5.7. And we check the violated test pairs in mutants to make sure that they are really due to the modification, instead of the real defects.

	Metamorphic Relation										
Mutant	**0**	**1.1**	**1.2**	**2.1**	**2.2**	**3.1**	**3.2**	**4.1**	**4.2**	**5.1**	**5.2**
original	0	0	0	0	0	0	0	0	0	0	0
v1	0	1.67	7.67	27	5.67	0	0	0	0	6.67	3.67
v2	0	0	0	0	0	0	0	0	0	4.67	3
v3	0	0	0	0	42.67	0	0	0	0	4.67	3.67
v5	0	2.33	0	0	0	0	0	2.33	0	6.67	3
v6	0	11.33	0	26.33	37	0	0	0	0	2	0
v7	0	9.67	0	0	1.67	0	0	0	0	4	1.67
v9	0	9	0	0	3.33	8.33	0	41.67	0	5	2
v10	0	1.33	22.67	34.33	94.67	0	0	0	0	4.33	5.33
v12	0	10.33	0	0	1.67	0	0	0	0	4	1.67
v13	0	0.33	22.33	0	0	0	0	0	0	3	3
v15	0	0	16.33	0	13.67	0	0	0	0	3.33	2.33
v16	0	10	0	26.33	37	0	0	0	0	2	0
v17	0	13.67	0	0	0	0	0	0	0	0	0
v18	0	11	0	0	0	0	0	0	0	0	0
v19	0	9.33	0	26.33	37	0	0	0	0	2	0
v20	0	0	0	0	43.67	0	0	0	0	2.33	1.67
v21	0	1	0	42.67	24	0.67	0	0	0	3.67	4
v22	0	68.33	0	0	0	0	0	0	0	0	0
v24	0	62.67	0	0	0	0	0	0	0	0	0
TOTAL	**0**	**15**	**4**	**6**	**12**	**2**	**0**	**2**	**0**	**15**	**16**

Table 5: Effectiveness of metamorphic relations for *k*NN

We applied the mutation operators in Table 2 to all selected files in Table 1, and randomly generated 30 mutants for both *k*NN and NBC. After excluding the mutants that caused compilation and runtime exceptions, we obtained 24 valid mutants for *k*NN and 26 mutants for NBC. Among the 24 feasible mutants for *k*NN, 19 were killed by metamorphic testing, and 20 out of the 26 feasible mutants were killed for NBC.

Table 5 shows the effectiveness of all metamorphic relations in Table 3 for *k*NN. The listed versions are the mutants with violations, that is, those killed by some metamorphic relations. Each cell except the last line of Table 5 records the percentage of violated input pairs among all valid input pairs, for a particular pair of metamorphic relation and mutant version. The last line records the total number of killed mutants of the corresponding MR.

It can be seen from Table 5 that our method is very effective in killing mutants: 19 out of 24 mutants have been killed by the current source inputs and all 11 metamorphic relations. After examining the five surviving mutants, we discovered that three out of the five mutants are equivalent mutants with respect to the current source inputs, the parameters in the command line, and all the 11 metamorphic relations. The reason for the equivalent mutants is that Weka is a large-scale program; even though we have selected the related program files for mutation analysis, we do not target all the functionality in these files. The parameters we used in the command line and the metamorphic relations that we have enumerated are only targeted for certain properties of the program. Thus in the three mutants, the modified statements are not executed using the current source inputs, the parameters in command line, and all the 11 metamorphic relations. Hence the actual effectiveness is 90.5%(19 out of 21 mutants).

The results in Table 5 also show that different metamorphic relations have different performance in detecting program faults. Among all 11 MRs, MR-1.1 and MR-5.1 had the highest killing rate (15 out of 21, 71.4%), while MR-0, MR-3.2 and MR-4.2 had the lowest killing rate (0 out of 21).

We also investigated the average violation percentage of all MRs. Since we enumerated all the metamorphic relations only by means of the background knowledge of the classifier without referring to the source code of the Weka implementation, and we also generated all mutants and test inputs randomly, our metamorphic relations are hence unbiased to any mutants under investigation. In this way, the average violation percentage for the MRs over all mutants (including all the survived mutants) can be used as an effectiveness measurement of metamorphic testing, that is, how likely a test input pair (source test input and follow-up test input) on average will reveal a violation. From Table 5, we can calculate that for *k*NN, the average percentage is 3.65% for all the mutants in table.

Similarly, we investigate the effectiveness of each selected metamorphic relation in the mutation analysis for NBC. The results are presented in Table 6.

