Clump: CLUstering on Many Predicates

Bryan Lemon
Lane Department of Computer Science and Electrical Engineering
West Virginia University
PO Box 6109
Morgantown, WV, 26506-6109
bryan@bryanlemon.com

Tim Menzies
Lane Department of Computer Science and Electrical Engineering
West Virginia University
PO Box 6109
Morgantown, WV, 26506-6109
tim@menzies.us

ABSTRACT

Current clustering algorithms are slow, with high order polynomial run-times. Current algorithms also cluster data by Euclidean distance. We propose an algorithm called Clump. Clump is a Naive Bayes classifier augmented by a rule tree based clusterer. Clump runs in low order polynomial time, and clusters by attribute similarity rather than full Euclidean distance. Clump clusters the data, and during testing, forms a naive bayes classifier with the data at each node.

Clump and other learners are tested against 7 datasets. These datasets [21] represent projects ranging from video guidance, a NASA dataset, to small appliance controllers, a SoftLab dataset.

Information about the results and conclusion will go here later.

1. INTRODUCTION

This paper demonstrates how an incremental clusterer can aid in classification. Clustering algorithms aid in classification by grouping like data together. This grouping helps to reduce the noise in the data, thereby reducing the false alarm rate [23]. Current clustering algorithms have a high order polynomial run-time, usually in the form of $O(n^2)$ or higher [1, 19]. Other clustering algorithms have faster run-times, but have execution time parameters that need to be tuned to each specific dataset [12]. We propose a self tuning clustering algorithm that runs in low order polynomial time.

The proposed solution is called Clump, standing for CLUstering on Many Predicates. This solution provides a method for the domain expert to create, audit, and modify the decision tree. By allowing the end user to oversee the creation of the decision tree, Clump provides a mechanism to repair broken rules, and provide domain specific insight. The rule tree also expands as needed, in response to the current intra-node entropy. This automated growth removes the necessity of dataset specific parameters from configuring the learner and training on the data.

With most clustering algorithms [1, 18, 19], the Euclidean distance between two instances is used as the nearness function. When a group of instances have a small Euclidean distance when compared to other instances in that group, they form a cluster. Depending on the clustering algorithm, there can be sub-clusters [19].

With Clump's human maintainability, it gains the benefit of expert systems with the generation speed of automated clustering. The following sections will describe Clump as it pertains to automated generation, and compare it to other rule based and statistical learners. Options for human maintainability are described in §7.

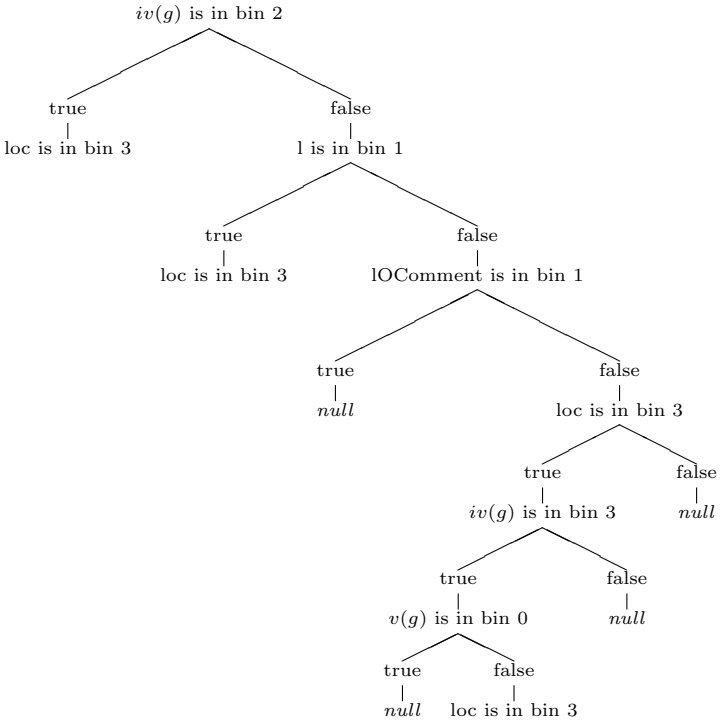

Figure 1: A sample rule tree for the KC1 dataset

2. PROPOSED SOLUTION

A rule tree classifier called Ripple Down Rules(RDR) was proposed by Paul Compton in his 1991 paper titled "Ripple Down Rules: Possibilities and Limitations" [5]. A basic Ripple Down Rule tree is defined as a binary tree where each node contains:

- A classification
- A predicate function
- A true branch
- A false branch

The true and false branches are other nodes that may or may not exist. The true and false branches are followed depending on the outcome of the predicate function during testing. The true and false nodes are created on demand, and are patches to the tree. Ripple Down Rule trees are human maintainable and explainable.

With Clump, if the amount of entropy[1] in a node exceeds a threshold value, a patch is created to reduce the intra-node entropy. Like Ripple Down Rule trees, the clusters made by Clump are also human maintainable. Figure 2 shows a sample rule tree generated by Clump.

