The Apache mod_ml module

Introduction

The mod_ml Apache module is a server side Apache interface that acts as an intermediary between Apache request processing and external machine learning tools. The mod_ml module is intended to simplify the process of gathering and acting on request data for machine learning applications.

The motivation for creating the mod_ml interface relates to the realization that processing large amounts of log data after the fact is excessively resource intensive and time consuming. Resources are consumed when log data is generated and stored to disk - possibly using network resources as well. In addition, to be useful, data in machine learning applications must be cleaned and preprocessed consuming more resources and time. In some cases the quality of the log data does not match that of the original request - during request processing the request time has millisecond precision, typically lost when the request is logged.

A second motivation is that data classified “on the fly” is more useful than data classified after the fact. HTTP requests classified via mod_ml can be acted on immediately at the time of the request. The request could be modified by mod_ml by adding cgi data, headers or environment variables. The request could be redirected after modification by mod_rewrite. Finally the classification output could be forwarded to external processes to alter system behaviour. An example of the latter approach might be an external service that works with a proxy server to spin up Amazon EC2 instances based on predicted demand and instance cost. The rules for when to add or remove instances could be modified on the fly based on a reinforcement learning scheme.

From the perspective of mod_ml, data analytics tasks can be roughly grouped into four stages:

1. data cleanup and formatting - can be done internally with regular expression processing or via external services
2. data analysis for model creation or logging (preprocessing) - done via external services
3. data analysis for classification (classification) - done via external services
4. the server response to classification (class response) - the server can be set up to change the request, redirect it or invoke external services based on classification results

For stage 1 the inputs are fields. These are possibly cleaned and concatenated into a feature string. The feature string can be formatted by mod_ml in a variety of ways via the MLOutFormat directive.

Stage 2 takes the feature string and sends it to an external process for preprocessing. This happens during the logging phase of Apache request processing. The external services that manage the actual preprocessing step can be network services (identified by the ip:port to access), unix sockets (identified by the socket path) or a program path to be started by mod_ml when the Apache child process starts. During operation these services accept formatted feature strings from mod_ml. The external processes may or may not send responses as mod_ml ignores these at this stage.

Stage 3 takes the same feature string and sends it to an external process for classification. In this case the response produced can be used by mod_ml to change the request or initiate an external action. The response is typically a simple string identifying the class of the input on some meaningful dimension.

Stage 4 depends on the class string produced by the classification step. At this stage mod_ml can modify the request or feed data to an external process. Data sent to an external process can consist of the class, feature string or both. Responses received at this stage may be logged but are not acted on further.

Installation and configuration

The mod_ml module was written to work with Apache 2.4 servers. It may not work with earlier 2.x servers and will certainly not work with 1.3 servers. See http://httpd.apache.org/ for general information on Apache installation and configuration.

The first step to install an Apache module is to copy the built module to the location where apache modules are stored. Look in the Apache configuration to see where this directory is when copying by hand.

The next step is to activate the module in Apache. In Centos, Fedora and RedHat systems, activate the mod_ml module by adding a LoadModule directive in httpd.conf. For Debian and Ubuntu systems put the LoadModule directive in an *ml.load* file to the */etc/apache2/mods-available* directory and create a link to this file in the *mods-enabled* directory. The directive should look like this:

```
LoadModule ml_module /path/to/apache/modules/mod_ml.so
```

Lastly, add the mod_ml Directives described elsewhere in this document to configure mod_ml's behaviour. Directives for mod_ml can be placed in the main server configuration in httpd.conf or in directory configuration files such as .htaccess. Directives in per directory configuration take precedence over Virtual Hosts which take precedence over the top level server configuration. In each case the lower level completely replaces the upper level directives.

For Ubuntu the top level configuration is put in a separate configuration file (e.g. *ml.conf*). This file should be saved in */etc/apache2/conf-available* and then linked in *conf-enabled*. Ubuntu has a number of tools to automate this process such as *a2disconf*, *a2enconf*, *a2dismod* and *a2enmod*.

