AN ABSTRACT OF THE THESIS OF

Peter Drake for the degree of Master of Science in Computer Science presented on August 9, 1995. Title: Constructive Induction for Improved Learning of Boolean Functions.

Abstract approved: 

Prasad Tadepalli

Supervised learning programs, such as decision tree learners and neural networks, often must learn Boolean functions. The concept being learned may not easily be expressed in terms of the atomic features given. Constructive induction automatically produces higher level features (combinations of the atomic features), which can improve learning performance.

The FLIP algorithm, introduced in this thesis, uses an information gain metric to search for useful conjunctions of atomic features. Given these conjunctions, a decision tree learner is shown to produce trees which are both smaller and more accurate, when learning both random CNF functions and functions from game-playing domains. Furthermore, evidence is provided that FLIP constructs even better features when it has access to training sets for additional functions related to the function being learned.
Constructive Induction for Improved Learning of Boolean Functions

by

Peter Drake

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed August 9, 1995
Commencement June 1996
Master of Science thesis of Peter Drake presented on August 9, 1995

APPROVED:

Redacted for privacy

___________________________
Major Professor, representing Computer Science

Redacted for privacy

___________________________
Chair of Department of Computer Science

Redacted for privacy

___________________________
Dean of Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for privacy

___________________________
Peter Drake, Author

Redacted for privacy
DEDICATION

For my beloved Heather.

Well we know where we’re going.

But we don’t know where we’ve been.

And we know what we’re knowing,

But we can’t say what we’ve seen.

And we’re not little children,

And we know what we want.

And the future is certain,

Give us time to work it out.

— Talking Heads
ACKNOWLEDGMENT

I would like to thank my major advisor, Prasad Tadepalli, whose extensive knowledge of the literature proved invaluable, and my minor advisor, Tom Dietterich. Thanks also go to Bruce D’ambrosio and Goran Jovanovic, my other committee members; to all of my professors, teachers, and predecessors; and to my wife Heather, who now knows much more about decision trees than the average historian.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>Previous Work</td>
<td>3</td>
</tr>
<tr>
<td>Inductive Learning</td>
<td>3</td>
</tr>
<tr>
<td>Decision Trees</td>
<td>5</td>
</tr>
<tr>
<td>Entropy and Information Gain</td>
<td>7</td>
</tr>
<tr>
<td>Constructive Induction</td>
<td>8</td>
</tr>
<tr>
<td>FLIP</td>
<td>15</td>
</tr>
<tr>
<td>Single-Task Learning</td>
<td>18</td>
</tr>
<tr>
<td>Rationale and Methodology</td>
<td>18</td>
</tr>
<tr>
<td>Beam Width</td>
<td>19</td>
</tr>
<tr>
<td>Training Set Size</td>
<td>19</td>
</tr>
<tr>
<td>CNF Size</td>
<td>21</td>
</tr>
<tr>
<td>Chess Endgame</td>
<td>24</td>
</tr>
<tr>
<td>Multitask Learning</td>
<td>26</td>
</tr>
<tr>
<td>Related CNF Functions</td>
<td>26</td>
</tr>
<tr>
<td>Poker Hands</td>
<td>27</td>
</tr>
<tr>
<td>Conclusions and Future Work</td>
<td>29</td>
</tr>
<tr>
<td>Bibliography</td>
<td>31</td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>Figure 1: A decision tree.</td>
<td>6</td>
</tr>
<tr>
<td>Figure 2: The replication problem. (Left branches for true, right for false.) Notice that the entire left subtree is replicated on the right.</td>
<td>9</td>
</tr>
<tr>
<td>Figure 3: The replication problem solved.</td>
<td>11</td>
</tr>
<tr>
<td>Figure 4: Unpruned tree for the data in</td>
<td>12</td>
</tr>
<tr>
<td>Figure 5: Error vs. beam width</td>
<td>19</td>
</tr>
<tr>
<td>Figure 6: Error vs. training set size</td>
<td>20</td>
</tr>
<tr>
<td>Figure 7: Size vs. training set size</td>
<td>21</td>
</tr>
<tr>
<td>Figure 8: Error vs. literals per clause</td>
<td>22</td>
</tr>
<tr>
<td>Figure 9: Size vs. literals per clause</td>
<td>23</td>
</tr>
<tr>
<td>Figure 10: Error vs. clauses</td>
<td>23</td>
</tr>
<tr>
<td>Figure 11: Error vs. training set size for Chess endgame data.</td>
<td>25</td>
</tr>
<tr>
<td>Figure 12: Size vs. training set size for Chess endgame data.</td>
<td>25</td>
</tr>
<tr>
<td>Figure 13: Error vs. additional functions</td>
<td>27</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1: (Reproduced from (Quinlan, 1993).) A set of training examples for a supervised learner.</td>
<td>4</td>
</tr>
<tr>
<td>Table 2: Incomplete data for $(A \lor B \lor C) \land (D \lor E)$.</td>
<td>12</td>
</tr>
<tr>
<td>Table 3: Error rates for Poker experiment</td>
<td>28</td>
</tr>
</tbody>
</table>
CONSTRUCTIVE INDUCTION FOR IMPROVED LEARNING OF BOOLEAN FUNCTIONS

