6

TOKENIZATION: THE BUILDING BLOCKS OF SPAM

Unlike older spam filters, in which the author programs the characteristics of spam, statistical filtering automatically chooses the characteristics (or "features") of spam and nonspam directly from each email. Two years from now, when spam has evolved in content, statistical filters will have learned enough to continue doing their job. This is because unlike older spam filters, in which the author programmed rules to identify spam, statistical filters automatically identify damning features of a spam based on message content.

Tokenization is the process of reducing a message to its colloquial components. These components can be individual words, word pairs, or other small chunks of text.

Data generated by the tokenizer is ultimately passed to the analysis engine, where it is interpreted. How the data is interpreted is important, but not necessarily as important as the quality of the data being passed. In other words, the way that a message is tokenized is more important than what we do with it later; even a simple change in tokenization can affect the accuracy

of the filter. From a philosophical point of view, this raises the question, "What is content?" If content were just words on a page, then tokenizing only complete alphabetical words should be sufficient—but content is much more than that, as we'll see throughout this book.

Tokenizing a Heuristic Function

The one heuristic aspect of statistical filtering is tokenization. Even though the process of identifying features is dynamic, the way those features are initially established—how they are parsed out of an email—is programmed by a human. Fortunately, languages change slowly, and only a few minor tweaks are necessary to adapt the tokenization process to handle some of the wrenches thrown at it by spammers. Tokenization is the type of heuristic process that is usually defined once at build time and rarely requires further maintenance. In light of its simplicity, many attempts are still being made to establish tokenization through artificial intelligence, to remove all sense of heuristic programming from the equation. Within a few years, filters should be able to efficiently perform their own type of "DNA sequencing" on messages, determining the best possible way to extract data. In fact, this is already being researched as a solution to filtering some foreign languages that don't use spaces or any other type of word delimiter.

Basic Delimiters

Besides deciding how best to break apart a message, there are many other issues to consider when tokenizing. For example, we need to determine what constitutes a delimiter (token separator) and what constitutes a constituent character (part of the token). Do we break apart some pieces of a message differently than others? What data do we ignore (if any)?

The fundamental goal of tokenization is to separate and identify specific features of a text sample. This starts with separating the message into smaller components, which are usually plain old words. So our first delimiter would be a space, since spaces commonly separate words in most languages. This makes it very easy to tokenize a phrase like the following:

```
For A Confidential Phone Interview, Please Complete Form & Submit.
```

which can be broken up into the following words:

```
For      A          Confidential   Phone   Interview,
Please   Complete   Form           &       Submit.
```

As we've learned, each word typically is assigned one of two primary dispositions—spam or nonspam. The example above will cover a lot of text, but we're left with a few punctuation issues. For example, is the word "submit" on its own likely to have a different disposition from the word "submit." with a

period after it? How about "interview" and "interview," containing a comma? In these cases, it makes sense to add some types of punctuation to the set of delimiters, as punctuation suggests a break in most languages. The following are some widely accepted punctuation delimiters:

- period (.)
- comma (,)
- semicolon (;)
- quotation marks (")
- colon (:)

Some other punctuation, such as the question mark, is a bit more controversial. Some authors believe that "warts" and "warts?" should be treated the same, in most cases as spammy tokens.

Including too much punctuation in the makeup of tokens could result in five or ten different permutations of a single word in the database. This can very rapidly diminish their usefulness. On the other hand, not having enough tokens can cause the tokens to become so common among both classes of email that they become uninteresting. The trick is to end up with tokens that would stick out in one particular corpus. If there were 100 spams about warts in the user's corpus, but only one posing a question in which "warts?" was used, the filter is likely to overlook this feature in the one message.

NOTE *I've found that treating a question mark as a delimiter results in slightly better accuracy (on the order of a few messages) in my corpus testing, as opposed to treating it as a constituent character. This could likely change in the future, however.*

Redundancy

Some types of punctuation are very useful; for example, the exclamation point makes a remarkable difference between "free" and "free!" and so you want to use some punctuation marks as constituent characters. One of the problems a filter author might run into when allowing these types of characters, however, is redundancy. Most would agree that there's no real difference between "free!" and "free!!!!" in a message, as both are equally condemning characteristics of spam. On the other hand, messages in which symbols are used to b!r!e!a!k up a word may behave a bit differently.

