Classifier architecture and Features.*

Greg Kochanski
http://kochanski.org/gpk

2006/02/28 10:49:18 UTC

1 Classifiers and Search Engines

A text classifier divides up the set of documents that you give it into several classes: where each class is defined by a group of training examples[1]. People who build text classifiers often use Bayes' Theorem (BsT) as a way of collecting many small bits of evidence into one overall score. Even if it is not used explicitly, many common classifier algorithms can be thought of as Bayesian[2]. For instance, any algorithm that adds up local scores, and then makes a decision based on the total score can be fit to the Bayesian mold: the local scores correspond to the log of a Bayes factor, and the total score is then the log of the posterior probability. (Recall that addition of logarithms is equivalent to multiplication, and you can see the equivalence to the odds-ratio formulation of Bayes' Theorem.) However, it should be pointed out that not all fit the mold, even though the mould is pretty broad.

In this part of the course, we discuss text classifiers, but some of what we say here can also apply to certain search engines. Although there are significant engineering differences[3], some search engines can be thought of as text classifiers. They divide up the universe of documents into those that are close to the query and those that are not.

2 Classifier Architecture

A Bayesian text classifier has the following components (see Figure 2):

1. An algorithm to extract a **feature vector** from some text.
2. An algorithm for going from the feature vector to a probability.
3. Bayes' Theorem.
4. Some decision rule to assign a class, based on the probabilities obtained from Bayes' Theorem.

To classify a document, one

1. computes the document's feature vector,
2. from the feature vector, one calculates the probability $P(\text{feature vector}|\text{Model})$ for all the models under consideration,

[1] This is known in the machine learning community as "supervised learning": there is a explicitly provided correct answer for each element of the training set, and the intent of the system is to reproduce those answers. "Unsupervised learning" is a catch-all term for systems where the correct answers are not provided explicitly.

[2] Although their authors may not have thought of them that way.

[3] The engineering differences arise because the search engine has a fixed dataset and a dynamically changing query, while the text classifier has a fixed query and dynamically changing data. The result of this reversal is that search engines and classifiers compute different things in advance.

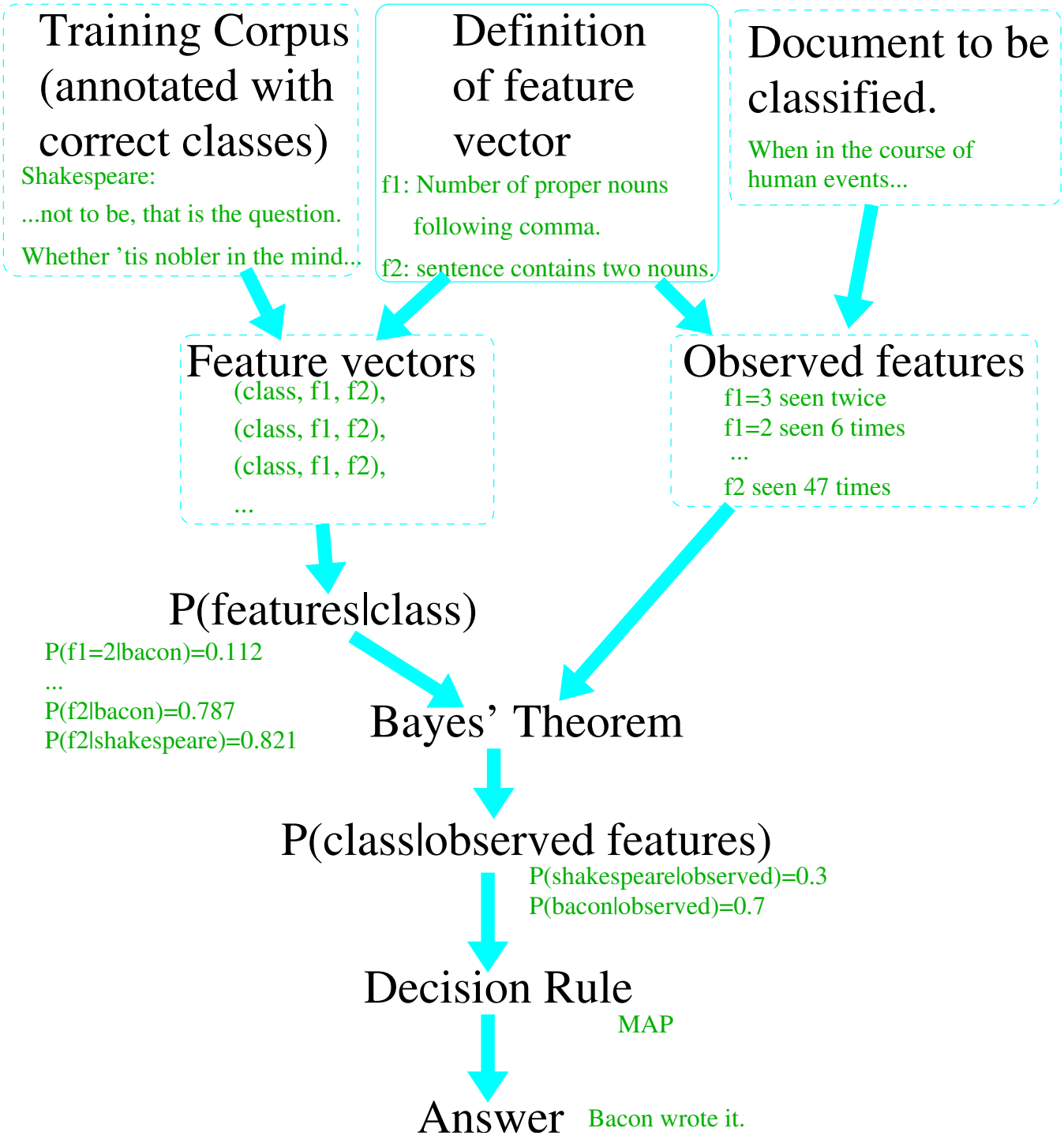


Figure 1: Architecture for a text classifier.

3. evaluates B'sT to get the posterior probabilities, (this may include some allowance for the fact that the features are usually correlated)

4. applies the decision rule to choose a class, and

5. announces the result.