Introduction

One of the central problems in artificial intelligence is that of machine learning. A system can be said to learn if it acquires new knowledge, or modifies itself so that it can use its current knowledge more effectively (Dietterich & Shavlik, 1990). Learning can be useful for a number of reasons. It may be prohibitive to acquire and store sufficient knowledge beforehand, and a dynamic domain may render ‘hardwired’ knowledge obsolete.

Most work in machine learning is in the area of inductive learning, where the system is asked to derive concepts from a set of examples.

One difficulty in inductive learning is that the concept being learned is not always easily expressed in the language used to describe the examples. The concept of ‘check’ is not a simple one when expressed in terms of the Cartesian coordinates of Chess pieces, and one would not want to describe what a chair looks like in terms of pixel brightnesses. Nonetheless, machine learning programs are routinely asked to learn concepts from just such low-level descriptions, because high-level descriptions can be difficult or even impossible to acquire.

The process of automatically producing high-level descriptions is constructive induction (Michalski, 1983). This thesis introduces the FLIP algorithm, which uses an information gain metric to construct conjunctions of atomic, Boolean features. A
number of experiments show that the C4.5 decision tree learner benefits significantly from access to the high-level features constructed by FLIP.

The next section discusses inductive learning, the decision tree method, the information gain metric used both by decision tree learners and the FLIP algorithm, and constructive induction. The FLIP algorithm is explained in the following section. The next two sections present experimental results demonstrating improvements in both single- and multi-task learning. The final section summarizes our conclusions and areas for future exploration.
Previous Work

**Inductive Learning**

Inductive learning systems can be classified as *unsupervised*, *reinforcement*, or *supervised* learners. Unsupervised learners use statistical methods to find patterns (e.g., clusters) in a collection of data points. Reinforcement learners learn mappings from domain states to actions by observing ‘rewards’ and ‘punishments’ for various state-action pairs. Supervised learners are given a collection of data points labeled with values, and learn functions mapping data points to values. Much of the work (including the work in this paper) is more specifically classification or concept learning, where the training data are labeled with discrete classes.

As this thesis deals with supervised concept learning, a formalization of the problem is in order. A supervised concept learning system is given a set $T$ of training cases. Each case $i$ consists of a set of feature values $\{F^i_0, F^i_1, \ldots, F^i_N\}$ and a class label $C^i$. (Where it is obvious from context, we will omit the superscript $i$.) The features can be of just about any form (e.g. Boolean or real), but each case has values for the same set of features. The class label must be one of a finite set $C = \{C_0, C_1, \ldots, C_M\}$.

