Keeping the Good Stuff In: Confidential Information Firewalling with the CRM114 Spam Filter & Text Classifier

William S. Yerazunis†[1], Mamoru Kato‡[2], Mitsunori Kori[2], Hideya Shibata[2], Kurt Hackenberg[1]

[1]Mitsubishi Electric Research Laboratories,
Cambridge, Massachusetts, USA
[2]Mitsubishi Electric Corporation Information Technology R&D Center,
Ofuna, Kamakura, Kanagawa, Japan

† Correspondence author (English) ‡ Correspondence author (Japanese)

Abstract: In this whitepaper we consider the problem of outbound-filtering of emails to prevent accidental leakage of confidential information, We examine how to do this with the GPLed open-source spam filter CRM114 and test the accuracy of this filter against a 10,000+ document corpus of hand-classified emails (both confidential and non-confidential) in Japanese. We look into what moving parts are involved in these filters, and how they can be set up. The results show that a hybrid of multiple CRM114 filters outperforms a human-crafted regular-expression filter by nearly 100x in recall, by detecting > 99.9% of confidential documents, and with a simultaneous false alarm rate of less than 6%. As the programmers creating the machine-learning programs don't know how to read or write Japanese, this problem is an almost ideal case of the Searle "Chinese Room" problem.

Introduction: In order to prevent inadvertent disclosure of confidential information, many companies have been forced to institute strong policies on email being sent from company servers. In some cases, these policies require every outgoing email from an employee to be reviewed by that employees manager before the email is released from the internal server to the Internet. In other cases, enforcement is purely reactive, but punishing a "leaking" employee doesn't un-leak the information that was released; the damage is still done.

A side effect of such policies is that users become tempted to "cheat" - that is, to circumvent the policy. Although such strong policies can be "system enforced", they impose a huge load on the management chain and greatly reduce the ability of the organization to rapidly respond to outside questions or issues. Finally, these policies may be simply ineffective if the second reviewer is not extremely well-versed in the current list of confidential versus non-confidential topics for the entire organization.

It should be noted that there are really three categories of employees:

1. The perfect user – never makes a mistake
2. The human user – needs some help occasionally
3. The malicious user – intentionally breaks policy (possibly a paid industrial spy)

Of these, we can only help users in category 2. Category 1 users (perfect) don't need our help, and category 3 users (industrial spies) will circumvent this plaintext filtering with encryption, steganography, or physical compromise such as a hidden USB memory stick, and this research does nothing to prevent category 3 leaks.

The goal of this research is to demonstrate that the “better accuracy than human” filtering technology achieved by the machine-learning community in adaptive spam filters can be ported over to the confidential information firewalling problem.

To jump ahead, the answer to this is “yes, we succeeded, here's how to do it”. We achieved > 99.9% capture of emailed confidential information, while only false-alarming on less than 5.3% of the non-confidential emails, against a test set of approximately 14,000 emails.

In order to understand the solution, we need to take a review of a few terms in information retrieval as well as a very quick review of the last three hundred years of machine learning.

Information Retrieval Concepts: There are two important concepts that document classification people need to think about that information retrieval people have developed – these are “recall” and “precision”.

“Recall” was very simply what fraction of the documents that should have been fetched actually were fetched. “Precision” was what fraction of the documents that were fetched that actually should have been fetched. A perfect system would rate 100% recall, and 100% precision.

Just so you don't think this is a solved problem, NIST is still running the TREC conference to push the limits of accuracy on text retrieval, and although Google is very good for what the grand majority of people need (finding pizza shops, car mechanics, and product reviews), there are still some big problems in information retrieval. For example, consider trying to do information retrieval on genomic papers or legal document subpeonas such as the ENRON corpus; almost all of the keywords that Google would search on appear in each and every document. Pagerank (or citations/references) form hubs and spokes, not a $1/r^n$ scale-invariant web. On this kind of information retrieval problem at the TREC conference, any score over 50% recall and 50% precision is considered very good performance.

It is clear that one can trade off recall for precision. Consider that you return every document on any query at all; you will (by definition) detect every document that might contain any possibly relevant material, so your recall will be 100%. However, your precision will be no better than the proportion of relevant documents in your entire corpus. You might only return the single most likely document, which would make your precision close to 100%, but guarantee that for any topic that had two or more documents that you never score above 50% recall.

Deciding what the optimal tradeoff between high recall and high precision is often more of a political or psychological process than a technical one. One “reasonable” way to do it is to calculate the cost of each kind of mistake, and then minimize the total expected cost. Under reasonable assumptions, this occurs when the probability of each kind of error times the cost of that kind of error are equal across all possible errors.

For this work, our goal was to drive the recall as high as possible (to at least 99%) while maintaining a >90% precision. Against a 14,000 email test set, our final machine-learning design gives a 99.93% recall versus a 99.3% precision, and a false alarm rate of 5.21%. This translates back to a relative cost of 3:1 but that is an artifact; if we could have achieved 99.99% recall (that is, a 1000:1 relative cost) we probably would be reporting that value rather than 99.9% recall.

Prior attempts at this problem: An obvious way to create a confidential-information detector is to list all of the suspicious keywords we would expect to see in confidential information. These telltales, or “tells” would be highly topical according to the organization. Simple keywording is not adequate as word stemming (prefixes, suffixes, and irregular forms) need to be accommodated.

