Representation Design and Brute-force Induction in a Boeing Manufacturing Domain

Patricia Riddle*
Boeing Computer Services
MS 7L-66
P.O. Box 24346
Seattle, WA 98124-0346

Richard Segal & Oren Etzioni
Dept. of Comp. Sci. & Eng.
University of Washington
Seattle, WA 98195

Abstract

We applied inductive classification techniques to data collected in a Boeing plant with the goal of uncovering possible flaws in the manufacturing process. This application led us to explore two aspects of classical decision-tree induction: (1) Preprocessing and postprocessing and (2) brute-force induction. For preprocessing and postprocessing, much of our effort was focused on the pre-processing of raw data to make it suitable for induction and the post-processing of learned rules to make them useful to factory personnel. For brute-force induction, in contrast with standard methods, which perform a greedy search of the space of decision trees, we formulated an algorithm that conducts an exhaustive, depth-bounded search for accurate predictive rules. We demonstrate the efficacy of our approach with specific examples of learned rules and by quantitative comparisons with decision-tree algorithms (C4 and CART).

1 Introduction

Continuous quality improvement in a manufacturing environment is a major challenge. The environment constantly changes because machines break down, vendors change, and new machines are added. We are exploring using inductive learning techniques to make process improvements in a Boeing manufacturing facility. We chose induction algorithms for our task for several reasons. These techniques have had many successes in real-world domains, they can focus on learning a single concept, and their results can be validated using test data and cross validation.

In this paper we will focus on the steps involved in applying induction algorithms to real-world data. We will discuss the obstacles involved in preprocessing the data (e.g., tuning the representation) and postprocessing the results (e.g., filtering out uninteresting patterns). We will discuss an example of this process where rules for process improvement were learned for a Boeing manufacturing environment. Initially we used IND (Buntine and Caruana, 1991) for inductive learning but later developed two new algorithms which are better suited to our domain.

The paper is organized as follows. In section 2 we discuss the Boeing manufacturing domain. Section 3 discusses the methodology we used for applying induction to our real-world domain. Section 4 discusses the new induction algorithms which we developed. Section 5 describes some of the useful rules we found using our methodology. In section 6 we discuss the related work. Section 7 discusses the future research. We are attempting to create a semi-automated toolbox which will allow factory personnel not familiar with induction to find patterns in manufacturing data. Section 8 concludes.

2 Domain

In today's factories, more and more of the manufacturing process is being handled by semi-automated cells. A semi-automated cell consists of multiple automated workstations. A nest of parts is automatically routed through a sequence of workstations. Operator intervention is only required in the case of a problem or in some designated manual operations (e.g., inspection). A semi-automated cell is controlled by a computer, the cell controller. When the cell encounters a problem, an alarm is sounded. The problem must then be fixed by the cell's operator. As alarm handling often involves processing delays, the cost of the recovery can add to product costs.

We are investigating patterns in operations records from a semi-automated work cell in a new Boeing plant. Examples of the types of patterns we would like to discover are "alarm B sounds thirty minutes after alarm A,"[1] "alarm C is twice as likely to sound on material X," and alarm cascades. An alarm cascade is when one problem may cause a chain reaction of related alarms. Another type of pattern we are exploring is predicting rejection tags, parts

---

[1]We can only provide a limited amount of concrete detail regarding the learning task due to Boeing's non-disclosure requirements.

which do not pass quality inspection. For example, we want to find patterns of the form "a rejection tag is four times as likely on parts made of batch C of material X." This information can be used to reduce the number of rejected parts; thus, saving money in wasted materials and manufacturing delays.

These patterns can be used to improve the manufacturing processes used in the factory. If a certain alarm is highly correlated with rejected parts, the factory might concentrate on preventing that alarm. If a certain batch of material is associated with several rejected parts, the factory might switch suppliers. This is important in our situation where the cell configuration is new because a significant portion of expertise for this cell configuration will be developed only during actual production. Even in more established situations where there exists a good base of expert knowledge, the factory changes rapidly; therefore, the knowledge about the factory must change to remain current. Learning is not required in real-time but can be done using an off-line analysis. The learned rules must be interpretable by the factory personnel so that they can implement changes to the factory process.

We prepared the data available for this domain using the methodology presented in Section 3. Two of the data sets we designed are as follows:

- *Failed part*: in the hope of increasing the factory's yield, we looked for patterns that predict when a manufactured part will fail inspection.

- *Occupancy time*: in the hope of increasing the factory's throughput, we searched for patterns that predict unusually long delays at the different machines.

The data set characteristics are shown below:

Data set	All Instances	Positive Instances	All Attributes	Continuous Attributes	All Tests	Continuous Tests
	519	34	1652	20	4380	1724
	1075	124	48	43	421	380

Continuous attributes are real valued; all other attributes are discrete valued. Tests are the total number of comparisons tested. For discrete attributes, there are two tests (equal and not equal) for each attribute value. For continuous attributes, there are two tests (greater than or less than) for each unique value for this attribute which appears in the data set. How these data sets were designed is the subject of the following section.

3 Using Induction in Real-World Domains

Despite the success of induction algorithms in real-world domains, there are several difficulties to overcome to produce good results with induction algorithms. The successful application of inductive algorithms requires three steps. The first step is deciding how to represent the data and how to configure the induction algorithm. The second step is running the induction algorithm. The third step is to process and analyze the results. In the following sections, we describe the preprocessing and postprocessing steps of the induction

3

process. The second step, running an induction algorithm, is usually not a problem when the other steps have been adequately addressed.

3.1 Preprocessing for Induction

Before an induction algorithm can be run, it is necessary to decide how to represent the data and how to configure the induction algorithm. Section 3.1.1 discusses the importance of choosing a good instance space. In section 3.1.2 we discuss how to determine relevant attributes. Section 3.1.3 discusses how the representation of the attributes affects learning. In section 3.1.4 we discuss how to represent time. Section 3.1.5 discusses setting the parameters of the induction algorithms.

Choosing Instances

In most traditional machine learning domains, the definition of an instance falls naturally out of the domain. One of the traditional machine learning data sets is the Cancer data set (Breiman *et al.*, 1984). An instance could be a person and all his symptoms and test results over the entire year, or an instance could be all the symptoms and test results performed during a particular day over all people. These two instance representations have the same content (i.e., the sets of exemplars are isomorphic), but they will result in radically different learned rules. For example, we could learn which symptoms are related to which diseases using the former representation, while we could learn whether certain symptoms or tests occurred more frequently during certain times of the year (e.g., expensive diagnostic tests are frequently run in the weeks preceding April 15th when income taxes are due) using the latter. In this particular example it is easy to see which representation is the better choice for each purpose, but in a new domain, designing the representation is not always straightforward.

