# CS351 Spring 2004, Project 1
# The Bayesian Spam Filter

MondoSoft, Inc.

January 26, 2004

**STATUS** Specification and requirements document

**VERSION** 1.0

**DATE** Jan 26, 2004

## 0 Changelog

**Version 1.0** Initial release. Jan 26, 2004.

## 1 Summary

In response to rising complaints about Unsolicited Bulk Email (UBE, also known as "spam"), MondoSoft Inc.[1] has decided to market a spam filter program, `SpamBGon` (TM), based on Bayesian statistical decision-making technology. To ensure interoperability of this filter program with other software, it must be capable of interfacing with standard UNIX mail program tools such as `procmail (1)`. Furthermore, MondoSoft has recognized the business opportunity to remarket a sub-component of this program as a high-performance standalone software module. Therefore, this project also includes `MondoHashTable`, a fully functional hashtable implementation of the `java.util.Map` interface. To improve marketability and demonstrate these programs' superior performances, both components (the hashtable and the full `SpamBGon` suite) will also be accompanied by rigorous scientific empirical performance evaluations[2].

## 2 Definitions

**ANALYZABLE** The sections of an email message subject to tokenization and statistical analysis. The ANALYZABLE section consists of the BODY, plus the FIELD-BODY of the HEADERS "From", "To", and "Subject".

[1] Your employer.

[2] Unfortunately, because MondoSoft's VCs pulled out during the bubble burst, MondoSoft is a bit strapped for cash to cover independent laboratory certification, so you get stuck with this job too.

**BODY** The main body of an email message; the part of an email message outside the HEADERs. Refer to RFC822 for details.

**CLASSIFICATION** The process of using statistics accumulated during TRAINING to label an UNLABELED message as SPAM or NORMAL.

**EVIL** See SPAM.

**FEATURE** Some measurable characteristic of an email message, such as the presence or absence of a specific word, the length of a line, etc.

**FIELD-BODY** The contents of a HEADER, following the FIELD-NAME. Refer to RFC822 for details.

**FIELD-NAME** The sequence of characters that identifies a field (i.e., specific HEADER) within the HEADERS section of an email message.

**HEADER** An email header or field, specifing information such as routing, date and time, subject, etc. Refer to RFC822 for details.

**MAILBOX** A folder format for storing multiple email messages that is widely used under Unix (e.g., by `mail`, `mailx`, `pine`, `mutt`, etc.). A MAILBOX consists of zero or more email messages concatenated together, separated by (single) blank lines. Each new email message is recognized by the presence of the token "From " at the beginning of a line. No other structure or control information is imposed on the file.

**MAY** A requirement that the product can choose to implement if desired. Can also indicate a choice among acceptable alternatives (e.g., "The program MAY do x, y, or z." indicates that the choice of behavior x, y, or z is up to the designer.)

**MUST** A requirement that the product must implement for full credit.

**MUST NOT** A behavior or assumption that must not be violated. Violating a MUST NOT restriction will result in a penalty on the assignment.

**NORMAL** Email that the USER wishes to receive.

**PRIOR** Or prior frequency estimate. The expected frequency of SPAM or NORMAL emails after TRAINING but *before* looking at a specific email message. Represents the proportion of SPAM and NORMAL email messages in the training data. Equivalent to the terms $\Pr[C_S]$ and $\Pr[C_N]$.

**POSTERIOR** Or posterior frequency estimate. The conditional probability estimate of a specific message being SPAM or NORMAL *after* analyzing the contents of the message. Equivalent to the terms $\Pr[C_S|\mathbf{X}]$ and $\Pr[C_N|\mathbf{X}]$.

**PUNCTUATION** Punctuation characters. For the purposes of this project, the punctuation characters are considered to be any characters other than WHITESPACE, letters, or digits.

**RECOVERABLE ERROR** An error condition that the software can ignore, correct, or otherwise recover from. The program MUST produce a warning message and then cleanly continue with no corruption or loss of valid data.

