How to Index Text

Clinton P. Mah

Walnut Creek, California

D R A F T

1 October 2022

1. History

People have been indexing text for a long time.[1] It began as a tool allowing scholars and students to locate material in a scroll or a codex without having to read through it all. For example, searchers with some kind of concordance can readily find the first occurrence of “Hittite” in Biblical source material. Before published indexes, searchers had to rely on librarians familiar enough with their collections to guide navigation through all the written material. Their job required a prodigious memory.

Nowadays, non-fiction books almost always include their own indexes, but when every book had to copied laboriously by hand from another manuscript, adding extra pages was a luxury. Readers began, however, to compile their own notes to facilitate later reference; and these were often also copied, edited, and shared as well. The printing press with movable type later made such copying easier, launching the information age long before the first digital computer.

Most literate people today understand the idea of indexing, but doing a good job of it requires nontrivial expertise. Unsupervised machine learning has become a popular automated approach to this problem in the 21st Century, since text text data today has grown far too voluminous for any single person to read through. Nevertheless, we cannot simply turn the problem over to some blackbox neural net. Whether manually or automatically compiled, a useful index must align with how humans view their information.

This extended survey will explore some nuts and bolts of good indexing in both intuitive and technical ways. It will be grounded in a somewhat historical approach, starting from the librarians in medieval monasteries. Their approach to indexing was rudimentary by necessity, but still suffices for many applications. Modern keepers of digital information, however, are forever aiming towards the next iteration of a system and trying to improve its effectiveness with small tweaks. This can sometimes will lead us to unexpected places.

The evolution of systems is not always progress, but by continually drawing upon knowledge of linguistics, statistical information theory, mathematical modeling, and the nature of our world, we can get a better grasp of basic problems and keep sharpening our tools. Effective indexing is a goal far short of understanding natural language, but doing this job accurately and reliably will allow all automated text processing to operate more intelligently and more transparently.

Text indexing can be more rigorous and compact. Properly done, it can allow us to organize search results for easier interpretation, to calibrate match thresholds for standing profiles scanning streams of text data, to cluster documents by content more flexibly, to divide long documents into coherent chunks, to do quick text summarization, to refine statistical tests of text authorship, and to introduce quality control in the management of real-time text data streams.

This all sounds like a tall order, but we can approach the overall problem by dividing and conquering. It is possible to do better text analysis without any big breakthrough in technology or even without any reliance on advanced machine learning. Statistical concepts will drive our thinking here, but it will be mostly at an undergraduate level. In the end, we can actually demonstrate our ideas in a real system running on commonplace computers to process real text data with a minimum of fuss and effort.

---

[1] Judith Flanders, Everything in Its Place, 2020.

2. Words, Words, Words

For proper indexing of text, we need to know more than a little about the language in which it is written. In the 21st Century, our text will often be in Unicode, which allows text in almost any language to be encoded digitally for processing. This does not, however, mean that some kind of automaton can now just be turned loose on any collection of Unicode text data to lay everything out for information users to dig into.

The problem of natural language processing is hard to solve, even when limited to the indexing of content. In particular, a solution for indexing Unicode text in Chinese cannot readily handle Unicode text in Vietnamese or in Japanese or in Korean. These languages are grammatically quite different from each other and have singular non-trivail writing systems: Chinese is ideographic, Vietnamese is alphabetic with many tonal diacritical marks, Korean employs a syllabary formatted to look ideographic, Japanese combines ideographs with two different syllabaries.

In contrast, English looks like an easy language to index, but probably only to those who speak it as their first language. The difficulties of English arise from chaotic spelling rules, peculiar punctuation, irregularities in inflections, complicated usage of verb auxiliaries and particles, itas plethora of synonyms, a casual regard to parts of speech, and a propensity for convoluted sentence construction.

English is nevertheless a good choice to work with because of its widespread usage. Mandarin Chinese has almost as many more speakers, but is non-alphabetic; and any indexing method for it will probably generalize poorly to most other languages. In English, text consists of words consisting of sequences of letters separated by spacing and by punctuation; in Chinese, there is no spacing, and readers have to know or guess where multi-character words begin and end.

As a first attempt at indexing a body of English text, we can just ignore all punctuation and just record and sort the occurrences of all distinct words. This was done in Strong's Comprehensive Concordance of the King James Version of the Bible.[2] If you have ever seen a printed copy of this index, however, then you will grasp the problem right away: the concordance is about the same size as the King James Version Bible itself. This is highly inefficient even when everything is completely digitized.

Efficiency is a perennial problem in data science. When seeking solutions in text processing, researchers often try to work with simplified models defined as an abstract space. The dimensionality of that space depends on how many basic elements go into the model. If we take all the possible forms of words in a collection of text like the Bible, then the count could be on the order of 10^6. Such numbers are hardly a stretch for 21st Century processing power, but why buy a GM Humvee if a Honda Fit will do?

High dimensionality in a model always has a cost, which needs to be justified. For example, the word "THE" is indexed in Strong's Concordance as occurring 17 times in just the first 5 verses of the first chapter of Genesis. Such information possibly might be of interest to a few specialized linguists, but no preacher or theologian or layman would really want to pay for such information. It is doubtful also whether a neural network operating with limited dimensionality in its input layer should even bother with it.

In English, words like "THE," "A," , "AN," "AND," and "OF" serve to define the structure of sentences. They are found in almost all English text, regardless of its content. Accordingly, they are called "grammatical" or "function" words, as opposed to "content" words like "GOD" and "LAW." These categories will vary across different natural languages, but in English, we could say that nouns, verbs, adjectives, and adverbs are content words, while all other parts of speech are function words.

English throws a curve ball at us, however, by allowing some words to be multiple parts of speech. For example, the word "IN" could be a noun ("I have an in with the boss."), an adjective ("the in crowd"), an adverb ("throw in the towel"), or a preposition ("in the weeds"). Fortunately, English has only a few

[2] Strong's Concordance.

problematic words so ambiguity, and we can deal with them as special cases. Our medieval monastery librarian probably would omit it from a remembered or an actually written out index.

Common practice is to identify several hundred words as being functional in English and bar them as separate entries in an index. It is easy to tailor such a listing to a particular text processing application, but the default in our hypothetical monastery would probably be to exclude as many function works as possible just to save ink. The result will be to eliminate about half of the word occurrences in a collection of English text from any index.

This ratio of functional to content word occurrences seems also true of languages other than English. In general, statisticians have observed that a small fraction of the words in any natural language will account for most of the total number of word occurrences in a large text sample. No one can really explain why, but the phenomenon is called Zipf's Law,[3] which is commonly stated as the relationship

$$F(w) = k \cdot \frac{1}{rank(w)}$$

where w is a word, $rank(w)$ is the ranking of w in the listing of all words in descending order of total frequency F, and k is total frequency of the most frequent word. For example, "THE" will typically be the most frequent word in a collection of English text, so that $rank$("THE") = 1. We can then set k to be the total frequency of "THE" in the collection and estimate the total frequency of all other words.

Words and other linguistic elements will vary greatly in frequency in text. This will affect their relative value for indexing because we want our elements to be fairly balanced. Zipf's Law, though not exact, gives some insight on where to find the elements most out of unbalance: they are going to be the most frequent ones and more likely to be grammatical function words like "THE." This is all the more reason to disregard them for indexing.

A complication here is that, in English, many function words are individually infrequent, while some content words can be quite frequent in a given corpus of text data. Exclusion from indexing of words by frequency alone is highly problematic. Fortunately, there are many reliable listings of English function words irrespective of frequency that any indexer can start from. Some people worry that will result in some loss of information, but tradeoffs are unavoidable, and words like "THE" are far more likely to add noise than signal to an index.

We should generally avoid relying solely on mechanical processes to make decisions about indexing. Human judgments about language are still trustworthy and desirable. For the time being, an unsupervised deep neural net will probably be inferior to a twelve-year-old at understanding natural language. Language is in our DNA, as it were. In any event, putting words like "THE" and "AND" by themselves into into an index has to make any medieval or modern reader look askance.

---

[3] Zipf's Law. Wikipedia.

3. Inflections

We have only begun to solve the dimensionality problem in creating indexes. It is a good start to remove a few hundred words mainly serving grammatical functions, but those are only a drop in the bucket. A typical written natural language will have over 10^4 distinct words that express content. Just count up, for example, the number of code points for distinct Chinese characters in Unicode. Almost every character can be read as a single word, and multiple characters can string together into longer, more complex words.

Alphabetic writing presents even more problems for indexing. Nouns, verbs, and adjectives in many languages can vary in their forms according to their usage in a sentence. These variants will look different, but are really the same noun, verb, or adjective and should really be listed only once in a compact index. In European languages like English, French, Spanish, or Russian, the various forms are distinguished by the endings of a word. In others like Arabic, there is systematic shifting of vowels.

The variation in word form can range from simple to complex within any family of languages. For example, Latin nouns can take on one of five grammatical cases (nominative, genitive, accusative, dative, and ablative) as well as singular or plural; and different categories of nouns can follow different patterns. This sounds quite challenging, but Russian is worse in having ten grammatical cases for nouns. The verbs of both Latin and Russian also change their form to reflect person, number, tense, aspect, and mood.

The systematic grammatical variation of nouns and verbs is called “inflection.” Some languages like Chinese really have none. Japanese employs Chinese characters (kanji) in its writing system, but tacks on inflections in its own katakana syllabary. Inflections are much less obvious in alphabetic languages when everything is written in the same 24 to 32 letters; and readers must somehow figure out which letters are the root of a word and which are inflections.

In English, inflections show up mainly as the suffixes “-S,” “-’S,” “-ED,” and “-ING.” So, the root word “INDEX” can become “INDEXES,” “INDEX’S,” “INDEXED,” and “INDEXING.” Someone compiling an index by hand would undoubtedly want to count all the variations as being the same, but for automatic approaches to indexing, the removal of such suffixes involves some rather complicated rules. For example, “PADDING” to “PAD,” “FADING” to “FADE,” “TYING” to “TIE.” Formulating such rules is tedious

Unfortunately, natural language inflections can get more complicated by the retention of archaic forms as irregularities. In English, some common verbs have an expected past tense “-ED” showing up as “-T”; for example, “SPEND” + “-ED” becomes “SPENT.” This nevertheless is an inflection and should recognized in indexing. Other irregular verbs have a distinct participle form departing from the usual “-ED” suffix, like “SHOW,” “SHOWED,” and “HAS SHOWN” versus “SNOW,” “SNOWED,” and “HAS SNOWED.”

In English , “-ER” and “-EST” suffixes indicate the comparative and superlative forms of adjectives, usually only when the root form has only a single syllable. For example, the adjective “RED” has the comparative form “REDDER” and the superlative form “REDDEST.” A multisyllabic adjective like “CURIOUS” indicates comparatives and superlatives in compound expressions “MORE CURIOUS” and “MOST CURIOUS,” but this is not inflection. It is something else to keep track of.

The removal of letters indicating the inflections of a word is called “stemming.” This was commonly done in the 20th Century indexing of text documents for information retrieval systems. Stemmers for particular languages can be downloaded from the Internet,[4] but while adequate for search engine support, their output may be too cryptic for humans to read. In English, some stemmers also can reduce words to the wrong root (e.g. “HATING” to “HAT”).

On the whole, English is much less inflected than other European languages, which should make it easier to do accurate and readable stemming. The main problems are the spelling rules of English like “TEACHES” and “TYING,” irregular forms like “SING,” “SANG,” “SUNG,” and borrowed words from other

[4] https://en.wikipedia.org/wiki/Stemming

languages with plural forms, giving us “CHERUBIM,” “OCTOPI,” “PHENOMENA,” and “CHILDREN.” These make accurate inflectional stemming a challenge, but it is still worth doing even imperfectly.

In English text, removing inflections should greatly reduce the dimensionality of automatically generated indexes. Typically, nouns and verbs will account for about three-quarters of the English content words that we might want to include in an index. In principle, we could just define a separate rule to handle each possible inflected form, but we can more efficiently accomplish almost the same thing with fewer than six hundred more general rules.[5]

The rules can be expressed as a series of tests highly specific to English. For example,

```
If a word ends in -SS, then there is no -S inflection.

If a word ends in -YSES, there is an -S inflection, but rewrite the root
to end with -YSIS.

If a word ends in -TIES, there is an -S inflection, but rewrite the root
to end with —TY.
```

These rules look quite complicated, but anyone literate in English will know them.

With a different alphabetic language, we would have to compile new rules of inflection and different irregularities to watch for. Sometimes we can be lucky; for example, French also has “-S” as a plural inflection similar to English. With languages like Hebrew, Finnish, Tamil, or Navajo, we are out of luck. If a language is well-known, however, we should be able to find a comprehensive grammar to give us a head start. For serious indexing in any language, however, such detailed preparation should always pay off.

[5] PyElly. GitHub.

4. Morphology

We can improve both manual and automatic indexing just by dropping of frequent grammatical function words and by mapping inflected nouns, verbs, and adjectives to root forms. Some machine learning advocates, however, will argue that such preprocessing of text data is unnecessary. If some unsupervised deep neural net is going to solve a problem like text indexing, then it might as well also figure out such details like functional words and inflectional endings. Unlabeled training data is plentiful on the Internet.

On the other hand, why make an artificial intelligence work much harder than it has to. With a challenging problem like indexing, we humans can actually offer some insight. After all, people are experts on language, having invented it and spent at least 1,000 generations refining it. Data is data, but we know how to read text and understand it and see no need for a machine learning system to start from A-B-C each time we want to build another index.

Linguistic knowledge has already helped us to make indexes for English text more compact and usable. We still need, however, to reduce the number of distinct words to track in text. Those words can show up in different forms as nouns, verbs, adjectives, and adverbs. but sometimes have the same root. For example, English has “REAL,” “REALLY,” “REALIZE,” “REALIZATION,” “REALISTICALLY,” “REALITY,” AND “REALISM.” We can make all of these all equivalent to “REAL.”.

In English, the part of speech of a word will usually be changed by adding a suffix. In the examples above, the adjective “REAL” + “-LY” becomes an an adverb; the adjective “REAL” + “-IZE” becomes a verb, which + “-ATION” becomes a noun; the adjective “REAL” + “-ISTIC” + “-ALLY” first is unchanged in part of speech after the first suffix, but then becomes an adverb; the adjective “REAL” + “-ITY” or + “-ISM” becomes a noun. All of this should be familiar to any 12-year-old ablrd to read English.

If you want most to minimize the dimensionality of our indexes for English text and are willing to sacrifice a little precision in indexing, then you can map all part-of-speech transformations of words back to their roots. In English, this can be accomplished with a morphological stemmer, much like our inflectional stemmer, but driven by a set of rules that can be applied recursively.[6] As with inflections, there will be many special cases to recognize and English spelling rules to apply in removal of suffixes

With many more suffixes to check, morphological stemmers will have to work harder to check whether the final letters in a word are an actual suffix. For example, “RELY” is not “RE” + “-LY,” “SEIZE” is not “SE” +”-IZE,” “CEMENT” is not “CE” + “-MENT,” and “NATION” is not “NAT” + “-ION.” None of these should be news to fluent readers of English with no special linguistic training. Varying the parts of speech of a word is a common rhetorical device that can help in identifying topics in English written text.

Suffixes are often taught in courses on English as a second language, where it is a quick way for learners to expand their vocabulary.[7] Usually, 50 or 60 suffixes are presented, including “-AL,” “-AN”, “-FUL,” “-HOOD,” “-IC,” “-ION,” “-IST,” “-IZE,” “-LY”, “-MENT,” “-NESS,” “-OUS,” “-RY,” and “-Y.” An indexing algorithm need not recognize every possible suffixes. On the other hand, being more comprehensive does not hurt, and sloppiness in linguistic details can make an index less credible to users.

Here are some examples of morphological stemming rules:

```
If a word is CEMENT, then there is no suffix.

If a word ends in -DIMENT, then take off -MENT and change the remaining I
to Y.

If a WORD ends in -NDIMENT, then there is no suffix.
```

---

[6] PyElly. GitHub.

[7] E.g., https://englishstudyonline.org/suffix/ , https://dictionary.cambridge.org/us/grammar/british-grammar/suffixes

```
If a WORD ends in -EDIMENT, then take off -MENT and change the remaining I
to E.

If a WORD is SEDIMENT, then there is no suffix.
```

The rule that matches the most number of letters at the end of a word takes precedence.

Because morphological stemming targets words derived from simpler words, we can always define a stemmer just by listing all the target words of interest and giving their derivations. This could be done in a single pass through a dictionary, or we could approach a solution gradually by looking first at subsets of words with frequent endings. No special linguistic or computational expertise would be called for. A few literate high school students could handle the task.

There is a point of diminishing returns for morphological stemming, however. With removal of a few dozen of the most common suffixes, a page of text may have only one or two words with other suffixes. This is different with inflectional suffixes in English, which can show up with any noun, verb, or adjective. Morphological stemming will reduce the dimensionality of an index less because only a small fraction of all words are formed by adding suffixes to roots, although bureaucratic language may be an exception here.

5. Word Importance

To build a good index, whether for a medieval monastic librarian or for a massive online digital archive, we have to keep its dimensionality in check. If an index has too many keys, it is wasteful of resources and is more cumbersome to use; if it has too few keys, then the index will be rarely usable at all. Up to now, we have been exploiting our linguistic knowledge to eliminate certain words and to combine other words for a more compact index that still allows readers to find efficiently what they are looking for.

For a single book of about three hundred pages, a publisher would want an index of about ten pages. At about 100 word entries per double-column page, that would total only 1,000 entries in all. We expect at least 10^4 unique words in our book, which means that we need a way of selecting only a tenth of those words for an index. For a book, this job could be done the author, an editor, or a hired professional, but for uniform access to large archives, some automation is almost unavoidable.

A common practice is employ some sort of statistical selection. We can start by defining the probability p_z of a word z as the fraction of its occurrences in a collection of text items with respect to occurrences of all words in that collection.

$$p_z = \frac{\sum_{i=1}^{N} f_{zi}}{\sum_{w \in W} \sum_{i=1}^{N} f_{wi}}$$

where N is the total number of items, W is the set of non-excluded words occurring in a collection, and f_{wi} is the frequency of word w in item i.

One way to get indexing keys is to select the top M words according to their p_z, where M might be one thousand or two thousand. This is reasonable if we have already excluded frequent functional words from W, but it still tends to favor the most general content words. These will give us a myopic view of a book or archive, which may lack enough detail needed for easy discovery of the information of interest to a reader. We have to be more picky about our indexing keys.

In statistics, any data distribution of a single variable can be described by a series of basic numbers. For f_{zi}, the first number (or 'moment') turns out to be p_z, a measure of its relative frequency in a collection. Now let us define a new probability

$$p_{zi} = \frac{f_{zi}}{\sum_{w \in W} f_{wi}}$$

which is the relative probability of word z in text item i; this allows us to define

$$V_z = \frac{\sum_{i=1}^{N} (p_{zi} - p_z)^2}{N}$$

This is a variance statistic. It measures how much the relative frequency p_{zi} of a word x changes across a collection of text. The formula above differs from the variance formula found in textbooks because it is based on relative frequencies. Such values are easier to compare when working with text items, which typically differ greatly in length. Variance V_z will be the second basic descriptive number (or 'moment') for a distribution, telling us about its spread versus its total mass as shown by p_z.

For indexing, the most useful word keys will distinguish some small subset of items, but yet still be fairly frequent. V_z and p_z give us a way of identifying such words, although specific thresholds for them will probably have to be adjusted for particular collections of text and particular indexing applications. In theory, we can bring more information to bear on our choices by going to higher-level statistics descriptive of distributions, but these will require much more computation for only slightly better word picks.

If we really want to get the very best set of M indexing words, where M is about a thousand, we would be better off choosing $2M$ candidates with V_z and p_z and then manually editing out half of them. This solution

would be more or less adequate for a medieval monastic librarian at least. It would be inadequate for a huge online digital online archive because of severely limited coverage, but could make a good starting point for a bigger indexing set relying on more than word as its keys. This will all come out later here.

In the meantime, we should take advantage of available statistical tools, along with our human linguistic expertise, to attack some difficult fundamental issues with text data. If we can index text data reliably, transparently, and compactly, then we can better frame the problems of information analysis and thereby unleash a host of computational technologies for stronger solutions. Before we can break the world record in the 100-meter dash, we have to learn first how to crawl.

