Applying Combinatorial Testing to Data Mining Algorithms

Jaganmohan Chandrasekaran, Huadong Feng, Yu Lei
Department of Computer Science & Engineering
University of Texas at Arlington
Arlington, USA
{jaganmohan.chandrasekaran, huadong.feng}@mavs.uta.edu,
ylei@cse.uta.edu

D. Richard Kuhn, Raghu Kacker
Information Technology Lab
National Institute of Standards and Technology
Gaithersburg, USA
{khun, raghu.kacker}@nist.gov

***Abstract*— Data mining algorithms are used to analyze and discover useful information from data. This paper presents an experiment that applies Combinatorial Testing (CT) to five data mining algorithms implemented in an open-source data mining software called WEKA. For each algorithm, we first run the algorithm with 51 datasets to study the impact different datasets have on the test coverage. We select one dataset that achieves the highest branch coverage. Next we construct positive and negative combinatorial test sets of configuration options and execute each test set with the selected dataset. Test effectiveness is measured using branch and mutation coverage. Our results suggest that when testing data mining algorithms: (1) larger datasets do not necessarily achieve higher coverage than smaller datasets; (2) test coverage increases progressively slower as test strength increases; and (3) branch coverage correlates well with mutation coverage.**

***Keywords*—Combinatorial Testing; Data mining; Machine learning; Input parameter modeling; Branch coverage; Mutation testing;**

I. INTRODUCTION

Big data applications are becoming more popular as large amounts of data are generated and collected in virtually every domain, e.g., e-commerce, social networking, and scientific computing [11,12]. These applications typically employ data mining algorithms to analyze data and discover useful information. Data mining algorithms include supervised learning algorithms, and un-supervised learning algorithms to perform tasks such as classification, clustering, association rule mining [36].

In this paper, we present an experiment that applies combinatorial testing (or CT) to software that implements data mining algorithms. Testing data mining algorithms has several challenges. First, data mining software typically involves complex computation and decision logic. This is because data mining algorithms can be quite sophisticated. Second, data mining software often deals with datasets that have complex structure. Thus, it can be difficult to model and characterize the input space. Third, many data mining algorithms are designed to process large amounts of data. However, it is impractical to test large amounts of data at the development stage when testing is frequently performed. In the remainder of this paper, we will refer to data mining algorithms and software that implements data mining algorithms simply as data mining algorithms, unless otherwise specified.

The goal of our experiment is to evaluate the effectiveness of CT for testing data mining algorithms. In our experiment, we apply CT to five data mining algorithms implemented in the Waikato Environment for Knowledge Analysis (WEKA) tool. These five algorithms include C4.5, K-Means, SVM, Apriori and EM, and they are identified to be the top five most influential data mining algorithms by the IEEE International Conference on Data Mining (ICDM) [44]. Each algorithm takes two types of input, including a dataset to be analyzed, and configuration options that are used to customize the behavior of the algorithm. In our experiment, the input dataset for each algorithm was selected from a collection of 51 public benchmark datasets provided by WEKA and UC Irvine [32].

To carry out our experiment, we first ran each algorithm with the default configuration on the 51 datasets to study the impact different datasets have on the test coverage, and selected one dataset that achieved the highest branch coverage. We then created an input parameter model (IPM) for the configuration options of each algorithm. An IPM consists of representative values of each configuration option as well as constraints that exist between these values. We performed both positive and negative testing of each algorithm using the selected dataset and the IPM of the configuration options. For positive testing, we created 1-way to 6-way test sets using the valid values in the IPM. For negative testing, we created a 1-way test set for the invalid values in the IPM. In the 1-way negative test set, each invalid value is covered by one test in which every other value is a valid value. Test effectiveness is measured in terms of both branch coverage and mutation coverage.

The major results of our experiment are summarized as follows:

- *Larger datasets do not necessarily achieve higher test coverage than smaller datasets.* The sizes of the datasets that are applicable to each algorithm range from as few as 14 instances to as many as 20,000 instances. However, almost all of these datasets achieved similar branch coverage. In some cases, very small datasets achieved higher coverage than very large datasets. For

example, for algorithm Apriori, the *weather.nominal* dataset has only 14 instances, but it achieved higher coverage than the *mushroom* dataset, which has 8124 instances. This suggests that the size of a dataset is not a dominating factor in deciding test coverage. Other factors, e.g., structure of a dataset, and relationship between different instances, might play a more significant role.

- *Test coverage of CT test set increases progressively slower with respect to increase of test strength.* In our experiment, test coverage increases more significantly when test strength increases from 1-way to 3-way. After 4-way testing, higher strength test sets no longer provide significant coverage improvement. This result is consistent with the results of other empirical studies that apply CT to general software applications.
- *Branch coverage correlates well with mutation coverage.* The results of our experiment suggest that in general, branch coverage correlates with mutation coverage. In particular, higher branch coverage often implies higher mutation coverage. This suggests that branch coverage could be used as a good indicator of fault detection effectiveness for data mining algorithms, since mutation coverage is expensive to measure.

To the best of our knowledge, our work is the first attempt to evaluate the effectiveness of CT to data mining algorithms. In general, little work has been reported on testing data mining algorithms. We believe that our experiment provides initial insights that can be useful for developing more effective testing techniques for data mining algorithms.

