CSI5387: Data Mining Project

Terri Oda

April 14, 2008

1 Introduction

Web pages have become more like applications that documents. Not only do they provide dynamic content, they also allow users to play games, send email, and do many other tasks that used to be reserved for traditional applications.

One of the major technologies enabling web application creation is JavaScript, which allows execution of code in the browser. Unfortunately, because it is so powerful, JavaScript is also abused by attackers. Often, the only suggested defense against malicious web scripts is to disable JavaScript [2]. Unfortunately, doing so disables much of the functionality of some web pages, and completely breaks others. As such, disabling JavaScript is not a viable option for many people, and few other options exist for protection from malicious web code.

Users might gain some protection if it were possible for them to run *some* JavaScript rather than only having the option of all or none. People might only really need to see menu code, or code that displays videos, and want to discard anything else. However, if we are to apply this sort of protection on existing web pages, there needs to be some way to determine class automatically.

For this project, I've chosen two classes to distinguish. The first is code used for displaying JavaScript menus, which is probably a fairly low-risk type of JavaScript that many users would wish to run. The second is advertising JavaScript, which is often more complex and may be something users wish to disable. Although it might have been more interesting from a security perspective to use malicious JavaScript as one of my classes, this data would be more difficult to gather. Since at the outset I was not even sure JavaScript could be classified in this manner, it seemed best to use easier-to-obtain training samples.

The goal of this project is to learn whether the "bag of words" approach used provides suitable attributes for classification, to explore properties of the resulting feature space, and to determine if reasonable accuracy could be achieved in distinguishing the two classes defined.

2 Approaches

2.1 J48

The first algorithm I applied is the J48 algorithm found in `weka.classifiers.trees.J48`. According to Weka, this produces a pruned or unpruned C4.5 decision tree as described in

[4]. The decision tree algorithm was chosen because I wanted to look at classifiers that were easy for a human to understand, so I could see how valid the rules were likely to be based on my knowledge of the domain.

2.2 Naive Bayes

The second algorithm used is the Naive Bayes algorithm found in `weka.classifiers.bayes.NaiveBayes`, which is described in greater detail in [3]. According to the Weka documentation, this is not an updateable classifier, as one might desire if one were building a system which could learn to adapt classes over time. Since my primary goal in using the Bayes classifier was to learn more about the contributions of attributes, it did not seem necessary to use an updateable classifier for this task.

The Bayes algorithm was chosen because it would give weights for each of the attributes used, giving me a better idea of how much different attributes contributed to the system as a whole. The hope was that if the decision trees seemed very easy to mislead, the Bayesian system would be more robust to changes.

2.3 Multilayer Perceptron

The third algorithm used is the Multilayer Perceptron found in `weka.classifiers.functions.MultilayerPerceptron`. This is a neural network which trains using back propagation. I chose this particular algorithm specifically because the resulting classifier is not very easy for a human to analyse and form simple rules based on. The hope that was if simple rules proved too easy to evade, a more complex classifier might offer an alternative.

3 Experimental results

3.1 Method

3.1.1 Choosing data files

The data files used were chosen from a variety of webpages that had JavaScript menus and/or advertisements. Most of these were noticed in the course of my regular browsing, although this number was supplemented by web searches for "JavaScript menu" and by visiting popular sites from the beginning of the Alexa global top 500 list [1].

I chose to look at complete files rather than just the relevant portion of the code. This has the advantage that they were easier to collect, since analysis of code can be time-consuming, but the disadvantage that some JavaScript libraries will contain code with different behaviours. For example, style code or form handling code might be grouped into the same file. However, while this may make the classification task more difficult, it seems fair as files would be a natural boundary for enabling partial JavaScript.

In my data set, there are 32 menu files and 32 advertising files. I chose to use identically-sized sets in order to avoid problems that occur with disbalanced class sizes.

After 32 instances, I was finding the same code more often than I was finding new instances, both for menu code and advertising code. I had anticipated this for the advertise-

ments due to most advertisements being served by a relatively small number of companies, but I was surprised to find some duplicate menu code as well, since these seemed more likely to be custom-made for each website. As it turns out, there are also several very popular libraries available for menus and menu animations, and these are commonly used. Although some effort was made to avoid duplicate code, it is possible that some are heavily derivative (although customised per site) and may have highly similar characteristics.

3.1.2 Parsing the data files

Since I wanted to work with JavaScript code rather than the text off web pages, I needed to make my own parser. This was accomplished with a small Perl script that separated out "words" from whitespace using the `split` function of Perl. I chose to separate words based on anything that was not alphanumeric or an underscore.

My original parser produced 17391 attributes from the 64 files. Unfortunately, this number was so large that Weka had difficulty dealing with the data, spitting out warnings or crashing entirely. Since Weka was unable to even apply filters to reduce the data set itself, I changed the parser to remove any attribute which appeared in only one file. This reduced the number of attributes to a more manageable 3323.