Directives for mod_ml configuration

The mod_ml module provides the following Apache directives. All directives are allowed in *.conf anywhere and in .htaccess when AllowOverride isn't set to None. The directives should be grouped in the following order:

```
<IfModule ml_module>
        MLEnabled on

        mod_ml feature configuration directives:

        MLDefFieldProc, MLFieldProc, MLOutProc and MLOutFormat directives

        a processor group consisting of (in order):

        MLFeatures, MLVars, MLOutProc and MLOutFormat directives
        MLClassResponse directives
        MLClassifier or MLPreprocess directives

        [ more processor groups ]
</IfModule>
```

The directives in detail:

MLEnabled
Syntax:
MLEnabled {on|off}

Enable or disable mod_ml. Other directives will be read but no processing will be done if set to “off.”

MLDefFieldProc and MLOutProc
Stage: data clean up
Syntax:
MLDefFieldProc|MLOutProc {regex|proc|sock|ip} {string}

Default processor for cleaning field data. Data is processed through this field processor if there is no other field specific mapping.
MLOutProc works in the same way as MLDefFieldProc but it is applied the concatenated feature string before it is sent off to be processed.

The processor can be

- regex - apply a regular expression to the field data internally
- proc - send field data to an external process started by mod_ml; the processed feature is expected to be returned
- sock - send field data to a unix socket; as with proc the mogrified feature is expected as the return value from the socket
- ip - send field data to an external network service; cleaned field data is expected to be returned

Notes:

- regexes must use m// with () captures or s/// to work.
- at the moment there is no password / nonce protection for network services or sockets - i.e. mod_ml won't send a nonce with the data
- in general it is expected that most applications will use only server local services for this type of processing given that it may be repeated many times for each request

MLFieldProc
Stage: data clean up
Syntax:
MLFieldProc {regex|proc|ip|sock} {processor} {cgi|uri|header|env|cookie|time|request} {field1 field2 ...}

Field/processor map. As with MLDefFieldProc raw data from a field is processed through one processor. Only one field cleaner can be mapped to a given field at the moment. The regex, proc, sock or ip all work the same as with MLDefFieldProc.

Fields types are identified by one field group identifier per directive:

- cgi - look for the field in cgi data - post or get variables
- uri - look for the field in the parsed uri data (options: *fragment, hostinfo, hostname, port, path, password, query, scheme, user*)
- header - look for the field in the http request headers
- env - look for the field in the subprocess environment or general environment variables
- time - look for the field in the request time. (options: *epoch (seconds), micros (epoch in microseconds), millis (epoch in milliseconds), ctime (string format), rfc822 (string format), or any valid strftime format*) OR the current wall clock time (options: *now (microseconds), elapsed [= now - micros]*)
- cookie - look for the field in a HTTP cookie

- request - look for the field in the HTTP request_rec struct itself (options: *args, content_type, content_encoding, filename, handler, hostname, method, protocol, path_info, range, status_line (e.g. "200 OK"), the_request, unparsed_uri, uri, vlist_validator*)

Use multiple directives for multiple field types. While fields are normally identified by how they are named in their field group and alternate name can be specified with an = sign as illustrated here: *myname=fieldname*. This custom name is used in the jsonfields output format for the name of the field.

MLFeatures
Stage: data formatting
Syntax:
MLFeatures {cgi|uri|header|env|cookie|time|literal|request|auth} {field1, field2 ...}
...

Field names of features in the order they should have in the field string. Multiple MLFeatures directives can be used to concatenate groups of fields together. The order of the MLFeatures directives as well as the order of the fields within a specific MLFeatures directive determines the order of the fields in the output string. Features and labels are associated with specific preprocessors, classifiers and class responders as defined by *MLPreProcess, MLClassifier* and *MLClassResponse* directives.

Apart from the field groups described for MLFieldProc, MLFeatures also supports two other pseudo field groups:

- literal - used to insert a literal string into a field string. For example, in vowpal wabbit the | character is important for identifying labels.
- auth - this is used to specify a password to be sent with the field data to preprocessing, classification or class response scripts. Multiple auth fields can be specified. These are always transformed into "nonces" (short for "number used once") in the form: *nonce=sha1(sha1(password)):salt*. The first half of the pair is the sha1 hash of a sha1 hash of the password plus the salt. The second half is the salt. For example:

```
nonce=60750304db2314d6c2aa234568ff2e66bda7562a:c58de859b5fd2103f32deea7fa9aa579c87ebca5
```

External services are expected to use the salt and their own copy of the password to recreate the feature string.