The remainder of this paper is organized as follows. Section II discusses the major design decisions made in our experiment. Section III presents the major results obtained from our experiment. Section IV briefly reviews related work, including existing work on CT and on testing data mining algorithms. Section VI provides our conclusion and outlines several directions for future work.

II. Experimental Design

In this section, we present the design of our experiments. We formulate our research questions, identify the subject algorithms and datasets, present our approach to Input Parameter Modeling (IPM) and test generation, and discuss the metrics used to measure test effectiveness.

A. Research Question

The goal of this project is to evaluate effectiveness of CT applied to data mining algorithms. We formulate the following research questions:

1) How do different datasets impact test coverage?
2) How effective is CT applied to data mining algorithms?
3) Is branch coverage a good indicator of fault detection effectiveness?

B. Subject Programs

WEKA is one of the most widely used data mining tools. WEKA is developed by University of Waikato, and implements a collection of data mining algorithms as different packages. The subject programs include five data mining algorithms implemented in the WEKA tool: (1) C4.5, which is a supervised learning algorithm that takes a collection of cases as input, and output a classifier that predicts the class to which a new case belongs using decision tree[44]; (2) K-Means, which is an unsupervised learning algorithm that performs clustering by partitioning a given dataset into *k* clusters such that the members of each cluster are similar to each other; (3) SVM, which is a supervised learning algorithm that uses the vector space to build a SVM classification model. The model predicts the class to which a new case belongs; (4) Apriori, which is an unsupervised learning algorithm that generates association rules by identifying frequent item sets; and (5) EM, which is an unsupervised learning algorithm that uses statistical models to perform clustering. These five algorithms are identified to be the top five most influential data mining algorithms [44].

Table I shows information about the WEKA packages that implement the five algorithm.

TABLE I – WEKA PACKAGE INFORMATION

Algorithm Name	Package Name in WEKA	# of Files	# of Classes	# of Branches	LOC	# of Configuration Parameters	# of Applicable Datasets
Apriori	weka.associations	5	5	580	1349	12	11
EM	weka.clusterers	6	10	736	1825	14	46
C4.5	weka.classifiers.trees.J48	17	17	696	1641	17	44
K-Means	weka.clusterers	5	7	699	1721	18	46
SVM	libsvm	6	18	1124	2138	17	44

C. Datasets

We selected our datasets from a collection of 51 public benchmarking datasets provided by WEKA and UC Irvine [32]. Table II shows the statistics of the 51 subject datasets.

Different algorithms require different types or formats of data. As a result, not every dataset is applicable to every algorithm. To determine the applicability of a dataset to a given algorithm, we run the dataset with the algorithm. A dataset is considered applicable to an algorithm if executing the dataset with the algorithm provides meaningful output without any exception. The number of applicable datasets for each algorithm is shown in the last column of Table I.

TABLE II – DATASET INFORMATION

	# of Attributes	# of Instances	Size in KB
Maximum	217	20000	1978.39
Minimum	2	14	0.483398
Average	23.92157	1466.902	229.5591
Standard Deviation	31.73506	3079.593	393.0123
Median	18	604	44.82715

D. Input Parameter Modeling

Before CT is applied, we must create the input parameter model (IPM) [21]. Each subject algorithm takes two types of input, including a dataset to be analyzed, and a set of configuration options that are used to customize the behavior of the algorithm. Our experiment focuses on CT of configuration options. As mentioned in Section V, CT of datasets is left for future work. Thus our modeling process mainly consists of identifying representative values for different configuration options.

In the following, we use the *Apriori* algorithm as an example to explain our approach. We categorize configuration options into two groups.

- *Group 1*: This group includes options with a set of predefined choices. For each option in this group, every predefined choice is identified as a representative value for this option.
  Figure 1 shows some configuration options of the Apriori algorithm. Consider as an example option "-T", which is used to specify the metric type. This option has four predefined choices, *Confidence, Lift, Conviction, and Leverage,* each of which is identified to be a representative value for this option.

```
-N <required number of rules output>
  The required number of rules. (default = 10)

-T <0=confidence | 1=lift | 2=leverage | 3=Conviction>
  The metric type by which to rank rules. (default =
confidence)

-C <minimum metric score of a rule>
  The minimum confidence of a rule. (default = 0.9)

-c <the class index>
  The class index. (default = last)
```

Figure 2. Example Configuration Options for Apriori

- *Group* 2: This group includes options that do not have a set of predefined choices. Instead, the user can input any value that is valid. For each option in this group, equivalence partitioning is used to identify representative values.
  We observe that in our subject programs, all the options in this group are of type Integer, Float, Double. We first identify boundary values that distinguish valid and invalid values, as shown in Figure 2. A boundary value itself may or may not be valid. In Figure 2, a square bracket indicates a valid boundary value, and a parenthesis indicates an invalid boundary value. Next, we partition valid and invalid values into different groups as needed, based on domain knowledge. The set of representative values include one representative value from every partition of valid and invalid values.

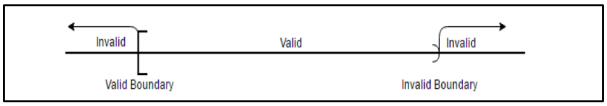

Figure 1. Equivalence Partitioning for Group 2 Configuration Options

Consider as an example the "- C" (minimum metric score) option in Fig. 1. This option allows the user to specify a threshold value for the selected metric type. Assume that the user chooses Confidence as the metric type, i.e., "-T 0 (Figure 1)". Based on domain knowledge, we identify the boundary values for minimum metric score as "0" and "1".