This table lists all mutants with violations. As above, each cell except the last line records the percentage of violated input pairs among all valid input pairs and the last line of the table records the total number of killed mutants of the corresponding MR.

For NBC, our method demonstrates a very good performance: 20 out of 26 mutants have been killed by the current source inputs and all nine metamorphic relations. And among the six surviving mutants, four are equivalent mutants with respect to the current source inputs, the parameters in the command line, and all the nine metamorphic relations. Hence the actual effectiveness is 20 out of 22

	Metamorphic Relation								
Mutant	**0**	**1.1**	**1.2**	**2.1**	**2.2**	**3.2**	**4.1**	**5.1**	**NBC**
original	0	0	0	0	0	0	0	0	0
v1	6.67	7.72	7.33	6.71	7.33	6.33	7.33	18.86	7
v2	0.74	0	0.33	0	0	0	0	0	0
v4	3.33	0	0	0	0	0	0	1.42	0
v5	45.56	30.87	29.67	25.5	28.33	37.67	52.33	67.97	29.33
v6	0.37	1.68	0.33	0	0	0	0.33	0	0
v7	4.82	10.07	1.33	2.35	1	1.33	1.33	7.12	1.33
v9	0.37	0	0	0	0	0	0	1.42	0
v11	0.37	1.34	0	0	0	0	0.33	0	0
v12	17.41	5.03	47	66.44	2.33	2.67	2.67	16.37	2.67
v15	81.85	80.54	79	90.27	87	79	79	89.32	79
v16	0.74	10.74	0	0	0	0	0	0.71	0
v17	50.74	53.02	50.33	41.61	50	51.33	50.33	63.7	50.33
v18	6.30	6.38	1.67	1.68	1.33	10.33	20.33	14.95	1
v19	7.41	8.73	0.67	1.34	0.33	4	8.33	11.39	0.67
v20	19.26	0.34	0.33	1.34	1	8.33	12	11.03	0.33
v21	33.33	0.67	0.33	0.67	0.67	0.33	0.33	13.88	0.33
v22	40	4.03	4	3.69	2.67	2.67	4	19.22	3.33
v24	0	2.35	0	0	0	0	0.33	0	0
v25	49.26	53.36	50	41.95	60.67	53.33	50	62.99	50
v26	0	2.01	0	0	0	0	0.33	0	0
TOTAL	**18**	**17**	**14**	**12**	**12**	**12**	**16**	**15**	**12**

Table 6: Effectiveness of metamorphic relations for NBC

mutants (90.9%).

Different from *k*NN, where some MRs kill none or a small number of mutants, in NBC, close to 50% of the mutants are killed by any given MR. For example, MR-0, which kills no mutants in *k*NN, can kill 18 mutants in NBC. And consequently the average violation percentage of the nine MRs over all the mutants in table is much higher than that in *k*NN. The average effectiveness of metamorphic testing for NBC is 11.19%.

5.2.2. Cross-validation Analysis

Apart from metamorphic testing, we also conducted cross-validation on these mutants. In the machine learning community, cross-validation is commonly used to assess how well the classification algorithms can model the classification process. In the context of applying cross-validation, it is often implicitly assumed that the implementation of the algorithm is correct. The focus is on the appropriateness of the classification algorithm to the given problem. While in our study, since we have investigated the well-known classification algorithms, we assume that they should perform well in cross-validation with reasonable datasets. Our major concern is whether the implementation of these algorithms is correct.

For most commonly used classification algorithms, a correct implementation for these algorithms should give a good cross-validation result when we use a reasonable dataset that contains discriminatory signals among samples of different classes; while an implementation with bad cross-validation performance justifies a further investigation which may lead to identification of software faults. However, in our experiments, we discovered that quite a few mutants can survive the cross-validation procedure, that is, the program that actually contains faults can still perform very well in cross-validation. As a consequence, this observation implies that proper software testing, particularly using the metamorphic testing technique proposed in this paper, is indispensable for these kind of machine learning applications.

In our experiments, we conducted k-fold cross-validation, which is a typical cross-validation method. In k-fold cross-validation, the original sample set is randomly partitioned into k subsets. Among the k subsets, a single subset is retained as the validation data for testing the classifier model, and the remaining ($k \leq 1$) subsets are used as training data. The cross-validation process is then repeated k times. The k results from the k folds then can be averaged or summarized (or otherwise combined) to produce a single estimation (McLachlan et al., 2004). In cross-validation, a classifier is simply evaluated in terms of its respective fraction of misclassified instances, noted as the error-rate. A lower error-rate means a better performance of a classifier. Usually an error-rate lower than 30% can be regarded as a reasonable one in a two-class classification problem.

