A mod_ml Bot Detection System

1. Introduction

The bot detection system is intended to give a web service provider the ability to automatically detect bots with a reasonable amount of certainty even if the bot is not identified explicitly.

At the centre of the system is the Apache mod_ml module. This module serves two functions: formatting and forwarding data on user behaviour for preprocessing and, secondly, as an intermediary in the classification process. Figure 1 illustrates these two functions.

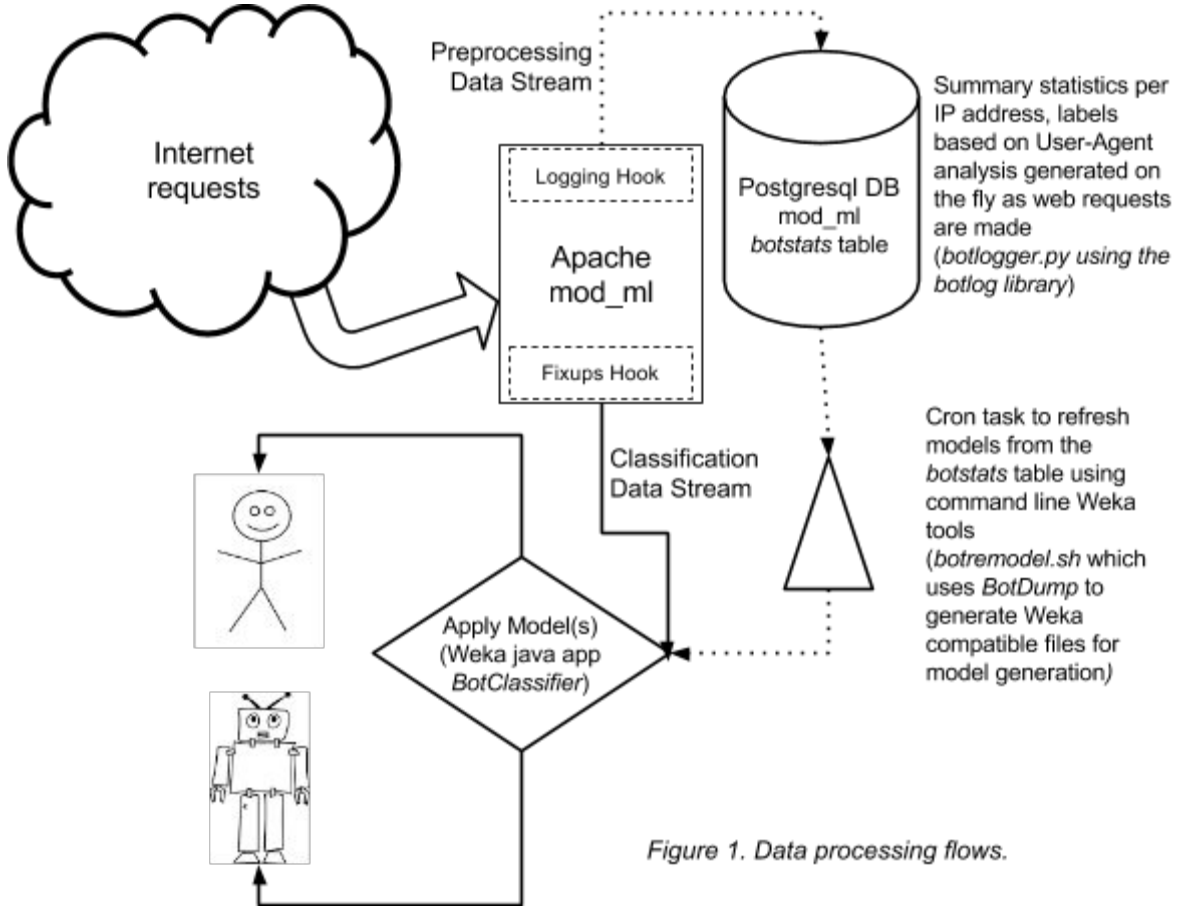


Figure 1. Data processing flows.

2. Preprocessing

Preprocessing uses the mod_ml MLPreProcess directive to forward encoded request data in the to an external preprocessing process. An example is shown below:

```
{"useragent":"xxx", "epoch":"1437481437361", "hour":"12", "REMOTE_ADDR":"45.74.2.131",
"content_type":"text/html", "HTTP_HOST":"206.12.16.219:8898", "status_line":"200 OK"}
```

The data sent to the preprocessor consists of information useful for classifying traffic and collecting timing statistics on visitors:

- **useragent** - the contents of the User-Agent header; this, classified as bot or human by the Google http://www.useragentstring.com API, is used to train the model; the data is cached in the mod_ml Postgresql database located on cloudsmall8 *botlabels* table
- **epoch** - the epoch time in milliseconds from the apache request_rec data structure; differences in time are used to generate features for model building
- **hour** - the hour of the request; once again differences in time are used to generate features for model building
- **REMOTE_ADDR** - the IP address of the requester; this, along with the HTTP_HOST, is used to group the stats and for classification
- **content_type** - the ratio of html vs non-html content is used as a feature
- **HTTP_HOST** - which web server served the request; used with the ip as a key for grouping stats
- **status_line** - the HTTP return code for the request; the ratio of 200/300 vs 400/500 is used as a feature for model building

Data for each REMOTE_ADDR/REQUEST_URI pair is saved in three tables in the mod_ml DB: *botlog, botlatest* and *botstats*:

The *botlog* and *botlatest* tables use the same structure as shown in Table 1.

Table 1: botlog/botlatest

Column Name	Column Type
logid	integer primary key
hour	integer
remote_addr	character varying(128)
status_line	character varying(64)
useragent	character varying(512)
epoch	bigint
content_type	character varying(64)
http_host	character varying(128)

The *botlog* table is a temporary buffer for incoming requests to the preprocessor. The *botlatest* table holds the last log entry for a given visitor. This information is used during processing to calculate the time difference between HTML page requests.