Next we identify the equivalence classes for valid and invalid values and select a representative value from each class. For valid values, we identify three equivalence classes: (1) {values that take every possible rule}, (2) {values that only take rules with 100% confidence}, and (3) {other values, i.e, values that do not take every possible rule, and do not require 100% confidence for each rule}. The first class consists of a single value, i.e., 0. Similarly, the second class consists of a single value, i.e., 1. The third class includes every value that is greater than 0 and less than 1. Thus, we select the following three representative values, including 0, 1, and 0.9. Note that "0.9" is the default value of this option as shown in Figure 1. In general, the default value is selected as the representative value for the equivalence class that contains the default value. Doing so helps to reduce number of representative values identified for each parameter.

For invalid values that are outside of the boundary values, we identify two equivalence classes: (1) {*value* | *value* < *lower boundary*}; and (2) {value | *value* > *higher boundary*}. A random value can be chosen from each of the two equivalence classes as the representative values.

In addition to identifying representative values for each configuration option, we have also identified constraints between different values. Constraints are used to prevent ACTS [46] from generating invalid combinations. For example, in the Apriori algorithm, when option "-A" is true, the only allowed metric type "-T" is *confidence*, i.e., option "-T" must take the value of 0. Table III shows the constraints that are identified for algorithm Apriori.

TABLE III – CONSTRAINTS IDENTIFIED FOR APRIORI

A = false => c = -1
A = true => T = 0
T = 0 => (T = 0.1 || C = 0.9)
T = 1 && A = false => (M = 0.1 || M = 0.95)
T = 1 || T = 3 => (T =1.1 || T = 1.5)
T = 2 => (C = 0.1 || C = 0.5)

E. Test Generation

We performed both positive and negative testing in our experiments. For positive testing, we created test sets that achieve 1-way to 6-way coverage for valid values using the *extend* mode from ACTS. The *extend* mode allows a test set to be built by extending an existing test set. By using the *extend* mode, every higher strength test set will be the superset of its lower strength test set(s). For negative testing, we generated a test set that achieves 1-way coverage for invalid values. That

is, each invalid value is covered by one and only one test in which every other value is a valid value. Note that in order to avoid potential mask effects, a negative test should contain at most one invalid value.

TABLE IV – SIZES OF TEST SETS

	Apriori	EM	J48	SimpleKMeans	LibSVM
1-way	7	3	4	4	5
2-way	33	11	14	16	21
3-way	132	37	48	49	76
4-way	478	91	133	136	232
5-way	1440	214	349	368	637
6-way	4055	463	835	911	1546
Negative	12	18	9	11	15

We used ACTS to generate both positive and negative test sets. Table IV shows the sizes of test sets of different strengths. Since the representative values are abstract values, the tests generated by ACTS are abstract tests. These abstract tests need to be translated to concrete tests prior to execution. For example, consider the option, "-c", representing Class Index, as shown in Figure 1. String "last" is an abstract value of this option that represents the last column (or attribute) of the input dataset. This abstract value must be mapped to the actual index of the last column in a dataset.

For each algorithm, we have written a script that performs automatic translation from abstract tests to concrete tests. The corresponding concrete value of an abstract value is calculated based on the selected input dataset. For example, abstract value "last" for option, -c, the concrete value when executing *weather.nominal* dataset for Apriori will be set to the actual last index, "5".

F. Metrics

In our experiments, we used branch coverage and mutation coverage to measure test effectiveness. We used JaCoCo to record branch coverage. JaCoCo is a free Eclipse plugin that measures statement and branch coverage at the byte code level [22].

We used an open-source mutation testing tool called PIT to measure mutation coverage [14]. We selected all available mutators that are provided by PIT for generating mutants. PIT uses JUnit tests to determine whether a mutant is killed. All JUnit tests must pass before PIT can be applied. We first ran each test case with the original programs and stored the output as the expected output. Then we created JUnit tests that check the actual output against the expected output. Whenever a passing JUnit test fails after executing a mutant, the mutant is considered killed.

PIT uses timeout to kill mutants that may never terminate. That is, if the execution of a mutant times out, then the mutant is considered killed. In order to reduce test execution time while preventing premature termination, we set the timeout value differently for each test set as follows. We first recorded the normal execution time taken by every test in a test set on the original program. This allowed us to find the longest execution time of a test set. If the longest execution time t is less than or equal to 10 seconds, we set the timeout value of the test set to be t plus 10 seconds. Otherwise, we set the timeout value to be t plus 100 seconds.

III. EXPERIMENTAL RESULTS

In this section, we present the results from our experiments. The coverage results for each algorithm are collected for the class files in the package that implement the algorithm, i.e., instead of every class file in the WEKA package. All the results and related files such as datafiles, scripts and experiment logs are publicly available at http://barbie.uta.edu/~hdfeng/.

A. Impact of Datasets

We executed each algorithm's default configuration with the 51 datasets. Some datasets are not applicable to a given algorithm, e.g., due to incorrect data type, insufficient number of attributes, and missing data of attributes. Table I (Section II) shows the number of datasets that are applicable to each algorithm.

