

User's Guide for 'Baywatch'

Michael J. Fromberger¹
<sting@cs.dartmouth.edu>

Version 2.5.3—May 30, 2004

This document describes Version 2 of the **Baywatch** Bayesian spam filtering program. This version represents a fairly significant rewrite of the previous version (which was numbered 1.3). You can download the latest version of the program from <http://www.dartmouth.edu/~sting/sw.shtml>. Version 1 was a Perl² 5 script; Version 2 is written in Python³.

The purpose of Baywatch is to act as a filter on incoming e-mail. Using a simple probabilistic technique, Baywatch tags each message as belonging to one of two categories—*good* mail, the stuff you want to see, and *spam*, the annoying unsolicited bulk commercial mailings that seem to be clogging up everybody's mailbox these days. As I write this, I now receive almost thirty spam messages each day. That doesn't sound too bad, until you realize that in the past seven months, that totals more than *six thousand* spam messages! Needless to say, I get many fewer legitimate messages than that! Thus, as I'm sure you can imagine, I am eager to keep the spam away from my mailbox.

What *is* spam, exactly? It is difficult to define spam universally—if nothing else, we are soon faced with the problem that one man's trash is another man's treasure, as the saying goes. But, as a Supreme Court Justice named Potter Stewart once said about obscenity, "I know it when I see it." So rather than trying to define spam in a rigorous and concrete manner, Baywatch works on the principle that *you* know which messages are spam just by inspection.⁴ By taking advantage of your own judgement about what's a good message, and what's not, Baywatch can help you filter out the trash.

1 Requirements

The Baywatch distribution consists of the following files:

¹Copyright © 2003–2004 Michael J. Fromberger, All Rights Reserved. Permission is hereby granted, free of charge, to any person obtaining a copy of this document, to use, copy, merge, publish, and distribute the document without limitation, subject to the following condition: This copyright notice and this permission shall be included in all copies or substantial portions of the document.

²<http://www.perl.org/>

³<http://www.python.org/>

⁴For an excellent article on this idea, see <http://paulgraham.com/spam.html>.

File	Description
<code>bay</code>	The main driver program for testing and training.
<code>baydb</code>	An tool to help manipulate the database of words.
Baywatch	A module containing the reusable guts of the program.
<code>data.py</code>	– An interface to the on-disk database of words.
<code>message.py</code>	– Routines for parsing e-mail messages.
<code>stats.py</code>	– Code to analyze and categorize messages.
<code>utils.py</code>	– Various utility subroutines.
<code>_version.py</code>	– An internal module.

In addition to these files, Baywatch also requires:

1. Python 2.3 or newer.⁵
2. The `gdbm` module. This is distributed with Python by default, but it is not built unless GDBM⁶ is installed on your system.
3. The `base64`, `bz2`, `cPickle`, `email`, `gzip`, `pwd`, `quopri`, `re`, and `tempfile` modules. All of these are part of the Python standard distribution.
4. A custom option-parsing library called `altopt`, which is available from <http://www.dartmouth.edu/~sting/sw.shtml>.
5. A collection of messages you consider to be “good” (*i.e.*, not spam). We will call this collection the **good corpus**. Baywatch assumes you have these messages saved in Unix mailbox format.
6. A collection of messages you consider to be spam. We will call this the **spam corpus**. Again, the program assumes the messages are in Unix mailbox format.

2 Installing Baywatch

To install Baywatch, copy the `bay` and `baydb` scripts to a directory that is part of your executable path (on a Unix system, you can add to this list by editing the `PATH` environment variable). You *may* need to edit the first line of each of these scripts, to invoke the correct version of the Python interpreter.

If you are using a Unix shell, or something similar, you can find out what version of Python you are running by typing:

```
% python -V
```

You can find out where it’s installed using:

```
% which python
```

You will also need to copy the Baywatch directory and `altopt.py` to some location where Python can find them. One easy solution is to put them in the same directory where you install `bay` and `baydb`.