The *botstats* table structure is listed in Table 2.

Table 2: botstats

Column Name	Data type	Notes
ip	character varying(64)	HTTP_HOST/REMOTE_ADDR
pages	bigint	number of html page requests
reqs	bigint	number of requests of any type
errs	bigint	number of 4xx and 5xx status_lines
diffs	integer[1000]	sample distribution of paired epoch time differences
n	integer	count of items saved in diffs column
sum	bigint	sum of items saved in diffs
mean	double precision	mean of diffs items
var	double precision	variance of diffs items
skew	double precision	asymmetricality of diffs distribution
kurtosis	double precision	flatness of diffs distribution (more -ve = flatter)
hourdiffs	integer[1000]	distribution of pairwise hour differences
hn	integer	count of hourdiffs
hsum	bigint	sum of hourdiffs
hmean	double precision	average hour difference
hvar	double precision	variance of hourdiffs
hskew	double precision	asymmetricality of hourdiffs distribution
hkurtosis	double precision	flatness of hourdiffs distribution
hours	integer[24]	html page request counts per hour
htn	integer	count of hourly reqs
htsum	bigint	sum of hourly reqs
htmean	double precision	average hourly reqs
htvar	double precision	variance of hourly reqs
htskew	double precision	asymmetricality of hourly reqs distribution
htkurtosis	double precision	flatness of hourly reqs distribution
uas	varchar(512)[10]	array of last 10 user agents associated with ip
class	integer (-1 human, 1 bot)	actual class based on www.useragentstring.com label
label	varchar(32)	label from *botlabels* table
prediction	integer (-1 human, 1 bot)	predicted class based on amalgamated predictons
sample	integer	identifies which rows were used to create models

The diffs and hourdiffs arrays are allowed to grow to 1000 items in length but the pages and reqs counts are allowed to grow to arbitrary sizes. Only HTML request time differences based on the content_type field are tracked. With this scheme only the most recently seen html file request needs to be saved in the database, greatly reducing the required space. For non HTML

requests counts are kept but they are not added to the distributions as these requests are highly correlated with the HTML document requests. In other words we are likely not gaining much new information by tracking them.

The *botstats.class* and *botstats.label* fields are mapped to the *botlog.useragent* via the *botlabels* table. Table 3 shows the structure of this table.

Table 3: botlabels

Column Name	Column Type
useragent	character varying(255)
label	character varying(32)
isbot	integer (0 human, 1 bot)

These labels are strictly used for supervised learning and evaluation of classification. The current list of labels from the Google www.useragentstrng.com service is along with their the arbitrary classification used for identifying bots for supervised learning. Table 4 shows how the User Agent strings were classified.

Table 4: label categories

User-Agent Label	Bot?
Mobile Browser	no
Browser	no
Offline Browser	no
unknown	(left as null)
Console	no
Cloud Platform	no
Feed Reader	yes
Librarie	yes
LinkChecker	yes
Crawler	yes
Validator	yes
Other	yes

See the *Testing* section for implementation details.

3. Classification

The goal of the project is to see if it is possible to classify web site visitor behaviour without extensive logging. Ideally the features used should only involve recent samples of behaviour for a given visitor. What is measured is not request history such as what would be found in the Apache access log but rather behavioural measures such as time of day of the requests and inter-request time. Bots are likely to behave differently in these dimensions than humans. For example the skew of the distribution of inter-request time differences for a bot is closer to 0 than that of a human.

Looking at the timing data as a whole it appears that there may be measurable differences between bot and human behaviour. Table 5 shows the t-scores of the bot vs human categories for mean time differences and request hour.

Table 5: t-scores for distributions of means

measure	t-score (*N1* = 172385, *N2* = 5354)
botstats mean inter-request time difference	-18
botstats mean inter-request hour difference	-26
botstats mean request hour	-14

Although the distributions of mean time differences would be considered highly significant if the distributions were normal they are not normal (Jarque-Bera scores are far above 0). Thus these results should be taken with a grain of salt.

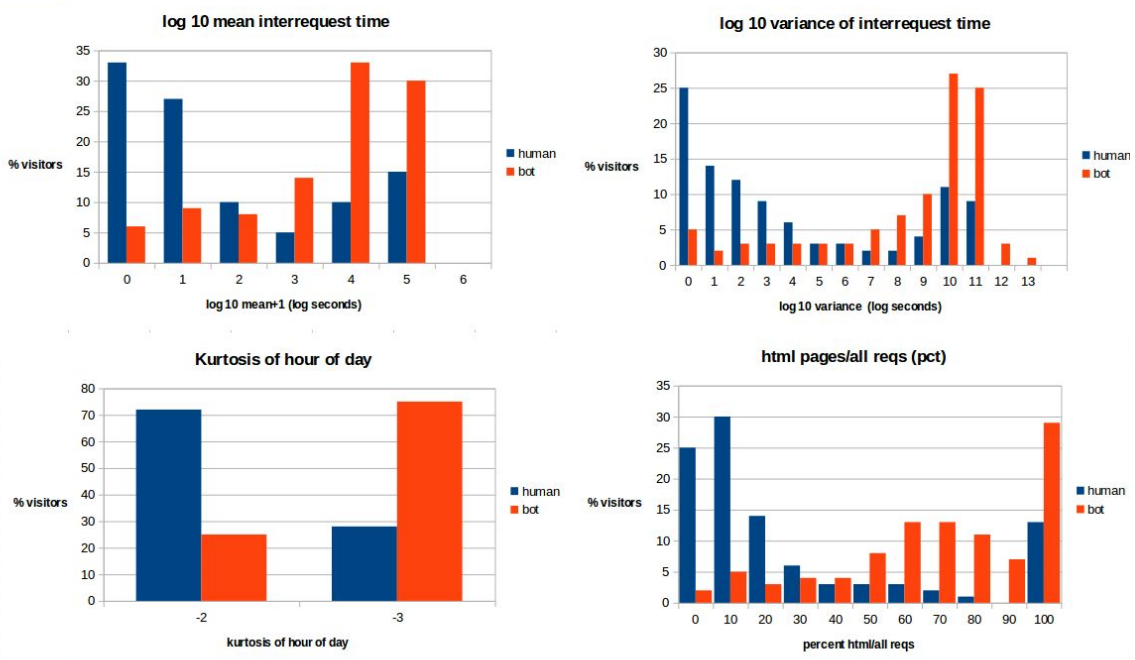
It is also possible that many bots do not identify themselves via the User Agent string. It may be possible to further improve the generated models by reclassifying some “human” outliers as bots based on request count or other aggregate measures reflecting a longer term history.