For example, consider the training set in Table 1, taken from (Quinlan, 1993). There are four features, namely Outlook, Temperature, Humidity, and Windy?. $C = \{\text{‘Play’}, \text{‘Don’t Play’}\}$.
<table>
<thead>
<tr>
<th>Outlook</th>
<th>Temperature (°F)</th>
<th>Humidity (%)</th>
<th>Windy?</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>sunny</td>
<td>75</td>
<td>70</td>
<td>true</td>
<td>Play</td>
</tr>
<tr>
<td>sunny</td>
<td>80</td>
<td>90</td>
<td>true</td>
<td>Don't Play</td>
</tr>
<tr>
<td>sunny</td>
<td>85</td>
<td>85</td>
<td>false</td>
<td>Don't Play</td>
</tr>
<tr>
<td>sunny</td>
<td>72</td>
<td>95</td>
<td>false</td>
<td>Don't Play</td>
</tr>
<tr>
<td>sunny</td>
<td>69</td>
<td>70</td>
<td>false</td>
<td>Play</td>
</tr>
<tr>
<td>overcast</td>
<td>72</td>
<td>90</td>
<td>true</td>
<td>Play</td>
</tr>
<tr>
<td>overcast</td>
<td>83</td>
<td>78</td>
<td>false</td>
<td>Play</td>
</tr>
<tr>
<td>overcast</td>
<td>64</td>
<td>65</td>
<td>true</td>
<td>Play</td>
</tr>
<tr>
<td>overcast</td>
<td>81</td>
<td>75</td>
<td>false</td>
<td>Play</td>
</tr>
<tr>
<td>rain</td>
<td>71</td>
<td>80</td>
<td>true</td>
<td>Don’t Play</td>
</tr>
<tr>
<td>rain</td>
<td>65</td>
<td>70</td>
<td>true</td>
<td>Don’t Play</td>
</tr>
<tr>
<td>rain</td>
<td>75</td>
<td>80</td>
<td>false</td>
<td>Play</td>
</tr>
<tr>
<td>rain</td>
<td>68</td>
<td>80</td>
<td>false</td>
<td>Play</td>
</tr>
<tr>
<td>rain</td>
<td>70</td>
<td>96</td>
<td>false</td>
<td>Play</td>
</tr>
</tbody>
</table>

Table 1: (Reproduced from (Quinlan, 1993).) A set of training examples for a supervised learner.

The learner’s task is to learn a mapping from feature vectors $F'$ to classes $C$. One hypothesis consistent with the data given above is: label the case as ‘Don’t Play’ if either the outlook is sunny and the humidity is over 75%, or the outlook is rain and it’s windy; otherwise label the case ‘Play’.

Criteria for judging supervised learning systems include:

- How well does the learned hypothesis predict the class of as-yet-unseen data points?
- How succinctly can the learned hypothesis be stated? (This is a concern for storage, for computing predictions, and for human understanding of learned hypotheses.)
- How efficient is the learner (in terms of both space and time)?
Quite a few methods for supervised learning are being studied. Among the more popular are neural networks, nearest neighbor methods, and decision trees. The experiments in this thesis deal only with decision trees.

**Decision Trees**

Decision tree algorithms use information-theoretic techniques to grow a tree of tests used to classify data points. The canonical works on these methods are (Breiman, et al., 1984) and (Quinlan, 1986). A more recent and very readable book is (Quinlan 1993), which traces the origin of decision trees “to the work of Hoveland and Hunt in the late 1950s, culminating in [(Hunt, et al., 1966)].”

A decision tree is either a leaf, which is labeled with one of the classes $C_i$, or a decision node, which is labeled with a test and has two or more children (which are themselves decision trees) corresponding to results of the test. If the tree is a leaf, any case to be classified is given the label at the leaf. Otherwise, the test at the root is applied to the case, and the case is classified according to the subtree corresponding to the result of the test.
Figure 1: A decision tree.

A decision tree corresponding to the example above is shown in Figure 1.

Suppose we have a new case, with outlook sunny, temperature 80, humidity 72, and windy true. The test at the root of the tree is on the outlook. Since the case has outlook sunny, we go down the left branch. The next test is on humidity. The case has humidity ≤ 75, so we again go down the left branch, arriving at a leaf. The case is labeled with the class of this leaf, namely 'Play'.

