Learning to remove Internet advertisements

Nicholas Kushmerick
Department of Computer Science, University College Dublin, Dublin 4, Ireland
nick@ucd.ie

Abstract

AdEater is a fully implemented browsing assistant that automatically removes advertisement images from Internet pages. Unlike related systems that rely on hand-crafted rules, AdEater takes an inductive learning approach, automatically generating rules from training examples. Our experiments demonstrate that our approach is practical: the off-line training phase takes less than six minutes; on-line classification takes about 70 msec; and classification accuracy exceeds 97% given a modest set of training data.

1 Introduction

Many Internet sites draw income from third-party advertisements, usually in the form of images sprinkled throughout the site's pages. If judged to be interesting or relevant, users can click on these so-called "banner advertisements", jumping to the advertiser's own site.

Some users prefer not to view such advertisements. Images tend to dominate a page's total download time, so users connecting through slow links find that advertisements substantially impede their browsing. Other users dislike paying for services indirectly through advertisers, preferring direct payment for services rendered. Finally, some users disagree with the very notion of advertising on the public Internet.

AdEater is a fully-implemented browsing assistant that automatically removes banner advertisements from Internet pages. Advertisements are removed *before* the corresponding images are downloaded, so pages download faster. Unlike related systems (see discussion in Sec. 4) that require hand-crafted rules, AdEater classifies advertisements with rules that are automatically generated by an inductive learning algorithm.

Fig. 1 shows AdEater in action. The system has been in continuous use by a small user community since July 1997; see 'www.cs.ucd.ie/staff/nick/research/ae' for instructions on running AdEater.

We proceed as follows. We first describe AdEater's architecture and implementation (Sec. 2). Our main contribution—the formalization of the task of learning advertisement-removal rules—is presented in Sec. 2.1. We go on to describe two experiments designed to evaluate our system (Sec. 3). First, we demonstrate that our approach is feasible: both the off-line learning and on-line classification modules are reasonably fast, and even modest amounts of training data yield high accuracy (Sec. 3.1). Second, we systematically explore the space of possible encodings for this learning task (Sec. 3.2). Finally, we discuss related work, future plans, and summarize our contributions (Sec. 4).

2 The AdEater system

Fig. 2 shows the architecture of the AdEater system. AdEater comprises three modules. First, during an preliminary off-line phase, a collection of training examples are gathered. Second, during an off-line training phase, an inductive learning algorithm processes these examples to generate a set of rules for discriminating advertisements from non-advertisements. The third module scans pages fetched by the user, removing images classified as advertisements by the learned rules.

In the remainder of this section, we first describe the central research issue: how to encode candidate advertisements in a way that is suitable as input to the inductive learning algorithm. We then describe each module in turn.

3rd Int. Conf. on Autonomous Agents, 1999.

Figure 1: Two Internet sites, before and after processing by AdEater. Advertisements are replaced by innocuous transparent images that simply say "ad". As desired, non-advertising images such as the Metacrawler logo and the photograph of Gerry Adams are left intact. Note that AdEater makes a small mistake on the Irish Times page: two navigational images at the top of the page near the date are incorrectly classified as advertisements.

2.1 Encoding instances

We treat the generation of advertisement-detection rules as an inductive learning task; see [10] for an introduction. An HTML page—both pages from which training examples are generated, and pages from which ads are to be removed during browsing—is processed to extract zero or more *instances*. Each instance corresponds to a candidate advertisement in the HTML page. Given a set of *training instances* that are preclassified as being an advertisement (AD) or not ($\overline{\text{AD}}$), the goal is to learn a *classifier* that maps instances to either AD or $\overline{\text{AD}}$.

The central research issue, therefore, is to choose an appropriate instance encoding. To enable rapid on-line removal, the encoding must be derivable directly from the raw HTML—if we were to encode image features such as color, then AdEater would have to download images before they could be removed, thereby defeating one of AdEater's purposes, speeding download time.

AdEater encodes instances using a fixed-width feature vector. Fig. 3 shows (a) an example HTML file, and (b) the 52-feature encoding of its three candidate advertisements. Specifically, the set of feature vectors are extracted from an Internet page with URL U_{base} according to the following procedure.