**RFC822** Document that specifies the syntax of standard internet email messages. Refer to this document for all specifications related to the format of email messages. Available at `http://www.ietf.org/rfc/rfc0822.txt`.

**SPAM** Email that the USER does not wish to receive.

**TRAINING** Mode or stage in which the software compiles statistics from data that is known to be SPAM or NORMAL.

**UNLABELED** An email message whose content (SPAM or NORMAL) is unknown.

**UNRECOVERABLE ERROR** An error condition from which recovery is impossible. The program MUST produce an error message describing the condition and then cleanly halt.

**USER** A single computer user or email recipient (potentially a mailing list).

**WHITESPACE** Non-printable characters including (but not limited to) space, horizontal and vertical tabs, newlines, and carriage returns. C.f., the Java JDK API call `Character.isWhitespace()`

# 3 Requirements

This section describes the elements that MUST be developed as part of this project. The designer MAY also choose to implement additional Java source files, programs, and/or shell scripts in support of the following items. This section only describes the general performance requirements for each element; for specific deliverable requirements, please refer to Section 5.

## 3.1 Hashtable

The fundamental unit of statistical analysis for the Bayesian spam filter is the statistic for an individual token, which boils down to counts of the number of occurrences of each distinct token seen in the TRAINING data (see Appendix A for details). To track the mapping between tokens and their counts, the `SpamBGon` suite will use a hash mapping, specifically, a from-scratch implementation of the `java.util.Map` interface, as documented in the Java 1.4.1 API specification. This module will be named `MondoHashTable.java` and MUST support the complete `java.util.Map` interface and contract specification. The `MondoHashTable` implementation MUST NOT use, access, refer to, or rely on the `AbstractMap` or any other implementation of the `Map` interface. The `MondoHashTable` implementation MAY employ the `java.util.AbstractSet` implementation to support the `Map.keySet()` and/or `Map.values()` operations.

As part of the project deliverables, the developer MUST demonstrate the performance of the `MondoHashTable` and show that it meets the quantitative requirements given in Section 4. To do so, it will probably be necessary to provide additional data members, methods, or subclasses

to track quantities such as number of allocations and reallocations, number of accesses, wall clock time, etc. The choice of which data/methods/subclasses to provide is up to the developer, but all such entities MUST be documented in the API documentation (c.f., Section 5.1).

The `MondoHashTable` will form the core of the first milestone submission; refer to Section 5.1 for details on the full submission requirements.

## 3.2 Tokenizers

The job of a tokenizer is to split the ANALYZABLE sections of an email message into small chunks, called tokens, which are the basic units of analysis. Many different types of tokens are possible—individual words, words plus punctuation characters, short strings of characters, HTML entities, MIME attachments, etc. A program can obtain different streams of tokens and, thus, different statistics from the same email message by changing the definition of a token (i.e., by changing the tokenizer module that turns an email message into tokens).

The `SpamBGon` software suite MUST provide at least three tokenizers:

**`NGramTokenizer`** This tokenizer splits the ANALYZABLE section of an email message into tokens of $n$ contiguous characters, where $n$ is a parameter to the tokenizer. This tokenizer MUST be able to omit WHITESPACE and PUNCTUATION characters. It MAY also provide user-selectable functionality to include WHITESPACE and/or PUNCTUATION characters.

**`WhiteSpaceTokenizer`** This tokenizer splits the ANALYZABLE section of an email message at WHITESPACE characters. Essentially, the goal of this tokenizer is to split out individual "words". This tokenizer MUST discard WHITESPACE and PUNCTUATION characters, though it MAY also provide a user-selectable option to preserve PUNCTUATION characters.

**One other tokenizer** Choice of this tokenizer is a design decision, but it MUST be documented, described, motivated (i.e., a rationale given for why it might be a useful tokenizer), and empirically tested. Possibilities for this tokenizer include (but are not limited to) a recognizer for dates and times (from the `Date` header), a tokenizer that recognizes HTML tags and their contents, a tokenizer that recognizes MIME messages as single entities, a tokenizer based on n-grams of words (rather than characters), etc.