6. Information Value

What we have seen so far for indexing of text is nothing really new. Other than the mathematical notation and the algorithmic expression of stemming rules, a medieval monastic librarian would be quite familiar with all our problems and solutions. The idea of statistical moments of the distribution of a random variable is 20th Century, but the basic ideas come from the 19th Century. An advanced-placement high school student might be learning them. The challenge is to put all our existing knowledge together.

Assuming the selection of a set X of M words for indexing some text data T, we next need some way to assess how helpful X will be for a reader. Information theory[8] provides a convenient measure. Suppose that we have words x is in X; then the information value H of X in its indexing of T could be expressed as

$$H(X) = -\sum_{x \in X} p_x \cdot log_2 p_x$$

where the probability p_x for word x is as defined previously. This is an entropy measure, developed originally to define the notion of temperature in a physical realm, but repurposed to describe unpredictability of an index key in an abstract information realm.

If w pick a text item at random without knowing anything about it, then any x in an ideal index set X should have the same prior probability of applying to it. If some x has prior probability higher than the others, then X is biased: it conveys less information about a collection of text data than it could ideally. When all probabilities are equal, $p_x = 1/M$; and $H(X) = log_2 M$, the number of binary digits (bits) needed to represent the number M. The is the maximum information X can convey about text data T.

Given Zipf's Law, we know that no index set X of nontrivial size will ever be ideal, but if $r = H(X)$, then indexing with X should allow us to distinguish between 2^r items. If the number of items in T is much greater than 2^r, then X is probably inadequate for indexing it. When we have two possible index sets X and X' of the same size, then we compute $H(X)$ and $H(X')$ to determine which is better. This is helpful to know when trying to set up an index for text in a systematic way.

We can of course increase the entropy of any index set simply by adding a bunch of new keys. Some additions, however, will be better than others, and now we have a tool to evaluate them. This entropy tool has been available for more than a half-century, but text information system have avoided it by assuming in effect that their indexing sets will always be infinite. This leads to great inefficiency and makes it difficult to move from simple searching to much more intelligent processing of text data.

This is the problem of text data dimensionality rearing its head up again. To make progress here, we must first recognize that problem and understand it. The most important step in seeking a solution is just to acknowledge that text data is quite different from other types of data and that its special characteristics will give us an opening to greater exploitation of it with already available tools. This is what we have been doing all along up to this point, and it will remain our strategy going forward.

[8] Shannon.

7. Multiple Keys For Text Items

Looking for the occurrences of a single key in an index may be enough for someone seeking a reference in the Bible or in the plays of Shakespeare. If we are scouring through months of subpoenaed email messages or abstracts of recent papers published on vaccine research, however, we probably want greater precision in our matchups to avoid being buried by them. This brings us into the province of search engines, which can readily be built on the word indexes we have explored so far.

The basic idea is to make matchups on more than a single index key. Computationally, this can be accomplished in various ways, but some are more friendly for human users than others. One of the pioneering approaches was introduced by IBM at the dawn of commercial computing in a mainframe text retrieval system called STAIRS.[9] This allowed users to search a collection of text by framing logical requirement for the occurrence of index keys in an item to be retrieved.

This is idea of a Boolean query, which is still supported in modern search engines like Google under the rubric of "advanced searching." An example of a query as a logical expression is ((MARTIN AND LUTHER) OR (PHILIPP AND MELANCHTHON) OR (HULDRYCH AND ZWINGLI)) AND REFORMATION NOT WITTENBURG. This is looking for the name of one of three leaders of the German Reformation in a text item not mentioning the city of Wittenburg.

Boolean queries can be quite precise, but they are equivalent to a bit of code, which requires programming skill to craft and inevitably to debug. This is unfriendly to most non-expert information seekers and also makes it difficult to support common search features like query by example ("find more items like this one") or relevance feedback ("adjust my retrieval to exclude items like this one"). Such searching will be less precise, but is friendlier to users. This is more than enough compensation for lower precision,

Since the 1970's, the main solution to the querying problem has been to represent both text items and queries as numeric vectors. Suppose we have an index set X, with keys $\{ x_j \}$ listed in some order. Then we represent item i in a collection as the vector $\mathbf{v}_i = [v_{1i} , v_{2i} , v_{3i}, \ldots]$, where

$$v_{ji} = f_{x_j i}$$

Where f_{xi} is as defined above. In this framework, a query is no longer a possibly complex logical expression, but another numerical vector $q = [q_1, q_2, q_3. \ldots]$ of weights associated with the index keys $x_1, x_2, x_3. \ldots$.

We can determine a degree of match between a query q and an item i by computing an inner product score

$$IP(q, i) = \sum_{j=1}^{M} q_j \cdot v_{ji}$$

This is a measure of relative match between a query q and an item i. Usual practice is to compute this score for every item i and return those with the K highest scores. It is possible that item with the top score may actually match a query poorly; that s, best may fall short of being good. An information seeker should see this right away, however, and could decide to look no further down in a retrieval list.

The important takeaway here is that the vector for a text item has the same structure as the vector for a query, although queries will get their vector numbers differently. This makes it easy to implement query by example: just compute the inner product score between the vector for an example item and the vectors for all other items. We can also easily adjust the contribution for a particular index key in scoring a match, which would be awkward to pull off in a Boolean query.

---

[9] IBM STAIRS. Wikipedia. https://en.wikipedia.org/wiki/IBM_STAIRS

8. Exploring Vector Spaces

The vector representation of text items was first advanced by Gerard Salton as a basis for text information retrieval.[10] This involved a radical simplification of the complexity of natural language text, but such modeling proved to be a useful framework for approaching important text data problems like searching, classification, and clustering. The neural networks underpinning 21st Century artificial intelligence often represent text input into vectors.

Numerical vectors are an important theoretical construct in modern mathematics and physics and are built into many computer programming languages. They have largely been useful for automated text data processing, but ithey can be an oversimplification. We must be careful to avoid becoming enthralled in an unwarranted impression of mathematical rigor and thinking that this gives some special insight into the problem of understanding natural language.

In particular, when we map text items into some abstract space of numerical vectors, it is tempting to believe that we no longer have to contend with the messiness of language and meaning. Instead, we might see only clean numbers lined up in neat rows. With vectors, we can draw upon centuries of mathematical tools to get quick results. Such results can sometimes seem impressive, but any kind of system here will have its deficiencies and has to be viewed critically if we ever want to do better.

With a vector model, every text item corresponds to a single point in a high-dimensional space defined by numerical coordinates. Such a space looks like an extension of the 2-dimensional X-Y plane of high school algebra, sprouting many thousands or even an infinite number of new orthogonal axes. Accordingly, we can compute a Euclidean distance *d* between every pair of points **y** and **z** using the Pythagorean theorem.

$$d(\mathbf{y}, \mathbf{z}) = \left[\sum_{j=1}^{?} (y_j - z_j)^2 \right]^{\frac{1}{2}}$$

where $\mathbf{y} = [\, y_1, y_2, y_3, \ldots \,]$ and $\mathbf{z} = [\, z_1, z_2, z_3, \ldots \,]$; these may not correspond to any text item.

In theory, the distance measure $d(\mathbf{y},\mathbf{z})$ describes how dissimilar point **y** is from point **z** in our vector space. In the real world of the medieval monastic librarian, however, a measure of similarity is more familiar and more useful. The measure *d* for vector representations of text items is also more sensitive to their length than our medieval monastic librarian would like. A piece of text consisting of a section A joined to a section B could show high dissimilarity with text consisting of a section A alone.

Our vector space is also awkward for modeling searches with queries as typically formulated by a human. These tend to be just a few words, which would map to a tiny region around the origin of the space, which would be the zero vector $\mathbf{0} = [\, 0 , 0 , 0, \ldots \,]$. A library of text to be searched would would each have many more words than queries, so that their vectors would be scattered far outside this central query region. This would suggest that all text item vectors would have about the same dissimilarity with any query.

All of this makes a simple vector space a difficult model for text indexing. Salton and others realized this problem and tried to get around it by normalizing all vectors representing text items or queries. If **v** is such a vector, then its normalized form becomes

$$\hat{\mathbf{v}} = \frac{1}{d(\mathbf{v}, \mathbf{0})} \cdot \mathbf{v}$$

This diminishes the problem of the disparity of query and item vectors, but our theoretical space becomes unrecognizable. All our normalized vectors now become points on the surface of a hypersphere! This is fine and dandy, but what does it actually mean anything to us humans?

---

[10] Gerard Salton.

There is an advantage to a hypersphere model. Any two points on the surface of that hypersphere will be connected by arc of a great circle connecting them. The line segments connecting the endpoints of that arc with the center of the hypersphere will form an angle that can be taken as a measure of distance between the points. The cosine of that angle for a pair of points **y** and **z** can be directly computed as the inner product of the normalized vectors $\hat{\mathbf{y}}$ and $\hat{\mathbf{z}}$ for those points.

$$cos(\hat{\mathbf{y}}, \hat{\mathbf{z}}) = \sum_{j=1}^{?} \hat{y}_j \cdot \hat{z}_j$$

This cosine measure will always be between 0 and 1 because $\hat{\mathbf{y}}$ and $\hat{\mathbf{z}}$ will always have non-negative vector components. It will be 1 if $\hat{\mathbf{y}}$ and $\hat{\mathbf{z}}$ align, 0 if they are orthogonal. This makes the cosine measure convenient for gauging similarity two text items or a text item and a query. It actually seems to work fairly well for that purpose, but this is all highly abstract and divorced from any intuition. We have to wonder whether there might be some other way of measuring similarity.

9. Back to Basics

An inner product similarity measure for numerical vectors is quite plausible in theory. The problem is in interpreting the similarity scores back in the real world. The usefulness of a type of score will depend on how well a choice of vector dimensions captures content of interest to given users and how well the scores can be scaled to achieve the kind of ranking of matched items that users want. This is entirely separate from the underlying mathematics of how to compute scores.

We meet the same issues in the scoring for Olympic figure skating competitions. The index set there is a standard repertory of moves and transitions. The scaling is in the bonuses for the difficulty of given specific moves and the penalties assessed by judges for mistakes. This all then somehow combines into an overall numerical performance score from 0.0 to 10.0 for determine a winner. This may or may be consistent with what a watching arena crowd thinks or what other judges award.

As information system builders, we have to make users happy. Mathematical elegance is fine for papers in technical journals, but for the medieval monastic librarian and his modern progeny, a system must just be fast, compact, transparent, and practical. The cosine measure of text similarity has several major deficiencies: infinitely many index keys, inconsistent scaling of scores, and a strange hyperspherical model of text content alien to how humans think.

Let us first get rid of infinite index sets. If every word, name, or designation will be an index key, then we can never set a fixed upper limit on index size when dealing with dynamic collections of text like a news stream, a scientific preprint archive, or the latest trends on social media. The obvious alternative here is just not to limit ourselves to a reasonably small curated subset of index words augmented with a set of some other keys that is finite.

In particular, we would propose to index English text with a set of just several thousand words. For any word in a text item outside the several thousand, we break it up into overlapping fragments that will completely cover the word and then index these separate fragments instead. If we limit the length of fragments to some n, then our total index size will be finite, although by increasing n, we can ever more closely approximate indexing with an unlimited number of words.

In English, we can take 2-, 3-, 4-, and 5-letter fragments of words for the latter index keys. Our approach is first to look up a word in our selected whole-word index keys. If not found, we then will break up that unknown word and look up each of its pieces. Can such an approach actually work? It will definitely be noisier than whole-word indexing, but we can run simple experiments with various kinds of text data that with proper selection of indexing fragments, a discernible signal emerges from the noise.

Once we have a complete finite index set for all words or their parts, we can more reliably estimate the probability of an index key occurring in a text item we have never seen before. This lets us model the statistical distribution of inner product similarity scores expected by chance for a pair of text items or for a text item versus a given query or another item. The mean and the variance of that those similarity score distributions can then allow us to determine the statistical significance of a given score.

This approach gives rise to a new kind of scaling for inner product similarity scores. With the cosine measure, it is hard to set a minimum threshold on a good match. A cosine above 0.9 is almost always good, but often a cosine of only 0.4 is also a good match. This brings much uncertainty to applications like the automatic clustering of text items, which requires a similarity score between every pair of items, or like text stream filtering when we want the similarity of an item against perhaps thousands of standing queries.

A large system relying on a cosine measure of text similarity is in effect running out of control. With statistically scaled match scores, managers can see what is going on and maybe decide to adjust an index set to improve performance. In the filtering streams of text, they can monitor the match scores for individual standing queries to see whether they are the expected ranges. If not, then alarms should be raised to alert everyone to unexpected major changes in newly arriving content.

At this point, someone may react with more than a bit of skepticism. This all sounds too good to be true. The kind of hybrid indexing described here, however, is firmly grounded on our knowledge of language and of English in particular, plus the application of some basic statistics. The concept of statistically scaled similarity also can be easily checked out by potential users on their own text data. It offers no magical solution, but within its limitations, we have expanses of clear ice to attempt quadruple axels.

10. Fragmentary Indexing

The American English alphabet contains 26 letters familiar to most preschoolers. There are 26 × 26 = 676 distinct pairs of letters: `AA`, `AB`, `AC`, `AD`, ..., `ZW`, `ZY`, `ZZ`. With these as keys, we can in theory index any English word. For example, the word "AARDVARK," which is unlikely to be selected as key for an index set, could be analyzed as follows

```
AARDVARK
AA
 AR
  RD
   DV
    VA
     AR
      RK
```

So, we represent the word "AARDVARK" with seven distinct 2-letter fragments, with `AR` showing up twice. This actually is fairly decent indexing in that finding an occurrence of `AA` is likely to indicate a text item containing "AARDVARK." We did not have to look for overlapping fragments in a word, but this will capture the maximum information conveyable with just 2-letter index keys. It also is a cleaner way to index words with an odd number of letters.

Still, our intuition is that 2-letter fragments are inadequate for the kind of indexing we want to supportP. With only 676 distinct keys, we would have an entropy of only 9 bits under the best circumstances. Given the number of rare 2-letter fragments like `QK`, `JJ`, `LH`, `PK`, or `ZW`, the entropy will probably be much lower than 9 bits. Nevertheless, the addition of just 676 2-letter fragments to any finite set of full-keys in an index set will be enough to achieve full coverage of all words in any text item, though in a noisy way.

One way to reduce the noise is use 3-letter fragments as index keys, but there are 17,576 distinct triples of 26 letters. We ideally want to have any index set, hybrid or otherwise, to have only about 10^4 keys for simpler processing. Most 3-letter combinations in any event will be extremely rare like `QXJ` or `LLL`, which will contribute little to indexing. We are better off selecting a subset of 3-letter fragments as keys and then combining then with all 2-letter fragments to achieve coverage of all words.

One quick way of choosing useful 3-letter fragments is to find about 200 of the most frequent 2-letter combinations in English text and append one other letter after them. This would result in 200 × 26 = 5,200 3-letter fragments plus 676 2-letter fragments for an index set along with some user-selected fragments. We would allow occurrences of 3-letter fragments to overlap in a word as done above with 2-letter fragments. This is again to maximize the information from indexing. (See Appendix A.)

Suppose we have the word "REMDESIVIR," the generic name of an antiviral drug that someone might want to search for, but is unavailable (yet) in some index set. Most of the 2-letter fragments are frequent enough to be extended to 3-letters, which might result in the following analysis:

```
REMDESIVIR
REM
 EMD
   DES
    ESI
     SIV
      IVI
        IR
```

The rule is that no fragment is recognized if it occurs entirely within a longer indexing fragment. The list of 2-letter fragments to be extended by one letter are taken from the classic cryptanalysis book of Gaines.[11] A list based on 21st Century medical text in English would probably have `VIR` instead of just `IR`.

This is just the beginning of what can be done to improve the quality of text indexing. We can go to 4-letter and 5-letter fragments, which we could identify from cryptanalysis resources. This means that an index set will have relatively few of them at most, but this can still help to reduce the noise in indexing with word fragments. Zipf's Law is on our side here; only a few longer fragments will be frequent enough to bother with. Full words and 3-letter fragments will still pull the most weight in indexing.

In particular, we have 456,976 distinct 4-letter sequences in a 26 letter alphabet. Putting them all into an index set is highly inefficient because almost all of them will be nonsense like QQQQ, IIOO, or KZLR. These will force us to work with index vectors of huge dimensionality without much benefit. Yet, we certainly can improve our indexing of English text with selected 4-letter fragments like ZZLE, LIEN, or NTRO. We just have to use our knowledge of English choose carefully.

The case of LIEN shows what we have to contend with here. LIEN is actually a full word, but it is a legal term rarely showing up in everyday discussion. As an indexing fragment, however, it be found in many places: a**LIEN**, resi**LIEN**ce, c**LIEN**t, sa**LIEN**t, and the Spanish word ca**LIEN**te, often seen in Southwestern place names. The fragment is more useful as an indexing feature than the full word.

Indexing with a word fragment like LIEN is still noisy, but less so than indexing with only 2- and 3-grams. Adding LIEN to an index set also makes the shorter fragments LI, IE, EN, LIE, and IEN less noisy because of not counting occurrences of fragments in text when they are completely contained in a longer indexing fragment. We cannot eliminate all noise in this way, but longer indexing fragments will improve our overall indexing as long they actually in target text.

We should improve ilndexing entropy with every longer word fragment added to an index set. It will take about a hundred 4-letter fragmens, however, to raise entropy by 0.01 bits, which calls for some substantial work to find enough suitable candidate indices out of 456,976 possibilities. At present, we are up to over 2,000 selected 4-letter fragments for text indexing, comparable to the numbers of 2-character and 3-letter indexing fragments. The number of 4-letter fragments will probably keep increasing

Indexing with word fragments of 5 letters is more challenging. The problem is that there are almost 12 million possibe 5-letter sequences alone, making the choice of a subset for indexing more difficult. Yet longer fragments will be less frequent in text that shorter ones, and so it will take many more of them to affect overall indexing entropy. It can be helpful to have a few hundred of them for indexing text, but it looks as if shorter fragments will have to pull the most weight in any index set.

[11] Gaines, Cryptanalysis.1939.

11. Modeling and Validating

With an index set X of M keys for a collection of text items, we can map each item to an M-dimensional numerical vector and can compute an inner product similarity measure $S(d,e)$ between any pair of items d and e. When $S(d,e) > S(d,e')$, we can say that item d is farther away from e than from e'. Without some kind of scaling, however, we cannot really interpret $S(d,e) > S(d',e')$ for four distinct items d, e, d', and e'. The cosine measure attempts such a scaling, but does so poorly.

When the probability of each key x_j in X can be estimated within a text collection, there is another option. First, we can make a simplifying assumption that those probabilities are more or less independent and use a multinomial statistical model to estimate the expected value $E[s(d,e)]$ and variance $Var[s(d,e)]$ of the inner products between random pairs of vectors for items d and e. (See the Appendix for the detailed mathematics of how all of this comes about.)

A multinomial model here will not be exact for text indexing because of our simplifying assumptions. Nevertheless, this can still be a helpful approximation for gauging the statistical significance of similarity measures. The idea is to define a new statistically scaled similarity score

$$\bar{s}(d,e) = \frac{s(d,e) - E[s(d,e)]}{\left[Var[s(d,e)]\right]^{1/2}}$$

We can interpret $\bar{s}(d,e)$ as the number of standard deviations (σ) that a given score exceeds a score expected by chance. If $s(d,e)$ scores had a normal distribution, 3σ would correspond to a statistical significance of $p \approx .013$, a bit more than one chance in a hundred of happening by chance. This is enough to get a paper published in a psychology journal, but when searching a collection of a million text items, that would mean about 13,000 possible matches. In most text searching, we might want a score of 4σ or more, although when matched items are ranked by match, a match threshold affects only search speed.

A bigger issue here is whether statistically scaled similarity makes any sense at all in practice. Our particular choice of indexing has to capture sufficient information about the content of our target text items and our similarity measure for items has to be fine-grained enough to distinguish between different items in a collection. Neither condition can be taken for granted; and so we need a standard framework for validating whatever indexing setup we decide to work with.

Historically, indexing has been evaluated by running retrieval experiments with a large set of test queries against items in a target collection that has been previously marked up to indicate the relevance of each item to each test query. The markup requires subjective human judgment and can be costly even for fairly small target collections. The resulting validation procedure of course may not satisfy some information seekers, who might want more specific testing of their own indexing on their own kinds of text data.

The validation problem is unavoidable for any indexing with word fragments because this approach will face much more skepticism than indexing with whole words. So, it was necessary to establish a kind of testing in which explicit human judgments of relevance could be entirely avoided. This had to be easy to set up for any subset of text data and had to provide quick results easily interpretable even by people without special technical expertise.