In our domain it was unclear how to define an instance. For example, should a nest or a part be an instance? Should a single alarm or a set of alarms be an instance? The appropriate instance representation is largely determined by the goal of learning. We mainly used the two instance representations described in Section 2. The first representation records whether a part was rejected and the factory history of this part. The second representation records the occupancy time at a station for a part and the factory history of this part. We also used other instance representations such as an alarm occurrence and the events associated with it (i.e., those events which occur within a time window). Each of these instance representations was designed around its class attribute.

We are presently using statistical tests and clustering techniques to suggest which attributes would be successful class attributes. For instance, we tried different levels of abstractions for the value of the attribute *RejectedPart* (e.g. "Rejected vs. Non-Rejected", "TypeX Rejected vs. TypeY Rejected vs. Non-Rejected"). Trial learning runs were then used to determine which level of abstraction produced the best learned rules. Of course, the level of abstraction which produces the best rule might change over time as the data changes. We also tried different discretizations of time based on its distribution curve. For

instance, when trying to predict length of time at a particular cell station, we discretized the amount of time spent at each cell. To choose an appropriate discretization, we used the distribution curve of actual values to determine what ranges should be used.

3.1.2 Relevant Attributes

In traditional domains such as medical diagnosis, the appropriate attributes are known. These domains have been explored by people for many years, and a set of relevant attributes have been devised. It was unclear in our domain which attributes would be important. We have large amounts of data from multiple sources. If all this data were represented as attribute value pairs, the resulting data file would contain tens of thousands of attributes. The inclusion of many irrelevant attributes may cause problems for induction. The number of attributes is the biggest factor in the computational cost of most induction algorithms. In the presence of noisy data, large numbers of irrelevant attributes can produce overly complex trees, requiring very good pruning methods or stopping criteria.

We devised a method to focus on only those attributes which are useful for determining class membership. We ran our inductive algorithms using all the available attributes. Attributes which were never used in a rule were left out of subsequent runs. We also left out attributes which dominated the rules but were deemed to be uninteresting correlations. Uninteresting correlations take one of two forms: correlations of which the factory personnel are already aware or correlations which are novel but unhelpful. For instance, an unhelpful correlation is that more parts are rejected on Wednesdays. This type of finding usually indicates that more data is needed to eliminate poor correlations. Until additional data can be collected, the attribute *Day-of-the-Week* can be removed. Subsequent learned rules may not be as highly rated as those containing *Day-of-the-Week = Wednesday* but they will be more useful to the factory personnel. Since which attributes are useful will change over time, the full set of attributes should be returned to on a periodic basis.

3.1.3 Representation Tuning

Attribute representation has an effect on what types of patterns can be learned. For example, the chess domain can be represented in one of two ways. We could store for each square the piece it contains (e.g., a3=White-King or c2=Null), or we could store for each piece its position on the chess board (e.g., White-King=a3). The information content of an exemplar is the same in both formulations (i.e., they are isomorphic). These seemingly small alterations in our representation of a chess board will affect the kinds of patterns that can be learned. For instance, the concept *square X is empty* is easily represented in the first formulation, while in the second formulation it is difficult to represent.

We have experimented with several different attribute representations. The representation of our attributes, including the class attribute, is fairly complex. We use derived attributes (e.g., generalization hierarchies and sets) which are combinations of other primitive attributes. These data structures can be employed either to speed up the induction algorithm or to allow the algorithm to learn results otherwise unobtainable. For instance,

5

failed part	AlarmsOccurred
yes	1,2
no	2,3,4
yes	5

Figure 1: A sample knowledge base using set attributes.

in the rejected part abstraction hierarchy stated earlier, if no good rules are learned for the class *RejectedPart*, the user can specify a lower level of abstraction and learn rules for *TypeX-RejectedPart* and *TypeY-RejectedPart*. There are correlations which appear only at specific levels of abstraction. Learning at multiple levels of abstraction makes it possible to combine results found at different levels. This creates additional learned rules and therefore a better chance of finding useful results.

We extended the basic decision-tree algorithms to handle set-valued attributes. An example of a set-valued attribute is *AlarmsOccurred* which contains the set of alarms that occurred while processing a specific nest. The *AlarmsOccurred* attribute may contain a value such as {*alarm2, alarm8, alarm5*}. We consider tests on set-valued attributes which use subset membership as a splitting criterion. For example, we could test the *AlarmsOccurred* attribute using the test {*alarm2, alarm5*} $\subseteq$ *AlarmsOccurred*.

Without our extension, it is necessary to represent set attributes using a separate binary attribute for each element in the set's domain. The binary attribute would indicate for a given instance whether or not that element appears in the instance's set value for the given attribute. With this representation, testing whether an instance's value contains a subset causes a separate test to be added to the tree for each element in the subset. Since decision-tree algorithms prefer small trees, this representation is biased toward testing on small subsets. Our extension allows all subsets of alarms to be treated equally.

Our method for handling sets is to reformulate set-valued attributes using several new binary attributes. We generate one test for every subset of a set which actually appears in the data. That is for each example, we generate one test for each subset of its value. Any set X that is not a direct subset of a set appearing in the data is necessarily not a subset of any of the instances. Therefore, the test *"X $\subset$ AlarmsOccurred"* will not be true for any example. For this reason, the tests produced by this algorithm are complete and do not leave out any meaningful tests. We will now show an example of this technique being applied. Figure 1 shows a data set with three examples. The values of *AlarmsOccurred* for these instances are {1,2}, {2,3,4}, and {5} respectively. We generated three tests using the first example: {*1*} $\subseteq$ *AlarmsOccurred*, {*2*} $\subseteq$ *AlarmsOccurred*, and {*1,2*} $\subseteq$ *AlarmsOccurred*. Each of these tests are converted to the new attributes 1, 2, and 1,2 respectively. These attributes are true when the test associated with the attribute is true and false otherwise. This process is repeated for each example. The result is the new set of attributes shown in Figure 2.

This algorithm is efficient when the size of the largest set appearing in the data is small.

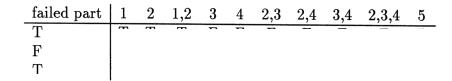| failed part	1	2	1,2	3	4	2,3	2,4	3,4	2,3,4	5
T										
F										
T										

Figure 2: . The knowledge base of Figure 1 reformulated using our set conversion algorithm to a representation using binary attributes.