These tokenizers MUST be interchangable; the USER MUST be able to select among the tokenizers at the command line during TRAINING and CLASSIFICATION. When a tokenizer requires additional parameters (e.g., the parameter $n$ for the `NGramTokenizer`), the program MUST provide a command-line interface to set such parameters.

## 3.3 Spam Filter

The Bayesian Spam Filter program suite MUST provide at least two client programs, `BSFTrain` and `BSFTest`. The suite MAY also provide additional programs for additional functionality, at the designer's option. Each program's interface is described below, followed by common options that MUST be supported by both programs.

**`BSFTrain`** This client is responsible for analyzing known examples of NORMAL and SPAM emails, building the naïve Bayes statistical models for each, and saving a durable copy of those models. This program also provides an interface to display a summary of the contents of the two naïve Bayes models.

This program MUST accept email inputs in the format specified by RFC822 as training input. It MUST also accept MAILBOX format files as training input, and recognize the individual messages that occur within the file as separate email entities. It MUST correctly handle zero-length input files.

This program MAY accept its training data files from standard input, but MAY provide direct file access to them (via an appropriately chosen command-line option). It MUST support some form of durable storage for the statistical models. The developer MAY use the Java serialization mechanism to implement this durable storage. `BSFTrain` MAY save its statistics in a single, combined file or in two separate files. This program MUST be capable of loading previously produced statistics files, updating them with new data, and saving the new updated files. This program MUST NOT lose data from previous TRAINING sessions during this operation—it MUST only add new data to existing.

`BSFTrain` MUST maintain two statistical models—one for SPAM and one for NORMAL email. When this program is initially run (i.e., if no statistics files exist at first), it MUST construct new, default statistics tables. The designer MAY choose to require a separate, "initialization" invocation to create and initialize the statistics files before TRAINING, or MAY choose to have that operation happen transparently.

This program MUST be capable of producing a human-readable dump of the current (possibly default) statistical models. The designer MAY choose any reasonable format for the dump, but the dump MUST, at the minimum, provide information about the relative frequency of all known tokens under both SPAM and NORMAL classes, the class PRIORs, and the total number of tokens read.

The `BSFTrain` client MUST support the following command-line options:

**`-s`** Treat input data as SPAM.

**`-n`** Treat input data as NORMAL.

**`-t`** Run in TRAINING mode. Compiles input data into statistics and updates the existing statistics tables (if any) with the new data.

**`-d`** Run in dump statistics mode. Produce a summary report that gives the statistics for both SPAM and NORMAL email. This report MAY be printed to standard output or MAY be written to a log file. If `BSFTrain` supports writing the dump to a log file, it MUST provide a command-line option for choosing that output file.

This program MUST also support the options listed below under **Common Options**.

`BSFTrain` MAY implement additional command-line options of the designer's choice, but such options MUST NOT conflict with the options given above or the options listed below under **Common Options**.

**`BSFTest`** This client is responsible for analyzing a single, UNLABELED email message and labeling it as SPAM or NORMAL.

This program MUST read its email message from standard input and write its (normal) output to standard output. Its assessment of the label for the email (either SPAM or NORMAL) MUST be written as an "`X-Spam-Status:`" HEADER, including the token `SPAM` or `NORMAL` and the likelihoods for each class. For example,

```
X-Spam-Status: NORMAL, ll(SPAM)=-2478.34, ll(NORMAL)=-1893.28
```

or

```
X-Spam-Status: SPAM, ll(SPAM)=-3789.6, ll(NORMAL)=-4279.62
```

`BSFTest` MAY also emit additional "`X-`" headers of the designer's choice. All such headers MUST be fully documented in the user documentation.

This program MUST accept the options listed below under **Common Options**. It MAY also accept other options that do not conflict with those, at the designer's option.

If `BSFTest` is invoked with no previously trained model (i.e., no statistics file(s) generated by `BSFTrain`), it MAY treat this as a RECOVERABLE or UNRECOVERABLE error, or it MAY silently initialize the appropriate statistics internally. If it chooses to initialize the statistics, it MAY write them out to file analysis or it MAY discard them.