For finite statistical indexing with set X, such validation can be done as follows:

a. Select a target collection of text. This should be large enough to have occurrences of most keys in X.

b. Compute the probability p_j of each x_j in X within the target collection.

c. Compute the entropy of X. This should be at least half of the maximum entropy of M bits, where M is the size of X.

d. The maximum probability of any index key in X should be less than 0.01.

e. Randomly select a subset T of $K \geq 100$ target text items for testing. These should be at least 2,000 characters long.

f. Divide each selected test item t_i in T into 2 roughly equal parts, t_i^a and t_i^b. The division should be on a sentence boundary.

g. For t_i^a and t_i^b, take the first $L >= 100$ index key occurrences and create corresponding vectors v_i^a and v_i^b.

h. Compute raw inner product similarity scores for every unique pair of vectors v_m^a and v_n^a. This will be the noise distribution. It should plot with one peak with a long tail to the right.

i. Compare the mean and variance of the noise distribution with predictions of the multinomial model.

j. Compute raw inner product similarity scores for every pair of vectors v_m^a and v_m^b. This will be the signal distribution.

k. Plot the noise and signal distributions and verify that there is a clear separation.

l. Repeat as required for different K and L.

The critical test is step k. The noise and the signal distributions will always overlap, but this should be mostly in their tails. It may well be that a high match score in the noise distribution may actually be for text items relevant to each other by chance. Similarly, the second half of an item may not be relevant to the first half. Such anomalies should usually be rare, however.

12. Experimental Results

The original tests of fragmentary indexing were back before the Internet made a wealth of text data available by download. Three small text collections were readily at hand then: some old wire service news stories, selected readings from an anthology for college freshman English, and abstracts on solid-state physics and electronics,. These have been lost over the past forty years, but anyone should easily be able to follow the procedures of the last section to get similar results with more recent data.

Our experiments with the three data sets used one index set of all 2-letter fragments plus selected 3-letter fragments. To maximize the information value of English fragments like “THE,” “ING,” or “ION” here, common grammatical words were dropped in test data and inflectional and morphological suffixes of words were removed as much as possible. This required extensive preprocessing of test data, but otherwise, word fragments are probably too noisy for proper indexing of text content.

The first test for our proposed word-fragment indexing is to check whether a multinomial model based on fragment probabilities is a reasonable description to the distributions of similarity scores actually seen in our test data sets. For each set of 100 items, we computed the 100×99/2 scores for all unique pairs of the first vectors obtained for each item as described in the preceding section. Each of these vectors had 100 index key occurrences.

First, we computed inner product scores from computed from actual frequencies. For our three test collections, here are the statistics for scores between 495 vector pairs versus model predictions:

2- and 3-Letter Fragment Indexing Experiment - Raw Counts				
	E[S]	μ_{sample}	$Var[S]^{1/2}$	σ_{sample}
News	16.2	14.6	4.5	8.5
Readings	14.4	13.3	4.2	6.4
Abstracts	23.3	22.3	5.8	12.5

The inner product similarities are a little lower than expected, but the variance is quite higher than expected. Nevertheless, we are in the ballpark can see some possibilities here. John Tukey, who wrote the standard reference book on descriptive statistics[12], recommends the trick of transforming raw counts from text data into their square-roots. This will make index keys with many occurrences in a text item less likely to dominate computations like an inner product. It gives us the following performance statistics.

2- and 3-Letter Fragment Indexing Experiment - Square-Roots				
	E[S]	μ_{sample}	$Var[S]^{1/2}$	σ_{sample}
News	14.7	13.7	4.5	6.2
Readings	13.6	12.9	4.0	4.6
Abstracts	20.3	19.6	5.2	7.9

A multinomial model looks better here, although actual variance is still higher than expected for all three sets of test data. On the whole, we decided to follow Tukey’s recommendation. We tried fourth-roots also, but the results were not much better than for square-roots.

[12] John Tukey, Exploratory Data Analysis.

2- and 3-Letter Fragment Indexing Experiment - Fourth-Roots				
	E[S]	μ_{sample}	$Var[S]^{1/2}$	σ_{sample}
News	14.2	13.3	4.1	5.5
Readings	13.2	12.5	3.9	4.2
Abstracts	19.1	18.7	4.9	6.7

We can do a more objective statistical assessment of the match between model predictions E[S] and Var[S] with actual distributions of similarity scores (see Appendix C for details). This will use Student's t-test to estimate the probability *p* that that the means and standard deviations of similarity scores for subsamples of test item pairs are outside of model predictions.

Significance Tests for Model Estimates With 2- and 3-Letter Fragments			
		p	
		μ	σ
	raw	0.001	0.001
News	square-root	0.001	0.001
	fourth-root	0.001	0.001
	raw	0.001	0.001
Readings	square-root	0.001	0.001
	fourth-root	0.001	0.106
	raw	0.018	0.001
Abstracts	square-root	0.050	0.001
	fourth-root	0.080	0.001

This supports using square-roots of index key frequencies for our item frequencies. The multinomial model has to be seen only as an approximation of similarity scores between items that can be applied to the engineering of practical text information systems. We also got similar results on trials with vectors based on 50 or 200 index key occurrences. Information users should easily be able experiment with their own data and their divisions of text into vectors.

Fragmentary indexing also notable in that it holds up fairly well across diverse types of text. We humans understand that reading a news story often requires different skills than reading an essay or a description of electron mobility in a new semiconducting material. Yet, many systems operate on the premise that words are words in all text and can always be attacked in the same way.

13. Signal Versus Noise

A multinomial distribution provides a reasonable estimate of the mean and standard deviation for inner product similarity scores computed for random pairs of text items. This will define a noise distribution. Similarity scores computed for related pairs of text items will define a signal distribution. If an indexing scheme is effective, then we should be able to tell the two distributions apart. Some overlap is inevitable, but the means for the two distributions should at least be well separated.

The problem with getting signal distribution is in finding related text items to generate it. Historically, this has involved subjective judgments of one item being relevant to another, which is time-consuming and expensive. It has limited the number and extent of standard benchmarks for information retrieval. How many times did the Cranfield collection[13] and its couple dozen of test queries with identified target documents constitute the framework of yet another research project?

In the Internet Age of terabytes of text data accessible in clouds, information scientists need to test their technology much more broadly and more narrowly at the same time. This was the motivation for taking text segments from the same document as an objective way to gather an unlimited number of examples of relevance for testing purposes. One only has to have plenty of documents long enough to extract text segments or some sort of threading to connect short items like email messages.

For just testing the viability of word fragment indexing, we can get by with only a small amount of text data, but that text still had to be fairly diverse. So, our three test collections consisted of 100 segments of news stories, 100 segments of literary readings, and 100 segments of technical abstracts, plus three sets of 100 related segments for each of the three collections. These are the results:

Similarity Scores for Unrelated and Related Pairs With 2- and 3-Letter Fragments

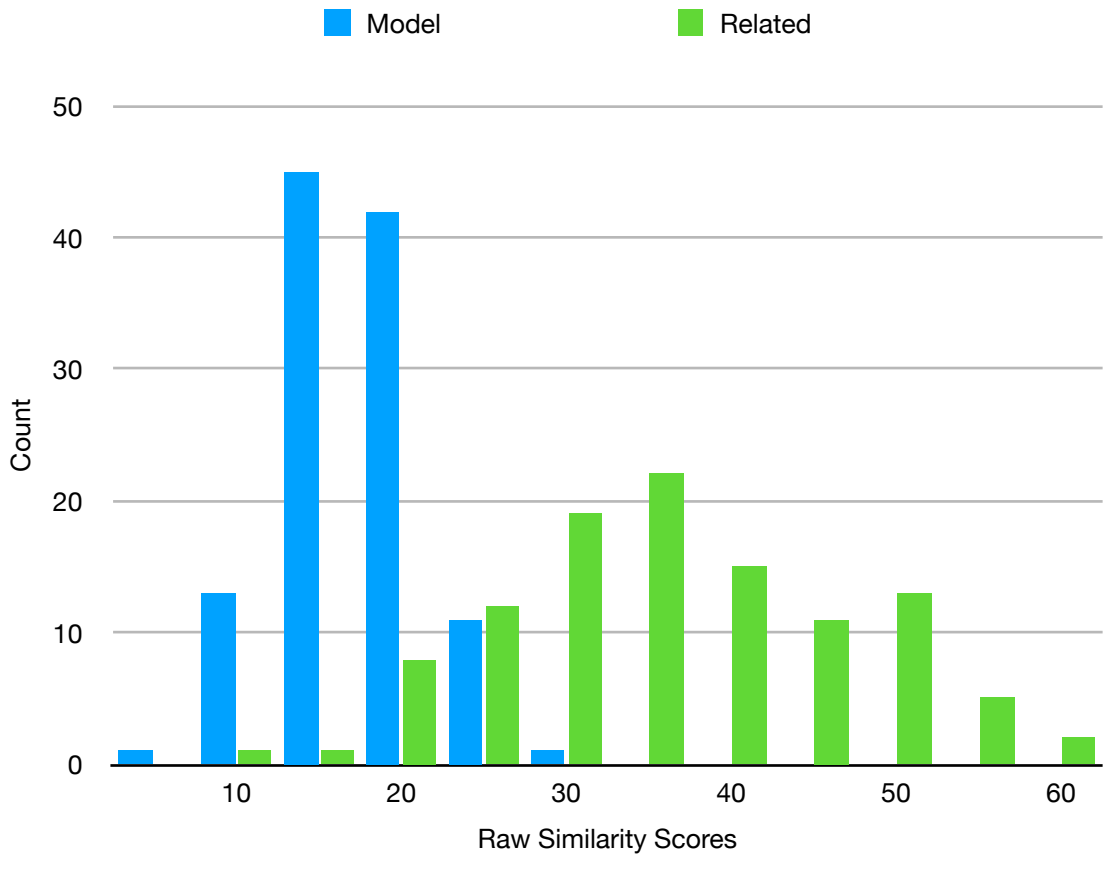
Collection		495 Unrelated Pairs		100 Related Pairs	
		μ	σ	μ	σ
	raw	16.2	4.5	42.6	16.3
News	square-root	14.7	4.2	34.1	10.6
	fourth-root	14.2	4.1	30.7	9.2
	raw	14.4	4.2	33.6	22.4
Readings	square-root	13.6	4.0	25.6	11.5
	fourth-root	13.2	3.9	23.1	8.8
	raw	23.3	5.8	54.5	21.2
Abstracts	square-root	20.3	5.2	44.0	12.7
	fourth-root	19.1	4.9	39.2	10.2

The literary readings show the least signal-versus-noise separation, but even so, the separation between the means of the noise and signal distributions are separated by more than 3 standard deviations of the noise distribution. This is a strong signal.

We can plot the noise and signal distributions of the news collection to get a picture of what is going on overall. The blue bars for noise are 100 similarity scores as predicted by a multinomial model for the News collection; the green for signal are actual scores for 100 related items for News. Scores range from 0 to 60, divided into 12 bins, with counts in each bin reported as blue and green bars for noise and signal.

---

[13] https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.88.9319&rep=rep1&type=pdf



News — 2- and 3-Letter Fragment Indexing Square-Roots of Index Key Frequencies — Unity Weighting		
	μ	σ
Noise (Model Predictions)	14.7	4.2
Signal (Related Pairs)	34.1	10.6

As can be seen, the noise and signal distributions overlap, but they are quite different. When target text items have to be found among a huge number of irrelevant items, the noise distribution will fatten and can spill into the signal distribution. This suggests possible problems if we try to index large collections with only 2- and 3-letter word fragments, but everyone should note that:

1. We can always improve our indexing by increasing the number of 3-letter fragments in our set of index keys. As of 2002, the number of frequent 2-letter fragments extended to get 3-letter fragments has increased from 200 to 216.
2. Our approach will actually be to use thousands of selected whole word keys as the starting point for text indexing. We will resort to word fragment keys only when a whole word is missing from our index set. This limits the extent to which indexing noise can arise from word fragment keys.
3. Weighting of index keys in an inner product similarity measure can sharpen our ability to distinguish between different kinds of text content. Up to now, the weights a_j in our computation of $S(d,e)$ for text items d and e have all been set to 1.

The bottom line, though, is that word fragment indexing conveys enough information to be workable.

14. Finite Indexing

Our experiments so far allow us to define various finite index sets to describe the content of any text item completely. Each such index set will consist of a limited number of full word keys selected for particular target text plus a limited number of short word fragments to back up the full-word keys. Whenever a word in a text item is not a selected key, we turn to its various fragments instead. This will be noisy in that an item could contain all the fragments of a word without containing the word itself.

Such noise is unavoidable, but we have seen that there is a real signal within indexing by word fragments. So, the challenge is to figure out how to control the noise as much as possible. The first step of course is just to recognize the sources of noise and deal with them systematically. That is the fundamental problem of all statistical analysis, even when the data has no text at all in it. There is probably no perfect solution here, but we can do better than just randomly slamming numbers together in hope of success.

The big shortcoming of indexing approaches like the vector spaces and the cosine measure of distance of text similarity is that of infinite dimensionality. In any living language, new words appear all the time. The Internet is especially fertile ground: LOL, WOOT, ransomeware, vlog, LGBTQ, and on and on. A finite set of keys gives us a better theoretical handle on the problem of text indexing at the cost of added noise, but this trade-off opens the way to practical engineering solutions.

To summarize, we will be working a finite index set X of M keys of hybrid provenance. From there, we can define an M-dimensional numerical vector $[f_{i1}, f_{i2}, ..., f_{iM}]$ for each of the N text items in a collection of interest, where f_{ij} = the square-root of the frequency of the jth index key in the ith item. The total frequency F_j of index key x_j in X can be computed as

$$F_j = \begin{cases} \sum_{i=1}^{N} f_{ij} & , if\ this\ sum > 0 \\ 0.5 & , otherwise \end{cases}$$

and then compute the probability p_j for each index key x_j in X as

$$p_j = \frac{F_j}{\sum_{j=1}^{M} F_j} > 0$$

This is impossible to compute with an index set of infinite size.

Our inner product similarity measure will be defined as before

$$S(d,e) = \sum_{j=1}^{M} a_j \cdot f_{dj} f_{ej}$$

and we use our probabilities p_j derive a multinomial model to estimate $E[S(d,e)]$ and $Var[S(d,e)]$ using

$$N_d = \sum_{j=1}^{M} f_{dj}$$

$$N_e = \sum_{j=1}^{M} f_{ej}$$

in the formulas of Appendix B.

Our similarity measure will then be scaled statistically.

$$\bar{S}(d,e) = \frac{S(d,e) - E[S(d,e)]}{Var[S(d,e)]^{1/2}}$$

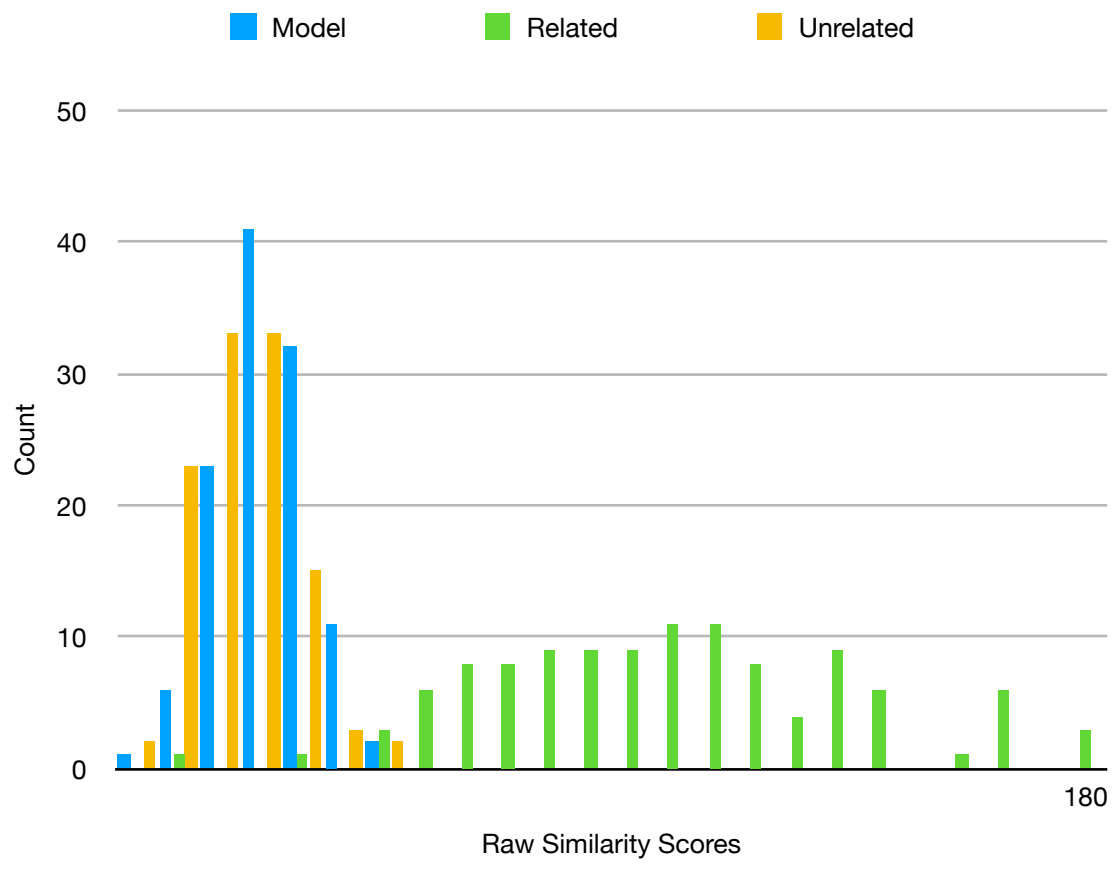
Our scaled similarity is in units of standard deviations (σ), which allows us to set minimum thresholds on similarity for any significant match. We can use the familiar normal distribution as a rough guide here: we want a scaled similarity of at least 3σ for a match, but would probably prefer 4σ to be safer. The cosine measure never shows up anywhere.

Working with a finite set of index keys makes it easier to tackle many basic problems of configuring a system for better performance. An example right in front of us is with setting the weights a_j built into our inner product similarity measure. Up to now, we have been setting $a_j = 1$, but intuition tells us that a matchup of rare index keys in a pair of text items is more important than a matchup of common index keys. With finite indexing, we can define various a_j based on p_j. and run experiments to validate them.

Our current favorite weighting is fairly simple, assuming only that $p_j > 0$ for all j.

$$a_j = \left(\frac{1}{p_j}\right)^{1/2}$$

Here is what happens when we rerun the signal-versus-noise test with 2- and 3-letter fragments indexing the News collection.



News — 2- and 3-Letter Fragment Indexing Square-Roots of Index Key Frequencies — $(1/p)^{1/2}$ Weighting		
	μ	σ
Noise (Model Predictions)	29.6	8.6
Noise (Unrelated Pairs)	24.7	10.6
Signal (Related Pairs)	92.3	33.7

The weights a_j have the net effect of raising the similarity scores for related pairs more than those for unrelated pairs. This should be evident in a visual comparison of the bar chart just above with the one in the previous section. There is a wider separation of signal versus noise. It is an open research question whether someone can do better than this with a different frequency transformation or a different weighting of index keys.

15. Searching

Searching for text information typically involves by formulating a query of some kind to be compared against a large number of text items. The hypersphere cosine similarity model of information retrieval treats a query as if it was another text item, mapping it to a point on the surface of a hypersphere. One then can compute n-dimensional great-circle cosine distances from the query point to other points representing text items that are candidates for retrieval.

Although this conceptual view of indexed searching has been employed widely for a half century, there is more than one way to skin the proverbial cat. The statistical model of text similarity as described here provides a cleaner and simpler alternative. We look at the nitty-gritty of text indexing as might be understood by a medieval monastic librarian, but adding the insight of modern statistical thinking. This will lead us to realize that a query often should NOT be just another text item.

