MeatSlicer: Spam Classification with Naieve Bayes and Smart Heuristics

Michael Salib

December 12, 2002

Abstract

MeatSlicer is a framework for experimenting with Naive Bayes spam filters. It allows researchers to compare the effects of different tokenization, stemming, and feature selection algorithms. Most importantly, MeatSlicer makes it easy to integrate heuristic tests into a Bayesian classification framework. Several experiments were conducted with MeatSlicer against large electronic mail datasets with surprising results. These results call into question the efficacy of stemming, fine grained tokenization, and sophisticated information theory derived feature selectors. They also suggest that while frequency based classifiers may be more robust, Bayesian classifiers that integrate heuristics perform better.

Contents

1 Introduction **3**

2 Core Design Decisions **4**

3 Design **5**
3.1 Tokenization 5
3.2 Stemming 5
3.3 Zero-Knowledge Probability Estimates 5
3.4 Feature Selection 5
3.5 Heuristics 6
3.6 Systems Issues 8

4 Experiments **8**
4.1 Experiment 1 9
4.2 Experiment 2 9
4.3 Experiment 3 9
4.4 Miscellaneous Experiments 9

5 Results **9**
5.1 Experiment 1 9
5.2 Experiment 2 11
5.3 Experiment 3 13

6 Contributions **14**

A Source Code **16**
A.1 meatslicer.py 16
A.2 bayes.py 19
A.3 heuristics.py 27
A.4 runTests.py 35
A.5 buildGraphs.py 38

1 Introduction

Unsolicited Commercial Email (spam) presents a growing threat to the viability of electronic mail as a communications medium. Many companies and individuals waste large sums of money each year transferring, storing, processing and sorting through (often fraudulent) commercial messages in which they have no interest. Current approaches to dealing with spam include:

1. Reject incoming mail at the mail server with a mail server blacklist. Since this rejection occurs before messages are even transferred, such techniques obviously cannot make use of content based filtering. Typically, network administrators configure their mail servers to check each mail server requesting a connection against a Realtime Blackhole List (RBL). RBL checks are usually conducted via the Domain Name System (DNS) with stylized lookups, although some large providers may access RBLs through DNS zone transfers or through Border Gateway Protocol routing.

   Although initially promising, RBL-based approaches have proven ineffective at stopping spam. The process of maintaining blacklists is difficult and expensive; one blacklist provider recently shut down their service after receiving a lawsuit. Moreover, blacklist compilers are always one step behind spammers: they can only respond after a spammer has generated enough activity to get on the blacklist. Finally, RBLs fare poorly against spammers that move quickly from one "throw-away" dialup ISP account to another, a practice that has become the norm in the spam community.

2. Reject incoming mail after it has arrived by using a content blacklist. This approach is typified by the Distributed Checksum Clearinghouse (DCC). Cooperating mail servers send checksums of all incoming bulk mail to master DCC servers, which then maintain a count on how often mail with a particular checksum has been seen in the wild. Anyone can then take a checksum of an incoming message and ask a DCC server how many copies of the corresponding message it has been told exist in the wild. The checksum DCC uses is a "fuzzy hash" designed so that messages that differ by only a few characters will tend have the same hash. This is critical since spammers have taken to adding small random elements to their messages in an attempt to avoid detection.

   This approach suffers from two major drawbacks: the need for whitelisting and trust concerns. DCC doesn't distinguish between solicited and unsolicited bulk mail: if its bulk, it ends up in the DCC database. Consequently, DCC should only be used in conjunction with a whitelist describing legitimate bulk messages or senders. The use of the whitelist adds significantly to the management overhead of using DCC. In addition, the integrity of the DCC system relies heavily on the trust relationships between parties in the system. For example, if DCC servers accepted spam submissions from anonymous users, the system could be subverted by spammers. These issues yet to be resolved.

3. Reject mail after it has arrived by applying clever heuristics. This approach is typified by SpamAssassan and its ilk. SpamAssassan runs a large number of simple tests against each message. Each test returns a numeric score and those scores are combined in a weighted sum (the weights are determined by a genetic algorithm). The total score indicates the likelihood that the given message is spam. While most of the tests used are content based, SpamAssassan has been adding tests that make use of the RBL mail server blacklists and DCC-like content blacklists.

SpamAssassan offers significant advantages over the previous two approaches because it makes use of far more information than they do. Also, instead of offering the user a binary spam/not spam decision, it offers a score indicating the likelihood of spam, allowing users to tailor its aggressiveness based on how much valid mail they're willing to see misclassified as spam. On the other hand, SpamAssassan is difficult to install and consumes more computational resources than the other approaches.

None of the approaches described above are sufficient for eliminating spam. Paul Graham describes [1] a different approach based on Bayesian Decision Theory. His approach works as follows. Given a corpus of already labelled messages containing N_s spam messages and N_n non spam messages, we tokenize each message. For each token k, we measure f_s^k and f_n^k, the number of times k occurred in the spam corpus and non-spam corpus respectively. If we assume that the occurrence of any token in a message is independent of the occurrence of any other token, Bayes Rule says that given a message M, the probability that M is spam given that token k occurs in M is:

$$P(M\, is\ spam|k) = \frac{\frac{f_s^k}{N_s}}{\frac{f_s^k}{N_s} + \frac{f_n^k}{N_n}}$$

Graham points out that given a sequence of conditionally independent conditional probabilities $p_1, p_2, \ldots, p_n$, the combined probability is:

$$\frac{p_1 \cdot p_2 \cdot \ldots \cdot p_n}{p_1 \cdot p_2 \cdot \ldots \cdot p_n + (1 - p_1) \cdot (1 - p_2) \cdot \ldots \cdot (1 - p_n)}$$

Of course, the assumption that the occurrence of a single word is statistically independent of other words in the same message is patently insane. However, practical uses of such models suggest that little accuracy is lost by violating this independence assumption. Bayesian classifiers that assume statistical independence are termed Naive Bayesian classifiers.

Following Graham's lead, I built a Naive Bayesian spam classifier. While Naive Bayes performs well, it is far from perfect. My experience has convinced me that Naive Bayes cannot solve the spam problem alone. However, I was intrigued by how the SpamAssassan team integrates many tools to produce a better, more robust tool. I suspected that while any one test might give a false positive (DCC would indicate spam even for solicited bulk mail for example), by using all the tests to accumulate evidence for or against a spam decision, much better results could be achieved. After conducting my experiments with the Naive Bayes classifier, I built a set of heuristic spam tests and attempted to integrate them into the Naive Bayes classifier. The result is MeatSlicer.

2 Core Design Decisions

The most important design decision I made was in choosing Naive Bayes as a learning method. I initially considered using a rule based system, but I was concerned about the dearth of industrial strength rule based applications. Toys don't count. In addition, I suspected that the binary nature of most rule based systems would not fare well in the noisy domain of spam filtering where classifiers are often faced with contradictory information: simple rule based systems would be too brittle. Using rules along with a scoring system would fix that problem, but the resulting system would be difficult to understand and almost impossible to train.

The need for smoothness and flexibility suggested that I consider neural nets. However, it seems like building and sizing nets appropriately is a black art; the dangers of over fitting are legion. I considered k-nearest neighbors as well and rejected it because of difficulties with feature selection. In other words, I was concerned about figuring out how to scale different dimensions correctly.

Ultimately, I built a Naive Bayes system because I felt that Naive Bayes was more transparent and insensitive to noise than other learning algorithms for this application. The underlying mental model seemed a lot easier to grasp and reason about than lots of invisible hyper-plane operations in a ridiculously high dimensional space. When I did a literature search, I discovered that much to the amazement of researchers, Naive Bayes remains competitive with a broad range of new learning algorithms.

3 Design

MeatSlicer is a framework for experimenting with different spam detection techniques. I'll now describe some of the features it implements that are used in spam filtering.

3.1 Tokenization

MeatSlicer performs message tokenization using regular expressions. I experimented with several different regular expression tokenizers during development. Of the two most interesting, the first simply look for tokens consisting of a contiguous sequence of letters, digits, and punctuation marks. It would thus tokenize the string "make3 money now!!" as "make3, money, now!!". The second tokenizer is more sophisticated; it looks for contiguous sequences of letters and digits or contiguous sequences of characters that are not letters, digits, or whitespace. It would thus classify the previous example as "make, 3, money, now, !!".

3.2 Stemming

MeatSlicer offers the option of using a stemming library from the Snowball project. Stemming strips off the most common morphological and inflexional endings from english words, reducing them to their linguistic root. For example, a good stemming algorithm should reduce the phrase "eating roasted chickens" to "eat roast chicken".

3.3 Zero-Knowledge Probability Estimates

Text classifiers like MeatSlicer frequently encounter input words that have been seen in one corpus but not the other. This raises the question of how we should treat such words. Graham suggests [1] assigning conditional spam probabilities of 0.01 and 0.99 for these words depending on whether they were absent from the spam corpus or the non-spam corpus. I found that I could dramatically improve performance by using $P = \frac{1}{N_s+N_n}$ for words that do not appear in the spam corpus and $P = 1 - \frac{1}{N_s+N_n}$ for the non-spam corpus. In these equations, N_s is the number of tokens in the entire spam corpus while N_n is the number of tokens in the non-spam corpus.

