MITSUBISHI ELECTRIC RESEARCH LABORATORIES
http://www.merl.com

A Unified Model of Spam Filtration

William S. Yerazunis, Shalendra Chhabra, Christian Siefkes, Fidelis Assis, Dimitrios Gunopulos

TR2005-085 December 2005

Abstract

A large number of spam filtering and other mail classification systems have been proposed and implemented in the recent past. This paper describes a possible unification of these filters, allowing their technology to be described in a uniform way, and allowing comparison between similar systems to be considered in a more analytic style. In particular, describing these filters in a uniform way reveals a large commonality of design and explains why so many filters have such similar performance.

MIT Spam Conference

This work may not be copied or reproduced in whole or in part for any commercial purpose. Permission to copy in whole or in part without payment of fee is granted for nonprofit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by permission of Mitsubishi Electric Research Laboratories, Inc.; an acknowledgment of the authors and individual contributions to the work; and all applicable portions of the copyright notice. Copying, reproduction, or republishing for any other purpose shall require a license with payment of fee to Mitsubishi Electric Research Laboratories, Inc. All rights reserved.

Copyright © Mitsubishi Electric Research Laboratories, Inc., 2005
201 Broadway, Cambridge, Massachusetts 02139

A Unified Model Of Spam Filtration

William S. Yerazunis[1], Shalendra Chhabra[2], Christian Siefkes[3], Fidelis Assis[4] and Dimitrios Gunopulos[2]

[1] Mitsubishi Electric Research Laboratories
Cambridge, MA, USA wsy@merl.com

[2] Computer Science and Engineering
University of California, Riverside
Riverside CA USA schhabra@cs.ucr.edu, dg@cs.ucr.edu

[3] Berlin-Brandenburg Graduate School in Distributed Information Systems[*]
Database and Information Systems Group, Freie Universitat Berlin
Berlin, Germany christian@siefkes.net

[4] Empresa Brasileira de Telecomunicações - Embratel
Rio de Janeiro, RJ, Brazil fidelis@embratel.net.br

Abstract: A large number of spam filtering and other mail classification systems have been proposed and implemented in the recent past. This paper describes a possible unification of these filters, allowing their technology to be described in a uniform way, and allowing comparison between similar systems to be considered in a more analytic style. In particular, describing these filters in a uniform way reveals a large commonality of design, and explains why so many filters have such similar performance.

Introduction

For initial convenience, we will consider the three common spam filtration styles in the current state-of-the-art – black hole listing, human-driven heuristic filtering, and machine-learning based filtering – as separate methods. Later, we will show that these methods are special cases of a more general form.

Black hole listing: determination that a site is emitting spam, and "turning off" that site by placing the site on a blacklist[i]. In extremis, this is the "Internet Death Sentence". Prime examples are the SpamCop Blocking List (SCBL)[ii] and the Composite Block List (CBL)[iii]. More recent implementations use a distributed blacklisting information system,

[*] The work of this author is supported by the German Research Society (DFG Grant nr. GRK 316)

such as John Zdzairski's peer-to-peer blacklist system.

Heuristic filtering: a human examines spam and nonspam texts for "likely features", and writes specific code to trigger on those features. These human-created features are weighted (either manually or by an optimization algorithm), and thresholded to determine the spam or nonspam nature of a document. A prime example of such a heuristic filter is SpamAssassin; other heuristic filters are used by major ISPs such as Earthlink[iv] and Yahoo[v].

Statistical filtering: a human classifies a training set of texts; a machine-learning algorithm then creates and weights features according to an internal optimization algorithm. A number of these filters have been implemented, such as D2S (Death2Spam)[vi], SpamBayes[vii], SpamProbe[viii], Bogofilter, DSPAM[ix], and the CRM114 Discriminator[x].

In this document, we will consider a unified description of these filters, and using that description, we will compare filters and consider the higher-level interactions of filters in an Internet-wide context.

Definition of Spam Filtering

To differentiate "spam filtering" from the more generalized problems of information retrieval, we'll define spam filtering as "given a large set of email readers who desire to reciprocally communicate without prearrangement with each other, and another set of email spammers, who wish to communicate with the readers who do ***not*** wish to communicate with the spammers, the set of filtering actions can the readers take to maximize their desired communication and minimize their undesired communications"[1]

In this sense we intentionally exclude "filters" such as single-use addresses, forward-computed micropayments, and referral-based systems, as those are all prearrangements of a sort.