Some authors will view punctuation as part of a token only if it appears at the end of the token. If an exclamation point appears elsewhere, it will be treated as a delimiter in most cases. So for those punctuation marks that are permitted, we should consider working some method of de-duplication into our tokenizer, where only the first occurrence of the punctuation is used. We essentially look at "free!!!", "free!!!!!!!!!!", and "free!" as the same token by truncating the extra chaff. I've found that using the exclamation point as a constituent character slightly improves accuracy, which is the opposite effect that question marks appeared to have. This is probably because more spams use an obnoxiously loud used-car-salesman type of pretense rather than

actually posing questions. Perhaps one day, spammers will become more philosophical, and then question marks will become just as useful as exclamation points.

Some filters permit a certain window size before the token is truncated; for example, tokens may be allowed to have up to three exclamation points before being truncated, giving the filter three different meanings for "free!", "free!!", and the extremely guilty and shameless "free!!!" One of the advantages to doing this, other than measuring the three levels of unbridled fervor, is that it allows a really obnoxious message that uses all three tokens to fill up more slots in the decision matrix.

It's important to truncate extraneous characters at some level because spammers could easily use *not* truncating them as a way to hide very spammy tokens; for example, a spammer wanting to hide the word "porn" could send "porn!!!" in the first spam and "porn!?!?!" the next time, so that in both cases the token would be considered a new token. Truncating will reduce both of these tokens to "porn!" or even "porn" if exclamation points are ignored all together. Tokens should generally be limited to only one acceptable punctuation mark at the end, or to an *N*-sized window of homogeneous punctuations at the most.

Other Delimiters

Other delimiters used by many applications include the following:

- brackets [ ]
- braces { }
- parentheses ( )
- mathematical operators + - / * = < >
- special characters | & ~ `
- the at sign @
- underscores and other rare characters

These delimiters frequently prevent the duplication of several different permutations of tokens, such as "when" and "(when". Other characters, such as the new line character, are also treated as delimiters. The nice thing about the way text is delimited is that it's going to result in unique tokens, even if the tokenization isn't perfect. This can be good or bad, but most of the time it's good. Even a token that isn't in human-readable format may be machine-readable and may occur with enough frequency to be a good identifier. In fact, Bayesian antivirus filtering uses an entirely different set of delimiters, because antivirus analysis involves the cataloging and analysis of several different binary sequences.

Exceptions

Some exceptions to the basic delimiters we've mentioned involve one-off instances where we actually want to preserve certain complete tokens. For example, IP addresses make for good spam markers, as do certain HTML characters like © and . If you're reading this book, there is most likely no shortage of spam in your inbox (or quarantine). Often the best way to discover new approaches to tokenization is to take a look at some of the text spammers are using in their samples. It's very important that the tokenizing approaches being used aren't biased against present-day spam.

The tokenizing algorithm should be generic in such a way that it can easily break down any kind of natural language or new type of message style, but it shouldn't be so plain vanilla that the features it generates are likely to appear as common in all email. It would be relatively easy to tokenize a message into individual characters, but that wouldn't be very useful, since the token "v" could occur in "viagra" or "violin." All-numeric tokens are generally not very useful on their own, but when combined with the proper punctuation (such as a dollar sign or exclamation mark) can make a significant distinction between "19" and "$19" or between "95" and "95!". Provide enough information to allow the token to be set apart from the rest, but not so much that it is unlikely to show up only a handful of times.

To some degree, this anal-retentive exercise is overrated. Any reasonable level of tokenization will most likely yield levels of accuracy above 99 percent, but making a mistake could cost a few misclassifications on occasion. I've found that using the question mark as a constituent character in my tests resulted in approximately 3 additional errors per 5,000. Experimentation and thorough testing is one of the best ways to decide on the tokenization approach that works best for the filter.