Before the project to produce a machine-learning filter for confidential information was started, we attempted to create a classic, regular-expression (regex) based filter. The individual regexes were human-tweaked by native Japanese speakers for best accuracy and in it's final version had 170 regexes. When we tested this regex-based filter against a database of 14,000 hand-classified emails, we found the following results:

- less than 90% alarm rate (actual “recall” of 89 %, meaning that 89% of the confidential information is stopped, which leaves 11% of the confidential information that would pass through the firewall unhindered)
- 96% “precision” (4% of alarms are false alarms)

This weak performance told us an important fact- human language meaning is difficult to capture in terms of keywords. It also told us that this was a difficult problem, even for well-motivated, funded projects with significant domain expertise and a significant body of test and demonstration data. With such a high “miss” rates (11%) we would not be able to give any significant level of confidence in security in corporate environments.

We will now take a medium-length detour into machine learning, and when we emerge, we'll talk about how to build and test a good text classifier. Although a deep understanding of all the moving parts is unnecessary, it's handy to at least have a vague knowledge of the mechanisms involved.

Machine Learning Filters: Machine learning filters, or “classifiers” (not to be confused with the governmental idea of “classified” information) have been around for hundreds of years, if you consider the work of Reverend Thomas Bayes, who probably went on to his just rewards in 1761. Indeed, Bayesian inference is the root of the field of machine learning.

The downside of basic Bayesian filtering is the assumption that every observation or “measure” must be independent for Bayes theorem to apply. This is immediately false in the case of a text classifier, as the inter-word dependency in human language is very large. Human language does not allow just any randomly-chosen word to follow any other word; rather, sentences have structure.

The fallacy of Bayes: Consider a basic dictionary of any human language- roughly 2000 words form the basic vocabulary and 10,000 words are the commonly understood vocabulary (this ignores stemming- the alteration of a root word to indicate time in the past, present, or future, to indicate plurals, possessives, and suchlike). If you allowed random and arbitrary word choices at each point in a sentence, each word position within a sentence would have roughly 10,000 possible choices; this would take about 13 bits/word to encode.

Compression techniques for plain ASCII text of randomized words show that it can be compressed to about 3 bits per character on the average (for example, /usr/dict/words, at 5 megabytes, compresses with gzip to about 1.5 megabytes, or 3.3 bits per character). This is about 2/3 of the maximum compression possible; meaningful and grammatically correct English text can be compressed to roughly 1 bit per letter.

This 1-bit-per-letter information rate tells us that the Bayes theorem assumption of word independence is being grossly violated; independent words shouldn't compress this well. Sure enough, if you actually build a text filter without some form of compensation for this assumption violation, you find the filter often putting out probabilities that are within one floating-point epsilon of 1.000000000 and 0.0000000000 exactly (if not exactly 1.0 and 0.0, then usually DBL_MIN and 1-DBL_MIN). Often, the result is those two values only and no values in between. Clearly it's not nice to violate the basic assumptions of Bayes theorem.

Given the huge impact of word interdependency in human text, how can Bayes rule possibly give anything close to valid results? Some Bayesian filters do it by restricting how many words are considered, for example, a certain popular spam filter uses only the 15 "most significant" words found in a text to decide whether it's spam or not. Because these significant words are usually not adjacent to each other in the text, this restores at least a vague conformance to the Bayes theorem's assumption of independence.

On the other hand, training these Bayesian filters is very simple- one must only keep track of the number of times each word is seen in confidential versus non-confidential texts. The ratio of occurrences is the probability observation, and the normalized product of the probability observations is the final result.

Fixing Bayes Rule with the Markov Manifold: Another way to avoid this horrible violation of Bayes theorem is to reject the individual words and consider only phrases of two words consecutively, or two words and three words consecutively, and so on. In fact, this works quite well with respect to accuracy (you can double a Bayesian spam filter's accuracy by doing this up to five words in sequence). The extreme probability value problem is mitigated but not completely eliminated by this. Interestingly, the full set of $2^4 = 16$ possible five-word sequences with the first word always included is suboptimal for accuracy; using only the endpoints (called "orthogonal sparse bigram" or OSB for short) is both faster and more accurate for spam filtering. Getting rid of the unigram (that is, the individual words, taken one at a time) also increases accuracy, paradoxically.

For example, the phrase "the quick brown fox" could be split into the eight phrases

the
the quick
the [skip] brown
the quick brown
the [skip] [skip] fox
the quick [skip] fox
the [skip] brown fox
the quick brown fox

but using just the following three phrases will usually give about twice the accuracy:

the quick
the [skip] brown
the [skip] [skip] fox

Those of you who know more math than is good for you will recognize this as constructing a Markov Random Field. Amazingly, this OSB method works, and even more amazingly, the rate at which various OSB phrase lengths contribute to accuracy in text classification correspond very well to the values of various phrase lengths contribute to learning real human language by real humans! Detmar Meurers and Tobin Mintz at OSU have done research in how humans learn languages, and found that the contribution of various “to be learned” phrase lengths for human students of both English and Chinese (and both neonatal and adult language learning) are very similar to the accuracy contribution of those same endpoint-only phrase lengths for machine-learning text classifiers.