T-scores may also be misleading as they only show whether there is a difference in summary statistics between two samples. For any individual item it may be difficult to distinguish between bot and human visitors for a significant number of cases. Furthermore, t-scores make the assumption that the underlying distributions being compared are normal. However, the request data distributions are generally quite flat. This applies to both individual visitors and also distributions of statistics for groups of visitors. Because the distributions are flatter than normal there is likely more overlap between the two distributions despite the fact that the t-scores suggest that they are distinct.

Looking at the distributions it is clear that for many quantity independent statistics bots and humans appear to be different. Figure 2 shows some examples of features with distinct distributions for bots. Examples of features where bots and humans have different distributions: inter-request time (the *botstats.diffs* array) mean and variance, kurtosis of html requests per hour of day (the *botstats.hours* array) and the percentage of requests that are html pages. Bots

tended to have much more varied inter-request times but also tended to have much longer mean inter-request times. Bots tended to visit at more uniformly distributed times of day. Bots also tended to download more html pages compared to other media (graphics, css etc).

Figure 2: some features that distinguish bots and humans - inter-request time (mean + variance), kurtosis of hour of day, relative frequency of html page requests out of all requests.



Weka turned out to be invaluable for testing different classifier models. The models that worked best with this data were decision trees. Functional classifiers such as SVM, Multilayer Perceptron and Logistic Regression were not as effective and in fact tended to fare worse than Naive Bayes (ROC AUC of .77). In fact, many of the models tested were even less accurate than Naive Bayes.

Artificially boosting the number of bot examples improved the performance of the generated classifiers. The number of human examples was 30 times larger than the number of human examples available for training and testing. Imbalances in the number of available examples are known to affect performance of many learning algorithms.

Some early output from Weka can be found in weka-results.txt. These results are misleading as they only test on the amplified data. In reality to get more than 50% bot detection over 150,000 examples had to be supplied (amplified to over 200,000 for training). Even with such a large dataset results were mixed. As the “expert” labelling by www.useragentstring.com assumes that all internet users are honest there are likely a number of misclassified IP addresses. To remedy this the “human” visitors that had request counts greater than one standard deviation larger than the mean were reclassified as bots (amounting to 1478 reclassifications out of 10047 total bots).

Functional classifiers such as SVM and Logistic Regression are known to work better with normalized data. However, with normalization many features were at or near 0. Thus there was potentially a problem with rounding errors. This rounding error proved to be a significant issue in practice.

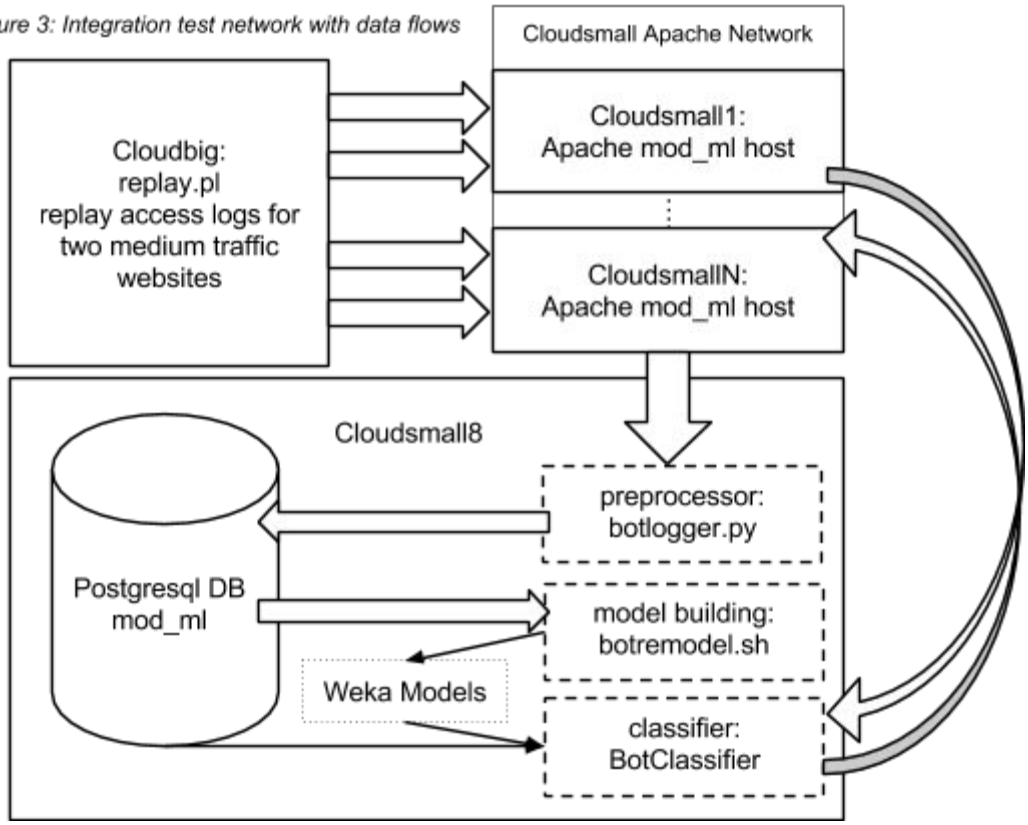
Weka was not the only machine learning platform tested. Vowpal Wabbit, which performed poorly with the reduced 6 normalized features (~75% correct overall using the hinge loss function), performed better with the full feature set without normalization. Using the full set of 16 raw features from the *botstats* table the scores of all the models, Weka or Vowpal Wabbit, improved significantly. See the Results section below for details.