However, before you can do the classification, you usually need to collect some training data to estimate all the probabilities P(feature vector|Model) (see 2, above). This training run involves:

1. computing the document's feature vector, for every document in the training set.

2. Counting the number of times different features occur,

3. Applying Good-Turing estimation (or something similar) to derive probabilities from the counts.

2.0.1 Testing

Finally, one needs to test the classifier to be able to know how well it works. The classifier must be tested on data that wasn't used in the training process. It is common for a classifier to perform worse on the testing set than on the training set.

The test set is the real performance, hopefully close to what you might expect if you collected another corpus; the results on the training set are artificially high. One can see how this might work by considering a N-gram classifier where N is large enough so that each N-gram appears only once. $N = 300$ will do, selecting a paragraph-sized chunk of text.

Take our English/Spanish classifier as an example. If this particular document is in the English training set and $N = 300$, our classifier will contain the information that $P(\text{Finally, one needs to test} \ldots |\text{English}) = 0.001$, while any reasonable estimate for $P(\text{Finally, one needs to test} \ldots |\text{Spanish})$ will certainly be an extremely small number. Consequently, if you test the classifier with this particular document you will find the Bayes ratio to be

$$\frac{P(\text{Finally, one needs} \ldots |\text{English})}{P(\text{Finally, one needs} \ldots |\text{Spanish})} = \frac{0.001}{\text{something tiny}} = \text{a large number.} \tag{1}$$

The classifier will then (correctly) report that this document is written in English, and so it will go for the rest of the training set. Such a classifier will get a score of 100% correct if you test it on the training set.

However, if you were to test it on documents that *weren't* in the training set, the Bayes factor would be based on probability estimates for N-grams that were observed in neither English nor Spanish. The Bayes factor would depend on the details of the probability estimation procedure, but – crucially – there is no reason for it to depend on the actual language of the test document. Consequently, such a classifier would perform at a chance level when tested with new, unused data. This chance level would correctly represent its utility in the wider world, away from the training set. A document from the wider world will simply not contain any of the N-grams that the classifier knows, so it is effectively Greek.

While the example presented above is extreme, this phenomenon is always present to one degree or another. As a general rule, results simply will not be accepted for publication if you test on your training set.

2.1 Details: Generating the Feature Vector

A feature vector is a list of features that describe the document. In the simplest case, features are Boolean events that are either true or false for the document as a whole, such as "The document contains the word 'elephant'." However, the only absolute requirement on a feature is that one must be able to calculate P(feature|Model) for all the models.

It is quite possible for features to have integer values, such as "the number of times that the 3-gram 'usu' appeared in the text." Even real-number values are possible, such as "the mean number of proper nouns per subordinate clause," or "the mean position of the verb in each sentence."

Virtually anything goes, so long as it can be computed automatically from the text. However, not all features are equally good for all purposes.

Different languages have different letter-to-sound rules, for instance, so one could expect that letter bi-grams or even a feature as simple as a letter unigram could do a good job of capturing the difference between two languages.

On the other hand, if one were trying to separate two different authors writing in the same language, unigram letter frequency would be worthless: even if the authors habitually chose different words, the overall letter frequencies of one author's set of English words doesn't differ much from any other set.

2.2 Rules for Selecting Features

2.2.1 Use Multiple Regions

One important principle when you define features is that if a document contains regions with drastically different characters, a given feature should be computed across just a single region. For instance, e-mail is an obvious example: The header of an e-mail is very different from the body.

Mathematically, if a document is composed of multiple regions, and each region has different statistics, one can show that the information gained by treating each region separately is larger than the information gained by treating the whole document as a unit.

As an example, consider a document that is half text and half image, and that we are trying to decide whether it is an English or a Spanish document. Further, assume that the image is encoded in some way (like JPEG) that all byte unigrams in the image are equally probable.

Now, some unigrams are, on their own, good discriminants between the two languages: "w", for instance, is quite rare in Spanish, accounting for only 1/80000 of all letters. It has a Bayes Factor of 1000, between the text in those two languages. However, in a document that is half image and half text, most occurrences of "w" will actually be bytes in the image: 1/256 of the image bytes will have the same value as a "w" in the text part. Consequently, in a document that's half image, 1/512 of the bytes will be "w" in Spanish, and 1.6% in an English document. The Bayes factor of "w" is then reduced from 1000 (in pure text) to 8 in a mixed document.

To complete the proof, we consider two "w"s, one in each of the two halves. If we treat the two halves of the document separately, we get a Bayes factor of 1000 from the first and 1 from the second, yielding an overall Bayes Factor of 1000. If we treat the document as a unit, we get a Bayes Factor of 8 from each of the two "w"s, for an overall Bayes Factor of 64 for the document.

A more linguistically-relevant example of the same principle shows up when one tries to assign authorship to a play. The names of the characters appear many times in a play, and are not necessarily common words in the author's language. They could even be foreign names, so that N-gram letter statistics could be wildly different from words that the author would normally use. Thus, names of characters can be considered a separate region of the play with different statistical properties. By the above proof, we will get more information about the document if we treat the names separately than if we treat the document as a whole.

2.2.2 Features should ignore unimportant differences

Another rule is that one wants to pick features that are insensitive to unimportant differences. For instance, we know that both long and short documents exist in both English and Spanish. Consequently, the length of a document is unlikely to be particularly informative.

If we really know, on theoretical grounds, that a parameter is uninformative, we should leave it out. We leave it out to avoid being confused by accidental correlations in the training corpus. Imagine we collect our documents from the web. It could easily be the case that Spanish documents on the web would tend to be shorter than English documents. (For instance, there are many absurdly long technical descriptions of software on the web, and the majority of them are in English.)

However, if we are planning to use our document classifier on a mix of novels and short stories, then any dependence on length that the system might learn is just noise. It is not true that all novels are in English and all short stories are in Spanish, so it is better to ignore length as a feature in this classifier.

If we decide that some feature is unimportant, then we should not use features that are tightly correlated with the unimportant feature, either. If the length is unimportant, then probably we don't want to use the number of subordinate clauses in the document as a feature either: long documents will have lots of subordinate clauses. What we should do (and this gets into the next point) is construct a feature that captures the (hypothetical) importance of subordinate clauses, but doesn't depend upon the document length.