TABLE V – BRANCH COVERAGE STATISTICS OF APPLICABLE DATASETS

	Apriori	EM	J48	SimpleKMeans	LibSVM
Maximum	28.79%	37.64%	36.64%	21.89%	34.96%
Minimum	26.72%	31.11%	8.33%	18.31%	20.46%
Mean	27.98%	35.34%	30.07%	21.04%	30.35%
Standard Deviation	0.68%	2.29%	4.8%	1.04%	3.77%
Median	28.02%	36.41%	29.89%	21.6%	31.23%

Table V presents some statistics about the branch coverage results of each selected algorithm with the applicable data sets. The results indicate that different datasets achieve similar coverage results despite significant differences in their sizes in terms of number of attributes and instances. Recall that as shown in Table II (Section II), some datasets contain as many as 20,000 instances while other datasets contain as few as 14 instances. However, the standard deviations of the branch coverage results are generally less than 5% as shown in Table V.

TABLE VI- APPLICABLE DATASETS FOR APRIORI

Dataset[a]	# of Attributes	# of Instances	Branch Coverage
vote	17	435	28.79%
weather.nominal	5	14	28.79%
splice	62	3190	28.62%
contact-lenses	5	24	28.28%
breast-cancer	10	286	28.28%
primary-tumor	18	339	27.76%
soybean	36	683	27.59%
supermarket	217	4627	27.59%
kr-vs-kp	37	3196	27.41%
mushroom	23	8124	26.72%

[a] Only 10 datasets' branch coverage are available instead of 11 as shown in TABLE I. Dataset *audiology* did not finish execution within 48 hours.

We point out that even some datasets have a very small number of instances, they can achieve higher branch coverage than the datasets with significantly more instances. Table VI shows the dataset and branch coverage information of the applicable datasets for the Apriori implementation. Consider dataset *weather.nominal* and *mushroom*. Dataset

weather.nominal has only 5 attributes and 14 instances. Dataset *mushroom* has 23 attributes and 8124 instances. However, *weather.nominal* achieved higher coverage than *mushroom*. Similar situations exist for other algorithms, which are not shown due to space limitation.

Based on the results of the 51 datasets for each algorithm, we selected one dataset that achieved the highest branch coverage for each algorithm for the rest of our experiment. If more than one dataset achieves the highest branch coverage, we break the tie by choosing the one with a smaller number of instances. For example, for *Apriori*, both *vote* and *weather.nominal* achieves the maximum branch coverage. To break the tie, we choose *weather.nominal*. The datasets selected for each algorithm are shown below;

- Apriori – *weather.nominal*
- EM – *segment-challenge*
- J48 – *credit-a*
- SimpleKMeans – *iris.2D*
- LibSVM – *primary-tumor*

> Finding 1: Larger datasets do not necessarily achieve higher branch coverage. In some cases, smaller datasets can achieve higher branch coverage than larger datasets.
>
> Implication 1: The size of a dataset is not a dominating factor for determining test effectiveness of a dataset. Instead, other characteristics must be considered, e.g., the dataset structure, and the relationship between different data instances. Also, it is possible to create small datasets that are effective for testing data mining algorithms.

B. *Branch Coverage Results of T-Way Testing*

Table VII shows the branch coverage results of the seven test sets, including the negative test set, 1-way to 6-way positive test sets. Table VII also shows the branch coverage results for the default configuration as a baseline, and the branch coverage results that combine 6-way test and negative test.

TABLE VII– BRANCH COVERAGE RESULTS OF T-WAY TESTING

Test set	Apriori	EM	J48	SimpleKMeans	LibSVM
Default Configuration	28.79%	37.64%	36.64%	21.89%	34.96%
Negative	66.03%	50.54%	55.32%	66.24%	28.47%
1-way	55.52%	52.99%	52.73%	59.51%	24.47%
2-way	66.55%	53.80%	54.60%	69.53%	43.77%
3-way	68.62%	53.94%	59.77%	70.39%	54.63%
4-way	68.62%	53.94%	59.77%	70.39%	54.80%
5-way	68.62%	54.08%	59.77%	70.39%	54.80%
6-way	68.62%	54.08%	59.77%	70.39%	54.89%
6-way &Negative	68.97%	55.3%	59.77%	70.67%	55.34%

We observe that negative test sets achieve relatively high coverage, in comparison with positive t-way tests, for all the algorithms except LibSVM. One possible reason is that the validity of a configuration option value is not checked until it is used. Thus, in some cases, a significant amount of the source code could have been executed before the system detects this invalid value. We plan to investigate this further in our future work.

We also observe that the total coverage achieved by combining the negative test set and the 6-way test set ranges from 55.34% to 70.67%. Other empirical studies [6, 7, 27, 29, 30, 43] have reported that 6-way test sets could detect all the faults. We plan to investigate this further in our future work. The following factors could have contributed to the fact that our coverage results are less than expected:

- *Limited domain knowledge*: In our experiments, we performed input parameter modeling based on our limited domain knowledge. The input parameter models could be refined to achieve higher branch coverage.

- *Testing only configuration options*: In our experiments, CT is only applied to configuration options. That is, we did not test combinations between configuration options and different datasets.