4. Testing

Integration test architecture

Testing was conducted over over a week-long period. Figure 3 shows the test set up for integration testing.

Figure 3: Integration test network with data flows



A three month set of Apache log data was rerun using a custom developed replay script (*/srv/cal/src/apache/access-logs/replay.pl* on cloudbig). These requests, received by two Apache servers (cloudsmall1 and cloudsmall2), are processed by mod_ml as described above

and sent to a preprocessor running on cloudsmall8 (*/srv/cal/src/apache/ml/analysis/botlogger.py*). This preprocessor reads the JSON sent from the mod_ml Apache servers and creates or updates entries in the *botlatest* and *botstats* tables.

On an hourly basis a script is run that classifies rows in *botstats* using www.useragentstring.com. While this step is part of the model building task it seemed to make sense to reduce the time to build models by doing some of the preparation on an ongoing basis.

Periodically samples of data gathered by *botlogger.py* were analyzed with Weka. The sampled *botstats* rows are saved in Weka ARFF format files in a subdirectory in */srv/cal/src/apache/ml/analysis/models* and models are generated. These models are saved as serialized java classes that can be later used by the classifier to classify visitors based on browsing behaviour.

The integration testing was done in two phases: a data gathering phase where a large number of requests were fed into the preprocessor to generate enough statistics that models could be built and a phase where the classification process was refined.

In the first phase classification is turned off to avoid overloading the preprocessing server. The preprocessing server, cloudsmall8, was a relatively light duty virtual machine with 4 cores and 8 GB of RAM. During the preprocessing phase it was at almost 100% CPU at all times, handling approximately 1.1 million preprocessing requests per hour over several days. However, this did not appear to be the upper limit on what the preprocessing system could handle. In practice the preprocessor was quite responsive.

In the second phase model testing was attempted, some of the results of which were described previously. Different models were tried on samples of the *botstats* data both through a local testing script (*bot1000.sh* in the *ml/analysis* directory) and via mod_ml Apache servers.

Configuration of mod_ml Apache servers

The cloudsmall1-8 servers all had *mod_ml.so* installed. In addition, a configuration file *ml_bot.conf* was added to the Apache configuration to manage sending data to the preprocessor and classifier. Because many of the links resent via the replay script were PHP pages the PHP apache module was temporarily removed from the configuration.

Figure 4 shows the contents of the ml_bot.conf file. This file contains directives for the mod_ml module. It defines two processes: an IP based preprocessor at port 39992 on cloudsmall8 and an IP based classifier at port 39999 on cloudsmall8. For more information on mod_ml directives see the mod_ml users guide.

Notice that in the *ml_bot.conf* file there are many directives of the form “MLFeatures header ...” Because the integration test was resending requests from existing log files, many of the fields normally populated by the web browser or crawler before sending the request would not be the

logged values. The request time and remote IP address are examples of fields that need to be changed to an arbitrary value when replaying a request. To get around this limitation without drastically reconfiguring the Apache server, headers were sent with the desired information and mod_ml was directed to look for the information in these headers.

Figure 4: the ml_bot.conf file

```
# data gathering for identifying bots

<IfModule ml_module>
	MLEnabled on

	# section for preprocessor
	# doesn't like /g and \s are stripped from control characters
	MLDefFieldProc regex "s/\s*$//s"
	MLOutFormat jsonfields
	MLFieldProc regex "s/^0*//" time hour=%H
	MLFieldProc ip cloudsmall8.cs.surrey.sfu.ca:39993 header epoch
	MLFieldProc regex "m/.*?:0*(\d+):/" header hour
	# expected input data: all the header stuff is added by curl at the client
	MLLabels header useragent=User-Agent
	MLFeatures header epoch hour
	# MLFeatures time epoch hour=%H
	# MLFeatures env REMOTE_ADDR
	# MLFeatures env REQUEST_URI
	# MLFeatures request status_line
	MLFeatures env HTTP_HOST
	MLFeatures header REMOTE_ADDR
	MLFeatures header status_line
	MLFeatures header content_type
	MLPreProcess ip cloudsmall8.cs.surrey.sfu.ca:39992

	# new section for classifier
	MLOutFormat raw
	MLFeatures env HTTP_HOST
	MLFeatures header REMOTE_ADDR
	MLOutProc regex "s#^\s*(\S+)\s+(\S+)\s*$#$1/$2#'
	MLClassResponse "YES" "env" "client=bot"
	MLClassResponse "NO" "env" "client=human"
	MLClassResponse "NOIP" "env" "client=error"
	MLClassResponse "NOTFOUND" "env" "client=unknown"
	MLClassifier ip cloudsmall8.cs.surrey.sfu.ca:39999

	SetHandler ml
</IfModule>
```

The preprocessor, due to its heavy resource use, must be careful to not get overloaded with irrelevant requests. To limit traffic, it checks the */etc/hosts* file to identify hosts that are local. All other hosts are rejected. However, for additional security iptables could also be set up to limit traffic from external hosts. Figure 5 shows rules that can be used for this purpose.