2.2.3 Features should be uncorrelated.

In §2.5, we discuss how to approximately compensate for correlations amongst features. It is better to avoid the correlations in the first place than to attempt to correct for them later.

As an example, if you counted

1. the number of words in a document,
2. the number of proper nouns in a document,
3. the number of dependent clauses in a document,
4. the number of center embeddings in a document,
5. the number of left embeddings in a document,
6. the number of right embeddings in a document,
7. the number of commas in a document, and
8. the number of acronyms in a document,

you would find that the resultant classifier was extremely sensitive to the length of the document, because all of those counts will generally increase as the document gets longer.

2.2.4 Features should be robust.

Features need to be calculable for imperfect documents. For instance, an algorithm that looks for proper nouns needs to do something sensible when confronted with a part number: is "CJRR4124124" a proper noun? Some documents will have thousands of part numbers, and they cannot be allowed to bias the statistics too badly.

2.2.5 Features should reflect human perception

People do not perceive text the same way a computer does. For instence, many people will not treat misspelled words as seperate lexical items, but will read right over them and never notice a difference that would be dramatic in terms of word frequencies and/or N-gram statistics.

This ties in with robustness (§ 2.2.4) in the sense that features should not be unduly perturbed by normal spelling and punctuation errors. There are several examples of Spam in Appendix A that make use of the difference between human and machine interpretation of text.

2.2.6 The Art of choosing Features

Beyond those quantitative rules, one is on one's own, when choosing features. You choose features to capture what you think are the most important aspects of the classes you care about. You try to choose features that are broad, in the sense that they appear in many documents in a class. Good selection of features depends upon a knowledge of the classes and the documents.

2.3 Example: features in a spam filter

The spam filter uses a large and complex set of features, but they come in three general categories:

- Letter 2-grams, 3-grams, and 4-grams.
  - On various header lines. N-grams from different header lines are kept separate.
  - On transformed header lines. Different header lines are transformed different ways, to try to focus on the relevant information.
    - As an example, the subject line is transformed as follows: All lowercase letters are replaced by "a", all uppercase letters by "A", and all digits by "0". This captures odd punctuation!!! and subject lines with TOO MUCH CAPITALISATION.

    - Lines containing addresses are split into the local part of an e-mail address and the domain name. The two parts have very different n-gram statistics.
    - Other header lines are treated in their own specialized manners.
  - On the text of the e-mail, after decoding, uncompression, *etc.*.
  - On transformations of the text.
    - Transformed by removing all HTML tags. This avoids dilution of text statistics by the very different statistics of HTML tags.
    - Transformed by retaining only HTML tags. HTML tags, by themselves, are a fairly good indicator of spam, but many of the addresses in the tags provide additional evidence.
    - The above two transformations, but mapping all letters to "a" or "A", and all digits to "0". (This focusses on punctuation.)

- Usage of different header fields and the MIME type(s) of the body.
- Lengths of the body. The lengths are labelled by the MIME type of each section of the body, so lengths of plain text sections are counted in separate features from the length of sections that contain HTML or images.
  - Lengths of each section of the body, at both coarse and fine resolution.
  - Lengths of each section of the body on the various types of transformed text.

2.4 Computing Probabilities of Feature Vectors

Normally, the algorithm to get the probabilities of observing different feature vectors (step 2) is not known *à priori*. In that case, it must be derived from a training corpus.

2.4.1 Boolean Features

In the simple case of Boolean features, one simply counts how many times a feature appears, and then uses Good-Turing estimation or ELE or other suitable technique to estimate the probability.

2.4.2 Integral and Real Features - Conversion to Boolean

If you have features that are not Boolean, you can convert them to events. You can convert a numerical feature to a bunch of binary features by just dividing the number line into intervals and seeing if the value lands in each interval.

For example, if you have a feature "length", which contains the mean length of a sentence (measured in words), you can convert it to Boolean features by setting up the intervals $0 < \text{length} \leq 1.5$, $1.5 < \text{length} \leq 2.5$, $2.5 < \text{length} \leq 3.5$, ... In general, you want to choose the size of the intervals so that each of the resulting Boolean features occurs at least a few times in the training corpus. If not, you may be able to adjust the size of the intervals.

This may not be the best way to go, though. It requires enough data to estimate all the probabilities for a sufficient number of intervals. This approach can be impractical if one needs the probability to depend on multiple features: the intervals become regions in a multidimensional space and the number of regions grows exponentially with the number of features that are involved.

2.4.3 Integral and Real Features - Analytic Approaches

Analytic approaches for evaluating the probabilities can be much better if there is reason to believe that the probability depends on more than one or two mutually dependent features. Analytic approaches consist of assuming some equation for the probability, such as a multivariate Gaussian, and finding values of the (in the Gaussian case) mean and covariance matrix that best reproduce the training data.

Such approaches are used in speech recognition systems, where one might typically have 30-60 mutually correlated features for one acoustic frame. A proper description is beyond the scope of this course.

2.5 Applying Bayes' Theorem

We have covered this in detail earlier in the course.

However, despite the fact that Bayes' Theorem needs either events that are statistically independent of each other or fully conditional probabilities, features in text classifiers almost never meet either of the requirements.

To partially compensate for these nearly unavoidable failings, it is common to weight different features.

Here, once again, is BsT displayed for repetitive application, on multiple models (where i selects the model):

$$P(M_i|D,C) = \frac{P(D|M_i,C)}{P(D|C)} \cdot P(M_i|C) \tag{2}$$

The ratio $\frac{P(D|M_i,C)}{P(D|C)}$ plays the role of the Bayes Factor in the two-model case: it is large if the data strongly supports model i over other models, and much smaller than one if the data reject i.

Now, if one had M features that were identical, and applied BsT sequentially on all the features, then you would end up with

$$P(M_i|D,C) = \left(\frac{P(D|M_i,C)}{P(D|C)}\right)^M \cdot P(M_i|C), \tag{3}$$

which would mean that the feature had a M-fold larger effect on the final probability than it should have had. The same logic applies if one has features that are not identical, but instead are highly correlated: the correlated part that shows up in all the features will have an incorrectly large effect on the posterior probability estimate.

If one can estimate how strongly correlated certain features are, one can partially correct for the correlation by using $(\frac{P(D|M_i,C)}{P(D|C)})^(1/M)$ in BsT, instead of $\frac{P(D|M_i,C)}{P(D|C)}$. The degree of correlation is expressed by M, where $M = 1$ for independent features and M is large for features that are part of a highly correlated group.