3.4 Feature Selection

Natural languages are sufficiently large that any reasonably sized corpus will contain many thousands of distinct word types [2]. That means that the task of classifying a new message involves working in a space with thousands of dimensions, a formidable task. In addition, not all features are equally useful in classification: the fact that a message has the word "the" in it is far less informative than the fact that it has the word Nigeria. MeatSlicer thus offers three different techniques for reducing the number of features it classifies with. These include an ad-hoc method, Mutual Information, and Hypothesis Testing.

The ad-hoc method was described by Graham in [1]. For each word k that appears in the training corpus, we calculate $|P(spam|k) - \frac{1}{2}|$. We then select the N words with the highest value. Intuitively, this procedure should yield the set of words with the highest resolving power. The correct value of N for an application can only be determined experimentally.

Mutual Information (MI) is a metric based on information theory concepts [4, 5, 2]. Also known as Information Gain, for a given word k, MI measures the entropy difference between the likelihood of any message being spam and the likelihood of a message being spam given that the message includes the word k. In this case, for a given word k, MI is given by

$$MI = \frac{f_s^k}{N_s + N_n} \log \frac{\frac{f_s^k}{f_s^k + f_n^k}}{\frac{N_s}{N_s + N_n}} + \frac{N_s - f_s^k}{N_s + N_n} \log \frac{\frac{N_s - f_s^k}{N_s + N_n - f_s^k - f_n^k}}{\frac{N_s}{N_s + N_n}} + \frac{f_n^k}{N_s + N_n} \log \frac{\frac{f_n^k}{f_s^k + f_n^k}}{\frac{N_n}{N_s + N_n}} + \frac{N_n - f_n^k}{N_s + N_n} \log \frac{\frac{N_n - f_n^k}{N_s + N_n - f_s^k - f_n^k}}{\frac{N_n}{N_s + N_n}}$$

In practice, as pointed out in [4], picking the right value of N is a black art; the optimal value varies wildly for different text classification tasks.

Finally, Rennie [4] suggests using hypothesis testing to rank words by the extent that their occurrence in messages is independent of whether or not those messages are spam. This metric is given by:

$$HT = 2f_s^k \log \frac{\frac{f_s^k}{f_s^k + f_n^k}}{\frac{N_s}{N_s + N_n}} + 2f_n^k \log \frac{\frac{f_n^k}{f_s^k + f_n^k}}{\frac{N_n}{N_s + N_n}}$$

This metric has the advantage that one does not need to calculate a threshold value N in order to use it like the previous two methods. Instead, one can specify a significance level. Regrettably, I did not have time to implement significance level feature selection in MeatSlicer. MeatSlicer uses hypothesis testing, but with a threshold parameter like the other two methods of feature selection.

3.5 Heuristics

MeatSlicer makes use of several heuristics to provide evidence for whether or not a message is spam. These include:

Blackhole Test This test collects the set of IP addresses used to relay the message and checks each of them against a collection of RBL-style blackhole servers. Each time a relay IP is found on a blacklist, the score increases, thus penalizing messages that are sent over many confirmed open relays while being more lenient to messages that came over a single relay that is listed by only one blacklist.

DCC Test This test uses the Distributed Checksum Clearinghouse to determine how many times similar messages have been seen before at other mail servers.

Hopcount Test This test simply measures the number of relays over which the message traveled. Since many non-spam messages are local (traveling from and to the same school or company), they should traverse fewer relays than other messages.

Country Test This test simply counts the number of distinct countries the message traveled through to get here. It does so by reading the IP addresses of all relays the message traveled over and then looking up the resulting addresses in an IP address to country name mapping database derived from public internet records. This is useful since much spam either originates overseas or travels through many different countries.

Distance Test Like the Country Test, this test gives a very rough measure of the distance a message has traveled in order to get here. It calculates the country in which each relay is located and then uses a dataset compiled from the CIA World Fact Book to calculate the coordinates of the center of each of those countries. It then uses those coordinates to determine the distance (in miles) between each pair of hops and returns the sum of those distances.

SMTP Test This test uses the Simple Mail Transport Protocol [3] to guess at the authenticity of the message. More specifically, it tries to determine if a message has been forged. It determines the return address by examining the message headers. It then uses the DNS to determine what the appropriate mail server for that address is by looking for a Mail Exchange (MX) record associated with that address. For example, given a return address of msalib@mit.edu, the SMTP test would determine that the associated mail server is pacific-carrier-annex.mit.edu or fort-point-station.mit.edu.

Many spam messages include a bogus return path that cannot be resolved by DNS at all. If there actually is a mail server associated with that address, this test connects to it using SMTP and tries to verify that the server is actually willing to accept mail for that user. Mail sent by legitimate users should pass this test without difficulty.

Message ID Test This test checks the message ID header. Any well formed message should have a well formed message ID. There is a slight penalty for message IDs that include a machine name that cannot be resolved by the test.

Non-Alphanumeric Character Tests These two tests measure what percentage of the subject and body of the message consist of non-alphanumeric characters.

Time Travel Test This test calculates the absolute value of the difference between when the message was received at its destination and when it claims to have been sent. Non-spam messages tend to be received soon after they were sent. Some spam messages have falsified headers that indicate they were sent in some future time so as to confuse user mail programs into displaying them at the top of a list of date sorted messages. Many other spam messages simply take a very long time to reach their destination since they're being routed over a long distance through many relays.

Forward and Reverse DNS Test This test determines whether the final relay that passed this message on was spoofing a legitimate mail server. It does so by finding the last relay's IP address from the headers and then reverse mapping that address to a domain name. It then forward maps that domain name and checks to see if the resulting address was identical to the IP address found in the header. Since anyone who rents an address block can set the reverse DNS entries to be what they like for their own address, spammers will often configure their reverse DNS to give the domain name of a legitimate company. However, since only the domain holder can change the IP address that the domain name forward maps to, we can easily detect such domain name spoofing. If the last relay doesn't even have a reverse mapping, that is also suspicious, but less so than failing the spoofing test.

3.6 Systems Issues

A system this complicated brings with it several difficult systems engineering challenges. Since I was examining very large quantities of mail, memory consumption became problematic. I was able to solve that problem by relying on generator-based lazy mailbox access, so I never had to have the entire mail spool loaded into memory at once.

A second problem was underflow. When calculating probabilities with many features, MeatSlicer would often run into situations where it had to operate on products of many probabilities. Such products would occasionally underflow, thereby causing division by zero errors and no end of suffering. I resolved that problem by using infinite precision arithmetic libraries in the few places where underflow was occurring.

Another problem was introduced by the heuristic tests. Many of these tests, such as the blackhole test, the DCC test, or the SMTP test, involve sending data over the network. The amount of data is generally very small: only a few packets per message. However, the latency imposed by network access is significant, especially given the volume of messages I was examining. To make matters worse, many of these network services use UDP, so packet delivery is not guaranteed at all. When a single packet gets dropped in the network, there's no way to tell that it never arrived at its destination. Instead, we must wait for a timeout and then try again. This latency means that a single experiment could take many hours to complete. In order to mask this latency, I built a thread system that processes messages in batch thereby ensuring that several different threads are always waiting on network data.

I also built a persistent object store to cache network test results. These features dramatically improved performance. However, multithreading introduced other problems. Because the persistent object store was not designed to support concurrent writes, it scrambled cached results initially. I solved this problem by introducing a writing queue onto which threads wanting to write to the persistent object store could place their write requests. This writing queue was serviced by one thread dedicated to taking requests off of it and writing them to the object store, thereby serializing writes and ensuring that one and only one thread wrote to the object store at any one time.

4 Experiments

In addition to the data provided by the class, I also used another data set consisting of a subset of my personal filtered inbox. My dataset had all MIME messages removed in order to simplify testing. I also had the system translate non-ASCII message encodings into plain text. Statistics on both data sets are listed in the table below.

	6.034 Dataset	Personal Dataset
Spam Messages	60	664
Non-spam Messages	99	5599
Total Messages	159	6263
Size (kilobytes)	188	39,391

MeatSlicer works by determining the probability that each testing message is spam, comparing that probability to a fixed threshold and classifying the message accordingly. Initially, I measured filter performance by calculating what percentage of spam and non-spam messages were misclassified for a given threshold. However, different systems can provide the same performance at different thresholds. In order to get a better picture of each filter's performance, I decided to measure spam filtering performance by calculating precision/recall curves, a commonly used metric in Information Retrieval.

Precision is the percentage of messages that my classifier labels as spam that actually are spam. Recall is the percentage of messages that actually are spam that my classifier labels as spam. For spam filtering, precision corresponds to how good the filter is at avoiding false positives, which is usually the most important thing for users. Recall corresponds to how good the filter is at avoiding false negatives. Better filters will provide higher recall for a given precision.