Figure 5: iptables rules for locking down cloudsmall8 in /root/iptables-whitelist.sh

```
#Flush existing rules
iptables -F
# Set up default DROP rule for eth0
iptables -P INPUT DROP
# Allow existing connections to continue
iptables -A INPUT -i eth0 -m state --state ESTABLISHED,RELATED -j ACCEPT
# Accept everything from the local network
iptables -A INPUT -i eth0 -s 206.12.16.0/24 -j ACCEPT
iptables -A INPUT -i eth0 -s 142.58.184.0/24 -j ACCEPT
...
```

The mod_ml module works by collecting fields from request data which a system administrator selects by using mod_ml Apache configuration directives. The mod_ml module processes the data and creates an output string that it can then send to a preprocessor or classifier. The mod_ml module can define an arbitrary number of these field strings and processes. In the *ml_bot.conf* file two different feature strings are created. One for the preprocessor with more detailed request information and a simpler IP based key for the classifier.

Any returned data from a preprocessor is ignored (unless trace logging is turned on in the Apache configuration). However, mod_ml can be set up to respond to classifier responses. In this case an environment variable “client” is set. This is a particularly powerful technique as other Apache modules such as mod_rewrite can be directed to respond to these environment variables. However, for the purposes of testing sending the response on to an IP service that can collect statistics on classification might also be a reasonable choice. Likewise the classifier can keep statistics on its own performance provided they do not significantly affect latency.

Preprocessing database setup on cloudsmall8

During the preprocessing phase of testing cloudsmall8 was very heavily loaded. Reducing preprocessing system latency was a challenge. What worked in this case was to make the preprocessing step as massively parallel as possible. The preprocessor, *botlogger.py*, used a library *botlog.py* to manage the Postrgresql connections. Figure 6 shows the structure of the preprocessor. The *botlog.py* managed the work that it was fed by *botlogger.py* through the *botlog.process()* function. This function took a list of JSON messages from mod_ml as input, decoded the messages and saved them to a temporary table *botlog*. Any new IP addresses found in the messages were assigned to one of a pool of threads managed by *botlog.py*. Each thread was solely responsible for messages from a specific group of IP addresses while there

were still messages available from that IP address in *botlog*. In all there were 512 processing threads managed by *botlog.py*.

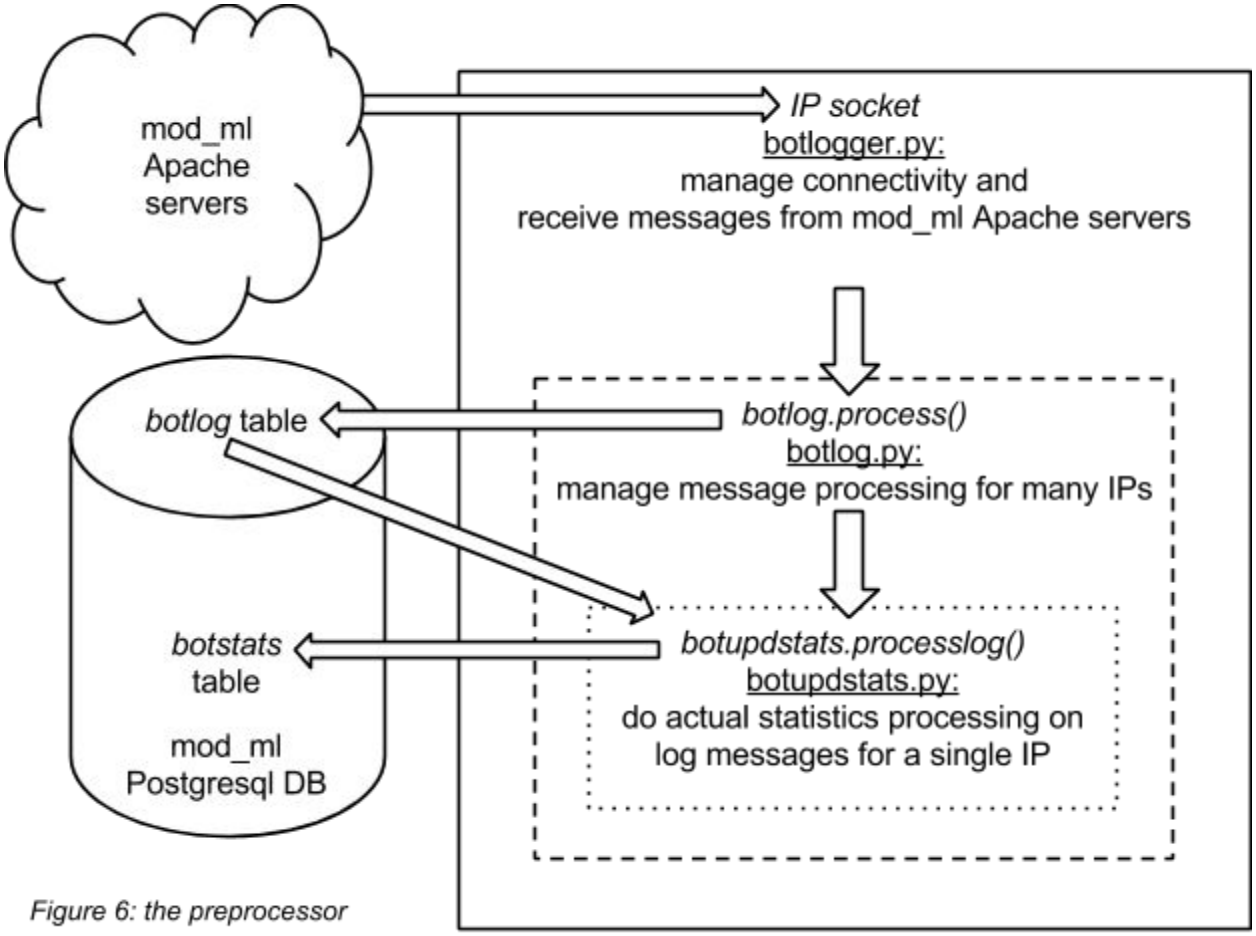


Figure 6: the preprocessor

Postgresql 9.3 was used for the database back end on cloudsmall8. To handle such a heavy load with minimum latency the Postgresql database was tuned. Three parameters were modified in */etc/postgresql/9.3/main/postgresql.conf* on cloudsmall8 to work with *botlog.py*:

- data directory was moved to */srv/cs8/postgresql* but path in postgresql.conf not changed
- max_connections - set to 600
- shared_buffers - set to 2 GB or ¼ of the available memory (recommended setting)
- effective_cache_size - set to 4 GB or ½ the available memory (recommended setting)

With these settings the preprocessor could handle 200 to 1000 messages per second gracefully.