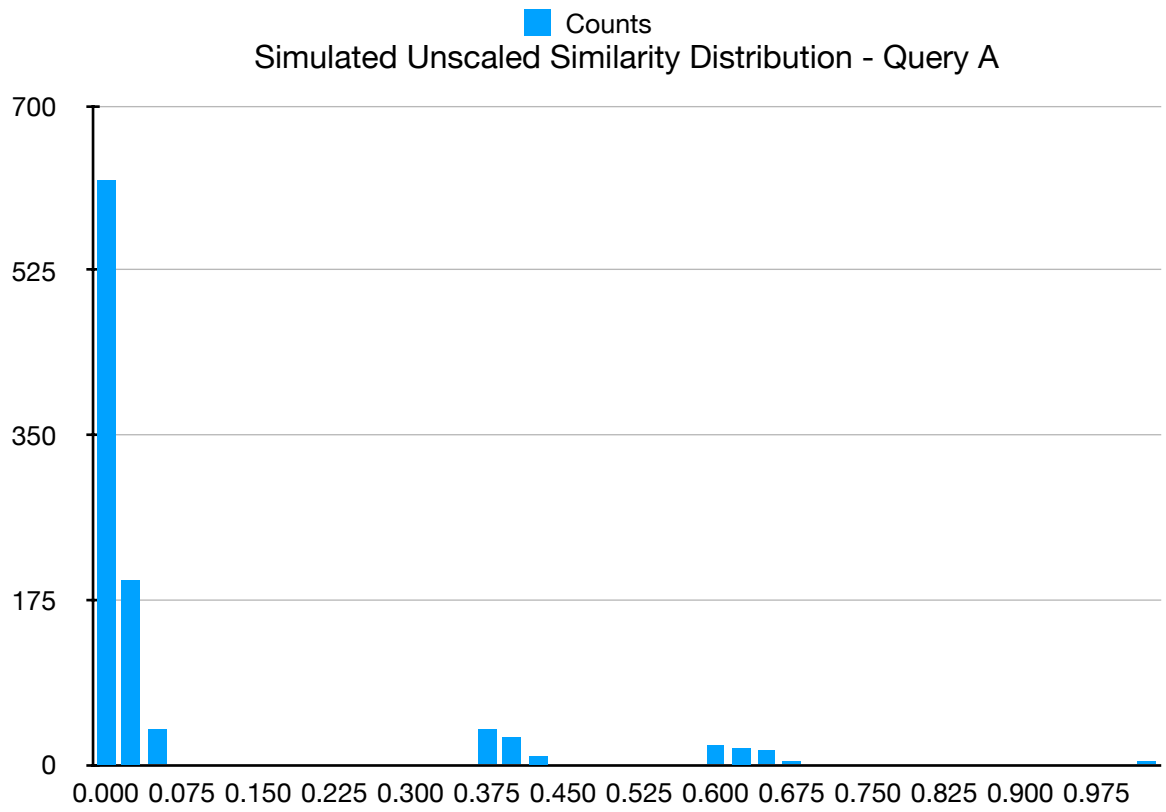
To scale inner product similarity measures for pairs of text items, one needs to know how likely a given numerical score could arise by chance given the length of both items. For a query versus a text item, however, we almost never care about the distribution of scores of all possible queries and all possible text items. That might be helpful information for a search engine designer, but seekers of answers to a question will be concerned more with the likelihood of a given score versus a specific fixed query.

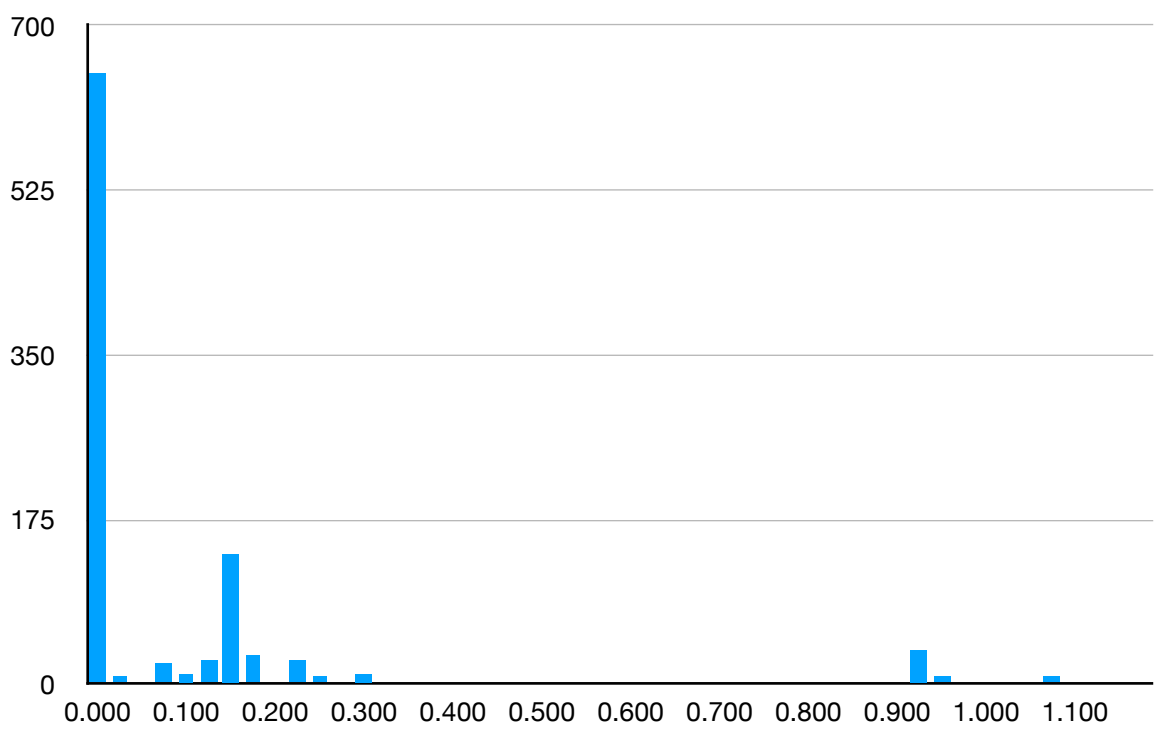
Our multinomial model of inner product similarity with fixed, finite indexing gives us a prediction for the query case also. It is different from from that for an arbitrary pair of text items. Statistically speaking, queries involve fewer degrees of freedom. This will simplify the multinomial model for $E[S_q(d)]$ and $Var[S_q(d]$. for a query q versus an arbitrary text item d (see Appendix C). The details are more nuanced than in models involving the surface of a hypersphere in a high-dimensional space.

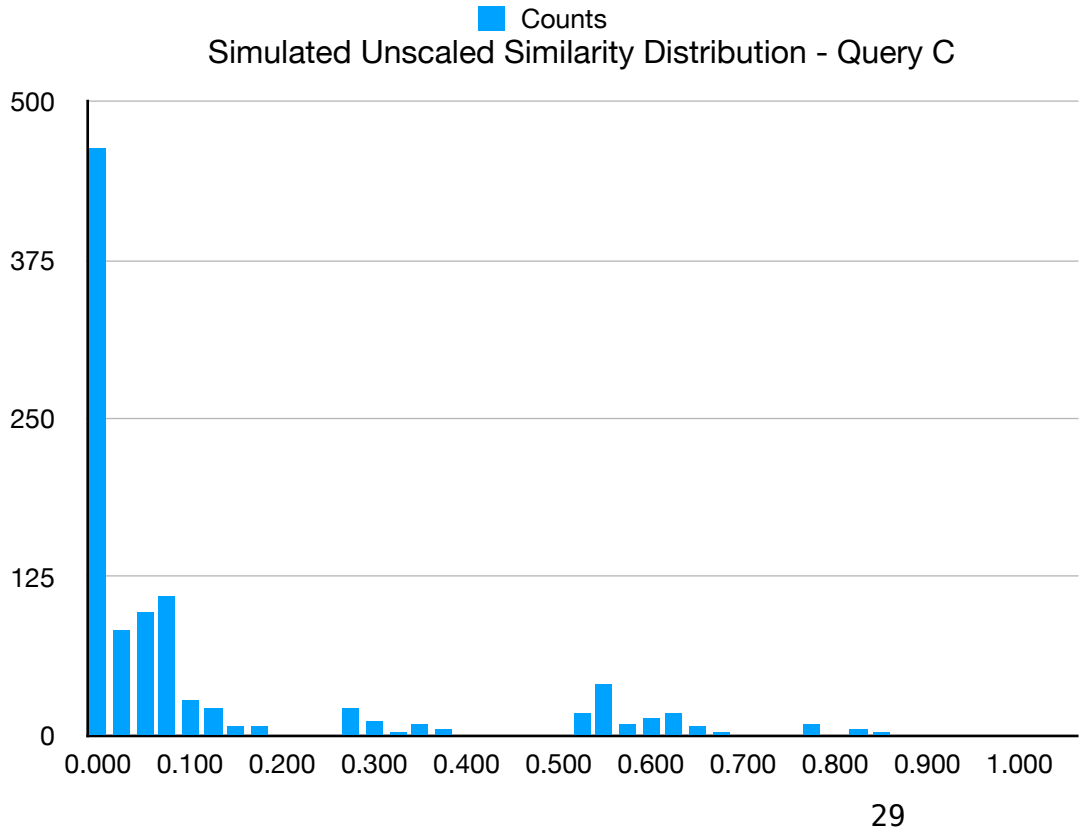
Statistically scaled similarity is advantageous when a stream of incoming text has to be matched against a large array of standing queries. With a single query with unscaled similarity, a searcher always can read a few of the top n best matches for the query. That chore becomes unmanageable when many thousands of standing queries need to be managed. Statistically scaled similarity by themselves can usually let us assess the success of matching without having to read through the matches collected by every standing query.

The distribution of scores of text items versus a query will usually be shaped nothing like the clean unimodal plots associated with a multinomial model. This is because queries tend to be quite short when compared to a typical text item. Consequently, inner products of vectors for such queries will interact only with only narrow projections of item vectors, allowing fewer degrees of freedom in scoring. The distributions of the similarity scores for different short queries will vary greatly.

We can build a simple statistical model to show how queries work. Start with an index set X of size M with keys x_j having probabilities p_j as in Zipf's Law. It is then straightforward to simulate an arbitrary query q to be matched against a large set of randomly generated item vectors with square-root frequencies. If we do this several times and plot the resulting similarity scores separately for each query, we can get something different each time, although these will may resemble each other somewhat. Here are three examples.


Counts
Simulated Unscaled Similarity Distribution - Query A
700
525
350
175
0
0.000 0.075 0.150 0.225 0.300 0.375 0.450 0.525 0.600 0.675 0.750 0.825 0.900 0.975


Counts
Simulated Unscaled Similarity Distribution - Query B
700
525
350
175
0
0.000 0.100 0.200 0.300 0.400 0.500 0.600 0.700 0.800 0.900 1.000 1.100


Counts
Simulated Unscaled Similarity Distribution - Query C
500
375
250
125
0
0.000 0.100 0.200 0.300 0.400 0.500 0.600 0.700 0.800 0.900 1.000

A short query and a fairly short text item will often have a low similarity score. The three charts above all show a high number of such scores in their first bin. The lowest range of similarity scores for a query should look like a distinct noise distribution, though it might be squeezed by the scale of its bin axis. We might also see one or more peaks of higher scores as a kind of signal distribution, but a query could quite well have no high-scoring matches. Here is a summary of the data for our query examples.

Query	μ	σ	Plotted	Unplotted	High Score	Scaled
A	0.0895	0.1870	1000	0	1.0445	5.11σ
B	0.1291	0.2088	1000	0	1.0548	4.43σ
C	0.1169	0.2467	996	4	1.3379	4.95σ

The highest scoring items for each query will usually be far enough above the mean of all scores to count as outliers. In a more thorough analysis, we would probably want to work the means and standard deviations of scores without the outliers. This will give us a better estimate of the noise for given query.

On the whole, the takeaway here is that any index set with keys having probabilities that follow Zipf's Law should adequately support searches of text collections when there are enough keys. The keys need to carry enough information as measured by their entropy to achieve the degree of discrimination required for retrieving relevant items for a query. It should not matter whether keys are whole words, word fragments, or some combination of the two.

16. Clustering

Many people think that text information systems are synonymous with search engines. This is the influence of tools like Google Search, which have turned the entire Internet into everyone's personal resource. Such broad access to information is unimaginable to a medieval monastic librarian. Nevertheless, there is a catch: information seekers must first ask the right question. If this is off the mark, then all the server farms and data warehouses in the world will be for nought.

To make a quest for information less of a guessing game, a helpful system can provide users with a content map of its collections of text. In a physical library, such a map might show how its shelves are laid out and where particular subjects can be found together. In a virtual library, a text item can be shelved in many different spots, making maps even more useful, though harder to create and maintain. Automatic clustering techniques can help here, and these can be made more effective by how text items are indexed.

One advantage of a finite indexing scheme is that the similarity between a pair of text items can be statistically scaled. With similarity scoring like the cosine measure, when $cos(A,X) > cos(A,Y)$, we might expect that X is more similar to A than Y is, but what $cos(A,X) > cos(B,Y)$ means is unclear. This will depend on the actual lengths of A. B, X, and Y, which are normalized away in the cosine measure, and also on the particular indices these items have in common.

The cosine measure has no clean interpretation for clustering. For example, consider a cosine similarity score of 0.33, which corresponds to a hyperspace angle of about 19.4°. Is this significant? The answer is that it will depend. A 0.33 score is less than half of the maximum cosine similarity score of 1.0, but when you run many searches with various query vectors, the top item on a ranked retrieval can often have a score of 0.33 and be relevant.

Statistically scaling of similarity is imperfect because of the imprecision of calculating the expected value and the variance of inner product scores expected by chance for pairs of item vectors. Yet our experiments show that we can get close enough for thresholds of 4σ or higher so that our scaled similarity can be reasonably reliable for automatically clustering large sets of text items. That is, $\bar{s}(A, X) > \bar{s}(B, Y)$ has a consistent meaning for all items A. B, X, and Y.

Clustering is computationally intensive. For a set of N text items, we have to calculate $N \bullet (N\text{-}1)/2$ scaled similarity scores. This means that we probably want to avoid trying to process a million items at one time. We get around this problem by clustering in batches of around a thousand items each. This will be done in two stages: (1) find compact highly connected cluster seeds and construct indexing profiles for them; (2) assign text items to clusters by matching a seed profile at above a minimum threshold.

A thousand items should produce about a hundred cluster seeds. About two-thirds of the original items will match at least one of the seed profiles; the rest go onto a residual list. When the next batch is processed, its items are first compared to current seed profiles and either assigned to existing clusters or appended to the list of residuals. Then the new residuals are analyzed to get new cluster seeds and seed profiles; yet unclustered items will be assigned to the new clusters or to a new residual list.

There are many algorithms used by data scientists to do clustering. They have names like minimum spanning tree, mutual nearest neighbor, k-means, DBSCAN, and so forth. These are designed to overcome problems arising from poorly scaled similarity measures or from the need to find clusters that are non-compact. With text data and proper indexing, these issues become less important. Since we do have statistical scaling of similarity, compact clustering is usually what we want.

So, we shall define a simple two-step clustering adjustable to various text data applications. We first want find a set of cluster seeds, each being a small compact group of mutually connected text items; these seeds may cover only a small subset of a text collection. We then build a query based on each cluster, and these all can then run with a minimum match thresholds in searches of the entire collection. An item may match more than one query, but its match score will usually be higher for some queries than for others.

To get starting cluster seeds, it is convenient to use a hierarchical clustering method. This in effect generates a multilevel tree where each subtree corresponds to possible grouping of text items in a collection at some similarity measure threshold. If a set of items is not mutually connected enough, then we can move further down in that subtree and split up the set by raising the threshold there. The exact procedure can vary, but if seed groupings are not compact, profiles built for them can be problematic.

17. Serial Grouping of Text Segments

Humans have a built-in sense of time. When we observe events around us, we try to make sense of them by ordering them into sequences of one thing after another. From this comes our stories. As their telling become more complex, we usually try to group single events into larger encompassing events and thereby construct a mosaic experience of reality.

Written text really involves a kind of story telling. Data scientists often think of text as an undifferentiated mass of characters waiting for slicing and dicing, but it is ultimately a linear phenomenon that any reader must get hold of as a first step toward understanding what is written. As it turns out, the indexing of text can be a helpful tool in bringing at least part of such understanding into the realm of computation.

The statistician John Tukey[14] in a short paper proposed a simple statistical clustering method for points on a line, which might be interpreted as events in time. This computed the interval between consecutive point and obtained their statistical distribution. If any interval is more than 2.3 standard deviations above the mean of this distribution, then it is a significant gap. All consecutive points without a gap between them can then be grouped together.

We can apply a similar procedure to text, where points are paragraphs in a document and where the interval between paragraphs is the inverse of the scaled similarity score between those paragraphs. If they are extremely short, we may want to several consecutive paragraphs into one unit. In any event, a grouping by gaps will give us a way to divide up a long document into shorter coherent sections for a more precise analysis of its content. This can be helpful in detect text items that actually news roundups.

Gapping also can be the basis for generating summaries automatically for a long document. In news stories especially, the first sentence of a paragraph typically serves as a topic sentence. If we take the topic sentence of the first paragraph of every gapped grouping, then we can roughly approximate a summary for the content of the entire document. This idea needs to be explored further, but seems quite promising.

[14] John Turkey

18. Filtering Text Streams

Tailored news streams are big business on the Internet. So much real-time information nowadays is available that users are willing to pay someone else to filter and organize what shows up their online inboxes. This is typically done by developing one or more detailed profiles of a person's interests, which then can be compared against each item arriving on one or more data feeds. Profiles can run in series or in parallel, and a system must make sure that final results are neither too much nor too little.

Filtering a text data stream differs from searching a database. In the latter, an information seeker works interactively, able to adjust a query in response what is being retrieved by it. In filtering, user profiles typically run unattended for long periods, and a system has to be much more precise in its matching to strike the proper balance for each information user. The ability to control every single profile closely is critical to successful system operation.

Profile matching with statistically scaled similarity brings many big advantages for real-time filtering. First, each user can set reliable minimum significance thresholds to hold down noisy matches. This is hard with a cosine similarity measure, which really can only rank matches. When running on a real-time text stream, a system has to decide almost immediately whether to pass on a new item; it cannot wait to see if something better might come along. Second, if an item does match at a high level of significance like 8 or 9 standard deviations, then a system can notify a user right away.

From a management standpoint, statistically scaled similarity also fits naturally into the statistical process control idea of W. Edwards Deming[15]. This is normally employed in manufacturing assembly lines, but can be applied to multi-stage filtering of text streams. The goal is keep a process under control, running as predictably as possible. In filtering, the match scores with a profile should stay within lower and upper bounds reflecting a noise model of a profile. A system will sound a warning when they fail to do so.

In manufacturing, warnings might mean that a tool is out of alignment or is worn and needs replacing. In text filtering, they probably mean instead that the content of an input stream has changed significantly and that noise models need to be updated for every profile. The recalculation can be carried out quickly, but the trick is to know when to do it. Any drift in the probability of indices is likely to be gradual, even with major changes in stream content. It is always essential to keep watching how unattended profiles are performing.

Statistical process control was actually built into an operational system for a commercial news aggregator a few decades ago. The software ran for a number of years, but difficulties in migrating it to more advanced platforms forced its abandonment. The effectiveness of statistical process control in text filtering remains to be tested in a thorough way, but from a systems management perspective, any large-scale processing of streamed text really needs some kind of systematic quality control.

---

[15] Deming, W. Edwards, Lectures on statistical control of quality., Nippon Kagaku Gijutsu Remmei, 1950

19. Adding Longer Fragments as Indices

We have been exploring the idea of a finite set of keys for content indexing of English text. This set so far consists of several thousand selected full words and about six thousand 2- and 3-letter fragments for backup indexing when a word is not a selected key. This scheme works reasonably well when we want to describe text items as *M*-dimensional numerical vectors. Results can be noisy, however, when we want to find items containing a single word that is not a selected index key.

We can reduce indexing noise by allowing longer word fragment as keys (see Appendix D). In particular, adding over 3,000 4- and 5-letter fragments can sharpen our indexing without blowing up our finite index set. In general, we could keep adding more 4- and 5-letter fragments and perhaps some with 6 or more letters; but we will soon reach a point of diminishing returns for general applications. To be useful for indexing, new word fragments have to be fairly frequent in our target text.

If the text is specialized, like medical journal articles, military intelligence, food blogging, or financial reporting, more fine tuning might be appropriate. This would be accomplished more easily, however, by adding more words, and possibly longer word fragments. An information system manager must choose where to make the tradeoffs on noise and coverage, but having a good set of word fragments for backup indexing should always broaden the range of options available. Index words must be limited.

In an operational setting, any particular scheme of indexing should periodically be re-evaluated for effectiveness. A finite index set , however, should make it simpler to watch the statistics of each key to catch significant changes. Such monitoring is essential to the processing of news feeds where the unexpected is normal; an information routing system has to function resiliently in a dynamic environment, while keeping alert for important breaking events.