⁵It is quite likely that Baywatch will work with Python 2.2, and possibly even earlier versions as well, but I have not tested it under any versions prior to Python 2.3b1.

⁶<http://www.gnu.org/software/gdbm/>

Alternately, Python has a special directory for local custom packages, located near where the interpreter itself is installed, *e.g.*:

```
/sw/lib/python2.3/site-packages, or  
  
/usr/local/lib/python2.3/site-packages
```

As long as the interpreter can find these libraries when required, there is nothing special about where you put them. You may find the PYTHONPATH environment variable to be useful in this regard.

2.1 First-Time Setup

Before Baywatch will be useful as a spam filter, you need to provide it with some training data. Training data consist of a collection of messages which you have passed judgement on—some good, some spam. The bigger and more diverse the body of messages you use to train the filter, the more likely the filter is to correctly categorize future messages.⁷ The program will extract the words from these messages and create a database of statistics (actually, two such databases), which it will later use to figure out whether some new message—hitherto unseen—is likely to be spam or not.

If you have files named `gm1`, `gm2`, and `gm3` containing ‘good’ messages, run:

```
bay learn -good gm1 gm2 gm3
```

For files named `sm1`, `sm2`, and `sm3` containing spam examples, run:

```
bay learn -spam sm1 sm2 sm3
```

These options may be combined to load both sorts of files at once, as in the following example:

```
bay learn -spam sm1 sm2 -good gm1 -spam sm3 sm4 -good gm2
```

The file names following the `-good` option will be assumed to be ‘good’ messages, while those following the `-spam` option will be assumed to be ‘spam’ messages. You can get some feedback while the program is working by adding the `-verbose` or `-v` option, *e.g.*:

```
% bay learn -spam badstuff -verbose  
[processing spam messages from file badstuff]  
- msg 1 [body] 49 words 49 new, [head] 21 words 21 new  
- msg 2 [body] 56 words 42 new, [head] 21 words 11 new  
- msg 3 [body] 107 words 90 new, [head] 14 words 8 new  
- msg 4 [body] 128 words 105 new, [head] 11 words 4 new  
- msg 5 [body] 155 words 103 new, [head] 14 words 2 new  
- msg 6 [body] 130 words 79 new, [head] 13 words 5 new  
6 messages processed.
```

Once you have trained the program, you don’t need to keep the training messages around anymore; all the relevant information is kept in a pair of GDBM files in your home directory. Of course, you will want to make sure you have good backups of that information, since the accuracy of the filter depends on it.

⁷Actually, this is an assumption of the naïve Bayesian categorization algorithm; it works well in practise, but is not a proven mathematical fact.

Note:

Files compressed by `gzip` or `bzip2` can be read directly by `bay` for training or testing purposes. If a file name is given which ends in `.gz` or `.bz2`, it will be uncompressed on the fly as it is read.

2.2 Testing New Messages

Once you have trained the filter, you can direct Baywatch to test a new message for the likelihood that it is spam. If your new message is in a file named `message.txt`, you can test it by running:

```
bay test message.txt
```

When you do this, `bay` reads in each message from the file, analyzes it, and writes it back out to the standard output. To each message, it adds a header similar to the following:

```
X-Bay-Status: REJECT; PS="1.00000"; PG="0.00000"; TW="23"; HW="8"
```

The first field is the **status** of the message. It can take on the values `ACCEPT`, `IGNORE`, or `REJECT`. The second field (`PG`) is the computed probability that the message in question should be considered spam. The third field (`PP`) is the computed probability that the message in question should be considered ‘good’. For a fairly detailed description of how these are computed, you can read the article “Bayesian Classification of Unsolicited E-Mail,” which accompanies the Baywatch distribution. But for the sake of intuition, the basic procedure is as follows:

1. The value of `PG` is computed by examining the message sender, the return path, and the Subject line.
2. The value if `PS` is computed by examining the message body (including any text-based attachments), along with the Subject line.
3. If `PG` is sufficiently high, the message is *white-listed*, and given a status of `ACCEPT`.
4. If not, but if `PS` is sufficiently high, the message is *black-listed*, and given a status of `REJECT`.
5. Otherwise, the message is given a status of `IGNORE`.