3.2 Results

3.2.1 J48 Decision Tree

The J48 algorithm was run with a confidence factor of 0.25 and a minimum number of objects of 2 (the default settings for J48 in Weka). I used a 10-fold cross validation for the test.

When I was first testing, I tried leaving out one particularly large menu file in order to reduce the attribute set slightly without needing to adapt the parser. The resulting tree, shown in Figure 1 was particularly interesting.

```
when <= 0
|   may <= 0
|   |   UL <= 0
|   |   |   CSS <= 0
|   |   |   |   backgroundColor <= 0: ads (32.0/1.0)
|   |   |   |   backgroundColor > 0: menus (6.0/1.0)
|   |   |   CSS > 0: menus (2.0)
|   |   UL > 0: menus (4.0)
|   may > 0: menus (6.0)
when > 0: menus (13.0)
```

Figure 1: J48 tree on 63 instances

The words "when" and "may" are comment words – much more likely to be used in descriptions of what the code does than within the JavaScript code itself. As such, they occur more often in menu code, which is often written to be editable and customized for a site, often containing many helpful comments. In contrast, advertisement code is generally intended not

to be read at best, intentionally obfuscated at worst. It is rare that the advertisement code has comments.

The next distinguishing word is "UL" which is also associated with menus. Since "UL" is the HTML tag for starting (or ending) an unordered list, it makes sense that it would be commonly used for menus, which are often represented as unordered lists of items.

Another distinguishing word is "CSS" which stands for Cascading Style Sheets, which are used to control layout and appearance of HTML. Again, this term does not appear in many ads, which are most often displaying a single image on the page, but is more common in menus where styles are being defined.

The word "backgroundcolor" proved to be a highly distinguishing feature, appearing in only one of the advertisement samples. In fact, the particular advertising sample in which is appeared is a slight anomaly even among advertisers in that it generates advertisements that appear when the user mouses over a keyword, while most advertisers display advertisements embedded in a page. Because it creates a box around the advertisement that appears, this code would have use for the backgroundcolor property to modify the appearance of the space surrounding the advertisement.

The final tree using all training instances is shown in Figure 2.

```
classname <= 0
|   direction <= 0
|   |   backgroundcolor <= 0: ads (30.0/1.0)
|   |   backgroundcolor > 0: menus (2.0)
|   direction > 0: menus (5.0)
classname > 0
|   10063 <= 0: menus (25.0/1.0)
|   10063 > 0: ads (2.0)
```

Figure 2: J48 tree on all attributes

The final accuracy of this tree was 68.75%, with 14 misclasified menus and 6 misclassified advertisements.

One interesting attribute is the one 10063, which occurs only in two of the files from one advertisement service, Konterra. They are part of code which appears to have been heavily obfuscated.

Classname is also an interesting choice, as it is again related to style definitions and is (predictably) more common in menu code as a result. The attribute "direction" seems a similarly unsurprising one to find only in menu code.

Suspecting that the attribute 10063 was fairly arbitrary, I experimented with reducing the number of attributes used for classification. The attributes were filtered using information gain (InfoGainAttributeEval + Ranker in Weka) and the set was reduced to 500, 50, and even 10 attributes.

When I began gathering data for this project, I worried that the results would be that the instances would be classified primarily by the words "ad" and "menu" – sure enough, one of these words appeared in the tree built on 50 attributes, shown in Figure 3.

```
classname <= 0
|   show <= 0
|   |   focus <= 0: ads (29.0)
|   |   focus > 0: menus (3.0)
|   show > 0: menus (5.0)
classname > 0
|   ads <= 0: menus (24.0/1.0)
|   ads > 0: ads (3.0/1.0)
```

Figure 3: J48 tree on 50 attributes

The most accurate of the trees proved to be the one built using only 10 attributes, which achieved an accuracy of 79.7%, misclassifying 6 ads and 7 menus. This tree is shown in Figure 4. In this small tree, all the classification is done based upon words highly associated with style and HTML appearance ("classname" and "hide").

```
classname <= 0
|   hide <= 0: ads (29.0/1.0)
|   hide > 0: menus (8.0/1.0)
classname > 0: menus (27.0/3.0)
```

Figure 4: J48 tree on 10 attributes

3.2.2 Naive Bayes

The second algorithm I experimented with is the Naive Bayes algorithm. I experimented with different numbers of attributes used as input to Naive Bayes, to see if I could gain any insight from the relative weights given to attributes. The attributes were filtered using information gain (InfoGainAttributeEval + Ranker in Weka) and the set was reduced to 500, 50, and even 10 attributes to get an idea of how much the many attributes contributed to the solution.