The number of tests added by this algorithm is exponential in the size of the biggest set appearing in the data and linear in the number of examples. Because of similar patterns appearing in the data, the actual number of tests generated is usually significantly smaller. Since decision-tree style algorithms are linear in the number of tests, when this efficiency criterion is met, it is also practical to apply decision-tree algorithms to the newly reformulated representation.

3.1.4 Representing Time

Traditional machine-learning domains do not contain time series data. Treating time as a discrete attribute is usually inappropriate. Treating time as a continuous attribute can also cause problems. When time is treated as a continuous attribute, as the data covers a larger and larger time range, instances which are further and further apart will have time values close enough to match. For example, if data was collected for a year, two instances which occurred during the same week would not match on time; but if we collected data for ten years they would. Another simple approach is to use time differences (i.e., time since a certain alarm last was generated) instead of actual time values. This solves the problem of learning that alarm A sounds every five hours. However, the use of time differences does not allow the system to learn that alarm A was generated at 5:00 A.M. every day. We use a combination of exact times and time differences which gives us the advantages of both representations.

3.1.5 Setting Inductive Parameters

The IND decision tree package has several parameters which allow it to mimic CART, C4, Bayes Trees, and MDL Trees. These parameters affect the speed of the algorithm and the accuracy of the resulting decision tree. We tested different settings of the parameters to determine which performed best on our data. The C4 setting performed best, but the performance depended on the class, the attributes, and the training set we chose. Therefore, the parameter settings cannot be determined *a priori* but must be chosen via experimentation.

3.2 Postprocessing of Induction

Induction algorithms are general, industrial strength tools that have been successfully used for related tasks in the past. There is good reason to expect that, when properly tuned, the algorithms will detect a host of useful patterns. However, past experience also indicates the limitations of these methods. For example, when used to detect errors in DEC's XCON database, the C4 algorithm generated 78 predictions that were classified by human experts as follows: 26 useful, 45 useless, 7 unknown (Schlimmer, 1991a).

We want to provide guidance to the user as to which rules are best. Once the induction process delivers a likely rule, we statistically verify this rule for two reasons. We must know the precise effect of the antecedent of the rule on the consequent (i.e., is it two times as likely?). Quantifying our results helps to focus on those results whose possible remedies could have the greatest impact. Secondly, we do statistical tests (e.g., χ^2 tests) to verify that the pattern detected is statistically meaningful given the entire set of data. This allows us to avoid presenting those rules which have a high accuracy but have a low level of statistical significance. A rule can be 100% accurate when the only exemplar satisfying the antecedent also satisfies the consequent. But this rule will have a low level of statistical significance. We use a χ^2 test to order the learned rules by their statistical significance. This will relegate at least some of the uninteresting rules to the bottom of the list.

4   New Induction Techniques

Decision-tree algorithms were developed to produce good decision trees. Decision trees are used to determine the class of new examples given their attribute values. For instance, a decision tree can be used to determine whether a patient should be placed in a cardiac intensive care unit given his vital signs. Or, a decision tree can be used to determine if a mushroom is edible given its physical description. But in our domain, the goal is not classification. We are seeking rules that predict when a manufactured part is likely to fail inspection or when a delay will occur at a particular machine. Such rules do not provide a comprehensive characterization of the manufacturing process. Instead, they facilitate the identification of relatively rare, but potentially important, anomalies. Since predictive rules of this sort are rare, but potentially valuable, we refer to them informally as *nuggets.*

We need to extract a few good branches from a decision tree or set of decision trees which a human can understand. This is an important change of focus. All the metrics used in decision-tree algorithms (e.g. Gini index, information gain) are focused on producing the best overall tree, whereas we would be happy with a poor tree with one really great branch. We would also like to iterate on the data and get another great branch. In addition, the statistics presently computed on trees, including any cross-validated pruning, are based on the accuracy or predictive ability of the whole tree. We are more interested in the accuracy or predictive ability of individual branches.

We have developed two new inductive algorithms that are better suited to finding the nuggets. Gold-digger is a modification of the standard decision-tree algorithm designed to

remove its bias for finding high-coverage rules. Brute expands on Gold-digger by replacing Gold-digger's greedy search with an exhaustive search. The ideas behind Gold-digger and Brute are best understood by considering why decision trees fail to find good nuggets. The main reason is that the classical test selection functions used to grow decision trees are inappropriate for finding nuggets. Classical test selection functions, such as the Gini index or ID3's information gain measure, have the following form:

$$\max_{t \in T} \left(\sum_{i=1}^{v(t)} \frac{p_i + n_i}{|E|} P(i) \right)$$

T	The set of possible tests.
E	The set of examples at the node with cardinality $|E|$.
$v(t)$	The number of possible values of the test t.
p_i	The number of positive examples that satisfy the ith value of t.
n_i	The number of negative examples that satisfy the ith value of t.
$P(i)$	A measure of the purity of the examples that satisfy the value i.

To evaluate each test, this function averages the purity of each branch, weighted by the number of examples that satisfy the test value at the branch. Although the exact measure of purity varies from one algorithm to another, standard measures exhibit fairly similar behavior in practice (Breiman *et al.*, 1984, Mingers, 1989).

Test selection functions of this form can lead decision-tree algorithms to overlook valuable nuggets for several reasons. Due to the weighted average, the function will prefer a test that results in a balanced tree, where purity is increased for most of the instances, over a test that yields high purity for a relatively small subset of the data, but low purity for the rest. Yet, a single high-purity branch might yield a critical nugget. An example of this bias is shown in Figure 3. In the figure, P and N denote the number of positive and negative examples respectively that satisfy the indicated test. Adding the test *T1 = True* appears to be a good choice because it yields a nugget with 100% accuracy on the training data. However, standard selection functions weight the purity of each branch of the test. Since the *T1 = False* branch has very low purity, the overall score for test *T1* is low. In fact, standard selection functions tend to prefer test *T2* because it has better then average purity for both of its branches.

To avoid missing valuable branches, we prefer a test selection function that computes the maximum of the branch scores rather than their weighted average:

$$\max_{t \in T} \left(\max_{i=1..v(t)} P(i) \right)$$

The maximum focuses on only one of the (potentially many) branches of the test and ignores the number of examples at each branch, considering only its purity.