4.1 Experiment 1

The first experiment involved running my classifier against both data sets while varying which tokenizer was used and whether or not stemming was performed.

4.2 Experiment 2

This experiment involved running my classifier against both data sets while varying which feature selection algorithm used. I used the tokenization and stemming parameters that produced the best results in Experiment 1 here.

4.3 Experiment 3

This experiment compared classifier accuracy for the naive Bayes classifier when using only word frequencies (as in the previous two experiments), only heuristics, and combinations of word frequencies and heuristics. I used the stemming and tokenization parameters that produced the best results in Experiment 1. Since many of the heuristic tests require access to message headers which are not available in the 6.034 data set, this experiment was only run against my personal data set.

4.4 Miscellaneous Experiments

Although I don't describe them here in detail, I did conduct several other experiments. For example, after conducting Experiment 2, I ran a full set of tests to verify that the stemming and tokenization parameters used really were optimal for each of the feature selectors tested, rather than just the ad-hoc feature selector. I also conducted tests to determine the optimal number of probabilities used for each type of feature selector.

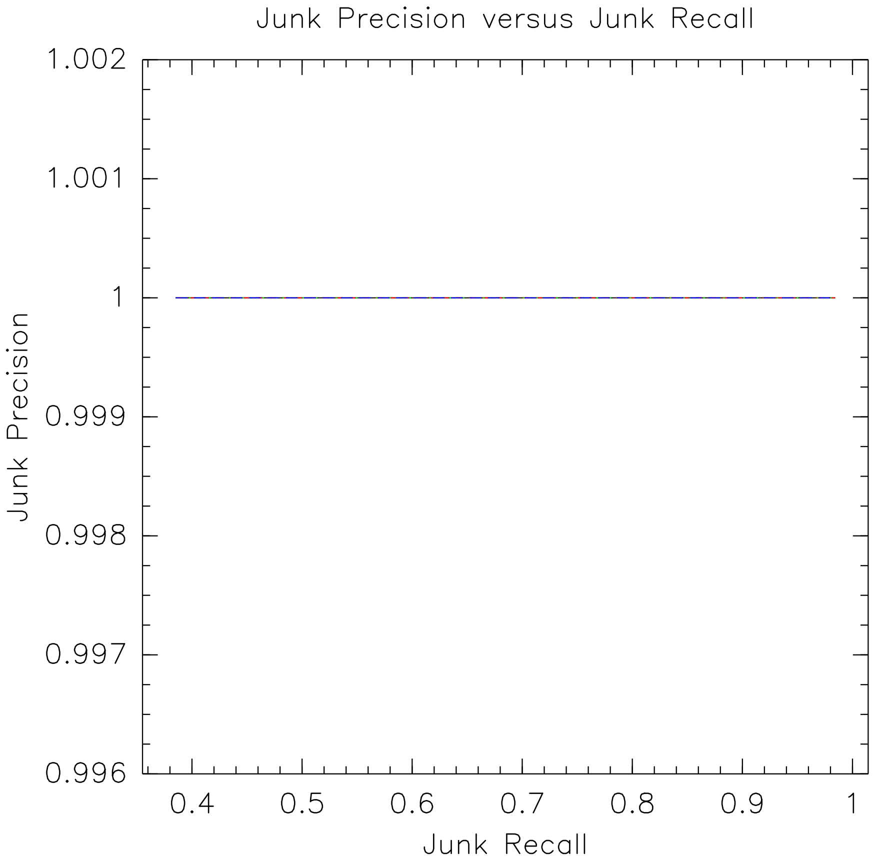
5 Results

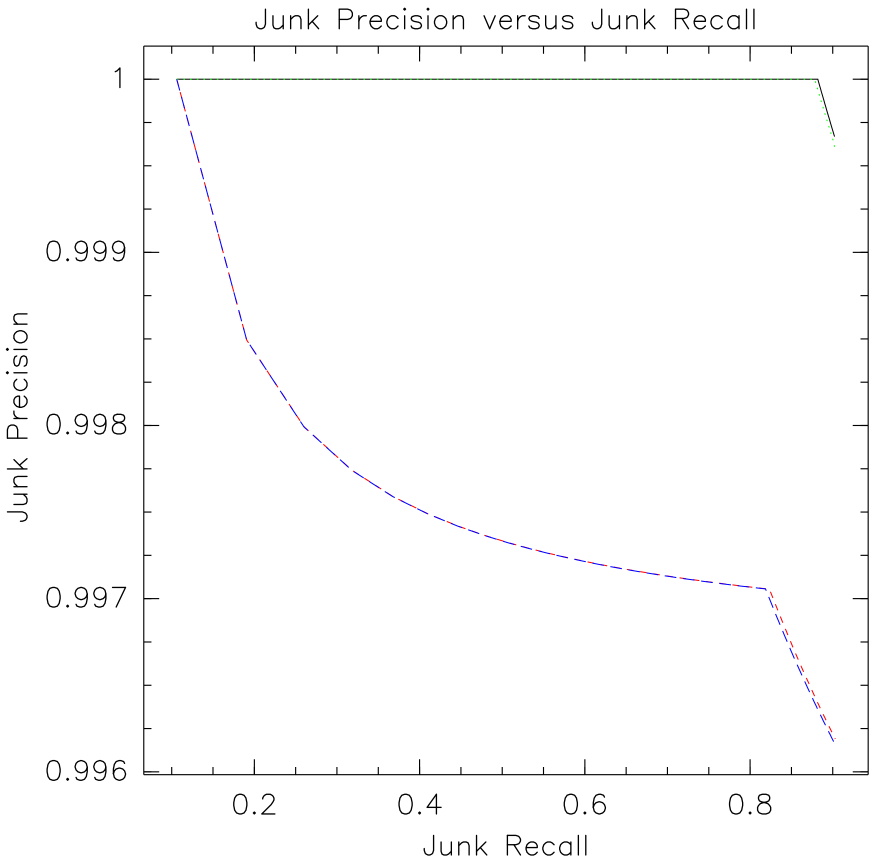
5.1 Experiment 1

The results are shown in the two figures below. Although its difficult to see because the curves overlap significantly, each graph consists of four curves. The first two curves show classifier performance without stemming for both of the two tokenizers. The second two curves show classifier performance for both of the tokenizers while employing a stemming algorithm.

The first graph shows classifier performance over the 6.034 data set. It indicates that for many thresholds, the classifier correctly classifies all of the messages regardless of whether stemming is enabled or which tokenizer is used. The second graph shows classifier performance over my personal data set. The two overlapping curves near the top correspond to the first tokenizer both with and without stemming. The bottom two curves correspond to the second tokenizer both with and without stemming. In each pair of curves, the lower of the two represents performance with stemming enabled.

These results indicate that stemming very slightly reduces classifier accuracy and that we're better off without it. What really improves performance is a tokenizer that doesn't split punctuation from adjacent words. Both of these results are counterintuitive.