**Common Options** Both client programs MUST support the following command-line options:

**`-m modelFileName`** Load/save SPAM/NORMAL statistical models from/to the specified filename. If `BSFTrain` saves all statistical models to a single file, the filename should be `modelFileName.stat`; if it uses separate files for SPAM and NORMAL statistics, their respective file names should be `modelFileName.sstat` and `modelFileName.nstat`.

**`-k tokenizerName`** This selects the type of tokenizer to be used in the current analysis. The program MAY also require additional command-line arguments specific to each tokenizer (e.g., the `NGramTokenizer` requires that the parameter $n$ be set). Failyure to provide a necessary tokenizer-specific option MAY be treated as a RECOVERABLE or UNRECOVERABLE ERROR.

If a program detects a discrepency between the type of tokenizer specified by `-k` and the type of tokenizer previously used to construct the saved statistics tables (if any), it MAY choose to interpret this as a UNRECOVERABLE or (if possible) a RECOVERABLE ERROR. It MUST NOT crash or produce incorrect results because of this condition. It MUST NOT destroy, corrupt, or overwrite previously existing saved statistics file(s).

# 4 Quantitative Requirements

This section describes the performance and IP requirements for the `SpamBGon` software suite.

- All programs MUST NOT crash, core dump, dump a stack trace, or throw an exception on any input.

- In the case of a RECOVERABLE ERROR, a program MUST issue a warning statement and continue processing. The program MAY choose to issue the warning statement to standard error or to a log file. If the warning is issued to a log file, the log file name and location MUST be a user-specifiable parameter to the program.
- In the case of an UNRECOVERABLE ERROR, a program MUST issue an error statement and terminate with a non-zero error condition. The program MAY use different exit codes to indicate different error conditions, but such codes MUST be documented in the user manual. The error message MUST be logged to the same destination that warning messages (from RECOVERABLE ERRORS) are.
- In the case of any ERROR, a program MUST NOT delete, corrupt, or damage existing statistics model files or any other "stateful" files employed by the program suite.
- The `MondoHashTable.java` module MUST NOT use or reference the `Hashtable`, `HashMap`, `AbstractMap`, `HashSet`, `TreeSet`, *or any of their subclasses*.
- For (substantially) reduced credit, `BFSTrain` and `BFSTest` MAY use the `HashMap` class in place of `MondoHashTable`. Note that this requirement exists only as an aid in case the programmer has difficulty getting `MondoHashTable` to work properly; for full credit the entire `SpamBGon` suite MUST employ `MondoHashTable` and MUST NOT employ or refer to any of the classes listed in the previous bullet point.
- The entire program suite MUST NOT employ or refer to the `StreamTokenizer` class.
- The programs MAY provide additional output for debugging purposes, *but* such output must be *disabled by default*. Any program MAY provide a command-line switch to enable debugging support when desired.
- The `SpamBGon` suite MAY use the `gnu.getopt.Getopt` and `gnu.getopt.LongOpt` classes to assist in handling command-line options.
- The programmer MAY ask permission of the instructor or the TA to use any classes outside the JDK that have not already been mentioned. The final programs MUST NOT use any class outside the JDK that have not been explicitly allowed.
- The `SpamBGon` suite MAY assume that all valid input is standard ASCII text in the range `(char)0`–`(char)127`, inclusive. If a program encounters a character outside this range, it MAY treat it it as a RECOVERABLE or UNRECOVERABLE ERROR or silently ignore it. If such characters are treated as RECOVERABLE or ignored, they MUST NOT disrupt the otherwise normal functioning of the program.
- All programs MUST NOT assume that all input is validly structured email. If a program encounter non-email input (e.g., lacking or corrupted HEADERS, invalid character sets, improper MIME boundaries, etc.) it MAY produce a RECOVERABLE or UNRECOVERABLE ERROR, but it MUST NOT crash, corrupt the statistics files, etc. If a program chooses to RECOVER from an ill-formed email, it MUST NOT corrupt the statistics tables with information from the illegal input; it MUST wait for the next valid input before continuing to update statistics tables.