The tests chosen near the root of a decision tree affect a large number of leaves. In order to develop compact trees, it makes sense to weight purity by the number of examples. Our

9

```
        T1                                    T2
   True/ \False                          True/ \False
      /   \                                 /   \
     ↓     ↓                               ↓     ↓
  P: 100  P: 450                        P: 175  P: 375
  N:   0  N: 450                        N: 325  N: 125
```
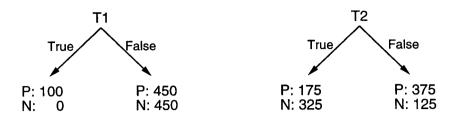
Figure 3: A numerical example illustrating how standard test selection functions can overlook nuggets. Both the Gini Index and the information gain function prefer T2 over T1, whereas our selection function prefers T1.

preferred function allows a test that achieves purity of 0.80, on a small subset of the data, to obscure a different test that achieves purity of 0.75 on the entire data set. However, such considerations arise because decision-tree algorithms build a single classifier for the entire data set. Gold-digger and Brute generate multiple, completely distinct, predictive rules. Thus, choosing the test with the maximal-purity branch to classify a small subset of the data does not commit us to use that test when classifying the rest of the data.

Another limitation of standard purity functions is that they treat negative and positive examples symmetrically. In many applications, the positive (or, equivalently, negative) examples represent some surprising occurrence or anomaly we wish to monitor. In our Boeing application, for example, we are interested in anticipating when manufactured parts will fail inspection. A rule predicting that a part will not fail inspection could be very accurate but quite useless. Thus, instead of treating negative and positive examples symmetrically, we prefer a function that selects the test with maximal purity relative to the positive examples and ignores the purity of negative examples altogether. The following purity function, which simply computes the proportion of positive examples, has the desired behavior:

$$P(i) = \frac{p_i}{\cdot \cdot \cdot}$$

Note that although our choice for $P(i)$ could be approximated by introducing the appropriate priors or misclassification costs into classical selection functions (Breiman et $al.$, 1984), choosing the test with the maximal accuracy branch cannot be approximated.

In summary, we believe that, when searching for nuggets, the max-accuracy test-selection function shown below is preferable to standard functions.[2]

$$\texttt{max-accuracy}(T, E) = \max_{t \in T} \left(\max_{i=1..v(t)} \frac{p_i}{p_i + n_i} \right) \tag{1}$$

This function is equivalent to the select-literal function employed by GREEDY3 (Pagallo and Haussler, 1990, page 85). We make one modification. The function in Eq. 1 runs the

---

[2]Note that the function applies readily to multi-outcome tests and to multi-class data sets. It merely requires labeling some subset of the classes as positive.

```
GoldDigger(trainEGS,pruneEGS):rule-set
    rule-set := empty;
    REPEAT
        rule := GoldDigger_GrowRule(trainEGS);
        PrunedRule := GoldDigger_PruneRule(rule,pruneEGS);
        trainEGS := RemovePositives_ThatMatchRule(PrunedRule,trainEGS);
        rule-set := rule-set + PrunedRule;
    UNTIL (positives(trainEGS) < MinPositives) or empty(PrunedRule);
    RETURN rule-set;
END

GoldDigger_GrowRule(T,EGS):rule;
    IF AllPositive(EGS) THEN RETURN MakeLeaf();
    BestTest := max-accuracy(T,EGS);
    EGS := match_test(EGS, BestTest);
    RETURN MakeNode(BestTest, GoldDigger_MakeRule(T,EGS));
END
```

Table   Gold-digger's algorithm. The `max-accuracy` function is defined in Eq. 1

risk of picking a test that covers very few examples. In the extreme, the test could have a branch that contains exactly one example — a positive one. In this case, the accuracy of the branch will be 100% and it will be chosen. To avoid this problem, Gold-digger ignores branches that contain fewer than `MinPositives` positive examples, where `MinPositives` is a constant that is set by the user. In our experiments with Gold-digger, `MinPositives` was set to 20% of the total number of positive examples. The addition of the `MinPositives` parameter had a significant impact on the function's success in practice.

4.1 Gold-Digger

Gold-digger is a greedy algorithm that aggressively searches for nuggets using the `max-accuracy` function. Gold-digger was derived from classical decision-tree algorithms by addressing the problems described in the previous section. The main difference between Gold-digger and a standard decision-tree algorithm is that Gold-digger expands only one branch of the decision tree, the branch that appears to contain the best predictive rule as measured by the `max-accuracy` function. By only expanding one branch, Gold-digger avoids the pitfalls associated with trying to build a balanced tree. Finally, Gold-digger utilizes a pruning method very similar to that used by CART (Breiman *et al.*, 1984). A pseudo-code description of Gold-digger appears in Table 1.

As described thus far, Gold-digger would only find a single predictive rule. This is unsatisfactory since there may be multiple nuggets to be uncovered in the data. The obvious

solution is to run Gold-digger multiple times. However, if Gold-digger is repeatedly run on the same training and pruning data, it will produce the exact same rule. We solve this problem by running Gold-digger multiple times, each time removing the positive examples covered by the learned rule from the training data. This practice guarantees that when we run Gold-digger again, it will attempt to find a pattern in a different subset of the initial training data. In essence, this iteration procedure learns a disjunctive-normal form description of the target concept, one disjunct at a time. When the number of positive examples drops below `MinPositives` or when the last learned rule is pruned away, further iterations would fail to produce additional rules, so the algorithm terminates.

In essence, a single Gold-digger iteration corresponds to a hill-climbing search through the space of conjunctive rules. The limitations of hill-climbing are well documented; Gold-digger may find itself in a local maximum and overlook critical nuggets. For illustration, consider the nugget "IF *Material=material-1* AND *Location=machine-3* THEN Status=alarm." For Gold-digger to learn this nugget, it must greedily choose to start a rule either with the test *Material=material-1* or with the test *Location=machine-3*. Gold-digger will choose one of these tests to start a rule only if one of them has the highest accuracy of any test that Gold-digger considers. When this condition is not met, the above nugget will be missed.

4.1.1 Experimental Results

We tested Gold-digger on the two data sets discussed in Section 2. To test the different algorithms, we split the data into two subsets: 70% for training and 30% for testing. Each algorithm used its own method for picking a subset of the training data for pruning. In each experimental run, the algorithms were given the same training and test data. To reduce bias due to the data split, we repeated the runs ten times, randomly selecting different examples for training and testing. The results shown in Table 2 are averaged over each run. The *base rate* is the percentage of the test data that is positive.[3]

The *accuracy* of a rule is the percentage of test examples matched by the rule that are correctly classified. For instance, if a rule matches ten examples and four of them are positive, then the rule's accuracy is 40%.[4] The *positive coverage* of a rule is the percentage of the positive test data it covers. The accuracy and coverage columns figures are averaged over each algorithm's rule set. The *cumulative positive coverage* column shows the total positive coverage of each algorithm's rule set. The *rules* column shows the number of rules found by each algorithm. The *statistically significant rules* column shows the number of rules deemed statistically significant by a χ^2 test at the level of $p = 0.005$. Finally, the *percent collapse* column shows, for each algorithm, the percentage of runs which failed to produce any rules.