Usually, M can be estimated with a bit of thought, by considering how many nearby features will be correlated. For instance, when the features are letter N-grams, M is typically between the length of the N-gram and the length of a typical word.

The equations can be recast into linear form by taking the log, in which case one is adding up many correlated terms,

$$\log(P(M_i|D,C)) = \sum_j M_j^{-1} \cdot \log\left(\frac{P(D_j|M_i,C)}{P(D_j|C)}\right) + \log(P(M_i|C)), \tag{4}$$

and M_j^{-1} acts as a weight factor, controlling the importance of the j^{th} feature. This approach can be related into techniques like Latent Semantic Indexing [1] and Principal Value Decomposition [3], by considering M to be a matrix of correlations, and M^{-1} to be a matrix inversion operation.

2.6 Decision Rules

Again, covered in earlier lectures. Often, one uses a MAP decision rule. However, the Spam filter we describe use a minimum Bayesian Risk decision rule: it assigns a cost of 1 to passing a spam message, but a cost of 100 to improperly dropping a non-spam message. The large ratio of costs biasses the classifier toward passing messages that are not clearly spam.

3 A Warning

Text classifiers only know about the universe through the window of the feature vector, and they only know about the part of the universe that you show it by your choice of training set. A classifier is only as good as the combination of the two, and will certainly obey the grand principle of "Garbage in, garbage out."

For instance, I once built a system to classify e-mails into categories. The intent was that you would set up some mail folders, put in some examples, and the computer would then sort your incoming mail. It was a fairly straightforward letter N-gram classifier, and it seemed moderately successful. I would divide my corpus of labelled e-mails into a training set and a test set, run it, and find that it classified the test set fairly well.

Fortunately or unfortunately, I then spent some time looking at which N-grams were most important in the classification and finding which words they came from. I had grand plans of helping the classifier to generalise by

finding the important words, then looking up their synonyms in WordNet[2] and using the synonyms to broaden the search.

To my surprise and dismay, it turned out that the most important N-grams were the initial digits of room numbers. At the time, it was standard procedure to put your room number in the address, and people in Bell Labs were clustered by their technical topics. All the astrophysicists were in Building One, D or E-wing, fourth floors; chemists in Building One, A or B wing, first or second floor; speech and language people in Building Two, D-wing, floors 3,4,5; and computer scientists were in other wings of Building two.

In retrospect, it shouldn't have been surprising that N-grams like `1E4`, `1A2`, `2D4` or `2B5` should do such a good job of separating mail for different technical projects. The system was just reflecting the separation that the human managers had already performed. Unfortunately, that meant that the performance of the system depended on two particular features of Bell Labs circa 1997, and a lucky definition of the classes of e-mail. If the classes hadn't been closely correlated to projects and the projects hadn't been split along academic disciplines, and the people clustered in hallways along the lines of academic disciplines, the software wouldn't have worked.

The system would not be portable to a different corpus, because it depended on a lucky correlation which (a) had nothing to do with the language used, and (b) would probably not work elsewhere. Indeed, when I took out the room numbers and names from the e-mails, the performance of the system dived to completely uninteresting levels.

So, the classifier had learned to classify projects by the room numbers of their participants, rather than the words used in the e-mails because the unintended specific correlation in the training data was better than the more general correlation I had hoped to capture.

Several morals can be taken from the story:

1. One should always ask why the system works (or doesn't), rather than just being content with a performance number.

2. Careful design of the feature vector is essential: I included the entire e-mail header because I wanted to include the subject line, and I didn't want to spend the time to find or build a parser for the headers so that I could pick out just the subject line. As a result of this sloppy choice of feature vectors (*i.e.* by including the To: and From: lines), I ended up wasting several weeks on a wild goose chase.

3. Corpora for training and testing text classifiers need to be as broad as possible, so that they don't contain strong accidental correlations that can fool you into believing you have succeeded.

   This example also points out that the common practice of dividing a corpus into a training set and a test set can sometimes be dangerous, because there may be correlations built into that corpus that are found nowhere else.[4]

4. Machine learning systems are not like humans. They do not share your unspoken assumptions.

References

[1] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.

[2] George A. Miller et al. *Wordnet: a lexical database for the English language.* Princeton University Cognitive Science Laboratory, http://www.cogsci.princeton.edu/ wn/, 2004.

[3] Michael E. Wall, Andreas Rechtsteiner, and Luis M. Rocha. Singular value decomposition and principal component analysis. In D.P. Berrar, W. Dubitzky, and M. Granzow, editors, *A Practical Approach to Microarray Data Analysis*, pages 91–109. Kluwer, Norwell, MA, 2003. LANL LA-UR-02-4001, http://arxiv.org/abs/physics/0208101.

[4] This ties into the subject of statistical sampling that we will cover soon. Any corpus is a sample of the entire language, but it is probably a biassed sample, in one way or another.

A Examples of Spam, circa January 2004.

Spam is an interesting case, much different from normal documents, because it is often specifically designed to be hard for a machine to process, while conveying its message to a human reader by way of a browser.

Here's a bit of simple HTML code. Spam filters have no problem with this, so long as they recognise it as HTML and parse it properly. NB: HTML tags are enclosed in <brackets>, and typically come in pairs: `<X>` and `</X>` give the text in between the property `X`.

```
    <a
href=http://200.216.233.100/masterloanz/><b>Home Improvement,
    Refinance, Second Mortgage, Home Equity Loans,
    and More!</b></a></font><font face=Verdana, Arial, Helvetica, sans-serif size=2><br>
    <br>
    You're eligible even with less than perfect credit!<br>
    <br>
    This service is <font color=#000000><b>100% FREE</b></font>
    to home owners and new home buyers without any obligation.
    <br>
    <br>
    Just fill out a quick, simple form and jump-start your future
    plans today!</font><br>
```

This next example is, statistically, much like many e-mails that you might get from your co-workers. (In terms of letter N-gram frequencies and even word N-gram frequencies.) It has relatively few words that mark it as spam. It's only at the semantic level that it is obviously junk.

```
Sender: owner-mailing_list_eurospeech@idiap.ch
Precedence: bulk
X-idiap-list: mailing_list_eurospeech
X-EFL-Spamscore: 1%
X-Spam-Prob: -22.96835
X-Spam-Rating: 3

 Permit me to inform you of my desire of going into business relationship
with you. I got your name and contact from the Benin Chamber of
Commerce and Industry. I prayed over it and selected your name among
other names due to its esteeming nature and the recommendations given to
me as a reputable and trust-worthy person that I can do business with
as per the recommendations.I must not hesitate to confine in you for
this simple and sincere business.
```

Here's an example that's probably designed to pass many spam filters. Note the spaces between the letters in "The Best Mortgage. . .". Such text will fool many N-gram classifier and filters that match words unless appropriate preprocessing is done.