Token Reassembly

Occasionally, tokens will turn out to be a little too small due to attempts by spammers to obfuscate them (we'll go into more detail about the different obfuscation attempts in the next chapter). When this happens, reassembling individual letters into a token can help improve accuracy. Let's look at an example of obfuscated text:

```
C/A/L/L/ N-O-W -  I/T/S F_R_E_E
```

If the tokenizer we're using considers underscores, dashes, and slashes to be token delimiters, then instead of ending up with 4 one-word tokens, we'll end up with 14 single-character tokens. Many filter authors believe it's healthy to allow these individual characters to tokenize, while others believe that the resulting information is too generalized to be a good indicator of anything, at least without the risk of false positives.

Filter authors who share the latter philosophy can use token reassembly to join the original tokens back together. Token reassembly isn't a perfect science, but it provides more useful tokens to work with. The tokens "VIA" and "GRA" are much more useful than individual characters and are definitely more indicative of spam. Token reassembly basically concatenates single-character tokens that are adjacent to one another, looking for larger amounts of whitespace amidst the slicing and dicing to make an educated guess about what words go together. Since statistical filtering involves *machine* learning and not *human* learning, tokens like this are very useful to the computer, even though they may not make much sense to us. For example, the token "VIA" really doesn't mean much, which is exactly why it makes a great indicator of spam—you'd rarely see the word "VIA" in a legitimate message unless you were talking about motherboards. The word "GRA" is even more rare in legitimate mail. The fact that these tokens aren't necessarily comprehensible to a human makes it easier to identify them in spams. My dataset considers some of these fractional words to be extreme indicators of spam:

```
agra        S: 00030    I: 00000    P: 0.9999
eacute      S: 00021    I: 00000    P: 0.9999
prematur    S: 00020    I: 00000    P: 0.9999
```

Degeneration

Another solution Graham introduced into tokenization is called *degeneration*. Degeneration allows an token that hasn't been seen before to be reduced in complexity (location, case, and punctuation) until it matches a simpler token. If no tokens match a given token, we make it simpler until we find a match. For example, consider the use of the word "FREE!!!" in the subject. If it has never been seen before in the subject, degeneration has us reduce the phrase until it matches something we have seen before.

```
Subject*Free!!!
Subject*free!!!
Subject*FREE!
Subject*Free!
Subject*free!
Subject*FREE
Subject*Free
Subject*free
FREE!!!
Free!!!
free!!!
FREE!
Free!
free!
```

```
FREE
Free
free
```

Degeneration has a lot of room for customization, including the order in which the tokens decrease in complexity. At the very least, degeneration of punctuation is a wise move. If the word "free!" doesn't exist in the dataset yet, it makes good sense to use the value from a similar token.

Header Optimizations

Most filter authors agree that a token in the subject header is very different from a token in the message body, and that a token that appears in two different headers is unique enough to warrant keeping track of. Header tokens are usually processed differently from body tokens in order to maintain the origin of each token. Let's look at an example of an email with a lot of useful header information.

```
From: bazz@xum2.xumx.com
To: bazz@xum2.xumx.com
Reply-To: mort239o@xum2.xumx.com
Subject: ADV: FREE Mortgage Rate Quote - Save THOUSANDS! kplxl
X-Keywords:

Save thousands by refinancing now. Apply from the privacy of your home and
receive a FREE no-obligation loan quote.
http://211.78.96.11/acct/morquote/

Rates are Down. YOU Win!
Self-Employed or Poor Credit is OK!

Get CASH out or money for Home Improvements, Debt Consolidation and more.
Interest rates are at the lowest point in years-right now! This is the perfect
time for you to get a FREE quote and find out how much you can save!
```

In the spam shown here, several different tokens stand out. First, if my email address happened to be bazz@xum2.xumx.com, I wouldn't expect to be seeing it in the From: header, but it would be very normal in the To: header. Seeing my own email address in the From: header would be a clear indicator of spam, since most people don't usually send email to themselves unless they've had too much to drink.