---

[3] Using the positive training examples directly as predictive rules performed far worse than the base rate and worse than any of the inductive algorithms tested. This demonstrates that our data sets are non-trivial, and the overlap between training and test data is very small.

[4] Our rules invariably predict a positive classification. The decision-tree output of C4 or CART is automatically converted to the equivalent rule set. Due to the nature of the application, we only consider rules that predict a positive classification. Note that C4 and CART refer to the IND (Buntine and Caruana, 1991) re-implementations of these algorithms.

The results shown are averaged over the runs that did not collapse.

Gold-digger performed better than either CART or C4 on the occupancy data. Gold-digger's rules were more accurate and had higher coverage than those produced by CART and C4. The higher coverage of Gold-digger's rules is particularly interesting because Gold-digger's selection function was designed to eliminate the high-coverage bias of standard selection functions. Although decision-tree algorithms prefer to add tests with high coverage, the cumulative effect of adding several tests to a branch can result in a rule with low positive coverage.

On the failed-part data, with exception of one run (CART on seed 6), both CART and C4 were unable to produce any rules. Most of the time, Gold-digger collapsed on this difficult data set. When Gold-digger did not collapse, it was only able to produce a single rule per run. Of the three rules found, only one was effective with 37.5% accuracy.

Gold-digger's difficulties with this data set indicate that its biggest limitation is how it iterates to find additional rules. Gold-digger iterates to find additional rules by removing the positive examples covered by the current rule and recursing on the remaining examples. If the current rule collapses during pruning, Gold-digger has no examples to remove and thus cannot learn additional rules. Therefore, a single poor rule found by Gold-digger can prevent it from finding any additional rules. Another problem with Gold-digger is its practice of discarding (scarce!) positive training examples at each iteration. As a result, Gold-digger can underestimate the number of positive examples covered by a new nugget and fail to learn it as a consequence. This occurs when the set of examples covered by the new nugget overlaps with the examples covered by previous nuggets. Furthermore, as positive data is repeatedly discarded, the algorithm finds itself with less and less grist for its inductive mill. In our experiments, Gold-digger was unable to find more than seven nuggets per run. We considered several possible improvements to Gold-digger's iterative procedure. However, the Brute algorithm (which dispenses with Gold-digger's iterations) appeared to be superior, so we did not pursue improvements to Gold-digger.

4.2   Brute

We devised Brute to address the limitations of the greedy, hill-climbing search and the ad hoc iterative procedure employed by Gold-digger. Brute performs a massive, brute-force search for accurate predictive rules. As demonstrated by the success of chess-playing machines such as Deep-thought, judicious application of brute-force search can yield impressive results. Is brute-force induction practical, though, in the absence of the special-purpose search hardware found in Deep-thought? We describe Brute's algorithm and then use both analytic calculations and experimental results to show that brute-force induction is feasible and yields valuable nuggets in practice.

In essence, Brute conducts an exhaustive depth-bounded search for accurate conjunctive rules. Brute's use of a depth bound was motivated by the observation that accurate predictive rules tend to be short and by the need to make the exhaustive search tractable. Brute avoids redundant search by considering conjuncts in a canonical order. This optimization guarantees

OCCUPANCY-TIME DATA (Base rate = 11.1%)

Algorithm	Accuracy	Positive Coverage	Cumulative Positive Coverage	Rules	Statistically Significant Rules	Percent Collapse
Brute (d=4)	38.4%	10.6%	91.0%	50.0	20.5	0%
— Top ten rules	50.8%	7.8%	43.8%	10.0	5.2	0%
Beam-Brute (d=5, b=10)	44.2%	12.8%	82.7%	22.9	12.5	0%
— Top ten rules	52.0%	8.2%	42.8%	10.0	5.6	0%
Gold-digger	46.0%	18.9%	61.6%	4.4	2.8	0%
C4	29.9%	2.9%	25.5%	9.2	2.1	0%
CART	21.6%	1.7%	34.3%	21.3	3.4	0%
— Top ten rules	21.7%	3.8%	21.7%	8.2	1.3	0%

FAILED-PART DATA (Base rate 7.2%)

Algorithm	Accuracy	Positive Coverage	Cumulative Positive Coverage	Rules	Statistically Significant Rules	Percent Collapse
Brute (d=2)	35.0%	23.5%	59.7%	50.0	25.3	0%
— Top ten rules	43.6%	20.0%	43.9%	10.0	5.8	0%
Beam-Brute (d=5, b=10)	26.8%	11.9%	51.8%	24.2	5.9	0%
— Top ten rules	30.5%	10.0%	32.0%	9.8	2.8	0%
Gold-digger	12.5%	11.1%	11.1%	1.0	0.3	70%
C4	—	—	—	—	—	100%
CART (seed 6)	27.8	4.9	44.4	9.0	3.0	—
Brute (seed 6)	42.9	31.1	44.4	10.0	9.0	

Note: CART collapsed on all seeds other than seed 6.

Table 2: Comparison of Brute, Gold-digger, CART, and C4 on the Boeing data sets. In the table, d refers to Brute's search depth, and b refers to the size of Beam-Brute's search beam.

```
BruteSearch(trainEGS, rule)
  IF NumConjuncts(rule) < MaxDepth THEN
    FORALL test in tests DO
      NewRule := AddConjunct(rule, test);
      IF PositivesCovered(NewRule) > MinPositives THEN
        IF DifferentThanParents(NewRule)
          THEN HeapInsert(BestRules, NewRule);
        Brute(trainEGS, NewRule);
      END
    END
  END
END.

Brute(trainEGS, rule)
  rule := EmptyRule();
  BruteSearch(trainEGS, rule);
  FOR the best R rules in BestRules DO
    Output(rule);
  END
END.
```

Table 3: A simplified version of the Brute algorithm for searching the space of predictive rules. *BestRules* stores the R best rules encountered, ranked according to their accuracy. *MaxDepth*, *MinPositives*, and R are set by the user.

that Brute will consider each rule, whose length is smaller than the depth bound, exactly once.[5] Brute retains the R rules found to be most accurate on the training data. In our experiments, R was set to 50. A pseudo-code description of Brute appears in Table 3.