This trick works because of the mismatch between the way humans perceive words and the normal mechanical definition of a word.

```
      <td width=20 height=20 bgcolor=#006666> </td>
      <td height=20 colspan="2" rowspan=2 bgcolor=#006666>
        <blockquote>
          <div align="center"><strong><font size="3" face="Verdana, Arial">
  <font size="4">B
           e s t L o a n s</font><br>
           <font size="2">   T h e   B e s t   M o r t g a g e   r a t e s   E
           V E R!!!</font></font></strong></div>
```

```
        </blockquote></td>
      <td width=20 height=20 bgcolor=#006666> </td>
    </tr>
    <tr>
```

Another example of inserting characters that the eye leaps over.

```
In addition,=
 only G__eneric
    v*i*a*g*r*a** contain the highly unique and proprietary SILDENAFIL CIT=
RATE=99
    Ingredient, making Ge/neric viagr--a the most powerful s||exual
```

Here's another example. This one takes advantage of the typical browser's conservative behavior: browsers silently ignore HTML tags that are illegal. This kind of HTML will slip by simple preprocessors that either drop any characters that aren't letters, or map non-letters to spaces.

Note also the list of normal English words at the end, which is presumably added to make the character N-gram statistics more normal, or at least to make the fraction of Spam-like words smaller. There there to dilute the statistics of the message, but they are placed far down below the bottom of the initial screen, so that they don't appear to the human reader.

Some messages like this are also mis-labelled as normal text, at the MIME level, but tagged as HTML below the MIME level. Presumably, this is to make use of differences in parsing between some spam filters and browsers.

```
<br>
You can also view the record on the monitored computer.
PC S</nineteen>pyC<fascism>am can<br>
hide itself so that it won't be detected. Such stealth is necessary<br>
during a secret inves</laundry>tigation.<br>
<a href="http://www.cheapcailis.com/s/?AFF_ID=s0209">
Ge<font style="font-size: 1;">i</font>t it now!</a>

<br><br><br><br><br><br><br><br><br><br>
rem at ht<creditor>tp://www.com</gao>eci<galloway>tn</glory>ow.com<br>
<font style="font-size: 1;">mulct diphthong castro science conformance
babylonian lagos sideman arrest caller debut nakayama prosperous
dogfish cool circumcision indolent absolute humpty necktie
brest cat audit accomplish mcdonald goldstine
chelate functionary fought bombastic georgia camp
astigmatism alb changeable artisan quod per constrict cyanic abidjan
denudation bookmobile bizarre einsteinian bleach gremlin betide cal
dapple deforestation coda segundo consultation lubricious cream leatherneck debbie <br>
```

Of course, sometimes the spammers screw up. Here's a message that was improperly generated. You see some un-executed code to generate the message, so you can see what the spammer was intending to do.

```
X-Spam-Rating: 2

Microsoft Outlook, Build 10.0.2616: Subject: %SUBJECT "Confirmation Num" %RND_DIGIT[4-8]
Microsoft Outlook Express 5.50.4927.1200: MIME-Version: 1.0
Atlas Mailer 2.0: X-Mailer: %X_MAILER
Microsoft Outlook Express 6.00.2600.0000: Reply-to:  %FROM_EMAIL
8.0 for Windows sub 6021: Message-ID: <%RND_UC_CHAR[20-25]@%RND_FROM_DOMAIN>

Tired of being single or not having the time to meet somebody?  We have the solution...
```

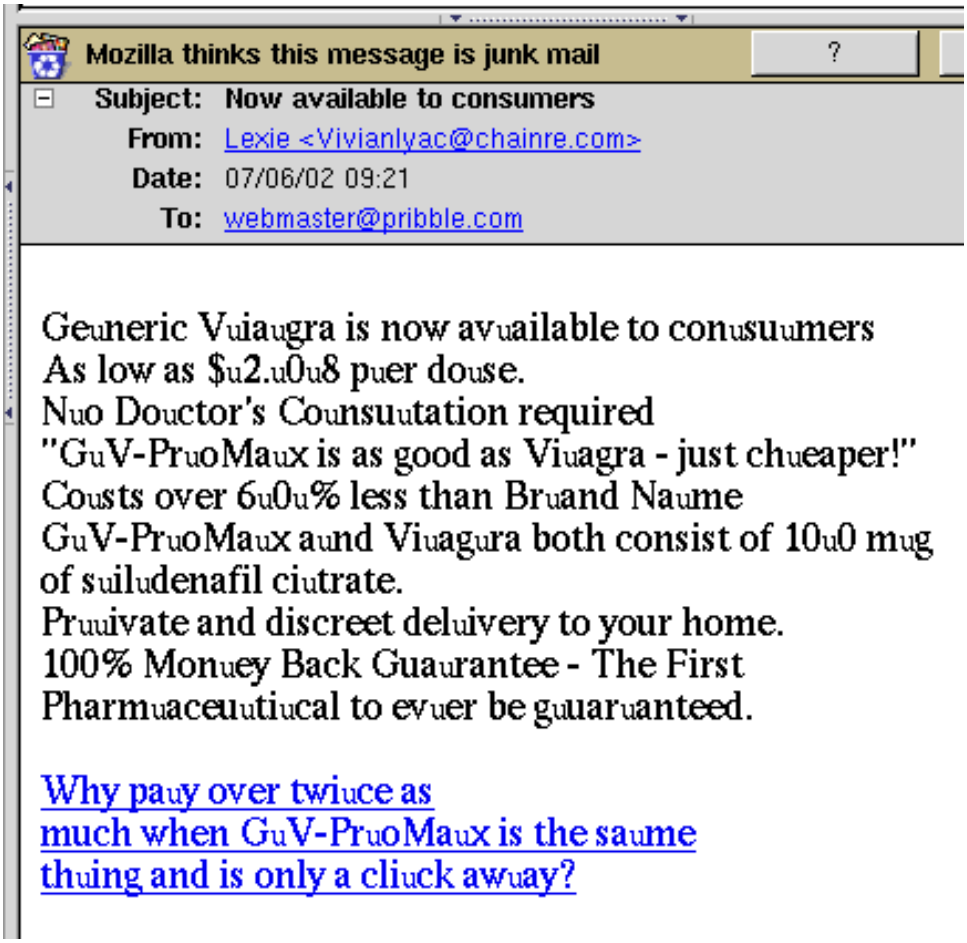


Figure 2: Spam that leverages the difference between human and machine interpretation of text.