Clump differs from standard clusterers. Most clustering algorithms group rows based on the Euclidean distance between two rows or cluster centroids [1, 12, 18, 19]. This Euclidean distance (Equation 1) is determined by the combined absolute value of the delta between the two rows or the row and the centroid.

$$\sum_{x=0}^{k} \Delta(|row_1[attribute_x] - row_2[attribute_x]|) \qquad (1)$$

Clump accomplishes its clustering, not by Euclidean distance, but by how relevant a particular feature is in splitting the data. The exact equations used in determining the relevancy of a give attribute are shown in §4.3 Equation 2 - Equation 5. Grouping data by relevancy accomplishes much the same as grouping data by Euclidean distance. It provides several benefits as well. There is an explanation as to why a particular instance belongs to a particular cluster. Also, the standard n^2 runtime of clustering algorithms is avoided because decisions made in generating the tree are made in respect to each row as its being examined, not to every row as each row is being examined.

One goal of Clump is to create rule trees that are maintainable. The maintenance can be accomplished in two ways. First, if a rule tree is small enough, a human to look at the tree, and remember most if not all of it by recall. This allows the human to notice additions/subtractions that could be made to the tree by utilizing domain specific knowledge. Second, a tree can be built during initial training, and while being used with real-time data, can be patched to adapt to the changing data. A second goal of Clump is to form the nearest neighbor structure from the data in linear or low order polynomial time.

Rule trees offer a significant advantage to frequency count learners such as Naive Bayes:

> Rule trees offer not just answers, they offer explanation.

Unlike Ripple Down Rules, Clump is not used for classification, but for clustering. Clump is used to find the structure within the data while avoiding the standard $O(n^2)$ clustering algorithms. While testing, Naive Bayes is used to do the classifying once the data has been limited by Clump as a clusterer.

9[1]Entropy is used to represent a mix of different classes in a general sense, not in the standard entropy calculation

This paper will explore the Clump/Naive Bayes combination while using 7 datasets from the Promise Data Repository [10]. The chosen datasets focus on software defect prediction.

3. RELATED WORK

The performance of Clump will be compared to several different learners in §5 and §6. Among these learners will be:

- RIpple DOwn Rules, also known as Ridor [9]
- j48[2]
- jRip[3]
- OneR [13]
- Naive Bayes [17]
- Locally Weighted Learning [7]

Four of the learners listed above are all rule based learners. Ridor is a version of a Ripple Down Rule tree. j48/C4.5 is an extended version of Quinlan's ID3 tree. jRip/RIPPER is an optimized version of Cohen's IREP. OneR is a 1R algorithm; as soon as one rule is matched, the processing stops. Locally Weighted Learning is Naive Bayes augmented by weighting training instances by their nearness to the test instance.

3.1 Classifier Algorithms

3.1.1 j48/C4.5

j48/C4.5 is based in the ID3 tree learner. It creates chooses its splits in data based off of normalized information gain of the attributes. Once an attribute is chosen, new child nodes are created for each of the values present in the chosen attribute. The algorithm then recurses on each of the children nodes. This differs from Clump because each node can split into at most 2 additional nodes: one for the true branch, and one for the false branch. Clump chooses a predicate function for the node, while j48 chooses an attribute to split on.

3.1.2 jRip/RIPPER

jRip, also called RIPPER, is a rule based algorithm rather than a rule tree algorithm. RIPPER stands for Repeated Incremental Pruning to Produce Error Reduction. RIPPER creates as series of individual rules, adding conjunctions until the rule only satisfies members of one class. The rules are then pruned to remove the rules that decrease the performance of the algorithm. RIPPER is closer to OneR than Clump because RIPPER creates a series of conjunctions. If the test data matches the first rule, the class of the first rule is chosen. The test data is passed down the rule list until it matches on rule or the final catch-all rule is chosen. RIPPER explores all possible rules, while Clump explores only the necessary rules to create the tree.

9[2]This is a Java representation of J. Quinlan's c4.5 learner. [20]

9[3]This is a Java representation of W. Cohen's RIPPER learner. [4]

	Clump	Ridor	Ripple Down Rules
How Rules are Chosen	The attribute value pair that decreases the mixuped-ness of the resulting dataset as described in §4.3.1 is used to patch the tree.	The attribute value pair that has the maximum info gain is used to patch the tree.	A human creates the patch by examining the training instance and creating a patch manually.
When to Make a Rule	When the mixuped-ness as defined in §4.3.1 passes a threashold.	When eneough training instances have been misclassified. This is a runtime configuration option.	When a training instance is misclassified.
Incrimental or Batch	Incrimental or batch	Batch only	Incrimental only

Figure 2: Comparing the difference between classifiers: Clump, Ridor, and Traditional Ripple Down Rules

3.1.3 OneR

OneR, also referred to as 1R, creates a series of rules. The rules are compared to the testing data one by one. As soon as a rule matches, processing terminates for the matching rows, and the matching rule's classification is assigned to the rows. Over-fitting avoidance is accomplished by requiring each rule to match a pre-determined number of rows. The attributes are ranked according to their error in classifying the training set, rather than some entropy or information gain metric. This learner is included as the "straw man" of the rule based classifiers.

3.1.4 Naive Bayes

Naive Bayes makes many assumptions about data. All attributes are:

- assumed to be equally important.
- statistically independent.
- do not predict values of other attributes

These assumptions rarely hold up to real world data, but empirically, they work quite well [6]. Naive Bayes is frequently augmented by different pre and post processing algorithms to attempt to reduce the its naivety while still maintaining its performance and runtime.