As described above, Brute would learn many variations of a single rule. Often, when the rule $A \wedge B$ is very accurate, the rule $A \wedge B \wedge C$ is also quite accurate, particularly when C rules out very few of the instances covered by the original rule. When this occurs, variations on a single rule can crowd out other useful rules from the limited rule set retained by Brute. Left unchecked, this problem would reduce the number of useful nuggets found by Brute. Our solution is only to keep the rule $A \wedge B \wedge C$ when it is significantly more accurate than its parents: $A \wedge B$, $A \wedge C$, and $B \wedge C$. We test for a significant increase in accuracy using a χ^2 test at the level of $p = 0.1$.

Brute would also learn many variations of rules that use continuous attributes if it were left unchecked. Often a rule containing a test of the form $val < Y$ will have similar perfor-

---

[5]Avoiding redundant search is referred to as *systematicity* in the planning literature (McAllester and Rosenblitt, 1991). Brute is systematic in this sense.

mance to the same rule with this test replaced with $val < Z$, where Z is numerically close to Y. To alleviate this problem, Brute only stores the rule with the test $val < Y$ such that for all other Z, $val < Y$ performs better than $val < Z$.

Brute uses two techniques to reduce its search time. First, it does not expand rules which do not cover enough positive examples to meet the `MinPositives` requirement. Any specialization of a rule that does not meet the `MinPositives` requirement will itself not meet the requirement and therefore does not need to be considered. Pruning rules with too few positive examples greatly reduces the search space. Second, Brute stops when it has found R rules with 100% accuracy. Since 100% accuracy is the highest possible value of Brute's scoring function, once it has found R rules with this score, no other rules need to be considered.

For the sake of tractability, Brute can reduce its branching factor to a fixed-width beam b. We refer to this variant of Brute as Beam-Brute. Beam-Brute uses Gold-digger's test-selection function to choose the b most promising tests at each branching point in its search. Beam-Brute dispenses with Brute's canonical ordering of tests, which results in some redundant search, but provides Beam-Brute with multiple paths to the same rule. This change is important. Since Beam-Brute does not carry out an exhaustive search, an inappropriate canonical ordering could interfere with Beam-Brute's ability to find valuable rules.

4.2.1 Experimental Results

Due to the large number of tests in the failed-part data set, Brute was only able to search nugget space to depth two. Even so, it did not collapse on any run and yielded quite a few nuggets, outperforming all other algorithms including Beam-Brute. Brute appears to be much less prone to collapse than the other algorithms including Gold-digger.

On the occupancy-time data set, Brute was able to search to depth four and outperform both C4 and CART. Beam-Brute was able to find slightly more accurate rules on average, but the difference is not statistically significant. Brute's average accuracy was 38.4% compared with only 21.6% for CART and 29.9% for C4.[6] Arguably, our comparison actually understates the quality of Brute's rule set because Brute outputs far more rules than C4 or CART. If we only consider Brute's top ten rules, its average accuracy increases from 38.4% to 50.8%. C4 found less than ten rules, and the average accuracy for CART's top ten rules was 21.7%.

Finally, both variants of Brute found far more statistically significant rules than C4 or CART on this data set. This is important since each statistically significant rule represents a potentially useful nugget. By generating many more statistically significant rules, Brute provides more opportunities to improve Boeing's manufacturing processes.

While Brute outperforms classical decision-tree algorithms on the Boeing data, we need to also consider its running time. Table 4 compares the running times of the different algorithms. We see that Brute's running time is acceptable and that Beam-Brute is exceedingly

---

[6]The difference in accuracy between Brute and C4 is statistically significant. Using a one-tailed, paired T-test we found p=0.01.

RUNNING TIME

	Failed-part data	Occupancy data
Brute (d=2)	0:49	—
Brute (d=4)	—	33:31
Beam-Brute (d=5, b=10)	2:22	0:18
Gold-digger	0:14	0:02
CART	21:34	0:22
C4	1:17	0:03

Table 4: A comparison of the running time of each algorithm on the two data-sets. All times are in minutes:seconds of CPU time.

fast (less than thirty seconds on the occupancy-time data and about two and one-half minutes on the failed-part data).

We conclude that Brute is the algorithm of choice for finding nuggets in data similar to the Boeing data. Will Brute or Beam-Brute be tractable on larger data sets? To answer this question, we turn to a complexity analysis of the algorithm.

4.2.2 Complexity Analysis

The number of decision trees is doubly exponential in the number of tests T. T can exceed 1,000 in real-world applications, implying that any straightforward enumeration of decision trees is impractical (Quinlan, 1986, page 87). However, we are seeking concise conjunctive rules. The number of conjunctive rules is singly exponential in T, and the number of concise conjunctive rules is even smaller. The number of rules of length d is the number of subsets of T of size d, which is $\binom{T}{d}$ (read as "T choose d"). Thus, the number of rules of length d or smaller is as follows:

$$R(T, d) = \sum_{i=1}^{d} \binom{T}{i} = \sum_{i=1}^{d} \frac{T!}{i!(T-i)!}$$

$R(T, d)$ is considerably smaller than T^d. Yet, for any fixed d, T^d is polynomial in T. It follows that when d is small, Brute's search space is manageable, even for very large T.

Brute evaluates each rule by counting how many training examples it covers. In the worst case, each evaluation takes time linear in the number of training examples E. Thus, the overall complexity of Brute is $O(R(T, d)E)$. We can calculate an upper bound for Brute's running time as a function of s, the machines speed, using the following formula:

$$\text{CPU seconds} \leq \frac{R(T, d)}{s} E$$

On a SPARC-10 workstation, Brute can evaluate approximately 75,000 rules per second when learning with 500 training examples. Therefore, on a SPARC-10, $s = 3.75x10^7$. We

17

can use this information to choose an appropriate depth bound for Brute. For example, given the occupancy-time data set, which contains 421 tests and 752 training examples, the formula predicts that a search to depth four will take no more than 7 hours. Brute's actual performance is significantly better at 33 minutes because it prunes a significant portion of the search space. Nevertheless, the estimate indicates that searching to depth four is in the realm of possibility.

We see that when the number of tests is large, Brute can only perform a shallow search. However, if the depth reached by Brute is not sufficient (and faster machines or more CPU time are not available!), Beam-Brute can be used to carry out deeper searches. On the failed-part data, Brute would take about 700 years to conduct a search to depth five, assuming it did not find enough 100% accurate rules to terminate early. However, Beam-Brute ,using a fixed-width beam of 10, requires less than three minutes and still produces good results. In general, the complexity of Beam-Brute is $O(b^{d-1}TE)$. Since the running time of both Brute and Beam-Brute is linear in the number of examples, both algorithms are able to process the large data sets often found in practical applications. Beam-Brute is particularly attractive for very large applications since both the depth bound and the branching factor of the search can be set by the user.