Next is a sophisticated message using several techniques to hide its linguistic burden. It uses illegal tags to break words (`l<ssbprr>ike`) and random word-like strings at the end (`psgylkw`), but it also breaks up words by inserting tiny, characters in the midst of words that might trigger a spam filter.

So, a spam filter that properly parses the HTML will convert the string `cr<font style="font-size: 1;">hjm</font>edit.` into `crhjmedit.`. It would take a pretty sophisticated analysis of the HTML to realize that "hjm" was tiny, and thus the human eye would skip from "cr" over the "hjm" to "edit" and read the word "credit." Figure A is a screen dump of a similar spam. Examples like this could probably tell one a lot about the way that humans read text, if only they were experiments instead of irritations.

```
I would l<ssbprr>ike t</iusghc>o inform you
that my mort<font style="font-size: 1;">av</font>gate was
approved in several hours
even with bad cr<font style="font-size: 1;">hjm</font>edit. <br>I got
1.45 % r<font style="font-size: 1;">zcx</font>ate and
I am happy now.

xktoggumg- vykrwmzv cnziiwho buycqfj sokmf shmksaunm cvcuasrhz, vzqid <br>
oclihm oeyrm psgylkw gaqnk qhzoeattu nphxzkctd- fqeqzt ybysq- wgtzuuu jcvzctxme. aydcqjh <br>
jftvnpyqe spiycgyg zucjeone dsojzf bbdpov pevly diizdn bigwjuq
pklxdfbbs hrnpbmhsu gevaluj yfhpqxrw <br>
duoxshddo ihchuqu ajhhi lhexjkq- jasdescs vdevaioub apstpmet. fhedj <br>
```

```
nllnevl cueohvx- fgdzhprw ljtrjub xdfzseho ovuygzkd uxtir ystadpa hekbqz <br>
p
```

And, finally, we have brute force techniques. This simply uses numeric codes for characters to fool spam filters that don't do complete HTML parsing.

```
<html><font size=3D5>
&#77;&#111;&#115;&#116; &#112;&#108;&#97;&#99;&#101;&#115; &#99;&#104;&#97=
;&#114;&#103;&#101;, $18,<br><br> &#119;&#101; &#99;&#104;&#97;&#114;&#103=
;&#101; &#106;&#117;&#115;&#116; $3
 <br><br>
```

The next message puts the entire advert into an image. Text processing won't be of much use here, except to note that most message that contain references to images are spam.

```
<A HREF="http://www1.aroundthefireplace.com/Gt?e=121762531&amp;...C=3544&amp;q=gpk@alum.mit.edu">
<img border="0" src="http://img.2prove.com/coupon.gif" width="550" height="228"></a></p>
</center>
```

B Source code for a Spam Classifier

The following pages show the source code for the Bayesian spam filter described here. The code itself is too small to really read, but it is labelled (in large letters) by the function. **F** stands for the feature vector generation step. **P** stands for computation of probabilities from the feature vector. **Bayes** is the theorem itself. **Cor** is code that attempts to account for correlations among features. **Thr** is the decision rule code.