Second, the word "Save" appears in both the subject line and the message body. I would expect to see it in the message body more frequently in legitimate mail—for example, "Save your files in the blue folder" or "Save me from this dreaded cubicle." Seeing the word "Save" in the subject header is much more suspicious, though, and it makes sense for me to have a different entry in the dataset for each of them.

The word "FREE" also shows up in both the subject line and message body, but in this case, they're both very guilty indicators of spam. The filter still benefits here because the tokens "FREE" and "Subject*FREE" now have the ability to take up two slots in my decision matrix, further condemning the spam. Header tokens are extremely useful for identifying both spam and legitimate mail.

Other types of header tokens are frequently found to be useful, and the set of delimiters used in the headers is usually slightly different from those used in the message body. For example, if I want to catch all of the IP addresses in the Received: headers, I would treat a period as a constituent character (part of the token) instead of a separator. If I wanted to tokenize the message-id, I'd also include the @ sign as a delimiter, as it is used to separate some pieces of the message-id.

Another advantage of including the header as part of the token is that it helps to create a virtual "whitelist" of users you trust. If I exchange a lot of correspondence with bobsmith@somedomain.com, tokens like "From*bobsmith" and "From*yourcompany.com" will start to appear in the dataset, usually with very innocent values. This works equally well in identifying the hostnames of trusted mail servers in the Received: header too.

URL Optimizations

Everyday innocent-sounding words like "order" and "cgi" often appear in the body of messages I receive from legitimate mailing lists. Seeing them appear in a URL, however, is much more suspicious. URLs are the spammers' preferred means of contact. It's much easier to run a scam using a website as your point of contact than it is to pay for the overhead of a phone system or mail processing department. Spammers also like their privacy, since the rest of the free world hates them, and they prefer that even customers not know how to contact them or the companies they spam for. Whether it's a link to click to visit a site or the URL of an image inside the message, URLs provide a lot of useful information specific to their own kind. Even non-sensible numbers will frequently stand out in URLs. This makes really good data for identifying not only spam but some legitimate mailing lists that use URLs in their unsubscribe tag lines. Users who are subscribed to some mailing lists that frequently include embedded advertisements (such as Yahoo Groups) will notice some specific characteristics of the URLs used in these advertisements that help the filter distinguish between advertising and real spam.

URLs are frequently tokenized differently than the rest of a message. The only delimiters usually used when tokenizing a URL are the slash, question mark, equal sign, period, and colon, although some filter authors perform the same basic type of token separation as they do in the rest of the message body. Tokenizing using URL-specific delimiters is done because the individual tokens are more frequently found based on their path in the URL, rather than on a specific context inside the URL. Regardless of how they are tokenized, URLs, when analyzed, can yield a lot of useful information. They

can be categorized as places you want to go and places you don't want to go. A spam containing places you don't want to go is just as informative as a legitimate message containing places you do.

```
Url*getitrightnowwholesale    S: 00026    I: 00000    P: 0.9999
Url*thesedealzwontlast        S: 00026    I: 00000    P: 0.9999
Url*biz                       S: 00008    I: 00000    P: 0.9998
Url*us                        S: 00000    I: 00050    P: 0.0001
Url*java                      S: 00018    I: 00000    P: 0.9999
Url*www                       S: 00000    I: 00030    P: 0.0001
Url*com                       S: 00000    I: 00033    P: 0.0001
Url*img                       S: 00066    I: 00000    P: 0.9999
```

Ironically, legitimate URLs seem to be rare among spammers, while the wild and obnoxious names always pop up—with the exception of "java," of course, which appeared as spammy only because this user doesn't use Java (not because Java programmers were spamming). The appearance of certain naming conventions, such as the extensive use of "img," makes the task of identifying malicious URLs pretty easy. If we wanted to, we could probably determine the disposition of the message based on the URL information alone.