Here are two examples showing the difference between indexing only with 2- and 3- letter fragments and with 2-, 3-, 4-, and 5-letter fragments. The original text in both cases is the same sentence

```
PGA champion Phil Mickelson was headed home to California after missing the cut at the
Charles Schwab Classic by one stroke.
```

After removing grammatical function words and inflectional and morphological endings, we get

pga champion phil mickelson head home california miss cut charle schwab classic stroke

The 2- and 3-letter fragment indexing here would be

```
pg  cha      phi  mic       hea  hom  cal       mis  cut cha    sch    cla     str
 ga  ham      hil  ick       ead  ome  ali       iss      har    chw    las     tro
      amp           cke                lif                arl     wab   ass     rok
       mpi           kel                ifo                rle           ssi      ke
        pio           els                for                              sic
         ion            son               orn
                                           rni
                                            nia
```

With the 4- and 5-letter fragments of Appendix D added to our indexing, we get

```
pg  champ    phil mic       head home cali       miss cut char   sch    class   stro
 ga    mpi         ick                lif                arl   chw     assi    roke
        pion        cke                ifo                rle    wab     sic
                     kel                for
                      els                orni
                        son                nia
```

As can be seen, 3-letter fragments are most of our index keys here even after adding over three thousand longer fragments to our index set. We get only eleven indices of 4 letters and two of 5 letters, indicated above in **bold face**. This outcome is due to indexing with about 5,600 3-letter fragments, as defined in Appendix A, compared to only about 3,200 fragments of 4, and 5 letters. Longer fragments also are just each less likely to be found in any sample of text than our selected 3-letter fragments.

In general, finite indexing should look at linguistically plausible letter sequences. We can generate these through a random Markov process following the probabilities that one letters will follow another in a word; but we would have then have to verify that such sequences are actually frequent enough in English text to be worth putting into a hardwired finite core index set. Some degree of sweat is unavoidable here; but fortunately, such low-level labor needs to be done only once, and its results can be easily shared.

Some letter sequences would be candidates for a finite index set without vetting them first for fairly high frequency of occurrence. In English, we have many words of Latin and Greek origin, and these tend to be formed by joining two ancient words from a classical language to make a new composite word in English. For example, THEOS (God) plus (LOGOS (word) make THEOLOGY, the study of religion. This common in academic language: PHOTOGRAPH, HOLOGRAM, STEREOPHONIC, FERROMAGNETIC, ANALOG, ASYMMETRIC, and so forth.

The frequency-driven process for finding the word fragments in Appendix D discovered GRAM, GRAPH, and OLOGY, but missed many that any high school graduate might immediately think of; for example, PSYCH, SOMA, ECTO, NOMY, MONO, PENTA, LYSIS, and so forth. Any serious system for finite indexing should allow users add on such fragmentary indices. These have to be limited in number to keep our overall indexing finite, but just one or two thousand extra slots here could help greatly.

User-provided fragmentary indices have to be integrated properly with other kinds of indices, however. In particular, we still want to let fragments overlap, while not counting a fragment if it is completely covered by a single longer indexing fragment. From a coding standpoint, this is easiest to implement when a user-provided fragment always must be at either the start or the end of a word. For example, suppose we have the word HYPERTROPHY. With the user-provided indices HYPER- and -TROPHY, our analysis would be

```
hypertrophy
hyper-
  pert
    -trophy
```

The 4-letter sequence PERT is in the listing of important 4-grams in Appendix D. It is fairly frequent in English (exPERT, PERTain, PERTurb, PERTly) and should counted in indexing even though the leading and trailing literals, HYPER- and -TROPHY, completely cover all of its letters. Its occurrence helps to connect the literals in text. Every little bit of information helps in indexing.

If noise were of no concern, we could index just with selected short n-letter fragments. Having user-defined leading and trailing literals, however, lets users tune their text processing applications. Staying within a finite limit of about 10^4 indices altogether will be a challenge, but we only need to be careful in adding index keys. They must be common enough to make significant contributions to similarity scores, but yet uncommon enough to discriminate between different subsets of text data.

The natural statistics of text data mean that a finite index set can be “imperfect” and still support useful analyses of content. Think that every index set of word fragments as just being on the way to something bigger and better, though still finite. This is how technology generally works: every solution invites further tinkering. Improvement will always be in small steps, but if sustained, the results are enduring.

20. Alphanumeric Indexing

Numbers can greatly complicate indexing. The total number of numbers is infinite, and it really makes no sense to include them from any index set. As part of terms, however, numbers are essential for technical, scientific, historical, and military applications. We can easily handle them in two ways:

- Have alphanumeric 2-grams as indices. These include alphabetic 2-grams like AB, GR, and XY, but add 2-grams with digits like A1, X5, 3B, and 44. When analyzing a term, it is convenient to ignore hyphens and other embedded punctuation, so that something like AK-47 would have the three indices AK, K4, 47. There would be 1,296 alphameric 2-grams, as opposed to 676 alphabetic 2-grams.
- Users can specifically define literal n-grams like 31416-, Y2K, 2024, -000, -000000, or COVID19-. These have to be at the start or at the end of a term or an entire term and would be treated like any other literal n-gram for finite indexing.

There seems to be little urgency in going beyond such an alphanumeric indexing capability. One would really has to have in mind a specific application that needed something more sophisticated.

21.Software For Finite Indexing

Active Watch (AW) is a software package implementing the basics of finite indexing as described here. It was originally developed in the C language for use in open-source data analysis by government organizations. It also has been employed by a commercial news aggregator, but issues with technical support and licensing eventually forced its abandonment. The current version of AW (v1.3.4) was rewritten in object-oriented Java about that time to make the technology more portable.

AW is available as open-source code under a BSD license. Its Java code can be freely downloaded from GitHub on the Web at https://github.com/prohippo/ActiveWatch . There are currently about 150 Java source files for a system to demonstrate the basic AW text item clustering capability. Users can feed AW with their own UTF-8 text files to obtain clusters providing a kind of content map of the data. Anyone can freely modify the software and expand its capabilities.

AW finite indexing is organized around alphanumeric word fragments, called n-grams. These will be extracted from each unstopped word or other term in a text segment in a way that gets the smallest set covering every character of the term, while maximizing the total length of those fragments. In general, indexing n-grams will overlap in a term, but aw will not count them if they are subsumed by a longer indexing n-gram for that term.

Currently, AW recognizes five types of indexing n-grams:

- The complete set of all 1,296 alphanumeric 2-grams, which will be adequate to cover every letter of any term with no punctuation in it. Theoretically, such n-grams would be enough to index any text, but this will be noisy. Alphabetic 2-grams tend to have similar frequency distributions in the text segments in any given language, which is exploited by cryptanalysts trying to crack simple encipherment. For example, they know that TH, IN, ER, RE, and AN will probably be the five most frequent 2-grams in any English text segment of a thousand characters or more.
- The persistent imbalance in the frequency of 2-grams in natural languages like English means lower entropy for alphanumeric 2-gram indexing. To even out the frequency of indexing 2-grams, we can introduce indexing 3-grams by taking the K most frequent alphabetic 2-grams and adding a single trailing letter. For example, with TH, we get THA, THB, THC, … , THY, THZ. From the first 216 most frequent alphabetic 2-grams, we can get 5,616 alphabetic 3-grams to index with.
- A hybrid index set of about 7,000 alphanumeric 2-grams plus some alphabetic 3-grams will support the running 0f the experiments in Section 12 to sh0w the effectiveness of n-gram indexing. Analysis with only 2- and 3-gram indices will still be somewhat noisy, but we can increase their signal by adding manually selected alphabetic 4-grams to our finite indexing set. AW so far recognizes 2,460 of these, all known to have fairly high frequency, unlike our derived 3-grams. For example, CONT, PART, WARE.
- There are probably more 4-grams that we might add to our word-fragment index set, but it is helpful to explore longer n-grams. Cryptanalysis resources often compile the most frequent 5-grams in English, which we can readily exploit. These are much rarer than fairly frequent alphabetic 4-grams, and so their inclusion in an index set has only a limited impact on overall entropy. AW currently recognizes 700 alphabetic 5-grams, including ANGER, DRIVE, PHONE, WORLD; these should be recognizable to most English readers, but are really fragments. On the whole, they would seem worth indexing with, and we probably should look for more of them. The usefulness of n-grams of 6 or more letters is as yet unclear.
- AW 2-, 3-, 4-, and 5-grams are hardwired into Java code. These were selected for English in general and should be usable for all applications. Changes to them will entail recompiling code, but AW allows for a easier option. Users can define up to 2,000 n-grams called literals: either (1) a leading sequence of specific letters at the start of a word; or (2) a trailing sequence at the end. These should be longer than 2 characters and can more closely approximate whole-word indexing. For example, COMMON-, EXCHANGE-, -ONOMY, -000000.

The details of AW indexing are still in flux. It seems, however, that AW can operate quite effectively with an index set of only about 10^4 word fragments. We cran further improve such performance for particular target corpora of text. Possibilities here go beyond adding more word fragments to index. AW is already quite competent at stopword removal and inflectional and morphological stemming in English, but these also can be tuned further. All such measures can add up in the quest for higher indexing entropy.

The current GitHub AW software focuses on automatic clustering of text items according to content. Finite n-gram indexing allows this to be done rigorously, controlled by thresholds with sound statistical interpretations. The AW clustering algorithm works in two stages: (1) identification of compact and densely linked cluster seeds; and (2) creation of a profile for each seed to determine which text items should be assigned to each cluster. Items may be put into more than one cluster.

The output of the AW clustering demonstration is a set of descriptive words or word roots for each cluster. These are usually enough to identify the topic associated with each cluster, although one has to figure out which words go with each other. The results are sensitive to the choice of indexing. AW users are encouraged to cluster their own text data with their own selection of literal n-grams (or none at all). The clusters for the sample text included in the AW GitHub distribution took less than 10 seconds to process.

Java code exists for other AW applications beyond clustering, but having been written decades ago, this will require extensive cleanup and updating even just to compile successfully. The source files are as yet unavailable from GitHub. None of this code is magic, or even cutting edge. AW exists to show that there is definitely a big payoff in paying proper attention to the nuts and bolts of linguistics and statistics in any automated text processing, even those eventually involving neural nets and other machine learning.

22. Going to Infinity by Finite Steps

Medieval monastic librarians invented finite indexing of text collections. Their methods were adequate for the needs of scholars in their time, but exponential growth of text data in our modern information age calls for a more automated and analytical approach. As it turns out, our index sets can remain finite, if we allow indexing keys to include common word fragments. This will be noisier than always indexing with full words, but it will provide benefits like statistically scaled measures of similarity between text items.

Think of indexing with word fragments as an approximation to indexing with full words. With just two-letter word fragments, text indexing has to be quite skimpy and rough, but alphanumeric 2-gram vector signatures can nevertheless cover every letter and digit of every term in a text item. The problem is the information content of 2-grams alone is too low to support the degree of content discrimination we often need for text processing, even if the probabilities of those 2-grams could be all the same. With only about 10^3 of them, they are a poor substitute for having about 10^5 words and other terms for indexing content.

As we have seen, an obvious next step is to augment our indices with 3-grams, but there are only about 10^4 alphabetic ones, and most of those are like QZQ, KJK, or ZRR, which are useless for content indexing. Nevertheless, 3-gram indices will be unavoidable for better discrimination in indexing; we just have to be careful in choosing them. The current AW approach indexes with all 3-grams beginning with a common alphabetic 2-gram. These cannot be guaranteed to cover every letter of every unstopped word in a text segment for indexing, but the gaps can be filled by 2-grams, which will cover all text.

The latest version of ActiveWatchhas added over 3,000 preselected alphabetic 4- and 5-gram indices for even better performance. Just expanding an index set of word fragments to should increase its information value, but proper selection of longer indices can also improve the entropy contribution of shorter indexing n-grams. For example, the 4-gram ASTE contains the two frequent 3-grams AST and STE. When AW can count an occurrence of ASTE, it no longer counts the subsumed occurrences of AST and STE, which should even out their frequencies overall and increase indexing entropy.

This all provides a general roadmap for how to keep improving our finite indexing of text. As we gain experience with systems like AW, we can continue to refine and extend n-grams to extend our fragmentary indexing and thereby get ever closer to a finite approximation to indexing with an unlimited and always growing set of words and other terms. Finding useful longer n-grams will get harder, but such labor has to be done only once for a particular language like English. English is worth the effort here.

Meanwhile, the current hybrid built-in finite index set of 2-, 3-, 4-, and 5-grams, plus user-supplied literal n-grams, seems to be doing a good job in automatic clustering of text items by content and in filtering the content of text streams. In particular, the descriptive keywords generated by AW for its clusters do make sense to a human reader and could be developed into a tool for rigorously mapping out large collections of English text for academic, legal, government, and commercial research.

None of this involves artificial intelligence or advanced machine learning. AW is good, however, at packing information into numerical vectors of finite dimensionality. Numerical vector representations of text still persist in automated information processing; and there would be tremendous advantages when those vectors have finitely many dimensions. Some people may remain uncomfortable about indexing with word fragments, but with ever longer n-gram indices, we can reduce the noise of indexing as much as we want.

In a sense, numerical vectors constitute an abstract engineering model of text to simplify its processing. As long as the vectors have the right statistical characteristics, it should not matter whether their numbers come from whole words, or word fragments, or some combination of these. Their linguistic origin is probably a detail that artificial neural net wranglers and practitioners of latent Dirichlet allocation would just as soon not have to worry about, a kind of conceptual sausage making.

Appendix A. Two-Letter Base Fragments

These are a selection of 216 English 2-letter fragments selected as seeds in the 2002 version of the ActiveWatch system. Each seed can be extended into 3-letter fragments with the addition of a single following letter, if there is one. An indexer may want to adjust this base set to reflect the letter statistics of an actual target collection of text.

```
"ab", "ac", "ad", "af", "ag", "ai", "al", "am",
"an", "ap", "ar", "as", "at", "au", "av", "aw",
"ba", "be", "bi", "bl", "bo", "br", "bu", "ca",
"cc", "ce", "ch", "ci", "ck", "cl", "co", "cr",
"ct", "cu", "da", "de", "di", "do", "dr", "du",
"ea", "eb", "ec", "ed", "ee", "ef", "eg", "ei",
"el", "em", "en", "eo", "ep", "er", "es", "et",
"ev", "ex", "fa", "fe", "ff", "fi", "fl", "fo",
"fr", "fu", "ga", "ge", "gh", "gi", "gl", "go",
"gr", "gu", "ha", "he", "hi", "ho", "hu", "ia",
"ib", "ic", "id", "ie", "if", "ig", "ik", "il",
"im", "in", "io", "ir", "is", "it", "iv", "ja",
"je", "jo", "ju", "ke", "ki", "la", "le", "li",
"ll", "lo", "lt", "lu", "ma", "mb", "me", "mi",
"mm", "mo", "mp", "mu", "na", "nc", "nd", "ne",
"nf", "ng", "ni", "nn", "no", "ns", "nt", "nu",
"nv", "oa", "ob", "oc", "od", "of", "og", "oi",
"ol", "om", "on", "oo", "op", "or", "os", "ot",
"ou", "ov", "ow", "pa", "pe", "ph", "pi", "pl",
"po", "pp", "pr", "pu", "qu", "ra", "rc", "rd",
"re", "rg", "ri", "rl", "rm", "rn", "ro", "rr",
"rs", "rt", "ru", "rv", "sa", "sc", "se", "sh",
"si", "sl", "so", "sp", "ss", "st", "su", "ta",
"te", "th", "ti", "tl", "to", "tr", "tt", "tu",
"ua", "ub", "uc", "ud", "ue", "ug", "ui", "ul",
"um", "un", "up", "ur", "us", "ut", "va", "ve",
"vi", "vo", "wa", "we", "wi", "wo", "xp", "ye"
```

For example, “TH” might be followed in a text segment by the letter “Q”; in this case, the resulting 3-letter fragment “THQ” would be recognized for indexing. This is a rare 3-letter fragment, but it will be useful for recognizing the occurrence of the word “EARTHQUAKE” when its full form is omitted from an index set.

Appendix B. Probabilistic Modeling

Given any finite index set X for text, regardless of how its keys were obtained, we can generally define a vector representation of any text item along with an inner product similarity measure for items d and e

$$s(d,e) = \sum_{j=1}^{M} a_j \cdot f_{jd} \cdot f_{je}$$

The a_j are importance weights that someone might assign to index key x_j in X. We can make $a_j = 1$ to avoid any differential weighting of keys.

The measure $s()$ is not yet scaled so that we can compare similarity scores between different pairs of text items of various lengths. Instead of converting item vectors to unit length as done for the cosine measure, we can approach this problem probabilistically with some simplifying assumptions: (1) all index keys x_j have a probability p_j that a given key occurrence in an item will be x_j; (2) given any text item d, with N_d occurrences of index keys, x_j on the average will occur $N_d \cdot p_j$ times with a variance of $N_d \cdot p_j \cdot (1 - p_j)$.

These two assumptions basically mean that the keys of index set X are independent of each other. Strictly speaking, this is untrue. Words in text generally do not occur independently; if you see “smoke” in some segment of text, then it becomes more likely that you will also see “fire.” When their letters can overlap, word fragments are certainly also not independent. For thinking about a technology, however, an approximate model can be of help, unlike models with no connection to reality.

If our assumptions hold approximately, we can employ the multinomial distribution in statistics to get a handle on our inner product similarity measure. First, let us define an auxiliary random variable

$$g_j = f_{jd} \cdot f_{je}$$

It does not matter what d and e are; they just have to be different. We compute the expected value of g_j from the expected value of f_{jd} times the expected value of f_{je}, since f_{jd} and f_{je}, will be independent.

$$\begin{aligned} E[g_j] &= E[f_{jd}] \cdot E[f_{je}] \\ &= N_d \cdot p_j \cdot N_e \cdot p_j \\ &= N_d N_e p_j^2 \end{aligned}$$

Now we need some other statistics for g_j. By definition

$$Var[g_j] = E[g_j^2] - E[g_j]^2$$

where

$$E[g_j^2] = E[f_{jd}^2] \cdot E[f_{je}^2]$$

and

$$\begin{aligned} E[f_{jd}^2] &= E[f_{jd}]^2 + Var[f_{jd}] \\ &= N_d[(N_d - 1)p_j^2 + p_j] \end{aligned}$$

$$E[f_{je}^2] = N_e[(N_e - 1)p_j^2 + p_j]$$

So

$$E[g_j^2] = N_d N_e \cdot [(N_d N_e - N_d - N_e + 1)p_j^4 + (N_d + N_e - 2)p_x^3 + p_j^2]$$

and

$$Var[g_j] = N_d N_e \cdot [-(N_d + N_e - 1)p_j^4 + (N_d + N_e - 2)p_j^3 + p_j^2]$$

Finally, we have the statistic

$$Cov[g_j, g_k] = E[g_j g_k] - E[g_j]E[g_k]$$

where

$$E[g_j g_k] = E[f_{jd} f_{kd}] \cdot E[f_{je} f_{ke}]$$

and

$$E[f_{jd} f_{kd}] = E[f_{jd}]E[f_{kd}] + Cov[f_{jd} f_{kd}]$$

If index keys x_j and x_k are independent, which should be approximately true in fairly long text items,

$$Cov[f_{jd} f_{kd}] = -N_d p_j p_k$$

$$Cov[f_{je} f_{ke}] = -N_e p_j p_k$$

We can now calculate

$$E[s(d,e)] = \sum_{j=1}^{M} a_j \cdot E[g_j]$$

$$Var[s(d,e)] = \sum_{j=1}^{M} a_j^2 \cdot Var[g_j] + \sum_{j=1}^{M} \sum_{k \neq j} a_j a_k \cdot Cov[g_j g_k]$$

With our substitutions from above

$$E[s(d,e)] = N_d N_e \cdot \sum_{j=1}^{M} a_j \cdot p_j^2$$

$$Var[s(d,e)] = N_d N_e \cdot [\sum_{j=1}^{M} a_j p_j^2 \; + \; (N_d + N_e - 2) \sum_{j=1}^{M} a_j p_j^3 \; -$$

$$(N_d + N_e - 1) \sum_{j=1}^{M} a_j p_j^4 \; - \; (N_d + N_e - 1) \sum_{j=1}^{M} \sum_{k \neq j} a_j a_k p_j^2 p_k^2 \;]$$

The summations in both $E[s(d,e)]$ and $Var[s(d,e)]$ can be precomputed, since they are the same for all item vector pairs.