Does that mean your brain is really constructing a Markov Random Field when it learns how to decide that one thing to say or write is OK, but another thing with the same words but in a different order is socially unacceptable? It's hard to say absolutely, but the evidence of Meurers and Mintz says that's the right way to bet. The implication is that we might be able to go one step further, and be able to categorize not just language semantics, but meaning, with a good enough Markov Random Field. The best experimentally-determined selection set of phrase lengths (including interior “don't care” placeholders) matches the Meurers & Mintz curves embarrassingly well.

One might conclude that the OSB variant of Markov classification makes the classifier nonlinear; this is correct but only in a very limited sense because the window of sensitivity of the nonlinearity (i.e. the ability to detect either “Mortgage” or “Viagra” but not both) is bounded by a short window of words (in our case, the window is six words wide.

Linearity in Classification: The Bayesian (and Markovian, both ordinary and OSB variation) classifiers common in spam filtering have the property of linearity – that is, whatever calculation they do can be reduced to mathematically finding which side of a straight line the unknown sample lies on. Training the classifier is the act of putting the dividing line in the right place. This “straight line” may well be a many-dimensional hyperplane, but it is a straight line (or flat plane) nonetheless.

This linearity is actually a very useful property, because it brings with it another feature- that concatenating two texts that are both on one side of the dividing hyperplane will never flip the result onto the other side of the hyperplane (thus “confidential” plus “confidential” can never become “not confidential”), and fractional mixtures (i.e. mostly one side, but with only a little bit of the other side) will lie along the line connecting the two documents.

During training, Bayesian and Markovian classifiers incrementally move the separating hyperplane as each new example is added to the times-seen-this-word (or phrase) database, and the probabilities are derived directly from the ratio of confidential versus non-confidential occurrences of each database entry seen. This learning method is slow, but sure; other methods are also known.

In Littlestone's Winnow algorithm (1988), learning is done by changing scoring weights by multiplication. Each possible word or phrase starts out with a score of 1.00 (words or phrases never seen before also get 1.00). To learn a new document, the document is broken up into words (or phrases) and whenever a word is present in a confidential document, the score for that word is multiplied by a small amount greater than one (1.33 works well), and when a word is present in a non-confidential document, the score for that word is multiplied by a number slightly less than one (0.8 works well). The resulting weighted sum is the conclusion, and if the sum is more than the number of words in the document, then the document is probably confidential, otherwise it is probably not confidential. Optionally, one may or may not allow the same score to be updated more than once for each document learned. An advantage of Winnow over straight Bayesian / Markovian is that it moves the hyperplane much more quickly which is an advantage when examples of one text variety or the other are rare in the training set.

In a Support Vector Machine (a.k.a. SVM, by Vapnik in 1963) the learning process is much more complicated; rather than going into the mathematics in depth we'll just state the end result is finding the dividing hyperplane that gives the biggest margin of error for any known example. In some sense, this maximizes the chances of correct answers, but even a single erroneous input will push the dividing hyperplane into a highly erroneous position. This is “Garbage In, Garbage Out” with a vengeance.

Newer SVM formulations allow some fraction of the known points to be “knowingly in error”, and give some robustness, but not a lot; the TREC 2008 conference test set of roughly 30,000 spam/nonspam texts contains a (probably unintentional) error around half-way through, where essentially the same text is attributed both as spam and again as nonspam. This single error knocks the SVM spam filters down from 99.99% accuracy to no better than 99.85%, and the SVM-based filters never recover this lost ground throughout the rest of the test set.

Limitations of Linear Classifiers: We can re-cast the weighted-sum or normalized-product description of a linear classifier into the same mathematical form as the description of an idealized neuron called a “perceptron” as described by McCullogh and Pitts in 1950. The good news is that this means a lot of very handy convergence theorems can be brought into play to optimize training.

Unfortunately, it also brings forth the Minsky-Papert Perceptron Theorem (from 1969) that shows a linear classifier (including not just McCullogh-Pitts neurons but also single-word Bayesian classifiers, Winnows, and SVMs) cannot compute something as simple as an XOR (try it- come up with weights for the words “Viagra” and “Mortgage” presence or absence that are over 1.00 when either one is present, but below 1.00 when both or neither are present.   But don't waste too much time on this, because it's been proven impossible).

To handle these harder problems (and we assumed that some of the confidential information issues would be exactly of this form- an XOR or worse). We expected small regions of confidentiality surrounded by a sea of innocuous documents, and so we applied more advanced methods.

One twist that SVM classifiers often use (but we didn't use in this work) is to preprocess the input data to transform a nonlinear problem into a linear one. This trick works, but you need to have some fairly good ideas about exactly what kind of transform will create this linear version of the classification problem. Since we can't read Japanese, we really couldn't even attempt this.

Nonlinear Classifiers: Unlike a linear classifier, nonlinear classifiers can't be reduced to a single straight line, even in many dimensions. Nonlinear classifiers can even have isolated regions, or islands of acceptance and rejection, inside larger areas of the opposite condition. For this reason, nonlinear classifiers are usually considered to be “more powerful” or “better” than linear classifiers, but at least for our task of classifying human texts into “confidential” versus “non-confidential”, the advantage of nonlinearity seems quite small.

K-nearest-neighbor classifiers: The ultimate nonlinear classifier is a very simple one; it's called the K-nearest-neighbor (KNN) classifier, first described by Fix & Hodges in 1951 (but they didn't publish it outside of the USAF). The principle of a KNN is simple – look at the K closest matches to the unknown; the unknown is most likely of the same type as the majority of it's nearest neighbors (to minimize risk of a voting tie, K is usually odd).   The only downside is that unless you have some very good sorting tricks, it's at least linear time to compare the unknown text to each and every known in your database.

