# 6.891 Final Project

# Email Filtering: Machine Learning Techniques and an Implementation for the UNIX Pine Mail System

**Yu-Han Chang**
M.I.T. A.I. Lab & L.C.S.
Cambridge, MA 02139
ychang@ai.mit.edu

December 10, 1999

## 1. Introduction

The problem of email filtering is a very practical one. As our lives become ever increasingly tied to the online world, the volume of email coming into our inboxes has also been increasing steadily. The need to make sense of all this information is critical if one is to retain their sanity amid spam, smut, silliness, and semblances of sense. The current solution usually consists of using a mail filtering program that can sort incoming mail based on user-specified rules. Many hours go into crafting filtering rules that can sort out mail from specific addresses or with specific subject lines. But keeping track of all these rules is yet again enough to drive one up the wall. New junk mail from new addresses keep coming through the door, and real correspondents change email addresses all the time.

Here is where machine learning comes to the rescue. If we can train a machine, or rather a computer program, to learn what email should go where, then we would no longer have to deal with the hassle of sorting email ever again. Junk mail would be discarded. Personal correspondence could be filtered into folders specific to the correspondent. Business or coursework mail could go to their respective folders. Life would again be simple, or at least organized.

Preferably, the machine would be an online-learner that can incrementally update its filtering abilities over time. Thus, when new types of email arrive, it can update the way it filters such mail accordingly. It learns as it watches us file this mail, and it can learn from its own mistakes. With such an intelligent mail filtering program at our side, we can finally just sit back, relax, and read our mail.

This paper presents techniques to achieve this goal based on well-understood machine learning algorithms. It also describes our implementation, which we call `sortmail`, that we have written for users of the Pine mail system under UNIX. Section 2 provides the theory, and Section 3 describes the implementation.

## 2. Techniques

Machine learning offers a myriad of different methods to go about classifying various objects into their respective categories. From estimating probability distributions to regression techniques to reinforcement methods to finding separators in feature spaces, we have a wide ranging set of tools at our disposal.

In our case, we must narrow our domain to the problem of text classification. Text classification takes a document consisting of a set of words and tries to categorize the document among a specified set of topics, or classes. So in our case, we thus classify each given email as belonging to a particular folder. However, we must realize that an email is not exactly a set of words, and we must decide how to extract the relevant words from the given email. This in itself is a topic of much research. We found these techniques rather interesting, and so Section 2.1. is devoted to a brief summary of these methods. The next two sections then outline two different machine learning techniques for text classification.

### 2.1. Feature Extraction from Email

An email message does not fall neatly into the domain of text classification. First of all, it contains much extraneous information that is irrelevant (or at least redundant) to our text classification problem. For example, the major portion of the email header need not be considered part of our document. Lines such as:

```
Received: from lychee ([170.1.11.124])
        by apple.webjuice.com (8.9.3/8.9.3) with SMTP id KAA43232;
        Wed, 8 Dec 1999 10:54:40 -0800 (PST)
        (envelope-from fyang@webjuice.com)
```

do not convey much useful information to us for classification purposes. We might imagine that some of these IP addresses contain useful information in the sense that they suggest which user has sent us the email. This in turn might suggest a particular classification since most messages from this particular user are filed in a particular folder.

However, we observe that most email programs hide this information from the user. Thus, if we are trying to mimic the user's preferred method of classification, we can also ignore this information. Even if it contains relevant information in the sense just described, this information must have strong correlation to other user-observe-able elements of the email message. This is because the user only uses the observed information for classification, and thus if the hidden information leads to correct classification, the only reason is because it is highly correlated to the observed inputs. Essentially, our learning problem seeks to learn the user's classification function, mapping documents to folders. If our learned function is equivalent to the user's true function, then it must mean that the inputs of our function can be transformed into the inputs of the user's function, and vice versa. Hence, we only use observed information in our learning problem. We thus use only the `To:`, `From:`, and

`Subject:` fields of each message's header as inputs for our classifier. We also include the entire text of the message body.

Once we have done this, there are still many refinements we can make to the document which will facilitate the learning process. Since we are classifying messages based on the existence of certain words in each message, we can simplify our problem by eliminating those words which appear with high frequency in all documents, regardless of classification. These are words such as "the", "he", or "it." Common pronouns, modifiers, simple abverbs and verbs all fall into this category. We call this list of words the *stoplist*. Since these words appear in almost all documents, we can ignore them without losing any information. Essentially this is just noise that does not help with classification. Andrew McCallum's text processing package `bow` contains a stoplist of 524 words [McCal], which we utilize in our implementation. When we encounter any of these words in the process of lexing the message, we can ignore the word. This enables us to concentrate on learning the words with actual significance.

Another means for refining the text classification problem is to use a process known as *stemming*. The purpose of stemming is to treat words with the same root as being equivalent. This eliminates the irrelevant differences between a document containing the word "exasperating" from a document containing the word "exasperate," for example. Both documents might concern the topic of final projects, but without stemming, they may have different classifications depending on the appearance of "exasperate" and "exasperating" in other documents. For example, suppose we had only two emails in our training set, one regarding Microsoft Windows with the words "exasperate," and one regarding someone's significant other that contains the word "exasperating." Then we receive an email for classification regarding the significant other but contains the word "exasperate." We would incorrectly classify the email in the Microsoft Windows folder if we did not use *stemming*. In this simplified example, stemming would at least make the classification ambiguous, forcing us to rely on the other words in the messages for clues. In our implementation, we use the *Porter stemming algorithm*, [McCal] which solves this problem by transforming all equivalent words into a token that is represented by their root form only. For example, the words `serve`, `service`, `serves`, and `served` would all be transformed into their root `serve`.

Finally, we note that it is important to lex the input stream correctly so that we preserve strings like a user's email address as one token. Email addresses are important since we recognize that people often sort messages based on the sender. Thus, we need to use a lexer that does not separate words using punctuation symbols such as `@` or `.`, or which takes special case to preserve email address. There are also many other cases such as web addresses or proper nouns that may exhibit this problem. In our implementation, we simply use a lexer that separates token by white space. This trades off the ability to match email addresses with its inability to recognize that similar tokens such as `http://www1.yahoo.com` and `http://www2.yahoo.com` should in fact be treated the same token.

These methods all serve to simplify our learning problem and to reduce to amount of noise inherent in the problem. Applying these methods to each email message transforms the message into an acceptable input document for our text classifier. Each message can now be viewed as a collection of words, and we can examine the frequencies with which these words appear in documents of various classes. However, we have one further dilemma. The English language contains tens of thousands of words, making the learning problem inefficient, if not intractable. Moreover, there are an infinite number of possible email addresses, web addresses, proper nouns, and the like. We would be left with a very inefficient classifier if we kept track of every word that our classifier ever encounters. This would lead to a unwieldy database that lists the frequency with which each of these words appears in documents of each class. Thus we need a good method for feature extraction.

As shown by Yang and Pederson [YP97], document frequency is a reasonable method for feature extraction in text classification problems. This uses a feature set that only includes words that appear with high frequency, which intuitively makes sense. We do not need to keep track of words that appear only once, since they are likely not to convey much information about that particular class. This assumes, of course, that we have many example points for each class. For example, say in our Microsoft Windows folder, we have one occurrence of the word `rigmarole` in one of the 100 messages in the folder. Since it occurs so infrequently, it most likely does not convey much useful classification information.

To make use of this method, we define the *age* of a word to be the number of documents we have classified since the first time the word was encountered. We remove a word when the value $\log_2 age - 1$ becomes greater than the document frequency of the word. In our implementation, this typically retains 2000-8000 words in our feature set. Now we have a complete method for transforming a given email message into a tractable set of input features. The feature space has a dimensionality of 2000-8000 dimensions. We then have pretty much the complete range of learning algorithms that we have covered in 6.891 available to us. Depending on implementation, some algorithms may not be practical for a problem of this size. We thus focus on two methods: Bayesian estimation, and Linear Separators. In practice, we observe that the methods work fairly well, leading to good classification rates.

## 2.2. Naïve Bayes Text Classification

As we have seen, Bayesian estimation is a simple technique for estimating the probability distribution of classes over feature data inputs. In the text classification domain, this is often referred to as Naïve Bayes text classification. It uses Bayes Rule to translate between feature data frequencies and the probability distributions on classes. [Dud97] In the text classification domain, Naïve Bayes thus estimates the probability that a given document belongs to a certain class by evaluating the probability that words will appear in a document of that class. The document is then classified according to the class in which it is most likely to belong. The probability of words appearing in documents of a certain class is computed from our training data. The training data consists of a set of labeled documents, each labeled with its correct class.

In our case, our training data consists of a user's previously sorted email. Each message has been stored in a particular folder, which denotes the document's class. As we discussed in the previous section, our message can be transformed into a set of features, specifically a set of words that appear in the message. Thus, given a particular set of messages in folders, we can calculate the probability that a word appears in an email from a particular folder. We can then apply Bayes Rule to derive the probability of a folder given a particular set of features.