The `TW` and `HW` fields give a count of how many “words” were encountered in the text (`TW`) or headers (`HW`) of the message.

3 Integration with Procmail

By itself, Baywatch doesn’t actually *separate* your e-mail, it merely marks it as being good or spam. But once this has been done, you can use this marking to determine the disposition of the message using other tools. My preferred method of doing this is to use `procmail`.

Here is a simple cookbook recipe for setting up Baywatch with `procmail`:

```
# Make sure the bay program is in your path
PATH=$PATH:/home/u/user/scripts
```

```

# This first recipe removes any pre-existing X-Bay- headers from the
# message. This will prevent someone from tricking your filters by
# adding their own versions of these headers.
:0 Hfhw
* ^X-Bay-.*
|formail -I "X-Bay-Status:" -I "X-Bay-Restrict:"

# Now, pass everything through the spam filter
# The - argument tells bay to read messages from standard input.
:0 fw
|bay test -input -

# Now you can select messages based on the X-Bay-Status tag.
:0:
* ^X-Bay-Status: REJECT;.*
spamtrap

# ... other filters ...

:0:
* ^X-Bay-Status: ACCEPT;.*
INBOX

```

One important note about using Baywatch with `procmail`. It is necessary that your database files be located somewhere they can be read by the system while delivering e-mail. In a typical Unix environment, this is not a big issue, since the mail system usually runs as a super-user, and mail is stored in a local directory. However, on some systems (notably AFS), the local system's super-user does not inherently have permission to access the contents of a shared volume.

Because of the way GDBM deals with I/O synchronization, even when you are only opening the database for read access, GDBM needs the ability to write to the database file itself, to insure mutual exclusion. If this does not work in your environment, you can turn off synchronization of readers using the `-unsync` option, *e.g.*:

```
bay test - -unsync
```

This is primarily a concern for the `test` subcommand, which runs unattended most of the time. There is some danger that if you run a test unsynchronized, while the database is being updated, that the reader might obtain inconsistent results (this could happen if you are adding new training samples while messages are arriving at the mail server, for instance).

4 Reference

This section describes all the options understood by the Baywatch tools. Any option name can be abbreviated to the shortest unique prefix of that name among all the options supported for a particular command. Where file names are required, the name `-` stands for the standard input. By default, output is written to the standard output unless otherwise directed.

The general format for each command is:

```
command <subcommand> [options/args]
```

The `<subcommand>` specifies which of a set of different possible actions the program should take. Option switches are given by strings beginning with a hyphen (*e.g.*, `-option`) and some take one or more additional arguments. Any argument that is not an option switch or one of its corresponding arguments is considered a “free” argument, and is passed to the program normally.

Options and free arguments may be mixed in any order, except that the arguments belonging to an option switch must immediately follow the switch itself.

4.1 Bay

The `bay` script is the primary front end for Baywatch. It supports the following subcommands:

learn

This permits you to add new training examples to the database. You will want to do this on occasions when you discover that the program has guessed incorrectly about the disposition of a new message.

Basic usage:

```
bay learn [-good files] [-spam files] [-verbose]
```

forget

This is the natural opposite of `learn`. If you make a mistake, or decide you no longer consider a message a good example, you can withdraw it from the training sample using this command.

Basic usage:

```
bay forget [-good files] [-spam files] [-verbose]
```

Note: Baywatch does not actually “know” which messages it has been trained on. You can tell it to forget messages it was never actually trained with in the first place. The `forget` subcommand simply causes the word frequencies of the message to be subtracted from the totals in the database. If the message *was* in the training sample, this has the effect of dropping that message from the sample.

test

This command is the interface for analyzing new messages against the existing data. It is this command that reads in and marks up messages with the `X-Bay-Status:` headers.