To grow a decision tree, a test must be chosen for the root. In the basic decision tree algorithm, the single feature offering the greatest information gain (explained below) is chosen for the test. For a Boolean feature, there will be branches for each of the two test outcomes 'true' and 'false'; finding branches for more complex attributes is explained in (Quinlan, 1993).

A decision tree is then grown on the subset of the data falling down each branch. This is repeated recursively, with branches ending in leaves when some stopping
criterion has been met. Common criteria include the lack of a good feature, a minimum number of data points, and all of the data being in the same class.

After the complete tree is grown, it is often pruned by replacing some of the subtrees with leaves. The rationale behind pruning is that it both makes the tree smaller and avoids overfitting, that is, including test nodes which derive from chance variations in the training data rather than aspects of the 'true' function being learned.

**Entropy and Information Gain**

(This subsection borrows heavily from (Quinlan, 1993).)

Information gain is defined in terms of entropy. Entropy is a measure of the expected amount of information conveyed by an as-yet-unseen message. (The message will be one of a known set.)

The amount of information conveyed by a message, in bits, is minus the base-two logarithm of the probability of that message. For example, if there are 8 equally probable messages, receiving any one of them conveys \(-\log_2(1/8) = 3\) bits of information. Less probable messages convey more information, and vice versa. The expected amount for any message is simply the sum over all possible messages, weighted by their probabilities.

In the context of supervised learning, the possible 'messages' are the classes in \(C\), and \(p_i(S)\) is the fraction of the cases in the training set \(S\) labeled with class \(C_i\). The expected information needed to classify a case in the training set (that is, the information conveyed by being simply told the class) is
The entropy of a set $S$ is given by:

$$
\text{entropy}(S) = -\sum_{i} p_i(S) \cdot \log_2 p_i(S)
$$

This measures the uncertainty or randomness in the set. A training set evenly split across the classes therefore has maximal entropy (1 if there are two classes), while one containing only examples of one class has entropy 0.

If the data can be divided into subsets by some useful test (e.g., Outlook from the example above), each subset will have less entropy than the whole set. We don’t know ahead of time into which of the subsets $S_j$ an unseen case will fall, but we can take another weighted sum:

$$
\text{entropy}_x(S) = \sum_{j} \frac{|S_j|}{|S|} \cdot \text{entropy}(S_j)
$$

where $X$ is the test (usually one of the features $F_i$) and the values of $j$ correspond to the possible results of the test.

The information gain of the test $X$ is simply

$$
gain(X) = \text{entropy}(S) - \text{entropy}_x(S)
$$

In growing a decision tree, we choose the feature with the greatest information gain at each step.

**Constructive Induction**

One drawback of decision trees is that trees describing certain functions contain a great deal of replication. Figure 2 shows the smallest decision tree for the Boolean function $(A \lor B \lor C) \land (D \lor E)$, where each letter is a Boolean feature. This *replication*
**problem** (Pagallo and Haussler, 1990) can lead to very large — and, with incomplete or noisy data, inaccurate — decision trees.

![Decision Tree Diagram]

**Figure 2:** The replication problem. (Left branches for true, right for false.) Notice that the entire left subtree is replicated on the right.

The replication problem is just one example of *feature interaction* or blurring (Rendell & Ragavan, 1993). (Technically, blurring also covers difficult one-feature concepts such as “n is odd”, but this is not relevant to the Boolean features used in this thesis.) When a feature provides useful information only in the context of other features, the information gain metric may not be able to find good features. The worst case is
parity, where the class of an example can be determined if all $n$ atomic features are known, but not if any $n-1$ of them are known.

Greedy methods such as decision trees fare particularly poorly on problems with high atomic feature interaction. Consider the 3-parity function, which is true if an odd number of the features $A$, $B$, and $C$ are true. With a complete data set, half of the cases fall into each class. If we split on any one feature, each subset also has half of its cases in each class, so no information is gained. Indeed, these relevant features would look no better than a spurious feature $D$ which is true for half of the cases.