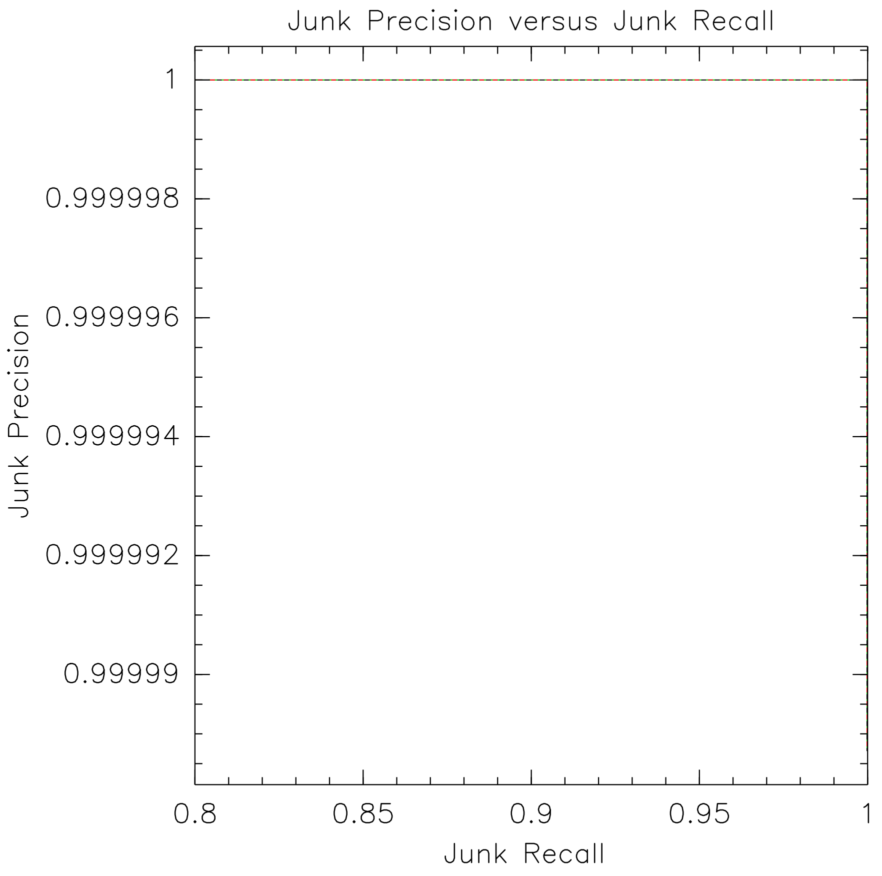


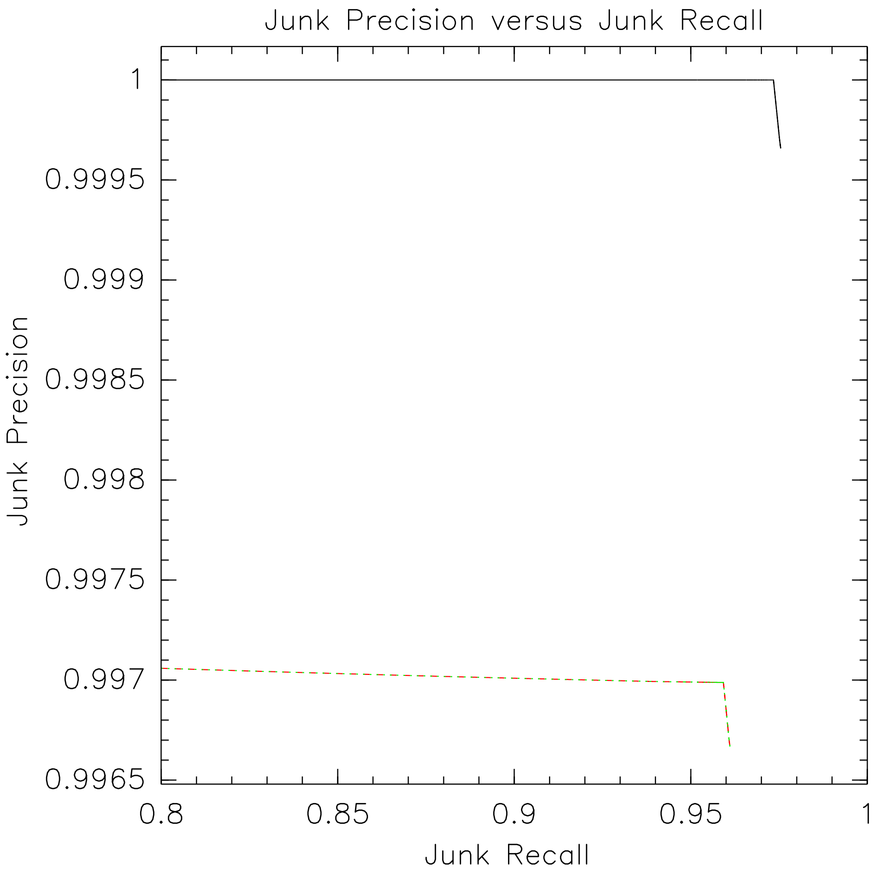
5.2 Experiment 2

The first of the following graphs compares classifier performance against the 6.034 dataset for the ad-hoc, mutual information and hypothesis testing feature selectors. The second graph shows the same tests executed against my personal data set.

For the 6.034 dataset, the three curves overlap completely. This means that the classifier correctly classifies all examples regardless of which feature selector was used. In contrast, on my personal data set, feature selector choice has a clear effect on performance. The top curve corresponds to the ad-hoc feature selector. The bottom curve actually consists of two overlapping curves for the mutual information and hypothesis testing feature selectors.

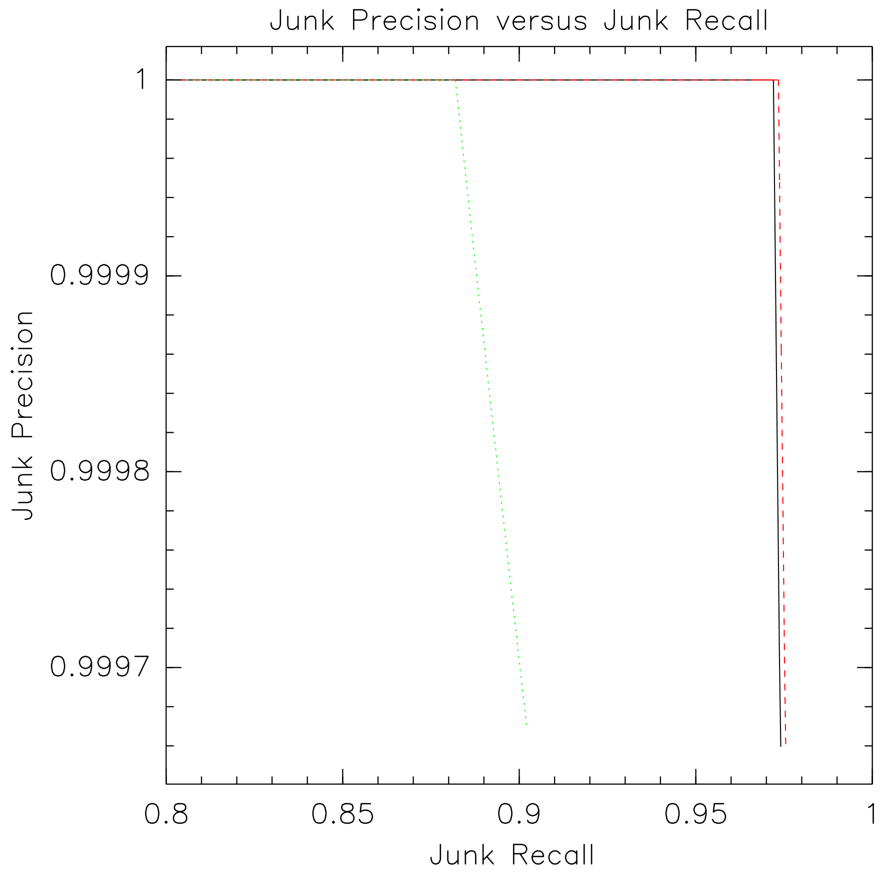
Contrary to expectations, the ad-hoc feature selector significantly outperformed both of the more sophisticated feature selectors.


Junk Precision versus Junk Recall
Junk Precision
1
0.999998
0.999996
0.999994
0.999992
0.99999
Junk Recall
0.8
0.85
0.9
0.95
1



5.3 Experiment 3

The results of Experiment 3 are shown in the graph below. In order from left to right, the curves represent word frequencies alone, combined heuristics with word frequencies, and heuristics alone. All three classifiers perform well, but surprisingly, the combination of heuristics and word frequencies performs slightly poorer than heuristics alone.



6 Contributions

MeatSlicer represents a novel approach to detecting spam. The key idea is that any heuristic test can be wrong, so we shouldn't rely on any one test exclusively; instead, we should use individual heuristic tests to accumulate evidence for or against the hypothesis that a particular message is spam. In fact, optimal performance is achieved when we rely solely on heuristic tests without any word frequency data.

This work provides other benefits as well. The use of corpus-based zero-knowledge probability estimators for unseen data parallels Good-Turing smoothing but may very well be a novel contribution. In addition, I discovered that the sophisticated feature selectors require far more probabilities to reach their optimal performance than the ad-hoc feature selector. This may explain why their optimal performance is inevitably lower than the ad-hoc one: more probabilities means more noise.

The other surprising results include the effects of tokenization and stemming. Stemming is slightly detrimental for this classifier; I can only speculate that it would prove more beneficial if I had used a bigram or trigram language model. In a similar vein, tokenization algorithms that don't break apart words from their punctuation seem to provide much better performance than those that do. This result needs careful study. As with stemming, it may be simply an artifact of my use of unigram language models, or it may indicate something more significant.

Finally, the poor performance of the sophisticated feature selectors suggests another avenue for research. Rennie suggests [4] that researchers have long overestimated the benefit of such classifiers; in some ways this work provides a good experimental verification of his theoretical criticisms of information theory based feature selectors.

References

[1] Paul Graham. A plan for spam, Aug 2002. Available at `http://www.paulgraham.com/spam.html`.

[2] Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing.* The MIT Press, Cambridge, Massachusetts, 1999.

[3] J. Postel. RFC 821: Simple mail transfer protocol. Technical Report 821, Internet Engineering Task Force, aug 1982. Available at `ftp://ftp.internic.net/rfc/rfc788.txt`.

[4] J. Rennie. Improving Multi-Class Text Classification with Naive Bayes. Master's thesis, Massachusetts Institute of Technology, 2001. Available at `ftp://publications.ai.mit.edu/ai-publications/AITR-2001-004.ps`.

[5] Mehran Sahami, Susan Dumais, David Heckerman, and Eric Horvitz. A bayesian approach to filtering junk E-mail. In *Learning for Text Categorization: Papers from the 1998 Workshop*, Madison, Wisconsin, 1998. AAAI Technical Report WS-98-05.