Naive Bayes runs in O(n) time, as no structure is built, no rules are learned, and no complex processing is done. Only frequency statistics are gathered during training, followed by simple arithmetic during testing. Missing values are handled by ignoring that attribute during calculations. The major drawback to Naive Bayes is that, although Naive Bayes offers conclusions, it offers little insight into how these conclusions were reached. By adding a decision tree to Naive Bayes, we hope to add the "why" to the "what" of Naive Bayes.

3.1.5 Locally Weighted Learning

Locally Weighted Learning, or LWL, is lazy naive bayes classifier. It is called lazy because, upon training, the data is just stored, leaving the computational work to occur during testing. Testing is accomplished one row at a time. As each row is tested, the records in the training set are weighted based on their Euclidean distance from the testing row. After the training instances are weighted, a standard naive bayes algorithm is applied.

This algorithm assumes that the data near the testing row holds the most relevance to it. It is possible that two rows can be near each other while never sharing a common attribute. It is also possible that two rows could be identical in several attributes, while having many that are substantially different. Clump attempts to balance these two approaches by grouping data by their similarities and minimizing entropy.

3.1.6 RIpple DOwn Rules/Ridor

Ridor, or RIpple DOwn Rules, by Compton is the first implementation of a Ripple Down Rule tree. The main difference between Ridor and Clump is that Ridor is a classifier and Clump is an incremental clusterer. Overlooking the difference in usage, we can compare the generation of the two learner's trees. Ridor generates its trees using information gain to split continuous attributes into two bins, and then uses a $\leq$ or $>$ operator to represent the split. Clump on the other hand splits its data into equal width bins before using its own scoring function(described in §4.3 to determine the chosen attribute and range to split on.

A traditional Ripple Down Rules tree as described in [5] has a rule with an "except" and "or" branch that represents the true patch and false patch respectively. A patch is created as new data enters the tree that violates a previous rule. By patching the tree, it can remain relevant even as new, previously unseen, data is discovered.

3.2 Clustering Algorithms

Clustering algorithms attempt to reduce the time needed to search for nearest neighbors. They accomplish this by forming a tree of the nodes in the training set. The children of each node contain the other nodes that share some "nearness" to the parent. The speedup in searching time needs to be balanced with the time necessary to create and maintain this tree.

3.2.1 Cover Trees

The Cover Tree algorithm was created by Beygelzimer et al. [1] in 2006. It forms a tree with the top level of the tree being of level i, where $i \geq$the number of levels in the tree. Starting at the root node, as the tree is descended, i decrements. At each node of the tree, the distance between any two points of the node's children is greater than 2^{i-1}. At least one point p in the node is within 2^i of any point q in the node's children.

Cover trees have a maximum insertion time of $O(c^6 nln(n))$ [1], where c is the dimensionality of the dataset. This insertion time is theoretical and represents the worst case scenario. While the creation time is high, the querying time is only $O(c^{12} lnn)$.

3.2.2 KD-Trees

A KD-Tree is a binary tree where the data is split at each node based on some dimension d, and a point along that axis [18]. A training row r is chosen to be the splitting row for a node. Any training row who's dth dimension is less than the dth dimension of r belongs to the left subtree, and the rest belong to the right subtree.

Querying the KD-Tree for the nearest neighbor takes at least $O(ln(N))$ time, and can take up to $O(N)$ time for some distributions. The more evenly the data is spread across the k-dimensional space, the closer the runtime will be to $O(ln(N))$. If the training data is clustered in a small subset of the space, and the testing data is not clustered in that subset, then the majority of the space will have to be searched. This will push the runtime closer to the worst case scenario.

3.2.3 Ball Trees

Ball trees are like Cover trees, where the parent nodes contain all of that node's children nodes. One benefit of ball trees over KD-trees is that ball trees do not need to partition the whole space [19]. Also, the children's balls are allowed to intersect with each other. A parent's ball is large enough to encompass all of its children and their balls. There are two main ways to construct the ball trees:

- Bottom Up: The tree is constructed from the leaves to the root node. This provides the optimum trees, but takes the longest to construct.
- Top Down: The tree is constructed from the root to the leaves. This provides the fastest construction time, but performs worse than trees generated with the Bottom Up method.

3.3 Burak Filter

The Burak filter was designed to aid in Cross Company defect prediction. As noted by Turhan et al., when Cross Company data was used in defect prediction, the recall and false alarm rates both increased drastically [**?**]. They determined that this was caused by the increase defect examples, both pertinent and extraneous. To remove the extraneous examples, the training data is filtered with respect to the testing data.

This filter used the Euclidean distance between the testing and training examples to find the k nearest neighbors per test instance. The nearest neighbors for each test instance are combined to form the training set. Any training instances that are within the k nearest neighbors of more than one test instance are only included once in the final training set. By training on only the nearest neighbors to the test instances, theoretically only the relevant training instances are examined.

The cost associated with generating this nearest neighbor information is exponential, in the order of $O(N_{train}N_{test})$. For large datasets, this preprocessing runtime is impractical as the neighbor information must be recalculated for each testing example. Clump is is proposed as an alternative to to Burak filter.

3.3.1 Clump versus Burak filter

Clump and the Burak filter [24] both are used as a clustering algorithm designed to aid a classifier. The Burak filter finds the k nearest neighbors, based on Euclidean distance, to each testing instance. This information is then passed to a classifier for final classification. Like the Burak filter, Clump also finds the nearest neighbors. The neighbors for Clump are determined by the training rows left in the various nodes of the Clump tree. The testing instances are run down the tree, and the resultant data is gathered. This process is described in more detail in §4.4.