One approach to dealing with feature interaction is constructive induction (Michalski, 1983). Constructive induction refers to the creation of new features which are functions of the atomic features. The hidden units of a backpropagation neural network may be said to perform constructive induction, creating compound features which greatly simplify the functions the output units are required to learn. On the 3-parity problem, a decision-tree learner would fare much better if it had access to the conjunctions $A \land B \land C$, $A \land \neg B \land \neg C$, $\neg A \land B \land \neg C$, and $\neg A \land \neg B \land C$.

In addition to guiding the learner to relevant combinations of features, constructive induction can provide a vocabulary of high-level features leading to more concise concept descriptions. For example, it can solve the replication problem shown in Figure 2. As Figure 3 shows, compound features produce a much smaller tree. A smaller tree can be stored more cheaply, computed more quickly, and understood by humans more easily.
Internal representations produced by constructive induction can also improve generalization, giving better performance in the case of noisy or incomplete data. Again, consider the function \((A \lor B \lor C) \land (D \lor E)\), but with the incomplete data shown in Table 2. Given this data, C4.5 (invoked with a minimum leaf size of 1) produces the inaccurate unpruned tree shown in Figure 4, and then prunes it to simply a test on \(B\)! Given the appropriate compound features, it produces the correct tree.
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

Table 2: Incomplete data for $(A \lor B \lor C) \land (D \lor E)$.

Figure 4: Unpruned tree for the data in Table 2.
Researchers are working on constructive induction specifically for decision trees. Two recent systems are FRINGE (Pagallo and Haussler, 1990) and LFC (Ragavan and Rendell, 1993).

FRINGE alternates tree-growing and feature construction. Features are constructed from the last nodes on the path from the root to each leaf in the most recently grown tree. For example, the rightmost leaf in Figure 2 would give rise to the feature \(~D\wedge E\). Features thus produced are used in successively grown trees, and may in turn be used to construct even larger features. The algorithm halts when no new features are found, or a limit on the total number of features is reached.

LFC also intersperses tree-growing and feature construction, performing a beam search between each recursive call to the tree-grower. Using clever search techniques which emphasize reducing the entropy of the subset of training data which satisfies the feature, LFC is able to evaluate many fewer features.

Another intriguing possibility is that constructive induction may be used to discover high-level features which are common to a set of related problems. For example, a vision program might fare better if it learns to recognize several objects, rather than just one or two, by developing an internal representation which ignores rotation and absolute brightness. A medical diagnosis program, learning to diagnose several diseases from a set of symptoms, might learn some intermediate conditions which are relevant to several diseases.

Work on multitask learning seems to focus on neural networks, where it is natural to train hidden units on a set of problems and output units on individual problems. (Caruana, 1993), (Thrun and Mitchell, 1994), and (Baxter, 1995) demonstrate that
learning related tasks can help by biasing the learner toward the proper part of the hypothesis space, or (in Baxter’s terminology) learning an internal representation which is useful for the whole set of tasks.

In summary, constructive induction (in the context of decision trees) can help supervised learning systems produce smaller, more accurate hypotheses, and can find compound features which are useful in learning several related functions.
FLIP

Every machine learning paper needs a clever algorithm, so we’ve dubbed our system the Feature Learning Inductive Preprocessor. As the name suggests, it finds compound features from a data set, which can then be fed to a standard supervised learner.

FLIP uses a beam search to find conjunctions of the atomic features and their negations. (AND and NOT are complete, so there is no need to learn disjunctions.) The ‘goodness’ measure is the information gain metric described above; since the entropy of the whole data set does not change, it is easier to simply calculate the $\text{entropy}_x$ of the feature in question.

In pseudocode, the algorithm is:

Initialize $\text{beam}$ to the set of literals (the atomic features and their negations), all marked ‘live’.

Until none of the features in $\text{beam}$ are ‘live’.

Mark every feature in the beam as ‘dead’.

Find each new feature which can be made by conjoining a literal to an extant feature. Mark the new features ‘live’.

Throw out all but the width features having the lowest $\text{entropy}_x$.