A Source Code

A.1 meatslicer.py

```
#!/usr/bin/env python
from random import shuffle
from heuristic import checks, checkNames
from bayes import messageFreqs, collection, filter
from Queue import Queue, Empty, Full
from threading import Thread, activeCount, currentThread
from thread import exit
from mailbox import PortableUnixMailbox
import email, time, shelve

def workerThread(message, resultQueue):
    print "worker thread started", currentThread().getName()
    check_results = [None] * len(checks)
    msgHash = hash(message.as_string())
    newIndices = range(len(checks))
    shuffle(newIndices)
    for i in newIndices:
        # implement caching by hashing on the (check_func.func_code,
        # message) key
        check_func = checks[i]
        cacheKey = str((check_func.__name__, msgHash))
        if cache.has_key(cacheKey):
            print '*'*4, "retrieving cached value for function", check_func.__name__
            check_results[i] = cache[cacheKey]
        else:
            try:
                startTime = time.time()
                check_results[i] = check_func(message)
                endTime = time.time()
                writeOutLog.put((cacheKey, check_results[i]))
                print '*'*4, "saving cached value for function", check_func.__name__, endTime - star
            except StandardError, einfo:
                print "unknown exception calculating check function", check_func.__name__
                print "\t\t", einfo
    print 'calculating frequencies now...'
    freqs = messageFreqs(message.get('Subject', '') +
                    message.get_payload(decode=1))
    freqs.unintegratedHeuristicResults = check_results
    resultQueue.put(freqs)
    print "worker thread dying", currentThread().getName()
    exit()

def writerThread():
    while 1:
        key, val = writeOutLog.get()
        cache[key] = val
        cache.sync()

def managerThread():
    while 1:
        if activeCount() <= maxWorkQueueSize:
            nextJob = workQueue.get()
            print "manager thread starting new worker thread"
            newWorker = Thread(target = workerThread,
                               args = nextJob)
            newWorker.start()
        else:
            time.sleep(0.001)

def main():
    # first we train...
    dataDirectory = 'data-mike/'

    spamResultQ = Queue()
    notSpamResultQ = Queue()
    spamCollection = collection(())
    notSpamCollection = collection(())

    for resultQ, fname, col in ((spamResultQ, 'spam.mbox', spamCollection),
                                (notSpamResultQ, 'nonspam.mbox', notSpamCollection)):
        # replace with mmap?
        f = file(dataDirectory + fname, 'r')
        totalMessagesSent = totalMessagesRecieved = 0

        for msg in PortableUnixMailbox(f, email.message_from_file):
            if not(msg.is_multipart()):
                print "new message on the table"
                # ignore MIME crap...
                workQueue.put((msg, resultQ))
                totalMessagesSent += 1

                # now see if any results are done yet...
                while 1:
                    try:
                        finishedJob = resultQ.get_nowait()
                        totalMessagesRecieved += 1
                        col.addMessage(finishedJob)
                    except Empty:
                        break
        f.close()
        while totalMessagesRecieved != totalMessagesSent:
            finishedJob = resultQ.get()
            totalMessagesRecieved += 1
            col.addMessage(finishedJob)
        #col.integrateHeuristicResults()
        print '//////'*5, "finished with loop for file", fname

if __name__ == '__main__':
    cache = shelve.open('cache.shelve')
    try:
        maxWorkQueueSize = 100
        workQueue = Queue(maxWorkQueueSize)
        writeOutLog = Queue()

        writer = Thread(target = writerThread)
        writer.start()
        manager = Thread(target = managerThread)
        manager.start()

        main()
    finally:
        cache.close()




## training:
## main thread reads message objects and tosses them on the work queue
## manager thread waits on work queue and dispatches incoming jobs to new
## worker threads. (he may also monitor job progress and dispatch a new worker
##                  thread if someones been stalled forever on a job)

## calls worker thread with message and output queue that result should go on
## (one queue for spam and one for nonspam?)

## worker thread shuffles checklist and calls each of the checks
## it also calls killspam.message on each message to calculate freqs
## together, killspam.message object (which only has freq results) and
## checks results list along with original email.message object are packaged
## up and stashed on the results queue

## in each loop through, the main thread checks to see if any new results
## are immediately available. if they are, it grabs them and sticks them
## in the right place. if not, it continues looping. outside of the loop
## (afterwards), it keeps issuing blocking reads on the result queue
## until all jobs have been accounted for.
## maybe we should stash check_results in messageFreqs;

# normalize both then sum and stash the normalization parameters in
# the filter instance... in filter.__call__, use the filter
# instance's normalization parameters
# to normalize the messages heuristic results with
# messageFreq.normalizeAndIntegrateHEuristic(norm param1, norm param2)

# write one normalization function that works on both collections and messageFreqs
# it will take the collection/messageFreq and smallest, largest and then add
# the heuristic "words" to the collection/messageFreq's countDict
# we call the norm function once for each collection in filter init
# and then once again on each messageFreq in filter's __call__
```