Basic usage:

```
bay test input-files [-unsync] [-verbose]
```

Note: The file name `-` can be used to indicate the standard input.

help

This command gives you a quick way to obtain a synopsis of the valid command-line options supported by the rest of the commands.

Basic usage:

<code>bay help</code>	Print a synopsis of available subcommands.
<code>bay help <i>commands</i></code>	Print a list of options for the specified subcommands.

The `learn` and `forget` commands support the following options:

-bodyonly

Only consider the words in the message body; do not look at the sender or return path headers.

This is useful if you get a message that would ordinarily be considered spam, but it comes from (say) a friend or a client. You can train the body as a spam message, and separately train the headers as good (see the `-headonly` option below).

-headonly

Only consider the sender, the return path, and the subject line; do not look at the words of the message body.

See also the `-bodyonly` option, above.

-verbose

Print out diagnostic information about each messages as it is absorbed into (or removed from) the database.

-count <num>

Process at most this number of messages from the input files.

-good <files>+

Specify one or more files of “good” training examples.

-skip <num>

Skip this many messages from the beginning of each input file.

-spam <files>+

Specify one or more files of “spam” training examples.

The `test` command supports the following options:

-bodyonly

Analyze only the text of the message body and the Subject line, not the other headers. With this option, the output message may have a status of `IGNORE` or `REJECT`, but never `ACCEPT`. See also `-headonly`.

-discard

Discard output. Ordinarily, output is written to the standard output; this option is useful for debugging. See also the `-select` option.

-filter

Select messages which match the given filter expression. See also the **-select** option. Selected messages are written to the output, unless **-discard** is specified.

-headonly

Analyze only the headers and subject of the message, not the message body or attachments. With this option, the output message may have a status of **ACCEPT** or **IGNORE**, but never **REJECT**. See also **-bodyonly**.

-normalize

Normalize frequency counts by dividing through by the size of the training corpus for good vs. spam messages, temporarily overriding the setting on the database. This is the default behaviour for newly-created databases, but it can be changed (see the section on **baydb** below).

-unnormalize

Do not normalize frequency counts; compute all frequencies as an absolute fraction of total words, rather than scaling. This option (temporarily) overrides the setting on the database.

-unsynced

Do not synchronize read-only accesses to the database with other users. This is sometimes necessary if the database lives on a network file system which is not writable by the process that is opening the database. Avoid this when possible.

-verbose

Print out diagnostic output. This option may be repeated multiple times for more detailed output.

-count <num>

Test at most the specified number of input messages from each file.

-input <file>

Specify an input file. The free arguments of the command are also taken as input file names.

-select <types> ...

Usually, every message that is read in is copied to the output. However, the **-select** option overrides this, causing only the messages with the specified tag or tags (**ACCEPT**, **IGNORE**, and **REJECT**) are sent to the output. This is useful for picking wrongly classified messages out of a file for separate training, *e.g.*:

```
bay test -select reject > oops
```

See also the **-filter** option. If both **-select** and **-filter** are specified, all the matching messages will be output.

-skip <num>

Skip this many messages from the beginning of each input file.

If the name of any input file ends in **.gz** or **.bz2**, then **bay** will attempt to treat the file as if it is compressed with **gzip** or **bzip2** respectively. Thus, you do not need to decompress these files in order to test them or learn messages from them.

4.2 BayDB

The `baydb` script helps you maintain your word databases. It understands the following commands:

export

Dump the specified database out into a plain text file, so that it can be ported to other systems. Binary database formats like GDBM are fast, but they don't work across platforms.

Basic usage:

```
baydb export <head|body> -output filename
```

```
baydb export -dbpath file -output filename
```

flush

Discard all the entries from the specified database. This can be used to “start over” if you want.

Basic usage:

```
baydb flush <head|body>
```

```
baydb flush -dbpath file
```

import

The natural reverse for `export`, this command loads in entries from a flat text file of the format written by the `export` command.