We used some simulation data for our cross-validation analysis. The simulated datasets that we adopted have similar sizes and formats as the randomly generated data used in the mutation analysis. They were produced and used in another bioinformatics study (Ho et al., 2008) that simulates microarray gene expression data containing realistic noise characteristics. Each simulated dataset contains five numeric attributes and 100 samples comprising five classes of 20 samples each. The expression level of each attribute is simulated with a normal distribution $N(\mu, \sigma^2)$. The same value is used for variance ($\sigma^2 = 2$) in all simulated datasets, and μ varies with three different rules, referred as Rule-1, Rule-1.5 and Rule-2 respectively in the paper. Each rule is to multiply the data in consecutive classes with a normal distribution with the same σ^2 and different $\mu * \gamma$ where γ is a multiplication factor according to the current rule. The value of γ is assigned as 1, 1.5 and 2 in corresponding rules, in order to approximate the effect of observing no, medium, and large signals for distinguishing among different classes. We utilize 300 datasets of each rule, hence have a total of 900 datasets for the cross-validation experiment.

In our experiments we conducted 10-fold cross-validation, which is commonly used with each simulated dataset acting as the training dataset. Table 7 presents the results for *k*NN, while Table 8 shows the performance of NBC. In each table, we list both the original version and the mutants that were killed by metamorphic testing. Each cell records the average error-rate among all 300 datasets for the corresponding rule.

It can be seen from Tables 7 and 8 that, for both *k*NN and NBC, in the original program and most mutants, the error-rates of the three rules have the same trend: Rule-2 $\leq$ Rule-1.5 $\leq$ Rule-1. This result supports our intuition that datasets which contain more discriminatory signals can be used to train a classifier to acquire a higher predictive ability. The cross-validation results of the original program (without mutants) are within our expectation. In Rule-1, all data are simulated in the same way regardless of the class label, so an error rate of 80% is expected given that there is a one-in-five chance in randomly choosing a sample with the correct label, given that the dataset contains 100 samples with 5 class labels (that is, 20 samples per class).

If we consider the results of Rule-2 as an example, we can find that many mutants have similar reasonable error-rates as the original program:

1. For Table 7 on *k*NN, even though the original version performs better than any other mutants, and 12 mutants have quite a high error-rate using the

Mutants	**Rule-1**	**Rule-1.5**	**Rule-2**
original	80.07	3.95	0.06
v1	79.78	10.58	1.55
v2	80.07	3.95	0.06
v3	79.97	80	80
v5	80.05	6.75	0.15
v6	80	80	80
v7	79.72	5.79	1.23
v9	80.04	44.4	40.84
v10	80.09	100	100
v12	79.72	5.79	1.23
v13	80.07	5.66	0.21
v15	80.12	100	100
v16	80	80	80
v17	80	80	80
v18	80	80	80
v19	80	80	80
v20	80.18	80	80
v21	79.91	80	80
v22	100	100	100
v24	100	100	100

Table 7: Cross-validation error rate for *k*NN.

Mutants	**Rule-1**	**Rule-1.5**	**Rule-2**
original	80.07	3.36	0.07
v1	80.61	49.68	60
v2	79.97	3.35	0.07
v4	80.19	3.43	0.09
v5	80	80	80
v6	79.97	3.36	0.07
v7	80	80	80
v9	80	3.34	0.08
v11	79.97	3.36	0.07
v12	80.18	18.20	3.49
v15	100	100	100
v16	80	80	80
v17	80.21	81.09	91.2
v18	80	80	80
v19	80	80	80
v20	79.90	27.24	39.89
v21	80	80	5.35
v22	80	80	60
v24	79.95	3.37	0.07
v25	80.19	81.05	91.04
v26	79.97	3.36	0.07

Table 8: Cross-validation error rate for NBC.

dataset of Rule-2, there are still six mutants having an error-rate close to 1%, and one mutant that has an error rate of 40%, using the same dataset.

2. Table 8 shows that performance for mutants is even better in NBC. In NBC, the original version is no longer the only one having the lowest error-rate; there are five mutants that have the same error-rate (0.07%). Actually half of the mutants acquire quite good classification performance (error rate less than 5.5%), and one mutant has a reasonable error-rate (39.89%).