A.2 bayes.py

```
#!/usr/bin/env python
from __future__ import division
from re import compile
from operator import mul
from Stemmer import Stemmer
import biggles
from math import log
from mx.Number import *
from operator import div
from Numeric import array, sum
from heuristic import checkNames

englishStemmer = Stemmer('english')

class messageFreqs:
    tokenRE1 = compile(r"""[A-z0-9\.\!,\;\:\?]+""")
    tokenRE2 = compile(r"""[A-z]+|[0-9]+|[^A-z0-9 \t\n\r\f\v]+""")

    def __init__(self, text, useStemmer=0, tokenRE=2):
        self.useStemmer = useStemmer
        if tokenRE == 1:
            self.tokenRE = self.tokenRE1
        elif tokenRE == 2:
            self.tokenRE = self.tokenRE2
        else:
            raise ValueError(
                'Bad tokenRE selector in messageFreqs constructor:' + str(tokenRE))

        self.countDict = {}
        tokenStream = self.tokenize(text)
        self.tokenCount = len(tokenStream)
        self.countDict = self.count(tokenStream)

        self.orderedTokenList = [(count, word) for (word, count)
                                 in self.countDict.items()]
        self.orderedTokenList.sort()

        assert self.check(), "Message object integrity check failure"

    def tokenize(self, text):
        lowerText = text.lower()
        tokens = self.tokenRE.findall(lowerText)
        if self.useStemmer:
            return map(englishStemmer.stem, tokens)
        else:
            return tokens

    def count(self, tokenStream):
        d = {}
        for token in tokenStream:
            d[token] = d.setdefault(token, 0) + 1
        return d

    def topWords(self, n):
        return [word for (count, word) in self.orderedTokenList[:n]]

    def check(self):
        total = 0
        for count in self.countDict.values():
            total += count
        return (total == self.tokenCount)


def messageListFile(fname):
    messageSplitterRE = compile(
        '---------- (?:NOT)?SPAM (?:[0-9]+) ----------')
    text = open(fname, 'r').read()
    messageTexts = [text for text in messageSplitterRE.split(text) if text]
    return map(messageFreqs, messageTexts)


class collection:
    def __init__(self, messageList):
        self.tokenCount = 0
        self.messageCount = 0
        self.countDict = {}
        self.unintegratedHeuristicVectors = []
        for m in messageList:
            self.addMessage(m)

    def addMessage(self, m):
        if hasattr(m, 'unintegratedHeuristicResults'):
            self.unintegratedHeuristicVectors.append(
                m.unintegratedHeuristicResults)
        self.tokenCount += m.tokenCount
        self.messageCount += 1
        d = self.countDict
        for word, count in m.countDict.items():
            d[word] = d.setdefault(word, 0) + count

    def probability(self, token):
        count = self.countDict.get(token, 0)
        return count, self.tokenCount



class filter:
    maxMessageWords = 1000000
    def __init__(self, spamCollection, notSpamCollection,
                 featureSelector=None, minSpamCount=0,
                 minNonSpamCount=0, minTotalCount=1,
                 maxProbabilityListLength=10):
        self.spamCollection = spamCollection
        self.notSpamCollection = notSpamCollection
        self.featureSelector = featureSelector
        self.minSpamCount = minSpamCount
        self.minNonSpamCount = minNonSpamCount
        self.minTotalCount = minTotalCount
        self.maxProbabilityListLength = maxProbabilityListLength
        self.cache = {}

        # create a candidate feature list comprised of words
        # that appear often enough...
        candidateWordSet = {}
        for archive in (self.spamCollection, self.notSpamCollection):
            for word in archive.countDict.keys():
                if not(candidateWordSet.has_key(word)):
                    s = spamCollection.probability(word)[0]
                    n = notSpamCollection.probability(word)[0]
                    if ((s >= minSpamCount) and
                        (n >= minNonSpamCount) and
                        (s + n >= minTotalCount)):
                        candidateWordSet[word] = s + n
        self.candidateWordSet = candidateWordSet
        self.wordTypeCount = len(self.candidateWordSet.keys())

        col1 = self.spamCollection
        col2 = self.notSpamCollection
        # list addition creates a shallow copy of both lists
        if col1.unintegratedHeuristicVectors:
            x = array(col1.unintegratedHeuristicVectors +
                      col2.unintegratedHeuristicVectors)
            smallest = min(x)
            largest = max(x)
            del x
            self.smallest, self.largest = smallest, largest
            for col in (col1, col2):
                col.unintegratedHeuristicResults = sum(
                    col.unintegratedHeuristicVectors)
                self.integrateHeuristicResults(col)

    def integrateHeuristicResults(self, obj):
        # obj can be either a collection or a messageFreq
##          print obj.unintegratedHeuristicResults
##          print self.smallest
##          print self.largest
##          normalizedData = div((obj.unintegratedHeuristicResults
##                                 - self.smallest),
##                                (self.largest - self.smallest))
        normalizedData = obj.unintegratedHeuristicResults
        for name, value in zip(checkNames, normalizedData):
            obj.countDict['heuristic:' + name] = value


    def spamProbability(self, word):
        spamCount, spamTotal = self.spamCollection.probability(word)
        notSpamCount, notSpamTotal = self.notSpamCollection.probability(word)
        if spamCount == 0:
            probSpam = 1. / (spamTotal + notSpamTotal)
        elif notSpamCount == 0:
            probSpam = 1 − (1. / (spamTotal + notSpamTotal))
        else:
            probSpam = (spamCount / spamTotal) / (
                (spamCount / spamTotal) + (notSpamCount / notSpamTotal))

        if word[:10] == 'heuristic:':
            goodness = 1e40
        else:
            goodness = abs(probSpam − 0.5)

        return (goodness, probSpam, word)
    def __call__(self, messageToFilter):
        if hasattr(messageToFilter, 'unintegratedHeuristicResults'):
            self.integrateHeuristicResults(messageToFilter)

        probabilities = []
        for word in messageToFilter.topWords(self.maxMessageWords):
            if self.candidateWordSet.has_key(word):
                probabilities.append(self.spamProbability(word))
                if not(self.cache.has_key(word)):
                    self.cache[word] = self.spamProbability(word)
                probabilities.append(self.cache[word])

        probabilities.sort()
        probabilities.reverse()

##          # make sure that heuristics are included...
##          d = {}
##          for goodness, prob, word in probabilities[:self.maxProbabilityListLength]:
##              d[word] = 1
##          for checkName in checkNames:
##              name = 'heuristic:' + checkName
##              if not(d.has_key(name)):
##                  probabilities.insert(0, self.spamProbability(name))

        p = [prob for (goodness, prob, word)
             in probabilities[:self.maxProbabilityListLength]]

        p = map(Float, p)
        c = reduce(mul, p, 1)
        d = reduce(mul, [1 - probSpam for probSpam
                         in p if (1-probSpam) > 0], 1)
        return div(c, (c + d))


class MIFilter(filter):
    def spamProbability(self, word):
        spamCount, spamTotal = self.spamCollection.probability(word)
        notSpamCount, notSpamTotal = self.notSpamCollection.probability(word)
        k = (spamTotal + notSpamTotal)
        if spamCount == 0:
            probSpam = 1 / k
        elif notSpamCount == 0:
            probSpam = 1 - (1 / k)
        else:
            probSpam = (spamCount / spamTotal) / (
            (spamCount / spamTotal) + (notSpamCount / notSpamTotal))

        fs = max(spamCount, 1 / k)
        Ns = spamTotal
        fn = max(notSpamCount, 1 / k)
        Nn = notSpamTotal
        N = Ns + Nn
        infoGain = ((fs / N) * log((fs/(fs+fn))/(Ns/N)) +
                    ((Ns - fs)/N) * log(((Ns - fs)/(N - (fs+fn)))/(Ns/N)) +
                    (fn / N) * log((fn/(fs+fn))/(Nn/N)) +
                    ((Nn - fn)/N) * log(((Nn - fn)/(N - (fs+fn)))/(Nn/N)))
        return (infoGain, probSpam, word)


class HTFilter(filter):
    def spamProbability(self, word):
        spamCount, spamTotal = self.spamCollection.probability(word)
        notSpamCount, notSpamTotal = self.notSpamCollection.probability(word)
        k = (spamTotal + notSpamTotal)
        if spamCount == 0:
            probSpam = 1 / k
        elif notSpamCount == 0:
            probSpam = 1 - (1 / k)
        else:
            probSpam = (spamCount / spamTotal) / (
                (spamCount / spamTotal) + (notSpamCount / notSpamTotal))

        fs = max(spamCount, 1 / k)
        Ns = spamTotal
        fn = max(notSpamCount, 1 / k)
        Nn = notSpamTotal
        N = Ns + Nn
        HT = 2 * (fs * log((fs / (fs + fn)) / (Ns / N)) +
                  fn * log((fn / (fs + fn)) / (Nn / N)))
        return (HT, probSpam, word)

def results(filter, spamMessages, notSpamMessages):
    print 'testing spam messages...'
    spamMessagesScores = map(filter, spamMessages)
    print 'testing not spam messages...'
    notSpamMessagesScores = map(filter, notSpamMessages)

    spamCorrect = spamInCorrect = 0
    notSpamCorrect = notSpamInCorrect = 0
    junkPrecision = []
    junkRecall = []
    for t in range(0, 10001):
        threshold = t / 10000.
        for m in spamMessagesScores:
            if m >= threshold:
                spamCorrect += 1
            else:
                spamInCorrect += 1
        for m in notSpamMessagesScores:
            if m < threshold:
                notSpamCorrect += 1
            else:
                notSpamInCorrect += 1
        junkPrecision.append(spamCorrect / (spamCorrect + notSpamInCorrect))
        junkRecall.append(spamCorrect / (spamCorrect + spamInCorrect))
    return junkPrecision, junkRecall

colors = [ "black", "red", "green", "blue", "magenta", "cyan" ]


if __name__ == '__main__':
    print 'Loading spam messages...'
    spamMessages = messageListFile('data/spam.txt')
    #spamMessages = messageListFile('data/spam.txt')

    print 'Loading non spam messages...'
    notSpamMessages = messageListFile('data/notspam.txt')
    #notSpamMessages = messageListFile('data/notspam.txt')


    print 'analyzing spam messages...'
    spamCollection = collection(spamMessages)
    print 'analyzing non spam messages...'
    notSpamCollection = collection(notSpamMessages)
    print 'filtering...'


    new_filter_output = ('red',
                         results(filter(spamCollection,
                                        notSpamCollection),
                                        #maxProbabilityListLength=500),
                                 spamMessages, notSpamMessages))
##    old_filter_output = ('blue',
##                         results(filter(spamCollection,
##                                        notSpamCollection),
##                                        #maxProbabilityListLength=500),
##                                 spamMessages, notSpamMessages))
    output = (new_filter_output, new_filter_output)#, old_filter_output)
    ## output = zip(colors,
    ##              [results(MIFilter(spamCollection,
    ##                                notSpamCollection,
    ##                                maxProbabilityListLength=x*10),
    ##                       spamMessages, notSpamMessages)
    ##               for x in range(6)])

    p = biggles.FramedPlot()
    p.title = "Junk Precision versus Junk Recall"
    p.xlabel = "Junk Recall"
    p.ylabel = "Junk Precision"

    for (color, (junkRecall, junkPrecision)) in output:
        print (junkRecall, junkPrecision)
        print hash(tuple(junkRecall)), hash(tuple(junkPrecision))
        p.add(biggles.Curve(junkRecall, junkPrecision, color=color))

##      f = MIFilter(spamCollection, notSpamCollection)
##      for x in range(4,6):
##          #minSpamCount = 1, minNonSpamCount = 1,
##          f.maxProbabilityListLength = 200 + 120*x
##          #f.maxProbabilityListLength = 150 + 75*x
##          jr, jp = results(f, spamMessages, notSpamMessages)
##          p.add(biggles.Curve(jr, jp, color=colors[x]))
##      jr, jp = results(filter(spamCollection, notSpamCollection),
##                       spamMessages, notSpamMessages)
##      p.add(biggles.Curve(jr, jp, type='dashed'))

    p.xrange = 0, 1
    p.write_eps('output.eps')
    p.show()




# hyper optimized version...too bad its no faster than the original...

## class messageListFile:
##      def __call__(self, fname):
##          # return a list of message objects given a text file
##          # containing a sequence of messages separated by...

##          f = file(fname, 'r')
##          f.seek(0,2) #seek to end of file
##          size = f.tell()
##          f.seek(0)
##          text = mmap(f.fileno(), size, prot=PROT_READ)

##          messageTexts = []
##          counter = 0

##          messageSplitterRE = compile(
##              '———- (?:NOT)?SPAM (?:[0-9]+) ———-')
##          match = messageSplitterRE.search(text)
##          startPosition = match.end()
##          match = messageSplitterRE.search(text, startPosition)
##          endPosition = 0
##          while match:
##              endPosition = match.start()
##              messageTexts.append(message(text[startPosition:endPosition]))
##              startPosition = match.end()
##              match = messageSplitterRE.search(text, startPosition)
##              counter += 1

##          text.close()
##          del text
##          f.close()
##          del f
##          gc.collect()
##          print counter
##          return messageTexts
## messageListFile = messageListFile()
```