The calculations above do not apply when we want to compare many item vectors against a fixed query vector q as in a search over a collection of text items. Our similar measure becomes just

$$S_q(d) = \sum_{j=1}^{M} q_j f_{jd}$$

where query vector q has the components $\{ q_j \}$.

With cosine similarity, the scaling of a query match score would be the same as for a score comparing two item vectors. Our probabilistic approach would have instead

$$E[s_q(d)] = \sum_{j=1}^{M} q_j \cdot E[f_{jd}]$$

$$Var[s_q(d)] = \sum_{j=1}^{M} q_j^2 \cdot Var[f_{jd}] \; + \; \sum_{j=1}^{M} \sum_{k \neq j} q_j q_k \cdot Cov[f_{jd} g_{kd}]$$

Substituting as before, this gives us the multinomial model predictions

$$E[s_q(d)] = N_d \cdot [\sum_{j=1}^{M} q_j p_j \ ]$$

$$Var[s_q(d)] = N_d \cdot [\sum_{j=1}^{M} q_j^2 p_j(1-p_j) \ - \ \sum_{j=1}^{M} \sum_{k \neq j} q_j q_k p_j p_k \ ]$$

These distributional parameters are much simpler to compute, and the bracketed summations can be precomputed once for each query. Except for automatic clustering, which compares pairs of item vectors, this simplified model for profile matching will be the workhorse of our text information processing.

Appendix C. Significance Testing

With a theoretical model and samples of experimental data, we can statistically quantify the degree of match between them. The χ^2 test is often used for this purpose, but this is most appropriate with experimental data distributions that are unimodal and symmetric. Inner product similarity measures, however, are unimodal and skewed toward higher scores.

For just scaling item similarity, however, the actual shape of a model distribution is unimportant as long as it can reliably estimate the mean and standard deviation of computed inner product similarity scores. We can test both these statistical parameters with two separate applications of Student's *t*-test. Our similarity scores for pairs of items can be divided into $n = 4$ subsets for our computations below.

Mean μ

The null hypothesis H_0 will be that the sample means of similarity scores are consistent with model predictions.

$$H_0: \quad \mu = \mu_0$$
$$H_a: \quad \mu \neq \mu_0$$

For the rejection of the null hypothesis with confidence 1 - α

$$|\bar{m} - E[S]| \geq \frac{s_m}{n^{1/2}} \cdot t_{\alpha/2,n-1}$$

where

$$\bar{m} = \sum_{i=1}^{n} \frac{m_i - E[S]}{n}$$

$$s_m = \sum_{i=1}^{n} \frac{(m_i - \bar{m})^2}{n-1}$$

Variance σ^2

The null hypothesis H_0 will be that the sample variances of similarity scores are consistent with model predictions.

$$H_0: \quad \sigma^2 = \sigma_0^2$$
$$H_a: \quad \sigma^2 \neq \sigma_0^2$$

For the rejection of the null hypothesis with confidence 1 - α

$$|\bar{d}| = \sum_{i=1}^{n} \geq \frac{s_d}{n^{1/2}} \cdot t_{\alpha/2,n-1}$$

where

$$\bar{d} = \sum_{i=1}^{n} \frac{log\ 100 \cdot s^2 - log\ 100 \cdot Var[S]}{n} = \sum_{i=1}^{n} \frac{d_i}{n}$$

$$s_d = \sum_{i=1}^{n} \frac{(d_i - \bar{d})^2}{n-1}$$

Appendix D. Major 4- and 5-Letter Fragments

We can improve the entropy of English word fragment indexing with some 4- and 5-letter fragments. These can subsume occurrences of common 3-letter fragments into longer, more specific fragments to level out the frequencies of indexing keys. Cryptanalysis resources for enciphered English text[16] can suggest longer indexing fragments, but these need filtering. They come from text with grammatical function words and inflectional and morphological endings kept and word separators removed.

For example, a common 4-letter fragment helpful for cryptanalysis is EDTH, which might come from the text segment "…ED TH…" as in "bakED THe cake" or "washED THat car." Frequency tables for such fragments could help in cracking a secret message with spaces removed to make it harder to analyze, but would be of little use to a medieval monastic librarian. Here is a handpicked set of 2,500 specific English 4-letter fragments appropriate for indexing content when word boundaries are recognized:

```
"aban", "abbl", "abit", "able", "abor", "acce", "acci", "acco", "ache", "acid", "acle", "acon", "acqu", "acre", "acro", "addr", "adem", "adet", "adge", "adle",
"admi", "adve", "aero", "agen", "aggl", "aggr", "agle", "agon", "agra", "agri", "ague", "aign", "aile", "aint", "aire", "aise", "alan", "alco", "alem", "alia",
"alle", "allo", "ally", "alom", "alon", "aloo", "alor", "alte", "alth", "alti", "alve", "alyt", "ambe", "ambi", "ambl", "amer", "amin", "amou", "ampl", "anal",
"anar", "anat", "ance", "anch", "anct", "andi", "andl", "aneu", "ange", "angl", "angr", "anne", "anon", "anso", "anta", "ante", "anti", "anyo", "apar", "appe",
"appl", "appo", "appr", "apse", "aqua", "arab", "arad", "aran", "arbi", "arbo", "arch", "area", "aren", "arge", "argo", "aris", "arma", "arne", "arni", "arom",
"aron", "arra", "arre", "arri", "arro", "arry", "arse", "arte", "arth", "arti", "arve", "asci", "asia", "asis", "aske", "ason", "assa", "asse", "assi", "assu",
"aste", "asto", "astr", "asur", "atar", "atch", "ater", "athe", "athl", "atin", "atio", "atri", "atta", "atte", "atti", "atto", "attr", "atul", "audi", "augh",
"ault", "aunt", "ause", "aust", "auth", "auto", "avai", "aven", "avid", "awar", "axis", "babe", "baby", "back", "bact", "bail", "bait", "bake", "bala", "bald",
"ball", "balm", "band", "bane", "bang", "bank", "bann", "barb", "bard", "bare", "bark", "baro", "barr", "base", "bate", "bath", "batt", "bble", "bead", "bean",
"bear", "beat", "beef", "beer", "beli", "bell", "belt", "bend", "bene", "bent", "berg", "best", "beve", "bibl", "bide", "bill", "bina", "bind", "bine", "bing",
"biol", "biom", "bion", "bird", "bite", "bjec", "blas", "blea", "blem", "blin", "blis", "bloo", "blow", "blue", "boar", "boat", "bode", "body", "bolt", "bomb",
"bond", "bone", "bonk", "book", "boom", "boon", "bore", "born", "boro", "bort", "bott", "boul", "brag", "brai", "bran", "brav", "brea", "bred", "brew", "brid",
"bris", "brit", "broa", "brow", "buck", "buff", "buil", "bulk", "bull", "bump", "bund", "bung", "bunk", "burg", "burn", "burs", "bury", "bush", "bust", "bute",
"butt", "byte", "cade", "cage", "cake", "cala", "calc", "cale", "cali", "call", "calm", "camp", "cana", "canc", "cand", "cane", "cann", "cant", "capa", "cape",
"capi", "capt", "carb", "card", "care", "carn", "carp", "carr", "cart", "case", "cash", "cask", "cast", "cata", "cate", "cath", "catt", "caus", "cava", "cave",
"ceal", "ceiv", "cele", "cell", "cend", "cens", "cent", "cept", "cere", "cern", "cert", "cess", "cest", "chal", "cham", "chan", "chap", "char", "chas", "chat",
"chea", "chem", "chen", "cheo", "chet", "chic", "chie", "chim", "chin", "chip", "chis", "choo", "chop", "chor", "chow", "chri", "chro", "chur", "cial", "cide",
"cien", "cile", "cind", "cise", "cite", "city", "civi", "cket", "ckle", "clan", "clap", "clar", "clat", "claw", "cler", "clim", "clin", "clip", "clos", "clot",
"club", "coal", "coar", "coat", "cock", "coco", "code", "coff", "cold", "coll", "colo", "colt", "colu", "comb", "come", "comm", "comp", "conc", "cond", "cone",
"conf", "cong", "conn", "cons", "cont", "conv", "cook", "cool", "coon", "coop", "cope", "cord", "core", "cork", "corn", "corp", "corr", "cosm", "cost", "cote",
"cott", "coun", "coup", "cour", "cove", "crab", "cram", "cran", "crap", "cras", "crat", "craw", "craz", "crea", "cred", "cree", "crep", "cret", "crew", "crib",
"crim", "crip", "cris", "crit", "crop", "crot", "crow", "cruc", "crui", "crum", "ctic", "cuff", "cula", "cule", "cult", "cumb", "curb", "curd", "curi", "curr",
"curt", "curv", "cuse", "cuss", "cust", "cute", "cybe", "cyte", "cyto", "dale", "damp", "dana", "danc", "dang", "dapt", "dard", "dare", "dark", "dart", "dash",
"data", "date", "daug", "dawn", "ddle", "dead", "deal", "dean", "dear", "deat", "deca", "dece", "deci", "decl", "deep", "deer", "defe", "defi", "dela", "dele",
"deli", "delt", "delu", "deme", "demi", "demn", "demo", "dens", "dent", "deny", "depe", "depo", "depr", "dera", "desc", "desi", "desp", "dest", "dete", "deut",
"deve", "devi", "dial", "dict", "dida", "dier", "diet", "diff", "digi", "dile", "dilu", "dime", "dine", "ding", "dirt", "disa", "disc", "dise", "dish", "disi",
"disp", "dist", "dite", "dium", "dive", "divi", "dock", "doct", "dole", "dome", "domi", "dona", "done", "dong", "doom", "door", "dorm", "dors", "dose", "dote",
"down", "doze", "drag", "draw", "drea", "dres", "drip", "driv", "dron", "drop", "drow", "drug", "drum", "duce", "duck", "dumb", "dump", "dunk", "dupl", "dusk",
"dust", "dway", "dyna", "each", "eagl", "eant", "eard", "earm", "earn", "eart", "ease", "easo", "east", "eate", "eath", "eave", "ebra", "ebri", "ebut",
"ecal", "eche", "echo", "ecia", "ecip", "ecla", "ecom", "econ", "ecor", "ecre", "ecto", "ectr", "edal", "eden", "eder", "edes", "edge", "edit", "educ", "eech",
"eeti", "eeze", "efea", "effo", "egen", "egim", "egis", "egot", "eigh", "eign", "elan", "eleb", "elec", "eleg", "elem", "eles", "elig", "elim", "elin", "elit",
"elle", "elon", "elte", "elve", "emai", "eman", "embe", "embl", "embr", "emed", "emem", "emen", "emer", "emis", "emol", "emon", "emor", "emot", "empe", "empl",
"empt", "enal", "enam", "ence", "ench", "enef", "enfo", "enge", "engl", "enne", "enom", "enon", "enor", "enro", "ense", "ensu", "ente", "enti", "entr", "eopl",
"epar", "epor", "epre", "eput", "equa", "eque", "equi", "erag", "eral", "erap", "erce", "erch", "erie", "eril", "erio", "eris", "erit", "erra", "erro", "erry",
"erse", "erso", "erty", "erua", "erve", "esca", "esig", "esis", "esse", "esso", "esth", "esti", "estl", "estr", "etai", "etal", "etch", "eter", "ethe", "ethi",
"etho", "ethy", "etro", "ette", "ettl", "etto", "etty", "eval", "evel", "even", "ever", "evic", "evid", "evil", "evol", "eway", "exam", "exce", "exit", "expa",
"expe", "expl", "exte", "extr", "face", "fact", "fail", "fair", "fake", "fall", "fals", "fame", "fang", "fant", "fare", "farm", "fasc", "fast", "fate", "favo",
"fear", "feat", "fect", "fede", "feed", "feel", "feit", "feli", "fell", "femi", "fend", "fern", "ferr", "fess", "fest", "ffer", "ffic", "ffle", "ffor", "fice",
"fici", "fict", "fide", "fiel", "figu", "file", "fill", "film", "fina", "find", "fine", "fini", "fire", "firm", "fish", "fiss", "fist", "flag", "flai", "flam",
"flan", "flat", "flee", "flex", "flim", "flip", "floo", "flop", "flor", "flou", "flow", "foil", "fold", "foli", "folk", "foll", "fond", "food", "fool", "foot",
"forc", "ford", "fore", "forg", "fork", "form", "fort", "foul", "frag", "frai", "fran", "frat", "free", "fres", "fric", "frug", "fter", "fuel", "fuge", "full",
"fume", "func", "fund", "fung", "furn", "fury", "fuse", "fuss", "gain", "gale", "gall", "gamb", "game", "gang", "gant", "garb", "gard", "gate", "gave", "gear",
"genc", "gend", "gene", "geni", "gent", "geon", "geri", "germ", "gest", "ggle", "gift", "giga", "gile", "gine", "gird", "girl", "give", "glad", "glam", "glan",
"glen", "glob", "gnal", "goat", "gold", "golf", "gone", "good", "goon", "gorg", "gory", "gote", "grad", "gram", "grap", "grat", "grav", "gray", "gree", "gren",
"grid", "grin", "grou", "grow", "grum", "guar", "guil", "gulf", "gull", "gust", "hack", "hail", "hair", "half", "hall", "halo", "hame", "hamp", "hanc", "hand",
"hane", "hank", "hant", "happ", "hard", "hark", "harm", "harp", "harr", "hart", "hase", "hast", "hate", "have", "hawk", "hbor", "head", "heal", "heap", "hear",
"heat", "heav", "heel", "heir", "hell", "helm", "help", "hema", "hemi", "hemo", "hend", "hera", "herb", "herd", "here", "heri", "hero", "hest", "hexa", "hibi",
"hick", "hief", "high", "hill", "hind", "hine", "hing", "hint", "hion", "hist", "hive", "hman", "hock", "hoke", "hold", "hole", "holy", "home", "hone", "honk",
"hony", "hood", "hook", "hool", "hope", "hore", "hori", "horn", "horr", "hose", "host", "hour", "hous", "hull", "humb", "hump", "hund", "hunk", "hunt", "hurl",
"hush", "hust", "hydr", "hype", "hypn", "hyst", "iber", "ibut", "ican", "icat", "icer", "icip", "icle", "iday", "idea", "idge", "idol", "ield", "iend", "ient",
"ieve", "ifle", "igat", "igen", "iger", "ight", "igni", "ilet", "ilia", "illa", "ille", "imag", "imen", "imet", "imit", "imme", "immo", "immu", "impa", "impe",
"impl", "impo", "impr", "imul", "inap", "ince", "inch", "inci", "inco", "inct", "inde", "indi", "indu", "inet", "infl", "info", "inge", "ingo", "ingu", "inno",
"inse", "insp", "inst", "insu", "inta", "inte", "inti", "intr", "inue", "inut", "inve", "invo", "ipal", "iple", "ipro", "ipse", "ique", "irch", "iron", "irra",
"irre", "irri", "isan", "isci", "isit", "isla", "isle", "ison", "ispo", "issu", "iste", "istl", "istr", "ital", "itar", "itch", "iter", "ithe", "ithm", "itio",
"itis", "itte", "ittl", "itud", "itut", "ival", "ivat", "ivel", "iver", "ivil", "ivor", "izen", "jack", "jail", "japa", "jazz", "jean", "ject", "jerk", "jers",
"jest", "jing", "join", "jour", "jump", "juni", "junk", "jure", "jury", "just", "keep", "kick", "kill", "kilo", "kind", "kine", "king", "kiss", "kitc", "knee",
"knit", "knot", "know", "labo", "lace", "lack", "lact", "lade", "lady", "lage", "laid", "laim", "lain", "lamb", "lamp", "land", "lane", "lang", "lank", "lant",
"lapi", "lard", "lare", "larg", "lari", "lark", "lash", "lass", "last", "late", "lati", "latt", "lava", "lave", "lawn", "lead", "lean", "lear", "leas", "leav",
"lebr", "lect", "leep", "left", "lega", "legi", "lend", "lent", "leon", "lert", "less", "lest", "lete", "lett", "leve", "libe", "libr", "lice", "lici", "lick",
"lict", "lide", "lief", "lien", "life", "liff", "lift", "lige", "ligi", "lign", "like", "limb", "lime", "limi", "limp", "lind", "line", "ling", "link", "lint",
"lion", "liqu", "lish", "list", "lite", "lith", "live", "llab", "llar", "llet", "lley", "llow", "load", "loan", "loat", "loca", "lock", "loft", "logi", "lone",
"long", "look", "loom", "loon", "loop", "lope", "loqu", "lord", "lore", "loss", "loud", "lout", "love", "lter", "luck", "lude", "lumb", "lume", "lump", "luna",
"lund", "lung", "lunk", "lunt", "lure", "lush", "lust", "lute", "lway", "mace", "mach", "mage", "maid", "mail", "main", "majo", "make", "mala", "male", "mall",
"manc", "mand", "mang", "mani", "mank", "mans", "mant", "manu", "mare", "mari", "mark", "marr", "mart", "mary", "masc", "mash", "mask", "mass", "mast", "mate",
"math", "matr", "matt", "matu", "maze", "mbar", "mbat", "mber", "mbin", "mble", "mbra", "meal", "mean", "meas", "meat", "mech", "meda", "medi", "medy", "meet",
"mega", "melo", "melt", "memb", "memo", "mend", "mens", "ment", "mera", "merc", "merg", "mess", "mest", "mete", "meth", "metr", "migr", "mile", "mili", "milk",
"mill", "mina", "mind", "mine", "mini", "mint", "minu", "mira", "mirr", "misc", "mise", "mish", "miss", "mist", "mite", "mmer", "mmis", "mmit", "mmon", "mmun",
"mock", "mode", "mold", "mona", "mond", "mone", "monk", "mono", "mont", "mony", "moon", "moor", "more", "morn", "mort", "mote", "moth", "moti", "moun", "move",
"movi", "mpac", "mpar", "mpas", "mpat", "mper", "mple", "mplo", "mpre", "mula", "mule", "muni", "musc", "muse", "mush", "musi", "musk", "must", "mute", "mutu",
"myst", "myth", "nack", "nage", "nail", "nake", "nald", "name", "nant", "narc", "nate", "nati", "natu", "naut", "ncia", "ncil", "ndam", "nday", "nder", "ndic",
"ndit", "ndle", "neck", "nect", "need", "nega", "nerg", "nest", "netw", "neut", "news", "nfor", "nfra", "ngag", "ngen", "nger", "ngin", "ngle", "ngra", "ngth",
"ngue", "nice", "nick", "nign", "nion", "nior", "nito", "nitr", "nkle", "nman", "nnel", "nock", "nome", "noon", "norm", "nose", "note", "noti", "noun", "nput",
"nque", "nsel", "nser", "nset", "nsio", "nsis", "nsit", "nsti", "nstr", "nsul", "ntam", "nten", "nter", "ntim", "ntin", "ntle", "ntra", "ntro", "ntry", "numb",
"nute", "oach", "oard", "oast", "oath", "obab", "ober", "occa", "occu", "ocen", "oche", "ocke", "octa", "octo", "ocus", "odel", "odge", "offe", "offi", "ogra",
"oice", "oise", "oist", "olat", "ollo", "oman", "omba", "ombi", "omen", "omeo", "omet", "omma", "ommo", "ompa", "ompe", "ompl", "onar", "onat", "onde", "ondo",
"ondu", "onen", "oney", "ongr", "onic", "onom", "onor", "onsc", "onst", "ontr", "oodl", "oose", "ooth", "open", "opin", "ople", "oppo", "opti", "orce", "orch",
"orde", "ordi", "orge", "orig", "orit", "orna", "orne", "orni", "orri", "orth", "ortr", "ortu", "osis", "osit", "ossi", "otch", "otel", "othe", "ottl", "otto",
"oubl", "ouch", "ough", "ould", "ounc", "ound", "oung", "ount", "ourn", "ouse", "oust", "oute", "outh", "outl", "oval", "ovel", "oven", "over", "ower", "oxid",
"oyal", "pace", "pack", "pact", "page", "paid", "pain", "pair", "pale", "pali", "pall", "palm", "pand", "pane", "pani", "pank", "pant", "para", "pard", "pare",
"pari", "park", "parl", "part", "pass", "past", "pate", "path", "patr", "patt", "pave", "peak", "peal", "pear", "peck", "pect", "pede", "peer", "pell", "pena",
"pend", "pene", "peni", "pent", "peop", "pera", "perc", "perf", "peri", "perk", "perm", "perp", "pers", "pert", "perv", "pess", "pest", "pete", "peti", "petu",
"phan", "phas", "phat", "phen", "phil", "phin", "phon", "phor", "phos", "phra", "phys", "pick", "pict", "pier", "pike", "pile", "pill", "pine", "ping", "pink",
"pion", "pipe", "pire", "pise", "pist", "plac", "plag", "plan", "plas", "plat", "play", "plea", "plet", "plex", "plic", "plot", "ploy", "plun", "pock", "poin",
"pois", "poke", "pole", "poli", "poll", "poly", "pond", "pone", "pong", "pont", "pony", "pool", "poon", "poor", "popu", "porc", "pore", "pork", "porn", "port",
"pose", "posi", "post", "pote", "poti", "pour", "pout", "ppea", "ppen", "pple", "ppli", "ppor", "pray", "prea", "prec", "pred", "pree", "preg", "prem", "prep",
"pres", "pret", "prev", "pric", "prim", "prin", "prio", "pris", "priv", "prob", "proc", "prof", "prog", "prom", "prop", "pros", "prot", "prov", "prox", "prud",
```