The main difference between Clump and the Burak filter, run-times aside, is in the amount of data returned. The Burak filter finds the k nearest neighbors. Enforcing a value of k can cause unwanted data to be gathered, or useful data to be set aside. If the testing row falls in a very specific region of the n dimensional space where few other instances fall, rows that are distant from the testing example can be included in training the final classifier. Likewise, if the testing row falls in a highly populated area, many useful rows can be left behind. Clump overcomes this by including only the rows which are similar to the testing row according to the rule tree generated during training.

4. DESIGN OF CLUMP

Clump is a rule based decision tree clusterer. At its core, it is a binary tree with nodes that can have $0-2$ children. Each node consists of a rule, its true and false conditions (if any), and a collection of training data that has reached that node. When testing, a testing example travels down the tree until it reaches a point where there are no children nodes for it to follow. The data that stored in that node is then passed to a naive bayes classifier for final classification.

Most clustering algorithms use the nearest neighbor calculation to determine which cluster a record belongs to. This record by record comparison takes $O(n^2)$ comparisons, each record must be compared against every other record to minimize the dissimilarity. Clump creates it's clusters, not by minimum dissimilarity, but by grouping records with similar attributes. Grouping by similar attributes leads to decreased run-times. Frequency counts can be gathered, and cached, which leads to decreased run-times. Frequency counts can be used to determine which attribute value pair to split on because of the types of rules created by Clump.

When training, Clump produces greedy rules, adding one attribute to the rule at each level of the tree. Clump performs local feature subset selection as it creates the rules, only considering attributes that have not been considered further up the tree. The most important feature, determined by the reduction in entropy, is chosen for the splitting criteria at each branch of the tree. When choosing the splitting criteria, the standard entropy calculation is not used. The entropy is determined by the frequency of the different classes represented in the training rows at each branch of the tree, relative to the overall frequency of each class. This allows some features that might only be important under specific circumstances to be used when needed, and ignored in the other parts of the tree.

4.1 Runtime Complexity

The theoretical runtime of Clump while training is O($(3kn+k)*log_{kn}$) where k is the number of columns in the dataset, and n is the number of rows. When training, each record is examined to determine which bin it belongs to (O($1kn$)). To create the bins, the minimum and maximum values of each attribute must be found (O($1kn$)). Next, a Naive bayes classification table is built with a runtime cost of O($1kn$). If the

	Clump	K-Means	Locally Weighted Learning
Clusters With	Dependant attributes	Independant attributes	Independant attributes
Clusters By	Attribute similarity	Euclidean distance	Euclidean distance
Clusters on	Attribute similarity	Nearest centroid	Nearest neighboor
Training behavior	Creates a decision tree with the relevant training instances at each leaf	Creates clusters to be used by another algorithm	Delays processing until testing time.
Testing behavior	Creates a naive bayes classifier based off of the training data at the node that the test instance resides at.	*Not applicable. K-Means does no classification*	Finds the k nearest neighboors and weights them according to the normalized euclidean distance from the test instance. It then builds a naive bayes classifier off of the weighted data.

Figure 3: Comparing the differnece between clustering algorithms: Clump, K-Means, and Locally Weighted Learning

data requires, a patch is created by examining the frequency count tables for an attribute value pair that will reduce the mixuped-ness of the data. This is repeated at each level of the tree with the amount of data, in both the row count and the column count, decreasing at each level of the tree.

With the current built-in classifier of Naive Bayes, the theoretical maximum runtime for testing is O(nkd), where d is the depth of the tree. This worst case scenario occurs when all the data is contained in a single cluster. The current maximum depth is limited to 15 levels. The theoretical minimum runtime for testing is O($\frac{nk}{d}$). This best case scenario occurs when the data is evenly distributed across n-dimensional space, causing each node in the tree to contain an equal number of rows.

4.2 Discretization

The rules are based on which bin the discretized attributes belong in. Rather than rules reading: "If attribute$_x$ < value then ...", the rules for Clump read: "If attribute$_x$ is in bin x then ...". Discretizing is done using equal width discretization [25]. Each attribute range is broken up into three different bins, with each bin having the same width, or the same distance between the minimum and maximum values for the bin. Three bins were chosen by experimental testing.

4.3 Training

Training, like most tree building algorithms is a recursive process. Initially, a root node is made and populated with the entire training set. The default class is also set to the majority class. This root node is then passed to the training function. The training function looks at the data in the node, and if there are $\leq$ 15 rows in the data, training terminates. If there are $\geq$ 15 rows in the data, an optimal splitting criteria is chosen. Two resultant nodes are created and added as children of the generating node. All data from the generating node that satisfies the optimal splitting criteria is added to the true child node, and all that does not is added to the false child node. The process then recurses until all nodes are created.

The optimal splitting criteria is used to create the rule, and split the data into two groups: the data that satisfies the splitting criteria, and the data that does not satisfy the splitting criteria. Each rule created is conditional on the node's parent's rule.