A.3 heuristics.py

```
#!/usr/bin/env python
from __future__ import division
import socket
from re import compile
from os import popen2
from operator import add
import GeoIP
from math import pi, cos, sin, acos
import DNS, smtplib
from email.Utils import parseaddr, parsedate_tz, mktime_tz
from geographic_data import *

# blackhole stuff...
ipRE = compile(r'\[\s*([0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3})\s*\]')

blackholes = [' blackholes.mail-abuse.org',
           'relays.osirusoft.com',
            'list.dsbl.org',
            'multihop.dsbl.org',
            'relay.ordb.org']

def blackhole_check(message):
    score = 0

    # first get the originating host and all relays...
    canidateHosts = ipRE.findall(''.join(
        message.get_all('received', '')))

    # then starting with the originator, iterate through
    #  all spam detectors...
    for host in canidateHosts:
        for spamTrap in blackholes:
            #print "\tTrying host %s with relay %s" % (host, spamTrap)
            # assume host is in dotted quad notation
            # returns true if they're evil spammers, otherwise false
            octets = host.split('.')
            assert len(octets) == 4
            octets.reverse()
            lookupHost = '.'.join(octets) + '.' + spamTrap
            try:
                addr = socket.gethostbyname(lookupHost)
                score += 1
            except socket.error:
                addr = ''
    return score


# DCC...
dccRE = compile("\=([0-9]+)")
magic_DCC_many_value = 1000

def dcc_check(message):
    # sample DCC output line:
    # X-DCC-sackHeads-Metrics: beg-for-more 1012; Body=0 Fuz1=0 Fuz2=0
    child_stdin, child_stdout = popen2("dccproc -QH")
    child_stdin.write(message.as_string())
    child_stdin.close()
    output = child_stdout.read()
    child_stdout.close()

    if output.lower().find('many') != -1:
        return magic_DCC_many_value
    else:
        scores = map(float, dccRE.findall(output))
        return reduce(add, scores, 0)


# hop count...
def hopcount_check(message):
    return len(message.get_all('received', ()))


# countries involved...
gi = GeoIP.new(GeoIP.GEOIP_MEMORY_CACHE)

def country_check(message):
    countries = {}
    canidateHosts = ipRE.findall(''.join(
        message.get_all('received', '')))
    for host in canidateHosts:
        countries[gi.country_code_by_name(host)] = 1
    return max(len(countries.keys()) - 1, 0)


# path distance...
def degreesToRadians(degrees, minutes, direction):
    rads = (degrees + (minutes / 60)) * pi / 180
    d = direction.lower()
    if d in 'sw':
        return -1 * rads
    elif d in 'ne':
        return rads
    else:
        raise ValueError('degrees spec broken')

EarthRadius = 3963.1 # miles
def latitudeLongitudeDistance(lat1, lon1, lat2, lon2):
    # returns distance in miles between those two places
    # given latitude and longitude in radians
    if (abs(lat1 - lat2) < 0.0001) and (abs(lon1 - lon2) < 0.0001):
        return 0
    return abs(EarthRadius *
            acos(cos(lat1)*cos(lon1)*cos(lat2)*cos(lon2) +
                cos(lat1)*sin(lon1)*cos(lat2)*sin(lon2) +
                sin(lat1)*sin(lat2)))

def distance_check(message):
    canidateHosts = ipRE.findall(''.join(
        message.get_all('received', '')))
    hostPairs = zip(canidateHosts, canidateHosts[1:])
    totalDistance = 0
    for host1, host2 in hostPairs:
        a = gi.country_name_by_name(host1)
        lat1, lon1 = countryLocationData.get(
            a, defaultCoordinates)
        if a and not(countryLocationData.has_key(a)):
            print '\t country not found:', a

        b = gi.country_name_by_name(host2)
        lat2, lon2 = countryLocationData.get(
            b, defaultCoordinates)
        if b and not(countryLocationData.has_key(b)):
            print '\t country not found:', b

        distance = latitudeLongitudeDistance(lat1, lon1, lat2, lon2)
        totalDistance += distance

    return totalDistance

## >>> from smtplib import SMTP
## >>> s=SMTP('pacific-carrier-annex.mit.edu')
## >>> s.helo()
## (250, 'pacific-carrier-annex.mit.edu Hello BEG-FOR-MORE.MIT.EDU [18.209.0.60], pleased to meet you')
## >>> s.docmd('RCPT', 'TO:msalib@mit.edu')
## (503, 'Need MAIL before RCPT')
## >>> s.docmd('MAIL', 'FROM:foo@beg-for-more.mit.edu')
## (250, 'foo@beg-for-more.mit.edu... Sender ok')
## >>> s.docmd('RCPT', 'TO:msalib@mit.edu')
## (250, 'msalib@mit.edu... Recipient ok')
## >>> s.docmd('RCPT', 'TO:msalibd@mit.edu')
## (550, 'msalibd@mit.edu... User unknown')
## >>> s.docmd('RCPT', 'TO:msalibd@mit.eddu')
## (250, 'msalibd@mit.eddu... Recipient ok')
## >>> s.quit()

# SMTP verify...
DNS.ParseResolvConf()

magic_smtp_no_address_score = 1000
magic_smtp_server_error = 500
magic_smtp_user_doesnt_exist = 500

# 4 possible cases:
# 1. no valid address
# 2. server reports an error when we chat
# 3. verify works
# 4. verify doesn't work but rcpt accepts mail
# 5. neither verify nor rcpt work
# if 252,502: verify doesn't work so try rcpt
# if 550: verify works and says this address is bad

#verify
#if 252, 502

def smtp_check(message):
    # first try the reply-to header
    repaddr = message.get('Reply-To', '')
    name, emailaddr = parseaddr(repaddr)
    if not(emailaddr):
        # try the from header
        fromaddr = message.get('From', '')
        name, emailaddr = parseaddr(fromaddr)
        if not(emailaddr):
            return magic_smtp_no_address_score

    # now we have a valid email address to play with...
    username, network = emailaddr.split('@')
    try:
        mailServers = DNS.mxlookup(network)
    except DNS.Base.DNSError:
        print 'DNS timeout on address', emailaddr
        return magic_smtp_no_address_score

    if not(mailServers):
        return magic_smtp_no_address_score

    # this extra conversation seems to slow us
    # down trmendously, probably due to anti spamming
    # throttling features in the mail servers, so lets
    # just ignore it. Besides, most of the value of this
    # test seems to come from the mxcheck on return address
    return 0


    # just pick of the highest scoring mail server
    mailServer = mailServers[0][1]
    try:
        server = smtplib.SMTP(mailServer)
    except (StandardError, socket.error, smtplib.SMTPException):
        print 'socket error: mail server refused connection'
        return magic_smtp_no_address_score

    try:
        server.helo()
        errorCode, stuff = server.docmd(
            'MAIL', 'FROM:foo@beg-for-more.mit.edu')
        if errorCode > 400:
            return magic_smtp_server_error
        errorCode, stuff = server.docmd('RCPT', 'TO:' + emailaddr)
        server.quit()
        #print errorCode, stuff, 'boo'
        if errorCode > 400:
            # user doesn't exist
            return magic_smtp_user_doesnt_exist
        else:
            # user does exist: they're legit...
            return 0
    except (smtplib.SMTPException, socket.error):
        print 'smtp exception'
        server.quit()
        return magic_smtp_server_error



# message id checks...
## 5 cases:
##     1. no header present : 1000
##     2. header present but empty : 1000
##     3. header present but not valid (check for more than 1 @) : 1000
##     4. header present and valid but DNS can't resolve name : 10
##     5. header present and valid and DNS can resolve name : 0

dnsNameRE = compile("^[A-z_\-0-9\.]+$")

magic_message_id_empty_or_invalid = 1000
magic_message_id_no_dns = 10

def message_id_check(message):
    value = message.get('message-id', '')
    if not value:
        return magic_message_id_empty_or_invalid
    pieces = parseaddr(value)[1].strip().split('@')
    if len(pieces) == 2:
        muaID, sendingHostName = pieces
        if not(dnsNameRE.match(sendingHostName)):
            #print '\t\t', sendingHostName, 'boo'
            return magic_message_id_empty_or_invalid
        # muaID will be completely random with no specified
        # structure whatsoever according to the RFC.
        # the host however should be a DNS name, although
        # not necessarily a FQDN or a globally addressable one
        try:
            ipAddress = socket.gethostbyname(sendingHostName)
            #print '\t message ID host and IP:', sendingHostName, ipAddress
            return 0
        except socket.error:
            #print '\t message ID host:', sendingHostName
            return magic_message_id_no_dns
    else:
        return magic_message_id_empty_or_invalid

# nonalphanumeric character checks...
nonAlphaChar = compile("[^A-z0-9\.,\;\:\'\"\s]+")

def nonAlphaNumericCharSubject_check(message):
    subj = message.get('Subject', '')
    if len(subj):
        return reduce(add, map(len, nonAlphaChar.findall(subj)), 0) / len(subj)
    else:
        return 0

def nonAlphaNumericCharBody_check(message):
    body = message.get_payload(decode=1)
    if len(body):
        return reduce(add, map(len, nonAlphaChar.findall(body)), 0) / len(body)
    else:
        return 0


#time travel checks...

# we take abs because spam tends to have both large positive
# large negative delays: either it takes forever to get here
# b/c its being routed around the world 8 times or its forged
# to appear like its coming in the future...

def time_travel_check(message):
    # pick the most recent one
    recievedLine = message.get_all('Received', None)
    if not(recievedLine):
        print '\tno recieved line?!'
        return 0
    recievedTimeString = recievedLine[0].split(';')[-1].strip()
    sendersTimeString = message.get('date', recievedTimeString)
    try:
        hoursDifferent = (mktime_tz(parsedate_tz(sendersTimeString)) -
                          mktime_tz(parsedate_tz(recievedTimeString))) / 3600.
        return abs(hoursDifferent)
    except:
        #print '\tsome random error'
        return 0


# forward
# reject mail from hosts without reverse DNS or whose PTR RRs don't
# match their forward DNS values.

## check reverse DNS of ip relay, and compare to forward DNS for spoofing.

# reject mail from hosts without reverse DNS or whose PTR RRs don't
# match their forward DNS values.

# if i control an IP block, I can set the rdns to be whatever i want
# including your domain name, but i can't change what that domain name
# will forward resolve to: take the relay ip address (it should be the
# first one on that line) from the second recieved line and rDNS it;
# then forward DNS that name to get an IP address; if that IP matches
# the relay IP you started with, the machine is legit.

# do we want to do this for all the first ip addresses in all the
# recieved lines? maybe all recieved lines except the first and/or the
# last?

def forwardReverse_check(message):
    line = message.get_all('Received')[1]
    x = ipRE.findall(line)
    if len(x) == 0:
        return 20
    recievedIP = x[0].strip()
    try:
        x = socket.gethostbyaddr(recievedIP)
    except socket.error:
        # we can't even get a reverse DNS on this IP...
        return 20
    candidateNames = [x[0]] + x[1]
    for candidate in candidateNames:
        try:
            candidateIPs = socket.gethostbyname_ex(candidate)[2]
            for candidateIP in candidateIPs:
                if candidateIP == recievedIP:
                    return 0
        except socket.error:
            pass
    return 10
```

```
checks = [smtp_check,
          blackhole_check,
          forwardReverse_check,
          dcc_check,
          distance_check,
          message_id_check,
          nonAlphaNumericCharSubject_check,
          nonAlphaNumericCharBody_check,
          time_travel_check,
          hopcount_check,
          country_check]

checkNames = [f.__name__ for f in checks]

if __name__ == '__main__':
    from mailbox import UnixMailbox
    import email

    #fname = 'data-mike/nonspam.mbox'
    fname = 'data-mike/spam.mbox'
    scores = count = 0
    for m in UnixMailbox(file(fname, 'r'), email.message_from_file):
        if not(m.is_multipart()):
            #s = smtp_check(m)
            #s = distance_check(m)
            #s = blackhole_check(m)
            #s = message_id_check(m)
            #s = nonAlphaNumericCharSubject_check(m)
            #s = nonAlphaNumericCharBody_check(m)
            #s = time_travel_check(m)
            s = forwardReverse_check(m)
            scores += s
            count += 1
            print s, m['Subject'], scores/count
```