Specifically, we have a set of classes, of folders, $F = \{ f_1, \ldots, f_m \}$. Each time we wish to classify an email document *e*, which we can view as a set of features, or words, $W = \{ w_1, \ldots, w_n \}$. As discussed in the previous section, for each email *e*, we only test the existence of currently tracked words to use as features. This set is *W*, and can be updated over time. The document *e* thus contains a subset of the words in *W*. We would like to calculate the probability that this document *e* is in each of the different folders $f_i$ , i.e.

$$P(f_i \mid e) = \frac{P(f_i) \prod_{w_j \in e} P(w_j \mid f_i)}{P(e)}$$

where $P(f_i)$ is the prior probability that any random email belongs to the class $f_i$, $P(w_j \mid f_i)$ is the probability that a randomly chosen word from an email in folder $f_i$ is the word $w_j$, and *P(e)* is the probability that a random email is the document *e*. With these probabilities, we can then classify the email *e* into the folder $f_i$ with highest probability $P(f_i \mid e)$. Since we are only using these $P(f_i \mid e)$ for comparison purposes, we can drop the scaling factor *P(e)*. Thus, to find the best folder *f**, we simply need to compute

$$f^* = \arg\max_{f_i \in F} P(f_i) \prod_{w_j \in e} P(w_j \mid f_i)$$

Calculating $P(f_i)$ is straightforward, and each $P(w_j \mid f_i)$ may also be calculated directly from the training data, or we can use the *m*-estimate. This estimate is useful when we are dealing with sparse data, and calculates $P(w_j \mid f_i)$ as

$$P(w_j \mid f_i) = \frac{n_j + 1}{n + |W|}$$

where $n_j$ is the total number of occurrences of the word $w_j$ in all of the message in folder $f_i$, *n* is the total number of words in the all the emails in folder $f_i$, and $|W|$ is the dimensionality of our feature space, i.e. the number of words we are tracking for classification purposes. This estimate captures the essential fact that we are trying to capture in the learning process. It says that even if the occurrences of a particular word $w_j$ are high only in a few messages in a folder, but that on average the word does not appear in a given message in the folder, we still favor documents containing this word $w_j$ to be in folder $f_i$.

The nice thing about this method is that it easily adapts to our online learning process. Each time we filter a new message, it is easy to update $W$ and to update $n_j$. Also, if we ever make a mistake, it is easy to adjust these values so that they take into account the incorrect classification. That is, we would simply subtract off the occurrences that we previously added, and then add them to the correct folder instead.

Thus Naïve Bayes is a simple and clear method for solving our given learning problem. In Section 3, we will show that in practice, Naïve Bayes works surprisingly well and may be all we really need for email sorting purposes.

### 2.3. Separation Techniques

In the previous section, we used a method which actually estimates the underlying distribution functions for the classes given the inputs. Under separation methods, we forgo this estimation of the distribution and directly solve for the separators between the classes. There are many different techniques for doing this, including linear discriminants, generalized discriminant functions, and Support Vector Machines. We first tried using Support Vector Machines, but after hours of cajoling the code, we gave up in "exasperation." Perhaps, we thought, our data has too many dimensions for this provided code to handle.

So instead we opted for the simplest of the methods, the linear discriminant. As we saw in the problem sets, linear discriminants are designed to only handle two classes of data. However, we also saw that they could be adapted handle multiple classes of data just as a perceptron can be adapted to accomplish this. In fact, we demonstrated that the linear discriminant is functionally equivalent to single layer perceptrons. To handle mutiple classes, we simply connect separate output nodes for each class. These outputs are the result of the transformation on the input vector, which in our cases is a vector denoting the occurrence of each of a set of words in the particular document. We then choose the output node with the highest value, which translates to the class with which we have the most confidence contains the document.

Of course, there is no guarantee that our data will be linearly separable; in fact, we would imagine that it most likely isn't. People tend to receive textually similar emails which they may classify differently. Hence we use the Minimum Squared Error Linear Discriminant. This linear separator minimizes the sum of the least squared errors over the training data. It uses the pseudo-inverse to compute this separator. Best of all, it is a very simple algorithm. For a full description, consult Duda, Hart, and Stork, Chapter 5. [Dud97]

We adapt the MSE Linear Discriminant to our email sorting problem. To be precise, we use the same transformations as described in the previous sections in order to transform each given email message into a well-behaved text document. That is, we apply stemming and utilize the stoplist as described, and we lex the document properly into a set of words. We also use document frequency to whittle our dimensionality down to something sane. We use the 2000 most frequent words to conduct our analysis. Thus, we have a 2000-dimension

input vector. The number of output nodes depends on the number of folders. Note that we do not use the age metric because we are not going to do online learning with linear discriminants. We only calculate the discriminants once on the initial training data. Further tuning will require the re-calculation of the discriminant over the entire new training data, which would then include the newly sorted messages.

Thus, our training data is represented as a matrix Y which each document as a row. We consider each folder in turn. Thus $Y$ is a $n$ x $|W|$ size matrix, where $n$ is the total number of training messages we have pre-sorted in the all of the folders in our training set, and $|W|$ is the number of feature words we are tracking. In our implementation, we have chosen $n$ = 2000. The vector $b$ contains either 1 or –1 depending on the correct folder for the email in that row of $Y$, i.e. $b_i$ = 1 if the message contained in row $i$ of $Y$ should be in folder $f_j$, or $b_i$ = -1 if it should be in any other folder $f_k$. We then solve for the weight vector $a$ that minimizes the squared error:

$$a = \hat{Y}b \qquad \text{where} \quad \hat{Y} = \left(Y^t Y\right)^{-1} Y^t \text{ is the pseudo-inverse of } Y$$

To evaluate a given document e, we simply compute $e^t a$. If the value is positive, then it is classified under folder $f_j$, if it is negative, it is classified under some other folder $f_k$. Once we have computed these vectors a for each pairing of folders, we can then compute all such $e^t a$ and choose the folder with the highest value. We then sort the message into this folder.

# 3. Implementation

The implementation for our email classifier is relatively general, though it was written using the Pine mail system as an example. The only real dependencies it has on Pine is the way Pine stores its mail folders, with a folder being one long text file containing all the messages in the folder. The messages are separated in the usual way, by the "`From `" at the beginning on a newline. We constructed different sorting programs for the methods described in the previous section, though they all use much of the same framework of code. Most importantly, they all use the same methods to transform each email message into a set of input features. The message is then classified and placed into the correct mail folder. The next section describes some of the nitty-gritty details about the implementation, and Section 3.3 presents some of the experimental results we gathered by running these programs. As shown, they do a pretty good job at sorting mail.

## 3.1. Technical Details to Gripe About

The Pine mail system is difficult to work with. Most problematic is the fact that it is not extendable at all. Thus, we needed to devise some method for filtering that did not rely on Pine itself for any information. In this sense, our developed program should be easily reconfigured to any other mailing system.

There are two methods for using `sortmail`, since the only input `sortmail` expects is the text of the message to be sorted, passed into `sortmail` on `STDIN`. The first method is to filter messages before they are placed in the pine inbox; the other is to filter messages once they are already in the user's inbox. The first method calls `sortmail` from the `.forward` file. On most UNIX systems, the `.forward` file is enabled and automatically forwards all the user's incoming mail. By placing the command

```
"|exec /home/y/c/ychang/bin/sort.pl"
```

we can call `sort.pl`, which is the `sortmail` executable for sorting incoming messages. The command also passes the incoming message to `sortmail` on `STDIN`. This allows `sortmail` to classify the incoming message and place in the correct mail folder for later reading.

The second method allows the user to manually call `sortmail` from within pine. We use Pine's pipe command, "`|`", to accomplish this. By using the pipe command, we can pass any selected messages in the current Pine folder into `sortmail` on `STDIN`. The messages are then classified by `sortmail` and placed in the correct folders. The program then displays a screen which shows the folders into which each message has been sorted. If the user wishes to keep the message in the Inbox as well, then no further actions need to be taken. Otherwise, if the user wishes to delete the message from the Inbox and retain only the copy in the sorted folders, pressing "`d`" will delete the messages from the Inbox.

If either method, if `sortmail` classifies an email incorrectly, the user can manually move the message into the correct folder. Every so often, the user should run the script `refile.pl` which scans the mail folders, detecting all of these manually re-filed messages. It treats these messages as having been incorrectly classified by `sortmail` and updates the `sortmail` knowledge base accordingly. There are some other minor features of the user interface, but these are the main functions that are regularly used.

Internally, the core coding framework of the `sortmail` system is based on `ifile`, written by Jason Rennie [Ren98]. Much of the code has been rewritten to support the expanded learning algorithms available in `sortmail`. All of this code is written in the `C` language and uses the text processing packages `bow` and `rainbow` by Andrew McCallum at CMU [McCal] for pre-processing the email messages into well-behaved text documents. Between `ifile` and the `bow` packages, all of the email lexing is taken care of. We implemented the actual learning algorithms that use this input, and we also implemented wrappers in order to manage the emails. On top of the `sortmail` core, we then added several `perl` wrappers which call the `sortmail` core with various arguments depending on the desired functionality. Here is a list of the major functionalities we use, some of which have already been mentioned above, listed by the `perl` wrapper executable name:

- **`sort.pl`** – takes a message or a concatenation of multiple messages from `STDIN`, sorts them into the correct folders by concatenating the messages onto the ends of their respective folders, and updates the knowledge base with updated word frequencies
- **`refile.pl`** – scans the mail folders for incorrectly sorted messages and updates the knowledge base accordingly to reflect the user's preferences.
- **`build.pl`** – builds the initial knowledge base of the probability distributions by analyzing the pre-sorted emails which have already been placed into correct folders by the user.

The Appendix contains a partial listing of the source code. Some of the system dependent coding decisions are described there, since they are not very relevant to the thrust of this paper. One of the major issues is the method by which we detect incorrectly filed messages. The solution was to add a unique message ID the first time we encounter each message and to keep a database listing the pairings of IDs and mail folders. Other similar issues are described in greater detail in the comments of the code.