These experimental data reveal that some mutants can achieve relatively good performances in cross-validation, despite the fact that these mutants are faulty implementations of the algorithms. Actually cross-validation has been widely adopted as the main method for evaluating a supervised classifier system for decades; however, it was never designed for the purpose of either verification or validation. But, most practitioners in the machine learning field have relied on the cross-validation method to check the correctness of the implementation of the algorithm. In other words, an additional way to verify the correctness of the implementation is necessary. Because of the oracle problem, metamorphic testing becomes necessary and suitable in testing these supervised machine learning programs. In fact, metamorphic testing is very powerful in detecting faults even for mutants with very low error-rate. For example, v1 of *k*NN has an ASR mutant in the EuclideanDistance.java file, line 182. The modification is:

```
result = diff * diff; //correct:  result += diff * diff;
```

However in Table 7, v1 for Rule-2 has the error-rate as low as 1.55%. Fortunately metamorphic testing is able to kill this mutant. Table 5 shows that MR-1.1, MR-1.2, MR-2.1, MR-2.2, MR-5.1 and MR-5.2 all reveal this mutant.

This result shows that the cross-validation technique is not sufficient to test the correctness of a supervised classification program. It is strongly recommended to adopt MT as the complement to this technique in order to provide more confidence of the software quality.

6. Related Work

Although there has been much work that applies machine learning techniques to software engineering in general and software testing in particular (e.g., Briand (2008)), we are not currently aware of any other work in the reverse sense: applying software testing techniques to machine learning applications. ML frameworks such as Orange (Demsar et al., 2004) provide testing functionality but it is focused on comparing the quality of the results, and not evaluating the "correctness" of the implementations. Repositories of "reusable" data sets have been collected (e.g., Newman et al. (1998)) for the purpose of comparing result quality, that is, how accurately the algorithms predict, but not for the software engineering sense of testing (to reveal defects).

Applying metamorphic testing to situations in which there is no test oracle was first suggested in Chen et al. (1998) and is further discussed in Chen et al. (2002). Metamorphic testing has previously been shown to be effective in testing different types of machine learning applications (Murphy et al., 2008), and has recently been applied to testing specific scientific computation applications, such as in bioinformatics (Chen et al., 2009). The work we present here seeks to extend the previous techniques to scientific computation domains that rely on machine learning.

7. Conclusion

As noted in Kelly and Sanders (2008), "scientists want to do science" and do not want to spend time addressing the challenges of software development. Thus, it falls upon the software engineering community to develop simple yet powerful methods to perform testing and validation. Our contribution is a set of metamorphic relations for classification algorithms, as well as a technique that uses these relations to enable scientists to easily test and validate the machine learning components of their software; this technique is also applicable to problem-specific domains as well. We hope that our work helps to improve the quality of the software being developed in the fields of computational science and engineering.

Acknowledgments

This project is partially supported by an Australian Research Council Discovery Grant (ARC DP0771733), as well as the National Natural Science Foundation of China (90818027 and 60721002), the National High Technology Development Program of China (2009AA01Z147), and the Major State Basic Research Development Program of China (2009CB320703). Ho is supported by an Australian Postgraduate Award and a NICTA Research Project Award. Murphy and Kaiser are members of the Programming Systems Laboratory, funded in part by CNS-0905246, CNS-0717544, CNS-0627473, CNS-0426623 and EIA-0202063, and NIH grant 1U54CA121852-01A1.

Reference

Andrews, J. H., Briand, L. C., Labiche, Y., 2005. Is mutation an appropriate tool for testing experiments? In: Proc. of the 27th International Conference on Software Engineering (ICSE). pp. 402–411.

Briand, L., 2008. Novel applications of machine learning in software testing. In: Proc. of the 8th International Conference on Quality Software(QSIC). pp. 3–10.

Chen, T. Y., Cheung, S. C., Yiu, S., 1998. Metamorphic testing: a new approach for generating next test cases. Tech. Rep. HKUST-CS98-01, Dept. of Computer Science, Hong Kong Univ. of Science and Technology.

Chen, T. Y., Ho, J. W. K., Liu, H., Xie, X., 2009. An innovative approach for testing bioinformatics programs using metamorphic testing. BMC Bioinformatics 10, 24–36.

Chen, T. Y., Huang, D. H., Tse, T. H., Zhou, Z. Q., 2004. Case studies on the selection of useful relations in metamorphic testing. In: Proc. of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC 2004). pp. 569–583.

Chen, T. Y., Tse, T. H., Zhou, Z. Q., 2002. Fault-based testing without the need of oracles. Information and Software Technology 44 (15), 923–931.