Ironically, URLs containing well-known web addresses are likely to appear as innocent or hapaxes. Not a single URL token containing the following words has ever appeared in my corpus as spammy:

- Url*microsoft
- Url*whitehouse
- Url*sco
- Url*linux
- Url*quicken
- Url*intuit
- Url*amazon
- Url*fbi

HTML Tokenization

One area that has plagued many filter authors is the decision as to what HTML to include and what other parts of the message to ignore—for example, should we ignore JavaScript? What about font tags? Most filters pay attention to all HTML tags except those on an exclusionary list—namely, a specific set of tokens that are common to all types of email. This approach works quite well, but there's still room for improvement. Ignoring data is always something to be concerned about, and you shouldn't do it unless you have good reason. The justification for ignoring some HTML data is that many people normally converse only with senders who do not use HTML.

This could cause any type of message with embedded HTML to be rejected as spam, which could be bad for the recipient if their boss suddenly started using an HTML-enabled mail client. The tags most filters ignore include

- td
- !doctype
- blockquote
- table
- tr
- div
- p
- body
- short tags, with fewer than *N* characters of content
- tags whose content contains no spaces

It is probably better to use an exclusionary list rather than an inclusionary one. You're more likely to miss a few tags or possibly to fail to name certain tags you never thought could be used in spam (for example, the object tag has recently become popular). If this happens, at worst the tag will sit and collect dust in the dataset with some neutral value or will fill up a decision matrix slot in error. If you fail to add a tag to an inclusive list, though, you're bound to ignore an important data point and may not even realize it.

Some of the HTML tags commonly used by spammers (which a filter should definitely be looking at) include the following:

```
APPLET    BGSOUND    FRAME    IFRAME
ILAYER    IMG        INPUT    LAYER
LINK      SCRIPT     A        AREA
BASE      DIV        LINK     SPAN
OBJECT    FONT       BODY     META
```

Some filters like to mark the tokens generated from HTML tags with an "HTML" identifier, while others go so far as to mark the particular tag the text belonged to (for example, "BODY:BGCOLOR=#FFFFFF"). Regardless of which tags the filter decides to keep and which get discarded, it's very important to handle HTML comments correctly. Spammers are using many tricks to obfuscate their text so that it's human readable, but not very machine readable. For example, the following may look like a complete mess in its machine-readable format:

```
Received: from 64.202.131.2 (h0007e9075130.ne.client2.attbi.com
[24.218.222.43])
Message-ID: <cp6-mh-rn-w$4pa2o965rl84@jn4yOhq1bcy>
From: "patsy stamm" <arthropathology71255@earthlink.net>
Reply-To: "patsy stamm" <arthropathology71255@earthlink.net>
Subject: Giving this to you
Date: Fri, 08 Aug 03 07:29:02 GMT
```

```
X-Mailer: MIME-tools 5.503 (Entity 5.501)
MIME-Version: 1.0
Content-Type: multipart/alternative;
        boundary="AD0E55.76_15.C"
X-Priority: 3
X-MSMail-Priority: Normal


--AD0E55.76_15.C
Content-Type: text/html;
Content-Transfer-Encoding: quoted-printable

Yes you he<!lansing>ard about th<!crossbill>ese weird <!cottony>little
pil<!domesday>ls
that are suppo<!=anabel>sed to make you bigger and of cou<!chord>rse you think
they're b<!soften>ogus snake potion. Well, let's look at the facts:
<strong>G<!eigenspace>RX2
has be<!waldron>en sold over 1.9 Mill<!audacity>ion times within the last 18
months</strong>...
With awe<!tapestry>some results for hun<!wield>dreds of thous<!locale>ands of
men all over the planet!  They all enjoy a seriously enhanced version of their
manh<!rescind>ood and <b>why shou<!seoul>ldn't you</b>?
```

But when the user clicks the message to read it, the HTML comments won't be visible, and the user will see this:

```
Yes you heard about these weird little pills
that are supposed to make you bigger and of course you think
they're bogus snake potion. Well, let's look at the facts: GRX2
has been sold over 1.9 Million times within the last 18 months...
With awesome results for hundreds of thousands of men all over the planet!
They all enjoy a seriously enhanced version of their manhood and why shouldn't
you?
```

A simple way to ensure that the message is tokenized correctly is to remove the HTML comments and reassemble the message.