Basic usage:

```
baydb import <head|body> [-flush] [-input file]
```

```
baydb import -dbpath file [-flush] [-input file]
```

lookup

This command permits you to look up information about words in the database. It prints out a synopsis of what it knows, for instance:

```
% baydb lookup body this that these
Word          freq  fspam  fgood  asпам  agood  pspam  pgood
that          4656   813   3843   2.07   2.03  0.5044  0.4956
these         439    225    214   0.57   0.11  0.8350  0.1650
this          3393  1809  1584   4.60   0.84  0.8460  0.1540
```

For each word, `freq` is the total number of times that word has occurred in the training corpus; `fspam` is the number of times it occurred in spam messages, and `fgood` is the number of times it occurred in good messages. The `asпам` and `agood` columns list the average number of times the word appeared per spam or good message; while `pgood` and `psпам` are the probabilities, given that this word appears, that the message is good or spam, respectively.

Words which did not occur in the training sample are shown with a frequency of 0.

Basic usage:

```
baydb lookup <head|body> words
```

The `lookup` command supports the following options:

-all

Display all the words in the database. This is the default behaviour if no words are given as free arguments on the command line. But, see also the **-filter** option.

-delete

If the **-filter** option is used, this causes each word that is selected by the filter to be deleted after it is displayed. There is no ‘undo’ capability, so be careful.

-normalize

This has the same function as the **-normalize** option of **bay test** (see above).

-unnormalize

This has the same function as the **-unnormalize** option of **bay test** (see above).

-dbpath *file*

Specify an alternate database file to use.

-filter *filter*

Specify the filter expression to use (see below).

-words *words ...*

Give specific words to be looked up.

setparam

Each of the two databases contains several configuration values which can be set using the **setparam** command of **baydb**. These include:

Parameter	Description
bias	The probability assigned to novel words ($0 \leq bias \leq 1$).
inset	The probability assigned to words that only appear in ‘good’ messages ($0 < inset \leq 1$).
interest	How many words to consider (0 = consider all).
normalize	Whether to normalize frequencies (0 = no, 1 = yes).
threshold	The probability cutoff for category inclusion ($0 \leq threshold \leq 1$).
wordbreak	The regular expression used to break up text into words (as a string).
wordlen	The maximum word length allowed (0 = no limit).

To set a parameter, you specify the database, the name of the parameter, and its new value:

baydb setparam <head|body> *parameter value*

For example:

baydb setparam body interest 25

showparam

Show the values of database parameters (see **setparam**).

Basic usage:

baydb showparam <head|body> [-param *names*]

baydb -dbpath *file* [-param *names*]

4.3 Filter Expressions

The `lookup` subcommand of `baydb` and the `test` subcommand of `bay` allow you to specify “filter expressions” to select certain words or messages out of the database (in the case of `lookup`) or the input (in the case of `test`). These filter expressions consist of a set of simple comparisons and Boolean connectives. The following properties of a word are understood for `lookup`:

Name	Description
<code>freq</code>	The total frequency of the word in the corpus.
<code>fspam</code>	The frequency of the word among the spam messages.
<code>fgood</code>	The frequency of the word among the good messages.
<code>aspm</code>	The average frequency of the word in a spam message.
<code>agood</code>	The average frequency of the word in a good message.
<code>pspm</code>	The probability that a message containing this word is spam.
<code>pgood</code>	The probability that a message containing this word is good.
<code>length</code>	The length of the word (in characters).
<code>word</code>	The text of the word itself.

For the filter expressions supported by the `test` subcommand of `bay`, only the `pspm`, `pgood`, and `status` fields are understood. The `status` parameter is the disposition string assigned by the tester (*i.e.*, `ACCEPT`, `IGNORE`, or `REJECT`).