- *Shared class files that contain unreachable code from implementations of other algorithms.* In WEKA, multiple algorithms are implemented within the same package. For example, the *weka.clusterers* package contains implementations of eight different clustering algorithms, e.g., SimpleKMeans, EM, Canopy, etc. Some portions of source code may only be reachable when executing its corresponding algorithm. As an example, SimpleKmeans algorithm is a variant of EM algorithm with the assumptions that clusters are spherical. In WEKA, EM algorithm uses the SimpleKMeans class to complete its first few steps of the clustering tasks. But most source code of SimpleKMeans are not semantically reachable because EM is only using a static configuration of SimpleKMeans algorithm as specific in the EM class file.

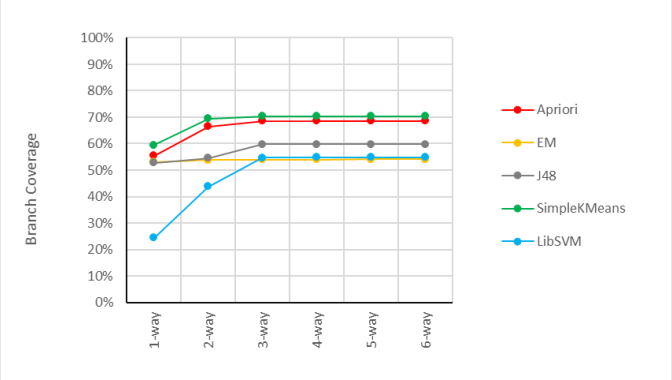

Figure 3 – Growth of Branch Coverage

Figure 3 shows how branch coverage increases with respect to test strength. The result is consistent with previous studies [15, 28]. That is, the coverage grows progressively

slower when test strength increases. Also, branch coverage stops increasing after 3-way testing for algorithms Apriori, J48, SimpleKMeans, and after 5-way testing for algorithms EM. For algorithm LibSVM, branch coverage continues to increase until 6-way testing as shown in Table VII.

> Finding 2: Branch coverage increases progressively slower as test strength increases. The coverage increase stops at a test strength that is relatively low.
>
> Implication 2: During CT, data mining algorithms display similar behavior as general software applications. CT has the potential to be effective for testing data mining algorithms.

C. *Mutation Coverage Results of T-Way Testing*

The mutation coverage results are unavailable for the following algorithms:

- KMeans – PIT cannot execute for this algorithm, due to a bug that is confirmed by PIT developers. Discussion of this bug is publicly available at https://github.com/hcoles/pitest/issues/300.
- EM – Mutation testing for EM was not able to complete within 48 hours due to the large number of mutators generated from the source code and the heavy computation of EM algorithm itself.

TABLE VIII – MUTANTS GENERATED FOR EACH ALGORITHM

Mutants	Apriori		J48		LibSVM	
ConditionalsBoundaryMutator	120	4%	113	3.11%	314	6.38%
ConstructorCallMutator	186	6.2%	137	3.77%	140	2.85%
experimental	104	3.47%	202	5.56%	204	4.15%
IncrementsMutator	113	3.77%	91	2.5%	214	4.35%
InlineConstantMutator	555	18.5%	488	13.43%	876	17.81%
InvertNegsMutator	0	0%	1	0.03%	46	0.94%
MathMutator	99	3.3%	213	5.86%	511	10.39%
NegateConditionalsMutator	282	9.4%	348	9.57%	551	11.2%
NonVoidMethodCallMutator	778	25.93%	984	27.07%	614	12.48%
RemoveConditionalMutator	564	18.8%	696	19.15%	1102	22.41%
ReturnValsMutator	112	3.73%	239	6.57%	131	2.66%
VoidMethodCallMutator	87	2.9%	123	3.38%	215	4.37%
Total	3000	100%	3635	100%	4918	100%

Table VIII shows the number of mutants generated by each mutator. Some mutators are generating significantly more mutants than others as shown in Table VIII:

- *NonVoidMethodCallMutator*: Incorrect method calls.
- *InLineConstantMutator*: Assigning an incorrect constant value to a variable.
- *RemoveConditionalMutator*: Incorrect conditional statements. "RemoveConditionalMutator" will change the conditions to a constant boolean value.

For *NegateConditionalsMutator*, "RemoveConditional-Mutator" and "ConditionalBoundaryMutator", these three mutators focus on generating mutants at conditional statements of the program, and these three mutators together generates over 30% of the total mutants for the three algorithms that are shown in Table VIII.

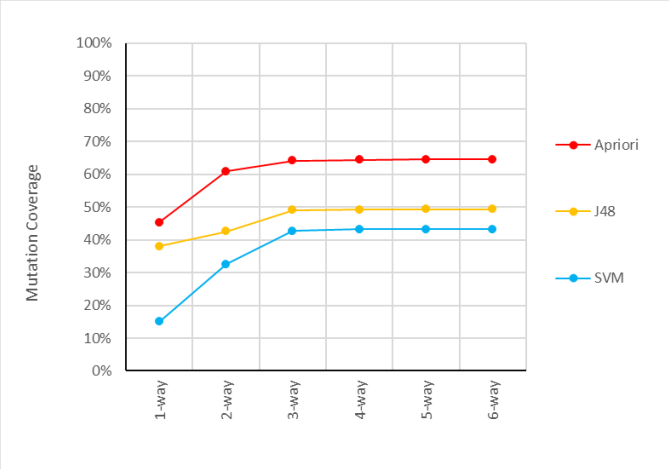

Figure 4 – Growth of Mutation Coverage

Table VII shows that branch coverage stops growing after 3-way testing for Apriori and J48. However, mutation coverage continues to grow for these two algorithms after 3-way testing, as shown in Table IX. This is because executing a line or branch of code does not necessarily expose faults that exist in the code or branch, especially when the computation or decision logic is more complex. However, the increase in mutation coverage is not significant after 3-way testing.