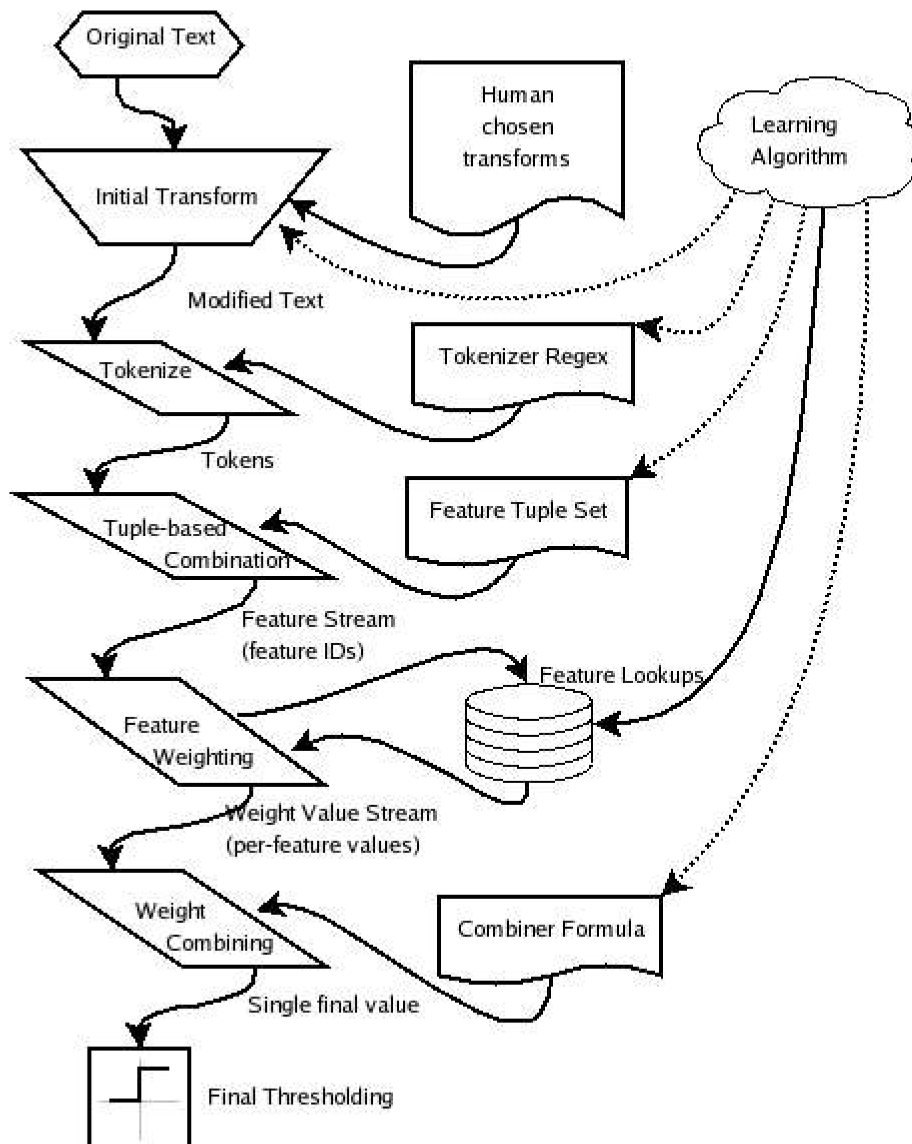
The Proposed Pipeline

We propose the following filtering pipeline as a generalized form for spam classification. Given an input text we perform the following sequential operations:

[1] Admittedly this is a different definition of "spam" than used by some authors who use "unsolicited commercial email (UCE); here we define "spam" to be "any email that I didn't want to get" (very much like Justice Potter Stewart's criterion of "I can't define it, but I know it when I see it", as applied to email instead of pornography)

1. Initial arbitrary transformation (a.k.a. MIME normalization)
2. Tokenization
3. Feature extraction
4. Feature weighting
5. Feature weight combination
6. Thresholding, yielding a go/no-go (or perhaps go / unsure / no-go) result.

Graphically this process can be represented as:



Note that this classification model functions irrespectively of the learning model used to update the databases involved; in the long term the learning model does not matter as long as the learning system produces (eventually) a satisfactory configuration. This

includes both single-user, multi-user, and multi-host distributed filtering systems.

By partitioning the filtering operation into this set of successive stages, we can hopefully gain some insight into the entire process.

Typically, the learning algorithms themselves do not change the feature generator parameters or the combiner formula, although it is always possible for a learning algorithm to perform such actions (or even to have a learning algorithm generate preprocessing steps). The dotted lines in the above diagram show these atypical data paths.