Very importantly for KNN lovers, Cover & Hart published a theorem in 1967 that proved that a KNN will converge to never be worse than 2x the error rate of the best possible classifier given the same input data, as the number of known inputs goes toward infinity.

The particular KNN we used is called a “hyperspace” or “infinite radius” KNN. In this form, all known texts get to vote, but their votes are weighted by similarity versus dissimilarity, as (shared words)^2 / (unshared words)^2 .

This creates the same kind of rubber-sheet model of the dividing surface as you see in books on astronomy and physics (if you substitute “mass1” and “mass2” for “shared words”, “distance” for “unshared words”, and multiply by the universal gravitational constant G, then you have the equation for Newtonian gravitation). This physical analog happens to give a very simple, very fast, and very accurate text stream classifier. The only disadvantage is that because a KNN or hyperspace classifier requires comparison of the unknown data with each known that it might be close to, the storage required is linear in the number of texts learned.

Compressive (LZ77) classification: Another way to do classification is by measuring compression. Kendall Willets gave a first hint of this application in 2003 in an article in Dr. Dobbs Journal. Willets fed gzip (an LZ77 compressor) some known spam followed by an unknown message, and then known nonspam followed by the same unknown message. The result was that the spam and nonspam have about a 15% compression difference given spam and nonspam preambles- more than enough to differentiate the two.

GZIP (and the Lempel-Ziv LZ77 algorithm it's based on) are what are termed “universal” compressors- they build their compression dictionary directly from the signal being compressed. Essentially, the first time a string is seen, it represents itself. Subsequent occurrences of a string are replaced with back-references to the most recent previous occurrence (actually, the back-reference is how many characters backward in the output stream to go, and how many characters to copy).

However, gzip (and the LZ77 algorithm) have a finite look-back window, typically no more than 32Kbytes (thus allowing a 2-byte escape sequence to denote a back reference versus among uncompressed text). However, this limits the amount of example text available to no more than 32Kbytes as well, with a severe impact on accuracy.

The version of LZ77 we created has a 32-bit (4 gigabyte) limit on lookback. It also doesn't actually calculate the compressed version (we'd only throw it away), rather we only calculate the final compressed length. We've found that there's a significant increase in accuracy by making a higher compression for longer matching run-lengths as well; we see our best accuracy when the compression matcher's score is the sum of each solitary compression length raised to the 1.35 power (we determined that value experimentally).

We didn't come up with the idea of making longer perfect sequences score more; a similar method is used by the genetics research community to find gene matches across species. Geneticists use an algorithm called BLAST (Basic Local Alignment Search Tool) to find significant sections of identical DNA; and then add “bonus points” by extending the match forward and backward across short segments of non-identical DNA. The result is that one identical long segment scores more highly than two shorter segments that cover the same total length in both the CRM114 compressive classifier and the BLAST gene-matcher.

Bit-Entropy Classification: In 2005 Andrej Bratko exhibited a new kind of text classifier at NIST's TREC conference- an entropy classifier. The idea of this classifier was to form a full-scale model, one word at a time, of the texts, and then measured the entropy of the unknown, using that entropic model of the known texts (lower entropy implying better prediction and hence better match). The classifier worked extremely well, but consumed rapidly gigabytes of memory and required emptying the entire database and retraining using only the last part of the data periodically due to this memory consumption. The performance was repeated by Gordon Cormack at NIST TREC 2006, working one byte at a time as an entropic predictor.

The classifier we built and tested is even more minimalistic, working one <u>bit</u> at a time (and hence being able to classify bit-packed symbols as we were not sure if the texts would contain bit-packing in the future). The bit-based format also gives a much more compact representation; we need only about 64 megabytes to model each class of text we want to recognize. The interesting parts of this classifier are precisely how to create the predictive model efficiently; we will skip over that but note that well-commented source code is freely available. There is a method for computing the optimal model, but we don't use it because it's computationally expensive. Instead, we create an approximate (but very good) model on the fly.

Training Methods: TEFT (Train Every Thing), TOE (Train Only Errors), SSTTT (Single Sided Thick Threshold Training), TTE (Train Till Exhaustion), DSTTT (Double Sided Thick Threshold Training).

The next major choice that affects how well a confidential information detector works is the training method. This doesn't refer to the "inside math" of the classifier itself, but rather to whether the classifier is told to add a particular known text to its database or not. This can be very important for some classifiers, and completely unimportant for other classifiers. There are at least four training methods in common use for spam classifiers; we brashly assumed that at least one of them will work well for confidential information detection.

The four methods are often known by their acronyms: TEFT (Train Every Thing), TOE (Train Only Errors), SSTTT (Single Sided Thick Threshold Training) and DSTTT (Double Sided Thick Threshold Training). For our purposes, we'll combine SSTTT and DSTTT and speak only of TTT – Thick Threshold Training.

For example, a Bayesian or Markov classifier needs careful selection of training examples; if a Bayesian is trained on every incoming example, it drifts toward a configuration where most unknown texts are of the more common type. Consider this the AI equivalent to "If all you have is a hammer, everything looks like a nail". For this reason, Bayes and Markov classifiers should not be trained in TEFT mode.