4.3 Discussion

While the number of decision trees appears to preclude any kind of exhaustive search, the number of concise conjunctive rules is much smaller. The number of concise conjunctive rules is polynomial in the number of tests, given any fixed bound on rule length. The complexity of an exhaustive search is still only linear in the number of training examples. These observations, coupled with the calculations in Section 4.2.2, led us to consider brute-force induction as a feasible option. Instead of creating decision trees and converting them into rules, we decided to conduct a massive (but direct) search for the rules.

As it turned out, thirty-five CPU minutes on a SPARC-10 workstation enabled us to conduct extensive searches in the context of our Boeing application. As shown in Section 4.1.1, this search uncovered sets of predictive rules with higher accuracy and superior coverage compared with the rules produced by standard decision-tree algorithms. Brute-force induction may not be feasible in cases which require full coverage of the data set or deep searches of the hypothesis space. However, our experimental results and complexity analysis suggest, that when seeking concise predictive rules, brute-force induction is an option worth considering.

The limitations of Gold-digger are that it is a greedy algorithm and so can always miss good rules when a combination of attributes does well while any single attribute of that combination does poorly. Since it has no coverage component, it can also focus in on rules which cover too small of a set to be considered statistically significant.

The limitations of Brute are obviously its computational complexity. As the feature space gets too large, it becomes infeasible to run brute without limiting its beam. Whether Beam-Brute is a better choice than other approximate algorithms depends mostly on the size of the

domain and how many conjuncts the rules must contain to be effective. Recent work (Holte, 1993) has suggested that many domains are adequately represented by surprisingly short rules.

5 The Boeing Success Story

Figure 4 shows the rules we found applying our methodology to the Boeing data. A rule of the form "If W, then it is N times more likely to Y," means that when W is true, Y is N times more likely to occur than the unconditional frequency of Y. These rules were used by factory personnel as clues on how to improve the factory. For instance, the first rule in Figure 4 led the factory personnel to examine the results of the material acceptance tests for batch B. Apparently, batch B had just barely passed the acceptance tests. As a result, they have implemented a new system for tracking batches and are currently working with the vendor of this material to improve quality. The second rule is interesting because stationA and stationB were supposedly identical machines. Their different roles in rejected parts led factory personnel to examine the equipment and scheduled maintenance in more detail to determine what changes could be made to bring stationB up to the high standards of stationA. The third rule led the factory personnel to examine how delays before stationB could cause rejected parts. This gives a sense of the type of rules which were derived and how they were used to improve the factory.

Several important aspects of this work for a manufacturing environment are: 1) the rules can be discovered automatically without much intervention from the factory personnel, 2) the rules are statistically validated both in terms of their statistical significance and in terms of their potential value for improving the manufacturing process, and 3) as the factory environment changes over time the techniques can easily be re-applied to derive new rules which reflect the new state of the factory.

6 Related Work

There is scant related work to report in the area of representation design. In (Evans and Fisher, In press), Evans and Fisher discuss how they determined the class attributes *Banded* and *NotBanded*. They also discuss the iterative process they used for attribute selection. (Fayyad *et al.*, 1992) discusses the use of derived attributes. In their domain, adding two user defined attributes increased accuracy so significantly that they chose to automate the process. They derived new attributes by applying induction recursively. (Buntine and Stirling, 1991) discusses altering inductively learned rules to make them more acceptable to human experts. In particular, they discuss the importance of generating statistics concerning the applicability and confidence in each rule.

Rivest describes an algorithm for PAC (Probably Almost Correct) learning decision lists in (Rivest, 1987) that has similarities to both Brute and Gold-digger. Rivest's algorithm uses the same iterative structure as Gold-digger, but replaces Gold-digger's greedy search

If the nest's material is A and it is from batch B, then it is 4 times as likely to have a TypeC reject.

A nest which goes through station B is 2 times as likely to be rejected as a nest which goes through station A.

If a nest is at station A before station B for over 32 minutes, then it is 4.5 times as likely to be a TypeC reject.

If a nest is at station X over 51 minutes, then it is 3 times as likely to be rejected

If the nest's material is X and it is at station Y before it goes to station Z, then it is 2 times as likely to be a TypeW reject.

A nest is 2 times as likely to get a TypeZ reject on a Friday.

Part A is 1.5 times as likely to get a TypeX reject

A nest which spends less than 9 minutes in station X is 6 times more likely to be OCC3 than a nest which spends more than 9 minutes in station X. OCC3 means that a nest spent 6 to 21 minutes in station Z.

If the nest's material is A, then the probability of alarm X reduces by 25%.

Alarm Y is almost 2 times as likely not to occur in the first five days of the month.

Figure 4: Rules learned in the Boeing factory domain.

algorithm with one that does exhaustive search. Since Rivest's algorithm is designed for PAC learning, it only searches for 100% accurate rules and assumes its data is noise free. Gold-digger can be viewed as an extension of Rivest's algorithm to perform a greedy rather than exhaustive search, to handle noisy data, and to find less than 100% accurate rules. Gold-digger provides two mechanisms to handle noisy data. First, it only considers rules which cover a minimum number of positive examples. Second, Gold-digger prunes its rules using independent pruning data. The change to greedy search from exhaustive search seems questionable in light of Brute's success. However, adding exhaustive search to Gold-digger would provide little benefit for finding nuggets. First, adding exhaustive search does nothing to alleviate Gold-digger's central problem of frequently collapsing. Second, as Brute illustrates, it is possible to search for multiple nuggets using a single exhaustive search. However, a single exhaustive search would not work for learning decision lists because the performance of each rule must be considered in relation to its position in the decision list.

GREEDY3, developed by Pagallo and Haussler, also extends Rivest's algorithm to do greedy rather than exhaustive search (Pagallo and Haussler, 1990). The result is very much like Gold-digger. GREEDY3 uses the same iterative structure as Gold-digger and uses a selection function very similar to max-accuracy. GREEDY3's selection function differs from Gold-digger's in that it does not have a MinPositives parameter. Like Rivest's algorithm, GREEDY3 only searches for 100% accurate rules and does not handle noisy data. GREEDY3 does have a pruning algorithm, but it is not designed to handle noise. GREEDY3's pruning algorithm is designed to eliminate crossover terms which can increase the size of a decision list.

Schlimmer's CARPER system (Schlimmer, 1991b) also pursues a branch rather than building an entire tree. However, CARPER does this as an efficiency optimization, not as a means of improving the classification accuracy of the algorithm. CARPER does not share Gold-digger's iterative structure or the max-accuracy function.