The classifier

The process of classification began with model building. The first step in model building for a supervised learning process such as classification is to create labelled data. The *botstatsclass.py* (in */srv/cal/src/apache/ml/analysis*) does this job by invoking the www.useragentstring.com API and caching the results. The www.useragentstring.com service, as described above, provides a web based API that accepts User Agent strings as input and responds with categorical labels. These labels can be classified as bot or human and the class stored in the *botstats.class* field. For purposes of classification -1 indicated a human and 1 indicated a bot, a convention used so that SVM classifiers could be built from the data.

Labelled data was processed to form models that could be used to classify inputs. Weka was found to be a very versatile tool for developing machine learning models. Because the boundary between bots and human visitors is likely to be nonlinear a number of different machine learning models were developed and tested. These models are often used together to classify a visitor based on saved statistics regarding previous visits.

Building models with a relatively large dataset is often time consuming. Model build times can range from less than 1 minute to more than 10 minutes depending on system load and type of model being built. Tools were developed that could be run in the background and permitted the building of a wide variety of models for testing.

Weka is a java based system and a set of java tools were developed to automate working with Weka classes. Luckily it was relatively easy to make a versatile model build system with a small number of simple programs.

The first such program was *BotDump*. This program samples data from *botstats* and builds Weka compatible ARFF format data files. *BotDump* takes as input the output file name and optionally the size of the sample (the default is 20,000 rows). It queries a random set of rows and automatically amplifies the less frequently found rows to ensure an even representation of classes in the resulting file. It then randomizes the resulting rows before saving them to disk. The ARFF format file can be used directly by Weka classifier classes via the command line. See Figure 7 for an overview of this process.

Finally, once one or more models for classification are created, the saved model can be used for classification. In the case of Weka these models are all serialized java code that can be unpacked and run against an input.

BotClassify is a java class that handles classification for mod_ml Apache clients. It binds to an IP socket and accepts as input visitor identifiers, essentially the IP address of the visitor, returning a response indicating whether it believes the visitor is a bot or a human. Figure 8 diagrams the process.