In the source code listings, we leave out the routines taken from `ifile` or `rainbow`. Please consult the proper references for that code. Alternatively, we will be posting a final version of `sortmail` on the web at `http://www.ai.mit.edu/~ychang` in the near future. The entire source code will be available in the download.

### 3.2. Experimental Results

To test the system, we conducted a variety of different experiments. Cross-validation techniques were used first, but we then wanted to capture the ability of the program to learn new information about the user's preference and also its ability to "extrapolate" from its information about currently known classes to categorize "unknown classes." We will explain what we mean by this shortly.

Because of the nature of email sorting, it is hard to come across different sets of personal email data sets that have been sorted based on different preferences. Thus, given the time span allowed by this project, I concentrated on my personal archive of mail, which in itself is pretty extensive, on the order of ten thousands of messages. I consider this archive to be a rather good representative of the types of emails college students might encounter, mixing personal, class, interview, business, retail, and junk emails. To run these experiments, I sorted a total of 1000 messages mail into twenty-five different folders.

My first round of testing involved simple cross validation. I first used to extract one and train on the rest version of cross-validation, validating on the extracted message. This yielded very good results under both learning algorithms over the entire one thousand message data set. Smaller numbers of folders tended to produce better results, as shown in the table below. However, this sometimes depended on the particular folders I removed from the training set. For example, as shown below, I tried two sets of ten messages. The set with more similar topics (set a) clearly proved to be harder to correctly classify.

| Naïve Bayes | |
|---|---|
| # folders | Accuracy |
| 10 (set a) | 96% |
| 10 (set b) | 92% |
| 20 | 89% |
| 25 | 88% |

| MSE Discriminant | |
|---|---|
| # folders | Accuracy |
| 10 (set a) | 97% |
| 10 (set b) | 91% |
| 20 | 84% |
| 25 | 82% |

In order to further test the learning algorithms, I constructed a hierarchy of classes for my mail. The top tier of the hierarchy are the folders under which each message is stored. I varied the number of folders in my various experiments by simply excluding some of the folders. Underneath the top directory are sub-categories for the emails. For example, under the junk mail folder, I separate the emails into types: random spam, mass forwards and chain mail, advertisements, and building notices. Under each type, I further separate the emails by mailer or specific topic, which usually means separating mail from different cyberstores or different groupings of spam (end of the world spam messages vs. you'll be rich messages for example).

Here is a small subset of the complete hierarchy:

```
6.891
      Final Project
      Problem Sets & Announcements
6.854
      …
Allen
      …
Contact
      …
…

Junk
      Random Spam
      Mass Forwards and Chain Mails
      Advertisements and Solicitations
      Building Notices
Eric Yeh
      …
Taiwan
      Politics
      Jiang's Visit to the U.S.
      Interesting Articles
      Harvard Taiwanese Cultural Society Annoucements
…
```

With this hierarchy, I wanted to test the generalization capabilities of the learning algorithms. That is, I would train the program with only training data from one sub-category of the folder, and then use the another sub-category as input. This is what is meant by extrapolation to unknown classes. The new test data sub-category has not been given to the program yet, so it is essentially unknown. That is, the word frequencies for that category have not yet been incorporated the classifier. We exhibit the results for the junk folder, which were less than spectacular. This is understandable, since the different subcategories intuitively have very different word distributions. Thus, whether we are trying to learn the underlying distribution or the separator between classes, we are going to have a hard time since we have no knowledge about an entire sub-category. For example, advertisements may frequently have the word `sale`, while chain mail might often have the word `forward`. Without having seen any advertisements, we would have no information regarding an occurrence of the word `sale`. We might even categorize it under the folder `activities`, which includes the announcements from various student groups, which sometimes include moving sales.

Removed sub-category shown, used that category as test input.

Naïve Bayes

| removed | Accuracy |
|---|---|
| Ads | 61% |
| Spam | 47% |
| Chain Mail | 15% |
| Notices | 8% |

MSE Discriminant

| removed | Accuracy |
|---|---|
| Ads | 56% |
| Spam | 50% |
| Chain Mail | 13% |
| Notices | 12% |

It is interesting to note the high correlation between our two different learning methods, especially when they seem to follow such different approaches. However, a closer evaluation reveals similarities due to the nature of the data. Essentially our feature space only cares about the existence of certain words in a given document. Thus, each feature is pretty much a binary random variable. While the Bayesian method tries to estimate the probability the variable has of occurring in documents of different types, the discriminant method tries to separate a class based on which words are important to that class. Hence they are doing very similar tasks. The Bayesian method assigns higher probability to the words that are more likely to appear in that document, leading to a higher probability for that class. The discriminant method assigns a larger weight to those same words for that class. Thus we are using the training data in similar ways in both cases.

We note that the example using the folder junk had a particularly high failure rate, but we use it to highlight this point. That is, the sub-categories within junk had very different contents. Thus, since our learning algorithm is learning based upon words and not semantics, it learns "content" rather than some more abstract notion of "worth" of a document. Thus, different types of junk mail will be categorized different based on content, even though they all have zero (or negative) worth to the user. When we ran the sub-category experiment on other folders, we had much higher rates of success. For example, within the Taiwan folder, we had sub-categories of politics, Jiang visit to the U.S., interest stories, and Harvard Taiwanese Cultural Society (TCS) announcements. Obviously, the contents of these sub-categories are much more correlated with one another than in the junk mail case.

Naïve Bayes

| removed | Accuracy |
|---|---|
| politics | 91% |
| Jiang | 100% |
| interest | 73% |
| TCS | 85% |

MSE Discriminant

| removed | Accuracy |
|---|---|
| politics | 90% |
| Jiang | 95% |
| interest | 60% |
| TCS | 73% |

Our success rate improves greatly when we consider sub-sub-categories. Since the success rate is essentially as good as if we trained using the unknown sub-sub-category included for almost all cases, we only exhibit the case with different cyberstores sending junk advertisements. Actually, `sortmail` excels at sorting junk mail from different merchants

but of the general type being an advertisement. Most of the error that is exhibited in our general test cases comes from other folders such as between `journal entries` and `henry`, who happens to be a close friend. Thus we can imagine that much of the content there is similar, and hence the learner sometimes get confused.

Naïve Bayes

| removed | Accuracy |
|---|---|
| Xoom.com | 100% |
| Netbuyer | 100% |
| B&N | 98% |
| Garden.com | 90% |

MSE Discriminant

| removed | Accuracy |
|---|---|
| Xoom.com | 100% |
| Netbuyer | 95% |
| B&N | 95% |
| Garden.com | 88% |

Clearly `sortmail` does a very good job of generalizing between junk mail from different merchants.

We make a note here regarding the dimensionality of the data. For both learning methods, we used a feature space of at least 2000 dimensions. If we decrease this substantially, it leads to drastically reduced learning capabilities, i.e. error rates shoot up. We take an example from the MSE Discriminants case:

MSE Discriminant

| Dimensions | Accuracy |
|---|---|
| 2000 | 82% |
| 1000 | 75% |
| 500 | 39% |
| 100 | 23% |

The extreme case, of course, is when we only keep 100 words, and we need to classify over 25 folders. Still, a classification success rate of 23% means that it is still doing pretty well, considering that random sorting only gets us 4% success in this case. These numbers clearly are highly dependent on the exact data which we are using, and in this case.

We make one final note regarding the online learning capability of the Naïve Bayes method. Based on casual user testing, the program quickly learned new sub-categories of data. Sometimes it took up to 10 re-files for it to see enough examples of that type to overcome some other preferences it had already built up. This number is highly dependent on the total number of messages trained and the frequency of those particular words in these messages of course. Overall, we were pretty satisfied with `sortmail`'s performance and view it as a useful life-simplifying tool.

## 4. Discussion and Future Work

Our `sortmail` program provides a useful tool for fairly accurate mail sorting. For most users, the naïve bayes classification scheme should be sufficient for general mail sorting purposes. Furthermore, the Bayesian method has the advantage of being an online learning algorithm. For most users, this is an important feature, since it allows the learning algorithm to progressively adapt to new preferences over time. A major reorganization of the mailboxes will still need a recomputation of the word frequency statistics in the new folders of course.

Some directions for future work include using different underlying machine learning algorithms to do the learning. We tried Support Vectors Machines, but couldn't get them to work. We'd like to try again. Also, neural nets would be interesting to test. Neural nets would have the advantage of being capable of online learning. This is built into the backward propagation process of adjusting the weights. Thus we can implement an online learning system similar to the one we constructed using the Bayesian estimation methods.

In addition to new implementing new learning algorithms, we might also try pre-processing incoming messages. That is, we can invoke more clever means of distilling information from the message. For example, we could extract message identifiers and relate messages that are "`In-Reply-To:`" another message.

Finally, some more work needs to be done to streamline the operation of `sortmail` in conjunction with Pine. Preferably we can modify the code so that messages which were not filtered will also be detected by `sortmail` so that the knowledge base can be updated accordingly. Currently `sortmail` only handles messages that are passed to it for sorting and only uses these messages to do online learning. Adapting `sortmail` to work with other mail programs is another goal.

## References

[Dud97] Duda, Richard O., Peter E. Hart, and David G. Stork. Pattern Classification, 2nd Edition. Pre-publication (to be published 1998), John Wiley & Sons, Inc.

[Joa98] Joachims, Thorsten. "Text categorization with support vector machines: Learning with many relevant features." In *European Conference on Machine Learning*, 1998.