[16] http://practicalcryptography.com/cryptanalysis/letter-frequencies-various-languages/english-letter-frequencies/

```
"pter", "ptim", "publ", "pugn", "pull", "puls", "pump", "puni", "punk", "pure", "purg", "push", "puss", "pute", "quad", "quan", "quar", "quen", "quer", "ques",
"quet", "quir", "quit", "quiz", "race", "rack", "ract", "rade", "radi", "raft", "rage", "raid", "rail", "rain", "rait", "rald", "rama", "ramb", "rame", "ramp",
"ranc", "rand", "rang", "rank", "rans", "rant", "rape", "rash", "rass", "rast", "rate", "rath", "rati", "rato", "rave", "rawl", "raze", "rbor", "rcen", "rday",
"rdle", "reac", "read", "reak", "real", "ream", "reap", "rear", "reas", "reat", "rebe", "rebo", "reca", "rece", "reci", "reck", "reco", "rect", "rede", "redi",
"redu", "reed", "reel", "reet", "refe", "refi", "refo", "refu", "regi", "regu", "rehe", "rela", "rele", "reli", "reme", "remo", "rend", "rent", "repe", "repo",
"repr", "repu", "resc", "resh", "resi", "reso", "resp", "ress", "rest", "resu", "reta", "reti", "retr", "retu", "reve", "revi", "rgen", "rget", "rian", "riat",
"ribe", "rice", "rich", "rick", "rict", "ride", "rief", "rien", "rier", "rife", "riff", "rifl", "rift", "rike", "rill", "rime", "rina", "rine", "ring", "rink",
"ripe", "ripo", "ript", "rise", "rish", "risk", "rist", "rita", "rith", "rive", "rize", "rman", "rmor", "rnet", "road", "roar", "robe", "robo", "rock", "rode",
"roga", "roid", "roil", "roke", "role", "roll", "roma", "romb", "romo", "rone", "rong", "roni", "ront", "rook", "room", "roon", "root", "rope", "roph", "rose",
"ross", "roth", "roud", "roun", "roup", "rous", "rout", "rove", "rovi", "rowd", "rown", "rran", "rren", "rres", "rror", "rrow", "rson", "rter", "rtle", "rtun",
"rude", "ruel", "ruin", "rule", "rumb", "rump", "rung", "runk", "runt", "rupt", "rush", "russ", "rust", "ruth", "sack", "sacr", "sade", "safe", "sage", "sail",
"sake", "sala", "sale", "sali", "salo", "salt", "salu", "salv", "sand", "sane", "sang", "sani", "sano", "sant", "sary", "sati", "saur", "scal", "scam", "scan",
"scar", "scen", "sche", "scho", "scin", "scon", "scor", "scot", "scou", "scra", "scri", "scro", "scru", "scur", "scus", "sday", "seal", "sear", "seas", "seat",
"secr", "sect", "secu", "sede", "sedi", "seed", "sele", "self", "sell", "semi", "send", "seni", "sens", "sent", "sequ", "serf", "seri", "sers", "sert", "serv",
"sess", "sett", "seum", "seve", "shad", "shal", "sham", "shan", "shap", "shar", "shea", "shee", "shel", "shie", "shin", "ship", "shir", "shoo", "shop", "shor",
"shot", "shou", "show", "shut", "sick", "side", "sign", "sile", "sili", "silv", "simi", "simu", "sing", "sink", "sion", "sire", "sist", "site", "size", "skel",
"sket", "skin", "skir", "slam", "slap", "slav", "slea", "slee", "slid", "slim", "slin", "slip", "slog", "slow", "slum", "sman", "snap", "snip", "snow", "soci",
"sock", "soft", "sola", "sold", "sole", "soli", "solu", "solv", "soma", "some", "song", "soph", "sorb", "sore", "sort", "sote", "soul", "sour", "spac", "span",
"spar", "spat", "spec", "spee", "spel", "spen", "spin", "spir", "spit", "spla", "spli", "spok", "spon", "spoo", "spot", "spri", "spro", "spun", "spur", "sput",
"squa", "sque", "squi", "ssen", "sset", "stab", "staf", "stag", "stai", "stak", "stal", "stam", "stan", "star", "stat", "stay", "stea", "stel", "stem", "sten",
"step", "ster", "stic", "stif", "stig", "stim", "stin", "stit", "stle", "stoc", "stol", "stom", "ston", "stop", "stor", "stra", "stre", "stri", "stro", "stru",
"stud", "stum", "subs", "subt", "succ", "suck", "suit", "sult", "sume", "summ", "sump", "sund", "supp", "supr", "surd", "sure", "surg", "surr", "surv", "susp",
"swim", "symb", "synt", "tabl", "tach", "tack", "tact", "tail", "tain", "take", "tale", "tali", "talk", "tall", "tame", "tamp", "tang", "tank", "tant", "tard",
"tare", "targ", "tarn", "tase", "task", "tate", "taut", "taxi", "teal", "team", "tear", "tech", "tect", "teen", "teer", "tele", "tell", "temp", "tena", "tenc",
"tend", "teno", "tens", "tent", "terc", "tere", "terf", "term", "tern", "terr", "terv", "test", "tetr", "text", "thar", "them", "theo", "ther", "thin", "thol",
"thon", "thor", "thre", "thro", "thun", "tice", "tick", "ticl", "tide", "tier", "tiff", "tige", "tile", "till", "tilt", "timb", "time", "tina", "tine", "ting",
"tink", "tint", "tinu", "tion", "tire", "tiss", "tita", "titu", "tman", "tock", "toil", "toke", "told", "tole", "toll", "tomb", "tomo", "tone", "tono", "tool",
"toon", "toot", "tore", "torn", "torr", "tort", "tory", "toss", "tour", "tout", "town", "tput", "trac", "trad", "trag", "trai", "tram", "tran", "trap", "trat",
"trav", "tray", "trea", "tree", "trem", "tren", "tres", "tria", "trib", "tric", "trig", "trim", "trin", "trio", "trip", "triv", "trix", "trod", "trol", "tron",
"trop", "trot", "trou", "troy", "truc", "true", "trum", "trun", "trus", "tter", "ttle", "tton", "tube", "tude", "tume", "tune", "turb", "turd", "ture", "turn",
"tuss", "tute", "twee", "twin", "type", "uage", "uard", "uary", "uate", "ubbl", "ubli", "ucle", "udge", "uenc", "uest", "ught", "uite", "ulat", "ulge", "ulle",
"ulti", "umbl", "umin", "ummy", "umor", "umph", "unce", "unch", "uncl", "unct", "unde", "undr", "unge", "ungu", "unic", "unio", "unit", "unkn", "upid", "uple",
"ural", "urch", "uret", "urge", "urre", "urry", "urse", "ussi", "uste", "ustr", "utch", "uten", "uter", "util", "utin", "utte", "vaca", "vacu", "vade", "vail",
"vain", "vale", "valu", "vamp", "vane", "vani", "vant", "vate", "veal", "vect", "vehi", "veil", "vein", "velo", "vend", "vene", "veni", "vent", "verb", "verd",
"verh", "veri", "vern", "vers", "vert", "vest", "vice", "vici", "vict", "vide", "view", "vile", "vill", "vinc", "vine", "viol", "virt", "visi", "vite", "vive",
"voca", "void", "voke", "volt", "volu", "vore", "vote", "vour", "wade", "wain", "wait", "wake", "walk", "wall", "want", "ward", "ware", "warm", "warn", "warr",
"wart", "wash", "wast", "wave", "weal", "wear", "weed", "week", "weep", "well", "west", "wher", "whip", "whis", "whiz", "whol", "whop", "wich", "wick", "wide",
"wife", "wild", "will", "wind", "wine", "wing", "wipe", "wire", "wise", "wish", "wist", "wolf", "wood", "wool", "word", "work", "worl", "worm", "worn", "wort",
"wrap", "writ", "xact", "xper", "xplo", "xtra", "yard", "yarn", "year", "ymph", "yoff", "yond", "youn", "yout", "zeal", "zero", "zest", "zone", "zoom", "zzle"
```

This indexing subset evolved over time and is somewhat arbitrary. If we had to come up with a 4-gram list again from scratch, it probably would turn out differently.

The goal was just to get a few thousand useful indices from the 456,976 possible sequences of 4 letters, of which almost all are nonsense. Built-in 2- and 3-grams would remain the backbone of AW indexing; but the selection account for more counts in a deomonstration system than either 2-grams or 3-grams separately. Good 4-gram indices shouldbe fairly common in English as word fragments so as to offer more specificity for indexing. Some 4-grams can sometimes be full words, but most are not.

The AW built-ins started with 750 4-letter fragments selected from cryptanalysis listings of the most frequent English 4-grams. Another 450 came from 4-letter fragments with their leading and trailing 3 letters both being common 3-letter fragments. The rest come from common 3-letter fragments plus one letter before or after, from frequent 4-letter words making compounds like CAMPFIRE and SANDHILL, and from familiar patterns like ZZLE and CKLE, far down in cryptanalysis lists because of rarer letters.

Around 2,500 4-letter fragments seem to be a reasonable indexing complement for about 7,000 2- and 3-letter fragments. The latter would still account for more instances of word fragments, but index occurrences of 4-letter fragments already can exceed those of 3-letter fragments in a collection of text. Furthermore, all of the above 4-grams were selected and are less noisy, unlike AW 2- and 3-grams.

Adding more common 4-letter fragments for indexing should always improve indexing. The contribution of a single 4-letter fragment will be tiny, but these will add up over time. Indexing with another hundred 4-letter fragments likely to occur more than one or twice in a text collection should increase entropy by about .01 bits. We just have to find them somehow.

We need not stop at 4-letter fragments for a index set of features. Zipf's Law means that some 4-letter fragments will be much more common than the the rest. These will be candidates for subsuming in a few 5-letter fragments frequent enough to be worth recognizing. Here are 700 of such longer fragments:

```
"abort", "abuse", "actic", "actor", "adult", "advan", "agent", "agree", "alleg", "allow",
"ament", "ameri", "anger", "anima", "ankle", "apple", "apply", "arden", "arena", "arren",
"assoc", "atern", "ation", "attle", "audio", "austr", "avail", "avern", "award", "battl",
"beach", "beard", "belly", "birth", "black", "blame", "blast", "blaze", "blend", "blind",
"block", "blood", "board", "brack", "brain", "bread", "break", "brick", "broad", "brook",
"broth", "brown", "brute", "build", "built", "busin", "cargo", "cause", "cease", "ceive",
"centr", "centu", "chain", "chair", "champ", "chanc", "chang", "chant", "chara", "charg",
"chase", "cheap", "check", "cheek", "cheer", "chees", "cheme", "chest", "chief", "child",
"chrom", "chron", "circl", "circu", "civil", "claim", "class", "clean", "clear", "clerk",
"cliff", "cline", "clock", "close", "cloth", "clown", "clude", "coach", "coast", "colon",
"comma", "commo", "compa", "compe", "confe", "confi", "conne", "const", "consu", "conta",
"conte", "contr", "conve", "cosmo", "counc", "count", "cours", "court", "cover", "crack",
"craft", "crash", "crawl", "craze", "cream", "creas", "creat", "creek", "cresc", "crest",
"crime", "cross", "crowd", "crown", "crude", "crush", "crust", "crypt", "cture", "curio",
"curre", "curve", "custo", "cycle", "dairy", "dance", "death", "decid", "defen", "democ",
"depar", "depen", "depth", "desig", "devel", "direc", "disco", "distr", "doubt", "draft",
"drama", "dream", "dress", "drift", "drink", "drive", "drome", "drone", "dynam", "ealth",
"earin", "earth", "eason", "easur", "educa", "egion", "egree", "eight", "elbow", "elect",
"elite", "ellow", "emand", "ember", "emplo", "enemy", "energ", "enjoy", "enter", "eport",
"equen", "equip", "ermin", "error", "escri", "estab", "estig", "estim", "ethan", "event",
"exert", "expec", "exten", "faith", "famil", "fashi", "fathe", "fault", "featu", "felon",
"fever", "field", "fight", "figur", "final", "finan", "finge", "flake", "flare", "flash",
"flavo", "flesh", "flict", "flood", "floor", "flush", "focus", "force", "fores", "forge",
"found", "frame", "fraud", "frien", "front", "frown", "fruit", "futur", "gener", "ghost",
```

```
"glass", "globe", "gover", "grace", "grade", "grain", "grand", "grant", "graph", "grass",
"green", "greet", "gress", "grind", "groom", "groun", "group", "grove", "guard", "guest",
"guide", "guile", "hance", "happe", "happy", "haven", "heart", "heath", "highw", "histo",
"hollo", "honey", "horse", "hotel", "house", "human", "icate", "icide", "icipa", "iddle",
"ident", "image", "imate", "immun", "impor", "index", "indus", "iness", "infor", "insta",
"instr", "integ", "inter", "intro", "inves", "islan", "issue", "isten", "itude", "janit",
"joint", "judge", "junct", "jungl", "knife", "knuck", "labor", "larce", "large", "later",
"laugh", "learn", "lease", "legit", "level", "light", "limit", "litic", "llage", "llege",
"lleng", "llion", "locat", "lunch", "macro", "major", "manag", "manor", "march", "marke",
"marsh", "mason", "maste", "match", "mater", "matio", "meado", "media", "medic", "membe",
"membr", "merce", "merge", "merse", "metal", "meter", "micro", "might", "milit", "milli",
"minat", "minor", "minut", "mmuni", "model", "money", "monit", "month", "motor", "mount",
"mouth", "movie", "music", "nance", "natio", "natur", "neigh", "neral", "neutr", "ngine",
"night", "niver", "noise", "novel", "nsist", "ntern", "nterp", "numer", "nurse", "ocean",
"offer", "offic", "ollar", "ology", "ommun", "ontra", "opera", "orbit", "order", "organ",
"otent", "other", "ouble", "ounce", "paign", "panel", "paper", "paren", "party", "patch",
"pater", "peace", "peopl", "perio", "phase", "phone", "photo", "piece", "pilot", "pital",
"pitch", "place", "plain", "plane", "plant", "plate", "plent", "plete", "plica", "plore",
"point", "polic", "polit", "posit", "pound", "power", "ppear", "pport", "pract", "prehe",
"press", "price", "pride", "prima", "princ", "pring", "print", "prise", "prize", "proba",
"proce", "produ", "progr", "proje", "prone", "proof", "prope", "prosp", "prote", "proto",
"prove", "publi", "pulse", "quart", "queen", "queer", "quent", "quest", "queue", "quick",
"quiet", "quire", "radio", "raise", "ranch", "range", "ratio", "reach", "recom", "recor",
"reduc", "refer", "refor", "regul", "rench", "repla", "reply", "repor", "repub", "resid",
"ridge", "right", "river", "roast", "roduc", "round", "route", "saint", "salut", "sault",
"scale", "scape", "scene", "schoo", "scien", "scope", "score", "scrap", "screw", "searc",
"seaso", "secur", "sembl", "senat", "sense", "shack", "shake", "shame", "shape", "share",
"shark", "sharp", "sheep", "sheet", "shell", "shift", "shine", "shire", "shirt", "shock",
"shore", "short", "shrap", "sight", "simil", "skate", "skill", "skull", "sland", "slate",
"sleep", "smart", "smash", "smile", "smith", "smoke", "socia", "solut", "solve", "sound",
"sourc", "space", "speak", "spear", "spect", "speed", "spell", "spend", "spill", "spine",
"spire", "spite", "splay", "split", "spoil", "sport", "sprin", "spruc", "squad", "squar",
"ssist", "stabl", "stack", "staff", "stage", "stain", "stair", "stake", "stall", "stamp",
"stand", "start", "state", "stati", "stead", "steal", "steam", "steel", "steep", "steer",
"stick", "still", "stock", "stone", "store", "storm", "story", "stove", "strai", "stran",
"strat", "stree", "strik", "strip", "struc", "study", "stuff", "stunt", "style", "suade",
"subst", "sugar", "super", "sweet", "sword", "table", "taste", "teach", "techn", "tempt",
"teria", "terra", "thank", "theat", "theme", "theor", "therm", "think", "thres", "throa",
"throw", "thumb", "tight", "title", "toast", "tooth", "touch", "tower", "toxic", "track",
"tract", "trade", "trail", "train", "tramp", "trans", "trave", "treat", "treme", "trend",
"trial", "trick", "trict", "troll", "trong", "troop", "troph", "truck", "truct", "trust",
"truth", "tutor", "twist", "ublic", "uffer", "ultur", "ument", "under", "union", "unive",
"ustle", "valen", "valle", "value", "veget", "velop", "venue", "verge", "veter", "video",
"villa", "ville", "visit", "vista", "voice", "volve", "waist", "waste", "watch", "water",
"whack", "whale", "wheel", "whelm", "whimp", "white", "whole", "winte", "witch", "woman",
"world", "worth", "wound", "wrack", "wreck", "wrist", "write", "yield", "youth", "ystem"
```

Some of these were gleaned from cryptanalysis and ESL resources, but others were added just after running across them by chance in regular reading. There are fewer built-in alphabetic 5-letter fragments than 4-letter ones because it is much harder to find 5-letter ones frequent enough for useful indexing of content. The current 700 include many common 5-letter words occurring in place names like PINECREST or FAIRFIELD or in reference to parts of human anatomy like WRIST or CHEEK or ELBOW.

As with 4-letter fragments, we want to hold down the number of 5-letter indexing fragments that cannot also be parts of longer words. We usually want legitimate word fragments like EN**GRAIN** or B**RIGHT** with what might seem to be the full words GRAIN and RIGHT. Otherwise, we will get closer to indexing with full words and face all the associated problems of dependent occurrence.

Nothing prevents us from putting word fragments of six or more letters into our finite index set, but we seem already at a point of diminishing returns. To have noticeable impact, the total frequency of a set of added indices must be significant relative to the total frequency of all indices. Unfortunately, Zipf's Law will work against us: the frequency of most fragments of longer length will be quite low. We need many more of them to improve indexing entropy; but these will be harder and harder to find in quantity.

For general-purpose processing of English text data collections of small to moderate size, our indexing scheme as described here should provide a good starting point. It defines a basic set of 1,296 alphanumeric 2-character fragments and 5,616 3-letter fragments, plus over 3,000 4- and 5-letter fragments described above. A user can then add up to 2,000 literal indices for even more specific coverage of particular content.

Appendix E. Java ActiveWatch System

The ActiveWatch system was developed originally in the C language around 1984 and later ran under IBM OS/2. To make it more platform-independent, a Java version was written around 1997, but was never released. In 2021, this code was recovered from a backup CD-ROM and then cleaned up and updated to provide a demonstration of finite statistical indexing. A subset is now available as open-source software from https://github.com/prohippo/ActiveWatch.

AW consists of a collection of Java modules serving as building blocks for putting together various statistical text processing capabilities. These modules all read their input from files and write their output into other files. The modules currently on GitHub implement automatic clustering of text items arriving in a series of batches. These consist of 11 main modules (SEGMTR, INDEXR, UPDATR, SEQNCR, SQUEZR, MLTPLR, CLUSTR, SUMRZR, CLSFYR, KEYWDR, WATCHR), plus 3 support modules (STPBLD, SUFBLD, LITBLD).