1. Each image enclosed in an <A> tag is a candidate advertisement; non-anchor images are rarely advertisements, and are therefore ignored. Let U_{dest} be the URL to which the anchor points, and let

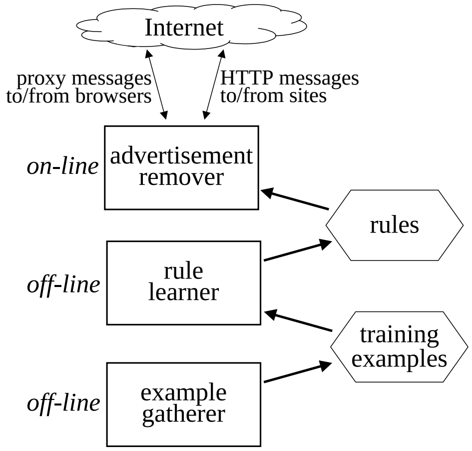


Figure 2: Architecture of the AdEater systen

U_{img} be the image's URL.

2. Three numeric features capture geometric i mation about the image: **height**, **width**, anc **pect ratio** (ratio of width to height). These tures are drawn directly from the HTML file. the image. Therefore, these features migh missing (indicated by "?") if the correspon `<IMG>` tag does indicate the height or width. example, no geometric features can be extra for instance C.

3. A single binary feature **local?** indicates whe U_{dest}'s and U_{img}'s servers are in the same I net domain. For example, if U_{dest} = a.host.cc page.html, then **local?** is 1 for U_{img} = b.h com/image.jpg, but 0 for U_{img} = elsewhere.c picture.gif.

4. An instance's **caption** is the words occurrir the enclosing `<A>` tag, ignoring punctuation case. A set of binary features encode each caµ word, each two-word phrase, and so on, thro K-word phrases. Caption features are then discarded if the phrase occurs fewer than at M times in the training set. For example, the caption feature "funded+by" is 1 for instances whose caption contains this two-word phrase (instance C only, in the example). Note that the specific caption features generated depend on the particular training instances; feature vectors have a fixed width respect to a given set of training instances.

5. An instance's **alt** text is the set of "alternate" words in the `<IMG>` tag. As with captions, the encoding contains one boolean feature for phrase of length each 1, 2, ..., K that occurs at least M times.

http://www.provider.com/index.html

```
A{  ...
    <A href="http://www.corp.com/sales.html">
      Our sponsor: <IMG src="http://www.corp.com/ads/thead.gif"
                    alt="click here" height="40" width="200"></A>
                    ...
B{  <A href="contact.html">
      Contact us: <IMG src="/images/contact.gif"
                   alt="contact info" height="50" width="40"></A>
                    ...
C{  <A href="http://www.mega.com/marketing.html">
      Funded by: <IMG src="http://www.mega.com/adverts/adimg.jpg"
                  alt="free stuff"></A>
                    ...
```

A	B	C	Feature	
40	50	?	**height**	
200	40	?	**width**	
5.0	0.8	?	**aspect ratio**	
0	0	1	**local?**	
1	0	0	"our"	**caption** features
1	0	0	"sponsor"	
1	0	0	"our+sponsor"	
0	1	0	"contact"	
0	1	0	"us"	
0	1	0	"contact+us"	
0	0	1	"funded"	
0	0	1	"by"	
0	0	1	"funded+by"	
1	0	0	"free"	**alt** features
1	0	0	"stuff"	
1	0	0	"free+stuff"	
0	1	0	"contact"	
0	1	0	"info"	
0	1	0	"contact+info"	
0	0	1	"click"	
0	0	1	"here"	
0	0	1	"click+here"	
1	1	1	"www.provider.com"	U_{base} features
1	1	1	"index"	
1	1	1	"index+html"	
1	0	0	"www.corp.com"	U_{target} features
1	0	0	"sales"	
1	0	0	"sales+html"	
0	1	0	"contact"	
0	1	0	"contact+html"	
0	0	1	"www.mega.com"	
0	0	1	"marketing"	
0	0	1	"marketing+html"	
1	0	0	"www.corp.com"	U_{img} features
1	0	0	"ads"	
1	0	0	"ads+thead"	
1	0	0	"thead"	
1	0	0	"thead+gif"	
0	1	0	"images+contact"	
0	1	0	"images"	
0	1	0	"contact"	
0	1	0	"contact+gif"	
0	0	1	"www.mega.com"	
0	0	1	"adverts"	
0	0	1	"adimg"	
0	0	1	"adverts+adimg"	
0	0	1	"adimg+jpg"	
AD	$\overline{\textbf{AD}}$	**AD**	**Classification**	

Figure 3: An example Internet page, and the encoding of its three instances.