Table 1 shows the accuracy of the Naive Bayes classifier with different numbers of attributes. Interestingly, just as we saw with the decision trees in Section 3.2.1, the classifier generally becomes more accurate as the number of attributes decreases. This may be an indication that there are a much smaller set of terms that are key in distinguishing advertisements from menus, and that other terms may simply contribute noise.

Note that the difference between classification at the 50 attribute mark versus the 10 attribute mark is only one misclassified instance.

The weight per attribute is fairly interesting, too. Consider Table 2, which shows the attributes used in classification and the means calculated for each. It seems that while the menu instances are being classified by what they are, the ads are being classified by what they are not. That is, the ads are being mostly classified by their missing attributes, and the menus are being classified by the attributes they do have.

# of	ads			menus			Total
Attributes	TP Rate	FP Rate	Precision	TP Rate	FP Rate	Precision	Accuracy
3323	0.938	0.219	0.811	0.781	0.063	0.926	85.9%
500	0.938	0.188	0.833	0.813	0.063	0.929	87.5%
50	0.938	0.063	0.938	0.938	0.063	0.938	93.8%
10	0.938	0.094	0.909	0.906	0.063	0.935	92.2%

Table 1: Accuracy of Naive Bayes using varying attribute numbers.

Class	Attribute	Mean	StdDev
Ads	this	0.41	0.49
	scrolling	0.41	0.49
	event	0.16	0.36
	hidden	0.16	0.36
	onclick	0.13	0.33
	getelementsbytagname	0.13	0.33
	visibility	0.09	0.29
	classname	0.09	0.29
	hide	0.06	0.24
	when	0	0.17
Menus	this	0.97	0.17
	event	0.78	0.41
	classname	0.75	0.43
	getelementsbytagname	0.75	0.43
	hidden	0.72	0.45
	onclick	0.69	0.46
	hide	0.66	0.48
	visibility	0.63	0.48
	when	0.41	0.49
	scrolling	0	0.17

Table 2: Attribute weights for 10-attribute Naive Bayes

3.2.3 Multilayer Perceptron

Unlike the decision tree and Bayes algorithms, which both produced results very quickly, the neural network quickly used all of the memory assigned to Weka when attempted on all attributes. Even using the reduced set of 500 it took a noticeably long time to prepare and it used up most of the CPU while doing so.

For all that work, it did produce a highly accurate classifier using 500 attributes, with 1 misclassified ad and 3 misclassified menus for a total of 93.75% accuracy.

Interestingly, unlike the other classifiers, the neural network did not become more accurate when using fewer attributes. At 50 attributes, it misclassified 3 ads and 6 menus for a total accuracy of 85.9%. At 10 attributes, it rose again to 92.2% accuracy with 2 ads and 3 menus misclassified.

3.3 Comparison and Discussion

The time taken to run the algorithms is summarized in Table 3. As you can see, the time taken to run the Multilayer Perceptron algorithm greatly exceeded the time needed for either J48 or Naive Bayes.

Algorithm	# of Attributes	Time (seconds)
j48	10	0
	50	0.01
	500	0.09
	3323	0.41
NaiveBayes	10	0
	50	0
	500	0.05
	3323	0.12
MultilayerPerceptron	10	0.11
	50	13.46
	500	1268.23

Table 3: Time taken to run algorithms

The accuracy results are summarized in Figure 5. The Naive Bayes and Multilayer Perceptron both perform better than J48, although because the time required to run the multilayer perceptron is so high, the Bayesian solution does seem to be the most promising one of the three, purely based on accuracy and time required to create the classifier.

However, in the course of examining the attributes I have noticed that many of them are related to style settings for changing the appearance of HTML. Most the attributes used for classification are found only in the menu files. This fact makes it very easy for an advertiser to make their code to "look like" a menu: they would simply need to provide some extra style information. As such, it would be very easy for code from one class to intentionally be mis-classified. This means that this approach may have only limited usefulness for a classification task where someone might want to intentionally create instances that look like they belong to another class.

4 Conclusions

My original goal involved finding a way to allow users a way to run "some" JavaScript that might be safer. I have found that given my two classes, a Naive Bayes algorithm can be used to find a classifier with reasonable classification accuray in minimal processing time.