Davis, M. D., Weyuker, E. J., 1981. Pseudo-oracles for non-testable programs. In: Proc. of the ACM Annual Conference. pp. 254–257.

Demsar, J., Zupan, B., Leban, G., Curk, T., 2004. Orange: From experimental machine learning to interactive data mining. Lecture Notes in Computer Science, 537–539.

Duran, J., Ntafos, S., 1984. An evaluation of random testing. IEEE Transactions on Software Engineering 10, 438–444.

Gewehr, J. E., Szugat, M., Zimmer, R., 2007. BioWeka - extending the Weka framework for bioinformatics. Bioinformatics 23 (5), 651–653.

Ho, J. W. K., Lin, M. W., Adelstein, S., dos Remedios, C. G., 2010. Customising an antibody leukocyte capture microarray for systemic lupus erythematosus: Beyond biomarker discovery. Proteomics - Clinical Applications in press.

Ho, J. W. K., Stefani, M., dos Remedios, C. G., Charleston, M. A., 2008. Differential variability analysis of gene expression and its application to human diseases. Bioinformatics 24, 390–398.

Kelly, D., Sanders, R., 2008. Assessing the quality of scientific software. In: Proc. of the 1st International Workshop on Software Engineering for Computational Science and Engineering(SECSE).

Knight, J., Leveson, N., 1986. An experimental evaluation of the assumption of independence in multi-version programming. IEEE Transactions on Software Engineering 12 (1), 96–109.

Ma, Y.-S., Offutt, J., Kwon, Y. R., June 2005. MuJava: An automated class mutation system. Journal of Software Testing, Verification and Reliability 15 (2), 97–133.

Manning, C. D., Schütze, H., 1999. Foundations of Statistical Natural Language Processing. The MIT Press.

McLachlan, G. J., Do, K.-A., Ambroise, C., 2004. Analyzing microarray gene expression data. Wiley.

Middendorf, M., Kundaje, A., Shah, M., Freund, Y., Wiggins, C. H., Leslie, C., 2005. Motif discovery through predictive modeling of gene regulation. Research in Computational Molecular Biology 3500, 538–552.

Mitchell, T., 1983. Machine Learning: An Artificial Intelligence Approach, Vol. III. Morgan Kaufmann.

Murphy, C., Kaiser, G., Hu, L., Wu, L., 2008. Properties of machine learning applications for use in metamorphic testing. In: Proc. of the 20th International Conference on Software Engineering and Knowledge Engineering (SEKE). pp. 867–872.

Newman, D. J., Hettich, S., Blake, C. L., Merz, C. J., 1998. UCI repository of machine learning databases. University of California, Dept of Information and Computer Science.

SVM Application List, 2006. `http://www.clopinet.com/isabelle/Projects/SVM/applist.html`.

Weyuker, E. J., November 1982. On testing non-testable programs. Computer Journal 25 (4), 465–470.

Witten, I. H., Frank, E., 2005. Data Mining: Practical Machine Learning Tools and Techniques, 2nd Edition. Morgan Kaufmann.

Xie, X. Y., Ho, J. W. K., Murphy, C., Kaiser, G., Xu, B. W., Chen, T. Y., 2009. Application of metamorphic testing to supervised classifiers. In: Proc. of the 9th International Conference on Quality Software(QSIC). pp. 135–144.

Appendix A.

In the appendix, we discuss the necessity of MRs for both *k*NN and NBC.

Appendix A.1. Necessary MRs for k-Nearest Neighbors

In our previous study (Xie et al., 2009), we adopted a total of 11 MRs on *k*NN, and 6 of them can be proved as necessary properties for *k*NN with any value of *k*.

1. MR-0: Consistence with affine transformation.
Each value in the training sample set and in the test case is transformed in this way: $kx+b$ ($k \neq 0$). Thus, this MR does not change the distance between s_i and t_s. The distance is:

$$dist(s_i', t_s') = \sqrt{\sum_j^m [(k * sa_j + b) - (k * a_j + b)]^2} = k\sqrt{\sum_j^m (sa_j - a_j)^2}.$$

Therefore MR-0 does not change the order in the *k* nearest neighbors and will still give the same prediction.

2. MR-1.2: Permutation of the attribute. It can be seen from the formula for calculating the distance that the result is not related to the order of the attributes. Thus, the permutation of the attributes will not affect the prediction result.