6. Additional sets of features are provided by the **base URL** U_{base}, the **destination URL** U_{dest}, and the **image URL** U_{img}. For each of these URLs, one binary feature corresponds to the server name. Then, punctuation and case are discarded in the rest of the URL, and (like caption and alt text), a set of binary features encode phrase of length 1, 2, ..., K that occurs at least M times in the training set. One-word phrases are ignored if they are members of a stop list containing low-information terms such as "http", "www", "jpg", "html", *etc.*

Note that the above procedure generates a family of encodings, one for each value of K (maximum phrase length) and M (minimum phrase count). In the current implementation, $K = 2$ and $M = 10$. For the training data gathered as described in Sec. 2.2, the encoding consisted of 1558 features: **height**, **width**, **aspect ra-**

tio, **local?**, 19 **caption** features, 111 **alt** features, 495 **base URL** features, 472 **destination URL** features, and 457 **image URL** features. In Sec. 3.2, we discuss varying K and M.

2.2 Gathering examples

The previous section described ADEATER's encoding of instances. We now describe the "example gatherer" module in Fig. 2, which generates a collection of such instances. Since both negative and positive examples of advertisements are important for effective learning, we needed to generate instances that would be classified as both AD and $\overline{\text{AD}}$.

AD's were generated using the ADGRABBER browsing assistant. ADGRABBER identifies candidate advertisements. It leaves the HTML visually intact, but the user can point out advertisements to ADGRABBER using a simple mouse gesture. The vector encoding of these advertisements are then stored for later use by the learning module. This process was used to generate 364 AD's. While these instances are certainly not chosen independently (an assumption often made in the computational learning literature), the deviations are apparently small, since the system works well in practice.

$\overline{\text{AD}}$'s were generated using a custom-built Internet spider that extracted images from randomly-generated URLs. We chose random URLs so that the instance space would be sampled as fairly as feasible. These random images were then manually classified as AD/$\overline{\text{AD}}$, yielding 2820 $\overline{\text{AD}}$'s and 95 additional AD's.

As mentioned earlier, deficiencies in the raw HTML mean that the features can not always be generated. Of the 3279 examples, 28% contained one or more missing features. Moreover, it is possible that some examples were misclassified. We have not attempted to verify the classifications, but anecdotal evidence suggests that the classifications are certainly imperfect.

2.3 Learning rules

Once the training examples have been generated, an inductive learning algorithm must process the rules, resulting in a classifier that maps new instances to AD/$\overline{\text{AD}}$. The learning algorithm should have several properties.

- The learned classifier must execute quickly, since it is invoked on-line by ADEATER to remove advertisements. (In contrast, since learning occurs off-line, relatively long learning times are acceptable.)

- The learning algorithm must not be overly sensitive to missing features or classification noise, because our task exhibits these properties.

- We encode instances with thousands of features, and it seems likely that many will be irrelevant. Therefore, the learning algorithm must scale well with the number of features, and be insensitive to irrelevant features.

- The specific URLs, caption phrases, *etc.* used in advertisements may well evolve over time. For example, an advertiser might change the URLs where images are stored. Therefore, the learned classifier will eventually be obsolete, and old training examples must be retired. One strategy is to simply relearn the classifier from scratch. But ideally, the learning algorithm would be *incremental* [9, 14], so that the classifier can be rapidly relearned given a set of updates to the training set.

Given these desiderata, we selected the C4.5rules learning algorithm [11]; Ripper [3] would probably be suitable as well. Both algorithms exhibit the desired properties except the last. Nearest-neighbor and other lazy algorithms are indeed incremental, but classification is very slow, and accuracy is sensitive to irrelevant features. Since C4.5rules learns fast enough in practice, the benefit does not outweigh the cost.

C4.5rules learns a set of rules, each a conjunction of tests together with a predicted classification if the tests are satisfied. For numeric features, tests are of the form "$f_i \leq \tau$" or "$f_i > \tau$", where τ is a constant real number. For binary features, tests are of the form "f_i" or "$\bar{f}_i$". For our application, C4.5rules learned a set of 25 rules. Two representative examples are as follows:

- If **aspect ratio** > 4.5833, **alt** doesn't contain "to" but does contain "click+here", and U_{dest} doesn't contain "http+www", then instance is an AD.