In addition, the following comparison operators are understood by all filter expressions:

Operator	Description
<code>a < b</code>	True if <i>a</i> is less than <i>b</i> .
<code>a <= b</code>	True if <i>a</i> is less than or equal to <i>b</i> .
<code>a > b</code>	True if <i>a</i> is greater than <i>b</i> .
<code>a >= b</code>	True if <i>a</i> is greater than or equal to <i>b</i> .
<code>a = b</code>	True if <i>a</i> is equal to <i>b</i> .
<code>a <> b</code>	True if <i>a</i> is not equal to <i>b</i> .
<code>a != b</code>	True if <i>a</i> is not equal to <i>b</i> .
<code>a like b</code>	True if string <i>a</i> matches the regular expression <i>b</i> .
<code>a unlike b</code>	True if string <i>a</i> does not match the regular expression <i>b</i> .

These may be combined using the logical operators `and`, `or`, and `not`, and parentheses may be used to enforce alternative orders of precedence. Strings may be specified by giving them within “double quotes”. Whitespace is not significant within an expression, although you will probably have to quote the whole expression to protect it from the shell.

A few simple examples might serve as the best instruction. Here is a filter expression that selects all the words which occur fewer than ten times in the corpus, and which are longer than 25 characters:

```
freq < 10 and length > 25
```

This one selects all the words whose spam probability is within 0.25 of 0 or 1:

```
pspm <= 0.25 or pspam >= 0.75
```

Here's a filter that selects all the words which occur exactly the same number of times in both spam and good messages:

```
fspam = fgood
```

And, finally, here's a filter that finds all the words which are at least ten characters long, occur at least 50 times overall, appear in both spam and good messages, and end in "ball":

```
len > 10 and freq >= 50 and fspam <> 0 and fgood <> 0 and word like "ball$"
```

Filters can also be used with the `test` subcommand of `bay` to select out messages with certain properties for additional training. For example, maybe you want to select all the messages whose good probability was less than 0.9, and collect them for further training. You can do this using a command like:

```
% bay test -input filename -filter 'pgood < 0.9' > outputfile
```

This is a useful technique if you have a file of messages which have been categorized correctly, but for which you feel the system could use some additional training, *e.g.*, the message was ignored by a very narrow margin. You can do this as follows:

```
% bay test -input filename -filter 'status = ignore and ps > 0.8' > outputfile
```

With `-filter`, as with `-select`, only those messages matching the given criteria are actually written to the output.

4.4 How Words are Extracted

One important question is how Baywatch identifies what constitutes a “word” in the text you ask it to analyze. Right now, the following rules determine what constitutes a word-delimiter:

1. Any consecutive run of one or more whitespace and punctuation characters are considered to delimit words, except as described in the following rules.
2. A single apostrophe (') or period (.) that occurs within a word does not delimit a word.
3. An apostrophe or period is considered to be *outside* a word if it precedes whitespace, the end of the input line, or another word delimiter, or occurs as part of a sequence of two or more apostrophes or periods.

The input is broken up into consecutive runs of characters which are separated by word-delimiters as defined by these rules, and these runs of characters are considered to be words. If the `wordlen` parameter of the database is set to a value > 0 , words longer than this value are discarded when learning new messages.

The following general procedure is used for extracting words from the body of a message:

1. Consider each part of the message whose primary MIME content type is `text`, *e.g.*, `text/plain` or `text/html`.
2. If the part is encoded with as `BASE64` or `quoted-printable`, the contents of the part are decoded.
3. The resulting text is split into words using the above rules.
4. All the resulting words are converted to lower-case, and counted; the resulting table of frequencies are used as the representation of the message body.⁸

When extracting words from the header, this procedure is used:

1. The `From:` and `Sender:` headers are extracted.
2. For each `Received` line of the envelope, except the most recent two, the imputed and canonical host names of the originating host are extracted.
3. The `Subject:` header is extracted.⁹
4. Each e-mail address is converted from `user@host` into separate `user` and `host` parts.
5. Each host name of the form `a.b.c` is converted into a list of words like `a`, `b`, `c`, `b.c`, and `a.b.c`.
6. All these pieces are converted to lower-case and counted to form a table of frequencies.

⁸This is usually called a “bag-of-words” model by the information retrieval community.

⁹The Subject is used both as part of the message body, and as part of its header.