Train Only Errors (or "TOE") does just what it says- we train only the example texts that are classified incorrectly. Usually this is done as a single pass through the known data and the accuracy in the final 10% of the pass is taken as the actual accuracy of the classifier. TOE is a good all-around training method, because it keeps the number of examples actually trained to a minimum, it converges rapidly (convergence in single pass training is guaranteed), and it usually gives very good results no matter what classifier is being used.

One can eke out another factor of two in accuracy by doing "thick threshold" training on a Bayesian or

Winnow system. In thick-threshold training, you train both errors and correct results if the result is not strong enough. For example, if a text is confidential, and the classifier reported a probability of confidentiality violation at 65%, the classifier got it “right”, and in TOE training nothing further would happen. In thick-threshold training (TTT for short) the threshold is usually set at something fairly strong, like 90% confidence or better, so a TTT system would train this confidential document even if TOE system wouldn't. This little bit more work is well worth the factor of two in accuracy.

A few classifiers support the concept of a “negative example”. For these classifiers, thick threshold training can be done both as a positive example of one class, and as a negative example of the other class (or classes). This is what makes “double sided thick threshold training” double-sided. If there's no such concept of a negative example, then it's single-sided thick threshold training.

At the other extreme, classifiers like SVMs work by looking for the most borderline examples and maximizing the margin of error on those examples only. This means that extra examples that are not borderline are simply **ignored**. The net result is that an SVM is usually trained TEFT (Train Every Thing) mode; it's one of the few classifiers that isn't actually hurt by TEFT because it ignores the extra examples in any case; an SVM's solution is determined solely by the examples with the smallest margins of error and extra examples have larger margins of error.

Another training method is TTE – Train Till Exhaustion. As the name implies, this is repeated training passes through the entire dataset until either perfect accuracy or total patience is exceeded. As a risk, TTE may never converge to perfect accuracy, and even if it does converge, it may *overtrain*. Overtraining is the phenomenon where an AI system learns individual cases as point solutions instead of making generalizations; the result is that accuracy on unseen examples falls drastically. We did not use TTE, however it has been reported to increase efficiency in spam filtering. Be wary, however, because multipass TTE is not guaranteed to converge.

The other hazard of TTE, overtraining, is most easily detected by *tenfold validation.* To do tenfold validation, the training set is split into ten roughly equal subsets. The first subset is held back, while the other nine subsets are trained, then finally the held-back subset is used as test data. Then the classifier is erased, subset number two is held back, subset one and subsets three through ten are trained, and then subset two is used as test data. This process is repeated ten times, so that each known example gets to be used as test data exactly once. If the test data part of each run is significantly less accurate than the final 10% of the training data, then the system is probably overtraining and your training methods are unsound; you should train over more known examples and fewer times (optimally, once only) on each example.

The Japanese text: One of the more interesting aspects of this project is that we worked on Japanese text, rather than English. Japanese is not “like” any western language in epigraphy or grammar. First of all, Japanese is not written in ASCII. Japanese uses the Latin alphabet (roughly 0 to 127 in ASCII) only for Western names, a few technical terms, metric units (i.e. volts, Hz, watts, etc) and as a form of extreme emphasis (the equivalent of a double-bold font).

The vast majority of the texts we were to classify are not written in ASCII. Instead, they are written in one of the three Japanese writing systems. Japanese uses a set of roughly 7000 Chinese-style “kanji” ideographic characters; each of these characters may have two possible single-word meanings (the “on” and the “kun” reading which are usually related ... but not always). The ideograms are often combined to make new words (for example, “Mitsubishi Electric” is written in four such ideographs).

Additionally, two separate polysyllabic alphabets known as “katakana” and “hiragana”; these alphabets encode syllables like “shi”, “tsu”, and “wa” as single characters. Katakana and hiragana cover the same space of sounds; thus the encoding is redundant between kanji, katakana, and hiragana, although the choice of which character set is used for any particular meaning is fairly strongly conventionalized. However, if a particular kanji is uncommon, an associated set of either katakana or hiragana will often be included (at least on the first usage) to give the pronunciation.

Once the words are known, the actual representation is just that- and nothing more. Unlike western languages, Japanese does not use whitespace to indicate breaks between words in any of the four epigraphies (latin, kanji, katakana, or hiragana). Punctuation (such as a period) is used between sentences, and a newline usually separates paragraphs, but other than that, written Japanese is a stream of mixed 8 and 16-bit characters without whitespace breaks. This poses an interesting problem for tokenization, as it's essentially impossible to know where one word ends and the next begins without deep knowledge of how the ideographs may be combined and without a full dictionary of stemmed Japanese words and knowledge of how the stemming interacts with grammar and common idioms.

In our work we settled for a simple tokenization- we defined the regex of tokenization to be **/./**- that is, a single byte at a time. This is an admitted weakness and we are considering ways to improve it. Note that the special case of the bit entropy classifier always works one bit at a time and so ignored the tokenization.

Testing Procedure: The Mitsubishi Electric ITC laboratory in Japan produced an initial test set of roughly 3000 examples of confidential and non-confidential documents, all written in Japanese, with the text in JIS encoding.   Attachments, if present, such as .DOC and .PDF were converted to Unicode, then the entire texts were converted to UTF-8 (a variable-width encoding). The UTF-8 texts were sent to MERL in Massachusetts and used for configuring classifier parameters and developing training methods. The resulting C code and CRM114 scripts were shipped back to Yokohama Japan, where ITC researchers tested the classifiers against a 14,000 email test set.