- If U_{base} does not contain "messier", and U_{dest} contains the "redirect+cgi", then instance is an AD.

Note that these are actual rules learned by C4.5rules: the rules have only been reformatted to make them easier to read, and the learning algorithm, not a person, identifies relevant phrases such as "click+here".

2.4 Removing advertisements

The module of ADEATER's to which users are exposed is the browsing assistant that removes advertisements from Internet pages as they are fetched. Candidate advertisements are identified in fetched pages, as with the example generator. The learned rules are then consulted to classify each example as AD/$\overline{\text{AD}}$. Advertisements are then removed from the Internet page by replacing U_{img} (the image's URL) with the URL of an inconspicuous low-bandwidth image.

The advertisement-removal module is implemented as a proxy server. Browser requests are passed to the AdEatersystem, which forwards the requested URL to the destination, and replaces the U_{img}'s in the returned HTML files as appropriate.

3 Evaluation

AdEater has been fully implemented. Anecdotal user feedback suggests that while the system occasionally makes mistakes, it is reasonably effective at removing advertisements. AdEater's off-line training phase using C4.5rules takes 5.8 CPU minutes. AdEater then requires about 70 msec to remove each image during the on-line classification phase (excluding network time for downloading the original HTML text). Note that this figure is much less that the time to download a typical image.

We have also conducted a series of more objective experiments, using the standard machine learning "cross validation" methodology. We first randomly partitioned the gathered instances into a *training* set containing 90% of the instances and a *test* set containing the remainder. We then invoked C4.5rules on the training set, and measured the performance of the rules on the test set. We cross validated our results in this way ten times.

Averaging across the ten trials, we found that the learned rules have an accuracy of 97.1%. To further understand the limitations of our approach, we have also measured the system's learning curve. A second experiment was designed to validate the particular features in our encoding.

3.1 Learning curves

To calculate a learning curve for our system, we gave the learning algorithm 10%, 20%, ..., 90% of the training data, and then calculated 10-fold cross-validated accuracy on the remainder. Fig. 4(a) shows the results, along with 95% confidence intervals after ten repetitions of this process. The observed accuracy asymptotically approaches the 97.1% figure reported earlier, and exceeds 93% with just 10% of the training data.

In one important respect, this method for calculating the learning curve does not reflect the true nature of the task. Recall that the specific set of features depends on the training set. For example, the **caption** phrase "click+here" is assigned a feature only if this phrase occurs at least M times in the training data. In order to replicate the learning task more faithfully, we re-ran the learning-curve experiment so that the feature set was re-calculated for each train/test split; see Fig. 4(b).

We conclude that our approach is feasible, because relatively few training examples are needed to achieve high accuracy.

3.2 Alternative encodings

In Sec. 2.1, we described the features used to encode candidate advertisements. A natural question is whether better encodings exist. As an initial investigation of this question, we have systematically explored nine encodings in this (infinite) space.

Recall that our encoding technique has two parameters: K, the maximum phrase length; and M, the minimum number of times a phrase must occur to be assigned a feature. In the standard encoding, $K = 2$ and $M = 10$. Other encodings are obtained by varying K and M. Holding M constant, the just-words encoding uses $K = 1$ (just one-word phrases are considered), while long-phrases uses $K = 4$. Holding K constant, the most-phrases encoding uses $M = 2$ (any phrases occurring more than once is considered), and freq-phrases uses $M = 100$. Setting $M = \infty$ results in no-phrases: phrase features are eliminated entirely. The last three encodings involve not using a stop list (no-stoplist), and ignoring the **local?** and **aspect ratio** features (no-local and no-aratio, respectively).

Fig. 5 lists the nine encodings. Recall that our encoding process generates different features depending on the training instances. The figure also shows the number of features generated for our 3279 training examples, as well as the 10-fold cross-validated accuracy.