3. MR-2.1: Addition of uninformative features. In this MR, we add a new attribute att_m to both the samples and the test case and assign them with the same value. Suppose the value of att_m is a. It is obvious that MR-2.1 will not change the distance between any sample s_i and test case t_s. The new distance is:

$$dist(s_i', t_s') = \sqrt{\sum_j^{m-1} (sa_j - a_j)^2 + (a - a)^2} = \sqrt{\sum_j^{m-1} (sa_j - a_j)^2}.$$

Therefore MR-2.1 does not change anything in the *k* nearest neighbors and will still give the same prediction.

4. MR-3.1: Consistence with re-prediction. Suppose the label of a test case is l_i. We put the test case back into the training sample set, and from the distance formula we can know that the distance between the new sample and the test case is 0. Thus, the number of samples with label l_i in the *k* nearest neighbors increases by 1, and obviously the proportion of samples with label l_i will increases. Therefore, the follow-up prediction remains the same, l_i, as the source prediction.

5. MR-3.2: Additional training sample. Suppose the label of a test case is l_i. MR-3.2 duplicates the samples with label l_i in the training sample set. These new samples have the same value as the old ones, thus the number of samples with label l_i increases in the *k* nearest neighbors (maximum is being doubled). Meanwhile, the samples with other labels are excluded from the *k* nearest neighbors. Thus, the proportion of samples with label l_i increases (maximum is being doubled). Therefore the follow-up prediction remains the same, l_i, as the source predication.

6. MR-4.2: Addition of classes by re-labelling samples. Suppose the label of a test case is l_i. MR-4.2 renames parts of the samples, which have labels other than l_i. This will not change the value of the distance between each sample and test case. It just changes the label of the distance. Thus it changes the label in the k nearest neighbors. This will not result in any changes in the number and proportion of samples with label l_i. It only may decrease the number and the proportion of samples which have labels other than l_i; therefore it will not affect the follow-up prediction.

The remaining MRs can be proved as not necessary properties for any k. Actually, those MRs usually lead to changing the distance between the training samples and the test case, thus the ranking of all distances and the proportion in the k nearest neighbors also change correspondingly. However if we fix k as 1, these MRs all become necessary properties.

The reason is apparent. Since all the samples are sorted ascendingly by the distance to test case (no duplicated samples in our experiments), when $k = 1$, the kNN classifier just picks up the first sample, and makes its label as the predicted result. Even though the MRs may change the distance between the samples and the test case, and consequently change the ranking, they do not affect the top position of all the sorted distances. Thus if we assign $k = 1$, these MRs become necessary properties and can be adopted in our mutation analysis.

Appendix A.2. Necessary MRs for Naïve Bayes Classifier

For NBC, we adopted 12 MRs in our previous study, and 9 of them can be proved as necessary properties.

1. MR-0: Consistence with affine transformation. To implement the calculation of an integral in a digital computer, it is necessary to define a small interval δ to calculate the area. In Weka, they use a variable called *Precision* as the interval. The *Precision* for att_j is defined as the average interval of all the values. For example, suppose there are five samples in the training sample set, and the values of att_j in the five samples are 2, 7, 7, 5, and 10. After sorting the values we have 2, 5, 7, 7, 10. Thus, *Precision* = [(5-2) + (7-5) + (10-7)] / (1+1+1) = 2.67. If all the values are the same, *Precision* (abbreviated *pr*) equals its default value, 0.01. In the computation, Weka rounds all the values x in both the training samples and the test case with *pr* as $rint(x / pr) * pr$, in which *rint* is the function to round to the nearest integer. This manipulation means that Weka treats all the values within ((2*k*-1)* *pr*/2, (2*k*+1)* *pr*/2] as *k***pr*, in which *k* is any integer. This manipulation may lead to a loss of precision; however, it provides a mechanism to disperse the continuous values in the mathematic model, in order to be make the model suitable for computer implementation.

In Weka, the small interval δ is the magnitude of precision. According to formula for calculating area, we have:

$$P(a_j \mid l_{ts} = l_k) = \frac{1}{\sigma\sqrt{2\pi}} \int_{a_j - pr/2}^{a_j + pr/2} e^{-(x - \mu)^2 / 2\sigma^2}\, dx.$$