Return all features in $\text{beam}$ comprising more than one literal.
For an example, consider complete data for the function $(A \land B \land C) \lor (D \land E \land F)$. In the first pass through the loop, every two-literal conjunction — $A \land B$, $A \land \neg B$, et cetera — is constructed. If width is set to 10, the 10 best features survive the first iteration. These features are:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Entropy</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \land F$</td>
<td>0.65</td>
</tr>
<tr>
<td>$D \land F$</td>
<td>0.65</td>
</tr>
<tr>
<td>$D \land E$</td>
<td>0.65</td>
</tr>
<tr>
<td>$B \land C$</td>
<td>0.65</td>
</tr>
<tr>
<td>$A \land C$</td>
<td>0.65</td>
</tr>
<tr>
<td>$A \land B$</td>
<td>0.65</td>
</tr>
<tr>
<td>$\neg C \land \neg F$</td>
<td>0.67</td>
</tr>
<tr>
<td>$\neg C \land E$</td>
<td>0.67</td>
</tr>
<tr>
<td>$\neg C \land D$</td>
<td>0.67</td>
</tr>
<tr>
<td>$\neg B \land \neg F$</td>
<td>0.67</td>
</tr>
</tbody>
</table>

The top six are the two-literal subsets of the terms of the function, and the next four are some of the clauses of the CNF negation of the function.

The next iteration produces the two terms of the function (entropy 0.78), the six two-literal subsets, and two of the CNF negation clauses. Another iteration finds some spurious four-literal features, and the last iteration finds nothing new, so the algorithm
terminates. At the front of the list of features returned are the “ideal” features $A\wedge B\wedge C$ and $D\wedge E\wedge F$.

The time complexity of the algorithm is bounded by the number of features considered times the number of iterations.

Let $m$ be the number of atomic features, and assume that the beam width is less than or equal to $2m$. Each iteration takes each of up to $2m$ features (the atomic features and their negations, in the first iteration), and combines each one with each other literal. Since there are $2m$ literals, no more than $4m^2$ features are generated in any iteration. We must consider at most all $n$ data points for each feature, so each iteration takes $O(m^2n)$ time.

Since the maximum feature length must increase by one each iteration, and no feature can contain more than $m$ literals, there can be at most $m - 1$ iterations. Thus, the time complexity of the algorithm is $O(m^3n)$.

Since the features are considered serially, the space taken by the algorithm is just the space needed to store the atomic features for all of the examples in the training set, which is $O(mn)$.

The multitask version of the algorithm keeps the features having the highest average information gain over all of the tasks, rather than the lowest entropy. If all of the problems are the same size, this effectively multiplies the time and space complexity by the number of problems.
Single-Task Learning

Rationale and Methodology

In this section, we show that FLIP provides significant improvement in both the accuracy and parsimony of decision trees grown for a single task. In all of these experiments, except where otherwise specified, the following methodology was used:

- C4.5 is used to grow all trees, and results are shown for the pruned trees.
- For ‘atoms only’ results, C4.5 was given only the atomic features. For ‘random conjunction’ results, C4.5 was given the atomic features and a number of random, two-literal conjunctions equal to the beam width. For ‘FLIP’ results, C4.5 was given the atomic features and the features constructed by FLIP.
- Training and testing sets were disjoint, with 100 data points each.
- Beam width was set to 20.
- Results reported are means over 20 trials.
- Tree size is as reported by C4.5. This is the number of nodes, including leaves.
- ‘Significant’ denotes the P < 0.05 level on a 2-tailed, paired t-test.
- There are 30 atomic features (some of which may be irrelevant).
- The learning tasks were random CNF functions of four, three-literal clauses. They were not guaranteed to be satisfiable or non-redundant (i.e., may contain clauses such as ‘A\&B\&B’).
**Beam Width**

In this experiment, the beam width parameter was varied. The errors for FLIP are shown in Figure 5. C4.5 using atoms only averages 27.85% test error, and the random conjunctions produced an average of 33.65% error. FLIP showed a significant improvement in accuracy at each beam width tried.