[McCal] McCallum, Andrew. CMU. `http://www.cs.cmu.edu/~mccallum/bow/rainbow/`

[Ren98] Rennie, Jason. "ifile: An Application of Machine Learning to E-Mail Filtering." CMU, December, 1998.

[Viola] Viola, Paul. 6.891 Lectures & Lecture Notes, Fall 1999.

# Appendix

Below is a listing of a portion of the source code needed for `sortmail`. Extraneuous portions of the code such as those for log messages or error messages have been expunged. Explanations and code decisions are described in the comments. The complete source code will be available for download at `http://www.ai.mit.edu/~ychang/`.

### sort.pl

```
#!/usr/local/bin/perl

# sortmail helper script
# called from .forward or from within pine as a wrapper to sortmail's filtering
# functionality
#
# by Yu-Han Chang <ychang@ai.mit.edu>

# accepts a message to be filtered on standard input
# and stores that message to the location determined by
# ifile.

$sortmail_binary = "/home/y/c/ychang/sortmail/sortmail";
$maildir = "/home/y/c/ychang/mail/";
$sortmail_args = "-h";
$inbox = "/home/y/c/ychang/mail/inbox";
$countfile = "/home/y/c/ychang/.sortmail.id.count";
$m2ffile = "/home/y/c/ychang/mail/.msg.to.folder.list";

&parse_args();

foreach $arg (@ARGV) {
    if ($arg =~ /^--help/ || $arg =~ /^-h/) { &print_usage; }
}

# get environment information
$home_dir = $ENV{"HOME"}."/";
if ($tmp2_file) {
    print TMP2 "home directory is $home_dir\n";
}

# update accuracies file ~/.sortmail_accuracy
$acc_file = $home_dir.".sortmail_accuracy";

$filters = 1;
$refiles = 0;

if (open(ACC, $acc_file))
{
    $line = <ACC>;
    $line =~ m/filters\s*=\s*(\d*)\s*refiles\s*=\s*(\d*)/;

    $filters += $1;
    $refiles += $2;

    $accuracy = int(($filters - $refiles)/$filters * 10000)/100;
```

```
    close(ACC);
}

if (open(ACC, "> $acc_file"))
{
    print ACC "filters = $filters  refiles = $refiles\n";
    if ($accuracy) { print ACC "Accuracy = $accuracy \%\n"; }
    close(ACC);
} else {
    print TMP "Was not able to write accuracies file\n";
}

# pipes message on stdin to the sortmail program, parses output

# Temporary file for storing a copy of incoming message
$tmp_file = "/tmp/ifile".(time() % 100000);

open(TMP, "> $tmp_file")
    || die "Could not open $tmp_file: $!\n";
@message = <STDIN>;
print TMP @message;
close(TMP);
chmod(0600, $tmp_file);

$command = "$sortmail_binary $sortmail_args --query-insert $tmp_file |";
open(IFILE, $command)
    || die "Could not execute \"$command\": $!\n";
@query_results = <IFILE>;
close(IFILE);

unlink $tmp_file;

# Temporary file for testing the writability of a directory
$tmp_file = ".ifile".(time() % 100000);

## find the best matching folder that doesn't have a .skip_me file
$best = shift(@query_results);
# print "Trying $best folder\n";
$best =~ m/^(\S+)/;
$best_folder = $1;


# Use the first folder which doesn't have a .skip_me file and which we
# can write to.
while ((-f $maildir."/".$best_folder."/.skip_me" && (@query_results > 0))
       || !(open(FOO, "> ".$maildir."/".$tmp_file))) {
    $best = shift(@query_results);
    # print "Trying folder $best";
    if ($best =~ m/^-+$/) { last; }
    $best =~ m/^(\S+)/;
    $best_folder = $1;
}

if (@query_results <= 0) {
    $best_folder = $inbox;
} else {
    close(FOO);
    unlink($maildir."/".$tmp_file);
}

# place file into the correct folder, but first add
# a unique identifier tag to the message
```

```
# get current count
open(COUNTFILE, $countfile);
$count = <COUNTFILE>;
close(COUNTFILE);
chop $count;
$count++;
open(COUNTFILE, "> $countfile");
print COUNTFILE "$count\n";
close(COUNTFILE);

# store message
open(VER, "$sortmail_binary --version |")
    || die "Error: could not execute $sortmail_binary: $!\n";
$version = <VER>;
chop $version;
close(VER);

if ($tmp2_file) {
    print TMP2 "$version => $best_folder\n";
}

print "\n\n$version => $best_folder\n";

open(FOLDER, ">> $maildir$best_folder")
    || die "Error: could not open mail folder $best_folder to write.\n";
$print_filter = 0;
foreach $line (@message)
{
    # Add the X-filter header at the end of the header section
    if ($line =~ m/^\n$/ && (!$print_filter))
    {
        print FOLDER "X-filter: $version => $best_folder\n";
        print FOLDER "X-sortmail: $count\n";
        $print_filter = 1;
    }
    # Rename old x-filter header if we are still in header section
    $line =~ s/^x-filter: /Old-x-filter: /i if (!$print_filter);
    # Make sure each line ends with feed-line
    $line .= "\n" if ($line !~ m/\n$/);

    print FOLDER $line;
}

# Print X-filter header now if we never did so before
print FOLDER "X-filter: $version => $best_folder\n" if (!$print_filter);
close(FOLDER);

# now print out id tag to appropriate msg list file so that we can
# later track this msg's movement between folders if user refiles.

open(LIST, ">> $maildir\.$best_folder\.msglist");
print LIST "$count\n";
close(LIST);

open(M2F, ">> $m2ffile");
print M2F "$count $best_folder\n";
close(M2F);

# end sort.pl
```

## refile.pl

```
#!/usr/local/bin/perl

# script to run periodically to check whether emails had been filed
# correctly.  looks thru log files to see where emails were filed and
# checks to see if they are still there.  if not, it updates the
# .idata stats file accordingly.

# by Yu-Han Chang <ychang@ai.mit.edu>


$sortmail_binary = "/home/y/c/ychang/sortmail/sortmail";
$sortmail_args = "-h";
$maildir = "/home/y/c/ychang/mail/";
$home_dir = $ENV{"HOME"}."/";
$m2flist = "/home/y/c/ychang/mail/.msg.to.folder.list";
$tmpfile = "/home/y/c/ychang/.sortmail.refile.tmp";
$list_creation_binary = "/home/y/c/ychang/6.891/make_lists.pl";
$countfile = "/home/y/c/ychang/.sortmail.id.count";

# keep track of total number of refiles so we can update accuracies file
$num_refiles = 0;

# open maildirectory

# we need to check which msg's have moved since the last time we ran
# this script.  that is, we need to identify all msg's that were
# improperly sorted

$true = opendir(MAILDIR, "$maildir");
@folders = readdir(MAILDIR);

# get rid of "." and ".." files
shift(@folders);
shift(@folders);

# open msg to folder translation table and store in memory
open(M2F, $m2flist);
@m2f = <M2F>;
close(M2F);

foreach $folder (@folders) {

    if (($folder =~ m/^\..*/) || (-f "$maildir".".$folder".".skip_me")) {
        print "skip $folder\n";
    }
    else {
        # get list of msgs that are supposed to be here
        open(LIST, "$maildir\.$folder\.msglist");
        @list = <LIST>;
        close(LIST);

        open(FOLDER, "$maildir$folder");
        print "examining folder $folder...\n";

        # flag if we are in header
        $in_header = 0;

        while (<FOLDER>)  {

            $line = $_;
```

```
# line is first line of a msg
if (/^From .*/) {

    # check if last msg was a new one
    # if so, we need to update .idata and stuff
    if ($msg_moved)  {

        $num_refiles++;

        # put msg into a temp file
        open(TMPFILE, "> $tmpfile");
        print TMPFILE $tmp_msg;
        close(TMPFILE);

        # passes message to sortmail program

        # create the sortmail command line
        if ($tmp_file)
        {
            $command = "$sortmail_binary $sortmail_args -g ";
        }
        else
        {
            $command = "$sortmail_binary $sortmail_args ";
        }
        if (! -f $maildir.$old_folder."/.skip_me") {
            $command .= "--delete=$old_folder ";
        }
        if (! -f $maildir.$folder."/.skip_me") {
            $command .= "--insert=$new_folder ";
        }
        $command .= "$tmpfile";

        print $command."\n";

        open(FP, "$command |")
            || &quick_finish("Could not execute \"$arg\": $!\n");
        while (<FP>) {print;}
        close(FP);

        # done with sortmail updating
    }
    $in_header = 1;
    $new_folder = $folder;
    $found_id = 0;

    # flag if this msg has been moved by user
    $msg_moved = 0;
    $tmp_msg = $line;
}

# else if line is separator btwn header and body in a msg
elsif (($line =~ m/^\n$/) && ($in_header)) {
    $in_header = 0;
    $tmp_msg .= $line;
}
# else if line is the sortmail msg id line is the header
# check if this msg id is supposed to be here
elsif (($line =~ m/^X-sortmail:\s*(\d*)\n$/) && ($in_header)) {
    $id = $1;
    $found_id = 1;
    $used_to_be_here = 0;
```

```
                foreach $number (@list)  {
                    if ($number == $id) {
                        $used_to_be_here = 1;
                        break;
                    }
                }
                # yup, so just move on...
                if ($used_to_be_here) {
                    # print "msg $id is correctly in folder $folder\n";
                    $tmp_msg .= $line;
                }

                # user moved this guy on us!  we sorted incorrectly!
                # shame on us.  we better fix our sorter and update our
                # database accordingly
                else {

                    # find what folder it used to be in by looking it
                    # up
                    foreach $entry (@m2f) {
                        if ($entry =~ m/^$id (\S*)\n$/)  {
                            $old_folder = $1;
                            print "-- msg $id used to be in $old_folder\n";
                        }
                    }
                    print "-- msg $id is new in folder $folder\n";
                    $msg_moved = 1;
                }
                $tmp_msg .= $line;
            }

            # else line is just any old line in the msg
            else {
                $tmp_msg .= $line;
            }
        }
    }
}

# update the accuracies file ~/.sortmail_accuracy
$acc_file = $home_dir.".sortmail_accuracy";

$filters = 0;
$refiles = $num_refiles;

if (open(ACC, $acc_file))
{
    $line = <ACC>;
    $line =~ m/filters\s*=\s*(\d*)\s*refiles\s*=\s*(\d*)/;

    $filters += $1;
    $refiles += $2;
    if ($filters > 0) {
        $accuracy = int(($filters - $refiles)/$filters * 10000)/100;
    } else {
        $accuracy = 0.0;
    }
    if ($accuracy < 0.0) {
        $accuracy = 0.0;
    }
    close(ACC);
}
```

```
if (open(ACC, "> $acc_file"))
{
    print ACC "filters = $filters  refiles = $refiles\n";
    if ($accuracy) { print ACC "Accuracy = $accuracy \%\n"; }
    close(ACC);
} else {
    print STDERR "Was not able to write accuracies file\n";
}

# remake lists
system($list_creation_binary);

# end refile.pl
```