One of the first systems to apply massive search to learning is Weiss, Galen, and Tadepalli's PVM system (Weiss et al., 1990). PVM uses a beam search and pruning heuristics to do a semi-exhaustive search for short classification rules. PVM differs from Brute in that PVM is designed to find classification rules rather than predictive rules and because its search is not complete. Since PVM is interested in classification rules, PVM considers rules containing both conjunction and disjunction. Brute only considers conjunctive rules. This difference arises from the slightly different learning tasks they address. For learning nuggets each disjunct in a disjunctive rule would have to be verified independently of the other disjuncts to determine whether the pattern it identifies is justified. For classification, it is necessary to consider how the individual disjuncts interact to form a classifier; thus, they cannot be considered independently. Brute's restriction to purely conjunctive rules greatly reduces the search space, allowing greater depths to be pursued.

Smyth and Goodman also recognized that exhaustive search of conjunctive rules can effectively be used for searching for nuggets (Smyth and Goodman, 1991). Their system, ITRULE, also performs a depth bounded exhaustive search for good predictive rules. ITRULE uses an evaluation function based on information theory that has a greater bias

towards finding high-coverage rules than our max-accuracy function. Brute's biggest improvement over the basic ITRULE algorithm is its ability to prevent syntactically similar rules from appearing in the final rule set. Brute does this by removing rules statistically similar to their parent rules and by removing rules that perform worse than the same rule with different values for their numeric comparisons.

Finally, a recent paper by Schlimmer uses exhaustive depth-bounded search to find determinations (Schlimmer, 1993). A determination for an attribute is a set of attributes which are relevant for determining the attribute's value. Although not directly useful for finding nuggets, this work does further illustrate that exhaustive search can be a valuable tool.

7   Future Directions

While induction has been credited with many successes, much work must be performed before an induction algorithm can be used. Applying induction requires careful tuning and analysis by a machine learning engineer. This is similar to the part played in the past by the knowledge engineer for expert systems. As expert systems developed, the field of knowledge acquisition formed to alleviate reliance on a knowledge engineer and to allow domain experts to build expert systems themselves. Similarly, it is now time for the field of inductive learning to allow domain experts to take charge of induction.

We are attempting to create a semi-automated toolbox which will allow factory personnel, who are not familiar with machine learning, to look for patterns in their own data. This paper discusses the steps we believe are involved and how these steps can be applied manually. We believe that automating this process as much as possible is the necessary next step for getting induction algorithms into everyday use in industrial settings.

A semi-automated toolbox must be able to be used directly by factory personnel. It would have to allow factory personnel to choose the algorithm parameters and data representation. Since the factory personnel are not machine learning specialists, the tuning of the inductive algorithm and the data representation must be automated. Since the output of the toolbox would go directly to the factory personnel, we would have to avoid producing uninteresting patterns. Since statistics are often misconstrued by non-statisticians, it is equally important that statistical information be clearly explained.

Here is a sketch of our current design for a semi-automated toolbox. Exploratory statistical tests and visualization techniques will be used to choose instances. Statistical techniques will be used to produce derived attributes from the primitive data, which can then be used by the induction algorithm. A set of complex data types such as sets and generalization hierarchies will be provided which will be automatically reformulated into attribute-value representations.

We will partially automate the process of setting induction parameters. Different parameters will favor rules with higher accuracies or greater statistical significance. A combination of heuristic rules (e.g., in Brute always use a low beam value if the number of attributes is very high) and experimentation (i.e., running the algorithm with a number of different parameter settings and returning the cumulative non-redundant best rules) should be used.

8 Conclusion

Induction has been credited with many successes over the past few years, but the difficulties encountered in applying these techniques to real-world domains have not been addressed. In this paper we discuss the various challenges to applying induction algorithms to real-world domains. We emphasize practical techniques for overcoming these challenges. We use a success story of applying induction algorithms to a Boeing factory domain as a case study. The Boeing application led us to seek concise predictive rules which cover small subsets of the data, instead of full-blown decision trees. We developed two new induction algorithms which were focused on finding nuggets.

For induction algorithms to have an impact on real-world problems, we need a methodology for handling these challenges. We believe it is time to analyze and to document the process of applying induction algorithms so that we can start automating it. We propose building a semi-automated toolbox to help non-specialists to apply induction effectively.

References

Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J., 1984. *Classification and Regression Trees*. Wadsworth.

Buntine, W. and Caruana, R., 1991. Introduction to IND and recursive partitioning. NASA Ames Research Center, Mail Stop 269-2 Moffet Field, CA 94035.

Buntine, W. and Stirling, D. 1991. Interactive induction. *Machine Intelligence*, 12:121–137.

Evans, B. and Fisher, D. Process delay analysis using decision tree induction. To appear in IEEE Expert.

Fayyad, U., Doyle, R., Weir, W. N., and Djorgovski, S., 1992. Applying machine learning classification techniques to automate sky object cataloguing. In *Proceedings of International Space Year Conference on Earth & Space Science Information Systems*.

Holte, R. C., 1993. Very simple classification rules perform well on most commonly used datasets. *Machine Learning*, 11(1):63–90.

McAllester, D. and Rosenblitt, D., 1991 Systematic nonlinear planning. In *Proceedings of AAAI-91*, pages 634–639.

Mingers, J., 1989. An empirical comparison of selection measures for decision-tree induction *Machine Learning*, 3(4):319–342.

Pagallo, G. and Haussler, D., 1990. Boolean feature discovery in empirical learning. *Machine Learning*, 5(1):71–100.

Quinlan, J R., 1986. Induction of decision trees. *Machine Learning*, 1(1):81–106.

Rivest, R. 1987. Learning decision trees. *Machine Learning*, 2:229–246.

Schlimmer, J. C., 1991. Database consistency via inductive learning. In Birnbaum, Lawrence A. and Collins, Gregg C., editors, *Proceedings of the Eighth International Machine Learning Workshop*, pages 640–644.

Schlimmer, J. C., 1991. Learning meta-knowledge for database checking. In *Proceedings of AAAI-91*, pages 335–340.

Schlimmer, J. C., 1993. Efficiently inducing determinations: A complete and systematic search algorithm that uses optimal pruning. In *Proceedings of the Tenth International Conference on Machine Learning*, Amherst, MA.

Smyth, P. and Goodman, R. M., 1991. Rule induction using information theory. In *Knowledge Discovery in Databases*, pages 159–176. MIT Press, Cambridge, MA.

Weiss, S. M., Galen, R. S., and Tadepalli, P. V., 1990. Maximizing the predictive value of production rules. *Artificial Intelligence*, 45:47–71.