Test Results: For each of the candidate classifiers, we ran a 10-fold validation. Each classifier except SVM was trained single-sided with a very thin thick threshold (SSTTT training), and each known example was trained at most once, to avoid overtraining.

The following table shows our typical results for the 3000-document test set; results on the larger test sets are similar.

Classifier	**Markov**	**Bit Entropy**	**SVM**	**Hyperspace**	**Compressive**
Correct Alarms	360	358	355	355	346
Alarm Misses	8	10	13	13	22
Recall (% correct alarms)	0.9783	0.9728	0.9647	0.9647	0.9402

Although there was no classifier that met the requirement of 99% or better recall, we noticed that the classifiers often got entirely <u>different</u> errors; thus a voting system can produce a better classifier than any single classifier.

For example, the first confidential document that the Markov OSB classifier failed to detect was document #78; the compressive classifier also got document #78 incorrect. However, the SVM classifier, the Hyperspace classifier, and the bit entropy classifier all correctly detected the confidential content in document #78 and triggered the alarm. Conversely, the confidential document #166 is not detected by bit entropy, the SVM, the compressive classifier, or they Hyperspace classifier, but it is detected by the Markov classifier.

Here are typical mutual-error rates for confidential information; the number in each cell is the number of confidential documents that both classifiers failed to detect. Note that this is ONLY the "recall" side of the problem; as long as our precision stays above 90% (that is, less than a 10% false-alarm rate) we chose to ignore that part of the problem.

	Markov	**Bit Entropy**	**Compressive**	**SVM**	**Hyperspace**
Markov	*[8]*	4	5	3	3
Bit Entropy	4	*[10]*	4	4	3
Compressive	5	4	*[22]*	7	7
SVM	3	4	7	*[13]*	5
Hyperspace	3	3	7	5	*[13]*
Sum of Cross Errors	***15***	***15***	***18***	***19***	***23***

Note that a "3" in the above represents an actual alarm rate (or "recall") of 99.18%. However, for robustness, we preferred to optimize further, and to use more classifiers to further minimize the missed alarm rate and maximize our recall, at the intentional expense of a slightly lower precision and higher false-alarm rate .

Note also that if the classifier errors were truly independent, we would expect $(13 / 368)^2 = 0.5$ misses, or 99.86% recall from any two classifiers ORed together. Since we don't see this, we can conclude that the classifier errors are still correlated.

The classification system we developed has an interesting structure- it is a big OR gate. If any one classifier in the set detects classified information, the alarm is triggered and the email is considered to be classified unless a human overrides the alarm.

For the initial tests of the OR-gate system, we took the alarm output as the logical OR of all five classifiers listed (Markov OSB, LZ-77 Compressive, SVM, Bit Entropy, and Hyperspace.) Testing against the 3000-document test set, we achieved a single "miss" out of the 368 confidential documents for a recall of 99.72%.

We then selected just three classifiers (the Markov OSB, SVM, and bit entropy classifiers) for ORing, and tested this on a much larger (14,000 Japanese emails) test set that was mostly confidential, rather than the 3,000 document test set that is mostly non-confidential. One of the reasons for choosing this subset of classifiers is that Markov OSB, SVM and Bit Entropy all have classification time that is linear in the length of the input text and do not depend on the number or length of any texts previously learned.

On the other hand, FSCM and Hyperspace do not have this characteristic; Hyperspace requires time proportional to the length of the learned examples times the unknown input text, and FSCM takes time proportional to the inverse of the abundancy of the shared substrings in the learned text and the unknown input. Therefore, as learning continues on large data sets, FSCM and Hyperspace take geometrically more time to learn, while the other three classifiers continue at the same speed.

This analysis disregards time to learn; Markov OSB, Bit Entropy, and Hyperspace are all (nearly) constant-time learners. SVMs can learn in near-linear time per learn cycle, but that becomes $O(n^2)$ time given that a learn cycle must be run on every error – and errors accumulate. As with classification,, FSCM's time to learn is proportional to the inverse of the abundancy of shared substrings.

Using 10-fold validation on this large data set, we achieved two results were better than 99.5% recall (that is, less than 0.5% of the confidential information got through the filter without setting off an alarm) and 99.5% precision (of all alarms, only 0.5% were incorrect). This 14,000 element test set is about 85% confidential documents, so the false alarm rate on non-confidential documents was under 3.5% overall (note the difference between precision (which is calculated over all alarms) versus false alarm rate on non-confidential documents (which is calculated on the non-confidential documents only)

Here's the actual error rate for one 10-fold validation shuffle of the 14,000 email test set (14,247 valid files in UTF-8 Japanese):

Classifier	**Markov**	**Bit Entropy**	**SVM**
Correct Alarms	12414	12506	12525
Alarm Misses	141	49	30
Recall (% correct alarms)	0.9888	0.9961	0.9976
Correct non-alarms	1673	1634	1633
False Alarms	17	56	57
Precision (% correct alarms of all alarms)	0.9986	0.9955	0.9955
False Alarm rate	0.0101	0.0331	0.0337

As we might expect, the system with the highest precision (Markov) is also the one with the lowest recall, as it's typical that one can trade off precision and recall.