In MR-0, each value *x* in the training set and the test case are transformed in this way: $\varphi = k*x+b$ ($k \neq 0$). According to the calculation of *pr*, *pr*' is set to be *k***pr* + *b*. According to the formula of mean value μ and mean square deviation σ, we have $\mu' = k*\mu + b$, and $\sigma' = k*\sigma$. And the formula for probability is as follows:

$$P(k * a_j + b \mid l_{ts} = l_k) = \frac{1}{\sigma'\sqrt{2\pi}} \int_{k*a_j+b-k*pr/2}^{k*a_j+b+k*pr/2} e^{-(\varphi - \mu')^2 / 2\sigma'^2}\, d\varphi$$

by substituting σ' with $k * \sigma$, and μ' with $k * \mu + b$, we have:

$$P(k * a_j + b \mid l_{ts} = l_k) = \frac{1}{k\sigma\sqrt{2\pi}} \int_{k*a_j+b-k*pr/2}^{k*a_j+b+k*pr/2} e^{-(\varphi - k\mu - b)^2 / 2k\sigma^2}\, d\varphi$$

by substituting φ with $k * x + b$, we have:

$$P(k * a_j + b \mid l_{ts} = l_k) = \frac{1}{k\sigma\sqrt{2\pi}} \int_{a_j-pr/2}^{a_j+pr/2} e^{-(kx + b - k\mu - b)^2 / 2k^2\sigma^2}\, d(kx + b)$$

$$\Longrightarrow P(k * a_j + b \mid l_{ts} = l_k) = \frac{1}{\sigma\sqrt{2\pi}} \int_{a_j-pr/2}^{a_j+pr/2} e^{-(x - \mu)^2 / 2\sigma^2}\, dx = P\,(a_j \mid l_{ts} = l_k).$$

It can be seen from the above formula that after the transformation, the probability will not change, thus the prediction result will not change either.

2. MR-1.1: Permutation of class labels. This MR reflects a key property of mathematical function such as NBC that the output of the classifier is deterministic, and is not affected by random permutation.

3. MR-1.2: Permutation of the attribute. It is known that in NBC, we assume all the attributes are independent, thus we have the following formula:

$$P(l_{ts} = l_k \mid a_0 a_1 \ldots a_{m-1}) = \frac{P(l_k) \prod\limits_j P(a_j \mid l_{ts} = l_k)}{\sum_i P(l_i) \prod_j P(a_j \mid l_{ts} = l_i)}$$

Therefore, changing the attribute order will not affect the prediction result.

Actually, it can be concluded that *all* classifiers should have a consistent result in this MR, assuming the attributes are independent to each other.

4. MR-2.1: Addition of uninformative features. In this MR, we add a new attribute att_m with identical value to both the samples and the test case. Suppose the value of att_m is a. For each $l_k \in \{l_0, l_1, ..., l_{n-1}\}$, the probability $P(l_{ts} = l_k \mid a_0a_1...a_{m-1})$ can be re-written in the following way:

$$P(l_{ts} = l_k \mid a_0a_1...a_m) = \frac{P(l_{ts} = l_k) \prod_j P(a_j \mid l_{ts} = l_k) * P(att_m = a \mid l_{ts} = l_k)}{\sum_i P(l_{ts}=l_i) \prod_j P(a_j|l_{ts} = l_i) * P(att_m=a_m|l_{ts}=l_i)}$$

Since the new attribute att_m has the same value a in all the samples, the mean value $\mu = a$ and the mean square deviation $\sigma = 0$. Thus the $P(att_m = a \mid l_{ts} = l_k)$ part is equal to 1 for all the $l_k \in \{l_0, l_1, ..., l_{n-1}\}$.

In Weka, since it is infeasible for computer to deal with the normal distribution with $\sigma = 0$, they give σ a default minimum of $pr/2*3$. Thus for each $l_k \in \{l_0, l_1, ..., l_{n-1}\}$, the numerator in the formula above will be changed by multiplying a constant value $P(att_m = a \mid l_{ts} = l_k)$, which is a little less than 1.

It follows that the probability for each $l_k \in \{l_0, l_1, ..., l_{n-1}\}$ changes in the same way. Thus the order of the probabilities will not change; consequently the prediction in the follow-up cases will remain the same as the one in the source cases.

5. MR-2.2: Addition of informative features. In this MR, we add a new attribute att_m to both the samples and the test case and assign the samples having the same label with the same value; meanwhile, we assign the new attribute's value in the test case as the one of its predicted label. For example, suppose there are three classes in the training samples, $\{l_0, l_1, l_2\}$, and the predicted label of the test case is l_0. In the MR-2.2 transformation, we add a new attribute and make it different among different classes, that is, for samples with l_0, the $att_m = a$; for samples with l_1, the $att_m = b$; for samples with l_2, the $att_m = c$; and for the test case, the $att_m = a$.