4.3.1 Scoring Function

When choosing the optimal splitting criteria, all possible splitting criteria for both positive and negative classes at a node are explored. Each possible split receives a score based on the relative frequency of the positive and negative records as described in Equation 2 - Equation 5.

$$P_{true} = \frac{F_{true} \mid v}{F_{true}} \tag{2}$$

$$P_{false} = \frac{F_{false} \mid v}{F_{false}} \tag{3}$$

$$Score_{true} = \frac{P_{true}}{P_{true} + P_{false}} \tag{4}$$

$$Score_{false} = \frac{P_{false}}{P_{ture} + P_{false}} \tag{5}$$

This scoring function removes the bias to choose classes with more overall support. This is done by normalizing the frequency of the class at a node by representing it as the percentage of all examples of class C present in the node. Comparing the relative frequency of the true versus the false nodes this way can show the presence or lack thereof of a bias towards one class versus the other.

4.3.2 Dependant/Independent Attributes

An attribute is independent [14] when the attribute value is disassociated from the class attribute. A dependant [14] attribute is when the attribute value is considered in conjunction with the class attribute. Most clustering algorithms consider independent attributes, building clusters without considering the classes of the intra node instances. Clump builds its clusters by reducing intra node entropy. To accomplish this, an attribute value is chosen according the the process is §4.3.1. The attribute chosen will create two sub clusters each containing a higher frequency of one class than the combine parent cluster.

4.3.3 Boosting

Traditional boosting [22] is where the training instances that fail classification with one learner will be used to train a second learner. This process can be repeated many times,

```
function Training(data)
  if(numRows == 0 || depth >= 15)
    exit;

  for(row in data)
    for(colum in columns)
      columnWidth = max(column) − min(column) / 3
      row[column].value = row[column].value − min(column) / columnWidth


  choices = GetColumnWeights(data)

  maxChoice = max(choices);

  rule = CreateRule(maxChoice, data)

  return rule
  rule.true = Training(data | rule)
  rule.false = Training(data | !rule)


function CreateRule(choice, data)
  rule.function = choice
  rule.true = Training(data | choice)
  rule.false = Training(data | !choice)
  return rule


function GetColumnWeights(data)
  for(column in columns)
    for(bin in column)
      trueData = data | column.value = bin && data.class = true
      falseData = data | column.value = bin && data.class = false
      column.bin.weight = max(trueData.size / data | data.class = true , -
falseData.size / data | data.class = false )
```

Figure 4: Pseudo code of the Clump Training process

```
function Testing(data)
  chunks = split(data)
  for(chunk in chunks set)
    for(row in chunk)
      gatheredData += GatherData(tree, row)

  nb = NaiveBayes(gatheredData)
  return nb(data)
```

Figure 5: Pseudo code of the Clump Testing process

resulting in a chain of learners that can then be pooled for final classification. This is commonly referred to as "toilet learning" as each successive learner is trained off of the dregs of the learner before it. Clump represents boosting by creating sub clusters of data when the data added to a cluster becomes too diverse. Clump creates sub clusters that have a lesser combined mean degree of diversity within each other than within the parent cluster.

4.4 Testing

When testing, one or more testing rows are combined to form a chunk of rows. Each row from a chunk is sent down the tree, and the data stored at the node where the row stops is combined. The row of data continues its travel down the tree, following the true and false branches as necessary until a leaf node is reached. The row then travels back up the tree until the last satisfied rule is reached. While combining the data, any duplicates are ignored[4]. This combined training data is then passed to a naive bayes classifier. Each row in the testing chunk is then passed to the naive bayes classifier for final classification.

When gathering the data for the naive bayes classifier, each node is marked if the data should be used in classification. Each node will only be included once, even if multiple chunk rows fall to that node. Some data may be included more than once because if a row stops at a parent node and another row stops at a child node, the data in the child node is also included in the parent node.

5. EXPERIMENT

Clump is being used as a software defect predictor for the NASA/MDP data sets[5] from the Promise Data Repository [21]. The NASA/MDP datasets used are: CM1, KC1, KC2, KC3, MC2, MW1, and PC1. These datasets were chosen for examination because they represent a wide variety of projects. The NASA datasets represent projects such as flight software, storage management, video guidance systems, and an experiment framework [24]. NASA utilizes sub contractors for the majority of its projects, so the seven projects represented in the MDP datasets represent the work of many different development teams.

The files have been modified from their original form to allow them to be used in cross company experiments[6]. They have been modified to unify the columns used, and their orderings. The data is stored in the arff format for easy integration with the WEKA experimenter [11].

Two different types of experiments are being run. First, Within Company, and second, Cross Company. Within Company trains the learning algorithms with a large portion of data from one data set, and tests the learned theories with the remainder of the data from that dataset. Cross Company analysis trains the learning algorithms using all but one of the available data sets, and tests the theories on the remaining whole dataset.

The Within Company data sets do a 5 by 5 way cross validation. This means that the instances are randomly ordered 5 times, and broken into 5 separate sets for each random ordering. Then 4 out of the 5 separate sets are used for training, and the remainder is used for testing in a round robin fashion.

The Cross Company experiments do a 7 way cross validation. The data is randomly ordered in the test dataset and in the combined training datasets 7 times, with each data set being the test dataset once.