## sortmail.c

```
/* sortmail - general email sorter
   by Yu-Han Chang <ychang@ai.mit.edu>, December 1999

   parts of sortmail are adapted from ifile.
   ifile is Copyright (C) 1997  Jason Rennie <jrennie@ai.mit.edu>

   This program is free software; you can redistribute it and/or
   modify it under the terms of the GNU General Public License
   as published by the Free Software Foundation; either version 2
   of the License, or (at your option) any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program (see file 'COPYING'); if not, write to the Free
   Software Foundation, Inc., 59 Temple Place - Suite 330,
   Boston, MA  02111-1307, USA.
   */

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#include <time.h>
#include <ifile.h>

#define SEMKEY  10439838

int semid;
struct sembuf sops;

arguments args;
extern struct argp argp;
int msgs_read;        /* number of messages actually read in */

/* variables for keeping track of time/speed of ifile */
clock_t DMZ_start, DMZ_end, DMZ2_start;

/* ifilter specific function prototypes */
int cmp(const void *e1, const void *e2);

/* Main program */
int
main (int argc, char **argv)
{
  char *data_file = NULL;   /* full path of idata file */
  char *home_dir = NULL;    /* full path of user's home directory */
  FILE *MSG = NULL;         /* file pointer for a message */
  category_rating * ratings;
  ifile_db idata;
  htable ** message = NULL;
  int i;
  int db_read_result = 0, db_write_result = 0;
  char *file_name;
  int trimmed_words;
```

```
/* Harry's semaphore stuff to protect two ifile jobs from stepping
 * on each other */
/* Find the Semaphore id */
if ((semid = semget(SEMKEY, 1, 0666)) < 0)
  if ((semid = semget(SEMKEY, 1, 0666|IPC_CREAT|IPC_EXCL)) < 0)
    {
           perror("semget");
           exit (-1);
    }

/* Wait for Semaphore to clear */
sops.sem_num = 0;
sops.sem_op = 0;
sops.sem_flg = 0;

if (semop(semid, &sops, 1))
  {
    perror("semop");
    exit (-1);
  }

/* Set the Semaphore to clear on exit */
sops.sem_num = 0;
sops.sem_op = 1;
sops.sem_flg = SEM_UNDO;

if (semop(semid, &sops, 1))
  {
    perror("semop");
    exit (-1);
  }

ifile_init_args(&args);
argp_parse (&argp, argc, argv, 0, 0, &args);

ifile_verbosify(ifile_verbose, "%d file(s) passed\n", args.num_files);
for (i=0; i < args.num_files; i++)
  ifile_verbosify(ifile_verbose, "file #%d: %s\n", i,
                      EXT_ARRAY_GET(args.file, char *, i));

/* Get home directory */
home_dir = getenv("HOME");
if (home_dir == NULL)
  ifile_error("Fatal: HOME environment variable not defined!\n");
ifile_verbosify(ifile_verbose, "home directory = %s\n", home_dir);

/* Get the database file name */
if (args.db_file != NULL)
  data_file = ifile_strdup (args.db_file);
else
  data_file = ifile_sprintf("%s/%s", home_dir, DEFAULT_DB_FILE);

/* remove the .idata file if requested */
if (args.reset_data)
  {
    ifile_verbosify(ifile_progress, "Removing %s...\n", data_file);
    system(ifile_sprintf("rm %s", data_file));
  }

ifile_db_init(&idata);
ifile_open_log(argc, argv);
ifile_default_lexer_init();
```

```
/* argument variables that still need to be handled:
 * skip_header, minus_folder, plus_folder */

/* read and lex the message(s) */
if (args.read_message == TRUE)
  {
    msgs_read = 0;
    DMZ_start = clock();
    if (args.num_files > 0)
           {
             ifile_verbosify(ifile_progress, "Reading messages...\n");
             message = (htable **) malloc(args.num_files*sizeof(htable *));
             for (i=0; i < args.num_files; i++)
               {
                 file_name = EXT_ARRAY_GET(args.file, char *, i);
                 MSG = fopen(file_name, "r");
                 if (MSG == NULL)
                  {
                    ifile_verbosify(ifile_quiet,
                                 "Not able to open %s!  No action taken.\n",
                                 file_name);
                    message[i] = NULL;
                  }
                 else
                  {
                    message[i] = ifile_read_message(MSG);
                    if (args.occur == TRUE)
                      ifile_bitify_document(message[i]);
                    if (message[i] != NULL)
                      msgs_read++;
                  }
                 if (args.verbosity >= ifile_debug || args.print_tokens)
                  ifile_print_message(message[i]);
                 fclose(MSG);
               }
           }
    else
           {
             message = (htable **) malloc(sizeof(htable *));
             ifile_verbosify(ifile_quiet, "Reading message from standard
           input...\n");
             message[0] = ifile_read_message(stdin);
             msgs_read++;
             if (args.verbosity >= ifile_debug || args.print_tokens)
               ifile_print_message(message[0]);
             args.num_files = 1;
           }
    DMZ_end = clock();
    ifile_verbosify(ifile_progress,
                        "Read %d messages.  Time used: %.3f sec\n", msgs_read,
                        ((float)(DMZ_end-DMZ_start))/CLOCKS_PER_SECOND);
  }

/* Don't do anything else if we are printing tokens */
if (args.print_tokens)
  exit (0);

/* Now read the idata database */
if (args.read_db == TRUE)
  db_read_result = ifile_read_db(data_file, &idata);

/* Do LOOCV queries if requested */
```

```
  if (args.loocv_folder != NULL)
    for (i=0; i < args.num_files; i++)
      {
             ifile_del_db(args.loocv_folder, message[i], &idata);
             ratings = ifile_rate_categories(message[i], &idata);
             qsort(ratings, idata.num_folders, sizeof(category_rating), cmp);
             ifile_print_ratings(stdout, ratings, &idata);
             ifile_free(ratings);
             ifile_add_db(args.loocv_folder, message[i], &idata);
      }

  /* if a query was requested, make the calculations and output the results */
  if (args.query == TRUE)
    {
      for (i=0; i < args.num_files; i++)
             if (message[i] != NULL)
               {
                 ratings = ifile_rate_categories(message[i], &idata);
                 qsort(ratings, idata.num_folders, sizeof(category_rating), cmp);
                 ifile_print_ratings(stdout, ratings, &idata);
                 if (args.query_insert)
                   ifile_add_db(ratings[0].category, message[i], &idata);
                 ifile_free(ratings);
               }
    }

  if (args.write_db == TRUE)
    {
      if (args.plus_folder != NULL)
             for (i=0; i < args.num_files; i++)
               if (message[i] != NULL)
                 ifile_add_db(args.plus_folder, message[i], &idata);

      if (args.minus_folder != NULL)
             for (i=0; i < args.num_files; i++)
               if (message[i] != NULL)
                 ifile_del_db(args.minus_folder, message[i], &idata);

      if ((args.plus_folder != NULL || args.query_insert == TRUE) &&
                args.minus_folder == NULL)
             {
               trimmed_words = ifile_age_words(&idata, msgs_read);
               ifile_verbosify(ifile_progress,
                            "Trimmed %d words due to lack of frequency\n",
                            trimmed_words);
             }

      db_write_result = ifile_write_db(data_file, &idata);
      if (db_read_result != 0 && db_write_result == 0)
             {
               ifile_verbosify(ifile_quiet, "Created new %s file.\n", data_file);
               /* set proper permissions */
               system(ifile_sprintf("chmod 0600 %s\n", data_file));
             }
    }

  ifile_close_log();

  return 0;
}
```