However, in analyzing the attributes used, I have found that the ones contributing most to both the decision tree and Bayesian classifiers are words that are associated with menus, while the advertisements seem to share fewer attributes that can distinguish them. This means that by adding attributes to an advertisement, it would be possible to make it look like a menu – if this is true of many classes in JavaScript, we may find it difficult to deal with the case where someone intends to have their code mis-classified. However, this also means

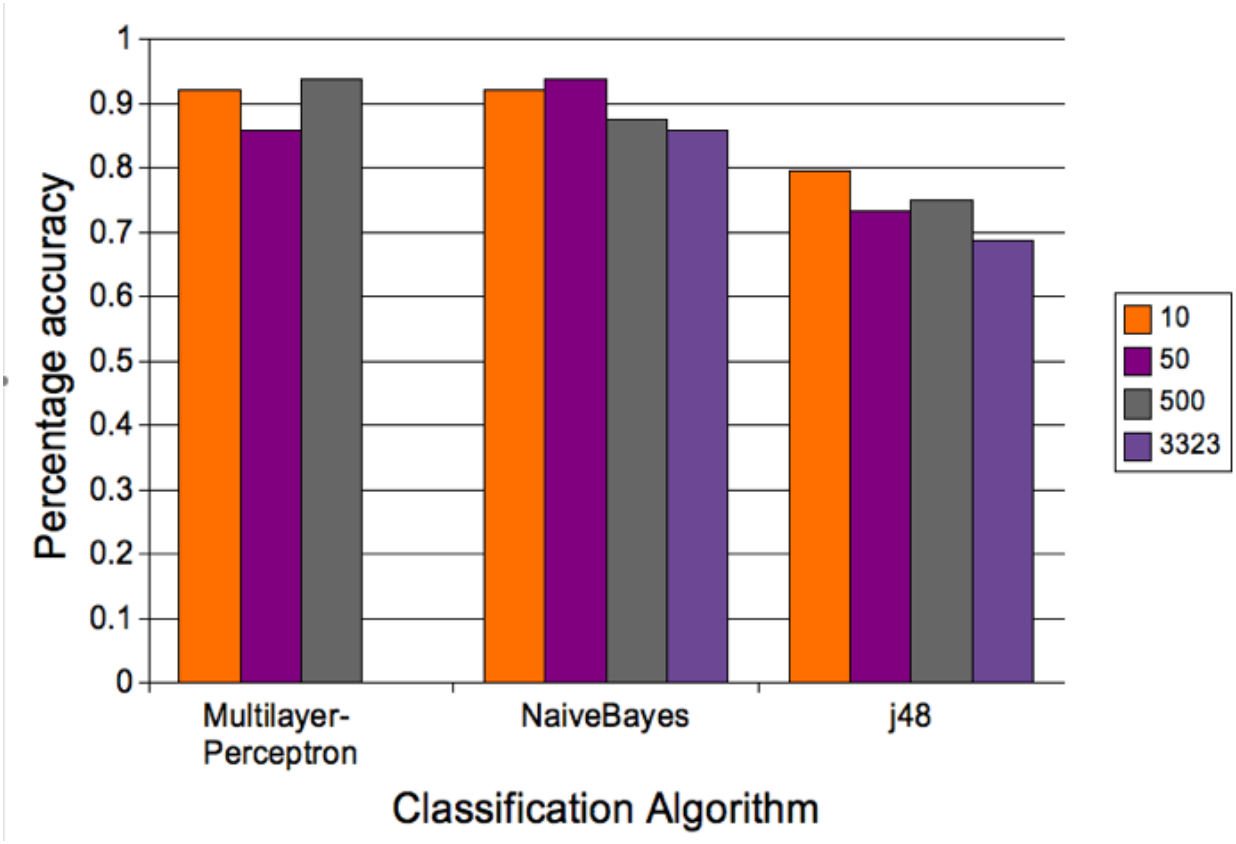

Figure 5: Classification Accuracy on JavaScript Samples

that code that is mis-classified can be easily adjusted to look more like the other instances of that class. (This is also a problem that occurs in spam detection, where spammers will use known good properties to make their messages appear less spammy to popular classifiers.)

More troubling is that with the menus and advertisements, the particular attributes were often words associated with style information or even comments, meaning that it would be fairly easy to put these extra words in without it interfering with other code.

It is not yet clear whether this would be true if more classes were chosen, however. It may be that additional classes might force the classifiers to look past stylistic information to information related to what the code actually does, yielding additional insights.

I have not yet formulated a way of determining attributes from code other than the simple "bag of words" approach used, but I would like to investigate other options further in the future. Since this was not the primary task for this assignment, I have not yet explored other options and do not know what my next steps might be in this area.

Although I do not believe that my current approach will yield good long-term results for choosing some JavaScript to run in a page, this project has been fairly illuminating as far as properties that distinguish different pieces of code. I would like to try out some other classes, including JavaScript related to games, videos, or used for collecting statistics on users. I would also be incredibly interested to simply gather a large collection of JavaScript code and run some clustering algorithms on it, to see if it naturally finds groupings that make any sense in terms of the code function as the user sees it.

References

[1] Alexa Internet, Inc. Alexa top 500 sites, April 2008. `http://www.alexa.com/site/ds/top_sites?ts_mode=global&lang=none`.

[2] CERT® Coordination Center. Frequently asked questions about malicious web scripts redirected by web sites, 2004. `http://www.cert.org/tech_tips/malicious_code_FAQ.html`.

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

[4] R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.