Since the denominator in the formula for each $l_k \in \{l_0, l_1, ..., l_{n-1}\}$ are the same, only the numerator will affect the result.

For l_0, the mean value of att_m is $\mu = a$; the mean square deviation of att_m is $\sigma = \delta$ (since it is hard to deal with a normal distribution with $\sigma = 0$, we assign a very small number to σ).

For l_1, the mean value of att_m is $\mu = b$; the mean square deviation of att_m is $\sigma = \delta$.

For l_2, the mean value of att_m is $\mu = c$; the mean square deviation of att_m is $\sigma = \delta$.

Thus the numerator in the formula for l_0 is multiplied by a value of $P(att_m = a \mid l_{ts} = l_0)$, which is quite close to 1. Also, the numerator in the formula for l_1 is multiplied by a value of $P(att_m = b \mid l_{ts} = l_1)$, which is quite close to 0. Last, the numerator in the formula for l_2 is multiplied by a value of $P(att_m = c \mid l_{ts} = l_2)$, which is quite close to 0.

Therefore the former highest possibility almost remains the same, while the other two decrease dramatically. Consequently the follow-up prediction will remain the same as in the source case.

6. MR-3.2: Additional training sample. Suppose the label of test case is l_i. MR-3.2 duplicates the samples with label l_i in the training data set. Those new samples have the same value as the old ones, thus the mean value and the mean square deviation of each attribute in l_i will not change. Meanwhile, the mean square deviation of each attribute in other labels will not change either. The only change is the proportion of each $l_k \in \{l_0, l_1, ..., l_{n-1}\}$: $P(l_{ts} = l_k)$; that is, $P(l_{ts} = l_i)$ increases, while $P(l_{ts} = l_k)$ for the other labels decreases.

Therefore the probability of t_s belonging to l_i increases, while the probability of t_s being one of the other labels decreases. The prediction is still l_i, as in the source case.

7. MR-4.1: Addition of classes by duplicating samples. Suppose we have labels $\{l_0, l_1, ..., l_{n-1}\}$, the number of each distinct label $l_i \in \{l_0, l_1, ..., l_{n-1}\}$ in the training sample set is *count*[*i*], and its corresponding proportion is *proportion*[*i*]. For each $l_i \in \{l_0, l_1, ..., l_{n-1}\}$, the mean value of att_j is μ_{ij}; the mean square deviation is σ_{ij}. Suppose the prediction in source case is l_k. Thus in the MR-4.1 transformation, we duplicate all samples with $l_i \in \{l_0, l_1, ..., l_{n-1}\}$ $(i \neq k)$ and rename them as l_i'. After duplication the μ_{ij} and σ_{ij} for the original labels remain the same value as the ones in the source case. The only change is the *proportion*[], which is as follows:

$$proportion'[i] = proportion[i] * \frac{\sum_{0}^{m-1} count[i]}{count[0] + 2\sum_{1}^{m-1} count[i]}$$

And for the new added label l_i', their μ, σ and *proportion*[] values are all the same for l_i. Therefore *proportion*[0] remains the highest value, and the prediction will not change in the follow-up case.

8. MR-5.1: Removal of classes. This MR transformation only changes the proportion of each class, rather than changing the distribution in each survived class. Suppose we have labels $\{l_0, l_1, ..., l_{n-1}\}$, the number of each distinct label $l_i \in \{l_0, l_1, ..., l_{n-1}\}$ in the training sample set is *count*[*i*], and its corresponding proportion is *proportion*[*i*]. For each $l_i \in \{l_0, l_1, ..., l_{n-1}\}$, the mean value of att_j is μ_{ij}; the mean square deviation is σ_{ij}. Suppose the prediction in the source case is l_0, and l_2 is the label being removed. Thus, after transformation, the μ and σ for each survived label remain the same as in the source case. The only change is the *count*[i] and the *proportion*[i], which changes as follows:

$$proportion'[i] = proportion[i] * \frac{\sum_{0}^{m-1} count[i]}{\sum_{0}^{m-1} count[i] - count[2]}$$

Therefore, the prediction remains the same as in the source case.

9. MR-NBC: Consistence with value permutation. NBC assumes that all the attributes are independent. If we permute the value of $attr_j$ among all samples with label l_i, this will not change the $P(attr_j \mid l_{ts} = l_i)$. Because for each $l_i \in \{l_0, l_1, ..., l_{n-1}\}$, the mean value μ_{ij} of att_j, and the mean square deviation σ_{ij} will not change in this permutation. Thus MR-NBC will not change the prediction made in the source case.