![Figure 5: Error vs. beam width](image)

**Training Set Size**

In the next experiment, we varied the training set size. Figure 6 shows the impressive results for FLIP: a significant improvement at every level. By the time the training set contains 250 cases, the trees grown with FLIP features average less than 1% training error.
Figure 6: Error vs. training set size

Figure 7 shows the tree sizes for the same experiments. The replication problem becomes apparent, as increasingly accurate atoms-only trees become enormous. The number of nodes stays relatively flat for FLIP, but it should be noted that the number of literals per node is not taken into account. Tentative experiments suggest that the total number of literals in the tree (the sum of the number of literals in each node) is about the same with or without FLIP; while FLIP trees may not be faster to compute, they are likely to be easier for humans to understand, especially if meaningful names for the constructed features can be devised.
Another way to vary the difficulty of learning CNF functions is to vary the number of literals per clause. We experimented with values from 1 to 6 for this parameter, holding the number of clauses at 4.

Figure 8 shows the effect on accuracy. FLIP provides a significant improvement at 1, 3, and 4 literals per clause, but never does significantly worse. It is interesting that FLIP provides the greatest improvement on those problems where raw C4.5 fares worst.

We suspect that the reason for the difficulty of learning 4-clause CNF functions around 3 literals per clause is that this is the point where the proportion of true cases in a complete data set is closest to 50%. With very few literals per clause, a CNF function is unlikely to be satisfied, so a learning system does not suffer much for guessing ‘always false’. With many literals, ‘always true’ becomes a good guess. Indeed, at 5 and 6 literals per clause, the error rates are commensurate with what would result from such a guess.
FLIP can help down at one literal per clause because it may be able to find a single conjunction for the entire function; at the high end, 100 data points may be insufficient for FLIP to learn long conjunctions corresponding to the complements of long clauses.

Figure 8: Error vs. literals per clause

As shown in Figure 9, the effect on tree size is similar. FLIP offers the most improvement on the difficult problems, and very little by the time six literals per clause is reached. For all three curves, the decision trees are almost trivial at the outside ends.
To further test the hypothesis about the 50% level, and to reassure ourselves of FLIP's generality, we ran another experiment. Holding the literals per clause at 5, we varied the number of clauses. 1000 training examples were given in each trial.

The results are shown in Figure 10. The difficulty levels off at around 20 clauses, where 53% of the assignments satisfy the function. Beyond that point, the difficulty of...
learning these very complex functions seems to compensate for the slowly decreasing satisfaction percentage.

FLIP provides significant improvement at every point up to 20 clauses, inclusive.

Chess Endgame

Turning to non-synthetic data, we experimented with a set of Chess endgame data (Quinlan, 1987). The features are Boolean, high-level features (Quinlan, 1983), and the data points are classified by whether white will lose in two plies. This experiment differed from those above in that:

- Results are means over 50 trials.
- There are 24 atomic features.
- Test and training sets were disjointly drawn from a fixed body of 928 examples.

The outcome is shown in Figure 11. FLIP provides a significant improvement at every level except 70 training examples (at which point the P-value is only slightly higher than 0.050). It is encouraging that FLIP helps even on top of hand-crafted, high-level features.
Figure 11: Error vs. training set size for Chess endgame data.

Figure 12 shows the tree sizes for the same experiment.

Figure 12: Size vs. training set size for Chess endgame data.
Multitask Learning

Caruana (1993) points out that learning more than one function from the same domain at the same time may be easier than learning the functions separately. "In effect, the information provided by the training signal for each task serves as a domain-specific inductive bias for the other tasks."

With this in mind, we ran experiments to see if FLIP could build better features when given training sets for several related functions. When constructing features for multiple concepts, FLIP is given a training set for each function in the group; in choosing the 'best' features to retain, FLIP chooses the features having the highest average information gain over all of the training sets. C4.5 is then given these features for the training set for the single function being learned.

Related CNF Functions

In the first experiment, we used a set of related CNF functions. Specifically, the functions were each the conjunction of four, three-literal clauses drawn from a pool of 10 clauses. The methodology was otherwise identical to that used in the previous section.