MLVars
Stage: data formatting
Syntax:
MLVars {cgi|uri|header|env|cookie|time|literal|request|auth} {field1, field2 ...}

One or more extra variables. These are not sent to the preprocessor or classifier but can be used to modify a preprocessor, classifier or class response IP process.

For example:
```
MLFieldProc proc /usr/local/micros2ip.pl time mylatestip
MLFieldProc proc /usr/local/micros2port.pl time mylatestport
MLVars time mylatestip=now mylatestport=now
MLClassifier ip %{time:mylatestip}:%{time:mylatestport}
```
would take the current wallclock time and use it to determine which IP address and port to send the feature string to. This allows for in-module load balancing based on variable data. Variable syntax is of the form *%{vartype:varname}*.

Features and vars are associated with specific preprocessors, classifiers and class responders as defined by *MLPreProcess, MLClassifier* and *MLClassResponse* directives.

MLSendLength
Stage: data formatting
Syntax:
MLSendLength {on|off}

Whether or not to send the length of the feature string before sending the feature string.
For example:
```
2237 "field1","field2", .... (up to 2237 characters)
```

This directive can be turned on and off for different classifiers and preprocessors as needed. Default is for the length *not* to be sent.

MLOutFormat
Stage: data formatting
Syntax:
MLOutFormat {raw|quoted|csv|jsonarray|jsonfields}

How to format the feature string:
- raw: no extra formatting is done beyond feature cleanup (see the *FieldProc directives)
- quoted: like raw but each feature is put in quotes (quotes in the field are escaped)
- csv: quoted list of features separated by commas
- jsonarray: unlabelled, quoted features in an array (e.g. ["fieldval1", "fieldval2", …])
- jsonfields: labelled, quoted features in a dictionary (e.g. { "name1": "val1", … })

MLPreProcess
Stage: preprocessing / model building / logging
Syntax:
MLPreProcess {sock|ip|proc} {action}

Process that creates model or does basic data intake. Any preprocessing happens during logging which is the final stage of Apache request processing as such mod_ml does not use returned output from this stage. Also, preprocessing will not affect response time for a specific request as the request has already been completed.

The sock, ip and proc the same as with the *FieldProc directives except that they are sent the entire feature string including any auth fields. Use *MLSendLength* to prepend the length.

The {action} is expected to be a single host:port, socket or path. If a command needs to be run with arguments, encapsulate it in a script that can be run without arguments. For IP based actions use *MLVars* with a *MLFieldProc* directive to dynamically set the IP address or port. This works for for any IP based classifier, preprocessor or class response directive.

Multiple MLPreProcess directives can be specified. These are run in the order they are found in the Apache configuration. The *MLFeatures* and *MLVars* directives before the *MLPreProcess* directive determine the labels and features that get sent to the preprocessors.

MLClassResponse
Stage: server response to classification
Syntax:
MLClassResponse {class} {cgi|uri|header|cookie|env|http|ip|proc|sock} {action [format string]}

What to do based on classification (each directive applied in order seen). These directives are associated with *MLClassifier* directives that follow them.

The {class} in the directive is the expected response from a classifier. If this exactly matches then the action specified is initiated.

The {action} can either be a *name=value* pair for the cgi, header, cookie and env action types or it can be the name of a script, host:port pair or unix socket. In this directive the action can have an argument, a format string. The format string can contain the special formatters %C or %F. %C is replaced by the class value and %F is replaced by the feature string. The format string is processed and sent to the external service as with the MLClassifier and MLPreProcess directives. The “HTTP” action type should be used as a last response for any group of class responses as this sends the numeric HTTP response code to apache ending processing.

For ip, proc and sock action types, the *MLFeatures* and *MLVars* directives before the *MLClassResponse* directives determine the labels and features that get sent to the class responders.

MLClassifier
Stage: classification
Syntax:
MLClassifier {sock|ip|proc} {string}

Process that makes decisions based on field input. This directive operates the same way as MLPreProcess except that it happens right before content handling (i.e. the content generation phase of Apache's HTTP request processing model).