## database.c

```
/* database routines */
/* adapted from ifile (C) 1997  Jason Rennie <jrennie@ai.mit.edu>

   This program is free software; you can redistribute it and/or
   modify it under the terms of the GNU General Public License
   as published by the Free Software Foundation; either version 2
   of the License, or (at your option) any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program (see file 'COPYING'); if not, write to the Free
   Software Foundation, Inc., 59 Temple Place - Suite 330,
   Boston, MA  02111-1307, USA.
   */

#include <time.h>
#include <math.h>
#include <pwd.h>
#include <unistd.h>
#include <sys/types.h>
#include <ifile.h>

#define LOG_2 0.69314718

extern arguments args;       /* info about command line arguments */
extern int msgs_read;        /* number of messages actually read in */

/* variables for keeping track of time/speed of ifile -*/
extern clock_t DMZ_start, DMZ_end, DMZ2_start;

/* given the age of a word, returns an integer which is the minimum
 * required frequency for the word to remain in the database */
long int
_trim_freq (long int age)
{
  if (age <= 0)
    return 0;
  else
    return (((long int)(log((float) age) / log(2.0))) - 1);
}

long int
_trim_freq0 (long int age)
{
  return 0;
}

/* Initializes the sortmail database */
void
ifile_db_init(ifile_db * idata)
{
  long int i;

  idata->num_folders = 0;
  idata->num_words = 0;
```

```
  idata->total_docs = 0;
  idata->total_freq = 0;

  /* words of AGE should be trashed if they have frequency less than returned
   * by this function */
  if (args.keep_infrequent)
    idata->trim_freq = _trim_freq0;
  else
    idata->trim_freq = _trim_freq;

  EXT_ARRAY_INIT(idata->folder_name, char *, IFILE_INIT_FOLDERS);
  EXT_ARRAY_INIT(idata->folder_freq, long int, IFILE_INIT_FOLDERS);
  EXT_ARRAY_INIT(idata->folder_msg, long int, IFILE_INIT_FOLDERS);
  for (i=0; i < IFILE_INIT_FOLDERS; i++)
    {
      /* EXT_ARRAY_SET(idata->folder_freq, int, i, 0);
         EXT_ARRAY_SET(idata->folder_msg, int, i, 0);
         printf("folder %d defaults:  freq = %d  msgs = %d\n", i,
                     EXT_ARRAY_GET(idata->folder_freq, int, i),
                     EXT_ARRAY_GET(idata->folder_msg, int, i));
                   */
    }
  htable_init(&(idata->data), IFILE_INIT_WORDS, (unsigned long (*)(const void *,
              long int)) hash);
}


/* Initializes a word entry of an ifile_db type.  Does NOT allocate memory. */
/* Written by Jason Rennie <jrennie@ai.mit.edu> */
void
ifile_db_entry_init (db_word_entry * wentry)
{
  wentry->word = NULL;
  wentry->age = 0;
  wentry->tot_freq = 0;
  wentry->freq = NULL;
}


/* Expects a valid file pointer which points to the beginning of an
 * ifile database (idata) file along with an ifile database structure.
 * Function reads in the header lines of file and stores information in IDATA.
 * Expects that IDATA is allocated and initialized.  DATA will be advanced
 * to the beginning of the end of the idata header.
 * Returns number of folders upon success, -1 upon failure. */
long int
ifile_read_header (FILE * DATA, ifile_db * idata)
{
  char line[MAX_STR_LEN*64];
  char *token = NULL;
  long int i;
  long int num;

  readline(line, sizeof(line), DATA);
  if (line[0] == '\0' || feof(DATA)) return -1;

  /* read folder names */
  token = strtok(line, " \t");
  for (i = 0; token != NULL; i++)
    {
      EXT_ARRAY_SET(idata->folder_name, char *, i, ifile_strdup(token));
      token = strtok(NULL, " \t");
    }
```

```
  idata->num_folders = i;

  readline(line, sizeof(line), DATA);
  if (line[0] == '\0' || feof(DATA)) return -1;

  /* read word frequencies */
  token = strtok(line, " \t");
  for (i = 0; token != NULL; i++)
    {
      num = atoi(token);
      EXT_ARRAY_SET(idata->folder_freq, long int, i, num);
      idata->total_freq += num;
      token = strtok(NULL, " \t");
    }

  if (i != idata->num_folders)
    ifile_verbosify(ifile_quiet, "Bad data file format - line #2\n");

  readline(line, sizeof(line), DATA);
  if (line[0] == '\0' || feof(DATA)) return -1;

  /* read document frequencies */
  token = strtok(line, " \t");
  for (i = 0; token != NULL; i++)
    {
      num = atoi(token);
      EXT_ARRAY_SET(idata->folder_msg, long int, i, num);
      idata->total_docs += num;
      token = strtok(NULL, " \t");
    }

  if (i != idata->num_folders)
    ifile_verbosify(ifile_quiet, "Bad data file format - line #3\n");

  return idata->num_folders;
}


/* Expects a valid file pointer which points to the beginning of the word
 * entry section of an ifile database (idata) file along with an ifile
 * database structure.  Function reads to the end of the file and stores
 * information in IDATA.  Expects that IDATA is allocated and initialized.
 * DATA will be advanced to the end of the file.
 * Returns number of word entries upon success, -1 upon failure.*/
long int
ifile_read_word_frequencies (FILE * DATA, ifile_db * idata)
{
  char line[MAX_STR_LEN*4];
  long int i = 1;

  readline(line, sizeof(line), DATA);

  while (!feof(DATA))
    {
      if (line == NULL || line[0] == '\0')
             ifile_verbosify(ifile_quiet, "Line # %d not in proper word entry
             format\n", (i+3));
      else
             idata->num_words += ifile_read_word_entry(line, idata);

      i++;
      readline(line, sizeof(line), DATA);
    }
```

```
  return idata->num_words;
}


/* Given a character array which contains a single word entry from an idata
 * file and a pointer to an ifile database, allocates and adds a
 * DB_WORD_ENTRY to IDATA->DATA.  Returns 1 if line contained a word entry,
 * 0 otherwise. */
/* written by Jason Rennie <jrennie@ai.mit.edu> */
long int
ifile_read_word_entry (char * line, ifile_db * idata)
{
  db_word_entry * wentry = malloc(sizeof(db_word_entry));
  char *token = NULL;
  long int folder, freq;
  long int num_folders = idata->num_folders;
  long int i;
  extendable_array * freq_array;

  /* initialize EXT_ARRAY things on the fly - do NOT do initialization
   * when creating the DB */
  wentry->freq = (extendable_array *) malloc(sizeof(extendable_array));
  freq_array = wentry->freq;
  EXT_ARRAY_INIT((*freq_array), long int, IFILE_INIT_FOLDERS);

  for (i=0; i < num_folders; i++)
    EXT_ARRAY_SET((*freq_array), long int, i, 0);

  token = strtok(line, " \t");

  /* if this is a blank line, don't add anything to IDATA */
  if (token == NULL)
    {
      free(wentry);
      free(freq_array);
      return 0;
    }

  /* add word entry to database of word entries */
  htable_put(&(idata->data), (void *) ifile_strdup(token), (void *) wentry);
  wentry->word = ifile_strdup(token);

  /* read the age of the word */
  token = strtok(NULL, " \t");
  if (token != NULL)
    wentry->age = atoi(token);
  else
    wentry->age = 0;

  /* read the per folder frequency */
  token = strtok(NULL, ":");
  while (token != NULL)
    {
      folder = atoi(token);
      token = strtok(NULL, " \t");
      if (token != NULL)
             {
               freq = atoi(token);
               wentry->tot_freq += freq;
               EXT_ARRAY_SET((*freq_array), long int, folder, freq);
             }
      token = strtok(NULL, ":");
```

```
    }
  return 1;
}


/* Given the name of an idata file this function parses the entire data file
 * and stored the read information into IDATA.
 * Returns 0 upon success, -1 upon failure. */
/* written by Jason Rennie <jrennie@ai.mit.edu> */
long int
ifile_read_db (char * data_file, ifile_db * idata)
{
  FILE * DATA;
  long int folders;
  long int words;

  ifile_verbosify(ifile_progress, "Reading %s from disk...\n", data_file);

  DATA = fopen(data_file, "r");

  if (DATA == NULL)
    {
      ifile_verbosify(ifile_quiet, "Not able to open %s for reading!\n",
                           data_file);
      return -1;
    }

  DMZ_start = clock();

  folders = ifile_read_header (DATA, idata);

  words = ifile_read_word_frequencies (DATA, idata);

  fclose(DATA);

  DMZ_end = clock();
  ifile_verbosify(ifile_progress,
                      "Read %d categories, %d words.  Time used: %.3f sec\n",
                      folders, words,
                      ((float)(DMZ_end-DMZ_start))/CLOCKS_PER_SECOND);

  return 0;
}


/* Given a message to rate and a database to get information from, calculates
 * a value for each folder which approximates the liklihood that the user
 * would have placed the message in that folder.  Returns an array of
 * category ratings */
category_rating *
ifile_rate_categories (htable * message, ifile_db * idata)
{
  long int i;
  float docval, nval, r;
  char *token = NULL;
  category_rating * ratings;
  hash_elem * elem;
  db_word_entry * wentry;
  long int print_calc = -1;  /* folder of which rating calculations are printed */
  float freq;

  if (args.folder_calcs != NULL)
    {
```

```
      print_calc = 0;
      for (i=0; i < idata->num_folders; i++)
             {
               if (strcmp(args.folder_calcs,
                        EXT_ARRAY_GET(idata->folder_name, char *, i)) == 0)
                 print_calc = i;
             }
    }

  ratings = malloc(sizeof(category_rating)*idata->num_folders);

  DMZ_start = clock();
  ifile_verbosify(ifile_progress, "Computing category ratings...\n");

  docval = (float)(idata->total_docs + idata->num_folders);

  if (args.folder_calcs)
    ifile_verbosify(ifile_debug, "Outputting calculations for folder \"%s\"\n",
                        (EXT_ARRAY_GET(idata->folder_name, char *, print_calc)));

  for (i=0; i < idata->num_folders; i++)
    {
      token = EXT_ARRAY_GET(idata->folder_name, char *, i);
      ratings[i].category = ifile_strdup(token);
      ratings[i].rating = 0.0;
    }

  for (elem = htable_init_traversal(message);
       elem != NULL; elem = htable_next_traversal(message, elem))
    {
      wentry = htable_lookup(&(idata->data), (char *) elem->index);
      if (wentry != NULL)
             {
               for (i=0; i < idata->num_folders; i++)
                 {
                   nval = (float) (EXT_ARRAY_GET(idata->folder_freq, long int, i)
             +
                                   idata->num_words);
                   freq = (float) EXT_ARRAY_GET((*(wentry->freq)), long int, i);
                   r = log((freq + 1.0) / nval);
                   if (i == print_calc)
                    ifile_verbosify(ifile_quiet, "word = %s  msg = %d  db = %d
             +rating = %.5f\n", (char *) elem->index, (long int) elem->entry,
             wentry->freq[i], r);
                   ratings[i].rating += ((long int) elem->entry) * r;
                 }
             }
      else
             if (print_calc >= 0 && print_calc < idata->num_folders)
               ifile_verbosify(ifile_quiet, "word = %s  msg = %d  db = 0\n",
                            (char *) elem->index, (long int) elem->entry);

    }
  DMZ_end=clock();
  ifile_verbosify(ifile_progress,
                     "Calculated category scores.  Time used: %.3f sec\n",
                     ((double)(DMZ_end-DMZ_start))/CLOCKS_PER_SECOND);
  return ratings;
}


/* Given the name of an idata file this function writes the information
 * stored in IDATA to disk.  The pre-existence of an .idata file in the
```

```
 * location is not checked for.
 * Returns 0 upon success, -1 upon failure. */
/* Written by Jason Rennie <jrennie@ai.mit.edu> for ifile */
long int
ifile_write_db (char * data_file, ifile_db * idata)
{
  FILE * DATA;
  long int folders;
  long int words;
  char *temp_data_file;
  char *user;
  char host[128];

  struct passwd *pwd = getpwuid (getuid ());

  if (!pwd)
    user = "unknown";
  else
    user = pwd->pw_name;

  if (gethostname (host, sizeof (host)))
    strcpy(host, "unknown");

  temp_data_file = ifile_sprintf ("%s.%s.%s", data_file, user, host);

  ifile_verbosify (ifile_progress, "Writing %s to disk...\n", data_file);

  DMZ_start = clock ();

  /* Open temporary data file for writing of database */
  DATA = fopen (temp_data_file, "w");
  if (DATA == NULL)
    {
      ifile_verbosify (ifile_quiet, "Not able to open %s for writing!\n",
                       data_file);
      return -1;
    }
  folders = ifile_write_header (DATA, idata);
  words = ifile_write_word_frequencies (DATA, idata);

  fclose (DATA);
  /* Rename file to regular database name */
  rename (temp_data_file, data_file);

  DMZ_end = clock ();
  ifile_verbosify (ifile_progress,
                       "Wrote %d folders, %d words.  Time used: %.3f sec\n",
                       folders, words,
                       ((float)(DMZ_end-DMZ_start))/CLOCKS_PER_SECOND);

  return 0;
}


/* Expects a valid file pointer which points to the beginning of an
 * ifile database (idata) file along with an ifile database structure.
 * Function reads in the header lines of file and stores information in IDATA.
 * Expects that IDATA is allocated and initialized.  DATA will be advanced
 * to the beginning of the end of the idata header.
 * Returns number of folders upon success, -1 upon failure. */
/* Written by Jason Rennie <jrennie@ai.mit.edu> */
long int
ifile_write_header (FILE * DATA, ifile_db * idata)
```

```
{
  long int i;
  long int num;

  for (i=0; i < idata->num_folders; i++)
    fprintf(DATA, "%s ", EXT_ARRAY_GET(idata->folder_name, char *, i));
  fputc('\n', DATA);

  for (i=0; i < idata->num_folders; i++)
    {
      num = EXT_ARRAY_GET(idata->folder_freq, long int, i);
      fprintf(DATA, "%ld ", num);
    }
  putc('\n', DATA);

  for (i=0; i < idata->num_folders; i++)
    {
      num = EXT_ARRAY_GET(idata->folder_msg, long int, i);
      fprintf(DATA, "%ld ", num);
    }
  putc('\n', DATA);

  return idata->num_folders;
}


/* Expects a valid file pointer which points to the beginning of the word
 * entry section of an ifile database (idata) file along with an ifile
 * database structure.  Function reads to the end of the file and stores
 * information in IDATA.  Expects that IDATA is allocated and initialized.
 * DATA will be advanced to the end of the file.
 * Returns number of word entries upon success, -1 upon failure.*/
/* Written by Jason Rennie <jrennie@ai.mit.edu> */
long int
ifile_write_word_frequencies (FILE * DATA, ifile_db * idata)
{
  long int i;
  hash_elem * elem;
  db_word_entry * wentry;
  long int freq;
  extendable_array * freq_array;
  long int num_words = 0;

  for (elem = htable_init_traversal(&(idata->data));
       elem != NULL; elem = htable_next_traversal(&(idata->data), elem))
    {
      wentry = (db_word_entry *) elem->entry;
      freq_array = wentry->freq;
      if (wentry->tot_freq == 0) continue;
      num_words++;
      fprintf(DATA, "%s %ld ", wentry->word, wentry->age);

      for (i=0; i < idata->num_folders; i++)
             {
               freq = EXT_ARRAY_GET((*freq_array), long int, i);
               if (freq > 0)
                 fprintf(DATA, "%ld:%ld ", i, freq);
             }
      putc('\n', DATA);
    }
  return num_words;
}
```

```
/* Adds EPOCHS to each word's age and eliminates words from the database which
 * are overly infrequent  Uses IDATA->TRIM_FREQ() to calculate which words
 * should be tossed.  Returns the number of trimmed words. */
long int
ifile_age_words (ifile_db * idata, long int epochs)
{
  long int i;
  hash_elem * elem;
  db_word_entry * wentry;
  long int wfreq, ffreq, new_freq;
  long int trimmed_words = 0;

  for (elem = htable_init_traversal(&(idata->data));
       elem != NULL; elem = htable_next_traversal(&(idata->data), elem))
    {
      wentry = (db_word_entry *) elem->entry;
      wentry->age += epochs;
      if (idata->trim_freq(wentry->age) > wentry->tot_freq)
             {
               /* update the word frequency values for each folder */
               for (i=0; i < idata->num_folders; i++)
                 {
                   wfreq = EXT_ARRAY_GET((*wentry->freq), long int, i);
                   ffreq = EXT_ARRAY_GET(idata->folder_freq, long int, i);
                   new_freq = (ffreq >= wfreq) ? (ffreq - wfreq) : 0;
                   EXT_ARRAY_SET(idata->folder_freq, long int, i, new_freq);
                   EXT_ARRAY_SET((*wentry->freq), long int, i, 0);
                 }
               wentry->tot_freq = 0;
               trimmed_words++;
             }
    }
  return trimmed_words;
}


/* if we wanted to make ifile more efficient, we would allocate our
 * idata->data->freq arrays so that they are one larger than the number
 * of folders.  This would make it so that we would never have to reallocate
 * these arrays since it is currently not possible to add messages to more
 * than one folder (per execution) */

/* Adds the word statistics from MESSAGE to FOLDER in IDATA */
void
ifile_add_db (char * folder, htable * message, ifile_db * idata)
{
  hash_elem * elem;
  db_word_entry * wentry;
  long int folder_index;
  long int freq = 0;
  long int folder_freq = 0;
  long int i;
  long int num_msgs;

  folder_index = -1;
  for (i=0; i < idata->num_folders; i++)
    {
      if (strcmp(folder,
                 EXT_ARRAY_GET(idata->folder_name, char *, i)) == 0)
             folder_index = i;
    }
```

```
  if (folder_index == -1)
    {
      EXT_ARRAY_SET(idata->folder_name, char *, idata->num_folders, folder);
      folder_index = idata->num_folders;
      idata->num_folders++;
    }

  for (elem = htable_init_traversal (message);
       elem != NULL; elem = htable_next_traversal (message, elem))
    {
      ifile_verbosify(ifile_debug, "adding... %s %d\n", (char *) elem->index,
                      (long int) elem->entry);
      wentry = htable_lookup (&(idata->data), (char *) elem->index);
      if (wentry == NULL)
             {
               wentry = (db_word_entry *) malloc(sizeof(db_word_entry));
               ifile_db_entry_init(wentry);
               wentry->freq = (extendable_array *)
             malloc(sizeof(extendable_array));
               wentry->word = (char *) elem->index;
               wentry->tot_freq = 0;
               wentry->age = 0;
               EXT_ARRAY_INIT((*wentry->freq), long int, IFILE_INIT_FOLDERS);
               freq = 0;
               htable_put (&(idata->data), (char *) elem->index, wentry);
               idata->num_words++;
             }
      else
             freq = EXT_ARRAY_GET((*wentry->freq), long int, folder_index);
      folder_freq += (long int) elem->entry;
      freq += (long int) elem->entry;
      idata->total_freq += (long int) elem->entry;
      wentry->tot_freq += (long int) elem->entry;
      EXT_ARRAY_SET((*wentry->freq), long int, folder_index, freq);
    }
  /* increase message count by one */
  num_msgs = EXT_ARRAY_GET(idata->folder_msg, long int, folder_index);
  EXT_ARRAY_SET(idata->folder_msg, long int, folder_index, (num_msgs+1));
  /* adjust the folder word frequency count */
  folder_freq += EXT_ARRAY_GET(idata->folder_freq, long int, folder_index);
  EXT_ARRAY_SET(idata->folder_freq, long int, folder_index, folder_freq);
}


/* Removes the word statistics of MESSAGE from FOLDER in IDATA.
 * if FOLDER does not exist, or any word of MESSAGE has a lower
 * frequency than in the database, an error message is printed. */
/* Written by Jason Rennie <jrennie@ai.mit.edu> */
void
ifile_del_db (char * folder, htable * message, ifile_db * idata)
{
  hash_elem * elem;
  db_word_entry * wentry;
  long int folder_index;
  long int freq;
  long int i;
  long int folder_freq = 0;
  long int num_msgs;

  folder_index = -1;
  for (i=0; i < idata->num_folders; i++)
    {
      if (strcmp(folder,
```

```
                    EXT_ARRAY_GET(idata->folder_name, char *, i)) == 0)
              folder_index = i;
    }

  if (folder_index == -1)
    {
      ifile_verbosify(ifile_quiet, "Folder \"%s\" does not appear to exist\n",
                          folder);
      return;
    }

  for (elem = htable_init_traversal (message);
       elem != NULL; elem = htable_next_traversal (message, elem))
    {
      wentry = htable_lookup (&(idata->data), (char *) elem->index);
      if (wentry == NULL)
              {
                ifile_verbosify(ifile_verbose, "Word \"%s\" does not exist in the
              database.  Skipping.\n", (char *) elem->index);
                continue;
              }
      else
              freq = EXT_ARRAY_GET((*wentry->freq), long int, folder_index);
      freq -= (long int) elem->entry;
      folder_freq -= (long int) elem->entry;
      if (freq <= 0)
              {
                ifile_verbosify(ifile_verbose, "Word \"%s\" has lower frequency in
              database than in message.\n  Setting database frequency to 0\n",
              (char *) elem->index);
                freq = 0;
              }
      EXT_ARRAY_SET((*wentry->freq), long int, folder_index, freq);
    }
  /* update the word frequency count for the folder */
  folder_freq += EXT_ARRAY_GET(idata->folder_freq, long int, folder_index);
  if (folder_freq < 0)
    folder_freq = 0;
  EXT_ARRAY_SET(idata->folder_freq, long int, folder_index, folder_freq);
  /* update the message count for the folder */
  num_msgs = EXT_ARRAY_GET(idata->folder_msg, long int, folder_index);
  num_msgs = (num_msgs >= 1) ? (num_msgs-1) : 0;
  EXT_ARRAY_SET(idata->folder_msg, long int, folder_index, num_msgs);
}
```