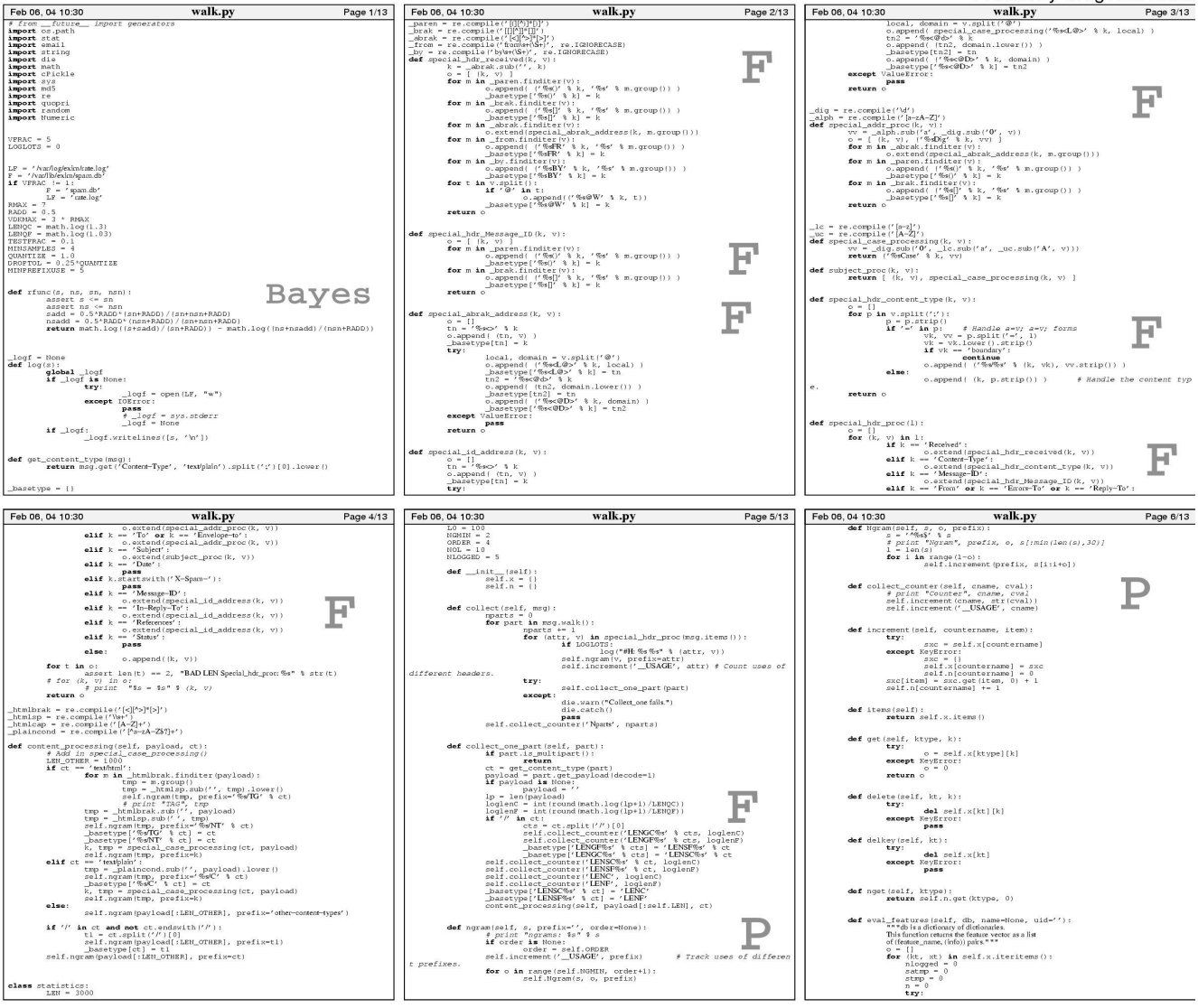
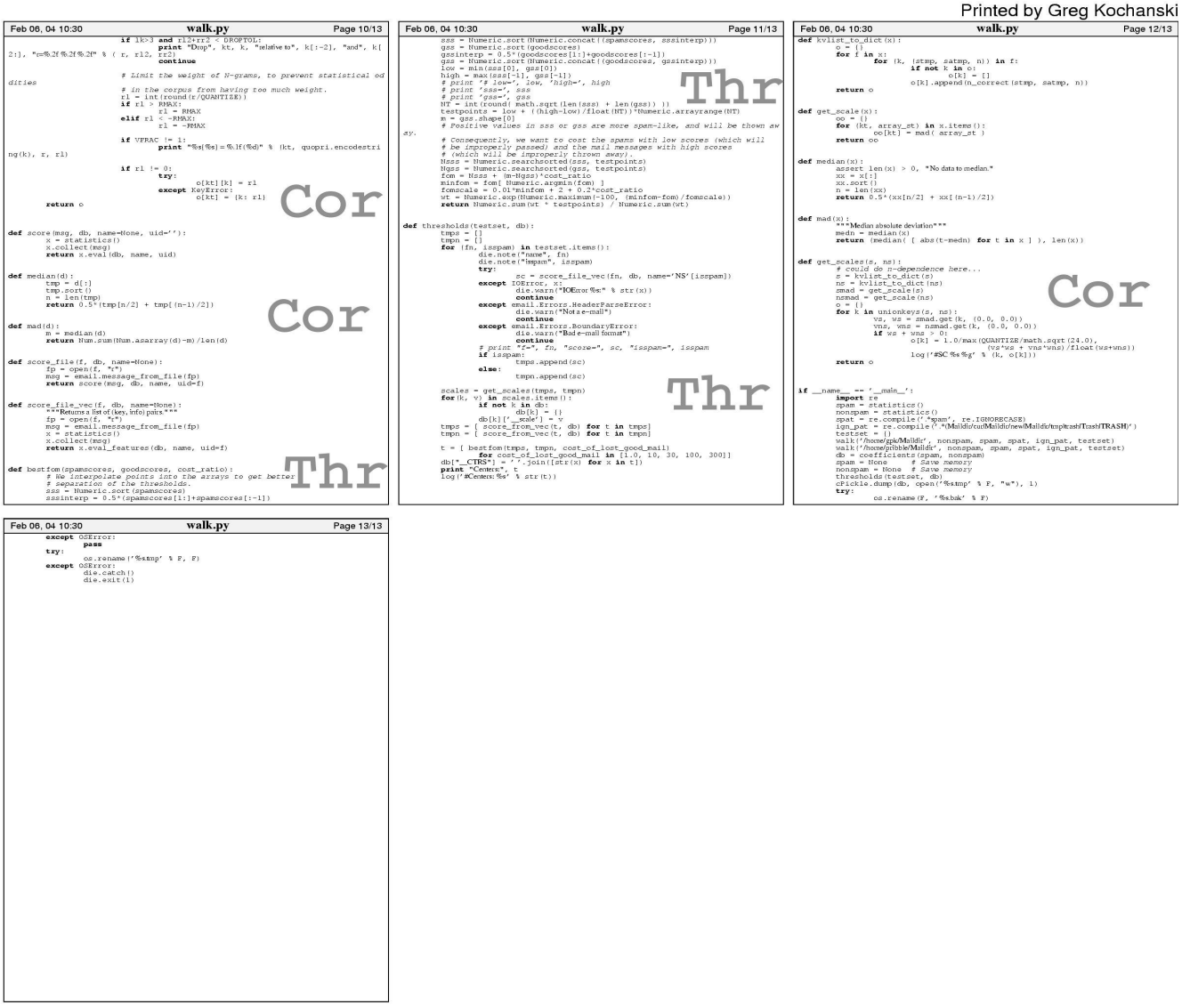


Figure 3: Source code for a spam filter.