Initial Transformation

The initial transformation (step 1) is often a null step- the output text is just the input text. Other common or often-proposed initial transformations are:

- character-set folding – forcing the character set used in the message to the character set deemed "most meaningful" to the end user. For a typical US-based user, this is base ASCII (also known as Latin-1) where accents are not significant.
- case-folding – removing case changes in the text (for example, lowercasing the entire text).
- MIME normalization – unpacking MIME encodings to a reasonable (and common) representation. In particular, the decoding of BASE64 texts is often useful, as some email systems will BASE64 encode a perfectly reasonable plain-ASCII text thereby concealing it's innocence, and some spammers will BASE64 encode their (incriminating) text to circumvent a spam filter that does not decode BASE64 attachments.
- HTML de-obfuscation– In some rare cases, HTML is an essential part of the message, but HTML also provides an incredibly rich environment to obscure content from a machine classifier while retaining the content for a human viewer. In particular, spammers often insert nonsense tags to break up otherwise recognizable words (*hypertextus interruptus)* and use font and foreground colors to hide hopefully dis-incriminating keywords. Similarly, over-enthusiastic email authors often overuse tags such as <bold> and <color> to the detriment of the parseability of a spam filter to differentiate their texts from spam.
- Lookalike transformations – spammers often substitute characters for other characters that "look alike", in order to avoid known spammish keywords. Examples are using

'@' instead of 'a', '1' (the numeral) or ! (the punctuation) instead of 'l' or “i” (the letters), and '$' instead of 'S'.
- OCR/machine vision operations – using OCR to recognize text in an included image, or machine vision techniques to form a text-based representation of images(such as a pornographic .jpg). This has often been proposed [Leigh, 2002[xi] among others] but the authors are unaware of any actual implementations doing OCR or machine vision classification.

It should be realized that not all systems use an initial transformation; some systems work perfectly well with no preprocessing whatsoever.

It should also be noted that the initial text-to-text transform is human-created, arbitrary, and rarely if ever 1-to-1 in terms of characters, words, or even lines of text. The arbitrary nature of this transformation is what we will eventually use to demonstrate that this unified filtering form includes not only statistical methods, but also heuristic methods and black/white lists.

Tokenization

In the tokenization step of the pipeline, the text is converted into a set of uniquely-valued tokens. We propose a two-step process in feature extraction:
- using a regular expression (a regex) to segment the incoming text into interesting parts (text strings)
- Using a lookup of some form to convert these text strings into unique values

For example, a lookup of textual tokens in a dictionary of tokens previously seen in learning corpus can guarantee unique numerical representations of tokens (textual tokens that haven't been seen in the learning corpus can be assigned a locally unique number but, by definition, no feature containing a previously unseen token exists in the learned corpora and so such tokens don't contribute to the useful classification feature set).

Alternatively, in a fast implementation textual tokens could simply be hashed to a moderately long numerical representation; a 64-bit representation gives roughly 1.8E19 different possible hashes and the occasional hash collision causes only a small decrease in accuracy (remember, half the time, the error is “in our favor”, as it will increase the margin by which a text is judged correctly).

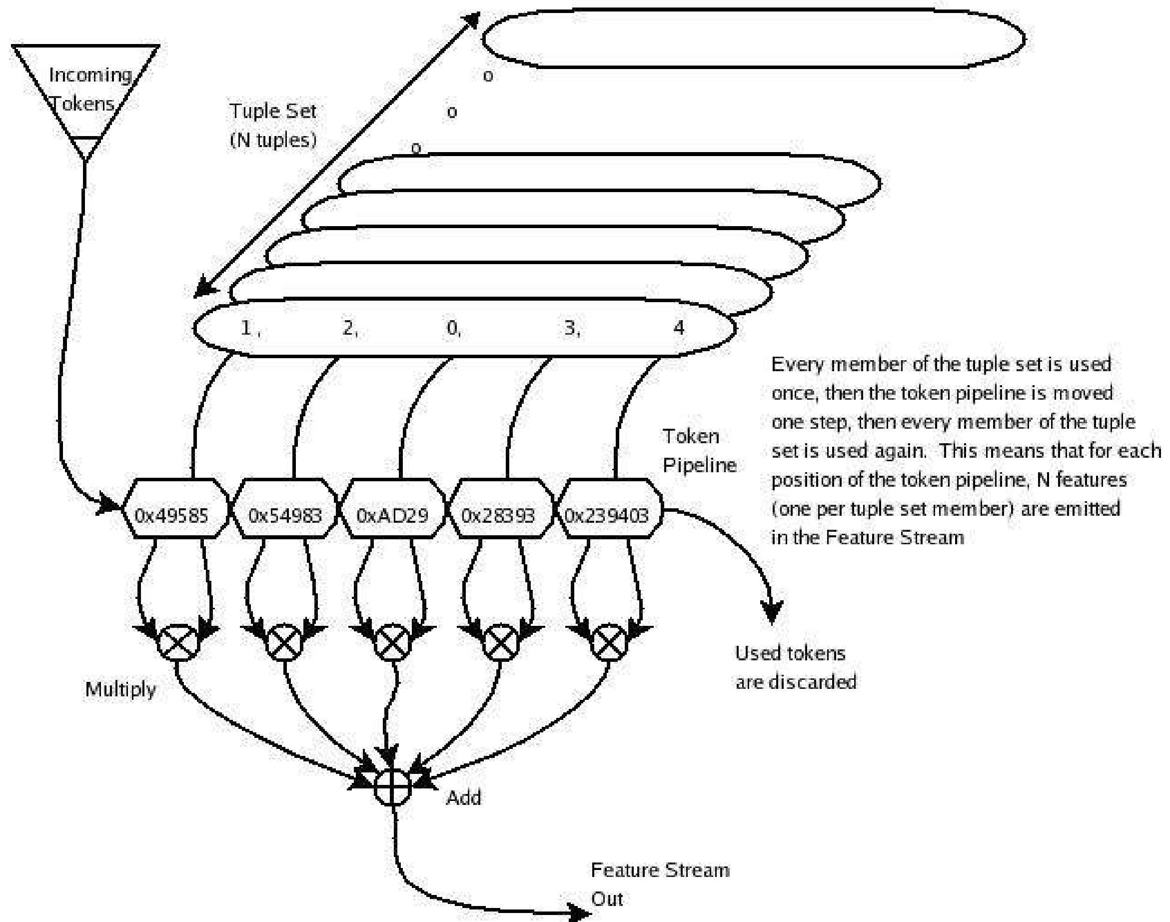
In prior work [Siefkes 2004[xii]] it was found that the choice of tokenizing regexes between an intuitively good regex such as [[:graph:]]+ and a carefully crafted HTML-aware regex