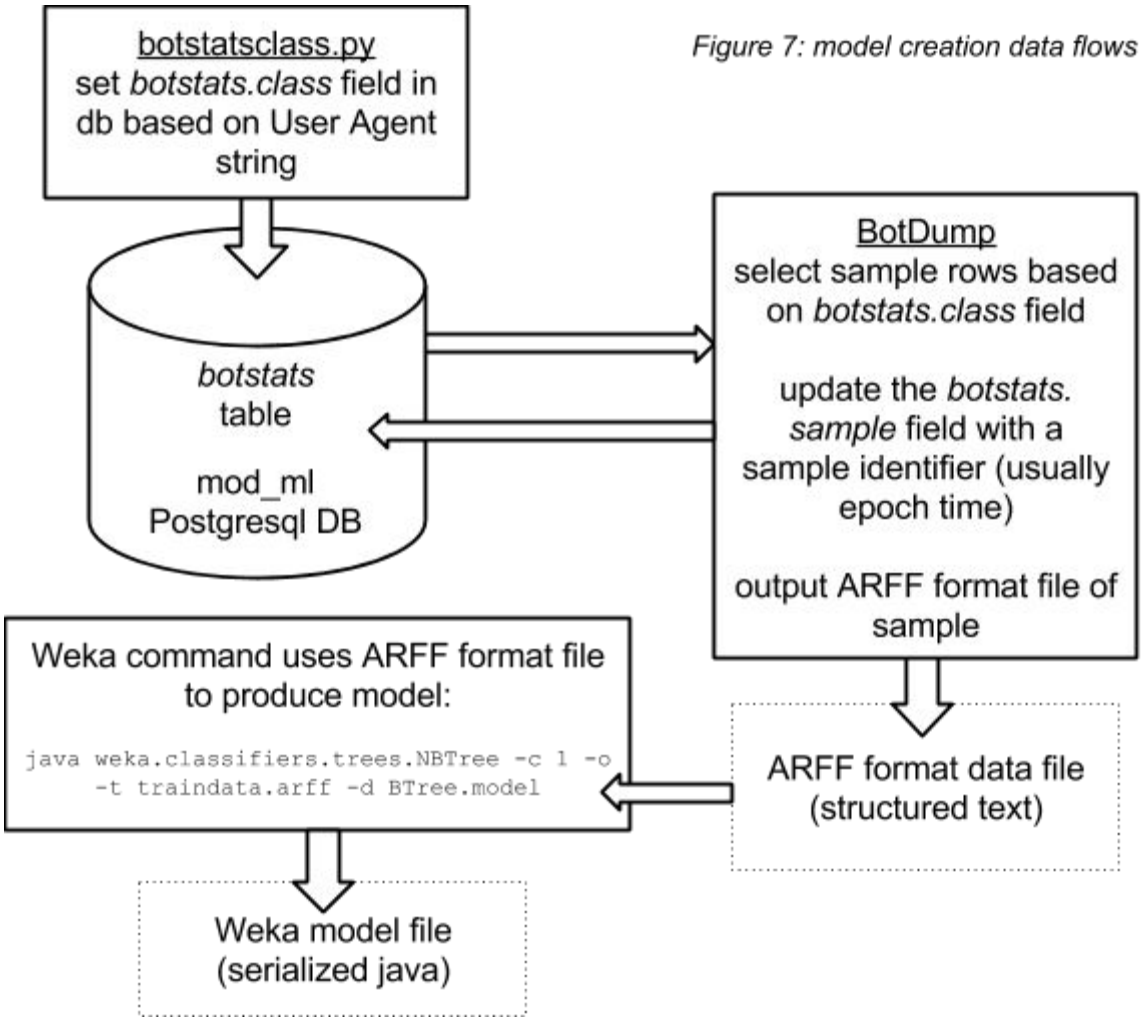


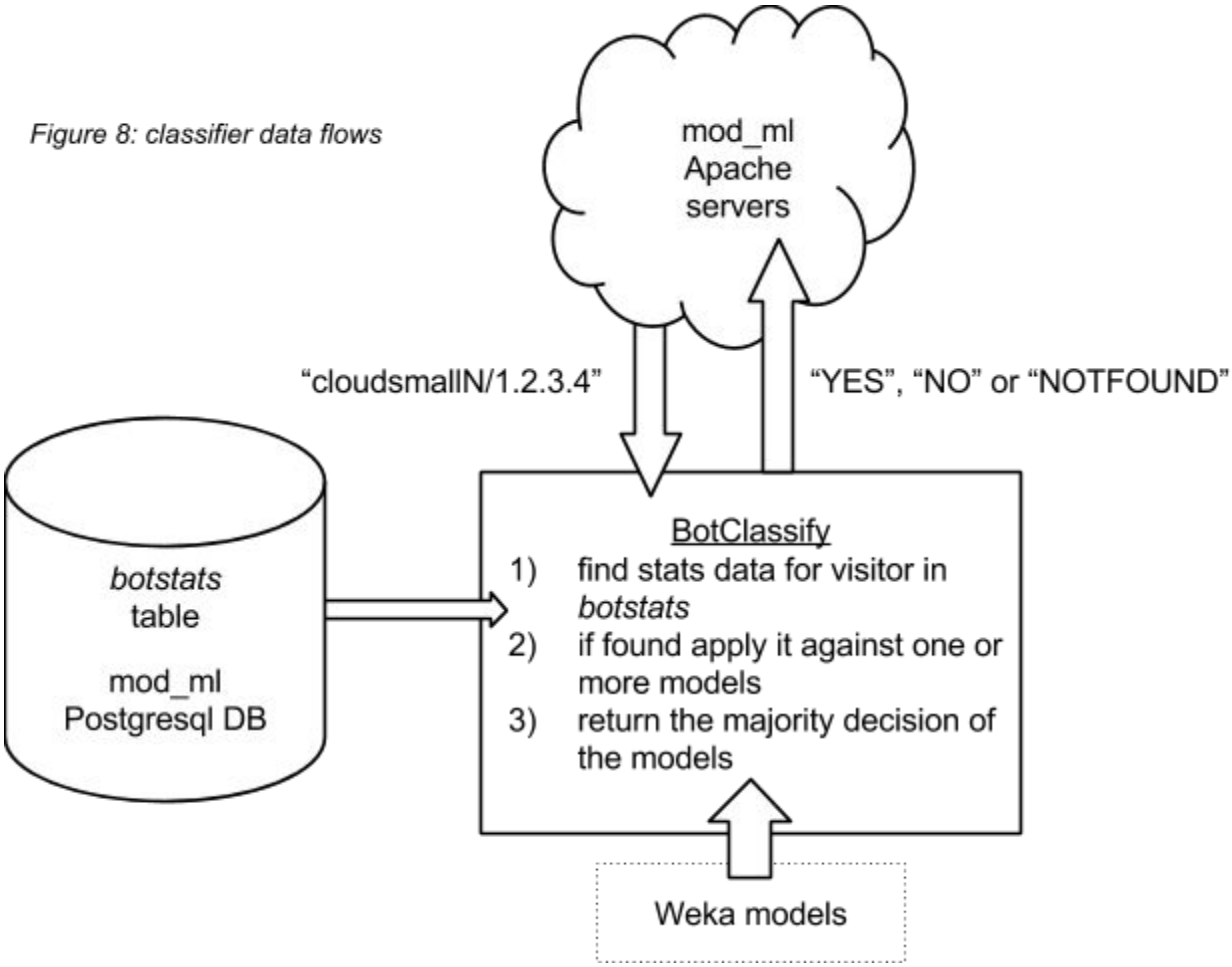
Figure 7: model creation data flows

When a mod_ml Apache client sends *BotClassify* data, it checks the visitor identifier for validity and looks up that visitor in the *botstats* table. If the visitor is not found *BotClassify* returns NOTFOUND to the mod_ml client. If the visitor is found the *botstats* data is formatted and applied against the provided classifiers. Each classifier determines whether the visitor is bot or human and the majority decision is returned as either a YES or a NO.

The mod_ml client takes this information and, if given the appropriate directives, acts on the output. For integration testing this output is of the form of environment variables that can be checked in the web server output or logs.

However, to test models it is much more efficient to use command line tools such as the *test1000.sh* script (also in the same directory as *BotClassify*). The *test1000.py* script will find 1000 *botstats* entries labelled as bots and 1000 labelled as humans and send them to a live BotClassifier instance. The output from the classifier is counted and the total correct is returned.

Figure 8: classifier data flows



5. Results

The four Weka algorithms for detecting bots were RandomForest, BFTree, SimpleCart and J48. The sample data used to create the models was a true stratified sample of bot human examples.