- Both `BFSTrain` and `BFSTest` programs MUST run in amortized $O(n)$ time for email input of size $n$.
- The `MondoHashTable` MUST support `get()`, `put()`, `remove()`, `size()`, and `isEmpty()` in amortized $O(1)$ time. The table MAY support key/value iteration in time proportional to the *capacity* of the table. For extra credit, it MAY support key/value iteration in time proportional to the number of keys/values (respectively). To receieve the extra credit, the designer must demonstrate this convincingly in the performance documentation.
- The `MondoHashTable` MUST NOT consume more than $O(n \cdot s)$ memory for $n$ distinct keys, where $s$ represents the combined size of a key/value pair.
- The `MondoHashTable` MUST support the `keySet()` and `values()` operations with only $O(1)$ space above that required by the hashtable itself. Specifically, these operations MUST NOT replicate the underlying hashtable, nor duplicate any keys or values.
- All user documentation MUST be grammatically correct and include correct spelling and usage. Notably, "Bayes" was a real person so all terminology including his name must be capitalized. E.g., "Bayesian spam analysis", "naïve Bayes", etc.
- The programmer MUST document any areas in which her or his software suite does not meet this specification. *WARNING!* The grade penalty will be higher if the instructors discover an undocumented program shortcoming or bug than if it is documented up front.

# 5 Deliverables

This section describes the content to be delivered at each stage of the project (one milestone and a final rollout). For the deadlines of these stages, please refer to Section 6.

## 5.1 Milestone 1: `MondoHashTable`

The first project component due is the `MondoHashTable` implementation. The deliverables for this milestone are:

**`MondoHashTable.java`** The main class file for the `MondoHashTable` implementation.

**Other Java source files** Any other supporting code files necessary to compile, load, and use the `MondoHashTable` module.

**API documentation** The handin MUST also include the full, compiled JavaDoc documentation for the `MondoHashTable` implementation. This documentation MUST include full descriptions of every public or protected method, field, sub-class, enclosed class, or constructor employed by `MondoHashTable`. This documentation hierarchy MUST be included in a sub-directory named `documentation/` within the submission tarball package.

**Performance documentation** The handin submission MUST include a document describing the performance of the hashtable implementation and demonstrating (via empirical experiments) that it meets the quantitative performance goals established in Section 4 of this document. The designer MAY choose any tests that he or she desires to establish the performance of his/her `MondoHashTable`, but MUST describe all tests and why they lead to the stated conclusions about performance. This document MUST be named `PERFORMANCE.extension`, but it MAY be a plain text, HTML, PDF, or PostScript document (with the appropriate `extension`). It MUST NOT be a Microsoft Word or other nonportable format document.

**Test cases** The submission tarball MUST include a subdirectory named `tests/` that includes all of the test data used to demonstrate the performance of the hashtable implementation.

**CVS log file(s)** For each Java source file, the submission tarball MUST include a corresponding `.log` file including the CVS log for that sourcecode. E.g., for the file `MondoHashTable.java`, the following CVS command will produce the appropriate log output:

```
cvs log MondoHashTable.java > MondoHashTable.log
```

At the programmer's option, this submission MAY also include:

**`BUGS.TXT`** This file documents any known outstanding bugs, missing features, peformance problems, or failures to meet specifications of your submission. Note that the penalty for such problems will be smaller if they're fully documented here than if the instructors discover them independently.

The submission directory MUST be named `lastname_p1m1` and the submission tarball MUST be named `lastname_p1m1.tar.gz`.

## 5.2 Rollout: The `SpamBGon` Suite

The second and final stage of the project is the rollout of the completed project. The deliverables for this stage are:

**`BSFTrain.java` and `BSFTest.java`** The two primary programs of the `SpamBGon` suite.

**Other Java source files** Any other supporting code files necessary to compile, load, and use the `BSFTrain` and `BSFTest` programs. *Note:* if these programs depend on external library code other than the Java JDK or the `gnu.getopt` suite, the submission tarball MUST either include the library whole or provide easy and explicit instructions on how and where to access such libraries. This documentation MUST be provided in the `README.TXT` file. The designer is responsible for ensuring that all copyright and distribution conditions are adhered to.