can affect overall accuracy by at least a factor of 25% on one standard corpus.

Tuple-based Combination

As used here, a tuple is a set of encoding constants, each member of the tuple is the corresponding encoding constant for a particular offset within the token stream. A zero valued encoding constant implies that the corresponding token stream element is disregarded; a nonzero element indicates an encoding of the feature depends on that particular token stream element offset. This is simply a dot product, applied repeatedly to incrementing offsets in the token stream.

Graphically represented, here is the tuple-to-feature translation:



In a perfect world, this dot product would have tuple terms that are large prime numbers, and be carried out in infinite-precision (bignum) arithmetic, which would produce unique

output values, thus yielding truly unique encodings for each tuple stream. In actual use, a fast implementation can use small integer tuple values and register-based (fixnum) arithmetic with a trivially small decrease in accuracy as long as the unique token values are not assigned in increasing order; a hash-based token converter works very well for this. Similarly to the previous simplification of hashing instead of true dictionary lookup, each collision caused by 64-bit finite arithmetic are rare enough that we can ignore the small decrease in accuracy.

For simplicity, we will represent all of our tuples in this paper in a normalized form. In this normalized form tuple elements which correspond to pipeline tokens which are to be disregarded will always have the tuple element value 0 (as expected), and all nonzero tuple elements will start at 1 and increase by 1 for each element that is to be regarded with unique position, and will reuse a previously used integer if the corresponding tuple element is to be considered interchangeable with a previous tuple element (previous here being both in the current tuple, and in all prior tuples in this tuple set).

Tuple-based feature generation often produces more features than the original text contained tokens; in some implementations there will be more features emitted from the feature generator than there were bytes of text in the original message.

Many statistical spam filters consider only single words to be features. This corresponds to the simple tuple set:

{ 1 }

In this case, each output feature is simply an input feature set, multiplied by 1. This gives the “typical” behavior of most word-at-a-time Bayesian spam filters.

The somewhat more interesting tuple set

{ 1, 0 }
{ 1, 2 }

yields the classic “digraph” feature set as used by DSPAM [Zdzairski 2003[xiii]], where each word is taken as a feature, each pair of words in sequence are also taken as a feature, and the sequence “foo bar” is not equivalent to the sequence “bar foo”

This tuple-based feature generation also allows representation of bag-based (order is disregarded, but count matters) and queue-based (order matters, but interspersed ignored elements don't) feature generators.

For example, here's an order-ignoring (“bag-based”) feature generator with a viewing

pipeline length of 4, which when slid over the pipeline, generates all pairs taken two at a time:

{ 1, 1, 0, 0 }
{ 1, 0, 1, 0 }
{ 1, 0, 0, 1 }

Note that the tuple coefficients for each token are each 1 – thus the same output numerical representation is generated without regard to the order of the incoming tokens.

Here's a very similar tuple set, however this one is order-sensitive (but not distance-sensitive):

{ 1, 2, 0, 0 }
{ 1, 0, 2, 0 }
{ 1, 0, 0, 2 }

Note that the coefficient of the first feature in the pipeline is 1 and the coefficient of the second feature in the pipeline is 2. This causes this particular tuple set to generate features where order is significant, but intervening tokens are not significant (for example, “foo bar”, “foo lion bar”, and “foo lion tiger bar” all generate the “foo bar” feature value, but “bar foo”, “bar lion foo”, and “bar lion tiger foo” generate a different (but identical between themselves) value.