The experiment compares several variations of Clump to other learners, mostly other rule tree learners: oner, naive bayes, j48, jRipper, Ridor, and locally weighted learning(LWL). The other learners come from Weka's Data Mining Software [11]. j48, jRipper, Ridor, and Clump represent rule tree methods. OneR represents a rule based method as well, but is not built on a rule tree. Naive Bayes and LWL are frequency count methods that are based on Bayes theorem [15]. All experiments are run using Ourmine [10], a shell environment for data mining experiments created at West Virginia University.

5.1 Parameters to Clump

9[4]A duplicate is defined as a specific training instance, not the combination of attribute values

9[5]The datasets can be obtained at `http://promisedata.org`

9[6]Cross company experiments use 6 of the 7 datasets for training, and the 7^{th} for testing

Runtime while	Naive Bayes	Ridor	Clump	LWL	j48/C4.5	RIPPER	OneR
Training	O(nk)	O($nk * log(kn)$)	O($3nk * log(kn)$)	O(n)	O($nk * log(n)$)	O($2nk * log(kn)$)	O(kn)
Testing	O(mk)	O(md)	O(mk) - O(mkd)	O(mnk)	O($m * log(m)$)	*Unlisted*	O(1)

Figure 6: Runtime Complexity of Clump, Naive Bayes [16], Ridor, OneR [2], LWL [7], j48/C4.5 [3], and RIPPER [8] on a dataset with n training cases, m testing cases, and k attributes.

```
for data in $datasets; do
  preProcess $data > processed.arff
  for((r=1;r<=$repeats;r++)); do
    seed=$RANDOM;
    for((bin=1;bin<=$bins;bin++)); do
      cat processed.arff |
        someArff --seed $seed
                 --bins $bins
                 --bin $bin
      for learner in $Learners; do
        $learner test.arff train.arff
          >> results.dat
      done
    done
  done
done
```

Figure 7: The pseudo code to run the experiment

	Within-Company		Cross-Company	
	Raw	Logging	Raw	Logging
Ridor	2.56	2.52	3.34	3.27
jRip	2.45	2.53	4.06	4.17
Clump	2	3	2	2

Figure 8: Number of rules

Clump takes five different parameters:

- Test file
- Training file
- Maximum level of mixedupedness
- Number of bins for discretization
- Split size

The test and training files are in the arff format described in Weka's documentation [11]. Each file has a header where the column information is included. The name of the data set, the names of the columns, and the type of column[7] is included. The body of the datafile contains a comma separated list of all instances in the dataset.

The maximum level of mixedupedness represents a threshold on the frequency of training instances not of the majority class allowed in a node before a new rule is created. This also effectively puts a lower bounds on the size of a node. A node can have no less than one more than twice the level of mixedupedness training instances.

The number of bins for discretization is used when creating the rules. The minimum and maximum values for a given attribute are taken. The range is then broken up into n equal width bins according to its value. More bins means a higher granularity in the rules created, but it also means a lower number of training instance in each node. Although it is often desired to reduce the number of instances in each cluster, there exists a lower bounds where too few instances can prevent new rules from being learned. The new rules could be prevented because the number of instances in each rule could violate the level of mixedupedness previously set.

The final parameter to Clump is the split size, or the number of testing instances to consider at one time. The chunk of testing instance being considered are placed in the rule tree. The training data stored in each node of the rule tree where a testing instance stops at is combined to form the training set for that chunk. This training set and the subset of the testing set is then passed to a Naive Bayes classifier for final classification. A higher split size will increase the number of training instances used in classification, and may add noise to the training set. Conversely, a small split size can gather too little data to make an accurate prediction.

5.2 Clump Configuration Values

Optimal values for the split size, number of bins, and level of mixedupedness were determined experimentally. Any value greater than or equal to 1 for the split size and level of mixupedness produced the same PD and PF results. If these values were ignored and set to 0, the number of rules created rose beyond the capabilities of Clump's runtime environment. The value with the largest impact on the performance of Clump was the number of bins used during discretization. A number of bins ranging between 2 and 10 were tested, and as the bin size chosen neared either extreme, the number of rules and runtime increased. A bin count of 3-5 produced the lowest run-time and rule counts. A split size of 5, bin minimum amount of mixupedness of 1, and 4 bins were chosen.

5.3 Experiments

There were two main experiments run: the Within-Company tests, and the Cross-Company tests. Each test was run twice. The first variation was run with the data in its raw form, and the second variation took the log of the numeric attributes before classification. Figure 9, Figure 10, Figure 11, and Figure 12 shows the results for the four different tests.

6. RESULTS

Figure 9 - Figure 12 show the results in quartile form. In both Within-Company and Cross-Company, when the numbers are not logged, Clump has a higher probability of detection(PD) than any of the other learners. At the same time, it also has the highest probability of failurePF. When the numerics are logged, all of the learners except Naive Bayes and Clump remain the same while Clump and Naive Bayes increase their PD. There are three things of note:

- In all of the experiments, The only learner that performs close to, or better than, Clump is Naive Bayes.