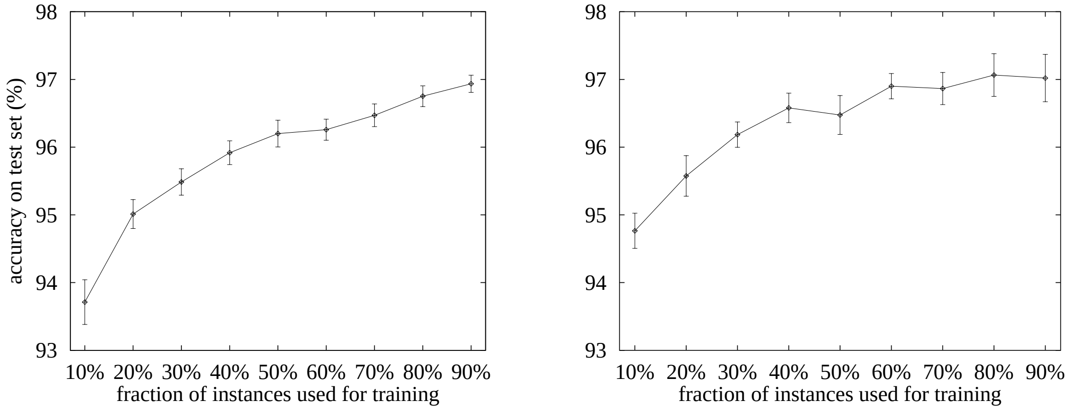
To compare the encodings, we quantified their "relative efficiency". The learning algorithm consumes resources (space and time) that depend directly on the number of features in the encoding. One encoding is preferable to another if an increase in consumed resources is compensated by improved accuracy. Thus a natural way to measure an encoding's inherent *efficiency* of an encoding e is to calculate the ratio of accuracy to number of features:

$$E_e = \frac{\text{accuracy when using encoding } e}{\text{number of features for encoding } e}$$

The *efficiency gain* of one encoding e over a second e' is the ratio $E_e/E_{e'}$; e is preferable to e' if $E_e/E_{e'} > 1$.

The last two columns in Fig. 5 show, for each encoding e, the efficiency E_e, and e's efficiency gain over over standard ($E_e/E_{\mathsf{standard}}$). Gains below one indicate encodings that are less efficient than standard.

We conclude that if users demand high accuracy ($> 97\%$), then standard is the best encoding. However, if user's were to tolerate AdEater making more mistakes, then the some of the other encodings are substantially more efficient. Remarkably, the "minimalist" no-phrases encoding achieves an accuracy of 93.5%. We leave further exploration of possible encodings to future work.



(a) (b)

Figure 4: Learning curves: (a) simple methodology, and (b) realistic methodology.

encoding e	K (max. phrase length)	M (min. phrase count)	use stop-list?	use '**local?**'?	use '**aspect ratio**'?	number of features	accuracy (%)	efficiency (E_e)	gain ($E_e/E_{\mathsf{standard}}$)
standard	2	10	yes	yes	yes	1558	97.2	0.06	1
just-words	1					1079	96.9	0.09	1.43
long-phrases	4					1979	97.1	0.05	0.79
most-phrases		2				9113	97.6	0.01	0.17
freq-phrases		100				35	96.6	2.8	44
no-phrases		∞				4	93.5	23	375
no-stoplist			no			1703	96.9	0.06	0.91
no-local				no		1557	97.1	0.06	1.0
no-aratio					no	1557	97.2	0.06	1.0

Figure 5: A comparison of eight variations on the standard encoding.

4 Discussion

Related work. Several systems remove advertisements from Internet pages or, more generally, modify the raw HTML according to some specification; examples include Muffin [muffin.doit.org], WebFilter [math-www.-uni-paderborn.de/~axel/NoShit], ByProxy [www.besiex.org/ByProxy] and Junkbusters [www.junkbusters.-com]. Unlike AdEater, these systems rely on hand-crafted filtering rules. Some of the systems support centralized repositories of manually-generated rules, though as the Internet grows, a learning approach is perhaps the only way to ensure scalability.

Smokey [13] detects "flames", annoying or harassing mail messages. Like AdEater (but unlike the mail filtering tools accompanying most mail environments), Smokey learns flame-detection rules. However, Smokey's ultimate task is very different from AdEater's: the system simply classifies entire mail messages as flames, rather than (for example) snipping out insulting passages. Machine learning has been applied to numerous Internet applications (see [2, 4, 5, 8] for just a few recent examples), but none of this work directly relates to our task

Learning to identify advertisements in Internet pages is a form of *information extraction*. Learning approaches have been taken for several such tasks; see [1] and earlier Message Understanding Conference proceedings, and the AAAI-99 Workshop on Machine Learning for Information Extraction. These approaches rely on linguistic regularities that are rarely present in the kinds of Internet pages with which we are concerned, though [6, 12] address these limitations.