We can create “contains” feature generators. For example, the following tuple set creates features where the three tokens must occur in the order specified, but as long as the entire feature sequence fits into the pipeline length (here, 5), the precise positioning does not matter. Here is the tuple set:

{ 1, 2, 3, 0, 0 }
{ 1, 2, 0, 3, 0 }
{ 1, 0, 2, 3, 0 }
{ 1, 2, 0, 0, 3 }
{ 1, 0, 2, 0, 3 }
{ 1, 0, 0, 2, 3 }

A useful tuple set for spam filtering is this tuple- this is the OSB (Orthogonal Sparse Bigram) tuple set:

{ 1, 2, 0, 0, 0 }
{ 1, 0, 3, 0, 0 }
{ 1, 0, 0, 4, 0 }

{ 1, 0, 0, 0, 5 }

Note that this tuple set does not generate the unigram (features representing single words, taken one at a time, do not appear in the output feature stream). This OSB tuple set is interesting because it has been shown experimentally to be particularly effective in spam filtering – in experiments with this tuple set, [Siefkes 2004] it was at least equal to the same tuple set including the unigram, and was often more accurate (it was also more accurate than the SBPH tuple set and significantly more accurate than the unigram alone). This is an interesting counterintuitive example where better accuracy is achieved with fewer features.

For reasonable tuple sets the pipeline length is usually quite small – there is evidence [Siefkes 2004] that pipeline lengths in excess of five or six do not increase accuracy.

Feature Weighting

The third step of the filtration pipeline is feature weighting. This weighting has several parts:

- a part of the weighting is based on the prior training of the filter with respect to this particular feature; this is simply a table (or database) lookup. Often, this part of the weighting is just the number of times each feature has been trained into the database representing each of the respective training corpora[2]. There are experimental indications that for some tuple sets, accuracies are higher if training and feature weighting are based on the number of trained documents it appears in, rather than the number of times the feature appears (thus the same feature repeated several times in a document is counted only once for the training or classification).
- a part of the weighting is based on the tuple itself- for example, it appears for some tuple sets and combiner rules to be advantageous to not weight all tuples evenly; tuples containing more nonzero terms appear to be more significant than mostly-zero tuples, and tuples with the nonzero terms adjacent or nearly adjacent are more significant than those tuples with the nonzero terms widely separated. The superincreasing Markovian weighting sets [Yerazunis 2003] using a full set of all possible combination tuples exhibit this behavior. Other tuple sets and combiner rules show no such effect (OSB tuples with Winnow-type combiner rules seem to work best with uniform tuple values).
- a part of the weighting may be based on metafeatures or "database constants"; for

[2] The reader should differentiate between "training texts offered" (the entire training text corpus) versus "training texts actually trained into the database" because many effective learning algorithms do not use all of the training texts offered; the superfluous texts do not have their features added to the database.

example, it appears advantageous to alter the weighting of features depending on the overall count of features learned, as well as the related count of example texts learned.

It is not necessarily the case that a feature's weight is a probability estimate. It is perfectly reasonable to define weight of a feature by reference to a “voodoo” value reference from a database generated by an arbitrary learning algorithm.

It is not reasonable to assume that every possible weighting generator will work with every possible combiner rule; in particular, Bayesian combiner rules need reasonable-valued probabilities as inputs (consider what happens to a Bayesian combiner rule if a local probability is greater than unity or less than zero?)

A simple and obvious per-feature probabilistic weighing formula to generate local probabilities for Bayesian combiner rules is:

$$\text{LocalWeight} = \frac{\text{TimesSeenInClass}}{\text{TimesSeenOverAllClasses}}$$

but unfortunately, this weighting yields absolute certainty when a feature has only been seen just once in a single class. A better-behaved per-feature weight as used in the Markovian filter option in CRM114 [Yerazunis 2002, [xiv]] is

$$\text{Weight} = \frac{\text{TimesSeenInClass}}{\text{TimesSeenOverAllClasses} + \text{Constant}}$$

where **Constant** is some relatively small integer, such as 1 to 100. This is one common form of *smoothing*.

Experimentally, we have found that a better local estimate of probabilistic weight takes into account the relative number of documents in each of the learned corpora; a simple renormalization of weights with respect to how many documents have been inserted into each database. One such formula is:

$$\text{Weight} = \frac{\text{TimesSeenInThisClass} * \text{DocumentsTrainedIntoThisClass}}{(\text{TimesSeenOverAllClasses} + \text{Constant}) * \text{TotalDocumentsActuallyTrained}}$$

gives a significant improvement in filtering accuracy.

Other Weight Generators

It is not necessarily the case that a feature's weight is a strict probability. Other weighting generators can be used as desired; it is perfectly reasonable to define weight of a feature by reference to a database produced by a learning algorithm.