**`README.TXT`** This file MUST describe how to compile, configure, and install the `SpamBGon` suite. It MUST also list any dependencies on additional software support libraries.

**Internal documentation** The handin MUST also include the full, compiled JavaDoc documentation for all Java source files in the submission tarball. This documentation MUST include full descriptions of every public or protected method, field, sub-class, enclosed class, or constructor employed by the code. This documentation hierarchy MUST be included in a sub-directory named `documentation/` within the submission tarball package.

**User documentation** The handin submission MUST include complete user-level documentation for the `SpamBGon` suite. This documentation MUST include instructions on how to use both `BSFTrain` and `BSFTest` including the functionality of all command-line options. The documentation MUST also describe the function and use of any additional programs included in the submission. User documentation MUST include information on the expected inputs and outputs of all programs, how to read and interpret the output, and information on all status and error messages that the programs could produce. This documentation MUST also include at least one example of how to run each program and how to interpret the output. This document MUST be named `USERDOC.extension`, but it MAY be be a plain text, HTML, PDF, or PostScript document (with the appropriate `extension`). It MUST NOT be a Microsoft Word or other nonportable format document.

**Performance documentation** The handin submission MUST include a document describing the performance of the `SpamBGon` suite, including its ability to differentiate SPAM from NORMAL email under different amounts of TRAINING data and under different tokenizers (including a small range of reasonable parameters for each parameterized tokenizer). This document MUST also include the designer's assessment of which tokenizer is superior and why or, if different tokenizers are superior under different conditions, what conditions are important to the success of each. The designer MAY choose any tests that she or he desires to establish the performance of her/his `SpamBGon` suite, but MUST describe all tests and why they lead to the stated conclusions about performance. Finally, this document MUST include the designer's assessment of how to improve the performance of the system (e.g., what other kind of tokenizer might be helpful, how to change the probability equations to improve accuracy, etc.) This document MUST be named `PERFORMANCE.extension`, but it MAY be a plain text, HTML, PDF, or PostScript document (with the appropriate `extension`). It MUST NOT be a Microsoft Word or other nonportable format document.

**Test cases** The submission tarball MUST include a subdirectory named `tests/` that includes all of the test data used to demonstrate the performance of the `SpamBGon` suite.

**CVS log file(s)** For each Java source file, the submission tarball MUST include a corresponding `.log` file including the CVS log for that sourcecode.

At the programmer's option, this submission MAY also include:

**`BUGS.TXT`** This file documents any known outstanding bugs, missing features, peformance problems, or failures to meet specifications of your submission. Note that the penalty for such problems will be smaller if they're fully documented here than if the instructors discover them independently.

Note that if the `MondoHashTable` code is not fully functional for Milestone 1, a revised version MAY be submitted in this handin. If `MondoHashTable` has been revised for this version, this submission tarball MUST include the necessary supporting documentation described under Milestone 1, as well as notes describing the added functionality/improvements between Milestone 1 and this handin.

# 6 Timeline

**Jan 26** Project specification handed out.

**Feb 6, 5:00 PM** `MondoHashTable` component due.

**Feb 20, 5:00 PM** Full project due.

# Appendix A: Bayesian Spam Analysis

The spam classification method you will be using is based on a Bayesian statistical model known as the "naïve Bayes" model. It's based on estimating the probabilities that a given UNLABELED message is either SPAM or NORMAL. More specifically, for an UNLABELED message, $\mathbf{X}$, you must evaluate the quantities $\Pr[C_N|\mathbf{X}]$ and $\Pr[C_S|\mathbf{X}]$, where $C_N$ denotes the class of NORMAL email messages and $C_S$ is the class of SPAM email messages. If you find that

$$\Pr[C_N|\mathbf{X}] > \Pr[C_S|\mathbf{X}] \tag{1}$$

, you can label the message $\mathbf{X}$ NORMAL, otherwise you label it SPAM.