Similarly to the above small test-set, here's the cross-error rates for the 14,000 email test set:

	Markov	**Bit Entropy**	**SVM**
Markov	158	30	34
Bit Entropy	30	105	35
SVM	34	35	87
Sum of Cross Errors	***64***	***65***	***69***

Now we see an interesting situation: the classifier with the worst recall (Markov) also has the best cross-error rate, even against the best classifier (the SVM).

Combining all three filters (Markov OSB, Bit Entropy, and SVM) in a large OR-alarm (that is, trigger the confidential information alarm whenever any one of these three filters alarms) yields just 11 cases where all three alarms failed to trigger, for an overall precision of 99.93%, more than ten times better precision than required for product use. Of course, the recall gets worse (more false alarms) for a total of 95 false positives and a precision of 99.24% (false alarm rate of 95/1690 = 5.95% of genuine non-confidential documents alarmed). There were 95 total false positives rather than 17+56+57 = 130, because 35 of the false positives were double false positives and 10 alarms were triple false positives.

Classifier	**Markov**	**Bit Entropy**	**SVM**	**OR of all Three Classifiers**
Correct Alarms	12414	12506	12525	12544
Alarm Misses	141	49	30	11
Recall (% correct alarms)	0.9888	0.9961	0.9976	**0.9991**
Correct non-alarms	1673	1634	1633	1595
False Alarms	17	56	57	95
Precision (% correct alarms of all alarms)	0.9986	0.9955	0.9955	**0.9924**
False Alarm rate	0.0101	0.0331	0.0337	**0.0595**

Note that these results are one "run" - that is, one particular random shuffling of the 14,000 input files into the 10 groups for tenfold validation. We then tried two other random shufflings into the 10 groups, and arrived at very similar results:

Classifier	Shuffle #1	Shuffle #2	Shuffle #3	**Average Result**
Correct Alarms	12544	12548	12546	
Alarm Misses	11	7	9	
Recall (% correct alarms)	0.9991	0.9994	0.9993	**0.99928**
Correct non-alarms	1595	1604	1607	
False Alarms	95	86	83	
Precision (% correct alarms of all alarms)	0.9924	0.9931	0.9934	**0.99297**
False Alarm rate	0.0562	0.5088	0.4911	**0.05207**

This result establishes a reasonable claim of 99.93% recall, 99.29% precision for the task of confidential information detection.

This result also confirms Cormack's observation at the 2006 NIST TREC conference about combining tex-categorization filters. Cormack created a virtual spam filter that was the voting result of all of the spam filters submitted for testing at NIST. Each of the 27 spam filters submitted to 2006 NIST TREC got one vote, no matter how accurate or inaccurate that filter was found to be during testing. The resulting voting filter was roughly twice as accurate as any of the individual filters it was composed of. Interestingly, Cormack then tried a PCA analysis to optimize the 27 actual per-filter weights and found that this much more complicated system was no more accurate, on a statistical significance basis, than the one-filter-one-vote method.

There exists a trade-off between recall rate (capture rate for confidential information) versus precision (rate of false alarms). Since an email triggering the confidential-information alarm can be immediately sent back to the sender for either cancellation or override (rather than being dropped email), we can emphasize recall (very few missed alarms) over precision (false alarm rate) without being costly in time or human effort. The filters run very fast (about 10 to 20 milliseconds per classification on a server-class machine such as a Core 2 Duo) so the actual time delay is below the threshold of human perception and the filter result could appear essentially instantaneously when the user clicks on “send”.

The Black Swan Problem: Machine intelligence has some of the same limitations as human intelligence – that the machine cannot learn situations that are not shown by example in the training set (sometimes known as “black swans”, something that exists but that have at the present time no known examples, such as the actual species of black swans that were unknown before 1697). In the situation of confidential information, the “black swans” represent confidential information types for which no examples are available. For example, a new project might have zero examples, yet might be considered to be of very high confidentiality; until examples exist, the machine learning system cannot learn them. For another example, a project might be so secret that no documents can be made available for training purposes, but a particular set of keywords is guaranteed to exist that will successfully filter out that confidential information.

The particular keywords used in this situation are specific to the organization. The keywords could be extracted easily by those who do not know about the contents of the documents, and it works efficiently for newly generated (not learned) formats of documents.

For this reason, the final version of the confidential information detection system uses the regex keywords as an input to the OR along with the machine learning classifiers, thus avoiding the Black Swan problem. Additionally, these keyword examples can inform the user just **why** a particular document should be considered classified information, by pointing to a single keyword in a single paragraph.

Adding the additional regex keyword filtering feature has nonzero cost. Testing against the 14,000 document test set, the Markov OSB + SVM + Bit Entropy + Regex filter combination keeps the 99.9% recall, but precision drops from 99.3% to 98.8%, and the false positive rate goes from 5.95% up to 9.41%

Searle's Chinese Room: The question of analyzing text in a language that you don't understand has been an interesting point for machine learning and artificial intelligence for many years. In 1980, John Searle proposed a fairly devious thought experiment along the following lines: You are locked in a room with a slot in the door. Bookshelves cover the walls. Each book contains instructions, one per page, that tell you to look at one or more pieces of paper, and depending on exact match of the squiggles and patterns you see, you are to write (or erase) a note to yourself, write a permanent squiggle pattern on another piece of paper, and perhaps push that piece of paper out through the slot. Then every instruction tells you the book number and page number to go to next. Then a piece of paper drops into the slot on the wall, and you start at book one, page one, and start performing the instructions, making squiggle patterns and notes to yourself. What you don't know is that the squiggle patterns are actually messages written in Chinese, and the people dropping these messages in your slot think they're conversing with a person who knows Chinese who's locked inside a little room trying to convince them that they are a real person who wants to get out of the room. So- assuming you do your job and the people outside the room let you out- then did you learn to speak Chinese? Or does the room itself somehow know how to speak Chinese?