For example, the Winnow algorithm uses additive weights stored in a database; each feature's weight starts at 1.0000 for each class. If the summed weights of the true class remain below a predefined threshold, each of the features found in the document is promoted by multiplying by a constant > 1 (typically in the range 1.1 to 1.35). Analogously, the weights of all features of an incorrect class are demoted by multiplying by a different constant < 1 (typically in the 0.8 to 0.9 range) if their sum exceeds a given threshold.

Other weighting generators may simply pass on a simple function of the observed and expected feature counts; a chi-squared weight generator is in this category.

Weight Combination

At this point in the pipeline, we now have a series of weights corresponding to each feature in the unknown input text. We now must combine these weights to obtain a single output result.

Some spam filters use completely linear weight combining rules – that is, their combining rules obey the laws of superposition, and the output value of a concatenated text is equal to the combining rule applied to the values of the unconcatenated text segments.

Other filters have nonlinear features in their combining laws, such as "ignore any weight below a particular threshold". This type of nonlinearity is easily accommodated by simple modification of the combining rules to set "newvalue = oldvalue" for each feature weight if the threshold of significance is not reached.

Other filters use a sorted-weight approach, typically "only use the most extreme N weights found in the document". This sorted-weight approach is represented as a function of the stream of weights emitted from the weight generator software; the only

difference is that there is local memory inside the chain rule.

Bayesian Combining

A very common combining formula is the generalized Bayesian probability combiner formula – this relates a local probability due to a given feature, a prior probability (the probability of a document being in a particular class before this feature was encountered) and the posterior probability (the updated probability, given that the feature was found in the document):

$$PposteriorThisClass = \frac{PpriorThisClass * PlocalThisClass}{\text{SUM over all classes} (PpriorThatClass * PlocalThatClass)}$$

Chi-Squared Combining

Another common combining rule is the chi-squared rule. In the chi-squared formulation, the observed occurrences of features are compared against the expected number of occurrences of those features.

The actual chi-squared formula for the chi-squared value of one exclusive feature is:

$$X^2 = ((Observed-Expected)^2 / Expected)$$

and then all of the chi-squared exclusive feature values are summed.

Note that the underlying mathematics of chi-squared calculation implies that the features must be exclusive; this is not normally the case for most spam filtering features. Some examples of exclusive features are binned time-of-arrival, originating IP address, and claimed “From:” originator; an example of a pair of nonexclusive feature is whether the words “squeamish” or “cheese” appeared in the text.

Another chi-square-related combining method is to “turn the table 90 degrees” and consider if the relationship between the expected and observed values of each feature supports the hypothesis that the document is a member of the set “Good”, a member of the set “Spam”, or a member of neither (that is, the null hypothesis).

In this method, the independent variables are the features, and the dependent variables are the categories “Good” and “Spam”, The per-class table looks like this (ignoring the ad-

hoc Yates correction, which is just subtracting 0.5 from the magnitude of the (Observed-Expected) term before squaring for any cell whose observed count is less than 5) [Garson 2004] [xv]:

Feature	*Observed Count in incoming text*	*Expected Count for Good*	*Expected Count for Spam*	X^2 *Good*	X^2 *Spam*
"foo"	0	5	0	5	0
"bar"	3	1	4	4	0.25
....	...	...	...	...	...
"xyzzy"	1	1	0	0	1
TOTAL				**9**	**1.25**

From this we see that the chi-squared result for "Good" is 9, which is very high (and indicates that a hypothesis that the unknown incoming text is a member of class "Good" is unlikely to be true). The chi-squared result for "Spam" is 1.25, quite low, and supports a hypothesis that the unknown incoming text is spam.

Winnow Combining

If the weight calculation and value updating is performed using the Winnow algorithm, the combining rule is particularly simple:

WeightOut = WeightIn + LocalWeight

That is, in Winnow, the sum of the Local Weights is the output of the fit; the higher the weight total, the better the fit (unlike chi-squared, where a low weight total is better fit).

Final Thresholding

After the weights are combined, a final thresholding is performed. For filters that use probability, the final decision threshold value is typically 0.5 (ambivalent probability). As some filter authors and filter users consider it preferable to falsely accept some spam in order to decrease the amount of falsely rejected nonspam, not all probabilistic filters use p = 0.5.

Other filter algorithms use a different final threshold; for example, in Winnow, the winning class is the class with the largest total weight score; in the current version of SpamAssassin (at this writing) the threshold is a weight of < 4.5 for nonspam, and the original “Plan for Spam” code used < 0.9 for nonspam.

Emulation of Other Filtering Methods

If this unified model of spam filtering is truly general, we must show that it is possible to represent all possible (or at least a large fraction of the useful) filters within it's framework. In one sense, this is trivial to prove, as the initial text-to-text transform can contain an arbitrary computation and all subsequent stages can operate in a pass-through mode.