## build.pl

```
#!/usr/local/bin/perl

# by Yu-Han Chang <ychang@ai.mit.edu> for sortmail

print "hello, welcome to sortmail\n";
print "we're now going to build a database using your existing
mail...\n\n";

# Full path of ifile binary (if not in $PATH)
$sortmail_binary = "/home/y/c/ychang/sortmail/sortmail";
$sortmail_args = "-h -v 2";
$scratchdir = "/scratch/ychang/mail/";
$maildir = "/home/y/c/ychang/mail/";

# gets environment information #
$home_dir = $ENV{"HOME"}."/";
$mail_path = "/home/y/c/ychang/mail/";

# reset old word statistics
print "$ifile_binary -r\n";
system "$ifile_binary -r";

$true = opendir(MAILDIR, "$maildir");
@folders = readdir(MAILDIR);

# get rid of "." and ".." files
shift(@folders);
shift(@folders);
# print "@folders\n";

# read each pine folder file and expand it into individual msg files
# in the appropriate folder scratch directory
foreach $folder (@folders) {
    if (($folder =~ m/^\..*/) || (-f "$maildir".".$folder".".skip_me")) {
        print "skip $folder\n";
    }
    else {
        $count=0;
        open(FOLDER, "$maildir$folder");
        print STDOUT "rm -f $scratchdir$folder\/\*\n";
        system("rm -f $scratchdir$folder\/\*");
        system("rmdir $scratchdir$folder");
        system("mkdir $scratchdir$folder");
        while (<FOLDER>)  {
            if (/^From .*/) {
                if ($count > 0) {
                    close(SCRATCH_FILE);
                }
                $count++;
                open(SCRATCH_FILE, "> $scratchdir$folder\/$count");
                print SCRATCH_FILE $_;
            }
            else {
                print SCRATCH_FILE $_;
            }
        }
        close(FOLDER);
    }
}
```

```
opendir(SCRATCHDIR, "$scratchdir");
@folders = readdir(SCRATCHDIR);
foreach $folder (@folders) {
    print "processing folder $folder...\n";
    local(@files);

    $dir = $scratchdir.$folder;
    $tmp_file = ".ifile".(time() % 100000);

    # Only accumulate data for folders which we have write permissions
    # to and which don't have .skip_me files
    if (! -f "$dir/.skip_me"
        && open(FOO, "> ".$dir."/".$tmp_file))
    {
        close(FOO);
        unlink($dir."/".$tmp_file);

        $num_files = 0;
        opendir(FILES, $dir);
        while ($_ = readdir(FILES))
        {
            if (m/^\d+$/)
            {
                $files[$#files+1] = $_;
            }
        }
        closedir(FILES);

        if (@files > 0)
        {
            chdir $scratchdir.$folder;
            $files_arr_size = @files;
            print STDERR "$ifile_binary $ifile_args -i $folder $scratchdir$folder/*
              [$files_arr_size]\n";
            system "$ifile_binary $ifile_args -i $folder ".join(' ', @files);
        }
    }
    else
    {
        print STDERR "Skipping $folder...\n";
    }
}
closedir(SCRATCHDIR);

# end build.pl
```