Essentially, we've now completed the first step in Searle's thought experiment. Unlike the authors of the regex-based matcher, none of the programmers who created these text classifiers can read Japanese. The classifiers themselves are written in ANSI C and contain no keywords or any other Japanese-specific information whatsoever. The training protocol is entirely mechanistic. The mathematical definition of all of the final classifiers (although perhaps not all of the learning algorithms) is understandable by anyone who can pass high school algebra. Yet, the resulting classifiers themselves are comparable in performance to the very best humans at the task of deciding whether a particular Japanese email is confidential or not, based solely on learned examples.

Dangers and Risks: Although the classifier system we have developed here is superior to any single human, there are dangers and risks associated with it.

First of all, this confidential information detector is a single strong gateway in a very low and mostly undefended wall. There is no protection against sending things like confidential or embarrassing .GIF or .JPG images, for PDF documents that are encoded as images rather than as text, or for texts that are in any way encrypted (even ROT-13 will defeat this text classifier unless some of the training documents are also ROT-13 encoded).

Second, this confidential information detector provides no protection associated with any channel except outgoing email. Physical access to a computer holding your unencrypted confidential data implies unimpeded access to that confidential data, and a confidential information detector does not change that sad fact.

The mere act of preparing to train a confidential information detector creates an additional risk: the training data is your organization's “crown jewels” - it is literally the organization's own opinion of the organization's most protected intellectual property, and a successful theft of the training data exposes <u>all</u> of your organizations IP value, patents, research (including work that may be subject to ITAR controls) contacts, business plans, and even information held under NDA from other companies.

Organizations that want to deploy this style of automatic outbound filtering should be advised that obtaining good training data (that is, all of your organizations most tightly held secrets) can be problematic, and you may find some justifiable resistance even if the mandate for the system comes from the highest levels of management's chain of command (and <u>not</u> merely from the IT department). Should the training data be leaked, the consequences may be dire (court proceedings are a possibility), and

The classifier training data itself can be reverse-engineered (by an offline dictionary attack, if nothing else) to extract text snippets of high value. The following theorem by Yerazunis and Raj show the futility of dataset obfuscation to hide topics of interest; a dictionary-based phrase attack will yield sufficient information to create a KNN classifier that can be reverse-engineered to reveal your organizations most valued topics.

Theorem 1: Classifier Obfuscation is Futile.

1. Assume the trained classifier is available to run, but it's interior state and data is perfectly encrypted (i.e. we assume any attacks on the encryption of the database or the algorithm itself will be 100% futile).
2. Assume you have the unlimited use of this trained and encrypted classifier.
3. Classify a large number (n approaching infinity) of randomly generated texts.
4. Train a KNN classifier with the resulting classification on these randomly generated texts.
5. By the Cover & Hart KNN theorem, the KNN will approach to within 2ε of the optimal classifier's performance.
6. Extract the KNN via a dictionary attack (or just TOE training on the KNN, and keep a record of what randomly generated texts were tagged as confidential data.
7. You now have a list of high-value keywords and fragments of text of the obfuscated high-value data.

This risk is not merely theoretical. In 2004 John Graham-Cumming demonstrated this attack successfully against a well-trained copy of SpamBayes, successfully obtaining a set of strings that could gateway spam through a SpamBayes anti-spam firewall.

An even more targeted attack can be created by grabbing texts from the web that mention topics related to the target organization and it's competitors in some way, such as product areas, markets, trademarks, press releases, patent filings, job postings, organizational charts, employee lists, etc. These texts can be converted to snippets and either used straight or "fuzzed" with each other or with a bank of acronyms to automatically generate a large volume of highly pertinent information. The resulting high-value "confidential" snippets will profile whatever the targeted organization considers to be high-value intellectual property assets. Fuzzing the highest-scoring snippets can lead to yet longer snippets; the time required for finding the single most probable stream is merely linear with a bad coefficient.

Conclusions and Availability:

We conclude that automatic classification of outgoing email is not only technologically feasible, but that the machine-learning approach is superior to human-mediated email classification. Not only is it orders of magnitude faster, but (assuming the humans are roughly as smart as the regexes the humans wrote) the machine-learning classifiers are roughly 100 times better recall in minimizing accidental leaking of confidential information, as well as having a false alarm rate of roughly half that of humans.

As of December 2009, Mitsubishi Electric Information Technology Corporation began offering this filtering capability in a retrospective mode in the LogAuditor Mail Saver product in Japan.

For those of you who like to roll your own customized solutions, the core software needed (either the CRM114 standalone language and the ANSI-C callable library) is available for free download from Sourceforge. The standalone language version is licensed under the GPL, and the C-callable library is licensed under the LGPL; other licenses can be negotiated. As this is open source, if you want to contribute, please do (take note, those of you who like to create Python bindings or JNI). Note that the standalone language classifiers are slightly older versions at this point; we are backporting the new C library back into the language as time and workload permits.