[7]The columns can be either continuous or discrete

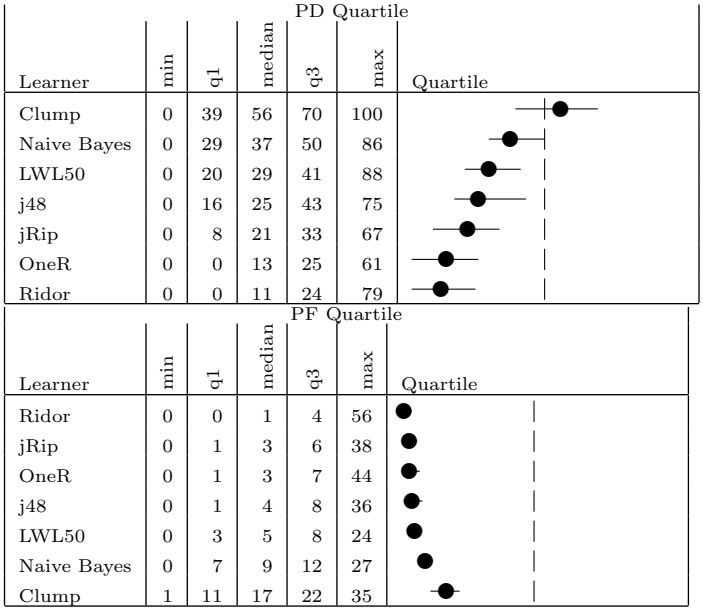

Figure 9: Results of the Within-Company tests with no preprocessing

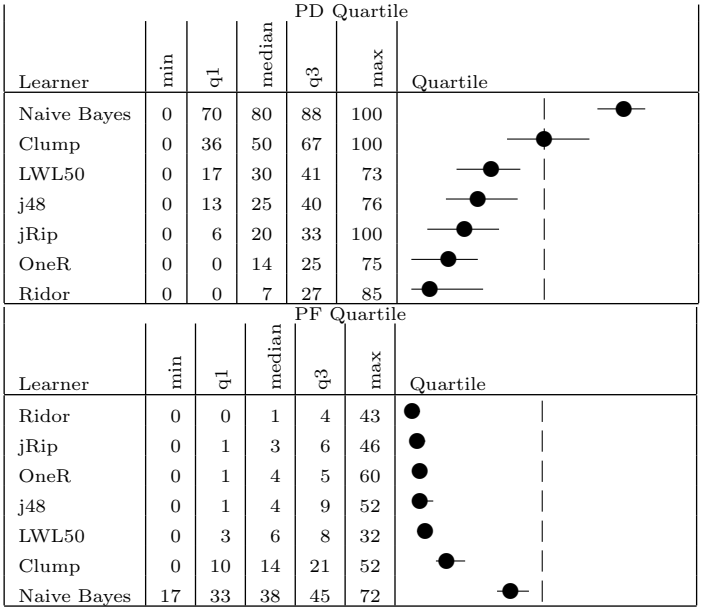

Figure 11: Results of the Within-Company tests with logging the numerics

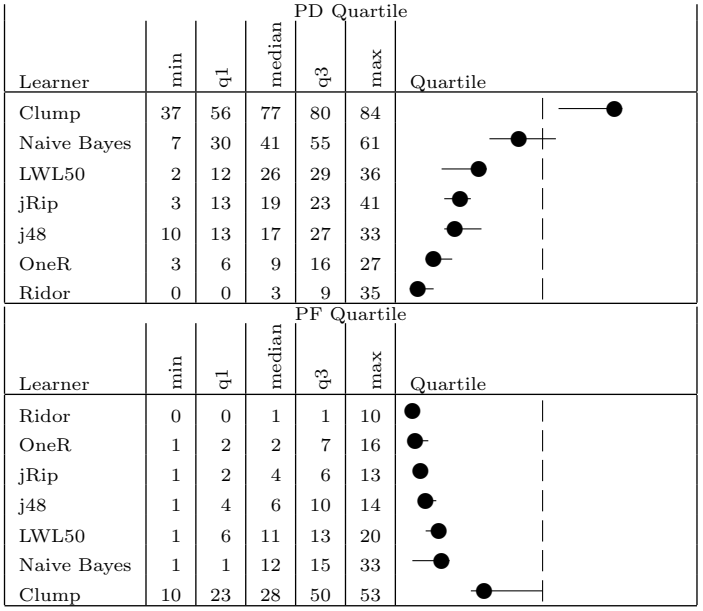

Figure 10: Results of the Cross-Company tests with no preprocessing

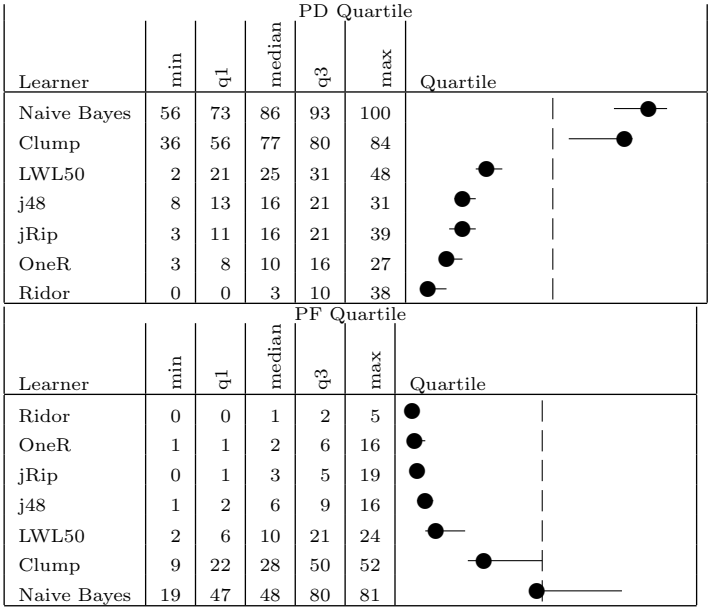

Figure 12: Results of the Cross-Company tests with logging the numerics

The PD's of other learners are between 22% and 52% lower than Clump with PF's that are only between

- Clump shows a high increase in PD when working on Cross-Company data, with a proportionally lower increase in PF.
- Taking the logarithm of the data increases the PD for Naive Bayes and Clump. The PF of Naive Bayes also increases, while the PF of Clump stays the same.