Despite this trivial proof, it's useful to consider how to use an optimized implementation with parameterized code in the unified filtering model to implement other types of filters such as heuristic filters and black/whitelists. The significant extent to which the same unified pipeline can be used demonstrates the significant amount of commonality between blacklists, heuristic filters, and statistical filters.

Emulating Whitelists and Blacklists in the Generalized Model

Voting-style whitelists:

Emulation of a voting-style whitelist/blacklist filtering in the generalized model is quite simple. All that we need to do is to look for whitelisted or blacklisted words in the input, and count them. If the whitelisted words outnumber the blacklisted words, the text is good, if the blacklisted words outnumber the whitelisted words, the text is spam, and it's indeterminate if there is a tie.

Here's a set of parameters for the generalized model that produce a whitelist/blacklist filter:

1. The initial text-to-text transform is “none”. That is, the output text is equal to the input text.
2. The token generator regex is [[:graph:]]+ (resulting in blank-delimited blacklisting and whitelisting)
3. The tuple set for feature generation is just { 1 } , giving feature Ids that correspond to single words.

4. The feature database is loaded only with those words that are blacklisted or whitelisted, with a value of +1 for whitelisted words, and -1 for blacklisted words. All other words return a 0.0 value.
5. The feature weight rule is “FeatureWeight = FeatureLookedUpValue”
6. The feature combiner rule is “NewScore = OldScore + FeatureWeight”
7. The final decision threshold is “ > 0 --> good, < 0 --> bad, otherwise unknown”

This particular implementation scores whitelist words and blacklist words equally; some people consider them equally valuable. For pure whitelisting or blacklisting, one could put only the respective whitelist or blacklist words into the feature database, or one could weight whitelist words with much higher weights than blacklist words (or vice versa).

Prioritized-rule Blacklists

Some whitelist/blacklist systems don't count votes; instead the rules themselves are ordered and the highest priority rule “wins”. We can easily emulate this method by assigning additive superincreasing weights to whitelist/blacklist entries. This can be done by setting the weight of each rule to be twice as high as the next least important rule as demonstrated by the following convention:

1. The weight of the lowest priority black/whitelist element is +|-1 (sign depends on whether the mail should be white- or blacklisted),
2. for the next-to-last priority list item assign the weight +|-2,
3. for the second-to-last priority list item, assign the weight +|-4.
4. repeat this doubling of weight magnitude for each increasingly-important priority list item.

In this way, the absolute weight of each rule is higher (by 1.00) than the sum of the absolute weights of all rules to follow, meaning that the decision of the highest-priority matching rule can never be reversed by lower-priority rules.

Emulation of Heuristic Filters in the Generalized Model

We now consider the question of the emulation of heuristic filters in the unified model. Heuristic filters are by definition created by expert humans, to trigger on specific features of the unknown text. The individual features may either accumulate a weight or score, or by themselves be sufficient to reject (or accept) an unknown text.

It is possible to form hybrid human+machine-created systems in this form. For example,

SpamAssassin [spamassassin.org[xvi]] has a feature library of several hundred human-written recognizers; the recognizers themselves have weights that are optimized by a genetic algorithm testing against a well-vetted base of spam and nonspam texts.

Emulation of these heuristic-feature-based filtering systems is easily performed by a set of rewrite rules:

1. generating a "local key", a string with a vanishingly small probability of appearing in an incoming text; this local key can be constant for a user, or be randomly generated for each incoming text.
2. executing a series of rewrite rules; each rewrite rule corresponds to one of the original heuristics. Whenever a rewrite rule matches the incoming text (corresponding to the original heuristic being triggered), the rewrite rule appends a new line at the end of the unknown text; this new line contains the local key followed by the heuristic rule's unique identifier. The text that matched the rewrite rule remains in the text, so that other heuristic matchers can continue to operate against that text.
3. the second-from-last rewrite rule deletes every line in the unknown text that does not start with the local key.
4. the last rewrite rule deletes every copy of the textual representation of the local key from the text, leaving only the unique heuristic identifiers.
5. The text emitted from the preprocessor is now just the unique identifiers of the heuristic rules that were satisfied by the original text.

The resulting output text is now taken one unique identifier at a time (that is, with the tuple set:

{ 1 }

and the respective weightings of the unique identifiers are then looked up. For example, if we were emulating SpamAssassin, the respective weightings stored in the database would be the relative point values of each of the heuristic features found, the local weighting formula would be just:

LocalWeight = DatabaseValueReturned

and the combining rule would be the summation of the local weights:

TotalWeight = TotalWeight + LocalWeight

The current version of SpamAssassin at this writing uses a threshold of 4.5, so the final decision threshold is:

$$TotalWeight – 4.5 > 0$$

Examples of Several Popular Spam Filters in the Generalized Model

Here, we will show several popular spam filters as expressed in the generalized model.