Word Pairs

Using word pairs, or *nGrams*, has recently become very popular among authors of statistical filters and adds a lot of benefits to standard single-token filtering. Pairing words together creates more specialized tokens. For example, the word "play" could be considered a very neutral word, as it could be used to describe a lot of different things. But pairing it with the word adjacent to it will give us a token that will inevitably stick out more when it occurs—for example, "play lotto." This approach helps improve the processing of HTML components by identifying the different types of generators used to create the HTML messages. Each generator, whether it's a legitimate mail client or a spam tool, has its own unique signature, which joining tokens together can

help to highlight. Tokenizers that implement these types of approaches are referred to as *concept-based tokenizers,* because they identify concepts in addition to content. We'll discuss the different implementations of nGrams in Chapter 11.

Sparse Binary Polynomial Hashing

Bill Yerazunis originally introduced the concept known as *SBPH*, or *sparse binary polynomial hashing*. SBPH is an approach to tokenization using word pairs and phrases. If it wasn't so effective at what it does, it would probably be a terrible idea—but Yerazunis has repeatedly astonished the spam-filtering community with the leaps in accuracy made by SBPH tokenization. Graham refers to SBPH with the same mixed feelings regarding its ingenuity and need for medication:

> Another project I heard about . . . was Bill Yerazunis' CRM114. This is the counterexample to the design principle I just mentioned. It's a straight text classifier, but such a stunningly effective one that it manages to filter spam almost perfectly without even knowing that's what it's doing.

SBPH tokenizes entire phrases, up to five tokens across, and allows for word skipping in between. It led the way in terms of accuracy for a long period of time, but it also created an enormous amount of data, which is one of the reasons it presently functions only in a train-on-error environment. SBPH provides the benefit of using the simplest, most colloquial tokens but giving special notice to more complex tokens as well—which are usually much stronger indicators of spam when they appear.

A few filters, such as CRM114, perform this type of word skipping, which will tokenize something like "manh+<!rescind>+ood" and also help the filter "see" the original token by performing the word skipping: "manh+ood." Since tokenization is an imperfect process, approaches like this generally provide more machine-readable tokens to deal with, without necessarily requiring much work. The more permutations of machine-readable tokens are created, however, the larger and more spread out the dataset will become, possibly affecting accuracy. The amount of data generated by SBPH generally turns a lot of filter authors off to it in favor of simple functions such as HTML comment filtering. We'll cover SBPH more in depth in Chapter 11.

Internationalization

The tokenization methods discussed thus far have covered only standard character sets. The issue of foreign languages will eventually require a solution. Most spam filters simply use wide characters as placeholders, such as the letter "z" or an asterisk. This functionality allows the filter to catch just about any messages written using a wide character set. Some users, however,

may expect to receive email from others speaking such a language, and for them this approach won't function well at all, filtering only based on header data. The rest of the body will look (to the filter) like this:

```
ZZZZZ,

ZZ ZZZZ ZZZ ZZZZZZZ ZZZ ZZZ Z ZZZZZZ Z ZZZZZZ
ZZZZ Z ZZZ ZZZZ ZZZZZZZ ZZ ZZZZZZ ZZ ZZZ ZZZZZZZ

ZZ,
ZZZZZZZZ
```

Some filters implement i18n internationalization, which lets their filter support some additional languages. To make matters more complicated, however, some languages don't use whitespace, making it very difficult to identify words at all. This commonly calls for more advanced solutions such as variable-length nGrams, which we'll discuss in Chapter 11.

Final Thoughts

We've run the gamut of approaches to tokenizing in this chapter. We'll learn more about tokenizing phrases in Chapter 11, and in Chapter 13 we'll cover another type of prefilter that actually despeckles the noise inherent in tokens. Tokenizing strives to define content by defining the construct and, more importantly, what the root components of content are. This is a noble quest, but, as with other areas of machine learning, is a function that may eventually be better left up to the computer. As new types of neural decision-making algorithms surface, the analysis of unformatted text may become one of the next forms of AI. Until this happens, tokenizing remains one of the few heuristic components of a statistical spam filter. It should therefore be respected and kept somewhat simple, so as not to require any maintenance in the years to come.