Table IX - MUTATION COVERAGE RESULTS OF T-WAY TESTING

Test set	Apriori	J48	LibSVM
Default Configuration	31.53%	29.66%	26.11%
Negative	59.9%	40.39%	19.46%
1-way	45.27%	38.05%	14.99%
2-way	60.93%	42.56%	32.61%
3-way	64.1%	49.05%	42.72%
4-way	64.47%	49.19%	43.19%
5-way	64.5%	49.35%	43.19%
6-way	64.53%	49.38%	43.21%
6-way&Negative	64.63%	49.38%	44.02%

Figure 4 plots the growth of mutation coverage with respect to test strength. The result is consistent with previous studies [15, 28] on branch coverage. Hence, Finding 2 and Implication 2 also apply to the coverage growth of mutation testing. The results of our experiment suggest that in general, branch coverage correlates well with mutation coverage. Thus, branch coverage could be used as a good indicator of fault detection effectiveness for data mining algorithms, since mutation coverage is expensive to measure.

When we analyzed the results of mutation coverage for the individual files of each algorithm, we discovered two special cases where mutation coverage for "apriori.java" for the Apriori algorithm decreased by one mutant from 4-way to 5-way and 6-way testing. As discussed in the Section II, the CT tests we created using ACTS used the *extend* mode. This means that every higher strength test set is a superset of its predecessor. Consequently, branch coverage and mutation coverage should not decrease from a lower strength test set to a higher strength test set. We have informed the lead

developer of PIT with this issue, but the exact cause has not been successfully identified.

> Finding 3: Branch coverage and mutation coverage seem to correlate well for data mining algorithms. That is, higher branch coverage seems to imply higher mutation coverage, and vice versa.
>
> Implication 3: Branch coverage could be used as a good indicator of fault detection effectiveness for data mining algorithms, since mutation coverage is expensive to

D. Threats to Validity

Threats to external validity occur when the experimental results could not be generalized to other subjects. Our subject programs implement the top five data mining algorithms identified in [44] and are from a widely used data mining tool, i.e., WEKA. The datasets used in our experiments have been used in other studies [32]. More experiments using data mining algorithms other than these five algorithms and using different datasets can further reduce threats to external validity.

Threats to internal validity are other factors that may be responsible for the experimental results. To prevent mistakes that could happen during the modeling process, two of the authors created the IPM independently and cross-checked them against each other. We have automated the execution of experiments using scripts, as an effort to minimize human errors. Furthermore, consistency of the results (executed by scripts) has been checked by two of the authors using their independently written scripts.

IV. Related Work

We first review previous work on applying CT to different types of software. Lei et al. [30] developed a t-way testing strategy for testing concurrent programs. Simos et al. [40] and Bozic et al. applied CT to perform security testing of web applications [6]. Li et al. applied CT to test three real-life industrial software systems that include an embedded system, a graphical operating system and a database management system [35]. Dhadyalla et al. applied CT to test automotive control software embedded in a hybrid electric vehicle [16]. Li et al. applied CT to ETL applications [34]. Note that ETL is a special type of big data applications. However, the work in [34] focuses on data transformation and management aspects, whereas our work focuses on algorithmic aspects. These existing works show that CT can be effectively applied to different domains. However, to our knowledge, our work is the first one that applies CT to data mining algorithms.

Second, we review existing work related to evaluating the effectiveness of CT. Khun et al. [31] investigated the fault detection effectiveness of t-way testing. Kuhn et al. [27] report a study that applies CT and random testing to detect deadlocks in a network simulator. Bell and Vouk discussed the effectiveness of pairwise testing and random testing to a network-centric software [2]. A number of studies have been reported that compares the effectiveness of CT and random testing [1, 5, 7, 17, 25, 41, 42]. There are also studies that investigate the code coverage effectiveness of t-way testing [13, 15]. Our work presented in this paper is the first effort to evaluate the effectiveness of CT to data mining algorithms.

Third, we review previous work related to testing data mining applications. Jeske et al. [24] developed a platform to generate realistic, synthetic data to test data mining tools. Data mining tools were evaluated in terms of their false positive and false negative error rates when executed with the synthetic data. Murphy et al. discussed how to identify metamorphic properties for performing metamorphic testing of data mining algorithms [38]. Metamorphic testing is one approach to addressing the test oracle problem. Murphy et al. [37] discussed approaches to test machine-learning applications that implement ranking algorithms. Our work is different in that we apply CT to test data mining algorithms.

Finally, we note that a significant amount of work has been reported on testing database centric applications [3, 8, 9, 10, 18, 33, 45]. Similar to work presented in [34], these work focuses on testing data management aspects. In contrast, our work focuses on the algorithmic aspects of data mining software.