Multiple MLClassifier directives can be specified and will be applied in the order they are seen but only the last class value returned will be used by mod_ml. The *MLFeatures* and *MLVars* directives before the *MLClassifier* directives determine the labels and features that get sent to the classifiers.

SetHandler ml

Use this directive for debugging. Normal page output will be replaced by debug output from mod_ml. See the Appendix on building mod_ml with debug information for more information on how to turn on debug features.

To get extended logging output use (in the main Apache configuration):
```
LogLevel trace1
```

Example configuration

The following is an example configuration showing how the directives work together in Apache:

```
<IfModule ml_module>
       MLEnabled on

       # doesn't like /g and \s are stripped from control characters
       MLDefFieldProc regex "s/\s*$//s"

       # remove the leading zero from hour, grab the hour from the time using strftime
       MLFieldProc regex "s/^0*//" time hour=%H

       MLFeatures time epoch hour=%H
       MLFeatures env REMOTE_ADDR
       MLFeatures env REQUEST_URI HTTP_HOST
       MLFeatures request status_line
       # the following will add a "mynonce": "pw:salt" string to the feature string
       # MLFeatures auth mynonce=somesecret

       # the user agent will be prepended to the feature string "useragent":"..."
       MLFeatures header useragent=User-Agent

       # output as key value pairs using json format
       MLOutFormat jsonfields

       # what to do if we see YES … lots apparently

       # this env variable is read by mod_rewrite
       MLClassResponse "YES" "env" "client=bot"
       # add a cgi field client=bot
       MLClassResponse "YES" "cgi" "client=bot"
       # similarly add a cookie - cookie must be in the form of name=value;
       MLClassResponse "YES" "cookie" "client=bot;"
       # make a HTTP header
       MLClassResponse "YES" "header" "client=bot"
       # then send class and feature string to these services
       MLClassResponse "YES" "proc" /usr/local/bin/caps.pl {"%C": %F}
       MLClassResponse "YES" "sock" /tmp/usock.sock {"%C": %F}
       MLClassResponse "YES" "ip" localhost:7777 {"%C": %F}
       # finally send a HTTP response - be careful, this will derail the response
       # MLClassResponse "YES" http 500

       # send the feature string to this classifier during the
       # fixups phase right before content generation
       MLClassifier proc /usr/local/bin/y.pl
       # send the feature string to this web service during the logging phase
       MLPreProcess ip cloudsmallN:39992

       # more processor groups ...

       # if this is set we will see the mod_ml stats instead of the page
       # SetHandler ml
</IfModule>
```

A feature string produced by this might look like this:

```
{"useragent":"Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:33.0) Gecko/20100101 Firefox/33.0",
"epoch":"1437481437361", "hour":"12", "REMOTE_ADDR":"45.74.2.131",
"REQUEST_URI":"/ottawa.craigslist.ca/ths/5108776913.html", "HTTP_HOST":"206.12.16.219:8898",
"status_line":"200 OK"}
```

The *{"%C": %F}* format string makes a json dictionary like this:

```
{"YES":{"User-Agent":"Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:33.0) Gecko/20100101
Firefox/33.0","epoch":"1437481437361","hour":"12","REMOTE_ADDR":"45.74.2.131","REQUEST_URI":"/
ottawa.craigslist.ca/ths/5108776913.html","HTTP_HOST":"206.12.16.219:8898","status_line":"200
OK"}}
```

Appendix: Debug and cross platform support

When the module is built, extra debug logging can be turned on by setting the ML_DEBUG C language define equal to 1. The apxs build command syntax:

```
sudo apxs -D ML_DEBUG=1 -o mod_ml.so -i -c mod_rewrite_funcs.c mod_ml.c
```

or to turn off debug information

```
sudo apxs -D ML_DEBUG=0 -o mod_ml.so -i -c mod_rewrite_funcs.c mod_ml.c
```

Starting the Apache server with ML_DEBUG=1 will cause a group of debug statements to be sent to standard output. These are output whether or not MLEnabled is set to "on." Similarly, the error log will show a large number of statements with each request showing the system state during the request. This much logging will slow processing down. Production systems should be built with ML_DEBUG=0.

The mod_ml module was developed and tested under Ubuntu Linux version 14.04 with Apache 2.4.7 and 2.4.16.