The trick is finding $\Pr[C_i|\mathbf{X}]$. Your program (specifically, `BSFTrain`) will estimate them by looking at a great many NORMAL and SPAM emails, called TRAINING DATA. The problem is that, even given a bunch of example emails, it's not immediately obvious what this conditional probability might be. Through the magic of Bayes' rule, however, we can turn this around:

$$\Pr[C_i|\mathbf{X}] = \frac{\Pr[\mathbf{X}|C_i]\Pr[C_i]}{\Pr[\mathbf{X}]} \tag{2}$$

The quantities $\Pr[C_N|\mathbf{X}]$ and $\Pr[C_S|\mathbf{X}]$ are called ***posterior probability estimates***—posterior because they're the probabilities you assign *after* you see the data (i.e., after you get to look at $\mathbf{X}$). The quantity $\Pr[C_i]$ is called the ***prior probability of class*** $i$, or simply the prior. This is the probability you would assign to a particular message being SPAM or NORMAL *before* you look at the contents of the message. The quantity $\Pr[\mathbf{X}]$ is the "raw" data likelihood. Essentially, it's the probability of a particular message occuring, across both SPAM and NONSPAM. The quantity $\Pr[\mathbf{X}|C_i]$ is called the ***generative model*** for $\mathbf{X}$ given class $i$.

It seems like we've taken a step backward. We now have three quantities to calculate rather than one. Fortunately, in this case, these three are simpler than the original one. First off, if all we want to do is *classify* the data, via Equation 1, then we can discard the raw data likelihood, $\Pr[\mathbf{X}]$ (to see this, plug Equation 2 into 1). Second, the term $\Pr[C_i]$ is easy—it's just the relative

probability of a message being SPAM or NORMAL, i.e., the frequency of SPAM or NORMAL emails you've seen:

$$\begin{aligned}\Pr[C_N] &= \frac{\text{\# NORMAL emails}}{\text{total \# emails}} \\ &= \frac{\text{\# NORMAL emails}}{\text{\# SPAM} + \text{\# NORMAL}} \\ \Pr[C_S] &= \frac{\text{\# SPAM emails}}{\text{total \# emails}} \\ &= \frac{\text{\# SPAM emails}}{\text{\# SPAM} + \text{\# NORMAL}}\end{aligned}$$

That leaves the generative model. Note that if I hand you a message and say "that's spam", you have a *sample* of $\Pr[\mathbf{X}|C_S]$. Your job is to assemble a bunch of such samples to create a comprehensive model of the data for each class. Essentially, $\Pr[\mathbf{X}|C_i]$ tells you what the chance is that you see a particular configuration of letters and words within the universe of all messages of class $C_i$.

Here we make a massive approximation. First, let's break up the message $\mathbf{X}$ into a set of lower level elements, called ***features***: $\mathbf{X} = \langle x_1, x_2, \ldots, x_k \rangle$. In the case of email, a feature might be a single character, a word, an HTML token, a MIME attachment, the length of a line, time of day the mail was sent, etc. For the moment, we won't worry about what a feature is (it's the tokenizer's job to determine that—see Section 3.2 for details); all we'll care is that you have *some* way to break it down into more fundamental pieces. Now we'll write:

$$\begin{aligned}\Pr[\mathbf{X}|C_i] &= \Pr[x_1, x_2, \ldots, x_k|C_i] &(3)\\ &\approx \Pr[x_1|C_i]\Pr[x_2|C_i]\cdots\Pr[x_k|C_i] &(4)\\ &= \prod_{j=1}^{k}\Pr[x_j|C_i] &(5)\end{aligned}$$

This is called the ***naïve Bayes approximation***. It's naïve because it *is* a drastic approximation (for example, it discards any information about the order among words), but it turns out to work surprisingly well in practice in a number of cases. You can think about more sophisticated ways to approximate $\Pr[\mathbf{X}|C_i]$ if you like (I welcome your thoughts on the matter), but for this project it's sufficient to stick with naïve Bayes.