In Figure 9

7. FUTURE WORK

Clump represents a new approach to clustering and classification. Traditional clustering approaches [1, 12, 18, 19] cluster on independent attributes, while Clump clusters on dependant attributes. Clump also creates a maintainable rule tree. This rule tree allows for human interaction both during the testing and training phases of the algorithm.

7.1 Human Interaction

We plan on extending the functionality of Clump by introducing the human element. During the training phase, we propose that humans be presented with the possible patches at each node of the tree, and choose which patch best addresses the data. Several variations on this can be investigated. It may be best to show the user a subset of the training data at the node, and allow the user to choose which rule is "best" free hand. Another possibility would be to present the user with the top X rules ordered randomly, as determined by the current scoring algorithm described in §4.3.1. When presented with the new rules, the user could be shown the score of each rule, the rule's score's rank, or be shown no information about the score at all. By allowing the human to interact with the rule tree, the knowledge of domain experts can be exploited. Another alternative to this model would be to only query the domain expert when the score between the top rule candidates is similar enough to be ambiguous.

7.2 Interface Options

We expect to also expand Clump with a GUI, making it no longer just command line based. A side effect of this move will make gathering user feedback easier and more user friendly. An alternative would be integrating Clump with the WEKA framework.

7.3 Rule Creation

Internally, Clump uses a custom algorithm to score the possible rules. We plan to explore alternative rule scoring algorithms, examining them for increased performance. Rules that test more than one attribute are of great interest. These conjunctions of rules could isolate clusters of information faster than singleton rules alone.

8. CONCLUSION

[1] A. Beygelzimer and J. Langford. Cover trees for nearest neighbor. pages 97–104, 2006.

[2] G. Buddhinath and D. Derry. A simple enhancement to one rule classification.

[3] E. C, S. Ruggieri, and S. Ruggieri. Efficient c4.5, 2000.

[4] W. W. Cohen. Fast effective rule induction. In *ICML*, pages 115–123, 1995.

[5] P. Compton, G. Edwards, B. Kang, L. Lazarus, R. Malor, T. Menzies, P. Preston, A. Srinivasan, and C. Sammut. Ripple down rules: Possibilities and limitations, 1991.

[6] P. Domingos and M. J. Pazzani. On the optimality of the simple bayesian classifier under zero-one loss. *Machine Learning*, 29(2-3):103–130, 1997.

[7] E. Frank, M. Hall, and B. Pfahringer. Locally weighted naive bayes. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, pages 249–256. Morgan Kaufmann, 2003.

[8] J. FÃijrnkranz and G. Widmer. Incremental reduced error pruning, 1994.

[9] B. R. Gaines and P. Compton. Induction of ripple-down rules applied to modeling large databases, 1995.

[10] G. Gay, T. Menzies, B. Cukic, and B. Turhan. How to build repeatable experiments. In *PROMISE '09: Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, pages 1–9, New York, NY, USA, 2009. ACM.

[11] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update, 2009.

[12] G. Hamerly and C. Elkan. Learning the k in k-means. In *in k-means, NIPS*, page 2004, 2003.

[13] R. C. Holte. Very simple classification rules perform well on most commonly used datasets. In *Machine Learning*, pages 63–91, 1993.

[14] A. Jakulin and I. Bratko. Analyzing attribute dependencies. In *PKDD 2003, volume 2838 of LNAI*, pages 229–240. Springer-Verlag, 2003.

[15] E. T. Jaynes. *Probability Theory: The Logic of Science (Vol 1)*. Cambridge University Press, June 2003.

[16] G. John and P. Langley. Estimating continuous distributions in bayesian classifiers. In *In Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 338–345. Morgan Kaufmann, 1995.

[17] M. Minsky. Steps toward artificial intelligence. pages 406–450, 1995.

[18] A. W. Moore. An intoductory tutorial on kd-trees, 1991.

[19] S. M. Omohundro. Five balltree construction algorithms. Technical report, 1989.

[20] J. R. Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[21] J. Sayyad Shirabad and T. Menzies. The PROMISE Repository of Software Engineering Databases. School of Information Technology and Engineering, University of Ottawa, Canada, 2005.

[22] R. E. Schapire. The boosting approach to machine learning: An overview, 2002.

[23] N. Srinivasan and V. Vaidehi. Reduction of false alarm rate in detecting network anomaly using mahalanobis distance and similarity measure. In *Signal Processing, Communications and Networking, 2007. ICSCN 07 International Conference on*, pages 366–371. IEEE, 2007.

[24] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Softw.*

Engg., 14(5):540–578, 2009.

[25] Y. Yang and G. I. Webb. A comparative study of discretization methods for naive-bayes classifiers. In *In Proceedings of PKAW 2002: The 2002 Pacific Rim Knowledge Acquisition Workshop*, pages 159–173, 2002.