V. Conclusion and Future Work

In this paper, we reported an experiment that applied CT to five data mining algorithms implemented in the WEKA tool. This is part of a larger effort that is aimed to develop effective CT-based methods for testing big data applications. The experiment allows us to obtain some initial understandings about the effectiveness of CT on data mining algorithms. In particular, the results of our experiment indicate that data mining algorithms behave in a way that is similar to general software. This suggests that CT has the potential to be effectively applied to data mining algorithms.

We plan to continue our work in the following three directions. First, we will perform detailed code analysis to better understand the results of our experiment. In particular, we want to investigate why some branches were executed by none of our test sets, and whether these branches could be executed by using different configuration options and/or datasets. Second, in our experiment, we only applied CT to configuration options. We plan to investigate how to apply CT to create representative datasets. The key challenge is to identify the characteristics of a dataset that could significantly impact the execution of the underlying algorithm. We can model these characteristics as abstract parameters, and then apply CT to these parameters to create representative datasets. Third, negative testing alone has shown great importance in achieving good coverage, we will perform further investigation and experiments on how we can better use negative testing to improve the coverage of CT [20].

VI. Acknowledgment

This work is supported by grant 70NANB15H199 from Information Technology Lab of National Institute of Standards and Technology (NIST).

Disclaimer: Certain software products are identified in this document. Such identification does not imply recommendation by the NIST, nor does it imply that the products identified are necessarily the best available for the purpose.

REFERENCES

[1] W. A. Ballance , S. Vilkomir, and W. Jenkins. Effectiveness of pair-wise testing for software with boolean inputs. Proceedings of the IEEE Fifth International Conference in Software Testing, Verification and Validation (ICST), 580-586, 2012.

[2] K.Z. Bell, and M.A. Vouk. On effectiveness of pairwise methodology for testing network-centric software. Proceedings of the 3rd International Conference in Information and Communications Technology for Enabling Technologies for the New Knowledge Society, 221-235, 2005.

[3] C. Binnig, D. Kossmann, and E. Lo. Testing database applications. Proceedings of the 2006 ACM SIGMOD international conference on Management of data, 739-741, 2006.

[4] M.N. Borazjany, L.S. Ghandehari, Y. Lei, R. Kacker and R. Kuhn. An input space modeling methodology for combinatorial testing. Proceedings of the Sixth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 372-381, 2013.

[5] M.N. Borazjany, L. Yu, Y. Lei, R. Kacker and R. Kuhn. Combinatorial testing of ACTS: A case study. Proceedings of the Software Testing, Verification and Validation (ICST), 591-600, 2012.

[6] J. Bozic, B. Garn, I. Kapsalis, D. Simos, S. Winkler, and F. Wotawa. Attack pattern-based combinatorial testing with constraints for web security testing. Proceedings of the IEEE International Conference In Software Quality, Reliability and Security (QRS), 207-212, 2015.

[7] R.C. Bryce, A. Rajan, and M.P. Heimdahl. Interaction testing in model-based development: Effect on model-coverage. Proceedings of the 13th Asia Pacific Software Engineering Conference (APSEC), 259-268, 2008.

[8] D. Chays, S. Dan, P.G. Frankl, F.I. Vokolos and E.J. Weyuker. A framework for testing database applications. Proceedings of the ACM SIGSOFT Software Engineering Notes, 25(5), 147-157, 2000.

[9] D. Chays, Y. Deng, P.G. Frankl, S. Dan, F.I. Vokolos and E.J. Weyuker. An AGENDA for testing relational database applications. Proceedings of the Software Testing, verification and reliability, 17-44, 2004.

[10] M.Y. Chan, and S.C. Cheung. Testing Database Applications with SQL Semantics. In CODAS, 363-374, 1999.

[11] M. Chen, S. Mao, and Y.Liu. Big data: A survey. Mobile Networks and Applications, 171-209, 2014.

[12] C.P. Chen and C.Y. Zhang. Data-intensive applications, challenges, techniques and technologies: A survey on Big Data. Information Sciences, 275, 314-347, 2014.

[13] E.H. Choi, O. Mizuno, O and Y. Hu. Code Coverage Analysis of Combinatorial Testing. Proceedings of the 4th International Workshop on Quantitative Approaches to Software Quality, p.34.

[14] H. Coles. Pit mutation testing. http: //pitest.org/, 2016.

[15] J. Czerwonka. On use of coverage metrics in assessing effectiveness of combinatorial test designs. Proceedings of the IEEE Sixth International Conference in Software Testing, Verification and Validation Workshops (ICSTW), 257-266, 2013.

[16] G. Dhadyalla, N. Kumari and T. Snell. Combinatorial testing for an automotive hybrid electric vehicle control system: a case study. Proceedings of the IEEE Seventh International Conference in Software Testing, Verification and Validation Workshops (ICSTW), 51-57, 2014.

[17] M. Ellims, D. Ince and M. Petre. The effectiveness of t-way test data generation. Proceedings of the International Conference on Computer Safety, Reliability, and Security, 16-29, 2008.

[18] M. Emmi, R. Majumdar and K. Sen. Dynamic test input generation for database applications. Proceedings of the International Symposium on Software testing and analysis, 151-162, 2007.

[19] A. Frank, and A. Asuncion, A. UCI Machine Learning Repository [http://archive. ics. uci. edu/ml]. Irvine, CA: University of California. School of Information and Computer Science, 213, 2010

[20] A. Gargantini, J. Petke, M. Radavelli and P. Vavassori. Validation of Constraints Among Configuration Parameters Using Search-Based Combinatorial Interaction Testing. Proceedings of the International Symposium on Search Based Software Engineering, 49-63, 2016.