Ok, so now we've blown out a single term that we didn't know how to calculate into a long product of terms. Is our life any better? Yes! Because each of those individual terms, $\Pr[x_j|C_i]$, is simply an observed frequency within the TRAINING data for the token $x_j$—you can get it simply by counting:

$$\Pr[x_j|C_i] = \frac{\text{\# of tokens of type } x_j \text{ seen in class } C_i}{\text{total \# of tokens seen in class } C_i}$$

For example, suppose that your tokens are individual words. When you're analyzing a new SPAM message during TRAINING, you find that the $j^{\text{th}}$ token is the word "tyromancy". Your

probability estimate for "tyromancy" is just:

$$\Pr[\text{"tyromancy"}|C_S] = \frac{\text{\# "tyromancy" instances in all SPAM}}{\text{total \# of tokens in all SPAM}}$$

So when you're TRAINING, every time you see a particular token, you increment the count of that token (and the count of all tokens) for that class. When you're doing CLASSIFICATION, you don't change the counts when you see a token. Instead, you just look up the appropriate counts and call that the probability of the token that you're looking at. So to calculate Equation 5, you simply iterate across the message, taking each token, and multiplying its class-conditional probability into your total probability estimate for the corresponding class. In pseudo-code,

Given: an email message, $\mathbf{X}$, and a label $C_i \in \{C_N, C_S\}$,

1. break $\mathbf{X}$ into its tokens, $\langle x_1, \ldots, x_k \rangle$
2. for each token, $x_j$
   (a) Increment the counter for token $x_j$ for class $C_i$
   (b) Increment the count of total tokens in class $C_i$
3. Increment the total number of email messages for class $C_i$

Figure 1: TRAINING pseudo-code

Given: an UNLABELED email message, $\mathbf{X}$

1. $p_N := \Pr[C_N]$
2. $p_S := \Pr[C_S]$
3. break $\mathbf{X}$ into its tokens, $\langle x_1, \ldots, x_k \rangle$
4. for each token, $x_j$
   (a) $p_N := p_N \cdot \Pr[x_j|C_N]$
   (b) $p_S := p_S \cdot \Pr[x_j|C_S]$
5. if $p_N > p_S$ then return NORMAL
6. else return SPAM

Figure 2: CLASSIFICATION pseudo-code

And you're done. There are, of course, an immense number of technical issues in turning this into a real program, but that's the gist of it. One practical issue, however, is underflow—if the product in Equation 5 has very many terms, your probability estimates ($p_N$ and $p_S$) will quickly become 0 and it will be impossible to tell the difference between the two classes. To overcome this,

instead of working directly with the *probabilities* of tokens, we'll work with the *log likelihood* of the tokens. I.e., we'll replace Equation 5 with $\log(\text{Equation 5})$. (Question 1: how does this change the algorithm in Figure 2? Question 2: does this leave the final classification unchanged? Why or why not?)

A second critical issue is what to do if you see a token in an UNLABLED message that you've never seen before. If all you're doing is using $\Pr[x_j|C_i] = \frac{\#\ x_j}{\text{total tokens}}$, then you have that $\Pr[x_j|C_i] = 0$ if you've never seen token $x_j$ before in your TRAINING data. This is bad. (Question: why is this bad? Hint: consider what happens to Equation 5 if one or more terms are 0.) So instead, we'll use an approximation to $\Pr[x_j|C_i]$ that avoids this danger:

$$\Pr[x_j|C_i] \approx \frac{(\#\ x_j \text{ tokens in class } C_i) + 1}{(\text{total \# of tokens in class } C_i) + 1}$$

This is called a ***Laplace correction*** or, equivalently, a Laplace smoothing. (It also happens to be a special case of a Dirichlet prior, but we won't go into that here.)

Now you have enough mathematical background and tricks to implement the SPAM filter. The rest is Java...