A.4 runTests.py

```
#!/usr/bin/env python
from __future__ import division
from bayes import messageFreqs, collection, filter, results, MIFilter, HTFilter
from heuristic import checks, checkNames
from mailbox import PortableUnixMailbox
import email, shelve
```

```
def test(dataDirectory, stemmerUsage, tokenizer):
    # first we train...
    spamCollection = collection(())
    notSpamCollection = collection(())

    spamMessages = []
    notSpamMessages = []

    for dataSet, col, messageList in (
        ('spam', spamCollection, spamMessages),
        ('notspam', notSpamCollection, notSpamMessages)):

        fname = dataDirectory + dataSet + '.mbox'
        dataFile = file(fname, 'r')

        for msg in PortableUnixMailbox(dataFile, email.message_from_file):
            if not(msg.is_multipart()):
                text = msg.get_payload(decode=1)
                if type(text) == type(''):
                    m = messageFreqs(text + msg.get('Subject', ''),
                                     stemmerUsage,
                                     tokenizer)
                    col.addMessage(m)
                    messageList.append(m)

    f = filter(spamCollection, notSpamCollection)
    m = MIFilter(spamCollection, notSpamCollection,
                 maxProbabilityListLength = 400)
    h = HTFilter(spamCollection, notSpamCollection,
                 maxProbabilityListLength = 400)
    # then we test...
    return [results(x, spamMessages, notSpamMessages) for x in (f, m, h)]

def part1Graphs():
    for dataDirectory in ('data/', 'data-mike/'):
        for stemmerUsage in (0, 1):
            for tokenizer in (1, 2):
                print "starting new run"
                f, m, h = test(dataDirectory, stemmerUsage, tokenizer)
                for name, data in (('standard', f),
                                   ('mi', m),
                                   ('ht', h)):
                    outputFileName = "%s_%s_%s_%s" % (dataDirectory[:-1],
                                                      name,
                                                      stemmerUsage, tokenizer)
                    f = file(outputFileName, 'w')
                f.write(repr(data))
                f.close()





def integratedTest(dataDirectory, stemmerUsage, tokenizer):
    # first we train...
    spamCollection = collection(())
    notSpamCollection = collection(())

    spamMessages = []
    notSpamMessages = []

    for dataSet, col, messageList in (
        ('spam', spamCollection, spamMessages),
        ('notspam', notSpamCollection, notSpamMessages)):

        fname = dataDirectory + dataSet + '.mbox'
        dataFile = file(fname, 'r')

        for msg in PortableUnixMailbox(dataFile, email.message_from_file):
            if not(msg.is_multipart()):
                text = msg.get_payload(decode=1)
                if type(text) == type(''):

                    msgHash = hash(msg.as_string())
                    checkResults = []
                    for func in checks:
                        cacheKey = str((func.__name__, msgHash))
                        if not(cache.has_key(cacheKey)):
                            print '*********missing cache key:', cacheKey
                        checkResults.append(cache.get(cacheKey, 0))

                    freqs = messageFreqs(text + msg.get('Subject', ''),
                                         stemmerUsage,
                                         tokenizer)
                    freqs.unintegratedHeuristicResults = checkResults

                    col.addMessage(freqs)
                    messageList.append(freqs)

    # heuristics only
    f = filter(spamCollection, notSpamCollection,
               maxProbabilityListLength = len(checks))
    heuristicsOnly = results(f, spamMessages, notSpamMessages)

    #combined
    f = filter(spamCollection, notSpamCollection,
               maxProbabilityListLength = len(checks) + 10)
    combined = results(f, spamMessages, notSpamMessages)
    return heuristicsOnly, combined

def part2Graphs():
    stemmerUsage = 0
    tokenizer = 1
    dataDirectory = 'data-mike/'
    print "starting new run"
    heuristicsOnly, combined = integratedTest(dataDirectory, stemmerUsage, tokenizer)
    for fname, data in (('heuristicsOnly', heuristicsOnly),
                        ('combined', combined)):
        f = file(fname, 'w')
        f.write(repr(data))
        f.close()

##      outputFileName = "%s_%s_%s" % (dataDirectory[:-1],
##                                     stemmerUsage, tokenizer)
##      f = file(outputFileName, 'w')
##      f.write(repr(data))
##      f.close()


if __name__ == '__main__':
    part1Graphs()
    cache = shelve.open('cache.shelve')
    #part2Graphs()
    cache.close()
```

A.5 buildGraphs.py

```
#!/usr/bin/env python
from __future__ import division
import sys
import biggles

p = biggles.FramedPlot()
p.title = "Junk Precision versus Junk Recall"
p.xlabel = "Junk Recall"
p.ylabel = "Junk Precision"

data = [(fname,) + eval(file(fname, 'r').read()) for fname in sys.argv[1:]]
```

```
styles = ['solid', 'dashed', 'dotted', 'longdashed']
colors = [ "black", "red", "green", "blue", "magenta", "cyan" ]

for (fname, junkPrecision, junkRecall), style, color in zip(data, styles, colors):
    p.add(biggles.Curve(junkPrecision, junkRecall, linetype = style, color=color))

p.yrange = 0.996, 1.002
#p.xrange = 0.8, 1.0
p.show()
p.write_eps('output.eps')
```