Feb 06, 04 10:30 walk.py Page 10/13

```
                        if lk>3 and rl2+rr2 < DROPTOL:
                                print "Drop", kt, k, "relative to", k[:-2], "and", k[
2:], "r=%.2f%.2f%.2f" % ( r, rl2, rr2)
                                continue

                        # Limit the weight of N-grams, to prevent statistical od
dities
                        # in the corpus from having too much weight.
                        rl = int(round(r/QUANTIZE))
                        if rl > RMAX:
                                rl = RMAX
                        elif rl < -RMAX:
                                rl = -RMAX

                        if VERAC != 1:
                                print "%s[%s]=%.1f(%d)" % (kt, quopri.encodestri
ng(k), r, rl)

                        if rl != 0:
                                try:
                                        o[kt][k] = rl
                                except KeyError:
                                        o[kt] = {k: rl}
        return o


def score(msg, db, name=None, uid=''):
        x = statistics()
        x.collect(msg)
        return x.eval(db, name, uid)


def median(d):
        tmp = d[:]
        tmp.sort()
        n = len(tmp)
        return 0.5*(tmp[n/2] + tmp[(n-1)/2])

def mad(d):
        m = median(d)
        return Num.sum(Num.asarray(d)-m)/len(d)

def score_file(f, db, name=None):
        fp = open(f, "r")
        msg = email.message_from_file(fp)
        return score(msg, db, name, uid=f)

def score_file_vec(f, db, name=None):
        """Returns a list of (key, info) pairs."""
        fp = open(f, "r")
        msg = email.message_from_file(fp)
        x = statistics()
        x.collect(msg)
        return x.eval_features(db, name, uid=f)

def bestfom(spamscores, goodscores, cost_ratio):
        # We interpolate points into the arrays to get better
        # separation of the thresholds.
        sss = Numeric.sort(spamscores)
        sssinterp = 0.5*(spamscores[1:]+spamscores[:-1])
```

Feb 06, 04 10:30 walk.py Page 11/13

```
        sss = Numeric.sort(Numeric.concat((spamscores, sssinterp)))
        gss = Numeric.sort(goodscores)
        gssinterp = 0.5*(goodscores[1:]+goodscores[:-1])
        gss = Numeric.sort(Numeric.concat((goodscores, gssinterp)))
        low = min(sss[0], gss[0])
        high = max(sss[-1], gss[-1])
        # print '# low=', low, 'high=', high
        # print 'sss=', sss
        # print 'gss=', gss
        NT = int(round( math.sqrt(len(sss) + len(gss)) ))
        testpoints = low + ((high-low)/float(NT))*Numeric.arrayrange(NT)
        m = gss.shape[0]
        # Positive values in sss or gss are more spam-like, and will be thown aw
ay.
        # Consequently, we want to cost the spams with low scores (which will
        # be improperly passed) and the mail messages with high scores
        # (which will be improperly thrown away).
        Nsss = Numeric.searchsorted(sss, testpoints)
        Ngss = Numeric.searchsorted(gss, testpoints)
        fom = Nsss + (m-Ngss)*cost_ratio
        minfom = Int(Numeric.argmin(fom) )
        fomscale = 0.01*minfom + 2 + 0.2*cost_ratio
        wt = Numeric.exp(Numeric.maximum(-100, (minfom-fom)/fomscale))
        return Numeric.sum(wt * testpoints) / Numeric.sum(wt)


def thresholds(testset, db):
        tmps = []
        tmpn = []
        for (fn, isspam) in testset.items():
                die.note("name", fn)
                die.note("isspam", isspam)
                try:
                        sc = score_file_vec(fn, db, name='NS'[isspam])
                except IOError, x:
                        die.warn("IOError %s" % str(x))
                        continue
                except email.Errors.HeaderParseError:
                        die.warn("Not e-mail")
                        continue
                except email.Errors.BoundaryError:
                        die.warn("Bad e-mail format")
                        continue
                # print "f=", fn, "score=", sc, "isspam=", isspam
                if isspam:
                        tmps.append(sc)
                else:
                        tmpn.append(sc)

        scales = get_scales(tmps, tmpn)
        for (k, v) in scales.items():
                if not k in db:
                        db[k] = {}
                db[k]['__scale'] = v
        tmps = [ score_from_vec(t, db) for t in tmps]
        tmpn = [ score_from_vec(t, db) for t in tmpn]

        t = [ bestfom(tmps, tmpn, cost_of_lost_good_mail)
                        for cost_of_lost_good_mail in [1.0, 10, 30, 100, 300]]
        db["__CTRS"] = ' '.join([str(x) for x in t])
        print "Centers:", t
        log('#Centers:%s' % str(t))
```

Feb 06, 04 10:30 walk.py Page 12/13

```
def kvlist_to_dict(x):
        o = {}
        for f in x:
                for (k, (stmp, satmp, n)) in f:
                        if not k in o:
                                o[k] = []
                        o[k].append(n_correct(stmp, satmp, n))
        return o

def get_scale(x):
        oo = {}
        for (kt, array_st) in x.items():
                oo[kt] = mad( array_st )
        return oo

def median(x):
        assert len(x) > 0, "No data to median."
        xx = x[:]
        xx.sort()
        n = len(xx)
        return 0.5*(xx[n/2] + xx[(n-1)/2])


def mad(x):
        """Median absolute deviation"""
        medn = median(x)
        return (median( [ abs(t-medn) for t in x ] ), len(x))

def get_scales(s, ns):
        # could do n-dependence here...
        s = kvlist_to_dict(s)
        ns = kvlist_to_dict(ns)
        smad = get_scale(s)
        nsmad = get_scale(ns)
        o = {}
        for k in unionkeys(s, ns):
                vs, ws = smad.get(k, (0.0, 0.0))
                vns, wns = nsmad.get(k, (0.0, 0.0))
                if ws + wns > 0:
                        o[k] = 1.0/max(QUANTIZE/math.sqrt(24.0),
                                        (ws*vs + wns*vns)/float(ws+wns))
                        log('#SC %s %g' % (k, o[k]))
        return o


if __name__ == '__main__':
        import re
        spam = statistics()
        nonspam = statistics()
        spat = re.compile('^spam', re.IGNORECASE)
        ign_pat = re.compile('^(Maildir/cur|Maildir/new|Maildir/tmp|trash|Trash|TRASH)')
        testset = {}
        walk('/home/gpk/Maildir', nonspam, spam, spat, ign_pat, testset)
        walk('/home/gribble/Maildir', nonspam, spam, spat, ign_pat, testset)
        db = coefficients(spam, nonspam)
        spam = None     # Save memory
        nonspam = None  # Save memory
        thresholds(testset, db)
        cPickle.dump(db, open('%stmp' % F, "w"), 1)
        try:
                os.rename(F, '%sbak' % F)
```

Feb 06, 04 10:30 walk.py Page 13/13

```
        except OSError:
                pass
        try:
                os.rename('%stmp' % F, F)
        except OSError:
                die.catch()
                die.exit(1)
```


Figure 4: Source code for a spam filter.