Their functions are as follows:

- SEGMTR divides an input UTF-8 text file into individual items with unique ID numbers.
- INDEXR generate a numerical vector describing each item with respect to finite set of keys.
- UPDATR computes the collection probabilities of index keys.
- SEQNCR selects specified items for eventual clustering.
- SQUEZR gathers selected item vectors into a compact single data file for computing pairwise similarity.
- MLTPLR computes statistically scaled similarity scores for every pair of items to be clustered.
- CLUSTR discovers compact highly interconnected cluster seeds.
- SUMRZR creates descriptive profiles for each cluster seed.
- CLSFYR compares item vectors with a set of cluster profiles and makes assignments of text items.
- KEYWDR compiles descriptive keywords for the set of items assigned to each cluster.
- WATCHR selects unclustered items that stand out because of their n-gram index probabilities.
- STPBLD compiles a binary table of stop words for text analysis.
- SUFBLD compiles a binary table of word suffixes to remove for text analysis.
- LITBLD compiles a binary table of user-defined leading and trailing literal indices.

After downloading AW Java source code from GitHub, you can generate all the demonstration modules by running the provided `build` command file. This will compile the required Java source files and collect the resulting class files into Jar container files for individual demonstration modules with the 6-letter names seen above. To run a module XXXXXX, you can just enter the command line

```
java -jar XXXXXX
```

The `awdemo` command file runs AW modules to cluster items found in a specified text input file ${1} with a pairwise clustering link threshold ${2}, specified in standard deviations.

Coordination between AW modules is by a control file that tracks batches of item already processed. The items, vectors, probabilities and other statistics, descriptive keywords, and various pointers are maintained in separate binary files.

AW also defines more than two dozen other top-level modules that can be plugged into this basic processing framework. These include modules implementing a keyword search engine and producing various HTML files reporting on AW output. they are not yet included in the GitHub repository. Users may write their own Java modules as well to customize their system.

Appendix F. Running the AW Demonstration

Whether or not you compile and jar the modules of the AW demonstration yourself, the next step is to run in the proper sequence. This all has to be done from a command line, which may seem inconvenient, but a primary goal of this entire effort was to get some working software out to the public as quickly as possible. AW has been in mothballs for almost thirty years, and it is showtime really right now.

The original Java ActiveWatch demonstration had a graphical user interface built on the AWT (Abstract Window Toolkit) library. This was rudimentary and is unnecessary for running AW modules. It may or may not be rebuilt from saved Java Source code, which probably needs extensive updating. The command line execution of AW should be adequate for evaluating its utility for prospective users.

The details of setup will depend on whether the platform will be Unix, Linux, or Windows. The details will vary, but it should be straightforward to translate them into one's particular environment. The descriptions here are from macOS 11.5.2 (Big Sur), which is built on Darwin Unix core. At some point, someone may want to develop some auto-installation packages.

Here is what you need to do:

1. Create a new file directory and copy in the AW distribution versions of the `stopword`, `suffix`, `action`, and `literal` files from the AW `table` directory.
2. Copy in the file `delims` from the AW `data` directory.
3. Copy in the `awdemo` shell script and edit it to run on your operating system.
4. Define environmental variable AWD to point to the directory where jar files for AW modules can be found. Or edit the `awdemo` shell script to refer explicitly to that directory.
5. Create a UTF-8 input file containing up to several thousand text items to be processed. Each item should have about two to six thousand characters; they each should be terminated by a single blank line, including the last item. Or you can just use the data file `text` in the AW `data` directory.
6. Run the awdemo script with the path of the text input file to process.
7. In the standard output from the script, look for the number of segments found by the Segmenter (SEGMTR) module, the entropy statistics produced by the Updater (UPDATR) class, the number of items assigned to clusters by the (CLSFYR) class, the listing of clusters with descriptive keywords by the Keyworder (KEYWDR) class, and the standout items unassigned to any cluster as found by the Watcher (WATCHR) class.

By default, SEGMTR will divide longer items into shorter segments for analysis. About two-thirds of these segments will typically then be assigned to at least one cluster under default processing options. You can change that processing by passing arguments to the various modules run by `awdemo`. Some of the options at the command line level for AW users are:

Module	Arguments		Comments
SEGMTR	–d D File1 File2 …	use delimiter file D, defaults to delims first input text file; default is text second	can process multiple text input files specified as successive command line arguments
INDEXR			
UPDATR	Mode	= “+”, “-“, “+-“, or “-+”; default is “+”	“+” means to add the new batch of text to statistics, “-“ means to drop the old batch of text from statistics
SEQNCR	Count	how many items of current batch to cluster; default is ALL	used for debugging
SQUEZR			
MLTPLR	Minimum	threshold for pairwise link; default is 6 standard deviations	
CLUSTR	Minimum	threshold for link density of seed; default is 0.5	link density is 1.0 when every item in a cluster seed is linked to every other one
SUMRZR	Minimum	threshold for cluster assignment; default is 6 standard deviations	
CLSFYR			
KEYWDR	Maximum	number of items to get candidate keywords from; default is 6	
WATCHR	Maximum	number of standout items to display; default is 24	

Appendix G. AW User-Editable Tables

In theory, the finite indexing framework defined in preceding sections should perform quite adequately right out of the box. Our modern information universe, however, puts constant pressure on participants to beat the other guy. Big rewards accrue to those who can find out something first or glean some hidden nugget from mountains of otherwise moldering data. Coming in second here earns little respect.

So, an information tool needs to allow for easy customization and sharpening to give its users some extra competitive edge. AW in particular accomplishes this through three user-editable tables to guide its finite indexing; these are defined by three text files in a file directory from which AW is running: `stopword`, `suffix`, and `literal`. They function as a medieval monastic librarian might expect.

- A stopword table tells AW what words to ignore in building its text indexes. For example, here are some actual entries in that table:

```
any
anybody
anyhow
unless
unlikely
until
```

- A suffix table specifies what word endings are to be removed prior to indexing. For example,

```
aerial      0  2
burete      2  5
onch        2 88
njury       1 11 =injure
nulus       2 10 =annule
ocuss       2  4
```

- A literal table what n-gram indices are to added to the AW built-in lexical n-gram index set, typically whole words or long prefixes and suffixes like PROTO- or -OLOGY. For example,

```
-endorse
-ession
-establish
-final
hunt
hurricane
hydro-
hyper-
```

The stopword file is straightforward. It is a straight list of words, one per line; they need not be in alphabet order. The suffix file is more complicated in that each entry consists of a sequence of letters to look for in at of a word plus a number specifying a condition for a match and another number for what to do after a match. The literal file lists possible trailing ones, starting with a "-" and possible leading ones with a "-" or with no final hyphen at all. You can freely edit these three files, although new AW users probably should them all alone unless getting maximum entropy of indexing is critical.

In a suffix file entry, the condition can be 0 = no suffix to be removed, 1 = take specified action in entry, 2 = take specified action only if only the matched sequence is preceded by a consonant, 3 = same as 2, but the preceding letter can also be a U. The possible actions are defined in an accompanying action file, indicating how many letters of a matched sequence to keep in a word root and other letters are to be added to it. If a 0 condition is specified, then the action should be non-zero, as seen in the example for AERIAL above.

The action file defines 125 different things that can happen after a match of an entry in a suffix file. Each of these possibilities occupies a single line in the action file; here are some actual examples:

```
  4ex.    =77
  0ounce. =78
  4sy.    =79
  3al.    =80
  1ble.   =81
```

These define actions 77 through 81. Each entry starts with a single digit indicating how many characters of a word ending table match are to be kept in a word root; this is followed by a sequence of letters that will be appended to finish off a root. In the case of action 81, it is taken in the suffix rule

```
    mility     3 81
```

This will be matched by the word HU-MILITY. Action 81 then puts back one letter of the suffix rule sequence (the M) and adds on BLE. This yields the word root HU-M-BLE or HUMBLE.

We can also define special-case rules describing how particular words are to be stemmed. This is indicated in a morphological rule that begins with a vertical bar (|) indicated the start of a word. For example,

```
    |men    1 118
```

This changes the plural MEN to the singular MAN with the action:

```
  1an.    =118
```

Morphological rules need not always be applied only to changes of parts of speech for a word root.

The AW STPBLD, SUFBLD, and LITBLD modules generate respective binary files stps, sufs, and lits to be loaded by other AW modules at run time to guide finite indexing. For just a basic demonstration, it is enough just to use just the stopword, suffix, and literal files provided in the AW GitHub distribution. The purpose of editing `stopword`, `suffix` (possibly along with `action`), and `literal` should be to increase indexing entropy. This generally can increase the total number of clusters found in the AW demonstration, but may decrease the total number of clustered text items.

Appendix H. ActiveWatch Indexing Probabilities

With the default set of built-in and user-defined n-grams, the AW demonstration system processing the sample Google News summaries will produce indexing probabilities still following Zipf's Law. The AW diagnostic utility tool DPRB can be run to list the N most frequent n-grams. This is a good initial check that AW is working as expected. Here is the actual DPRB output for the top 64 n-grams in a recent run.

```
probability range= 0.000011 : 0.004653

6170 non-zero indices with sum of probabilities=1.000000
based on 89398 total occurrences of indices
computed entropy = 11.6 bits, 91.9 percent of maximum
(saved entropy= 91.9)

64 indices with highest percentages of occurrence

 preside-          (  335):  percent=0.4653
 trum              (11270):  percent=0.4329
 rump              (10894):  percent=0.4307
 office-           (  311):  percent=0.3501
 state             (12073):  percent=0.3423
 year              (11449):  percent=0.3277
 pro-              (  337):  percent=0.2975
 report-           (  363):  percent=0.2763
 time              (11200):  percent=0.2651
 new               ( 5117):  percent=0.2383
 people-           (  320):  percent=0.2282
 20                ( 8735):  percent=0.2260
 lead              (10136):  percent=0.2204
 us                ( 8439):  percent=0.2125
 ore               ( 5671):  percent=0.2025
 -son              (   45):  percent=0.1890
 day               ( 2909):  percent=0.1890
 north-            (  306):  percent=0.1846
 act               ( 2046):  percent=0.1834
 house             (11775):  percent=0.1823
 -russia           (    1):  percent=0.1745
 01                ( 8664):  percent=0.1734
 man               ( 4822):  percent=0.1700
 -nation           (   42):  percent=0.1667
 news              (10384):  percent=0.1667
 law               ( 4649):  percent=0.1644
 rea               ( 6161):  percent=0.1644
 country-          (  161):  percent=0.1600
 part              (10556):  percent=0.1566
 call              ( 9263):  percent=0.1555
 ko                ( 8075):  percent=0.1532
 police-           (  327):  percent=0.1477
 work              (11436):  percent=0.1454
 tige              (11196):  percent=0.1443
 invest-           (  264):  percent=0.1432
 ama               ( 2183):  percent=0.1421
 force             (11728):  percent=0.1421
 move              (10335):  percent=0.1409
 old               ( 5540):  percent=0.1398
 tain              (11134):  percent=0.1398
 fire              ( 9756):  percent=0.1376
 estig             (11697):  percent=0.1365
 elect             (11680):  percent=0.1353
 america-          (   87):  percent=0.1331
 -unite            (   27):  percent=0.1309
 republic-         (  365):  percent=0.1309
 know              (10103):  percent=0.1309
```

```
count              (11616):  percent=0.1309
late               (10130):  percent=0.1298
govern-            (  230):  percent=0.1275
south-             (  392):  percent=0.1275
long               (10201):  percent=0.1275
campaign-          (  114):  percent=0.1242
politic-           (  329):  percent=0.1242
atte               ( 9130):  percent=0.1242
white              (12186):  percent=0.1230
dona               ( 9523):  percent=0.1208
nald               (10361):  percent=0.1208
usa                ( 7305):  percent=0.1197
atta               ( 9129):  percent=0.1186
senate-            (  381):  percent=0.1163
land               (10116):  percent=0.1141
face               ( 9709):  percent=0.1130
tack               (11131):  percent=0.1119

total percentage of occurrences for top 64 = 11.54
```

The Google News sample is small and has only 6,170 out of about 10,000 n-gram indices defined by default for AW. The entropy of indexing was 11.6 bits, or about 92 percent of the maximum entropy for the indices occurring in the Google data. This is actually quite good, but could improve a bit if we would recognize literal n-grams like TRUMP- and KOREA-.

The built-in AW n-grams were selected mostly according to their frequency in general English. They should not be biased toward any particular content. The AW default literal n-grams, however, were originally compiled for news text decades ago at the turn of the 21st Century.

Even when AW is running right out of the box on the Google News text, AW default literal n-grams occurred 20 times in the top 64. Alphabetic 5-grams show up 7 times; alphabetic 4-grams appear most at 23 times, more often than literals. The largest category of built-in AW n-indices will be 3-grams, with over 5,600 defined, but these occurred only 10 times in the top 64. Alphanumeric 2-grams occurred only 4 times, notably US from U.S. in text and KO from KOREA.

DPRB output such as above is helpful in suggesting where longer n-grams could improve AW indexing. Currently, the best strategy for improved indexing is by adding alphabetic 4-grams, but finding enough of them to make a noticeable difference in indexing can be a challenge. Frequent 3-grams are clues to new 4-gram index candidates.

For example, take the 3-grams MAN or ORE listed in the top 64. Add a single letter to either their front or their back . If these 4-grams are not already in the AW built-in index set, look for multiple words containing or common names them. If you can find at least three or four such words or names, then those 4-grams are worth building in. They often may be more easily handled as literals, however.

(Little can be done with the 3-gram MAN, which often is a word by itself or at the end of a longer word like FIREMAN. The 3-gram ORE mostly comes from the names KOREA and MOORE, the latter a senatorial candidate caught up with a history of pursuing underage teenage girls. It is each user's call as to whether these 5-grams should be added as literals; they are probably not general enough to include as built-ins.)

Another ten or twenty 4-gram indices will of course make negligible contributions to AW indexing entropy when there are about 10^4 indices altogether. Everything will add up, however. Built-in 4-grams actually teach AW about English to make it smarter in indexing. This is short of artificial intelligence, but is close to how human children learn their first language in small lessons that just keep on coming.

Appendix I. Actual ActiveWatch Clustering

The ActiveWatch (AW) demonstration analyzes a set of 611 short segments of text collected in 2017 and 2018 from stories in Google News. With a set of about 10,000 literal and built-in n-grams compiled in 2000 and 2021, AW v1.3.4 found 78 distinct clusters of three or more text segments significantly related to each other by a scaled similarity of at least 8 standard deviations. AW then identified the words or word roots most contributing to that similarity of segments within each cluster. Here are the top 16 such words or word roots for each of the top five clusters by seed size:

zimbabwe	mugabe	robert	preside
mnangagwa	succeed	emmerson	harare
rule	independ	capital	lead
power	decade	govern	africa

explode	detonate	suspect	austin
device	police	bomb	pack
report	brian	texas	hour
kill	allege	rock	dead

ventura	santa	california	evacuate
fire	county	thoma	barbara
scorch	flame	angele	communite
new	order	reside	push

cambridge	facebook	analyt	data
million	zuckerberg	mark	profile
employ	firm	committee	scrutiny
london	usa	share	work

submarine	argentine	crew	juan
navy	san	south	miss
coast	report	ara	atlantic
spokesman	plata	communice	know

The last five of the 78 clusters have smaller seed sizes. This will result in fewer being assigned to each ckuster and fewer words or word roots to choose from as descriptors of the content of the clusters.

organize	trump	kushner	website
jare	familiar	message	hillary
thousand	email	clinton	campaign
sent	person		

miami	collapse	florida	dade
bridge	pedestr	university	internation
construct	day		

superintend	school	district	florida
pennsylvania	classroom	authorit	

theresa	britain	prime	ministr
london			

pennsylvania	philadelphia	neighbor	kilometr
west	multiple	minute	communite
mile	shoot		

AW will generate an n-gram profile showing the indices most associated with the content of each cluster seed. This will be used to assign text segments to that cluster, including those outside of its original seed. Individual words and word roots in clustered text can also be scored by the total weight of their n-gram indices for such a profile; and the highest scoring ones can then become the cluster keywords shown above.

AW could also derive descriptive phrases for each cluster, but just words and word roots can provide a fairly readable summary of a cluster as a major news topic. For example, the top cluster above is about a military coup ousting the long-ruling President Robert Mugabe of Zimbabwe. The next cluster is about a bombing in Austin, TX. The ordering of the top five clusters has been stable even with the recent addition of several hundred new alphabetic 4-gram indices and over a hundred 5-gram indices..

AW produced its 78 clusters in under 10 seconds on an Apple MacBook Pro laptop (2017) with Java openjdk version "11.0.11" 2021-04-20. Only 474 segments out of 611 were actually assigned to clusters with seeds formed at a minimum link threshold of 8 standard deviations. AW was running without any tuning of word-fragment indices for words or names expected to be newsworthy in 2017 or 2018. In particular, the name TRUMP was unknown and had to be analyzed into the 4-grams TRUM and RUMP.

The AW clustering procedure begins by computing scaled similarity measures for every pair of starting text segments. Its initial cluster seeds come from a minimal spanning tree algorithm, along with constraints on seed size and connectivity of seed members. This will allow AW to work from a set of highly interconnected items when creating its n-gram profiles for each cluster seed.

The clusters produced by AW will generally depend on the particular indexing employed and on the thresholds set at each stage of clustering. Strongly linked segments will tend to stay together regardless, but the details of final groupings can vary considerably. Making final cluster assignments with a n-gram profile means that a segment can be in more than one cluster, though each segment will usually have its highest match score for one particular profile.

Clusters with small seeds will tend also to have fewer text segments eventually assigned to them. This typically means fewer descriptive word or word root candidates to choose from and lower overall descriptive quality. Unassigned text segments will be recorded in a residual file; this can be combined with any new incoming segments as input to the next round of clustering.

Appendix J. Unclustered Text Items

When clustering, we typically can choose tighter or looser groupings by adjusting the thresholds of our algorithms. With text items, however, loose groupings can become incoherent and hard to describe. If we want useful clusters, we have to allow a significant portion of a starting set of text items to be unassigned to any cluster. In ActiveWatch, these are called “residuals.” When monitoring a text input stream by clustering, we often want to look at residuals to see what we are missing.

We have many options here, including random sampling, but with a reliable estimate of the probability of any indexing n-gram occurring in a text item, AW can be more selective. In particular, we can identify residuals that stand out in having unusually many low-probability n-gram indices. The latest version of AW has a new module (WATCHR) able to score an item by the sum of the probabilities of its indices. when its sum of indexing frequencies is above some minimum threshold.

The WATCHR module is still experimental, but when run against the AW cluster analysis of Google News sample data, it produces some non-random results. Here the top sixteen residual items in ascending order by sum of index probabilities:

```
 1)  0::399   0.02202510 (  80)
 2)  0::266   0.02524665 (  85)
 3)  0::171   0.02588425 (  85)
 4)  0::104   0.02894919 ( 124)
 5)  0::69    0.03360254 ( 129)
 6)  0::283   0.03455335 ( 106)
 7)  0::427   0.03548178 (  82)
 8)  0::70    0.03580617 ( 124)
 9)  0::339   0.03619768 ( 133)
10)  0::60    0.03627598 ( 145)
11)  0::155   0.03634309 ( 105)
12)  0::351   0.03679053 ( 118)
13)  0::570   0.03710374 ( 120)
14)  0::231   0.03743932 ( 116)
15)  0::24    0.03771897 (  84)
16)  0::212   0.03965413 (  87)
```

The AW ID for each item subsegment is in the second column; their sums of probabilities are in the third column, and their sums of index counts is in the fourth. As can be seen, the first four items stand out from the rest in their low sums of probabilities. Here are the starting sentences of the first six items:

0::399 Samsung just announced the Galaxy S9, and now we’re getting details on how much retailers and wireless carriers are going to charge for it.

0::266 The Supreme Court began hearing arguments Tuesday in one of the term’s most anticipated cases: whether the First Amendment protects a Colorado baker from creating a wedding cake for a same-sex couple.

0::171 Wednesday was a typical afternoon but an exciting one — Thanksgiving was the next day, and Jamie Billquist and his wife, Rosemary, would partake in one of their favorite traditions: the Turkey Trot.

0::104 Why was Piglet staring down the toilet? He was looking for Pooh. Today, November 19th is Toilet Day.

0::69 Tesla plans to reveal an electric commercial truck on Thursday night, which CEO Elon Musk said will "blow your mind out of your skull and into an alternate dimension."

0::283 A baby boy mistakenly declared dead by an Indian hospital last week, passed away on Wednesday morning.

These items probably had too little immediate followup in a news stream to be picked up as a cluster with from one small batch of Google News stories. AW requires that a cluster seed must have at least three interconnected items. Many residuals also seem to be “soft news.” Some deeper digging is needed here before we can fully validate the processing done by AW, but so far there is no indication that AW results are out of line.