Summary and future work. We have described the AdEatersystem, a browsing assistant that automatically learns advertisement-detection rules, and then applies those rules to remove advertisements from Internet pages during browsing. In controlled experiments, AdEater achieves very high levels of accuracy while consuming modest resources (both processing rime and preclassified training examples). User's have provided anecdotal confirmation that these results carry over to the "real world" as well.

While building a working system has been our priority, we have made two interesting technical contributions. First, we have formalized our task as one of inductive learning. Second, we have systematically explored the space of possible features for encoding the examples.

While AdEater is fully implemented, several additional features could be adde:

- Note that AdEater made two mistakes on the bottom example in Fig. 1. (However, the result is reasonable: the mistakes involve two small navigational images, the "important" images are intact, and the advertisements are removed.) These problems could be repaired by gathering more training examples. An important next step, therefore, would be to extend the user interface so users could add misclassified images to the training set and re-invoke the learner.

- Second, some user's might prefer one-sided errors (*e.g.*, when in doubt leave images intact). We know of no easy way to bias C4.5rules in this manner, but extending the learning algorithm to do so would be intersting.

- We already mentioned that our task is ideal for exploring "incremental" learning, in which a classifier is modified based on an updates to the training instances, rather than being relearned from scratch. As described above, nearest-neighbor and other lazy learning algorithms are incremental but are undesirable for other reasons. Incorporating an incremental decision tree or rule learning algorithm (*e.g.*, [9, 14]) would improve overall efficiency.

- Our encoding results in many features, and we use a crude *feature selection* mechanism, implicitly embodied in the values of K, M and the stop-list. It would be useful to exploit more sophisticated feature selection strategies (*e.g.* [7]).

Acknowledgements. Much of this research was conducted during the author's 1998 stay at Dublin City University. AdEater is built on top of the Muffin proxy server [muffin.doit.org] and the C4.5rules learning algorithm [11]. Thanks to Barry Smyth for helpful discussion.

References

[1] ARPA. *Proc. 7th Message Understanding Conf.* Morgan Kaufmann, 1998.

[2] D. Billsus and M. Pazzani. Learning collaborative information filters. In *Proc. 15th Int. Conf. Machine Learning*, 1998.

[3] W. Cohen. Fast effective rule induction. In *Proc. 12th Int. Conf. Machine Learning*, 1995.

[4] M. Craven, D. DiPasquo, D. Freitag, A. McCallum, T. Mitchell, K. Nigam, and S. Slattery. Learning to extract symbolic knowledge from the World Wide Web. In *Proc. 15th Nat. Conf. AI*, 1998.

[5] R. Doorenbos, O. Etzioni, and D. Weld. A scalable comparison-shopping agent for the World-Wide Web. In *Proc. Autonomous Agents*, pages 39–48, 1997.

[6] D. Freitag. Information extraction from HTML: Application of a general machine learning approach. In *Proc. 15th Nat. Conf. AI*, 1998.

[7] D. Koller and M. Sahami. Toward optimal feature selection. In *Proc. 13th Int. Conf. Machine Learning*, 1996.

[8] N. Kushmerick, D. Weld, and R. Doorenbos. Wrapper Induction for Information Extraction. In *Proc. 15th Int. Joint Conf. AI*, 1997.

[9] R. Michalski, I. Mozetic, J. Hong, and N. Lavrac. The multi-purpose incremental learning system AQ15 and its testing application to three medical domains. In *Proc. 5th Nat. Conf. AI*, 1986.

[10] T. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[11] J. Quinlan. Simplifying decision trees. *Int. J. Man-Machine Studies*, 27:221–34, 1987.

[12] S. Soderland. Learning information extraction rules for semi-structured and free text. *Machine Learning*, 34(1), 1999.

[13] E. Spertus. Smokey: Automatic recognition of hostile messages. In *Proc. Conf. Innovative Applications of Artifical Intelligence*, 1997.

[14] P. Utgoff. Incremental induction of decision trees. *Machine Learning*, 4:161–86, 1989.