Table 6 summarizes a series of tests that consisted of sending either 1000 visitor ids labelled as bot and 1000 ids labelled as human to a classifier process. In addition, latency was measured based on round trip time to send data to the IP based classifier service (*vwbotclassify.py* for Vowpal Wabbit models and *BotClassifier* as described above for Weka models). Most of the classifier latency represents communications and database processing. The Vowpal Wabbit models did not have this overhead during these tests. In a production situation getting data out of the database for classification is likely to be a hurdle. Using a technology such as *memcached* to store known visitor ids will probably be necessary to reduce latency.

With more than 200,000 examples the Weka models took a very long time to build. SimpleCart and RandomForest took over an hour to build and BFTree took more than 2 hours. J48 was by far the fastest, taking only 5 minutes. The Weka model files were quite large (BFTree being over 100MB in size). The size of these complex models may affect processing time. Vowpal Wabbit tended to produce small models taking less than a minute to build that were also quick to test.

Table 6: a comparison of some Weka and Vowpal Wabbit models

Model Type	Sample	Latency	1000 bot	1000 human	F-score
SimpleCart	153845	250 ms	51.1%	94.5%	0.8426
BFTree	153845	257 ms	52.7%	94.9%	0.8493
RandomForest	153845	236 ms	61.8%	97.9%	0.8879
J48	153507	249 ms	66.5%	96.3%	0.8975
(retest)		267 ms	57.8%	97.2%	0.8732
Ensemble of All Above		287 ms	76.8%	96.6%	0.9288
Vowpal Wabbit logistic loss function	153507	0.9 ms*	85.7%	71.1%	0.8789
Vowpal Wabbit hinge (svm) loss function	153507	0.9 ms*	89.1%	71.9%	0.8920

** the Vowpal Wabbit scripts did not use individual database lookups*

[Given the strong results of the Vowpal Wabbit models tools were developed to take advantage of them for testing. The *vwbotclassify.py* script is a drop in replacement for *BotClassify* that can be used in exactly the same way as *botclassifier.sh*. It uses *vwbotclassify.sh* to make predictions on input. Tools for making the Vowpal Wabbit models were added to *botremodel.sh*.]

With an F-score of 0.9288 the ensemble model performed the best overall. However, the Vowpal Wabbit models outdid Weka at finding bots. Having some way to combine predictions from both types of models could possibly enhance the accuracy of the classifier.

6. Conclusion

Much of the work on machine learning technology has been in the theoretical area of developing algorithms that can make sense out of large, ambiguous datasets not unlike the common web traffic logs used for testing in this project. However, applied systems research is another area where there is tremendous potential to apply these theoretical insights to real human problems.

This project explored the practical problems involved in integrating machine learning technology into a web server. The practical problem of gracefully identifying whether a given user is a human or web robot was used as a way to identify some of the challenges and opportunities of this type of design.

Latency in machine learning technology turned out to be a major issue at every stage. The sheer volume of data that can pass through even a moderately busy website could easily overwhelm someone casually trying to make sense of the data. To remedy this problem and to try and reduce the need for logging, a sampling based design was implemented. This had the effect of reducing over 18 million rows of log entries to a more manageable 1 million statistics table. Furthermore, quasi-time invariant features were identified that could potentially distinguish humans from robots without the requirement for maintaining extensive log records.

Machine learning models made from these features were moderately successful. The best performance being an ensemble of models over 92% correct overall. However, there is clearly room for improvement. This present work provides a framework for exploring this problem further.

The tools developed were designed to be reused and proved themselves to be very robust in practice. There are other potential applications that could be contemplated. Using machine learning to improve system provisioning is one area that might lead to some very effective applications.

The remaining challenges are to get the latency down to a reasonable level for online classification via Apache mod_ml - this might be tricky as there are two possible bottlenecks: database access and classification complexity. There are a number of ways to mitigate these issues (caching, in memory databases). The second challenge is to find a reasonable way to combine models from different machine learning frameworks. This challenge is more difficult as most machine learning platforms are not designed to interoperate with each other.

7. Glossary

The Witten et al. references are to:
Witten, I., Frank, E., Hall, M. (2011). Data mining: Practical machine learning tools and techniques, 3rd Edition. Burlington, MA: Morgan Kaufman.

Apache

Is a popular web server with an extensible, modular design.
Reference: http://httpd.apache.org/

BFTree

Or “Best First Tree” is an example of an ensemble tree building method. Multiple tree stumps are built using a relatively simple regression scheme and added together. The regression models are combined such that more successful ones are given more weight.
Reference: http://weka.sourceforge.net/doc.stable/weka/classifiers/trees/BFTree.html

J48

Is an implementation of the C4.5 algorithm which takes a “divide and conquer” approach to tree building. The C4.5 algorithm attempts to maximize mutual information when deciding on an attribute to split on at any given node. C4.5 uses a relatively simple error estimation scheme based on a user given confidence limit to identify when it needs to prune: either eliminate or replace a subtree.
Reference: Witten et al. p 193

Random Forest

Is another ensemble method that creates multiple trees from subsets of features and subsets of the training data (typically ⅔ is used for each tree and replaced)
Reference: https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm

Simple Cart

The Simple Cart, or “Classification and Regression Tree,” algorithm is like J48/C4.5 except that it uses a different method to prune the tree. It uses *cost complexity* to evaluate what subtrees can be eliminated. Cost complexity combines the size of the candidate for elimination and the error reduction produced by that tree. Many solutions are tested against held out data.
Reference: Witten et al. p 202

Vowpal Wabbit

Is a popular, very versatile machine learning toolkit it can be used from the command line or operated as a server. It encapsulates an enormous number of different machine learning techniques in one package and can be used on stand alone systems or in a distributed environment.
Reference: https://github.com/JohnLangford/vowpal_wabbit/wiki

Weka

Is another popular machine learning toolkit. It distinguishes itself with a convenient to use GUI environment, database integration and the ability design and conduct experiments.
Reference: http://www.cs.waikato.ac.nz/ml/weka/