The results are shown in Figure 13. (For comparison, atoms only and random conjunctions averaged 22.85% error.) FLIP's improvement with no additional functions was significant. It never performed worse than that, but none of the improvements are significant (due to large variances), and the correlation is only -0.36. Judging from this experiment, related functions may help, but the evidence is not conclusive.
Further experiments are underway involving different numbers of additional functions and more trials. Unfortunately, results are not available at the time of this writing.

**Poker Hands**

A more successful experiment involved learning Poker hands. Here, each hand was encoded as a series of 20 bits, four bits for the rank of each card. (Suits were not accounted for). The training and testing sets were disjoint, and no data set contained duplicate bitstrings. The same cards, arranged in a different order, would produce a different bitstring, but FLIP has no way to exploit such a permutation. Each data set contained 50 positive and 50 negative examples. The methodology was otherwise like that described in the previous section.
In defining hands, superior hands were not taken into account. For example, every instance of ‘three of a kind’ was also an instance of ‘one pair’.

The error rates are shown in Table 3. The ‘FLIP (single)’ column used features grown by FLIP only on the training set for the concept being learned. The ‘FLIP (all)’ column used features grown for the whole group of concepts. Results marked with asterisks (*) indicate a significant improvement over atoms only. Clearly, FLIP was aided by the data from the other concepts in most cases.

<table>
<thead>
<tr>
<th>Hand</th>
<th>Atoms only</th>
<th>Random conj</th>
<th>FLIP (single)</th>
<th>FLIP (all)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pair</td>
<td>48.05</td>
<td>47.00</td>
<td>48.25</td>
<td>45.85</td>
</tr>
<tr>
<td>Two Pair</td>
<td>49.20</td>
<td>47.93</td>
<td>47.05</td>
<td>46.30*</td>
</tr>
<tr>
<td>Three of a Kind</td>
<td>45.10</td>
<td>45.80</td>
<td>44.80</td>
<td>42.20*</td>
</tr>
<tr>
<td>Full House</td>
<td>39.10</td>
<td>39.95</td>
<td>39.30</td>
<td>34.15*</td>
</tr>
<tr>
<td>Four of a Kind</td>
<td>33.10</td>
<td>33.80</td>
<td>29.25*</td>
<td>30.40</td>
</tr>
</tbody>
</table>

Table 3: Error rates for Poker experiment

It is somewhat surprising that FLIP could do any good at all; it is, after all, unable to construct features such as ‘the first four bits are the same as the second four’.

Examination of the data showed that FLIP often learned conjunctions like that equivalent to ‘all of the cards have their low-order bit on’, i.e., ‘all of the cards are odd-numbered’.

This is indeed a useful high-level feature, as any hand possessing this feature is more likely to be an instance of any of the concepts in question.

While FLIP did help here, the error rates are still quite high. Further experiments, varying the training set size, beam width, and number of trials, are underway.
Conclusions and Future Work

We have shown that features generated by FLIP significantly improve the performance of C4.5. The trees grown are both smaller and more accurate. FLIP offers the most improvement for those problems where C4.5 fares most poorly, and never significantly degrades performance.

Furthermore, in at least one domain, FLIP was able to improve its performance by exploiting domain-specific bias in the form of training data for other concepts from the same domain.

One severe limitation of the current FLIP system is that it can only handle Boolean features. Information theoretic techniques can be used to find good decision-tree splits for other types of features (Quinlan, 1993), so it may be possible to construct compound features in a similar manner.

Another problem, noticed just before the completion of this thesis, is that the best features (those which make it into the beam) are often very similar. When FLIP returns a set of very similar features, those not chosen as the root of the decision tree become significantly less useful in its wake. It might be profitable to explore a scheme giving credit for uniqueness, increasing the diversity of the features returned after the last iteration.

Finally, since FLIP is a preprocessor, there is no real reason to use the features it constructs only in decision trees. Other inductive learning methods, such as neural networks or genetic algorithms, may be able to take advantage of FLIP-constructed
features. It would also be interesting to see if such features could help provide more useful 'nearness' metrics for nearest-neighbor methods.
Bibliography