## make_lists.pl

```
#!/usr/local/bin/perl

$maildir = "/home/y/c/ychang/mail/";
$m2flist = "/home/y/c/ychang/mail/.msg.to.folder.list";

$true = opendir(MAILDIR, "$maildir");
@folders = readdir(MAILDIR);

print "creating updated msg id lists...\n";

# get rid of "." and ".." files
shift(@folders);
shift(@folders);
# print "@folders\n";

open(M2F, "> $m2flist");

# read each pine folder file and extract msg id numbers and output
# them into a file .$folder.msglist
foreach $folder (@folders) {
    $in_header = 0;

    if ($folder =~ m/^\..*/) {}
    else {
        open(FOLDER, "$maildir$folder");
        open(LIST, "> $maildir\.$folder\.msglist");
        while (<FOLDER>)  {
            $line = $_;
            if (/^From .*/) {
                $in_header = 1;
            }
            if ($line =~ m/^\n$/ && ($in_header)) {
                $in_header = 0;
            }
            if (($line =~ m/^X-sortmail:\s*(\d*)\n$/) && ($in_header)) {
                $id = $1;
                print LIST "$id\n";
                print M2F "$id $folder\n";
            }
        }
    }
    close(LIST);
    close(FOLDER);
}
close(M2F);

print "successfully created.\n";

# end make_lists.pl
```

## add_ids.pl

```
#!/usr/local/bin/perl

# adds a unique id tag to each message so that sortmail can keep track of where
# they are going
# for simplicity, we're jsut going to keep a count starting
# from good old 1.

# new users of sortmail need only run this script once on their existing presorted
# mailboxes.  After this, all messages should retain their assigned ID numbers.

$scratchdir = "/scratch/ychang/mail/";
$maildir = "/home/y/c/ychang/mail/";
$countfile = "/home/y/c/ychang/.sortmail.id.count";
$tmpfile = "/scratch/ychang/mail/.sortmail.id.mail.tmp";

# read in current id count
open(COUNTFILE, "$countfile");
$count = <COUNTFILE>;
chop $count;
close(COUNTFILE);

# read folder names in mail directory
$true = opendir(MAILDIR, "$maildir");
@folders = readdir(MAILDIR);

# get rid of "." and ".." files
shift(@folders);
shift(@folders);
# print "@folders\n";

foreach $folder (@folders) {
    open(FOLDER, "$maildir$folder");
    open(TMP, "> $tmpfile");

    if (($folder =~ m/^\..*/) || (-f "$maildir".".$folder".".skip_me")) {
        print "skip $folder\n";
    }
    else {
        while (<FOLDER>)  {
            if (/^From .*/) {
                $print_filter = 0;
            }
            $line = $_;
            # Add the X-filter header at the end of the header section
            if ($line =~ m/^\n$/ && (!$print_filter))
            {
                print TMP "X-sortmail: $count\n";
                $print_filter = 1;
                $count++;
            }
            # Rename old x-filter header if we are still in header section
            $line =~ s/^x-filter: /Old-x-filter: /i if (!$print_filter);
            $line =~ s/^X-sortmail: /Old-x-sortmail: /i if (!$print_filter);
            # Make sure each line ends with feed-line
            $line .= "\n" if ($line !~ m/\n$/);

            print TMP $line;
        }
    }
    close(FOLDER);
```

```
    close(TMP);
    system("mv $tmpfile $maildir$folder");
}

open(COUNTFILE, "> $countfile");
print COUNTFILE "$count\n";
close(COUNTFILE);

# end add_ids.pl
```