[21] M. Grindal and J. Offutt. Input parameter modeling for combination strategies. Proceedings of the 25th conference on IASTED International Multi-Conference Software Engineering, 255-260, 2007.

[22] M. Hoffmann, B. Janiczak, E. Mandrikov and M. Friedenhagen. Jacoco code coverage tool. Online , 2016

[23] D.R. Jeske, P.J. Lin, C. Rendon, R. Xiao and B. Samadi. Synthetic data generation capabilties for testing data mining tools. Proceedings of the IEEE Military Communications conference(MILCOM), 1-6, 2006.

[24] D.R. Jeske, B. Samadi, P.J. Lin, L. Ye, S. Cox, R. Xiao et.al., Generation of synthetic data sets for evaluating the accuracy of knowledge discovery systems. Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining, 756-762, 2005.

[25] N. Kobayashi, T. Tsuchiya and T. Kikuno. Applicability of non-specification-based approaches to logic testing for software. Proceedings of the IEEE International Conference on Dependable Systems and Networks, 337-346, 2001.

[26] R. Kuhn, R. Kacker, Y. Lei and J. Hunter. Combinatorial software testing. Computer, 42(8), 2009.

[27] D.R. Kuhn, R. Kacker and Y. Lei. Random vs. combinatorial methods for discrete event simulation of a grid computer network. Proceedings of ModSim World, 83-88, 2010.

[28] D.R. Kuhn, R. Kacker and Y. Lei. Introduction to CT. CRC press, 2013.

[29] D.R. Kuhn, R. Kacker and Y. Lei. Estimating t-Way Fault Profile Evolution During Testing. Proceedings of the IEEE 40th Annual Computer Software and Applications Conference (COMPSAC) Vol. 2, 596-597, 2016.

[30] Y. Lei, R.H. Carver, R. Kacker and D.Kung. A combinatorial testing strategy for concurrent programs. Proceedings of the Software Testing, Verification and Reliability, 17(4), 207-225, 2007.

[31] D.R. Kuhn, D.R. Wallace and A.M. Gallo. Software fault interactions and implications for software testing. Proceedings of the IEEE transactions on software engineering, 30(6), 418-421, 2004.

[32] M. Lichman. UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science, 2013.

[33] B. Li, M. Grechanik and D. Poshyvanyk. Sanitizing and minimizing databases for software application test outsourcing. Proceedings of the IEEE Seventh International Conference on Software Testing, Verification and Validation, 233-242, 2014.

[34] N. Li, Y. Lei, H.R. Khan, J. Liu and Y. Guo. Applying combinatorial test data generation to big data applications. Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ACM) , 637-647, 2016.

[35] X. Li, R. Gao, W.E. Wong, C. Yang and D. Li. Applying Combinatorial Testing in Industrial Settings. Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (QRS), 53-60, 2016.

[36] T.M. Mitchell, Machine learning. Burr Ridge, IL: McGraw Hill, 45(37), 870-877, 1989.

[37] C. Murphy, G.E. Kaiser and M. Arias. An Approach to Software Testing of Machine Learning Applications. In SEKE, 167, 2007.

[38] C. Murphy, G.E. Kaiser, L. Hu and L.Wu. Properties of Machine Learning Applications for Use in Metamorphic Testing. In SEKE (Vol. 8), 867-872, 2008.

[39] G. Paynter, L. Trigg, E. Frank and R. Kirkby. Attribute-relation file format (ARFF). Online] http://www. cs. waikato. ac. nz/ml/weka/arff. Html, 2008.

[40] D.E. Simos, K. Kleine, L.S.G. Ghandehari, B. Garn, and Y. Lei. A Combinatorial Approach to Analyzing Cross-Site Scripting (XSS)

Vulnerabilities in Web Application Security Testing. Proceedings of the International Conference on Testing Software and Systems, 70-85, 2016.

[41] P.J. Schroeder, P. Bolaki, and V. Gopu. Comparing the fault detection effectiveness of n-way and random test suites. In Proceedings of International Symposium on Empirical Software Engineering, 49-59, 2004.

[42] S. Vilkomir, O. Starov and R. Bhambroo. Evaluation of t-wise approach for testing logical expressions in software. Proceedings of the IEEE Sixth International Conference in Software Testing, Verification and Validation Workshops (ICSTW), 249-256, 2013.

[43] H. Wang, C. Xu, J. Sui and J. Lu. How Effective Is Branch-Based Combinatorial Testing? An Exploratory Study. Proceedings of the IEEE International Conference in Software Quality, Reliability and Security (QRS), 41-52, 2016.

[44] X. Wu, V. Kumar, J.R. Quinlan, J. Ghosh, Q. Yang, H. Motoda et al. Top 10 algorithms in data mining. Knowledge and information systems, 14(1), 1-37, 2008.

[45] X. Wu, Y. Wang and Y. Zheng. Privacy preserving database application testing. Proceedings of the ACM workshop on Privacy in the electronic society, 118-128, 2003.

[46] L. Yu, Y. Lei, R.N. Kacker and D.R. Kuhn. Acts: A combinatorial test generation tool. Proceedings of the IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST), 370-375, 2013.