Classic Paul Graham 2002 "A Plan For Spam" model:

Preprocessor: lowercase, remove HTML comments, remove numeric-only constants

Tokenizer: [[-'$a-z]]+

Feature Generator: single tuple - { 1 }

Lookups: count of good occurrences "G", count of bad occurrences "B"

Weight Generator:

IF (G+B < 5) : 0.5

ELSE (Ceiling 0.99 (Floor 0.01 (Bad / (Good+Bad)))

Weight Combiner: Classic Bayesian, top 15 scorers only

Final Decision Threshold: 0.9

D2S model (GET RICHARD JOWSEY TO CHECK THIS):

Preprocessor: lowercase, remove HTML comments, add specific markers for FROM and TO fields in header

Tokenizer: [[a-z]]+

Feature Generator: single tuple - { 1 }

Lookups: count of good occurrences "G", count of bad occurrences "B"

Weight Generator:

(Ceiling 0.99 (Floor 0.01 (Bad / (Good+Bad)))

Weight Combiner: Classic Bayesian

Final Decision Threshold: 0.5

CRM114 (2002 model):

Preprocessor: remove HTML comments, expand BASE64's

Tokenizer: [[:graph:]]

Feature Generator: SBPH (Sparse Binary Polynomial Hash) tuple set of window length 4 (note that CRM114 in 2002 did not use differing weightings depending on the tuple. CRM114 moved to a window-length 5 tuple set in early 2003):

{ 1 0 0 0 }
{ 1 2 0 0 }
{ 1 0 3 0 }
{ 1 2 3 0 }
{ 1 0 0 4 }
{ 1 2 0 4 }
{ 1 0 3 4 }
{ 1 2 3 4 }

Lookups: count of good occurrences “G”, count of bad occurrences “B”

Weight Generator:

0.5+ Good / ((Good+Bad)/2)+16)

(NOTE: 2002 CRM114 did not yet have Superincreasing Markovian weightings that depended on the tuple being used)

Weight Combiner: Classic Bayesian, score everything.

Final Decision Threshold: accept if Pgood > 0.5

SpamAssassin model:

Preprocessor: all 300+ SpamAssassin heuristics (using the feature Ids as in current SA),

deleting all non-featureID text as a final step before tokenizing)

Tokenizer: [[:graph:]]

Feature Generator: single tuple - { 1 }

Lookups: precalculated per-feature weights of good occurrences “G”, weight of bad occurrences “B”

Weight Generator:

Simple lookups, no G/B combination.

Weight Combiner: Simple addition

NewScore = OldScore + B - G

Final Decision Threshold: accept if < 4.5

Conclusions and Further Work

This paper shows that a single unified pipeline, controlled by a relatively small set of input parameters (a set of rewrite rules, a regex, a set of tuples, a mapping/lookup table, and a chain combining rule) can describe nearly all of the spam filters of all variants typically available today. By showing the commonality of these filters, we hope to stimulate creative thought to advance the state of filtering art, with some hope of advancing the entire field of information retrieval.

[i] Blacklisting notes – http://www.email-policy.com/Spam-black-lists.htm

[ii] Spamcop can be found at http://www.spamcop.net/bl.shtml

[iii] the CBL can be found at http://cbl.abuseat.org/

[iv] Earthlink can be found at http://www.earthlink.net

[v] Yahoo can be found at http://www.yahoo.com

[vi] D2S can be found at http://www.death2spam.com

[vii] Spambayes can be found at http://spambayes.sourceforge.net

[viii] SpamProbe can be found at http://spamprobe.sourceforge.net

[ix] DSPAM can be found at http://dspam.nuclearelephant.com/

[x] CRM114 can be found at http://crm114.sourceforge.net

[xi] Darren Leigh, January 2003, personal communication

[xii] Christian Siefkes, Fidelis Assis, Shalendra Chhabra, and William S. Yerazunis. Combining Winnow and Orthogonal Sparse Bigrams for Incremental Spam Filtering. In Jean-Francois Boulicaut, Floriana Esposito, Fosca Giannotti, and Dino Pedreschi, editors, Proceedings of the 8th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD 2004), volume 3202 of Lecture Notes in Artificial Intelligence, pages 410-421. Springer, 2004.

[xiii] John Zdziarski's DSPAM is available at http://dspam.nuclearelephant.com/

[xiv] William S, Yerazunis, "The Bayesian Filtering Plateau at 99.9% Accuracy", Presented at the MIT Spam Conference, January 2004; available at http://crm114.sourceforge.net/Plateau_Paper.pdf.

[xv] David Garson's Chi-Squared Significance Tests, tutorial, available online at http://www2.chass.ncsu.edu/garson/pa765/chisq.htm

[xvi] The SpamAssassin web page is at http://spamassassin.apache.org/; the .com and .net variants are commercial domain-squatters.